Sparse Approximation and Dictionary Learning Using Cloud K-SVD for Image Denoising

Christian Marius Lillelund
201408354@post.au.dk

Henrik Bagger Jensen
201304157@post.au.dk

Master's Thesis

Section of Electrical and Computer Engineering
Department of Engineering
Faculty of Science and Technology
Aarhus University
Denmark

Sparse Approximation and Dictionary Learning Using Cloud K-SVD for Image Denoising

Christian Marius Lillelund and Henrik Bagger Jensen

M.Sc. Computer Engineering, Aarhus University

January 3, 2020

Title page

Thesis title	Sparse Approximation and Dictionary Learning Using Cloud K-SVD for Image Denoising
Thesis ECTS	30
University	Aarhus University
Faculty	Faculty of Science and Technology
Department	Department of Engineering
Section	Section of Electrical and Computer Engineering
Master's program	Computer Engineering
Authors	Christian Marius Lillelund and Henrik Bagger Jensen
Student numbers	201408354 and 201304157
Date of submission	January 3rd, 2020
Supervisor	Christian Fischer Pedersen

Preface

This master's thesis is written for the Department of Engineering at the Faculty of Science and Technology, Aarhus University. It is part of the study program Computer Engineering, and was written in the autumn of 2019.

We would like to thank our adviser Christian Fischer Pedersen for suggesting the initial problem domain, supervising our thesis, and for providing relevant data, that allowed us to carry out the experiments we wanted.

All source files associated with this thesis are found at: https://github.com/thecml/thesis-cloudksvd

Aarhus, January 3rd, 2020

*****

Abstract

Introduction: Ever since the Internet got traction in the 1990s, the total sum of data available online has been ever increasing. As of September 2014, there were 1 billion websites on the Internet and estimates say, that the big four service companies (Google, Amazon, Microsoft and Facebook) store at least 1,200 petabytes between them. With this surge of data, there has been a growing interest in the study of sparse approximation of signals for applications such as compression of data, regularization in statistics and denoising of acoustic signals and images. **Methods**: A central problem in sparse approximation is the design of a dictionary that contains fundamental signal-atoms, where signals can be described as linear combinations of these atoms. For this task, we can either choose a dictionary that has been designed for us (e.g. Haar, DCT) or make our own by adapting a random dictionary to a set of training signals, as done in the acclaimed K-SVD algorithm from 2006. Ten years later in 2016, a new dictionary learning algorithm called cloud K-SVD saw the light of day, an extension to K-SVD that offers several benefits when data is bulky or governed by privacy concerns: Cloud K-SVD can train data at mulitple local nodes and hereby produce a mutual dictionary to represent low-dimensional geometric structures in all the data combined by alternating between sparse approximation of the signals and updating the dictionary to better fit the data in a collaborative fashion. **Results:** In this thesis, we analyze cloud K-SVD and demonstrate its efficiency at learning geometric structures in both synthetic and real data. We also show its ability to remove Gaussian noise from natural images as a benchmark application and from high-dimensional medical computerized tomography (CT) images as a practical application. The results suggest that cloud K-SVD is well-suited for constructing a mutual dictionary among multiple nodes on both synthetic and real data. Moreover they show that cloud K-SVD can remove quantifiable levels of noise from benchmark and medical images, but at the cost of a large memory consumption and a lot of network traffic overhead. **Future work:** The next step would be to deploy our solution to an enterprise-system that can handle larger data loads, for example images in full resolution and in 3D, than we are currently able to. Moreover other protocols, other than HTTP over Ethernet, should be investigated for communicating between nodes, as other protocols may improve performance.

Keywords— Sparse approximation, Dictionary learning, Distributed systems, Consensus, Image Denoising, Kubernetes

Resume

Introduktion: Internettet blev for alvor populært i 1990'erne og siden er den samlede mængde af data, som er tilgængeligt online, kun steget. I september 2014 var der en milliard hjemmesider på internettet, og det siges at de fire store internet-firmaer (Google, Amazon, Microsoft og Facebook) sidder på mindst 1200 petabytes hver især. Denne tendens har gjort det interessant at undersøge, hvordan datasignaler kan repræsenteres med mindre data til rådighed, såkaldte *sparse* approximationer. Det finder nytte ved blandt andet komprimering af data, indenfor statistik og til at fjerne støj fra lyd- og billedsignaler. **Metoder:** Et centralt problem ved disse *sparse* approximationer er at kunne lave et såkaldt *dictionary* (opslagsværk, red.), som består af grundlæggende signalvektorer, som så kan bruges til at repræsentere andre signaler ved linearkombinationer af disse grundlæggende vektorer. Til det kan man enten vælge et *dictionary*, som er lavet i forvejen (f.eks. Haar og DCT), eller lave et selv ved at træne det efter bestemte træningssignaler, hvilket var tilfældet med den meget anerkendte *K-SVD* algoritme fra 2006. Ti år senere i 2016 så en ny algoritme, til at træne disse *dictionaries*, dagens lys, nemlig cloud K-SVD. Denne er en udvidelse til K-SVD fra 2006 med flere nye fordele, når man arbejder med store datamængder eller data som er beskyttet mod deling. Cloud K-SVD kan træne et *dictionary* på flere computere og hermed producere et samlet fælles *dictionary*, der kan repræsentere lav-dimensionelle geometriske strukturer i den samlede datamængde. Dette opnås ved dels at lave en *sparse* approximation af dataet og dels at træne et *dictionary* til bedre at repræsentere den data, man træner, gennem samarbejde imellem computere. **Resultater:** I dette speciale analyserer vi cloud K-SVD og demonstrerer hvor effektiv den er til at lære geometriske strukturer i både syntetisk og rigtig data. Vi viser også dens evne til at fjerne normalfordelt støj fra billeder, der enten er konstruerede til formålet eller produceret af en ct-scanner. Vores resultater peger på, at cloud K-SVD er god til at lave et fælles *dictionary* blandt flere computere på baggrund af forsøg med både syntetisk og rigtig data. Ydermere viser de også, at cloud K-SVD kan fjerne målbare mængder af støj fra testbilleder og billeder fra en ct-skanner, men på bekostning af et betydeligt hukommelsesforbrug og megen netværkstrafik. **Fremtidigt arbejde:** Det næste naturlige skridt vil være at udrulle vores applikation på et system, der kan håndtere større datamængder, f.eks billeder i fuld størrelse og i 3D, end vores nuværende system kan. Desuden bør andre netværksprotokoller end HTTP over Ethernet undersøges til kommunikation mellem computere, da de måske kan forbedre systemets ydeevne.

Contents

Chapter 1

Introduction

We live in an era that has been dubbed the Big Data era by the computer science and engineering community, as large volumes of a wide variety of valuable data continue to be generated at a non-stop pace from a broad range of data sources [1]. For instance, data gathered exclusively by social networks translates to millions of images, videos and messages alone. According to statistics, almost 500 TB social data is produced every day [2]. With the digital transformation we see today, all kinds of information has started being present in cyberspace in unprecedented quantities. This phenomenon has encouraged researchers to experiment and invent new ways of processing and storing large amounts of data that would otherwise have proved to be infeasible with current technologies and knowledge.

This study addresses how advances in computer engineering in the areas of distributed systems and machine learning can help solve problems in data processing and representation that traditionally required immense resources or unrestricted access to all available data at once [2]. In today's world, scenarios exist where data is presented to the system in such a large quantity that any processing or information gathering would be intangible. The presented data could have been subjected to large amounts of noise or data regulation laws could be in place, like the recently introduced GDPR regulations in all European Union member states, that prohibit personal data of interest to be sent around freely.

We address the problem of big data and how we can learn to process it better, since it has become relevant in a number of fields. These include the audio and sound industry, where distortion and noise can easily ruin an otherwise brilliant recording (case 1), the entertainment industry, where online video streaming services get hamstrung by a lack of proper coverage and bandwidth limits (case 2), and finally in the health-care sector, where images from computed tomography (CT) scanners are easily susceptible to noise (case 3). Improvements in these fields could benefit avid podcast listeners, quality-concerned television viewers or medical doctors like radiologists, respectively. The next section will present these three examples as practical cases and investigate possible solutions.

1.1 Practical cases and SOTA

Case 1, audio: The first case where our initial problem could find relevance is in problems with audio recordings where unwanted noise easily sneaks in. We consider an auditorium meant for presentations, lectures and speeches, where computer devices with microphones are placed around the hall to record all speech. An example of this case is illustrated in figure 1.1. These capturing devices record all acoustic signals both real speech and noise coming from the audience and outside the room. The degree of external noise may vary depending on the time of day. Each device has a CPU, some on-board memory and a connection over wireless to an external server.

Its job is to capture all acoustic signals, use some method to remove any unwanted noise and deliver the final result to a sink that can collect and store multiple signals. The sink must balance speech and noise accordingly - this means excluding nodes from taking part in producing the final speech signal if they only pick up noise on their end. For this case, the recording and denoising part is located at the capturing device placed in the hall and aggregation of multiple incoming acoustic signals is done somewhere else. Given the auditorium is large, more capturing devices and multiple sinks may be needed. We consider this case to be interesting for denoising acoustic signals in large halls, at concerts, stadiums and places with a lot noise.

State of the art: In order to enhance the audio quality and remove unwanted noise we find the *DANSE* algorithm made by A. Bertrand and M. Moonen in [3]. We will not go into detail on the theory behind the algorithm, but we are still interested in some of their research. The special part about their algorithm and their papers, is the research of recording audio on multiple stations or nodes located at different spots in some confined area, and looking at the processing of said recordings in what is a distributed system (multiple nodes basically). They go from doing a sequential node updating system [3] to a more complicated system that is simultaneous and supports asynchronous node updates in [4]. With some modifications to their algorithm they show promise and efficiency in the simultaneous and asynchronous node update, which in the sense of this case 1, is exactly what we could look into.

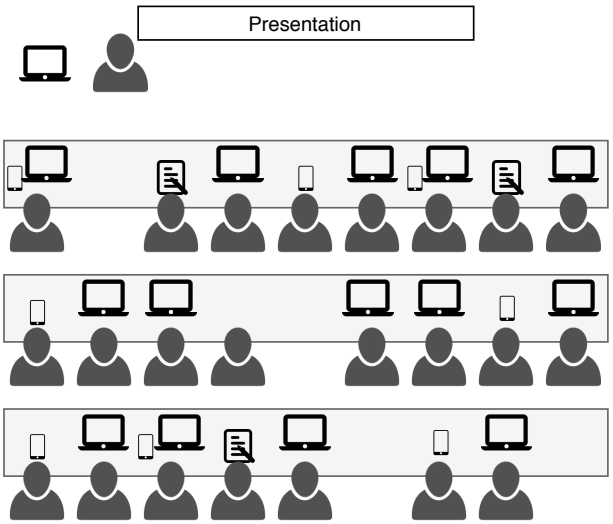

Figure 1.1: *A practical example of case 1 set in an auditorium, where a presenter wants to record a presentation without any external interference or noise.*

Case 2, online streaming: An online video or game streaming service like Netflix[1] or Twitch[2] offer a ton of content in high-definition (HD) quality over the Internet for consumers. At the moment, viewers must specify the level of picture quality in advance for the whole received signal rather than parts of it. In areas of the world where Internet bandwidth capabilities are limited and delays may occur, this may force the viewer to select a low definition (SD) version of the entire video signal. Ideally, one should be able to pick a hybrid where interesting parts of the signal is kept in HD and others in SD, such as a birds-eye view of a football game where players are well-detailed but the audience is not. An example of this case is illustrated in figure 1.2.

State of the art: A technique called coding tree unit (CTU) has been investigated and used in multiple studies [5] [6] [7]. The most recent relevant study stem from 2017 where Shen et al. in [8] encode 3D videos that contain multi-view texture frames using the 3D-HEVC standard. Each coding unit (CU) is responsible for splitting entire video segments into sub-segments or sub-CU's recursively. At each CU depth, the technique allows different types of frame modes to represent parts of the video differently. This ties back to our original problem for case 2 since these coding units allow splitting the received video stream into multiple separated parts and use a different coding strategy, for example a higher resolution, in each individual part. This way patches can be extracted from both wanted and less wanted parts of the stream signal and passed to the

[1]Netflix is an American media-services provider and production company that offers online streaming of film and television series, see: https://www.netflix.com

[2]Twitch is a live streaming video platform where people or organizations can broadcast esports competitions, music broadcasts or creative content, see: https://www.twitch.tv/

HEVC algorithm to compute CU's that match the viewers expectations. For example by putting more emphasis on CU's that represent the wanted parts (the players in the football match) of the signal and less on CU's that represent the unimportant parts (the audience watching the match). Since the amount of data tend to be large for 3D video encoding and decoding, such an algorithm need considerable amounts of computer resources to perform.

Figure 1.2: *Case 2 illustration of HD and SD patches, in a 9 × 6 grid style. Modified screenshot of a CS:GO match from Twitch*[a].

[a](11/10 2019) RERUN: CS:GO - FaZe Clan vs. mousesports [Overpass] Map 2 - Group A - ESL EU Pro League Season 10.

Case 3, medical images: For the third case we turn our attention to the health-care sector and a problem they face that implicate image distortion. People suffering from the chronic autoimmune disease rheumatoid arthritis (RA) are having regular scans using a CT scanner and follow-up checks at the hospital. This is to help establish a definitive diagnosis for the individual and monitor disease progression. Specifically, radiologists and doctors use these images to measure cartilage and bone damage in the wrists and hands of patients, a measurable consequence of RA. Unfortunately, the images produced by the CT scanner are often contaminated by static noise and unwanted artifacts as illustrated in figure 1.3. The additive noise degrades image quality and makes the job of radiologists and doctors harder. Moreover, these images exist in large dimensions and data come in such large quantities that it can prove difficult for a single computer to process. If one were to try to remedy some of the flaws here, it is important to consider privacy concerns with this kind of data as well, as such images are produced for medical purposes and should never be distributed anywhere to any system on the Internet without the patient's consent.

State of the art: The problem that medical staff face appears fairly simple, but there are multiple facets to it: First, the data produced by the CT scanners come in high dimensionality and can take up a considerable amount of both storage and memory capacity in systems. In order for a single system to process entire images it must have the necessary resources at hand in terms of processing power (CPU) and available memory to load and process the data in its entirety. Second, there exists no noise-free versions of the images, which means that there are no reference points to use for an algorithm to understand how it is supposed to look or to measure its performance. It would be up to the radiologists to score noise-filtered images as either improved or worsened. Third, since this is real medical data and stem from actual patients, there are laws in-place to prohibit distribution of the data and any third-part involvement in the processing that has not been

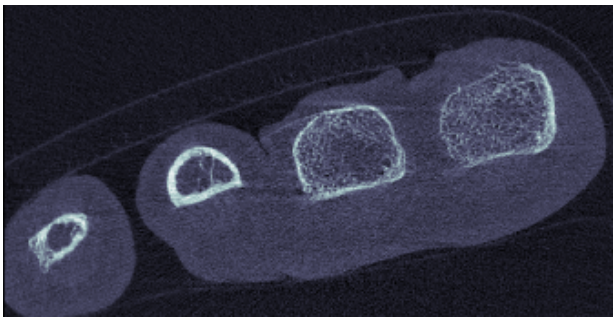

Figure 1.3: *A sample captured by a CT scanner from a dataset that has been put together by the DanACT study group. The group was established in 2014 with the goal of estimating effectiveness and the safety of various protocols for treating patients RA. Though it may not be visible in this picture, the sample contains a substantial amount of static noise and some unwanted artifacts.*

already approved. To reiterate, the problem here is manifold and any SOTA solution needs to deal with all facets of it.

By reviewing recently released studies and literature we come across an interesting algorithm purposed by H. Raja and W. Bajwa in [9] and considered by S. Aleti et al. in [10] that address the multiple problems we have just described with medical image data. Their algorithm is called cloud K-SVD and is an extension to a method developed by Aharon et al. in 2006 called K-SVD [11]. In layman's terms, the algorithm leverages techniques from the fields of digital signal processing and machine learning, called sparse approximation (SA) and dictionary learning (DL) respectively, to learn of and recognize geometric structures in image data. When the algorithm has been exposed to enough training data, it can restore or rebuild the original data by making an approximation of it. Such reconstructions have been shown in studies to effectively reduce noise and distortion in images due to how the approximations are made [12] [13] [14]. Cloud K-SVD is interesting because it actually addresses all three problems in case 3: First, the algorithm does not assume that training data is centrally available, which allows dividing the spacious CT images into blocks and store them separately on more than one computer. Second, because of how the approximation's work, cloud K-SVD should be able to solve our noise problem without relying on noiseless reference images. Third, though cloud K-SVD is a distributed algorithm, it performs all computations locally and only shares the residual errors (how well it has done) with its peers. In other words, medical images are not sent around to all participating computers, but rather kept at individual sites the entire time.

1.2 Where we can contribute

By reviewing recent research in the area, we see that the number of *field-tested* sparse approximation and dictionary learning methods are sparse. By *field-tested* we mean methods that have been tested with real data, on real systems and that are backed by experiments done in the field. Those that have been done often assume that data is centrally available to every node [15], they do not consider node failure in scenarios where the dictionary is spatially separated across multiple nodes [14] and they often have to make assumptions about the data and network. As of the this writing, we were not able to locate any scientific papers or studies that documented use and results of implementing cloud K-SVD in real distributed system for such a use case as case 3, medical images. In [9], Cloud K-SVD was tested with synthetic and image data for classification purposes, but not for image denoising and not in a real distributed setup where network delays, asynchronous communication concerns and package loss play a role. Moreover the experiments done in [9] assumes data is centrally available, which may not be the situation for our third case.

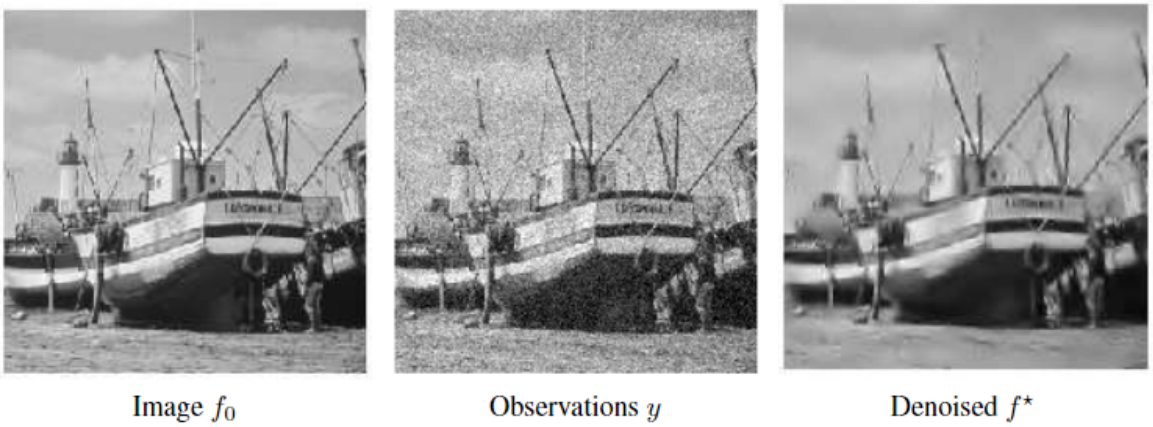

Figure 1.4: *A showcase of what denoising means. To the left is the original image f_0 of a boat without any noise, in the middle are the observations made from a noisy version of f_0 and to the right is the denoised version of f_0. Source: [16].*

1.3 Problem definition

The focus of the present master's thesis is to design, implement, test, compare, and document variants of the already existing cloud K-SVD algorithm for solving sparse inverse problems via dictionary learning with applications to image reconstruction and denoising. As a novelty, the feasibility of implementing and using cloud K-SVD for denoising of natural images will also be investigated. The algorithm must be optimized to (1) learn and make approximations of distributed synthetic data to prove its correctness, and (2) learn and reconstruct distributed patches that have been extracted from noiseless and noisy benchmark images and noisy medical images. Thus the algorithm must be proven functional via concrete experiments. To quantify the quality of the variants of cloud K-SVD, comparative evaluations should be carried out via objective metrics; in the comparisons, time and error aspects should also be considered. As an overall assessment of cloud K-SVD, evaluations of its scalability and practical use should be conducted.

1.4 Outline

Let us offer a brief outline of the thesis. Chapter 2 provides an introduction to the domains and scientific fields related to cloud K-SVD. This includes signal processing history, how dictionary learning works and what other studies have done. This chapter is intended as a layman's introduction to the domain without going into theoretical details. Chapter 3 and 4 provide in-depth information about the mathematics behind it, algorithms and details on how to develop modern distributed systems. Chapter 5 explains how we have designed and implemented cloud K-SVD, which precedes chapter 6 where the algorithm is demonstrated and documented via concrete experiments. Chapter 7 is the discussion and conclusion to the thesis.

Chapter 2

Historical background and studies

Cloud K-SVD is a dictionary learning algorithm that draws on the expertise and knowledge of multiple scientific fields and study groups. Most prominent are the areas of statistics, digital signal processing and computer engineering which have been the main contributing factors. In this chapter we will go through the main contributions, state of the art studies and inventions that have laid the groundwork for distributed dictionary learning and how it can be facilitated.

2.1 A look at modern signal processing

The foundation of modern digital signal processing techniques is established on the pioneering work that Nyquist, Shannon and other engineers did on recovering continuous-time signals from a uniformly distributed set of samples [17] [18]. They showed that any signal could be recovered exactly from a set of samples captured at more than the Nyquist frequency, which is twice the highest frequency in the input signal. This later became Nyquist–Shannon's celebrated theorem on noiseless coding and underlies many modern digital signal processing applications like visual electronics, medical imaging devices, radio receivers or sound synthesis for virtual reality (VR) systems. Ideally in any signal, we would like as much information preserved at the time of capture to later identify all interesting bits about it, and if we have to transmit it over any noise-prone media, then all over information would be conserved, and this is what the Nyquist–Shannon theorem helps us do. Moving on from 1949 when this was first discovered, the amount of data available in the world and exposed to our capturing devices has grown exponentially, whilst advances in computer science has given us immensely more powerful computers and devices than what was present in 1949. This means that sampling at the proposed Nyquist rate today could lead to an overflow of samples or simply be to costly hardware-wise, because the demanded frequency would be too high. Researchers realized that going forward, signals had to undergo some form of compression to even allow data acquisition and later recovery. This gives rise to a technique called *transform coding* that exploits the concept of sparsity: There exists a basis or frame that provides a sparse or compressible representation of a signal. Sparse in this context means that a signal of length N can be represented with K nonzero coefficients, where $K \ll N$. Sparsity leads to dimensionality reduction and is an efficient compression technique [19]. A compressible signal can be reproduced with only K nonzero coefficients. Many real-world signals are not entirely sparse, but often their representation in a certain basis has only a few large and many small coefficients. These mentioned aspects play a role in compressed sensing (CS), a emerging framework to potentially reduce the sampling and computational cost of large signals, because we have prior information about their sparsity in some domain. So saying we can reduce the frequency and number of samples captured. It allows for solving ill-posed linear

systems. In other words, systems where superposition[1] holds that do not have a unique solution. Typically such problems need to be reformulated and some assumptions have to be made to find a solution. The area of compressed sensing rose to prominence around 2005 and 2006 thanks to the work of Emmanuel Candès, David Donoho and others who showed that we need only a small set of measurements to exactly recover a signal originally in finite-dimensions given it can be approximated by a sparse vector and some other mild conditions apply [19] [20] [21, chapter 3].

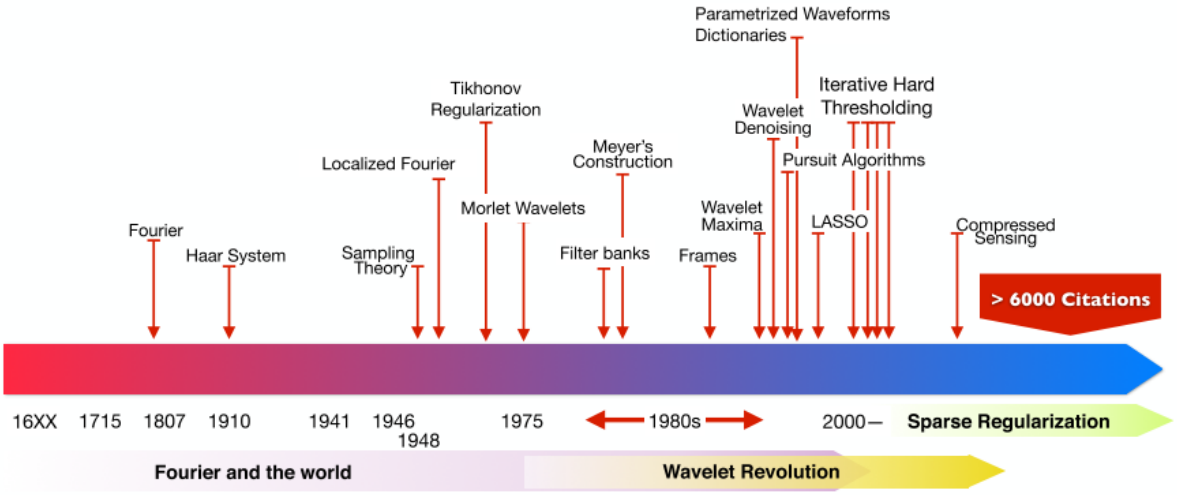

Figure 2.1: *A historical overview of the digital signal processing topics that relates to CS and sparse regularization. It shows that both are novel areas of research and studied extensively from the beginning of the 21st century. Source: [22].*

In *statistics*, simple linear regression is an approach to modeling the relationship between a response variable (the dependent variable) and one or more predictor variables (the independent variables). Many recovery algorithms (or model selection tools as they were called) such as *ridge* and least absolute shrinkage and selection operator (LASSO) regression that appear in CS have their roots in statistics. When statisticians faced a problem where the number of variables N were far greater than the number of measurements M, i.e. $N \gg M$, it could be a daunting task to have to estimate all N parameters to find the response. Fortunately it was discovered that in practice only a few predictors were needed to estimate the true output, as these algorithms or selector tools could be leveraged to shrink the number of variables N by identifying the most relevant ones and disregard the rest. They did so by introducing bias in the estimation of coefficients on purpose to reduce the variability (number of variables) in the estimate. They were generally used when the goal was to predict the response for values of predictors that had not yet been discovered. These methods produced far simpler linear models with less redundant data, removing information that was already represented in the model, and the resulting estimates generally had a lower mean squared error (MSE) than the traditional ordinary least squares (OLS) estimates, particularly when multicollinearity (a predictor in the model can be linearly predicted from another) is present or when overfitting (a resulting estimator that corresponds too closely or exactly to a particular set of training data) is a problem. Especially the *LASSO* regression technique has been adopted by CS.

[1] Superposition is a linear time-invariant (LTI) system's ability to process signals individually and then sum up at the end to process all signals simultaneously.

2.2 Applications of the compressed sensing framework

In the area of *telecommunications*, Candes et al. show in [23] that ℓ_1 minimization can recover a function f of a signal vector reliably after it has been corrupted in a completely arbitrary fashion by some error. They prove that under some suitable conditions for a coding matrix A that encodes f, the input f can be recovered exactly by solving a convex optimization problem, even when large fractions of the signal is corrupted. This discovery is useful in radio and data communications that usually rely on redundancy checks or repetition schemes to make error corrections. In *medicine*, engineers Shental et al. show that CS can be used for screening individuals for known disease alleles [24]. It boils down to identifying a handful of persons from a population of 4000 that potentially carry a known rare allele. This is done by forming a randomized pool of samples used to train a sensing matrix. They are then able to identify the original gene carriers and show, via computer simulations, that CS can recover these alleles in larger groups than what was possible before.

Additionally the area of *radiology*, one that deals with magnetic resonance imaging (MRI), has leveraged CS to reduce the duration of a typical MRI scan by undersampling in the transfer-domain of the MR images [25]. The magnetic field measured is basically a range of Fourier samples that can be inversely transformed to make an image. Traditionally, undersampling would induce aliasing upon reconstruction, but when the image has a sparse basis, it can be recovered efficiently often using a discrete cosine or wavelet transform. The deduction in scan times improve patient comfort and reduce scan costs for the hospital.

It should be clear how CS differs from classical signal sampling. First, instead of sampling a signal at some fixed points in time, CS acquires measurements by calculating the inner products between the signal and some overall sensing or measurement function. The input signal could be an image captured by an MRI scanner, a noisy acoustic impulse response or continuous temperature readings from a sensor. Second, the way signal recovery works is different in the traditional and compressed sensing world: Nyquist-Shannon uses the cardinal sine function or sinc to reconstruct a continuous band-limited signal, where the normalized version of the sinc is a Fourier transform of the rectangular function without scaling as shown in figure 2.2. In CS, recovery is either archived through an iterative algorithm that *greedily* selects pieces of already known data to estimate the output signal or a regularization approach that uses convex optimization to find a minimizer that can estimate the output. Recent work by Tropp et al. have shown that the former can provide excellent results in terms of recovery time and error when compared to the latter [26] [27].

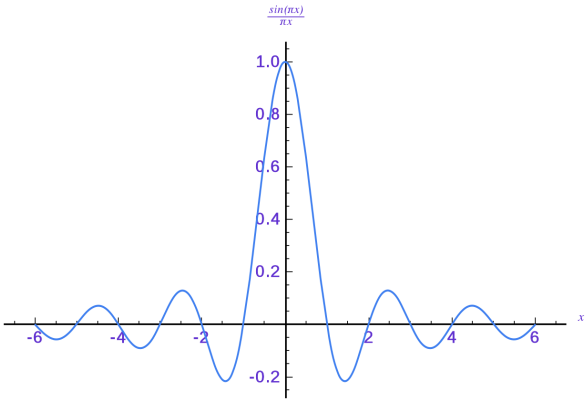

Figure 2.2: *The sinc function traditionally used to reconstruct continuous signals.*

In the recovery phase of CS, state-of-the-art recovery algorithms use methods from sparse approximation to successfully approximate a sparse solution using compressive sampled measurements and something called a dictionary, which is a set of basis functions that can be used to solve a linear system of equations $\boldsymbol{Y} = \boldsymbol{DX}$ by approximating $\boldsymbol{X}$, where $\boldsymbol{Y}$ are sampled measurements, $\boldsymbol{D}$ is the dictionary and $\boldsymbol{X}$ is the solution. We can either chose a predetermined dictionary or train it based on obtained signal samples. The solution $\boldsymbol{X}$ is sparse if it has few nonzero elements compared to the dimensionality of the original signal. This explanation of recovery is very general

and only provided for the reader to grasp the concept and understand the idea. We will go further into compressed sensing and sparse approximation in chapter 3 and relate it to our case.

Lastly in wireless sensor networks, the usage of CS can have a significant value in reducing the power costs of sending and receiving data between nodes, especially large data sets and high-dimensional samples. Moreover the amount of data we can possibly transfer is often bottlenecked by network capacity limitations [28]. Leveraging knowledge from distributed source coding, such as the *Slepian-Wolf* framework for lossless distributed coding, CS can be expanded into distributed compressive sensing (DCS) [29] [30] [31]. The result is going from the primarily intra-signal correlation structure of CS to the intra-signal and inter-signal correlation structure of DCS. Baron et al. study three example models for jointly sparse signals under the assumption that there exists a common sparsity component which is present in all signals [28]. The authors show practical algorithms that can jointly recover multiple signal ensembles from incoherent measurement bases. Their results in simulation show promise in joint decoding over separate decoding as it depends on the sparsity of what is called the common component. They argue that DCS is immediately applicable to a range of problems in sensor arrays and networks as these three joint-sparsity models show advantages over the standard CS framework in a DCS setting. A distributed wireless sensor network (WSN) also sets the scene in [32] where Wang et al. purpose a novel dictionary learning algorithm called online dictionary learning-based compressive data gathering (ODL-CDG) used for online training using compressive data (CS measurements) in a network of nodes. They consider parameters such as the quantity of nodes, transmission range, initial energy and data size. In their simulation results they show that ODL-CDG can enhance the recovery accuracy in the presence of noise, and reduce the energy consumption in network nodes compared to traditional dictionary learning approaches such as K-SVD. Keeping energy consumption low helps enhance the lifetime of the network. Lastly they argue that their online algorithm can outperform predetermined dictionaries like the DCT dictionary or other learning approaches such as K-SVD and CK-SVD however these claims are mainly based on energy consumption and not applicable for a distributed system with a stable power source.

2.3 The need for distributed systems

In recent years we have seen an increase in the number of systems being split up and distributed, as the need for offloading the workload from a single computing entity to multiple has grown. This is also the case for many Internet of Things (IoT) based systems that are either constrained by some physical or logical separation so they have to be distributed. Formally, we denote such a system as one with no global clock and no shared memory that contain one or multiple nodes with their own private memory that run a sequence of sequential or parallel processes. From the first mechanical computers in the late 40's, past Apple's Macintosh in 1984 and when the first smartphones start appearing in the 2000's, to the era of modern cloud computing, we have seen a shift from centralized and single processor computing to distributed with multiple processors involved. This is due to a few reasons: The costs of hardware and power requirements have gone down, see Koomey's law [33], making it more feasibly to link multiple computers instead of buying one large. Modern systems require a high degree of fault tolerance, which distributing a system can assist with. Also it opens up for improved concurrency and parallelism when you can allocate computing tasks to multiple nodes, facilitated by the fact that the storage capabilities of computers are increasing too, see Kryders law [34]. Lastly the number of transistors in integrated circuits does not double about every two years as Moore's law predicts [35], instead it has slowed down and may even fall off in the near future. According to Intel, in 2015 their pace of advancement has slowed, as we saw processors starting at the 22nm feature in 2012 and continuing to 14nm. This leads towards the need for horizontal scaling (adding more nodes) rather than vertical (increasing the computational capabilities of one node) when allocating

computer resources. Modern cloud technologies that we will make use of follow this approach. To summarize, the era of distributed systems is certainly upon us and new ways of utilizing computational resources are still being evolved. It would be a reasonable statement to argue for the need to explore distributed applications and how this can be merged with already established theories and practices.

2.4 Dictionary learning in distributed systems

Dictionary learning is a method to efficiently form what is called a *dictionary* that best represent the structures in your data of interest. A dictionary consists of multiple basis functions that can be used to solve a linear system of equations $\boldsymbol{Y} = \boldsymbol{DX}$ by approximating $\boldsymbol{X}$ and $\boldsymbol{D}$ given $\boldsymbol{Y}$. In this case, $\boldsymbol{Y}$ are sampled measurements of the signal, $\boldsymbol{D}$ is the dictionary we create and $\boldsymbol{X}$ is a signal matrix. The desire to efficiently train a generic dictionary for sparse signal approximation led M. Aharon, M. Elad and A. Bruckstein to develop the K-SVD (K-means and the singular value decomposition together) algorithm in 2006 [11]. It basically works by alternating between (1) sparse approximation of the signals using a fixed dictionary and (2) updating the atoms (columns) in the dictionary based on the latest approximation. The main advantages they argue is that the algorithm is simple, efficient, and converges quickly with respect to other previously proposed dictionary learning methods [11]. In their experiments they recover corrupted natural images using the learned dictionary. Aharon et al. do not assume any compressive technique have been used on the input signals, hence they train on fully-fledged samples. We will see next that there exists alternatives to this approach, which can save memory and reduce processing time without sacrificing accuracy.

A paper by Chouvardas et al. [36] put forward in 2015 extends the research by Aharon et al. and turns to distributed computing. They purpose a novel algorithm for what is called online distributed dictionary learning that can better cope with the mounting data capacity concerns modern applications face. They resort to the same learning philosophy as [15], that is training a distributed field of nodes by sequentially providing them bits of the data, one step at the time, until all data has been learned by the model. In sparse approximation this is applicable when using high-dimensional measurement vectors as training input. Chouvardas et al. form an ad-hoc network of inter-connected nodes that constitutes their decentralized dictionary and computes it cooperatively: Each node obtains its own data locally, does a sparse approximation step using the least-angle regression (LARS) algorithm which computes the LASSO solution followed by a local dictionary update step, a two-step approach that mimics K-SVD, and finally exchanges training results with other nodes in the network. This way the nodes work together to find a new improved dictionary estimate by employing the distributed recursive least-squares (DRLS) assuming a fully-connected network, i.e. a network where there exists a path between any two nodes in the network. This again poses some inter-dependency concerns which are not addressed in the paper and seem to not have been deployed and tested in a real-life context. Using synthetic data, they validate the performance of the combined DRLS and online dictionary learning distributed residual least squares (OnDiRLS) and find that the cooperation among nodes make the OnDiRLS equate the performance of a scheme where all data is centrally available. Though both perform better than a scheme where data is not centrally available and where nodes compute the dictionary without cooperation and act as individual learners. Finally they compare the purposed OnDiRLS and the now widely acclaimed K-SVD algorithm in real image denoising by extracting 8×8 sized patches, a total of 62001, from a noisy 256×256 natural image and randomly distribute the resulting data vectors to five separate nodes for the OnDiRLS case and a single matrix for the K–SVD case. A dictionary is then trained for each method and a sparse approximation is made that removes noise from the original test image. They observe that the performance of the OnDiRLS is slightly better compared to that of the K–SVD when peak signal

to noise ratio (PSNR) is low and that the K–SVD leads to better denoising when PSNR is high. Distributing the operation has apparently improved its robustness to noise.

A cooperative dictionary learning algorithm that targets big data applications is also what H. Raja and W. Bajwa purpose in [9]. Their algorithm, denoted cloud K-SVD is a learning scheme where multiple cloud nodes at geographically-distributed sites collaboratively learn a dictionary to represent a low-dimensional approximation of the total data quantity. That is, the cloud K-SVD creates an adaptive overcomplete dictionary by gathering high-dimensional data from all these sites such that every sample can be represented via a sparse approximation of the dictionary. The cloud K-SVD accomplishes this goal without an exchange of each sample of data between nodes. In order to have a global dictionary the nodes must communicate by a consensus protocol in order to come to an agreement on the importance of each dictionary atom, that is the nodes collaborative exchange and average atoms from their local dictionary until concordance of the global one. Practically the total data set is split between nodes and trained individually by the same two-step approximate-and-learn approach we saw in OnDiRLS before global consensus of the dictionary is attempted by a finite number of iterations to find the best distributed dictionary. Technically, the algorithm uses power iterations to gather the approximation from all neighboring nodes over weighted links and estimate the mutual dominant eigenvectors that span all subspaces of the data. In a synthetic data experiment they find that the convergence behavior of the cloud K-SVD was determined by the number of consensus iterations within each power method iteration and that the cloud K-SVD and centralized (canonical) K-SVD have similar performance but both of them outperform the local K-SVD variant with no inter-node dictionary strengthening teamwork. When it comes to natural image classification of the well-known MNIST[2] database, a solid detection rate for five classes puts cloud K-SVD and centralized K-SVD in front. The overall findings indicate that one should consider information exchange and joint estimation of mutual improved dictionary elements when designing an online learning algorithm. For real-world application, consensus iterations, additional computations and synchronous updates are potential challenges.

Anaraki and Hughes proposed in [37] that the dictionary can be trained from compressed samples as well. They call this novel algorithm compressive K-SVD (CK-SVD) since it uses the traditional K-SVD algorithm with CS measurements instead of fully-fledged samples. They contrast their approach to common sparse approximation approaches, where all data is assumed fully available at training time, though in practice it may not be. The cost of capturing signals or the sheer amount of measurements needed may motivate use of compressive sampling and CK-SVD in their view, so it is basically just a generalization of K-SVD and therefore preserves its convergence properties [37]. In a prearranged test, they show that CK-SVD is able to recover almost all atoms compared to a generated ideal dictionary. They extend their test to noisy input data, and show that CK-SVD is robust in the sense that it can still recover the generated dictionary atoms very well even from noisy CS measurements. Finally they argue that adaptive dictionary learning methods can surpass non-adaptive ones and that compressed measurements can often suffice without loss of information.

One advantage of CK-SVD is that the whole input data or at least a large enough portion of it does not have to be available at training time because it can get by using just compressive measurements. In a real-world scenario, the size of the input data might be too big to even fit in memory or the costs of capturing may be too high. Another technique that can handle such an inconvenience is *online learning*, which essentially suggests iteratively updating the dictionary when new data points become available in a stream. The approach is two-part as in K-SVD: (a) Find a sparse approximation that includes the latest received sample and then (b) update the dictionary via a minimization algorithm. This way we can gradually update the dictionary as

[2]MNIST is a large database of handwritten digits that is often used for benchmarking machine-learning algorithms.

new data becomes ready for sparse approximation and help reduce the amount of memory needed. This factor is the motivation behind work done by Chen et al. [15] where they place only a portion of the dictionary elements on each computer node in a distributed system. The concatenated set of elements from all participating nodes would then constitute the actual dictionary. They argue this distributed online approach is usable in big data scenarios where models are often very large and must be spread over multiple nodes or even over spatially separated locations and cases where it is simply not feasible to aggregate all dictionaries in one place due to communication and privacy concerns. They argue that because of the complexity and size of existing and future learning tasks, it is a given that learned dictionaries become increasingly demanding in terms of memory and processor requirements as well, so future work should focus on scenarios where the dictionary need not be available in a single location but instead spread out over multiple locations, in their view. They conduct a somewhat artificial experiment with 196 logically distributed nodes, a collective dictionary of size 100×196 so one atom for every node, with a 0.2 percent chance that a node is connected to another and 1 million patch samples of size 10×10 from a non-calibrated natural image dataset. Tests are done by denoising a corrupted image using the trained de-centralized dictionary and then compared to a traditional centralized method as in [38]. They present respectable denoising results using their novel distributed model, however it does not take time into account, network communication delay or inter-node dependency concerns (if one node goes offline, what happens to the atom on that node) as the test setup is entirely simulated on a single physical computer.

Chapter 3

Signal processing theory

We will now go through the main theory in our thesis, describe key concepts and methods and compare different models and real-life practices as it lays the groundwork for our design and implementation of cloud K-SVD. We start off in section 3.1 with a look at relevant literature on signal norms originating from mathematics that we use in both compressive sensing (CS) and sparse approximation (SA) theory, then turn our attention to the notion of sparsity, which is a recurring motif in the text, and also look at the properties of a compressible signal and the incoherent basis. We then look more specifically at signal sampling by CS in section 3.2 and describe the definitions and theorems in literature behind concepts like the null space property (NSP) and the restricted isometric property (RIP) criteria. We extend our review to the signal recovery case for compressive sensing in section 3.3 with focus on the sensing basis and measurements used in recovery. These sections conclude the compressed sensing part and is mostly compiled from work done by R. Baraniuk, M. Davenport, M. Duarte, C. Hegde, E. Candès and E. Ollila [39] [40] [19] [41] with all essential topics included.

The next part will focus on sparse approximation theory. In section 3.4 we relate the theories and concepts learned in CS on how they work differently in sparse approximation and change our view slightly, as we are now more interested in accurate recovery protocols from either noiseless or noisy data already sampled than we were in the CS methodology. We provide an introduction to the essence of SA in section 3.5, introduce the multiple-measurement vector model (MMV) and end with a list of requirements that should guide any recovery protocol. Since this thesis is mainly about SA, its theory and applications, we spend a good amount of time here explaining the concept. In section 3.6 we build upon our SA knowledge to introduce the concept of dictionary learning, a working subject of SA that exploits ways of improving what is called a dictionary or codebook used in signal recovery by adaptively learning it using training data. Here we also introduce a major algorithm for dictionary learning called *K-SVD* purposed by Aharon et al. in 2006 [11], an algorithm that constructs the dictionary by having it adapt to the data it is exposed to. This algorithm was originally designed for centralized systems however, so we widen our view and look at consensus theory for distributed systems in section 3.7. Here we examine different protocols for reaching consensus in a distributed system and we incorporate this line of thought into dictionary learning by looking into a protocol for distributed model learning called cloud K-SVD in section 3.8 [9].

3.1 Signal models and Norms

Compressed sensing (CS) builds on two fundamental principles: Sparsity and incoherence. Sparsity expresses the idea that the bandwidth or space used by a continuous signal may be far greater than the actual information it contains, so it occupies more room than it needs. In other words,

we can significantly reduce or compress the signal without any noticeable loss of information and ideally we want a signal with only a few nonzero coefficients. Furthermore we can leverage the fact that many natural occurring signals have a sparse frequency spectrum [40], for example smooth signals are sparse in the Fourier basis and piecewise smooth signals are sparse in a wavelet basis [29]. We say a signal is K-sparse, meaning it has at most K nonzero elements and its support $rsupp(\boldsymbol{X}) = i : \boldsymbol{x}_i \neq 0$ has cardinality less or equal to K. This means a K-sparse signal can be approximated well by a linear combination of a small set of vectors from a known basis. Incoherence is a measure between a measurement matrix $\boldsymbol{\Phi}$ and a orthonormal basis $\boldsymbol{\Psi}$ (Wavelet, Fourier) [42] were we choose to expand our signal in $\boldsymbol{x}$: $\boldsymbol{x} = \boldsymbol{\Psi z}$. As a general point of view, the vector $\boldsymbol{x}$ may contain coefficients of a signal $f \in \mathbb{R}^N$ in some orthonormal basis $\boldsymbol{\Psi}$ with elements $\boldsymbol{\psi}_i$:

$$f(t) = \sum_{i=1}^{N} \boldsymbol{z}_i \boldsymbol{\psi}_i(t) \tag{3.1}$$

For example we can expand the signal as sinusoids, wavelets or n-degree polynomials [43]. We write a decomposition of 3.1 as $\boldsymbol{x} = \boldsymbol{\Psi} f$ where $\boldsymbol{\Psi}$ is an $N \times N$ matrix with waveforms or likewise $f = \boldsymbol{\Psi} \cdot \boldsymbol{x}$. Thus we approximate:

$$\boldsymbol{x} \approx \sum_{i=1}^{K} \boldsymbol{z}_{ni} \boldsymbol{\psi}_{ni} \tag{3.2}$$

where $K \ll N$ and we say $\boldsymbol{x}$ is K-sparse in $\boldsymbol{\Psi}$ and call $\boldsymbol{\Psi}$ the sparse basis.

The elements of $\boldsymbol{\Psi} = \{\boldsymbol{\psi}_i\}_{i \in \mathcal{I}}$ where $\mathcal{I} = 1, 2, \ldots, N$ span the vector space $\mathcal{V} = \mathbb{R}^N$ and are linearly independent, so we can take a vector from the basis and represent a unique part of $\boldsymbol{x}$. A signal with a sparse representation in $\boldsymbol{\Psi}$ must be spread out in the domain it was captured in [40] and in CS, we strive for low coherent pairs given by an incoherence measurement $\mu(\boldsymbol{\Phi}, \boldsymbol{\Psi})$ from 0 to 1 as the largest correlation between any two elements of $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$. The more incoherent the two bases are, the fewer samples are needed. For an N-sample signal that is K-sparse, roughly cK projections of the signal into the incoherent basis are needed to reconstruct it with high success ($c \approx 3$). So instead of sampling N times, only cK measurements are needed and some incoherent basis [29]. For the latter part, independent and identically distributed (i.i.d.) Gaussian vectors often provide a universal measurement matrix that is incoherent with any basis. We say a signal $\boldsymbol{x}$ is compressible if the magnitude of sorted coefficients N of $\boldsymbol{x}$ decay by $N_s \leq C_1 s^{-q}, s = 1, 2, \ldots, n$ [39, chapter 2]. The larger values of q, the faster our coefficients decay and the more we can compress the signal. Sensing basis $\boldsymbol{\Phi}$ represents a dimensionality reduction step, as it transforms a space $\mathbb{R}^N$ into $\mathbb{R}^M$ where $M < N$ to form undersampled linear measurements $\boldsymbol{y} = \boldsymbol{\Phi x}$ that preserve essential information about $\boldsymbol{x}$. Later we want to find a solution for $\boldsymbol{x}$ by using these measurements, our measurement matrix $\boldsymbol{\Phi}$ and norm minimization strategies [39, chapter 3], which section 3.2 explains.

Norms are used to express some measurable strength of a signal or to denote a approximation error between the original signal $\boldsymbol{x}$ and the estimated one $\hat{\boldsymbol{x}}$. If we were to simply map $\boldsymbol{x} \in \mathbb{R}^N$ to $\hat{\boldsymbol{x}} \in \mathbb{R}^{N-1}$, we want to find a $\hat{\boldsymbol{x}}$ that minimizes some approximation error $\| \boldsymbol{x} - \hat{\boldsymbol{x}} \|_p$. The norm of $\boldsymbol{x}$ for p between $[1, \infty]$ defined by:

$$\| \boldsymbol{x} \|_p = \Big(\sum_{i=1}^{N} | x_i |^p \Big)^{1/p} \tag{3.3}$$

This and all subsequent norms are defined in finite-dimensional Hilbert space and taken from Boyd and Vandenberghe's definition in [44, appendix A]. For $p = \infty$ the norm is defined by:

$$\| \boldsymbol{x} \|_\infty = \max_{i=1,2,\ldots,N} | x_i | \tag{3.4}$$

For $p < 1$, the corresponding unit sphere is non-convex, as this norm does not satisfy all the axiomatic requirements of a norm [39, chapter 2] [45], so we call the ℓ_0 a quasinorm:

$$\| \boldsymbol{x} \|_0 = \lim_{p \to 0} \| \boldsymbol{x} \|_p^p = | \, supp(\boldsymbol{x}) \, | \tag{3.5}$$

While this norm satisfies the triangle inequality, $\| u + v \|_0 \leq \| u \|_0 + \| v \|_0$, the homogeneity property is not met: for $t \neq 0, \| tu \|_0 = \| u \|_0 \neq t \| u \|_0$. Lastly we introduce a mixed or matrix norm $\ell_{p,q}$ for $p, q \geq 1$ that can compute the norm of a $M \times N$ matrix $\boldsymbol{A}$ that contains columns $(\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n)$. This norm is useful for measuring the total strength of a signal matrix or as an error estimator between some approximation and the original matrix of signals. It is defined by:

$$\| \boldsymbol{A} \|_{p,q} = \Big(\sum_{j=1}^{n} (\sum_{i=1}^{m} | \, \boldsymbol{a}_{ij} \, |^p)^{q/p} \Big)^{1/q} \tag{3.6}$$

The $\ell_{2,1}$ variant of the p, q is useful for error correction in sparse approximation since the error for each column is not squared, i.e. it applies the ℓ_p norm to rows and ℓ_q norm to the resulting vector. It is defined by:

$$\| \boldsymbol{A} \|_{2,1} = \sum_{j=1}^{n} \| \boldsymbol{a}_j \|_2 = \sum_{j=1}^{n} \Big(\sum_{i=1}^{m} | \, \boldsymbol{a}_{ij} \, |^2 \Big)^{1/2} \tag{3.7}$$

A special case for the $\ell_{p,q}$ norm where $p = q = 2$ is the Frobenius norm, and $p = \infty$ yields the maximum norm. The Frobenius norm is the square root of the sum of the absolute squares of its elements and is defined by:

$$\| \boldsymbol{A} \|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} | \, \boldsymbol{a}_{ij}^2 \, |} = \sqrt{trace(\boldsymbol{A}^* \boldsymbol{A})} \tag{3.8}$$

The trace function returns the sum of diagonal entries of the square matrix $\boldsymbol{A}$. Since $\boldsymbol{A}$ contains columns that can be treated as K independent d-dimensional signals $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_K]$ we can write the Frobenius norm as:

$$\| \boldsymbol{A} \|_F^2 = \sum_{k=1}^{K} \| \boldsymbol{a}_k \|_2^2 \tag{3.9}$$

Figure 3.1 shows the unit sphere spanned by the regular p norms in two-dimensional space. The choice of p has a significant impact on the approximation error. By definition for $p \geq 1$, the unit sphere is convex and centrally symmetric but it is non-convex for $p < 1$, as its epigraph (the possible set of points above the graph of the function, or the curved line shown in figure 3.1) is a non-convex set. If we tried to draw a straight line from one corner to the other it would fall outside the boundaries of the set enclosed by the epigraph. This property of sets and functions allows us to use effective and well-known convex optimization methods to solve problems like least-squares or linear programming. Boyd and Vandenberghe [44, chapter 2] provide a more thorough look at what convexity means in optimization. Finding a sparse vector $\boldsymbol{x}$ is in fact a difficult combinatorial problem, but luckily there exists convex methods to do exactly that, which we will explore in the recovery phase.

The vector space $\mathbb{R}^N$ has the usual inner product:

$$< \boldsymbol{x}, \boldsymbol{z} >= \boldsymbol{z}^T \boldsymbol{x} = \sum_{i=1}^{N} x_i z_i \tag{3.10}$$

The matrix space $\mathbb{R}^{M \times N}$ has a inner product defined as well:

$$< \boldsymbol{X}, \boldsymbol{Z} >= Tr(\boldsymbol{Z}^T \boldsymbol{X}) = \sum_{i=1}^{M} \sum_{j=1}^{N} \boldsymbol{X}_{ij} \boldsymbol{Z}_{ij} \tag{3.11}$$

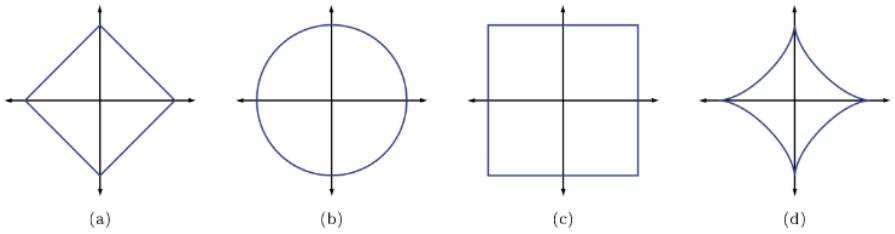

Figure 3.1: *Unit spheres in $\mathbb{R}^2$ for ℓ_p norms. (a) shows ℓ_1 norm for $p = 1$, (b) shows ℓ_2 norm for $p = 2$, (c) shows ℓ_∞ norm for $p = \infty$ and (d) shows the ℓ_p quasinorm for $p = \frac{1}{2}$. Source: [39]*

.

Figure 3.2 shows the approximation from a point $\boldsymbol{x}$ in $\mathbb{R}^2$ to $\hat{\boldsymbol{x}}$ in $\mathbb{R}^1$ with various ℓ_p norms. In compressed sensing theory, the norm of vectors and matrices plays an important role in terms of sparsity and dimensionality reduction. If p is large, it will spread out the error more evenly between coefficients, but if p is small, it will put more weight on a single or few coefficients and visually the edges will be more pointy as in figure 3.1. A signal's compressibility is determined by the ℓ_p space it belongs to. If $\boldsymbol{X}$ is a finite sequence of samples $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$, it only has a representation ℓ_p space for some value of p if its norm is finite. As p get smaller, the elements of $\boldsymbol{x}$ must decay faster to converge for the norm to be bounded and the size of ℓ_p decreases as well. See figure 3.1. Recall for $p = 0$ it merely counts the number of non-zero values.

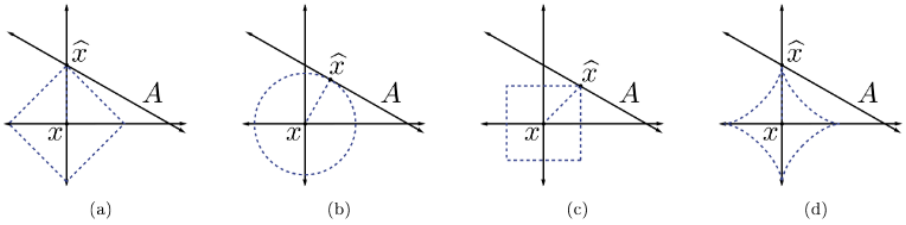

Figure 3.2: *The approximation of a point in $\mathbb{R}^2$ to a subspace A in $\mathbb{R}^1$ using various ℓ_p norms. (a) shows approximation in ℓ_1 norm, (b) shows it in the ℓ_2 norm, (c) shows it in the ℓ_∞ norm and (d) shows it in the ℓ_p quasinorm.*

It is tempting to apply the seemingly effective ℓ_0 quasinorm to find a sparse solution $\hat{\boldsymbol{x}}$ as it simply counts the number of nonzero entries in $\hat{\boldsymbol{x}}$. When p goes to zero, the curve of the norm becomes an indicator function that returns 0 for $\boldsymbol{x}_i = 0$ and 1 for every other value. However this problem is known to be NP-hard, because a ℓ_0 minimization problem would have to iteratively try each possible combination of some coefficient set to zero in a brute-force manner to find a solution. All combinations of the $\frac{N}{K}$ possible sparse subspaces must be evaluated. Moreover in a real scenario, a signal vector would rarely be representable by a vector of coefficients containing many zeros [chapter 1.7][46]. With the ℓ_1 and ℓ_2 norms we get a sum of all the elements that

can be minimized with a cost function $J(\boldsymbol{x}\hat{\boldsymbol{x}})$. Donoho has shown that the ℓ_1 is in fact a good approximation of the ideal ℓ_0 norm [47]. Later in the signal recovery section we will see why the ℓ_1 and ℓ_2 norms are obvious candidates for solving minimization problems in terms of single and multiple signal vector recovery.

3.2 Signal sampling in compressed sensing

The protocols we design to sense and sample sparse continuous-time signals must capture essential information and compress it effectively without trying to comprehend the entire signal [40, p. 2]. Certain rules govern how we should acquire M linear measurements given a signal $\boldsymbol{x} \in \mathbb{R}^N$ so that they can be recovered at a later time. A place to start would be transform coding, a subcategory of data compression that samples the entire signal with regards to Nyquist rate, then transforms it to a known basis where it is sparse or compressible. Upon transmission, only the significant coefficients of the signal would be sent. This technique is used in some applications, but we want to focus on compressed sampling where we do not sample at the Nyquist rate for reasons explained in chapter 2. Consider a finite signal vector $\boldsymbol{x} \in \mathbb{R}^N$, it can be represented as a linear combinations of columns taken from a basis $\boldsymbol{\Psi} \in \mathbb{R}^{N \times N}$ as $\boldsymbol{x} = \boldsymbol{\Psi z}$ where $\boldsymbol{z}$ is the data of interest, usually not compressed, but presumably sparse in nature. It is now represented by $\boldsymbol{x}$ in the basis $\boldsymbol{\Phi}$. For us to leverage compressed sensing at this point we must find a known pre-determined basis where the signal $\boldsymbol{x}$ is sparse, then sample it by a $M \times N$ measurement matrix $\boldsymbol{\Phi}$ with $M \ll N$ to get measurements:

$$\boldsymbol{y} = \boldsymbol{\Phi x} = \boldsymbol{\Phi \Psi z} \tag{3.12}$$

We say $\boldsymbol{\Phi}$ represents dimensionality reduction as it maps $\mathbb{R}^N$ to $\mathbb{R}^M$ with M much smaller than N. It is obvious at this point that $\boldsymbol{\Phi}$ is underdetermined, i.e it has a lot more columns than rows, so any reconstruction of $\boldsymbol{x}$ from $\boldsymbol{y}$ measurements is ill-posed unless we apply some restrictions on $\boldsymbol{\Phi}$ which includes the spark of $\boldsymbol{\Phi}$, the null property (NSP), the restricted isometry property (RIP) and bounded coherence to guarantee uniqueness in recovery. We will briefly review the theory here, but more details are found in [39].

First we denote the null space of $\boldsymbol{\Phi}$ by:

$$\mathcal{N}(\boldsymbol{\Phi}) = \{\boldsymbol{z} : \boldsymbol{\Phi z} = \vec{0}\} \tag{3.13}$$

To recover all sparse signals $\boldsymbol{x}$ from measurements $\boldsymbol{y} = \boldsymbol{\Phi x}$, then for any pair of distinct signal vectors $\boldsymbol{x}, \hat{\boldsymbol{x}} \in \sum_K = \{\boldsymbol{x} : \| \boldsymbol{x} \|_0 \leq K\}$ the two measurements $\boldsymbol{\Phi x}$ and $\boldsymbol{\Phi}\hat{\boldsymbol{x}}$ must be distinct as well, otherwise they would map to the same signal. Put formally, if we consider $\boldsymbol{\Phi x} = \boldsymbol{\Phi}\hat{\boldsymbol{x}}$ then $\boldsymbol{\Phi}(\boldsymbol{x} - \hat{\boldsymbol{x}}) = 0$ with $\boldsymbol{x} - \hat{\boldsymbol{x}} \in \sum_{2K}$, then $\boldsymbol{\Phi}$ uniquely represents all $\boldsymbol{x} \in \sum_K$ if and only if $\mathcal{N}$ contains no vectors in $\sum_{2K}$ by [section 3.2][39]. This property can be categorized as *the spark* by definition:

Definition 1 *The spark of a given matrix $\boldsymbol{\Phi}$ is the smallest number of columns of $\boldsymbol{\Phi}$ that are linearly dependent.*

Then the following theorem holds by proof in [39, section 3.2.1]:

Theorem 1 *For any vector $\boldsymbol{y} \in \mathbb{R}^M$, there exists at most one signal $\boldsymbol{x} \in \sum_K$ such that $\boldsymbol{y} = \boldsymbol{\Phi x}$ if and only if $spark(\boldsymbol{\Phi}) < 2K$.*

This leads to the requirement that $M \geq 2K$ where $2K = cK$ for $c = 2$ as we explained in the start of section 3.1. The spark property works for exactly sparse signals, however if they are only

approximately sparse the null space property (NSP) has to be introduced on $\mathbf{\Phi}$ by [39, section 3.2.2]. We will not explain the exact details, though the vectors in the null space of $\mathbf{\Phi}$ should be separated on a large set of indices, so that we can uniquely recover a sparse $\boldsymbol{x}$.

If the measurements are subjected to noise, the matrix $\mathbf{\Phi}$ needs to satisfy the restricted isometry property (RIP) which is a more rigorous version of the spark and NSP as shown by proof in [39, section 3.4]. RIP ensures spatial separation of any pair of K sparse vectors in $\mathbf{\Phi}$, such that the dimensionality reduction step is an isometric mapping [39, section 2.3.1.1].

Definition 2 *A matrix $\mathbf{\Phi}$ satisfies the restricted isometry property (RIP) of order K if there exists a $\delta_K \in (0,1)$ such that*

$$1 - \delta_K \leq \frac{\| \mathbf{\Phi} x \|_{2,2}}{\| \boldsymbol{x} \|_{2,2}} \leq 1 + \delta_K \tag{3.14}$$

holds for all $\boldsymbol{x} \in \sum_K = \{\boldsymbol{x} : \| \boldsymbol{x} \|_0 \leq K\}$

If a matrix $\mathbf{\Phi}$ satisfies definition 2 for order $2K$, then by 2 we interpret that $\mathbf{\Phi}$ preserves the distance between any K sparse vectors. This ensures that a variety of algorithms will be able to recover a sparse signal $\boldsymbol{x}$ from noisy measurements $\boldsymbol{y}$. The task is now to form matrices of size $M \times N$ that satisfy the RIP of order K. These can be built deterministically as shown in [48] where $M = O(K^2 \log N)$ and [49] where $M = O(KN^\alpha)$ which unfortunately require M to be quite large. For any real world application, we want to make M as small as possible. This can be archived by forming randomized measurement matrices that satisfy the RIP and incoherence with high probability if M is sufficiently large using:

- Gaussian measurements: The $M \times N$ sensing matrix $\mathbf{\Phi}$ takes entries from a independently sampled Gaussian distribution with zero mean and variance M^{-1} [section 3.4][19].
- Binary measurements: $\mathbf{\Phi}$ has independently sampled elements from a symmetric Bernoulli distribution $P(\mathbf{\Phi}_{mi} = \pm 1/\sqrt{M})$ [section 3.4][19].
- Fourier measurements: $\mathbf{\Phi}$ has elements obtained from the discrete Fourier transform (DFT) by selecting M rows uniformly at random [section 3.4][19].
- Incoherent measurements: $\mathbf{\Phi}$ is obtained by selecting M rows uniformly at random from an $N \times N$ orthonormal matrix $U = \mathbf{\Phi}\mathbf{\Psi}^*$ with unit norm. U maps the signal from the $\mathbf{\Psi}$ to the $\mathbf{\Phi}$ domain. The coherence between the measurement/sparsity basis ($\mathbf{\Phi}$, $\mathbf{\Psi}$) is called the mutual coherence. It is defined by:

$$\mu(\mathbf{\Phi}, \mathbf{\Psi}) = \sqrt{N} \max_{i,j} |< \boldsymbol{\phi}_i, \boldsymbol{\psi}_j >| \tag{3.15}$$

Random matrices provide several advantages over deterministic ones. Firstly, the measurements are *democratic*, meaning we can recover a signal using a large enough subset of measurements. This makes the recovery phase more robust to noise and loss. Secondly, if we assume $\boldsymbol{x}$ is sparse in some basis $\mathbf{\Psi}$, we require that the product $\mathbf{\Phi}\mathbf{\Psi}$ satisfies the RIP. If $\mathbf{\Phi}$ was deterministic, we would need to take $\mathbf{\Psi}$ into account in our construction of $\mathbf{\Phi}$, but not when it is chosen randomly. If $\mathbf{\Phi}$ is drawn from a Gaussian distribution and $\mathbf{\Psi}$ is a basis, then $\boldsymbol{A}\mathbf{\Phi}\mathbf{\Psi}$ will also be a Gaussian distribution that will satisfy the RIP if M is sufficiently large [39, section 3.5].

Lastly we should impose a condition on $\mathbf{\Phi}$ that guarantees uniqueness in recovery. The coherence of a matrix can provide such property given $\mathbf{\Phi}$ and is defined by:

Definition 3 *The coherence of a matrix $\mathbf{\Phi}$, $\mu(\mathbf{\Phi})$, is the largest absolute inner product between any two columns $\boldsymbol{\phi}_i, \boldsymbol{\phi}_j$ of $\mathbf{\Phi}$:*

$$\mu(\mathbf{\Phi}) = \max_{i \leq i < j \leq N} \frac{|< \boldsymbol{\phi}_i, \boldsymbol{\phi}_j >|}{\| \boldsymbol{\phi}_i \|_2 \| \boldsymbol{\phi}_j \|_2} \tag{3.16}$$

The coherence is always in the range $\mu(\mathbf{\Phi}) \in \left[\sqrt{\frac{N-M}{M(N-1)}}, 1\right]$, where the lower bound is known as the Welch bound. When $N \gg M$ the lower bound is approximately $\mu(\mathbf{\Phi}) \geq 1/\sqrt{M}$. Often this coherence property can be easily computed compared to criteria such as the spark, NSP and the RIP which essentially require us to consider $\binom{N}{K}$ submatrices. Coherence can be related to said criteria as well by the Gershgorin circle theorem used in [section 3.6][39] leading to:

Lemma 1 *The spark condition. For any matrix $\mathbf{\Phi}$, it holds that:*

$$spark(\mathbf{\Phi}) \geq 1 + \frac{1}{\mu(\mathbf{\Phi})} \tag{3.17}$$

By merging theorem 1 with lemma 1 we can make the following condition on $\mathbf{\Phi}$ that guarantees uniqueness with proofs in [section 3.6][39]:

Theorem 2 *The guarantee for uniqueness in recovery. If it holds that:*

$$K < \frac{1}{2}\Big(1 + \frac{1}{\mu(\mathbf{\Phi})}\Big) \tag{3.18}$$

then for each measurement vector $\boldsymbol{y} \in \mathbb{R}^M$ there exists at most one signal $\boldsymbol{x} \in \sum_K$ such that $\boldsymbol{y} = \mathbf{\Phi}\boldsymbol{x}$.

Theorem 2 together with the Welch bound puts an upper bound on the sparsity parameter K that guarantees uniqueness in recovery with coherence: $K = O(\sqrt{M})$. For a low coherence μ, K can be relatively large, whereas K has to be small for a high coherence. A low coherence implies a high incoherence which implies a smaller number of samples have to be acquired and vice versa. The RIP can be related to the coherence property as well by this lemma [39, section 3.6]:

Lemma 2 *If $\mathbf{\Phi}$ has unit-norm columns and coherence $\mu = \mu(\mathbf{\Phi})$, then $\mathbf{\Phi}$ satisfies the RIP order K with $\delta = (K-1)\mu$ for all $K < 1/\mu$.*

It becomes clear that K and M are dependent upon each other and the coherence of $\mathbf{\Phi}$. Studies have been done to compare deterministic and randomized matrices for $\mathbf{\Phi}$ to obtain a lower bound of $\mu(\mathbf{\Phi}) = 1/\sqrt{M}$, such as the Gabor frame generated from the Alltop sequence [50]. The idea is to restrict the number of measurements needed to recover a K-sparse signal to $M = O(K^2 logN)$. The squared dependency on K can be addressed with an assumption of average-case/probabilistic behavior in recovery and a probabilistic prior on the set of K-sparse signals $\boldsymbol{x} \in \sum_K$ [51]. Then if $\mathbf{\Phi}$ has low coherence by $\mu(\mathbf{\Phi})$ and spectral norm by $\| \mathbf{\Phi} \|_2$ and if $K = O(\mu^2(\mathbf{\Phi}) logN)$, then $\boldsymbol{x}$ can be recovered from measurements $\boldsymbol{y} = \mathbf{\Phi}\boldsymbol{x}$ with high probability. We can replace the Welch bound to obtain $K = O(M \log N)$ which is a linear dependency bound on measurements [39, section 3.6].

3.3 Signal recovery in compressed sensing

A signal recovery problem seeks a signal vector $\hat{\boldsymbol{x}}$ or signal matrix $\hat{\boldsymbol{X}}$ approximation from a small number of linear measurements $\boldsymbol{y} = \mathbf{\Phi}\boldsymbol{x}$ with some computational cost. The cost can be the number of K vectors extracted from sparse basis $\mathbf{\Phi}$, in other words the sparsity of $\boldsymbol{x}$, how long it takes to complete the recovery or an error approximation term. In a realistic and practical scenario said measurements are often imperfect, contain noise and are not exactly sparse. In any practical case, we strive for a recovery procedure that is said to be *stable*: small changes in the measurements must reflect small changes in the restored signal. This clearly depends on a

number of factors such as original signal sparsity, contaminated noise in the input, the number of measurements and their disparity. We will set the scene for both noiseless and noisy signal recovery via sparse approximation in the next sections.

This section will mainly detail the recovery phase of compressed sensing, whilst next section will explain sparse approximation theory.

An illustrated example of a compressed sensing case can be seen in figure 3.3. Step 1 (red arrow) is to transform the data of interest $\boldsymbol{z}$, with K-sparsity, into useful or readable data $\boldsymbol{x}$, sparsity level being kept. This can be done using the orthonormal basis $\boldsymbol{\Psi}$ of choice. Step 2 (yellow) is to compress the useful data $\boldsymbol{x}$ into a compressed representation $\boldsymbol{y}$ composed only of nonzero elements. This can be done using the measurement matrix $\boldsymbol{\Phi}$ which is randomly generated to maintain restricted isometric property (RIP). In step 3 (blue arrow), the compressed data can then be stored or transfered depending on the scenario. In step 4 (green arrow), sparse recovery of the data $\hat{\boldsymbol{x}}$ can be done, when data is needed by utilizing a sparse approximation algorithm.

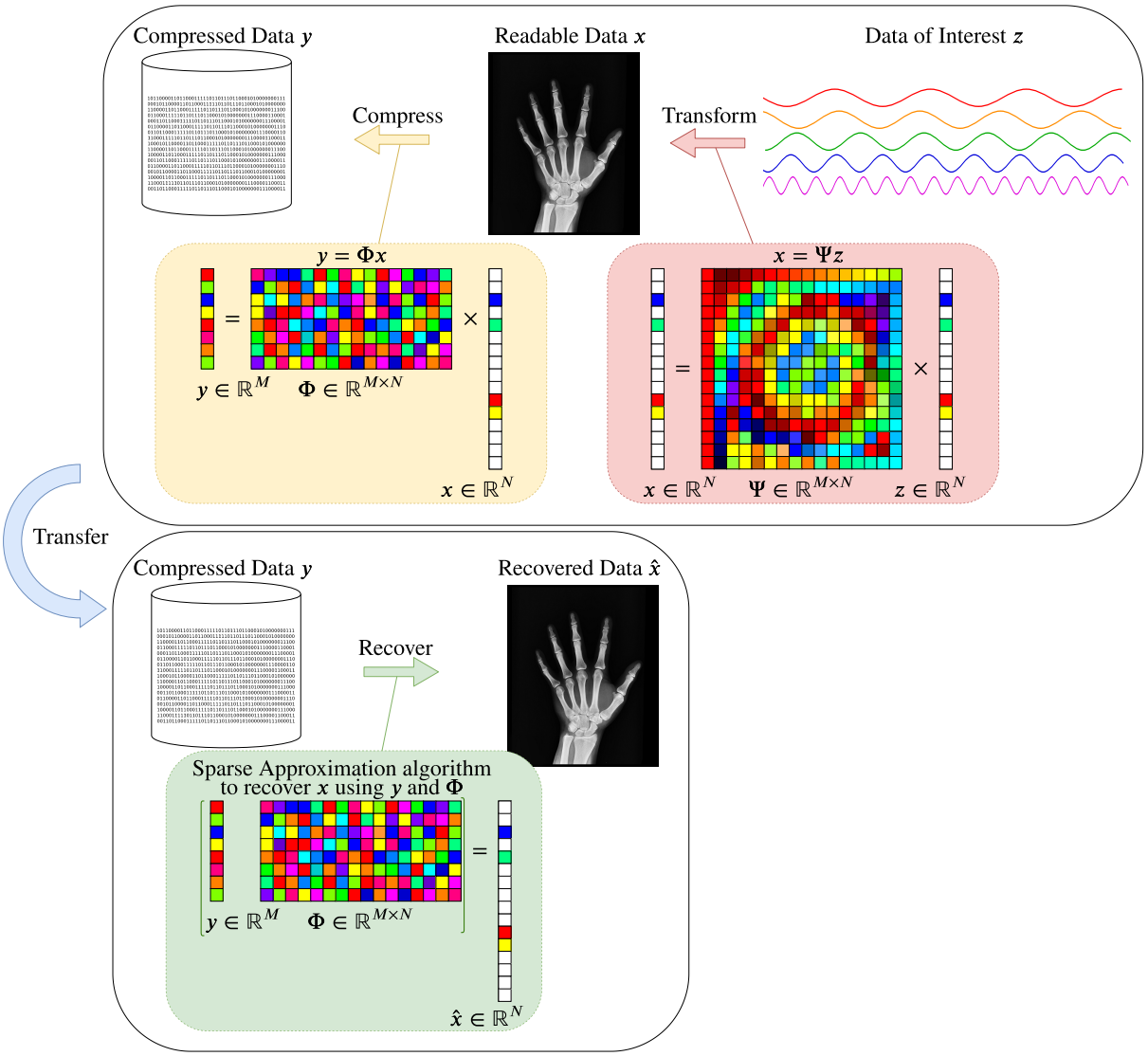

Figure 3.3: *An illustration of all the steps done in compressed sensing.*

Simple signal recovery have two facets that are characteristic of both compressed sensing and sparse approximations:

1. A signal vector $\hat{\boldsymbol{x}} \in \mathbb{R}^N$ is approximated using a linear combination of vectors from a measurement matrix $\boldsymbol{\Phi}$ and the measurement $\boldsymbol{y}$.

2. The procedure finds a compromise between the approximation error and the number of indices or K signals in $\boldsymbol{x}$ that are nonzero.

In an ideal scenario, we want to observe all N coefficients of the signal, but often only a subset of these are available for observations $\boldsymbol{y}_i =< \boldsymbol{x}, \boldsymbol{\phi}_i >$ where $i \in M$ and $M \subset 1, 2, \ldots, N$ with $M \ll N$. With this information we can attempt to recover $\hat{\boldsymbol{x}}$ by $\boldsymbol{y} = \boldsymbol{\Phi}\hat{\boldsymbol{x}}$, where $\hat{\boldsymbol{x}}$ is a solution to an optimization problem like:

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \parallel \boldsymbol{x} \parallel_0 \text{ subject to } \boldsymbol{x} \in B(\boldsymbol{y}) \tag{3.19}$$

where $B(\boldsymbol{y})$ is an added constraint on $\hat{\boldsymbol{x}}$ that ensures it is consistent with measurements $\boldsymbol{y}$.

$\parallel \boldsymbol{x} \parallel_0$ returns the ℓ_0 norm which simply counts the number of nonzero coefficients in $\boldsymbol{x}$, also known as the sparsity constraint, so 3.19 simply gives the most sparse signal consistent with the measurements. We assume that $\boldsymbol{x}$ is approximately sparse. If the input is noise free, by [39, section 4.1] we set $B(\boldsymbol{y}) = \boldsymbol{x} : \boldsymbol{\Phi x} = \boldsymbol{y}$. If the input contains bounded noise, we set $B(\boldsymbol{y}) = \boldsymbol{x} :\parallel \boldsymbol{\Phi x} - \boldsymbol{y} \parallel_p \leq \epsilon$ with $p = 1, 2, \infty$. We can further expand this equation as $\boldsymbol{x} = \boldsymbol{\Psi z}$ which yields $B(\boldsymbol{y}) = \boldsymbol{x} : \boldsymbol{\Phi\Psi z} = \boldsymbol{y}$ or $B(\boldsymbol{y}) = \boldsymbol{x} :\parallel \boldsymbol{\Phi\Psi z} - \boldsymbol{y} \parallel_p \leq \epsilon$. The latter problem is often called the LASSO after [52]. As explained in section 3.1, solving the objective function in 3.19 is nonconvex and finding a minimum is NP-hard. Instead we make the problem more tractable by replacing the $\parallel \cdot \parallel_0$ quasinorm with the convex $\parallel \cdot \parallel_1$ version as in [39, section 4.1]. Using the ℓ_1 furthermore promotes sparsity in the solution as accounted for in section 3.1 and better eliminates white noise in higher dimensions than ℓ_2, we shall rephrase 3.19 as:

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \parallel \boldsymbol{x} \parallel_1 \text{ subject to } \boldsymbol{x} \in B(\boldsymbol{y}) \tag{3.20}$$

If $B(\boldsymbol{y})$ is also convex, then 3.20 is solvable using known optimization techniques. For all possible solutions $\boldsymbol{y} = \boldsymbol{\Phi}\hat{\boldsymbol{x}}$, any technique would choose those that minimize the ℓ_1 norm of $\hat{\boldsymbol{x}}$. It can be proven that if $\boldsymbol{y}$ is adequately sparse, then the approximation $\hat{\boldsymbol{x}}$ via ℓ_1 minimization is exact in a noiseless setting as if we knew the K largest coefficients beforehand [40, p. 3]. This leads to theorem 3:

Theorem 3 *If we assume the sequence of coefficients in $\boldsymbol{y} \in \mathbb{R}^N$ formed by $\boldsymbol{x}$ in a basis $\boldsymbol{\Psi}$ is K-sparse and $\boldsymbol{\Phi}$ is sampled uniformly at random for M measurements, then if*

$$M \geq C \cdot \mu^2(\boldsymbol{\Phi}, \boldsymbol{\Psi}) \cdot K \cdot \log N \tag{3.21}$$

is true for some positive constant C, the solution to 3.20 is exact with high probability [40, p. 4-6].

It follows suit with a need for a low squared coherence between the sensing matrix and the orthonormal basis. If $\mu(\boldsymbol{\Phi}, \boldsymbol{\Psi})$ is close to one, then on the order of $K \log N$ samples are enough instead of N [40, p. 4-6]. Also the number of samples suggested by 3 do not have to be carefully selected as almost any sample set of this size will work. In other words: Sample nonadaptively in a incoherent basis and use linear programming after sampling to restore an approximation of the signal. This basically encodes the data in compressed form then decodes or decompress it using ℓ_1 minimization. There exists cases where a signal f would vanish in the sparse basis $\boldsymbol{\Psi}$ as described by Candès et al. in [40, p. 5], however the theorem ensures that the portion of sets for which exact recovery does not occur are small and unimportant. As long as the sampling size is sufficiently large, the risk of failure is zero.

3.4 The transition to sparse approximation

Sparse approximation (SA) is a subcomponent of compressed sensing and finds use in especially denoising applications of acoustic impulse responses (speech) signals, image restoration and data compression for later recovery in communications systems where bandwidth and throughput is a concern. The following sections of this thesis will mainly deal with SA in theory and practice, but often relate that back to compressed sensing. The theory and techniques we accounted for in section 3.3 is the foundation of sparse approximation, but there are some differences between the world of CS and SA in theory and concept that are stated here:

- In CS, the signal vector or matrix is approximated using a underdetermined measurement matrix $\boldsymbol{\Phi}$. In SA, we use a overcomplete $M \times N$ dictionary $\boldsymbol{D}$ containing N atoms to solve an underdetermined systems of equations $\boldsymbol{y} = \boldsymbol{D}\boldsymbol{x}$. The system has more unknowns than equations, ie. fewer rows than columns.
- In CS, the measurement matrix contains elements from an independently sampled Gaussian distribution. In SA, the atoms in the dictionary are selected to resemble the structures of the input signal and it can be either predetermined or trained from a set of realizations of the real signal. A predetermined $\boldsymbol{D}$ can be constructed using the same types of measurements as in CS (Wavelet, Fourier etc.), where training usually happens at runtime. This is also called dictionary learning.
- In CS, $\boldsymbol{\Phi}$ is nondeterministic and random in nature. In SA, the dictionary $\boldsymbol{D}$ is often a deterministic, unalterable matrix, which leads to different considerations and analysis than with CS. As an example of SA, in statistical model selection, the matrix contains response variables from a statistical study (answers, feedback) with some sparsity since only a few regressors (columns) are significant.

3.5 Sparse approximation

Sparse approximation is a technique to find good recovery of unknown signal vectors that are known to have a sparse representation in some domain [53] [54] [41]. In this context, sparse means that the length of the original signal is larger than the number of non-zero coefficients in the recovered version. Input signals can be sampled from a variety of sources, like acoustic signal bands, a large set of natural image patches or compressed data that have been undersampled from a medical imagining system. Sparse approximation is done by using various linear combinations of the same elementary signal, that typically model different coherent structures in the input, so that we can transform the original signal in an appropriately recovered sparse version, using as few of these elementary signal vectors called atoms as possible [53] [54]. We typically choose the atoms from a large linearly dependent collection, called a dictionary, that can be either predetermined or trained based on the input signals [55] [11]. Using a predefined dictionary of elementary signal vectors, such as the Fourier, Haar or Wavelet bases, are sufficient in most recovery cases and often simple to deploy. However a dictionary that has been trained on a sample set of some data that shares the structure of that it needs to recover can improve the speed and accuracy in a recovery problem compared to a traditional predefined dictionary [56] [11].

Sparse approximation finds use in areas like denoising, image compression, transform coders like JPEG2000[19] or classification [39] where we want to reduce noise in a recovered signal through sparsity, apply compression to only preserve the most significant coefficients in a signal or correctly classify a sequence of letters based on learned sparse signals. These methods are useful in big data scenarios like many modern machine-learning based applications or in systems that process data from social networks.

Also computers that rely on battery power and have high networks communications cost, like a wireless sensor network (WSN), could benefit from only sending and receiving signals that have been made sparse yet still contains all the information necessary [28]. Today's world sees a wide usage of sensors and Internet of things (IoT)-technology in construction work, in the health care sector and in everyday consumer amusement, that demands low battery consumption, a high throughput and real-time data feeds. Methods are needed that can reduce the complexity of ever-growing signals whilst maintaining transparency and essential information structures. To understand how sparse approximation works, we first look at linear inverse problems.

Linear inverse problems is an emerging field in mathematics and engineering as it allows us to understand the parameters and aspects of problems, that cannot directly be observed. An inverse problem starts with a set of observations (effects) and then calculates the factors that produced them (causes). The inverse is called a forward problem. Interestingly, these problems often cannot provide exact solutions from estimates, as they are not computationally viable. We can therefore rely on techniques that can either reduce the dimensionality of these unknown parameters, lower the number of iterations required or find an approximation of the exact solution that we can accept. In digital signal processing, sparse approximation (SA) is an inverse problem that starts with the observations $\boldsymbol{y}$ and a predetermined or trained knowledge base about the original signal, called the dictionary $\boldsymbol{D}$, and then approximates $\hat{\boldsymbol{x}}$ as an approximation of $\boldsymbol{x}$ by a linear combination of elementary or trained signals in this dictionary. We do this to either compress the original signal $\boldsymbol{x}$ by saving only its nonzero coefficients $x_1, x_2, \ldots, x_K$ given that $\boldsymbol{x} \in \mathbb{R}^N$ has a sparse representation in a lower dimension or to reliable recover this approximation $\hat{\boldsymbol{x}}$ of $\boldsymbol{x}$. If $\boldsymbol{x}$ has been subjected to random induced noise, jitter from a faulty radio receiver or other kinds of degradation we want to recover a noiseless version by finding the solution $\hat{\boldsymbol{x}}$ to a ℓ_0 minimization problem subject to $\boldsymbol{y} = \boldsymbol{D}\hat{\boldsymbol{x}}$ as a noiseless version of x, assuming it has a sparse representation. In the multi measurement vector (MMV) model by [41], we have $Q > 1$, let $\boldsymbol{Y} = [\, \boldsymbol{y}_1 \; \boldsymbol{y}_2 \; \cdots \; \boldsymbol{y}_Q \,]$ denote the observed data vector as:

$$\boldsymbol{Y} = \boldsymbol{D}\boldsymbol{X} + \boldsymbol{E} \tag{3.22}$$

$$\text{i.e.} \quad \boldsymbol{y}_i = \boldsymbol{D}\boldsymbol{x}_i + \boldsymbol{\epsilon}_i \quad i = 1, 2, \ldots, Q \tag{3.23}$$

$$\text{where} \quad \boldsymbol{D} = \begin{bmatrix} \boldsymbol{d}_1 & \boldsymbol{d}_2 & \cdots & \boldsymbol{d}_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{d}_1 & \boldsymbol{d}_2 & \cdots & \boldsymbol{d}_M \end{bmatrix}^T \tag{3.24}$$

Dictionary $\boldsymbol{D}$ usually has fewer row vectors than column vectors and is overcomplete, i.e. it contains redundant data, $\boldsymbol{X} = [\, \boldsymbol{x}_1 \; \boldsymbol{x}_2 \; \cdots \; \boldsymbol{x}_Q \,]$ are the unobserved signal vectors and $\boldsymbol{E} = [\, \boldsymbol{\epsilon}_1 \; \boldsymbol{\epsilon}_2 \; \cdots \; \boldsymbol{\epsilon}_Q \,]$ are unobserved noise vectors. In the MMV model [41], we write the matrix model of our system of equations as $\boldsymbol{Y} = \boldsymbol{D}\boldsymbol{X} + \boldsymbol{E}$ which can be formulated as a minimization problem:

$$\min_{\boldsymbol{X} \in \mathbb{R}^{N \times Q}} \| \boldsymbol{Y} = \boldsymbol{D}\boldsymbol{X} \|_2 \leq \epsilon \quad \text{subject to} \quad \forall i, \| \boldsymbol{x}_i \|_0 \leq K \tag{3.25}$$

is the general linear problem with an exact constraint that is often relaxed:

$$\min_{\boldsymbol{X} \in \mathbb{R}^{N \times Q}} \| \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X} \|_2 \leq \epsilon \quad \text{subject to} \quad \forall i, \| \boldsymbol{x}_i \|_0 \leq K \tag{3.26}$$

where $\boldsymbol{Y} = [\, \boldsymbol{y}_1 \; \boldsymbol{y}_2 \; \cdots \; \boldsymbol{y}_Q \,] \in \mathbb{R}^{M \times Q}$, $\boldsymbol{X} = [\, \boldsymbol{x}_1 \; \boldsymbol{x}_2 \; \cdots \; \boldsymbol{x}_Q \,] \in \mathbb{R}^{N \times Q}$.

3.26 is the conventional least-squares (LS) minimization problem of the residual matrix. Such problems as 3.25 and 3.26 are well studied in literature and there exist many practical methods to solve them. When only a single signal vector is present ($Q = 1$), the MMV model reduces to the single measurement model (SSA) used in traditional CS and SA. The goal in MMV is similar to standard signal recovery in section 3.3, but instead of recovering the sparse signals $\boldsymbol{x}_i$ separately,

we attempt to simultaneously/jointly recover all signals at once [41]. The key assumption is that the nonzero values of $\boldsymbol{X}$ occur at a mutual location set, i.e. the matrix $\boldsymbol{X}$ is assumed to be K-rowsparse, which means at most K rows in $\boldsymbol{X}$ are nonzero. Formally put, the row-support of $\boldsymbol{X}$ is less or equal to K:

$$rsupp(\boldsymbol{X}) = i \in 1, 2, ..., N : \boldsymbol{x}_{ij} \neq 0 \text{ for some } j \leq K \tag{3.27}$$

We have previously denoted this as the quasinorm of $\boldsymbol{X}$ that counts the number of nonzero elements. With joint estimation we can potentially get better recovery results by leveraging the assumption that signal vectors share a common support set [53] [54]. Our objective can then be accordingly formulated using the MMV model [41]: Find a row sparse approximation of the signal matrix $\boldsymbol{X}$ based on the matrix $\boldsymbol{Y}$, the dictionary $\boldsymbol{D}$ and the sparsity level $\boldsymbol{K}$. The optimization problem in 3.26 that does so is NP-hard, therefore approximation algorithms have been developed that exploit the joint sparsity in different ways. Two notable categories are convex optimization methods (basis pursuit denoising, in short BPDN) and greedy pursuit algorithms. The first method considers problems in penalized (Lagrangian) form and replaces the nonconvex ℓ_0-quasinorm in 3.26 of the signal matrix with the convex $\ell_{p,1}$ norm, which enforces row sparsity in the solution. For an appropriate Lagrange multiplier λ, we then turn the solution to 3.26 into an unconstrained optimization problem with the updated matrix norm either using $\ell_{\infty,1}$ as Tropp et al. did in [53]:

$$\min_{\boldsymbol{X} \in \mathbb{R}^{N \times Q}} \| \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X} \|^2 + \lambda \| \boldsymbol{X} \|_{\infty,1} \tag{3.28}$$

which applies the ℓ_∞ norm to the rows and ℓ_1 to the resulting vector, or using the $\ell_{2,1}$ norm as Malioutov did in [57]:

$$\min_{\boldsymbol{X} \in \mathbb{R}^{N \times Q}} \| \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X} \|^2 + \lambda \| \boldsymbol{X} \|_{2,1} \tag{3.29}$$

which applies the Euclidean norm to the rows and ℓ_1 to the resulting vector.

λ in 3.28 and 3.29 is the Lagrange multiplier and a function of $\boldsymbol{D}$, $\boldsymbol{Y}$ and $\boldsymbol{E}$. It is basically a fixed penalty term that we specific beforehand which determines the level of sparsity in $\boldsymbol{X}$. The higher λ, the more impact the error term has in the minimization problem. If $\lambda = 1$ in 3.28 and 3.29, the equations reduce to 3.26 with only the quasinorm replaced by a mixed norm term. Such formulations as 3.28 and 3.29 are often called the multivariate LASSO solutions [41]. The LASSO (least absolute shrinkage and selection operator) was in fact invented by researchers in statistics for model selection purposes at the same time basis pursuit came to light for signal processing purposes [46, chapter 5.3.3]. Even though the LASSO in statistics and BPDN in sparse approximations target different applications, they have the same mathematical formulation and end goal, for example compare 3.26 to 3.28 and 3.29. The latter make use of the $\ell_{2,1}$ norm which is considered more robust to outliers or dependent heavy-tailed noise since the error of each data point (the column) is not squared. Adding to that, a known problem with least squares minimization in 3.28 and 3.29 is the strong weight that the Euclidean ℓ_2 norm places on large residuals and the very small weight it places on small residuals. This means that large outliers can have a large impact on the result.

The second notable category is greedy pursuit algorithms. They work by iteratively comparing different correlations between a selected subset of atoms (columns of $\boldsymbol{D}$) with the current residual matrix and then updating the row support with the atom (column) that can reduce the residual the most as in 3.26. These algorithms have a built-in greediness that keeps adding atoms as long as the residual ℓ_2 error $\boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X}$ decreases and $rsupp(\boldsymbol{X}) \leq K$. They typically require a real valued K as a fixed hyperparameter, thus the optimal number of atoms or sparsity level may require iterative tries to find, but is often easier to compute than the optimal irrational

penalty parameter λ for the LASSO variants in 3.28 and 3.28 [41]. Greedy pursuit algorithms are known for being efficient, fairly simplistic and provide good approximations of sparse signals as proved by [26]. Numerical experiments have also shown that greedy algorithms can perform better than BPDN in some cases [27], however they fall short when two atom vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_j$, that correspond to active signals $\boldsymbol{y}_i$ and $\boldsymbol{y}_j$, are both close to a third atom vector $\boldsymbol{d}_k$. If $\boldsymbol{y}_i \approx \boldsymbol{y}_j$, the cross-talk onto the $\boldsymbol{d}_k$ atom is larger than the hidden signals onto $\boldsymbol{d}_i$ or $\boldsymbol{d}_j$, therefore a greedy algorithm updates $\boldsymbol{x}_k$ rather than $\boldsymbol{x}_i$ or $\boldsymbol{x}_j$. Once such a mistake is made, a greedy algorithm never recovers [58, chapter 7.3].

The design of sparse recovery methods should be guided by a list of requirements that tell us if a certain method is viable or not. We compiled a short list here:

- **Minimal number of measurements:** The sparse recovery algorithm should only require the same measurements M to recover a K-sparse signal. Any excessive needs for measurements may prolong sampling times, increase time complexity and take up more space.

- **Performance:** Any algorithm should meet some guarantee in terms of recovery accuracy and stability. In section 3.2 we mentioned prerequisites like incoherence, the RIP or the NSP on $\boldsymbol{\Phi}$ or $\boldsymbol{D}$, for exact or approximate recovery of any signal $\boldsymbol{x}$. Obviously the choice of $\boldsymbol{\Phi}$ or $\boldsymbol{D}$ plays an important role as well. Possible error metrics include mean squared error (MSE), Hamming distance or probability of exact recovery (PER). On a final note, added noise or perturbations of the input signal should not throw the algorithm off balance.

- **Sparsity:** We want to recover a signal from either clear or noisy measurements using as few elementary signals or atoms as possible to keep the dimensionality low and the signal sparse. Recall that some applications rely heavily on the low sparsity in a signal like the image compression format JPEG or nodes transmitting data over a limited network bandwidth.

- **Speed:** Our algorithm should converge towards an approximate solution within some expected time frame. This of course needs to count in complexity, signal size and expected error. Also one should adopt a common stopping criteria when comparing algorithms like thresholding the error or execution time.

- **Scalability:** Since we base our work on the MMV model, that is we make jointly sparse approximations from multiple signals, it becomes even more interesting to evaluate aspects of scalability in an algorithm and how a task can be shared amongst distributed computers. Cloud computing and online distributed services are the talk of the town, so naturally any approximation algorithm should consider this too.

To summarize, subset selection problems that select a subset of size k atoms from a total of n possible to optimize some criterion are generally NP-hard [59]. Basis pursuit algorithms, in short BPDN like the LASSO from statistics, the iteratively reweighted least squares (IRLS) [46, chapter 5.3.2] and the least-angle regression (LARS) [46, chapter 5.3.3] finds an $\boldsymbol{X}$ that solves a undetermined system of equations $\boldsymbol{Y} = \boldsymbol{DX}$ via an exhaustive search for an ideal combination of atoms that minimize the error. These are also called relaxation methods because they usually replace the set size constraint (i.e., ℓ_0 norm) with a convex relaxed constraint ℓ_1 [46, chapter 3.2.1]. A greedy strategy, most often implemented by the well-studied and efficient orthogonal matching pursuit (OMP) algorithm and the likes, abandon this kind of search and instead finds a series of local optimal updates that would reduce the error the most [46, chapter 3.1.2] [60]. Generally, the OMP isolates parts of the signal that are coherent with respect to a given dictionary and use these for reconstruction. An obvious weakness of the greedy approach is that the process of selecting an atom is sub-optimal: Our residual at a current iteration tells whether to select

an atom or not and once it has been selected, it cannot be removed. We can end up with a set of sub-optimal atoms selected at an early stage that are not correlated and the algorithm will therefore converge to a poor result. To avoid ending up in this sub-optimal blind alley our dictionary needs to represent the structures in the signal we want to approximate. Next section will review the theory behind creating a robust dictionary that can model any data and also compare this approach to the traditional analytic learning method using predesigned transforms. The OMP algorithm can be seen in appendix F.

3.6 Dictionary learning in sparse approximation

Many use cases in computer vision, image and acoustic processing are preceded by identification of geometric features in image and audio signals that are important to comprehend and process it. As the amount of data available in today's age grows excessively, in what is coined the big data era, and due to memory and storage constraints of most processing systems, we need ways to process and understand images and audio signals of high-dimensionality without an immerse overhead and poor throughput. This is the working methodology in sparse approximation as we have seen and also what we seek in a subfield of sparse approximation called dictionary learning. Recall that in order to recover an approximation of signal $\boldsymbol{X}$ from $\boldsymbol{Y}$ sampled measurements, we solve the linear inverse problem $\boldsymbol{Y} = \boldsymbol{DX}$ for $\boldsymbol{X}$. In the general case using what is called a dictionary that basically guides the recovery process in using the best non-zero transform coefficients because it contains a database of the geometric features, in the data of interest. Gauging recovery success can then be done by examining the level of sparsity in the recovered representation, or use a reference signal to estimate the error. The area of dictionary learning deals with designing such a guidebook.

Dictionary learning is a nonlinear data-centric training framework, emerged over recent years, that records geometric structures in data signals (acoustic, audio, natural images etc.) [11]. The assumption is that samples, in for example image patches, can be sparse approximated by a linear combinations of a few columns in a suitable overcomplete basis (a dictionary) [61]. Let $\boldsymbol{Y} \in \mathbb{R}^{M \times Q}$ be a matrix of vectorized image patches, then each patch $\boldsymbol{y}_i \in \mathbb{R}^M$ has a sparse representation in a dictionary $\boldsymbol{D} \in \mathbb{R}^{M \times N}$ if $\boldsymbol{Y} \approx \boldsymbol{DX}$ and the number of non-zero row entries in $\boldsymbol{X} \in \mathbb{R}^{N \times Q}$ is smaller than the number of column atoms in $\boldsymbol{D} \in \mathbb{R}^{M \times N}$ [61]. The dictionary $\boldsymbol{D}$ is then a dimensionality reduction or *sparsifying* step that can either be created by a predefined model, such as wavelets [16], curvelets [62] or it can be learned from data [11] [13] [38].

Using a predefined dictionary such as the Fourier, Haar or DCT bases are sufficient in most recovery cases and they are often simpler to employ than making a new from scratch. However to improve the speed and precision in a recovery problem, a dictionary that has been trained on a sample set of some data, that shares the structure and characteristics of that it needs to recover, is often better than traditional predefined dictionaries. A learned basis is shown to have better coding efficiency and also have better presentation power than traditional bases in many applications [56]. Since predefined dictionaries do not adapt to the underlying structure in the images, they do not always result in best image representations [11]. This does not mean that a learned dictionary is always better than a predefined one, in fact a learned dictionary is often limited in the sense that it will only work for a specific problem, where the traditional ones work in a more general matter. In particular with big data, in a setting of ever-accumulating data sets, it may not be feasible to use all available images to build representations or learn features. This indicates that if a dictionary can be quickly and efficiently trained at the start of recovery, or adaptively as the reconstruction process proceeds, it could potentially outshine the predefined one, given data structures do not change drastically, and our systems can process the dimensionality in the data we present them. Another case when dictionary learning can be applied is when it is unknown or lost and therefore needs to be re-estimated in order to re-create the original data

through sparse approximation.

To provide a mathematical overview of the core question in dictionary learning, the optimization problem can be written as seen in 3.30 [46, chapter 12.2.1]. Suppose we have been given a set of Q vectorized images $\boldsymbol{Y} = [\, \boldsymbol{y}_1 \; \boldsymbol{y}_2 \; ... \; \boldsymbol{y}_Q \,] \in \mathbb{R}^{M \times Q}$, where M is the dimension of each image. The problem of learning a dictionary of N atoms that can be used for sparse approximation given the measurements $\boldsymbol{Y}$ can be expressed in a form for each training signal $\boldsymbol{y}_i$ and coefficient vector $\boldsymbol{x}_i$:

$$\min_{\boldsymbol{D}, \{\boldsymbol{x}_i\}_{i=1}^{Q}} \sum_{i=1}^{Q} \| \, \boldsymbol{x}_i \, \|_0 \;\; \text{subject to} \;\; \| \, \boldsymbol{y}_i - \boldsymbol{D}\boldsymbol{x}_i \, \|_2 \leq \epsilon, 1 \leq i \leq Q \tag{3.30}$$

$\boldsymbol{x}_i$ represents a coefficient vector for the individual signal, $\boldsymbol{y}_i$ represents a training signal, $\boldsymbol{D} \in \mathbb{R}^{M \times N}$ is the overcomplete dictionary (i.e. $N > M$) with ℓ_2-norm columns and ϵ represents the model deviation or error. The intention is to estimate $\boldsymbol{D}$, assuming that the deviation ϵ is known. If a solution is found with exactly the cardinality K or fewer entries, it is a feasible model. With this knowledge, 3.30 can be reformulated by reversing the penalty and constraint for a case, where the sparsity needs to be constrained. The formula can be seen in 3.31 [46, chapter 12.2.1].

$$\min_{\boldsymbol{D}, \{\boldsymbol{x}_i\}_{i=1}^{Q}} \sum_{i=1}^{Q} \| \, \boldsymbol{y}_i - \boldsymbol{D}\boldsymbol{x}_i \, \|_2^2 \;\; \text{subject to} \;\; \| \, \boldsymbol{x}_i \, \|_0 \leq K, 1 \leq i \leq Q \tag{3.31}$$

Both equations 3.30 and 3.31 can be rewritten to a simplified form by concatenating all vectors column-wise, resulting in $\boldsymbol{Y}$ forming an $M \times Q$ matrix and $\boldsymbol{X}$ forming an $N \times Q$ matrix. The resulting equations can be seen in 3.32 and 3.33 in what is the general form of dictionary learning:

$$\min_{\boldsymbol{D}, \boldsymbol{X}} \| \, \boldsymbol{X} \, \|_0 \;\; \text{subject to} \;\; \| \, \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X} \, \|_2 \leq \epsilon \tag{3.32}$$

$$\min_{\boldsymbol{D}, \boldsymbol{X}} \| \, \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X} \, \|_2^2 \;\; \text{subject to} \;\; \| \, \boldsymbol{X} \, \|_0 \leq K \tag{3.33}$$

With the core questions defined in equations 3.30-3.33, and prior work showing that exact determination of sparsest representations prove to be an NP-hard problem [63], we can now look at two established dictionary learning algorithms; MOD and K-SVD. These two algorithms does the very same thing, learn a dictionary from training data, but with very different approaches.

The method of optimal directions (MOD) is a dictionary learning algorithm that aims to estimate the dictionary $\boldsymbol{D}$ only through the knowledge of the signals $\boldsymbol{Y}$ and the original cardinality K. The algorithm start by making a dictionary $\boldsymbol{D}_{(0)}$, which is constructed of random entries of $\{\boldsymbol{y}_i\}_{i=1}^{Q}$. *Step 1: Sparse approximation*; using the knowledge of equation 3.31, the sparse signal representation $\hat{\boldsymbol{x}}_i$ is approximated, $1 \leq i \leq Q$, through the use of a pursuit algorithm, by using the randomly generated dictionary $\boldsymbol{D}_{(0)}$ as a fixed dictionary. *Step 2: Dictionary Update*; update the dictionary, $\boldsymbol{D}_{(k)}$, with the new $\boldsymbol{X}_{(k)}$ formed by the newly acquired $\hat{\boldsymbol{x}}_i$, $1 \leq i \leq Q$. This is done using equation 3.34 [46, chapter 12.2.2].

$$\begin{aligned} \boldsymbol{D}_{(k)} &= arg \min_{\boldsymbol{D}} \| \, \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X}_{(k)} \, \|_F^2 \\ &= \boldsymbol{Y}\boldsymbol{X}_{(k)}^T (\boldsymbol{X}_{(k)}\boldsymbol{X}_{(k)}^T)^{-1} \\ &= \boldsymbol{Y}\boldsymbol{X}_{(k)}^{\dagger} \end{aligned} \tag{3.34}$$

Step 1 and 2 will then be reiterated, until the stopping rule is satisfied. The stopping rule is when the changes in the error $\| \, \boldsymbol{Y} - \boldsymbol{D}_{(k)}\boldsymbol{X}_{(k)} \, \|_F^2$ is small enough. The MOD algorithm cannot promise to reach a global minimum, but tests show that the algorithm is very effective at

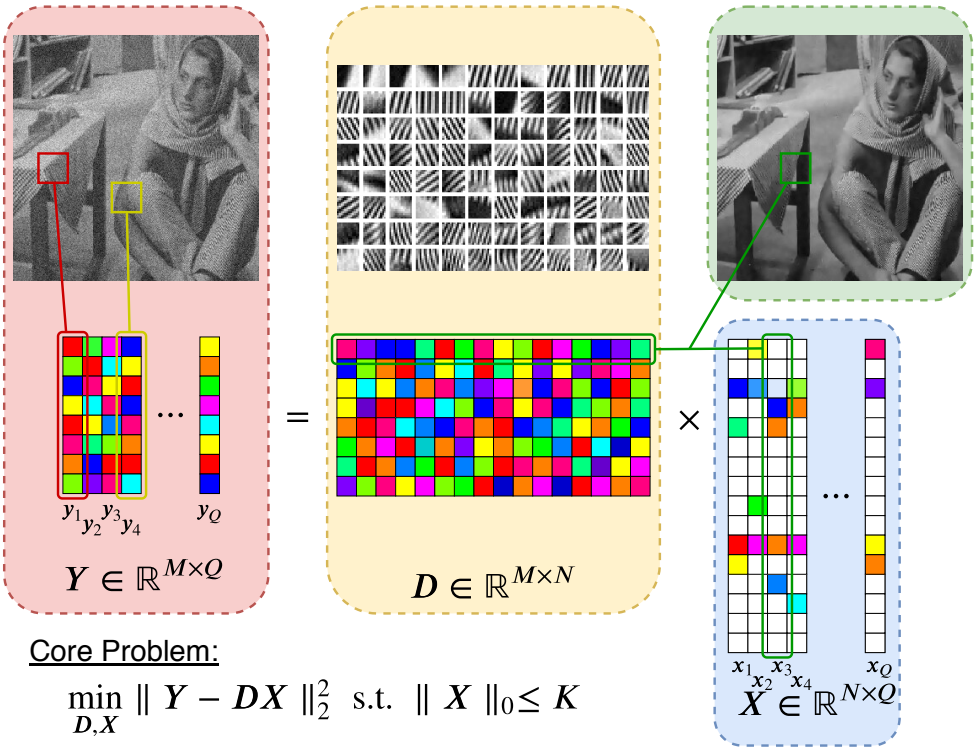

Figure 3.4: *A practical example of dictionary learning for denoising. The noisy picture represented in the red block is divided into individual patches illustrated as vectors; $\boldsymbol{y}_i$. These patches form the matrix $\boldsymbol{Y} = [\, \boldsymbol{y}_1 \; \boldsymbol{y}_2 \; \cdots \; \boldsymbol{y}_Q \,]$. With a dictionary learning algorithm, such as MOD or K-SVD, minimizing the core problem will yield a dictionary $\boldsymbol{D}$, the yellow block, as well as a sparse representation of the patches $\boldsymbol{X} = [\, \boldsymbol{x}_1 \; \boldsymbol{x}_2 \; \cdots \; \boldsymbol{x}_Q \,]$, the blue block. The dictionary is constructed of the most significant structures of the patches and the least significant ones are thrown away. If the noise is indeed randomly generated, this means there is no structure to it, and that will be the first to be discarded. Using data the matrices $\boldsymbol{D}$ and $\boldsymbol{X}$ the image can be reconstructed without the noise.*

estimating the original dictionary [11] by an approximation. The complete MOD algorithm can be seen in appendix F.

K-SVD is an algorithm that builds on knowledge from the clustering method *K-means* and is basically made as an extension of said algorithm. It is a versatile algorithm that works using any sparse approximation algorithm, it has the same sparsification problem as MOD and uses a similar two-step approximate-and-learn approach. That is, K-SVD alternates iteratively between sparse approximation of the input signal using the current dictionary and an update process of the dictionary atoms using the singular value decomposition (SVD) method to fit the data from the sparse approximation step. Because of the high non-convexity of the problem in 3.33, it is impossible to promise that the K-SVD will reach a global minimum and can get caught in local minima or even saddlepoints, however tests show that the algorithm is very effective, even more so than the MOD at approximating the original dictionary [11].

The main contribution of K-SVD is that the dictionary update step is performed sequentially atom-by-atom in a simple fashion instead of using matrix inversion as in the MOD. Moreover it adds further acceleration to the process by updating both the current atom and its associated sparse coefficients simultaneously [64]. This provides a fast and efficient algorithm for dictionary learning that requires less resources than the MOD. In practice, the K-SVD is an effective method for representing small signal patches because the result of the training process is a non-structured

dictionary which is relatively costly to apply to signals of large size [64]. For K-SVD, the dictionary update method as seen in 3.34 is rewritten in 3.35 [46, chapter 12.2.3].

$$\begin{aligned} \| \boldsymbol{Y} - \boldsymbol{DX} \|_F^2 &= \left\| \boldsymbol{Y} - \sum_{j=1}^{N} \boldsymbol{d}_j \boldsymbol{x}_j^T \right\|_F^2 \\ &= \left\| \left(\boldsymbol{Y} - \sum_{j \neq j_0} \boldsymbol{d}_j \boldsymbol{x}_j^T \right) - \boldsymbol{d}_{j_0} \boldsymbol{x}_{j_0}^T \right\|_F^2 \\ &= \| \boldsymbol{E}_{j_0} - \boldsymbol{d}_{j_0} \boldsymbol{x}_{j_0}^T \|_F^2 \end{aligned} \tag{3.35}$$

where $\boldsymbol{x}_j^T$ is the j'th row of $\boldsymbol{X}$, and $\boldsymbol{E}_{j_0}$ is the error, which can be obtained using SVD. Using the standard SVD in this case yields some problems in the form of a dense vector $\boldsymbol{x}_{j_0}^T$, which is the result of increasing the number of non-zeros in $\boldsymbol{X}$. To fix this problem, instead of taking the complete set of the error $\boldsymbol{E}_{j_0}$, only a subset will be taken, to keep the wanted cardinality of the signals. To do this, a restriction operator $\boldsymbol{P}_{j_0}$ is introduced. This operator will be multiplied on the right side of $\boldsymbol{E}_{j_0}$ in order to remove the columns, corresponding to where the entries in the row $\boldsymbol{x}_{j_0}^T$ are zero. This is done in order to keep the columns that affects the non-zero values, and remove the ones that has no effect, thereby keeping the cardinality without losing data. This results in the restriction definition of $\boldsymbol{x}_{j_0}^T$ as $(\boldsymbol{x}_{j_0}^R)^T = \boldsymbol{x}_{j_0}^T \boldsymbol{P}_{j_0}$. With the new sub-matrix $\boldsymbol{E}_{j_0} \boldsymbol{P}_{j_0}$ an approximation using SVD can be found, updating both the dictionary $\boldsymbol{D}$ and the sparse representations of the signals $\boldsymbol{X}$. The complete K-SVD algorithm is detailed in appendix F.

Another approach is to not use the full SVD and instead use a numerical approach to finding $\boldsymbol{d}_{j_0}$ and $\boldsymbol{x}_{j_0}^T$ proposed on equation 3.36 and 3.37 [46, chapter 12.2.3].

$$\min_{\boldsymbol{x}_{j_0}^R} \| \boldsymbol{E}_{j_0} \boldsymbol{P}_{j_0} - \boldsymbol{d}_{j_0} (\boldsymbol{x}_{j_0}^R)^T \|_F^2 \quad \Rightarrow \quad \boldsymbol{x}_{j_0}^R = \frac{\boldsymbol{P}_{j_0}^T \boldsymbol{E}_{j_0}^T \boldsymbol{d}_{j_0}}{\| \boldsymbol{d}_{j_0} \|_2^2} \tag{3.36}$$

$$\min_{\boldsymbol{d}_{j_0}} \| \boldsymbol{E}_{j_0} \boldsymbol{P}_{j_0} - \boldsymbol{d}_{j_0} (\boldsymbol{x}_{j_0}^R)^T \|_F^2 \quad \Rightarrow \quad \boldsymbol{x}_{j_0}^R = \frac{\boldsymbol{E}_{j_0} \boldsymbol{P}_{j_0} \boldsymbol{x}_{j_0}^R}{\| \boldsymbol{x}_{j_0}^R \|_2^2} \tag{3.37}$$

More recently a novel paradigm has been purposed called *parametric* dictionary training that aim to address these issues by combining the strengths of both predetermined and trained dictionaries. Figure 3.5 is a visual comparison of the two methods. Studies have been done on synthetic noisy speech data and show that parametric dictionaries can yield a better representation in terms of mean squared error (MSE) than non-parametric learning methods like the K-SVD that result in unstructured dictionaries [65] [66].

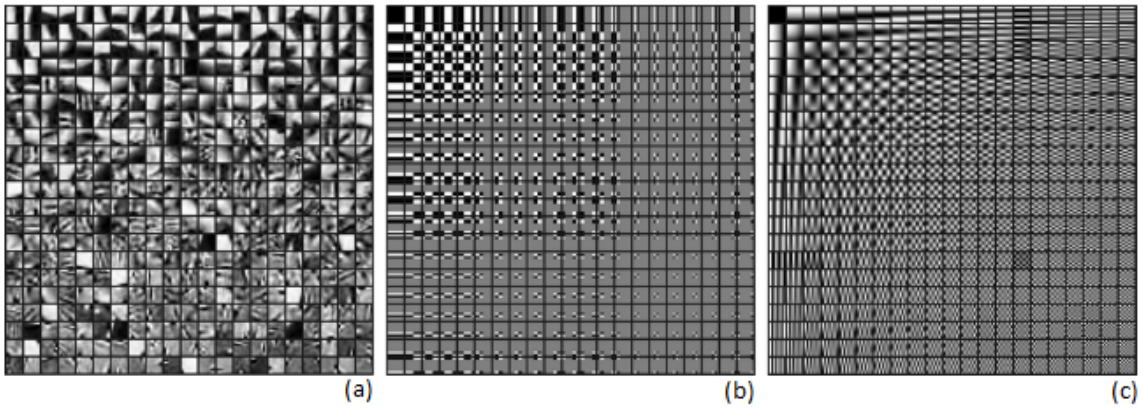

Figure 3.5: *(a) is a dictionary learned from image patches. (2) and (3) is the overcomplete Haar and DCT dictionary respectively for comparison. Source: [11].*

3.7 Consensus and power iterations

Cloud K-SVD is a distributed version of the K-SVD algorithm that trains a dictionary on multiple distributed nodes instead of just one. We will get to the mechanics of it in the next section, but first explain some fundamental theory about it. For cloud K-SVD to work, it needs to reach a consensus among nodes, which we will start examining from a general point of view.

A major difference between centralized and distributed systems is that we can experience partial divergence of data in a distributed one, since the systems do not share a global memory space or clock. Each node has its own private data space and executes requests as a sequence of sequential or parallel processes. If a group of nodes with the same specifications and the same preconditions executed the same request, say a web site query, the final result may vary depending on the resources available to the individual node at runtime, network throughput and other non-deterministic factors. Computers behave differently and run differently depending on their environment and configuration.

There exist a fundamental problem in distributed computing called the consensus problem: It requires multiple processes or nodes in a system to come to an consensus on a single value for computation. This value or state could be the result of a transaction, the current time of day or leader-election. Consensus protocols achieve reliability on networks involving multiple nodes, making sure all nodes conform to the current state of the system. If bilateral computations are needed, processes must talk to each-other, share their current state via message passing and somehow agree on one value in the original problem. This operation can spell trouble, as communication links between nodes may fail or become unreliable, so a consensus protocol must take this into account. One acclaimed example is the Raft algorithm by Diego Ongaro and John Ousterhout published in 2014 [67] that works by electing a leader node that has to manage and replicate a shared log across multiple nodes. The idea with such a *joint consensus* scheme is that at any point in time there is only one leader, and if the leader fails, a new one is elected with a high degree of certainty. This protocols dictate a certain correct state of the system, like the distributed digital ledger *blockchain*, where transactions are recorded between people across many nodes so that the record cannot be altered retroactively without altering all subsequent blocks and the consensus of the network. We can achieve this by having processes agree on one majority value, which requires more than half the nodes in a system to reach consensus. A less strict implementation which does not dictate a final true state but rather an average is the consensus averaging protocol. This protocol is used in cloud K-SVD to reach consensus about how the global dictionary should look among nodes. In the next phrases we will explain averaging

consensus and briefly touch upon its rigorous older brother, corrective consensus.

Averaging consensus

Averaging consensus [68] works by having nodes collaboratively share their state, for example the current residual vector in a dictionary learning scenario, then by averaging steer the pool of nodes towards a mutual averaged consensus. We consider a multi-hop network of a group of H nodes modeled as a undirected graph $\mathcal{G} = (\mathcal{V}, \varepsilon)$ where $\mathcal{N} = 1, 2, \ldots, H$ represents the H nodes and the edge set $\epsilon = (i, j) : i, j \in \mathcal{V}, p_{i,j} > 0$ consists of vertex pairs. A pair (i, j) represents the directed edge from j to i, which means node j can send traffic to node i. In theory the link is stable and has no packet loss, however in practice things such as link congestion, node failures and delays can occur. The averaging protocol assumes that each node i holds some value $\boldsymbol{z}_i \in \mathbb{R}$ and that it must compute the average $\bar{\boldsymbol{z}} = \sum_{i=1}^{H} \boldsymbol{z}_i / H$ using linear distributed iterations [69]. Each node defines a local state variable $\boldsymbol{x}_i(t = 0) = \boldsymbol{z}_i$ and iteratively updates the value with a weighed average of its neighbors state variables [70]. $\boldsymbol{x}_i(t)$ converges to $\bar{\boldsymbol{z}}$ under certain conditions [70]. How the actual communication takes place is conceived at the implementation stage, as node i can either send its state update $\boldsymbol{x}_i(t)$ synchronously to j and thus wait for a reply, have a second thread that continuously sends out updates asynchronously or instead have j pull $\boldsymbol{x}_i(t)$ from i whenever it needs it. This section concludes with implementation samples of two asynchronous consensus algorithms. After receiving their neighbor's state, nodes update their state variables as:

$$\boldsymbol{x}_i(t+1) = \boldsymbol{x}_i(t) + \sum_{j=1, j \neq i}^{H} \boldsymbol{W}_{ij}(t) \Big(\boldsymbol{x}_j(t) - \boldsymbol{x}_i(t) \Big) \tag{3.38}$$

where $\boldsymbol{W}(t) \in \mathbb{R}^{H \times H}$ is a weight matrix defined as $\boldsymbol{W}(t) := \mathcal{I} - \epsilon \boldsymbol{L}(t)$, where $\boldsymbol{L} = [l_{ij}]$ is the graph Laplacian matrix of the network [68].

Let $E(\boldsymbol{W})$ denote the expectation of the weight matrix $\boldsymbol{W}$. It can be fixed for the entirety of the algorithm or computed randomly at each iteration to simulate packet loss. If we select the constant ϵ to satisfy the constraint $0 < \epsilon < 1/max_{i,t}(\boldsymbol{d}_i(t))$, so the second largest eigenvalue of $E(\boldsymbol{W})$ is smaller than 1 in magnitude, 3.38 will converge to a common value, i.e. $\lim_{t \to \infty} \boldsymbol{x}(t) = \alpha \mathbb{1}$, where $\mathbb{1}$ is an eigenvector with all entries equal to one, i.e. $\mathbb{1} = (1, ..., 1)^T$ [71]. When the weight matrix is balanced, i.e. $\mathbb{1}^T \boldsymbol{W}(t) = \mathbb{1}^T$ then the nodes converge to the correct average $\alpha = \bar{\boldsymbol{z}}$.

Corrective consensus

Corrective consensus is an extension to averaging consensus and introduces a set of auxiliary variables $\boldsymbol{\phi}_{ij}^{(t)}$ at every node i used to store the residual difference. It updates them at time $t + 1$ as follows [70]:

$$\boldsymbol{\phi}_{ij}(t+1) = \boldsymbol{\phi}_{ij}(t) + \boldsymbol{W}_{ij}(t)(\boldsymbol{x}_j(t) - \boldsymbol{x}_i(t)),\ \boldsymbol{\phi}_{ij}(0) = 0 \tag{3.39}$$

In other words, the accumulated residual difference between node i and j for iteration $t + 1$ is the stored residual variable $\boldsymbol{\phi}_{ij}$ for iteration t plus the weight for the node link times the residual between signal vectors $\boldsymbol{x}_j(t)$ and $\boldsymbol{x}_i(t)$, respectively. $\boldsymbol{\phi}_{ij}$ represents the amount of change node i has made to its signal vector $\boldsymbol{x}_i(t)$ by exchanging data with node j. If we consider only symmetric packet losses for the link between i and j, it follows from 3.39 that $\boldsymbol{\phi}_{ij} + \boldsymbol{\phi}_{ji} = 0$ if $\boldsymbol{W}_{ij}(t) = \boldsymbol{W}_{ji}(t)$ holds for all t [70]. Accounting for packet losses, a situation where $\boldsymbol{x}_i(t)$ is lost at the j end is not impossible, so we have $\boldsymbol{W}_{ij}(t) \neq \boldsymbol{W}_{ji}(t)$ and consequently $\boldsymbol{\phi}_{ij} + \boldsymbol{\phi}_{ji} \neq 0$, thus we would see a shift away from the global average $\hat{\boldsymbol{z}}$ since node j would never take into account the residual in $\boldsymbol{x}_i$. To update state variables $\boldsymbol{x}_i$ and $\boldsymbol{\phi}_{ij}$, we perform M averaging iterations and at every iteration k perform a corrective one m times [70]:

$$\boldsymbol{x}_i(k+1) = \boldsymbol{x}_i(k) - \sum_{j=1}^{H} \boldsymbol{\Delta}_{(i,j)}(k)/2 \boldsymbol{\phi}_{ij}(k+1) = \boldsymbol{\phi}_{ij}(k) - \boldsymbol{\Delta}_{(i,j)}(k)/2 \tag{3.40}$$

where $\boldsymbol{\Delta}_{(i,j)}(t) = \boldsymbol{\phi}_{ij} + \boldsymbol{\phi}_{ji}$, that is the accumulated approximation residual between node i and j. If we assume an asynchronous pull implementation, node i would periodically start up a separate thread to fetch the $\boldsymbol{\phi}_{ji}$ from its neighbors, calculate the difference $\boldsymbol{\Delta}_{(i,j)}(t)$ and adjust its state variable $x_i(t)$ accordingly. If no residual can be retrieved from node j, the algorithm will simply skip the corrective step for that node and retry after n averaging consensus steps. Furthermore corrective consensus has been shown by proof to converge to $\bar{\boldsymbol{z}}$ given enough n and m iterations [72].

Power iterations

We want to calculate the most dominant eigenvector in the residual error via power iterations and afterwards use consensus passes on all neighboring nodes by multiplying the eigenvector with some weight to estimate a new approximation, as in 3.38. Assume $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ is symmetric, vector $\boldsymbol{x}$ is an eigenvector of $\boldsymbol{A}$ with eigenvalue λ if $\boldsymbol{Ax} = \lambda \boldsymbol{x}$. Vector $\boldsymbol{x}$ is a dominant eigenvector if there are no other eigenvectors with an eigenvalue larger than $\| \lambda \|$ in absolute value. In this case, λ is a dominant eigenvalue and $\| \lambda \|$ is the spectral radius of $\boldsymbol{A}$. More specifically, the power method applied in each iteration is defined as: Given a unit 2-norm $\boldsymbol{q}^{(0)} \in \mathbb{R}^N$, the *power method* produces a sequence of vectors $\boldsymbol{q}^{(k)}$ as follows [73, chapter 8]:

$$\begin{aligned} &\text{for } k = 1, 2, \ldots \\ &\quad \boldsymbol{z}^{(k)} = \boldsymbol{A}\boldsymbol{q}^{(k-1)} \\ &\quad \boldsymbol{q}^{(k)} = \boldsymbol{z}^{(k)} / \| \boldsymbol{z}^{(k)} \|_2 \\ &\quad \lambda^{(k)} = [\boldsymbol{q}^{(k)}]^T \boldsymbol{A}\boldsymbol{q}^{(k)} \\ &\text{end} \end{aligned} \tag{3.41}$$

If $\boldsymbol{q}^{(0)}$ is not deficient and $\boldsymbol{A}$'s eigenvalue of maximum modulus is unique, then the $\boldsymbol{q}^{(k)}$ converge to an eigenvector [73, chapter 8]. A concrete example is the PageRank algorithm [74], named after one of the founders of Google, Larry Page. It calculates the importance ranking for web pages. This is based on how many other sites in a connected graph that link to a specific page. PageRank was the first algorithm used by Google to rank web pages in their search engine, but not the only algorithm as of today. The iterative method to calculate the PageRank score is a power iteration that takes an adjacency matrix where $\boldsymbol{M}_{i,j}$ represents the link from j to i, a fixed number of iterations and a damping factor and returns a vector of page ranks such that $\boldsymbol{v}_i$ is the i'th rank from $[0, 1]$, where $\boldsymbol{v}$ is rescaled so that each column adds up to one for all pages. The page ranks are the dominant right eigenvector of the adjacency matrix $\boldsymbol{M}$ [74].

Jelasity et al. [75] extend the thought behind PageRank as they purpose the use of asynchronous distributed power iterations in a chaotic dynamic network, where nodes or network links are added or removed at will. Here, nodes find the dominant eigenvector of large and sparse matrices, like an adjacency matrix in PageRank, collaboratively by message passing. The adjacency or neighborhood matrix in this case is fully distributed, meaning the elements of the matrix are held by individual network nodes, one vector element per one node. Jelasity et al. considers cases where the dominant eigenvalue differs from one. Figure 3.6 shows asynchronous calculation of the dominant eigenvector of a weighted neighborhood matrix A and 3.7 shows gossip-based average approach, where a node connects to basically a random peer and then average the response with what they currently have. Gossiping is a way in peer-to-peer connected distributed systems to ensure that data is disseminated to all members of a group. They are further studied in [76, 77].

```
1: loop {Active Thread}
2:    wait(Δ)
3:    for each j ∈ out-neighbors_i do
4:       send A_ji w_i to j
5:    b_i ← Σ_{k∈in-neighbors_i} b_ki
6:    w_i ← b_i
```

```
1: loop {Passive Thread}
2:    x ← receive(∗)
3:    k ← sender(x)
4:    b_ki ← x
```

Figure 3.6: *An asynchronous power iteration algorithm that calculates the dominant eigenvector of a weighted neighborhood matrix A at node i [75].*

```
1: loop {Active Thread}
2:    wait(Δ_r)
3:    j ← getRandomPeer()
4:    send r_i to j
5:    r_j ← receive(j)
6:    r_i ← (r_i + r_j)/2
```

```
1: loop {Passive Thread}
2:    r_j ← receive(∗)
3:    send r_i to sender(r_j)
4:    r_i ← (r_i + r_j)/2
```

Figure 3.7: *An asynchronous gossip-based consensus algorithm. It uses a stochastic model to forward its vector element to a random node j and averages the return value with the current state [75].*

3.8 The cloud K-SVD and distributed learning

We have already discussed how dictionary learning can be done centrally however with the constraints that big data introduce, where no single system can cope with entire data sets, and security concerns of many modern data applications like privacy, confidentiality and copyright, it is evident that dictionary learning should be performed in a distributed manner by a pool of nodes that offer redundancy, horizontal scalability and local access to data on a individual node basis, so that no one else will get to access or modify the information. In section 3.7 we accounted for the various ways that nodes could reach consensus in a distributed system and finally explained in brief how averaging node consensus works between multiple nodes asynchronously. We set out in section 1.3 to experiment and evaluate a distributed sparse approximation and dictionary learning algorithm like cloud K-SVD in a real setup with real data input, however before realizing the goal, it is important to establish a sound understanding of what cloud K-SVD adds to the recently discussed K-SVD in section 3.6 and the algorithms that compromise it. Cloud K-SVD is like regular K-SVD as we described in section 3.6 with basically an added consensus step, where nodes exchange their current residual coding error to archive a collaborative global dictionary that best represent the structures in the input data. One way of achieving consensus is by averaging as described i section 3.7. Cloud K-SVD defines three kinds of iterations, which are important to keep in mind for the rest of the section and for the experiments later on: The number of overall iterations t_d, the number of power iterations t_p and the number of consensus iterations t_c. For example by setting $t_d = 1 \wedge t_p = 3 \wedge t_c = 5$ we run the algorithm one time in its entirety with exactly three power iterations that each include five consensus iterations. In theory, the algorithm is performed on a undirected graph $\mathcal{G} = (\mathcal{V}, \varepsilon)$ where $\mathcal{V} = 1, 2, ..., H$ represents the set of all H nodes and ε represents edges. Each node has access to its own data, $(i, i) \in \varepsilon$ and connections are represented by $(i, j) \in \varepsilon$. The following phrases will describe some technical details of the algorithm.

We assume a distributed setting, where local data is held at different nodes. Each node is denoted by H_i with local data $\boldsymbol{Y}_i \in \mathbb{R}^{M \times Q_i}$ for a total of H nodes, so the total data amount is $Q = \sum_{i=1}^{H} Q_i$ and can be represented as one matrix $\boldsymbol{Y} \in \mathbb{R}^{M \times Q}$ [9]. The goal of cloud K-SVD is for each node H to collaboratively learn the dictionary $\boldsymbol{D}$ by solving the minimization problem that finds the signal matrix using local data, then find the dominant eigenvector denoted $\boldsymbol{q}$ using a power method of the square, positive-semidefinite residual matrix $\boldsymbol{M}$ obtained by $\boldsymbol{Y} - \boldsymbol{DX}$

calculations locally and then at each consensus iteration t_c they must reach an estimate of the value of $\boldsymbol{q}$ for $\boldsymbol{M}$ where $\boldsymbol{M} = \sum_{i=1}^{H} \boldsymbol{M}_i$ and H is the total amount of nodes. Since the number of global samples is larger than the number of samples at each node, collaborative learning should outperform local learning, where all data has to be processed by every node. Figure 3.8 shows how global data $\boldsymbol{Y}$ is distributed at local nodes H.

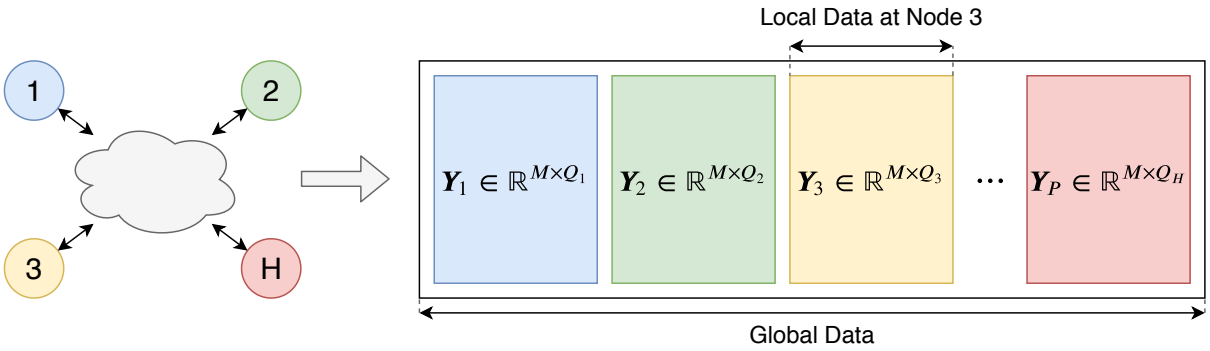

Figure 3.8: *In cloud K-SVD, we train a distributed data model based on spatially separated local node data. Source: [9].*

We now go into more detail with the algorithm in algorithm 1. It is essential for our thesis and therefore included in full here. First step is to compute the local signal matrix $\hat{\boldsymbol{X}}_i^{(t_d)}$ using any sparse approximation algorithm for example OMP at iteration t_d using the previous dictionary data from $t_d - 1$ without collaboration similar to K-SVD. A greedy algorithm like OMP decomposes a signal into linear combinations, or an approximation of the columns of some dictionary. It is guaranteed to converge in no more than N steps, where N is the number of dictionary columns [78]. A convex optimization method like LASSO can also be applied here [21]. The approximation step is as follows:

$$\forall q, \hat{\boldsymbol{x}}_{i,q}^{(t_d)} = arg \min_{\boldsymbol{x} \in \mathbb{R}^N} \| \boldsymbol{y}_{i,q} - \hat{\boldsymbol{D}}_i^{(t_d-1)} \boldsymbol{x} \|_2^2 \text{ subject to } \| \boldsymbol{x} \|_0 \leq K \tag{3.42}$$

where $\boldsymbol{y}_{i,q}$ *and* $\hat{\boldsymbol{x}}_{i,q}^{(t_d)}$ *denote the* q^{th} *sample and the signal vector at node* i*, respectively.*

$\hat{\boldsymbol{X}}_i^{(t_d)} = \left[\hat{\boldsymbol{x}}_{i,1}^{(t_d)} \quad \hat{\boldsymbol{x}}_{i,2}^{(t_d)} \quad \cdots \quad \hat{\boldsymbol{x}}_{i,Q_i}^{(t_d)}\right]$. It has been shown that computing the sparse coefficients locally is acceptable at each iteration as long as the dictionary atoms in $\hat{\boldsymbol{D}}_i^{(t_d-1)}$ remain close to each other [9].

Next is the dictionary update step. For this, cloud K-SVD uses a distributed power iterations model first described in section 3.7 to find the dominant eigenvector denoted $\boldsymbol{q}$ of a square, positive-semidefinite residual matrix $\boldsymbol{M}$ [9] we have previously described, which is defined to have no eigenvalues less than or equal to zero. The power method is denoted as $\hat{\boldsymbol{q}}^{(t_d)} = \boldsymbol{M}\hat{\boldsymbol{q}}^{(t_d-1)}$, where $\hat{\boldsymbol{q}}^{(t_d)}$ is the estimate of the dominant eigenvector and approaches $\boldsymbol{q}$ for every iteration ($t_d \to \infty$). We denote the initial value of $\boldsymbol{q}$ as $\boldsymbol{q}^{init}$ and may be a non-zero vector. All nodes begin with starting vector $\boldsymbol{q}^{init}$ and matrix $\boldsymbol{M}_i$ [79]. At each iteration of the power method, the nodes reach a average consensus on an estimate of $\boldsymbol{q}$ for the matrix $\boldsymbol{M}$ where

$$\boldsymbol{M} = \sum_{i=1}^{H} \boldsymbol{M}_i \tag{3.43}$$

and H *represents the total amount of nodes.*

If we apply the power method to $\boldsymbol{M}$, we observe that:

Algorithm 1: The cloud K-SVD algorithm by [9].

Input: Local data $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_H$, problem parameters N and K, and doubly stochastic matrix $\boldsymbol{W}$

Output: Dictionary $\hat{\boldsymbol{D}}$.

Generate $\boldsymbol{d}^{ref} \in \mathbb{R}^M$ and $\boldsymbol{D}^{init} \in \mathbb{R}^{M \times Q}$ randomly, set $t_d \leftarrow 0$ and $\hat{\boldsymbol{D}}_i^{(t_d)}$, $i = 1, 2, \ldots, H$. `// Initialization`

while *stopping criteria not satisfied* **do** `// Sparse Approximation`

  $t_d \leftarrow t_d + 1$.

  The i^{th} site solves $\forall q, \hat{\boldsymbol{x}}_{i,q}^{(t_d)} \leftarrow \arg\min_{\boldsymbol{x} \in \mathbb{R}^K} \| \boldsymbol{y}_{i,q} - \hat{\boldsymbol{D}}_i^{(t_d-1)} \boldsymbol{x} \|_2^2$ s.t. $\| \boldsymbol{x} \|_0 \leq K$

  for $n = 1$ **to** N **do** `// Dictionary Update`

    $\hat{\boldsymbol{E}}_{i,n,R}^{(t_d)} \leftarrow \boldsymbol{Y}_i \hat{\boldsymbol{\Omega}}_{i,n} - \sum_{j=1}^{n-1} \hat{\boldsymbol{d}}_{i,j}^{(t_d)} \hat{\boldsymbol{x}}_{i,j,T}^{(t_d)} \hat{\boldsymbol{\Omega}}_{i,n}^{(t_d)} - \sum_{j=n+1}^{N} \hat{\boldsymbol{d}}_{i,j}^{(t_d-1)} \hat{\boldsymbol{x}}_{i,j,T}^{(t_d)} \hat{\boldsymbol{\Omega}}_{i,n}^{(t_d)}$

    $\hat{\boldsymbol{M}}_i \leftarrow \hat{\boldsymbol{E}}_{i,n,R}^{(t_d)} \hat{\boldsymbol{E}}_{i,n,R}^{(t_d)^T}$

    Generate $\boldsymbol{q}^{init}$ randomly, set $t_p \leftarrow 0$ and $\hat{\boldsymbol{q}}_i^{(t_p)} \leftarrow \boldsymbol{q}^{init}$

    while *stopping criteria not satisfied* **do** `// Power Method`

      $t_p \leftarrow t_p + 1$

      Set $t_c \leftarrow 0$ and $\boldsymbol{z}_i^{(t_c)} \leftarrow \hat{\boldsymbol{M}}_i \hat{\boldsymbol{q}}_i^{t_p - 1}$

      while *stopping criteria not satisfied* **do** `// Consensus Averaging`

        $t_c \leftarrow t_c + 1$

        $\boldsymbol{z}_i^{(t_c)} \leftarrow \sum_{j \in \mathcal{N}_i} \boldsymbol{w}_{i,j} \boldsymbol{z}_i^{(t_c-1)}$

      end

      $\hat{\boldsymbol{v}}_i^{(t_p)} \leftarrow \boldsymbol{z}_i^{(t_c)} / [\boldsymbol{W}_1^{t_c}]_i$

      $\hat{\boldsymbol{q}}_i^{(t_p)} \leftarrow \hat{\boldsymbol{v}}_i^{(t_p)} / \| \hat{\boldsymbol{v}}_i^{(t_p)} \|_2$

    end

    $\hat{\boldsymbol{d}}_{i,n}^{(t)} \leftarrow \operatorname{sgn}\Big(\langle \boldsymbol{d}^{ref}, \hat{\boldsymbol{q}}_i^{(t_p)} \rangle \Big) \hat{\boldsymbol{q}}_i^{(t_p)}$

    $\hat{\boldsymbol{x}}_{i,n,R}^{(t)} \leftarrow \hat{\boldsymbol{d}}_{i,n}^{(t)^T} \hat{\boldsymbol{E}}_{i,n,R}^{(t_d)}$

  end

end

Result: The desired dictionary $\hat{\boldsymbol{D}}_i^{t_d}$ and associated signal $\hat{\boldsymbol{X}}_i^{t_d}$, $i = 1, 2, \ldots, H$

$$\hat{\boldsymbol{q}}^{(t_d)} = \boldsymbol{M} \hat{\boldsymbol{q}}^{(t_d-1)} = \Big(\sum_{i=1}^{H} \boldsymbol{M}_i \Big) \hat{\boldsymbol{q}}^{(t_d-1)} = \sum_{i=1}^{H} \boldsymbol{M}_i \hat{\boldsymbol{q}}_i^{t_d - 1} \tag{3.44}$$

as in [9] and [10].

Each node maintains a $\hat{\boldsymbol{q}}^{(t_d-1)}$ that holds the previous error. Consensus averaging is then used to find the last summation of the above equation. Since the output $\hat{\boldsymbol{q}}^{(t)}$ requires both consensus averaging and the power method step, the local estimate at each node is affected by error from both algorithms. If we perform enough consensus iterations t_c and power iterations t_p, each node's estimate of the dominant eigenvector will converge to the true value for the residual $\boldsymbol{M}$ [53]. To compute an estimate of a local dictionary, given by $\hat{\boldsymbol{D}}_i^{t_d} \forall i = 1, 2, \ldots, H$, to form a global dictionary estimate given by $\hat{\boldsymbol{D}}^{t_d}$ at iteration t_d we update atoms $\boldsymbol{d}_{i,n}$ at each node by collaboratively finding the total error which that atom must minimize. This requires us to find local error matrix called $\hat{\boldsymbol{E}}_{i,n,R}^{(t_d)}$. To do so, we first need to find the indices used by the dictionary at each node, denoted $\boldsymbol{\omega}_n$. In this case, $\boldsymbol{\omega}_{i,n}^{t_d}$ refers to the indices used by the dictionary atom

$\boldsymbol{D}_{i,n}$ at iteration t_d:

$$\boldsymbol{\omega}_{i,n}^{(t_d)} = \left\{ q \parallel 1 \leq q \leq Q_i, \boldsymbol{x}_{i,n,T}^{(t_d)}(q) \neq 0 \right\} \forall i \tag{3.45}$$

where $\boldsymbol{x}_{i,n,T}^{(t_d)}(q)$ *denotes the* q^{th} *element of* $\boldsymbol{x}_{i,n,T}^{(t_d)}$.

To collectively store $\boldsymbol{\omega}_{i,n}^{(t_d)}$ we define $\boldsymbol{\Omega}_{i,n}^{(t_d)} = Q \times \boldsymbol{\omega}_{i,n}^{(t_d)}$ as a binary matrix with ones in $(\boldsymbol{\omega}_{i,n}^{(t_d)}(q), q)$. Thus we can define $\hat{\boldsymbol{E}}_{i,n,R}^{(t_d)} = \hat{\boldsymbol{E}}_{i,n}^{(t)} \boldsymbol{\Omega}_{i,n}^{(t_d)}$. $\hat{\boldsymbol{E}}_{i,n}^{(t_d)}$ is given with this equation:

$$\hat{\boldsymbol{E}}_{i,n}^{(t_d)} = \boldsymbol{Y}_i - \sum_{j=1}^{n-1} \hat{\boldsymbol{d}}_{i,j}^{(t_d)} \hat{\boldsymbol{x}}_{i,j,T}^{t_d} - \sum_{j=n+1}^{N} \hat{\boldsymbol{d}}_{i,j}^{(t_d-1)} \hat{\boldsymbol{x}}_{i,j,T}^{t_d-1} \tag{3.46}$$

where j *is the dictionary atom for* $1, 2, \ldots, N$, *except for* $j = i$.

Here we update the atoms in numerical order, as the first atoms that have already been updated are in iteration t_d and those that have not yet been updated are in $t_d - 1$. Next step is to calculate the SVD as in K-SVD of $\hat{\boldsymbol{E}}_{n,R}^{(t_d)} = \sum_{i=1}^{N} \hat{\boldsymbol{E}}_{i,n,R}^{(t_d)}$ for all nodes to find $\boldsymbol{U}_{1,n}$ and $\boldsymbol{V}_{1,n}$. Again, we make the following updates:

$$\left(\hat{\boldsymbol{d}}_{i,n}, \hat{\boldsymbol{x}}_{i,n,R} \right) = (\boldsymbol{U}_{1,n}, \boldsymbol{\Delta}_{(1,1)} \boldsymbol{V}_{1,n}) \forall i \tag{3.47}$$

By definition we know that:

$$\hat{\boldsymbol{x}}_{i,n,R} = \hat{\boldsymbol{d}}_{i,n} \hat{\boldsymbol{E}}_{i,n,R}^{(t)} = \boldsymbol{U}_{1,n} \hat{\boldsymbol{E}}_{i,n,R}^{(t)} = \boldsymbol{\Delta}_{(1,1)} \boldsymbol{V}_{1,n} \forall i \tag{3.48}$$

To find an approximation of $\hat{\boldsymbol{E}}_{i,n,R}^{(t_d)}$, we use a distributed power method to find its dominant eigenvector. For this to work, $\hat{\boldsymbol{E}}_{n,R}^{(t_d)}$ must be a square and positive semi-definite matrix. This can be archived by multiplying the matrix by its transpose as in $\hat{\boldsymbol{E}}_{n,R}^{(t_d)} \hat{\boldsymbol{E}}_{n,R}^{(t_d)^T}$ and applying the power method to the resulting matrix defined as:

$$\hat{\boldsymbol{M}}^{(t_d)} = \sum_{n=1}^{H} \hat{\boldsymbol{M}}_n^{(t_d)} = \sum_{n=1}^{H} \hat{\boldsymbol{E}}_{n,R}^{(t_d)} \hat{\boldsymbol{E}}_{n,R}^{(t_d)^T} \tag{3.49}$$

We then find the dominant eigenvector of $\hat{\boldsymbol{M}}_n^{(t_d)}$. By definition, it is the U_1 of $\hat{\boldsymbol{E}}_{n,R}^{(t_d)}$, since for any real matrix $\boldsymbol{A}$, the left-singular vectors of $\boldsymbol{A}$ are the eigenvectors $\boldsymbol{A}\boldsymbol{A}^T$. With this in mind, $\hat{\boldsymbol{d}}_{i,n}$ is the result of our distributed power method. We locally compute $\hat{\boldsymbol{d}}_{i,n}^{(t_d)^T} \hat{\boldsymbol{E}}_{i,n,R}^{(t_d)}$ to set $\hat{\boldsymbol{x}}_{i,n,R}$ and lastly set $\hat{\boldsymbol{x}}_{i,n} = \hat{\boldsymbol{x}}_{i,n,R} \boldsymbol{\Omega}_{i,n}^{(t_d)^T}$ to place the zeros into the row of coefficients. Now the dictionary update for one atom is done and when all atoms have been updated, the entire dictionary update step is complete. The loop repeats for each dictionary learning iteration (t_d) [10].

Chapter 4

Cloud computing theory

This part of the theory tells the story of distributed cloud computing, its concepts and applications in section 4.1. Here we provide an appetizer for hot topics in the software industry such as microservices, containers, the Docker project and Kubernetes, all of which we use in our design and implementation of cloud K-SVD. Section 4.2 gives a layman's introduction to microservices with an example of an e-commerce store provided by C. Gammelgaard in [80], before examining how we can build and run these microservices in an encapsulated and closed environment using Docker containers in section 4.3. Finally we introduce an orchestration engine that can create, start and stop these containers called Kubernetes in section 4.4 that will make up our environment for all practical experiments.

4.1 Concepts of cloud computing

Cloud computing, as an on-demand availability of computer system resources, offers a scalable way to allocate more power to an online software solution, better performance as nodes can work simultaneously and flexibility for re-provisioning, adding or expanding resources via scaling. Compared to having just a single but performance-decent workstation to a modern cloud network like Google Cloud or Amazon Web Services (AWS) with seemingly a limitless number of nodes at your disposal, it begs the question whether we should perform all computation of these mathematical instruments in the cloud rather than relying on a single computer. In order to implement cloud K-SVD, something of particular interest are small isolated services called microservices, the container-based method that the Docker tool offers and a way of managing these containers in Kubernetes. We will review general cloud development concepts here.

Microservices: A modern version of the service-oriented architecture (SOA) style that organizes a large application into smaller, isolated and loosely coupled services with a fine-grained interface and a single responsibility. These smaller entities are self-substantiated and can be deployed independently of other services, however they often depend on other to perform a task. This approach improves modularity, parallelizes development as microservices can be developed, refactored and deployed separately and moreover makes the application easier to understand as each microservice should only contain the business logic needed for its capability. In our case, we will split the implementation of cloud K-SVD into several lightweight microservices to independently scale certain parts of the system.

Containers: Back in the old days, we could only run one application per server and the open-systems world of Windows and Linux did not have the features to safely and efficiently run multiple applications on the same server. To the rescue comes virtual machines (VM) and CPU virtualization technologies that allowed businesses to run multiple applications on a single server, only with the caveat that these required a dedicated OS which took up a lot of space

and resources on a host OS. Google and other have since been using container technologies to address the shortcomings of the VM model. A container is a lightweight implementation of the VM model that does not require its own full-blown OS. It simply uses the resources of the host OS. If a single node runs multiple containers, they would all share the host's OS. Because a container is not a full-blown OS, starts times are faster. There is no kernel inside of a container that needs locating, decompressing, and initializing like a regular OS would, not to forget all the hardware initializing a normal kernel bootstrap does. The single shared kernel, down at the OS level, is already started, so containers typically start in less than a few seconds. Usually a container contains some microservice application source code, the dependencies needed to run it and a tiny set of OS libraries to communicate with the underlying system. What is smart is that containers are usually very lightweight, fast, portable and can normally run independently on either Windows or Linux systems. Modern containers started in the Linux world and Google LLC has contributed many container-related technologies to the Linux kernel, however this remained for the few and it was not until Docker came along that containers were effectively democratized and available to the masses.

Docker: Docker is a container engine and software application that can create, manage and orchestrate containers and is being developed and maintained by Docker Inc. based on the open-source project Moby and freely available on GitHub. The engine is what runs containers on a host OS similar to how the core hypervisor technology ESXi runs virtual machines in WMware. The Docker client allows developers to create either Windows or Linux images that contain the application, all the code and dependencies to run it and a set of basic operating system libraries. One can think of docker images as similar to VM templates - A VM template is like a stopped VM — a Docker image is like a stopped container. Once Docker has created the image, it is usually pushed to a repository and later pulled by either Docker itself or Kubernetes to create an actual container, a runtime instance of an image.

Kubernetes: The word *Kubernetes* is Greek for "helmsman", i.e. the person holding a ship's steering wheel, and that is what it does with containers - it creates, manages and removes them when they are no longer needed. Kubernetes is an open-source container-orchestration project out of Google that was made public in 2014. It can automate the creation, deployment, scaling and control of containers or so-called pods in a live environment and uses Docker as its default container runtime engine. In Kubernetes, the term "pods" is used to denote a logical runtime environment for one or multiple containers. It is basically a logical classification of one or mulitple tightly-coupled containers. A pod then run on one or a multiple of nodes. For developers, Kubernetes abstracts away the hardware infrastructure and exposes the whole data center as a single enormous computational resource, see figure 4.1. This way we can deploy and re-deploy software applications without knowing anything about the actual servers, how many there are, their operating-system or perhaps quirky node-specific configurations. Everything is just one endpoint. Even if our application has multiple parts (for example a front-end and a back-end service), Kubernetes automatically selects the best server (node) to run each service on and enables them to easily find and communicate with each other.

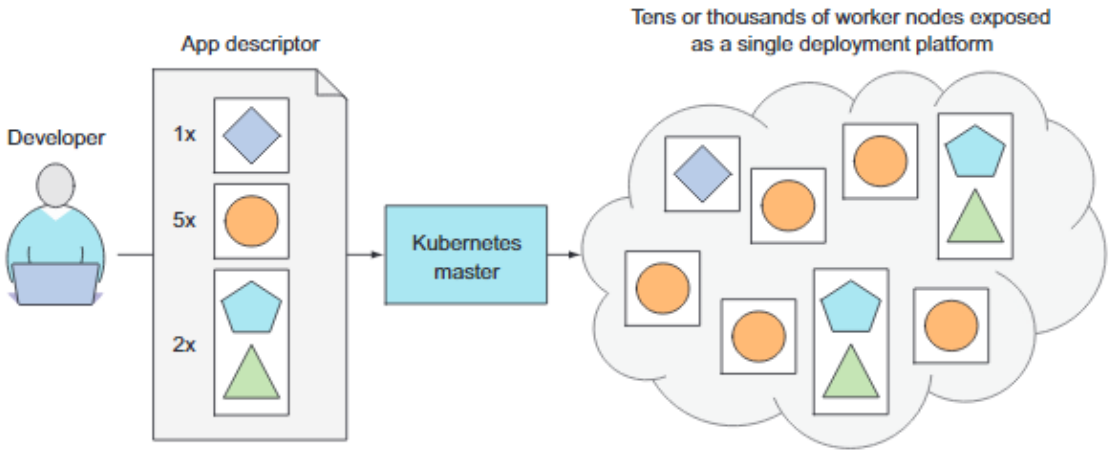

Figure 4.1: *Kubernetes exposes an entire data center as a single resource. The developer merely provides a list with applications to run and Kubernetes does the rest. Source: [81].*

4.2 Microservices in the cloud

Microservices is a software development method that has gained traction in recent years because of the way it eases development and deployment of application parts at the same time compared to a traditional monolith program. The way it works it by following a lightweight variant of the service-oriented architecture (SOA) design style that organizes the application as a collection of distributed loosely-coupled services. These services are isolated parts of the application (that is, they contain all the code and dependencies needed to run as one entity) and can be taken down for maintenance and upgrades at any point in time without affecting the other. They are self-reliant in case of network partitions, communication delays or service disruptions. They are often built with a single narrowly-focused capability that a remote API exposes to the rest of the system. If we think of an e-commerce store like Amazon and break down its capabilities, we may come up with a list of functionality pieces that can be placed in several separate microservices:

- Receive a list of stock at the warehouse.
- Presenting products in a list to the customer.
- Calculating shipping costs to domestic or foreign addresses.
- Determine if loyal customers are eligible to a bonus.

Next step is to make a microservice for each capability in our list as an individual service that we can examine, test, run and decommission independently of the other microservices. In practice, a service often rely on others to accomplish an action, for example if the customer wants to place an order, it may involve multiple services, but the idea is that they run in their own separate process and do not influence each other. In a real scenario, often a centralized API gateway serves user requests by instructing a set of microservices to each perform a specific task. Any communication between the microservices should favor event-based asynchronous collaboration over making synchronous remote calls, but this depends on the application and use case. Some calls like retrieving a list of products can be made synchronous, while maintaining a bonus database for loyal customers can simply be notified whenever someone purchases a product that comes with bonus point. A lot goes into the design and develop process of microservices

that we will not cover here, but it should be clear that a microservice is just one of many pieces that constitutes a large application and that it is responsible for providing a single capability.

This brings us to why the microservice approach is useful for implementation cloud K-SVD in our thesis: It allows splitting up the application in separate parts, build them as containers and scale them independently via the Kubernetes management tool. Several characteristics [80] of a microservice are especially useful in our case:

- It is responsible for providing a single capability. This means that we can place the pure cloud K-SVD algorithm in a microservice and then rely on that service to provide this capability. Another microservice could provide other necessary services. This makes the distributed application a lot more clearer and easier to grasp. This also means that if the capability or feature we need to scale is in a single service, we need only scale that particular one and not all the others.
- Is is individually deployable. We can build our application with its dependencies as an isolated piece of software that can be deployed on its own, for example a preprocessing service can be deployed without worrying about another unrelated service that does postprocessing work. Software containers is a way of achieving this.
- It runs in one or more separate processes or threads. This ensures that services will not trip over each other and can each be evicted from the system by terminating its thread. It also allows us to monitor resource consumption on a process-level.
- It owns and stores the data that belongs to its capability. This means we can manage a local data store and how to access it on a per microservice-level, so other services need not be concerned of this detail. This reduces complexity in our solution, as the service that does cloud K-SVD should not be burdened by having to know the peculiarities of properly handling input data.
- It is small enough for a single team. We want to manage and control services of relative small size, so any changes that should only implicate one particular service will only do so. If we compare the monolith approach, a change one place may require adjustments and work another because of tight coupling.
- It is scalable. Especially related to what Kubernetes offer, we want to scale our application on a capability-level, meaning that if we need more power to help prepare input data, we can easily power up some more services.

It should be clear by now that microservices is merely a working methodology of how to compartmentalize an application that has multiple working parts. To work in the cloud, where infrastructure is often heterogeneous and of an unknown quantity, the next step is to employ a container approach where our application, dependencies and configuration files are bundled in an image that contains everything it needs to run. Docker is a tool that can create and run such lightweight containers, which we will review next.

4.3 Building containers with Docker

Docker is a middleware engine and client toolkit invented by Docker Inc. that can create and run an application as a whole or in separate parts in a closed environment called a container. It is basically an abstraction at the application layer that package code, system tools, dependencies and configuration files together so what when the container is deployed, no local libraries or certain systems are needed as the application already have what it needs. A running container

uses the OS kernel of its host computer and can share a platform with multiple other containers, each running in an isolated, consistent and disposable environment in user space. Containers typically take up less space than traditional VM's for this reason. The Docker client is a user interface that can upload commands and application content to the part of Docker called daemon. The daemon can package application code and everything it needs to run into a image specified by a Dockerfile. The Dockerfile designates the base image, like a lightweight distribution of Linux x64, together with all dependencies and a command instruction set for the application when it is deployed. When the Docker daemon builds the image, a new layer is created for each individual command in the Dockerfile. After pulling all the layers of the base image and the dependencies, Docker will add a new layer on top of them and add your application code (for example app.js) to that. Then it will create yet another layer that will specify the command that should be run when the image is deployed and executed. The complete image then contains a lightweight Linux operating-system, all your files and program dependencies, the application itself and the instruction set that tells the container runtime how it should run it [81]. The process is illustrated in figure 4.3.

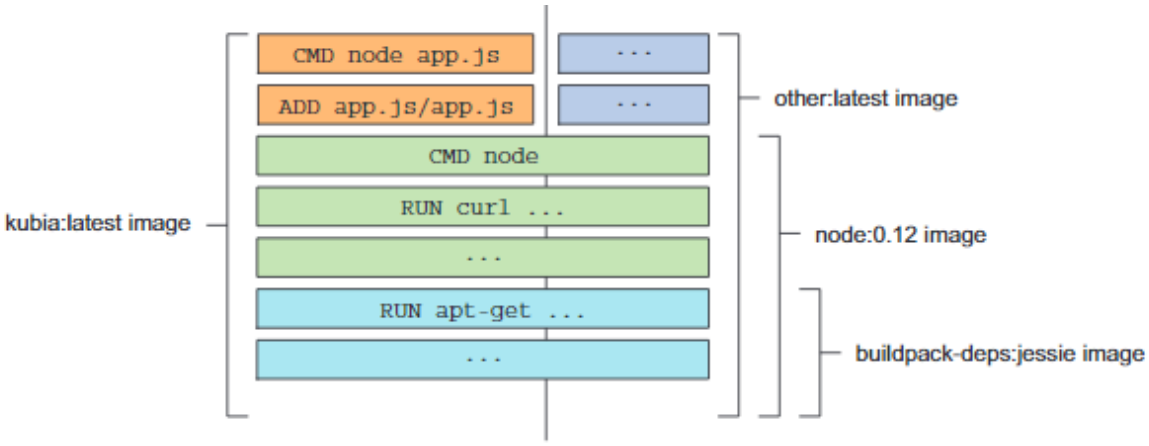

Figure 4.2: *A container image consists of layers that can shared among other images. Source: [81].*

Image files are often built locally at the developer's desk, labeled and tagged with the current release version and then pushed to a image repository like Docker Hub, where a container orchestration engine like Kubernetes can fetch and deploy them in a live environment. You can see in figure 4.2 how a final bundled image looks.

The idea with Docker and containers is really about abstracting and masking the complex underlaying structure of the heterogeneous application interfaces that a program needs, while providing a simple, accessible and consistent way of creating and running application anywhere. Say goodbye to manual interventions and quirky software configurations and hello to consistent application portability. In the next section we will look at Kubernetes, which is a container orchestration platform that can automate the whole process of fetching built images, deploying them in a distributed environment and frequently perform health checks on running containers. It uses a Docker as its default container runtime, which means that every Kubernetes node uses the Docker runtime to execute a container in. Recall that a container is simply a runtime instance of an image. Moreover Kubernetes can also manage the computational resources allocated to each container and scale appropriately if a high demand is suddenly placed on the system.

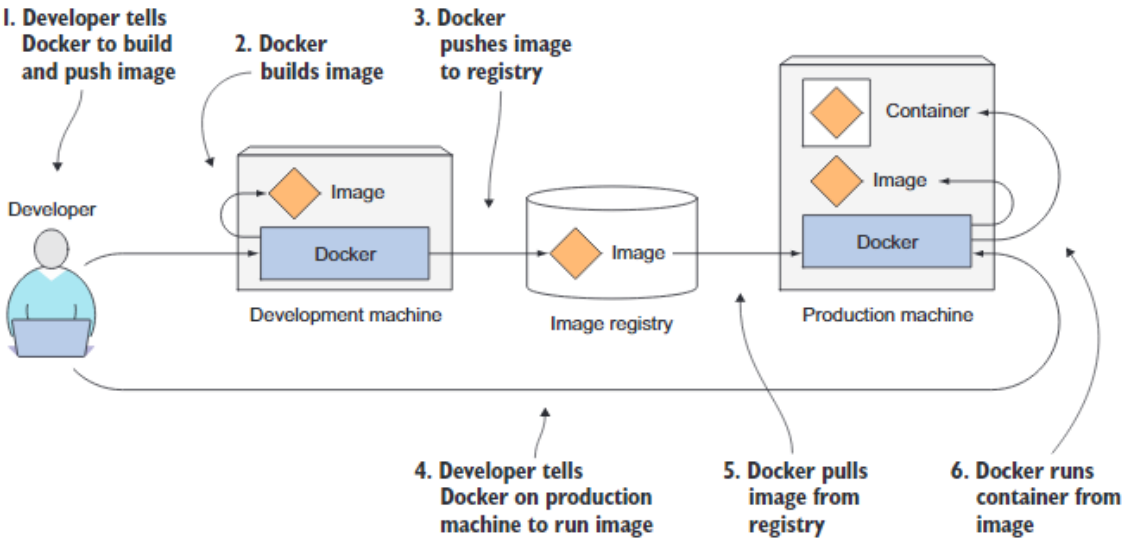

Figure 4.3: *How Docker builds, manages and deploys images. Source: [81].*

4.4 Controlling containers with Kubernetes

Kubernetes is an open-source container orchestration engine made by Google in 2014 based on a decade-old internal system called *Borg* and later *Omega* to assist developers and system administrators manage the vast amount of applications and services Google had running. It not only simplified development, but Kubernetes helped Google archive a much higher utilization of their server infrastructure. Because a company like Google maintains and runs servers on such as massive-scale, even small improvements in resource utilization per device mean a substantial cost reduction when added up. Today, Google processes 40,000 search requests every second and 3,5 billion searches a day. Over the last two years alone, 90 percent of the data in the world was generated, so we truly live in a big data era. This is why we need a smart way to utilize the resources we have, to scale up the number of services when applicable, to run multiple instances of a service in a single computer and reduce the system overhead by relying on lightweight containers that have a single capability. Kubernetes makes all of this possible for a modern distributed system and has since its inception become a hallmark for modern cloud computing. It does however have a slightly complex architecture with multiple components interacting with each other in complex ways, which we will try to demystify next. This will provide a summarization of what Kubernetes is and what it does, not a complete exposition of the architecture. For further reading, we recommend the book on Kubernetes by Marko Lukša [81].

We have seen a birds-eyes of what is in Kubernetes, basically this cloud of heterogeneous computational resources that expose a single endpoint for developers to deploy applications to. At the hardware-level, Kubernetes is composed of many nodes of two types [81], see figure 4.4:

- The *Kubernetes Control Plane*, which is hosted on a master node and manages the whole Kubernetes cluster. This is responsible for keeping track of running instances, schedule new *pods* and store the cluster configuration.
- Worker nodes that run the containerized applications you deploy in a container runtime like Docker.

The Control Plane: If we look at the first type, the master node, it consists of four essential components that make it work. These can run on single computer node or be replicated to

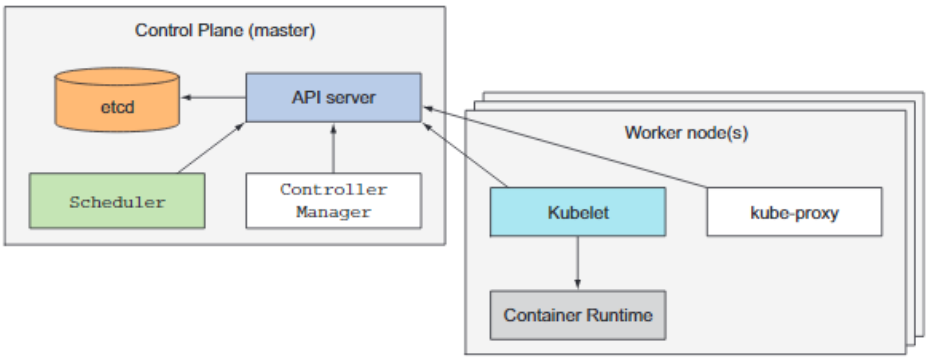

Figure 4.4: *The two essential Kubernetes components. Source: [81].*

improve data center resilience. First, the *Kubernetes API Server* component that allows the worker nodes to read and update resources in a database in a consistent and reliable manner. With an API you streamline the access to your database while enforcing things like authentication and communication protocol. The *Kubernetes API Server* does the same. Next is the *Scheduler*, an integral part of Kubernetes. It schedules your apps by assigning worker nodes to each part of it. For example an application with a frontend and backend service, it could assign the former to two running nodes and the latter pod to a single one depending on load. Next is the *Controller Manager*, a service that manages cluster functions, handles node failures and performs pod replication in case of failure or scaling. Lastly we have the *etcd*, a distributed database that stores the cluster configuration. *The Control Plane* manages the cluster, stores the configuration and keeps track of running nodes. It relocates pod's if a node fails or reschedules your running instances to a single node if the demand is low. It does not however run any application code or answer user requests, this is what the worker nodes are for.

The worker nodes: A Kubernetes worker node runs your containerized application. It hosts a Docker *container runtime* that can turn an image into a running instance. You can opt to go with another engine like *rkt*, but since Docker was the first container platform to make containers mainstream, it has become the default runtime in Kubernetes. Next is the *Kubelet* component, a node agent that manages containers on its node. It downloads specifications from the API server and ensures that the containers described in those specifications are running and healthy. Lastly we have the *kube-proxy*, which is a network proxy component that maintains network rules. These allow communication to and from the pods running in the node from network sessions inside or outside of the cluster. If a pod needs information from the API server, it typically goes through the kube-proxy.

Examine figure 4.6 to get an understanding of how applications are deployed in Kubernetes. We start by packing our application into one or more container images, then we push these to a registry and post an app description to the Kubernetes API server that tells it what container images to deploy, which are co-located (placed on the same node) and which are not. We can also specific the number of replicas with want of each pod. Recall a pod is a logical containment of one or more containers that Kubernetes refer to. Two containers should be put in the same pod if they are tightly-coupled, like a worker that does cloud K-SVD with its own dedicated storage service. A pod can be replicated on multiple nodes, but a pod can never span two nodes. Figure 4.5 shows how a typical pod can look with one main container and two supporting containers for data access. Additionally, the descriptor also says which services should be exposed to both internal or external clients. When the API server processes the description, the *Scheduler's* job is to schedule containers onto the available pool of workers nodes based on what resources they require and what is available in the cluster. *The kubelet* on those nodes then orders the container

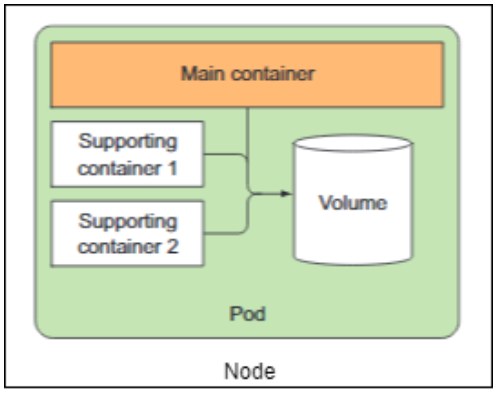

Figure 4.5: *One example of a pod. It should contain tightly-coupled containers, typically a main container and a few supporting ones for logging or data access purposes.*

runtime (Docker) to fetch the required container images and start the containers. We see in figure 4.6 that the container with an orange circle-shaped image needs five replicas, whereas the one with a triangle and pentagon inside needs two. The latter is co-located, so it should be deployed as a pod on the same node.

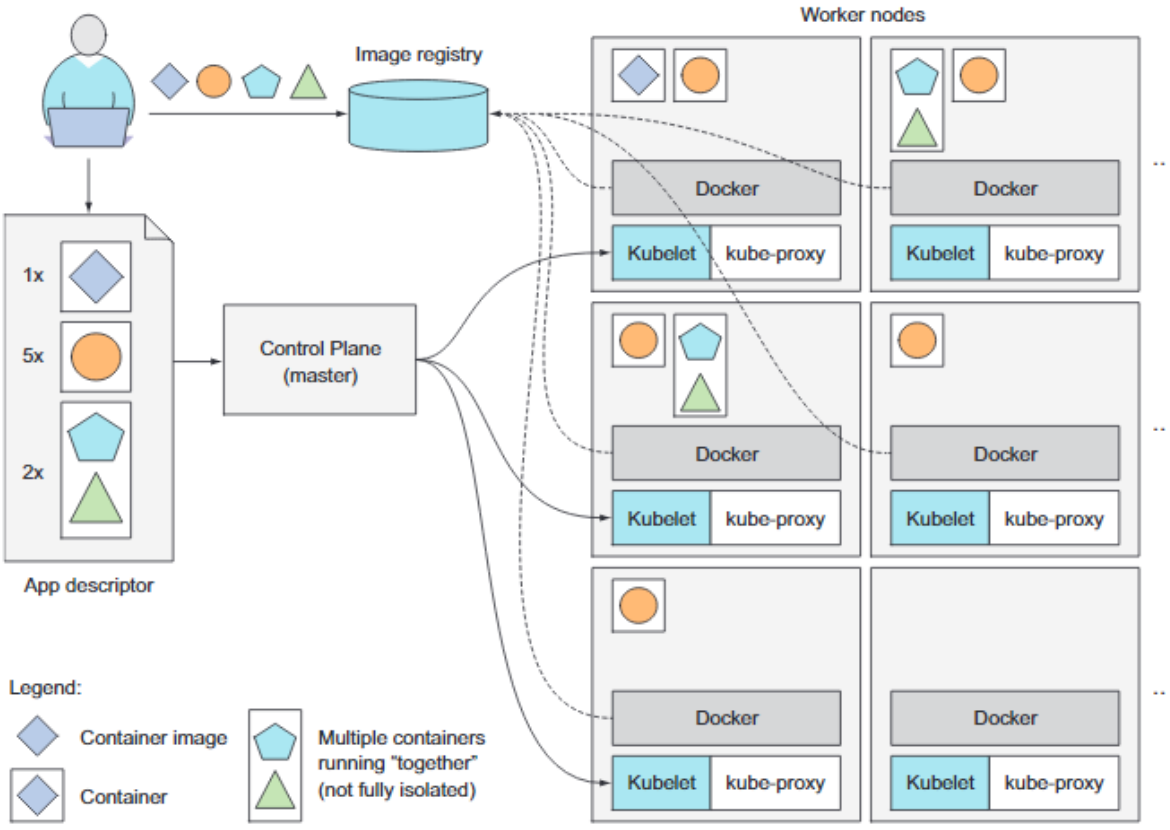

Figure 4.6: *An basic view of the Kubernetes architecture and what the components do. Source: [81].*

If a pod needs to collaborate with another pod, it can do so by a simple DNS lookup or query the Kubernetes API server directly to get either the address of a service that is backed by the pod it wants to contact or get the address of the pod itself. Services are a way in Kubernetes to load-balance requests between a pool of pods and decouple the callee from the caller. A service offers clients a single, constant point of entry to a group of pods that provide the same capability,

like a website or file server [81]. Depending on client needs, a service can also provide a list of nodes offering the capability instead of a single endpoint. We should bear in mind that pods are ephemeral: They may come and go at any time, whether it is because they have been closed down to make room for other pods, someone scaled down the number of pods or because a cluster node has failed [81]. This is why it makes sense to add an additional layer between the client and the pods.

We went with Kubernetes because it is a state-of-the-art platform with a high relevance for the distributed computing industry and because we can create a containerized application with all our microservices inside that can be tested and evaluated in a simulated setup and later deployed in an enterprise-state cluster without changing anything in the application. Kubernetes is platform agnostic and a container can be deployed anywhere at any cloud provider, be it Google Cloud, Amazon Web Services or in our own on-premise configuration. The applications are decoupled from the infrastructure. This makes Kubernetes ideal for testing and evaluating of distributed systems algorithms built as encapsulated microservices that can immediately be transfered to the real world. Kubernetes gives us fault-tolerance via self-healing, service scaling based on current or past load and a better utilization of the hardware because it can calculate performance metrics on the fly and move applications around freely in the cluster.

Chapter 5

Design and implementation

Up to this point, we have provided a detailed view of the theoretical concepts behind sparse approximation, dictionary learning and how *cloud K-SVD* work. This chapter will introduce our solution design and how we have implemented it to facilitate tests and experiments. Since we were not able to opt for a fully-fledged cloud solution at for example *Google Cloud* or *Amazon Web Services*, with equipment provided by the university we were able to build an on-premise *simulation* setup of a cloud system by installing Kubernetes on four Raspberry Pi Model 4's. This way we could easily deploy our microservices (our software) to the cluster as *pods*, monitor it in real time and scale the number of pods based on a concrete test scenario. By using an on-premise system in a laboratory environment, we can evaluate *cloud K-SVD* without interfering factors that may arose at a cloud provider, like heterogeneous hardware, unpredictable resource allocation and eventual network delays that are out of our control, though providers like Google or Amazon in general provide stable infrastructure components and a managed, production-ready environment for distributed systems at a considerable cost, of course.

Recall that pods in Kubernetes act like regular computers, you can think of. They process their own sequence of events, have an equal amount of resources at their disposal and can be started and stopped independently. Pods run on physical cluster hardware (the Raspberry Pi's), which we call nodes. They provide pods with the operating-system API's and resources like processing time, OS libraries and memory they need.

5.1 Overall design and solution

To run cloud K-SVD effectively and handle data of large quantities, we chose a layered software architecture. This means that we decomposed our system into three independent software components that basically do three separate things: First block (preprocessor) processes all the incoming training data, divides it in equal-sized data portions and forwards it. Second block (worker) receives the preprepared data, performs a number of cloud K-SVD iterations, see section 3.8, and forwards the result to the last block in the chain, the third block. The third block (postprocessor) receives the result and does some postprocessing tasks like aggregates data, saves statistics and expose all this through a public API we can access from our test computer. Figure 5.1 is the system model.

Figure 5.2 shows a small sequence diagram of the interaction between the preprocessor, worker and postprocessor pod. It shows all steps in our implementation and the cloud K-SVD protocol at a high level. The drawing has been simplified a bit, hence data is sent from preprocessor to individual workers directly, not through each other as it may seem. This is also the case when data is saved in the end.

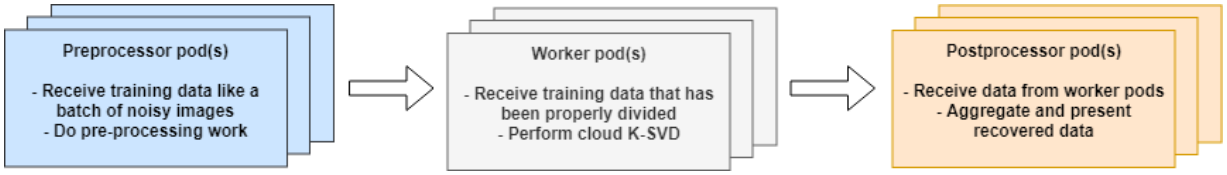

Figure 5.1: *The system model. We create three different pods: Preprocessor, worker and postprocessor. By making this distinction, we can define, deploy and scale them independently in Kubernetes, which allows us to better scale the system by load and performance requirements.*

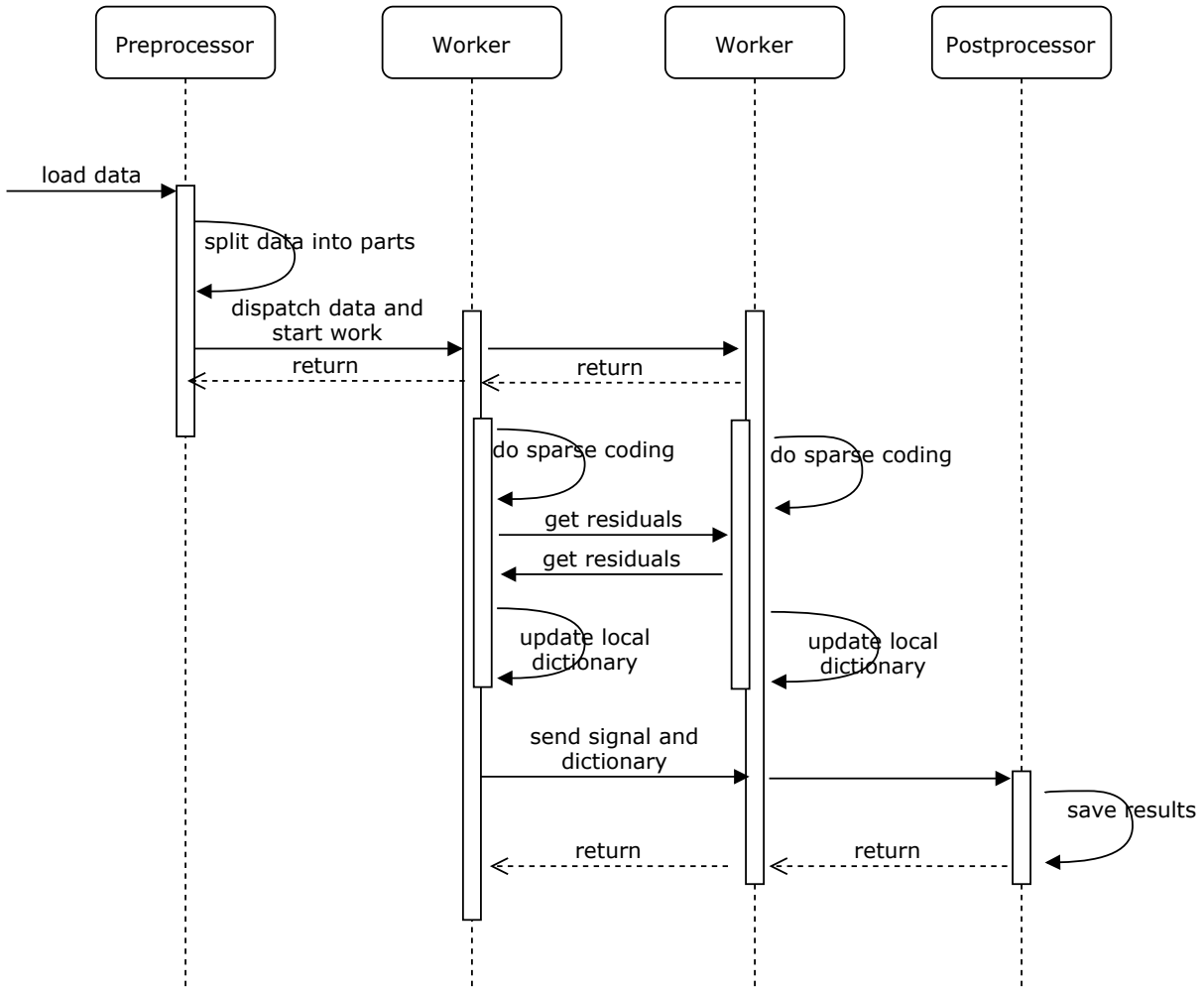

Figure 5.2: *A brief sequence diagram that demonstrates how data is loaded onto two workers. Cloud K-SVD is then started and at the end a signal matrix and a local dictionary is produced that we collect at the postprocessor pod.*

5.2 Cluster considerations and operating-systems

The main purpose of the system is to facilitate a solution to our main problem: Evaluate how cloud K-SVD can be used for dictionary learning and approximations, see 1.3. This means a long list of experiments have to be performed for different kinds of input data and for certain parameters of the algorithm. Other studies that tested cloud K-SVD [15] [32] [9] relied on virtual machines (VM) or the Python *MPI* library when testing distributed aspects, however the problem of using virtual machines on a single computer is that eventual problems in communication links are not exposed and even though modern computer systems have multi-core processors, it is far

from the same. Evaluating an algorithm across multiple virtual machines but on the same CPU die is much different than evaluating it across several independent systems, that work together. In reality, communication delays exist since data packets have to be put, transmitted and received over actual wires, multiple processing units have to work simultaneously and collaboratively by message passing exchange data, which is asynchronous in nature, and other variables just to mention a few of the factors that play a role in a distributed system. Furthermore, testing such an algorithm in a centralized computer makes it impossible to show and evaluate the strengths of real horizontal hardware distribution and scaling in terms of processing time, memory consumption and load. These are reasons we opted for a real distributed system to evaluate cloud K-SVD.

We conduct all experiments in this thesis using a Linux Kubernetes cluster installed on four Raspberry Pi's. These are nodes in the system and can host multiple Kubernetes pods of type preprocessor, worker or postprocessor. In appendix B, all the configuration is explained and how the cluster is configured technically. By using an on-premise system in a laboratory environment, it allows us to easily monitor and evaluate essential parts of the algorithm in a live distributed system without interfering factors that may arose at a cloud provider. We decided to use Raspberry Pi's as the hardware platform because of their low expenses, they are easy to come by and have a straight-forward configuration procedure. A Raspberry Pi is a small single-board computer ideal for prototyping, learning and making low-cost distributed systems when bundled together. Furthermore the platform has become a household name in the field of IoT-computing. The Internet offers lots of guides on how to get them configured and running with Kubernetes and other kinds of applications, if that is of interest. We did a preliminary investigation of the pros and cons of various Linux operating-systems for Kubernetes and made several installation attempts on all of them. This produced some findings we have included as well in table 5.1, where different operating-system options are listed for comparison. Note that all support Kubernetes (K8S), but vary in documentation and ease of setup. Our findings here are mostly empirical learnings that can benefit future studies that want to use Kubernetes and Docker.

The most suited operating-system was not the only factor, we took into account, however. We discovered two kinds of installation media for Kubernetes, namely *K8S*, which is the original full-blown version that uses *kubeadm* and *Docker*, and a more lightweight one called *K3S*, that comes with less preinstalled packages, ideal for IoT-setups and low-cost hardware. Table 5.2 shows a comparison of the installation media:

Because of the low-cost of Raspberry Pi nodes, their widespread use in IoT-systems, the lightweight nature of *K3S* and its ease of setup, we decided to create a cluster that consists of four Raspberry Pi model 4 nodes using *Raspbian Lite* as the operating-system and K3S as the Kubernetes installation version. The complete setup procedure, configuration and hardware list can be found in appendix B on page 109.

OS	Pros	Cons
HypriotOS 32bit	Designed to run Docker natively and requires only a few configuration steps to run Kubernetes.	Documentation on HypriotOS is unfortunately limited as of this writing and the distribution has not seen widespread recognition.
Raspbian Lite 32bit	Easy to setup and is the official OS for Raspberry Pi devices. Is well documented and tested.	Not configured nor ready for a Kubernetes installation out of the box. A lot of manual work has to be put in to make it compatible with a Kubernetes installation, in our experience.
Arch Linux 32/64bit.	Smallest OS of all that we evaluated. Has the ability to be customized to suit a particular need and comes with a 64bit version as well as 32bit.	The configuration process is very time consuming. A lot of Linux specific knowledge on how to install certain packages and perform Docker network configurations is required.
Ubuntu Server 32/64bit.	Popular OS for servers. A lot of documentation exists and is easy to get started with.	Not made natively for Raspberry Pi's, although a version exists for IoT-devices.

Table 5.1: *The pros and cons of the Linux distributions we evaluated for Kubernetes use.*

Kubernetes	Pros	Cons
K8S	Official open-source Kubernetes distribution with good and elaborate documentation that is readily accessible.	Originally designed for large-scale distributed computing systems and enterprise-level servers, like the Amazon AWS or Google Cloud Engine, not for a small Raspberry Pi cluster.
K3S	A Lightweight bare-bones version of Kubernetes, made for small clusters, thus it ships with an very quick installation procedure ideal for our purposes.	Documentation is limited.

Table 5.2: *The pros and cons of the Kubernetes media we evaluated. K8S is the default installation version from official repositories meant for AMD64 architectures, where K3S is a lightweight Kubernetes installation meant for IoT-appliances like ARM64/ARMv7 architectures and other resource-constrained systems.*

5.3 Implementation details

In this section we explain some technical details about the cluster, put words on the software libraries we have used and other implementation details. Figure 5.3 is an overview of the final Kubernetes cluster and illustrates pod entities and data flow.

The following phrases are short technical notes about pod configuration and design choices. Note we use a single preprocessor and postprocessor pod in all experiments for simplicity, but their numbers can be scaled if needed for the application at hand.

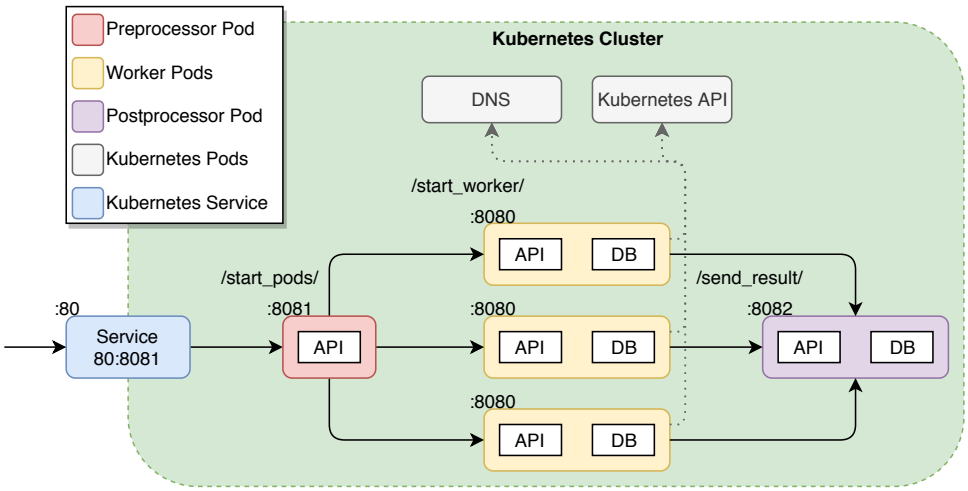

Figure 5.3: *A graphically overview of the Kubernetes cluster. For all experiments, we use this three-layered setup with a single preprocessing pod, multiple worker pods scaled accordingly and finally a single postprocessing pod. Data arrive at the service that fronts the preprocessing pod and is then distributed to worker pods. These workers run our implementation of cloud K-SVD and collaboratively reach a consensus of a final dictionary and signal matrix that best represent the data they were trained with. All pods can query the Kubernetes API for cluster information as well as a CoreDNS pod to resolve service hostnames. Some pods contain a utility container for data storage as well.*

Service pod: The service pod fronts the single preprocessing pod and maps incoming HTTP requests on port 80 from the external LAN to an internal target port 8081. All requests to the service will be forwarded accordingly to this configuration defined in an iptables list.

Preprocessing pod: Uses the Python AIOHTTP library to receive and handle asynchronous HTTP requests forwarded to it by the front service. The pod offers a single endpoint that takes initial data as a HTTP body and forwards this in parts to the worker pods participating in the algorithm. It queries the secure Kubernetes API by token authentication for pod lists. The preprocessor is data agnostic.

Worker pods: Consists of two containers: A Python application that uses AIOHTTP and AIOREDIS libraries to receive and handle asynchronous HTTP requests to start algorithmic work made by the processing pod, and a Redis database where it can store intermediate data like the residual vector for cloud K-SVD as well as computed results statistics and final result data. Redis[1] is an open-source key-value database perfect for storing non-relational data. We use the NUMPY library for all matrix and vector computations. Workers exchange information using the AIOHTTP web client that comes with the Python package and these exchanges are asynchronous

[1] `https://redis.io/`

as well. At any point in time can a worker request the residual or status of another worker, since each request including the one from the preprocessor is handled in individual threads. Like the preprocessing pod, the workers query the Kubernetes API for lists of fellow worker pods. We set specific parameters for cloud K-SVD and data communication as environment variables made possible by the Kubernetes Downward API[2].

Postprocessing pod: Provides a AIOHTTP web API to receive processed data from worker pods. If data has been separated at the start of processing, it is reassembled at this pod. Eventual data aggregation takes place here as well. Its technical details mimic those of the processing pod since it relies on AIOHTTP for web API services.

All API interfaces can be found in appendix C. The Dockerfiles for the project can be found in appendix D. The deployment configuration files can be found in appendix E. We use *Dockerhub.io* as the default repository for all Docker images and *GitHub.com* for source code. Below is a short list of software library versions.

- Node architecture: linux/ARMv7
- Node operating-system: Raspbian Lite 32bit
- Kubernetes version: v1.16.3-k3s.2
- Docker version: 2.1.0.5 (stable)
- Python version: 3.7.4 32-bit

[2] `https://kubernetes.io/docs/tasks/inject-data-application/environment-variable-expose-pod-information/`

Chapter 6

Experiments and results

This chapter details practical experiments and results of using cloud K-SVD to approximate various signals, learn a mutual dictionary in a distributed system and denoise both benchmark and medical images. With practical experiments we address the problems raised in the problem definition, see 1.3. We will benchmark cloud K-SVD's accuracy in recovery of sparse signal vectors, execution times and how well we are able to scale the algorithm to multiple worker pods (logical computing entities running on real nodes) in Kubernetes, whilst maintaining data separation at each node. Thus in a real scenario, to collaboratively archive a global dictionary to effectively reduce the residual coding error at every site, the pods must perform a number of consensus iterations to accommodate for either approximate or remote training residual errors. A set of experiments will be conducted to analyze the behavior of the cloud K-SVD algorithm, realized on a small scale IoT-system.

6.1 Introduction to experiments

In the first round of experiments, synthetic data signals are used to demonstrate efficacy of cloud K-SVD for proper dictionary learning and data representation. Such signals are generated as a sparse combination of predetermined dictionary elements from a random distribution. In the second round, we evaluate cloud K-SVD performance when training a distributed dictionary on samples from a set of ten natural images. Here, data is appropriately divided between the number of pods in a given experiment at the input level and we measure their ability to learn and approximate signal vectors based on data with common geometric structures. In the third round, we return to our base case to motivate an application of cloud K-SVD that can benefit from distributed learning collaborative between sites as we evaluate our implementation of cloud K-SVD on obtained medical image data from patients who suffer from rheumatoid arthritis. These images are produced by image systems with a lot of induced noise, so we shall test how well cloud K-SVD can built an effective distributed dictionary for denoising applications. For all experiments, we consider a network of P Kubernetes worker pods that can receive, compute and send data as a self-contained and autonomous unit operating under the supervision of a Kubernetes deployment controller. For these experiments, we scale the number of pods manually for a deployment to demonstrate efficiency of the algorithm when more pods are added to the working pool and input data is further distributed. All experiments are performed in a real cluster and that pods share the same networking media, hence network delays and potential traffic congestion will play a role in the final results.

Generally, we consider three different variants of cloud K-SVD that our experiments will focus on [79] [36]:

- Centralized K-SVD: A single worker pod $P = 1$ receives all data signals $\boldsymbol{Y} \in \mathbb{R}^{M \times Q}$ for the entire sparse approximation and dictionary learning task and does no intermediate consensus iterations among peers, since the pod has all the data itself. This model is an implementation of the traditional K-SVD scheme [11]. Upon completion, the single worker pod will have computed a global dictionary $\boldsymbol{D}$ to approximate all of the data.

- Local K-SVD: Pod P_i receive $\boldsymbol{Y}_i \in \mathbb{R}^{M \times Q_i}$ data signals and perform the dictionary learning task locally without any collaborative consensus iterations among peers. This model distributes the data signals, but pod P_i will not become familiar with pod P_j's data due to no consensus iterations. This model mimics the centralized version, but distributes the data among multiple working pods.

- Cloud K-SVD: Pod P_i receive $\boldsymbol{Y}_i \in \mathbb{R}^{M \times Q_i}$ data signals and perform dictionary learning collaboratively by first calculating the signal vector $\boldsymbol{X}_i$ via sparse approximation and then estimate the residual error via consensus iterations as shown in section 3.8. Here, all worker pods P work together by either averaging or corrective consensus to lower the residual error and properly estimate the signal vectors by a local dictionary $\boldsymbol{D}_i$ that accommodates data at node j. For all cloud K-SVD experiments, we use averaging consensus.

Note that all experiments are done with one preprocessing and one postprocessing pod. These pods are not active in any part of a experiment and are used only to split and aggregate data. The following parameters are configurable for cloud K-SVD, though some will often be fixed at start to evaluate certain performance metrics for various data inputs. For centralized K-SVD and the local version, some parameters are omitted in nature. For all experiments we note the execution time and the approximation error in the final signal. Sparse approximation (SA) and dictionary learning (DL) parameters are further detailed in section 3.5 and 3.6, respectively, whilst Kubernetes parameters are from section 4.4.

- Iterations t_d, t_p and t_c: The cloud K-SVD algorithm performs three types of iterations, that are nested from left to right: First, a number of t_d cloud iterations that compute the sparse signal vector via OMP and update the corresponding atoms in the dictionary update step. Second, a number of t_p power iterations for each atom N. Third, a number of t_c consensus iterations that fetches the residual error from neighboring pods by averaging or corrective consensus. See more in section 3.7 and 3.8.

- Atoms N: The number of atom column vectors in the dictionary $\boldsymbol{D} \in \mathbb{R}^{M \times N}$ and the length of the signal row vectors $\boldsymbol{X} \in \mathbb{R}^{N \times Q}$. As we evaluate overcomplete dictionaries in the experiments, it follows that $N \gg M$. The initial dictionary at pods is usually initialized with a fixed number atoms N and dimension M with randomly generated data.

- Signal size M: The length of the data signals in $\boldsymbol{Y} \in \mathbb{R}^{M \times Q}$ and atoms in the dictionary $\boldsymbol{D}$.

- Data signals Q: The number of signal column vectors in $\boldsymbol{Y}$ and $\boldsymbol{X}$.

- Sparsity level K: Number of non-zero coefficients in the signal vectors of $\boldsymbol{X}$.

- Pod weight W: A scalar that is multiplied by the difference between residual vectors q_i and q_j in the consensus step. It defines how much empathize is put on vectors q_j at node i in range $[0, 1]$.

- Number of pods P: Number of worker pod replicas in Kubernetes that run cloud K-SVD in a single experiment.

To evaluate the error in the approximations and the error in image recovery we use a mix of metrics, specially *MSE*, ℓ_2-*norm*, *PNSR* and *SSIM*, all of which are recognized as reliable instruments for error estimation in sparse approximation and image processing, see appendix A.

6.2 Experiments using synthetic data

Purpose

The first set of experiments run cloud *K-SVD* with synthetic data. This kind of data works well for behavior testing and early evaluation of our implementation as it is simple to generate in considerable amounts and is deterministic in a sense that can we generate the same data set over and over again for every experiment. We have the following objectives in view:

- Evaluate the correctness and convergence behavior of the distributed power method component in cloud *K-SVD* as a function of the number of t_d, t_p and t_c iterations using synthetic distributed data. Recall that t_d is the number of overall cloud *K-SVD* iterations, t_p is the number of power iterations and t_c is the number of consensus iterations. See 3.8 for a recap.
- Establish if the disparity between local dictionaries $\boldsymbol{D}_i$ and $\boldsymbol{D}_j$ is reduced as a function of the number of cloud *K-SVD* iterations (t_d, t_p and t_c).
- Measure cloud *K-SVD* ability to cope with large amounts of data signals Q in terms of accuracy in recovery and execution times.
- Compare the behavior of cloud *K-SVD* as a distributed algorithm with variants such as the centralized *K-SVD* for $P = 1$ and the local *K-SVD* for $P > 1$ and $t_c = 0 \wedge t_p = 0$.

Data and setup

The pods will have a preallocated set of resources available (a Kubernetes pod limit) and a upper bound set of the maximum amount of resources they can acquire (a Kubernetes request). All pods will have the same number of resources available (CPU and memory). This is done to avoid pod starvation, where a working process in a pod is constantly denied necessary resources to process its work because they have been reserved by another. By setting a resource and request limit we explicitly tell the Kubernetes controller to reverse the necessary number of resources before allocating a pod to a node.

We consider two kinds to test for the synthetic data: A consensus test (1) to evaluate convergence behavior of cloud K-SVD and a data test (2) to evaluate performance in approximation and recovery in terms of executions and error. This way, (1) mainly tests the collaborative part and compares the similarity in resulting $\boldsymbol{D}$ dictionaries, whereas (2) fixes the number of consensus and power iteration to focus mainly on overall approximation error and execution times. For both tests, we use a built-in Python method to generate training data and split it evenly at the preprocessing pod so that every worker pod P obtains an equally sized pool of data samples. Training data has been generated as sparse linear combinations of sparsity K from a random $\boldsymbol{D}_i$, of size $M \times N$ with atoms that are i.i.d. uniformly distributed entries with normalized ℓ_2 columns, in the range $[0, 1]$. Hence training data for P_i has been generated using $\boldsymbol{D}_i$ and $\boldsymbol{X}_i$ drawn from a random distribution. All dictionary columns have been normalized to ℓ_2-norm. No additional noise is added to the data signals at this stage. Sparse approximation is done using our own implementation of SOMP cross-compiled to C-code, from Python, for performance reasons

since the number of samples Q is large for both tests. For the consensus iterations, we assign a uniform weight w of some numeric value in the range $[0, 1]$ to each pod.

For the consensus tests (1), the following configuration is used:

- A training data dimension $M = 20$ and quantity set to $Q = 2000$
- Number of dictionary atoms is $N = 50$
- Sparsity is set to $K = 3$
- Number of iterations is $t_d = 50$.
- Pod quantity is set to $P = 4$.
- Number of collaborative iterations is set to $t_p = 2, 3, 4, 5$ and $t_c = 1, 5, 10$.

For the data tests (2), the following configuration is used:

- A training data dimension $M = 20$ and quantity set to $Q = 200, 800, 2000, 8000, 20000$.
- Number of dictionary atoms is $N = 50$
- Sparsity is set to $K = 3$
- Number of iterations is $t_d = 50$.
- Pod quantity is set to $P = 1, 4, 8, 16$.
- Number of collaborative iterations is set to $t_p = 3$ and $t_c = 5$.

Expectations

We expect the following observations to be expressed in the experiments:

- For cloud K-SVD, the MSE between dictionaries $\boldsymbol{D}_i$ and $\boldsymbol{D}_j$ should decrease as a function of the number of collaborative iterations t_d, t_p and t_c. This would indicate convergence to a mutual global state $\boldsymbol{z}$. Recall however that such iterations are network-based and require pods to actively communicate over the wire, so increasing these should also mean longer execution times for the whole algorithm.

- For centralized K-SVD, we expect a single pod to have slower execution times for the sparse approximation stage since it has to process all data signals Q at every iteration, but be faster at the dictionary update step because it can iterate and change its dictionary freely without having to obtain a mutual consensus first.

- For local K-SVD, we expect pods to become well-acquainted with data Q_i and efficiently learn a dictionary that can approximate these signals, however since we do no collaborative iterations here, we do not expect the pods to become familiar with data Q_j. This can be expressed by a large MSE between dictionaries $\boldsymbol{D}_i$ and $\boldsymbol{D}_j$.

Results

We start with test results for consensus tests (1). Figures 6.1 and 6.2 show the average ℓ_2-norm error and MSE, respectively, between dictionaries $\boldsymbol{D}_i \; \forall i = 1, 2, \ldots, P = 4$ as a function of the number of t_d iterations. Here, we fix the parameter t_c for all experiments and notice the change in error values when we increase t_p. For both metrics, a lack of convergence is observed for $t_c = 1$ no matter the number of t_p. When we set t_c higher, we get a much clear picture of the error between the dictionaries as they drop collectively towards zero on a logarithmic scale.

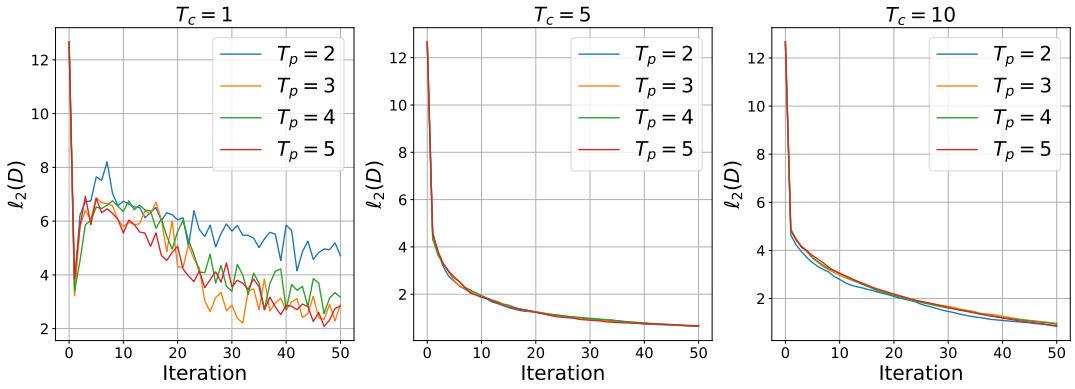

Figure 6.1: *The average ℓ_2-norm of the error between dictionaries of each pod at different t_p and t_c. $t_d = 0, 1, \ldots, 50$, $M = 20$, $N = 50$, $Q = 2000$, $K = 3$, $P = 4$.*

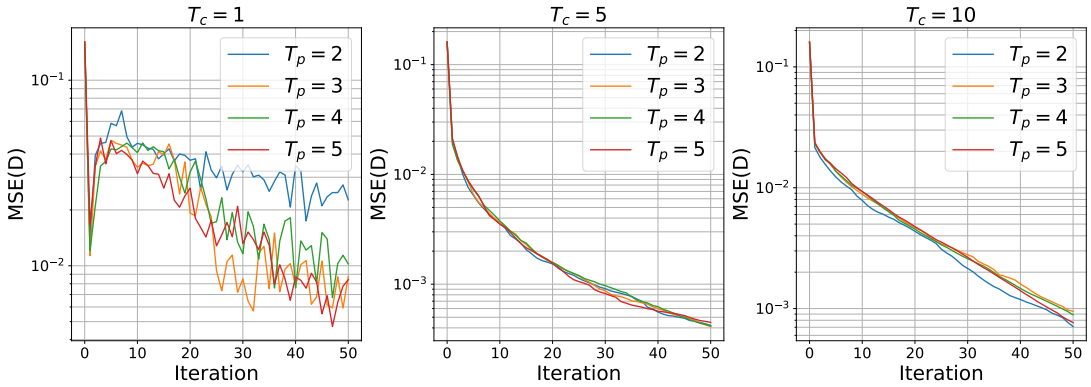

Figure 6.2: *The average MSE between dictionaries of each pod at different t_p and t_c. $t_d = 0, 1, \ldots, 50$, $M = 20$, $N = 50$, $Q = 2000$, $K = 3$, $P = 4$.*

Figures 6.3, 6.4 and 6.5 show the average ℓ_2-norm error, MSE and PSNR, respectively, between the original $\boldsymbol{Y}$ and the approximation $\hat{\boldsymbol{Y}}$ as a function of t_d iterations. Again, we see some turbulence in the error for $t_c =$ and no matter the number of t_p, however all experiments for $t_c = 1$ drops in error as the number of all iterations t_d goes up, accordingly. We attribute the unsteady behavior to a lack of consensus iterations for $t_c = 1$ in the plot to the left. Setting a higher t_c permits pods to properly propagate their errors to all peers, thus converging at a smaller error for $\hat{\boldsymbol{Y}}$. Also observe that we in fact get better results in terms of $\boldsymbol{Y}$ error for $t_c = 1$, which is naturally since the pods then solely approximate their own data, like local K-SVD, and do not take into account any data at other pods. This means that piece-by-piece $\boldsymbol{Y}$ is well represented, but the dictionary $\boldsymbol{D}$ lacks a common denominator.

Table 6.1 show a list of average execution times for $P = 4$ pods using synthetic data tracked for the collaborative K-SVD step. The purpose is to show how consensus and power iterations impact the time it takes to complete the K-SVD step on average in the algorithm measured in seconds. The signal approximation step, OMP, is not included here because it does not depend on the number of consensus or power iterations, thus remain unchanged throughout all experiments here. Timings here accompany the experiments we just showed in the plots above and numbers are rounded to nearest single decimal. The trend in 6.1 is clear: The more consensus iterations we do per power iteration, the longer the algorithm takes to complete the K-SVD step.

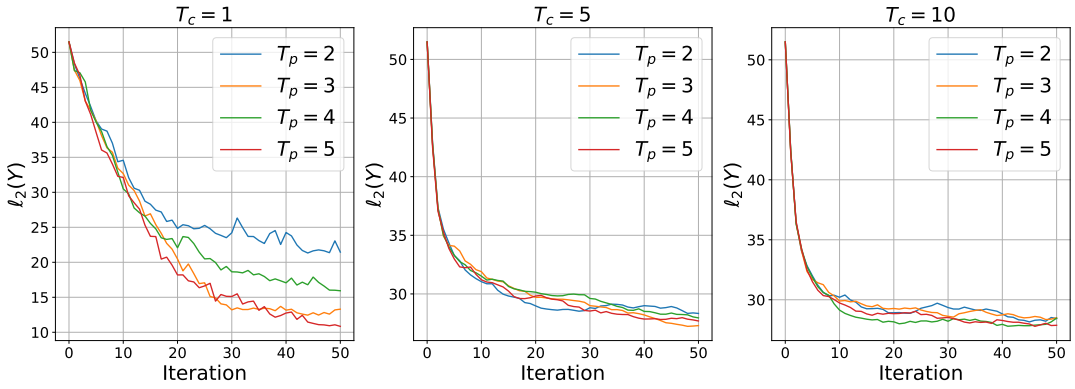

Figure 6.3: *The ℓ_2-norm of the error between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$ at different t_p and t_c. $t_d = 0, 1, \ldots, 50$, $M = 20$, $N = 50$, $Q = 2000$, $K = 3$, $P = 4$.*

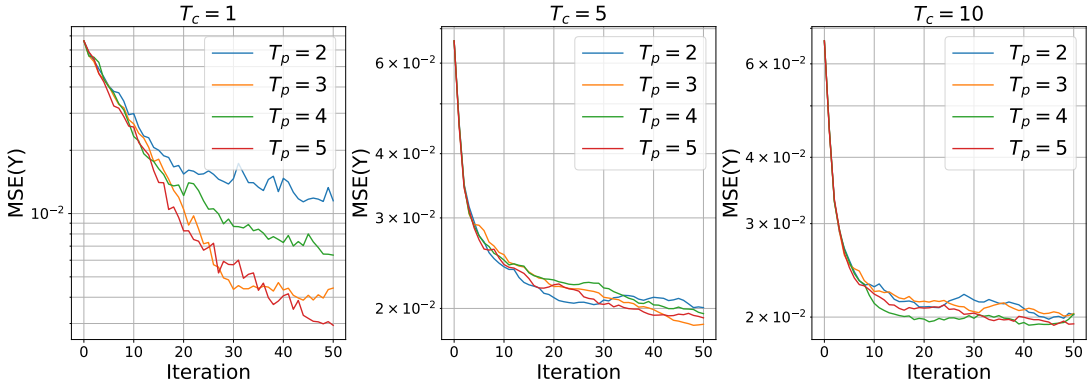

Figure 6.4: *The MSE between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$ at different t_p and t_c. $t_d = 0, 1, \ldots, 50$, $M = 20$, $N = 50$, $Q = 2000$, $K = 3$, $P = 4$.*

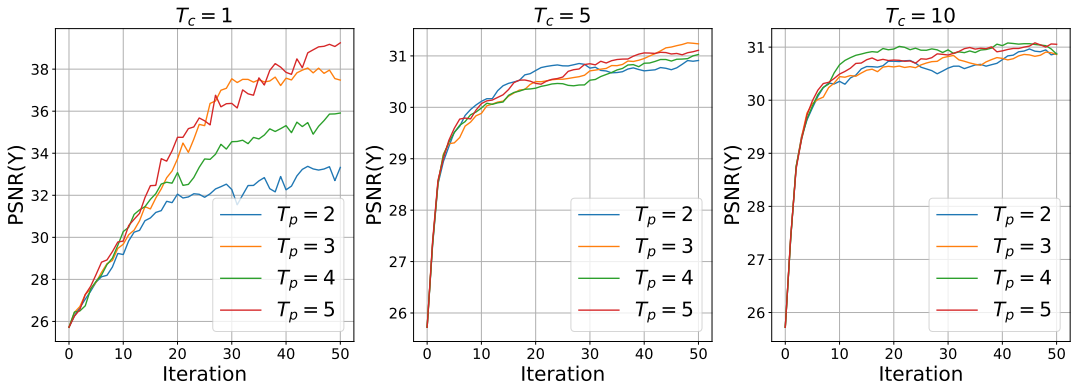

Figure 6.5: *The SSIM between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$ at different t_p and t_c. $t_d = 0, 1, \ldots, 50$, $M = 20$, $N = 50$, $Q = 2000$, $K = 3$, $P = 4$.*

t_c	t_p			
	2	**3**	**4**	**5**
1	$4.5s$	$6.6s$	$8.3s$	$9.8s$
5	$18.4s$	$26.9s$	$35.2s$	$43.1s$
10	$34.5s$	$51.2s$	$67.9s$	$84.2s$

Table 6.1: *The average time of a K-SVD iteration with different t_p and t_c. $t_d = 50$, $M = 20$, $N = 50$, $Q = 2000$, $K = 3$, $P = 4$.*

We now turn to the data tests (2) and start with a look at results for the centralized K-SVD case, where we change the number of training signals Q for each run and observe the error as a function of the number of iterations t_d. Figure 6.6 show error scores between $\hat{\boldsymbol{Y}}$ and $\boldsymbol{Y}$ for centralized K-SVD when the number of signals Q increase and a fixed dictionary size N. Obviously, the error depends on the number of training signals as it is easier to represent a few signals than many with a dictionary of the same size. We had to halt the experiments for $Q = 20000$ (the purple line) at a iteration count $t_d = 20$ because of memory and time constraints, however the error curve stabilizes around the 20 mark.

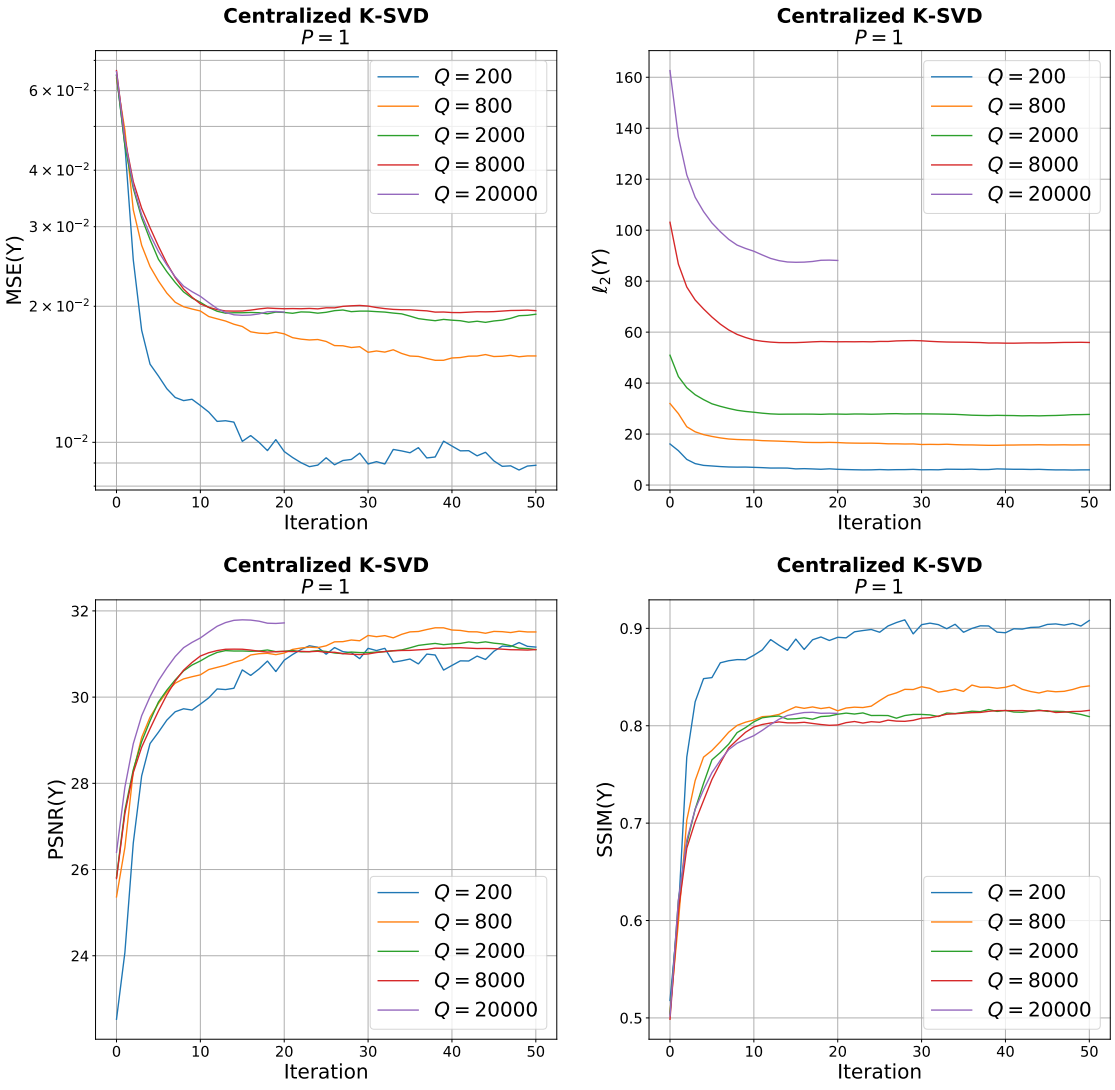

Figure 6.6: *The MSE, ℓ_2-norm of the error, PSNR and SSIM between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$ at different Q. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$, $P = 1$.*

The following tests use a local K-SVD configuration for data tests (2). This means $P > 1$ and data is distributed to all pods but not trained collaboratively. Figures 6.7 and 6.8 show average MSE scores between dictionaries $\boldsymbol{D}_i$ $\forall i = 1, 2, \ldots, P$, and MSE scores for the reconstruction $\hat{\boldsymbol{Y}}$ compared to the original. We set the number of iterations t_d to 50 for all runs. Even with a high number of iterations t_d, the error between dictionaries is stationary due to a lack of consensus between pods and they actually diverge somewhat when the number of signals is low and the

pod count is high. For $\boldsymbol{Y}$, the reconstruction error is significant for a low number of signals and deteriorates slightly when that parameter goes up.

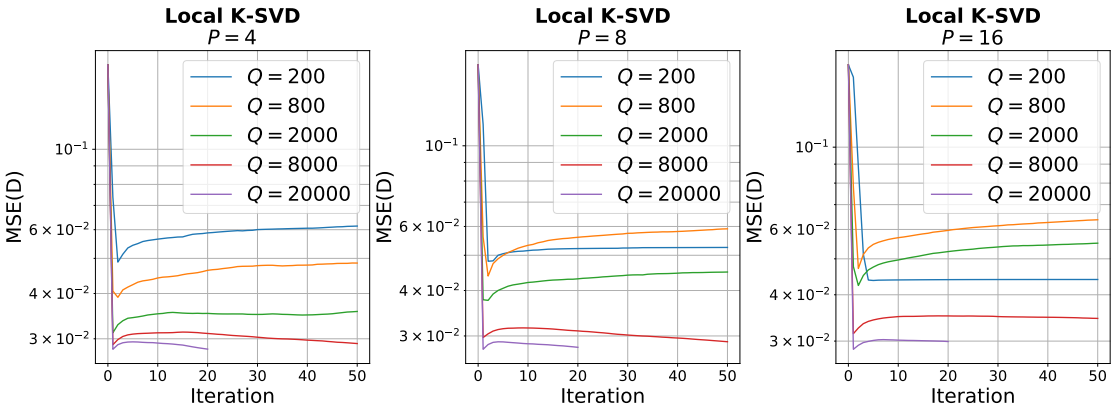

Figure 6.7: *The average MSE between dictionaries of each pod at different Q and P. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$.*

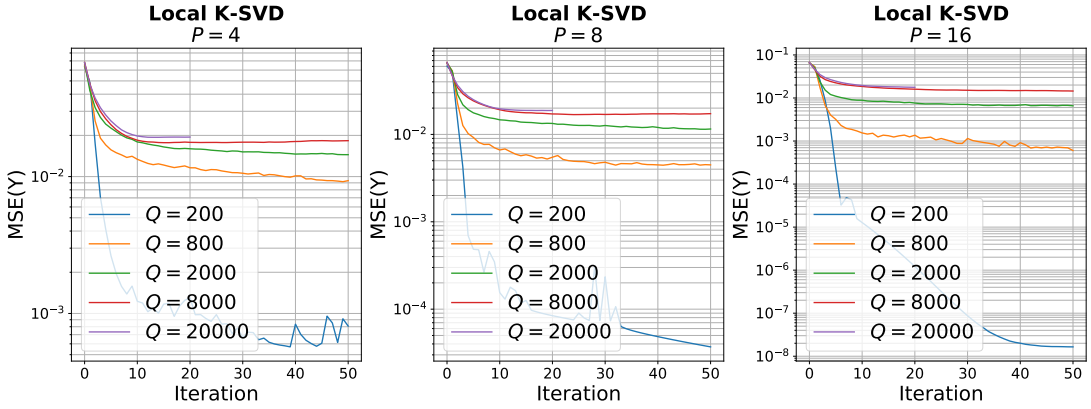

Figure 6.8: *The MSE between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$ at different Q and P. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$.*

Figure 6.9 is merely the SSIM for the reconstruction of $\boldsymbol{Y}$. Recall that a score of 1 is perfect reconstruction and anything below 0.5 is poor. As expected, it is easier to reconstruct few signals than many, for example $Q = 200$ versus $Q = 8000$, although the SSIM is certainly begin improved upon as a function of the number of overall iterations.

Next we use a cloud K-SVD configuration for data tests (2), in other words true cloud K-SVD with $t_p = 2$ and $t_c = 5$ set. So here the number of pods is $P > 1$, data is distributed and trained in a collaborative fashion. Again, we show the MSE metric for $\boldsymbol{D}$ and $\boldsymbol{Y}$ as in the local K-SVD experiments for comparison. Figures 6.10 and 6.11 show average MSE scores between dictionaries and MSE for the reconstruction $\hat{\boldsymbol{Y}}$. This time the pods can exchange residuals, so for the MSE between dictionaries we see vastly improved scores now in cloud K-SVD compared to traditional local K-SVD. Again we cut off the purple line, $Q = 20000$, at iteration $t_d = 20$ because of memory and time constraints. For $\hat{\boldsymbol{Y}}$, a large number of signals is naturally harder to approximate than a small number, hence the MSE is better at for example $Q = 800$, the orange line, than at $Q = 8000$, the red line. MSE for $\hat{\boldsymbol{Y}}$ drops as the number of pods increase, simply because data is more distributed when more pods participate.

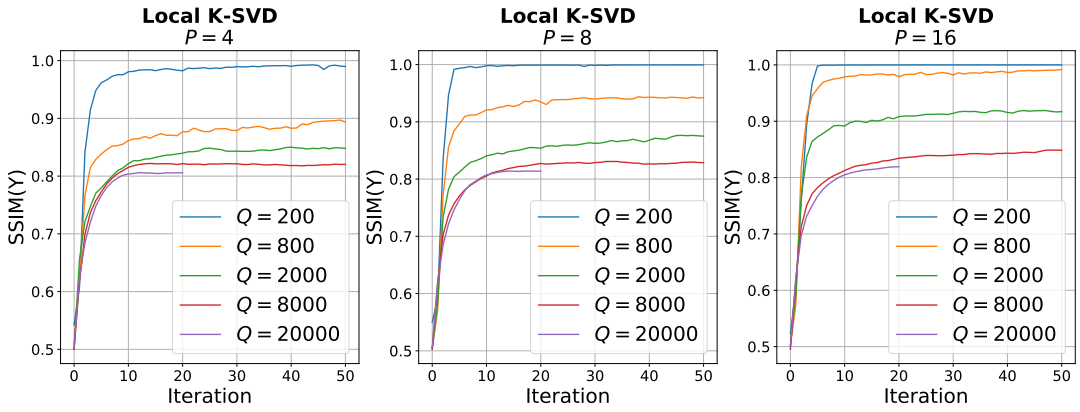

Figure 6.9: *The SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$ *at different* Q *and* P. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$.

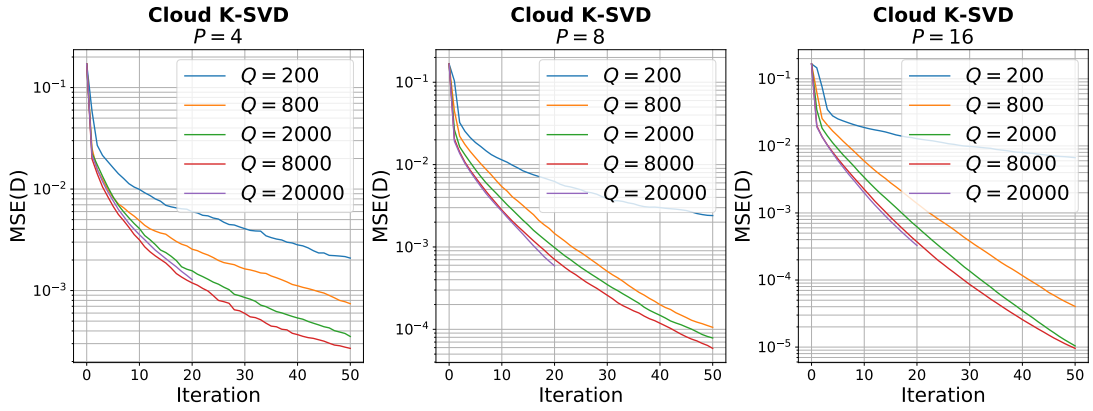

Figure 6.10: *The average MSE between dictionaries of each pod at different* Q *and* P. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$.

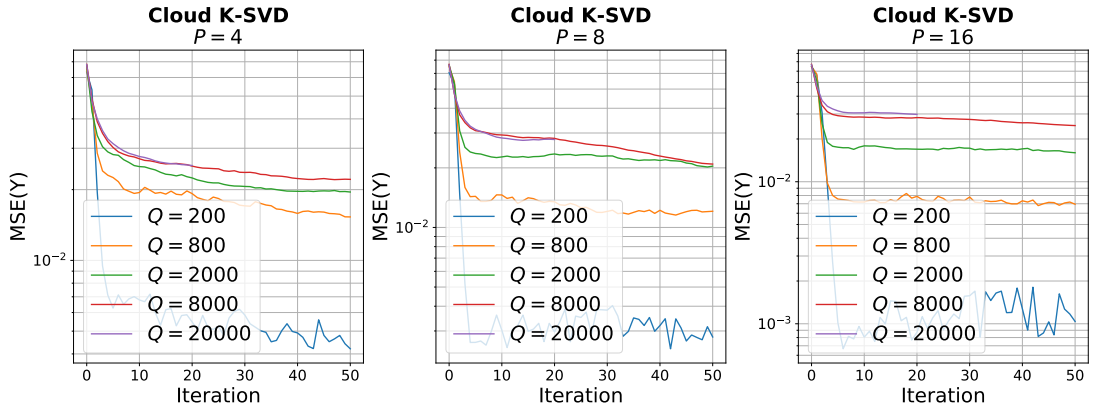

Figure 6.11: *The MSE between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$ *at different* Q *and* P. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$.

Figure 6.12 is the SSIM for the reconstruction of $\boldsymbol{Y}$. Recall that a score of 1 is perfect reconstruction and anything below 0.5 is poor. As expected, it is harder to reconstruct the signals when consensus is enabled (cloud K-SVD) compared to disabled (local K-SVD), as compared to Figure 6.9.

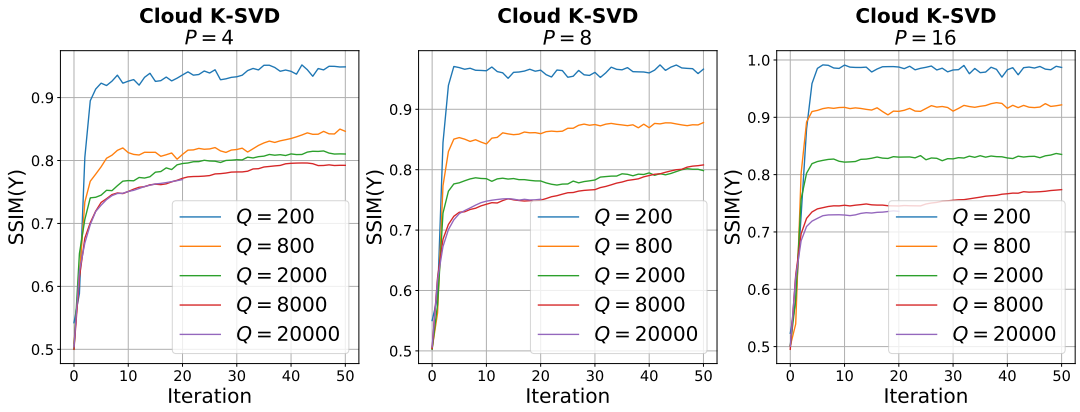

Figure 6.12: *The SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$ *at different* Q *and* P. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$.

Figures 6.13, 6.14 and 6.15 compare average MSE scores between dictionaries, now local versus cloud K-SVD, when we change the number of participating pods so that $P = 4, 8, 16$. We see a clear improvement in MSE when we allow the pods to communicate and exchange residual information to make a common dictionary (plots to the right), compared to traditional K-SVD with distributed data (plots to the left). For cloud K-SVD with $P = 8$ and $P = 16$, with signals $Q = 200$, the blue line stagnates after the 20th iteration which we believe is the result of a lack of training signals. With only a few samples, it is harder to approximate a function than with many, so the approximation is inherently more heterogeneous with fewer samples than with many.

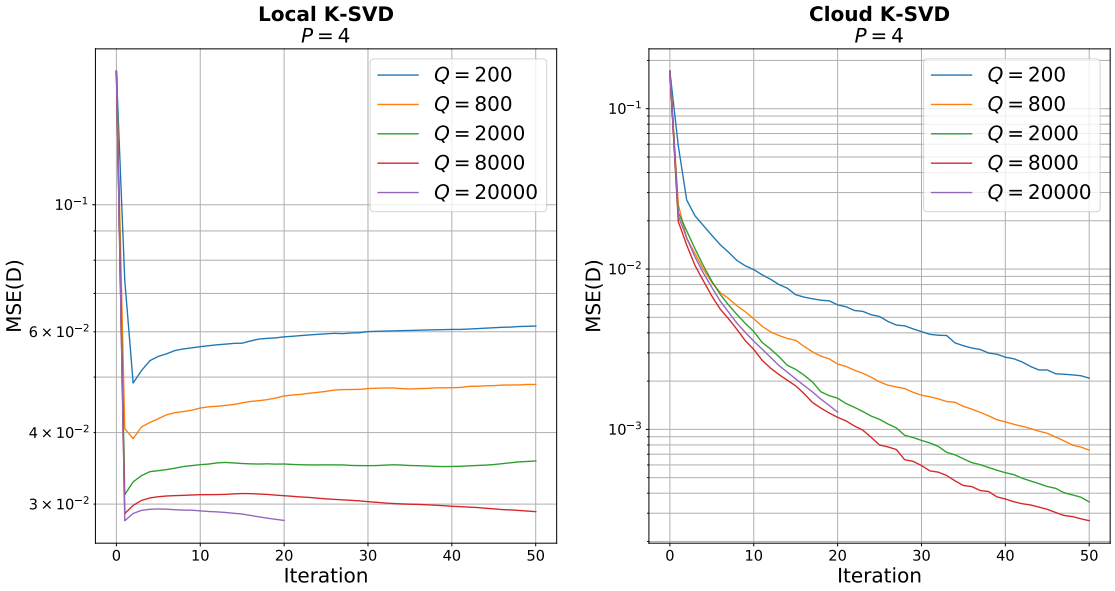

Figure 6.13: *Comparison of the average MSE between dictionaries of each pod at different* Q *using local and cloud K-SVD.* $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$, $P = 4$.

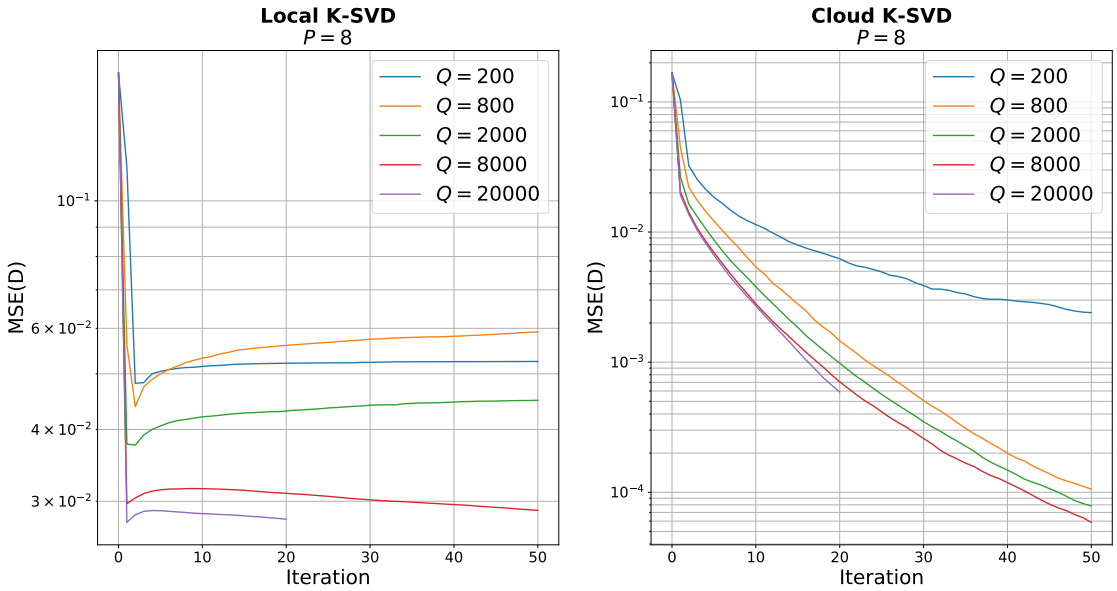

Figure 6.14: *Comparison of the average MSE between dictionaries of each pod at different Q using local and cloud K-SVD. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$, $P = 8$.*

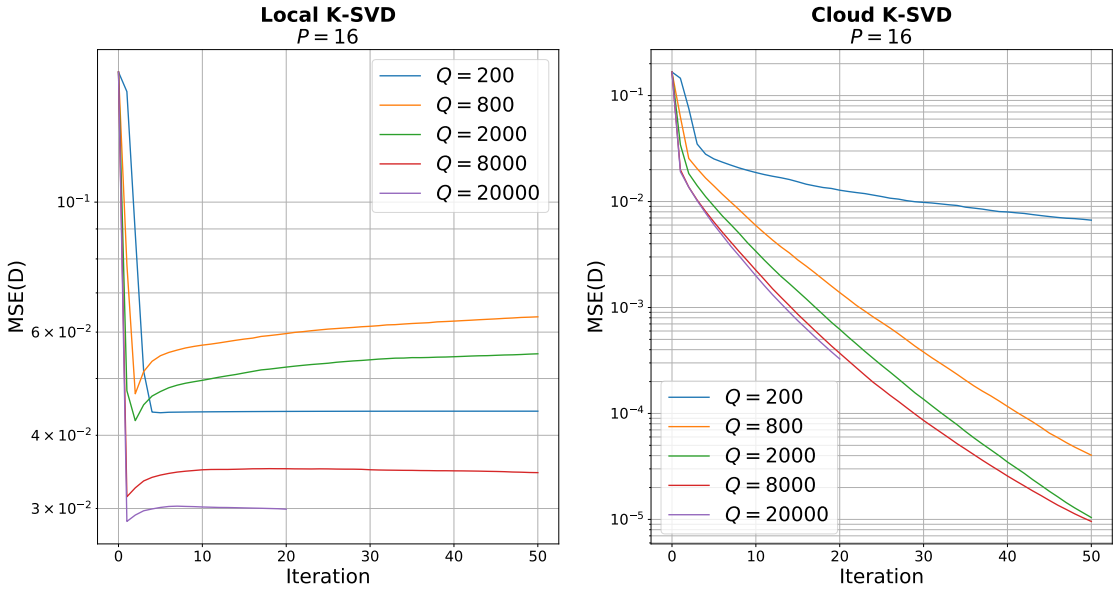

Figure 6.15: *Comparison of the average MSE between dictionaries of each pod at different Q using local and cloud K-SVD. $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$, $P = 16$.*

Figure 6.16 show PSNR scores between local and cloud K-SVD for the reconstruction of $\boldsymbol{Y}$ with just $P = 4$. We see that both K-SVD's reduce the error in $\boldsymbol{Y}$ by an equal amount for $Q > 200$, it just happens a lot faster for local (some $t_d = 10$) than cloud (some $t_d = 40$). This trend can be explained as cloud K-SVD has to take into account every other signal at each iteration, which steers the coefficients in the signal matrix $\boldsymbol{X}$ and atoms in the dictionary $\boldsymbol{D}$ away from the ideal values that correspond to its own training data. Thus with cloud K-SVD we obtain

a more common dictionary and signal matrix, but the price is plainly more iterations hence longer execution times if we want to get the same PSNR numbers in cloud K-SVD as in local K-SVD.

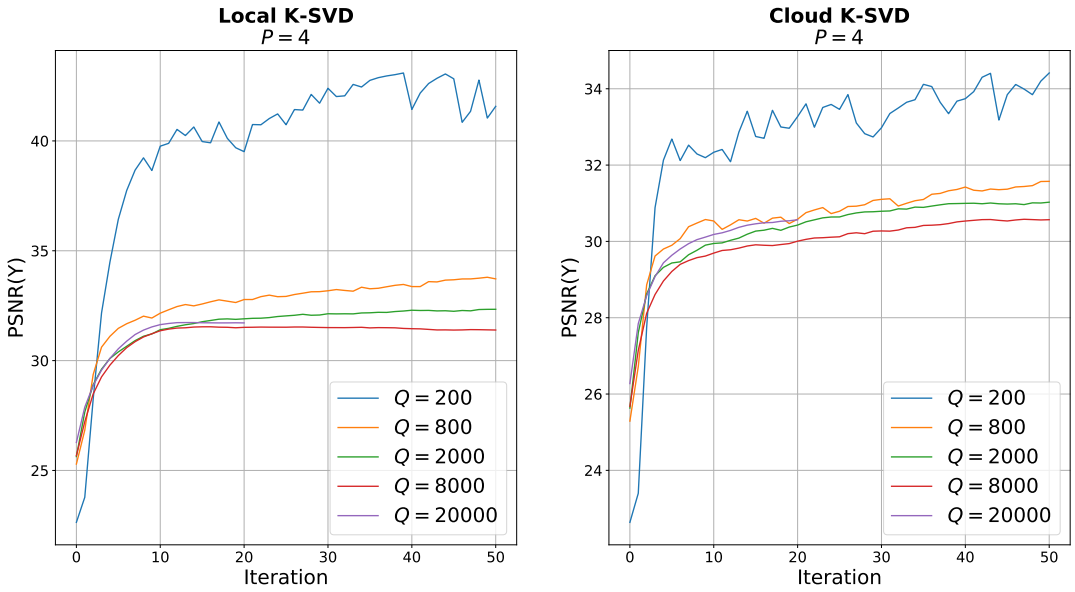

Figure 6.16: *Comparison of the PSNR between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$ *at different* Q *using local and cloud K-SVD.* $t_d = 0, 1, \ldots, 50$, $t_p = 3$, $t_c = 5$, $M = 20$, $N = 50$, $K = 3$, $P = 4$.

Table 6.2 shows average execution times for centralized, local and cloud K-SVD configurations when we change the number of training signals Q and pods P that participate in the test and measure the time it takes to complete the OMP and dictionary update step, respectively. We only show parts of the data here to demonstrate the trend and consequences more data have for the various configurations. As always, the complete data set is found in the result directory[1]. In table 6.2 we see that the time it takes to complete an OMP step is related to Q (a high Q means longer OMP) and K-SVD to consensus iterations and P (more pods means lengthy dictionary update steps). Between local and cloud, it obviously takes longer to update the dictionary in the latter because we have to complete power and consensus iterations.

P	Q				
	200	**800**	**2000**	**8000**	**20000**
Centralized $P = 1$	OMP: 0.4s, K-SVD: 0.1s	OMP: 1.5s, K-SVD: 0.2s	OMP: 3.7s, K-SVD: 0.5s	OMP: 14.7s, K-SVD: 2.3s	OMP: 36.8s, K-SVD: 6.9s
Local $P = 4$	OMP: 0.1s, K-SVD: 0.1s	OMP: 0.4s, K-SVD: 0.1s	OMP: 0.9s, K-SVD: 0.2s	OMP: 3.7s, K-SVD: 0.5s	OMP: 9.2s, K-SVD: 1.6s
Cloud $P = 4$	OMP: 0.2s, K-SVD: 24.8s	OMP: 0.7s, K-SVD: 25.8s	OMP: 1.6s, K-SVD: 27.0s	OMP: 5.9s, K-SVD: 28.9s	OMP: 12.6s, K-SVD: 32s

Table 6.2: *The average time of a OMP and K-SVD iteration with different data size* Q *and pod quantity* P.

[1] See results/syntheticData/averageTimes.json

Discussion

The purpose of these experiments is to get an overall evaluation of the practical usage of the centralized, local and cloud K-SVD. We start by testing the consensus part of cloud K-SVD (1) by looking at the changes made to the dictionary and signal approximations with different power t_p and consensus t_c iterations. As expected, when we increase t_c the pods reach consensus faster, and the average error between the dictionaries of the pods becomes lower, as seen in figures 6.1 and 6.2. However we did not expect the fact that the power iterations t_p had a lesser effect on the overall consensus. Looking at the synthetic data reconstruction, as seen in figures 6.3, 6.4 and 6.5, we find that by increasing t_c we are not as efficient at reconstructing the original signal $\boldsymbol{Y}$ as we are with fewer t_c, but increasing t_p has an overall good effect on the reconstruction. We believe that the reason behind t_c having that effect on the signal is because of the dictionary $\boldsymbol{D}_i$ of each pod tends towards each other making those dictionaries into a global one and thereby creating a mutual dictionary rather than one specialized to a certain pod; $\boldsymbol{D}_i \approx \boldsymbol{D}_j$.

Examining the average execution times of the K-SVD iteration, see table 6.1, there is no surprise. The greater number of t_p and t_c iterations, the longer it takes for an iteration of K-SVD to complete.

In the next tests (2) we observe the centralized, local and cloud K-SVD with different amount of data Q and pods P. We start by observing the centralized K-SVD and how it performs when reconstructing data of different sizes. We observe that by increasing the amount of data, the harder it is to reconstruct it all, as shown in figure 6.6, but that the overall error settles with 2000 and above at $MSE(\boldsymbol{Y} - \hat{\boldsymbol{Y}}) = 2 \cdot 10^{-2}$ and $SSIM(\boldsymbol{Y} - \hat{\boldsymbol{Y}}) = 0.8$, which is decent and expected on synthetic data. Since this was a test on centralized K-SVD, it only has one dictionary because there only is a single pod. We then start by comparing centralized K-SVD results to the local K-SVD ones. The local K-SVD does not have a consensus step, which means that the pods do not reach consensus of a dictionary and $\boldsymbol{D}_i \neq \boldsymbol{D}_j$, as shown in figure 6.7. However the signal reconstruction is an entire different matter. Since each pod only got a fraction of the data and need not reach consensus among peers, they can specialize in their part of the data and thereby do a better reconstruction job than centralized K-SVD, see figure 6.9. Going from local K-SVD to cloud K-SVD, we see a big difference in the consensus between pods. The average MSE between dictionaries of pods drop, and the more pods, the lower average MSE, as seen in figure 6.10. This unfortunately also impacts the error of our reconstructed signal in the sense that it starts to look like centralized K-SVD, see figure 6.12. This is however expected, since cloud K-SVD should be reminiscent of the centralized K-SVD in its results.

As a last part of these experiments, we observe the average execution times of OMP and K-SVD for the centralized K-SVD with one pod, and a total of four pods for both local and cloud K-SVD, see table 6.2. By increasing the amount of training data, the OMP iteration step is significantly impacted, while opposite, the K-SVD step is slowed-down by whether the consensus step is enabled or not.

6.3 Experiments using image patches

Purpose

The second set of experiments evaluates how well cloud K-SVD can learn and encode geometric structures represented in natural images divided into patches and how well it can reconstruct these by a sparse approximation and a dictionary that has been trained on the same structures. The act of patch learning is an established method to measure adaptiveness and later accuracy in recovery for both local and distributed dictionary models and is often used in denoising scenarios [64] [15] [61]. We have the following objectives in view:

- Demonstrate and measure cloud *K-SVD*'s ability to learn a new dictionary from completely uncharted image patches in terms of learning time and how effectively such learned dictionary (or dictionaries at multiple pods) can be used to jointly approximate the original image by a reconstruction process.
- Compare the performance of cloud *K-SVD* on noiseless (1) and noisy data (2) in terms of accuracy and execution times as a function of iterations t_d, patch size M, the number of patches Q and the atom count N in the dictionary.
- Establish if the patch recovery procedure will be positively or negatively effected by the number of collaborative power t_p and consensus iterations t_c in terms of accuracy and execution times.

Figure 6.17 is a simple model of our setup for the denoising case (2) in Kubernetes. It shows how $\boldsymbol{Y}$ is split and later recovered in a noiseless version with $P = 3$.

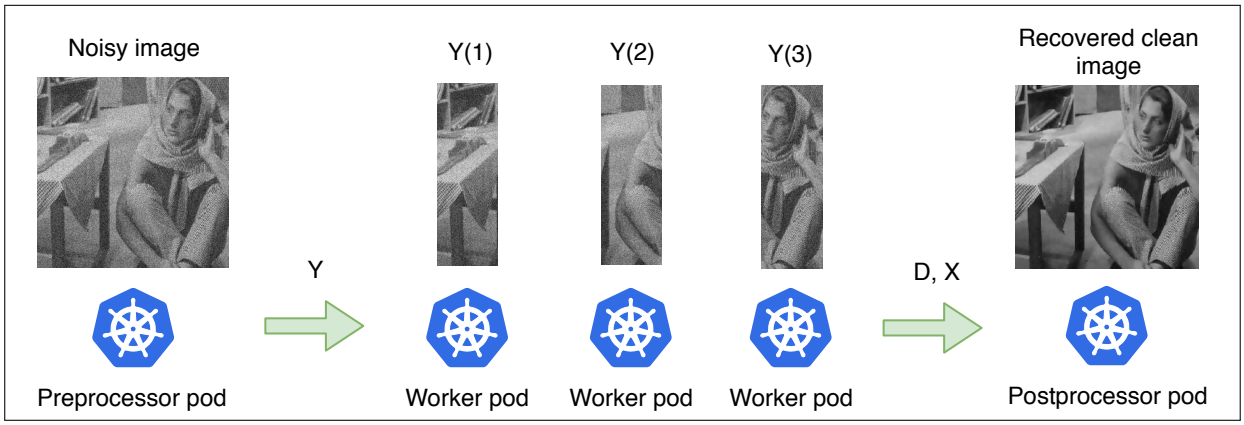

Figure 6.17: *How we do denoising of natural images. The preprocessor splits data in three equal sized pieces for $P = 3$, the worker pods processes their respective part and the postprocessor puts it all back together again.*

Data and setup

We consider two kinds of test for patch learning: A noiseless training data case (1) and a noisy one (2). The first case is designed to focus solely on the learning and coding process of clean patches without possible implications from distortions reductions in image quality. The second case features training patches contaminated with Additive White Gaussian Noise (AWGN) commonly found in natural images and demonstrates the algorithm's ability to effectively remove distortions and unwanted artifacts [82]. Both cases are done using a non-calibrated natural image dataset designed for image processing tasks. The set consists of eight gray-scaled 8-bit images chosen with respect to their dimensionality and variety in contours. Figure 6.18 shows thumbnails of all

training images. Table 6.3 shows properties such as the original resolution, the resolution when it has been downscaled by a factor $\alpha = 2$, image mean and variance.

Figure 6.18: *Thumbnails of the eight training images used in the patch experiments.*

Name	**Raw resolution**	**Resolution/$\alpha = 2$**	**Mean values**	**Variance values**
Castle	304×200	152×100	115	11.18
Lenna	480×512	240×256	124	8.96
China	427×640	214×320	114	26.95
Flower	427×640	214×320	68	9.97
Chelsea	300×451	150×226	117	4.05
Camera	512×512	256×256	118	15.07
Astronaut	512×512	256×256	112	22.27
Face	150×150	75×75	139	15.52

Table 6.3: *Test images for patch learning with their respective attributes. Downscaling by a decimation factor α creates blocks of $\alpha \times \alpha$ size and reduces elements in each block to a local mean. Mean values are in the range $[0, 255]$ and show the average intensity contribution of individual pixels. The variance indicates how much each pixel varies from its neighboring pixel.*

The images are relatively high in resolution, so for the sake of performance we apply a decimation factor α in the range $[1 - 5]$ to some images before processing. An α of 1 means no downscaling takes place, whereas a α of 2 means that the total number of pixels have been cut in half, column and row wise, compared to the original resolution. This reduces memory consumption and lowers execution times drastically allowing us to perform more experiments. For patch learning, we introduce a new variable PS that denotes the resolution of extracted patches, for example 5×5 or 8×8. For both cases, all possible overlapped patches of sizes PS are extracted from each training image, counting for example $Q = (256 - 8 + 1)^2 = 62001$ patches in total for a patch size of $PS = 8 \times 8$, resolution 256×256 and $M = 64$, see definition 4. The total amount of training data is then $Total = M \times Q$. Patch data will be used for training the distributed dictionary as in [36]. The dimension of an input data sample is then M (vertically stacked columns) for patch size PS and the total amount of training data is then $T_t = M \times Q$. The resulting data vectors are gathered in a single matrix $Y \in \mathbb{R}^{M \times Q}$ before being divided in equally-sized parts and distributed to P pods for the proposed algorithm case. For (2) we add a layer of Gaussian distributed noise by a mean μ and variance σ^2 to all training data in $\boldsymbol{Y}$ at the preprocessing stage. Worker pods are then responsible for making a sparse approximation that reduces the amount of noise in the resulting $\hat{\boldsymbol{Y}}$ data matrix.

Definition 4 *Given image size = (x, y) and patch size = (a, b) of scalar values, the number of patches Q is:*

$$\begin{aligned} &Q = (x - a + 1) \times (y - b + 1) \\ &\Downarrow x = y \wedge a = b \\ &Q = (x - a + 1)^2 \end{aligned} \tag{6.1}$$

For each experiment, a random $\boldsymbol{D}_i$ of size $M \times N$ is generated for each pod with i.i.d. uniformly distributed entries with normalized ℓ_2 columns in the range $[0, 1]$, as was the case in 6.2. Each dictionary is distributed to pods P_i before start. Then all worker pods in the network is in charge of their own local dictionary, which they will evolve as the training process commences. Again, we us our own implementation of SOMP cross-compiled to C-code from Python and assign the weight scalar w values in the range $[0, 1]$.

Generally $N \gg M$, so for both tests the following holds unless something else is stated:

- $(M = 25) \implies (PS = 5 \times 5) \implies (N = 50)$.
- $(M = 36) \implies (PS = 6 \times 6) \implies (N = 100)$.
- $(M = 49) \implies (PS = 7 \times 7) \implies (N = 100)$.
- $(M = 64) \implies (PS = 8 \times 8) \implies (N = 150)$.
- $(M = 81) \implies (PS = 9 \times 9) \implies (N = 200)$.

For the noiseless tests (1), the following configuration is used:

- The patch size is set to $PS = 5 \times 5, 6 \times 6, 7 \times 7, 8 \times 8, 9 \times 9$.
- Training data dimension is $M = 25, 36, 49, 64, 81$.
- Number of dictionary atoms is set to $N = 50, 100, 150, 200$.
- Sparsity is set to $K = 3, 5, 7$.
- Number of iterations is $t_d = 10$.
- Pod quantity is set to $P = 4$.
- Number of collaborative iterations is set to $t_p = 3$ and $t_c = 5$.

For the noisy tests (2), the following configuration is used:

- The patch size is set to $PS = 5 \times 5, 7 \times 7, 9 \times 9$.
- Training data dimension is $M = 20, 36, 49, 64, 81$.
- Number of dictionary atoms is set to $N = 50, 100, 150, 200$.
- Sparsity is set to $K = 3, 5, 7$.
- Number of iterations is $t_d = 10$.
- Pod quantity is set to $P = 4$.
- Number of collaborative iterations is set to $t_p = 3$ and $t_c = 5$.
- The noise mean is $\mu = 1$ and variance $\sigma^2 = 0.001, 0.005, 0.01$.

Expectations

We expect the following observations to be expressed in the experiments:

- For both cases, we expect the OMP alone to have longer execution times as a function of the number of data samples Q. This is naturally as it has to process more data per iteration. As we add more pods P to the pool of workers, OMP should perform better on the other hand, as the total amount of data is further divided. Moreover as the dimensionality of a patch M goes up, we would have to increase sparsity K as well to keep the error in check. For the dictionary size, there should be a clear-cut connection between the number of atoms N and how long the overall algorithm takes to place.

- For the noiseless case, we expect cloud K-SVD to be able to reconstruct a near-original version of the training image. The MSE when comparing the two should decrease as a function of the number of t_d iterations. We expect a slightly higher MSE in a case where the number of t_c and t_p iterations increase, though on other hand pod P_i would become more aware of the error of pod P_j.
- For the noisy case, we expect an *acceptable* level of denoising. We will use MSE, PSNR and SSIM to measure how effective cloud K-SVD is at denoising and compare the denoised version with the original one.

Results

We start by examining results for the noiseless case (1). For all experiments in (1), we set $P = 4$. Figure 6.19 shows the average MSE between dictionaries $\boldsymbol{D}_i \; \forall i = 1, 2, \ldots, P = 4$ as a function of t_d iterations when consensus is either enabled or disabled. We see a clear drop in MSE between the dictionaries when consensus is enabled (cloud K-SVD) compared to regular K-SVD with distributed data (local K-SVD). In other words, the pods become aware of each other's data this way thus produce a dictionary that can be used to represent the entire signal $\boldsymbol{Y}$ better, not just $\boldsymbol{Y}_i$.

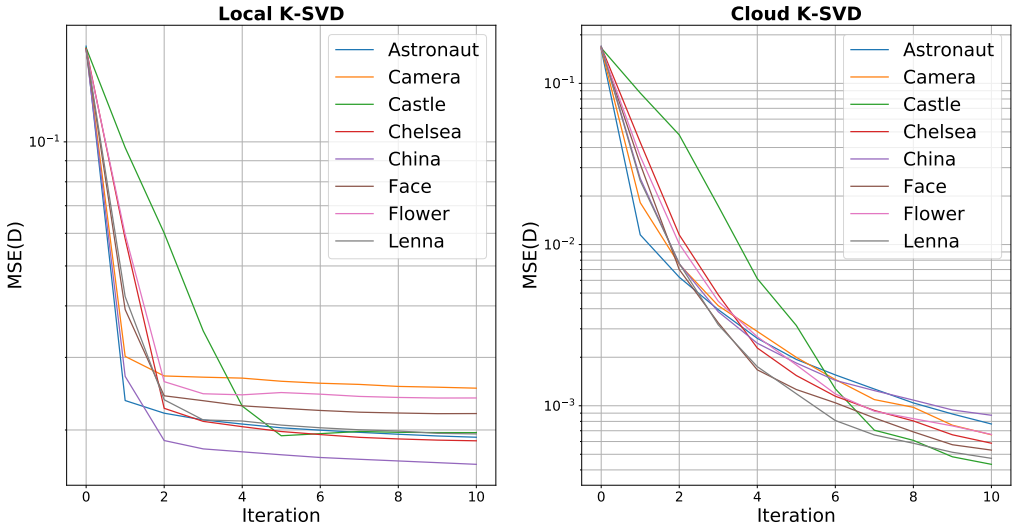

Figure 6.19: *The average MSE between dictionaries of each pod for different noiseless images.* $t_d = 0, 1, \ldots, 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 3$, $P = 4$.

Figures 6.20 and 6.21 show actual reconstruction from noiseless "Castle" and "Face" patches, respectively. We show the difference here between cloud and local K-SVD for $P = 4$, but little disparity is seen visually. This is contributed to the fact that we reconstruct $\boldsymbol{Y}_i$ by dictionary $\boldsymbol{D}_i$ and $\boldsymbol{X}_i$. If instead $\boldsymbol{D}_j$ was used to reconstruct $\boldsymbol{Y}_i$, we would expect a better reconstruction had consensus been enabled.

Figure 6.20: *Reconstructed images of Castle after local and cloud K-SVD, compared to the original.* $t_d = 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 3$, $P = 4$.

Figure 6.21: *Reconstructed images of Face after local and cloud K-SVD, compared to the original.* $t_d = 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 3$, $P = 4$.

Figures 6.22 and 6.23 show the MSE and SSIM score, respectively, when comparing the reconstructed $\hat{\boldsymbol{Y}}$ to $\boldsymbol{Y}$ in the cloud K-SVD or local K-SVD case as a function of iterations t_d. We observe similar behavior between the two variants, mainly due the fact that $\hat{\boldsymbol{Y}}_i$ is made from a corresponding dictionary $\boldsymbol{D}_i$ and signal $\boldsymbol{X}_i$. Here, "China" lacks a bit in the MSE and SSIM department, whereas "Flower" mimics the original image a lot closer. It seems the algorithm does favor more simple images, that contain lots of repeated contours and few specific structures, but shy away from those that have a great deal of small and distinctive details.

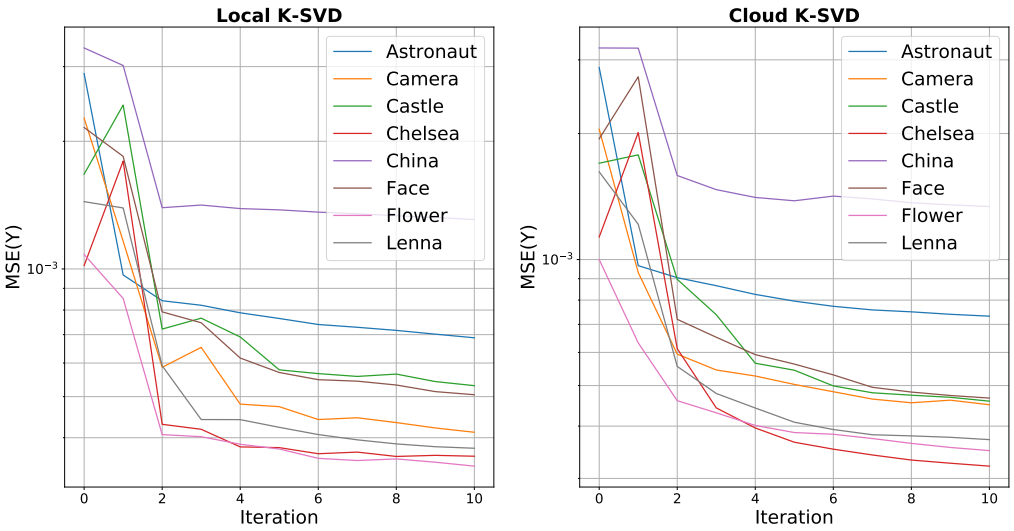

Figure 6.22: *The MSE between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$ *for different noiseless images.* $t_d = 0, 1, \ldots, 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 3$, $P = 4$.

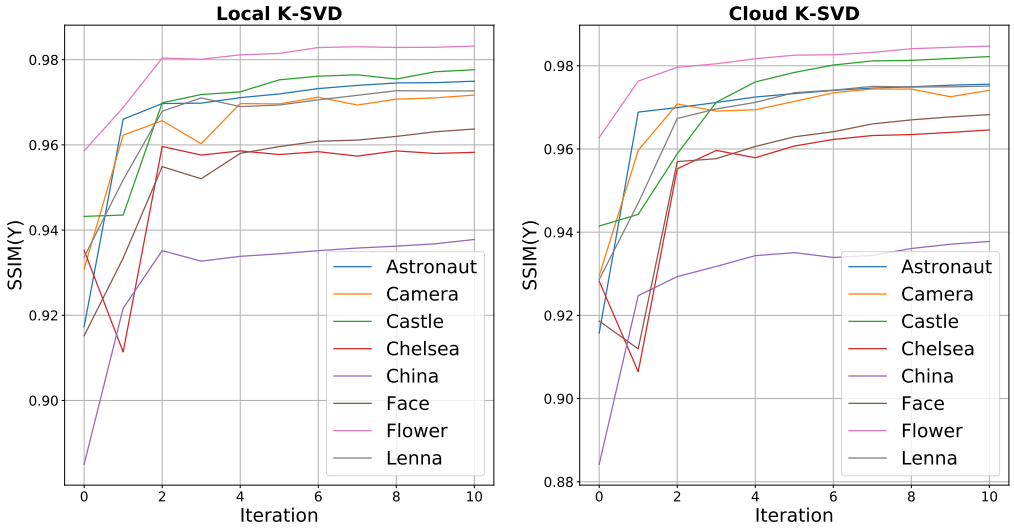

Figure 6.23: *The SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$ *for different noiseless images.* $t_d = 0, 1, \ldots, 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 3$, $P = 4$.

Figure 6.24 shows the "Castle" image reconstructed using cloud K-SVD with $K = 3, 5, 7$. There is a significant improvement in detail for $K = 7$ compared to the other two, as it appears less blurry and there is more detail in the reconstruction when K is increased. Recall that K the number of non-zero coefficients in the signal vector $\boldsymbol{X}$, so a large K should allow more detail to be kept. However to how sparse approximation works, it is difficult to completely remove the blurriness factor when reconstructing images.

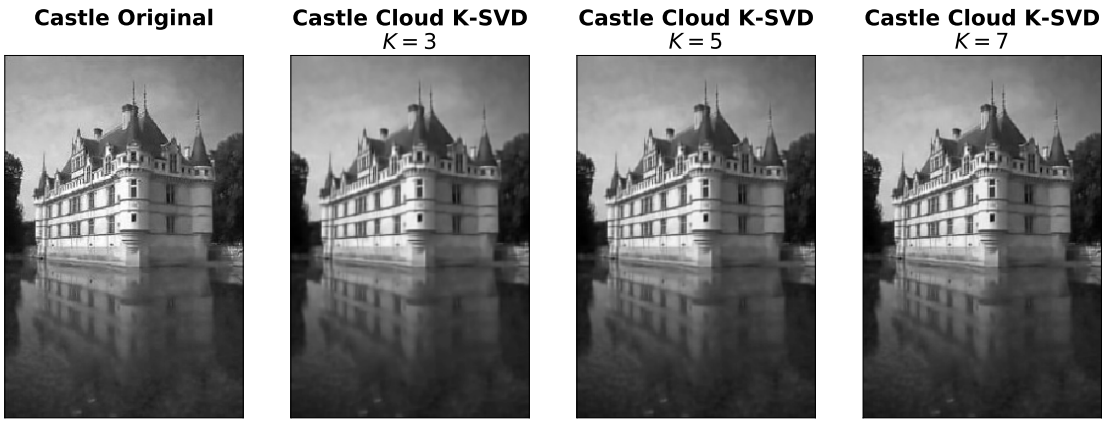

Figure 6.24: *Reconstructed images of Castle after cloud K-SVD, with varying K. $t_d = 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $P = 4$.*

Figures 6.25, 6.26 and 6.27 all show reconstructions of the "Castle" image in the cloud K-SVD case. This time we include specific error metrics for various K values like the ℓ_2-norm, MSE, the PSNR and SSIM. We see definitive improvements in the numbers when K is increased.

Figure 6.25: *Reconstructed image of Castle after cloud K-SVD, compared to the original. $t_d = 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 3$, $P = 4$.*

Figure 6.26: *Reconstructed image of Castle after cloud K-SVD, compared to the original. $t_d = 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 5$, $P = 4$.*

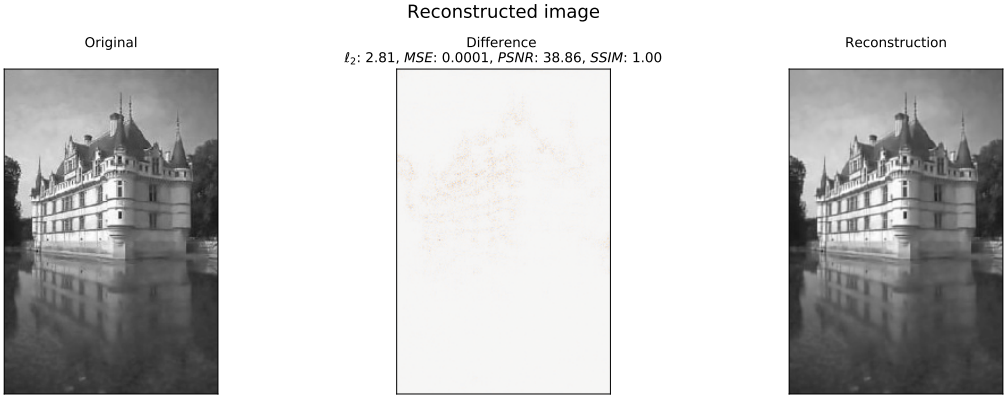

Figure 6.27: *Reconstructed image of Castle after cloud K-SVD, compared to the original.* $t_d = 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $K = 7$, $P = 4$.

Figure 6.28 shows how the error metrics are effected when setting the sparsity level $K = 3, 5, 7$ as a function of t_d iterations. We notice that a higher K seem to decrease MSE, ℓ_2-norm, PSNR and SSIM between $\boldsymbol{Y}$ and $\hat{\boldsymbol{Y}}$ significantly for the noiseless case.

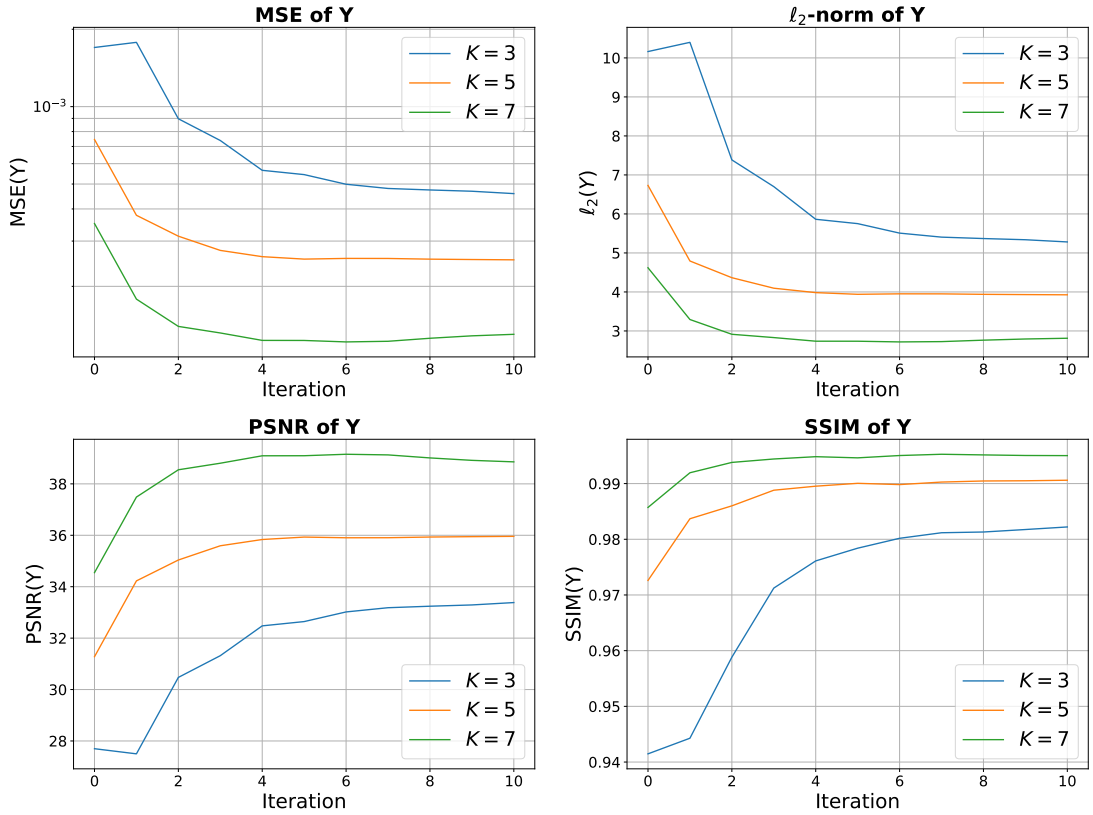

Figure 6.28: *The MSE, ℓ_2-norm of the error, PSNR and SSIM between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$, of the Castle image, at different K.* $t_d = 0, 1, \ldots, 10$, $t_p = 3$, $t_c = 5$, $M = 25$, $N = 50$, $P = 4$.

Figures 6.29 and 6.30 show reconstruction errors as a function of the number of iterations t_d when the patch dimension is increased ($M = 25, 36, 49, 64, 81$) and a visual presentation of the same experiments, respectively. Recall that patch size PS is the resolution of extracted patches and M is the dimension (e.g. for $PS = 5 \times 5$ then $M = 25$). By rule of thumb, $N \gg M$, so we increase N as well. Here all experiments are done with noiseless training data from the "Face"

image. We see a lower MSE and ℓ_2-norm between $\boldsymbol{Y}$ and $\hat{\boldsymbol{Y}}$ when the patch size is lower, for example $M = 25$, by comparison to a larger patch size, for example $M = 49$. The PSNR and SSIM exhibit similar behavior when the patch size goes up. Remember that we seek a low MSE and ℓ_2-norm, but a high PSNR and SSIM.

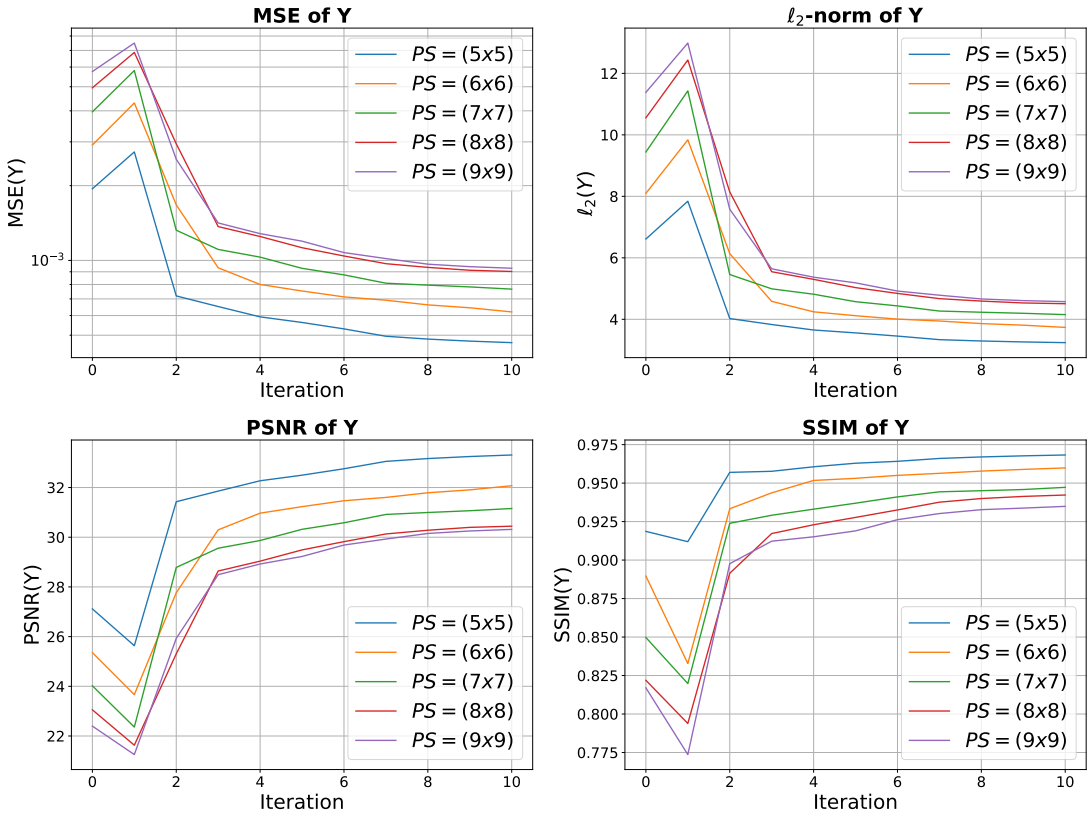

Figure 6.29: *The MSE, ℓ_2-norm of the error, PSNR and SSIM between the original signal $\boldsymbol{Y}$ and the reconstructed signal $\hat{\boldsymbol{Y}}$, of the Face image, at different M and N. $t_d = 0, 1, \ldots, 10$, $t_p = 3$, $t_c = 5$, $K = 3$, $P = 4$.*

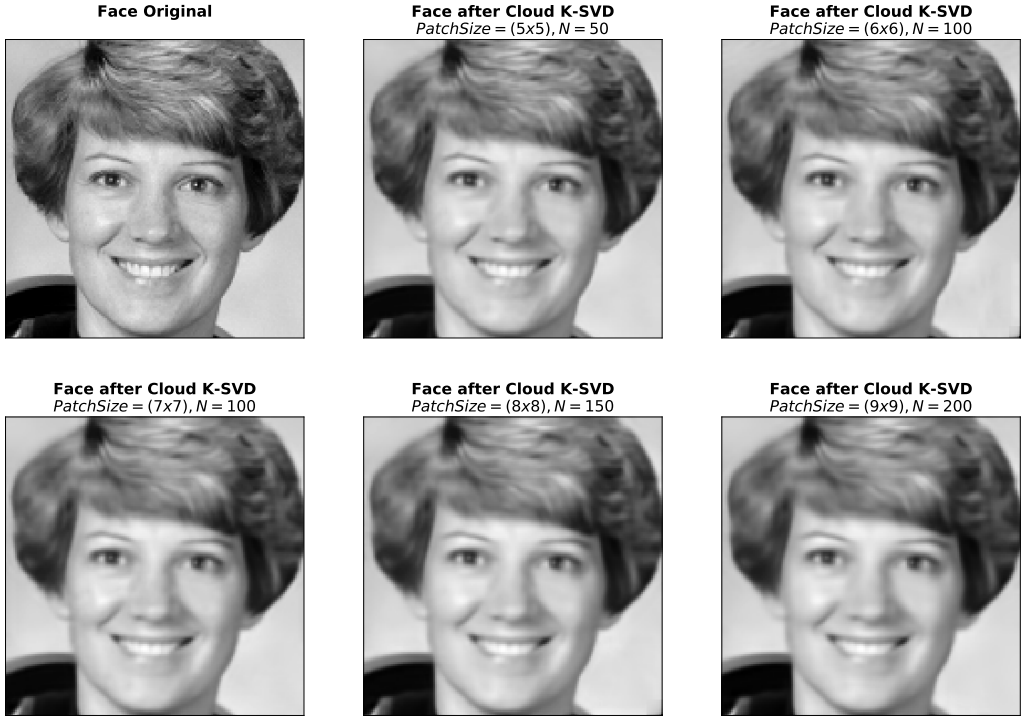

Figure 6.30: *Visual representation of reconstructions after cloud K-SVD from noiseless version of Face, at varying data sizes M and N. $t_d = 10$, $t_p = 3$, $t_c = 5$, $K = 3$, $P = 4$.*

Table 6.4, 6.5 and 6.6 show average execution times for P pods tracked for the OMP and K-SVD step for values of $K = 3, 5, 7$, $M = 25, 36, 49, 64, 81$ and $\alpha = 1, 2$ in local K-SVD and cloud K-SVD runs, respectively. For all runs we set $P = 4$. The OMP step seems to be impacted significantly by a higher K, as expected, whilst the K-SVD step is clearly doing alright with K set high. On the other hand, OMP is not really slowed down by using larger patch dimensionality M but K-SVD takes a hit as it increases. This is because generally $N \gg M$, usually twice its size, so the dictionary size N gets bigger as M go up. Because K-SVD updates the dictionary per atom, it naturally takes longer to update a large dictionary than a small one.

	Astronaut	**Camera**	**Castle**	**Chelsea**	**China**	**Face**	**Flower**	**Lenna**
Local OMP	$28s$	$29.4s$	$27.3s$	$15.1s$	$30.6s$	$9.8s$	$30.7s$	$27.6s$
Cloud OMP	$34.2s$	$32.6s$	$31.6s$	$18.5s$	$34.2s$	$12.5s$	$35.6s$	$31.2s$
Local K-SVD	$7.5s$	$7.1s$	$6.7s$	$3.3s$	$8s$	$2.5s$	$7.6s$	$6.8s$
Cloud K-SVD	$42.3s$	$43.1s$	$43.5s$	$33.9s$	$44.9s$	$31.9s$	$46.4s$	$42.8s$

Table 6.4: *The average Execution times of OMP and K-SVD iterations, for local K-SVD ($t_p = 0, t_c = 0$) and cloud K-SVD ($t_p = 3, t_c = 5$) using noiseless patches. $K = 3$, $M = 25$, $N = 50$. Castle and Face at $\alpha = 1$, the rest at $\alpha = 2$*

	Castle, $K = 3$	**Castle**, $K = 5$	**Castle**, $K = 7$
Local OMP	$27.3s$	$45.2s$	$64.4s$
Cloud OMP	$31.6s$	$51.9s$	$74.2s$
Local K-SVD	$6.7s$	$8.7s$	$11.1s$
Cloud K-SVD	$43.5s$	$45.4s$	$43.1s$

Table 6.5: *The average Execution times of OMP and K-SVD iterations, for local K-SVD ($t_p = 0, t_c = 0$) and cloud K-SVD ($t_p = 3, t_c = 5$) using Castle with different sparsity K values. $M = 25$, $N = 50$, $\alpha = 1$.*

	Face $M = 25$ $N = 50$	**Face** $M = 36$ $N = 100$	**Face** $M = 49$ $N = 100$	**Face** $M = 64$ $N = 150$	**Face** $M = 81$ $N = 200$
Local OMP	$9.8s$	$10s$	$10.1s$	$10.6s$	$10.9s$
Cloud OMP	$12.5s$	$14.3s$	$13.3s$	$16.3s$	$15.9s$
Local K-SVD	$2.5s$	$4.8s$	$6.6s$	$12.9s$	$17.7s$
Cloud K-SVD	$31.9s$	$63.4s$	$71.5s$	$120.5s$	$157.9s$

Table 6.6: *The average Execution times of OMP and K-SVD iterations, for local K-SVD ($t_p = 0, t_c = 0$) and cloud K-SVD ($t_p = 3, t_c = 5$) using Face with different data sizes M and N values. $K = 3$, $\alpha = 1$.*

We now turn our attention to the noisy training case (2). We add a layer of Gaussian distributed noise by mean $\mu = 1$ and variance $\sigma^2 = 0.001, 0.005, 0.01$ to all training images and then extract Q noisy patches. Figure 6.31 shows the MSE, PSNR and SSIM scores for $\hat{\boldsymbol{Y}}$ as a function of t_d iterations compared to the original noiseless $\boldsymbol{Y}$ after cloud K-SVD has reconstructed the "Face" image from noisy patches. This is the severe noise case. Here, the variance is set to $\sigma^2 = 0.01$ various configurations of PS, K and N is set as seen in the legend. We see a break-even around iteration 4 for most cases and some configurations even tend to do worse as the number of iterations go up. Because the noise level is quite high (heavy-tailed Gaussian), at some iteration the algorithm now starts reconstructing the noise because it has only been exposed to these noisy measurements.

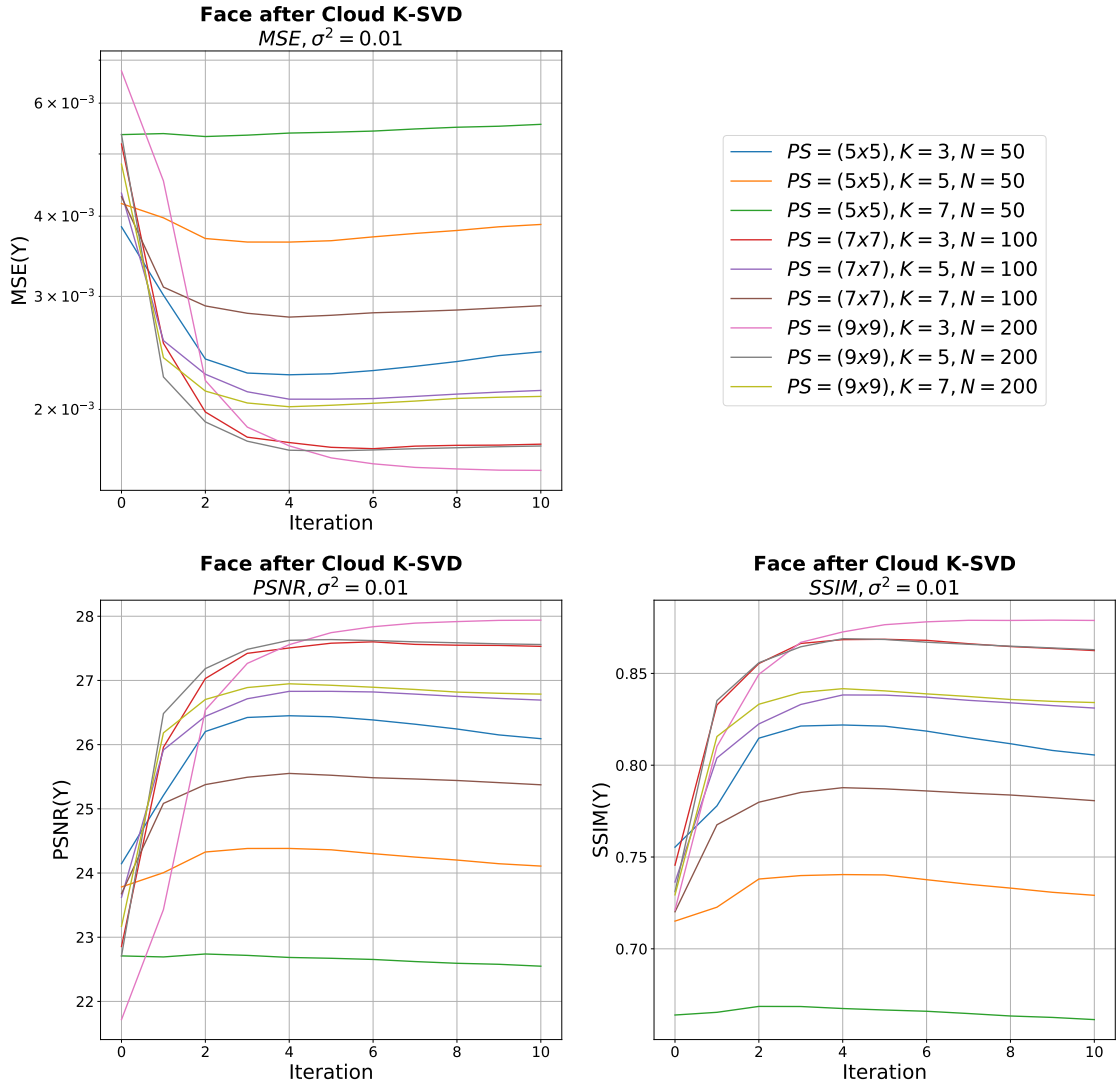

Figure 6.31: *The MSE, PSNR and SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$*, of the Face image with added Gaussian noise* ($\sigma^2 = 0.01$)*, at different* M*,* N *and* K*.* $t_d = 0, 1, \ldots, 10$*,* $t_p = 3$*,* $t_c = 5$*,* $P = 4$*.*

Figure 6.32 mimics the data setup in 6.31, but with a variance set to $\sigma^2 = 0.005$. This is the middle ground for the noise level in our experiments. Clearly, a low patch size and a high sparsity (the green line) struggles to reduce noise, whilst a low sparsity configuration does a better job. By looking at the results, an acceptable configuration is a patch size around 7×7 thus $M = 49$, an atom count of 100 and a sparsity of $K = 3$ for our training data.

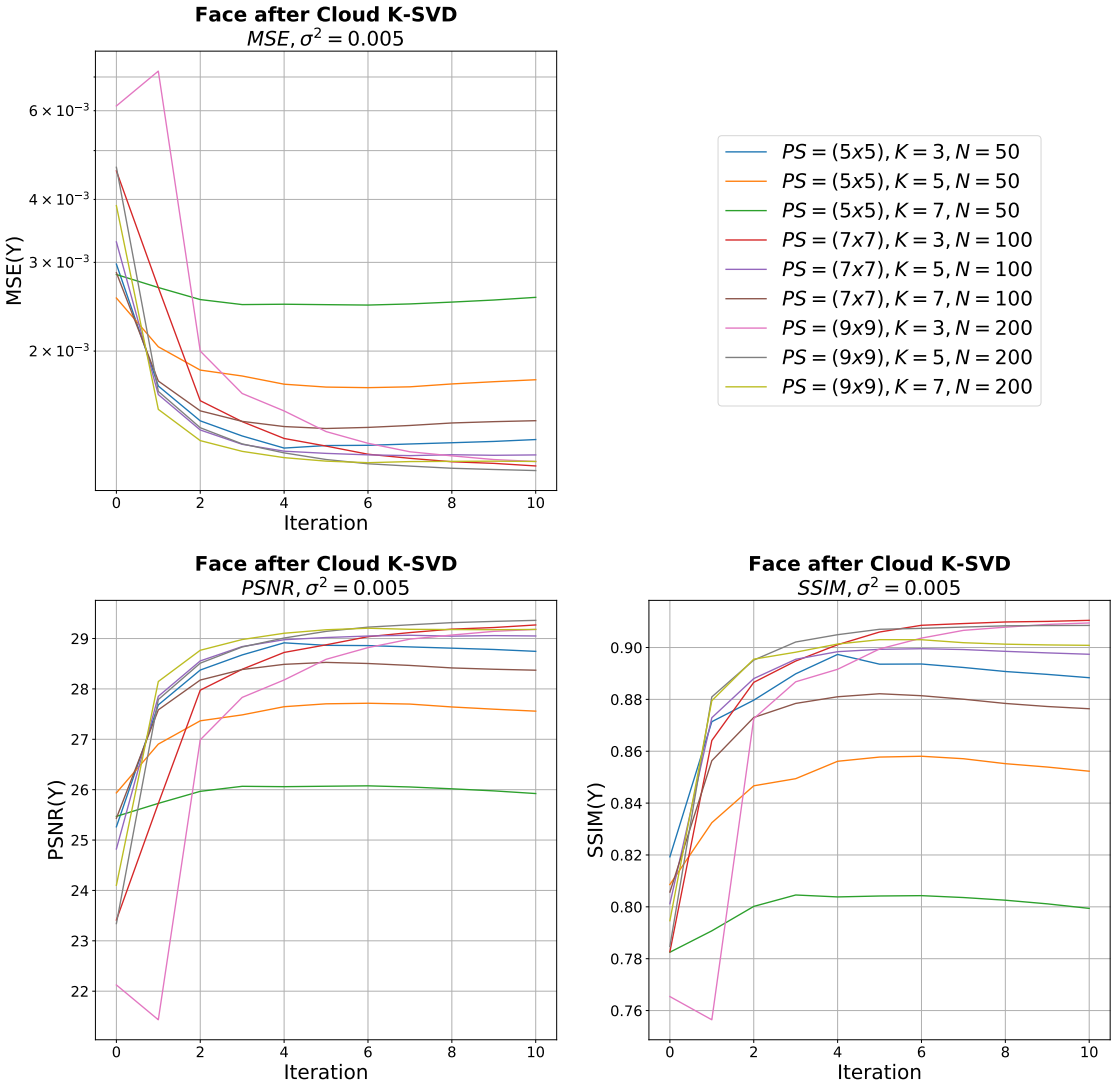

Figure 6.32: *The MSE, PSNR and SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$*, of the Face image with added Gaussian noise* ($\sigma^2 = 0.005$)*, at different* M*,* N *and* K*.* $t_d = 0, 1, \ldots, 10$*,* $t_p = 3$*,* $t_c = 5$*,* $P = 4$*.*

Figure 6.33 mimics the data setup in 6.31, but with a variance set to $\sigma^2 = 0.001$. We see that a high pixel size PS with a low sparsity K setting is no longer ideal when the noise variance is low. When K is low and PS is high (the pink line), cloud K-SVD struggles to properly reconstruct the image. It shows that cloud K-SVD performs better with high sparsity in case there is little to no noise in the input signal, but does better with a low sparsity in case the input is contaminated or distorted in any way.

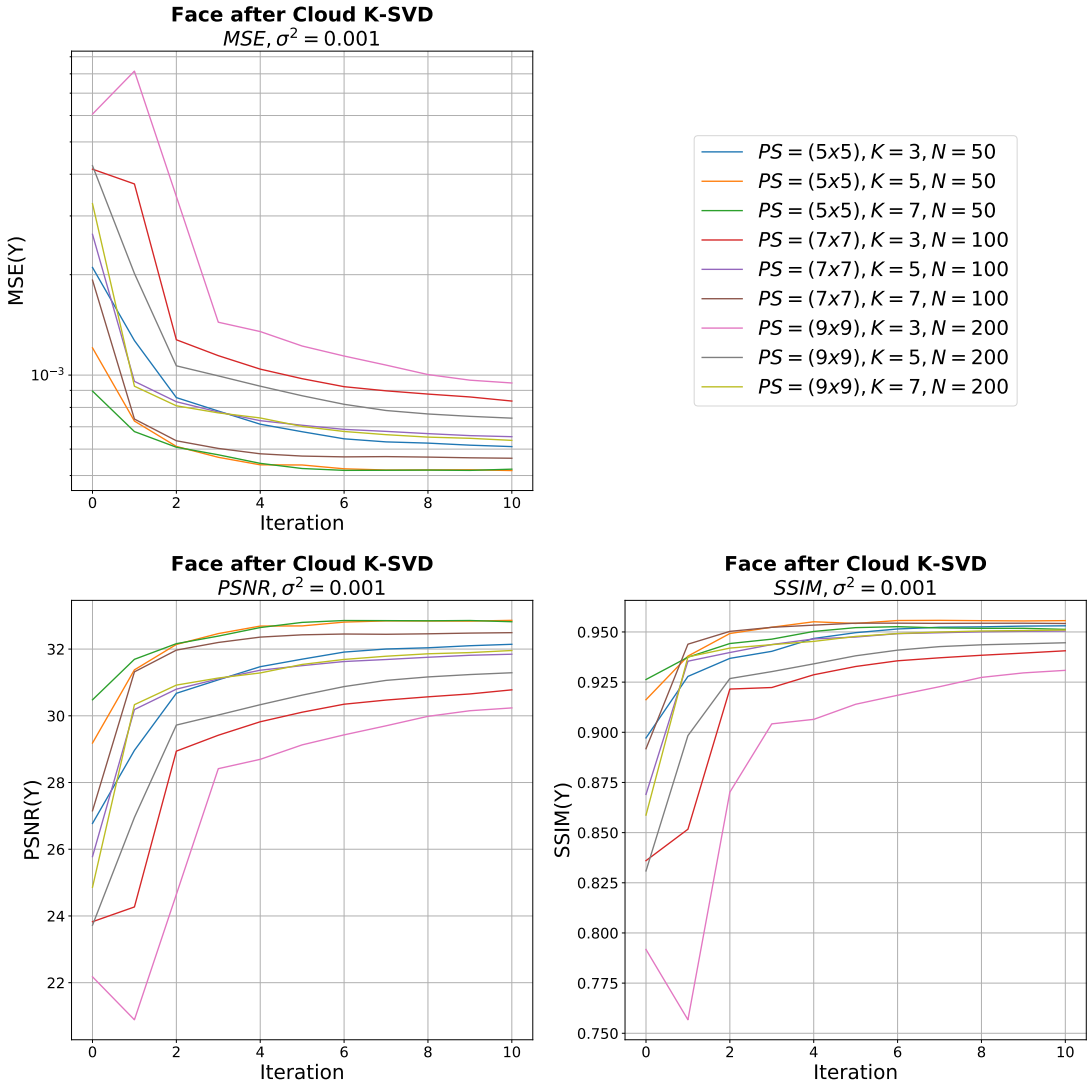

Figure 6.33: *The MSE, PSNR and SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$*, of the Face image with added Gaussian noise* ($\sigma^2 = 0.001$)*, at different* M*,* N *and* K*.* $t_d = 0, 1, \ldots, 10$*,* $t_p = 3$*,* $t_c = 5$*,* $P = 4$*.*

Figure 6.34 shows the "Face" image now contaminated with heavy-tailed Gaussian noise by mean $\mu = 1$ and $\sigma^2 = 0.01$ and accompanied error scores between that and the original. Figure 6.35 is then our reconstruction attempts with cloud K-SVD using a sparsity of $K = 3, 5, 7$, patch size $PS = 5 \times 5, 7 \times 7, 9 \times 9$ and the number of atoms $N = 50, 100, 200$. We see significant noise reductions with sparsity $K = 3$, a large patch size of $PS = 9 \times 9$ and a high number of dictionary atoms $N = 200$. Even if K is high, the PS and N makes up for it because cloud K-SVD then gets a larger pool of training data compared to a low PS and N.

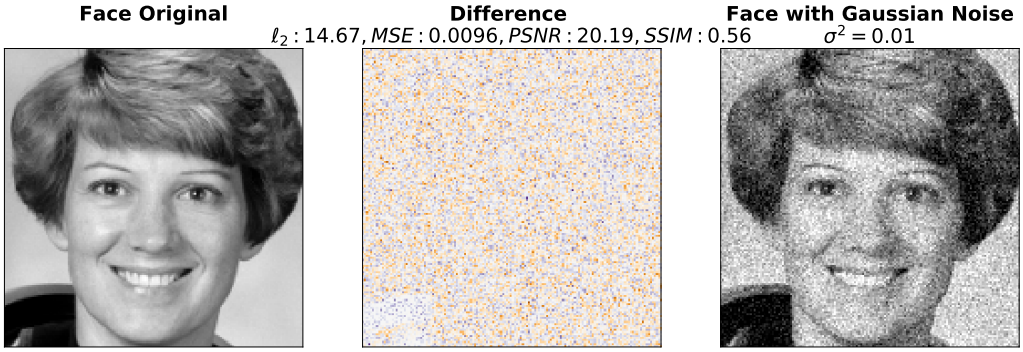

Figure 6.34: *Original image of Face, compared to the noisy version.* $\sigma^2 = 0.01$.

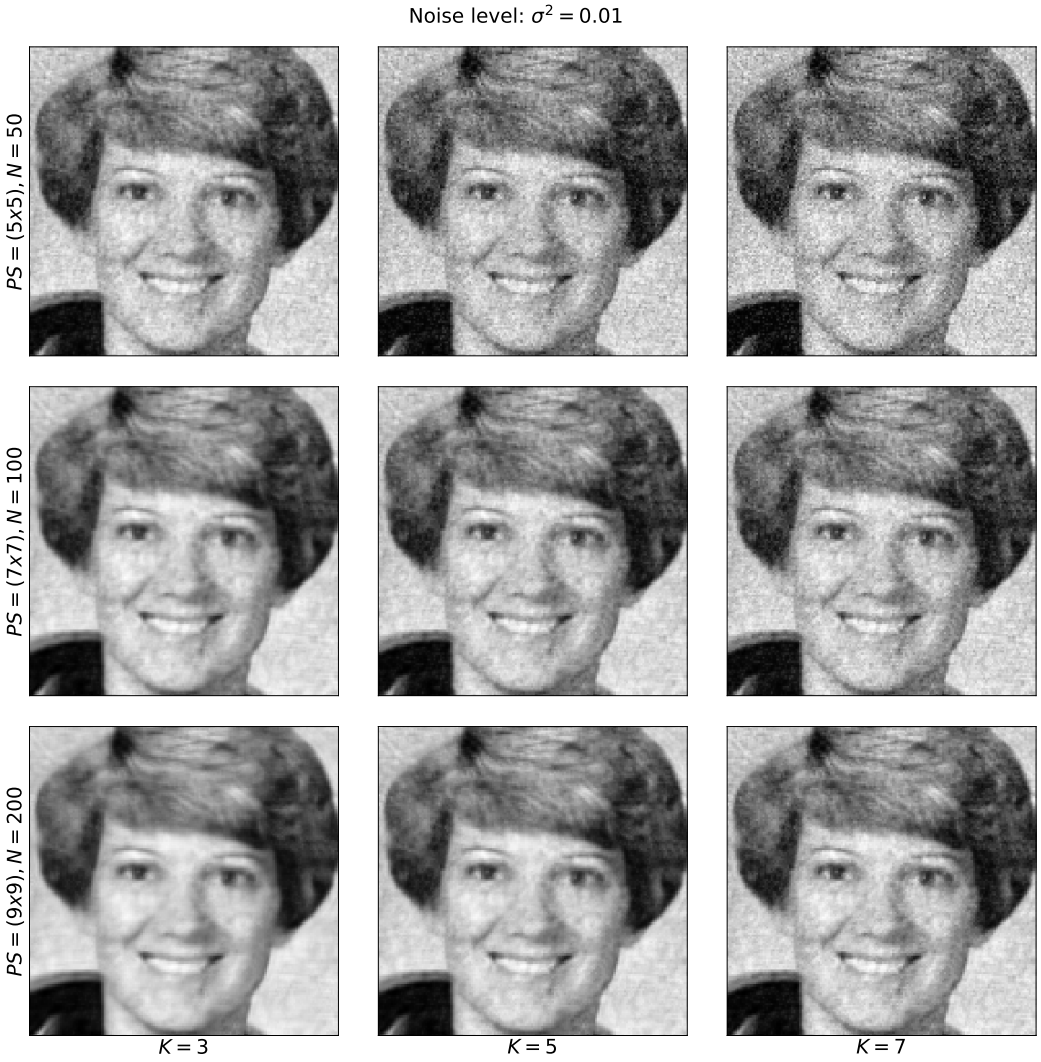

Figure 6.35: *Visual representation of reconstructions after cloud K-SVD from noisy version of Face, at varying* M, N *and* K. $\sigma^2 = 0.01$, $t_d = 10$, $t_p = 3$, $t_c = 5$, $P = 4$.

Figures 6.36 and 6.37 turns the heat down a bit, as we now set the noise variance to $\sigma^2 = 0.005$. The configuration is similar to that of 6.34 and 6.35 however.

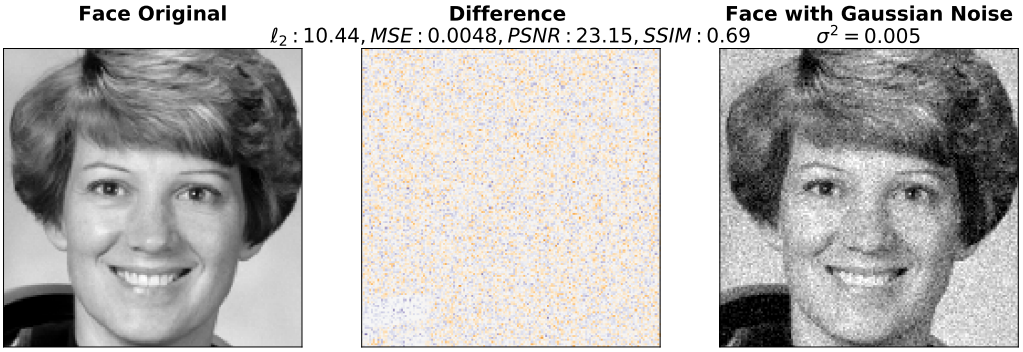

Figure 6.36: *Original image of Face, compared to the noisy version.* $\sigma^2 = 0.005$.

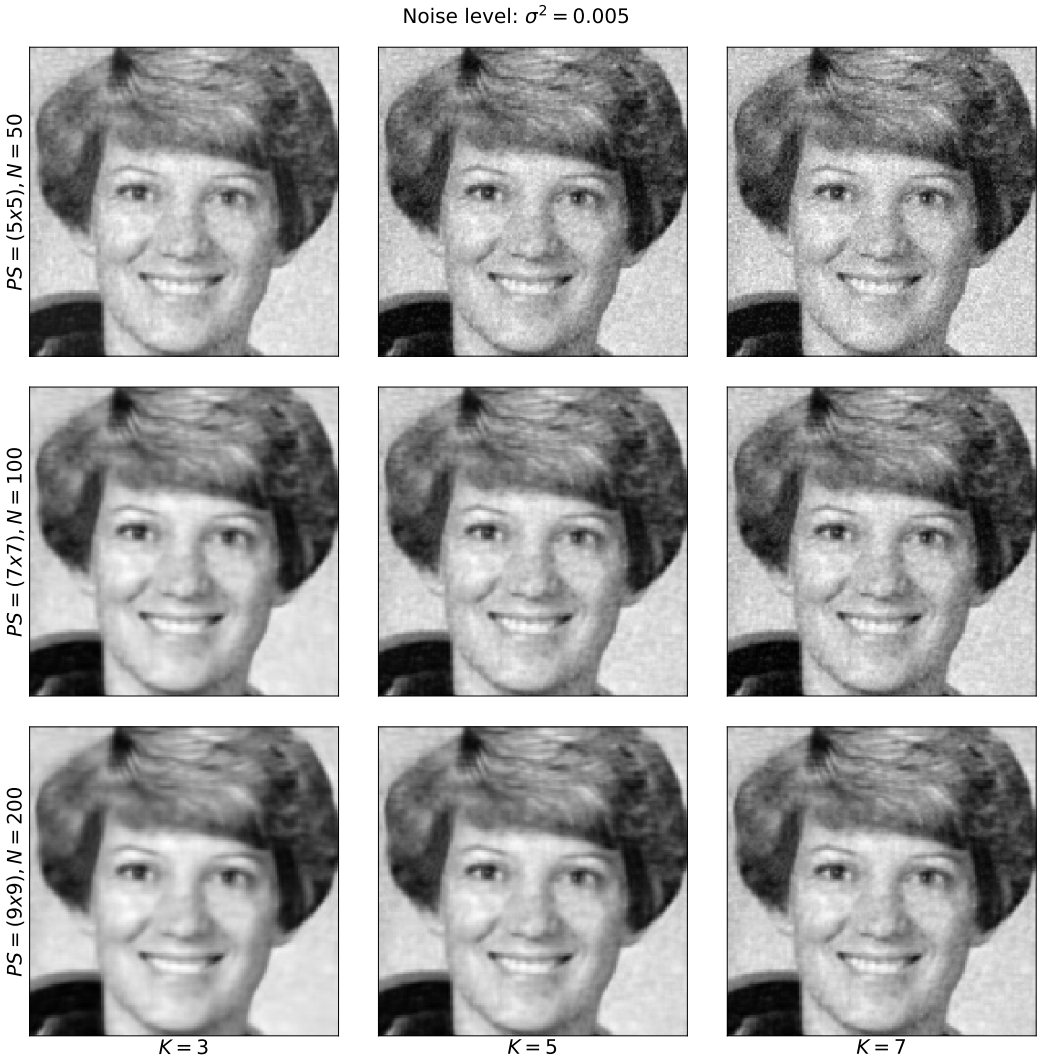

Figure 6.37: *Visual representation of reconstructions after cloud K-SVD from noisy version of Face, at varying* M, N *and* K. $\sigma^2 = 0.005$, $t_d = 10$, $t_p = 3$, $t_c = 5$, $P = 4$.

Figures 6.38 and 6.39 show the reconstruction of "Face" with the least amount of noise as $\sigma^2 = 0.001$.

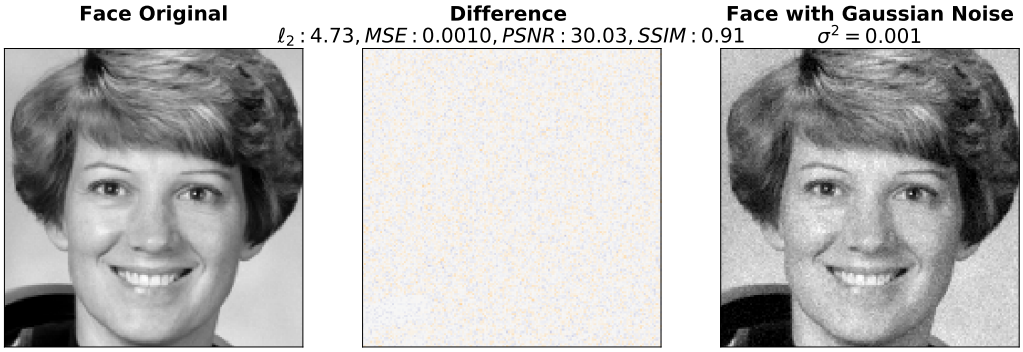

Figure 6.38: *Original image of Face, compared to the noisy version.* $\sigma^2 = 0.001$.

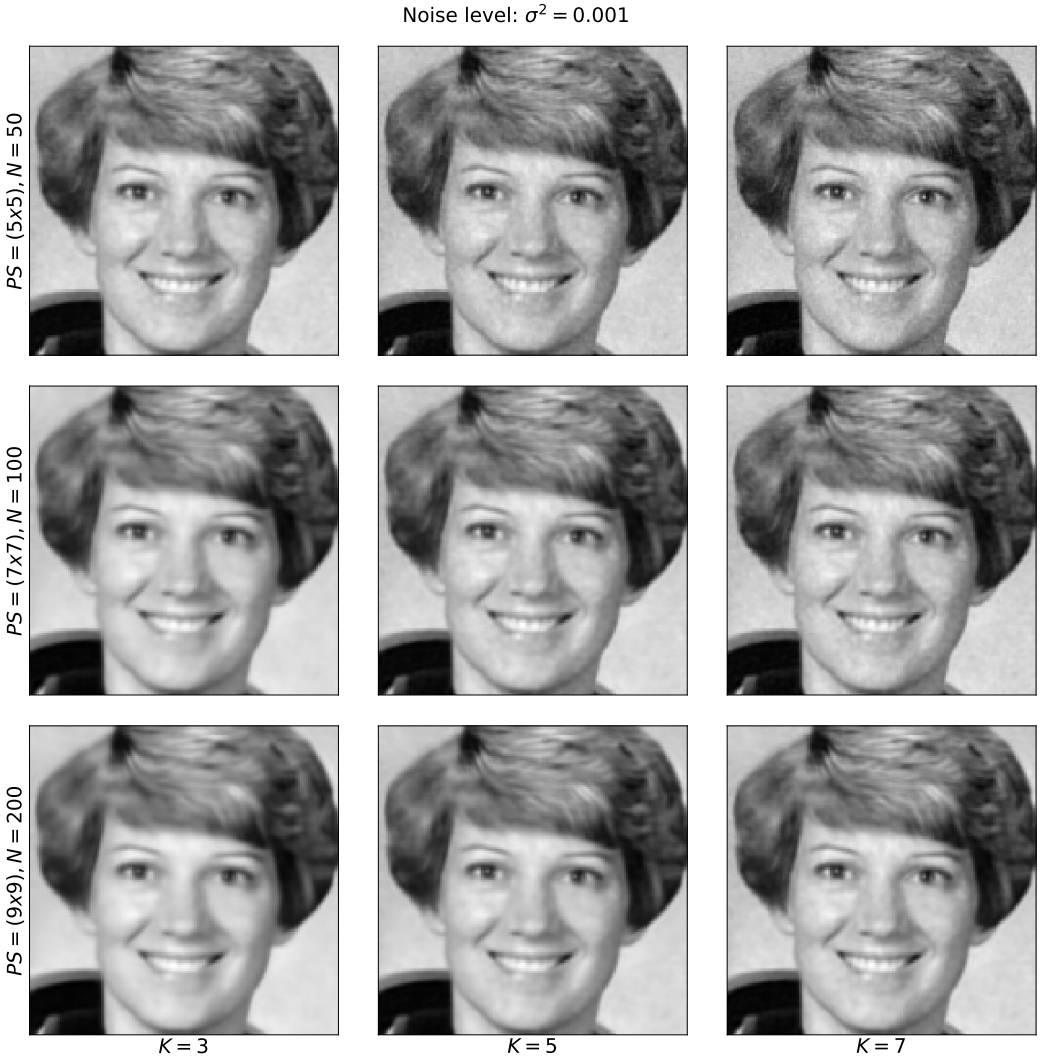

Figure 6.39: *Visual representation of reconstructions after cloud K-SVD from noisy version of Face, at varying* M, N *and* K. $\sigma^2 = 0.001$, $t_d = 10$, $t_p = 3$, $t_c = 5$, $P = 4$.

We chose not to include execution times for the noisy experiments (2), because they mimic the same trends as the ones in the noiseless case (1). It is no matter if training images are clean

or contaminated with additive noise, OMP and the collaborative K-SVD behave the same. All timings are included in the source material.

Discussion

The purpose of these experiments was to measure how well cloud K-SVD did at learning geometric structures in patches that had been extracted from either noiseless or noisy images. To get an overall picture of the algorithm's performance, we used eight different images to test the noiseless case. However because of the hardware limitations of the cluster, we needed to downscale some of them when testing. We start by looking at the effect of local versus cloud K-SVD on the dictionary (1), results shown in figure 6.19. We observe that much like the test on synthetic data, enabling the consensus step makes the individual dictionaries $\boldsymbol{D}_i$ of each pod converge. Looking at the reconstruction of the image itself, see figures 6.22 and 6.23, the overall results argue that cloud and local K-SVD perform the same when reconstructing clean images. The *China* image had the largest error when compared to the original, and *Chelsea* and *Flower* had the lowest error. *China* is special, since it contains a lot of detailed straight lines, which proved difficult to reconstruct. We also see that the higher variance of any image, see table 6.3, the greater an overall MSE of the reconstructed image was observed when compared to the original, see figure 6.22.

Moving forward, we test the impact of setting sparsity using *Castle*. This image was used because no downscaling was needed for a patch size of (5×5). The test was conducted with sparsity K of $\boldsymbol{X}$ with values 3, 5 and 7. The reconstruction results, see figure 6.28, show that higher sparsity means lower reconstruction error. This is of course expected, since more data naturally would reconstruct with more details. The final reconstruction is seen in figures 6.25, 6.26 and 6.27.

After sparsity, we look at the data size for, mainly M and N, by changing the patch sizes used from (5×5) up to (9×9). For this test we used only the *Face* image, because it did not need downscaling. The reconstruction error, see figure 6.29, show that smaller patches leads to a smaller error. In other words, the more data each patch or signal holds, the harder it is to reconstruct said patch or signal from a dictionary. The reconstructed images are in figure 6.30, where we also observe that the image becomes less detailed as each data signal increases in size.

Looking at execution times, see tables 6.4, 6.5 and 6.6, we notice a clear distinction between local and cloud K-SVD. The biggest time difference is for the K-SVD step itself, as expected, because the consensus step is embedded in the K-SVD. What might come as a surprise is the increased time spent to do OMP when the number of consensus iterations is slightly increased. We do not know the reason for this, but because the dictionary $\boldsymbol{D}$ is of more common nature on the individual pod's when consensus is enabled, it might be harder for OMP to reconstruct compared to a very specialized dictionary. If we look further into situations where we change K of $\boldsymbol{X}$, data dimensions M and atoms N, we also observe consequences in execution times. For the change in sparsity, we see that as we increase the sparsity, we prolong the time it takes to complete the OMP step, while the K-SVD step only increase slightly. If we observe the change in data size, M and N, we see that as it increases, the time it takes to complete K-SVD goes up as well, while OMP stays the same.

After testing on noiseless images it is clear that both sparsity K of $\boldsymbol{X}$ and data sizes M and N have a significant impact on the reconstruction process in different ways. We extended our experiments to noisy images (2) using three levels of Gaussian noise on the *Face* image and only using cloud K-SVD. The results are in figures 6.31, 6.32 and 6.33. They show, that the more noise we induce in training images, the less detail we are able to properly reconstruct, since keeping details will also keep noise. To give an example, exhibit the denoising of *Face* distorted with Gaussian noise that had variance $\sigma^2 = 0.01$ in figure 6.31. We note, that the denoising

effect comes when the patch size is set to (9×9) and sparsity is set to $K = 3$. Compared to tests before, this should be the scenario with least detail maintained compared to the original, however because the noise is so prominent, it is actually to our advantage. Compare this result to the denoising of *Face* with Gaussian noise that had variance $\sigma^2 = 0.001$ in figure 6.33. Here we see the exact opposite, namely the best denoising effects show with a small patch size like (5×5) and a high sparsity. Previous experiments point to that this setup (small patch sizes) will maintain most detail compared to one with large patch sizes and small sparsity. This was rather expected, since by only inducing a small amount of noise, the image still looks a lot like our original. The images themselves are exhibited in figures 6.35, 6.37 and 6.39. The denoising effect can be observed by looking at the images.

Through experiments, we find that when denoising images with cloud K-SVD, the parameters of the algorithm need to be adjusted to the amount of noise in images. There is no clear and correct configuration of cloud K-SVD. When the noise is severe, we have to remove a lot of detail in order to reduce the noise, on the contrary when it is more manageable, we do not have to remove that much detail to reduce it. Details can be effectively removed by increasing data/patch sizes, M and N, and decreasing sparsity K. We can recover details better by doing the opposite, that is decrease data/patch sizes, M and N, and increase sparsity K.

6.4 Experiments using medical images

Purpose

The final set of experiments will test if cloud *K-SVD* can remove noise from real medical images that exhibit the hands and wrists of people suffering from the chronic condition rheumatoid arthritis (RA). Noise and artifacts are often the result of patient motion, the scanner itself or by the amount of joint changes in the images. Image quality is a crucial factor for doctors and radiologists to measure disease progression and observe the amount of joint destruction it has wrought, since this information is an indicator of disease activity in RA [83]. All training data has been anonymized prior to evaluation and obtained as part of the DanACT research study with permission to use it for laboratory tests and experiments, but not for redistribution. The DanACT study was established in 2014 with the goal of estimating effectiveness and the safety of various protocols for treating patients with RA. The study group includes rheumatology departments in Aarhus, Silkeborg, Horsens, Gråsten, Svendborg and Odense, Denmark. We have the following objectives in view:

- Measure how efficient cloud *K-SVD* is at removing additive white Gaussian noise and unwanted artifacts from medical images.
- Measure if cloud *K-SVD* can remove noise and undesirable artifacts in the training images without removing structures and information necessary for disease characterization. In other words, can we reduce the amount of noise without sacrificing critical information such as joint edges and visible bone erosions.

Data and setup

All training images exist exclusively in noisy versions and all comparisons are therefore done with their original noisy version. A single dataset consists of 330 images slices that stem from concatenating three individual scans covering a total scan region of $2.7cm$ by capturing 110 image slices with a length of $0.9cm$ each. This way the doctors archive a complete scan of the joints of interest. Each dataset has a voxel[2] size of $82 \times 82 \times 82\mu m$, with resolutions varying between scans. Increasing the image voxel size results in an increased signal-to-noise ratio, but it brings a diminution in spatial resolution. For our experiments, we consider only the 2D representation of the images for performance reasons.

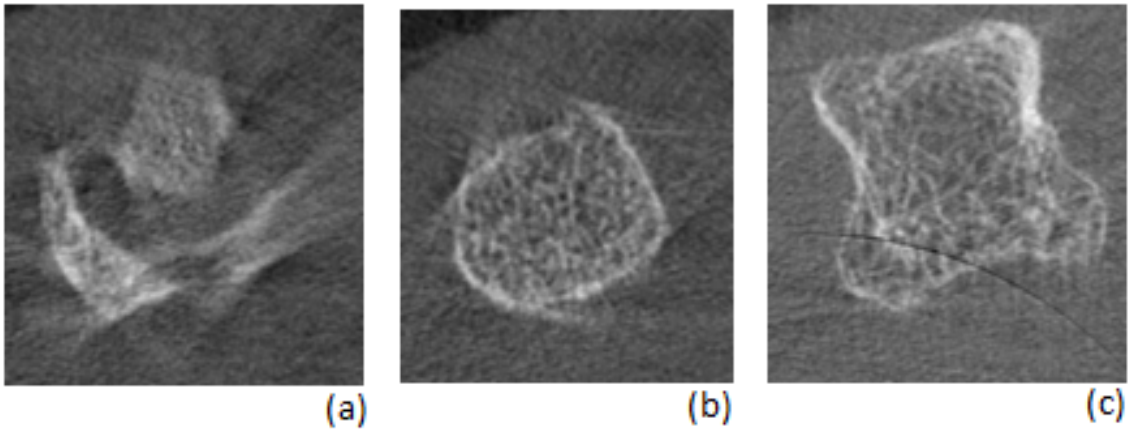

Figure 6.40: *Three noisy images from the dataset. (a) shows that joint change can cause noise in the respective slices. (b) shows patient motion which causes noise that blurs the border of the bone. (c) shows ring artifacts at the border region of the bone which can make it hard to distinguish.*

Examination of the image data reveals that we mostly are dealing with three kinds of noise that all degrade image quality: The first and most common is addictive Gaussian noise, something cloud *K-SVD* in theory should be excellent at reducing. The second kind of noise is patient motion. When the individual moves his or her hand any point during the scan it is considered patient motion. Its effects can causes blurring and stripe through the images as the second image in figure 6.40 shows. It takes the scanner eight minutes to acquire all images in a single scan, and it can be difficult for people to hold their hand or wrist still for that long. Thus patient motion artifacts are highly present in the images. The last kind of noise is ring artifacts seen in the third image in figure 6.40. These are typically caused by miscalibration of the scanner or by defective detector elements [84]. They can be corrected or removed through proper scanner recalibration or by postprocessing methods. The main problem with these ring artifacts is that they share pixel intensity values with actual bone structures and radiologists can therefore find it difficult to distinguish the two.

For our algorithm, things such as patient motion and ring artifacts can prove more problematic than simple Gaussian noise, since there exist some correlation in those versions of noise and it might reside in greater eigenvectors, which we normally want to maintain in the image.

For all experiments, we only test the cloud K-SVD configuration with number of pods $P = 4$ and consensus enabled, on the zoomed in image shown on Figure 6.41.

[2]A voxel is the 3D analogue of a pixel and relates to both the pixel size and thickness.

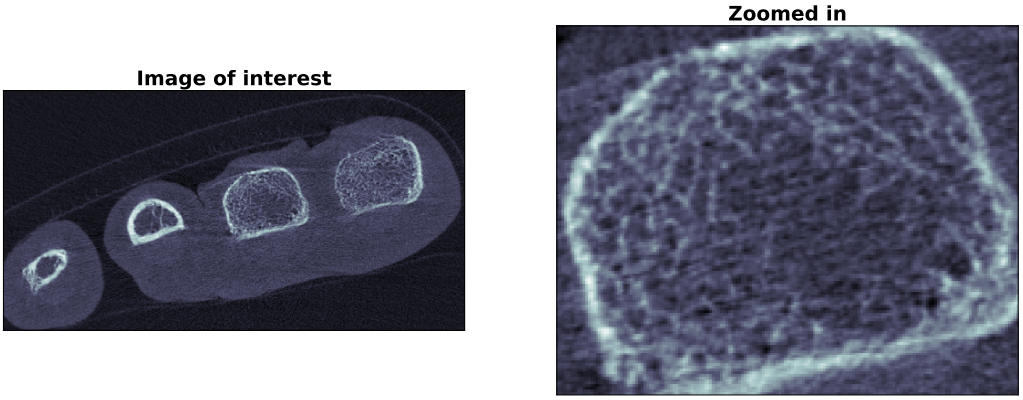

Figure 6.41: *Image of interest to be tested in the following experiments. The image has an overall bad quality since all three versions of noise are embedded in the image. The zoomed image will be used because of hardware limitations. In the zoomed version both Gaussian and ring artifact noise are present.*

Expectations

We expect the following observations to be expressed in the experiments:

- We expect to see effects of denoising in the medical images. However since all training material has noise embedded directly in the data, we will not be able to measure cloud *K-SVD*'s effect the same way we did in section 6.3. The expectation is that there will be improvements to the data by removing the noise with Gaussian characteristics.

Results

Figure 6.42 shows graphs for MSE, PSNR and SSIM scores for the recovered $\hat{\boldsymbol{Y}}$ as a function of t_d iterations when compared to the original noisy $\boldsymbol{Y}$ after cloud K-SVD. We see that trials with a low patch size (5×5) and high sparsity $K = 7$ does reasonably well in all tests, whilst larger patch sizes and low sparsity seem to struggle. PSNR results show notable lower numbers than with patch learning, which indicate that CT training images contain considerable amounts of noise, however cloud K-SVD seem to improve these as a function of iterations t_d. Figure 6.42 indicates that between four to six iterations is a rational setting, as improvements diminish after this point.

Figure 6.43 shows actual reconstruction from noisy medical images exhibiting a bone at different configurations of sparsity K, patch size PS and dictionary size N. Although difficult to see, a higher sparsity setting retain some degree of meaningful details and unwanted artifacts in the image, while a lower sparsity practically removes both.

Execution times when denoising medical images show the same trends as those from patch learning without any noise. OMP and K-SVD are not affected by what training data we train with, but rather the sparsity and patch size (dimensionality) of it. We refer to Table 6.4, 6.5 and 6.6 in section 6.3 for trends in execution times. All timings are per usual included in the source material.

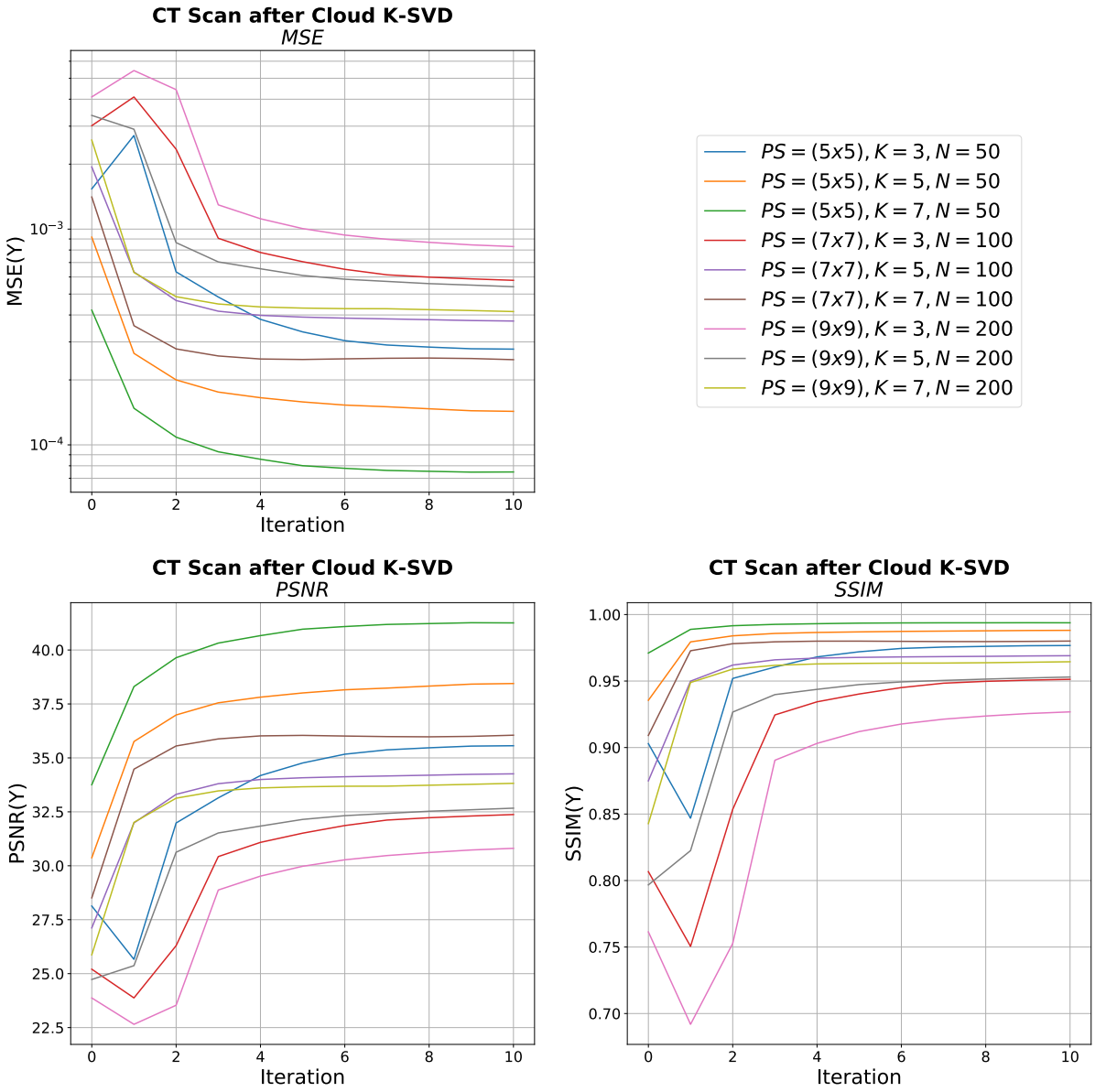

Figure 6.42: *The MSE, PSNR and SSIM between the original signal* $\boldsymbol{Y}$ *and the reconstructed signal* $\hat{\boldsymbol{Y}}$*, of the medical image, at different* PS/M*,* N *and* K*.* $t_d = 0, 1, \ldots, 10$*,* $t_p = 3$*,* $t_c = 5$*,* $P = 4$*.*

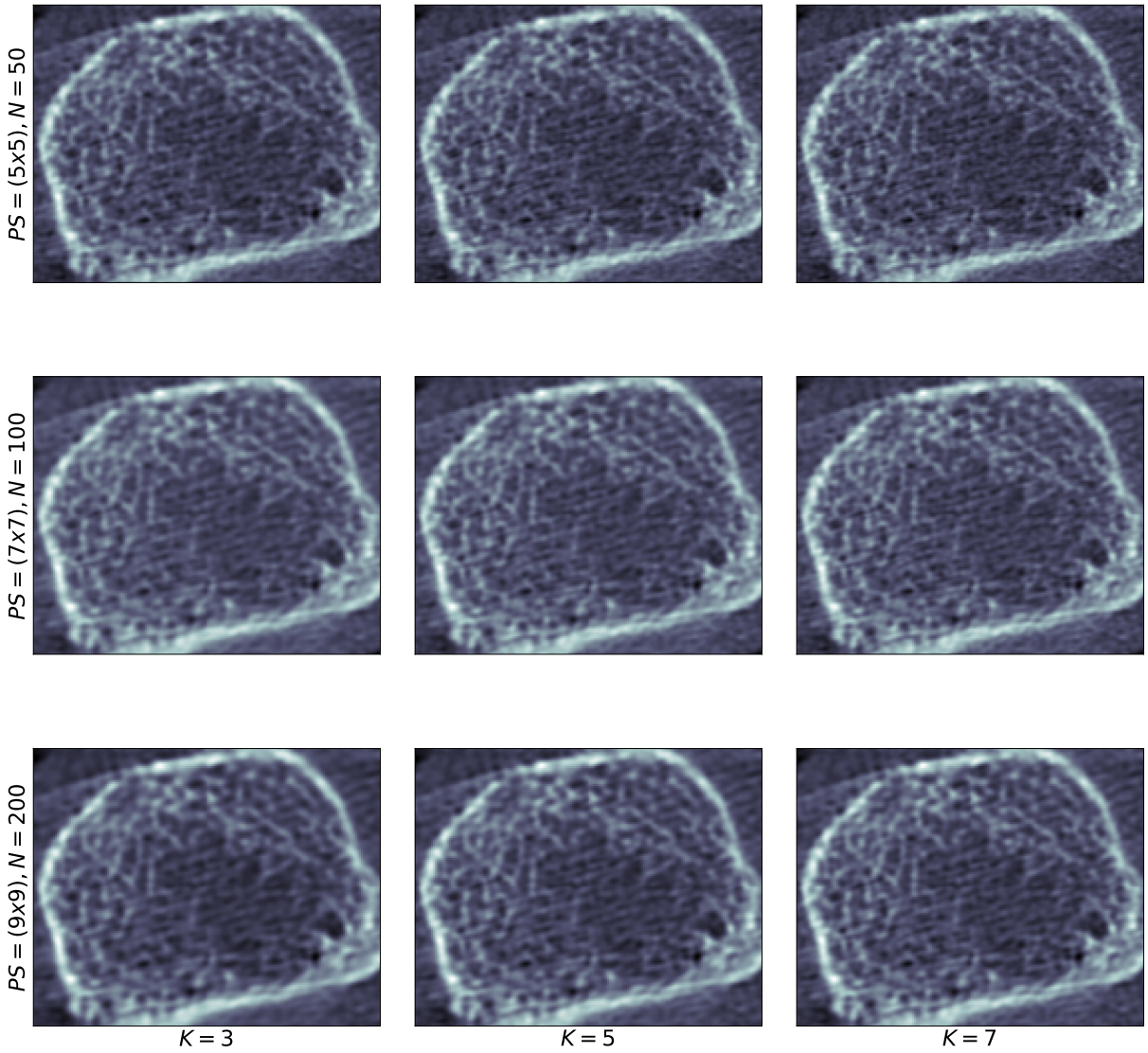

Figure 6.43: *Visual representation of reconstructions after cloud K-SVD from noise-filled version of medical image, at varying PS/M, N and K. $t_d = 10$, $t_p = 3$, $t_c = 5$, $P = 4$.*

Discussion

The purpose of these experiments was to test how well cloud K-SVD could denoise medical images. Because of hardware limitations however, we were only able to test on a small portion of a medical data in which the noise was embedded. We refrained from downscaling the image, since that would alter the noise structure. Results from the process are in figure 6.43 and unlike earlier tests, it is hard to verify whether cloud K-SVD has removed only noise or crucial details of the image. We clearly see that our experiment could not remove the ring artifact in the upper left corner. Ring artifacts were expected to be the hardest kind of noise to remove, since it has clear structure unlike white Gaussian noise. By observing the corners of the images, we also see a difference in the amount of Gaussian noise. The image with a patch size of (9×9) and a sparsity of $K = 3$ shows an overall improvement related to reducing Gaussian noise however at the cost of details in the rest of the image. By checking errors between the original and reconstructed image, see figure 6.42, we see that what was true for benchmark images, in section 6.3, is also true when using medical ones. We therefore believe that by tweaking the parameters of the algorithm and running on better hardware, cloud K-SVD would be able to improve these images even more.

Chapter 7

Discussion and conclusion

The chapter will present and discuss the lessons we learned during this master's thesis from mostly a practical standpoint, our conclusion and discuss eventual future work that naturally would follow our work. The objective of this thesis was to implement and document cloud K-SVD in a real distributed system for a good many types of data. In order to do so, we started by explaining fundamental concepts and mathematics that concerned cloud K-SVD before documenting how it actually worked, and investigating weather it was capable of denoising both benchmark and real medical images to an acceptable standard. A lot goes into the design and implementation of proper modern software, that should work in the cloud, be scalable under load and smoothly deploy on new hardware, hence we spent time discussing all relevant technologies and methods used to give the reader a better understanding of our work. Until now, we have presented our implementation of cloud K-SVD in our environment with the constrains we chose to set and enforce in order to present a tangible, transparent and applicable solution in the real world. Obviously, we did face challenges and made interesting observations, which the next section will discuss.

7.1 Lessons learned

We made some interesting discoveries and observations with the design, implementation and results of cloud K-SVD that are worth putting into words.

Compressed sensing in cloud K-SVD

By advice of Arora et al. in [85], any K-SVD algorithm should smartly initialize the dictionary $\boldsymbol{D}$ with i.i.d drawn measurements from a Gaussian distribution due to the same reasons as with the sensing matrix $\boldsymbol{\Phi}$ in compressed sensing, namely the restricted isometry property (RIP). When the dictionary $\boldsymbol{D}$ obeys the RIP, signal information is properly preserved because the dictionary ensures an isometric dimensionality mapping of all sparse signals. When the dictionary satisfies the RIP of order cK for some constant c and sparsity K, the RIP for a matrix $\boldsymbol{D}$ provides a guarantee that we can obtain successful K-sparse recovery of a signal using a variety of algorithms, like the OMP, and that the recovery process is stable in ℓ_2 under measurement noise [86]. We made a single observation with $\boldsymbol{D}$ initialized to fixed deterministic scalar values (for example just ones and twos) for all elements so that it basically had to start from scratch and saw inferior results as opposed to using Gaussian measurements. Thus in practice, we used compressed sensing theory when designing our experiments for cloud K-SVD.

The hunt for a global dictionary

The original cloud K-SVD algorithm was created with the purpose of reaching a global consensus among network nodes of the dictionary $\boldsymbol{D}$. To do so, cloud K-SVD was designed to use a consensus algorithm in the power method step that gathered error information from all other nodes before updating the dictionary with a largest error found between the node itself and all other nodes. We implemented the averaging consensus in order to keep it simple, which have turned out to be really effective in practical experiments. What we could gather from the original cloud K-SVD paper, is that they only assume the network graph is connected, not complete. Thus their configuration does not assume that all nodes can communicate directly, but rather have to ask random neighbors nodes to retrieve residual error data. In our test setup, we assume a complete node (pod) graph, so that every P pod can communicate with $P-1$ pods. In case the number of pods is $P = 16$, each single pod is able to communicate with 15 other pods. This is the single biggest reason for the amount of time it takes to complete a cloud K-SVD iteration compared to a local K-SVD one, but it ensures that all pods reach a consensus of the global dictionary. What was not tested in the experiments was fewer and random pod communication links, which we expect would make the execution faster but may deteriorate how quickly pod's reach consensus.

The denoising effect

Cloud K-SVD is a dictionary learning algorithm designed for image classification with distributed workload in mind. However since the dictionary learning part uses sparse approximation as a tool in the construction of a dictionary, we decided to experiment with one of the traditional features that sparse approximation algorithms have been used for in previous works, namely denoising. This results in a algorithm with a few minor modifications, now able to denoise natural images that have been distributed. Using cloud K-SVD for denoising, as documented in the experiments we have done, did prove success. We did hope to see a greater denoising effect on the medical CT images in our experiments, than what turned out to be the final result, however this thesis was always intended as a proof-of-concept about the effects of denoising. The algorithm was proven on noisy benchmark images with a tangible amount of training data available and have also shown the same effects on a medical image from a CT scanner.

Synchronization issues

When implementing cloud K-SVD, we did face a challenge with proper worker synchronization that is worth mentioning. Recall that the algorithm iteratively updates atoms N_i in dictionary D_i by fetching residual error vectors from neighboring peers, such that its own approximation of a signal and dictionary is swayed in direction of the global data Y rather than the local Y_i. The dictionary update step expects to pull residual vectors that correspond to the atom it is currently updating, but because worker pods operate on different cluster hardware nodes that are under unequal load, the amount of resources that are put at a pod's disposal is different from node to node. This meant that some cloud K-SVD iterations completed before others and that pods made disjointed updates to their local dictionary. So when pod p_i requests pod p_j's current residual vector for atom n_1, p_j could be at dictionary atom n_2 and return the residual of that instead! We saw two possible solutions to this: Either we had to force synchronization after each atom update or store the residual vectors per atom in the database of the individual worker pod. We chose the latter solution, because it allowed pods to continue execution even if they were ahead of the pact and no unnecessary synchronization messages had to sent or received.

HTTP, streaming and WebSocket

In the current design we used the HTTP/TCP protocol for all communications between pods, and the JSON data format to serialize payload data packets, however there exists alternative protocols, such as HTTP streaming and WebSocket, that deserve some attention here. With

traditional HTTP, we get a protocol that has broad support libraries (like AIOHTTP), supports asynchronous operations in a request-response style and abstracts a lot of complexity from our solution so that we can focus on business logic rather than data access logic. HTTP with TCP would in many eyes, including our own, be the tool-of-choice to establish reliable connection paths between a heterogeneous set of pods and exchange data in a connection-oriented way. Though the simplicity come at a cost that includes some message overhead per request/response, works only half-duplex (that is, data can only be sent one way at a time) and since cloud K-SVD relies on repeatably sending and receiving consensus data, any additional overhead can quickly pile up.

Another way of doing pod-to-pod communication is with HTTP streaming and WebSockets protocols, that both have less overhead than traditional HTTP. Streaming is a push-style data transfer method that allows the web server to continuously send data over a single HTTP connection, that stays open indefinitely and is only closed when the entire algorithm has completed. With HTTP streaming, the server holds on to the client connection and keeps the response channel open, so that it can push data through it. This way, the client can listen for updates and receive them instantly without opening or closing the connection each time, as we do now with traditional HTTP. Thus streaming reduces the number of HTTP header messages and eliminates the need for polling simply because it keeps the connection open. A possible disadvantage of streaming is the abundance of open connections we might see and if a pod goes down or a network partition occurs, connections would hang until closed preemptively.

WebSocket is an application layer protocol like HTTP (OSI model layer 7), but supports full duplex (pods can send and receive data simultaneously), uses a push-pull paradigm instead of just pull, supports data pipelining and behave like HTTP streaming when sending data. That is, both use an underlaying TCP/IP model to streamline payload and provide guarantees of transport reliability through flow control, proper segmentation/desegmentation of packets and error checks. WebSockets require more manual work to configure and maintain compared to HTTP REST services[1], but can reduce message overhead and allow pods to push new data as soon as it is ready. See figure 7.1 for a brief summary of HTTP and WebSocket.

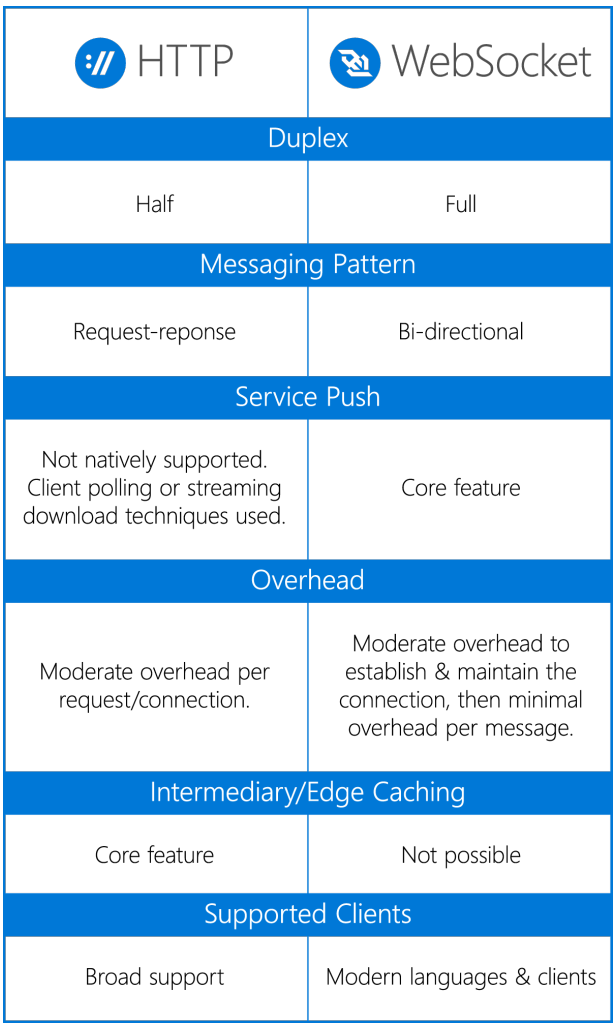

Figure 7.1: *The HTTP protocol and WebSocket head-to-head with some key differences that could benefit a solution with a lot of data requests and responses. Credits to the Windows Apps Team at Microsoft for the image.*

[1]Representational state transfer (REST), see: https://en.wikipedia.org/wiki/Representational_state_transfer

Memory use, downscaling and consensus

One particular downside of using IoT-hardware, with inherent resource constraints for something like cloud K-SVD, is that we had to constantly keep down the dimensionality of training signals to not cause memory problems on the nodes. Large data loads, for example a natural image in 400×640 resolution, could cause severe out-of-memory exceptions and segmentation faults on worker pods, so we had to carefully select which images we would use for patch learning and in some cases apply a downscaling factor α to make it palatable for the system. Unfortunately, downscaling is not really ideal here, because it actually blurs the image before any patch learning or reconstruction work is done. This can make it hard to distinguish between downscaling and cloud K-SVD effects when exhibiting the final result. One would think that creating more pods would mitigate this, because the training signals would then be further split up at the preprocessing level, however the physical memory on nodes does not increase or decrease with the number of pods and for each new worker pod, Kubernetes has to set aside a pool of resources for that new pod. The medical images were high in resolution as well, so we had to apply similar tactics and make compromises to even demonstrate some degree of denoising, which is a bit unfortunate. A way out is obviously getting better hardware and more memory, which we were sadly not able to this time around.

For all experiments, we used averaging consensus to produce an average at worker pod p_i for p_{i+1} residual vectors, so that p_i would steer its error in the direction of the general consensus among p_{i+1} pods. This method can be extended in a way so that the accumulated error is stored at individual pods and can be propagated throughout the network at some fixed interval. We call this corrective consensus and accounted for its details in section 3.7, however we chose not to re-evaluate our experiments with correct consensus, since we had already gotten sound results in terms of dictionary convergence among pods with just simply average consensus. Though if possible, we would definitively have made experiments to see if the error between dictionaries (how incoherent they are) would improve faster using corrective consensus instead.

Compiling Docker images for ARM

When building Docker images, the lengthy part of the process is building so called wheels (Python packages) for dependencies such as NUMPY, SCIPY and REDIS. This is especially true when compiling for the ARMv7 architecture, as in our case, since the main Python repositories do not contain precompiled versions of these libraries for ARMv7. It is a lot easier to get your hands on precompiled libraries for more popular architectures, like the AMD64, than it is for ARMv7. When building IoT-solutions with Docker, one should take the time to find precompiled wheels for medium to large libraries, for example NUMPY, as they drastically reduce the time it takes to build images.

Docker images and scaling

Because we bundle the application code and dependencies into Docker images (one for each type of pod), the entire application can be deployed anywhere on any Docker-supporting Kubernetes cluster, for example in at lab setup on more capable hardware or at a cloud provider like Amazon Web Services (AWS) or Google Cloud that provide a plug-and-in cluster for applications. The reason we did not use a cloud provider is mainly due to the costs associated with ordering and managing a cluster, even a small one, and since an IoT-platform like four Raspberry Pi's in a cluster with adequate memory and processing power is actually enough to make a sound demonstration of cloud K-SVD. Also the software industry, developers and system administrators are becoming increasingly attracted to cloud-technology and the paradigm behind it for building applications, so it makes sense to embrace such technologies (Docker and Kubernetes) when building any application today.

Kubernetes in general, IoT and K3S

Kubernetes is used as the management tool for orchestrating the pods in the cluster. Kubernetes did show to be surprisingly problematic to configure on an IoT-platform, when you factor in the mine of guides there exists on the Internet. We ran into problems with different Linux operating-systems and issues with K8S (the full Kubernetes software suite) that had trouble running on the Raspberry Pi's. By investigating and testing of several operating-systems and versions of Kubernetes, we wrote chapter 5, which takes a step back and looks at the pros and cons we saw with different operating-systems and versions. The real problem is that Kubernetes is still in experimental phase when it comes to IoT/Linux and it comes bundled with many redundant things as well, that you may not need for a small setup. We ended up finding a scaled-down version of Kubernetes called K3S. The 1.0 was released mid-project and targeted low-cost hardware, for such systems like a Raspberry Pi cluster. K3S did show itself as a very able Kubernetes version, since it is lightweight and easy to set up with a utility tool called K3SUP.

The overall impression of using Kubernetes is that, when it is already configured and working properly, it is excellent for software deployments, especially when you develop for a microservice architecture. However the hurdle it is to get to that point, where everything just works and runs flawless, without being an expert in Linux, is quite tedious and problematic. Especially if you are using low-cost hardware that K8S is not made for.

7.2 Conclusion

In this thesis, we have documented and described how to design, implement and test the dictionary learning algorithm cloud K-SVD on a real distributed computer system for solving sparse approximation and dictionary learning problems. For the implementation, we configured our own Kubernetes cluster on four Raspberry Pi nodes and deployed a total of three Docker images that made up our application. We showed with experiments that simple synthetic data can be used to train and verify the behavior of cloud K-SVD and that such data is enough to demonstrate convergence of heterogeneous local dictionaries to a common one. As a novelty, we then extended our experiments and demonstrated cloud K-SVD's ability to learn a dictionary from a mine of natural image patches in order to reconstruct both benchmark and real medical images that had been contaminated with noise. In general, we considered three variants of the algorithm (centralized K-SVD, local K-SVD and cloud K-SVD) to compare their strengths and weaknesses in a distributed setting. We had to make certain design choices when implementing and testing cloud K-SVD on a low-cost IoT-setup, which include setting runtime memory and processing power limitations on nodes and downscaling a portion of our test images to make it viable, however we were still able to demonstrate core aspects and applications of sparse approximation and dictionary learning with cloud K-SVD on real data. More work is needed to demonstrate viability with full-sized data sets, which necessarily has to be on a system with adequate resources, and also to explore other consensus algorithms and networks protocols that can help reduce the network communications costs in cloud K-SVD.

7.3 Future work

For future work, there exists many possible adaptations and changes that can be made to cloud K-SVD and our system. Here we will give some thoughts on possible improvements and elaborate on where to continue.

Communication protocols like HTTP streaming or WebSockets, mentioned in section 7.1, could have a potential impact on reducing the time each consensus step takes in cloud K-SVD over traditional HTTP. By streaming data over a single open connection, or by limiting some of the data transfer overhead, it could reduce network communication costs.

Consensus iterations have shown to be very a time consuming and expensive affair when every node (pod) needs to communicate with all its neighbors, as implemented with the averaging consensus protocol, detailed in section 3.7. Further investigation and experiments with consensus algorithms and neighbor communication could have a significant say in accelerating the consensus iteration part.

Sparse approximation is used to produce a signal matrix in cloud K-SVD. We implemented the greedy orthogonal matching pursuit (OMP) algorithm, detailed in section 3.5, which is a very effective and well-tested at making sparse approximations. There exists a different branch of algorithms, that differ from greedy, known as convex optimization methods, which should be evaluated via actual experiments as well to see if they where able to da better job than OMP.

Hardware limitations have been a recurring problem while testing on real images, which meant we had to downscale or crop several training images in our experiments. Moving to a more powerful hardware platform would allow testing of images in full resolution and evaluate the denoising effects on a bigger scale.

Classification is a hot topic in machine-learning today and is also what cloud K-SVD was originally designed for [9] [10]. Using cloud K-SVD to solve a classification problem may show new and interesting results when done in a real distributed setup.

Online dictionary learning updates node dictionaries sequentially in response to streaming data. In our solution, we flush all reminiscences of previous training data between experiments and we do not consider scenarios, where the dictionary is remembered and only improved when new training data becomes available. Online learning would be an interesting feature to add next and would not require that much extra work.

Nomenclature

Numbers

N	Quantity of elements in sparse representation of signal	$N \gg M$
M	Quantity of elements in non-sparse representation of signal	$M \ll N$
Q	Quantity of measurements	
K	Sparsity (quantity of non-zero elements)	
ϵ	Model deviation or error	
λ	Lagrange multiplier or eigenvalue	
H	Quantity of nodes in distributed setup	
P	Quantity of pods in distributed setup	
Q_i	Quantity of measurements for node i	$i = 1, 2, \ldots, H$

Vectors

$\boldsymbol{z}$	Measurement vector of single raw data	$\boldsymbol{z} \in \mathbb{R}^N$
$\boldsymbol{z}_i$	Single row or column of the matrix $\boldsymbol{Z}$, where i represents the entry	
$\boldsymbol{x}$	K-sparse signal vector	$\boldsymbol{x} \in \mathbb{R}^N$
$\hat{\boldsymbol{x}}$	Recovered/Reconstructed signal, estimate of original $\boldsymbol{x}$	$\hat{\boldsymbol{x}} \in \mathbb{R}^N$
$\boldsymbol{x}_i$	Single row or column of the matrix $\boldsymbol{X}$, where i represents the entry	
$\boldsymbol{y}$	Non-sparse representation of $\boldsymbol{x}$	$\boldsymbol{y} \in \mathbb{R}^M$
$\boldsymbol{y}_i$	Single row or column of the matrix $\boldsymbol{Y}$, where i represents the entry	
$\boldsymbol{d}$	Single row or column $\boldsymbol{D}$, often written as $\boldsymbol{d}_i$, where i represents the entry	
$\boldsymbol{\psi}$	Single row or column $\boldsymbol{\Psi}$, often written as $\boldsymbol{\psi}_i$, where i represents the entry	
$\boldsymbol{\phi}$	Single row or column $\boldsymbol{\Phi}$, often written as $\boldsymbol{\phi}_i$, where i represents the entry	
$\boldsymbol{q}$	The dominant eigenvector	

Matrices

$\boldsymbol{Z}$	Multi measurement vector of multiple $\boldsymbol{z}$-signals	$\boldsymbol{Z} \in \mathbb{R}^{N \times Q}$
$\boldsymbol{X}$	Multi measurement vector of multiple $\boldsymbol{x}$-signals	$\boldsymbol{X} \in \mathbb{R}^{N \times Q}$

$\hat{\boldsymbol{X}}$	Recovered/Reconstructed multi measurement vector, estimate of original $\boldsymbol{X}$	$\hat{\boldsymbol{X}} \in \mathbb{R}^{N \times Q}$
$\boldsymbol{X}_i$	Local multi measurement vector	$\boldsymbol{X} \in \mathbb{R}^{N \times Q_i}$
$\boldsymbol{Y}$	Multi measurement vector of multiple $\boldsymbol{y}$-signals	$\boldsymbol{Y} \in \mathbb{R}^{M \times Q}$
$\boldsymbol{Y}_i$	Local multi measurement vector	$\boldsymbol{Y} \in \mathbb{R}^{M \times Q_i}$
$\boldsymbol{D}$	The dictionary	$\boldsymbol{D} \in \mathbb{R}^{M \times N}$
$\boldsymbol{D}_i$	Local dictionary of pod i	$\boldsymbol{D} \in \mathbb{R}^{M \times N}$
$\boldsymbol{\Psi}$	The orthonormal basis	$\boldsymbol{\Psi} \in \mathbb{R}^{N \times N}$
$\boldsymbol{\Phi}$	The measurement matrix, also represented as a dictionary	$\boldsymbol{\Phi} \in \mathbb{R}^{M \times N}$
$\boldsymbol{E}$	The residual matrix, also known as the error matrix	
$\boldsymbol{M}$	Positive-semidefinite residual matrix	$\boldsymbol{M} = \boldsymbol{E}\boldsymbol{E}^T$
$\boldsymbol{U}$	Left unitary matrix of standard SVD	
$\boldsymbol{\Delta}$	Diagonal matrix of standard SVD	
$\boldsymbol{V}$	Right unitary matrix of standard SVD	
$\boldsymbol{W}$	Doubly stochastic weight matrix	
$\boldsymbol{P}$	Restriction operator	

Formula

ℓ_p	The p-norm
$\ell_{p,q}$	The mixed p,q-norm
ℓ_0	The quasinorm
ℓ_1	The Manhattan Distance or Taxicab norm
ℓ_2	The Euclidean norm
ℓ_∞	The infinity norm
rsupp(x)	The row-support of vector x, indexes with non-zero entries
$\mu(x)$	Coherence of matrix x
$\mu(x_1, x_2)$	Coherence between matrix x_1 and matrix x_2
$x^\dagger$	The conjugate transpose of matrix x

Other Symbols

$\mathbb{R}^x$	Vector set with x elements
$\mathbb{R}^{x_1 \times x_2}$	Matrix set with x_1 rows and x_2 columns

Bibliography

[1] Kevin Taylor-Sakyi. Big Data: Understanding Big Data. *CIM Magazine*, 11(1), 2016.

[2] Chencheng Li, Pan Zhou, Yingxue Zhou, Kaigui Bian, Tao Jiang, and Susanto Rahardja. Distributed Private Online Learning for Social Big Data Computing over Data Center Networks. *2016 IEEE International Conference on Communications, ICC 2016*, 2016.

[3] Alexander Bertrand and Marc Moonen. Distributed Adaptive Node-Specific Signal Estimation in Fully Connected Sensor Networds - Part I: Sequential Node Updating. *IEEE Transactions on Signal Processing*, 58(10):5277–5291, 2010.

[4] Alexander Bertrand and Marc Moonen. Distributed Adaptive Node-Specific Signal Estimation in Fully Connected Sensor Networks - Part II: Simultaneous and Asynchronous Node Updating. *IEEE Transactions on Signal Processing*, 58(10):5292–5306, 2010.

[5] Joel Sole, Rajan Joshi, Nguyen Nguyen, Tianying Ji, Marta Karczewicz, Gordon Clare, Félix Henry, and Alberto Duenas. Transform coefficient coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1765–1777, 2012.

[6] Gary J. Sullivan, Jens Rainer Ohm, Woo Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012.

[7] Jens Rainer Ohm, Gary J. Sullivan, Heiko Schwarz, Thiow Keng Tan, and Thomas Wiegand. Comparison of the coding efficiency of video coding standards-including high efficiency video coding (HEVC). *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1669–1684, 2012.

[8] Liquan Shen, Ping An, and Zhi Liu. Context-adaptive based CU processing for 3D-HEVC. *PLoS ONE*, 12(2):1–23, 2017.

[9] Haroon Raja and Waheed U. Bajwa. Cloud K-SVD: A Collaborative Dictionary Learning Algorithm for Big, Distributed Data. *IEEE Transactions on Signal Processing*, 64(1):173–188, 2016.

[10] Sakth Aleti, Augustus Chang, and Parth Parikh. An Introduction to Cloud K-SVD. page 5, 2014.

[11] Michal Aharon, Michael Elad, and Alfred Bruckstein. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation Michal. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, 2006.

[12] Gabriel Peyré. The Numerical Tours of Signal Processing - Advanced Computational Signal and Image Processing. *Signal Processing*, pages 1–15, 2010.

[13] Michael Elad and Michal Aharon. Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries. *IEEE Transactions on Image Processing*, 15(12):3736–3745, 2011.

[14] Jianshu Chen, Zaid J. Towfic, and Ali H. Sayed. Dictionary Learning Over Distributed Models. *IEEE Transactions on Signal Processing*, 63(4):1001–1016, 2015.

[15] Jianshu Chen, Zaid J. Towfic, and Ali H. Sayed. Online dictionary learning over distributed models. *2014 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP)*, pages 3902–3906, 2014.

[16] Stéphane Mallat. *A Wavelet Tour of Signal Processing: The Sparse Way*. Elsevier Inc., 2009.

[17] B Y H Nyquist. Certain Topics in Telegraph Transmission Theory. pages 617–644, 1949.

[18] Claude E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[19] Emmanuel J. Candès. Compressive sampling. *International Congress of Mathematicians, ICM 2006*, 3:1433–1452, 2006.

[20] Richard G. Baraniuk. Compressive sensing. *Handbook of Mathematical Methods in Imaging: Volume 1, Second Edition*, (July):205–256, 2015.

[21] E Candes, J Romberg, and Terence Tao. Stable Signal Recovery from Incomplete and Inaccurate Measurements. *Science*, 40698(8):1–15, 2005.

[22] Ayush Bhandari. Introduction to Sparse Approximation.

[23] Emmanuel Candes, Mark Rudelson, Terence Tao, and Roman Vershynin. Error Correction via Linear Programming. pages 668–681, 2010.

[24] Noam Shental, Amnon Amir, and Or Zuk. Identification of rare alleles and their carriers using compressed se(que)nsing. *Nucleic Acids Research*, 38(19):1–22, 2010.

[25] M Lustig, Jh Lee, Dl Donoho, and Jm Pauly. Faster Imaging with Randomly Perturbed, Undersampled Spirals and L1 Reconstruction. *Proceedings of the 13th ...*, page 50, 2005.

[26] Joel A. Tropp. Greed is Good: Algorithmic Results for Sparse Approximation. *IEEE Transactions on Information Theory*, 50(10):2231–2242, 2004.

[27] Ren Huamin, Pan Hong, Søren Ingvor Olsen, and Thomas B. Moeslund. Greedy vs. L1 Convex Optimization in Sparse Coding. 2015.

[28] Dror Baron, Marco F. Duarte, Michael B. Wakin, Shriram Sarvotham, and Richard G. Baraniuk. Distributed Compressive Sensing. *Chinese Journal of Sensors and Actuators*, 26(10):1446–1452, jan 2009.

[29] M.F. Duarte, S. Sarvotham, D. Baron, M.B. Wakin, and R.G. Baraniuk. Distributed Compressed Sensing of Jointly Sparse Signals. *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.*, pages 1537–1541, 2005.

[30] Qun Wang and Zhiwen Liu. A novel distributed compressed sensing algorithm for multichannel Electrocardiography signals. *Proceedings - 2011 4th International Conference on Biomedical Engineering and Informatics, BMEI 2011*, 2:607–611, 2011.

[31] Dennis Sundman, Saikat Chatterjee, and Mikael Skoglund. Parallel pursuit for distributed compressed sensing. *2013 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2013 - Proceedings*, pages 783–786, 2013.

[32] Donghao Wang, Jiangwen Wan, Junying Chen, and Qiang Zhang. An Online Dictionary Learning-Based Compressive Data Gathering Algorithm in Wireless Sensor Networks. *Sensors (Switzerland)*, 16(10), 2016.

[33] Jonathan G. Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of Historical Trends in the Electrical Efficiency of Computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011.

[34] Chip Walter. Kryder's law. *Scientific American*, 293(2):32–33, 2005.

[35] Gordon E. Moore. Cramming more components onto integrated circuits. *Journal of Integrated Design and Process Science*, 38(8), 1965.

[36] Symeon Chouvardas, Yannis Kopsinis, and Sergios Theodoridis. An online algorithm for distributed dictionary learning. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2015-Augus:3292–3296, 2015.

[37] Farhad Pourkamali Anaraki and Shannon M. Hughes. Compressive K-SVD. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5469–5473, 2013.

[38] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research*, 11:19–60, 2010.

[39] Richard Baraniuk, Mark A. Davenport, Marco F. Duarte, and Chinmay Hedge. An Introduction to Compressive Sensing. pages 1–112, 2011.

[40] E.J. Candes and M.B. Wakin. An Introduction To Compressive Sampling. *IEEE Signal Processing Magazine*, 25(2):21–30, 2008.

[41] Esa Ollila. Robust Simultaneous Sparse Approximation. *Modern Nonparametric, Robust and Multivariate Methods*, 2015.

[42] Yonina C. Eldar and Gitta. Kutyniok. *Compressed Sensing: Theory and Applications*. Cambridge University Press, 2012.

[43] Emmanuel J. Candes and Terence Tao. Decoding by Linear Programming. *IEEE Transactions on Information Theory*, 51(12):4203–4215, 2005.

[44] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[45] Richard Metzler and Hidegoro Nakano. Quasi-norm spaces. 1964.

[46] Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[47] David L. Donoho. For Most Large Underdetermined Systems of Equations, the Minimal L1-norm Near-Solution Approximates the Sparsest Near-Solution. *Communications on Pure and Applied Mathematics*, 59(7):907–934, 2006.

[48] Ronald A Devore. Deterministic constructions of compressed sensing matrices. 23:918–925, 2007.

[49] Piotr Indyk. Explicit constructions for compressed sensing of sparse signals. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, page 7, 2007.

[50] Matthew A. Herman and Thomas Strohmer. High-resolution radar via compressed sensing. *IEEE Transactions on Signal Processing*, 57(6):2275–2284, 2009.

[51] Joel A. Tropp. Norms of random submatrices and sparse approximation. *Comptes Rendus Mathematique*, 346(23-24):1271–1274, 2008.

[52] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. 58(1):267–288, 1996.

[53] Joel A. Tropp, Anna C. Gilbert, and Martin J. Strauss. Algorithms for simultaneous sparse approximation. Part I: Greedy pursuit. *Signal Processing*, 86(3):572–588, 2006.

[54] Joel A. Tropp. Algorithms for simultaneous sparse approximation. Part II: Convex relaxation. *Signal Processing*, 86(3):589–602, 2006.

[55] Kenneth Kreutz-Delgado, Joseph F Murray, Bhaskar D Rao, Kjersti Engan, Te-Won Lee, and Terrence J Sejnowski. Dictionary Learning Algorithms for Sparse Representation. 15(2), 2003.

[56] Michael S. Lewicki and Bruno A. Olshausen. Probabilistic framework for the adaptation and comparison of image codes. *Journal of the Optical Society of America A*, 16(7):1587, 1999.

[57] Dmitry M Malioutov, Müjdat Çetin, and Arthur C Smith. A Sparse Signal Reconstruction Perspective for Source Localization with Sensor Arrays. *Ieee Transactions on Signal Processing*, 53(8):3010–3022, 2003.

[58] Arthur E.C. Pece. The Problem of Sparse Image Coding. *Journal of Mathematical Imaging and Vision*, 17(2):89–108, 2002.

[59] B. K. Natarajan. Sparse approximate solutions to linear systems. 24(2):227–234, 1995.

[60] Stephane G. Mallat and Zhifeng Zhang. Matching Pursuits With Time-Frequency Dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[61] Tong Wu, Anand D. Sarwate, and Waheed U. Bajwa. Active Dictionary Learning for Image Representation. *Unmanned Systems Technology XVII*, 9468:946809, 2015.

[62] Emmanuel J. Candès and David L. Donoho. New Tight Frames of Curvelets and Optimal Representations of Objects with Piecewise C2 Singularities. *Communications on Pure and Applied Mathematics*, 57(2):219–266, 2004.

[63] G. Davis, S. Mallat, and M. Avellaneda. Adaptive Greedy Approximations. *Constructive Approximation*, 13(1):57–98, 1997.

[64] Ron Rubinstein, Alfred M. Bruckstein, and Michael Elad. Dictionaries for Sparse Representation Modeling. *Proceedings of the IEEE*, 98(6):1045–1057, 2010.

[65] Mahdi Ataee, Hadi Zayyani, Massoud Babaie-Zadeh, and Christian Jutten. Parametric dictionary learning using steepest descent. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 1978–1981, 2010.

[66] Dorina Thanou, David I. Shuman, and Pascal Frossard. Learning Parametric Dictionaries for Signals on Graphs. *IEEE Transactions on Signal Processing*, 62(15):3849–3862, 2014.

[67] Diego Ongaro and John Ousterhour. In Search of an Understandable Consensus Algorithm (Extended Version). page 18, 2014.

[68] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Consensus and Cooperation in Networked Multi-Agent Systems. *Proceedings of the IEEE*, 98(7):1354–1355, 2010.

[69] Reza Olfati-Saber and Richard M. Murray. Consensus Problems in Networks of Agents With Switching Topology and Time-Delays. *Automatic Control, IEEE Transactions on*, 49(9):1520–1533, 2004.

[70] Yin Chen, Roberto Tron, Andreas Terzis, and Rene Vidal. Corrective consensus with asymmetric wireless links. *Proceedings of the IEEE Conference on Decision and Control*, pages 6660–6665, 2011.

[71] Alireza Tahbaz-Salehi and Ali Jadbabaie. On Consensus Over Random Networks. *44th Annual Allerton Conference on Communication, Control, and Computing 2006*, 3:1315–1321, 2006.

[72] Yin Chen, Roberto Tron, Andreas Terzis, and Rene Vidal. Corrective consensus: Converging to the exact average. *Proceedings of the IEEE Conference on Decision and Control*, pages 1221–1228, 2010.

[73] Gene H. Golub and Charlese F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 2013.

[74] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. page 17, 1999.

[75] Márk; Jelasity, Geoffrey Canright, and Kenth Engø-Monsen. Asynchronous Distributed Power Iteration with Gossip-based Normalization. *Euro-Par 2007*, pages 514–525, 2007.

[76] Márk Jelasity, Rachid Guerraoui, Anne Marie Kermarrec, and Maarten Van Steen. The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3231:79–98, 2004.

[77] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-Based Aggregation in Large Dynamic Networks. *ACM Transactions on Computer Systems*, 23(3):219–252, 2005.

[78] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad. Orthogonal Matching Pursuit: Recursive Function Approximat ion with Applications to Wavelet Decomposition. *Conference Record of the Asilomar Conference on Signals, Systems & Computers*, 1:40–44, 1993.

[79] Haroon Raja and Waheed U. Bajwa. A convergence analysis of distributed dictionary learning based on the K-SVD algorithm. *IEEE International Symposium on Information Theory - Proceedings*, 2015-June:2186–2190, 2015.

[80] Christian Horsdal Gammelgaard. *Microservices in .NET Core with examples in Nancy*. Manning Publications Co., 2017.

[81] Marko Luksa. *Kubernetes in Action*. Manning Publications Co., 2018.

[82] Daniel Zoran and Yair Weiss. Scale invariance and noise in natural images. *Proceedings of the IEEE International Conference on Computer Vision*, pages 2209–2216, 2009.

[83] Josef S. Smolen, Daniel Aletaha, and Iain B. McInnes. Rheumatoid arthritis. *The Lancet*, 388(10055):2023–2038, 2016.

[84] F. Edward Boas and Dominik Fleischmann. CT artifacts: Causes and reduction techniques. *Imaging in Medicine*, 4(2):229–240, 2012.

[85] Sanjeev Arora, Rong Ge, and Ankur Moitra. New Algorithms for Learning Incoherent and Overcomplete Dictionaries. *Journal of Machine Learning Research*, 35:779–806, 2014.

[86] Tong Zhang. Sparse recovery with orthogonal matching pursuit under RIP. *IEEE Transactions on Information Theory*, 57(9):6215–6221, 2011.

[87] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 13(4):13, 2004.

Appendix A

Image and signal quality assessment metrics

The ℓ_2 norm of the difference between signals Y and $\hat{Y}$:

$$\ell_2 = \Big(\sum (Y - \hat{Y})^2 \Big)^{1/2} \tag{A.1}$$

The total mean square error (MSE) is defined as [65]:

$$MSE = \frac{1}{N} F(D) = \frac{1}{N} \parallel Y - DX \parallel_2^2 = \frac{1}{N} \sum_{r=1}^{N} \parallel y_r - Dx_r \parallel_2^2 \tag{A.2}$$

where N is the number of estimates and F is a cost function that can estimate the effectiveness of the dictionary to approximate the signal well in this example and is explicitly dependent on the dictionary matrix. For image recovery, we view the *MSE* as the error between an original noise-free $w \times h$ monochrome gray-scaled image I and its noisy approximation $\hat{I}$:

$$MSE = \frac{1}{wh} \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} [I(i,j) - \hat{I}(i,j)]^2 \tag{A.3}$$

We consider the peak signal-to-noise ratio (PSNR) as well for images which is a ratio between the maximum possible power of a signal and the power of noise that changes the accuracy of its representation measured in terms of the logarithmic decibel scale. Using the definition of *MSE* in A.3, the *PSNR* (in dB) is defined as:

$$PSNR = 10 \times \log_{10} \Big(\frac{MAX_I^2}{MSE} \Big) \tag{A.4}$$

where MAX_I is the maximum possible pixel value of the image, i.e. $MAX_I = 2^B - 1$ where B is bits per sample. For example if an image is represented using 8 bits per sample, the MAX_I is 255.

Lastly we consider the structural similarity (SSIM) metric, which is a perception-based model that calculates an index for the similarity between two images. That is, the *SSIM* an initial uncompressed reference image without distortion and a received second image that can be distorted. The difference with respect to the former metrics (MSE and PSNR) is that these approaches estimate absolute-error, while *SSIM* considers image degradation as perceived change in structural information, luminance masking and contrast masking terms[87]. Structural information relates to a perception that the amount of inter-dependency between pixels is strong when they are spatially close. Such dependencies contain important information about the structure of objects

in the image. Luminance masking means that image distortions are often less visible in bright regions, while contrast masking means that distortions appear less visible where there is a lot of activity in the image. The *SSIM* index is calculated on various windows of an image. The measure between two equal-sized windows a and b of the same width and height is defined as:

$$SSIM(a,b) = \frac{(2\mu_a\mu_b + c_1)(2\sigma_{ab} + c_2)}{(\mu_a^2 + \mu_y^2 + c_1)(\sigma_a^2 + \sigma_b^2 + c_2)} \tag{A.5}$$

where μ_a is the average of a, μ_b is the average of b, σ_a^2 is the variance of a, σ_b^2 is the variance of b, σ_{ab} is the covariance of a and b, $c_1 = (k_1 \times L)^2, c_2 = (k_2 \times L)^2$ are variables to stabilize the division and L is the dynamic range of the pixel values, same as MAX_I for *PSNR*. By default $k_1 = 0.01$ and $k_2 = 0.03$. More information about the index calculation can be find in a paper by Wang et al. in [87, chapter section III-B].

Appendix B

Configuring Kubernetes on a multi-node setup

This section will explain how we installed and configured our own Kubernetes cluster on four Raspberry Pi nodes. The purpose was to create a local distributed environment, that would emulate a real-world cloud setting and because the university provided a set of Raspberry Pi nodes free of charge. Thanks to excellent work by enthusiasts like Lucas Käldström, Kasper Nissen, Alex Ellis and many more with their respective projects, binaries and scripts already exist to deploy and setup a multi-node cluster in many ways. To start with, we used the following hardware in our setup:

Hardware used for the cluster:

- 4 × Raspberry Pi 4 Model B 4GB RAM
- 4 × Raspberry Pi official power supplies
- 3 × 16GB SD cards (for the worker nodes)
- 1 × 32GB SD card (for the master node)
- 1 × 5-Port 10/100/1000 Ethernet switch
- 4 × Ethernet cables
- 1 × Wireless Router
- A Windows computer with a Linux VM installed

The installation has been inspired primarily by such guides as "Will it cluster? K3S on your Raspberry Pi"[1] a blog guide by Alex Ellis and "k3sup (said 'ketchup')"[2] a GitHub repository for an installation package by Alex Ellis as well.

The hardware setup of the cluster can be seen in figure B.1. The four Raspberry Pi's are connected through the Ethernet switch, which then connects to the wireless router. This is then connected to the Internet and a regular Windows computer is able to connect locally with the Kubernetes cluster through a wireless connection to the router.

To start with, all Raspberry Pi's should be booted on an SD card with an operating-system installed that supports Kubernetes. Different OS's have been tested, but the one used in the final setup was a Raspbian Buster Lite version released in 2019-09-26. Raspbian Buster Lite is a barebone version of the Raspbian operating-system, which is the official Linux OS for Raspberry Pi[3].

[1] https://blog.alexellis.io/test-drive-k3s-on-raspberry-pi/
[2] https://github.com/alexellis/k3sup
[3] https://www.raspberrypi.org/downloads/raspbian/

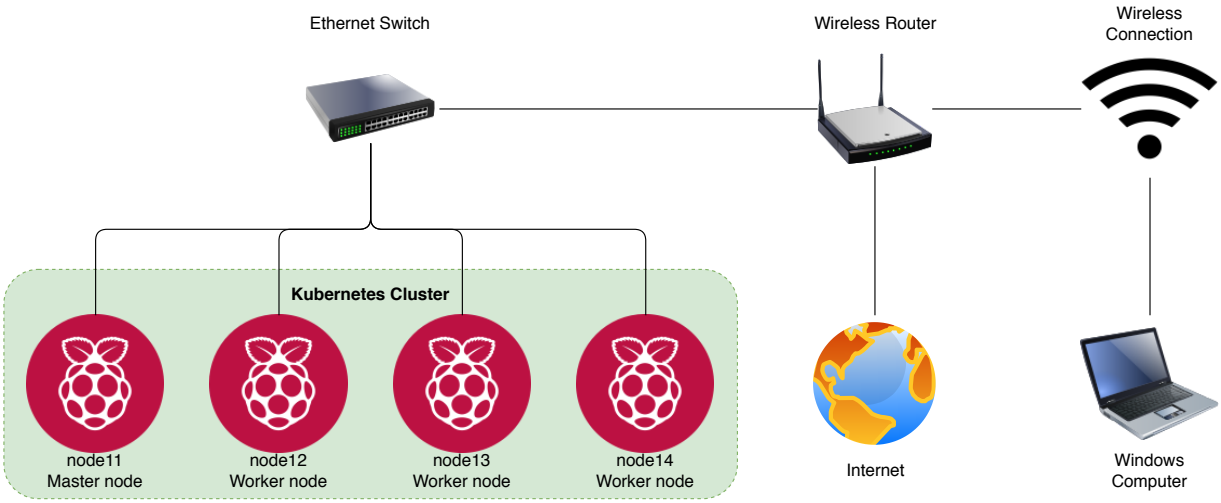

Figure B.1: *Kubernetes Cluster Hardware Setup.*

To flash the Raspbian Buster Lite OS onto the four SD cards, we used balenaEtcher[4] from a Windows computer. Before inserting the SD cards, an empty file named 'ssh' was created in the /Boot folder of the newly flashed SD cards. The SD cards were then unmounted and inserted into their respected Raspberry Pi's, the 32GB is inserted into the Raspberry Pi that is intended to become the *master* node in Kubernetes and the remaining three 16GB SD cards are inserted into the three Raspberry Pi's intended to become *worker* nodes. All Raspberry Pi's are then turned on.

We entered th each Raspberry Pi's via SSH to take note for its LAN IP-address. One can also connect a monitor to them and note the IP-address at startup. These are listed in the itemize below for completeness. Ideally, these IP's should be static:

- 192.168.1.151
- 192.168.1.152
- 192.168.1.153
- 192.168.1.154

To communicate over SSH, we used Putty[5] but any SSH connection tool will suffice like PowerShell. Log in with username "pi" and password "raspberry". Before configuring the Raspberry Pi's, we went to update their date and time. We did this semi-manually using the commands as shown in listing B.1. The timezone may vary depending on your location.

Listing B.1: *Raspberry Pi: Manual time and date setup.*

```
sudo cp /usr/share/zoneinfo/Europe/Copenhagen /etc/localtime

sudo date -s "$(wget -qSO- --max-redirect=0 google.com 2>&1 \
| grep Date: | cut -d' ' -f5-8)Z"
```

To change the Raspberry Pi's configuration, enter the command `sudo raspi-config` and configure the following:

- Set the GPU memory split to 16mb.

[4] https://www.balena.io/etcher/
[5] https://www.putty.org/

- Change the hostname to something different from each other, like node11, node12 etc.
- Change the password for the pi user (we used "dndmasters").
- Change the Internet location.

We now need to enable container features in the kernel, so enter the command `sudo nano /boot/cmdline.txt` and add the text in listing B.2 to the end of the line. Save and exit by using `Ctrl + X`.

Listing B.2: *Raspberry Pi: Setup cgroup.*

```
cgroup_enable=cpuset cgroup_memory=1 cgroup_enable=memory
```

The Raspberry Pi's are now configured and ready to be made into a Kubernetes cluster of one master and three worker nodes. We will not be needing a direct SSH connection to the nodes anymore, as we will be using a Linux virtual-machine installed locally on a Windows computer for the rest of this guide.

We decided to use K3S, which is a lightweight version of Kubernetes without Enterprise-grade packages and fewer dependencies. K3S is very useful for small setups and IoT-systems such as a Raspberry Pi cluster. To install K3S we used k3sup, which is a tool that uses SSH to install K3S on a remote Linux host. To be able to use k3sup, we needed to generate an SSH-key and distribute it to all nodes in the cluster, so that the k3sup tool can do its work. The commands to do so can be seen in listing B.3.

Listing B.3: *Linux VM: SSH-key installation on Raspberry Pi's.*

```
ssh-keygen

ssh-copy-id pi@192.168.1.151
ssh-copy-id pi@192.168.1.152
ssh-copy-id pi@192.168.1.153
ssh-copy-id pi@192.168.1.154
```

Now we are ready to install K3SUP on the Linux VM. Open a terminal and write the commands shown in listing B.4.

Listing B.4: *Linux VM: K3SUP setup on Linux.*

```
curl -sLS https://get.k3sup.dev | sh
sudo install k3sup /usr/local/bin/

k3sup --help
```

You should be presented with the different possible commands that can be used with K3SUP. What we are mostly interested in is the `install` and `join` commands. First create the master node by writing the `install` command like shown in listing B.5.

Listing B.5: *Linux VM: k3sup install to sertup master node.*

```
k3sup install --user pi --ip 192.168.1.151
```

When the command has been run you should be able to setup the worker nodes using the `join` command as shown in listing B.6.

Listing B.6: *Linux VM: k3sup join to sertup worker nodes.*

```
k3sup join --server-user pi --server-ip 192.168.1.151 \
--user pi --ip 192.168.1.152

k3sup join --server-user pi --server-ip 192.168.1.151 \
--user pi --ip 192.168.1.153

k3sup join --server-user pi --server-ip 192.168.1.151 \
--user pi --ip 192.168.1.154
```

After a couple of minutes the Raspberry Pi's should be up and running. To be able to use `kubectl` directly from the Linux VM, we can setup `kubectl` using the kubeconfig that was returned after the master node completed setup following `k3sup install`. The file may be protected which can give some problems when `kubectl` tries to use for example the config file. What we did was make a copy of the file named kube.config. This was done using `nano` to access the file and then save it with the name kube.config. Afterwards the commands shown in listing B.7 can be run to check whether the new configuration has been accepted and is working.

Listing B.7: *Linux VM: Setting up kubectl config file.*

```
export KUBECONFIG=/home/<your user>/kube.config

kubectl get node
```

If `kubectl` was installed correctly and K3S is working properly, the terminal should display all nodes of the Kubernetes cluster with their name, status, roles, age and version. The result of the command is in B.2.

```
bagger@ubuntu:~$ kubectl get node
NAME     STATUS   ROLES    AGE     VERSION
node14   Ready    worker   4d23h   v1.15.4-k3s.1
node13   Ready    worker   4d23h   v1.15.4-k3s.1
node11   Ready    master   4d23h   v1.15.4-k3s.1
node12   Ready    worker   4d23h   v1.15.4-k3s.1
```

Figure B.2: *Linux VM: `kubectl get node`.*

By running the command `kubectl get node -o wide` the terminal should display all nodes of the Kubernetes cluster with more information regarding internal and external IP's, OS-, kernel- and container-versions. The result of the command is in B.3.

```
bagger@ubuntu:~$ kubectl get node -o wide
NAME     STATUS   ROLES    AGE     VERSION         INTERNAL-IP     EXTERNAL-IP
node14   Ready    worker   4d23h   v1.15.4-k3s.1   192.168.1.114   <none>
node13   Ready    worker   4d23h   v1.15.4-k3s.1   192.168.1.113   <none>
node11   Ready    master   4d23h   v1.15.4-k3s.1   192.168.1.111   <none>
node12   Ready    worker   4d23h   v1.15.4-k3s.1   192.168.1.112   <none>

OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
```

Figure B.3: *Linux VM:* `kubectl get node -o wide`.

Since we are configuring an experimental setup for lab tests only, we promote the default service-account to administrator so that pods will be able to query all endpoints at the Kubernetes API for information, see listing B.8. However in an enterprise or production environment any service-account should have the least privileges assigned as possible and not be a member of the administrator role.

Listing B.8: *Admin role for default service account.*

```
kubectl create clusterrolebinding cluster-system-default \
--clusterrole=cluster-admin \
--user=system:serviceaccount:default:default
```

To connect from a Windows computer, it is possible to install `kubectl` to use in Windows Command Line. Do this by following the official guide[6] and setup the config file `/Users/<your user>/.kube/config`. The result can be seen in figure B.4 for our cluster.

```
C:\Users\bagge>kubectl get node
NAME     STATUS   ROLES    AGE    VERSION
node13   Ready    worker   5d3h   v1.15.4-k3s.1
node11   Ready    master   5d3h   v1.15.4-k3s.1
node12   Ready    worker   5d3h   v1.15.4-k3s.1
node14   Ready    worker   5d3h   v1.15.4-k3s.1

C:\Users\bagge>kubectl get node -o wide
NAME     STATUS   ROLES    AGE    VERSION         INTERNAL-IP     EXTERNAL-IP
node11   Ready    master   5d3h   v1.15.4-k3s.1   192.168.1.111   <none>
node12   Ready    worker   5d3h   v1.15.4-k3s.1   192.168.1.112   <none>
node14   Ready    worker   5d3h   v1.15.4-k3s.1   192.168.1.114   <none>
node13   Ready    worker   5d3h   v1.15.4-k3s.1   192.168.1.113   <none>
OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
Raspbian GNU/Linux 10 (buster)   4.19.75-v7+      containerd://1.2.8-k3s.1
```

Figure B.4: *Windows Command Line:* `kubectl get node` *and* `kubectl get node -o wide`.

[6] `https://kubernetes.io/docs/tasks/tools/install-kubectl`

Appendix C

Pod API interface

All pods communicate via REST[1] API interfaces, that define a set of actions and resources, that can be enabled or requested. We use Ethernet[2] as the network technology, TCP/IP[3] as the protocol stack, HTTP[4] as the application protocol for all messages sent and received between pods and JSON[5] as the data interchange format. The API interfaces are defined below per pod type (preprocessor, worker and postprocessor). All HTTP codes are HTTP response codes[6].

Action	**API**	**Description**	**Arguments**	**Returns**
POST	/load_data/	Receives a global dictionary, global training data and specification and distributes the data to all worker pods available.	Takes D, Y and a specification as body parameters.	Returns 200 if work was loaded correctly, otherwise returns 500.
POST	/start_work/	Takes and empty body and instructs all worker pods to start running cloud *K-SVD* at once.	None.	Returns 200 if all pods were started correctly, otherwise returns 500.

Table C.1: *API interfaces for the preprocessor pod template.*

[1]Representational state transfer: https://en.wikipedia.org/wiki/Representational_state_transfer
[2]Ethernet: https://en.wikipedia.org/wiki/Ethernet
[3]TCP/IP: https://en.wikipedia.org/wiki/Internet_protocol_suite
[4]HTTP: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[5]JSON: https://en.wikipedia.org/wiki/JSON
[6]HTTP response codes: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Action	API	Description	Arguments	Returns
POST	/load_data/	Receives a local dictionary, local training data and a specification from the preprocessor and stores everything in Redis.	Takes D_i, Y_i and a specification as body parameters.	Returns 200 if work was loaded correctly, otherwise returns 500.
POST	/start_work/	Spawns a task that runs cloud *K-SVD* with that latest loaded data and sets the local pod status to running.	None.	Returns 200 if work was started correctly, otherwise returns 500.
GET	/status/	Returns the current status of the worker pod.	None.	Returns 200 and a JSON response with the status.
GET	/qresidual/	Returns the local q_i for atom k (n) used in cloud *K-SVD*.	Takes a string k as request parameters.	Returns 200 and a JSON response with the residual if it was found, otherwise returns 404.
GET	/training_data/	Returns the currently loaded training data.	None.	Returns 200 and a JSON response with the training data if it was found, otherwise returns 404.

Table C.2: *API interfaces for the worker pod template.*

Action	API	Description	Arguments	Returns
POST	/save_results/	Receives a final D_i, Y_i, a list of all D_i and a list of all Y_i per iteration tp and a dictionary with statistics for the current run as results and saves everything in Redis.	Takes D_i, Y_i, a list of D_i, a list of Y_i and a dictionary with results as body parameters.	Returns 200 if results were saved correctly, otherwise returns 500.
GET	/get_result/	Returns the final D_i, the final Y_i, a list of all D_i and a list of all Y_i per iteration tp and a dictionary with statistics as the results.	None.	Returns 200 and a JSON response with the results, otherwise returns 404.

Table C.3: *API interfaces for the postprocessor pod template.*

Appendix D

Dockerfiles for Docker images

Listing D.1 shows the Dockerfile for the preprocessor image.

Listing D.1: *Dockerfile for preprocessor image.*

```
# Using official python runtime base image
FROM python:3.7.4

# Set the app directory
WORKDIR /usr/src/app

# Update packages and install libraries
RUN apt-get update && apt-get install -y python3-pip \
&& apt-get install -y python3-numpy && apt-get install -y libatlas
   -base-dev

# Upgrade pip
RUN pip install --upgrade pip

# Install numpy using precompiled wheels
RUN pip install --index-url=https://www.piwheels.org/simple/ \
NUMPY CCHARDET PYCARES CFFI MULTIDICT YARL HIREDIS PYYAML

# Install other requirements
RUN pip install AIOHTTP AIODNS AIOREDIS NEST_ASYNCIO KUBERNETES

# Copy all to current dir
COPY . .

# Make port 8080 available for webapi
EXPOSE 8081

# Define the command to be run when launching the container
CMD ["python","-u","./app.py"]
```

Listing D.2 shows the Dockerfile for the worker image.

Listing D.2: *Dockerfile for worker image.*

```
# Using official python runtime base image
FROM python:3.7.4

# Set the app directory
WORKDIR /usr/src/app

# Update packages and install libraries
RUN apt-get update && apt-get install -y python3-pip \
&& apt-get install -y python3-numpy && apt-get install -y libatlas
   -base-dev

# Upgrade pip
RUN pip install --upgrade pip

# Install using precompiled wheels
RUN pip install --index-url=https://www.piwheels.org/simple/ \
SCIPY CYTHON CCHARDET PYCARES CFFI MULTIDICT YARL HIREDIS PYYAML

# Install other requirements
RUN pip install AIOHTTP AIODNS AIOREDIS NEST_ASYNCIO KUBERNETES

# Copy all to current dir
COPY . .

# Run Cython
RUN python setup.py build_ext --inplace

# Make port 8080 available for webapi
EXPOSE 8080

# Define the command to be run when launching the container
CMD ["python","-u","./app.py"]
```

Listing D.3 shows the Dockerfile for the postprocessor image.

Listing D.3: *Dockerfile for postprocessor image.*

```
# Using official python runtime base image
FROM python:3.7.4

# Set the app directory
WORKDIR /usr/src/app

# Update packages and install libraries
RUN apt-get update && apt-get install -y python3-pip \
&& apt-get install -y python3-numpy && apt-get install -y libatlas
   -base-dev

# Upgrade pip
RUN pip install --upgrade pip

# Install numpy using precompiled wheels
RUN pip install --index-url=https://www.piwheels.org/simple/ \
NUMPY CCHARDET PYCARES CFFI MULTIDICT YARL HIREDIS PYYAML

# Install other requirements
RUN pip install AIOHTTP AIODNS AIOREDIS NEST_ASYNCIO KUBERNETES

# Copy all to current dir
COPY . .

# Make port 8082 available for webapi
EXPOSE 8082

# Define the command to be run when launching the container
CMD ["python","-u","./app.py"]
```

Appendix E

YAML deployment files for pods

Listing E.1 shows the deployment template file for the preprocessor.

Listing E.1: *YAML deployment template for the preprocessor*

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: preprocapi-app
spec:
selector:
matchLabels:
app: preprocapi-app
strategy:
type: RollingUpdate
replicas: 1
template:
metadata:
name: preprocapi-app
labels:
name: preprocapi-app
app: preprocapi-app
spec:
containers:
- name: preprocapi-container
image: thecml/sparse-preprocapi:1.0
imagePullPolicy: Always
env:
- name: APP_ENV
value: development
- name: PORT
value: "8081"
- name: PREPROC_WEB_PORT
value: "8081"
- name: WORKER_WEB_PORT
value: "8080"
- name: KUBERNETES_ENABLED
value: "1"
- name: WORKER_API_LABEL
```

```
value: "app=workerapi-app"
- name: DEBUG_MODE
value: "1"
ports:
- containerPort: 8081
name: http
protocol: TCP
```

Listing E.2 shows the deployment template file for the worker.

Listing E.2: *YAML deployment template for the worker*

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: workerapi-app
spec:
selector:
matchLabels:
app: workerapi-app
strategy:
type: RollingUpdate
replicas: 4
template:
metadata:
name: workerapi-app
labels:
name: workerapi-app
app: workerapi-app
spec:
containers:
- name: redis-container
image: redis
env:
- name: APP_ENV
value: development
- name: PORT
value: "6379"
ports:
- containerPort: 6379
name: http
protocol: TCP
- name: workerapi-container
image: thecml/sparse-workerapi:1.0
imagePullPolicy: Always
env:
- name: APP_ENV
value: development
- name: PORT
value: "8080"
```

```
- name: POD_IP
valueFrom:
fieldRef:
fieldPath: status.podIP
- name: WEB_PORT
value: "8080"
- name: KUBERNETES_ENABLED
value: "1"
- name: TIME_OUT
value: "1"
- name: WORKER_API_LABEL
value: "app=workerapi-app"
- name: DEBUG_MODE
value: "0"
- name: POSTPROC_WEB_PORT
value: "8082"
- name: POSTPROC_API_LABEL
value: "app=postprocapi-app"
ports:
- containerPort: 8080
name: http
protocol: TCP
resources:
requests:
memory: "506314Ki"
cpu: "500m"
limits:
memory: "1012629Ki"
cpu: "1000m"
```

Listing E.3 shows the deployment template file for the postprocessor.

Listing E.3: *YAML deployment template for the postprocessor*

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: postprocapi-app
spec:
selector:
matchLabels:
app: postprocapi-app
strategy:
type: RollingUpdate
replicas: 1
template:
metadata:
name: postprocapi-app
labels:
name: postprocapi-app
```

```
app: postprocapi-app
spec:
containers:
- name: redis-container
image: redis
env:
- name: APP_ENV
value: development
- name: PORT
value: "6379"
ports:
- containerPort: 6379
name: http
protocol: TCP
- name: postprocapi-container
image: thecml/sparse-postprocapi:1.0
imagePullPolicy: Always
env:
- name: APP_ENV
value: development
- name: PORT
value: "8082"
- name: POD_IP
valueFrom:
fieldRef:
fieldPath: status.podIP
- name: WEB_PORT
value: "8082"
- name: DEBUG_MODE
value: "1"
ports:
- containerPort: 8082
name: http
protocol: TCP
```

Appendix F

Algorithms

Algorithm 2: The K-SVD [46] [Chapter 12.2.3]

Input: Matrix $\boldsymbol{Y}$

Output: Dictionary $\boldsymbol{D}$.

Initialize $k = 0$ and initialize dictionary $\boldsymbol{D}_{(0)} \in \mathbb{R}^{M \times N}$, either by using random entries, or using N random chosen examples. `// Initialization`

Normalize the columns of $\boldsymbol{D}_{(0)}$. `// Normalization`

while *stopping criteria not satisfied* **do**

  Increment k by 1.

  Use a pursuit algorithm to approximate the solution of `// Sparse Approximation`

$$\hat{\boldsymbol{x}}_i = arg\min_{\boldsymbol{x}} \| \boldsymbol{y}_i - \boldsymbol{D}_{(k-1)}\boldsymbol{x} \|_2^2 \text{ subject to } \| \boldsymbol{x} \|_0 \leq k_0$$

  obtaining sparse representations $\hat{\boldsymbol{x}}_i$ for $1 \leq i \leq Q$ These form the matrix $\boldsymbol{X}_{(k)}$.

  for $j_0 \leftarrow 1$ **to** m **do** `// K-SVD Dictionary Update`

    Define the group of examples that use the atom d_j

$$\Omega_{j_0} = \{i \mid 1 \leq i \leq Q, \boldsymbol{X}_{(k)}[j_0, i] \neq 0\}$$

    Compute the residual matrix

$$\boldsymbol{E}_{j_0} = \boldsymbol{Y} - \sum_{j \neq j_0} \boldsymbol{d}_j \boldsymbol{x}_j^T$$

    where $\boldsymbol{x}_j$ are the j'th **rows** in the matrix $\boldsymbol{X}_{(k)}$.

    Restrict $\boldsymbol{E}_{j_0}$ by choosing only the columns corrosponding to Ω_{j_0}, and obtain $\boldsymbol{E}_{j_0}^R$.

    Apply SVD decomposition $\boldsymbol{E}_{j_0}^R = \boldsymbol{U}\Delta\boldsymbol{V}^T$. Update the dictionary atom $\boldsymbol{d}_{j_0} = \boldsymbol{u}_1$, and the representations by $\boldsymbol{x}_{j_0}^R = \Delta[1,1] \cdot \boldsymbol{v}$.

  end

  If change in $\| \boldsymbol{Y} - \boldsymbol{D}_{(k)}\boldsymbol{X}_{(k)} \|_F^2$ is small enough, stop. `// Stopping Criteria`

end

Result: The desired result dictionary $\boldsymbol{D} = \boldsymbol{D}_{(k)}$

Algorithm 3: The Method of Optimal Directions (MOD) [46][Chapter 12.2.2]

Input: Matrix $\boldsymbol{Y}$
Output: Dictionary $\boldsymbol{D}$.

Initialize $k = 0$ and initialize dictionary $\boldsymbol{D}_{(0)} \in \mathbb{R}^{M \times N}$, either by using random entries, or using N random chosen examples. `// Initialization`

Normalize the columns of $\boldsymbol{D}_{(0)}$. `// Normalization`

while *stopping criteria not satisfied* **do**

  Increment k by 1.

  Use a pursuit algorithm to approximate the solution of `// Sparse Approximation`

$$\hat{\boldsymbol{x}}_i = arg\min_{\boldsymbol{x}} \| \boldsymbol{y}_i - \boldsymbol{D}_{(k-1)}\boldsymbol{x} \|_2^2$$

  obtaining sparse representations $\hat{\boldsymbol{x}}_i$ for $1 \leq i \leq Q$ These form the matrix $\boldsymbol{X}_{(k)}$.

  Update the dictionary by the formula `// MOD Dictionary Update`

$$\boldsymbol{D}_{(k)} = arg\min_{\boldsymbol{D}} \| \boldsymbol{Y} - \boldsymbol{D}\boldsymbol{X}_{(k)} \|_F^2 = \boldsymbol{Y}\boldsymbol{X}_{(k)}^T(\boldsymbol{X}_{(k)}\boldsymbol{X}_k^T)^{-1} = \boldsymbol{Y}\boldsymbol{X}_{(k)}^\dagger$$

  If change in $\| \boldsymbol{Y} - \boldsymbol{D}_{(k)}\boldsymbol{X}_{(k)} \|_F^2$ is small enough, stop. `// Stopping Criteria`

end

Result: The desired result dictionary $\boldsymbol{D} = \boldsymbol{D}_{(k)}$

Algorithm 4: The Orthogonal Matching Pursuit (OMP)[39][Chapter 5.3.3]

Input: Dictionary/Measurement matrix $\boldsymbol{D}$ and signal measurement $\boldsymbol{y}$
Output: Sparse representation $\hat{\boldsymbol{x}}$.

Initialize $\hat{\boldsymbol{\theta}}_0 = 0$, $\boldsymbol{r} = \boldsymbol{y}$, $\Omega = \emptyset$, $i = 0$ `// Initialization`

while *stopping criteria not satisfied* **do**

  $i \leftarrow i + 1$

  $\boldsymbol{b} \leftarrow \boldsymbol{D}^T\boldsymbol{r}$ `// Form residual signal estimate`

  $\boldsymbol{\Omega} \leftarrow \boldsymbol{\Omega} \cup \text{supp}(T(\boldsymbol{b}, 1))$ `// Add index of residual's largest magnitude`

  $\hat{\boldsymbol{x}}_i|_\Omega \leftarrow \boldsymbol{D}_\Omega^\dagger\boldsymbol{x}, \boldsymbol{x}_i|_\Omega^C \leftarrow 0$ `// Form signal estimate`

  $\boldsymbol{r} \leftarrow \boldsymbol{y} - \boldsymbol{D}\hat{\boldsymbol{x}}_i$ `// Update measurement residual`

  If residual $\boldsymbol{r}$ is small enough or $\hat{\boldsymbol{x}}$ is K-sparse, stop. `// Stopping criteria`

end

Result: The desired approximate $\hat{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{x}}_i$