Matrix and tensor methods for dictionary learning for sparse representations

Špigel, Fran

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:148465

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Fran Špigel

MATRIX AND TENSOR METHODS FOR DICTIONARY LEARNING FOR SPARSE REPRESENTATIONS

Diplomski rad

Voditelj rada:
Zlatko Drmač

Zagreb, Travanj, 2021

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Roditeljima i majstorima

Contents

Contents iv

Introduction 1

1 Sparse Representations 2
- 1.1 Sparse Recovery . . . 2
- 1.2 Discrete Fourier Transform . . . 6
- 1.3 Discrete Cosine Transform (DCT) . . . 8
- 1.4 Wavelet Transform . . . 12
- 1.5 Sparse Coding . . . 13
- 1.6 Classification . . . 17
- 1.7 Compressed Sensing . . . 21

2 Dictionary Learning 24
- 2.1 K-Means Clustering . . . 24
- 2.2 Singular Value Decomposition . . . 29
- 2.3 K-SVD Algorithm . . . 31
- 2.4 Stochastic Gradient Descent . . . 36
- 2.5 Principal Component Analysis . . . 39

3 Tensor-based methods 41
- 3.1 Overview of tensor arithmetic . . . 41
- 3.2 Generalized Tensor Compressive Sensing . . . 42
- 3.3 Tensor-Based Dictionary Learning . . . 44
- 3.4 Multilinear Principal Component Analysis . . . 45

Bibliography 55

Introduction

Sparse representations are an area of study with significant applications in data compression, classification and even transformation (such as removing dead pixels from an image). In this thesis, we will construct an overview of some methods for representing visual signals sparsely, as well as test their applications on different data sets. We will explore the applications and some common problems with sparse representations, as well as their solutions. We will also include Python code for some of the algorithms discussed in the paper.

Chapter 1 covers mechanisms for obtaining sparse representations - in other words, finding sparse vectors a which solve expressions of the form $x = Ba$ given some signal vector x and a fixed $N \times K$ dictionary matrix B. We will focus on visual signals (images), but the results and processes described herein have a wide range of applications. We will show the compressive potential of Fourier and cosine transformations, cover the Matching Pursuit and Basis pursuit algorithms, as well as explore the use of sparse representations in classification tasks.

Chapter 2 covers how to not only obtain the sparse representation a, but also optimize the dictionary B, in order to get even sparser or more accurate representations. We will go over a a few algorithms for the k-means clustering problem, recall some basics of singular-value decomposition, and finally review the K-SVD algorithm for dictionary learning.

Finally chapter 3 generalizes the discussed methods to the context of tensors, covering tensor-based compressed sensing as well as tensor dictionary optimization.

Chapter 1

Sparse Representations

Sometimes, when observing naturally produced signals, it can be intuited that they exist in a subspace of the set of all possible signals. More specifically, let our signals have the form $x = (x^1, x^2, \ldots, x^N)^\top$, and we are presented with P signals, arranged in a matrix X such that each column represents one signal. These could, for example, be black-and-white image matrices, each unravelled into a vector of length N. Suppose that each pixel on the image could only assume the value 0 or 1. With a 16×16 pixel image, that means 2^{256} possible configurations. However, only a tiny fraction of those would reasonably represent something like a cat, or a dog, or a human face. So the question we naturally ask ourselves is - what is the 'natural image' subspace of the 'pixel configuration' vector space?

1.1 Sparse Recovery

Our assumption is that there is some set $B = [b_1, b_2, \ldots, b_K]$, where $K \ll N$, such that all natural 16×16 images exist within the linear span of B. This would imply that we can find representations a_i such that for each signal x_i, $x_i = Ba_i, i = 1, \ldots, P$. However, we will show that, numerically, it is simpler to generate an *overcomplete* dictionary, where $K \gg N$, but impose a sparsity requirement on the representation vectors. And so our problem can be formulated as:

$$\min_{B,A} \|X - BA\|_F, \quad \|a_i\|_0 \leq S, \quad i = 1, 2, \ldots, P, \tag{1.1}$$

where $X = [x_1, x_2, \ldots, x_P]$ is an $N \times P$ matrix of P N-dimensional signals, such as image vectors or sound waves, $A = [a_1, a_2, \ldots, a_P]$ is a $K \times P$ matrix of P K-dimensional representation vectors, and $B = [b_1, b_2, \ldots, b_K]$ is an $N \times K$ overcomplete dictionary (sometimes called a code book), with each dictionary atom b_i having unit length. The first term is a measure of the error in our approximation, and the last term is the sparsity requirement

imposed on each representation vector, where $S \ll N$ - in other words, all representation vectors are S-sparse. The sparsity mapping, sometimes called a 'pseudo-norm', is defined as the limit of the ℓ_p norm as p approaches 0. In short, it simply counts the number of nonzero elements of a vector. It is technically not a norm (since it does not satisfy the homogeneity requirement), however we will be using the notation $\|\cdot\|_0$ to signify this mapping, as it is a natural continuation of the well-defined ℓ_p family of norms.

Expression (1.1) is actually composed of several significant sub-problems. One of them is more fundamental: for a given signal vector x, and a fixed dictionary B, find its individual representation $a = (a^1, \ldots, a^K)^\top$ which optimizes the following:

$$\min_a \|x - Ba\|_2 , \quad \|a\|_0 \leq S. \tag{1.2}$$

In other words, "minimize a cost function with a strict sparsity constraint imposed on the solution space". There is another, similar approach to (1.2):

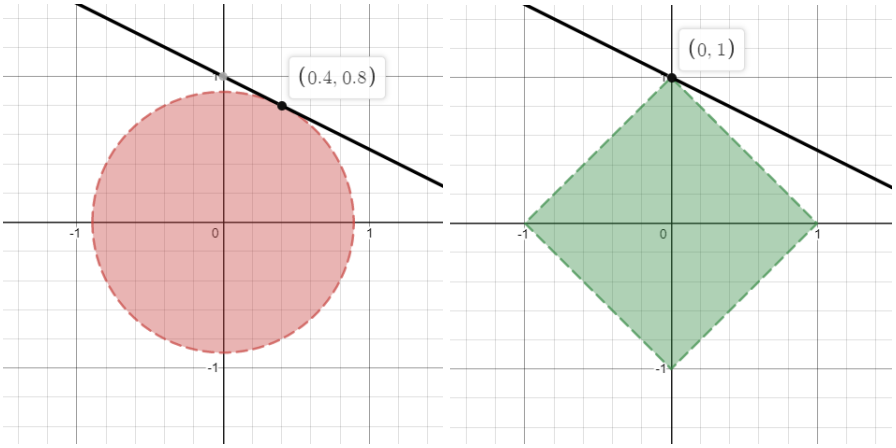
$$\min_a \|a\|_0 , \quad x = Ba, \tag{1.3}$$

"Find the sparsest solution to an under-determined system of linear equations". (1.2) and (1.3) are not exactly the same problem, but their solution spaces overlap significantly, and they often have the same minimum, so they are both worth considering. Additionally, they are dealing with a similar issue - the problem with (1.2) is that the set of all S-sparse vectors is not convex, so many of the standard approaches to solving it numerically are inadequate. (1.3) has the issue of its error function being discontinuous, so again most optimization methods will not work. This makes both of these problems NP-hard ([18]). To solve this issue, we will generally substitute the sparsity mapping in these problems with the ℓ_1 norm. The ℓ_2 norm would not suit our purposes - given a particular solution space, which is an $(K - N)$-dimensional manifold in K dimensions, ℓ_2 norm would simply find the solution which is closest to the origin, and that does not guarantee any sparsity. $\|\cdot\|_0$ retrieves the intersections of the manifold with basis axes (or at least, axis-spanned subspaces), and ℓ_1 norm comes close to approximating that behavior.

We can illustrate this point in 2 dimensions, using 1-dimensional solution manifolds, as is the case in fig. 1.1: if our solution manifold were to take the shape of a 1-dimensional line displaced from the origin in 2-dimensional space, we can see that minimizing with respect to ℓ_2 norm yields the solution $(0.4, 0.8)$, which is the closest to the origin, but minimizing with respect to ℓ_1 yields $(0, 1)$ which is the *sparsest* one. Additionally, we can see by intuition that minimizing any 1-d manifold in 2-d space will yield either a sparse solution (one that lies on an axis), or that the manifold will coincide perfectly with one side of the colored square, in which case sparse solutions will at least be minimums, even if they are not unique minimums.

And so we re-formulate (1.3) into a very similar, but much more solvable version:

Figure 1.1: Minimal solution w.r.t. ℓ_2 norm (left) and ℓ_1 norm (right)



$$\min_{\tilde{a}} \|\tilde{a}\|_1 , \quad x = Ba \tag{1.4}$$

There is an important result that guarantees a solution for the above expression.

Definition 1.1.1. *An $m \times n$ complex matrix A is said to have the null-space property of order S (NSP_S) if, for all index sets I with $S = |I| \leq n$ we have that*

$$\forall \eta \in Ker(A) \setminus \{0\}, \quad \|\eta_I\|_1 < \|\eta_{I^C}\|_1 ,$$

where

$$\eta_{I_i} = \begin{cases} \eta_i, & i \in I \\ 0, & i \notin I, \end{cases}$$

and I^C is the complement of I.

Theorem 1.1.2. *Let B be a $m \times n$ complex matrix. Then every S-sparse signal $a \in \mathbb{C}^n$ is the unique solution to the ℓ_1-relaxation problem:*

$$\min_{\tilde{a}} \|\tilde{a}\|_1 : B\tilde{a} = Ba \tag{1.5}$$

if and only if A satsfies the nullspace property with order S.

Proof. First, assume that the first statement holds true, and let η_I be some arbitrary element of Ker(A). Note that $\eta = \eta_I + \eta_{I_C}$, so by linearity we have

$$A(-\eta_{I^C}) = A(\eta_I - \eta) = A(\eta_I) - A(\eta) = A(\eta_I).$$

Since η_I is the unique solution to (1.5), it follows that $\|\eta_I\|_1 < \|\eta_{I^C}\|_1$.

Conversely, assume that B has the nullspace property, and let a be S-sparse and b another, not necessarily S-sparse, vector such that $a \neq z$ and $Bb = Ba$. Define the nonzero vector $\eta = a - b$. Note that $\eta \in Ker(A)$. Let I be the support of a. Then, by the triangle inequality,

$$\|a\|_1 \leq \|a - b_I\|_1 + \|b_I\|_1 = \|\eta_I\|_1 + \|b_I\|_1 < \|\eta_{I^C}\|_1 + \|b_I\|_1 = \| - b_{I^C}\|_1 + \|b_I\|_1 = \|b\|_1.$$

□

Unfortunately, testing NSP is NP-hard ([17]). Fortunately, there is a similar property with the same implication:

Definition 1.1.3. *Let $B = [b_1, \ldots, b_k]$ be an $n \times k$ matrix. For every integer $1 \leq S \leq k$, we define the S-*restricted isometry constants *δ_S to be the smallest quantity such that B_I obeys*

$$(1 - \delta_S)\|c\|^2 \leq \|B_I c\|^2 \leq (1 + \delta_S)\|c\|^2,$$

*for all index sets I such that $|I| \leq S$ and all real coefficients $(c_j)_{j \in I}$. Matrix B is also said to satisfy the S-*restricted isometry property *Similarly, we define the S, S'-restricted orthogonality constants $\theta_{S,S'}$ for $S + S' \leq |I|$ to be the smallest quantity such that*

$$\left|\langle B_{I^C}, B_{I'^{C'}} \rangle\right| \theta_{S,S'} \cdot \|c\|\|c'\|$$

holds for all disjoint index sets I, I' such that $|I| \leq S$, $|I'| \leq S$.

Theorem 1.1.4. *Suppose $S \geq 1$ is such that*

$$\delta_S + \theta_S + \theta_{S,2S} < 1,$$

and let a be a real vector supported on a set I with $|I| \leq S$. Let $x := Ba$. Then a is the unique minimizer to

$$\min \|d\|_{\ell_1}, \quad Bd = x.$$

The proof may be found in [4]. Even though the restricted isometry property is also NP-hard to test ([1]), random Gaussian, Bernoulli and partial Fourier matrices have been shown to satisfy it with exponentially high probability ([9]).

Yet another reformulation of (1.1) is sometimes used, where the sparsity restraint is relaxed, and converted into an error factor for the cost function:

$$\min_{B,A} \|X - BA\|_F + \lambda \sum_{i=1}^{P} \|a_i\|_1 , \tag{1.6}$$

where λ is some arbitrary coefficient that governs the importance of sparsity in the solution.

If a signal domain can be sparsely represented ('sparsified'), the first logical application arises naturally in data compression. If a set of N-dimensional vectors can be adequately represented using only S variables, we can immediately reduce storage space requirements by approximately a factor of $\frac{1-S}{N}$ (e.g. if $N = 100$, $S = 20$, we can store the data using 80% less space). Another application can be found in solving classification problems. More on that in section 1.6.

Several signal domains have already been thoroughly analyzed and sparsified. One such domain is visual signals, or images represented as pixel tensors. We will describe here several methods for sparsifying pixel data: discrete cosine transforms, wavelet transforms and the ubiquitous discrete Fourier transform.

1.2 Discrete Fourier Transform

The Fourier Transform is a method for decomposing a function f into a set of underlying frequencies represented therein - it is a spectral decomposition:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx. \tag{1.7}$$

Here, $\hat{f}(\xi)$ tells us how strongly the frequency ξ is represented in the function f. Once we obtain the Fourier decomposition, we can losslessly reconstruct the original function, using the inverse Fourier transform:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i x \xi} d\xi.$$

If we discretize the functions in question, such that $f(x_i) = f_i; \hat{f}(\xi_i) = \hat{f}_i$, we get the Discrete Fourier Transform (DFT) and its inverse:

$$\hat{f}_k = \sum_{n=0}^{N-1} f_n \cdot e^{\frac{-i2\pi}{N} kn} \tag{1.8}$$

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} \hat{f}_k \cdot e^{\frac{i2\pi}{N} kn}. \tag{1.9}$$

Two more steps are necessary in order to cement the DFT as a cornerstone of modern telecommunications technology. As it stands, the DFT can be re-written in matrix form, using a Vandermonde matrix:

$$\hat{f} = Ff$$

$$\begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_N & \omega_N^2 & \dots & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & \dots & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \dots & \omega_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{bmatrix} \tag{1.10}$$

$$f = \frac{1}{N} F^* \hat{f}$$

$$\omega_N = e^{-i2\pi/N}.$$

Finally, this formulation leads to the crucial method of performing DFT: the Fast Fourier Transform.

Fast Fourier Transform

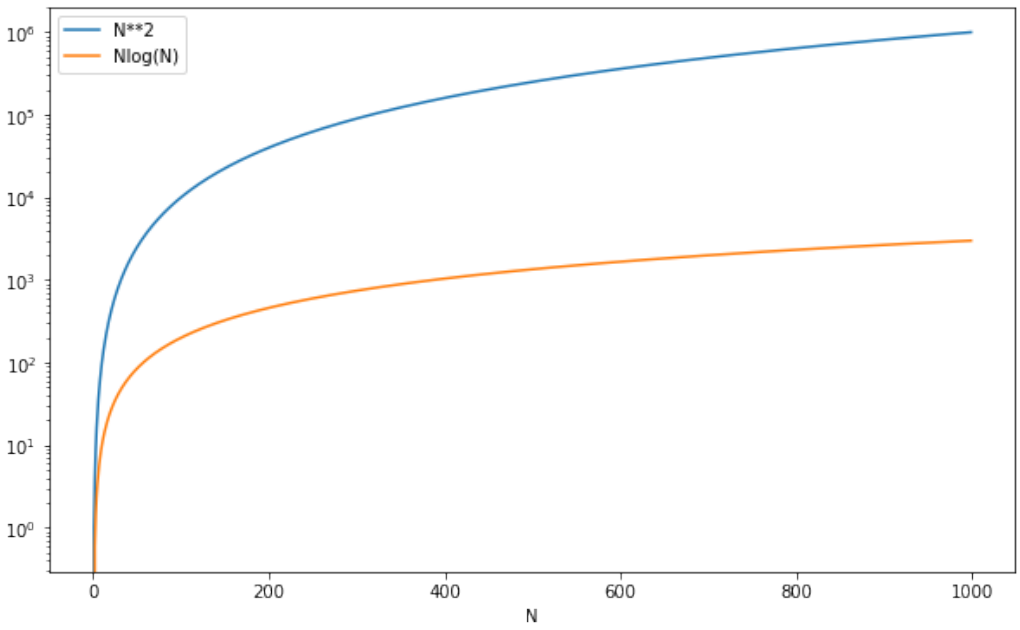
A matrix-vector multiplication, using a full matrix such as the one in (1.2) has $O(N^2)$ complexity, because N^2 multiplications need to be completed to calculate the result. This causes problems if we try to apply it to truly large problems, where N can be in the millions or even billions. In 1965, J. W. Cooley and John Tukey published a divide-and-conquer algorithm which reduces this complexity to only $O(N \log(N))$. Fig. 1.2 illustrates the difference in computation time with this kind of reduction in complexity. We can see that even with relatively light-weight data with $N = 1,000$, FFT represents a 300-fold decrease in time requirement. With $N = 1,000,000$, the decrease factor would be over 10^5.

FFT makes the DFT not only highly accurate, but also computationally cheap, which why it is now ubiquitous in modern telecommunications.

DFT-based compression and other applications

DFT is a fundamentally important result in numerical mathematics. It sees uses in a variety of applications - from data compression to solving partial differential equations. Its use in data compression is very easy to see - since usually only low frequencies are present in signals, a signal can be spectrally decomposed, and all high frequencies discarded without major data loss. Then this truncated data vector may be stored or transported, and the inverse DFT is applied when the signal needs to be analyzed in its natural form. Alternatively, many operations can be executed in the spectral domain itself, which often means sparse variables, making calculations faster. To illustrate the compressive potential of DFT,

Figure 1.2: A logarithmic scale representation of the difference in computation time between a DFT implementation using matrix multiplication and the FFT



we can look at fig. 1.3, and see that even with 95% of the frequency values truncated (discarded), the image is essentially indistinguishable from the original. In practice, instead of using a cutoff like that, we would simply truncate everything outside of a small circle or square in the center of the coefficients, since that is easier to communicate to the receiver of our signal.

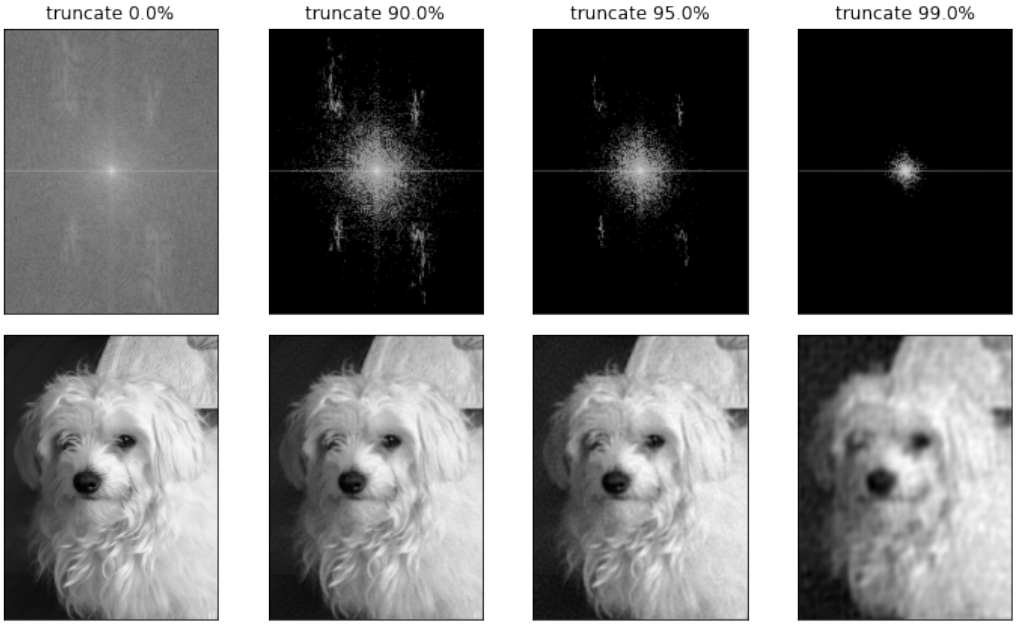
1.3 Discrete Cosine Transform (DCT)

The DCT is based on the Fourier Transform - we assume that a signal is periodic in nature, and attempt to replicate it using a basis of cosine waves with different periodicities. With 1-dimensional signals, our basis might look something like this:

$$\tilde{b}_i : \mathbb{R} \longrightarrow [-C_i, C_i]$$
$$\tilde{b}_i(x) := C_i \cos(\pi x i),$$

where C_i is a normalizing factor such that $\|\tilde{b}_i\| = 1, i = 1, \ldots, K$, regardless of the space being considered. This function would naturally be quantized into an N-dimensional, normalized vector with:

Figure 1.3: DFT-compressed picture of a dog. The top row shows a logarithmic representation of the corresponding frequency magnitudes (DFT coefficients), with some fraction of them being truncated to 0 (each coefficient with absolute value smaller than some cutoff point). The bottom row represents the reconstructed image using the inverse DFT operation.



$$\bar{b}_i^j := \tilde{b}_i(jh), j = 0, 1, \ldots, N$$
$$b_i := \frac{b_i^*}{\|b_i^*\|_2},$$

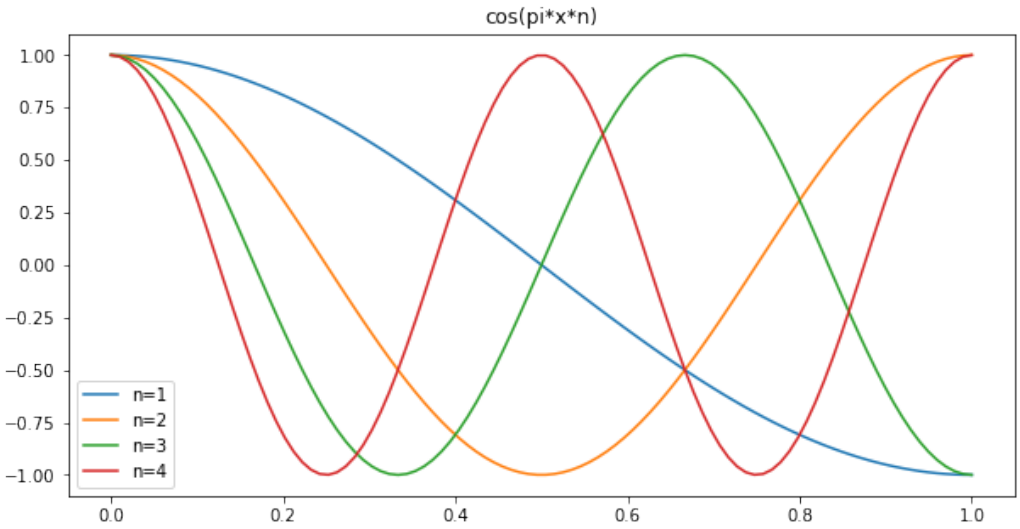
where h is some discretization factor used to quantize a domain such as $[0, 1]$

Arranging b_i as columns into a matrix, we would obtain a dictionary B. Then, given a signal x, we would employ sparse coding in order to find coefficients $a = a^1, \ldots, a^k$ s.t. $x = Ba$. More on exactly how to obtain these coefficients in section 1.5. Fig. 1.4 illustrates some bases that might be used in 1-d DCT (before normalization).

2D-DCT

Images, however, are inherently 2-dimensional signals, so the question of how to employ these 1-dimensional bases to represent them presents itself naturally, and the answer is

Figure 1.4: Some basis functions for 1-dimensional DCT before normalization



elegant in its simplicity. First of all, we cannot cheat by reshaping the 2-d matrix into a vector, because this creates a lot of discontinuities in the signal, something that DCT is not good at dealing with. Instead, we have to create 2-dimensional basis vectors. We do this by defining functions such as:

$$\tilde{b}_{i,j}(x,y) := \tilde{b}_i(x)\tilde{b}_j(y) = C_{i,j}\cos(\pi x i)\cos(\pi y j).$$

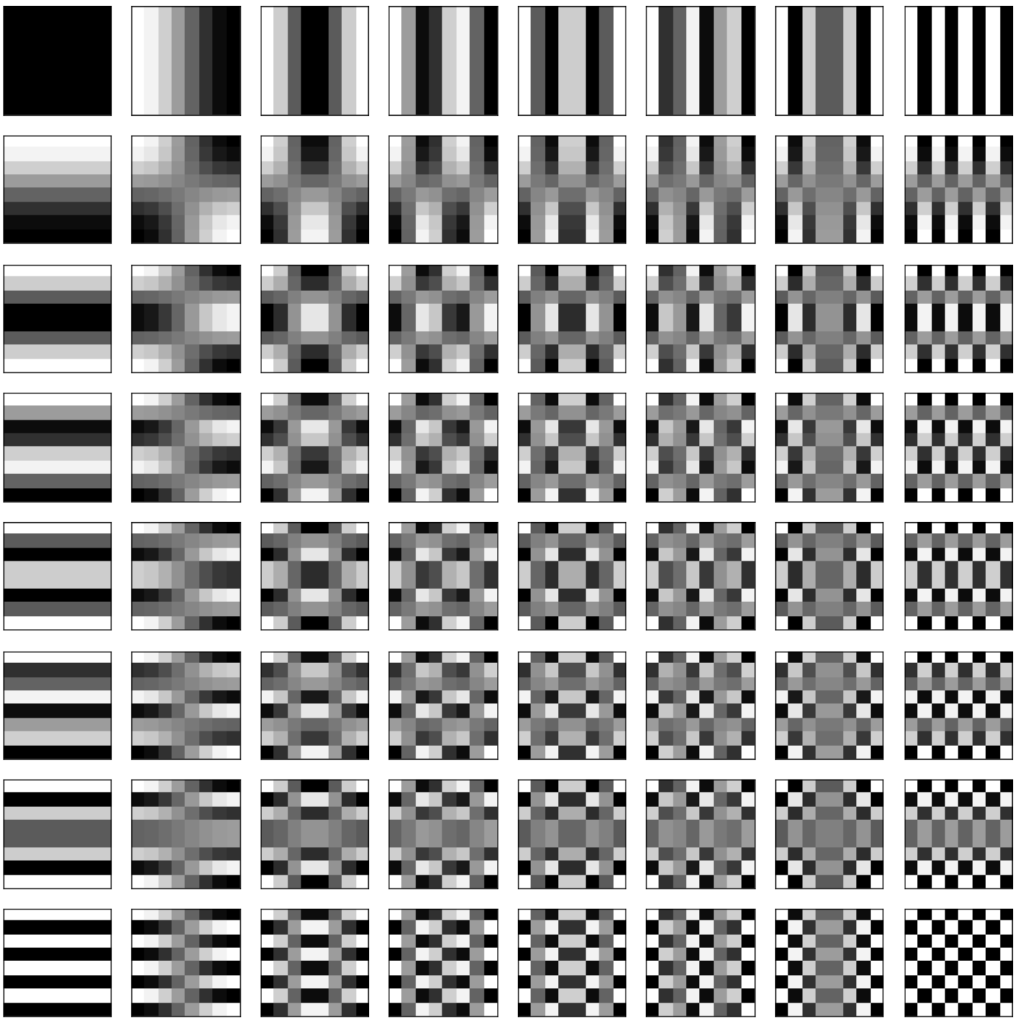
Such functions can also be quantized into matrices using:

$$b_{i,j} = b_i \bigotimes b_j,$$

where $\bigotimes$ denotes the outer product of two vectors, creating an $n \times n$ rank-1 matrix. In practice, the result looks like fig. 1.5.

Using these basis vectors, or matrices (depending on how you look at them), the signal matrix is divided into 8×8 pixel patches, and then each patch is represented as a linear combination of the 64 basis matrices. Note that no compression has occurred in this step, as we are still representing 64-dimensional signals in 64-dimensional space. However, in almost all applications (especially with natural images, meaning real-world photographs), the coefficients corresponding to the bases in the upper left corner of fig. 1.5 are generally larger then the ones in the bottom right. As such, we can truncate the solution by setting any sufficiently small coefficients to 0, thereby achieving a high degree of compression without significantly affecting the image itself. This is the process used for JPEG compression. The

Figure 1.5: Basis matrices for 2-dimensional DCT

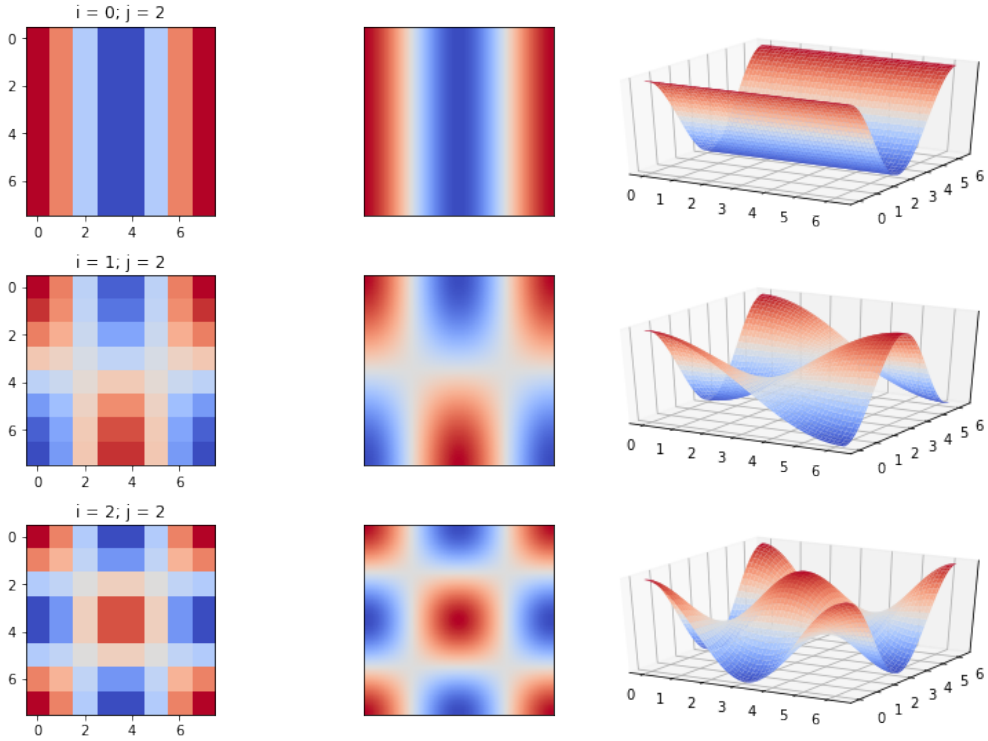


resulting coefficient matrix will be sparse, with most nonzero values located in the first row and first column.

Fig. 1.7 shows the DCT in action. It is exceedingly hard to tell the difference between the two images visually, and yet the one on the right was compressed to 10% of its original volume.

However, DCT has its weaknesses. Because it tends to ignore the contribution of the finer basis matrices (the ones on the bottom right of fig. 1.5), it has trouble representing images with fast, discontinuous switches in intensity. This rarely occurs in natural images, but does come up with images of text. This weakness results in so-called 'artifacts'

Figure 1.6: A visualization of the 2D-DCT basis vectors. The left column represents the actual vectors as shown in fig. 1.5, with i and j values denoting the row and column of the corresponding vector ($b_{i,j}$). The middle column represents the function $\tilde{b}_{i,j}$, and the right column is a 3d representation of $\tilde{b}_{i,j}$



- miscolored pixels surrounding such image patches, created because it is difficult to approximate a Heaviside function (or rather, a sequence of Heaviside functions) with cosine waves. This is evident in fig. 1.8.

One alternative that aims to remedy this issue is the wavelet transform.

1.4 Wavelet Transform

The Wavelet Transform, or Discrete Wavelet Transform (DWT) is similar to the DCT in that it uses basis functions to reconstruct an original signal. But the difference is that wavelets are not periodic like cosine waves - they are localized in time, meaning they have a much easier time dealing with irregular, discontinuous image patches. They are sensitive to fine details in a signal. And there is a plethora of wavelets to choose from, depending on the task at hand - Morlet, Daubechies, Haar etc. Some examples can be seen in fig. 1.9

Figure 1.7: Left: original image. Right: Image reconstructed after being compressed via DCT by a factor of 10



Figure 1.8: Artifacts created when applying the DCT to pictures of text



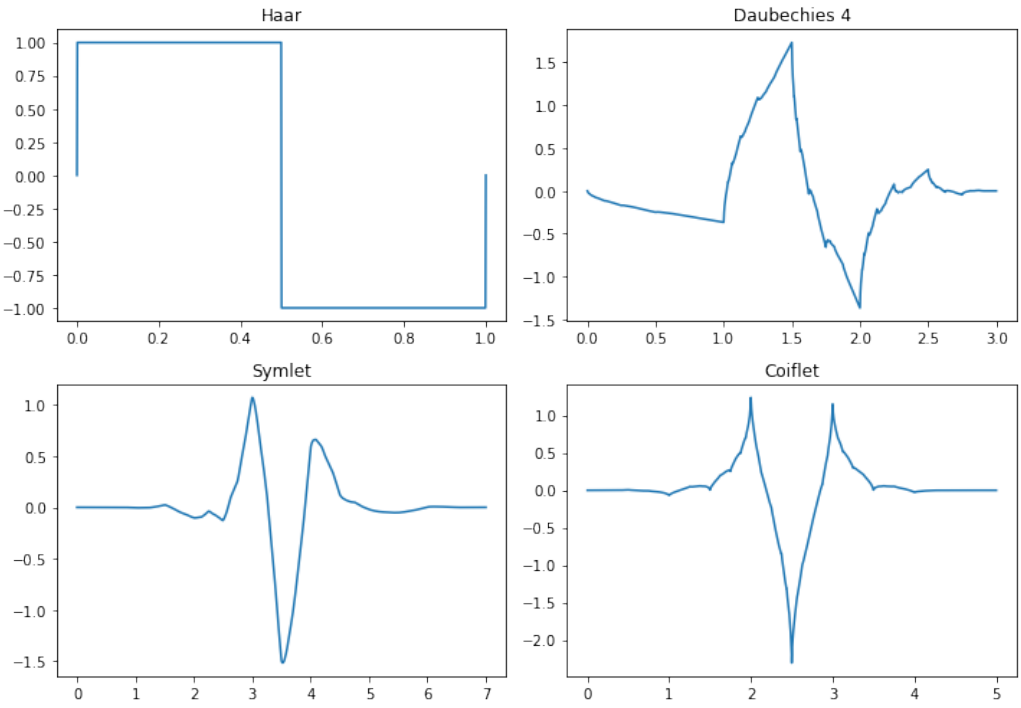
1.5 Sparse Coding

Assuming an adequate dictionary (the methods for acquiring this are discussed in chapter 2) is available, we require a mechanism for finding the sparsest representation vector, as discussed in section 1. Recall that the problem at hand is:

$$\min_{A} \|X - BA\|_F , \quad \|a_i\|_0 \leq S, \quad i = 1, \ldots, P, \tag{1.11}$$

where $A = [a_1, \ldots, a_P]$. The mapping $\|\cdot\|_0$ may be replaced with ℓ_1 to simplify the calculation. The first method for this is perhaps the most obvious: matching pursuit.

Figure 1.9: Some examples of various wavelets



Matching Pursuit

In matching pursuit (or MP), we attempt to find the solution coefficients one by one, starting with the largest one by absolute value. The motivation stems from linear spans of orthonormal bases, where

$$x = Ba = \sum_i \alpha_i b_i = \sum_i \langle b_i, x \rangle b_i.$$

The coefficients α_i in that case can be obtained simply by taking the inner product of x with any basis vector b_i. We apply the same reasoning in the case of our overcomplete dictionary - where the vectors are normalized, but not orthogonal or even independent. We again look at the inner product of x with each basis vector, then take the index associated with the highest value, and assign that inner product as the coefficient for the corresponding

dictionary atom. We then subtract that atom from x, and repeat the process.

Algorithm 1: Matching Pursuit

for $i = 1 : S$ **do**
  find j_0 s.t. $|\langle x, b_{j_0}\rangle| \geq |\langle x, b_j\rangle|$, $j = 1, 2, \ldots, K$;
  $a^{j_0} \leftarrow \langle x, b_{j_0}\rangle$;
  $x \leftarrow x - a^{j_0} b_{j_0}$
end

```python
import numpy as np

def matching_pursuit(B, v, l):
    """Returns an n-sparse vector a that minimizes
        the expression ||v - Ba||.

    Arguments:
    B: an (n x k) numpy.array dictionary
    v: an (n x 1) numpy.array signal vector that is
        to be approximated

    Returns:
    a (k x 1) numpy.array l-sparse array of coefficients
        which minimize ||v-Ba||

    """
    v_ = np.copy(v)
    coeff = np.zeros(B.shape[1])
    for i in range(l):
        c = v_.transpose()@B
        arg = np.argmax(np.abs(c))
        current_coeff = c[arg]
        if current_coeff == 0:
            break
        coeff[arg] += current_coeff
        v_ = v_-current_coeff*B[:,arg]
    return coeff
```

MP is a greedy algorithm that does not necessarily produce the globally optimal solution, but its solution is 'good enough'. Crucially, the process may be stopped at any point S' in order to obtain an S'-sparse solution. In the case of $K = S = N$, MP simply solves a complete system of linear equations.

Basis Pursuit

Basis pursuit (BP) is another pursuit algorithm which takes a slightly different approach. The problem it solves is (1.3), except we are again substituting the ℓ_1 norm in place of the sparsity map in order to generate a convex solution space:

$$\min \|a\|_1, \quad x = Ba, \tag{1.12}$$

so instead of finding the best solution with a given sparsity restriction, it finds the *sparsest* solution within a given solution space (assuming that $x = Ba$ is an underdetermined system of equations with a non-trivial solution space). This is accomplished by approaching (1.3) as an optimization problem. Recall that the canonical form for a linear optimization problem is

$$\max_{\tilde{x}} c^\top \tilde{x}, \quad L\tilde{x} \leq v, \quad \tilde{x} \geq 0, \quad \tilde{x} \in \mathbb{R}^m,$$

where $c^\top \tilde{x}$ is an objective function, $L\tilde{x} \leq v$ is a collection of inequality constraints, and $\tilde{x} \geq 0 \Leftrightarrow \tilde{x}_i \geq 0, i = 1, \ldots, m$ is a set of bounds. We can reformulate (1.12) thus:

$$m \Leftrightarrow 2K; \quad L \Leftrightarrow (B, -B); \quad v \Leftrightarrow x; \quad \tilde{x} \Leftrightarrow (u; v); \quad a \Leftrightarrow u - v, \tag{1.13}$$

where u and v are the positive and negative parts of a (specifically, $u_i = \max(a_i, 0)$ and $v_i = \max(-a_i, 0), i = 1, \ldots, K$. They are concatenated into one long vector of free variables $\tilde{x} = (u_1, \ldots, u_K, v_1, \ldots, v_K)$ which is used as the variable to be optimized. Thus, any positive values of a can be represented as values of u, with the corresponding value of v being 0, and vice-versa - any negative values are represented in v, with the corresponding value of u being 0. This way, the concatenated vector $\tilde{x}$ is positive in every variable, simplifying the linear program. All that remains is to set $c = (-1, -1, \ldots, -1)$, and the full linear optimization problem looks like:

$$-\tilde{x}_1 - \tilde{x}_2 - \ldots - \tilde{x}_m \to \max$$

$$\left[\begin{array}{cccc|cccc} b_1^1 & b_2^1 & \ldots & b_K^1 & -b_1^1 & -b_2^1 & \ldots & -b_K^1 \\ b_1^2 & b_2^2 & \ldots & b_K^2 & -b_1^2 & -b_2^2 & \ldots & -b_K^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ b_1^N & b_2^N & \ldots & b_K^N & -b_1^N & -b_2^N & \ldots & -b_K^N \end{array}\right] \begin{bmatrix} u_1 \\ \vdots \\ u_K \\ v_1 \\ \vdots \\ v_K \end{bmatrix} \leq \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

$$u_i, v_i \geq 0, \quad i = 1, \ldots, K,$$

and this system can be solved by any linear optimization method, such as the simplex algorithm or the interior-point method. More on BP and MP may be found in [7] or [5].

1.6 Classification

As previously stated, one useful application of sparse representations besides data compression is in classification. Classification is the task of determining the class of some object represented by a vector, given that we have previously seen a number of examples of each possible class by way of a labeled training data set.

Assume we are given such a set of training data - n-dimensional vectors each belonging to one of c classes, such that each class is represented by p vectors. We might construct a dictionary B as follows:

$$\begin{aligned} B &= \Big[B_1, \ldots, B_c\Big] \\ &= \Big[x_{11}, \ldots, x_{1p}|, x_{21}, \ldots, x_{2p}|, \ldots, |x_{c1}, \ldots, x_{cp}\Big]. \end{aligned}$$

If we then represent a new vector y using this dictionary:

$$y = \sum_{i=1}^{c} \sum_{j=1}^{p} a_{ij} x_{ij},$$

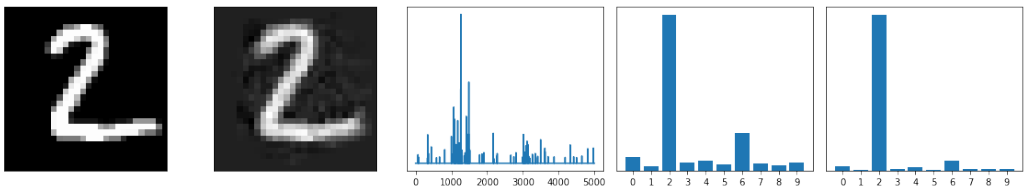
or in other words, $y = Ba$, we can assume that, given sufficient training samples of the k-th class, B_k, any new test vector y that belongs to class k will lie approximately in the linear span of the training samples from that class. Note that in the above expression, a_{ij} is a scalar, but x_{ij} is a column vector. Therefore, the coefficients associated with basis elements from class k ($a_{k1}, \ldots, a_{kn}$) should have the highest values, allowing us to determine the underlying class. We will test this premise in the next section, as well as delve into slightly more detail regarding the classification mechanism.

Classifying MNIST

MNIST is a basic benchmark for testing classification algorithms, and the method described above works quite well. In our case, we constructed the dictionary using 500 examples of each class (digit), represented their pixel-value matrices unraveled into 784-dimensional vectors.

In almost all cases with MNIST, the digit in question was represented almost exclusively by other images of the same digit, making classification easy. All that remains is to design some selection method - an arbitrary mapping from the space of representations on to the set of all classes. For example, we might simply pick the class that the highest coefficient corresponds with. Or, we might create a histogram of all the coefficients, then pick the class that is most-represented. Or we could square the values before making the

Figure 1.10: MNIST classification using sparse coding. The first image is the original. The second image is its reproduction using a linear combination of 100 dictionary atoms. The third image is a representation of the coefficients associated with each dictionary atom. The fourth image is a histogram of those values. The fifth image is a histogram of the squares of those values.



histogram, in order to suppress small values and promote high values. There are similarities between these options and ℓ_p vector norms. In our case, chose a sparsity requirement of $S = 10$, using a dictionary of $K = 5,000$ atoms. Using all 3 of the classification rules outlined above, we achieved an accuracy of 94%, over a testing set of $10,000$ images. In fact, all three rules nearly always classified the image the same way, differing in only 20 cases.

In fig. 1.11, we can see some of the images that were incorrectly classified, as well as their reconstructions and the incorrect predictions. Fig. 1.12 shows us which digits were most commonly misclassified. We can see that classification for digits 0, 1 and 6 was the most accurate, while the inaccuracies were evenly distributed over the other digits.

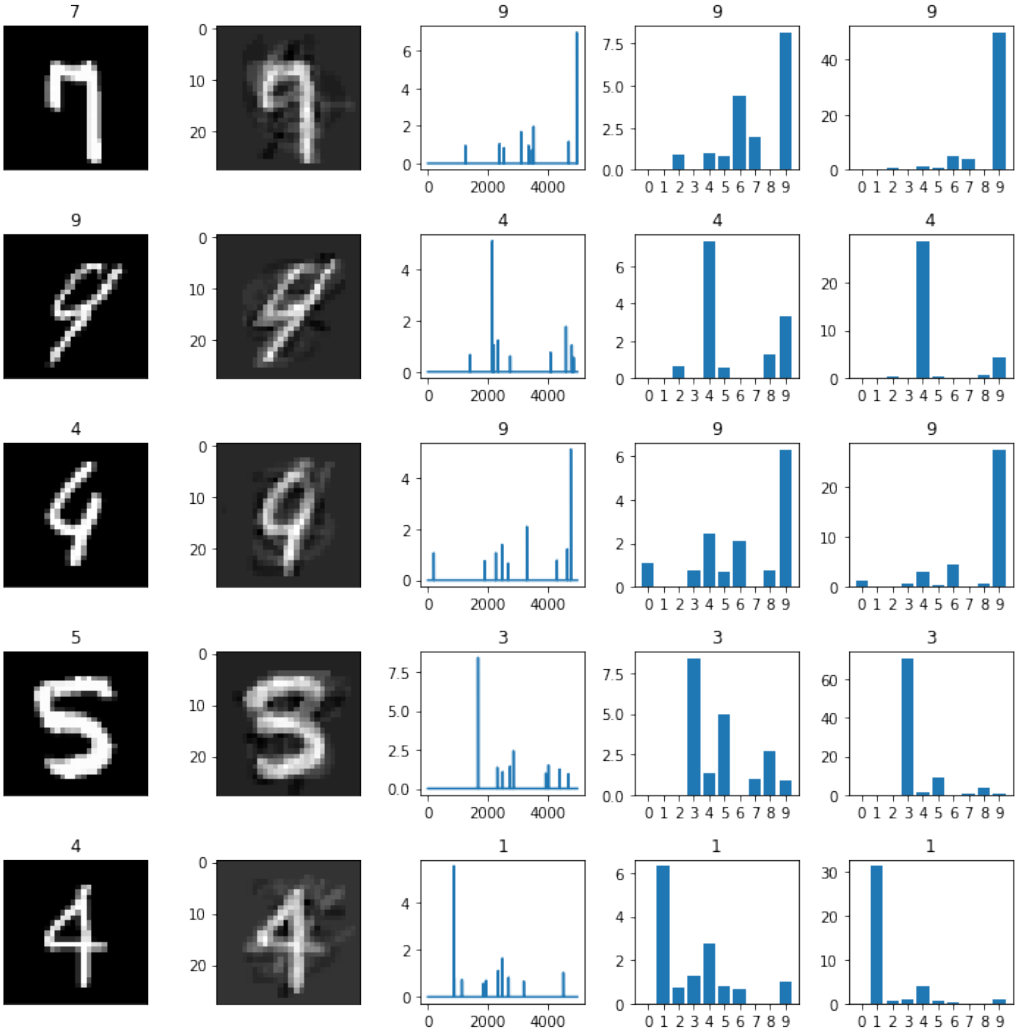
This approach is not without issues, however. If the angle between two of the class-specific linear spans ($[B_i]$ and $[B_j]$) is too small, then MP might misrepresent some signal vectors. An example of such a case is illustrated in fig. 1.13. In this case,

$$B = \begin{bmatrix} 1 & 0 & 0.67 \\ 0 & 1 & 0.67 \\ 0 & 0 & 0.1 \end{bmatrix}; \quad x = \begin{bmatrix} 0.7 \\ 0.7 \\ 0 \end{bmatrix},$$

and let us assume that b_1 and b_2 represent one class (which spans the entire x-y plane), and b_3 belongs to another class. Matching pursuit will pick b_3 as the optimal representation in the first iteration, and after that the remaining component, $x - \langle x, b_3 \rangle b_3$ is very small and not much use in classification. Specifically, the sparse representation derived from MP for this example is $a = (0.01, 0, 1.024)^\top$, so any classification rule would classify x as a member of class 2 - the one represented by b_3, even though it belongs to the linear span of b_1 and b_2, as illustrated in fig. 1.13.

We can formalize this. Assume that our classifier works by applying matching pursuit to a signal vector to solve (1.1), then observing the resulting coefficients in a and which classes are represented the most. We can show that the most represented basis vector

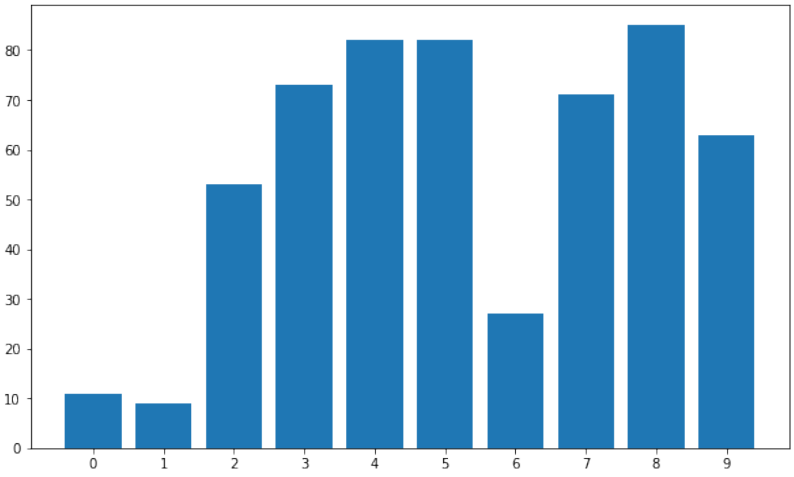
Figure 1.11: Examples of misclassified images. The columns represent the same things as in fig. 1.10, with the addition of titles, which represent the ground truth (column 1), and the wrong guess for each classification rule (columns 3, 4 and 5).



derived by MP will simply be the closest one. Practice has shown that usually, classification comes down to simply picking the class of this most represented vector, as its coefficient will be orders of magnitude bigger than the second most represented one.

Proposition 1.6.1. *Let $B = \{b_1, \ldots, b_k\}$ be a set of n-dimensional unit vectors, and let x be am n-dimensional signal vector. Then the most parallel member of B is also the closest*

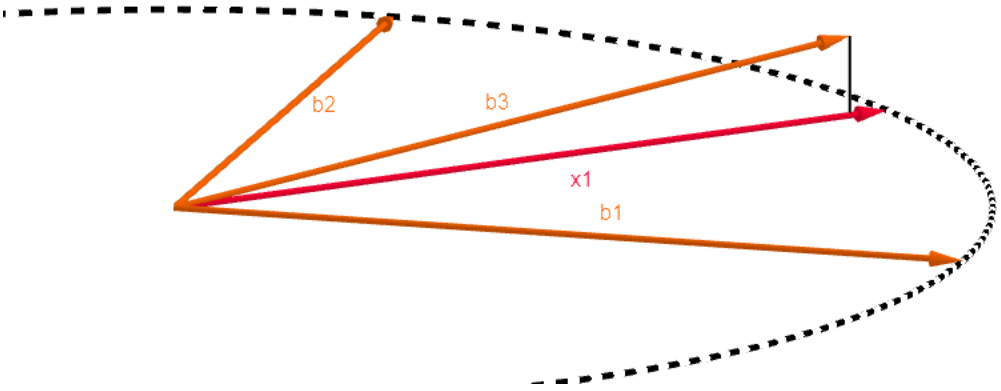
Figure 1.12: A chart representing how many images of each digit were mis-classified



member of B. In other words, for some index i,

$$\langle x, b_i \rangle \geq \langle x, b_j \rangle, \forall j \Rightarrow \|x - b_i\|_2 \leq \left\|x - b_j\right\|_2, \quad j = 1, \ldots, k$$

Figure 1.13: Sparse classification error - basis vectors are shown in orange, and the target signal vector is red.



Proof. Assume $\langle x, b_i \rangle \geq \langle x, b_j \rangle$, $j = 1, \ldots, K$. Then,

$$\begin{aligned} \|x - b_i\|_2^2 &= \langle x - b_i, x - b_i \rangle \\ &= \langle x, x \rangle - 2\langle x, b_i \rangle + \langle b_i, b_i \rangle \\ &\leq \langle x, x \rangle - 2\langle x, b_j \rangle + \|b_i\|_2^2 \\ &= \langle x, x \rangle - 2\langle x, b_j \rangle + \left\|b_j\right\|_2^2 \\ &= \langle x, x \rangle - 2\langle x, b_j \rangle + \langle b_j, b_j \rangle \\ &= \left\|x - b_j\right\| \end{aligned}$$

□

Thus, a classifier constructed this way is very similar to a simple nearest-neighbor classifier.

Remark 1.6.2. *The reasoning above does not take into account the case where $\angle(x, b_i) > \frac{\pi}{2}$, but that should be a rare occurrence in principle, when b_i is the best-matching basis vector.*

1.7 Compressed Sensing

Data compression has massive real-world applications and benefits in and of itself. However, compressed sensing is an idea that builds on it even further. The core concept is simple: instead of collecting and then discarding most of it via compression, why not collect only a fragment of the data to begin with? For example, instead of taking a full $m \times n$ picture, we might only test a few, sparsely distributed pixels, and then infer the complete photograph from those. The idea is relatively simple to state mathematically: let x be a signal, and y be a measurement of only some of its variables, given by

$$y = Cx,$$

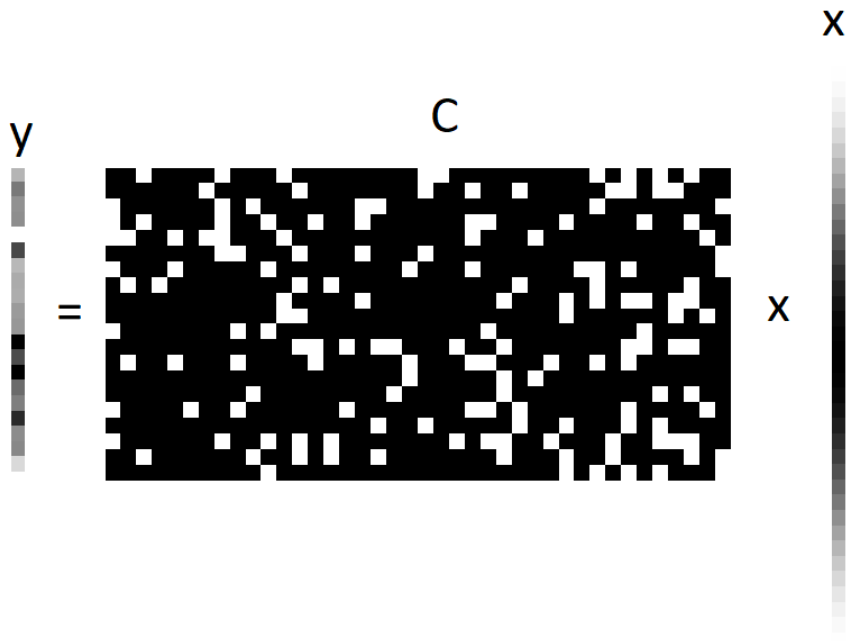
where C represents some sort of a selection matrix. For example, C could choose one of every three variables in x, or be a binomially distributed, such that each element c_{ij} has a chance p of being either 0 or 1, as illustrated in fig. 1.14.

Assuming we know $y = Cx$, and have an adequate dictionary B which sparsifies x, we have

$$y = Cx = CBa = \Theta a,$$

where $\Theta = CB$. So the task remains the same as before - finding the sparsest a which satisfies $y = \Theta a$. As discussed in section 1.5, this is an NP-hard problem, unless the

Figure 1.14: Binomially distributed matrix taking a compressed measurement of a signal vector x



sparsity constraint is relaxed to the ℓ_1 norm instead, in which case we can use basis pursuit or another sparse coding algorithm to derive the representation a.

Compressive sensing has a wide array of applications. Whenever we have some mapping from an uncountable domain, but we can only afford to measure its value in a finite number of locations, we can use compressive sensing to approximate a discretization of the mapping over the rest of the domain. In that case, the selection matrix C would signify the points at which we are measuring (sensing) the signal. For example, if we were looking at an image with pixels missing from some locations (due to a faulty camera or a corruption of the digital data, for example), we might construct C by first taking an identity matrix, and then eliminating the rows which correspond to the dead pixels in the vectorized image. Here are some more examples:

- Studying ocean region properties (water flow, heat, salinity etc) by measuring it at discrete points via sensor buoys
- Studying the flow of a fluid through some geometry by placing a number of sensors at specific points inside a pipe or cylinder

- Recovering visual data from a corrupted image with missing pixels (this process is also known as 'inpainting')
- Removing objects (such as unwanted text) from an image by treating it as dead pixels
- Reconstructing a photograph that was taken by only measuring a few pixels on purpose - this approach removes some of the technological and storage requirements from the camera and places them on the receiving end. For example, a deep space satellite could use a lower resolution camera to take photographs, then send the unencoded (but still low-volume) data back to earth. Current satellites have to compress the data on-site, increasing computational requirements, before sending it back.

Additionally, in the next chapter, we will discuss methods of not only deriving a sparse representation given a fixed dictionary, but optimizing the dictionary itself. That means we will be able to optimize the *placement* of the sensors mentioned above, in order to get more accurate measurements.

Chapter 2

Dictionary Learning

In chapter 1 we established a number of methods for obtaining the representation vector that satisfies $x = Ba$, given an adequate dictionary B. Now, we will discuss methods for obtaining the dictionary itself. In other words, we are coming back to the initial problem (1.1) and its ℓ_1 counterpart. Recall that it was stated as:

$$\min_{B,A} \|X - BA\|_F, \quad \|a_i\|_1 < l, \quad i = 1, 2, \ldots, P,$$

where a_i are columns of A. We will first review the some k-means clustering methods, including Lloyd's algorithm, which is a special case of the K-SVD algorithm discussed afterwards.

2.1 K-Means Clustering

K-means clustering refers to the task of grouping a number of data points into distinct clusters, or minimizing the 'within-cluster sum-of-squares' (WCSS):

$$E(C) = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - c_i\|_2^2, \tag{2.1}$$

where $C = \{C_1, \ldots, C_k\}$ is a partition of the set of data points x_i, and c_i are the centroids of C_i, given by

$$c_i = \frac{\sum_{x \in C_i} x}{|C_i|}, \quad i = 1, \ldots, k, \tag{2.2}$$

$|C_i| \in \mathbb{N}$ being the (finite) number of elements in C_i.

This problem is also NP-hard - even with only two clusters, and even in only two dimensions ('planar' k-means), as per [15]. The most common approach to solving it

numerically is an iterative process known as 'Lloyd's algorithm' or 'naive k-means'. It is so common, in fact, that it is sometimes referred to as simply 'k-means algorithm'.

Algorithm 2: Lloyd's algorithm

determine initial centroids $c_1, \ldots, c_k$
for $i = 1 : I$ **do**
  for x *in* X **do**
    find closest centroid c_i
    assign x to subset C_i
  end
  for $i = 1 : k$ **do**
    $c_i \leftarrow \sum_{x \in C_i} x/|C_i|$
  end
end

```
def kmeans(X, n, iter):
    """A basic implementation of the k-means algorithm. Clusters the
    signal vectors in X into n groups, and returns the centroid of each
    group.

    Arguments:
    X: (p x m) numpy.array() - data matrix where each row represents one
     data point
    n: integer - determines the desired number of centroids/clusters
    iter: integer - pre-determined number of iterations

    Returns:
    assign: (p x 1) numpy.array() - the i-th element is the index of the
     group that the i-th data vector belongs to
    centroids: (n x m) numpy.array() - the i-th row represents the
    coordinates of the centroid of the i-th group
    """
    # initialization
    centroids = np.outer(np.linspace(0,1,n), np.ones(X.shape[1]))
    assign = np.zeros(X.shape[0], dtype=int)
    d = np.zeros((X.shape[0], n))
    count = np.zeros(n)
    for i in pb(range(iter)):

        # assignment
        d = np.array([np.sum((X-c)**2, axis=1) for c in centroids]).
    transpose()
        assign = np.argmin(d, axis=1)
        count = np.array([np.sum(assign==j) for j in range(n)])

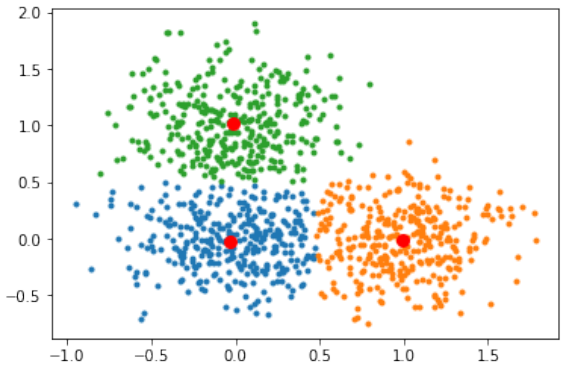
```

```
25          # fix empty clusters
26          for j in np.where(count==0)[0]:
27              assign[random.randint(0,X.shape[0])] = j
28
29          # recalculate centroids
30          centroids = np.array([np.sum(X[assign==j], axis=0)/np.sum(assign
    ==j) for j in range(n)])
31
32      # final assignment
33      d = np.array([np.sum((X-c)**2, axis=1) for c in centroids]).
    transpose()
34      assign = np.argmin(d, axis=1)
35      return assign, centroids
```

Here, I is some arbitrary pre-determined number of iterations until convergence is reliably reached. Alternatively, some stopping criteria may be used, such as per-iteration error gain, or some target error value. As for the initial centroids, there are several possible approaches. One is to generate a random k-partition of the data, then assign the centroids accordingly, using (2.2). Another is to scatter initial centers randomly throughout the dataset (this requires knowing or calculating the lower and upper bound of the space ahead of time), or simply distributing them evenly across each dimension - $c_i := (m_1 + h_1 i, m_2 + h_2 i, \ldots, m_n + h_n i)$, where m_i and M_i are the minimum and maximum possible values in the i-th dimension, and $h_i = (M_i - m_i)/k, i = 1, \ldots, n$, or even randomly selecting n of the data points themselves and setting the initial centers to their locations.

The weakness of K-means is that it is not invariant for different starting point assignments /center placements, and it can get trapped by local minima. This is sometimes circumvented by running the algorithm repeatedly with various starting assignments, then taking the solution with the best error score.

Figure 2.1: Clustering of some synthetic data



The applications of clustering methods are vast, and data compression is certainly one

of them. We can imagine an image as a collection of pixels located in RGB pixel-space - that is, each pixel has three defining axis values: red, green and blue, and we consider its position in the image as just a label. We can then apply clustering to the points. Then we can transform the image by replacing each pixel value with the value of its centroid. This way, we get a faithful representation of the original image, using only a few colors. We can specify the transformation as:

$$T : V_{pixel} \longrightarrow C$$
$$T(x) = \underset{c \in C}{\operatorname{argmin}} \|x - c\|_2,$$

where V_{pixel} is the space of all possible RGB pixel values (equivalent to $[0, 1]^3$, $C = \{c_1, \ldots, c_n\}$ is the set of all derived centroids, so T simply maps any pixel to its closest centroid.

Figure 2.2: Clustering applied to an RGB Image with only red and green components. Left: original image. Middle: color-clustered image using $n = 50$ clusters. Right: red-green pixel space with Voronoi cells to represent the clusters.

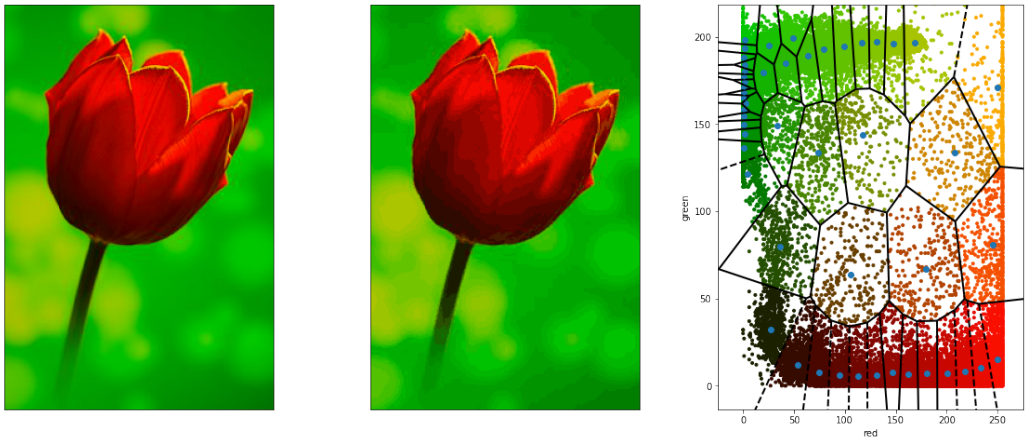


Fig. 2.2 shows how an image can be relatively faithfully reconstructed using only 50 different colors. The compression gain here is that we no longer need to store three floating-point values for each pixel, but instead a single integer with value up to 50, meaning we only need 6 bits of storage per pixel, as opposed to the 192 bits that the original pixels would've required.

Advanced K-means Implementations

Being such a fundamental and versatile algorithm, K-means has been thoroughly studied over the years, and several improvements and optimizations have been established. We will briefly cover some of them here.

In [2], a more efficient method for centroid initialization is established by first taking J sub-samples of the total pool of data, and clustering them to get J sets of centroid approximation. Then that set of approximations is itself clustered, and the resulting centroids are used as initial centroids when running the algorithm over the entire data set. In practice, this initialization has been shown to be so powerful as to require very little optimization afterwards in order to complete clustering.

In [11], the algorithm is restructured in order to speed up the cluster assignment: $p_i \longleftarrow \text{argmin}_j \left\|x_i - c_j\right\|_2$, where c_j are centroids. In short, they compute a tree structure over all the data points, then use a recursive function to assign each data point to a cluster, at each step of the algorithm, instead of solving the problem above by a brute force search over all possible values. This approach achieves more CPU-time gains for higher values of k (meaning more clusters/centroids). The results shown in [11] display consistent, 80-95% decreases in time needed to complete the algorithms, as compared to vanilla k-means.

In [20], the authors present a completely new approach to the problem. The first two items in this section are simply optimizations of Lloyd's algorithm, whereas this is an entirely new algorithm. First, the sum-of-squares cost function,

$$E(C) = \sum_{i=1}^{k} \sum_{j=1}^{|C_i|} \left\|a_j^{(i)} - c_i\right\|_2^2$$

is reformulated as

$$E(C) = \text{trace}(A^\top A) - \text{trace}(X^\top A^\top A X), \tag{2.3}$$

where $A = [a_i, \ldots, a_n]$ is the data matrix, and X is an $n \times k$ orthonormal matrix:

$$X = \begin{matrix} |C_1| \\ |C_2| \\ \vdots \\ |C_k| \end{matrix} \begin{bmatrix} e_1/\sqrt{|C_1|} & & & \\ & e_2/\sqrt{|C_2|} & & \\ & & \vdots & \\ & & & e_k/\sqrt{|C_k|} \end{bmatrix},$$

and e_i are $|C_i|$-dimensional column vectors with each value equal to 1, so minimizing the cost function E is equivalent to maximizing the right-hand side of (2.3). If we also ignore the structure of X and *relax* the constraint to any orthonormal matrix, we get a new maximization problem:

$$\max_{X^\top X=I} \left(\text{trace}(X^\top A^\top A X)\right), \tag{2.4}$$

which, according to Ky Fan's theorem, can be solved by finding the first k eigenvectors of $A^\top A$ (meaning the ones corresponding to the k largest eigenvalues). Finally, a cluster assignment is computed using pivoted QR decomposition.

2.2 Singular Value Decomposition

The Singular Value Decomposition algorithm is a factorization of a real or complex matrix that generalizes the eigenvalue decomposition of square normal matrices to any $m \times n$ matrix. The eigenvalue decomposition would factorize the square $n \times n$ matrix A as $A = Q\Lambda Q^{-1}$, where Λ is a diagonal matrix of eigenvalues, and Q is a square $n \times n$ matrix such that each column is a right eigenvector corresponding to the eigenvalue in the same column of Λ. SVD of an $m \times n$ matrix A looks similar:

$$A = U\Sigma V^*, \tag{2.5}$$

where U and V^* are unitary matrices of left and right singular vectors respectively, U being an $m \times m$ matrix, V an $n \times n$ matrix, and Σ being an $m \times n$ diagonal matrix of singular values. Singular values work much the same way as eigenvalues and eigenvectors, except that they are necessarily non-negative real numbers. It holds that

$$Av_i = s_i u_i, \quad A^* u_i = s_i v_i, \quad i = 1, \ldots, \min(m, n),$$

where u_i and v_i are columns of U and V, respectively. Furthermore, (2.5) can be rewritten as

$$A = \sum_{i=1}^{r} s_i u_i v_i^\top,$$

where $\text{rank}(A) = r$. This leads to an important conclusion about approximation. If we define a new matrix with

$$A_k = \sum_{i=1}^{k} s_i u_i v_i^\top,$$

it holds that A_k is the best rank-k approximation for A (Eckart-Young-Mirsky theorem, [8]), and its residual is $R_k = A - A_k = \sum_{i=k+1}^{r} s_i u_i v_i^\top$. In other words,

$$\underset{\tilde{A}:\text{rank}(\tilde{A})=k}{\text{argmin}} \ \text{RSS}_A(\tilde{A}) = A_k, \tag{2.6}$$

where

$$\mathrm{RSS}_A(\tilde{A}) := \left\|A - \tilde{A}\right\|_F^2 = \sum_{i,j} |a_{ij} - \tilde{a}_{ij}|^2$$

is the 'residual sum of squares' error function, or the sum of the squares of elements of the residual of an approximation.

SVD Pseudoinverse

One of the many applications of the SVD is to generate a Moore-Penrose pseudoinverse, which is to say a matrix A^+ with the following properties:

- $AA^+A = A$
- $A^+AA^+ = A^+$
- $(AA^+)^* = AA^+$
- $(A^+A)^* = A^+A$

A^+ exists for any matrix A. SVD is used to generate this matrix by first decomposing $A = U\Sigma V^*$, then defining Σ^+ by replacing every non-zero diagonal entry in Σ with its reciprocal and transposing the resulting matrix. Then we can obtain

$$A^+ = V\Sigma^+U^*.$$

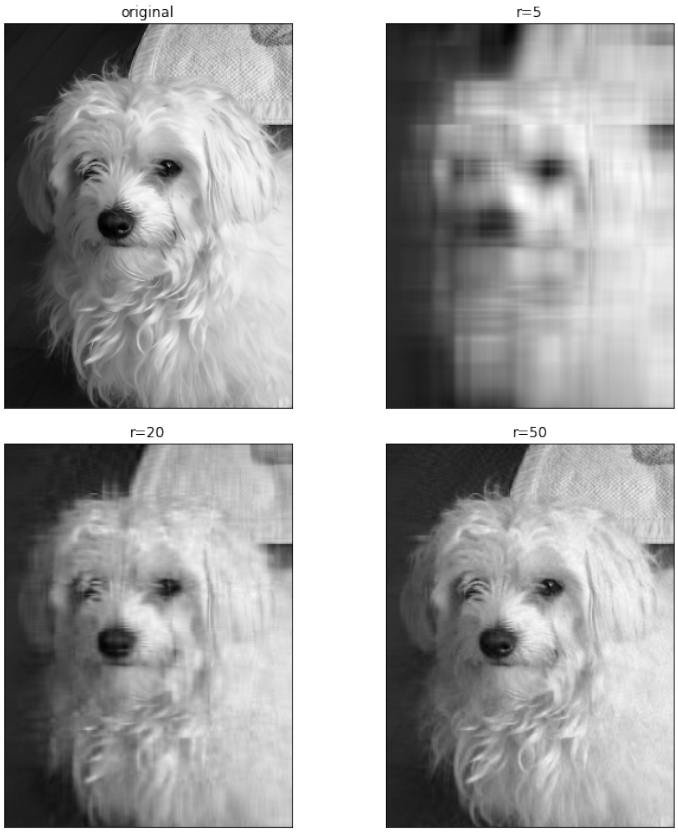
SVD Compression

As mentioned in the previous section, singular value decomposition can be used to generate limited-rank approximation of matrices. This is directly applicable to image compression. Instead of saving the full image matrix, M, we can save the first r columns from U and V and the first r values from Σ and recover a faithful reconstruction of the original image. In fact, a very convenient property of SVD is that the importance, or contribution, of each column in U or V corresponds to the magnitude of the corresponding singular value in S. Thus, the smaller the index i, the more important is the rank-1 matrix $u_i s_i v_i^\top$. Observe the natural image example shown in fig. 2.3

In fact, if we take a look at the singular values themselves, we can see that they usually have fast decay, which is why most of the information energy is conserved in the first few.

Clearly, approximating this image with only the first 50 columns yields a very faithful reproduction. Since the original dimension is 400×300, and we are now only storing one 400×50 matrix, one 300×50 matrix and a 50×1 vector, we are saving as much as 80% of the storage space, which is a remarkable compression rate.

Figure 2.3: A 300×400 image of a dog approximated with matrices of various ranks



2.3 K-SVD Algorithm

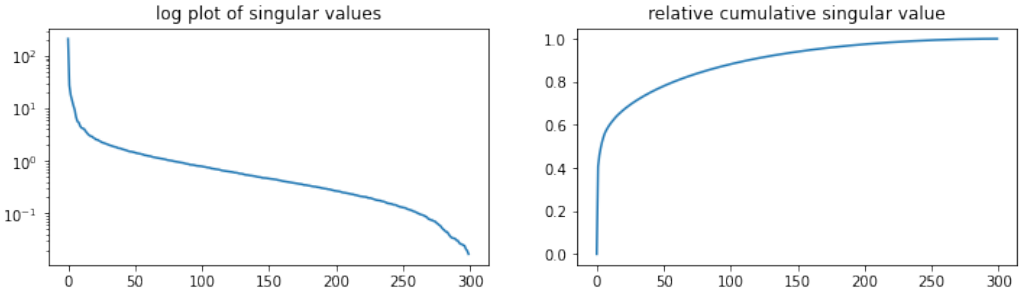
Now we finally arrive to K-SVD - a method to dynamically learn a dictionary for sparse representations. The idea is essentially similar to K-Means - it is an iterative process, with each iteration consisting of two steps. In one step, we optimize the representation matrix A, and in the other step we optimize the dictionary B itself, one column (atom) at a time. We will describe the dictionary optimization below.

Let us assume we have some dictionary B and representation A that minimizes (1.4). We want to obtain a better dictionary, $\tilde{B}$, and the corresponding representation $\tilde{A}$ s.t.

$$||X - \tilde{B}\tilde{A}|| < ||X - BA||$$

As stated, we will be optimizing one dictionary atom - column - at a time. Let us select a column b_k, and fix all other columns. The error function is now

Figure 2.4: The singular values from the decomposition of the pixel matrix of the original image in fig. 2.3



$$\|X - BA\|_F^2 = \left\| X - \sum_{\substack{i=1 \\ i \neq k}} b_i a_i^\top - b_k a_k^\top \right\|_F^2 .$$

In the above expression, the terms in the sum are outer products of vectors, meaning it is a sum of rank-1 matrices. The terms $a_i^\top$ represent rows of A. We can combine the first two terms, which are fixed, into a single matrix E_k:

$$\left\| E_k - b_k a_k^\top \right\|_F^2 . \tag{2.7}$$

$b_k a_k^\top$ is, again, a rank-1 matrix, and so task is to find the best rank-1 approximation for E_k, which is exactly what SVD accomplishes (recall eq. (2.6)). However, there is one more preparatory step to take. If we were to utilize SVD right now to optimize the expression (2.7), we would most likely end up with a full vector for $a_k^\top$, but that vector has to remain sparse. As such, we will extract a submatrix from E_k by taking all the rows which actually use the atom b_k in their representation. In other words, we will construct a set of indices, ω_k with

$$\omega_k = \{i | 1 \leq i \leq K, a_{ki} \neq 0\}.$$

So we are checking the k-th row of A to see which data vectors x_i use the atom b_k in their representation. If a_{ki} is non-zero, then the atom is being used to represent x_i. Once ω_k is obtained, we define a filter matrix Ω_k as a matrix of size $N \times |\omega_k|$ with ones on the $(\omega_k(i), i)$ entries and zeros elsewhere. Now we can multiply vectors and matrices with Ω_k in order to extract the rows or columns according to ω_k. So we can define $E_k^R := E_k \Omega_k$ and $a_k^R = x_k^\top \Omega_k$, and we get

$$\left\|E_k\Omega_k - b_k a_k^\top \Omega_k\right\|_F^2 = \left\|E_k^R - b_k a_k^R\right\|_F^2 .$$

Again, $b_k x_k^R$ is a rank-1 matrix which we can freely optimize (we simultaneously optimize both b_k and a_k^R), so now we can employ SVD. We decompose $E_k^R = U\Sigma V^\top$, and recall that $u_1 s_1 v_1^\top$ is the best rank-1 approximation for E_k^R (u_1 and $v_1^\top$ being the left and right singular vectors corresponding to the highest singular value s_1 of E_k^R), and so all we have to do is define the optimized $\tilde{b}_k$ as u_1 and a_k^R as $s_1 v_1^\top$. This way we even keep the dictionary atom normalized. We repeat the process for each atom in turn, and that completes a single iteration. The final algorithm is as follows:

Algorithm 3: K-SVD Algorithm

Initialize dictionary B
for $i = 1 : l$ **do**
  `// sparse coding`
  for $p = 1 : P$ **do**
    $a_p \longleftarrow \text{argmin}_{a:\|a\|_0 \le l} \left\|x_p - Ba\right\|$; `// e.g. via matching pursuit`
  end
  `// optimize dictionary`
  for $k = 1 : K$ **do**
    $\omega_k \longleftarrow \{i | 1 \le i \le K, a_{ki} \neq 0\}$
    define Ω_k as specified earlier in the chapter
    $E_k = X - BA + b_k a_k^\top$; `// outer product`
    $E_k^R = E_k\Omega_k$
    compute SVD: $E_k^R = U\Sigma V^\top$
    $b_k\Omega_k \longleftarrow u_1$
    $a_k\Omega_k \longleftarrow s_1 v_1^\top$
  end
end

```python
import numpy as np

def learn_dictionary(B, A, X):
    """Optimizes the dictionary B to better minimize the
        expression ||X-BA||

    Arguments:
    B: (n x k) np.array - dictionary
    A: (k x p) np.array - coefficient matrix
    X: (n x p) np.array - signal matrix (each column is a signal)

    Returns:
```

```
    None
    """
    K = B.shape[1]

    for k in range(K):
        # filter out signals which do not
        # use the given dictionary atom
        filter = (A[k, :] != 0)
        if np.sum(filter) == 0: continue

        A_k = A[:,filter]
        X_k = X[:,filter]
        E_k = (X_k - B@A_k + np.outer(B[:,k], A_k[k,:]))

        # SVD
        u, d, v = np.linalg.svd(E_k)
        B[:,k] = u[:,0]
        A[k,filter] = d[0]*v[0,:]
```

```
def k_svd(X, B, l, iter, stop):
    """Applies the K-SVD algorithm to optimize arguments B and A
        to minimize the expression ||X-BA||

    Arguments:
    X: (n x p) np.array - signal matrix
    B: (n x k) np.array - initial dictionary
    l: integer - the sparsity constraint - all columns of
        the resulting A matrix will be at least l-sparse
    iter: integer - maximum number of iterations to run
    stop: float - minimum percentage error gain. If the solution
        improves less than that in an iteration, the process
        is considered to have converged Smaller values mean more
        accurate solutions, but also more iterations.
    """

    N, P = X.shape
    K = B.shape[1]
    A = np.zeros((K, P))
    Xnorm = np.linalg.norm(X)

    # obtain initial coefficient matrix
    for p in range(P):
        c = matching_pursuit(B, X[:,p], l)
        A[:,p] = c
    progress = [np.linalg.norm(X-B@A)/Xnorm]

    for i in range(iter):
        # optimize dictionary
```

```
        learn_dictionary(B, A, X)

        # optimize coefficient matrix
        for p in range(P):
            c = matching_pursuit(B, X[:,p], l)
            A[:,p] = c
        err = np.linalg.norm(X-B@A)/Xnorm
        progress.append(err)

        # stopping condition
        impr = 1-(progress[-1]/progress[-2])
        if impr < stop: break
    return B, A, progress
```

The dictionary can be initialized in a number of ways, for example by simply including a random subset of K of the training vectors. Note that assignments such as $b_k\Omega_k \longleftarrow u_1$ signify that we are only updating those rows of b_k with indices in ω_k.

Testing on Synthetic Data

The algorithm was first tested using some synthetic data. The data was generated by first creating a 'real' $N \times K$ dictionary, B_0, and populating its columns with vector representations of various cosine waves:

$$v_{kn} = \cos\left(\frac{nhk}{2}\right), n = 1, \ldots, N; k = 1, \ldots, K$$
$$b_k = \frac{v_{k,:}}{\left\|v_{k,:}\right\|}, k = 1, \ldots, K$$
$$B = [b_1, \ldots, b_K].$$

Then we generated a $K \times P$ sparse coefficient matrix A_0, so we could obtain the data matrix $X = BA$. This way, our problem has a 'true' l-sparse solution - the matrices B_0 and A_0, and we can compare the derived solutions - B and A - to it. In our case, we used

$$N = 20; \quad K = 50; \quad P = 1000; \quad L = 5.$$

Then, we commence K-SVD iterations by first initializing B to be the first k signal vectors. During operation, we measure the accuracy of our model with a simple normalized loss function:

$$E_X(B, A) = \frac{\|X - BA\|_F}{\|X\|_F}.$$

Figure 2.5: Some of the dictionary atoms used to generate synthetic data

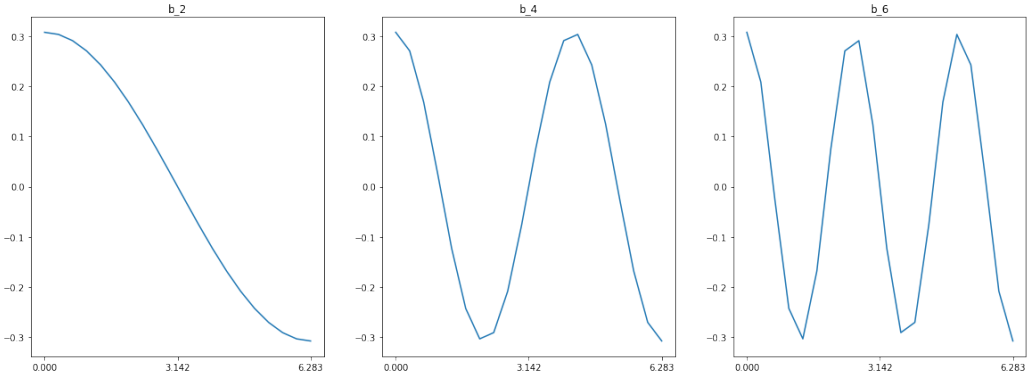
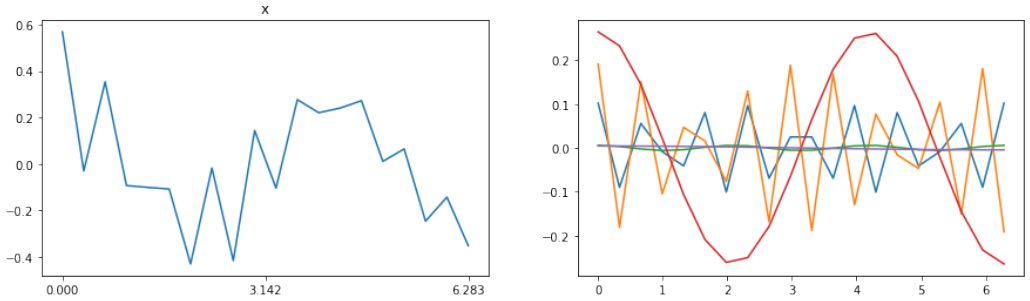


Figure 2.6: Left: an example of a synthetically generated signal vector. Right: the component dictionary atoms that, when summed, produce the signal on the left.



Our implementation of K-SVD flattens out after 21 iterations with $E_X = 0.069$ after decreasing exponentially, as illustrated in fig. 2.7.

2.4 Stochastic Gradient Descent

SGD is another way to learn a dictionary. We can begin by regarding our problem as a simple optimization problem, with a fixed representation matrix A:

$$\min_B ||X - BA|| : ||b_i|| = 1, \forall i$$

If normalized dictionary atoms were not a constraint, this would technically be solvable by a simple gradient descent iteration over all the variables of B. With the constraint, however, we are forced to use the slightly modified *projected* gradient descent, where after each step, we normalize all the atoms in order to project back on to the allowed solution space

Figure 2.7: The error E_X occurring in each iteration of K-SVD applied to synthetic data

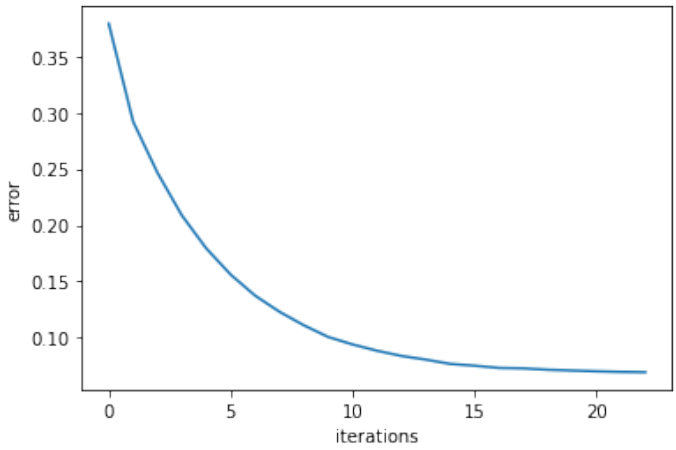
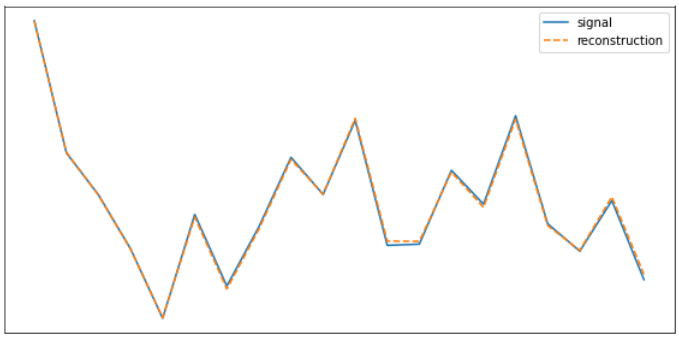


Figure 2.8: A synthetic signal vector (blue, solid) approximated by a 5-sparse coefficient vector using a 20×50 dictionary (the approximation is orange and dashed)



(the set of all dictionaries with unit columns). To formalize, we design a cost function, c_A, then use its derivatives to iterate our solution:

$$c_A(B) = ||X - BA||$$
$$B_{k+1} = B_k - \delta \nabla c_A(B_k),$$

where δ is usually some predetermined step size.

Finally, because we are dealing with a potentially vast number of training examples and dictionary variables, the standard gradient descent is simply too cumbersome memory-demanding to execute. And so, we use the Stochastic Gradient Descent (SGD) instead. In simple terms, SGD can be thought of as such: instead of considering all training examples

with each step of our optimization, we instead optimize for one training example at a time. To achieve this, we must simply re-design out cost function at each step, to only take into account the current training example:

$$c_{A,i}(B) = \|x_i - Ba_i\|$$
$$B_{k+1} = B_k - \delta \nabla c_{A,p(k)}(B_k),$$

where $p : \mathbb{N} \to \{1, \ldots, N\}$ is some endless randomization of the training data. We can visualize SGD this way: each example-specific cost function represents a different mountain we are trying to climb, with a slightly different peak. Each step we take takes us towards a different peak. We expect the overarching cost function, c_A, to have its peak somewhere in the middle of all of them, or perhaps at their mean. And so we can expect to approach it, while approaching each of the other ones in turn.

The SGD algorithm described above only optimizes the dictionary B for some fixed representation matrix A. However, we can take inspiration from the K-SVD and K-means algorithm, and optimize each of them in turn, in each iteration. We optimize A for a fixed B, then optimize B for a fixed A, and repeat. We could even use SGD to do both simultaneously, optimizing all the variables in B and A, and projecting onto the set of all unit-column dictionaries with each iteration, however this would lead to either non-sparse representation matrices, or we would have to include a sparsity constraint in our cost functions.

Algorithm 4: SGD Algorithm

Initialize dictionary B
for $i = 1 : l$ **do**
  `// define cost function`
  define $c_i(B, A) = \|x_{p(i)} - Ba_{p(i)}\| + \lambda \|a_{p(i)}\|_1$, e.g. via finite differences
  obtain $D_i = \nabla_{B,A} c_i(B_i, A_i)$
  `// optimize dictionary and representation`
  $[B_{i+1}; A_{i+1}] = [B_i, A_i] - \delta D_i$
  `// normalize dictionary atoms`
  for $k = 1 : K$ **do**
    $B^k_{i+1} = \frac{B^k_{i+1}}{\|B^k_{i+1}\|}$
  end
end

The optimization step is meant to signify that we are interpreting $[B_i, A_i]$ as a single, flattened vector. We are calculating the derivative of the cost function with respect to all variables in B and A simultaneously, and optimizing them both at the same time. λ is some arbitrarily chosen sparsity constraint.

2.5 Principal Component Analysis

Principal Component Analysis (PCA) is a method that lets us rewrite our data using a different basis such that each consecutive basis axis introduces as much variance as possible to the data.

Definition 2.5.1. *Given a set of n-dimensional data vectors $X = \{x_1, x_2, \ldots, x_M\}$, we define variance as*

$$V(X) = \sum \left(X - \bar{X}\right)^2,$$

where $\bar{X} = \frac{1}{M} \sum X$ is the mean of the data.

Loosely speaking, variance is a measure of how spread out the data is. If we take only the i-th component of each vector, we can measure how spread out the data is along the i-th axis. As we can see in fig. 2.9, we can maximize the variance along a single axis by rewriting the data using PC1 and PC2 as our axes.

The most common application of PCA is dimensionality reduction. Once we have obtained the principal components, we can simply elect only to use the first k axes to represent the data. In the case of fig. 2.9, we would keep most of the information available by projecting the data onto PC1.

In order to compute the principal components, we first form a covariance matrix C:

$$C = \frac{1}{n-1} \left(X - \tilde{X}\right)^* \left(X - \tilde{X}\right),$$

then computing the eigendecomposition for it:

$$CP = PD,$$

where $P = [p1, \ldots, p_n]$ is a matrix of right eigenvectors, and D is a diagonal matrix of corresponding eigenvalues. P and D should be sorted according to the magnitudes of the eigenvalues, starting with the highest. Each eigenvector represents one principal component, and they are now sorted in a decreasing order of significance - meaning that truncating all but the first k components will keep as much variance (or energy) as is possible in k dimensions. In other words, we may define the transformation matrices $P_k = [p_1, \ldots p_k]$ and then write

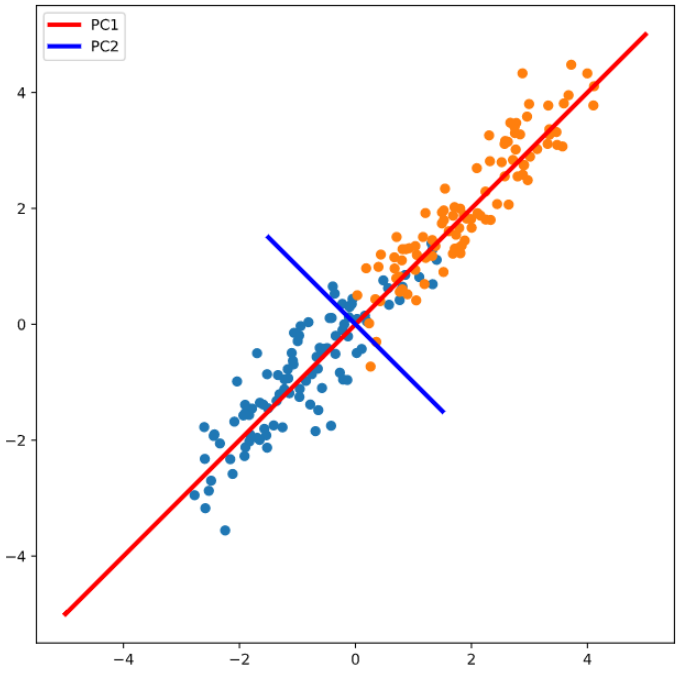
$$Y = P^\top X$$
$$Y_k = P_k^\top X.$$

It can be shown that Y_k are the representations of X in k dimensions with the highest possible variance and, thus, will retain the most information:

$$V\left(P_k^\top X\right) \geq V\left(\tilde{P}_k^\top X\right),$$

for all orthonormal $n \times k$ matrices $\tilde{P}_k$.

Figure 2.9: PCA applied to a 2-dimensional data set. PC1 and PC2 are the new axes - the data is highly spread out (has high variance) along PC1. We can rely on information from PC1 alone in order to classify the data and predict whether it belongs to the orange or the blue category.



Chapter 3

Tensor-based methods

Tensors are useful tools in data processing, because data so often appears in the shape of multidimensional arrays. For example, RGB images are essentially a $m \times n \times 3$ tensors (see fig. 3.1). One way to approach this problem is to simply unravel this tensor into a 2D matrix, or even a 1D vector, as we have done to satisfy the input format of all algorithms discussed so far. However, keeping the original structure of the input data can sometimes increase precision, or decrease computation time by simplifying calculations. First, we will go over some basics of tensor arithmetic.

3.1 Overview of tensor arithmetic

Tensors are n-order data structures, in the sense that scalars are 0-dimensional, vectors are 1-order and matrices are 2-order. All three of these can be thought of as low-order tensors, in the same way that a column vector may be thought of as an $n \times 1$ matrix.

Mode-r fibers of a tensor $\mathcal{A} = [a_{\alpha_1,\ldots,\alpha_R}] \in \mathbb{R}^{M_1 \times \ldots \times M_R}$ are M_r-dimensional vectors obtained by fixing every index but α_r.

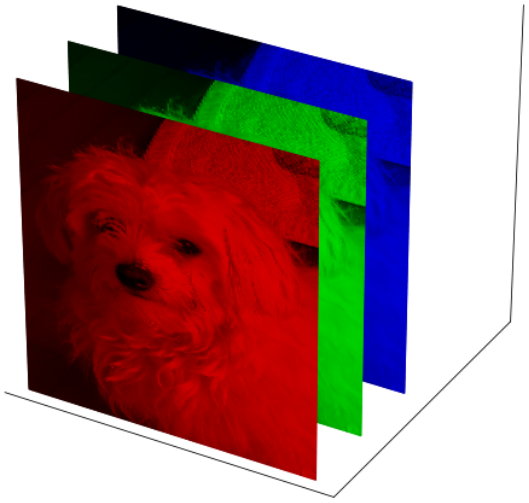
The mode-r unfolding of a tensor is a transformation that converts an R-dimensional tensor $\mathcal{A} \in \mathbb{R}^{M_1 \times \ldots \times M_R}$ into a matrix $A_{(r)} \in \mathbb{R}^{M_r \times (M_1 M_2 \ldots M_{r-1} M_{r+1} \ldots M_R)}$ where the r-th index is used as a row index, and all other indices are aligned along the columns in reverse cyclical ordering. The r-mode folding denotes the inverse of that transformation and is denoted by $[\mathcal{A}]_r^{-1}$, so that

$$\left[[\mathcal{A}]_{(r)} \right]_{(r)}^{(-1)} = \mathcal{A}.$$

The r-mode product between a tensor $\mathcal{A}$ and a matrix $U \in \mathbb{R}^{N_r \times M_r}$ is defined as

$$\mathcal{A} \times_r U = \left[U \, [\mathcal{A}]_{(r)} \right]_{(r)}^{(-1)} \in \mathbb{R}^{M_1 \times \ldots \times N_r \times \ldots \times M_R},$$

Figure 3.1: An RGB image decomposed into its three color channels



in other words, we would first r-mode unfold the tensor, then perform regular matrix multiplication, then r-fold it back into a tensor. The r-th dimension is changed to match the first dimension of U.

The Kronecker product between matrices $A \in \mathbb{R}^{I \times J}$ and $B \in \mathbb{R}^{K \times L}$ is a matrix of size $(I \cdot K) \times (J \cdot L)$ defined by:

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1J}B \\ a_{21}B & a_{22}B & \cdots & a_{2J}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}B & a_{I2}B & \cdots & a_{IJ}B \end{bmatrix}.$$

Thus, the Kronecker product of two vectors is just their outer product - a rank-1 matrix.

3.2 Generalized Tensor Compressive Sensing

In [13], Schonfield and Friedland construct a tensor-based framework for compressive sensing. The idea is based on something called the 'CANDECOMP/PARAFAC tensor decomposition' (a.k.a. tensor rank decomposition, or TRD), which is itself a generaliza-

tion of the SVD decomposition discussed in sec. 2.2. TRD of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_M}$ has the following form:

$$\mathcal{A} = \sum_{i=1}^{r} \lambda_r a_{1,i} \otimes a_{2,i} \otimes \cdots \otimes a_{M,i},$$

where M is the order of the tensor and $a_{m,i} \in \mathbb{R}^{I_m}$ are vectors. In matrix rank terms, $\mathcal{A}$ can be said to have rank $\min_m I_m$. And if $M = 2$, TRD is just SVD.

Two more results are needed in order to explain generalized tensor compressive sensing (GTCS):

Theorem 3.2.1. *Let $X = [x_{ij}] \in \mathbb{R}^{N_1 \times N_2}$ be S-sparse. Let $U_i \in \mathbb{R}^{m_i \times N_i}$ and assume U_i has NSP_S property for $i = 1, 2$. Define*

$$Y = [y_{pq}] = U_1 X U_2^\top \in \mathbb{R}^{m_1 \times m_2}.$$

Then X can be recovered uniquely using the following procedure. Let $y_1, \ldots, y_{m_2} \in \mathbb{R}^{m_1}$ be the columns of Y. Let $z_i^* \in \mathbb{R}^{N_1}$ be a solution of

$$z_i^* = \min \left\{ \|z_i\|_1 , U_1 z_i = y_i \right\}, i = 1, \ldots, m_2. \tag{3.1}$$

Then each z_i^* is unique and S-sparse. Let $Z \in \mathbb{R}^{N_1 \times m_2}$ be the matrix with columns $z_1^*, \ldots z_{m_2}^*$. Let $w_1^\top, \ldots, w_{N_1}^\top$ be the rows of Z. Then the j-th row of X is the solution $u_j^* \in \mathbb{R}^{N_2}$ of

$$u_j^* = \min \left\{ \left\| u_j \right\|_1 , U_2 u_j = w_j \right\}, j = 1, \ldots, N_1. \tag{3.2}$$

Proof. Let $Z = XU_2^\top \in \mathbb{R}^{N_1 \times m_2}$. Assume that $z_1^*, \ldots, z_{m_2}^*$ are the columns of Z. Note that z_i^* is a linear combination of the N_2 columns of X, given by the i^{th} row of U_2. Since X is S-sparse, z_i^* has at most s nonzero entries. Note that $Y = U_1 Z$, it follows that $y_i = U_1 z_i^*$. Since U_1 has NSP_S, we deduce the equality (3.1). Observe next that $Z^\top = U_2 X^\top$. Hence the column w_j of $Z^\top$ is $w_j = U_2 u_j^*$. Since X is S-sparse, each u_j^* is S-sparse. The assumption that U_2 has NSP_S property implies (3.2). □

Theorem 3.2.2. *Let $\mathcal{X} = [x_{i1}, \ldots, x_{id}] \in \mathbb{R}^{N_1 \times N_d}$ be S-sparse. Let $U_i \in \mathbb{R}^{m_i \times N_i}$ and assume that U_i has NSP_S for $i = 1, \ldots, d$. Define*

$$\mathcal{Y} = [y_{j1}, \ldots, y_{jd}] = \mathcal{X} \times_1 U_1 \times \cdots \times_d U_d \in \mathbb{R}^{m_1 \times \cdots \times m_d}.$$

Then $\mathcal{X}$ can be recovered uniquely from $\mathcal{Y}$ using the following recursive procedure. Unfold $\mathcal{Y}$ in mode 1,

$$Y_{(1)} := [\mathcal{Y}]_{(1)} = U_1 [\mathcal{X}]_{(1)} \left[\otimes_{k=d}^{2} U_k \right]^\top \in \mathbb{R}^{m_1 \times (m_2 \cdot \ldots \cdot m_d)}.$$

Let $y_1, \ldots, y_{m_2 \cdot \ldots \cdot m_d}$ *be the columns of* $Y_{(1)}$. *Then* $y_i = U_1 z_i$ *where each* $z_i \in \mathbb{R}^{N_1}$ *is* S*-sparse. Recover each* z_i *using sparse coding (discussed at length in sec. 1.5). Let* $\mathcal{Z} = \mathcal{X} \times_2 U_2 \times \cdots \times_d U_d \in \mathbb{R}^{N_1 \times m_2 \times \cdots \times m_d}$ *with its mode-1 fibers being* $z_1, \ldots, z_{m_2 \cdot \ldots \cdot m_d}$. *Unfold* $\mathcal{Z}$ *in mode 2,*

$$Z_{(2)} := [\mathcal{Z}]_{(2)} = U_2 [\mathcal{X}]_{(2)} \left[\otimes_{k=d}^{3} U_k \otimes I \right]^\top \in \mathbb{R}^{m_2 \times (N_1 \cdot m_3 \cdot \ldots \cdot m_d)}.$$

Let $w_1, \ldots, w_{N_1 \cdot m_3 \cdot \ldots \cdot m_d}$ *be the columns of* $Z_{(2)}$. *Then* $w_j = U_2 v_j$ *where each* $v_j \in \mathbb{R}^{N_2}$ *is* S*-sparse. Recover each* v_j *using sparse coding. Continue the above procedure for mode* $3, \ldots, d$ *and* $\mathcal{X}$ *can be reconstructed in series.*

In their paper, Schonfield and Friedland compare their method to another tensor-based compressed sensing approach called 'Kronecker compressive sensing', and surpass it on both accuracy and speed.

3.3 Tensor-Based Dictionary Learning

Problem (1.1) - transforming a set of data into a domain where it is sparsely represented - can be formulated in terms of tensors instead of matrices. This is precisely the topic of [19] (Quan, Huang, Ji). They look at dynamic textures - series of images that represent a short video clip or animation, so each texture has a temporal component - and attempt to develop a method analog to K-SVD (discussed in sec. 2.3). For a data tensor $\mathcal{X} \in \mathbb{R}^{M_H \times M_V \times M_T \times N}$, the expression to be optimized is given as

$$\min \|\mathcal{X} - C \times_1 D_H \times_2 D_V \times_3 D_T\|_F^2, \tag{3.3}$$

subject to the condition that $D_H \in O(M_H, \mathbb{R})$, $D_V \in O(M_V, \mathbb{R})$, $D_T \in O(M_T, \mathbb{R})$, where

$$O(M, \mathbb{R}) = \left\{ D \in R^{M \times M} : D^\top D = D D^\top = I \right\}$$

is the orthogonal group, $C \in \mathbb{R}^{M_h \times M_v \times M_T \times N}$, and finally $\left\| [C]_{(4)(i)} \right\|_0 \leq S$ for all possible i. In this expression, C is the sparse coding tensor, and there is a separate dictionary for each mode of the data tensor - the two spatial dimensions, as well as the temporal dimension. The algorithm to optimize (3.3) is analogous to K-SVD. In fact, if $M_V = M_T = 1$, then they are the same algorithm, so this is a generalization.

Learning algorithm

First, all three dictionaries are initialized to some starting values. Then, sparse coding is employed in order to solve

$$\min_C \|\mathcal{X} - C \times_1 D_H \times_2 D_V \times_3 D_T\|_F^2 \tag{3.4}$$

for fixed D_H, D_V, D_T. Afterwards, the dictionaries are updated by optimizing the following set of expressions individually:

$$\begin{cases} D_T := \text{argmin}_{D \in O(M_T, \mathbb{R})} \|\mathcal{X} - C \times_1 D_H \times_2 D_V \times_3 D\|_F^2 \\ D_V := \text{argmin}_{D \in O(M_V, \mathbb{R})} \|\mathcal{X} - C \times_1 D_H \times_2 D \times_3 D_T\|_F^2 \\ D_H := \text{argmin}_{D \in O(M_H, \mathbb{R})} \|\mathcal{X} - C \times_1 D \times_2 D_V \times_3 D_T\|_F^2 \end{cases} . \tag{3.5}$$

These two steps are repeated (alternated) some pre-determined K number of iterations. [19] additionally names and provides two propositions that validate the steps listed above - they show that explicit solutions for (3.4) and (3.5) exist, and that SVD may be used to obtain them. We will also outline them here:

Proposition 3.3.1. *Given $\mathcal{X} \in \mathbb{R}^{M_H \times M_V \times M_T \times N}$, $D_H \in S_{M_H}, D_V \in S_{M_V}$ and $D_T \in S_{M_T}$, the minimization problem*

$$\underset{C \in \mathbb{R}^{M_H \times M_V \times M_T \times N}}{\text{argmin}} \|\mathcal{X} - C \times_1 D_H \times_2 D_V \times_3 D_T\|_F^2$$

subject to $\left\|[C]_{(4)(i)}\right\|_0 \leq T$ for all possible i, has an explicit solution given by

$$C^* = \left[T_S\left([\mathcal{X} \times_3 D_T^\top \times_2 D_V^\top \times_1 D_H^\top]_{(4)}\right)\right]_{(4)}^{-1},$$

where $T_S(\cdot)$ denotes the operator that keeps the largest S elements of each row of the matrix in terms of magnitudes while setting the rest to zero.

Proposition 3.3.2. *Let $\{D_r : D_r \in O(M_r, \mathbb{R})\}_{r=1}^R$ be a set of orthogonal matrices. Given $\mathcal{X}, C \in \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_R \times N}$, the minimization problem*

$$\underset{A \in S_{M_r}}{\text{argmin}} \|\mathcal{X} - C \times_1 D_1 \cdots \times_{r-1} D_{r-1} \times_r A \times_{r+1} \cdots \times_R D_R\|_F^2$$

has an explicit solution given by $A = PQ^\top$, where P and Q denote the orthogonal matrices defined by the following SVD:

$$[\mathcal{X} \times_R D_R^\top \times_{R-1} D_{R-1} \cdots \times_{r+1} D_{r+1}^\top]_{(r)} [C \times_1 D_1 \cdots \times_{r-1} D_{r-1}]_{(r)}^\top = P\Sigma Q^\top$$

3.4 Multilinear Principal Component Analysis

Multilinear Principal Component Analysis (MPCA) is a generalization of the concepts discussed in section 2.5 to the domain of tensors. The core idea remains the same: find some new set of orthogonal bases (axes) to represent the data in, such that they consecutively

maximize its variance. Then, the may represent the data using only the first few principal component axes, trusting that we will still keep most of the information. Let us state the problem explicitly. Let $\mathcal{X} = \{\mathcal{X}_1, \ldots, \mathcal{X}_M\}$, be a dataset of tensor objects with matching dimensions: $\forall i, \mathcal{X}_i \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. Our objective is to find a multilinear transformation $\left\{\tilde{U}^{(n)} \in \mathbb{R}^{I_n \times P_n}, n = 1, \ldots, N\right\}$ that maps the original tensor space $\mathbb{R}^{I_1} \otimes \cdots \otimes \mathbb{R}^{I_N}$ into a tensor subspace $\mathbb{R}^{P_1} \otimes \cdots \otimes \mathbb{R}^{P_N}$ (where $P_n < I_n$, for $n = 1, \ldots, N$: $\mathcal{Y}_m = \mathcal{X}_m \times_1 \tilde{U}^{(1)^\top} \times_2 \tilde{U}^{(2)^\top} \cdots \times_N \tilde{U}^{(N)^\top}, m = 1, \ldots, M$ such that $\left\{\mathcal{Y}_m \in \mathbb{R}^{P_1} \otimes \cdots \otimes \mathbb{R}^{P_N}, m = 1, \ldots, M\right\}$ captures most of the variance observed in the original tensor objects, assuming that variance are measured by the total tensor scatter $\Psi_{\mathcal{Y}}$, where

$$\Psi_{\mathcal{Y}} = \sum_{m=1}^{M} \left\| \mathcal{Y}_m - \bar{\mathcal{Y}} \right\|_F^2,$$

$$\bar{\mathcal{Y}} = \frac{1}{M} \sum_{m=1}^{M} \mathcal{Y}_m$$

or, in other words, we must optimize the following expression:

$$\left\{\tilde{U}^{(n)}, n = 1, \ldots, N\right\} = \underset{\tilde{U}^{(1)}, \ldots, \tilde{U}^{(N)}}{\operatorname{argmax}} \Psi_{\mathcal{Y}}. \tag{3.6}$$

One solution to the problem is obtained by optimizing the projection matrices $\tilde{U}_n$ one by one, similar to the alternating least squares method. Updating all N matrices constitutes one cycle or iteration, and we repeat the process for some predetermined number of iterations K, or until we see no more improvement in the representation (which we can determine by keeping track of $\Psi_{\mathcal{Y}}$ throughout the process. The method depends on the following critical result:

Theorem 3.4.1. *Let $\left\{\tilde{U}^{(n)}, n = 1 \ldots, N\right\}$ be the solution to eq. (3.6). Then, for given projection matrices $\tilde{U}^{(1)}, \ldots, \tilde{U}^{(n-1)}, \tilde{U}^{(n+1)}, \ldots, \tilde{U}^{(N)}$, the matrix $\tilde{U}^{(n)}$ consists of the P_n eigenvectors corresponding to the largest P_n eigenvalues of the matrix*

$$\Phi^{(n)} = \sum_{m=1}^{M} (X_{m(n)} - \bar{X}_{(n)}) \cdot \tilde{U}_{\Phi^{(n)}} \cdot \tilde{U}_{\Phi^{(n)}}^{\top} \cdot (X_{m(n)} - \bar{X}_{(n)})^{\top}, \tag{3.7}$$

where

$$\tilde{U}_{\Phi^{(n)}} = \left(\tilde{U}^{(n+1)} \otimes \tilde{U}^{(n+2)} \otimes \cdots \otimes \tilde{U}^{(N)} \otimes \tilde{U}^{(1)} \otimes \tilde{U}^{(2)} \otimes \cdots \otimes \tilde{U}^{(n-1)}\right).$$

See [14] for proof and more details. With that in mind, we may now construct the MPCA algorithm:

Algorithm 5: MPCA

// center the data
$\bar{\mathcal{X}} = \frac{1}{M}\sum_{m=1}^{M} \mathcal{X}_m$
$\bar{\mathcal{X}}_m = \mathcal{X}_m - \bar{\mathcal{X}}$
// initialize
set $\tilde{U}^{(n)}$ to consist of the eigenvectors corresponding to the most significant P_n eigenvalues of $\Phi^{(n)*} = \sum_{m=1}^{M} \bar{X}_{m(n)} \cdot \bar{X}_{m(n)}^{\top}$, for $n = 1, \ldots, N$
// optimize
calculate $\left\{\bar{\mathcal{Y}}_m = \bar{\mathcal{X}}_m \times_1 \tilde{U}^{(1)^{\top}} \times_2 \tilde{U}^{(2)^{\top}} \cdots \times_N \tilde{U}^{(N)^{\top}}, m = 1, \ldots, M\right\}$
calculate $\Psi_{\bar{\mathcal{Y}}_0} = \sum_{m=1}^{M} \left\|\bar{\mathcal{Y}}_m\right\|_F^2$
for $k = 1 : K$ **do**
  // update projection matrices
  for $n = 1 : N$ **do**
    set $\tilde{U}^{(n)}$ to consist of the P_n eigenvectors corresponding to the largest P_n eigenvalues of the matrix $\Phi^{(n)}$ as defined in eq. (3.7)
  end
  recalculate $\left\{\bar{\mathcal{Y}}_m, m = 1, \ldots, M\right\}$ and $\Psi_{\mathcal{Y}_k}$
  if $\left|\Psi_{\mathcal{Y}_k} - \Psi_{\mathcal{Y}_{k-1}}\right| < \epsilon$ **then**
    break
  end
end
// output
the feature tensor after projection is obtained as $\left\{\mathcal{Y}_m = \mathcal{X}_m \times_1 \tilde{U}^{(1)^{\top}} \times_2 \tilde{U}^{(2)^{\top}} \cdots \times_N \tilde{U}^{(N)^{\top}}, m = 1, \ldots, M\right\}$

```python
import numpy as np
import tensorly as tl

def mpca(data, target_dims, max_iter=50):
    """Computes a multilinear principal component analysis
    for a set of tensor-shaped data points.

    Arguments:
    data: a list of numpy arrays. All should have the same
        shape (D_1, D_2,..., D_n)
    target_dims: the desired projection dimensions
        (P_1, P_2,..., P_n). Each P_i should be smaller than the
        corresponding D_i.
```

```
    max_iter: maximum number of iterations after which the process
        stops. The process will stop earlier if the projection variance
    stops improving.
    """
    N = len(data[0].shape)  # data tensor order
    M = len(data)           # number of data tensors

    # Center data
    X_mean = sum(data)/M

    # Initialize projection matrices U_n
    U = []
    for n in range(N):
        # Define Phi_n
        Phi_n=sum([tl.unfold(X-X_mean, n)@(tl.unfold(X, n).transpose())
                   for X in data])

        # Get eigenvectors and sort them
        eig_val, eig_vec = np.linalg.eig(Phi_n)
        p = np.argsort(np.abs(eig_val))[::-1]
        eig_vec = eig_vec[p]

        # Define U_n as first P_n eigenvectors
        U_n = eig_vec[:, :target_dims[n]]
        U.append(U_n)

    # Calculate projection Y
    Y = []
    for m in pb(range(M)):
        Y_m = r_mode_product(data[m], U[0].T, 0)
        for n in range(1, N):
            Y_m = r_mode_product(Y_m, U[n].T, n)
        Y.append(Y_m)

    # Calculate Psi
    Psi = []
    Psi.append(get_variance(Y))

    # Main optimization loop:
    for k in range(max_iter):
        for n in range(N):

            # Obtain projection matrix U_phi_n
            U_phi_n = np.ones((1, 1))
            for n2 in range(n+1, N):
                U_phi_n = np.kron(U_phi_n, U[n2])
            for n2 in range(0, n):
```

```
                U_phi_n = np.kron(U_phi_n, U[n2])

            # Obtain Phi_n
            Phi_n = sum([tl.unfold(X-X_mean, n)@U_phi_n@U_phi_n.T @
                         tl.unfold(X-X_mean, n).T for X in data])

            # Update U_n:
            # 1. solve eigenproblem for Phi_n
            eig_val, eig_vec = np.linalg.eig(Phi_n)
            p = np.argsort(np.abs(eig_val))[::-1]
            eig_vec = eig_vec[:, p]

            # 2. define U_n as first P_n eigenvectors
            U[n] = eig_vec[:, :target_dims[n]]

        # Recalculate projection Y
        Y = []
        for m in range(M):
            Y_m = r_mode_product(data[m], U[0].T, 0)
            for n in range(1, N):
                Y_m = r_mode_product(Y_m, U[n].T, n)
            Y.append(Y_m)

        # Recalculate Psi
        Psi.append(get_variance(Y))
        # If there is no longer any variance gain, return
        if Psi[-1] < Psi[-2]:
            break

    return U, Psi
```

Testing on synthetic data

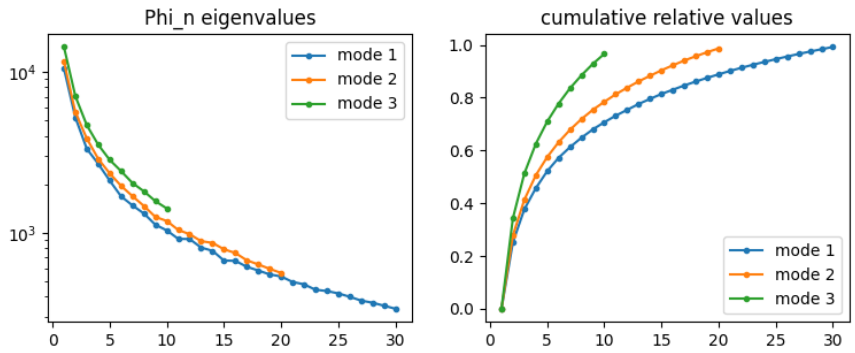
In order to test the MPCA algorithm, we have applied it to the same synthetic data as that used in [14]. First, a tensor $\mathcal{B}_m \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is randomly generated with each entry being drawn from a normal distribution with mean 0 and variance 1. Then, using some random orthogonal matrices $C_n \in \mathbb{R}^{I_n \times I_n}, n = 1, 2, 3$, the data tensor $\mathcal{A}_{\Updownarrow}$ is given as

$$\mathcal{A}_m = \mathcal{B}_m \times_1 C_1 \times_2 C_2 \times_3 C_3 + \mathcal{D}_m, m = 1, \ldots, M,$$

for M data points, where $\mathcal{D}_m$ is a noise tensor with entries drawn from a normal distribution with mean 0 and variance 0.01.

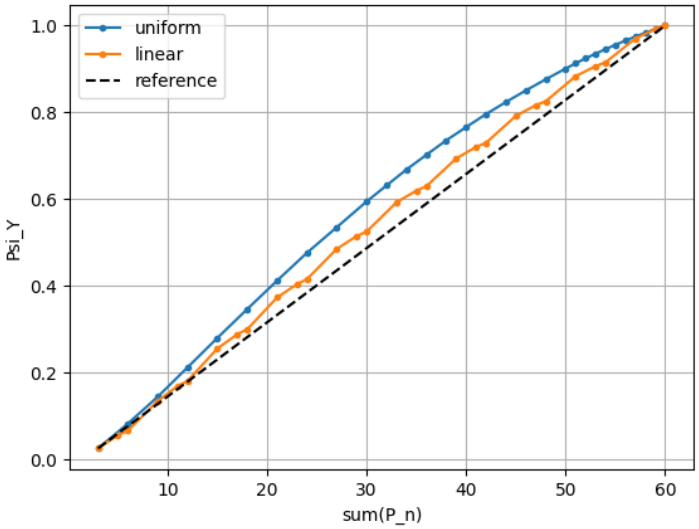
In regular, vector-based, PCA, dimensionality reduction comes down to choosing the first p principal components, with the value for p directly corresponding to the amount of variance the new model will keep. However, in MPCA, we have N diffferent values to

Figure 3.2: MPCA eigenvalue analysis on synthetic data. Left: a logarithmic plot of the magnitudes of eigenvalues of $\Phi^{(n)}$ as defined in (3.7). Right: a cumulative graph of their relative values (represented as fraction of the sum total)



choose - not just one. Looking at fig. 3.2, we can see that the first p eigenvalues in each mode are roughly the same. As such, we can intuit that a good strategy for choosing the truncated dimensions $P_1, \ldots, P_N$ is to define an integer, p, and then set each P_n to be equal to $\min(I_n, p)$.

Figure 3.3: Variance retention relative to dimensionality. The y-axis represents the relative variance of a projected data set ($\frac{\Psi_y}{\Psi_x}$). The x-axis displays the sum of the dimensions used in the projection ($\sum_n P_n$). The two colored lines represent different strategies of picking the number of dimensions in each mode. 'uniform' attempts to pick the same number of dimensions in each mode (for example, (9,9,9), (10,10,10), (11,11,10), etc.), while 'linear' tries to pick the same fraction of the maximum possible dimensions in each mode (for example (3,2,1), (6,4,2), (9, 6, 3) etc. 'reference' is just a straight line, representing what the variance would look like if we just randomly assigned the dimensions P_n.



Sažetak

Iako ovo nije cjelokupan pregled, proučili smo nekolicinu efektivnih metoda za rijetku reprezentaciju podataka, tj. rješavanje $\min_a \|x - Ba\|$ sa uvjetom rijetkosti na a te povezane probleme. Počeli smo u poglavlju 1 objašnjavajući metode koje se najčešće koriste u modernom procesiranju signala, kao što su DCT i DFT, te smo upoznali neke općenite metode za rijetke reprezentacije - matching pursuit i basis pursuit. Spomenuli smo neke od problema koji su sadržani u ovom području, pogotovo činjenicu da je suštinski problem NP-težak, te kako se suočiti sa tim problemima.

U poglavlju 2, upoznali smo se sa nekolicinom metoda za optimizaciju ne samo rijetke reprezentacije, nego i rječnika B. Diskutirajući k-means i SVD, izgradili smo znanje potrebno za razumjevanje K-SVD algoritma, te smo ga testirali na sintetičkim podacima. Također smo ukratko pregledali još jedan pristup sa širokom primjenom - 'principal component analysis', nužan za razumjevanje MPCA algoritma kojeg spominjemo kasnije.

U zadnjem poglavlju, poglavlju 3, ulazimo u domenu tenzora. Ovo je prirodan korak budući da se puno stvarnih podataka, pogotovo vizualnih, može puno prirodnije prikazati u obliku tenzora višeg reda. Ponovili smo malo osnovne aritmetike tenzora, te upoznali dvije metode - jedna je metoda za učenje rječnika napravljena za analizu dinamičkih tekstura, a druga je multilinearna PCA, generalizacija PCA algoritma iz prethodnog poglavlja u prostor tenzora.

Primjene rijetkih reprezentacija su široke, i nove metode se razvijaju vrlo brzo kako bi se suočili sa modernim problemom sve veće količine podataka koje prikupljamo i šaljemo preko telekomunikacijskih mreža, koje treba analizirati i komprimirati.

Summary

While this is by no means an exhaustive review, we have overviewed a number of effective methods for representing data sparsely, which is to say, solving $\min_a \|x - Ba\|$ with sparsity constraints on a and related problems. We started in chapter 1 by going over methods most widely used in modern signal processing, such as the DCT and DFT, and then introduced some general-purpose methods for sparse coding - namely matching pursuit and basis pursuit. We touched on some of the problems inherent in sparse coding (the fact that the base problem is NP-hard), and how to circumvent those issues.

In chapter 2, we introduced a number of methods for not only optimizing the sparse representation, but the dictionary B as well, and sometimes simultaneously. By discussing k-means clustering and singular-value decompositions, we built the knowledge base necessary to understand the K-SVD algorithm, and we tested it on some synthetic data. We also briefly overviewed another approach with broad applications - the principal component analysis, necessary to understand MPCA discussed later on.

In the last chapter, chapter 3, we enter the domain of tensors. This is a natural step, as much real-world data, especially visual data, is more naturally represented as tensors of higher orders instead of vectors or matrices. We went over some basic tensor arithmetic, and then reviewed two methods. One is a dictionary learning method built to analyze dynamic textures, and the other is multilinear principal component analysis - a generalization of PCA to tensor space.

The applications of sparse coding and dictionary learning are vast, and new methods are being developed rapidly in order to deal with the modern problem of ever more data being accumulated, sent over communication channels and needing to be analyzed or compressed.

Circum Vitae

Fran Špigel was born on September 10th, 1993 in Zagreb. He went to Otok elementary school, then attended the 18th linguistic gymnasium highschool in Zagreb. In 2012, he enrolled in the mathematics section of Prirodoslovno-Matematički Fakultet (PMF) in Zagreb, and in 2017 he enrolled in the applied mathematics graduate course of the same university.

Bibliography

[1] Afonso Bandeira, Edgar Dobriban, Dustin Mixon, and William Sawin. Certifying the restricted isometry property is hard. *Information Theory, IEEE Transactions on*, 59:3448–3450, 06 2013.

[2] P. S. Bradley and Usama M. Fayyad. Refining initial points for k-means clustering. pages 91–99. Morgan kaufmann, 1998.

[3] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[4] E. J. Candes and T. Tao. Decoding by linear programming. *IEEE Transactions on Information Theory*, 51(12):4203–4215, 2005.

[5] Scott Shaobing Chen, David L. Donoho, and Michael A. Saunders. Atomic decomposition by basis pursuit. *SIAM J. Sci. Comput.*, 20(1):33–61, December 1998.

[6] Xuemei Chen, Haichao Wang, and Rongrong Wang. A null space property approach to compressed sensing with frames. 02 2013.

[7] Piotr J. Durka. Matching pursuit. *Scholarpedia*, 2(11):2288, 2007. revision #140500.

[8] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1:211–218, 1936.

[9] Fan Yang, Shengqian Wang, and Chengzhi Deng. Compressive sensing of image reconstruction using multi-wavelet transforms. In *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*, volume 1, pages 702–705, 2010.

[10] John A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[11] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[12] Syed Ali Khayam. The discrete cosine transform (dct): Theory and application1. *Course Notes, Department of Electrical & Computer Engineering*, 01 2003.

[13] Qun Li, Dan Schonfeld, and Shmuel Friedland. Generalized tensor compressive sensing. pages 1–6, 07 2013.

[14] Haiping Lu, Konstantinos N Plataniotis, and Anastasios N Venetsanopoulos. Mpca: Multilinear principal component analysis of tensor objects. *IEEE transactions on Neural Networks*, 19(1):18–39, 2008.

[15] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard, 2009.

[16] M Mozammel, Hoque Chowdhury, and Amina Khatun. Image compression using discrete wavelet transform. *International Journal of Computer Science Issues*, 9, 07 2012.

[17] B. K. Natarajan. Sparse approximate solutions to linear systems. *SIAM Journal on Computing*, 24(2):227–234, 1995.

[18] Vishal M. Patel and Rama Chellappa. *Sparse Representations and Compressive Sensing for Imaging and Vision*. Springer Publishing Company, Incorporated, 2013.

[19] Y. Quan, Y. Huang, and H. Ji. Dynamic texture recognition via orthogonal tensor dictionary learning. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 73–81, 2015.

[20] Hongyuan Zha, Xiaofeng He, Chris Ding, Horst Simon, and Ming Gu. Spectral relaxation for k-means clustering. *Adv. Neural Inf. Process. Syst.*, 14, 04 2002.

[21] Lingsong Zhang, J. Marron, Haipeng Shen, and Zhengyuan Zhu. Singular value decomposition and its visualization. *Journal of Computational and Graphical Statistics - J COMPUT GRAPH STAT*, 16, 02 2007.