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, Veljača, 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. _____

Contents

Contents **iii**

Introduction **1**

1 Sparse Representations **3**
1.1 Discrete Cosine Transform (DCT) 5
1.2 Wavelet Transform 8
1.3 Discrete Fourier Transform 9
1.4 Sparse Coding 10
1.5 Classification 13
1.6 Compressed Sensing 16

2 Dictionary Learning **19**
2.1 K-Means Clustering 19
2.2 Singular Value Decomposition 22
2.3 K-SVD Algorithm 24
2.4 Stochastic Gradient Descent 31

Bibliography **33**

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 paper, 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

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's say our signals have the form $x = (x^1, x^2, \ldots, x^n)$, 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?

Our assumption is that there is some basis $B = [b_1, b_2, \ldots, b_k]$, 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 $x_i = Ba_i, \forall i$. 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:

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

Here, $X = [x_1, x_2, \ldots, x_p]$ 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 $l \ll n$ - in other words, all representation vectors are l-sparse. The l_0-norm is defined as the limit of the l_p norm as p approaches infinity. In short, it simply counts the number of non-0 elements of a vector.

Expression 1.1 is actually composed of several significant sub-problems. One of them is more fundamental: for a given signal vector x, find its individual representation a =

$(a^1, \ldots, a^k)$ which optimizes the following:

$$\min_a \|x - Ba\|_2 : \|a\|_0 \leq L \tag{1.2}$$

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:

$$\min \|a\|_0 : x = Ba \tag{1.3}$$

"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 l-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 won't work. This makes both of these problems NP-hard ([7]). To solve this issue, we will generally substitute the l_0 norm in these problems with the l_1 norm. The l_2 norm wouldn't suit our purposes - given a particular solution space, which is an n-dimensional manifold in k dimensions, l_2 norm would simply find the solution which is closest to the origin, and that doesn't guarantee any sparsity. l_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 aren't unique minimums.

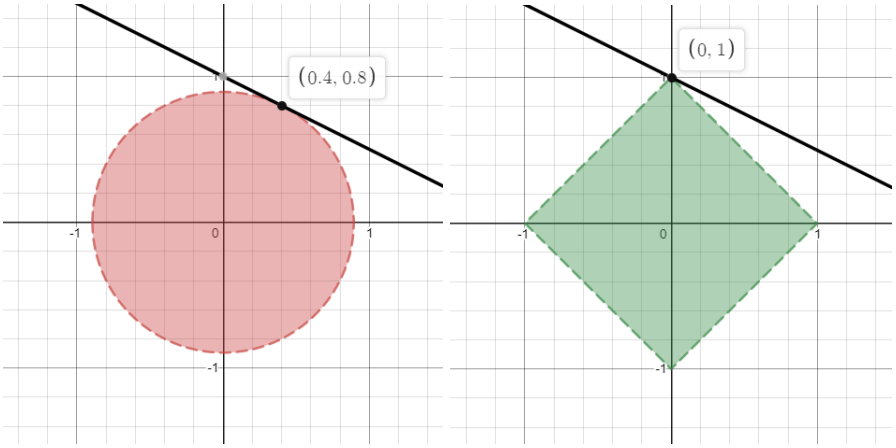
And so we re-formulate 1.1 into a very similar, but much more solvable version:

$$\min_{B,A} \|X - BA\|_F, \quad \|a_i\|_1 \leq l, \quad \forall i = 1, 2, \ldots, p \tag{1.4}$$

Yet another version is sometimes used, where the sparsity restraint is relaxed somewhat, and converted into an error factor for the cost function:

$$\min_{B,A} \|X - BA\|_F + \lambda \sum_{i=1}^{p} \|a_i\|_1 , \tag{1.5}$$

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



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

If a signal domain can be 'sparsified', if you will, the first logical application arises naturally in data compression. If a set of n-dimensional vectors can be adequately represented in k dimensions, we can immediately cut storage space requirements by a factor of $\frac{k}{n}$. Another application can be found in solving classification problems. More on that in section 1.5.

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.

1.1 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:

$$\tilde{b}_i(x) := C_i \cos(\pi x i),$$

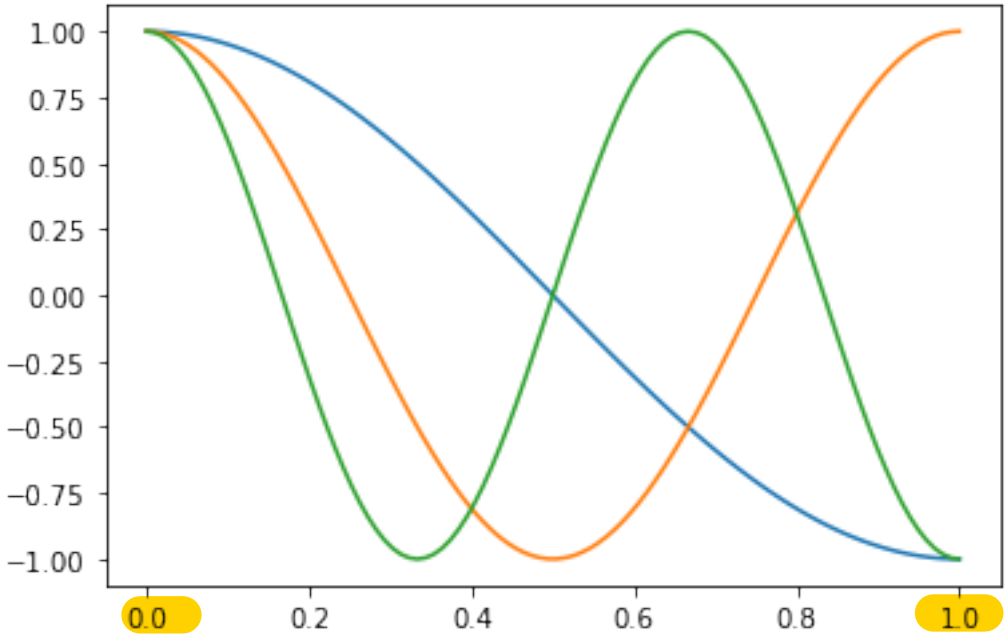
where C_i is a normalizing factor such that $\|\tilde{b}_i\| = 1, \forall i,$ regardless of the space being considered. This function would naturally be quantized into an n-dimensional, normalized vector with:

$$b_i^{*j} := b_i(jh), j = 0, 1, \ldots, n$$
$$b_i := \frac{b_i^*}{\|b_i^*\|_2}$$

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 1.4. Fig. 1.2 illustrates some bases that might be used in 1-d DCT (before normalization).

Figure 1.2: Basis functions for 1-dimensional DCT



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 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 isn't good at dealing with. Instead, we have to create 2-dimensional basis vectors. We do this by defining functions such as:

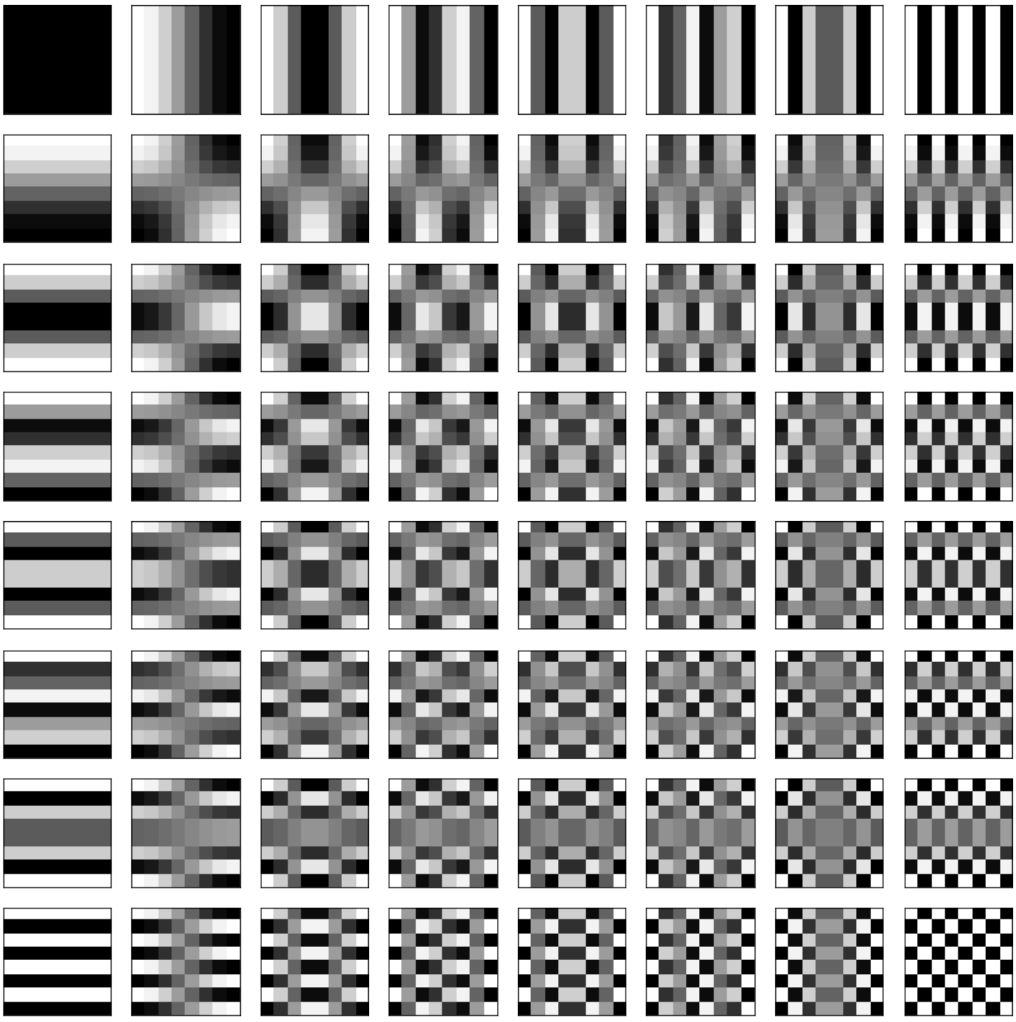
$$\tilde{b}_{i,j}(x,y) := \tilde{b}_i(x)\tilde{b}_j(y) = C_{i,j}\cos(\pi x i)\cos(\pi y j)$$

Such functions can also be quantized into matrices using:

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

Here, $\otimes$ signifies the outer product of two vectors, creating an $n \times n$ rank-1 matrix. In practice, the result looks like fig. 1.3.

Figure 1.3: Basis matrices for 2-dimensional DCT



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 most cases with natural images, the coefficients corresponding to the bases in the upper left corner of fig. 1.3 will be drastically 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 resulting coefficient matrix will be sparse, with most non-0 values located on the first row and first column.

Figure 1.4: Left: original image. Right: Image reconstructed after being compressed via DCT by a factor of 10

Fig. 1.4 shows the DCT in action. As you can see, 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. 1.3), it has trouble representing images with fast, discontinuous switches in intensity. This rarely occurs in nature, but does come up with images of text. This weakness results in so-called 'artifacts' - miscolored pixels surrounding such image patches, created because it is difficult to approximate a Heaviside function with cosine waves. This is evident in fig. 1.5.

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

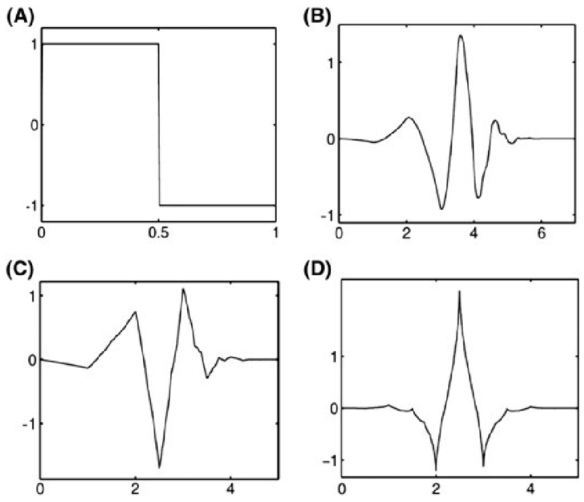
1.2 Wavelet Transform

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

Figure 1.5: Artifacts created when applying the DCT to pictures of text

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. 1.6

Figure 1.6: (A): Haar wavelet; (B): Daubechies wavelet; (C): sym3 wavelet; (D): coif1 wavelet



1.3 Discrete Fourier Transform

We would be remiss to discuss sparse coding without going over DFT. Similar to the DCT, the Fourier Transform represents functions in terms of the frequencies present in their signal - it is a spectral decomposition:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx \tag{1.6}$$

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

$$\hat{f}_k = \sum_{n=0}^{N-1} f_n \cdot e^{\frac{-i2\pi}{N} kn} \tag{1.7}$$

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

DFT is a fundamentally important result in numerical mathematics. It sees uses in a variety of applications - from data compression to solving PDE. 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 drastically faster. To illustrate the compressive potential of DFT, we can look at fig. 1.7, 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.

1.4 Sparse Coding

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

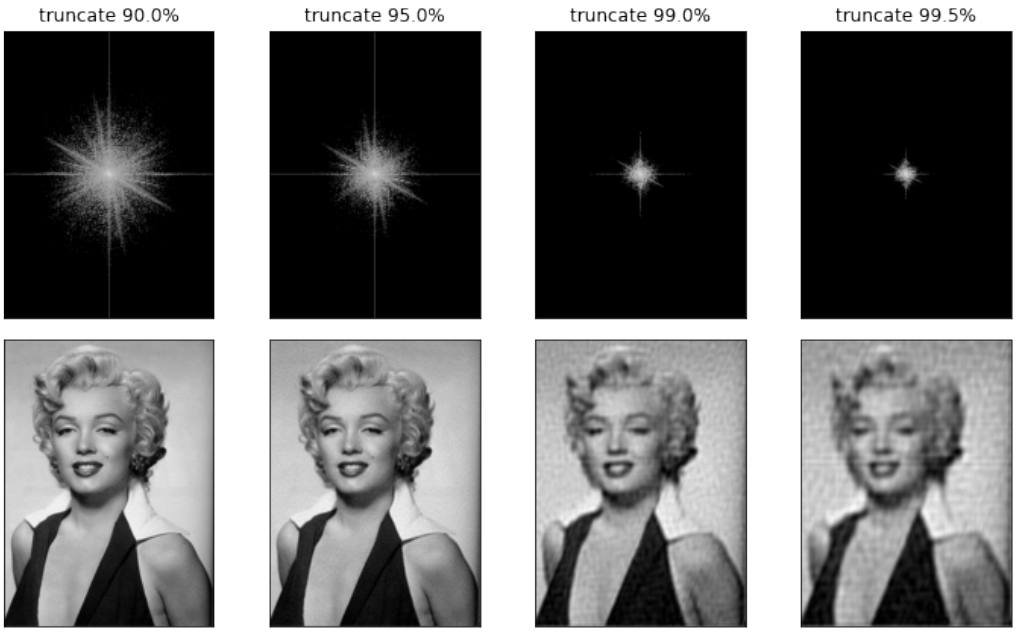
$$\min_A \left(\|X - BA\|_F \right), \quad \|a_i\|_0 \leq l, \quad \forall i \tag{1.9}$$

Here, the l_0 norm may be replaced with l_1 to simplify the calculation. The first method for this is perhaps the most obvious: matching pursuit.

Matching Pursuit

In 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

Figure 1.7: DFT-compressed picture of Marilyn Monroe. The top row represents 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.



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

Clearly, the coefficients 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 $i = 1 : l$ **do**
  find j_0 s.t. $|\langle x, b_{j_0}\rangle| \geq |\langle x, b_j\rangle|, \forall j = 1, 2, \ldots, k$;
  $a^{j_0} \leftarrow \langle x, b_{j_0}\rangle$;
  $x \leftarrow x - a^{j_0} b_{j_0}$
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 doesn't necessarily produce the globally optimal solution, but its solution is 'good enough'. Crucially, the process may be stopped at any point l' in order to obtain an l'-sparse solution. In the case of $k = l = n$, MP simply solves a complete system of linear equations.

Basis Putsuit

BP is another pursuit algorithm which takes a slightly different approach. The problem it solves 1.3:

$$\min \|a\|_1 \quad \text{s. t.} \quad x = Ba$$

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 standard form for a linear, convex optimization problem is

$$\min c^T x \quad \text{s.t.} \quad Ax = b, x \geq 0, x \in \mathbb{R}^m$$

where c^T is an objective function, $Ax = b$ is a collection of equality constraints, and $x \geq 0$ is a set of bounds. We can reformulate 1.3 thus:

$$m \Leftrightarrow 2n; \quad A \Leftrightarrow (B, -B); \quad b \Leftrightarrow x; \quad x \Leftrightarrow (u; v); \quad a \Leftrightarrow u - v \tag{1.10}$$

u and v are the positive and negative parts of a. They are concatenated into one long vector $x = (u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, v_n)$ 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 x is positive in every variable. All that remains is to set $c = (1, 1, \ldots, 1)$ and the system can be solved by any convex optimization method, such as the simplex algorithm or the inner-point method. More on BP and MP may be found in [3] or [2].

1.5 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:

$$B = \left[B_1, \ldots, B_c\right]$$
$$= \left[x_{11}, \ldots, x_{1p}|, x_{21}, \ldots, x_{2p}|, \ldots, |x_{c1}, \ldots, x_{cp}\right]$$

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

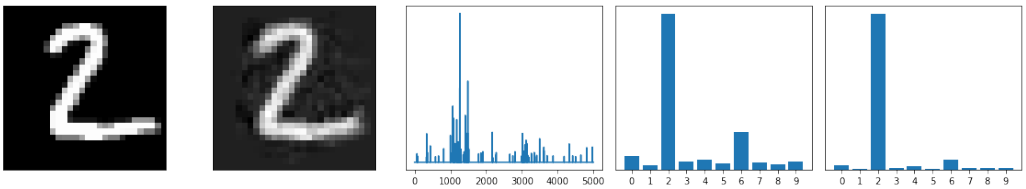
$$y = \sum_{i=1}^{c} \sum_{j=1}^{p} a_{ij} x_{ij}$$

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 cass k will lie approximately in the linear span of the training samples from that class. 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.

Figure 1.8: 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.



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 histogram, in order to suppress small values and promote high values. There are similarities between these options and l_p vector norms.

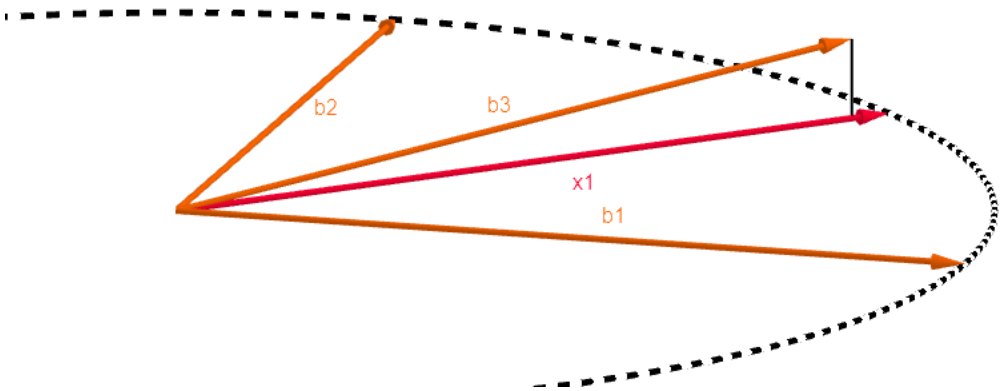
This approach is not without issues, however. If the angle between two of the class-specific linear spans ($[B_i]$ and $[B_j]$) is too small, then MP might mis-represent some signal vectors. An example of such a case is illustrated in fig. 1.9. In this case,

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

and let's 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.

so MP stops and the classification algorithm assigns x to the second class, even though it belongs to the linear span of the first class.

Figure 1.9: Sparse classification error - basis vectors are shown in orange, and the target signal vector is red.



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

$$\langle x, b_i \rangle \geq \langle x, b_j \rangle, \forall j \Rightarrow \|x - b_i\|_2 \leq \left\|x - b_j\right\|_2, \forall j$$

Proof. Assume $\langle x, b_i \rangle \geq \langle x, b_j \rangle, \forall j$. Then,

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

□

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

Remark 1.5.2. *The reasoning above doesn't 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.*

1.6 Compressed Sensing

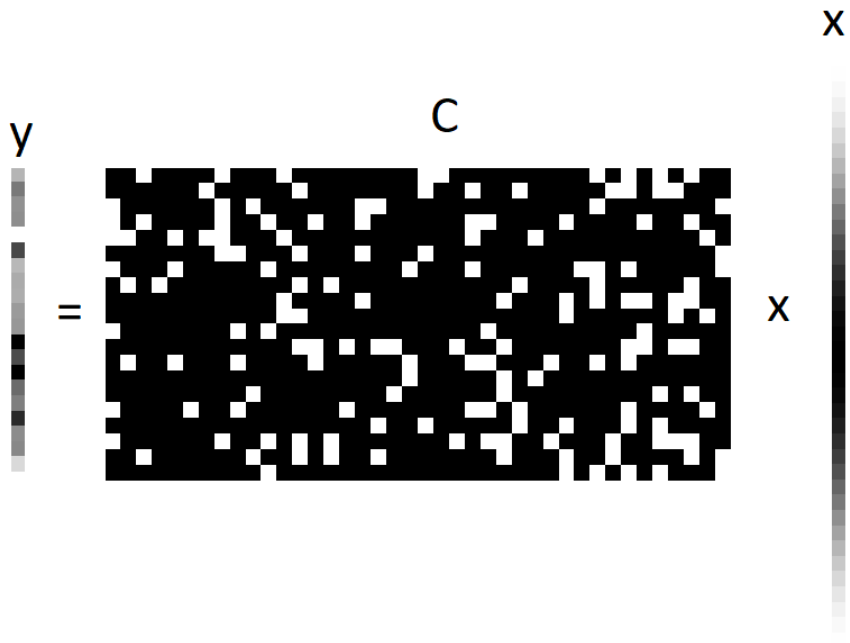
Data compression has massive real-world applications and benefits in and of itself. However, compressed sensing is an idea that builds on it even further. The core concept is simple: instead of collecting and then discarding most of it via compression, why not collect only a fragment of the data to begin 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

$$y = Cx$$

C here 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. 1.10.

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

Figure 1.10: Binomially distributed matrix taking a compressed measurement of a signal vector x



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

where $\Theta = CB$. So the task remains the same as before - finding the sparsest a which satisfies $y = \Theta a$. Until recently, this was considered to be an NP-hard problem, as discussed in 1.4.

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:

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

We will first review the K-Means clustering method, as a special case of the K-SVD algorithm introduced afterwards.

2.1 K-Means Clustering

K-Means is a simple, intuitive, yet highly funcitonal method to cluster high-dimensional data points. The task is to minimize the following error function:

$$E(S) = \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|_2 \tag{2.1}$$

Here, $S = \{S_1, \ldots, S_k\}$ is a partition of the set of data points x_i, and μ_i are the centroids of S_i, given by

$$\mu_i = \frac{\sum_{x \in S_i} x}{s_i}, \quad \forall i = 1, \ldots, k \tag{2.2}$$

s_i being the number of elements in S_i.

(2.1) is commonly referred to as the 'within-cluster sum of squares' (WCSS). Let us briefly recall the basic k-means algorithm used to attack it.

Algorithm 2: K-Means Clustering

determine initial centroids $\mu_1, \ldots, \mu_k$
for $i = 1 : l$ **do**
  for x *in* X **do**
    find closest centroid μ_i
    assign x to subset S_i
  end
  for $i = 1 : k$ **do**
    $\mu_i \leftarrow \sum_{x \in S_i} x / s_i$
  end
end

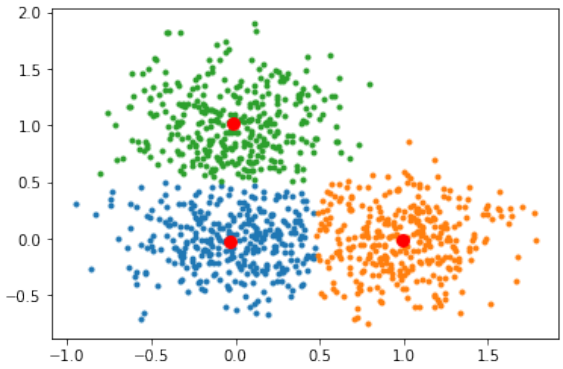
```
def kmeans(X, n, iter):
    """A basic implementation of the k-means algorithm. Clusters the
    signal vectors in X into n groups, and returns the centroid of each
    group.

    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)])

```

```
        # fix empty clusters
        for j in np.where(count==0)[0]:
            assign[random.randint(0,X.shape[0])] = j

        # recalculate centroids
        centroids = np.array([np.sum(X[assign==j], axis=0)/np.sum(assign
    ==j) for j in range(n)])

    # final assignment
    d = np.array([np.sum((X-c)**2, axis=1) for c in centroids]).
    transpose()
    assign = np.argmin(d, axis=1)
    return assign, centroids
```

Here, l 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 $h_i = (M_i - m_i)/n, \forall i$, 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:

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

where $C = \{c_1, \ldots, c_n\}$ 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.

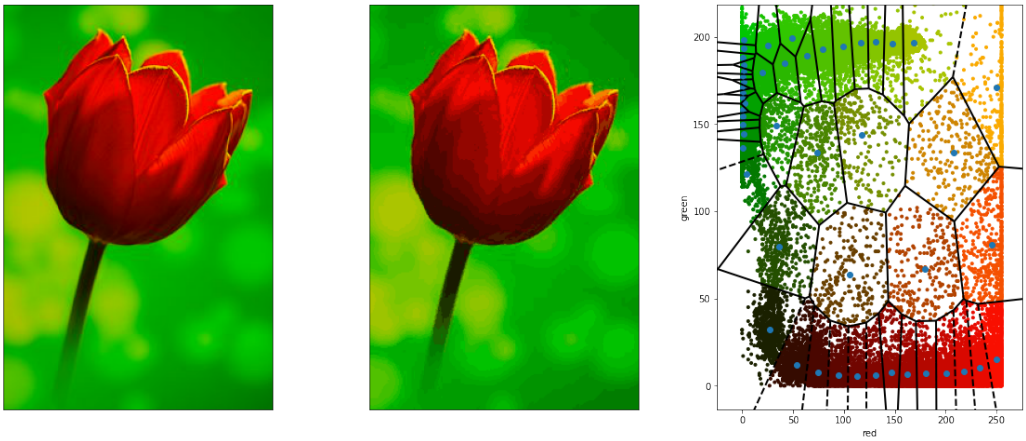


Fig. 2.2 shows how an image can be relatively faithfully reconstructed using only 50 different colors. The compression gain here is that we no longer need to store three floating-point values for each pixel, but instead a single integer with value up to 50, meaning we only need 6 bits of storage per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

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 $m \times n$ ma-

trix. The eigenvalue decomposition would factorize the matrix as $A = Q\Lambda Q^*$, where Λ is a diagonal matrix of eigenvalues, Q is a matrix such that each row is the left eigenvector corresponding to the eigenvalue in the same row of Λ and, accordingly, $Q*$ has right eigenvectors as columns. Each matrix in the decomposition has the same square dimensions as A. SVD of an $m \times n$ matrix A looks similar:

$$A = U\Sigma V^*, \tag{2.3}$$

where U and V^* are again vectors of left singular vectors, U being an $m \times m$ matrix, V an $n \times n$ matrix of right singular vectors, and Σ being an $m \times n$ rectangular 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. They still retain the property

$$Au = \sigma u; \quad A^*v = \sigma v$$

for left singular vector u and right singular vector v, both of unit length, and their corresponding singular value σ.

Furthermore, 2.3 can be rewritten as

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

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

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

it holds that A_k is the best rank-k approximation for A, and its residual is $R_k = A - A_k = \sum_{i=k+1}^{r} s_i u_i v_i^T$. In other words,

$$\underset{A:\text{rank}(A)=k}{\text{argmin}} \text{RSS}(A) = A_k, \tag{2.4}$$

with RSS being the 'residual sum of squares' error function, or the sum of the squares of elements of the residual of an approximation ([8])

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 $u_i s_i v_i^T$. 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 drop off exponentially, which is why most of the information energy is conserved in the first few.

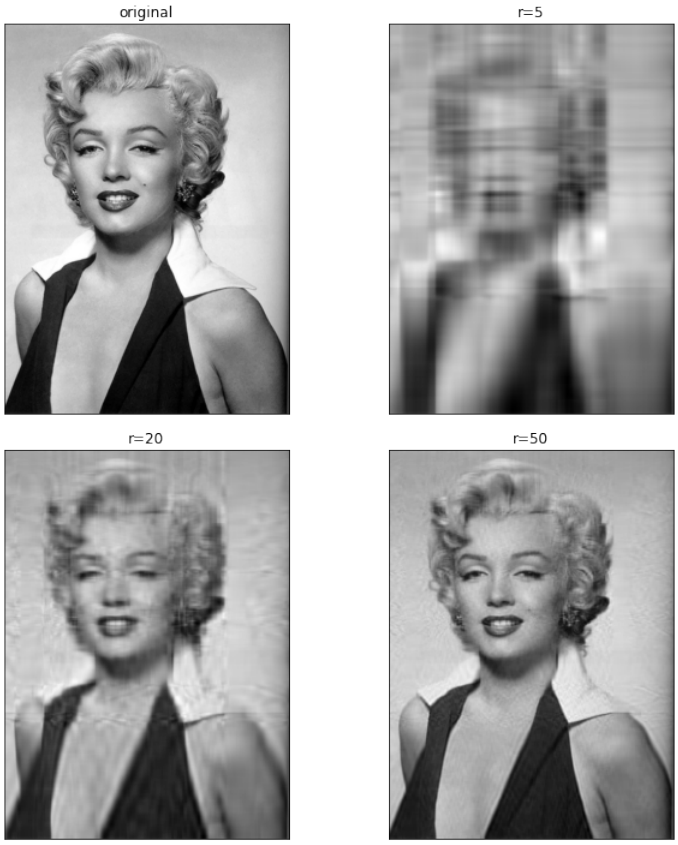
Clearly, approximating this image with only the first 50 columns yields a very faithful reproduction. Since the original dimension is 439×600, and we are now only storing one 600×50 matrix, one 439×50 matrix and a 50×1 vector, we are saving as much as 80% of the storage space, which is a remarkable compression rate.

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's 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.

Figure 2.3: A 439x600 image of Marilyn Monroe approximated with matrices of various ranks



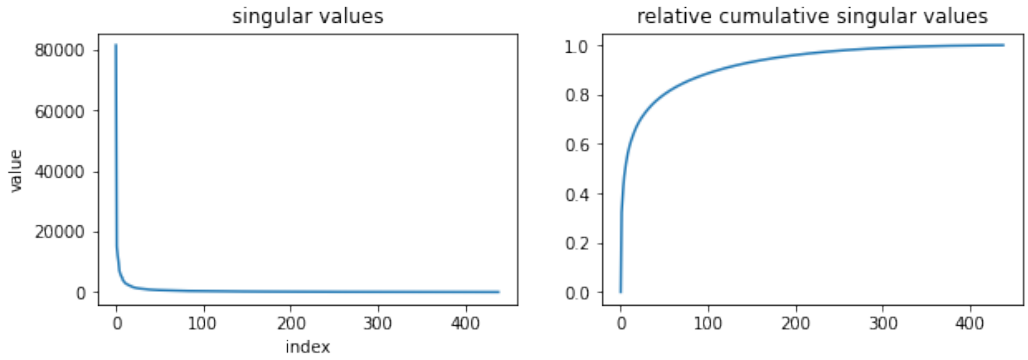
$$||X - \tilde{B}\tilde{A}|| < ||X - BA||$$

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

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

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^T represent rows of A. We can combine the first two terms, which are fixed, into a single matrix E_k:

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



$$\left\|E_k - b_k a_k^T\right\|_F^2 \tag{2.5}$$

$b_k a_k^T$ 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. 2.4). However, there is one more preparatory step to take. If we were to utilize SVD right now to optimize the expression 2.5, we would most likely end up with a full vector for a_k^T, 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 $a_k^R = x_k^T \Omega_k$, and we get

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

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 $E_k^R = U \Sigma V^T$, and recall that $u_1 s_1 v_1^T$ is the best rank-1 approximation for E_k^R (u_1 and v_1^T 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 $s_1 v_1^T$. 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**
    $a_p \longleftarrow \text{argmin}_{a:\|a\|_0 \leq l} \left\| x_p - Ba \right\|$; // e.g. via matching pursuit
  end
  // optimize dictionary
  for $k = 1 : K$ **do**
    $\omega_k \longleftarrow \{i | 1 \leq i \leq K, a_{ki} \neq 0\}$
    define Ω_k as specified earlier in the chapter
    $E_k = X - BA + b_k a_k^T$; // outer product
    $E_k^R = E_k \Omega_k$
    compute SVD: $E_k^R = U \Sigma V^T$
    $b_k \Omega_k \longleftarrow u_1$
    $a_k \Omega_k \longleftarrow s_1 v_1^T$
  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 don't
        # 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

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

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

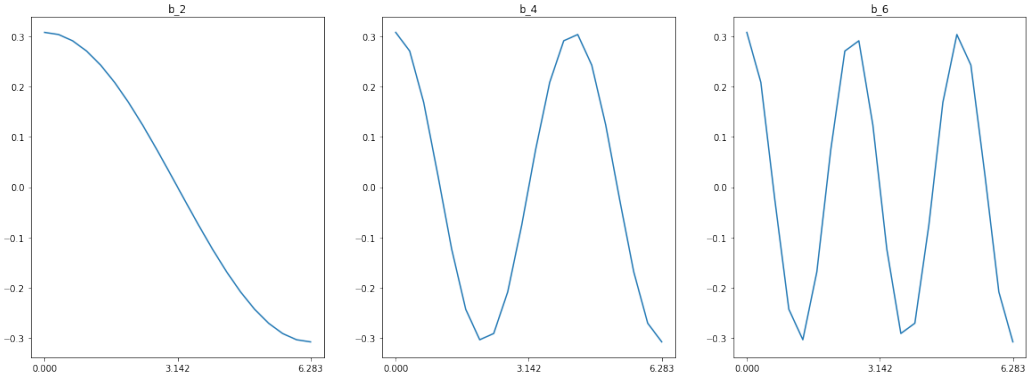
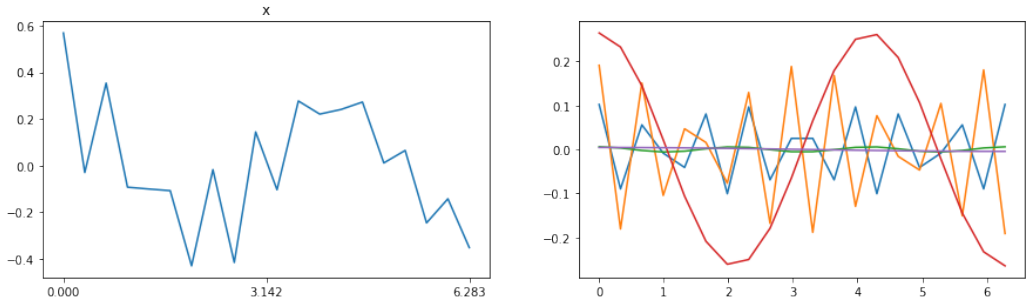


Figure 2.6: Left: an example of a synthetically generated signal vector. Right: the component dictionary atoms that, when summed, produce the signal on the left.



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:

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

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

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

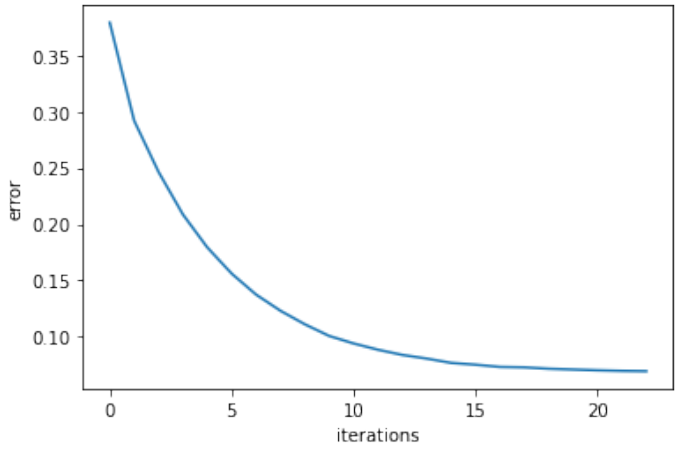
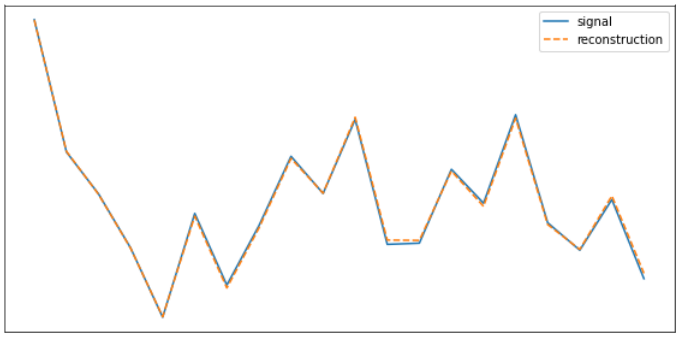


Figure 2.8: A synthetic signal vector (blue, solid) approximated by a 5-sparse coefficient vector using a 20×50 dictionary (the approximation is orange and dashed)



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 weren't a constraint, this would technically be solvable by a simpe 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 (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 pre-determined 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.

Bibliography

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

[2] 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.

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

[4] J. 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.

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

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

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

[8] 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.