Math 5467: Introduction to the Mathematics of Image and Data Analysis

Jeff Calder

University of Minnesota
School of Mathematics
jwcalder@umn.edu

May 10, 2022

Contents

Chapter 1

Introduction

1.1 Course Information

This course is a modern introduction to the mathematics of image and data analysis. The course will cover the discrete Fourier and Wavelet transforms, with applications to image and audio processing. We will also cover the mathematics of common data analysis algorithms, including principal component analysis (PCA), data ranking (e.g., Google's PageRank for ranking webpages), and clustering algorithms such as k-means and spectral clustering. Time-permitting, we will give an introduction to machine learning (ML), and cover basic ML classifiers, neural networks (in particular, convolutional neural networks for image classification), and graph-based learning.

The course will cover both mathematical theory and practical applications. We will use Python for all computational work in this course. Students will get hands on experience working with real data through a series of computational projects that will be completed throughout the term, on topics such as audio or image compression, facial recognition, or image classification. We will start the course with a gentle introduction to Python; no prior knowledge is required. See the course website, below, for details on how to get access to Python:

http://www-users.math.umn.edu/~jwcalder/5467S21/index.html

1.2 Python

These lecture notes are accompanied by Python notebooks that can be found on the course website, and in links throughout these notes. Below are links to

Google Colab Python notebooks with basic introductions to various aspects of Python programming.

Introduction to Python Notebooks:

1. Introduction to Python
2. Introduction to Numpy
3. Reading and writing images and audio in Python
4. Introduction to Pandas

Below are some exercises to be completed in Python (most are in the notebooks above as well).

Exercise 1.2.1. Write a Python function that approximates $\sin(x)$ using the Taylor expansion $\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$. Test your function for simple known values of $\sin(x)$, such as $\sin(0) = 0$, $\sin(\pi/4) = \frac{1}{\sqrt{2}}$, $\sin(\pi/2) = 1$, and $\sin(\pi) = 0$, etc. How accurate is the approximation?

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc. △

Exercise 1.2.2. Write a Python function that computes the square root of a positive number using the Babylonian method. The Babylonian method to compute $\sqrt{S}$ for $S > 0$ constructs the sequence x_n by setting $x_0 = S$ and iterating

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right).$$

Your code can take as input a tolerance parameter $\varepsilon > 0$, and should iterate until $|x_n^2 - S| \leq \varepsilon$, and then return x_n. Test your square root function to make sure it works.

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc. △

Exercise 1.2.3. Write a Python program that uses the Sieve of Eratosthenes to find all prime numbers between 2 and a given integer n.

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc. △

Exercise 1.2.4. Write a Python function that computes the largest magnitude eigenvalue of a square matrix with the power iteration. The power iteration is

$$x_{n+1} = \frac{Ax_n}{\|Ax_n\|}.$$

For a diagonalizable matrix, the iteration converges to the eigenvector of A with largest magnitude eigenvalue. The eigenvalue is

$$\lambda = \lim_{n\to\infty} \|Ax_n\|.$$

Compare your function to the true eigenvector and eigenvalue for small matrices where you can compute it by hand, to check that your function works.

In this exercise you may use Numpy. Try to write your code with only one loop, over the iterations in the power method. △

Exercise 1.2.5. Write a Python function that numerically approximates π via the integral expression

$$\pi = 4\int_0^1 \sqrt{1-x^2}\,dx.$$

Your function should not use any loops. Use a Numpy array and Numpy functions instead. How many decimal places of π can you accurately compute?

In this exercise you may use Numpy. Can you write the code without any loops? △

The following exercise may be of interest, but does not involve Python.

Exercise 1.2.6. Prove that the iteration in the Babylonian method above converges quadratically to the square root of x. In particular, show that the error $\varepsilon_n = \frac{x_n}{\sqrt{x}} - 1$ satisfies

$$\varepsilon_{n+1} = \frac{\varepsilon_n^2}{2(\varepsilon_n+1)}.$$

From this, we get that $\varepsilon_n \geq 0$ for $n \geq 1$, and so

$$\varepsilon_{n+1} \leq \frac{1}{2}\min\{\varepsilon_n^2, \varepsilon_n\}.$$

Why does the inequality above guarantee convergence (i.e., that $\varepsilon_n \to 0$ as $n \to \infty$)? △

1.3 Background on audio, images, and data analysis

A major theme in this course is that of finding a good basis to represent your data. Usually data (e.g., images, audio, video, etc.) are captured in

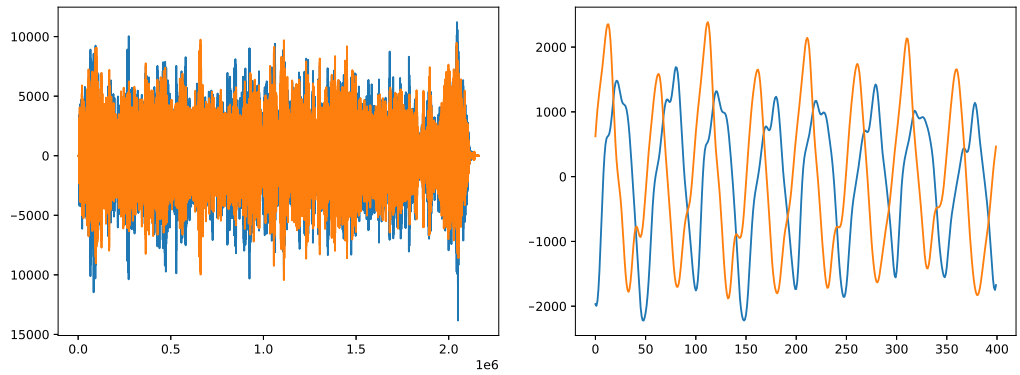


Figure 1.3.1: Example of a stereo audio signal from a piece of classical music. The left figure shows both channels over the whole song, while the right figure shows a short clip.

formats that are convenient for acquiring the data, but very inconvenient for storage or other kinds of processing. Much of this course is concerned with finding new bases for image and audio data that are better for tasks like image compression or image classification, among others. A change of basis can be linear, which the reader should be familiar with from linear algebra, or nonlinear as in modern deep neural networks. The change of basis can also be either hand-crafted (like in Fourier or Wavelet analysis), or learned from the data, as in Principal Component Analysis (PCA) or deep learning. Before we proceed with the course, we review the formats in which images and audio are typically recorded and stored.

1.3.1 Audio

A digital audio signal is a series of discrete samples of a continuous audio signal generated by the movement of the vibrating membrane of a microphone (of course, the vibrations in the membrane are caused by the rarefaction and compression of air, e.g., sound waves). Audio signals can be captured at different sampling rates and bit-depths. Standard CD quality audio has two channels (left and right) sampled at 44,100 Hz with 16-bits per sample. This means each channel has 44,100 samples per second, each encoded as a (signed) 16-bit integer to represent the magnitude of the sound wave at that sample. By the Sampling Theorem, this sampling rate allows CD audio to represent frequencies up to 22,050 Hz, which is well above what most people can hear. Figure 1.3.1 shows stereo audio signals for a piece of classical music.

The bit-rate for CD quality audio is the number of bits used per second of

audio, which we can compute as

$$\underbrace{2}_{\text{Channels}} \times \underbrace{44,100}_{\text{Samples per Second}} \times \underbrace{16}_{\text{Bits per Sample}} = 1,411,200 \text{ bits/second.}$$

Often this is written in terms of kilobits (kbit)—a kilobit is 1000 bits—so CD audio has a bitrate of 1,411 kbit/s (also denoted 1,411 kbps). A 4 minute long song would thus take

$$\underbrace{4}_{\text{Minutes}} \times \underbrace{60}_{\text{Seconds per Minute}} \times 1,411 \text{ kbits/second} = 338,640 \text{ kbits,}$$

of space on disk. In terms of megabits (Mbit)—a megabit is 1000 kilobits—this is 339 Mbits. The reader may be more familiar with kilobytes (kB) and megabytes (MB). A byte is 8 bits, and so 339 Mbits is actually around 42 MB.[1]

The reader who is familiar with digital audio files will probably know that an mp3 audio file for a 4 minute song is usually around 5 MB, depending on the bitrate, which is much less than the 42 MB we computed for the raw data. In fact, a very high quality mp3 bitrate is 320 kbit/second[2], which is less than one quarter of the bitrate of CD quality audio. Many mp3 audio files are compressed at much lower bitrates than this, with sometimes acceptable results. A main question we will address in this course is how to compress audio signals (and later images and video), without destroying important information. Other important tasks in audio processing are classifying audio signals (e.g., determining which song is playing automatically), and speech to text (e.g., determining what was said in an audio sample).

1.3.2 Images

A digital image is a discrete sampling of a two dimensional continuous signal given by the light hitting a rectangular image sensor in a digital camera. Thus, we can think of an image as a two-dimensional array of pixels. An image can be grayscale, in which case each pixel has a single number associated with it, representing the brightness, or image intensity, at that location. Figure 1.3.2 shows an example of a grayscale image, the famous cameraman image. On the right in the figure, we show the image plotted as a function (e.g., a surface),

[1] Note we are using base 10 to define kilo and mega. It is also common to use base 2, so that a kilobit is $2^10 = 1024$ bits, and a megabit is 1024 kilobits.

[2] It is widely accepted, except by some audio enthusiasts, that 320 kbit/second mp3 audio is indistinguishable to the human ear from CD quality audio.

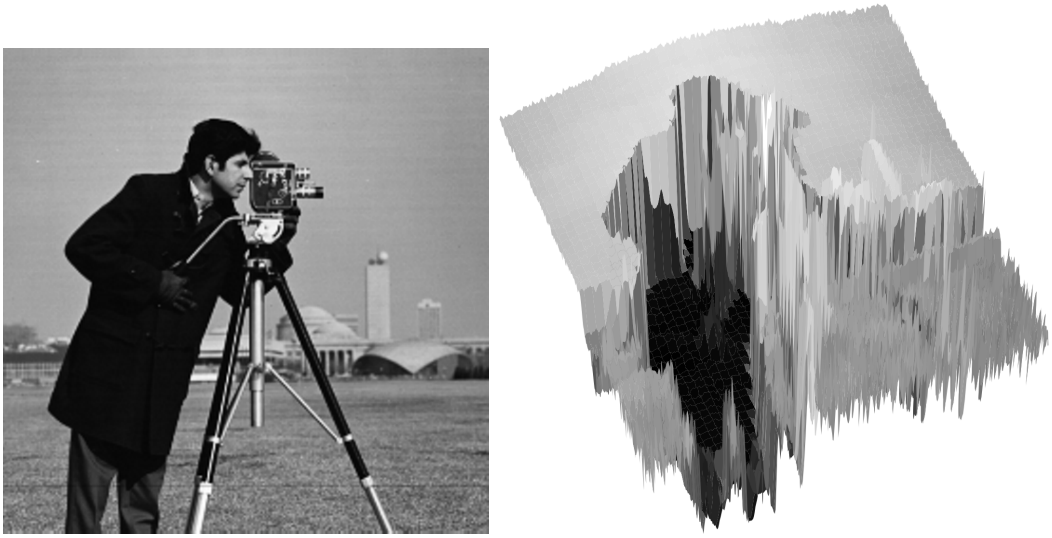


Figure 1.3.2: Example of a grayscale digital image. The left shows the image in its usual form, while the right side depicts the image intensity as a surface (i.e., a function of 2 variables).

over the two-dimensional plane, where the height of the surface is the intensity of the pixel at that location. In a color image, each pixel is associated with several numbers, representing the intensity of different colors at that point, which mix to produce the correct color. The most common colorspace is RGB (red–green–blue), in which each pixel has three numbers representing the amount of red, green, and blue present at each pixel in the image. Thus, a grayscale image can be thought of as a matrix, while a color image can be thought of as 3 (or possibly more) matrices, one for each color channel. There are other colorspaces that are frequently used (e.g., by printers) that have more color channels, or use different colors. For example, in hyperspectral satellite imagery, it is not uncommon to have over 200 color channels in an image, each corresponding to a different frequency band of the electromagnetic spectrum.

The cameraman image is of size 512×512, and so it has $262,144$ pixels. This is quite small by today's standards. For example, a high-end modern smartphone has a camera with 12 million pixels (12 MP). For a color image, this requires storing 36 million numbers. Images usually have a bit depth of 8-bit, allowing $2^8 = 256$ different values for each color channel, or about 16 million different colors. Some newer smartphones use higher bit depths of 10-bits or 12-bits, yielding over 1 billion different colors. Let's consider an 8-bit

color image with 12 megapixels. The 36 million 8-bit numbers that are stored take up 36 MB of space. A 10-bit image would take 45 MB of space and a 12-bit image would take 54 MB of space. High end digital cameras can have even higher resolution, currently up to around 46 million pixels, yielding 8-bit images that take around 130 MB of space, and 14-bit images that take up to 240 MB.

However, images are rarely stored or transmitted in raw form, since it is a colossal waste of space. Common image compression algorithms, like jpeg, can compress images to roughly one tenth of their size, without noticeable loss in image quality. Building on our discussion of audio compression, we will learn in this course how to compress images without removing important information. Aside from image compression, there are many other important tasks in computer vision and image processing, such as image segmentation (determining the object of interest in an image), object recognition (determining what the object is), image restoration (removing noise or blur), image inpainting (recovering lost portions of images), and image classification. We will touch on some of these other applications in the course.

A few words about video compression. We will not touch on this in the course, but since video is simply a sequence of images, one could apply image compression in this setting as well. However, there is a lot of temporal information that is missed by such an approach, and modern video compression uses motion tracking and compresses only the differences between subsequent frames of video, which has significant advantages over image compression alone.

1.3.3 Data science and machine learning

Finally, this course is not only about image and audio processing. Part of the course will cover some fundamentals of data science and machine learning from a mathematical perspective. In data analysis, one typically has many datapoints, and the goal is to uncover structure in the data to perform tasks like classification, clustering, dimension reduction, etc. Many problems in machine learning are related to image analysis (e.g., image classification), but are substantially different in character than the image and audio compression problems described in the preceding sections. In machine learning, we are given many (possibly thousands or millions) of images and the task is to understand how the images naturally group together (e.g., clustering), and how to automatically distinguish images from different classes (e.g., image classification).

Figure 1.3.3 shows an example of some images from the MNIST dataset,

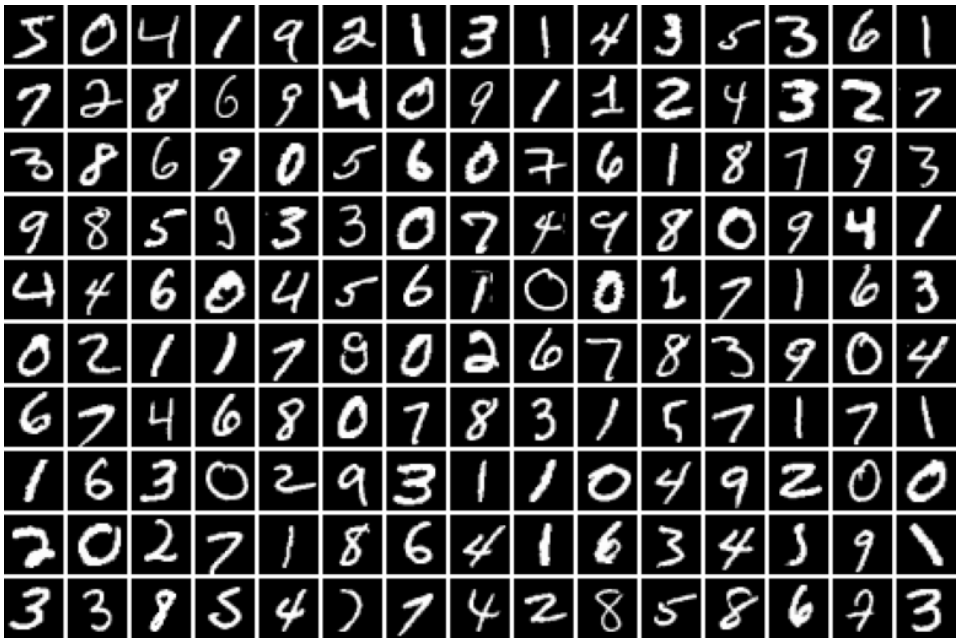


Figure 1.3.3: Example of some of the MNIST digits. Each image is a 28×28 pixel image of a handwritten digit.

which we will use for illustrating machine learning throughout the course. The MNIST dataset contains 70,000 grayscale images of handwritten digits 0 through 9. Each image is very small, containing only 28×28 pixels. We can still consider the image compression problem, but here the goal would be to compress the entire dataset, and not a single image. A more common task is to train a machine learning classifier to recognize the number written in each image. This is called optical character recognition, and is very commonly used for many tasks, including archiving old newspapers or books, and teaching self-driving cars to read street signs and house numbers. We will cover various methods for image classification in the course, starting with a method based on principal component analysis (PCA), and then moving on to more sophisticated methods based on Fourier and Wavelet features, and deep convolutional neural networks.

Chapter 2

Linear Algebra Review

We recall some basic facts about linear algebra in this section.

2.1 Notation

We will use capital letters like $A, B, C, \ldots$ for matrices, and lower case letters, such as $x, y, z, \ldots$ for vectors. Scalars will be denoted by lower case letters a, b, c, or $a_1, a_2, \ldots$. When not specified, vectors are of length n and matrices of size $m \times n$. Vectors are always treated as column vectors. If we have more than a handful of vectors, we will use subscripts to denote a collection of vectors, so $x_1, x_2, \ldots, x_p$ will always refer to a collection of p vectors. For example, $e_1, e_2, \ldots, e_n$ will denote the n standard basis vectors in $\mathbb{R}^n$. The reader should be careful not to confuse x_i with the coordinates of the vector x. We will rarely need to notate the coordinates of vectors or matrices, and if we do, we will use the notation $x(i)$ for the i^{th} coordinate of x, or $e_i^T x$. Similarly, we will denote the entries of a matrix A as $A(i, j)$.

The *dot product* of vectors x and y is given by the product $x^T y$, treating x and y as $n \times 1$ column vectors. To write this out in coordinate notation we have

$$x^T y = \sum_{i=1}^{n} x(i)y(i).$$

The dot product induces a norm

$$\|x\| = \sqrt{x^T x} = \sqrt{x(1)^2 + x(2)^2 + \cdots + x(n)^2}$$

on the Euclidean space $\mathbb{R}^n$. The quantity $\|x - y\|$ is exactly the Euclidean distance between x and y. A commonly used algebraic expansion is

$$\|x \pm y\|^2 = \|x\|^2 \pm 2x^T y + \|y\|^2. \tag{2.1.1}$$

This can be verified immediately by writing $\|x \pm y\|^2 = (x \pm y)^T(x \pm y)$ and expanding. A very important identity is the Cauchy-Schwarz inequality

$$x^T y \leq \|x\|\|y\|. \tag{2.1.2}$$

To prove the Cauchy-Schwarz inequality, we first assume $\|x\| = \|y\| = 1$, that is, x and y are normal vectors (see below). Then we have to show that $x^T y \leq 1$. To see this we compute via (2.1.1) that

$$0 \leq \|x - (x^T y)y\|^2 1 - 2(x^T y)^2 + (x^T y)^2 = 1 - (x^T y)^2,$$

from which $x^T y \leq 1$ follows. For general $x, y \in \mathbb{R}^n$ we use the result for normal vectors to show that

$$\left(\frac{x}{\|x\|}\right)^T \frac{y}{\|y\|} \leq 1,$$

from which (2.1.2) immediately follows by rearranging (note that if either $\|x\| = 0$ or $\|y\| = 0$ the result is trivial, so we are safe dividing by these norms). Finally, using (2.1.1) and (2.1.2) we have

$$\|x + y\|^2 = \|x\|^2 + 2x^T y + \|y\|^2 \leq \|x\|^2 + 2\|x\|\|y\| + \|y\|^2 = (\|x\| + \|y\|)^2,$$

which reduces, upon taking square roots of both sides, to the *triangle inequality*

$$\|x + y\| \leq \|x\| + \|y\|. \tag{2.1.3}$$

By induction we can extend the triangle inequality to more than two vectors, that is

$$\left\| \sum_{i=1}^{m} x_i \right\| \leq \sum_{i=1}^{m} \|x_i\|. \tag{2.1.4}$$

The triangle and Cauchy-Schwarz inequalities are two of the most important inequalities in analysis.

We say two vectors x and y are *orthogonal* if $x^T y = 0$. A vector x is a *normal* vector if $\|x\| = 1$, and two orthogonal vectors x and y that are also normal vectors are called *orthonormal*. A sequence of vectors $v_1, v_2, \ldots v_p$ is called orthogonal (resp. orthonormal) if each pair v_i, v_j is orthogonal (resp. orthonormal), provided $i \neq j$. Consider a vectors x in the span of orthonormal vectors $v_1, v_2, \ldots, v_p$, that is

$$x = \sum_{i=1}^{p} a_i v_i.$$

Then the norm of x can be computed by

$$\|x\|^2 = \sum_{i=1}^{p} a_i v_i^T \sum_{j=1}^{p} a_j v_j = \sum_{i=1}^{p}\sum_{j=1}^{p} a_i a_j v_i v_j^T = \sum_{i=1}^{p} a_i^2. \tag{2.1.5}$$

Exercise 2.1.1. Given vectors x and y, both of length n, we will sometimes find it useful to construct the rank-one matrix xy^T. The matrix is called "rank-one" since the range of xy^T is spanned by the vector x, and is thus one-dimensional so the matrix has rank equal to one. Using the definition of matrix multiplication, xy^T is the $n \times n$ matrix whose (i, j) entry is $x(i)y(j)$. Let $x_1, x_2, x_3, \ldots, x_m$ be a collection of vectors of length n. Show that

$$\sum_{i=1}^{m} x_i x_i^T = X^T X,$$

where X is the $m \times n$ matrix whose i^{th} row is x_i^T, which can be written as $X = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^T$. △

2.2 Projection

Let $L \subset \mathbb{R}^n$ be the linear subspace spanned by the collection of orthonormal vectors $v_1, v_2, \ldots, v_p$, where $p \leq n$. That is

$$L = \left\{ \sum_{i=1}^{p} a_i v_i \ : \ a_i \in \mathbb{R} \right\}.$$

In this case, L is p-dimensional. The *projection* of a point $x \in \mathbb{R}^n$ onto L, denoted $\text{Proj}_L x$, is the closest point in the subspace L to x. That is, $\text{Proj}_L x \in L$ satisfies

$$\|\text{Proj}_L x - x\| \leq \|y - x\| \ \text{ for all } y \in L.$$

We recall here some basic properties of the projection. First, we claim that

$$\text{Proj}_L x = \sum_{i=1}^{p} (x^T v_i) v_i \tag{2.2.1}$$

To see this, let us write $y \in L$ as

$$y = \sum_{i=1}^{p} a_i v_i,$$

and compute

$$\|x-y\|^2 = \|x\|^2 - 2x^T y + \|y\|^2 = \|x\|^2 - 2\sum_{i=1}^{p} a_i x^T v_i + \sum_{i=1}^{p} a_i^2.$$

Minimizing over a_i yields $a_i = x^T v_i$, which establishes the claim.

Since the v_i are orthonormal, we have by (2.1.5) that

$$\|\mathrm{Proj}_L x\|^2 = \sum_{j=1}^{p} (x^T v_i)^2 \tag{2.2.2}$$

It can be useful to write (2.2.1) in matrix form. Let V be the $n \times p$ matrix whose columns are $v_1, v_2, \ldots, v_p$, that is

$$V = \begin{bmatrix} v_1 & v_2 & \cdots & v_p \end{bmatrix}.$$

Then we have

$$\mathrm{Proj}_L x = VV^T x. \tag{2.2.3}$$

Note that since the columns of V are orthonormal, we have $V^T V = I$, and so

$$(VV^T)^2 = VV^T VV^T = VV^T. \tag{2.2.4}$$

This is a natural property, and simply says that if we apply the projection twice, the second operation leaves the point unchanged, since it already lies in L. It is also clear that VV^T is a symmetric matrix.

The *residual* is the difference $x - \mathrm{Proj}_L x$. The residual is orthogonal to L, and we thus say the projection is *orthogonal projection*. Indeed, we compute

$$(x - \mathrm{Proj}_L x)^T V = (x - VV^T x)^T V = x^T V - x^T VV^T V = 0.$$

Note we used that $V^T V = I$ in the third equality above. Subsequently, we have

$$\|x\|^2 = \|x - \mathrm{Proj}_L x + \mathrm{Proj}_L x\|^2 = \|x - \mathrm{Proj}_L x\|^2 + \|\mathrm{Proj}_L x\|^2,$$

and so

$$\|x - \mathrm{Proj}_L x\|^2 = \|x\|^2 - \|\mathrm{Proj}_L x\|^2. \tag{2.2.5}$$

The matrix corresponding to the residual mapping is $I - VV^T$, and, as with the projection, the residual matrix satisfies

$$(I - VV^T)^2 = I - 2VV^T + VV^T VV^T = I + VV^T. \tag{2.2.6}$$

This is the analogous property to (2.2.4). The matrix $I - VV^T$ is also clearly a symmetric matrix.

It is sometimes useful to project onto *affine spaces*. An affine space has the form $x_0 + L$[1], where L is a linear subspace. The key difference between affine and linear subspaces is that linear subspaces must contain the origin 0, while an affine space translates the origin to a new point x_0. To project onto an affine subspace, we simply translate the affine space and the point x to the origin, project onto L, and translate back. That is, the projection onto the affine space $A = x_0 + L$, denoted Proj_A, is given by

$$\text{Proj}_A x = x_0 + \text{Proj}_L(x - x_0). \tag{2.2.7}$$

We can also write this as

$$\text{Proj}_A x = x_0 - \text{Proj}_L x_0 + \text{Proj}_L x.$$

Exercise 2.2.1. Let L be a linear subspace of $\mathbb{R}^n$.

(i) Show that $\|\text{Proj}_L x\| \leq \|x\|$.

(ii) Show that $\text{Proj}_L x = x$ if and only if $x \in L$.

(iii) Show that if $\text{Proj}_L x = x$ for all $x \in \mathbb{R}^n$, then $L = \mathbb{R}^n$.

△

Exercise 2.2.2. Let V and W be orthogonal linear subspaces of $\mathbb{R}^n$. This means that for each $v \in V$ and $w \in W$ we have $w^T v = 0$. Define

$$V + W = \{v + w \,:\, v \in V \text{ and } w \in W\}.$$

Show that

$$\text{Proj}_{V+W} x = \text{Proj}_V x + \text{Proj}_W x.$$

△

2.3 Diagonalization of symmetric matrices

Every symmetric matrix can be diagonalized. That is, for any symmetric matrix A, there exists an orthogonal matrix Q and a diagonal matrix D such that

$$A = QDQ^T.$$

[1]The notation means $x_0 + L = \{x_0 + y \,:\, y \in L\}$.

An orthogonal matrix is a square matrix whose columns are orthonormal vectors. In this case, the columns of Q are exactly the eigenvectors of the matrix A, which are orthogonal due to the symmetry of A. An orthogonal matrix also has the property that all rows are orthonormal and thus

$$Q^T Q = I = QQ^T.$$

Thus, the inverse of Q is Q^T, and vice versa.

We generally arrange the eigenvalues of A from smallest to largest

$$\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n.$$

The diagonal matrix D has exactly the eigenvalues λ_i on the diagonal, so $D(i,i) = \lambda_i$. Thus the decomposition $A = QDQ^T$ acting on a vector x can be interpreted as changing basis into the coordinates of the eigenvectors by taking $Q^T x$, then multiplying by the diagonal matrix D, and then converting back to the standard coordinates.

Exercise 2.3.1. Let Q be an orthogonal matrix. Show that $\|Qx\| = \|x\|$. △

Exercise 2.3.2. Let A be a symmetric matrix, and consider the optimization problem

$$\min\{x^T Ax \,:\, \|x\| = 1\}. \tag{2.3.1}$$

Since the set $\{x \in \mathbb{R}^n \,:\, \|x\| = 1\}$ is compact (closed and bounded), and the function $x \mapsto \|Ax\|$ is continuous, the optimization problem (2.3.1) admits a minimizer. Show that every minimizer x^* is an eigenvector of A with smallest eigenvalue. What happens if we switch the min to a max in (2.3.1)? [Hint: Diagonalize A as $A = QDQ^T$ and write write the optimization problem in terms of $y = Q^T x$. Compute the optimal y and convert back to the x coordinates.] △

Exercise 2.3.3. We say a square matrix A is *positive semi-definite* if $x^T Ax \geq 0$ for all vectors x. Show that a symmetric matrix is positive semi-definite if and only if all its eigenvalues are nonnegative. [Hint: Diagonalize A.] △

2.4 Vector calculus

It will sometimes be useful to differentiate real-valued functions of a vector x, and we record some basic identities here. We recall that for a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient ∇f is defined by

$$\nabla f(x) = \left(\frac{\partial f}{\partial x(1)}, \frac{\partial f}{\partial x(2)}, \ldots, \frac{\partial f}{\partial x(n)} \right).$$

For example, for the function $f(x) = x(1)^2 - x(2)^2$ on $\mathbb{R}^2$, the gradient is $\nabla f(x) = (2x(1), -2x(2))$. The gradient is important since it characterizes minima and maxima of f, via the necessary condition $\nabla f = 0$.

For a linear function $f(x) = y^T x$, we clearly have $\nabla f(x) = y$. For a quadratic function $f(x) = x^T A x$, where A is an $n \times n$ matrix, we have

$$\nabla f(x) = (A + A^T)x. \tag{2.4.1}$$

Indeed, we compute

$$\begin{aligned}
\frac{\partial f}{\partial x(k)} &= \frac{\partial}{\partial x(k)} \sum_{i=1}^{n} \sum_{j=1}^{n} A(i,j)x(i)x(j) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} A(i,j)(\delta(i,k)x(j) + \delta(j,k)x(i)) \\
&= \sum_{j=1}^{n} A(k,j)x(j) + \sum_{i=1}^{n} A(i,k)x(i) \\
&= \sum_{i=1}^{n} (A(k,i) + A(i,k))x(i),
\end{aligned}$$

which establishes the claim. Above, the notation $\delta(i,j)$ is the *Kronecker delta*, which satisfies $\delta(i,j) = 1$ when $i = j$ and $\delta(i,j) = 0$ when $i \neq j$.

The Hessian of f, denoted $\nabla^2 f(x)$, is the $n \times n$ matrix of mixed second derivatives

$$\nabla^2 f = \left(\frac{\partial^2 f}{\partial x(i) \partial x(j)} \right)_{i,j=1}^{n}.$$

In other words

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x(1) \partial x(1)} & \cdots & \frac{\partial^2 f}{\partial x(1) \partial x(n)} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x(n) \partial x(1)} & \cdots & \frac{\partial^2 f}{\partial x(n) \partial x(n)} \end{bmatrix}.$$

Due to the equality of mixed partial derivatives

$$\frac{\partial^2 f}{\partial x(i) \partial x(j)} = \frac{\partial^2 f}{\partial x(j) \partial x(i)},$$

the Hessian matrix $\nabla^2 f$ is symmetric. As an example, for the quadratic function $f(x) = x^T A x$, the Hessian is the constant matrix $\nabla^2 f(x) = A + A^T$. Indeed, (2.4.1) can be written as

$$\frac{\partial f}{\partial x(i)} = \sum_{k=1}^{n} (a_{ik} + a_{ki}) x_k.$$

Differentiating in $x(j)$ above we obtain

$$\frac{\partial^2 f}{\partial x(i)\partial x(j)} = (a_{ij} + a_{ji}),$$

which shows that $\nabla^2 f = A + A^T$.

Exercise 2.4.1. Assume A is a symmetric matrix. Show that

$$\nabla \|Ax\|^2 = 2A^2 x, \tag{2.4.2}$$

and

$$\nabla^2 \|Ax\|^2 = 2A^2. \tag{2.4.3}$$

△

Exercise 2.4.2. Let A be a symmetric matrix. Show that any minimizer x of (2.3.1) is an eigenvector of A with eigenvalue $\lambda = x^T Ax$, without using that A is diagonalizable. [Hint: Any minimizer of (2.3.1) is also a minimizer of the Rayleigh quotient

$$f(x) = \frac{x^T Ax}{x^T x}.$$

Compute $\nabla f(x)$, set $\nabla f(x) = 0$, and use that $\|x\| = 1$.] △

2.5 Taylor expansion

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, it is often useful to approximate f by simpler functions. When those simpler functions are polynomials, and the approximation is local in space, we obtain the well-known Taylor expansion. In this section we will work out first and second order Taylor expansions for functions on $\mathbb{R}^n$.

Let $x, y \in \mathbb{R}^n$ and define the function $g : \mathbb{R} \to \mathbb{R}$ by

$$g(t) = f(x + t(y - x)).$$

We first work with a one dimensinal Taylor expansion of g. By the Fundamental Theorem of Calculus we compute

$$\begin{aligned} g(1) &= g(0) + \int_0^1 g'(t)\, dt \\ &= g(0) + \int_0^1 g'(0)\, dt + \int_0^1 g'(t) - g'(0)\, dt. \end{aligned}$$

Therefore

$$g(1) = g(0) + g'(0) + R, \tag{2.5.1}$$

where

$$R = \int_0^1 g'(t) - g'(0)\, dt.$$

We now compute, via the chain rule, that

$$\begin{aligned} g'(t) &= \frac{d}{dt} f(x + t(y - x)) \\ &= \sum_{i=1}^{n} \frac{\partial f}{\partial x(i)}(x + t(y - x))(y(i) - x(i)) \\ &= \nabla f(x + t(y - x))^T (y - x). \end{aligned}$$

Therefore, applying Cauchy-Schwarz (2.1.2) we have

$$\begin{aligned} |g'(t) - g'(0)| &= |\nabla f(x + t(y - x))^T (y - x) - \nabla f(x)^T (y - x)| \\ &\leq \|\nabla f(x + t(y - x)) - \nabla f(x)\| \|x - y\|. \end{aligned}$$

This suggests the following definition.

Definition 2.5.1. We say that ∇f is L-Lipschitz if

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \tag{2.5.2}$$

for all $x, y \in \mathbb{R}^n$.

Assuming ∇f is L-Lipschitz we find that

$$\begin{aligned} |R| &\leq \int_0^1 |g'(t) - g'(0)|\, dt \\ &\leq \int_0^1 \|\nabla f(x + t(y - x)) - \nabla f(x)\| \|x - y\|\, dt \\ &\leq \int_0^1 L\|x + t(y - x) - x\| \|x - y\| \\ &= L\|x - y\|^2 \int_0^1 t\, dt \\ &\leq \frac{L}{2}\|x - y\|^2. \end{aligned}$$

We now substitute this bound and the identites $g(0) = f(x)$, $g(1) = f(y)$ and $g'(0) = \nabla f(x)^T(y - x)$ into (2.5.1) to obtain

$$f(y) = f(x) + \nabla f(x)^T(y - x) + R \tag{2.5.3}$$

where

$$|R| \leq \frac{L}{2}\|x - y\|^2. \tag{2.5.4}$$

To simplify notation, we introduce Big O notation.

Definition 2.5.2. Given $f, g : \mathbb{R}^n \to \mathbb{R}$, we write $f = \mathcal{O}(g)$ to mean that there exists a constant $C > 0$ such that $|f(x)| \leq Cg(x)$ for all $x \in \mathbb{R}^n$.

Using the Big O notation, we can summarize our result in the following Theorem.

Theorem 2.5.3 (First Order Taylor Expansion)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ and assume ∇f is L-Lipschitz. Then*

$$f(y) = f(x) + \nabla f(x)^T(y - x) + \mathcal{O}(L\|x - y\|^2). \tag{2.5.5}$$

Remark 2.5.4. In using the Big O notation, we are being a bit less precise, compared to our previous bound on the remainder (2.5.4), since the Big O notation hides a constant $C > 0$ that is not specified by the notation. In the case of Theorem 2.5.3, the constant is $C = \frac{1}{2}$. In some cases we will use the explicit value of this constant. In general, it is merely important that C does not depedend on the given function f.

The first order Taylor expansion approximates f by the linear function

$$T(y) = f(x) + \nabla f(x)^T(y - x),$$

with quadratic errors $\mathcal{O}(L\|x - y\|^2)$. We now consider a second order Taylor expansion, which approximates f by a quadratic function, and sheds more light on the quadratic error term $\mathcal{O}(L\|x - y\|^2)$.

As before, we define the function g by

$$g(t) = f(x + t(y - x)),$$

except this time we compute a second order Taylor expansion of g. This requires two applications of the Fundamental Theorem of Calculus, as follows

$$\begin{aligned} g(1) &= g(0) + \int_0^1 g'(t)\,dt \\ &= g(0) + \int_0^1 g'(0) + \int_0^t g''(s)\,ds\,dt \\ &= g(0) + \int_0^1 g'(0)\,dt + \int_0^1 \int_0^t g''(s)\,ds\,dt \\ &= g(0) + g'(0) + \int_0^1 \int_0^t g''(0)\,ds\,dt + \int_0^1 \int_0^t (g''(s) - g''(0))\,ds\,dt \\ &= g(0) + g'(0) + \frac{1}{2}g''(0) + R. \end{aligned}$$

where

$$R = \int_0^1 \int_0^t (g''(s) - g''(0))\,ds\,dt.$$

We now compute, via the chain rule, that

$$\begin{aligned} g''(t) &= \frac{d^2}{dt^2} f(x + t(y-x)) \\ &= \frac{d}{dt} \sum_{i=1}^n \frac{\partial f}{\partial x(i)}(x + t(y-x))(y(i) - x(i)) \\ &= \sum_{i=1}^n \sum_{j=1}^n \frac{\partial^2 f}{\partial x(i)\partial x(j)}(x + t(y-x))(y(i) - x(i))(y(j) - x(j)) \\ &= (y-x)^T \nabla^2 f(x + t(y-x))(y-x). \end{aligned} \tag{2.5.6}$$

Therefore, applying Cauchy-Schwarz we have

$$\begin{aligned} |g''(s) - g''(0)| &= \left|(y-x)^T \left(\nabla^2 f(x + s(y-x)) - \nabla^2 f(x)\right)(y-x)\right| \\ &\leq \left\|\left(\nabla^2 f(x + s(y-x)) - \nabla^2 f(x)\right)(y-x)\right\| \|x - y\|. \end{aligned}$$

To proceed further, we need to define the norm of a matrix. The definition below gives the operator norm of a matrix A, induced by the Euclidean norm on vectors $x \in \mathbb{R}^n$.

Definition 2.5.5. Given a matrix $A \in \mathbb{R}^{n\times m}$, the operator norm $\|A\|$ of A is given by

$$\|A\| := \max_{\substack{x\in\mathbb{R}^n \\ x\neq 0}} \frac{\|Ax\|}{\|x\|}.$$

We note that for any $x \in \mathbb{R}^n$ we have $\|Ax\| \leq \|A\|\|x\|$. Applying this above yields

$$|g''(s) - g''(0)| \leq \|\nabla^2 f(x + s(y - x)) - \nabla^2 f(x)\|\|x - y\|^2.$$

This motivatives the following definition.

Definition 2.5.6. We say that $\nabla^2 f$ is L-Lipschitz if

$$\|\nabla^2 f(x) - \nabla^2 f(y)\| \leq L\|x - y\|$$

for all $x, y \in \mathbb{R}^n$.

Using this definition yields

$$\begin{aligned} |g''(s) - g''(0)| &\leq \|\nabla^2 f(x + s(y - x)) - \nabla^2 f(x)\|\|x - y\|^2 \\ &\leq L\|x + s(y - x) - x\|\|x - y\|^2 \\ &= Ls\|x - y\|^3. \end{aligned}$$

It follows that

$$|R| \leq L\|x - y\|^3 \int_0^1 \int_0^t s \, ds \, dt = \frac{L}{6}\|x - y\|^3.$$

Combining all the observations above we obtain the following second order Taylor expansion result.

Theorem 2.5.7 (Second Order Taylor Expansion)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ and assume $\nabla^2 f$ is L-Lipschitz. Then*

$$f(y) = f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}(y - x)^T \nabla^2 f(x)(y - x) + \mathcal{O}(L\|x - y\|^3). \tag{2.5.7}$$

Remark 2.5.8. As with the first order Taylor expansion in Theorem 2.5.3, the Big O notation obscures the constant. In this case we can see from the argument above that the constant in the Big O notation is $C = \frac{1}{6}$.

Finally, we consider a first order Taylor expansion of the gradient ∇f. While this can be handled by Theorem 2.5.3 applied to the components of ∇f, we derive a shaper matrix-vector form below.

Let us define

$$g_i(t) = \frac{\partial f}{\partial x(i)}(x + t(y - x)).$$

Then, as in the derivation of Theorem 2.5.3 we have

$$g_i(1) = g_i(0) + g_i'(0) + R_i,$$

where

$$g_i'(t) = \nabla \frac{\partial f}{\partial x(i)}(x + t(y-x))^T (y-x),$$

and

$$R_i = \int_0^1 g_i'(t) - g_i'(0)\, dt.$$

Now, notice that

$$\begin{aligned} g_i'(t) &= \sum_{j=1}^{n} \frac{\partial^2 f}{\partial x(i)\partial x(j)}(x + t(y-x))(y(j) - x(j)) \\ &= [\nabla^2 f(x + t(y-x))(y-x)]_i. \end{aligned}$$

Applying these observations for $i = 1, \ldots, n$, and putting them into vector form, we obtain

$$\nabla f(y) = \nabla f(x) + \nabla^2 f(x)(y-x) + R,$$

where

$$R = \int_0^1 [\nabla^2 f(x + t(y-x)) - \nabla^2 f(x)](y-x)\, dt.$$

Note that the term inside the integral above is a vector; the integral is simply defined component-wise. Using that the norm of the integral is bounded by the integral of the norm, we have

$$\begin{aligned} \|R\| &= \left\| \int_0^1 [\nabla^2 f(x + t(y-x)) - \nabla^2 f(x)](y-x)\, dt \right\| \\ &\leq \int_0^1 \|[\nabla^2 f(x + t(y-x)) - \nabla^2 f(x)](y-x)\|\, dt \\ &\leq \int_0^1 \|\nabla^2 f(x + t(y-x)) - \nabla^2 f(x)\| \|x-y\|\, dt \\ &\leq \int_0^1 L\|x + t(y-x) - x\| \|x-y\|\, dt \\ &= L\|x-y\|^2 \int_0^1 t\, dt \\ &= \frac{L}{2}\|x-y\|^2, \end{aligned}$$

provided $\nabla^2 f$ is L-Lipschitz. This yields the following theorem.

Theorem 2.5.9 (Gradient Taylor Expansion)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ and assume $\nabla^2 f$ is L-Lipschitz. Then*

$$\nabla f(y) = \nabla f(x) + \nabla^2 f(x)(y - x) + \mathcal{O}(L\|x - y\|^2). \tag{2.5.8}$$

As before, we note that the constant in the Big O notation is $C = \frac{1}{2}$.

Exercise 2.5.10. Let A be a symmetric and positive semi-definite matrix, and let λ_{max} denote the largest eigenvalue of A. Use an argument similar to Exercise 2.3.2 to show that

$$\|A\| = \lambda_{max}. \tag{2.5.9}$$

For this reason, the operator norm is also called the *spectral norm*. How does (2.5.9) change if A is symmetric but not necessarily positive semi-definite? How about if A is not symmetric? △

2.6 Convex functions

Here, we review some basic theory of convex functions.

Definition 2.6.1. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *convex* if

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y) \tag{2.6.1}$$

for all $x, y \in \mathbb{R}^n$ and $\lambda \in (0, 1)$.

Definition 2.6.2 (Strongly convex)**.** Let $\mu \geq 0$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *μ-strongly convex* if $f - \frac{\mu}{2}\|x\|^2$ is convex.

Exercise 2.6.3. Show that f is μ-strongly convex for $\mu \geq 0$ if and only if

$$f(\lambda x + (1 - \lambda)y) + \frac{\mu}{2}\lambda(1 - \lambda)\|x - y\|^2 \leq \lambda f(x) + (1 - \lambda)f(y) \tag{2.6.2}$$

for all $x, y \in \mathbb{R}^n$ and $\lambda \in (0, 1)$. The statement in (2.6.2) is often given as the definition of strong convexity. △

We now give several different characterizations of strong convexity for functions on the real line $\mathbb{R}$.

Lemma 2.6.4. *Let $f : \mathbb{R} \to \mathbb{R}$ be twice continuously differentiable and $\mu \geq 0$. The following are equivalent.*

(i) f *is* μ*-strongly convex.*

(ii) $f''(x) \geq \mu$ *for all* $x \in \mathbb{R}$.

(iii) $f(y) \geq f(x) + f'(x)(y-x) + \frac{\mu}{2}(y-x)^2$ *for all* $x, y \in \mathbb{R}$.

(iv) $(f'(x) - f'(y))(x-y) \geq \mu(x-y)^2$ *for all* $x, y \in \mathbb{R}$.

Proof. It is enough to prove the result for $\mu = 0$. Then if any statement holds for $\mu > 0$, we can define $g(x) = f(x) - \frac{\mu}{2}x^2$ and use the results for g with $\mu = 0$.

The proof is split into three parts.

1. (i) $\implies$ (ii): Assume f is convex. Let $x_0 \in \mathbb{R}$ and set $\lambda = \frac{1}{2}$, $x = x_0 - h$, and $y = x_0 + h$ for a real number h. Then

$$\lambda x + (1-\lambda)y = \frac{1}{2}(x_0 - h) + \frac{1}{2}(x_0 + h) = x_0,$$

and the convexity condition (2.6.1) yields

$$f(x_0) \leq \frac{1}{2}f(x_0 - h) + \frac{1}{2}f(x_0 + h).$$

Therefore

$$f(x_0 - h) - 2f(x_0) + f(x_0 + h) \geq 0$$

for all h, and so

$$f''(x_0) = \lim_{h \to 0} \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} \geq 0.$$

2. (ii) $\implies$ (iii): Assume that $f''(x) \geq 0$ for all $x \in \mathbb{R}$. Let $x, y \in \mathbb{R}$ and assume, without loss of generality, that $y > x$. Then by the Fundamental Theorem of Calculus, applied twice, we have

$$\begin{aligned} f(y) &= f(x) + \int_x^y f'(t)\,dt \\ &= f(x) + \int_x^y f'(x) + \int_x^t f''(s)\,ds\,dt \\ &\geq f(x) + \int_x^y f'(x)\,dt \\ &= f(x) + f'(x)(y-x), \end{aligned}$$

which establishes the result.

3. (iii) $\implies$ (iv): Assume (iii) holds. Then we have

$$f(y) \geq f(x) + f'(x)(y - x)$$

and

$$f(x) \geq f(y) + f'(y)(x - y).$$

Inserting the second inequality into the first yields

$$f(y) \geq f(y) + f'(y)(x - y) + f'(x)(y - x).$$

Rearranging we obtain

$$(f'(x) - f'(y))(x - y) \geq 0,$$

which establishes the claim.

4. (iv) $\implies$ (i): Assume (iv) holds. Then f' is nondecreasing and so $f''(x) \geq 0$ for all $x \in \mathbb{R}$, hence (ii) holds, and so does (iii). Let $x, y \in \mathbb{R}$ and $\lambda \in (0, 1)$, and set $x_0 = \lambda x + (1 - \lambda)y$. Define

$$L(z) = f(x_0) + u'(x_0)(z - x_0).$$

By (iii) we have $f(z) \geq L(z)$ for all z. Therefore

$$f(\lambda x + (1 - \lambda)y) = f(x_0) = \lambda L(x) + (1 - \lambda)L(y) \leq \lambda f(x) + (1 - \lambda)f(y),$$

and so f is convex. □

We now proceed with a higher dimensional analog of Lemma 2.6.4.

Theorem 2.6.5. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice continuously differentiable. The following are equivalent.*

(i) f is μ-strongly convex.

(ii) The smallest eigenvalue of $\nabla^2 f$ is at least μ.

(iii) $f(y) \geq f(x) + \nabla f(x)^T(y - x) + \frac{\mu}{2}\|x - y\|^2$ for all $x, y \in \mathbb{R}^n$.

(iv) $(\nabla f(x) - \nabla f(y))^T(x - y) \geq \mu\|x - y\|^2$ for all $x, y \in \mathbb{R}^n$.

Proof. Again, we may prove the result just for $\mu = 0$. The proof follows mostly from Lemma 2.6.4, with some additional observations.

1. (i) $\implies$ (ii): Assume f is convex. Since convexity is defined along lines, we see that $g(t) = f(x + tv)$ is convex for all $x, v \in \mathbb{R}^n$, and by Lemma 2.6.4 $g''(t) \geq 0$ for all t. Fix $x \in \mathbb{R}^n$ and note that by (2.5.6) we have

$$0 \leq g''(0) = v^T \nabla^2 f(x) v, \tag{2.6.3}$$

for all $v \in \mathbb{R}^n$. Taking v to be an eigenvector of $\nabla^2 f$, with eigenvalue λ, we have

$$0 \leq v^T \nabla^2 f(x) v = v^T \lambda v = \lambda \|v\|^2,$$

and therefore $\lambda \geq 0$. Thus, the smallest eigenvalue of $\nabla^2 f(x)$ is nonnegative (at least zero).

2. (ii) $\implies$ (iii): Assume (ii) holds and let $g(t) = f(x + tv)$ for $x, v \in \mathbb{R}^n$. By (2.6.3) we have $g''(t) \geq 0$ for all t, and so by Lemma 2.6.4

$$g(t) \geq g(s) + g'(s)(t - s)$$

for all s, t. Let $y \in \mathbb{R}^n$ and set $v = y - x$, $t = 1$ and $s = 0$ to obtain

$$f(y) \geq f(x) + \nabla f(x)^T (y - x).$$

3. (iii) $\implies$ (iv): The proof is similar to Lemma 2.6.4.

4. (iv) $\implies$ (i): Assume (iv) holds, and define $g(t) = f(x + tv)$ for $x, v \in \mathbb{R}^n$. Then we have

$$(g'(t) - g'(s))(t - s) = (\nabla f(x + tv) - \nabla f(x + sv)) \cdot v(t - s) \geq 0$$

for all t, s. By Lemma 2.6.4 we have that g is convex for all $x, v \in \mathbb{R}^n$, from which it easily follows that f is convex. □

Chapter 3

Principal Component Analysis

One of the most important tasks in data analysis is that of finding simpler structures in data. One of the simplest mathematical structures is a linear subspace. In this lecture we will discuss how to find the best linear subspace approximating a collection of data points. This process is called principal component analysis (PCA). Projecting onto this best linear subspace effectively reduces the dimensionality of our data to the dimension of the subspace, yielding a useful dimension reduction algorithm. After developing the mathematical theory of PCA, we will explore applications to image compression and classification.

3.1 Fitting the best linear subspace

Let $x_1, x_2, \ldots, x_m$ be a collection of vectors in $\mathbb{R}^n$, which we treat as our data points. Figure 3.1.1 shows an example of a point cloud in $\mathbb{R}^2$ which is roughly one dimensional. Depending on the application, we may wish to approximate the data points by a linear or an affine subspace of $\mathbb{R}^n$. To find a line of best fit for the data in Figure 3.1.1, we would seek a one dimensional affine subspace that best approximates the data.

We proceed with the analysis for an affine space, and we will see that we can immediately reduce to the linear case. We seek an affine subspace $A = x_0 + L$, where L is a k-dimensional linear subspace of $\mathbb{R}^n$ and $x_0 \in \mathbb{R}^n$, that best approximates our data in the mean squared sense. That is, we seek to minimize the mean squared error

$$E(x_0, L) = \sum_{i=1}^{m} \|x_i - \mathrm{Proj}_A x_i\|^2 \tag{3.1.1}$$

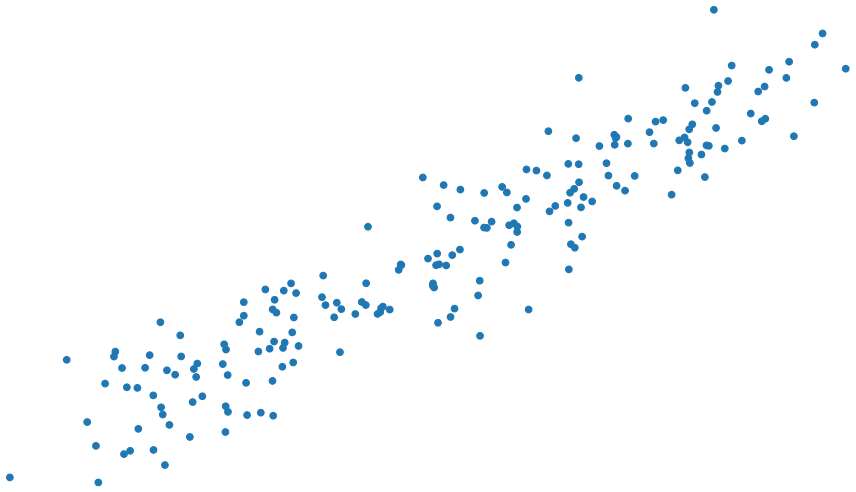


Figure 3.1.1: Example of a point cloud in $\mathbb{R}^2$. In this case, PCA would find the line of best fit through the point cloud.

over all k-dimensional linear subspaces L and translations $x_0 \in \mathbb{R}^n$. Let us write L as the span of orthonormal vectors $v_1, v_2, \ldots, v_k$, and let V be the matrix whose columns are the v_i.

We first consider the translation x_0. Recalling (2.2.7) we can write the energy E as

$$E(x_0, L) = \sum_{i=1}^{m} \|x_i - x_0 - \mathrm{Proj}_L(x_i - x_0)\|^2.$$

Then by (2.2.1) we have

$$E(x_0, L) = \sum_{i=1}^{m} \|(I - VV^T)(x_i - x_0)\|^2.$$

Differentiating in x_0 using (2.4.2) we find that the optimal value for x_0 must satisfy

$$0 = \nabla_{x_0} E(x_0, L) = -2 \sum_{i=1}^{m} (I - VV^T)^2 (x_i - x_0) = 0.$$

Let us define the mean, or centroid, of the data to be

$$\overline{x} = \frac{1}{m} \sum_{i=1}^{m} x_i, \tag{3.1.2}$$

and recall that $(I - VV^T)^2 = (I - VV^T)$. Bringing the summation above inside we obtain

$$0 = (I - VV^T)(x_0 - \overline{x}) = x_0 - \overline{x} - \mathrm{Proj}_L(x_0 - \overline{x}).$$

Therefore $x_0 - \overline{x} = \mathrm{Proj}_L(x_0 - \overline{x}) \in L$. It is clear that the energy E is unchanged by adding an element of L to x_0, so we will take $x_0 - \overline{x} = 0$, or $x_0 = \overline{x}$, for simplicity.

Exercise 3.1.1. Show that if $v \in L$ then $E(x_0 + v, L) = E(x_0, L)$. △

The discussion above allows us to reduce to the case of fitting a linear subspace to our data. Indeed, if we wish to fit an affine subspace, the optimal offset is the centroid $\overline{x}$, and we can simply center our data, by replacing x_i with $x_i - \overline{x}$, and reduce to the problem of fitting a linear subspace to the centered data. Thus, without loss of generality we consider the problem of minimizing the energy

$$E(L) = \sum_{i=1}^{m} \|x_i - \mathrm{Proj}_L x_i\|^2 \tag{3.1.3}$$

over all k-dimensional linear subspaces L of $\mathbb{R}^n$. Equivalently, we can view the problem as minimizing E over the orthonormal vectors $v_1, v_2, \ldots, v_k$ that span L. To proceed further, we rewrite the energy in a more convenient form.

Lemma 3.1.2. *The energy $E(L)$ can be expressed as*

$$E(L) = \mathrm{Trace}(M) - \sum_{j=1}^{k} v_j^T M v_j, \tag{3.1.4}$$

where M is the covariance matrix *of the data, given by*

$$M = \sum_{i=1}^{m} x_i x_i^T. \tag{3.1.5}$$

Remark 3.1.3. By Exercise 2.1.1, the covariance matrix M can also be written as $M = X^T X$, where X is the $m \times n$ matrix whose i^{th} row is x_i^T, which can be written as $X = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^T$. In practice, one is usually given the data matrix X directly, and the formula $M = X^T X$ is both more convenient and more efficient in many programming languages.

Proof of Lemma 3.1.2. We recall (2.2.2) and (2.2.5), and compute

$$\begin{aligned} E(L) &= \sum_{i=1}^{m} \left(\|x_i\|^2 - \|\mathrm{Proj}_L x_i\|^2 \right) \\ &= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{i=1}^{m} \sum_{j=1}^{k} (v_j^T x_i)^2 \\ &= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{j=1}^{k} \sum_{i=1}^{m} v_j^T x_i x_i^T v_j \\ &= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{j=1}^{k} v_j^T \left(\sum_{i=1}^{m} x_i x_i^T \right) v_j \\ &= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{j=1}^{k} v_j^T M v_j. \end{aligned}$$

To complete the proof, we note that

$$\mathrm{Trace}(M) = \sum_{i=1}^{m} \mathrm{Trace}(x_i x_i^T) = \sum_{i=1}^{m} \|x_i\|^2. \qquad \square$$

The first term in (3.1.4) is independent of L, so we may focus on the second term. Due to the minus sign, Lemma 3.1.2 shows that minimizing E is equivalent to *maximizing* the quantity

$$\sum_{j=1}^{k} v_j^T M v_j$$

over orthonormal vectors $v_1, v_2, \ldots, v_k$. We note that the covariance matrix M is a symmetric matrix, thus it is diagonalizable. This means there exists an orthogonal matrix P, whose columns are the orthonormal eigenvectors of M, and a diagonal matrix D, whose diagonal entries are the corresponding eigenvalues, such that $M = PDP^T$. We arrange the eigenvalues from largest to smallest, so $D_{ii} = \lambda_i$ and

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n. \tag{3.1.6}$$

Let $p_1, \ldots, p_n$ denote the corresponding orthonormal eigenvectors of M, which are just the columns of P.

We claim that M is positive semi-definite, that is, that $\lambda_i \geq 0$ for all i. To see this, simply note that

$$\lambda_j = p_j^T M p_j = \sum_{i=1}^{m} p_j^T x_i x_i^T p_j = \sum_{i=1}^{m} (p_j^T x_i)^2 \geq 0. \tag{3.1.7}$$

Since $p_j^T x_i$ is the projection of x_i onto p_j, we see that λ_j is simply the variance of the data in the direction p_j (up to a normalizing factor of $1/m$). We now compute

$$\begin{aligned}
\sum_{j=1}^{k} v_j^T M v_j &= \sum_{j=1}^{k} v_j^T P D P^T v_j \\
&= \sum_{j=1}^{k} \|D^{1/2} P^T v_j\|^2 \\
&= \sum_{j=1}^{k} \sum_{i=1}^{n} \lambda_i (p_i^T v_j)^2 \\
&= \sum_{i=1}^{n} \lambda_i \sum_{j=1}^{k} (p_i^T v_j)^2 \\
&= \sum_{i=1}^{n} \lambda_i \|\mathrm{Proj}_L p_i\|^2.
\end{aligned}$$

Notice that

$$\sum_{i=1}^{n} \|\mathrm{Proj}_L p_i\|^2 = \sum_{j=1}^{k} \sum_{i=1}^{n} (p_i^T v_j)^2 = \sum_{j=1}^{k} \|v_j\|^2 = \sum_{j=1}^{k} 1 = k,$$

where the identity $\|v_j\|^2 = \sum_{i=1}^{n} (p_i^T v_j)^2$ follows from the fact that the p_i form an orthonormal basis for $\mathbb{R}^n$. Thus, we must choose the v_i so as to distribute the weights $\|\mathrm{Proj}_L p_i\|^2$ among the largest eigenvalues as much as possible. A natural choice is to set $v_i = p_i$ for $i = 1, \ldots, k$. Then $p_i \in L$ for $i = 1, \ldots k$, and the p_i are orthogonal to L for $i \geq k+1$, yielding

$$\|\mathrm{Proj}_L p_i\|^2 = \begin{cases} 1, & \text{if } 1 \leq i \leq k \\ 0, & \text{otherwise.} \end{cases}$$

This choice of v_i then yields

$$\sum_{j=1}^{k} v_j^T M v_j = \sum_{i=1}^{k} \lambda_i. \tag{3.1.8}$$

The exercise below verifies that this choice of $v_i = p_i$ is indeed optimal.

Exercise 3.1.4. Suppose that λ_i satisfy (3.1.6) and let $a_1, \ldots, a_n$ satisfy $0 \leq a_i \leq 1$ and $\sum_{i=1}^n a_i = k$, where k is an integer $1 \leq k \leq n$. Show that

$$\sum_{i=1}^{n} \lambda_i a_i \leq \sum_{i=1}^{k} \lambda_i. \qquad \triangle$$

We summarize our findings in a theorem.

Theorem 3.1.5. *Let $p_1, p_2, \ldots, p_n$ be the orthonormal eigenvectors of the covariance matrix M, defined in (3.1.5), with corresponding eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, given in decreasing order. The energy $E(L)$, defined in (3.1.3), is minimized over k-dimensional linear subspaces $L \subset \mathbb{R}^n$ by setting*

$$L = span\{p_1, p_2, \ldots, p_k\}$$

and the optimal energy is given by

$$E(L) = \sum_{i=k+1}^{n} \lambda_i.$$

Proof. All that remains to prove is the formula for $E(L)$. This follows from Eq. (3.1.8), Lemma 3.1.2, and the identity

$$\text{Trace}(M) = \sum_{i=1}^{n} \lambda_i. \qquad \square$$

The vectors $p_1, p_2, \ldots$ in Theorem 3.1.5 are called the *principal components* of the data, and the eigenvalues $\lambda_1, \lambda_2, \ldots$ describe the amount of variation of the data in the direction of each principal component, due to (3.1.7). In Figure 3.1.2 we plot the principal components of the point cloud from Figure 3.1.1, scaled to length $2\sqrt{\lambda_i}$ to match the variation in each direction of the data.

Let $P_k = \begin{bmatrix} p_1 & p_2 & \cdots & p_k \end{bmatrix}$. Then, recalling Section 2.2, the projection of a vector x onto the PCA subspace L is given by

$$\text{Proj}_L x = P_k P_k^T x.$$

The coordinates of the point x in the subspace L are the contents of the length k vector $P_k^T x$. In this sense, we can view PCA as dimension reduction, since the length n vector x is approximated by the length k vector $P_k^T x$. Here, PCA gives a dimension reduction from $\mathbb{R}^n$ to $\mathbb{R}^k$.

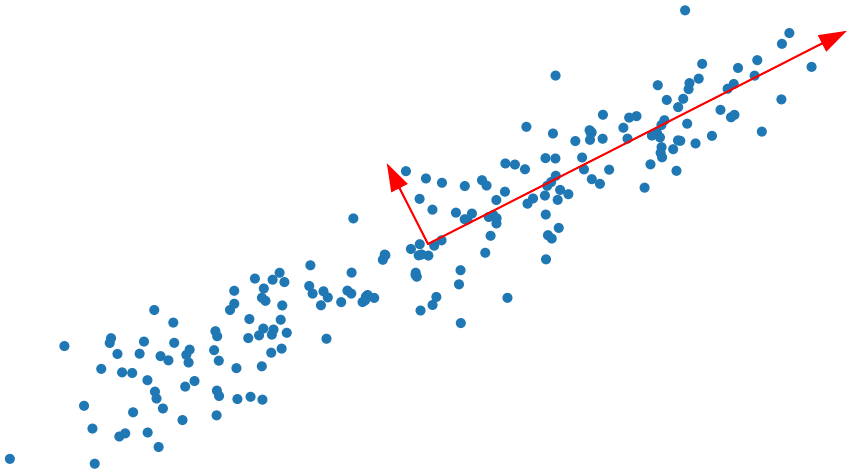


Figure 3.1.2: A depiction of the principal directions obtained by running PCA on the point cloud from Figure 3.1.1. The principal directions are scaled by $2\sqrt{\lambda_i}$ for visualization.

3.2 PCA dimension reduction

Python Notebook: .ipynb

We now give a brief application of PCA to dimension reduction. The steps for dimension reduction to $\mathbb{R}^k$ are outlined below. We assume we are given an $m \times n$ data matrix X

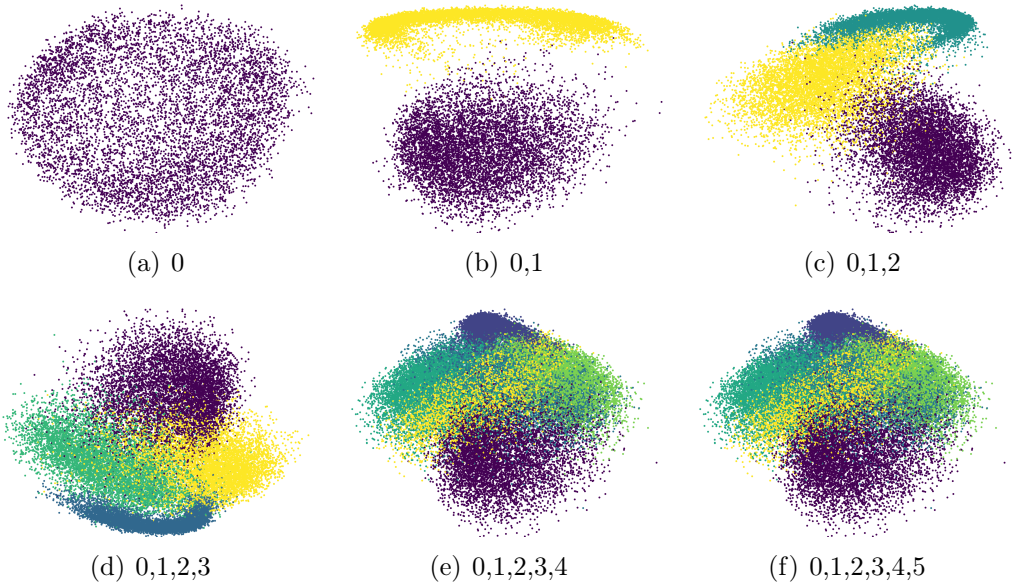
1. Compute the PCA covariance matrix $M = X^T X$, with the option of centering X first.

2. Compute the top k eigenvectors of M, and store them in a matrix P of size $n \times k$.

3. Compute the PCA dimension reduced dataset $B = XP$.

Note that the matrix B has dimensions $m \times k$, and the rows of B contain exactly the coordinates of each data point (row of X) in the PCA basis consisting of the top k eigenvectors of M. To lift the projection back to $\mathbb{R}^n$, the formula would be XPP^T. This would defeat the purpose of dimension reduction, but is useful in applications of PCA to compression, which is discussed in Section 3.5 below (essentially, multiplication by P^T is the decompression stage in PCA-based compression).

In Figure 3.2.1 we show the two dimensional PCA dimension reduction of some digits in the MNIST dataset. We start with just the zeros, and proceed



(a) 0 (b) 0,1 (c) 0,1,2

(d) 0,1,2,3 (e) 0,1,2,3,4 (f) 0,1,2,3,4,5

Figure 3.2.1: Plots of subsets of the MNIST dataset reduced to 2 dimensions through PCA, by projecting the data onto the top two principal components. The plots are colored by the underlying digit label. We start with just the zeros, and incrementally add digits up to 0–5. We note that PCA is able to well-separate the digits up to 0–3, and when we add digits 4 and 5, we see a significant amount of overlap between clusters.

by adding one digit at a time, up to digit 5. We note that the PCA embedding into $\mathbb{R}^2$ is able to well-separate the digits up to 4 digits (0–3). Beyond this, the digit clusters overlap significantly.

3.3 How many principal directions?

A basic question concerns how to chose the number of principal components k to use in PCA. A standard way to do this is to instead specify how much of the variation in the data one wishes to capture with the subspace L. Let $0 < \alpha \leq 1$ describe this quantity. For example, $\alpha = 0.95$ is interpreted as requiring that the subspace L capture 95% of the variation in the data. Since the eigenvalue λ_i describes the amount of variation in the principal direction p_i, we can simply choose k as small as possible while ensuring that

$$\sum_{i=1}^{k} \lambda_i \geq \alpha \sum_{i=1}^{n} \lambda_i.$$

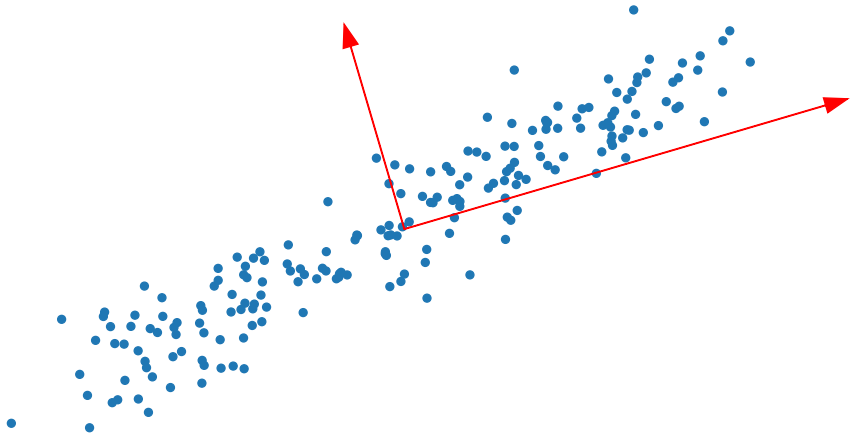


Figure 3.4.1: An illustration of how PCA is sensitive to outliers. Here, we used the same point cloud as in Figure 3.1.1, but added a single outlying point to the lower right of the point cloud (not depicted), whose distance from the point cloud is twice the length (longest dimension) of the point cloud. Since PCA minimize the sum of squared errors, it is overly concerned with approximating outliers, whose distance is far from the inlying point cloud.

As in the proof of Theorem 3.1.5, we have $\sum_{i=1}^{n} \lambda_i = \text{Trace}(M)$ and so we may rewrite the condition as

$$\sum_{i=1}^{k} \lambda_i \geq \alpha \, \text{Trace}(M). \tag{3.3.1}$$

The condition (3.3.1) can be checked without computing all of the eigenvalues of M, which may be computationally intensive in high dimensional applications, where n is very large. Instead, one can use an iterative eigenvalue solver, which finds the eigenvectors in order of decreasing eigenvalue, and stop the first time (3.3.1) holds.

3.4 Robust PCA

While PCA is simple to work with mathematically and computationally, it can be very sensitive to outliers in the data. This is due to its use of the mean squared error in (3.1.3), which strongly penalizes outliers. We show an example in Figure 3.4.1 of how a single outlier can lead to a large negative affect on the ability of PCA to accurately fit the main part of the point cloud.

Thus, it is important to remove outliers before applying PCA. Outlier detection can, however, be difficult, and so there has been substantial interest in more robust versions of PCA that are not sensitive to outliers. Many variants of robust PCA have been proposed in the literature. Some are based on minimizing a sum of distances energy of the form

$$F(L) = \sum_{i=1}^{m} \|x_i - \mathrm{Proj}_L x_i\|. \tag{3.4.1}$$

By omitting the square, we are placing a far lower penalty on severe outliers, and can achieve better performance. However, it is far more computationally challenging to minimize $F(L)$, compared to the mean-squared error $E(L)$, since there is no longer a simple relationship to the eigenvectors of the covariance matrix.

Exercise 3.4.1. Consider the weighted PCA energy

$$E_w(L) = \sum_{i=1}^{m} w_i \|x_i - \mathrm{Proj}_L x_i\|^2,$$

where $w_1, w_2, \ldots, w_m$ are nonnegative numbers (weights).

(i) Show that the weighted energy E_w is minimized over k-dimensional subspaces $L \subset \mathbb{R}^n$ by setting

$$L = \mathrm{span}\{p_1, p_2, \ldots, p_k\},$$

where $p_1, p_2, \ldots, p_n$ are the orthonormal eigenvectors of the covariance matrix

$$M_w = \sum_{i=1}^{m} w_i x_i x_i^T,$$

with corresponding eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, given in decreasing order.

(ii) Show that the weighted covariance matrix can also be expressed as

$$M_w = X^T W X,$$

where W is the $m \times m$ diagonal matrix with diagonal entries given by $w_1, w_2, \ldots, w_m$, and

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^T.$$

(iii) Show that the optimal energy is given by

$$E_w(L) = \sum_{i=k+1}^{n} \lambda_i.$$

(iv) Suppose we minimize E_w over affine spaces $x_0 + L$, so

$$E_w(x_0, L) = \sum_{i=1}^{m} w_i \|x_i - x_0 - \mathrm{Proj}_L(x_i - x_0)\|^2.$$

Show that an optimal choice for x_0 is the weighted centroid

$$x_0 = \frac{\sum_{i=1}^m w_i x_i}{\sum_{i=1}^m w_i}.$$ △

Project 3.4.1 (Robust PCA)**. Python Notebook:** .ipynb

Robust PCA refers to a class of algorithms that minimize an energy of the form

$$E(x_0, L) = \sum_{i=1}^{m} \Phi(\|x_i - x_0 - \mathrm{Proj}_L(x_i - x_0)\|^2), \tag{3.4.2}$$

Choosing $\Phi(s) = s$ yields ordinary PCA, while $\Phi(s) = \sqrt{s}$ yields the robust sum of distances energy F given in (3.4.1). In this project you will implement an iteratively re-weighted least squares (IRLS) algorithm for minimizing the robust PCA energy (3.4.2), and test it on some data with outliers. Please refer to the Python notebook linked above and complete the steps below.

1. Write a Python function to implement the weighted PCA from Exercise 3.4.1.

2. Write Python code to minimize (3.4.2) using the IRLS method, which solves a sequence of weighted PCA problems

$$(x_0^{k+1}, L^{k+1}) = \min_{(x_0, L)} \sum_{i=1}^{m} w_i^k \|x_i - x_0 - \mathrm{Proj}_L(x_i - x_0)\|^2,$$

where

$$w_i^k = \frac{\Phi(\|x_i - x_0^k - \mathrm{Proj}_{L^k}(x_i - x_0^k)\|^2)}{\|x_i - x_0^k - \mathrm{Proj}_{L^k}(x_i - x_0^k)\|^2}.$$

Start with only a handful of iterations. You can then play with a stopping condition that checks how much the weights w_i^k change each iteration.

You may find that you divide by zero, or very small numbers, in the definition of the weights w_i^k, which can cause problems. A common way to regularize is to define

$$w_i^k = \frac{\Phi(\|x_i - x_0^k - \text{Proj}_{L^k}(x_i - x_0^k)\|^2)}{\|x_i - x_0^k - \text{Proj}_{L^k}(x_i - x_0^k)\|^2 + \varepsilon},$$

where ε is a small number, say $\varepsilon = 10^{-5}$.

3. Try your Robust PCA method on data with outliers. How many outliers do you need to add to break the algorithm?

△

3.5 PCA-based Image Compression

Python Notebook: .ipynb

We now give a first application of PCA to the problem of image compression. To apply PCA in this context, we must break an image up into pieces in such a way that the pieces are expected to be particularly simple and can be expressed well by PCA with a small number of principal components.

The simplest way to do this would be split the image up into its rows or columns. Let X be an $m \times n$ matrix representing an $m \times n$ grayscale image. If we split the image up by rows, then this amounts to applying PCA directly to the matrix X, where each row is considered as a data point. In this case we do not center the data, so based on Remark 3.1.3, the covariance matrix is $M = X^T X$. Let k denote the number of principal components to keep and let $P_k = \begin{bmatrix} p_1 & p_2 & \cdots & p_k \end{bmatrix}$ be the matrix whose columns are the principal components, and let L_k denote the span of $p_1, \ldots, p_k$. For a given vector x, PCA approximates x by its projection onto L_k, that is $\text{Proj}_{L_k} x = P_k P_k^T x$. Since we work with row vectors in X, let's consider the transpose of this quantity, so that $x_i^T P_k P_k^T$, is the PCA approximation of the i^{th} row of X, if $X = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^T$. Thus, the PCA approximation of X is given by

$$X \approx X P_k P_k^T.$$

Why does this amount to compression? To compute the right hand side above, we only need to store the matrix P_k, which is $n \times k$, and the matrix XP_k, which is $m \times k$. The latter matrix XP_k represents the coordinates of each row of X in the basis of L_k. Thus, instead of storing the $m \times n$ matrix X, we store two

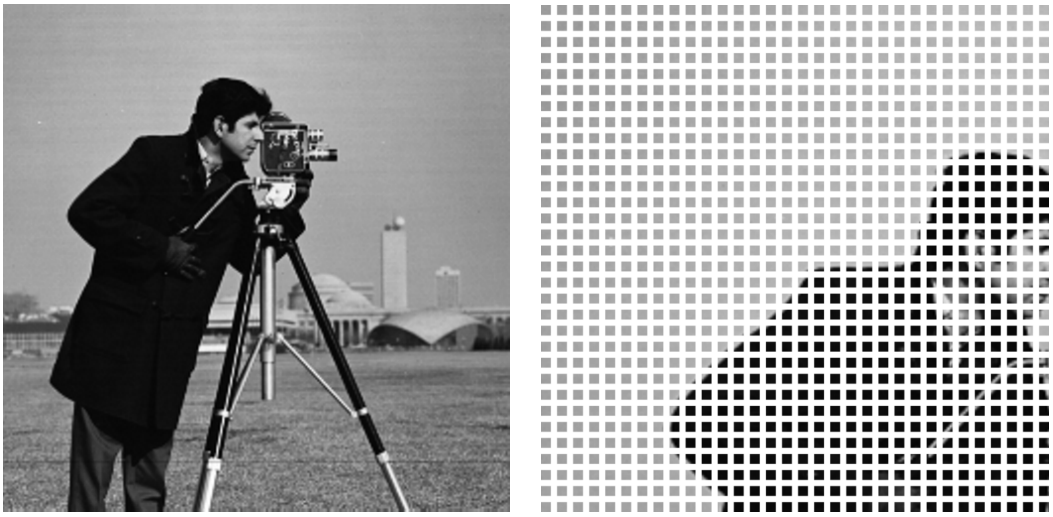


Figure 3.5.1: The cameraman image and the decomposition of part of the image into patches of size 8×8.

matrices of size $n \times k$ and $m \times k$. Often $m = n$ (the image is square), and so the compression ratio is is $n : k$.

While row-wise compression of images is an obvious first choice, the method misses important structure in the image, and the rows are not necessarily simple pieces of the image to compress. Since each row spans the entire image, the pixel values across the entire row may be entirely unrelated and have very little redundancy (which is useful for compression). Furthermore, why not use columns? Neighboring pixels vertically in the image should also have some correlation, but this is entirely ignored by row-wise image compression.

A better way to split up an image for compression is to use patches, or blocks, that are localized in space. The pixel intensities do not often vary rapidly in local areas of the image, and so small patches are expected to be well-approximated by a low dimensional PCA approximation. We work with the 512×512 cameraman image shown in Figure 1.3.2. We use 8×8 pixel patches in a regular grid, so the image contains $64 \times 64 = 4096$ patches, each containing $8 \times 8 = 64$ pixels. Figure 3.5.1 shows some of the patches of the cameraman image. Working with patches instead of rows requires a small amount of preprocessing of the image, to split it into patches. This produces a matrix X of size 4096×64, and we apply PCA to this matrix, instead of to the image itself, as we did in the row-wise compression example above. After this preprocessing, the compression proceeds exactly the same as in the row-wise compression example, and the decompressed image then needs to be

reconstructed from its patches.

The reconstruction error in image compression is measured with the peak signal to noise ratio (PSNR). The PSNR computation is based on the mean squared error

$$\text{MSE} = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(I(i,j) - I_0(i,j))^2.$$

Here, I_0 is the original image and I is the reconstructed image after compression. Both images have size $m \times n$. The PSNR computation also uses the peak signal value, S_{peak}, which is the largest possible value of the pixel intensity. For 8-bit images, the largest value is $S_{\text{peak}} = 2^8 - 1 = 255$. The PSNR is then given by

$$\text{PSNR} = 10\log_{10}\left(\frac{S_{\text{peak}}^2}{MSE}\right).$$

The PSNR is measured in decibels dB. PSNR values of 20 dB to 30 dB are very low quality images you may see on wireless devices, while 30 dB to 50 dB are respectable, and above 50 dB are very good quality compressions. This discussion is for 8-bit images, and the values would change for higher bit-depth images.

Figure 3.5.2 shows the compressed and difference cameraman images at three different compression ratios, with PSNR ranging from 34 dB up to 51 dB. The reader should note the blocking-type artifacts at higher compression ratios. These are caused by the decomposition of the image into patches, which allows for the reconstructed patches to differ greatly near the patch boundary. In Figure 3.5.3 we plot the PSNR versus compression ratio for patch-based (or block-based) image compression and row-wise image compression. This clearly shows the advantage of using block-based compression instead of row-wise compression.

The same types of blocking artifacts are present in older jpeg compression algorithms, which are based on the same principle of splitting the image into patches.[1] Instead of using PCA to find a basis for the patch space, the jpeg algorithm uses the Fourier series basis, which we will learn about later in the course. We show in Figure 3.5.4 the first 30 principal components obtained by applying PCA to the image patches. The principal components start of as low frequency, smooth, features, while the later components describe more high frequency content, like texture. When we study Fourier analysis later in

[1] The more recent jpeg2000 algorithm uses wavelet-based compression that does not require splitting the image into patches, and does not have blocking artifacts.

(a) Compression Ratio: 12.6:1, PSNR: 34.12 dB

(b) Compression Ratio: 6.3:1, PSNR: 36.61 dB

(c) Compression Ratio: 2.1:1, PSNR: 51.04 dB

Figure 3.5.2: Examples of PCA-based image compression on the cameraman image at different compression ratios. Left is original, center is compressed, and right is the difference image.

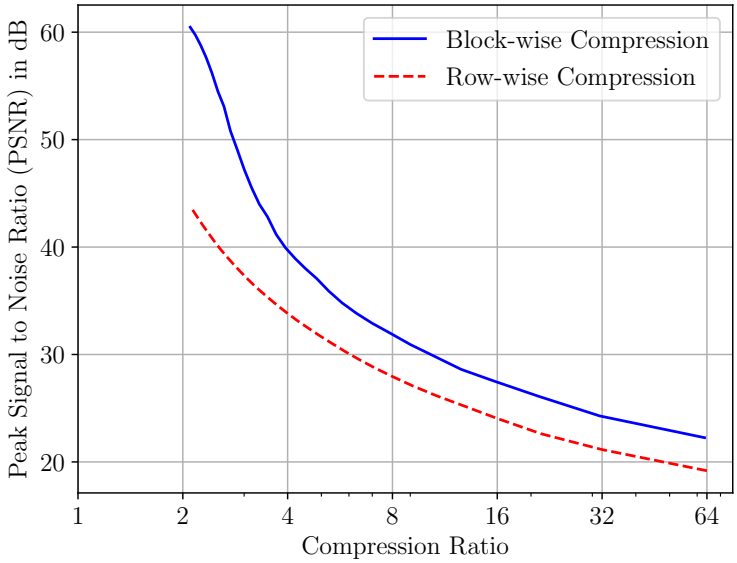


Figure 3.5.3: PSNR vs Compression Ratio for block-wise and row-wise compression of the cameraman image.

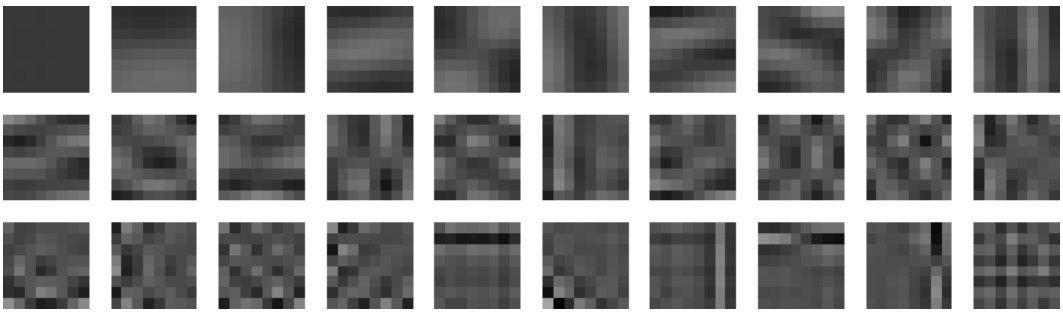


Figure 3.5.4: The first 30 principal components extracted via PCA on 8×8 pixel patches of the cameraman image.

the course, we will find that the Fourier basis for patch space looks strikingly similar to the principal components in Figure 3.5.4.

Let us also mention one aspect of compression we are omitting from this discussion. Normally the compressed data, which is XP_k and P_k, would be further compressed with a lossless compression algorithm (like zip compression) to save additional space. This adds further compression above the lossy

compression we have described in this section. The mathematics of lossless compression are unfortunately beyond the scope of this course.

Project 3.5.1 (Audio Compression)**. Python Notebook:** .ipynb

This project will use the same PCA-based compression algorithm for audio compression. There are two parts to the project, and the notebook above contains some code to get you started.

1. Use the same block-based image compression algorithm, described above, for audio compression. You can use any audio file you like; the python notebook linked above downloads a classical music sample from the course website that you can use. A stereo audio signal is an array of size $n \times 2$, where n is the number of samples. Use blocks of size $N \times 2$ for compression.

2. When you play back the compressed audio file, you will likely hear some static noise artifacts, even at very low compression rates. These are caused by blocking artifacts, where the signals do not match up on the edges of the blocks, which introduces discontinuities into the signal (which are very high in frequency). This is similar to the blocking artifacts we observed in image compression (see Figure 3.5.2), however, the artifacts are more noticeable in audio than in images.

   To fix this, audio compression algorithms use overlapping blocks, and apply a windowing function in order to smoothly patch together the audio in each block. The blocks are structured so that half of the first block overlaps with half of the second block, and so on. To implement this in python, just shift the signal by half of the block width, and apply the `images_to_patches` function on the original and shifted signals. Then compress and decompress both signals. After decompressing, and before converting back from the patch format to the audio signal, you'll need to multiply by a windowing function to smooth the transition between patches. If the patch size is $2 \times N$, then each channel should be multiplied by a window function w_i, $i = 0, 1, \ldots, N-1$. A common window function that is used, for example, in mp3 compression, is

$$w_i = \sin^2\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)\right).$$

   After you decompress and apply the window, undo the shift and add the signals together to get the decompressed audio. Does this improve the audio quality?

Figure 3.6.1: Mean images from each MNIST class.

As a note, in order to make sure the shifted signals add up correctly, we need that

$$w_i + w_{i+N/2} = 1.$$

As an exercise, the reader should check that the window function above satisfies this condition, which is called the Princen-Bradley condition.

△

3.6 PCA-based Handwritten Digit Recognition

Python Notebook: .ipynb

We now give a further application of PCA to the problem of handwritten digit recognition on the MNIST dataset, depicted in Figure 1.3.3. Recall MNIST is a dataset consisting of 70,000 images of handwritten digits 0 through 9. Each image is a 28×28 pixel grayscale image, which we can view as a vector in $\mathbb{R}^{784}$ by flattening the image array, since $28^2 = 784$.

PCA allows us to deduce a simple linear model for the images belonging to each digit. We choose a number of principal components k, and use PCA to learn affine subspaces $A_0, A_1, \ldots, A_9$ for each of the 10 digits. A new image of a handwritten digit is classified by projecting the image onto each of the 10 affine spaces, and finding which it is closest to. In order have held-out testing images that are not used for constructing the affine subspaces, we use the first 60,000 images in the dataset to build the affine spaces, and the last 10,000 to test the accuracy of the digit recognition. This is a standard training/testing split for the MNIST dataset.

To describe this mathematically, let $P_0, P_1, \ldots, P_9$ denote the $784 \times k$ matrices whose columns are the k principal components for each class. In this case, we center the data before applying PCA, so let $\overline{x}_0, \overline{x}_1, \ldots, \overline{x}_9$ denote the mean images from each class. The mean MNIST images are shown in Figure 3.6.1, and the first 10 principal components per class are shown in Figure 3.6.2. To project a flattened MNIST image $x \in \mathbb{R}^{784}$ onto the affine space A_i, we recall from Section 2.2 that the formula for projection onto an affine space is

$$\text{Proj}_{A_i} x = \overline{x}_i + P_i P_i^T (x - \overline{x}_i).$$

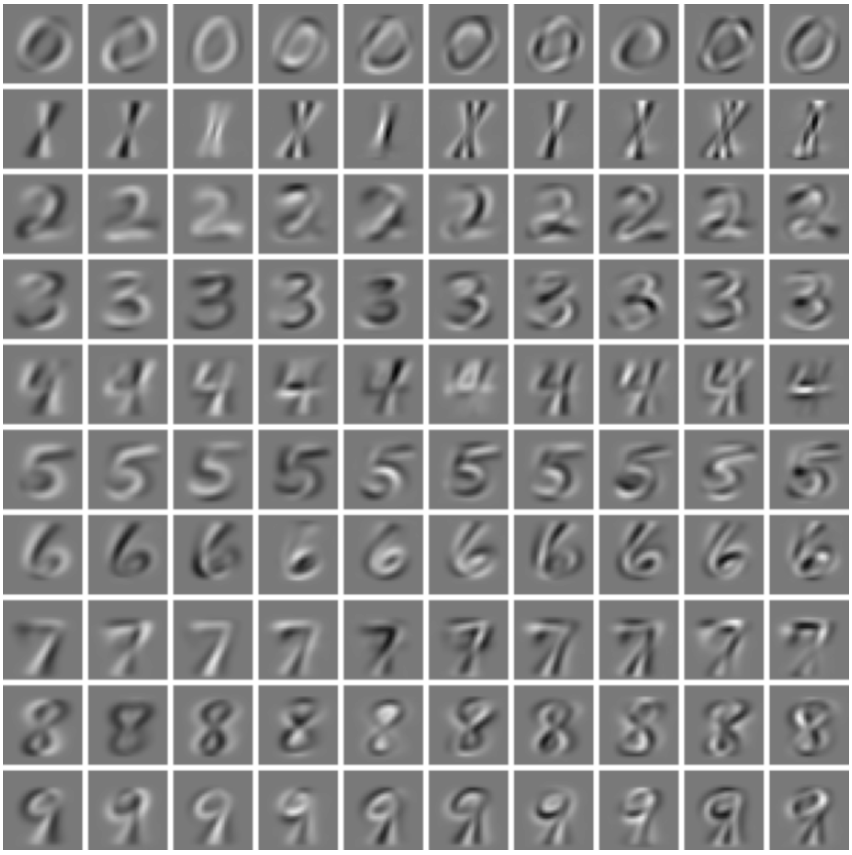


Figure 3.6.2: The first 10 principal components from each class of the MNIST dataset.

To check how far x is from the affine space A_i, we compute the norm of the difference between x and its projection, that is

$$d_i(x) := \|x - \text{Proj}_{A_i} x\| = \|(I - P_i P_i^T)(x - \overline{x}_i)\|.$$

The quantity d_i is the distance between x and the affine space A_i. We can then classify the image x by choosing the closest affine space, that is, the label $\ell_i(x)$ for image x is given by

$$\ell_i(x) = \underset{0 \le i \le 9}{\arg\min}\, d_i(x).$$

Running this on the MNIST dataset produces a classifier that achieves 95.8% accuracy on the 10,000 held out testing images. This is a very good result for such a basic algorithm. State of the art deep neural networks can achieve

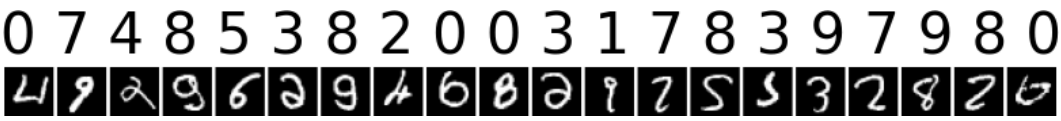


Figure 3.6.3: Some MNIST images that were incorrectly classified, along with the incorrectly predicted label.

over 99% accuracy. Figure 3.6.3 shows some of the incorrectly labeled digits, along with the incorrectly predicted label. We can see many of the incorrectly classified digits are poorly written and thus difficult to classify.

Remark 3.6.1. The PCA-based classifier we have constructed is a simple affine mapping $(I-P_iP_i^T)(x-\overline{x_i})$, followed by a nonlinear operation—computing the norm of this quantity. We then check which of $d_0(x), d_1(x), \ldots, d_9(x)$ is smallest to decide on the label for the image x. This is similar to a single layer neural network (a neuron is a linear function composed with a nonlinear activation function). We will cover the basics of neural networks later in the course.

Remark 3.6.2. In Python, and other programming languages, it is inefficient to compute with single images, so it is useful to write the formulas above in terms of a matrix X, whose rows are images from the MNIST dataset. Then we must work with the transposes of the quantities above, so

$$(X - \overline{x}_i^T)(I - P_iP_i^T)$$

represents the difference between the projections of all rows of X onto the i^{th} affine space A_i.

Let us mention briefly that the expression $X-\overline{x}_i^T$ is intended to mean that we subtract the row vector $\overline{x}_i^T$ from all rows of X. This type of expression is allowed in Python and other languages like Matlab. To be absolutely correct mathematically, the expression should be written as

$$(X - \mathbf{1}\overline{x}_i^T)(I - P_iP_i^T)$$

where $\mathbf{1}$ is the all-ones vector whose length is the same as the number of rows in X.

Project 3.6.1 (EigenFaces)**. Python Notebook:** .ipynb

In this project you will explore a PCA-based facial recognition algorithm that is known as *EigenFaces*. The Python notebook above has code to get

you started, including downloading and viewing a database of face images. The dataset has 2414 images of faces, from 38 different subjects. Each person appears many times in the database under different lighting conditions, etc. The goal of face recognition is to match a new image of a face to an image in an existing database. Please refer to the Python notebook above while completing the project steps below.

1. To have held-out data for testing, we will first split the face dataset into training and testing sets. Use the `train_test_split` function in `sklearn.model_selection` to split the dataset randomly into training (70%) and testing (30%).

2. Run PCA on the training images to learn an affine space that well approximates the training set. You can leave the number of principal components, k, as a parameter; around $k = 100$ gives good results.

3. Project both the training images and testing images onto the affine space, using the lower dimensional coordinates of the affine space. If the testing (or training) images are stored in a matrix X, where each row is a flattened image, and $\overline{x}$ is the mean face image computed by PCA in Step 1 (as a row vector), then the PCA coordinates are given by

$$(X - \overline{x})P,$$

   where P is the matrix containing the principle components as columns. Then match each testing image to the training image that is closest in Euclidean distance in the PCA coordinates.

4. You may not get very good results with the method above (around 60% accuracy). Try using the Mahalanobis distance instead of Euclidean distance to match faces in the PCA coordinates. For two vectors x and y of length k, representing the PCA coordinates of two images, the Mahalanobis distance is given by

$$d_M(x, y) = \sum_{i=1}^{k} \lambda_i^{-1}(x(i) - y(i))^2,$$

   where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$ are the eigenvalues of the covariance matrix (obtained by PCA in Step 1). The Mahalanobis distance rescales the norm along each coordinate to match the variation in the data in the corresponding principal direction. You should be able to get around 90% accuracy with the Mahalanobis distance, with some variation depending on the random training/testing split.

$\triangle$

Chapter 4

Clustering

We now turn to the problem of clustering, or grouping, data. Figure 4.1.1 shows a sample dataset consisting of 500 points that appear to belong to three distinct clusters. Two of the clusters are very close together, compared to the third, which more isolated. The goal of clustering is to separate the points in Figure 4.1.1 into the three natural clusters. In general, when working with real data and not synthetic examples, it is difficult to visualize the "natural clusters" within data, and it can be difficult to define what constitutes a good or bad clustering (since there are various natural ways one can group data).

In this section we will study two popular and widely used algorithms for clustering: k-means clustering, and spectral clustering. We will study the mathematics behind each algorithm, to the extent that we can in this course, and highlight the advantages and drawbacks of each method. We will also give applications to real data, by considering the problem of clustering pairs of digits from the MNIST dataset.

4.1 k-Means Clustering

Python Notebook: .ipynb

The k-means algorithm aims to find a single good representative point from each of k clusters. The dataset is then clustered into k groups by assigning each datapoint to the cluster corresponding to the closest such representative point in the Euclidean distance. To describe the setting mathematically, let $x_1, x_2, \ldots, x_m$ by a dataset consisting of m points in $\mathbb{R}^n$. Let $c_1, c_2, \ldots, c_k$ be the cluster centers, which are vectors in $\mathbb{R}^n$ and are yet to be determined. The k-means algorithm is guided by the task of minimizing the k-means clustering

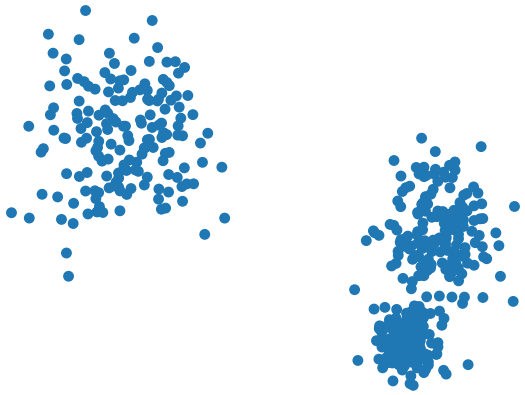


Figure 4.1.1: An example of a point cloud that has three clusters, one of which is substantially separated from the other two.

energy

$$E(c_1, c_2, \ldots, c_k) = \sum_{i=1}^{m} \min_{1 \le j \le k} \|x_i - c_j\|^2. \tag{4.1.1}$$

Minimizing E over the cluster centers (also called "means") $c_1, \ldots, c_k$ aims to find k points that well-represent the dataset, in the sense that all points are close to at least one c_j in the squared Euclidean distance. If we are able to minimize E, then the j^{th} cluster in the dataset, denoted Ω_j, consists of all points x_i that are closer to c_j than they are to any other cluster center. That is

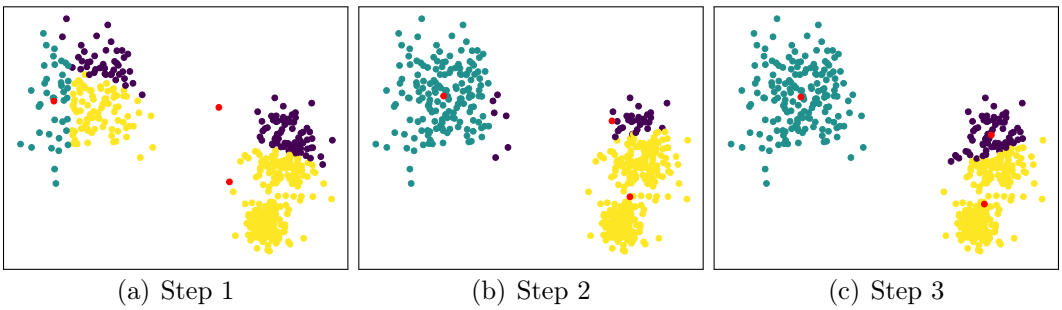
$$\Omega_j = \left\{ x_i \,:\, \|x_i - c_j\|^2 = \min_{1 \le \ell \le k} \|x_i - c_\ell\|^2 \right\}.$$

It turns out that minimizing the k-means clustering energy E is very hard computationally (it has been shown to be NP-hard). However, it is possible to construct a simple algorithm that descends on the energy E, is provably convergent (to a local minimizer), and often gives good clustering results. This is called the *k-means algorithm*, and is outlined below.

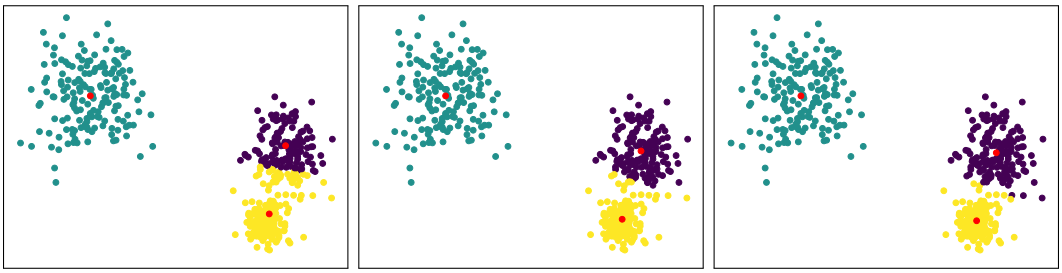
k-means algorithm: We start with some randomized initial values for the means $c_1^0, c_2^0, \ldots, c_k^0$, and iterate the steps below until convergence.

1. Update the clusters

$$\Omega_j^t = \left\{ x_i \,:\, \|x_i - c_j^t\|^2 = \min_{1 \le \ell \le k} \|x_i - c_\ell^t\|^2 \right\}. \tag{4.1.2}$$



(a) Step 1 (b) Step 2 (c) Step 3



(d) Step 4 (e) Step 5 (f) Step 8

Figure 4.1.2: An illustration of the intermediate steps in the k-means clustering algorithm. The red points are the cluster centroids. The algorithm converged in 8 steps, but steps 6,7,8 showed very little change in the clustering.

2. Update the cluster centers

$$c_j^{t+1} = \frac{1}{\#\Omega_j^t} \sum_{x \in \Omega_j^t} x. \tag{4.1.3}$$

Above, the notation $\#\Omega_j^t$ denotes the number of points in the j^{th} cluster Ω_j^t at the t^{th} step in the algorithm. Hence, c_j^{t+1} is exactly the mean of the j^{th} cluster Ω_j^t. The k-means algorithm generates a sequence of clusterings $\Omega_j^0, \Omega_j^1, \Omega_j^2, \ldots$ and cluster means $c_j^0, c_j^1, c_j^2, \ldots$, for $j = 1, \ldots, k$, that get progressively better in the sense that the k-means clustering energy (4.1.1) is decreasing (which we prove below). The algorithm converges when the clusters (and hence the cluster means) do not change from one iteration to the next, that is $\Omega_j^t = \Omega_j^{t+1}$ for all $j = 1, \ldots, k$. We also note that a point x_i may be equally close to more than one cluster center, and in this case we can make any reasonable choice of which cluster to assign it to, such as the cluster whose index is smallest.

We show in Figure 4.1.2 an illustration of some of the intermediate steps in applying k-means clustering with $k = 3$ (i.e., the 3-means clustering algorithm) to the point cloud from Figure 4.1.1. The algorithm converged in 8 steps to

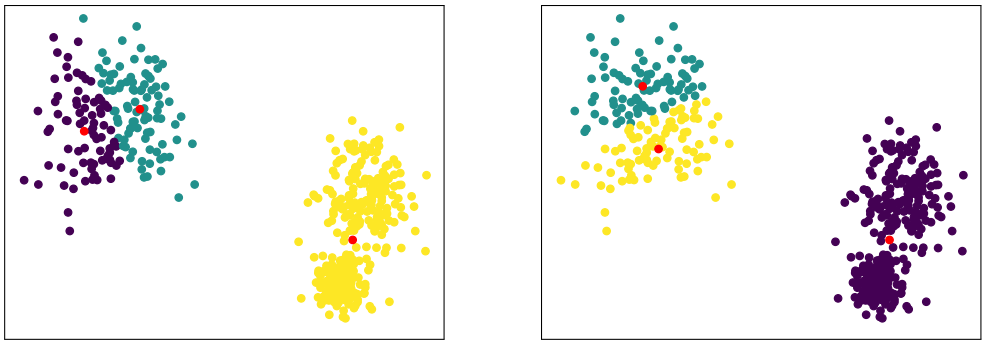


Figure 4.1.3: Two examples of poor clusterings obtained by the k-means algorithm. The final clustering obtained by k-means is not unique, and depends on the random initial condition.

a good clustering, although this depends on the randomized initial condition. For some initializations the algorithm converged in fewer iterations, sometimes as few as three, while for other initializations the algorithm took longer to converge. The clustering obtained can also depend on the initial condition. We show in Figure 4.1.3 two examples of poor clusterings obtained by 3-means clustering of the same point cloud.

The k-means algorithm often gives good results, but it does not find a global minimizer of the k-means clustering energy E defined in (4.1.1), and can converge to local minimizers that give poor clustering results. Nevertheless, we can prove that the k-means algorithm descends on the energy E, and converges in a finite number of iterations. The proof requires a preliminary lemma, which shows that the centroid (or mean) minimizes the sum of squared distances to the cluster center.

Lemma 4.1.1. *Let $y_1, y_2, \ldots, y_m$ be points in $\mathbb{R}^n$, and define the function $f : \mathbb{R}^n \to \mathbb{R}$ by*

$$f(x) = \sum_{i=1}^{m} \|y_i - x\|^2.$$

Then the unique minimizer of f is the centroid

$$c = \frac{1}{m} \sum_{i=1}^{m} y_i.$$

Proof. First, we claim that

$$f(c) = \sum_{i=1}^{m} (\|y_i\|^2 - \|c\|^2). \tag{4.1.4}$$

To see this, we first compute

$$\|c\|^2 = c^T c = \left(\frac{1}{m}\sum_{i=1}^{m} y_i\right)^T \left(\frac{1}{m}\sum_{j=1}^{m} y_j\right) = \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m} y_i^T y_j.$$

From this, it follows that

$$\begin{aligned} f(c) &= \sum_{i=1}^{m} \|y_i - c\|^2 \\ &= \sum_{i=1}^{m} \left(\|y_i\|^2 - 2y_i^T \frac{1}{m}\sum_{j=1}^{m} y_j + \|c\|^2 \right) \\ &= \sum_{i=1}^{m} \|y_i\|^2 - \frac{2}{m}\sum_{i=1}^{m}\sum_{j=1}^{m} y_i^T y_j + m\|c\|^2 \\ &= \sum_{i=1}^{m} \|y_i\|^2 - 2m\|c\|^2 + m\|c\|^2 \\ &= \sum_{i=1}^{m} \|y_i\|^2 - m\|c\|^2 = \sum_{i=1}^{m} (\|y_i\|^2 - \|c\|^2), \end{aligned}$$

which establishes the claim.

Now, using (4.1.4) we obtain

$$\begin{aligned} f(z) &= \sum_{i=1}^{m} \|y_i - z\|^2 \\ &= \sum_{i=1}^{m} \left(\|y_i\|^2 - 2z^T y_i + \|z\|^2 \right) \\ &= \sum_{i=1}^{m} \left(\|y_i\|^2 - \|c\|^2 \right) + \sum_{i=1}^{m} \left(\|c\|^2 - 2z^T y_i + \|z\|^2 \right) \\ &= f(c) + m\|c\|^2 - 2mz^T c + m\|z\|^2 \\ &= f(c) + m\|c - z\|^2 \end{aligned}$$

for any $z \in \mathbb{R}^n$. Therefore $f(c) \leq f(z)$ for all $z \in \mathbb{R}^n$, with equality if and only if $z = c$. This completes the proof. □

We can now prove convergence of the k-means algorithm.

Theorem 4.1.2. *The k-means algorithm descends on the energy (4.1.1), that is*

$$E(c_1^{t+1}, c_2^{t+1}, \dots, c_k^{t+1}) \leq E(c_1^t, c_2^t, \dots, c_k^t). \tag{4.1.5}$$

Furthermore, we have equality in (4.1.5) if and only if $c_j^{t+1} = c_j^t$ for $j = 1, \dots, k$, and hence the k-means algorithm converges in a finite number of iterations.

Proof. The proof is based on re-writing the k-means energy in the following way:

$$E(c_1^t, c_2^t, \dots, c_k^t) = \sum_{j=1}^{k} \sum_{x \in \Omega_j^t} \|x - c_j^t\|^2.$$

This follows from the definition of the clusters Ω_j^t defined in (4.1.2). Now, it follows from Lemma 4.1.1 that

$$\sum_{x \in \Omega_j^t} \|x - c_j^{t+1}\|^2 \leq \sum_{x \in \Omega_j^t} \|x - c_j^t\|^2$$

with equality if and only if $c_j^{t+1} = c_j^t$, due to the definition of c_j^{t+1} in (4.1.3) as the mean of the j^{th} cluster Ω_j^t at step t. Therefore

$$\sum_{j=1}^{k} \sum_{x \in \Omega_j^t} \|x - c_j^{t+1}\|^2 \leq E(c_1^t, c_2^t, \dots, c_k^t)$$

with equality if and only if $c_j^{t+1} = c_j^t$ for $j = 1, \dots, k$. Since the k-means energy uses the squared distance to the closest cluster center, we trivially have

$$\begin{aligned} E(c_1^{t+1}, c_2^{t+1}, \dots, c_k^{t+1}) &= \sum_{i=1}^{m} \min_{1 \leq j \leq k} \|x_i - c_j^{t+1}\|^2 \\ &\leq \sum_{j=1}^{k} \sum_{x \in \Omega_j^t} \|x - c_j^{t+1}\|^2 \\ &\leq E(c_1^t, c_2^t, \dots, c_k^t), \end{aligned}$$

again with equality if and only if $c_j^{t+1} = c_j^t$ for $j = 1, \dots, k$.

We now show that this implies convergence of the k-means algorithm. Note that if $c_j^{t+1} \neq c_j^t$ for some j (so the algorithm has not converged), then we proved above that the energy is strictly decreasing, so

$$E(c_1^{t+1}, c_2^{t+1}, \dots, c_k^{t+1}) < E(c_1^t, c_2^t, \dots, c_k^t).$$

Hence, prior to convergence, we can never revisit the same clustering

$$\Omega_1^t, \Omega_2^t, \ldots, \Omega_k^t$$

at any step in the k-means algorithm. Indeed, if we were to revisit the same clustering at some future step of the algorithm, then since the means $c_1^{t+1}, c_2^{t+1}, \ldots, c_k^{t+1}$ depend only on the clusters $\Omega_1^t, \Omega_2^t, \ldots, \Omega_k^t$, we would revisit the same configuration of cluster centers, which is impossible since the k-means energy is strictly decreasing with each iteration prior to convergence. Since there are only a finite number of possible ways to cluster the dataset into k groups, and the k-means algorithm cannot revisit any given clustering, the algorithm must eventually converge. □

Remark 4.1.3. The proof of Theorem 4.1.2 uses strict energy monotonicity to prove convergence. The proof is non-quantitative, meaning it does not say anything about how many iterations the k-means algorithm may take to converge. In practice, the algorithm often converges very quickly, in only a handful of iterations, but it is possible for k-means to take substantially longer to converge. Indeed, in the worst case, the algorithm must visit every possible clustering before converging. Even for the 2-means problem with n points, there 2^n possible ways to cluster the data, so the search space is exponentially large.

We also remark that convergence of the k-means algorithm simply means that the cluster centers stop changing from one iteration to the next. This does not mean the algorithm has converged to a minimizer of the k-means energy (4.1.1), and in general the algorithm does not find global minimizers. In fact, due to the random choice of initialization, the algorithm can find different clusterings every time it is executed.

Exercise 4.1.4 (Robust k-means clustering)**.** The k-means clustering algorithm is sensitive to outliers, since it uses the squared Euclidean distance. We consider the robust k-means energy

$$E_{robust}(c_1, c_2, \ldots, c_k) = \sum_{i=1}^{m} \min_{1 \le j \le k} \|x_i - c_j\|. \tag{4.1.6}$$

The robust k-means algorithm attempts to minimize (4.1.6). We start with some randomized initial values for the means $c_1^0, c_2^0, \ldots, c_k^0$, and iterate the steps below until convergence.

1. Update the clusters as in (4.1.2).

2. Update the cluster centers

$$c_j^{t+1} \in \arg\min_{y \in \mathbb{R}^n} \sum_{x \in \Omega_j^t} \|x - y\|. \tag{4.1.7}$$

Complete the following exercises.

(i) Show that the Robust PCA algorithm descends on the energy E_{robust}.

(ii) The cluster center (4.1.7) does not admit a closed form expression and is sometimes inconvenient to work with in practice. Consider changing the Euclidean norm in (4.1.6) to the ℓ^1-norm $\|x\|_1 = \sum_{i=1}^n |x(i)|$, and define

$$E_{\ell^1}(c_1, c_2, \ldots, c_k) = \sum_{i=1}^m \min_{1 \le j \le k} \|x_i - c_j\|_1.$$

Formulate both steps of the k-means algorithm so that it descends on E_{ℓ^1}. Show that the cluster centers c_j^{t+1} are the coordinatewise medians of the points $x \in \Omega_j^t$, which are simple to compute.

(iii) Can you think of any reasons why the Euclidean norm would be preferred over the ℓ^1 norm in the k-means energy?

△

Exercise 4.1.5 (Optimal clustering in 1D)**.** We consider here the 2-means clustering algorithm in dimension $n = 1$. Let $x_1, x_2, \ldots, x_m \in \mathbb{R}$ and recall the 2-means energy is

$$E(c_1, c_2) = \sum_{i=1}^m \min\left\{(x_i - c_1)^2, (x_i - c_2)^2\right\}.$$

Throughout the question we assume that the x_i are ordered so that

$$x_1 \le x_2 \le \cdots \le x_m.$$

For $1 \le j \le m-1$ we define

$$\mu^-(j) = \frac{1}{j}\sum_{i=1}^j x_i, \quad \mu^+(j) = \frac{1}{m-j}\sum_{i=j+1}^m x_i,$$

and

$$F(j) = \sum_{i=1}^j (x_i - \mu^-(j))^2 + \sum_{i=j+1}^m (x_i - \mu^+(j))^2.$$

(i) Explain how $F(j)$ differs from the 2-means energy $E(c_1, c_2)$, and why minimizing $F(j)$ over $j = 1, \ldots, m-1$ and setting $c_1 = \mu_-(j_*)$ and $c_2 = \mu^+(j_*)$ will give a solution at least as good as the 2-means algorithm (here, j_* is a minimizer of $F(j)$).

(ii) By (i) we can replace the 2-means problem with minimizing $F(j)$. We will now show how to do this efficiently. In this part, show that

$$F(j) = \sum_{i=1}^{m} x_i^2 - j\mu^-(j)^2 - (m-j)\mu^+(j)^2.$$

Thus, minimizing $F(j)$ is equivalent to maximizing

$$G(j) = j\mu^-(j)^2 + (m-j)\mu^+(j)^2.$$

(iii) Show that we can maximize G (i.e., find j_* with $G(j) \leq G(j_*)$ for all j) in $O(m \log m)$ computations. Hint: First show that

$$\mu^-(j+1) = \frac{j}{j+1}\mu^-(j) + \frac{x_{j+1}}{j+1},$$

and

$$\mu^+(j+1) = \frac{m-j}{m-j-1}\mu^+(j) - \frac{x_{j+1}}{m-j-1}.$$

and explain how these formulas allow you to compute all the values $G(1), G(2), \ldots, G(m-1)$ recursively in $O(m \log m)$ operations, at which point the maximum is found by brute force.

(iv) [Challenge] Implement the method described in the previous three parts in Python. Test it out on some synthetic 1D data. For example, you can try a mixture of two Gaussians with different means.

△

4.1.1 Clustering MNIST digits

Python Notebook: .ipynb

We now consider a brief application of k-means clustering to real data. Again, we use the MNIST dataset of handwritten digits, and we evaluate the 2-means algorithm for clustering pairs of MNIST digits. We consider all pairs of MNIST digits (there are around 7000 images per digit), yielding $\binom{10}{2}$

Digit	1	2	3	4	5	6	7	8	9
0	99.1	95.0	95.0	96.5	86.8	92.9	96.9	94.5	95.6
1		92.7	96.1	97.9	91.1	96.4	95.8	94.3	97.2
2			89.7	95.7	94.6	93.8	95.6	91.2	95.7
3				97.3	66.3	97.6	96.4	80.0	94.0
4					88.2	95.6	95.3	95.7	52.7
5						91.4	87.2	52.3	94.2
6							99.1	96.6	98.9
7								95.9	60.5
8									59.8

Table 4.1.1: Accuracy for binary (2-means) clustering of pairs of MNIST digits. We see most pairs of digits are easy to separate, while a few pairs, such as (4,9), (5,3), (7,9), (5,8), and (8,9), are more difficult.

binary clustering problems, each with around 14000 datapoints in dimension $\mathbb{R}^{784} = \mathbb{R}^{28\times 28}$. The k-means algorithm converged very quickly, in around 15 iterations taking around 1 second per clustering problem. Table 4.1.1 shows the clustering accuracy obtained by the 2-means algorithm for each pair of MNIST digits. The numbers can vary depending on the random choice of initial condition. We can see that many pairs of digits are very easy to cluster into the correct classes with the 2-means algorithm, while a handful of pairs of digits, such as (4,9), (5,3), (7,9), (5,8), and (8,9) are more difficult. It is also natural to run the 10-means algorithm on the whole MNIST dataset, but the algorithm takes a long time to converge and generally gives poor clustering results.

4.2 Spectral Clustering

Python Notebook: .ipynb

The k-means clustering algorithm works well for clusters that are roughly spherical (e.g., blob data). For clusters with more complicated geometries, a single cluster center may not be a good representative of the whole cluster, and Euclidean distance to cluster centers may not be a good indication of which cluster a datapoint belongs to. We show an example of this on the famous 2-moons dataset in Figure 4.2.1. This dataset has two clusters that have a nonlinear, and non-convex, shape, and are interleaved so that a single there

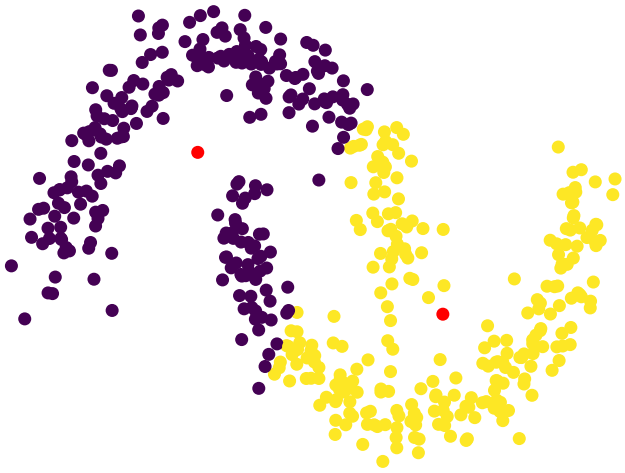


Figure 4.2.1: 2-means clustering on the two-moons dataset fails to uncover the true clusters.

are no good choice of Euclidean cluster centers for either cluster. In this case, 2-means clustering performs poorly. In this section we will study another algorithm, called *spectral clustering*, which can uncover more complicated cluster geometries.

Spectral clustering takes a different perspective on clustering, compared to k-means clustering. In k-means clustering we attempted to find k cluster centers, and clustered the data by finding, for each datapoint, which cluster center is closest in Euclidean distance. In contrast, spectral clustering aims to ensure that points that are nearby spatially are assigned to the same cluster. It does not compute cluster centers, and makes no assumption on the shape or geometry of the clusters.

Spectral cluster requires the computation of an adjacency matrix, or *weight matrix*, for the dataset, which encodes the similarities between pairs of points. If our dataset consists of m points $x_1, x_2, \ldots, x_m$ in $\mathbb{R}^n$, the weight matrix W is an $m \times m$ symmetric matrix, where the entry $W(i,j)$ represents the similarity between datapoints x_i and x_j. The similarity is always nonnegative ($W(i,j) \geq 0$), and should be large when x_i and x_j are close together spatially, and small (or zero), when x_i and x_j are far apart. A common choice for the weight matrix is Gaussian weights

$$W(i,j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right), \tag{4.2.1}$$

where the σ is a free parameter that controls the scale at which points are connected by strong similarities $W(i,j)$ in the weight matrix. In fact, the

weight matrix W endows the dataset with a graph structure, where each pair of points (x_i, x_j) is connected by an edge with edge weight $W(i,j)$. Other choices of weight matrix are possible, such as the k-nearest neighbor graph explored in Section 4.2.2, so we proceed with a general nonnegative and symmetric $m \times m$ weight matrix W in what follows.

While spectral clustering can be applied to general problems with k clusters, it is simplest to explain for the binary clustering problem, where we seek $k = 2$ clusters. Let $z \in \{0,1\}^m$ be a vector giving the cluster labels for each datapoint, so $z(i) \in \{0,1\}$ is the cluster label for x_i. Here, we have assigned one cluster to have the label 0 and the other cluster the label 1. Given a weight matrix W, we wish to find a binary clustering for which x_i and x_j are assigned to the same cluster when the weight $W(i,j)$ is large (since this means the points are nearby). It is thus natural to consider the graph cut energy

$$E(z) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)|z(i) - z(j)|^2. \tag{4.2.2}$$

The graph cut energy sums the edge weights $W(i,j)$ corresponding to pairs of points (x_i, x_j) that are assigned to different clusters, so that $z(i) \neq z(j)$. A graph cut approach to clustering amounts to attempting to minimize E over cluster labels z, which attempts to ensure that similar datapoints, where $W(i,j)$ is large, get assigned to the same cluster. In the presence of outliers, this can give poor clusterings, since it tends to put a single outlier in one class, and all other datapoints in the other class. Thus, it is more common to consider a balanced graph-cut energy, which penalizes unbalanced clusters. One possibility is the balanced energy

$$E_{balanced}(z) = \frac{\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)|z(i) - z(j)|^2}{\sum_{i=1}^{n} z(i) \sum_{j=1}^{n} (1 - z(j))}. \tag{4.2.3}$$

The denominator in the balanced energy is the product of the sizes of the two clusters, which is largest when the classes are balanced. Minimizing a balanced graph-cut energy gives much better clusterings, but it turns out to be a very hard problem computationally (in fact, it is NP hard).

Spectral clustering is a tractable relaxation of the graph cut problem described above. Instead of enforcing that the cluster labels $z(i)$ belong to the discrete set $\{0,1\}$, we relax the problem and allow $z \in \mathbb{R}^m$ to be any real-valued vector. This introduces the problem that any constant vector $z = t\mathbf{1}$ is a minimizer of the graph-cut energy E, where $t \in \mathbb{R}$ and $\mathbf{1}$ is the all ones vector. We could instead consider minimizing the balanced energy (4.2.3), but

it is more convenient to use slightly different balancing conditions. The two natural balancing conditions used in spectral clustering are

$$\mathbf{1}^T z = \sum_{i=1}^{m} z(i) = 0 \quad \text{and} \quad \|z\|^2 = \sum_{i=1}^{m} z(i)^2 = 1. \tag{4.2.4}$$

The two constraints in (4.2.4), taken together, ensure that the any minimizer z of $E(z)$ is not a trivial constant labeling $z = t\mathbf{1}$, and that the classes are balanced to some degree. This leads to the *binary spectral clustering* problem:

$$\text{Minimize } E(z) \text{ over } z \in \mathbb{R}^m \text{ subject to (4.2.4).} \tag{4.2.5}$$

After finding a minimizer z^*, the clustering is obtained by the sign of the minimizer, so $\{x_i : z^*(i) > 0\}$ is one cluster, while $\{x_i : z^*(i) \leq 0\}$ is the other cluster.

4.2.1 The graph Laplacian and Fiedler vector

We now show how to solve the binary spectral clustering problem (4.2.5), which will explain the *spectral* part of the name.

Definition 4.2.1. Let W be a symmetric $m \times m$ matrix with nonnegative entries. The associated *graph Laplacian* matrix L is the $m \times m$ matrix

$$L = D - W \tag{4.2.6}$$

where D is the diagonal matrix with diagonal entries

$$D(i,i) = \sum_{j=1}^{m} W(i,j).$$

We now show that the graph cut energy can be rewritten in terms of the graph Laplacian matrix L.

Lemma 4.2.2. *Let W be a symmetric $m \times m$ matrix with nonnegative entries. Then the graph cut energy E defined in (4.2.2) can be expressed as*

$$E(z) = z^T L z, \tag{4.2.7}$$

where $L = D - W$ is the graph Laplacian matrix.

Proof. We simply compute

$$\begin{aligned}
E(z) &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)|z(i)-z(j)|^2 \\
&= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)(z(i)^2 - 2z(i)z(j) + z(j)^2) \\
&= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)z(i)^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)z(i)z(j) \\
&\qquad\qquad + \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)z(j)^2 \\
&= \frac{1}{2}\sum_{i=1}^{m} D(i,i)z(i)^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)z(i)z(j) + \frac{1}{2}\sum_{j=1}^{m} D(j,j)z(j)^2 \\
&= \sum_{i=1}^{m} D(i,i)z(i)^2 - \sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)z(i)z(j) \\
&= z^T D z - z^T W z = z^T L z,
\end{aligned}$$

which completes the proof. □

We recall from Exercise 2.3.2 that minimizing the quantity $z^T L z$, subject to the constraint $\|z\| = 1$, is related to finding eigenvectors of L. Due to (4.2.4) we have an additional constraint $\mathbf{1}^T z = 0$ that needs to be accounted for. Before proceeding, we record some basic properties of the graph Laplacian.

Lemma 4.2.3. *Let $L = D - W$ be the graph Laplacian corresponding to a symmetric matrix W with nonnegative entries. The following properties hold.*

(i) *L is symmetric.*

(ii) *L is positive semi-definite (i.e., $z^T L z \geq 0$ for all $z \in \mathbb{R}^m$).*

(iii) *All eigenvalues of L are nonnegative, and the constant vector $z = \mathbf{1}$ is an eigenvector of L with eigenvalue $\lambda = 0$.*

Proof. (i) Both D and W are symmetric, so L is as well.
(ii) By Lemma 4.2.2 we have

$$z^T L z = E(z) \geq 0,$$

for any $z \in \mathbb{R}^m$, thus L is positive semi-definite.

(iii) By Exercise 2.3.3, all eigenvalues of L are nonnegative, since L is positive semi-definite. To check that $z = \mathbf{1}$ belongs to the kernel of L, let $y = L\mathbf{1} = D\mathbf{1} - W\mathbf{1}$, and check that

$$y(i) = D(i,i) - \sum_{j=1}^{m} W(i,j) = 0$$

for all $i = 1, \ldots, m$. This completes the proof. □

Since L is a real symmetric matrix, it is diagonalizable, and there exists an orthonormal basis $v_1, v_2, \ldots, v_m$ of $\mathbb{R}^m$ consisting of eigenvectors of L, with corresponding eigenvalues

$$0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_m.$$

That is, $Lv_i = \lambda_i v_i$. Note that $\lambda_1 = 0$ by Lemma 4.2.3 (iii).

Definition 4.2.4. The second eigenvector v_2 of the graph Laplacian L is called the *Fiedler vector*.

The Fiedler vector is the first non-trivial eigenvector of L, when they are ordered by increasing eigenvalue. The first eigenvector is constant, $v_1 = \frac{1}{\sqrt{m}}\mathbf{1}$, and is considered trivial.

It turns out, due to the theorem below, that the Fiedler vector is exactly the solution of the binary spectral clustering problem (4.2.5). Thus, spectral clustering amounts to assigning points to one of two clusters depending on the sign of the Fiedler vector. This is also the reason for the name *spectral clustering*—we are using the spectrum (i.e., the eigenvectors) of the graph Laplacian matrix for clustering.

Theorem 4.2.5. *The Fiedler vector v_2 solves the spectral clustering problem (4.2.5).*

Proof. A minimizer of (4.2.5) exists, due to the fact that we are minimizing a continuous function $E(z)$ over a closed and bounded set

$$\{z \in \mathbb{R}^m \,:\, \|z\| = 1 \;\text{ and }\; \mathbf{1}^T z = 0\}.$$

Let z be a minimizer of (4.2.5), and write z in the basis of eigenvectors of L as

$$z = \sum_{i=1}^{m} a_i v_i.$$

Since $\|z\| = 1$ we have by (2.1.5) that

$$\sum_{i=1}^{m} a_i^2 = 1. \tag{4.2.8}$$

Since $v_1 = \frac{1}{\sqrt{m}}\mathbf{1}$, and the v_i are orthonormal vectors, the condition $\mathbf{1}^T z = 0$ is equivalent to

$$0 = \mathbf{1}^T z = \sqrt{m} v_1^T \sum_{i=1}^{m} a_i v_i = \sqrt{m} \sum_{i=1}^{m} a_i v_1^T v_i = \sqrt{m} a_1.$$

Therefore $a_1 = 0$. We now compute, by Lemma 4.2.2, that

$$E(z) = z^T L z = z^T L \sum_{i=2}^{m} a_i v_i = \sum_{i=2}^{m} a_i z^T L v_i = \sum_{i=2}^{m} \lambda_i a_i z^T v_i = \sum_{i=2}^{m} \lambda_i a_i^2,$$

since $z^T v_i = a_i$. Since $\lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_m$, $a_1 = 0$, and (4.2.8) holds, we have

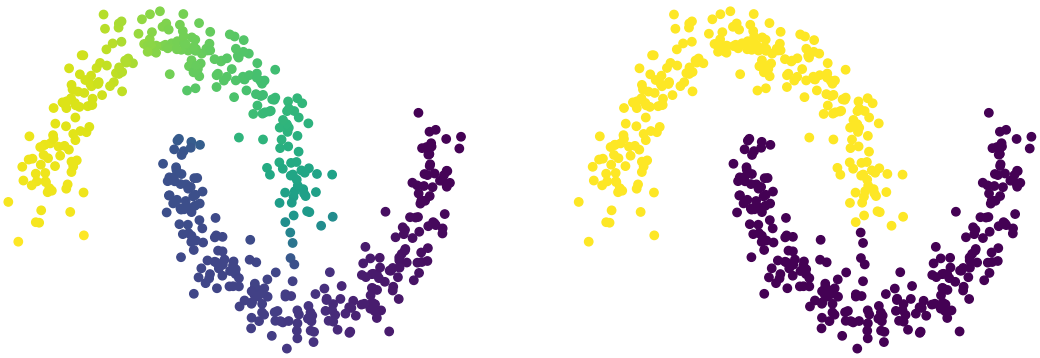
$$E(z) = \sum_{i=2}^{m} \lambda_i a_i^2 \geq \lambda_1 \sum_{i=2}^{m} a_i^2 = \lambda_1.$$

Therefore, setting $a_2 = 1$ and $a_i = 0$ for $i \neq 1$ minimizes $E(z)$. This amounts to $z = v_2$, which completes the proof. □

We return briefly to the two-moons clustering problem from Figure 4.2.1, which was not clustered correctly by k-means. We applied spectral clustering with Gaussian weights $W(i, j)$ (see (4.2.1)) with $\sigma = 0.15$. Figure 4.2(a) shows the Fiedler vector colored with lowest values dark blue, and highest values yellow. Figure 4.2(b) shows the result of spectral clustering on the two-moons dataset, where the clusters are based on the sign of the Fiedler vector. Spectral clustering finds the correct clustering for the two-moons dataset, which illustrates the ability of spectral clustering to handle more complicated nonlinear cluster geometry.

Remark 4.2.6. We briefly remark that spectral clustering is not only used for binary clustering, and there are versions of the algorithm that work for k classes. The general idea is to compute the first k eigenvectors $v_1, v_2, \ldots, v_k$ of the graph Laplacian $L = D - W$ and to perform a *spectral embedding* of the data into $\mathbb{R}^k$. The spectral embedding $\Phi_k : \mathbb{R}^n \to \mathbb{R}^k$ is

$$\Phi_k(x_i) = (v_1(i), v_2(i), \ldots, v_k(i)).$$



(a) Fiedler vector (b) Spectral Clustering

Figure 4.2.2: (a) The Fiedler vector and (b) spectral clustering on the 2-moons dataset.

That is, at each datapoint x_i, we evaluate all k eigenvectors of the graph Laplacian and those values are the coordinates of the points x_i in the k-dimensional embedded space. This is a very useful method for dimension reduction. The data is then clustered in the spectral embedding space $\mathbb{R}^k$, often using the k-means clustering algorithm. There are several different variants of spectral clustering for more than two classes, based on using different normalizations for the graph Laplacian (two common normalizations are $D^{-1}L$ and $D^{-1/2}LD^{-1/2}$), and on using further normalizations of the points in the embedded space $\mathbb{R}^k$. We refer to [19] for more information on spectral clustering.

4.2.2 Clustering MNIST digits

Python Notebook: .ipynb

We now return to the problem of clustering MNIST digits from Section 4.1.1, to see if we can improve upon the results of k-means clustering. The choice of Gaussian weights (4.2.1) is not practical for real data for a couple of reasons. First, it results in a dense matrix W, where all entries need to be stored, which is not tractable for large datasets. Second, there are difficulties using the same scale σ over the whole graph, since some areas of the graph may be very dense and a smaller connectivity scale σ would be appropriate, while some areas may be more sparse and require a larger σ to ensure they are well-connected to neighboring points in the graph.

A better way to construct a graph over a dataset in practice is to build a

Digit	1	2	3	4	5	6	7	8	9
0	99.8	98.9	99.5	99.8	99.5	98.7	99.7	99.2	99.3
1		97.0	99.3	99.1	99.4	99.7	98.8	99.1	99.6
2			98.3	99.5	99.1	99.5	98.0	98.6	99.3
3				99.6	82.3	99.6	99.0	91.8	97.9
4					99.6	99.3	98.9	98.9	53.4
5						97.9	99.8	90.0	98.3
6							99.8	99.0	99.7
7								99.1	70.9
8									97.0
9									

Table 4.2.1: Accuracy for binary spectral clustering of pairs of MNIST digits. The results are generally an improvement over 2-means clustering (see Table 4.1.1), but we still see some pairs of digits, such a (4,9) and (7,9) are hard to separate.

k-nearest neighbor graph. To do this, we find for each x_i, the k nearest points x_j in the Euclidean distance, and we assign positive edge weights $W(i,j)$ only for these k-nearest neighbors. All other weights are zero, and the matrix is stored in a sparse matrix format so the zero entries do not need to be stored in memory, nor used in computations. The weights can be defined as $W(i,j) = 1$ when j is a k-nearest neighbor of i, or we can use Gaussian weights with a length scale that adapts to the graph. In this experiment we used

$$W(i,j) = \exp\left(-\frac{4\|x_i - x_j\|^2}{d_k(x_i)^2}\right),$$

where $d_k(x_i)$ is the Euclidean distance from x_i to its k^{th} nearest neighbor.

A minor problem with k-nearest neighbor graphs is that they are not symmetric, that is $W(i,j) \neq W(j,i)$. This means the graph Laplacian matrix $L = D - W$ will not be symmetric, and hence it may not be diagonalizable, the spectrum may be complex-valued, and the Fiedler vector will have no meaning. Thus, we always symmetrize the weight matrix for a k-nearest neighbor graph in some reasonable way before applying spectral clustering. In this experiment, we replace W with $\frac{1}{2}(W + W^T)$ to symmetrize. Other choices are possible, such as symmetrizing the weights directly

$$W(i,j) = \exp\left(-\frac{4\|x_i - x_j\|^2}{d_k(x_i)d_k(x_j)}\right).$$

We show in Table 4.2.2 the results of binary spectral clustering of all pairs of MNIST digits using a $k = 10$ nearest neighbor graph. We generally see a good improvement over the corresponding results for the k-means algorithm given in Table 4.1.1. However, there are still some pairs of digits that are difficult to separate, such as $(4, 9)$ and $(7, 9)$. We will return to this example later in the lecture notes, and we will see how the results can be further improved by using neural network autoencoders.

Chapter 5

PageRank

Python Notebook: .ipynb

Having introduced graph-based methods briefly in Section 4.2, we turn here to study the PageRank algorithm, which was used by Google to rank internet search results until around 2006. While PageRank originally gained popularity as Google's search engine, it has found a wide range of applications in other fields, including biology (GeneRank), chemistry, ecology, neuroscience, physics, sports, and computer systems. We refer to [9] for a review of PageRank and a summary of the abundance of applications outside of search engines.

PageRank is a graph-based ranking algorithm that ranks websites based on their link structure. The idea behind the PageRank algorithm is to take a random walk on the internet for a long time (by randomly following links on each webpage visited) and record how often each website is visited along the way. The websites that are visited more often get higher ranks than those that are visited less often. While this is the core of the idea behind PageRank, there is a small problem with a simple random walk on a graph like the internet—the walker may get stuck in small disconnected components of the graph that have no outgoing links. The walker will then spend all of its time in a small part of the internet (which may be very sensitive to where the walk starts), and will be unable to rank the majority sites.

To resolve this issue, PageRank uses a *random surfer*, which is a random walker that at random times teleports to a completely random website, one that is not necessarily connected to the current website by a link. This allows the surfer to escape poorly connected pockets of the internet and produce a ranking for all pages. Furthermore, specific choices of the type of teleportation lead to localized or personalized versions of PageRank, which are discussed in Section 5.2.

To describe the PageRank algorithm mathematically, the starting point is an adjacency matrix W, which encodes the links between websites. Often the adjacency matrix is a binary matrix with

$$W(i,j) = \begin{cases} 1, & \text{if site } i \text{ links to site } j \\ 0, & \text{otherwise.} \end{cases}$$

The adjacency matrix W is an $n \times n$ matrix, where n is the number of webpages to rank. The matrix W is a very large matrix that is usually very sparse. That is, most of the entires of W are zero, since most websites link to only a small fraction of the internet. Matrix operations with W are thus tractable if one stores the matrix in a sparse format, so that the zero entries are not stored in memory, and not used in matrix/vector multiplications.

To define a random walk on a graph, we need to define a probability transition matrix P, which is an $n \times n$ matrix whose (i,j) entry is the one-step transition probability

$$P(i,j) = \text{Probability of stepping from } j \text{ to } i.$$

Clicking on a link at random from webpage j leads to the transition probabilities

$$P(i,j) = \frac{W(j,i)}{\sum_{k=1}^{n} W(j,k)}.$$

The columns of P are probability distributions, and hence for all j we have

$$\sum_{i=1}^{n} P(i,j) = 1. \tag{5.0.1}$$

Exercise 5.0.1. Show that $P = W^T D^{-1}$, where D is the diagonal matrix with diagonal entries $D(i,i) = \sum_{j=1}^{n} W(i,j)$. △

To model the random teleportation of the surfer, we flip a biased coin at each step, and with probability $\alpha \in [0,1)$ the surfer takes a random walk step, and with probability $1-\alpha$ the surfer teleports somewhere else at random in the internet. The teleportation step is itself drawn at random, according to a teleportation probability distribution v, which satisfies $v(i) \geq 0$ and $\sum_{i=1}^{n} v(i) = 1$. The teleportation distribution can be uniform $v(i) = 1/n$, in which case the random surfer jumps uniformly at random to another site on the internet. Another typical choice is a localized PageRank, obtained by setting $v(i) = \delta_{ij}$, where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise. In this case, the random surfer always jumps back to the same website j each time it teleports, which leads to

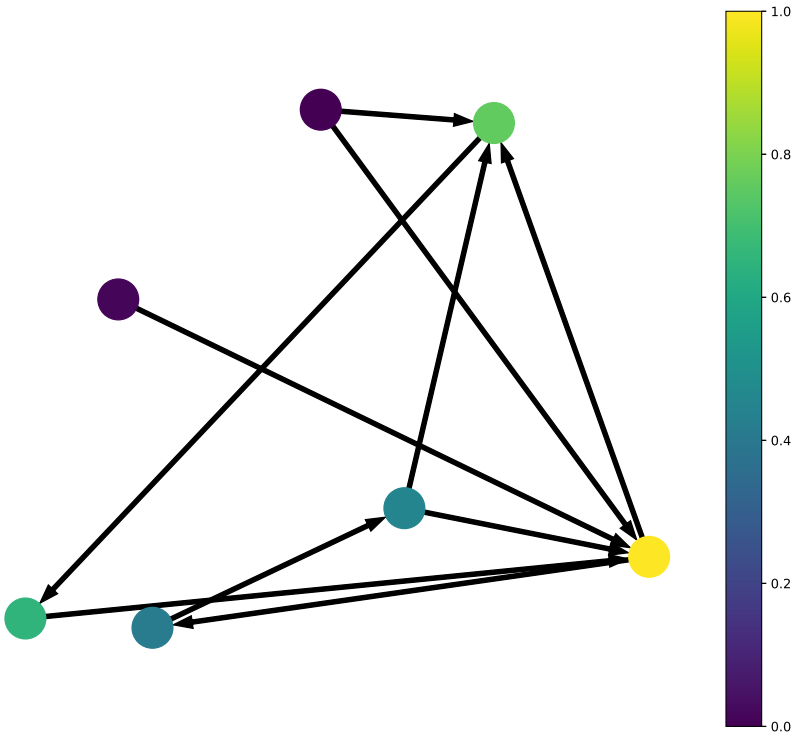


Figure 5.0.1: An illustration of the PageRank vector on a toy graph with $\alpha = 2/3$. The colors from blue to yellow indicate the value of the PageRank vector at each node, and the black arrows indicate directed links between webpages.

a ranking of all pages relative to j. This can be useful in retrieval problems. There are of course a continuum of intermediate teleportation distributions between the uniform random distribution and localized PageRank.

Let $x_0(i)$ denote the probability that the random surfer is initially at site i at step $k = 0$ of the walk. We can take x_0 to be any probability vector (i.e., $x_0(i) \geq 0$ and $\sum_i x_0(i) = 1$). For instance, setting $x_0(i) = \delta_{ij}$ amounts to initially placing the random surfer at page j. Any other distribution x_0 amounts to placing the surfer at node j with probability $x_0(j)$ at the initial time. For $k \geq 1$ define

$$x_k(i) = \text{Probability that the random surfer is at page } i \text{ on step } k.$$

To see how x_k transitions to x_{k+1} requires some probability. We condition on the location of the surfer at step k, and on the outcome of the coin flip, to

obtain

$$x_{k+1}(i) = (1-\alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)x_k(j).$$

We can write this in matrix/vector form as

$$x_{k+1} = (1-\alpha)v + \alpha P x_k. \tag{5.0.2}$$

The PageRank vector is defined as the limit of x_k as $k \to \infty$—we will prove the limit exists below. Figure 5.0.1 shows an illustration of the PageRank vector (small values are blue and large are yellow) on a toy graph with $\alpha = 2/3$. The black arrows indicate links between websites. Notice that the two lowest ranked sites (the dark blue ones) have no incoming edges, so they are not visited except for on teleportation steps of the random surfer, which is one of the reasons for their low ranking. On the other hand, the highest ranked yellow node has many incoming edges and is visited very often by the random surfer.

5.1 Convergence of the random surfer

If we send $k \to \infty$ in the PageRank iteration (5.0.2), and if x_k converges to some vector x, then clearly x should be a solution of the linear equation

$$x = (1-\alpha)v + \alpha P x. \tag{5.1.1}$$

We call (5.1.1) the *PageRank* problem, and its solution x is the *PageRank* vector. In this section, we will prove that the probability distribution x_k of the random surfer converges as $k \to \infty$ to the solution of the *PageRank* problem (5.1.1).

It will be more convenient to work in the ℓ_1-norm $\|\cdot\|_1$ defined by

$$\|x\|_1 = \sum_{i=1}^{n} |x(i)|.$$

In the ℓ_1-norm, the transition matrix P is non-expansive, according to the following proposition.

Proposition 5.1.1. *We have* $\|Px\|_1 \le \|x\|_1$.

Proof. The proof is a straightforward computation

$$\|Px\|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} P(i,j)x(j) \right| \le \sum_{i=1}^{n} \sum_{j=1}^{n} P(i,j)|x(j)| = \sum_{j=1}^{n} |x(j)| = \|x\|_1,$$

where we used (5.0.1) in the third step. This completes the proof. □

We first establish that the PageRank problem (5.0.2) has a unique solution.

Lemma 5.1.2. *Let $v \in \mathbb{R}^n$ and $0 \leq \alpha < 1$. Then there is a unique vector $x \in \mathbb{R}^n$ solving the PageRank problem (5.1.1). Furthermore, the following hold.*

(i) *We have $\sum_{i=1}^n x(i) = \sum_{i=1}^n v(i)$.*

(ii) *If $v(i) \geq 0$ for all i, then $x(i) \geq 0$ for all i.*

Remark 5.1.3. Taking (i) and (ii) together shows that whenever v is a probability distribution (i.e., $v(i) \geq 0$ and $\sum_i v(i) = 1$), the same is true of the unique solution x of the PageRank problem (5.1.1).

Proof of Lemma 5.1.2. Note that we can re-write (5.1.1) as $Ax = v$ where

$$A = (1-\alpha)^{-1}(I - \alpha P).$$

To prove there is a unique solution x, we need only show that the kernel of A is trivial. Then the mapping $A : \mathbb{R}^n \to \mathbb{R}^n$ is injective (one-to-one), and hence it is also surjective (onto), by the rank-nullity theorem.

To show that $\ker(A) = \{0\}$, let $z \in \ker(A)$. Then $Az = 0$ and so $z = \alpha P z$. By Proposition 5.1.1 we have

$$\|z\|_1 = \|\alpha P z\|_1 = \alpha \|Pz\|_1 \leq \alpha \|z\|_1.$$

Since $\alpha < 1$ we must have $\|z\|_1 = 0$, and so $z = 0$, which establishes that A is injective, and so $Ax = v$ has a unique solution x, for each v.

We now prove (i). We use (5.0.1) to obtain

$$\begin{aligned}
\sum_{i=1}^n x(i) &= \sum_{i=1}^n \left((1-\alpha)v(i) + \alpha \sum_{j=1}^n P(i,j)x(j) \right) \\
&= (1-\alpha) \sum_{i=1}^n v(i) + \alpha \sum_{j=1}^n x(j) \sum_{i=1}^n P(i,j) \\
&= (1-\alpha) \sum_{i=1}^n v(i) + \alpha \sum_{j=1}^n x(j).
\end{aligned}$$

Re-arranging we have

$$(1-\alpha) \sum_{i=1}^n x(i) = (1-\alpha) \sum_{i=1}^n v(i).$$

Since $\alpha < 1$, we can divide by $1-\alpha$ to complete the proof of (i).

We finally prove (ii). Assume $v(i) \geq 0$ for all i. We first compute, using (5.1.1), that

$$\begin{aligned} |x(i)| &= \left|(1-\alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)x(j)\right| \\ &\leq (1-\alpha)|v(i)| + \alpha \left|\sum_{j=1}^{n} P(i,j)x(j)\right| \\ &\leq (1-\alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)|x(j)|. \end{aligned}$$

Summing both sides over $i = 1, \ldots, n$ and using (5.0.1) we find that

$$\begin{aligned} \sum_{i=1}^{n} |x(i)| &\leq (1-\alpha)\sum_{i=1}^{n} v(i) + \alpha \sum_{i=1}^{n}\sum_{j=1}^{n} P(i,j)|x(j)| \\ &= (1-\alpha)\sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} |x(j)|. \end{aligned}$$

Re-arranging and using part (i) we have

$$(1-\alpha)\sum_{i=1}^{n} |x(i)| \leq (1-\alpha)\sum_{i=1}^{n} v(i) = (1-\alpha)\sum_{i=1}^{n} x(i).$$

Since $\alpha < 1$ we can divide by $1-\alpha$ to deduce that

$$\sum_{i=1}^{n} |x(i)| \leq \sum_{i=1}^{n} x(i).$$

It follows that $x(i) = |x(i)| \geq 0$ for all i, which completes the proof. □

Remark 5.1.4. It is common to re-write the PageRank problem (5.1.1) as an eigenvector problem. Let us assume we are in the setting where v and x are probability distributions, so $x(i) \geq 0$ and $\sum_{i=1}^{n} x(i) = 1$. Noting that $\mathbf{1}^T x = \sum_{i=1}^{n} x(i) = 1$ we can re-write (5.1.1) as

$$x = (1-\alpha)v\mathbf{1}^T x + \alpha P x = \left((1-\alpha)v\mathbf{1}^T + \alpha P\right) x.$$

The matrix

$$P_\alpha := (1-\alpha)v\mathbf{1}^T + \alpha P \tag{5.1.2}$$

is exactly the probability transition matrix for the random surfer, and the PageRank problem $P_\alpha x = x$ is an eigenvector problem, which merely states that the PageRank vector is the invariant (or stationary) distribution of the induced Markov chain. In this light, Lemma 5.1.2 (ii) is a direct consequence of the Perron-Frobenius Theorem.

We can now prove convergence, with a linear rate, of the random surfer's distribution x_k to the PageRank vector.

Theorem 5.1.5. *Let $v \in \mathbb{R}^n$ and $0 \leq \alpha < 1$. Let x_k satisfy the PageRank iteration (5.0.2), and let x be the unique solution of the PageRank problem (5.1.1) (i.e., the PageRank vector). Then we have*

$$\|x_k - x\|_1 \leq \alpha^k \|x_0 - x\|_1. \tag{5.1.3}$$

Proof. We simply subtract

$$x = (1-\alpha)v + \alpha P x$$

from

$$x_k = (1-\alpha)v + \alpha P x_{k-1}$$

to obtain

$$\|x_k - x\|_1 = \|\alpha P x_{k-1} - \alpha P x\| = \alpha \|P(x_{k-1} - x)\|_1.$$

Using Proposition 5.1.1 we obtain

$$\|x_k - x\|_1 \leq \alpha \|x_{k-1} - x\|_1.$$

The proof is completed by induction. □

Remark 5.1.6. Since $0 \leq \alpha < 1$, Theorem 5.1.5 shows that the PageRank iterations x_k converge to the PageRank vector x as $k \to \infty$ at the linear rate of α in the ℓ_1-norm. In particular, the convergence can be very slow if α is close to one. This shows another advantage to introducing the teleportation step; not only does it guarantee convergence of the PageRank iterations, but it gives the user control over the convergence rate.

In practice, the standard way to compute the solution of the PageRank problem (5.1.1) is via the PageRank iteration (5.0.2), since it is simple to compute (especially with large sparse matrices), and can be terminated early to obtain an approximate solution. The PageRank iteration (5.0.2) can be interpreted as the power iteration for computing the largest eigenvector of a matrix. Indeed, following Remark 5.1.4, we can rewrite the PageRank iteration as $x_{k+1} = P_\alpha x_k$, where P_α is defined in (5.1.2). This is exactly the power iteration, except the normalization step appears to be omitted (normally it would be $x_{k+1} = P_\alpha x_k / \|P_\alpha x\|$). The normalization is not needed since P is non-expansive (see Proposition 5.1.1).

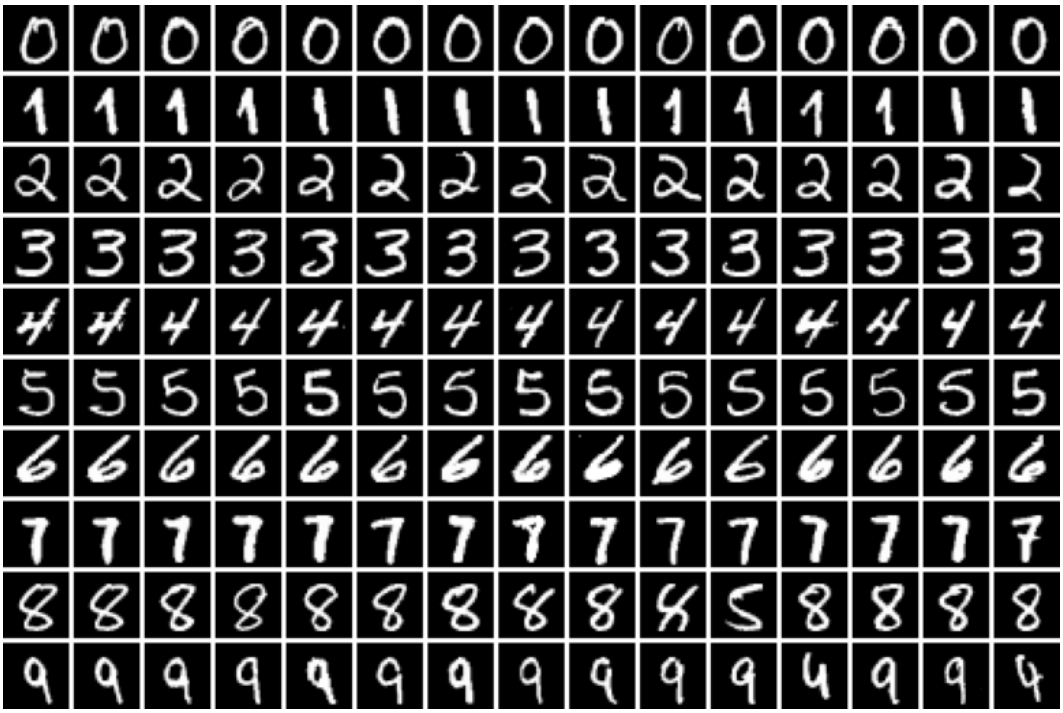


Figure 5.2.1: An example of using personalized PageRank for image retrieval. In each row the image on the left is the query image, and the following 14 images are the top retrieved images using personalized PageRank.

5.2 Personalized PageRank for image retrieval

Python Notebook: .ipynb

We give here an example of using personalized, or localized, PageRank for image retrieval. Image retrieval takes a query image and tries to find similar images in a dataset. We consider the MNIST dataset of handwritten digits, and construct a k-nearest neighbor graph over the dataset as described in Section 4.2.2. To retrieve images similar to image j, we set the teleportation distribution to be $v(i) = \delta_{ij}$, in order to rank all images based on their similarity to image j. We then solve the PageRank problem (5.1.1) with the iteration (5.0.2), and return the top ranked images. We took one image from each of the 10 MNIST digits and ran personalized PageRank to retrieve the top 14 similar images. Figure 5.2.1 shows the results. The image on the left is the query image, and the following 14 images are the retrieved images. We note

that most digits are from the same class, and are in fact written in a very similar way to the query digit. There are a handful retrieved digits that are from different classes than the query; for example, a 5 in the 8 row and some 4's in in the 9 row. There is also what appears to be an 8 in the 7 row, but this is hard to distinguish by eye.

Chapter 6

The Discrete Fourier Transform

We saw in Section 3.5 how to use PCA to find a good basis to represent image blocks in order to perform image compression. The basis is one for which only a few basis functions are required to reconstruct with good accuracy most of the blocks in the image. We experimented with a similar PCA-based method for audio compression in Project 3.5.1. In this case, the basis vectors learned by PCA are good at representing very short segments of audio files, such as music or speech. PCA finds the best such basis; the one that captures the most variation in the data with the fewest basis vectors.

It is instructive to take a closer look at these basis functions. For image compression we show the first 30 principal component vectors in Figure 3.5.4. These start off as low frequency images that are roughly constant over the block, and gain additional higher frequency components as we look at higher principal components. This means that most blocks in the image are well-approximated by these slowly varying roughly constant image blocks. Similarly, for audio compression we show the first 4 principal components on length 64 blocks of an audio file in Figure 6.0.1. Here, some of the basis functions strongly resemble low frequency pure tones, i.e., sin and cos functions.

While PCA finds the best basis, and is adapted to the data, it is costly to compute the principal components, since this requires solving large eigenvector problems. This is impossible to do in embedded environments (e.g., a digital camera, smartphone, video surveillance, digital TVs, etc.), which have limited processing power and have to process images and video in real-time. In some settings, individual image frames in video must be processed in real-time, and only a few lines of each frame can be stored in memory. In such cases, a learned basis, such as that obtained via PCA, is intractable, and instead a hand-crafted basis with similar properties is desired.

The Fourier Transform is a hand-crafted change of basis that is extremely

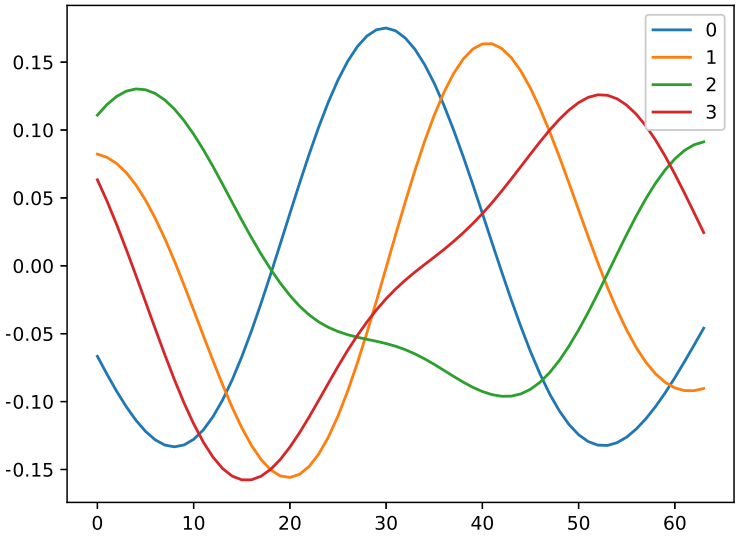


Figure 6.0.1: The first 4 principal components computed during PCA-based audio compression, following Project 3.5.1. Two of the basis functions strongly resemble the trigonometric functions sin and cos.

useful for analyzing audio, images, and video. The Fourier Transform expresses a signal (i.e., a function) in terms of a basis consisting of pure tones, that is, cos and sin waves with different shifts and of different frequencies. We will see that the Fourier basis shares many similarities with a PCA-basis, namely that images and audio signals are well-approximated locally in space/time by only a few Fourier basis functions. This is called *sparsity*, and audio and image signals are locally sparse in the Fourier domain. This sparsity is what allows lossy compression algorithms, like jpeg for images and mpeg for video, to work so well. These algorithms use Discrete Fourier Transforms, instead of a PCA-based compression, and are selective about which Fourier basis function to encode, and which to discard at each instance in time or location in the image.

We will study the Discrete Fourier Transform in these notes, which applies to discretized signals (e.g., vectors), but the main ideas are similar across the different variants of the Fourier Transform. The advantage of studying the Discrete Fourier Transform is that we are studying exactly the algorithm used in practice, and we do not need to use advanced mathematical tools like measure theory and L^p spaces, which are needed to study the continuous Fourier transform.

We briefly mention that there are many other advantages to a hand-crafted

change of basis, like the Fourier Transform, beyond the computational advantages of not needing access to an eigensolver. The change of basis itself can be computed far more efficiently with a Fast Fourier Transform (FFT) in $O(n \log n)$ operations, compared to the matrix multiplication required for a general change of coordinates, which requires $O(n^2)$ operations. Furthermore, with PCA we did not know before hand what our basis functions would look like and could only do a limited analysis of their properties. On the other hand, for a hand-crafted transform, we know the basis functions explicitly and can perform an in-depth mathematical analysis of their properties. The Discrete Fourier Transform has many very nice mathematical properties that are worth studying on their own, independent of the applications to image or audio analysis. Furthermore, we will find that some properties of the Discrete Fourier Transform, such as its compatibility with convolutions, make it a very useful tool for solving difference equations, which are discretizations of partial differential equations. In fact, one can argue that the most important applications of the Fourier Transform are no longer in image processing or computer vision, since deep convolutional neural networks outperform Fourier (and Wavelet) methods by orders of magnitude. However, in scientific computing problems, like solving linear partial differential equations, the Fourier Transform is still one of the most useful tools, and will not be replaced anytime soon. As such, these notes will emphasize a variety of applications of Fourier and Wavelet methods, including to solving partial differential equations.

It is important to point out that we will make use of complex numbers in this section. The letter i will always denote the imaginary number $i = \sqrt{-1}$, and will never be used for an index of summation, unless it is a typo. This convention is not followed in any other sections of these notes.

6.1 Complex numbers and Euler's formula

We recall that a complex number has the form $z = a + ib$ where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. The set of all complex numbers is denoted $\mathbb{C}$. For a complex number $z = a + ib$, the *complex conjugate*, denoted $\overline{z}$, is given by

$$\overline{z} = a - ib.$$

For two complex numbers z_1, z_2 we have $\overline{z_1 z_2} = \overline{z_1}\,\overline{z_2}$. The *modulus* of z, denoted $|z|$, is given by

$$|z| = \sqrt{a^2 + b^2} = \sqrt{z\overline{z}}.$$

The complex exponential of $z \in \mathbb{C}$ is defined by the Taylor series expansion

$$e^z = \sum_{k=0}^{\infty} \frac{z^k}{k!}.$$

The Taylor series is absolutely convergent in the whole complex plane. A very important identity involving the complex exponential is Euler's identity

$$e^{it} = \cos t + i \sin t \tag{6.1.1}$$

for all real numbers $t \in \mathbb{R}$. In particular, the complex exponential is a 2π-periodic function, so $e^{i(t+2\pi k)} = e^{it}$ for any integer k. We also note that

$$\overline{e^{it}} = \cos t - i \sin t = \cos(-t) + i \sin(-t) = e^{-it}.$$

Exercise 6.1.1. Prove Euler's identity in three different ways.

1. Substitute Taylor expansions for e^{it}, $\cos t$, and $\sin t$ in (6.1.1), and show that both sides are equal.

2. Write the complex number e^{it} in polar coordinates

   $$e^{it} = r(\cos\theta + i \sin\theta),$$

   where $r = r(t)$ and $\theta = \theta(t)$ are real numbers. Differentiate both sides in t, and equate real and imaginary parts to show that $r'(t) = 0$ and $\theta'(t) = 1$. Use the initial values $r(0) = 1$ and $\theta(0) = 0$ to show that $r(t) = 1$ and $\theta(t) = t$.

3. Write $f(t) = \cos t + i \sin t$ for $t \in \mathbb{R}$. Show that $f'(t) = if(t)$. Use this to show that

   $$\frac{d}{dt}(f(t)e^{-it}) = 0.$$

   Therefore $f(t) = ze^{it}$ for a complex number z. Use that $f(0) = 1$ to show that $z = 1$.

△

Exercise 6.1.2. Use Euler's formula (6.1.1) to prove the following trigonometric identities.

(i) $\cos(s+t) = \cos(s)\cos(t) - \sin(s)\sin(t)$

(ii) $\sin(s+t) = \sin(s)\cos(t) + \cos(s)\sin(t)$

[Hint: Apply Euler's formula to both sides of $e^{i(s+t)} = e^{is}e^{it}$.] △

6.2 The Forward and Inverse Transforms

Python Notebook: .ipynb

Let $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. It will be useful to think of complex vectors of length n as functions $f : \mathbb{Z}_n \to \mathbb{C}$. Since the trigonometric functions cos and sin are periodic, it will be convenient to work with periodic signals, so we are really thinking of $\mathbb{Z}_n$ as the cyclic group of integers modulo n (written $\mathbb{Z}_n = \mathbb{Z}/n$). When integers $p, q \in \mathbb{Z}_n$ are added, subtracted, or multiplied, the result is interpreted modulo n. For example, in $\mathbb{Z}_4$, $2 + 2 = 4 = 0 \bmod 4$. This has the effect of extending a signal $f : \mathbb{Z}_n \to \mathbb{C}$ to a periodic function $f : \mathbb{Z} \to \mathbb{C}$ of period n, so that $f(k + mn) = f(k)$ for all integers k and m.

Let $L^2(\mathbb{Z}_n)$ denote the vector space of functions $f : \mathbb{Z}_n \to \mathbb{C}$. We define the inner product on $L^2(\mathbb{Z}_n)$ by

$$\langle f, g \rangle = \sum_{k=0}^{n-1} f(k)\overline{g(k)}.$$

We note that $L^2(\mathbb{Z}_n)$ is nothing other than $\mathbb{C}^n$—it is simply more convenient to take the viewpoint of functions instead of vectors. It is important to note that the inner product is not symmetric, and in fact

$$\langle f, g \rangle = \overline{\langle g, f \rangle}.$$

The norm of $f \in L^2(\mathbb{Z}_n)$ is defined by $\|f\| = \sqrt{\langle f, f \rangle}$.

The Discrete Fourier Transform (DFT) is an orthogonal change of basis in $L^2(\mathbb{Z}_n)$ that expresses a function $f :\in L^2(\mathbb{Z}_n) \to \mathbb{C}$ in terms sinusoidal basis functions of different frequencies. Due to Euler's formula (6.1.1), it is far more convenient to work with complex exponential functions of the form

$$k \mapsto e^{2\pi i \sigma k} = \cos(2\pi\sigma k) + i \sin(2\pi\sigma k). \tag{6.2.1}$$

The real and imaginary parts of (6.2.1) are exactly the sinusoidal basis functions we are interested in, and $\sigma \geq 0$ is the corresponding frequency (i.e., the number of cycles per unit). It is possible to avoid complex analysis and work with the sinusoidal functions directly (indeed, there are transforms called the cosine and sine transforms). However, encoding the sinusoids into the complex exponential via Euler's formula greatly simplifies the analysis and leads to an elegant mathematical theory.

Now, we do not need all of the frequencies $\sigma \geq 0$ for a change of basis. In face, since $L^2(\mathbb{Z}_n)$ is n-dimensional (it is just $\mathbb{C}^n$), we should just need n

basis vectors. To see which frequencies to choose, note that any $f \in L^2(\mathbb{Z}_n)$ can be extended to a periodic function on $\mathbb{Z}$ with period n. Thus, we should only need to choose frequencies σ resulting in n-periodic complex exponentials. This means the smallest period $T = 1/\sigma$ of the sinusoid should divide evenly into n, in other words, $\sigma = \ell/n$ for some integer $\ell = 0, 1, \ldots, n-1$. Note that we do not take $\ell = n$ since then $\sigma = 1$ and $e^{2\pi i \sigma k} = 1 = e^0$ for all k, which is the same as taking $\ell = 0$. In fact, due to the periodicity of the complex exponential, taking any other integer values for ℓ is redundant.

Thus, it is natural to consider the n complex exponential functions

$$u_\ell(k) := e^{2\pi i k \ell/n}, \quad \ell = 0, 1, \ldots, n-1. \tag{6.2.2}$$

It is often useful to note that we can set $\omega = e^{2\pi i/n}$ and write $u_\ell(k) = \omega^{k\ell}$. The complex number ω is an n^{th} root of unity, meaning that $\omega^n = e^{2\pi i} = 1$. We also have $\overline{\omega} = e^{-2\pi i/n} = \omega^{-1}$.

It is important to point out that the frequency of the sampled function u_ℓ does not always match the frequency of the continuous function it is sampled from, due to an effect known as aliasing. Indeed, since ω is an n^{th} root of unity we have

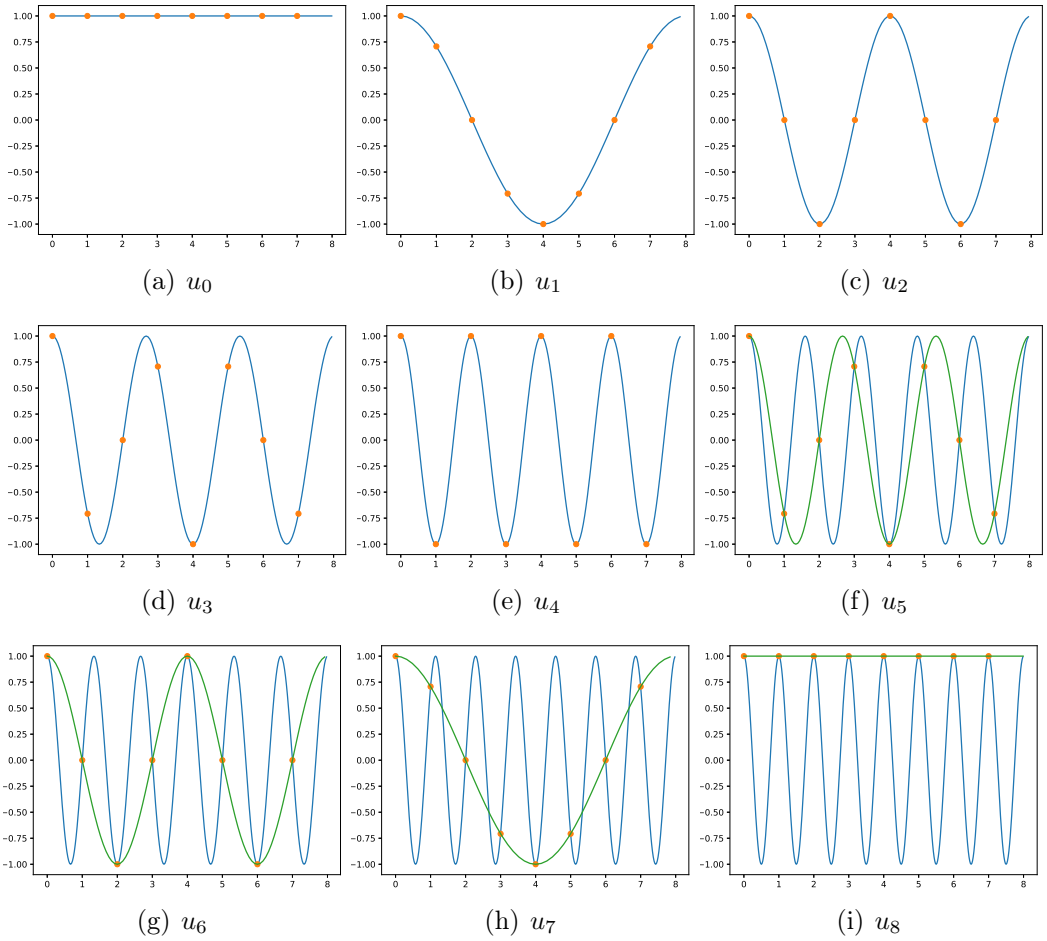
$$u_{n-\ell}(k) = \omega^{(n-\ell)k} = \omega^{nk}\omega^{-\ell k} = \overline{u_\ell(k)}. \tag{6.2.3}$$

So at the discrete level, the functions $u_{n-\ell}$ and u_ℓ have the same frequency, and are in fact just complex conjugates of each other. Hence, the exponentials u_ℓ with the highest frequencies correspond to $\ell \approx n/2$. The frequencies increase with ℓ for $0 \le \ell \le n/2$, and decrease with ℓ for $n/2 < \ell \le n-1$. We give an illustration of this for $n = 8$ in Figure 6.2.1.

It turns out that the complex exponential functions u_ℓ are mutually orthogonal.

Lemma 6.2.1. *The functions $u_0, u_1, \ldots, u_{n-1}$ are orthogonal. In particular*

$$\langle u_\ell, u_m \rangle = \begin{cases} n, & \text{if } \ell = m \\ 0, & \text{otherwise.} \end{cases} \tag{6.2.4}$$



(a) u_0 (b) u_1 (c) u_2

(d) u_3 (e) u_4 (f) u_5

(g) u_6 (h) u_7 (i) u_8

Figure 6.2.1: Real parts of the discrete Fourier modes u_ℓ for $n = 8$. The orange dots show the real parts of $u_\ell(k)$ for $k = 0, 1, \ldots, 7$, which are sampled from the blue curves. For u_5, u_6, u_7, u_8, the sampled frequency is aliased to the lower frequency modes u_3, u_2, u_1, u_0, respectively, which is equivalent to sampling the green curve. The discrete Fourier mode with the highest frequency is u_4.

Proof. We compute

$$\begin{aligned}\langle u_\ell, u_m \rangle &= \sum_{k=0}^{n-1} u_\ell(k)\overline{u_m(k)} \\ &= \sum_{k=0}^{n-1} \omega^{k\ell}\overline{\omega^{km}} \\ &= \sum_{k=0}^{n-1} \omega^{k\ell}\omega^{-km} = \sum_{k=0}^{n-1} \left(\omega^{\ell-m}\right)^k .\end{aligned}$$

If $\ell = m$ then we have $\omega^{\ell-m} = \omega^0 = 1$ and $\langle u_\ell, u_m \rangle = n$. If $\ell \neq m$ then we note that the expression above is a (truncated) geometric series in the variable $r := \omega^{\ell-k}$, and we thus have

$$\langle u_\ell, u_m \rangle = \frac{r^n - 1}{r - 1}.$$

The proof is completed by noting that $r^n = (\omega^n)^{\ell-k} = 1$, since ω is an n^{th} root of unity. □

Since $u_0, u_1, \ldots, u_{n-1}$ are orthogonal, they are linearly independent and span $L^2(\mathbb{Z}_n)$. The DFT is simply the change of basis that writes a function $f \in L^2(\mathbb{Z}_n)$ in this new basis, that is, in the form

$$f = \frac{1}{n} \sum_{\ell=0}^{n-1} c_\ell u_\ell.$$

Note that the $\frac{1}{n}$ factor is simply a convention, and could have been omitted, in which case it would appear in the formula for c_ℓ below. To determine the coefficients c_ℓ, we take the inner product of both sides above with u_m and use the orthogonality shown in Lemma 6.2.1 to obtain

$$\langle f, u_m \rangle = \frac{1}{n} \sum_{\ell=0}^{n-1} c_\ell \langle u_\ell, u_m \rangle = c_m.$$

Therefore we have $c_m = \langle f, u_m \rangle$, and we can write f in the form

$$f = \frac{1}{n} \sum_{\ell=0}^{n-1} \langle f, u_\ell \rangle u_\ell.$$

The coefficients c_ℓ can be written out as

$$c_\ell = \langle f, u_\ell \rangle = \sum_{k=0}^{n-1} f(k)\overline{u_\ell(k)} = \sum_{k=0}^{n-1} f(k)\omega^{-k\ell} = \sum_{k=0}^{n-1} f(k)e^{-2\pi i k\ell/n}.$$

This leads us to the definition of the DFT, which simply computes the coefficients c_ℓ.

Definition 6.2.2. The *Discrete Fourier Transform (DFT)* is the mapping $\mathcal{D} : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ defined by

$$\mathcal{D}f(\ell) = \sum_{k=0}^{n-1} f(k)\omega^{-k\ell} = \sum_{k=0}^{n-1} f(k)e^{-2\pi i k\ell/n},$$

where $\omega = e^{2\pi i/n}$.

It is important to point out that the DFT has symmetries when applied to real-valued signals.

Proposition 6.2.3. *If $f \in L^2(\mathbb{Z}_n)$ is real-valued (i.e., $f(k) \in \mathbb{R}$ for all k), then*

$$\mathcal{D}f(\ell) = \overline{\mathcal{D}f(n-\ell)}.$$

Proof. The proof follows from (6.2.3). Indeed, since f is real-valued, we use (6.2.3) to compute

$$\mathcal{D}f(\ell) = \langle f, u_\ell \rangle = \langle f, \overline{u_{n-\ell}} \rangle = \overline{\langle f, u_{n-\ell} \rangle} = \overline{\mathcal{D}f(n-\ell)}. \qquad \square$$

Proposition 6.2.3 shows that we only need to record the first half of the DFT coefficients, $\mathcal{D}f(\ell)$ for $0 \le \ell \le \frac{n}{2}$, when transforming a real-valued signal.

We immediately have the following inversion theorem.

Theorem 6.2.4 (Fourier Inversion Theorem)**.** *For any $f \in L^2(\mathbb{Z}_n)$ we have*

$$f(k) = \frac{1}{n} \sum_{\ell=0}^{n-1} \mathcal{D}f(\ell) \omega^{k\ell} = \frac{1}{n} \sum_{\ell=0}^{n-1} \mathcal{D}f(\ell) e^{2\pi i k\ell/n}. \tag{6.2.5}$$

Proof. Let $g(k)$ denote the right hand side of (6.2.5). We need to show that $g(k) = f(k)$. We compute

$$\begin{aligned} g(k) &= \frac{1}{n} \sum_{\ell=0}^{n-1} \mathcal{D}f(\ell) \omega^{k\ell} \\ &= \frac{1}{n} \sum_{\ell=0}^{n-1} \sum_{j=0}^{n-1} f(j) \omega^{-j\ell} \omega^{k\ell} \\ &= \frac{1}{n} \sum_{j=0}^{n-1} f(j) \sum_{\ell=0}^{n-1} \overline{u_j(\ell)} u_k(\ell) \\ &= \frac{1}{n} \sum_{j=0}^{n-1} f(j) \langle u_k, u_j \rangle = f(k), \end{aligned}$$

due to Lemma 6.2.1. $\square$

Based on Theorem 6.2.4, we make the following definition.

Definition 6.2.5 (Inverse Discrete Fourier Transform)**.** The *Inverse Discrete Fourier Transform (IDFT)* is the mapping $\mathcal{D}^{-1} : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ defined by

$$\mathcal{D}^{-1} f(\ell) = \frac{1}{n} \sum_{k=0}^{n-1} f(k) \omega^{k\ell} = \frac{1}{n} \sum_{k=0}^{n-1} f(k) e^{2\pi i k\ell/n}.$$

Remark 6.2.6. By Theorem 6.2.4, we have $\mathcal{D}^{-1}\mathcal{D}f = f$ for all $f \in L^2(\mathbb{Z}_n)$. The other direction, $\mathcal{D}\mathcal{D}^{-1}f = f$, follows from standard linear algebra facts, but it is also very easy to show using the argument in Theorem 6.2.4. Indeed, we have

$$\mathcal{D}(\mathcal{D}^{-1}f)(k) = \sum_{\ell=0}^{n-1} \mathcal{D}^{-1}f(\ell)\omega^{k\ell} = \frac{1}{n}\sum_{\ell=0}^{n-1}\sum_{j=0}^{n-1} f(j)\omega^{j\ell}\omega^{-k\ell} = f(k),$$

where the final equality is proved in a similar way as in Theorem 6.2.4.

Remark 6.2.7. Define the $n\times n$ complex-valued matrix with entries $W(k,\ell) = \omega^{k\ell}$, that is

$$W = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \tag{6.2.6}$$

Then the DFT can be expressed via matrix multiplication as $\mathcal{D}f = \overline{W}f$. The inverse DFT can be expressed as $\mathcal{D}^{-1}f = \frac{1}{n}Wf$. In both cases we treat f as a vector $f \in \mathbb{C}^n$. Theorem 6.2.4 (Fourier Inversion) can be restated as saying that $W\overline{W} = nI$.

Exercise 6.2.8. Show that the DFT enjoys the following basic shift properties.

1. Recall that $u_\ell(k) := e^{2\pi ik\ell/n} = \omega^{k\ell}$. Show that

$$\mathcal{D}(f\cdot u_\ell)(k) = \mathcal{D}f(k-\ell).$$

2. Let $T_\ell : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ be the translation operator $T_\ell f(k) = f(k+\ell)$. Show that

$$\mathcal{D}(T_\ell f)(k) = e^{-2\pi ik\ell/n}\mathcal{D}f(k).$$

[Hint: You can equivalently show that $\mathcal{D}^{-1}\left(f\cdot u_\ell\right)(k) = \mathcal{D}^{-1}f(k-\ell)$, using an argument similar to part 1.]

△

6.3 The Fast Fourier Transform (FFT)

Python Notebook: .ipynb

For $f \in L^2(\mathbb{Z}_n)$, computing $\mathcal{D}f$ or $\mathcal{D}^{-1}f$ requires on the order of n^2 operations (multiplications and additions) if we use perform the computation exactly as in Definition 6.2.2. Indeed, the sum defining $\mathcal{D}f(\ell)$ has n terms, and we have to compute n coefficients, for $\ell = 0, 1, \ldots, n-1$. From the viewpoint of the DFT as matrix multiplication discussed in Remark 6.2.7, we are multiplying an $n \times n$ matrix by a vector, which requires $O(n^2)$ operations. If we always computed the DFT in this way, it would be too slow for most applications. Indeed, taking $n = 1000$ samples in an audio file with a sample rate of 44.1 kHz is only 22 ms of audio, but would take millions ($n^2 = 10^6 = 1$ million) of operations to compute the DFT of very short audio sample. One whole second of audio has $n = 44,100$ samples, and it would take on the order of $n^2 \approx 2 \times 10^9$ operations to compute the DFT. This quickly becomes computationally intractable.

It turns out that many of the $O(n^2)$ operations performed to compute the DFT as per Definition 6.2.2 are redundant, and if the computation is done carefully it can be reduced to $O(n \log n)$ operations. The resulting algorithm is called the Fast Fourier Transform (FFT). The computational efficiency of the FFT is perhaps the dominant factor in the widespread applicability of the Fourier Transform in problems like image and audio compression, and solving partial differential equations, among many others. The FFT has been ranked as one of the top 10 algorithms of 20th Century by IEEE Computing in Science & Engineering magazine, and according to Gilbert Strang (MIT), is "the most important numerical algorithm of our lifetime."

The FFT is based on subsampling the signal into its even and odd samples, applying the DFT to these two components separately. The algorithm then proceeds recursively by applying the even/odd subsampling to each component from the first step. The whole idea is based on the Lemma 6.3.1 below, which allows us to efficiently compute the DFT of a signal from the DFT of its even and odd samples.

Before presenting the lemma, we introduce some additional notation. For $f \in L^2(\mathbb{Z}_n)$ with n even, the even and odd parts of f, denoted f_e and f_o, respectively, are the functions in $L^2(\mathbb{Z}_{\frac{n}{2}})$ defined by

$$f_e(k) = f(2k) \quad \text{and} \quad f_o(k) = f(2k+1),$$

for $k = 0, 1, \ldots, \frac{n}{2} - 1$. In this section we will work with the DFT for signals of different lengths so we will denote by $\mathcal{D}_n$ and $\mathcal{D}_n^{-1}$ the forward and inverse

DFT, respectively, on $L^2(\mathbb{Z}_n)$. It is important to note that $\mathcal{D}_1$ is the identity, since $\mathcal{D}_1 f(0) = f(0)$.

Lemma 6.3.1. *For each $f \in L^2(\mathbb{Z}_n)$ with n even we have*

$$\mathcal{D}_n f(\ell) = \mathcal{D}_{\frac{n}{2}} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}} f_o(\ell), \tag{6.3.1}$$

for $\ell = 0, 1, \ldots, n-1$.

Proof. The proof is a straightforward computation based on splitting the sum defining $\mathcal{D}f(\ell)$ into even and odd parts. We have

$$\begin{aligned}
\mathcal{D}_n f(\ell) &= \sum_{k=0}^{n-1} f(k) e^{-2\pi i k \ell/n} \\
&= \sum_{k=0}^{\frac{n}{2}-1} f(2k) e^{-2\pi i 2k\ell/n} + \sum_{k=0}^{\frac{n}{2}-1} f(2k+1) e^{-2\pi i (2k+1)\ell/n} \\
&= \sum_{k=0}^{\frac{n}{2}-1} f_e(k) e^{-2\pi i 2k\ell/n} + e^{-2\pi i \ell/n} \sum_{k=0}^{\frac{n}{2}-1} f_o(k) e^{-2\pi i 2k\ell/n} \\
&= \sum_{k=0}^{\frac{n}{2}-1} f_e(k) e^{-2\pi i k\ell/(n/2)} + e^{-2\pi i \ell/n} \sum_{k=0}^{\frac{n}{2}-1} f_o(k) e^{-2\pi i k\ell/(n/2)} \\
&= \mathcal{D}_{\frac{n}{2}} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}} f_o(\ell),
\end{aligned}$$

which completes the proof. □

Remark 6.3.2. In (6.3.1), it is important to point out that $\mathcal{D}_n f \in L^2(\mathbb{Z}_n)$ and $\mathcal{D}_{\frac{n}{2}} f_e, \mathcal{D}_{\frac{n}{2}} f_o \in L^2(\mathbb{Z}_{\frac{n}{2}})$. For $\frac{n}{2} \leq \ell \leq n-1$, the periodicity of $\mathbb{Z}_{\frac{n}{2}}$ gives that

$$\mathcal{D}_n f(\ell) = \mathcal{D}_{\frac{n}{2}} f_e(\ell - \tfrac{n}{2}) + e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}} f_o(\ell - \tfrac{n}{2}).$$

This is important to keep in mind in practical implementations, since indexing arrays in Python or Matlab does not work cyclically.

At a high level, the FFT works by using Lemma 6.3.1 to recursively split the problem up into smaller subproblems. Each split reduces the length of the signals by half, and after $\log_2 n$ such splits, the problem is reduced to computing the DFT of very short signals, say, of length $n = 1$, which can be computed in constant time. Each time we recombine the results from subproblems using (6.3.1) we incur a cost of $O(n)$ operations, yielding an overall complexity of $O(n \log_2 n)$.

The description above is just a summary of the main ideas. We now consider the FFT algorithm in detail and prove that the computational complexity is $O(n \log_2 n)$. From now on we restrict our analysis to signal lengths n that are powers of 2, so $n = 2^k$ for some positive integer k. This ensures that every time we split we end up with even-length signals that can be further split evenly, and makes the FFT algorithm simpler to state.

In Algorithm 6.3 we show Python code for the FFT algorithm using a recursive implementation. The algorithm checks if the input signal has length $n = 1$, and if so, it returns f, since $\mathcal{D}_1$ is the identity (as noted above). If $n \geq 2$ then the algorithm splits the signal into its even and odd parts $f_e =$ `f[::2]` and $f_o =$ `f[1::2]`, takes their DFT by calling `fft` recursively, and then recombines the resulting DFTs $\mathcal{D}f_e$ and $\mathcal{D}f_0$ using (6.3.1) to compute $\mathcal{D}f$. We also point out that Steps 11 and 12 simply extend $\mathcal{D}f_e$ and $\mathcal{D}f_o$ to functions on $\mathbb{Z}_n$ by periodicity, so that (6.3.1) can be applied (see Remark 6.3.2). We also recall (see Step 13) that `1j` $= \sqrt{-1}$ in Python.

Algorithm 6.3.1 The Fast Fourier Transform (FFT) in Python

```
1  import numpy as np
2
3  def fft(f):
4      n = len(f)
5      k = np.arange(n)
6      if n == 1:
7          return f
8      else:
9          Dfe = fft(f[::2])
10         Dfo = fft(f[1::2])
11         Dfe = np.hstack((Dfe,Dfe))
12         Dfo = np.hstack((Dfo,Dfo))
13         return Dfe + np.exp(-2*np.pi*1j*k/n)*Dfo
```

We now carefully analyze the computational complexity of the FFT. In order to do so, it is important to consider how one chooses to count operations. Here, we will count an operation of addition, subtraction, multiplication or division of *real* numbers as a single operation. Applying functions like cos or sin to real numbers will also count as one operation. Operations on complex numbers take multiple real operations. For example, adding two complex numbers

$$(a + ib) + (c + id) = (a + b) + i(c + d)$$

requires two real additions (adding $a+b$ and $c+d$). The complex number $a+ib$ is stored in memory as a pair (a,b), with the understanding that b is the imaginary part of the complex number, and a is the real part. So the addition operation between $(a+b)$ and $i(c+d)$ is not a real operation and does not need to be computed, since the result is simply stored as $(a+b, c+d)$. The same can be said for the multiplication of i and $c+d$; these numbers are not multiplied by the computer (how could they be?), so this does not count as an operation. The result of this discussion is that addition or subtraction of complex numbers takes two real operations.

Multiplication of complex numbers takes more than two real operations, since we must write

$$(a+ib)(c+id) = (ac-bd) + i(ad+bc).$$

So multiplying two complex numbers requires computing $ac-bd$ and $ad+bc$, which takes 6 real operations (4 multiplications and 2 additions). Note it is also possible to multiply numbers in polar coordinates with fewer operations, since

$$r_1 e^{i\theta_1} r_2 e^{-i\theta_2} = r_1 r_1 e^{i(\theta_1+\theta_2)}$$

requires only 2 real operations (one multiplication and one addition). However, adding or subtracting in polar coordinates is more expensive, since one must convert back to Cartesian coordinates $a+ib$ first via Euler's Formula. Thus, we will conduct our analysis assuming the complex numbers are stored in Cartesian format, requiring 2 operations per addition or subtraction and 6 operations for a multiplication.

We define

$$A_n = \text{Number of real operations taken by the FFT on } L^2(\mathbb{Z}_n).$$

We note that since $\mathcal{D}_1$ is the identity, which requires no operations to compute, we have $A_1 = 0$. We can use Lemma 6.3.1 to obtain a recursion for A_n in terms of $A_{\frac{n}{2}}$. Indeed, using (6.3.1) to compute $\mathcal{D}f(\ell)$ for $\ell = 0, 1, \ldots, n-1$ requires computing $\mathcal{D}_{\frac{n}{2}} f_e$ and $\mathcal{D}_{\frac{n}{2}} f_o$, which takes $2A_{\frac{n}{2}}$ operations. Then we need to perform the complex addition and multiplication in (6.3.1), which takes $2+6=8$ operations. This has to be done n times (for $\ell = 0, 1, \ldots, n-1$), yielding $8n$ operations. Thus, we find that

$$A_n = 2A_{\frac{n}{2}} + 8n. \tag{6.3.2}$$

Notice we did not allocate any computation time to the complex exponential $e^{-2\pi i \ell/n}$. This quantity is independent of the signal f and can be computed

once, offline, and the values can be stored in memory. Thus, we do not include this in the operation count of the algorithm.

The recursion (6.3.2) allows us to solve explicitly for A_n, as shown in the following lemma.

Lemma 6.3.3. *Let $n = 2^k$ for a positive integer k, and assume A_n satisfies (6.3.2) for $n \geq 2$ and $A_1 = 0$. Then we have*

$$A_n = 8\, n \log_2 n. \tag{6.3.3}$$

Proof. We can iterate (6.3.2) again to obtain

$$A_n = 2\left(2A_{\frac{n}{2^2}} + 8\tfrac{n}{2}\right) + 8n = 2^2 A_{\frac{n}{2^2}} + 8n + 8n,$$

and

$$A_n = 2^2\left(2A_{\frac{n}{2^3}} + 8\tfrac{n}{2^2}\right) + 8n + 8n = 2^3 A_{\frac{n}{2^3}} + 8n + 8n + 8n.$$

Iterating this k times, by induction, we have that

$$A_n = 2^k A_{\frac{n}{2^k}} + 8nk.$$

Since $n = 2^k$ we have $A_{\frac{n}{2^k}} = A_1 = 0$ and $k = \log_2 n$, which yields

$$A_n = 8\, nk = 8\, n \log_2 n. \qquad \square$$

Lemma 6.3.3 shows that the FFT takes at most $8\, n \log_2 n$ operations to execute, which is an order of magnitude improvement over the naive implementation based on Definition 6.2.2 that takes $O(n^2)$ operations. It turns out this complexity can be improved to $5n \log_2 n$ by removing some redundant computations.

Remark 6.3.4. As in Remark 6.3.2, we consider writing (6.3.1) in different ways depending on whether $\ell \leq \frac{n}{2} - 1$ or not. Since

$$e^{-2\pi i(\ell + \frac{n}{2})/n} = e^{-2\pi i \ell/n} e^{-\pi i} = -e^{-2\pi i \ell/n}$$

and $\mathcal{D}_{\frac{n}{2}} f_e(\ell), \mathcal{D}_{\frac{n}{2}} f_o$ are $\frac{n}{2}$ periodic, we can write (6.3.1) as

$$\mathcal{D}_n f(\ell) = \mathcal{D}_{\frac{n}{2}} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}} f_o(\ell), \text{ and}$$

$$\mathcal{D}_n f(\ell + \tfrac{n}{2}) = \mathcal{D}_{\frac{n}{2}} f_e(\ell) - e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}} f_o(\ell)$$

for $\ell = 0, 1, \ldots, \frac{n}{2} - 1$. Notice that there are now only $\frac{n}{2}$ complex multiplications to perform, since they are common between the two parts. We still need n

additions, so the number of real operations to combine $\mathcal{D}_{\frac{n}{2}} f_e(\ell)$ and $\mathcal{D}_{\frac{n}{2}} f_o$ into $\mathcal{D}f$ is reduced from $8n$, as above, to $6\frac{n}{2} + 2n = 5n$. The recursion (6.3.2) becomes

$$A_n = 2A_{\frac{n}{2}} + 5n$$

and the complexity is thus $A_n = 5n \log_2 n$.

The computational complexity can be further improved to $4n \log_2 n$ by considering a 3-way split, where the odd terms are further split in half before the recursion. Exercise 6.3.5 explores this algorithm, called the split-radix FFT, which enjoyed its role as the fastest FFT for power-of-two n for quite some time, until the relatively recent work [13] made some modifications to the algorithm and improved the complexity to $\frac{34}{9} n \log_2 n$. The split radix methods are variants of the general Cooley-Tukey FFT algorithm [7]. The Cooley-Tukey FFT can be applied to any n, not just powers of 2, although the constants in the computational complexity are worse for non powers of two.

Exercise 6.3.5 (Split-radix FFT)**.** Assume n is a power of 2 and let $f \in L^2(\mathbb{Z}_n)$. Define $f_e \in L^2(\mathbb{Z}_{\frac{n}{2}})$, and $f_{o,1}, f_{o,2} \in L^2(\mathbb{Z}_{\frac{n}{4}})$ by

$$f_e(k) = f(2k), \quad f_{o,1}(k) = f(4k+1), \quad \text{and} \quad f_{o,2}(k) = f(4k+3).$$

(i) Show that

$$\mathcal{D}_n f(\ell) = \mathcal{D}_{\frac{n}{2}} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,1}(\ell) + e^{-2\pi i 3\ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,2}(\ell). \tag{6.3.4}$$

(ii) The FFT algorithm based on the 3-way split in (6.3.4) is called the split-radix FFT algorithm. As in Remark 6.3.4, there are a lot of redundant computations in (6.3.4), and these must be accounted for in order to realize the improved complexity of the split-radix FFT. Show that

$$\begin{aligned}
\mathcal{D}_n f(\ell) &= \mathcal{D}_{\frac{n}{2}} f_e(\ell) + \ (e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,1}(\ell) + e^{-2\pi i 3\ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,2}(\ell)), \\
\mathcal{D}_n f(\ell + \tfrac{n}{2}) &= \mathcal{D}_{\frac{n}{2}} f_e(\ell) - \ (e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,1}(\ell) + e^{-2\pi i 3\ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,2}(\ell)), \\
\mathcal{D}_n f(\ell + \tfrac{n}{4}) &= \mathcal{D}_{\frac{n}{2}} f_e(\ell + \tfrac{n}{4}) - i(e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,1}(\ell) - e^{-2\pi i 3\ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,2}(\ell)), \\
\mathcal{D}_n f(\ell + \tfrac{3n}{4}) &= \mathcal{D}_{\frac{n}{2}} f_e(\ell + \tfrac{n}{4}) + i(e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,1}(\ell) - e^{-2\pi i 3\ell/n} \mathcal{D}_{\frac{n}{4}} f_{o,2}(\ell)),
\end{aligned}$$

for $0 \le \ell \le \frac{n}{4} - 1$. This gives all the outputs of $\mathcal{D}f(\ell)$ and reduces the number of multiplications and additions required.

(iii) Explain how the observations in Part (ii) allow you to compute $\mathcal{D}_n f$ from $\mathcal{D}_{\frac{n}{2}} f_e$, $\mathcal{D}_{\frac{n}{4}} f_{o,1}$ and $\mathcal{D}_{\frac{n}{4}} f_{o,2}$ using $6n$ real operations. [Note, multiplications with ± 1 or $\pm i$ do not count, since they amount to negation of real or imaginary parts, which can be absorbed into the next operation by changing it from addition to subtraction or vice versa]

(iv) Show that part (iii) implies that the number of real operations taken by the split-radix FFT, denoted again as A_n, satisfies the recursion

$$A_n = A_{\frac{n}{2}} + 2A_{\frac{n}{4}} + 6n.$$

Explain why $A_1 = 0$ and $A_2 = 4$. Use this to show that $A_n \leq 4n \log_2 n$. [Hint: Define $B_n = A_n - 4n \log_2 n$ and show that B_n satisfies

$$B_n = B_{\frac{n}{2}} + 2B_{\frac{n}{4}}$$

with $B_1 = 0$ and $B_2 = -4$. Use this to argue that $B_n \leq 0$ for all power-of-two n.] [Note: If one is more careful about redundant computations (there are additional multiplications with ± 1 or $\pm i$ that can be skipped), then the complexity of the split-radix FFT algorithm is actually $4n \log_2 n - 6n + 8$ real operations].

△

Remark 6.3.6. Before concluding this section, we remark that the FFT algorithm can be easily extended to compute the inverse DFT. The analogous identity to (6.3.1) for $\mathcal{D}_n^{-1}$ is

$$\mathcal{D}_n^{-1} f(\ell) = \frac{1}{2}\mathcal{D}_{\frac{n}{2}}^{-1} f_e(\ell) + \frac{1}{2} e^{2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}}^{-1} f_o(\ell). \tag{6.3.5}$$

The proof of (6.3.5) is left to an exercise (below). The inverse FFT is formulated very similarly to the FFT, just using (6.3.5) in place of (6.3.1). As an alternative, one can always use the identity

$$\mathcal{D}_n^{-1} = \frac{1}{n}\overline{\mathcal{D}_n \overline{f}}$$

to compute the inverse DFT efficiently using a single forward FFT, two complex conjugation operations, and an elementwise division.

Exercise 6.3.7. Prove that (6.3.5) holds under the same assumptions as in Lemma 6.3.1. △

6.4 Parseval's Identities

Recall that for a real-valued orthogonal matrix Q, the inverse of Q is the transpose Q^T. Since the DFT is an orthogonal change of coordinates, a similar property holds. Indeed, the following lemma shows that the IDFT $\mathcal{D}^{-1}$ is the *adjoint* of the DFT $\mathcal{D}$, up to the factor $1/n$.

Lemma 6.4.1. *For each $f, g \in L^2(\mathbb{Z}_n)$ we have*

$$\frac{1}{n}\langle \mathcal{D}f, g\rangle = \langle f, \mathcal{D}^{-1}g\rangle.$$

Proof. We compute

$$\begin{aligned}
\langle f, \mathcal{D}^{-1}g\rangle &= \sum_{k=0}^{n-1} f(k)\overline{\mathcal{D}^{-1}g(k)} \\
&= \frac{1}{n}\sum_{k=0}^{n-1} f(k)\overline{\sum_{\ell=0}^{n-1} g(\ell)\omega^{k\ell}} \\
&= \frac{1}{n}\sum_{\ell=0}^{n-1} \overline{g(\ell)}\sum_{k=0}^{n-1} f(k)\omega^{-k\ell} \\
&= \frac{1}{n}\sum_{\ell=0}^{n-1} \overline{g(\ell)}\mathcal{D}f(\ell) = \frac{1}{n}\langle \mathcal{D}f, g\rangle,
\end{aligned}$$

which completes the proof. □

An immediate consequence of Lemma 6.4.1 are Parseval's Identities, which state that the DFT preserves the inner product on $L^2(\mathbb{Z}_n)$, and hence also preserves the norm, up to the $\frac{1}{n}$ factor.

Theorem 6.4.2 (Parseval's Identities)**.** *Let $f, g \in L^2(\mathbb{Z}_n)$. Then it holds that*

(i) $\langle f, g\rangle = \frac{1}{n}\langle \mathcal{D}f, \mathcal{D}g\rangle$, *and*

(ii) $\|f\|^2 = \frac{1}{n}\|\mathcal{D}f\|^2$.

Proof. To prove (i), we use Lemma 6.4.1 and Theorem 6.2.4 to write

$$\langle f, g\rangle = \langle f, \mathcal{D}^{-1}\mathcal{D}g\rangle = \frac{1}{n}\langle \mathcal{D}f, \mathcal{D}g\rangle.$$

To prove (ii), we take $f = g$ in part (i). □

Remark 6.4.3. Of course, a similar statement holds for the inverse transform $\mathcal{D}^{-1}$. Indeed, Lemma 6.4.1 and Theorem 6.2.4 imply

$$\frac{1}{n}\langle f, g\rangle = \frac{1}{n}\langle \mathcal{D}\mathcal{D}^{-1}f, g\rangle = \langle \mathcal{D}^{-1}f, \mathcal{D}^{-1}g\rangle.$$

Setting $f = g$ yields $\frac{1}{n}\|f\|^2 = \|\mathcal{D}^{-1}f\|^2$.

6.5 Convolution and the DFT

One of the most important properties of the DFT is that it turns convolutions into products. This property makes the DFT useful for solving partial differential equations, and for efficiently computing certain convolutions.

We first define the discrete convolution on the cyclic group $\mathbb{Z}_n$.

Definition 6.5.1. The *discrete cyclic convolution* of $f, g \in L^2(\mathbb{Z}_n)$, denoted $f * g$, is the function in $L^2(\mathbb{Z}_n)$ defined for each k by

$$(f * g)(k) = \sum_{j=0}^{n-1} f(j)g(k-j). \tag{6.5.1}$$

We note that the definition of the convolution makes use of the fact that $\mathbb{Z}_n$ is a cyclic group when $k-j$ falls outside of $0, 1, \ldots, n-1$ (i.e., the values wrap around). We leave some basic properties of the convolution to an exercise.

Exercise 6.5.2. Let $f, g, h \in L^2(\mathbb{Z}_n)$. Show that the following hold.

(i) $f * g = g * f$;

(ii) $f * (g * h) = (f * g) * h$;

(iii) $f * (g + h) = f * g + f * h$.

△

The main result of this section concerns the DFT of a convolution.

Lemma 6.5.3 (Convolution and the DFT)**.** *For $f, g \in L^2(\mathbb{Z}_n)$ we have*

$$\mathcal{D}(f * g) = \mathcal{D}f \cdot \mathcal{D}g. \tag{6.5.2}$$

Proof. We compute

$$\begin{aligned}
\mathcal{D}(f * g)(\ell) &= \sum_{k=0}^{n-1} (f * g)(k)\omega^{-k\ell} \\
&= \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} f(j)g(k-j)\omega^{-k\ell} \\
&= \sum_{j=0}^{n-1} f(j)\omega^{-j\ell} \left(\sum_{k=0}^{n-1} g(k-j)\omega^{-(k-j)\ell} \right).
\end{aligned}$$

Since $\mathbb{Z}_n$ is cyclic, for any function $h \in L^2(\mathbb{Z}_n)$ and any $j \in \mathbb{Z}_n$ we have

$$\sum_{k=0}^{n-1} h(k) = \sum_{k=0}^{n-1} h(k-j).$$

Thus, the term in brackets above is exactly $\mathcal{D}g(\ell)$ and we have

$$\mathcal{D}(f * g)(\ell) = \sum_{j=0}^{n-1} f(j)\omega^{-j\ell}\mathcal{D}g(\ell) = \mathcal{D}f(\ell)\mathcal{D}g(\ell). \qquad \square$$

Remark 6.5.4. Lemma 6.5.3 is arguably one of the most important properties of the DFT. It allow for efficient computation of convolutions via the FFT. Indeed, we can use (6.5.2) to write

$$f * g = \mathcal{D}^{-1}(\mathcal{D}f \cdot \mathcal{D}g).$$

Using the FFT, we can compute the convolution with two forward and one inverse transformation, which takes $O(n \log n)$ operations. Computing the convolution by the definition (6.5.1) takes $O(n^2)$ operations.

Also, as we explore in Exercise 6.5.6, all discrete derivatives of f can be viewed as convolutions of f with a particular choice of g. Thus, Lemma 6.5.3 turns differential equations, involving discrete derivatives, into algebraic equations which are easy to solve. This enables the use of the FFT for efficiently solving linear constant coefficient partial differential equations. We explore this further in Section 6.6.

Remark 6.5.5. Of course, a similar property holds for the inverse DFT $\mathcal{D}^{-1}$. To see this, we use the identity $\mathcal{D}^{-1}f = \frac{1}{n}\overline{\mathcal{D}\overline{f}}$ to obtain

$$\mathcal{D}^{-1}(f * g) = \frac{1}{n}\overline{\mathcal{D}\overline{f * g}} = \frac{1}{n}\overline{\mathcal{D}\overline{f}} \cdot \overline{\mathcal{D}\overline{g}} = n\mathcal{D}^{-1}f \cdot \mathcal{D}^{-1}g.$$

Exercise 6.5.6. Discrete derivatives (difference quotients) can be interpreted as convolutions. Complete the following exercises.

(i) For $f \in L^2(\mathbb{Z}_n)$ define the backward difference

$$\nabla^- f(k) = f(k) - f(k-1).$$

Find $g \in L^2(\mathbb{Z}_n)$ so that $\nabla^- f = g * f$ and use this with Lemma 6.5.3 to show that $\mathcal{D}(\nabla^- f)(k) = (1 - \omega^{-k})\mathcal{D}f(k)$, where $\omega = e^{2\pi i/n}$.

(ii) For $f \in L^2(\mathbb{Z}_n)$ define the forward difference

$$\nabla^+ f(k) = f(k+1) - f(k).$$

Find $g \in L^2(\mathbb{Z}_n)$ so that $\nabla^+ f = g * f$ and use this with Lemma 6.5.3 to show that $\mathcal{D}(\nabla^+ f)(k) = (\omega^k - 1)\mathcal{D}f(k)$.

(iii) For $f \in L^2(\mathbb{Z}_n)$ define the centered difference by

$$\nabla f(k) = \frac{1}{2}(\nabla^- f(k) + \nabla^+ f(k)) = \frac{1}{2}(f(k+1) - f(k-1)).$$

Use parts (i) and (ii) to show that

$$\mathcal{D}(\nabla f)(k) = \frac{1}{2}(\omega^k - \omega^{-k})\mathcal{D}f(k) = i\sin(2\pi k/n)\mathcal{D}f(k).$$

(iv) For $f \in L^2(\mathbb{Z}_n)$, define the discrete Laplacian as

$$\Delta f(k) = \nabla^+\nabla^- f(k) = f(k+1) - 2f(k) + f(k-1).$$

Use parts (i) and (ii) to show that

$$\mathcal{D}(\Delta f)(k) = (\omega^k + \omega^{-k} - 2)\mathcal{D}f(k) = 2(\cos(2\pi k/n) - 1)\mathcal{D}f(k). \quad \triangle$$

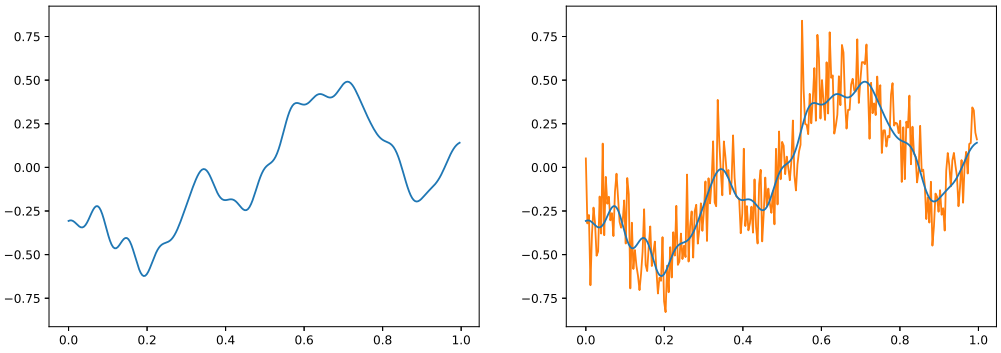
Exercise 6.5.7. Consider the Poisson equation on $\mathbb{Z}_n$

$$-\Delta f(k) = g(k) \quad \text{for } k \in \mathbb{Z}_n. \tag{6.5.3}$$

The source term $g \in L^2(\mathbb{Z}_n)$ is given, and $f \in L^2(\mathbb{Z}_n)$ is the unknown we wish to solve for. The Laplacian Δ is defined in Exercise 6.5.6. Use Exercise 6.5.6 (iii) to derive a solution formula for $f(k)$. Is there a condition you need to place on $\mathcal{D}g$ for your solution formula to make sense? [Hint: Take the DFT of both sides of (6.5.3), solve for $\mathcal{D}f$, and then apply the inverse DFT $\mathcal{D}^{-1}$. Be careful not to divide by zero when you solve for $\mathcal{D}f$.] $\triangle$

6.6 Application: Signal denoising

Noise arises in most signal acquisition processes. For example, in a microphone, air molecules constantly bombard the diaphragm, causing random vibrations not associated with the sound one wishes to record, and electronic circuitry generates thermal noise due to the natural random motions of electrons inside



(a) Clean signal (b) Noisy signal

Figure 6.6.1: Example of a clean signal, and a noisy version, corrupted with additive Gaussian noise. The clean signal is superimposed over the noisy one in (b) for reference.

conductors. Sensors on digital cameras are also prone to noise in the image capture process, for similar reasons, and this can be especially pronounced in low light situations. Noise can also be introduced after signal acquisition. For instance, noise can arise as the result of compression artifacts or corruption in the transmission of signals wirelessly. In Figure 6.6.1 shows a synthetic signal and a noisy version corrupted by additive Gaussian noise (at each sample point we add an independent Gaussian random variable to the signal). Many types of noise are well approximated by Gaussian noise due to the Central Limit Theorem in probability.

Let $f \in L^2(\mathbb{Z}_n)$ represent a noisy signal, like in Figure 6.6.1(b). While we are interested primarily in real-valued signals, the analysis is similar for complex-valued signals $f \in L^2(\mathbb{Z}_n)$, so we proceed in generality. The goal of signal denoising is to recover the clean signal in Figure 6.6.1(a), which amounts to removing the removing the noise. In the sections below, we discuss Tikhonov regularization and Total Variation regularization for denoising.

6.6.1 Tikhonov regularization

Python Notebook: .ipynb

A widely used and successful method for signal and image denoising is the regularized variational approach. While there are many versions of this, we focus here on a simple case known as Tikhonov regularization, which denoises

the signal by minimizing the function $E : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ defined by

$$E(u) = \sum_{k=0}^{n-1} |u(k) - f(k)|^2 + \lambda \sum_{k=0}^{n-1} |u(k) - u(k-1)|^2, \tag{6.6.1}$$

where $\lambda \geq 0$ is a parameter. The first term in E is called a data fidelity term, and it encourages the minimizer of E (i.e., the denoised signal) to remain close to the noisy signal f. The second term aims to measure the amount of noise in the signal u, by summing squared differences of neighboring signal values. Clean signals should give much smaller values for this term, compared to noisy signals. There is a parameter $\lambda \geq 0$ in the function E that controls the tradeoff between these two terms. For small λ we expect the minimizer of E to remain very close to f and we will thus have very little denoising. For large $\lambda > 0$, we expect to get a very smooth signal with the noise removed, but we may not accurately reconstruct the clean signal. Finding a good value of λ that removes noise while preserving important signal information is an important consideration in variational methods.

In this section, we show how to use the theory we have developed for the DFT to efficiently find the minimizer of E in order to denoise signals. For this, it is useful to rewrite the Tikhonov regularized functional E. We recall first the definitions of discrete derivatives from Exercise 6.5.6. The backward difference $\nabla^- : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is defined by

$$\nabla^- u(k) = u(k) - u(k-1),$$

while the forward difference is $\nabla^+ : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is defined by

$$\nabla^+ u(k) = u(k+1) - u(k).$$

The Laplacian $\Delta : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is the composition of the forward and backward differences, in either order, so $\Delta = \nabla^+\nabla^- = \nabla^-\nabla^+$. In terms of the values of $u(k)$, the Laplacian is given by

$$\Delta u(k) = u(k+1) - 2u(k) + u(k-1).$$

Since our signals are one dimensional, we should think of the Laplacian as just the second derivative of u, and indeed, it is exactly the difference quotient approximation of $u''(x)$ for a smooth function u.

In terms of this new notation, we can write E as

$$E(u) = \|u - f\|^2 + \lambda \|\nabla^- u\|^2. \tag{6.6.2}$$

The denoising problem is to minimize E over $u \in L^2(\mathbb{Z}_n)$, that is

$$\min_{u \in L^2(\mathbb{Z}_n)} E(u). \tag{6.6.3}$$

We now characterize minimizers of (6.6.1) as the solution of a differential equation known as the Euler-Lagrange equation.

Theorem 6.6.1. *Let $\lambda \geq 0$ and $f \in L^2(\mathbb{Z}_n)$. Then there exists a unique solution $u \in L^2(\mathbb{Z}_n)$ of the optimization problem (6.6.3). Furthermore, the minimizer u is also characterized as the unique solution of the Euler-Lagrange equation*

$$u - \lambda \Delta u = f. \tag{6.6.4}$$

Before proving the theorem, we need to record some properties of ∇^+, ∇^- and Δ. These properties are the discrete analog of integration by parts formulas.

Proposition 6.6.2. *For all $u, v \in L^2(\mathbb{Z}_n)$ the following hold.*

(i) $\langle \nabla^- u, v \rangle = -\langle u, \nabla^+ v \rangle$

(ii) $\langle \nabla^+ u, v \rangle = -\langle u, \nabla^- v \rangle$

(iii) $\langle \Delta u, v \rangle = \langle u, \Delta v \rangle$

Proof. We first prove (i). Let $u, v \in L^2(\mathbb{Z}_n)$ and compute

$$\begin{aligned}
\langle \nabla^- u, v \rangle &= \sum_{k=0}^{n-1} \nabla^- u(k) \overline{v(k)} \\
&= \sum_{k=0}^{n-1} (u(k) - u(k-1)) \overline{v(k)} \\
&= \sum_{k=0}^{n-1} u(k) \overline{v(k)} - \sum_{k=0}^{n-1} u(k-1) \overline{v(k)} \\
&= \sum_{k=0}^{n-1} u(k) \overline{v(k)} - \sum_{k=-1}^{n-2} u(k) \overline{v(k+1)}.
\end{aligned}$$

Now, since $\mathbb{Z}_n$ is cyclic, we have

$$\sum_{k=-1}^{n-2} u(k) \overline{v(k+1)} = \sum_{k=0}^{n-1} u(k) \overline{v(k+1)}$$

and so

$$\langle \nabla^- u, v\rangle = \sum_{k=0}^{n-1} u(k)\overline{(v(k)-v(k+1))} = -\langle u, \nabla^+ v\rangle. \qquad \square$$

To prove (ii), we use Part (i) to obtain

$$\langle \nabla^+ u, v\rangle = \overline{\langle v, \nabla^+ u\rangle} = -\overline{\langle \nabla^- v, u\rangle} = -\langle u, \nabla^- v\rangle.$$

To prove (iii), since $\Delta = \nabla^+\nabla^-$ we can use parts (i) and (ii) to find that

$$\begin{aligned}\langle \Delta u, v\rangle &= \langle \nabla^+\nabla^- u, v\rangle \\ &= -\langle \nabla^- u, \nabla^- v\rangle \\ &= \langle u, \nabla^+\nabla^- v\rangle \\ &= \langle u, \Delta v\rangle,\end{aligned}$$

which completes the proof.

Remark 6.6.3. Notice in the proof of Proposition 6.6.2(iii) that we can take $u = v$ to obtain the identity

$$\|\nabla^- u\|^2 = \langle \nabla^- u, \nabla^- u\rangle = -\langle \Delta u, u\rangle.$$

Similarly, we can also write $\|\nabla^+ u\|^2 = -\langle \Delta u, u\rangle$.

Proof of Theorem 6.6.1. The proof is split into 3 parts.

1. We first show that the solution of (6.6.4) has a unique solution $u \in L^2(\mathbb{Z}_n)$. To see this, we first establish uniqueness. Suppose that u and v satisfy (6.6.4). Then by subtracting the equations satisfied by u and v, we find that $w = u - v$ satisfies

$$w - \lambda \Delta w = 0.$$

We take the inner product with w on both sides and use Remark 6.6.3 to obtain

$$0 = \langle w - \lambda \Delta w, w\rangle = \langle w, w\rangle - \lambda\langle \Delta w, w\rangle = \|w\|^2 + \lambda \|\nabla^- w\|^2.$$

Therefore $\|w\|^2 = 0$, and so $0 = w = u - v$. Therefore $u = v$ and so the solution, if it exists, is unique.

The existence of a solution to (6.6.4) now follows from linear algebra, since the equation $u - \lambda\Delta u = f$ can be written as $Au = f$ where $A = I - \lambda\Delta$. The uniqueness proof above shows that the kernel of A is trivial, so $A : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is one-to-one. By the rank-nullity theorem A is also onto, so for every

$f \in L^2(\mathbb{Z}_n)$ there exists $u \in L^2(\mathbb{Z}_n)$ such that $Au = f$. This completes the fist step of the proof.

2. We now show that the unique solution of (6.6.4) solves the optimization problem (6.6.3). Let $u \in L^2(\mathbb{Z}_n)$ be the solution of (6.6.4), which exists due to Part (i). To show that u solves the optimization problem (6.6.3), we need to show that for all $w \in L^2(\mathbb{Z}_n)$ we have

$$E(u) \leq E(w). \tag{6.6.5}$$

Let $w \in L^2(\mathbb{Z}_n)$, set $v = w - u$, and define the function $e : \mathbb{R} \to \mathbb{R}$ by

$$e(t) = E(u + tv).$$

Then we have

$$E(w) - E(u) = e(1) - e(0) = \int_0^1 e'(t)\, dt.$$

The goal of the proof is to show that $e'(t) \geq 0$, so that $E(w) - E(u) \geq 0$. To do this, we first need a simple algebraic identity. For any $g, h \in L^2(\mathbb{Z}_n)$ we have

$$\|g + h\|^2 = \langle g + h, g + h\rangle = \langle g, g\rangle + \langle g, h\rangle + \langle h, g\rangle + \langle h, h\rangle.$$

Since $\langle h, g\rangle = \overline{\langle g, h\rangle}$ and $z + \overline{z} = 2\mathrm{Re}z$ for any $z \in \mathbb{C}$, we have

$$\|g + h\|^2 = \|g\|^2 + 2\mathrm{Re}\langle g, h\rangle + \|h\|^2.$$

We now apply this to both terms in $e(t) = E(u + tv)$ to obtain

$$\begin{aligned} e(t) = E(u + tv) &= \|u + tv - f\|^2 + \lambda\|\nabla^-(u + tv)\|^2 \\ &= \|u - f + tv\|^2 + \lambda\|\nabla^- u + t\nabla^- v\|^2 \\ &= \|u - f\|^2 + 2t\mathrm{Re}\langle u - f, v\rangle + t^2\|v\|^2 \\ &\quad + \lambda\|\nabla^- u\|^2 + 2\lambda t\mathrm{Re}\langle\nabla^- u, \nabla^- v\rangle + \lambda t^2\|\nabla^- v\|^2. \end{aligned}$$

By Proposition 6.6.2 we have

$$\langle\nabla^- u, \nabla^- v\rangle = -\langle\nabla^+\nabla^- u, v\rangle = -\langle\Delta u, v\rangle.$$

Therefore, we can simplify the expression above to read

$$e(t) = E(u) + 2t\mathrm{Re}\langle u - \lambda\Delta u - f, v\rangle + t^2\left(\|v\|^2 + \lambda\|\nabla^- v\|^2\right).$$

Differentiating in t we have

$$e'(t) = 2\mathrm{Re}\langle u - \lambda\Delta u - f, v\rangle + 2t\left(\|v\|^2 + \lambda\|\nabla^- v\|^2\right). \tag{6.6.6}$$

Since u solves (6.6.4), the first term vanishes, and so $e'(t) \geq 0$ for $t \geq 0$, which completes the proof of this step.

3. Finally, we show that any solution of the optimization problem (6.6.3) must solve the Euler-Lagrange equation (6.6.4). Since solutions of (6.6.4) are unique, this shows that minimizers E are also unique, and completes the proof. Let $u \in L^2(\mathbb{Z}_n)$ be a solution to the optimization problem (6.6.3). Let $v \in L^2(\mathbb{Z}_n)$, to be determined later, and as in Part (ii) we define $e(t) = E(u+tv)$. Since $E(u) \leq E(w)$ for all $w \in L^2(\mathbb{Z}_n)$, we have $e(0) = E(u) \leq E(u+tv) = e(t)$. Therefore e has a minimum at $t = 0$ and so $e'(0) = 0$. Using (6.6.6) we have

$$0 = e'(0) = 2\mathrm{Re}\langle u - \lambda\Delta u - f, v\rangle. \tag{6.6.7}$$

Setting $v = u - \lambda\Delta u - f$ we obtain

$$0 = 2\mathrm{Re}\|u - \lambda\Delta u - f\|^2,$$

which shows that $u - \lambda\Delta u = f$, as desired. □

Remark 6.6.4. A key step in the proof of Theorem (6.6.4) is differentiating the function $e(t)$, which amounts to computing

$$\frac{d}{dt}E(u+tv).$$

This is called computing a *variation* of E in the direction of v, and is a fundamental idea in the *calculus of variations*. In fact, this computation is part of the definition of the *Gateaux derivative* of E, and denoted $dE(u) : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$, and defined by

$$dE(u)v := \frac{d}{dt}\Big|_{t=0} E(u+tv).$$

The Gateaux derivative is simply the directional derivative of E in the direction v. The proof of Theorem 6.6.1 simply used the fact that minimizers u of E must satisfy $dE(u)v = 0$ for all $v \in L^2(\mathbb{Z}_n)$; that is, their directional derivatives in every direction are zero. It follows from (6.6.7) that

$$dE(u)v = e'(0) = 2\mathrm{Re}\langle u - \lambda\Delta u - f, v\rangle.$$

The gradient of E, denoted ∇E, is defined as the mapping $\nabla E : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ satisfying

$$dE(u)v = \mathrm{Re}\langle \nabla E(u), v\rangle.$$

In this case, the formula for the Gateaux derivative above shows that the gradient of E is

$$\nabla E(u) = u - \lambda \Delta u - f.$$

Thus, the Euler-Lagrange equation (6.6.4) is simply the necessary condition $\nabla E(u) = 0$ satisfied by minimizers of E.

The solution of (6.6.4) satisfies a maximum principle, which says that its values cannot exceed the maximum and minimum values of the noisy signal f. This is a desirable property of denoising algorithms.

Proposition 6.6.5 (Maximum Principle)**.** *Let $\lambda \geq 0$ and let $f \in L^2(\mathbb{Z}_n)$ be real-valued. Let $u \in L^2(\mathbb{Z}_n)$ be the solution of (6.6.4), which is also real-valued. Then we have*

$$\min_{\mathbb{Z}_n} f \leq u \leq \max_{\mathbb{Z}_n} f. \tag{6.6.8}$$

Proof. The proof uses a maximum principle argument. Let $k \in \mathbb{Z}_n$ be a point at which u attains its maximum value over $\mathbb{Z}_n$. Then $u(k) \geq u(j)$ for all $j \in \mathbb{Z}_n$, and so

$$\Delta u(k) = u(k+1) - 2u(k) + u(k-1) \leq u(k) - 2u(k) + u(k) = 0.$$

Inserting this into (6.6.4) we have

$$f(k) = u(k) - \lambda \Delta u(k) \geq u(k),$$

and so $\max_{\mathbb{Z}_n} u = u(k) \leq f(k) \leq \max_{\mathbb{Z}_n} f$.

Similarly, if $k \in \mathbb{Z}_n$ is a point where u attains its minimum value over $\mathbb{Z}_n$ then $\Delta u(k) \geq 0$ and we find that $\min_{\mathbb{Z}_n} u = u(k) \geq \min_{\mathbb{Z}_n} f$. □

So far we have not made any connection between denoising and the DFT. We now show how to use the FFT to efficiently solve the Euler-Lagrange equation (6.6.4) to obtain the denoised signal. This is an example of the application of the FFT to numerically solving partial differential equations. We simply take the DFT on both sides of (6.6.4) and use linearity of the DFT to obtain

$$\mathcal{D}u - \lambda \mathcal{D}(\Delta u) = \mathcal{D}f. \tag{6.6.9}$$

We now use Exercise 6.5.6(iv) to obtain

$$\mathcal{D}(\Delta u)(k) = 2(\cos(2\pi k/n) - 1)\mathcal{D}u(k).$$

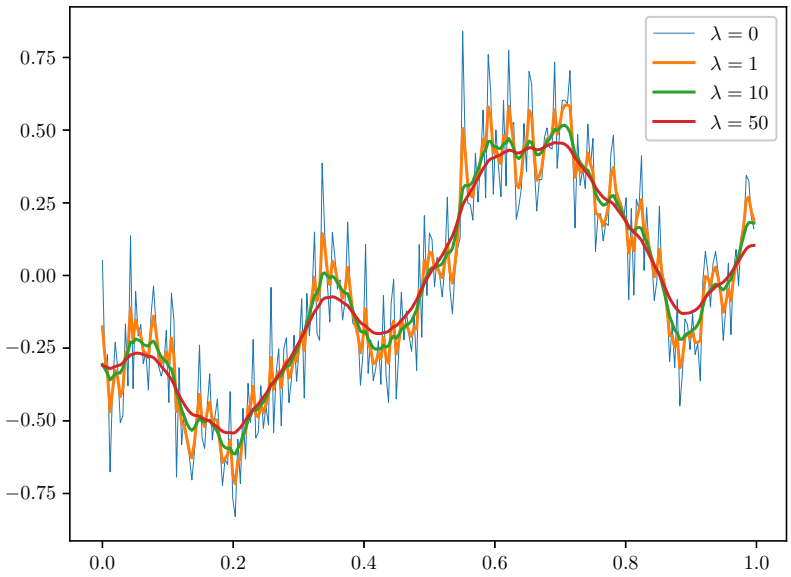


Figure 6.6.2: Example of Tikhonov denoising for $\lambda = 0, 1, 10, 50$, using the noisy signal from Figure 6.6.1.

This is the main utility of the DFT for solving partial differential equations; it turns derivatives into multiplication. We now substitute this into (6.6.9) to obtain

$$\mathcal{D}u(k) - 2\lambda(\cos(2\pi k/n) - 1)\mathcal{D}u(k) = \mathcal{D}f(k).$$

We can solve algebraically for $\mathcal{D}u(k)$ now, yielding

$$\mathcal{D}u(k) = G_\lambda(k)\mathcal{D}f(k),$$

where

$$G_\lambda(k) = \frac{1}{1 + 2\lambda - 2\lambda\cos(2\pi k/n)}.$$

Taking the inverse transform on both sides we obtain

$$u = \mathcal{D}^{-1}(G_\lambda \mathcal{D}f). \tag{6.6.10}$$

This gives a very efficient solution method, which solves the discrete differential equation (6.6.4) in $O(n \log n)$ operations using the FFT. Figure 6.6.2 shows examples of denoised images for various values of λ.

Remark 6.6.6. When using the DFT for signal denoising, the signal is implicitly extended periodically to $\mathbb{Z}_n$. This can introduce a sharp discontinuity

in the signal, since $f(0)$ and $f(n-1)$ may not be similar, leading to boundary effects in the denoised signal. To eliminate this, we first take the even extension of the signal (see (6.8.4)) before applying the solution formula (6.6.10). The even extension extends a signal periodically without introducing discontinuities. As an alternative, one may use the Discrete Cosine Transform in place of the DFT in the solution formula (6.6.10). Section 6.8 covers the Discrete Cosine Transform.

The Fourier solution formula (6.6.10) also gives us insight into how Tikhonov regularization works to denoise a signal. Indeed, from (6.6.10) we see that each Fourier coefficient $\mathcal{D}f(k)$ is simply multiplied by $G_\lambda(k)$ before taking an inverse transformation to obtain u. This acts to attenuate certain frequencies, where G_λ is small. In signal processing, this type of operation is called *filtering*. In Figure 6.6.3(a) we show the Tikhonov filters G_λ for various values of λ. All the Tikhonov filters are *low-pass* filters, which means the lower frequencies get multiplied by $G_\lambda \approx 1$ and are unchanged, while the higher frequencies are attenuated. The amount of attenuation of high frequencies increases with λ. This attenuation is exactly how the noise is removed from the signal; noise typically contains very high frequency components.

We can also rewrite the formula (6.6.10) in the signal domain using the convolution property (Lemma 6.5.3). Indeed, defining $g_\lambda = \mathcal{D}^{-1}G_\lambda$ so that $G_\lambda = \mathcal{D}g_\lambda$ we have

$$u = \mathcal{D}^{-1}(\mathcal{D}g_\lambda \cdot \mathcal{D}f) = g_\lambda * f.$$

Thus, the denoised signal u is merely the convolution of g_λ with f. The function g_λ is often called a *kernel*. Figure 6.6.3(b) shows the kernel g_λ for various values of λ. The convolution to compute $u(k)$ should be thought of as a weighted average, weighted by g_λ, of the values of f nearby k. Indeed, we have

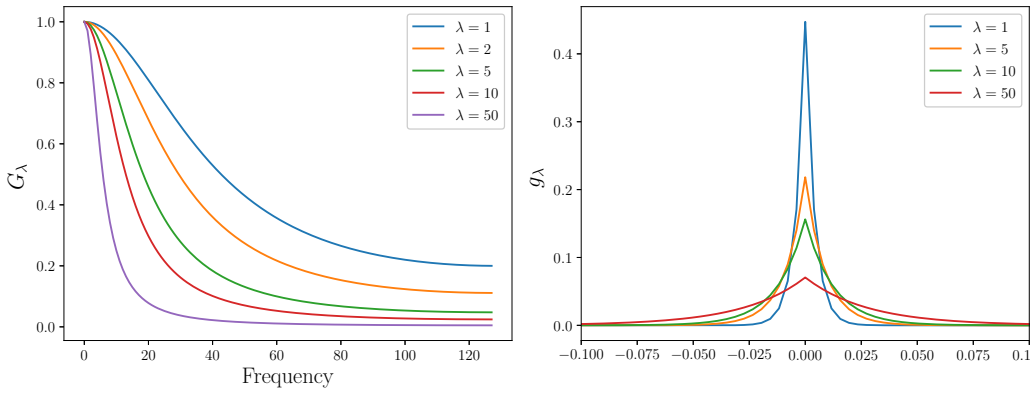
$$u(k) = (g_\lambda * f)(k) = \sum_{j=0}^{n-1} g_\lambda(j) f(k-j).$$

As we increase λ we can see in Figure 6.6.3(b) that the width of the kernels increases, meaning we are averaging over a wider neighborhood of the signal, yielding a smoother and less noisy signal.

6.6.2 Total Variation regularization

Python Notebook: .ipynb

Tikhonov regularization works well on signals that are smooth and do not have substantial high frequency content. Indeed, as we showed in Section



(a) Filters G_λ (b) Kernels g_λ

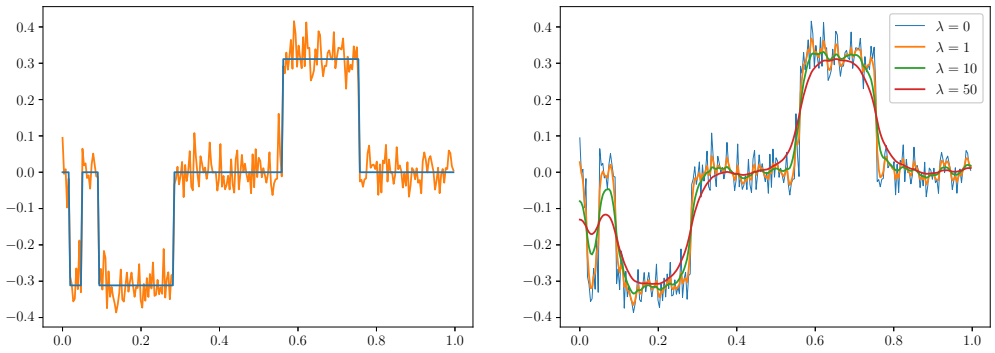
Figure 6.6.3: Examples of the (a) Tikhonov filters G_λ and (b) Tikhonov kernels g_λ for different values of λ. The filters work by allowing low frequencies to pass through, and attenuating high frequencies to a certain degree, depending on how large λ is. In the signal domain, this amounts to locally averaging the noisy signal with the kernel g_λ, whose width grows with λ.

6.6.1, Tikhonov regularization is just applying a linear filter to the signal that attenuates the high frequencies. Tikhonov regularization does not work well on signals that have sharp changes, like bar codes, or in two or three dimensions, natural images. Here, any sharp change in the signal (say an edge in an image, or a stripe in a bar code) contributes substantially to the higher end of the spectrum, and Tikhonov regularization excessively blurs the edges. Figure 6.6.4 shows an example of this, using Tikhonov regularization on a piecewise constant signal, with additive Gaussian noise. The sharp transitions in the signal are lost after Tikhonov denoising.

A better approach to denoising is Total Variation regularization, which minimizes the energy

$$E(u) = \frac{1}{2}\sum_{k=0}^{n-1} |u(k) - f(k)|^2 + \lambda \sum_{k=0}^{n-1} |u(k) - u(k-1)|. \tag{6.6.11}$$

The second term above is call the *Total Variation* of the signal u. We note that the only difference between (6.6.1) and (6.6.11) is that the Total Variation regularizer takes the sum of the absolute value of the differences $|u(k)-u(k-1)|$, instead of the squares $|u(k) - u(k-1)|^2$ that are used in Tikhonov regularization. Hence, Tikhonov regularization cares far more about minimizing the largest changes in the signal, which smoothes away the edges, while Total



(a) Noisy signal (b) Tikhonov denoising

Figure 6.6.4: Example of (a) a noisy piecewise constant signal (e.g., a bar code) and (b) the results of Tikhonov regularized denoising with various values of λ. Tikhonov denoising does not preserve the sharp changes in the signal between regions, and instead overly smoothes these edges.

Variation does not place any preference over which size changes are minimized. The following exercise explores this.

Exercise 6.6.7. Assume that $f \in L^2(\mathbb{Z}_n)$ is real-valued and increasing; that is $f(k) \leq f(k+1)$ for $k = 0, \ldots, n-2$.

(i) Show that

$$\sum_{k=1}^{n-1} |f(k) - f(k-1)| = f(n-1) - f(0).$$

Thus, the Total Variation norm does not care how the signal gets from $f(0)$ to $f(n-1)$, as long as the signal is monotone (i.e., not oscillating). Total Variation regularization assigns the same energy to a sharp jump as it does to a continuous ramp, so neither is preferred over the other.

(ii) Consider the Tikhonov regularizer

$$T = \sum_{k=1}^{n-1} |f(k) - f(k-1)|^2,$$

and suppose we fix $f(0) = 0$ and $f(n-1) = n-1$. As above, we assume f is increasing, so there are no oscillations. How does the Tikhonov energy T of a sharp jump, where $f(k) = 0$ for $k \leq n/2$ and $f(k) = n-1$ for $k > n/2$, compare to a smooth ramp $f(k) = k$? Which ones does Tikhonov regularization prefer?

△

For the rest of this section we will work with real-valued signals $f : \mathbb{Z}_n \to \mathbb{R}$. The theory is similar, but needlessly more complicated, for complex valued signals. For this purpose, we define the space $L^2(\mathbb{Z}_n; \mathbb{R})$ of real-valued signals $f : \mathbb{Z}_n \to \mathbb{R}$. Note that $L^2(\mathbb{Z}_n; \mathbb{R}) \subset L^2(\mathbb{Z}_n)$ is a linear subspace $L^2(\mathbb{Z}_n)$, so it inherits the inner-product structure and norm, etc.

We will proceed in generality, studying regularizers of the form

$$\sum_{k=0}^{n-1} \Phi(u(k) - u(k-1)) = \sum_{k=0}^{n-1} \Phi(\nabla^- u(k)), \tag{6.6.12}$$

where $\Phi : \mathbb{R} \to \mathbb{R}$ is a twice continuously differentiable, convex, and even function satisfying $\Phi(0) = 0$. We note that Tikhonov regularization corresponds to $\Phi(t) = t^2$, while Total Variation corresponds to $\Phi(t) = |t|$. Convexity can be defined in many different, but equivalent, ways. Here, we take the definition that $\Phi'' \geq 0$, so that Φ' is an increasing function. That is $\Phi'(s) \leq \Phi'(t)$ whenever $s \leq t$. Since Φ is even we have $\Phi'(0) = 0$ and so Φ is decreasing for $t < 0$ and increasing for $t > 0$. We also have that

$$(\Phi'(t) - \Phi'(s))(t - s) \geq 0 \tag{6.6.13}$$

for all $s, t \in \mathbb{R}$. This is an important monotonicity property used in the proofs in this section. To see why (6.6.13) holds, note that if $t \geq s$, then $t - s \geq 0$ and $\Phi'(t) - \Phi'(s) \geq 0$ since Φ' is increasing, so (6.6.13) holds. If $t \leq s$ then $\Phi'(t) - \Phi'(s) \leq 0$ and $t - s \leq 0$, so (6.6.13) still holds.

Remark 6.6.8. The assumption that Φ is continuously differentiable means that our theory does not apply directly to the Total Variation regularizer, where $\Phi(t) = |t|$ is not differentiable at $t = 0$. However, as we'll see below when we solve the Total Variation problem numerically, we will approximate it by $\Phi(t) = \sqrt{t^2 + \varepsilon^2}$ for a small $\varepsilon > 0$, which is necessary for numerical stability of our solution scheme. This approximation is a smooth function to which all our theory applies.

In order to simplify notation, we write $\Phi(\nabla^- u) : L^2(\mathbb{Z}_n; \mathbb{R}) \to L^2(\mathbb{Z}_n; \mathbb{R})$ for the function

$$\Phi(\nabla^- u)(k) = \Phi(\nabla^- u(k)).$$

Then the regularizer above can be written more simply as $\|\Phi(\nabla^- u)\|_1$, where we recall the ℓ_1-norm is given by $\|f\|_1 = \sum_{k=0}^{n-1} |f(k)|$. The regularized energy for denoising is then given by

$$E_\Phi(u) = \frac{1}{2}\|u - f\|^2 + \lambda \|\Phi(\nabla^- u)\|_1, \tag{6.6.14}$$

and the denoised signal is found by minimizing E_Φ. This section aims to study E_Φ mathematically, to prove a minimizer exists, is unique, and to find the corresponding Euler-Lagrange equation. Then we will explore how to minimize E_Φ numerically. The situation is far different compared to Tikhonov regularization, since here the Euler-Lagrange equation will be nonlinear. The techniques and tools in this section are thus somewhat different, and rely more on nonlinear and convex analysis.

We first prove the existence of a minimizer of (6.6.12).

Lemma 6.6.9. *For any $f \in L^2(\mathbb{Z}_n;\mathbb{R})$ and $\lambda \geq 0$, there exists $u \in L^2(\mathbb{Z}_n;\mathbb{R})$ minimizing (6.6.14), i.e., $E_\Phi(u) \leq E_\Phi(w)$ for all $w \in L^2(\mathbb{Z}_n;\mathbb{R})$. Furthermore, u satisfies*

$$\min_{\mathbb{Z}_n} f \leq u \leq \max_{\mathbb{Z}_n} f. \tag{6.6.15}$$

Proof. Let $f_{max} = \max_{\mathbb{Z}_n} f$ and $f_{min} = \min_{\mathbb{Z}_n} f$. Define the truncation operator $T : \mathbb{R} \to \mathbb{R}$ by

$$T(x) = \begin{cases} f_{min}, & \text{if } x < f_{min} \\ x, & \text{if } f_{min} \leq x \leq f_{max} \\ f_{max}, & \text{if } x > f_{max}. \end{cases}$$

The truncation operator is Lipschitz continuous with constant 1, that is

$$|T(x) - T(y)| \leq |x - y|.$$

To prove this, we may take $x > y$ without loss of generality and write

$$|T(x) - T(y)| = T(x) - T(y) = \int_y^x T'(s)\,ds \leq \max_{s\in\mathbb{R}} T'(s)|x - y|.$$

Note that we can ignore the points where T is not differentiable, $x = f_{min}, f_{max}$, since we can always split the integral to avoid them. Since at all points where T is differentiable we have $T'(x) = 0$ or $T'(x) = 1$, it follows that $\max_{s\in\mathbb{R}} T'(s) = 1$, which establishes the claim.

Let $u \in L^2(\mathbb{Z}_n;\mathbb{R})$ and define $w \in L^2(\mathbb{Z}_n;\mathbb{R})$ by $w(k) = T(u(k))$, that is, by truncating the values of u. We claim that

$$E_\Phi(w) \leq E_\Phi(u). \tag{6.6.16}$$

That is, E_Φ decreases by truncation. To see this, note that for each k, the Lipschitz property of T yields

$$|w(k) - f(k)|^2 = |T(u(k)) - T(f(k))|^2 \leq |u(k) - f(k)|^2,$$

and

$$|T(u(k)) - T(u(k-1))| \leq |u(k) - u(k-1)|.$$

Since Φ is increasing for $t \geq 0$, so we can use the evenness of Φ to write

$$\begin{aligned}\Phi(w(k) - w(k-1)) &= \Phi(|T(u(k)) - T(u(k-1))|) \\ &\leq \Phi(|u(k) - u(k-1)|) \\ &= \Phi(u(k) - u(k-1)).\end{aligned}$$

This establishes the claim (6.6.16).

Now we define the set

$$M = \{u \in L^2(\mathbb{Z}_n; \mathbb{R}) \,:\, f_{min} \leq u(k) \leq f_{max} \text{ for all } k\}.$$

The function E_Φ is continuous and the set M is closed and bounded (it's a rectangle when $L^2(\mathbb{Z}_n; \mathbb{R})$ is viewed as $\mathbb{R}^n$), so E_Φ attains its minimum value over M at some $u \in M$. To see that u minimizes E_Φ over $L^2(\mathbb{Z}_n; \mathbb{R})$, we take any other $v \in L^2(\mathbb{Z}_n; \mathbb{R})$ and define the truncation $w(k) = T(v(k))$. By (6.6.16) we have $E_\Phi(w) \leq E_\Phi(v)$, and since the truncation w belongs to M we have $E_\Phi(u) \leq E_\Phi(w)$, which completes the proof. □

We now prove the solution is unique and characterize the Euler-Lagrange equation.

Lemma 6.6.10. *Let $f \in L^2(\mathbb{Z}_n; \mathbb{R})$ and $\lambda \geq 0$. Then the minimizer $u \in L^2(\mathbb{Z}_n; \mathbb{R})$ of E_Φ is unique and is characterized as the unique solution of the Euler-Lagrange equation*

$$u - \lambda \nabla^+ \Phi'(\nabla^- u) = f. \tag{6.6.17}$$

Proof. Let $u \in L^2(\mathbb{Z}_n; \mathbb{R})$ be a minimizer of E_Φ, which exists due to Lemma 6.6.9. We first show that u satisfies (6.6.17). We take a variation of E_Φ, as in the proof of Theorem 6.6.1 and Remark 6.6.4. Let $v \in L^2(\mathbb{Z}_n; \mathbb{R})$ and define

$$e(t) := E_\Phi(u + tv).$$

Then, following a similar argument as in Theorem 6.6.1 we have

$$\begin{aligned}e'(t) &= \frac{d}{dt}\left(\frac{1}{2}\|u + tv - f\|^2 + \lambda \sum_{k=0}^{n-1} \Phi(\nabla^- u(k) + t\nabla^- v(k))\right) \\ &= \langle u - f, v\rangle + t^2\|v\|^2 + \lambda \sum_{k=0}^{n-1} \Phi'(\nabla^- u(k) + t\nabla^- v(k))\nabla^- v(k).\end{aligned}$$

Setting $t = 0$ and noting that $e'(0) = 0$ we have

$$\begin{aligned} 0 = e'(0) &= \langle u - f, v\rangle + \lambda \sum_{k=0}^{n-1} \Phi'(\nabla^- u(k))\nabla^- v(k) \\ &= \langle u - f, v\rangle + \lambda\langle \Phi'(\nabla^- u), \nabla^- v\rangle \\ &= \langle u - f, v\rangle - \lambda\langle \nabla^+\Phi'(\nabla^- u), v\rangle \\ &= \langle u - \lambda\nabla^+\Phi'(\nabla^- u) - f, v\rangle, \end{aligned}$$

where we used Proposition 6.6.2 in the third step above. Setting $v = u - \lambda\nabla^+\Phi'(\nabla^- u) - f$ we find that

$$\|u - \lambda\nabla^+\Phi'(\nabla^- u) - f\|^2 = 0$$

and so u satisfies (6.6.17).

The proof will be completed by showing that solutions of (6.6.17) are unique. Let $u, v \in L^2(\mathbb{Z}_n; \mathbb{R})$ be solutions of (6.6.17). Therefore

$$u - \lambda\nabla^+\Phi'(\nabla^- u) - (v - \lambda\nabla^+\Phi'(\nabla^- u)) = 0,$$

which can be simplified to read

$$u - v - \lambda\nabla^+ \left(\Phi'(\nabla^- u) - \Phi'(\nabla^- v)\right) = 0.$$

Now take the inner product with $u - v$ on both sides to obtain

$$\|u - v\|^2 - \lambda\langle \nabla^+ \left(\Phi'(\nabla^- u) - \Phi'(\nabla^- v)\right), u - v\rangle = 0.$$

In the second term, we use the integration by parts formula from Proposition 6.6.2(ii) to obtain

$$\|u - v\|^2 + \lambda\langle \Phi'(\nabla^- u) - \Phi'(\nabla^- v), \nabla^- u - \nabla^- v\rangle = 0.$$

By (6.6.13), and the assumption that $\lambda \geq 0$, we have

$$\lambda\langle \Phi'(\nabla^- u) - \Phi'(\nabla^- v), \nabla^- u - \nabla^- v\rangle \geq 0.$$

It follows that $\|u - v\|^2 = 0$, and so $u = v$, which completes the proof. □

We now turn to the problem of numerically computing the denoised signal by solving the Euler-Lagrange equation (6.6.17). This equation is nonlinear, and we can no longer simply take the DFT on both sides to obtain a solution formula, as we did in Section 6.6.1. Here, we use gradient descent to

solve (6.6.17) numerically. Recalling Remark 6.6.4 and examining the proof of Lemma 6.6.10 we see that the gradient of E_Φ is given by

$$\nabla E_\Phi(u) = u - \lambda \nabla^+ \Phi'(\nabla^- u) - f.$$

Gradient descent takes small steps in the direction of $-\nabla E_\Phi$ until convergence. We start with some initial guess for the denoised signal, u_0, and iterate

$$u_{j+1} = u_j - dt\,(u_j - \lambda \nabla^+ \Phi'(\nabla^- u_j) - f), \tag{6.6.18}$$

where $dt > 0$ is a time step. A good choice for the initial iterate u_0 is, for example, the solution of Tikhonov denoising given in Section 6.6.1.

The choice of the time step in important. If the time step is taken too large, then the iteration (6.6.18) is unstable and non-convergent, and if it is chosen too small then the convergence can take prohibitively long. To see how to choose an appropriate time step, we use a Von Neumann analysis, which uses the DFT to analyze stability of numerical schemes. The Von Neumann analysis applies only to linear equations, so it is common in practice (though not rigorous, see below) to approximate the equation by a similar linear equation. To see how to do this, note that the linear Tikhonov setting is $\Phi(t) = t^2$, where $\Phi'(t) = 2t$ and $\Phi''(t) = 2$. In general, we can make the approximation

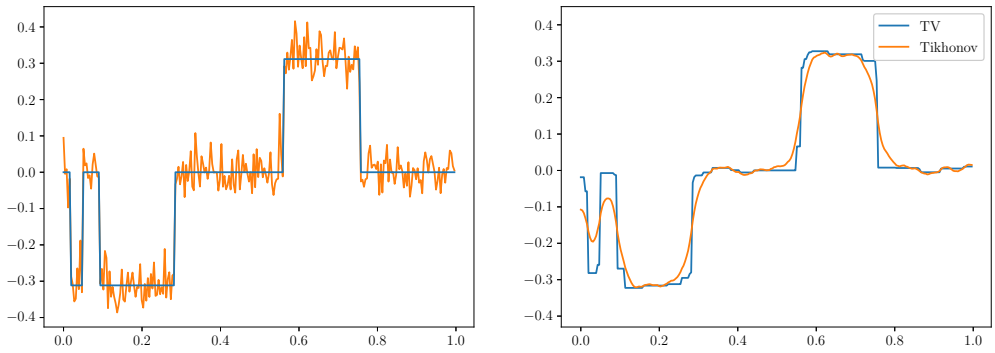
$$\begin{aligned}
\nabla^+ \Phi'(\nabla^- u)(k) &= \Phi'(\nabla^- u(k+1)) - \Phi'(\nabla^- u(k)) \\
&\approx \Phi''(\nabla^- u(k))(\nabla^- u(k+1) - \nabla^- u(k)) \\
&= \Phi''(\nabla^- u(k)) \nabla^+ \nabla^- u(k) \\
&= \Phi''(\nabla^- u(k)) \Delta u(k).
\end{aligned}$$

Thus, for the Von Neumann analysis we replace $\nabla^+ \Phi'(\nabla^- u)$ with $C_\Phi \Delta u$, where $C_\Phi = \max_{t \in \mathbb{R}} |\Phi''(t)|$. We also set $f = 0$ for simplicity; the scheme needs to be stable in this case, and it turns out nonzero choices for f do not affect stability considerations. This yields the simplified linear equation

$$u_{j+1} = u_j - dt\,(u_j - C_\Phi \lambda \Delta u_j).$$

We perform a Von Neumann analysis by taking the DFT of both sides and using Exercise 6.5.6 to obtain

$$\begin{aligned}
\mathcal{D}u_{j+1}(k) &= \mathcal{D}u_j(k) - dt\,(\mathcal{D}u_j(k) - 2C_\Phi \lambda (\cos(2\pi k/n) - 1)\mathcal{D}u_j(k)) \\
&= \underbrace{(1 - dt + 2C_\Phi \lambda dt\,(\cos(2\pi k/n) - 1))}_{\lambda_k} \mathcal{D}u_j(k).
\end{aligned}$$



(a) Noisy signal (b) Denoised

Figure 6.6.5: Total Variation (TV) denoising versus Tikhonov denoising on the piecewise constant signal from Figure 6.6.4. Total Variation denoising better preserves edges (e.g., sharp changes) in the signal, and is superior at reconstructing piecewise constant signals. We used $\lambda = 0.1$ and $\varepsilon = 10^{-5}$ for TV denoising and $\lambda = 20$ for Tikhonov.

Hence, the Fourier modes are simply being multiplied by λ_k at each iteration. In order to ensure stability, so the solution does not blow up, we need $|\lambda_k| \leq 1$. Using that $-1 \leq \cos(2\pi k/n) \leq 1$ we have

$$\lambda_k \leq 1 - dt \quad \text{and} \quad \lambda_k \geq 1 - dt(1 + 4C_\Phi \lambda).$$

Thus, to ensure $-1 \leq \lambda_k \leq 1$ for all k we require that $1 - dt(1 + 4C_\Phi \lambda) \geq -1$ or

$$dt \leq \frac{2}{1 + 4C_\Phi \lambda}. \tag{6.6.19}$$

In numerical analysis of partial differential equations, this time step restriction is called a Courant-Friedrichs-Lewy, or CFL, condition.

Figure 6.6.5 shows the solution of Total Variation (TV) denoising compared to Tikhonov denoising. We chose $\lambda = 0.1$ for TV denoising and $\lambda = 20$ for Tikhonov denoising. These were chosen to yield similar amounts of denoising in the constant regions of the signal. As discussed in Remark 6.6.8, we approximate the function $\Phi(t) = |t|$ for TV denoising with the smooth approximation $\Phi_\varepsilon(t) = \sqrt{t^2 + \varepsilon^2}$. In this case

$$\Phi_\varepsilon'(t) = \frac{t}{\sqrt{t^2 + \varepsilon^2}}$$

and $0 \leq \Phi_\varepsilon''(t) \leq \frac{1}{\varepsilon}$. Therefore $C_\Phi = \frac{1}{\varepsilon}$, and the time step restriction from the Von Neumann analysis yields

$$dt \leq \frac{2}{1 + 4C_\Phi \lambda} = \frac{2\varepsilon}{\varepsilon + 4\lambda}.$$

In the experiment we chose $\varepsilon = 10^{-5}$ and ran the gradient descent iterations (6.6.18) until u_j satisfied (6.6.17) up to an error tolerance of 10^{-3}, which took around 70,000 iterations. We can see that choosing $\varepsilon > 0$ is necessary to obtain a numerically stable computational method (though see Remark 6.6.13 below). There are other methods for minimizing energies with Total Variation regularizers that do not need to make this smooth approximation, for example, primal dual methods [4] and the Split-Bregman approach [10].

It is important to point out that the Von Neumann stability analysis for nonlinear equations like (6.6.18) is not mathematically rigorous. We made a heuristic simplification to obtain a linear equation, and to make this rigorous we would have to relate the solutions of the linearized equation to the original nonlinear equation (6.6.18) in some way, which is generally difficult. Nevertheless, it is very common to perform these heuristic stability analyses, since they usually give the correct CFL time step stability condition, and it is very easy to check in practice if the time step restriction is correct, and if it is tight.

In many cases we can say a great deal more if we analyze the gradient descent equation (6.6.18) using techniques from nonlinear and convex analysis. In Theorem 6.6.11 below, we give a rigorous proof of convergence of the gradient descent scheme (6.6.18) under a similar, but more restrictive, time step condition.

Theorem 6.6.11. *Let $f \in L^2(\mathbb{Z}_n; \mathbb{R})$ and $\lambda \geq 0$. Let u_j be the iterations of the gradient descent scheme (6.6.18) and let u be the solution of (6.6.17). Assume that the time step dt in (6.6.18) satisfies*

$$dt < \frac{2}{1 + 16C_\Phi^2 \lambda^2}. \tag{6.6.20}$$

Then u_j converges to u as $j \to \infty$, and the difference $u_j - u$ satisfies

$$\|u_{j+1} - u\|^2 \leq \mu \|u_j - u\|^2 \tag{6.6.21}$$

where

$$\mu := (1 - dt)^2 + 16C_\Phi^2 dt^2 \lambda^2 < 1. \tag{6.6.22}$$

Remark 6.6.12. Theorem 6.6.11 shows that whenever the time step restriction (6.6.20) holds, gradient descent (6.6.18) converges at the linear rate μ to the solution u of (6.6.17). The time step restriction is not strict, and gradient descent may converge for larger time steps, and may exhibit a better convergence rate in practice. Indeed, one often uses the less restrictive Von Neumann time step (6.6.19) in practice.

Proof. We note that the gradient descent equation (6.6.18) can be written as

$$u_{j+1} = (1 - dt)u_j + dt\lambda\nabla^+\Phi'(\nabla^- u_j) + dtf.$$

The solution u of (6.6.17) satisfies

$$u = (1 - dt)u + dt\lambda\nabla^+\Phi'(\nabla^- u) + dtf.$$

Subtracting these equations and writing $e_j = u_j - u$ we obtain

$$e_{j+1} = (1 - dt)e_j + dt\lambda\nabla^+ \left(\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\right).$$

We take the squared norm on both sides to obtain

$$\begin{aligned}
\|e_{j+1}\|^2 &= \|(1 - dt)e_j + dt\lambda\nabla^+ \left(\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\right)\|^2 \\
&= (1 - dt)^2\|e_j\|^2 + dt^2\lambda^2\|\nabla^+(\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u))\|^2 \\
&\qquad + 2dt(1 - dt)\lambda\langle\nabla^+ \left(\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\right), u_j - u\rangle \\
&= (1 - dt)^2\|e_j\|^2 + dt^2\lambda^2\|\nabla^+(\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u))\|^2 \\
&\qquad - 2dt(1 - dt)\lambda\langle\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u), \nabla^- u_j - \nabla^- u\rangle,
\end{aligned}$$

where we used the integration by parts formula from Proposition 6.6.2(i) in the last line. By (6.6.13), the last term is negative or zero, so we can drop it to obtain the inequality

$$\|e_{j+1}\|^2 \leq (1 - dt)^2\|e_j\|^2 + dt^2\lambda^2\|\nabla^+(\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u))\|^2. \tag{6.6.23}$$

We now note that for any $f \in L^2(\mathbb{Z}_n; \mathbb{R})$ we have

$$\|\nabla^+ f\|^2 \leq 4\|f\|^2.$$

To see this, we use the inequality $2ab \leq a^2 + b^2$ (obtained by expanding

$(a-b)^2 \geq 0$) to compute

$$\begin{aligned}\|\nabla^+ f\|^2 &= \sum_{k=0}^{n-1}(f(k+1)-f(k))^2 \\ &= \sum_{k=0}^{n-1}(f(k+1)^2 - 2f(k+1)f(k) + f(k)^2) \\ &\leq 2\sum_{k=0}^{n-1}(f(k+1)^2 + f(k)^2) \\ &\leq 2(\|f\|^2 + \|f\|^2) = 4\|f\|^2.\end{aligned}$$

Likewise we have $\|\nabla^- f\|^2 \leq 4\|f\|^2$. Using this estimate in (6.6.23) we have

$$\|e_{j+1}\|^2 \leq (1-dt)^2\|e_j\|^2 + 4dt^2\lambda^2\|\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\|^2 \tag{6.6.24}$$

Since

$$\Phi'(t) - \Phi'(s) = \int_s^t \Phi''(\tau)\,d\tau \leq C_\Phi(t-s)$$

for $t \geq s$ (recall $C_\Phi = \max \Phi''$), we have

$$|\Phi'(t) - \Phi'(s)| \leq C_\Phi|t-s|$$

for any $t, s \in \mathbb{R}$. Therefore

$$\|\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\|^2 \leq \|C_\Phi|\nabla^- u_j - \nabla^- u|)\|^2 = C_\Phi^2\|\nabla^- e_j\|^2 \leq 4C_\Phi\|e_j\|^2.$$

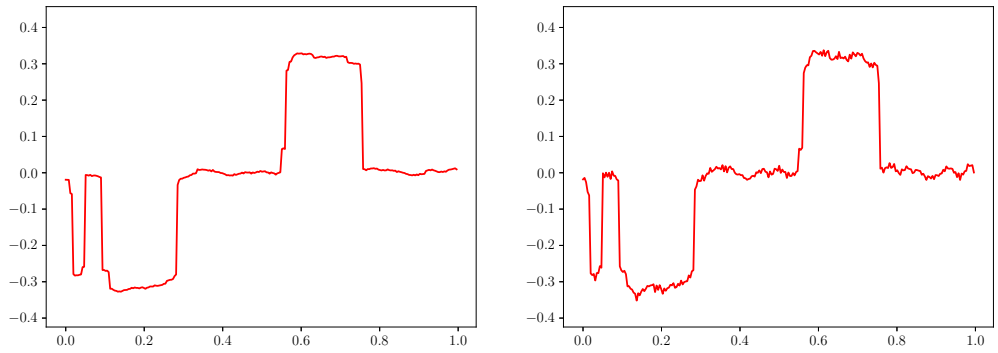
Inserting this into (6.6.24) we have

$$\begin{aligned}\|e_{j+1}\|^2 &\leq (1-dt)^2\|e_j\|^2 + 16C_\Phi^2 dt^2\lambda^2\|e_j\|^2 \\ &\leq \left((1-dt)^2 + 16C_\Phi^2 dt^2\lambda^2\right)\|e_j\|^2.\end{aligned}$$

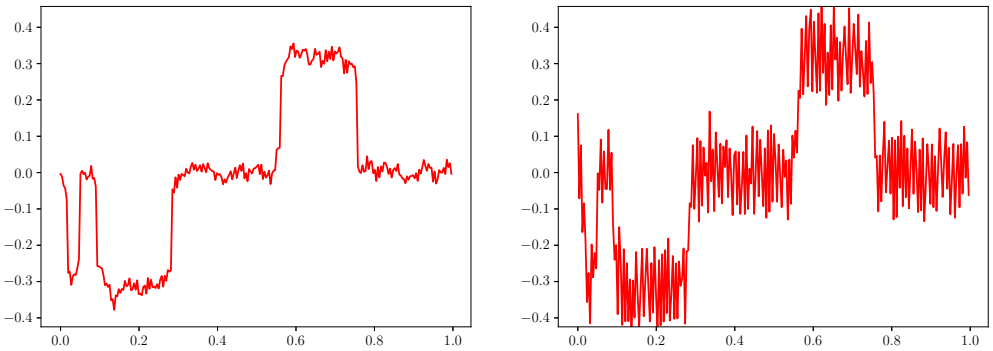
Thus, gradient descent converges when

$$\mu := (1-dt)^2 + 16C_\Phi^2 dt^2\lambda^2 < 1,$$

Since in this case we have $\|e_j\|^2 \leq \mu^j\|e_0\|^2 \to 0$ as $j \to \infty$. The proof is completed by checking that $\mu < 1$ whenever □



(a) $dt = 0.01$ (b) $dt = 0.05$



(c) $dt = 0.1$ (d) $dt = 0.5$

Figure 6.6.6: Example of the steady state obtained by the TV gradient descent (6.6.18) with $\varepsilon = 10^{-10}$ and various time steps dt, all of which are above the CFL stability condition, which is $dt \sim 5 \times 10^{-10}$. There are nonlinear effects that lead to a weaker (non-convergent) stability at much larger time steps that is still useful for denoising. For example, at $dt = 0.01$ the result in the figure took only 200 iterations to obtain, compared to the result in Figure 6.6.5, which took about 70,000 iterations at $\varepsilon = 0.01$.

Remark 6.6.13. It turns out it is possible to set $\varepsilon = 0$, i.e., use the true TV regularizer $\Phi(t) = |t|$, and still obtain a stable numerical method that results in good denoising results (and converges rapidly). In Figure 6.6.6 we show the denoising results with various values of dt and $\varepsilon = 10^{-10}$. In this case the CFL condition gives a stability condition of $dt \leq 5 \times 10^{-10}$, and we consider much larger time steps of $dt = 0.01, 0.05, 0.1, 0.5$. In each case the gradient descent does not converge, but it does settle down to a steady state solution with some persisting oscillations. For $dt = 0.01$ the oscillations are quite small and could

be acceptable for a denoising result.

This is a remarkable phenomenon that does not hold for linear equations, where the Von Neumann analysis gives sharp CFL time step stability conditions that cannot be violated without finite time blow up of the solution. Here, there are nonlinear effects that prevent oscillations from growing beyond a certain amplitude. Even for larger time steps dt, the gradient descent iterations never grow unbounded, and instead the oscillations in the signal grow to a certain amplitude, dependent on the choice of dt, and then they remain that large and continue to oscillate. The scheme is nonconvergent, but if the oscillations are sufficiently small, they would be within the quantization error from saving a signal or image as 8-bit or 16-bit integers, which is commonly done in practice.

While there is not a rigorous mathematical understanding of this phenomenon, we can perform a rough heuristic argument that gives some good insight. Essentially, in the CFL condition (6.6.19) we took a very loose upper bound by setting $C_\Phi = \max \Phi''$. Recall we used the approximation

$$\nabla^+ \Phi'(\nabla^- u)(k) \approx \Phi''(\nabla^- u(k)) \Delta u(k)$$

to obtain a linear equation, but we replaced Φ'' by C_Φ. Instead, we can leave Φ'' and treat the equation locally, so the equivalent linear equation becomes

$$u_{j+1} = u_j - dt\,(u_j - \lambda \Phi''(\nabla^- u) \Delta u_j).$$

We treat this equation locally, so Φ'' is roughly constant, the CFL condition (6.6.19) becomes

$$dt \leq \frac{2}{1 + 4\Phi''(\nabla^- u)\lambda}. \tag{6.6.25}$$

Now, we flip the CFL condition around and write it as a condition on Φ'', in the form

$$\Phi''(\nabla^- u) \leq \frac{1}{4\lambda} \left(\frac{2}{dt} - 1 \right). \tag{6.6.26}$$

From this, we can see that if we pick a time step dt, then the scheme will be stable locally wherever the gradient $\nabla^- u$ satisfies (6.6.26). Hence, some parts of the solution will be stable, while others will be unstable and have growing oscillations. But as the oscillations grow, the CFL condition (6.6.26) changes, leading to a nonlinear interplay between the gradient and time step stability condition.

For example, with the TV regularizer with $\varepsilon > 0$, i.e., $\Phi(t) = \sqrt{t^2 + \varepsilon^2}$, we have

$$\Phi''(t) = \frac{\varepsilon^2}{(t^2 + \varepsilon^2)^{3/2}} \leq \frac{\varepsilon^2}{t^3}.$$

Therefore, the condition (6.6.26) holds whenever

$$\frac{\varepsilon^2}{|\nabla^- u|^3} \leq \frac{1}{4\lambda}\left(\frac{2}{dt} - 1\right),$$

which can be rewritten as

$$|\nabla^- u|^3 \geq \frac{4\lambda\varepsilon^2 dt}{2 - dt}. \tag{6.6.27}$$

Hence, oscillations will grow until the gradient $\nabla^- u$ is large enough so that (6.6.27) holds, and then the scheme becomes stable and those oscillations will stop growing. This gives a heuristic explanation for Figure 6.6.6. For more details on this phenomenon, we refer the reader to [2], where this phenomenon is described in more detail.

Exercise 6.6.14. In the proof of Theorem 6.6.11 we only used that Φ is convex, namely $\Phi'' \geq 0$. In general, Φ is often strongly convex, which means $\Phi'' \geq c_\Phi$ for some positive constant $c_\Phi > 0$. In this exercise we assume Φ is strongly convex with constant c_Φ.

(i) Show that for every $t, s \in \mathbb{R}$ we have

$$(\Phi'(t) - \Phi'(s))(t - s) \geq c_\Phi (t - s)^2.$$

Use this to show that (in the context of the proof of Theorem 6.6.11) that

$$\langle \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u), \nabla^- u_j - \nabla^- u \rangle \geq c_\Phi \|\nabla^- e_j\|^2.$$

(ii) Modify the proof of Theorem 6.6.11 using Part (i) instead of (6.6.13) to show that

$$\|u_{j+1} - u\|^2 \leq (1 - dt)^2 \|u_j - u\|^2$$

provided the time step is restricted so that

$$dt \leq \frac{c_\Phi}{c_\Phi + 2C_\Phi \lambda}.$$

(iii) Show that if $C_\Phi = c_\Phi$ (which means that $\Phi(t) = \frac{1}{2}C_\Phi t^2$) then the time step restriction becomes

$$dt \leq \frac{1}{1 + 2C_\Phi \lambda},$$

which is only slightly more restrictive than the Von Neumann condition (6.6.19).

△

6.7 Multi-dimensional DFT

We now briefly study the Discrete Fourier Transform (DFT) in higher dimensions. Let

$$\mathbb{Z}_n^d = \underbrace{\mathbb{Z}_n \times \mathbb{Z}_n \times \cdots \times \mathbb{Z}_n}_{d \text{ times},}$$

where $d \geq 1$. Elements of $\mathbb{Z}_n^d$ are denoted by the same letters, i.e., k, ℓ, as elements of $\mathbb{Z}_n$. In this case $k \in \mathbb{Z}_n^d$ has d components $k = (k(1), k(2), \ldots, k(d))$. We denote the dot product of $k, \ell \in \mathbb{Z}_n^d$ by

$$k \cdot \ell = \sum_{j=1}^{d} k(j)\ell(j).$$

We denote by $L^2(\mathbb{Z}_n^d)$ the space of function $f : \mathbb{Z}_n^d \to \mathbb{C}$ equipped with the inner product

$$\langle f, g \rangle = \sum_{k \in \mathbb{Z}_n^d} f(k)\overline{g(k)}.$$

We also have the induced norm $\|f\|^2 = \langle f, f \rangle$. The *discrete cyclic convolution* of $f, g \in L^2(\mathbb{Z}_n^d)$ is given by

$$(f * g)(k) = \sum_{\ell \in \mathbb{Z}_n^d} f(\ell)g(k - \ell).$$

Note that the sums above are short-hand for d sums, and we have

$$\sum_{k \in \mathbb{Z}_n^d} = \sum_{k(1)=0}^{n-1} \sum_{k(2)=0}^{n-1} \cdots \sum_{k(d)=1}^{n-1}.$$

Definition 6.7.1. The *(multi-dimensional) Discrete Fourier Transform (DFT)* is the mapping $\mathcal{D} : L^2(\mathbb{Z}_n^d) \to L^2(\mathbb{Z}_n^d)$ given by

$$\mathcal{D}f(k) = \sum_{\ell \in \mathbb{Z}_n^d} f(\ell) e^{-2\pi i k \cdot \ell / n}. \tag{6.7.1}$$

The *(multi-dimensional) Inverse Discrete Fourier Transform (IDFT)* is the mapping $\mathcal{D}^{-1} : L^2(\mathbb{Z}_n^d) \to L^2(\mathbb{Z}_n^d)$ given by

$$\mathcal{D}^{-1}f(\ell) = \frac{1}{n^d} \sum_{k \in \mathbb{Z}_n^d} f(k) e^{2\pi i k \cdot \ell / n}. \tag{6.7.2}$$

We note that the same symbols $\mathcal{D}$ and $\mathcal{D}^{-1}$ are used for the one dimensional and higher dimensional DFTs. The value of d will normally be clear from context.

It is important to point out that the multi-dimensional DFT can be viewed as applying d one dimensional DFTs to the individual coordinates. Indeed, we consider the case of $d = 2$ where we can write

$$\begin{aligned}
\mathcal{D}f(k) &= \mathcal{D}f(k(1), k(2)) \\
&= \sum_{\ell(1)=0}^{n-1} \sum_{\ell(2)=0}^{n-1} f(\ell(1), \ell(2)) e^{-2\pi i (k(1)\ell(1) + k(2)\ell(2))/n} \\
&= \sum_{\ell(1)=0}^{n-1} e^{-2\pi i k(1)\ell(1)/n} \left(\sum_{\ell(2)=0}^{n-1} f(\ell(1), \ell(2)) e^{-2\pi i k(2)\ell(2)/n} \right).
\end{aligned}$$

The term in brackets above is the one dimensional DFT of f in the second coordinate $\ell(2)$, and the outer sum computes the one dimensional DFT of the result in the first coordinate. A similar observation can be made for the case of $d \geq 3$. This means most properties of the one dimensional DFT carry over to the multi-dimensional case, and the FFT can be directly used to efficiently compute multi-dimensional DFTs.

We will briefly review the important properties of the multi-dimensional DFT. The proofs are very similar to the one dimensional setting, so we leave them to exercises.

Theorem 6.7.2. *For every $f \in L^2(\mathbb{Z}_n^d)$ we have $f = \mathcal{D}\mathcal{D}^{-1}f = \mathcal{D}^{-1}\mathcal{D}f$. Furthermore, the following properties hold for each $f, g \in L^2(\mathbb{Z}_n^d)$.*

(i) $\langle f, g \rangle = \frac{1}{n^d} \langle \mathcal{D}f, \mathcal{D}g \rangle$,

(ii) $\|f\|^2 = \frac{1}{n^d}\|\mathcal{D}f\|^2$,

(iii) $\mathcal{D}(f * g) = \mathcal{D}f \cdot \mathcal{D}g$.

Exercise 6.7.3. Prove Theorem 6.7.2. △

The discrete derivatives introduced in Section 6.6.1 can be extended to multi-dimensional versions. Let $e_1, e_2, \ldots, e_d \in \mathbb{R}^d$ be the standard basis vectors in $\mathbb{R}^d$. We define the forward difference in the j^{th} direction, $\nabla_j^+ : L^2(\mathbb{Z}_n^d) \to L^2(\mathbb{Z}_n^d)$, by

$$\nabla_j^+ u(k) = u(k + e_j) - u(k).$$

Similarly, the backward difference ∇_j^- by

$$\nabla_j^- u(k) = u(k) - u(k - e_j).$$

The discrete Laplacian Δ is defined by

$$\Delta u = \sum_{j=1}^{d} \nabla_j^+ \nabla_j^- u.$$

The analogous result to Proposition 6.6.2, concerning discrete integration by parts, holds in the multi-dimensional setting.

Proposition 6.7.4. *For all $u, v \in L^2(\mathbb{Z}_n^d)$ and $j = 1, 2, \ldots, d$, the following hold.*

(i) $\langle \nabla_j^- u, v\rangle = -\langle u, \nabla_j^+ v\rangle$

(ii) $\langle \nabla_j^+ u, v\rangle = -\langle u, \nabla_j^- v\rangle$

(iii) $\langle \Delta u, v\rangle = \langle u, \Delta v\rangle$

Exercise 6.7.5. Prove Proposition 6.7.4. △

The DFTs of the multi-dimensional gradients and Laplacian are similar to their one dimensional counterparts. We leave the results to an exercise, which is the generalization of Exercise 6.5.6.

Exercise 6.7.6. Complete the following exercises.

(i) Show that $\mathcal{D}(\nabla_j^- f)(k) = (1 - \omega^{-k(j)})\mathcal{D}f(k)$, where $\omega = e^{2\pi i/n}$.

(ii) Show that $\mathcal{D}(\nabla_j^+ f)(k) = (\omega^{k(j)-1)})\mathcal{D}f(k)$.

(iii) Show that

$$\mathcal{D}(\Delta f)(k) = 2\mathcal{D}f(k) \sum_{j=1}^{d} (\cos(2\pi k(j)/n) - 1).$$ △

6.7.1 Application: Image denoising

Python Notebook: .ipynb

We now consider an application of the multi-dimensional DFT to image denoising. This section follows closely Sections 6.6.1 and 6.6.2, so we only sketch the results here, leaving some of the proofs to exercises. While most images are in $d = 2$ or $d = 3$ dimensions, we proceed in generality here. We consider the general regularized variational approach to denoising, as in Section 6.6.2, where we minimize the function

$$E_\Phi(u) = \frac{1}{2}\|u - f\|^2 + \lambda \sum_{j=1}^{d} \|\Phi(\nabla_j^- u)\|_1, \tag{6.7.3}$$

where $f \in L^2(\mathbb{Z}_n^d)$ is the noisy image and the denoised image is the minimizer u. The function $\Phi : \mathbb{R} \to \mathbb{R}$ is the regularizer, which we recall from Section 6.6.2 is assumed to be twice continuously differentiable, convex, and satisfies $\Phi(0) = 0$. We also recall that since these conditions imply Φ is nonnegative, we have

$$\|\Phi(\nabla_j^- u)\|_1 = \sum_{k \in \mathbb{Z}_n^d} \Phi(\nabla_j^- u(k))).$$

The choice of $\Phi(t) = \frac{1}{2}t^2$ leads to Tikhonov image denoising, while $\Phi(t) = |t|$ (or the regularized $\Phi(t) = \sqrt{t^2 + \varepsilon^2}$) leads to Total Variation (TV) regularization.

To compute the gradient of E_Φ, we follow the notation in Remark 6.6.4 and the proof of Lemma 6.6.10 to obtain

$$\frac{d}{dt}\Big|_{t=0} E_\Phi(u + tv) = \langle u - f, v\rangle + \lambda \sum_{j=1}^{d} \langle \Phi'(\nabla_j^- u), \nabla_j^- v\rangle, \tag{6.7.4}$$

for any $v \in L^2(\mathbb{Z}_n^d)$.

Exercise 6.7.7. Prove that (6.7.4) holds. △

We use Proposition 6.7.4 to obtain

$$\frac{d}{dt}\Big|_{t=0} E_\Phi(u + tv) = \langle u - f, v\rangle - \lambda \sum_{j=1}^{d} \langle \nabla_j^+ \Phi'(\nabla_j^- u), v\rangle = \langle \nabla E_\Phi(u), v\rangle,$$

where

$$\nabla E_\Phi(u) = u - f - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u).$$

Therefore, the Euler-Lagrange equation $\nabla E_\Phi(u) = 0$ becomes

$$u - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u) = f. \tag{6.7.5}$$

In the case of Tikhonov regularization, where $\Phi(t) = \frac{1}{2}t^2$, the equation becomes linear and we have

$$u - \lambda \Delta u = f. \tag{6.7.6}$$

For Tikhonov regularization, we solve the denoising equation (6.7.6) by taking the DFT on both sides and using Exercise 6.7.6 to obtain

$$\left(1 - 2\lambda \sum_{j=1}^{d} (\cos(2\pi k(j)/n) - 1)\right) \mathcal{D}u(k) = \mathcal{D}f(k).$$

Solving for u we obtain

$$u = \mathcal{D}^{-1}(G_\lambda \cdot \mathcal{D}f) = g_\lambda * f, \tag{6.7.7}$$

where

$$G_\lambda(k) = \left(1 - 2\lambda \sum_{j=1}^{d} (\cos(2\pi k(j)/n) - 1)\right)^{-1}$$

and $G_\lambda = \mathcal{D}g_\lambda$.

For general nonlinear Φ, we solve (6.7.5) with gradient descent, which iterates

$$u_{j+1} = u_j - dt \left(u_j - \lambda \sum_{m=1}^{d} \nabla_m^+ \Phi'(\nabla_m^- u_j) - f\right). \tag{6.7.8}$$

To determine the CFL stability condition for the time step dt, we follow a Von Neumann analysis for the simplified linear equation

$$u_{j+1} = u_j - dt \left(u_j - \lambda C_\Phi \sum_{m=1}^{d} \nabla_m^+ \nabla_m^- u_j - f\right),$$

which is equivalent, after dropping the forcing term f, to

$$u_{j+1} = (1 - dt)u_j + dt C_\Phi \lambda \Delta u_j. \tag{6.7.9}$$

Taking the multi-dimensional DFT on both sides and using Exercise 6.7.6 we find that

$$\mathcal{D}u_{j+1}(k) = \underbrace{\left(1 - dt + 2dtC_\Phi\lambda \sum_{m=1}^{d} (\cos(2\pi k(m)/n) - 1)\right)}_{\lambda_k} \mathcal{D}u_j(k).$$

We need $|\lambda_k| \leq 1$ for all $k \in \mathbb{Z}_n^d$ to ensure stability. Note that

$$1 - dt(1 + 4dC_\Phi\lambda) \leq \lambda_k \leq 1.$$

Therefore, to ensure $\lambda_k \geq -1$ we obtain the CFL condition

$$dt \leq \frac{2}{1 + 4dC_\Phi\lambda}. \tag{6.7.10}$$

Figure 6.7.1 compares Tikhonov regularization to Total Variation regularization. Notice the edges are better preserved in TV denoising. Figure 6.7.2 shows the results of TV denoising with larger values for λ, which favor the regularizer more heavily and lead to smoother images.

Exercise 6.7.8. Formulate and prove the analogous results to Lemma 6.6.9, Lemma 6.6.10, and Theorem 6.6.11 in the multi-dimensional setting. What is the corresponding time step restriction, analogous to (6.6.20)? △

Project 6.7.1 (TV Image Inpainting)**. Python Notebook:** .ipynb

A common problem in image processing is to fill in missing pieces of an image, which may have been corrupted by writing on an image of damage to a painting, for example. It may also be desirable to remove an object from an image, which can be treated by the same methods. The process of filling in missing parts of an image is called image inpainting. Figure 6.7.3 shows an example of part of the cameraman image that has been corrupted by writing "Math 5467" on the image, along with the result of TV regularized inpainting (from this project) to fill in the missing parts of the image.

For image inpainting, we assume we know the region of the image that is corrupted and needs to be inpainted. Let $f \in L^2(\mathbb{Z}_n^d)$ denote the corrupted image, and let $\Lambda \subset \mathbb{Z}_n^d$ denote the set of uncorrupted (i.e., good) pixels. The TV regularized inpainting problem is to minimize

$$E_\Phi(u) = \frac{1}{2}\|\delta_\Lambda \cdot (u - f)\|^2 + \lambda \sum_{j=1}^{d} \|\Phi(\nabla_j^- u)\|_1, \tag{6.7.11}$$

(a) Cameraman (b) Noisy

(c) Tikhonov ($\lambda = 5$) (d) TV ($\lambda = 0.1$)

Figure 6.7.1: A comparison of Tikhonov and Total Variation regularized image denoising on the cameraman image.

where $\delta_\Lambda(k) = 1$ if $k \in \Lambda$ and $\delta_\Lambda(k) = 0$ otherwise. Essentially we treat inpainting as a denoising problem where the image f is known on only part of the domain.

Complete the following steps for this project, using the Python notebook linked above.

(a) TV ($\lambda = 0.5$) (b) TV ($\lambda = 1$)

Figure 6.7.2: Examples of TV denoising with larger values of λ.

(a) Corrupted image (b) TV inpainting

Figure 6.7.3: Example of a corrupted image and the result of TV inpainting.

(i) Show that

$$\nabla E_\Phi(u) = \delta_\Lambda \cdot (u - f) - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u).$$

(ii) Use gradient descent to minimize the TV inpainting functional to inpaint the corrupted image in Figure 6.7.3. Note that the CFL condition on the time step dt in gradient descent is the same as in TV denoising.

△

Project 6.7.2 (TV Image Deblurring)**. Python Notebook:** .ipynb

Another common task in image processing is image deblurring (i.e., deconvolution). An image blur is the convolution of an image f with a blurring kernel g,

$$f_{blur} = g * f.$$

(a) Cameraman (b) Out-of-focus blur

Figure 6.7.4: An example of an out-of-focus blurring of the cameraman image.

For example, an out-of-focus blur, the blurring kernel g is the characteristic function of a disk,

$$g(k) = \begin{cases} 1, & \text{if } \|k\| \leq r \\ 0, & \text{otherwise.} \end{cases}$$

The radius r of the disk controls the amount of blur. Any convolution-based blurring operation can be computed with a DFT using the convolution property, as

$$f_{blur} = \mathcal{D}^{-1}(G \cdot \mathcal{D}f), \tag{6.7.12}$$

where $G = \mathcal{D}g$ is the filter.

Complete the following steps for this project, using the Python notebook linked above.

(i) Try a naive deblurring based on inverting the filter G. That is, try computing

$$f_{deblurred} = \mathcal{D}^{-1}(G^{-1} \cdot \mathcal{D}f_{blur}).$$

(ii) You should find the naive deblurring works well in a perfectly noise-free setting. Try adding noise after blurring, in the form

$$f_{blur+noise} = f_{blur} + \eta$$

where η is independent Gaussian noise (each $\eta(k)$ is an Gaussian random variable with mean zero and standard deviation $\sigma > 0$). Then try the naive deblurring in this noisy setting, which is

$$f_{deblurred} = \mathcal{D}^{-1}(G^{-1} \cdot \mathcal{D} f_{blur+noise}).$$

What do you see? While the blurring operator is technically invertible, it has vanishingly small eigenvalues and taking the inverse is not stable in the presence of noise. Noise is always present in signal acquisition, and it is impossible to work in noise-free settings, like in Part (i), in practice.

(iii) In order to deblur in the presence of noise, we use Total Variation regularization, which minimizes the function

$$E_\Phi(u) = \frac{1}{2}\|g * u - f\|^2 + \lambda \sum_{j=1}^{d} \|\Phi(\nabla_j^- u)\|_1,$$

where $f = f_{blur+noise}$. Show that

$$\nabla E_\Phi(u) = g^T * (g * u - f) - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u),$$

where $g^T := \mathcal{D}^{-1}\overline{G}$. [Hint: Take a variation, and then use Parseval's identity to simplify.]

(iv) Use gradient descent to minimize the TV regularized deblurring energy E_Φ and deblur the image. Note that the CFL time step restriction is the same as in TV denoising.

△

6.8 The Discrete Cosine and Sine Transforms

It is often useful in practical applications to avoid complex numbers and work with real-valued transformations. Indeed, many embedded systems do not have support for complex arithmetic, and due to the conjugate symmetry

$$\mathcal{D} f(\ell) = \overline{\mathcal{D} f(n - \ell)}$$

for real valued signals f, the FFT computes twice as many coefficients are are needed for real-valued signals, which is wasteful.

To motivate the Discrete Cosine and Sine Transforms, we recall the Fourier Inversion Theorem 6.2.4 yields

$$f(k) = \frac{1}{n}\sum_{\ell=0}^{n-1} \mathcal{D}f(\ell) e^{2\pi i k\ell/n}. \tag{6.8.1}$$

Throughout this section we assume that f is real-valued, that is $f(k) \in \mathbb{R}$ for all k. In this case, the imaginary part of the right hand side above must be identically zero. The remaining real part of the right hand side will give us a representation formula for f in terms of only real-value quantities, and will lead us to the Discrete Cosine and Sine Transforms.

To do this, we substitute the definition of $\mathcal{D}f(\ell)$ into (6.8.1) to obtain

$$f(k) = \frac{1}{n}\sum_{\ell=0}^{n-1}\sum_{j=0}^{n-1} f(j) e^{-2\pi i j\ell/n} e^{2\pi i k\ell/n}. \tag{6.8.2}$$

We use Euler's Formula to simplify the complex exponentials as follows

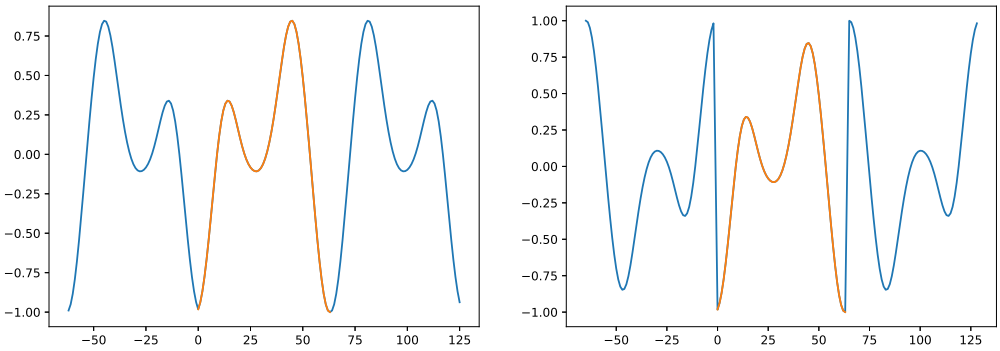
$$\begin{aligned} e^{-2\pi i j\ell/n} e^{2\pi i k\ell/n} &= (\cos(2\pi j\ell/n) - i\sin(2\pi j\ell/n))(\cos(2\pi k\ell/n) + i\sin(2\pi k\ell/n)) \\ &= (\cos(2\pi j\ell/n)\cos(2\pi k\ell/n) + \sin(2\pi j\ell/n)\sin(2\pi k\ell/n)) \\ &\quad + i\left(\cos(2\pi j\ell/n)\sin(2\pi k\ell/n) - \sin(2\pi j\ell/n)\cos(2\pi k\ell/n)\right). \end{aligned}$$

Since $f(k)$ is real-valued, we can ignore the imaginary part above (it must vanish when summed in (6.8.2)), and substitute the real part into (6.8.2) to obtain

$$\begin{aligned} f(k) = \frac{1}{n}\sum_{\ell=0}^{n-1}\left(\sum_{j=0}^{n-1} f(j)\cos(2\pi j\ell/n)\right)\cos(2\pi k\ell/n) \\ + \frac{1}{n}\sum_{\ell=0}^{n-1}\left(\sum_{j=0}^{n-1} f(j)\sin(2\pi j\ell/n)\right)\sin(2\pi k\ell/n). \end{aligned} \tag{6.8.3}$$

This gives a representation of f in a basis of real-valued cos and sin functions. To derive the Discrete Cosine Transform from this, we play a trick to remove the sin terms. The trick is to take an even extension of the function f to a domain that is (nearly) twice as large. That is, starting with $f : \mathbb{Z}_n \to \mathbb{R}$, we define the even extension $f_e : \mathbb{Z}_{2(n-1)} \to \mathbb{R}$ by

$$f_e(k) = \begin{cases} f(k), & \text{if } 0 \le k \le n-1, \\ f(2(n-1)-k), & \text{if } n \le k \le 2(n-1)-1. \end{cases} \tag{6.8.4}$$



(a) Even extension (b) Odd extension

Figure 6.8.1: Example of the even and odd extensions of a signal on $\mathbb{Z}_{64}$

The reason for using $\mathbb{Z}_{2(n-1)}$ instead of $\mathbb{Z}_{2n}$ is that we do not wish to duplicate the first and last points $f(0)$ and $f(n-1)$ when reflecting evenly. Figure 6.8.1 shows an example of the even extension f_e of a signal $f \in L^2(\mathbb{Z}_{64})$. The figure also shows the odd extension, which is defined in Exercise 6.8.2.

We now apply the representation formula (6.8.3) to the even extension f_e, taking $2(n-1)$ in place of n, to obtain

$$f_e(k) = \frac{1}{2(n-1)} \sum_{\ell=0}^{2(n-1)-1} A_\ell \cos\left(\frac{\pi k \ell}{n-1}\right) + \frac{1}{2(n-1)} \sum_{\ell=0}^{2(n-1)-1} B_\ell \sin\left(\frac{\pi k \ell}{n-1}\right), \tag{6.8.5}$$

where

$$A_\ell = \sum_{j=0}^{2(n-1)-1} f_e(j) \cos\left(\frac{\pi j \ell}{n-1}\right) \quad \text{and} \quad B_\ell = \sum_{j=0}^{2(n-1)-1} f_e(j) \sin\left(\frac{\pi j \ell}{n-1}\right). \tag{6.8.6}$$

Since we took the even extension of f, it turns out that $B_\ell = 0$. Indeed, we

compute

$$\begin{aligned} B_\ell &= \sum_{j=0}^{2(n-1)-1} f_e(j) \sin\left(\frac{\pi j \ell}{n-1}\right) \\ &= \sum_{j=0}^{n-1} f_e(j) \sin\left(\frac{\pi j \ell}{n-1}\right) + \sum_{j=n}^{2(n-1)-1} f_e(j) \sin\left(\frac{\pi j \ell}{n-1}\right) \\ &= \sum_{j=0}^{n-1} f(j) \sin\left(\frac{\pi j \ell}{n-1}\right) + \sum_{j=n}^{2(n-1)-1} f(2(n-1)-j) \sin\left(\frac{\pi j \ell}{n-1}\right) \\ &= \sum_{j=0}^{n-1} f(j) \sin\left(\frac{\pi j \ell}{n-1}\right) + \sum_{k=1}^{n-2} f(k) \sin\left(\frac{\pi(2(n-1)-k)\ell}{n-1}\right), \end{aligned}$$

where we made the change of variables $k = 2(n-1) - j$ in the last line. We now note that

$$\sin\left(\frac{\pi(2(n-1)-k)\ell}{n-1}\right) = \sin\left(2\pi\ell - \frac{\pi k \ell}{n-1}\right) = -\sin\left(\frac{\pi k \ell}{n-1}\right).$$

Substituting this above we obtain

$$\begin{aligned} B_\ell &= \sum_{j=0}^{n-1} f(j) \sin\left(\frac{\pi j \ell}{n-1}\right) - \sum_{k=1}^{n-2} f(k) \sin\left(\frac{\pi k \ell}{n-1}\right) \\ &= f(0)\sin(0) + f(n-1)\sin(\pi\ell) = 0. \end{aligned}$$

We can in fact make a similar argument to simplify A_ℓ. Following the argument above yields

$$A_\ell = \sum_{j=0}^{n-1} f(j) \cos\left(\frac{\pi j \ell}{n-1}\right) + \sum_{k=1}^{n-2} f(k) \cos\left(\frac{\pi(2(n-1)-k)\ell}{n-1}\right).$$

We now use the fact that

$$\cos\left(\frac{\pi(2(n-1)-k)\ell}{n-1}\right) = \cos\left(2\pi\ell - \frac{\pi k \ell}{n-1}\right) = \cos\left(\frac{\pi k \ell}{n-1}\right)$$

to obtain

$$A_\ell = \sum_{j=0}^{n-1} f(j)\cos\left(\frac{\pi j\ell}{n-1}\right) + \sum_{k=1}^{n-2} f(k)\cos\left(\frac{\pi k\ell}{n-1}\right) \tag{6.8.7}$$
$$= f(0)\cos(0) + f(n-1)\cos(\pi\ell) + 2\sum_{j=1}^{n-2} f(j)\cos\left(\frac{\pi j\ell}{n-1}\right)$$
$$= f(0) + (-1)^\ell f(n-1) + 2\sum_{j=1}^{n-2} f(j)\cos\left(\frac{\pi j\ell}{n-1}\right).$$

Now, we aim to substitute $B_\ell = 0$ and the expression for A_ℓ into (6.8.5) to get the Discrete Cosine Transform. However, the nonzero part of the expression in (6.8.5) still needs some simplification. We compute

$$\sum_{\ell=0}^{2(n-1)-1} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right) = \sum_{\ell=0}^{n-1} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right) + \sum_{\ell=n}^{2(n-1)-1} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right)$$
$$= \sum_{\ell=0}^{n-1} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right) + \sum_{j=1}^{n-2} A_{2(n-1)-j} \cos\left(\frac{\pi kj}{n-1}\right)$$
$$= A_0 + (-1)^k A_{n-1} + \sum_{\ell=1}^{n-2} (A_\ell + A_{2(n-1)-\ell}) \cos\left(\frac{\pi k\ell}{n-1}\right).$$

By Exercise 6.8.1, below, we finally arrive at

$$\sum_{\ell=0}^{2(n-1)-1} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right) = A_0 + (-1)^k A_{n-1} + 2\sum_{\ell=1}^{n-2} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right). \tag{6.8.8}$$

Exercise 6.8.1. Show that $A_\ell = A_{2(n-1)-\ell}$. △

Substituting $B_\ell = 0$, (6.8.7) and (6.8.8) into (6.8.5) and restricting to $0 \le k \le n-1$, where $f_e(k) = f(k)$, we obtain

$$f(k) = \frac{1}{2(n-1)}(A_0 + (-1)^k A_{n-1}) + \frac{1}{n-1}\sum_{\ell=1}^{n-2} A_\ell \cos\left(\frac{\pi k\ell}{n-1}\right), \tag{6.8.9}$$

where A_ℓ is given by

$$A_\ell = f(0) + (-1)^\ell f(n-1) + 2\sum_{k=1}^{n-2} f(k)\cos\left(\frac{\pi k\ell}{n-1}\right). \tag{6.8.10}$$

The Discrete Cosine Transform (DCT) is the transformation that maps a real valued function $f : \mathbb{Z}_n \to \mathbb{R}$ to the coefficients $A_0, A_1, \ldots, A_{n-1}$. The inverse DCT is the mapping that takes the coefficients $A_0, \ldots, A_{n-1}$ and reconstructs the function $f(k)$ via (6.8.9). Just like with the DFT (see Section 6.3), there is a fast recursive algorithm for computing the DCT using only $O(n \log n)$ operations. The DCT is the fundamental tool behind image and audio compression algorithms, like mp3 and jpeg. We leave the Discrete Sine Transform to an exercise.

Exercise 6.8.2. Derive the Discrete Sine Transform (DST) reconstruction formula

$$f(k) = \frac{1}{n+1} \sum_{\ell=0}^{n-1} B_\ell \sin\left(\frac{\pi(k+1)(\ell+1)}{n+1}\right), \tag{6.8.11}$$

where

$$B_\ell = 2 \sum_{k=0}^{n-1} f(k) \sin\left(\frac{\pi(k+1)(\ell+1)}{n+1}\right),$$

by following the arguments in this section, except for replacing the even extension f_e with the odd extension $f_o : \mathbb{Z}_{2(n+1)} \to \mathbb{R}$ defined by

$$f_o(k) = \begin{cases} 0, & \text{if } k = 0 \\ f(k-1), & \text{if } 1 \le k \le n, \\ 0, & \text{if } k = n+1 \\ -f(2(n+1)-1-k), & \text{if } n+2 \le k \le 2(n+1)-1. \end{cases} \tag{6.8.12}$$

△

6.8.1 DCT-based image compression

Python Notebook: .ipynb

We briefly explore an application of the Discrete Cosine Transform to image compression. The idea of signal or image compression with the DCT is to compute the DCT coefficients A_ℓ, as per (6.8.10), and threshold the smallest coefficients to zero. The nonzero coefficients are stored and the image or signal is decompressed by taking the inverse DCT, as per (6.8.9). In the case of image compression, we use the two dimensional DCT, which is computed row-wise and then column-wise, similar to the multi-dimensional DFT introduced in Section 6.7. Also, as in Section 3.5 on PCA-based image compression, we split

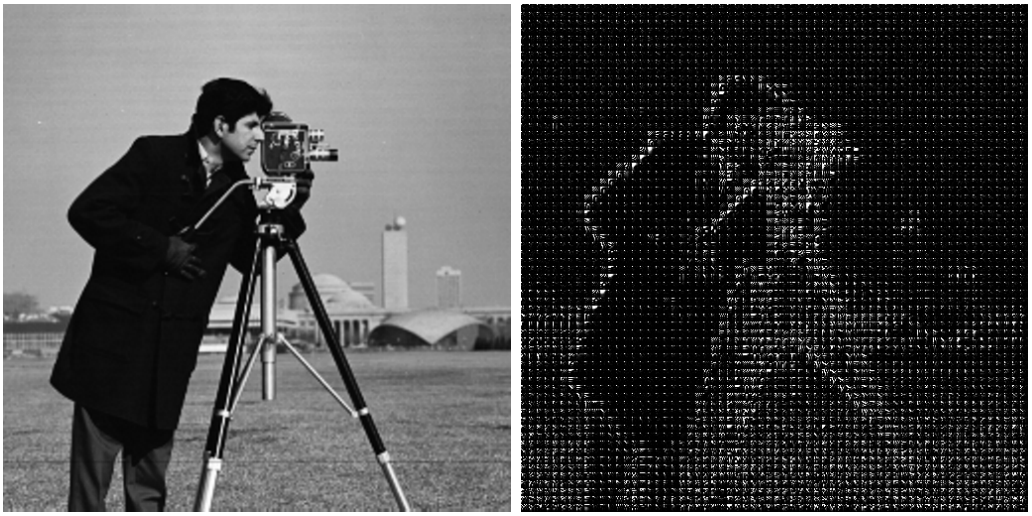


Figure 6.8.2: The cameraman image and its Discrete Cosine Transform (DCT) coefficients computed on 8×8 blocks.

the image up into 8×8 blocks and perform the DCT separately on each block. This works because small blocks in images are usually very simple and can be well-approximated by only a few DCT coefficients. Figure 6.8.2 shows the DCT coefficients of the cameraman image on 8×8 blocks.

We show the results of the decompressed cameraman images at different compression ratios in Figure 6.8.3 for the PCA-based compression from Section 3.5 and the DCT-based compression. We show the plots of PSNR vs compression ratio in Figure 6.8.4. DCT-based compression is generally superior at low and moderate compression ratios. This is because we keep only the most significant DCT modes on each block, and these may differ from block to block, wheres in PCA-based compression we used the top k PCA coefficients over the whole image, which are good for all blocks on average, but might yield poor quality reconstructions on some blocks. The PCA-based compression does work better than DCT at very high compression ratios, reflecting the fact that when we use only a very small number of basis images for the blocks, a PCA-based basis is better adapted to the particular image (in fact, it is learned from the image, while DCT is not).

6.9 The Sampling Theorem

In these notes we generally deal with discrete signals, and the assumption is that they have been sampled from a continuous signal at a particular sampling

(a) Compression Ratio: 12.6:1, PSNR: (PCA: 34.12 dB, DCT: 39.46 dB)

(b) Compression Ratio: 6.3:1, PSNR: (PCA: 36.61 dB, DCT: 45.16 dB)

(c) Compression Ratio: 2.1:1, PSNR: (PCA: 51.04 dB, DCT: 51.14 dB)

Figure 6.8.3: Comparison of PCA-based and DCT-based image compression on the cameraman image at different compression ratios. Left is original, center is PCA, and right is DCT.

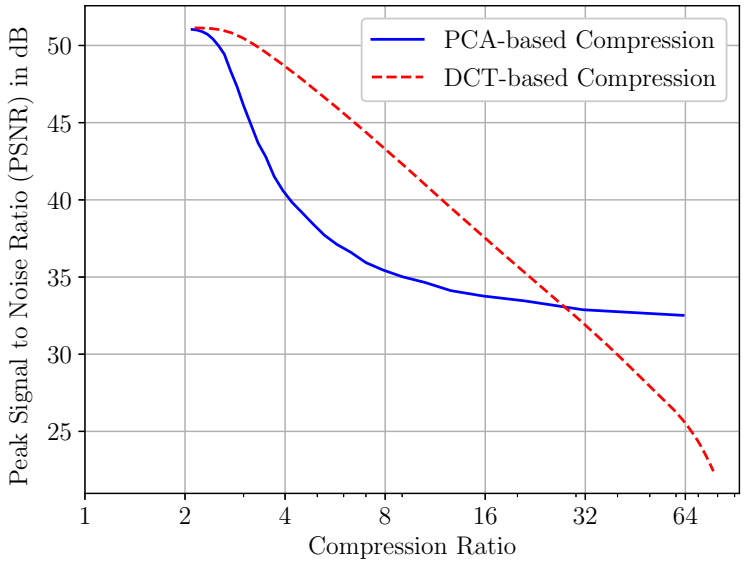


Figure 6.8.4: PSNR vs Compression Ratio for PCA-based and DCT-based compression of the cameraman image. DCT-based compression is better at low compression ratios, while PCA-based compression performs better at higher compression ratios.

frequency. In audio processing, the continuous signal is exactly the displacement of the diaphragm on the microphone, which encodes the sound waves striking its surface. The main question we will address in this section is how fast do we need to sample a continuous signal in order to capture all the important information in the signal. We will find that for a sufficiently fast sampling rate, the signal can be reconstructed perfectly from its samples, provided it is *band-limited* (which means it has a maximum frequency). We will also show how to reconstruct the signal from its samples with a simple convolution operation. This theory is referred to as the Sampling Theorem. Often then name Shannon-Nyquist Sampling Theorem is used, since it was proved by Claude Shannon in a 1949 paper, and some ideas, in particular the critical sampling rate, were due to work by Harry Nyquist previously. However, many other researchers had discovered the result independently, some before Shannon did, so it has become more common to use the term "The Sampling Theorem".

The standard version of the sampling theorem says that if a signal $u : \mathbb{R} \to \mathbb{R}$ contains no frequencies greater than σ_{max}, then u can be perfectly reconstructed from its evenly spaced samples provided the sampling frequency

is greater than $2\sigma_{max}$. The critical frequency $2\sigma_{max}$ is called the Nyquist rate. Furthermore, the signal u can be reconstructed from its samples via the Sinc Interpolation formula

$$u(t) = \sum_{j=-\infty}^{\infty} u(jh)\ \mathrm{sinc}\left(\frac{t-jh}{h}\right), \tag{6.9.1}$$

where h is the sampling period and $\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$ is the normalized Sinc function (defining $\mathrm{sinc}(0) = 1$). The sampling frequency is $\frac{1}{h}$, so the Nyquist rate condition for the Sampling Theorem is that $\frac{1}{h} > 2\sigma_{max}$, or $h < \frac{1}{2\sigma_{max}}$. At sampling intervals $h > \frac{1}{2\sigma_{max}}$ aliasing occurs, where frequencies that are too high to be represented at the sampling rate are aliased to lower frequencies, creating artifacts that make reconstruction of the signal from its samples impossible.

The proof of the version of the Sampling Theorem described above requires the continuous Fourier Transform, which we do not cover in these notes. We will instead prove a similar version of the Sampling Theorem for *periodic* signals, which can be done with the DFT theory developed in these notes and some very basic facts about Fourier series. In particular, we assume our signal $u : \mathbb{R} \to \mathbb{R}$ is periodic with period 1, and has no frequency larger than σ_{max}, where σ_{max} is a positive integer. This means that the signal u has the Fourier Series representation

$$u(t) = \sum_{k=-\sigma_{max}}^{\sigma_{max}} c_k\, e^{2\pi i k t}. \tag{6.9.2}$$

The complex numbers c_k are the Fourier Series Coefficients of u and we can think of the Fourier Series as the limit as $n \to \infty$ of the DFT. Since we do not study the Fourier Series in these notes, we will take the representation (6.9.2) as a starting place, and investigate whether we can reconstruct u from its evenly spaced samples $u(jh)$ for a sampling period $h > 0$ and $j \in \mathbb{Z}$. Naturally this boils down to determining whether we can deduce the Fourier Series Coefficients c_k from the samples $u(jh)$. The answer is positive precisely when the sampling frequency exceeds the Nyquist rate. We furthermore obtain a reconstruction formula similar to Sinc Interpolation, as shown in the following result.

Theorem 6.9.1. *Suppose that u is given by (6.9.2) and let $h = 1/n$ for $n \in \mathbb{N}$ with $n > 2\sigma_{max}$. Assume also that n is odd. Then $u(t)$ can be reconstructed*

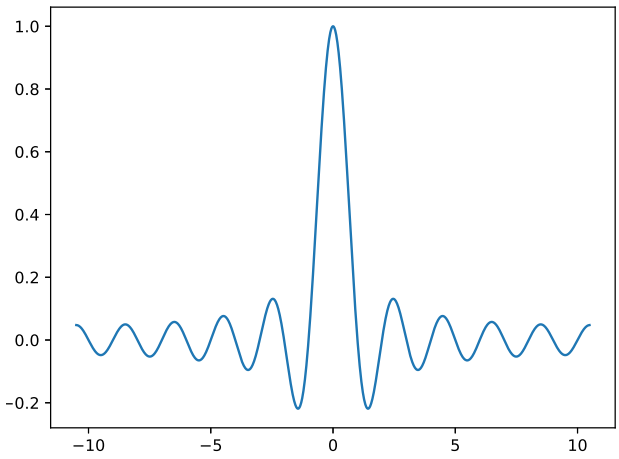


Figure 6.9.1: Depiction of the Sinc-like kernel $S(t) = \operatorname{sinc}(t)/\operatorname{sinc}(ht)$ for $n = 21$ and $h = 1/21$. The kernel is periodic with period $n = 21$.

from its evenly spaced samples $u(jh)$ and furthermore we have

$$u(t) = \sum_{j=0}^{n-1} u(jh) S\left(\frac{t - jh}{h}\right), \tag{6.9.3}$$

where $S(t)$ is given by

$$S(t) = \frac{\operatorname{sinc}(t)}{\operatorname{sinc}(ht)}.$$

Remark 6.9.2. Technically $S(t)$ is undefined when ht is a nonzero integer, since $\operatorname{sinc}(ht) = 0$. We define S by continuity in this case (see Exercise 6.9.3). We also note that the assumption that n is odd in Theorem 6.9.1 is only for convenience, and a similar result holds for even n. We show a depiction of $S(t)$ in Figure 6.9.1.

Proof. Define the sampling function $f : \mathbb{Z} \to \mathbb{C}$ by $f(j) = u(jh) = u(j/n)$. Since u is 1-periodic, f is n-periodic on $\mathbb{Z}$, so we may think of f as a function in $L^2(\mathbb{Z}_n)$. By (6.9.2) we have

$$f(j) = \sum_{k=-\sigma_{max}}^{\sigma_{max}} c_k \, e^{2\pi i k j/n} = \sum_{k=-\sigma_{max}}^{\sigma_{max}} c_k \omega^{kj},$$

where $\omega = e^{2\pi i/n}$. Recalling the orthogonality of the exponential functions $u_\ell(j) = \omega^{j\ell}$ (see Lemma 6.2.1), a natural way to try and obtain the coefficients

c_k from the sampled function $f(j)$ is to compute the inner product

$$\langle f, u_\ell \rangle = \sum_{j=0}^{n-1} f(j)\omega^{-j\ell} = \sum_{j=0}^{n-1} \sum_{k=-\sigma_{max}}^{\sigma_{max}} c_k \omega^{j(k-\ell)} = \sum_{k=-\sigma_{max}}^{\sigma_{max}} c_k \sum_{j=0}^{n-1} \omega^{j(k-\ell)}.$$

As in the proof of Lemma 6.2.1, it is tempting to say that the right hand side is nc_ℓ. However, there are in fact contributions from all k such that $k = \ell$ modulo n, that is $k = \ell + np$ for some integer p, since in that case $\omega^{j(k-\ell)} = \omega^{jnp} = 1$ since $\omega^n = 1$. This reflects the aliasing of high frequencies when the sampling rate is insufficiently fast. Hence, we have

$$\langle f, u_\ell \rangle = n \sum_{k=-\sigma_{max}}^{\sigma_{max}} c_k \delta_{\{k=\ell \bmod n\}},$$

where $\delta_{\{k=\ell \bmod n\}} = 1$ when $k = \ell$ modulo n, and zero otherwise. In order to determine the coefficient c_k, we need the sum above to have exactly one term, namely c_ℓ. This is the case whenever $\sigma_{max} < \frac{n}{2}$, or $n - 1 \geq 2\sigma_{max}$, so that if $k = \ell + np$ and $|k| \leq \sigma_{max}$ we must have $p = 0$. In this case we have $\langle f, u_\ell \rangle = nc_\ell$, and by (6.9.2) and the fact that $h = 1/n$ we have

$$u(t) = \frac{1}{n} \sum_{k=-\sigma_{max}}^{\sigma_{max}} \langle f, u_k \rangle \, e^{2\pi i k t}.$$

Since $\langle f, u_k \rangle = 0$ for $\sigma_{max} < |k| \leq \frac{n-1}{2}$, we can expand the range of k in the sum to range from $k = -\frac{n-1}{2}$ to $k = \frac{n-1}{2}$ to obtain

$$\begin{aligned} u(t) &= \frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} \langle f, u_k \rangle \, e^{2\pi i k t} \\ &= \frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} \sum_{j=0}^{n-1} f(j) e^{-2\pi i k j/n} e^{2\pi i k t} \\ &= \sum_{j=0}^{n-1} u(jh) \left(\frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} e^{2\pi i k (t - jh)} \right). \end{aligned}$$

The proof is completed by invoking Exercise 6.9.3, below. □

Exercise 6.9.3. Let n be odd. Show that for $t \notin \mathbb{Z}$ we have

$$\frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} e^{2\pi ikt} = \frac{\operatorname{sinc}(nt)}{\operatorname{sinc}(t)}.$$

What happens when $t \in \mathbb{Z}$? $\triangle$

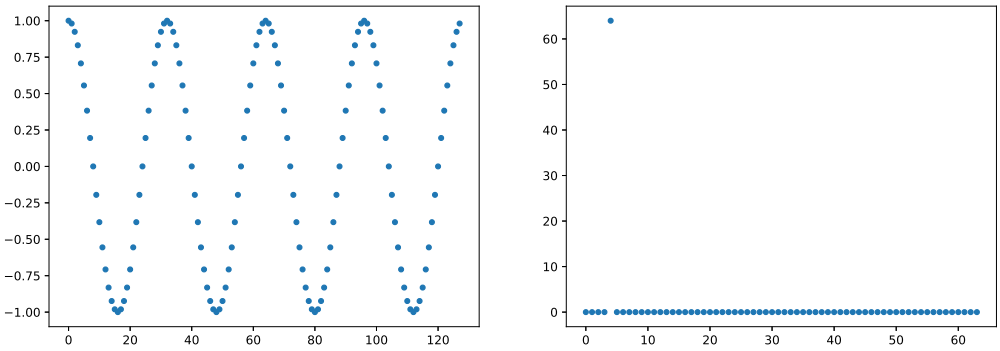
Chapter 7

The Discrete Wavelet Transform

The Discrete Fourier Transform (DFT) basis functions $u_\ell(k) = e^{2\pi ik\ell/n}$ are completely localized in the frequency domain. That is $\mathcal{D}u_\ell = \delta_\ell$ has only one nonzero entry. This allows us to decompose a signal or image very precisely into different frequency components, which is one of the most useful properties of the DFT in practice. This comes at the expense of complete delocalization in the time (or for images, space) domain; i.e., the function u_ℓ is nonzero at all grid points. Figure 7.0.1 shows the real part of u_ℓ and its DFT to illustrate the dichotomy between time and frequency localization. This delocalization means we are measuring frequencies across the entire signal, and not locally in time.

In practical applications of the DFT, we address this issue of delocalization of the basis functions by breaking the signal or image up into small blocks, and applying the Fourier transform on each block separately. This is done, for example, in Section 3.5 when using PCA for image compression, and Section 6.8.1 when using the Discrete Cosine Transform (DCT) for image compression. See, for example, Figure 6.8.2 for an illustration of how the DCT is applied locally to an image to sidestep the delocalization of the DCT basis.

When viewing the FFT or DCT on blocks as a transformation of the entire image, we are essentially applying a windowed version of the FFT, using a window function that is exactly equal to 1 on each block, and zero elsewhere (the window function multiplies the Fourier basis functions, as in Figure 7.0.2). This discontinuous window function, which is a rather crude attempt to localize the DFT basis, leads to the blocking artifacts observed in block-based image compression (see Figure 6.8.3). Figure 7.0.2 shows the DFT of a windowed version of the Fourier basis function from Figure 7.0.1, which shows that localizing in the time domain leads to delocalization in the frequency domain. In fact, there is a fundamental limit to how much a function and its



(a) $v(k) = \cos(2\pi k\ell/n)$ (b) $|\mathcal{D}v(k)|$

Figure 7.0.1: A plot the real part of a Fourier basis function and its Discrete Fourier Transform (DFT). The function v ($\ell = 4, n = 128$) is completely delocalized (most values are nonzero), while its DFT is highly localized (all values vanish except one).

DFT can both be localized. The uncertainty principle [8] states that

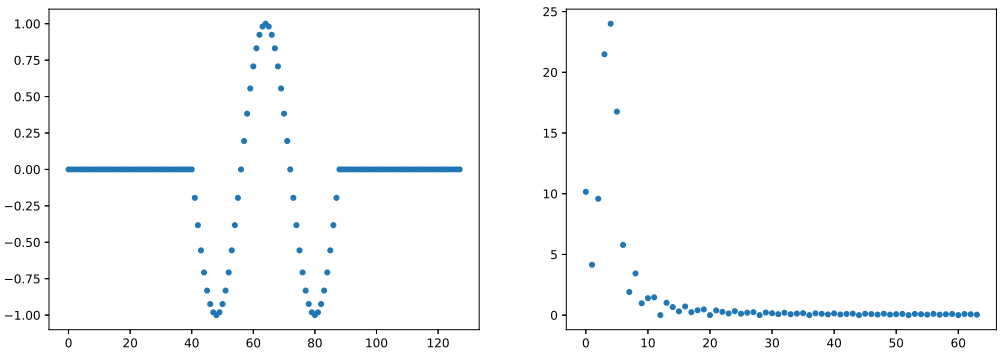
$$\|f\|_0 \|\mathcal{D}f\|_0 \geq n, \tag{7.0.1}$$

where $\|f\|_0$ is the number of nonzero values of f. This says that it is impossible for both f and $\mathcal{D}f$ to both be localized (i.e., have mostly zero entries). This bound is saturated by the Fourier basis functions u_ℓ, which satisfy $\|u_\ell\|_0 = n$ and $\|\mathcal{D}u_\ell\|_0 = 1$.[1]

Wavelets provide a principled approach to designing a frequency transformation whose basis functions are localized, as much as is possible given the uncertainty principle, in both the time and frequency domains. In contrast with the DFT, the wavelet transform decomposes an image into its frequency components at *multiple scales*, allowing fine scale details to be distinguished from large ones. Like the DFT, the Wavelet transform has proven to be very useful for image denoising and compression, and some nonlinear variations on the wavelet transform (e.g., the scattering transform [3]) are currently used in deep learning for feature extraction.

This section offers a brief introduction to the Wavelet Transform. We focus mostly on Haar Wavelets, and introduce the main ideas with concrete

[1] In quantum mechanics, the probability distributions of the position and momentum of a particle are the Fourier Transforms of each other. In this context, the uncertainty principle (7.0.1) says that their distributions cannot both be localized, meaning we cannot determine both the position and momentum of a particle with high precision. This is known as Heisenberg's uncertainty principle.



(a) Windowed Fourier basis function (b) Windowed DFT

Figure 7.0.2: A plot the real part of a windowed Fourier basis function and its Discrete Fourier Transform (DFT). The windowed function is more localized in space, while its DFT is less localized in the frequency domain, compared to Figure 7.0.1.

examples, before giving the more abstract definitions. We also consider some applications to image denoising, compression, and classification.

7.1 The 1D Haar Wavelet

The Wavelet Transform is based on repeatedly decomposing a signal into a low frequency part, called the *approximation coefficients*, and a high frequency part, called the *detail coefficients*. This is best illustrated at first with an example. Consider the following length $n = 8$ signal

`Signal: (7,5,6,3,2,5,4,1)`

The first level of the Haar Wavelet Transformation breaks up the signal into blocks of size 2, so `(7,5)`, `(6,3)`, `(2,5)`, and `(4,1)`. For each block `(a,b)`, the approximation coefficient is the sum `a+b`, while the detail coefficient is the difference `b-a`. Thus, a one-level Haar Wavelet Transformation yields

`Approximation Coeff: (12,9,7,5) and Detail Coeff: (-2,-3,3,-3)`

Of course, the original signal can be easily reconstructed from the approximation and detail coefficients, so it is clear the transform is invertible.

The approximation and detail coefficients are normally placed together in an array, with the approximation coefficients coming first, so the one-level Haar Wavelet transform of our signal is

`1-level Haar Wavelet Transform: (12,9,7,5,-2,-3,3,-3)`

The Wavelet Transformation can be continued to further levels by applying the same procedure to the *approximation coefficients* only. The approximation coefficients `(12,9,7,5)` are split into blocks `(12,9)` and `(7,5)`, and the approximation coefficients are `(21,12)` while the detail coefficients are `(-3,-2)`. The 2-level Haary Wavelet Transform is then

`2-level Haar Wavelet Transform: (21,12,-3,-2,-2,-3,3,-3)`

We can now perform one final Wavelet Transformation on the remaining approximation coefficients `(21,12)`, yielding the 3-level transform

`3-level Haar Wavelet Transform: (33,-9,-3,-2,-2,-3,3,-3)`

At this point, we have only a single approximation coefficient remaining and cannot perform any further iterations of the Wavelet Transform. Since each level of the Wavelet Transform is invertible, the whole process is, regardless of how many levels are used.

We can view the 3-level Haar Wavelet Transformation as multiplication of the signal $f = (7, 5, 6, 3, 2, 5, 4, 1)$ by the matrix

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}. \tag{7.1.1}$$

The rows can be interpreted as the basis functions for the Haar Wavelet Transformation. Notice that the rows are mutually orthogonal, so the Wavelet Transformation is an orthogonal change of coordinates, like the DFT.

In general, we can define a one-level Haar Wavelet Transform of a signal $f \in L^2(\mathbb{Z}_n)$ of length $n = 2^k$ as the signal $\mathcal{W}_1 f$ given by

$$\mathcal{W}_1 f(j) = \begin{cases} f(2j+1) + f(2j), & \text{if } 0 \le j \le \frac{n}{2} - 1 \\ f(2j-n+1) - f(2j-n), & \text{if } \frac{n}{2} \le j \le n-1. \end{cases} \tag{7.1.2}$$

The first line in the definition of $\mathcal{W}_1 f$ corresponds to the approximation coefficients, while the second line is the detail coefficients. The second level of the

Haar Wavelet Transformation acts only on the approximation coefficients, so we have

$$\mathcal{W}_2 f(j) = \begin{cases} \mathcal{W}_1 f(2j+1) + \mathcal{W}_1 f(2j), & \text{if } 0 \le j \le \frac{n}{4} - 1 \\ \mathcal{W}_1 f(2j - \frac{n}{2} + 1) - f(2j - \frac{n}{2}), & \text{if } \frac{n}{4} \le j \le \frac{n}{2} - 1 \\ \mathcal{W}_1 f(j), & \text{if } \frac{n}{2} \le j \le n - 1 \end{cases} \tag{7.1.3}$$

In general, the ℓ^{th}-level Haar Wavelet Transformation is

(7.1.4)

$$\mathcal{W}_\ell f(j) = \begin{cases} \mathcal{W}_{\ell-1} f(2j+1) + \mathcal{W}_{\ell-1} f(2j), & \text{if } 0 \le j \le \frac{n}{2^\ell} - 1 \\ \mathcal{W}_{\ell-1} f(2j - \frac{n}{2^{\ell-1}} + 1) - f(2j - \frac{n}{2^{\ell-1}}), & \text{if } \frac{n}{2^\ell} \le j \le \frac{n}{2^{\ell-1}} - 1 \\ \mathcal{W}_{\ell-1} f(j), & \text{if } \frac{n}{2^{\ell-1}} \le j \le n - 1 \end{cases}$$

The Haar Wavelet Transform is simple to implement recursively in a programming language like Python. Algorithm 7.1 gives the Python code for a 1D Haar Wavelet Transform of a signal whose length is a power of 2 (for other length signals, is is common to pad with zeros or use a reflection to increase the signal length). The inverse transform is given in Algorithm 7.1. The structure of the Wavelet Transform is very similar to the FFT discussed in Section 6.3, except that only half of the remaining signal is processed at each level, instead of the entire signal. For a signal of length $n = 2^k$, the maximum depth (i.e., number of levels) of the Wavelet Transform is k, and at each step the length of the signal to be processed reduces by half. Thus, the number of operations for the first level is $2n$, the second level is $2\frac{n}{2}$, the third level is $2\frac{n}{4}$, and so on. After k levels the number of operations is

$$2n \left(1 + \frac{1}{2} + \cdots + \frac{1}{2^{k-1}}\right) \le 4n.$$

Thus, the Wavelet Transform takes a linear number of operations $O(n)$, in contrast with the FFT which takes $O(n \log_2 n)$ operations.

7.2 2D Haar Wavelet Transform

The 2D Haar Wavelet Transformation acts on images in a similar way to the 1D version acting on signals. We assume we have an image of size $n \times n$, where $n = 2^k$. One level of the 2D Haar wavelet transformation breaks up the image into 2×2 blocks and computes 1 approximation coefficient and 3 detail coefficients corresponding to horizontal, vertical, and diagonal details. The formulas for the approximation and detail coefficients can be found by

Algorithm 7.1.1 The Haar Wavelet Transformation in Python

```
import numpy as np

def haar_wavelet(f,depth):
    g = np.zeros_like(f)
    n2 = len(f)>>1
    g[:n2] = f[::2] + f[1::2] #Approximation coeff
    g[n2:] = f[1::2] - f[::2] #Detail coeff
    if depth >= 2:
        g[:n2] = haar_wavelet(g[:n2],depth-1)
    return g
```

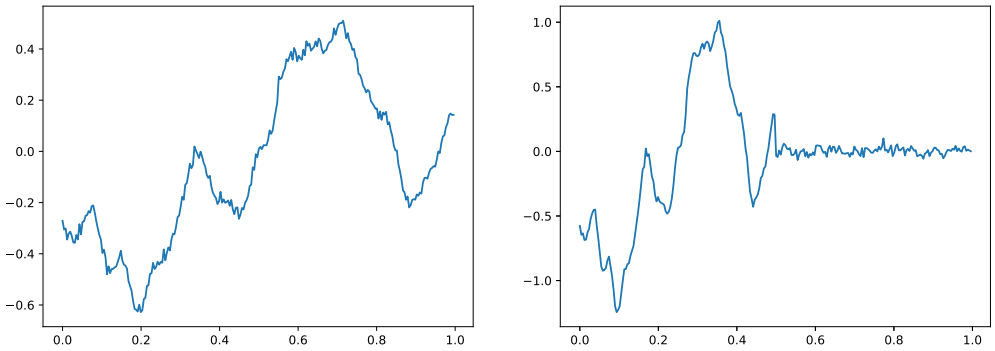
Algorithm 7.1.2 The Inverse Haar Wavelet Transformation in Python

```
import numpy as np

def inverse_haar_wavelet(f,depth):
    if depth == 0:
        return f
    else:
        n2 = len(f)>>1
        h = inverse_haar_wavelet(f[:n2],depth-1)
        g = np.zeros_like(f)
        g[1::2] = (h + f[n2:])/2
        g[::2] = (h - f[n2:])/2
        return g
```

applying the 1D Haar Transformation to the rows of the image, and then afterwards to the columns, in a similar way that the multi-dimensional DFT can be decomposed into 1D DFTs of rows and columns (see Section 6.7). For a 4×4 patch of the image

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$



(a) Noisy Signal (b) Haar Wavelet Transform (1 level)

Figure 7.1.1: One level of the Haar Wavelet Transformation applied to a noisy signal. Notice that much of the noise appears in the detail coefficients. Wavelet based denoising and compression algorithms are based on thresholding the detail coefficients.

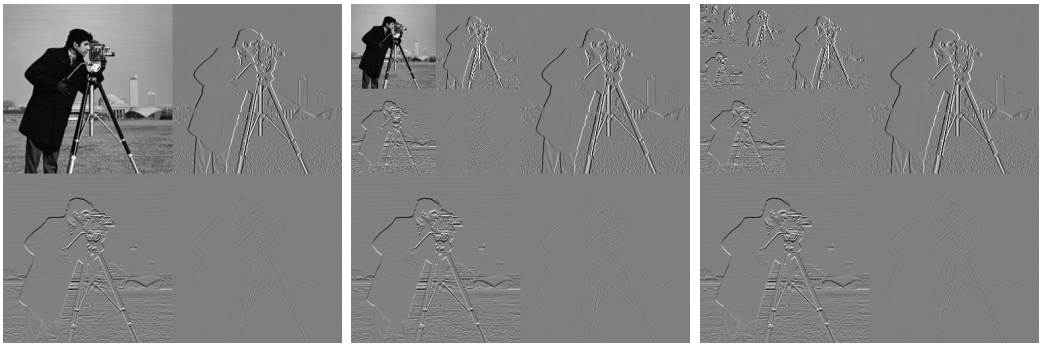
the approximation, vertical, horizontal, and diagonal detail coefficients, denoted A, V, H, D, respectively, are given by

$$\begin{aligned} A &= a + b + c + d \\ H &= -a - b + c + d \\ V &= -a + b - c + d \\ D &= a - b - c + d. \end{aligned}$$

The inverse transform is simple to obtain; indeed, we have

$$\begin{aligned} a &= \frac{1}{4}(A - H - V + D) \\ b &= \frac{1}{4}(A - H + V - D) \\ c &= \frac{1}{4}(A + H - V - D) \\ d &= \frac{1}{4}(A + H + V + D). \end{aligned}$$

Thus, one level of the 2D Haar Wavelet Transformation is clearly invertible. Further levels of the transform are obtained by applying the transform again to the approximation image. Figure 7.2.1 shows the 2D Haar Wavelet Transformation applied the cameraman image at levels 1 and 2, and 9, which is the largest possible depth for a 512×512 image.



(a) 1 Level (b) 2 Levels (c) 9 Levels

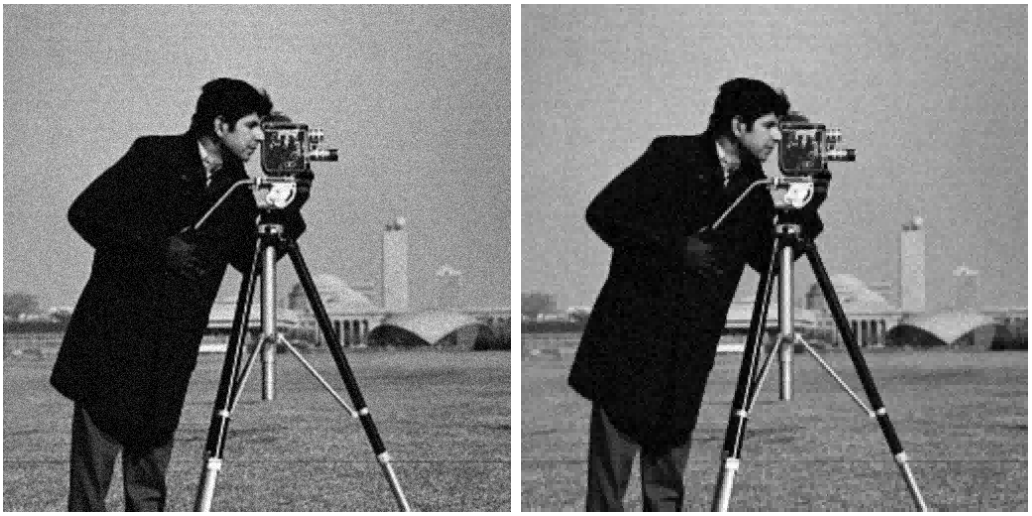
Figure 7.2.1: The Haar Wavelet Transformation of levels 1, 2, and 9 on the cameraman image. The approximation image is placed in the upper left corner, the horizontal detail in the lower left, the vertical detail in the upper right, and the diagonal detail in the lower right.

7.3 Wavelet denoising and compression

Python Notebook: .ipynb

We now give some brief applications of the Haar Wavelet Transformation to image denoising and compression. In both settings, the strategy is to threshold the detail coefficients to zero. We use a basic hard thresholding here, and the results can be greatly improved with a more sophisticated form of wavelet coefficient shrinkage. Figure 7.3.2 shows wavelet denoising of a noisy cameraman image. Here, we used a 2-level Haar Wavelet Transformation and set the detail coefficients with absolute value less than 0.5 to zero.

Figure 7.3.2 shows the results of Haar Wavelet based image compression at different compression ratios. Here, we used a 3-level Haar Wavelet Transformation, and we set the detail coefficients smaller than a given threshold to zero. We see the compressed images appear to lose resolution as the compression ratio increases. In Figure 7.3.3 we give a comparison of DCT-based compression (from Section 6.8.1) to wavelet-based image compression. The main difference is that DCT-based compression exhibits regular blocking artifacts on an 8×8 grid, while wavelet-based methods show fewer blocking artifacts. On the other hand, wavelet-based compression exhibits more low resolution artifacts.



(a) Noisy image (b) Haar Wavelet denoising

Figure 7.3.1: Example of using a 2-level Haar Wavelet Transformation to denoise an image. Detail coefficients with absolute value less than 0.5 were set to zero.

(a) 13:1 (b) 22:1 (c) 32:1

Figure 7.3.2: Example of using a 3-level Haar Wavelet Transformation to compress an image. Detail coefficients with absolute value less than 0.25 were set to zero in (a), 0.5 in (b), and 1 in (c).

7.4 Wavelet-based image classification

The Wavelet Transform can also be used as a feature extractor for image classification, and these ideas foreshadow the development of convolutional neural networks in Section 8.4.5. In Figure 7.4.1 we show the 1-level Haar Wavelet

(a) Haar Wavelet Compression (b) DCT Compression

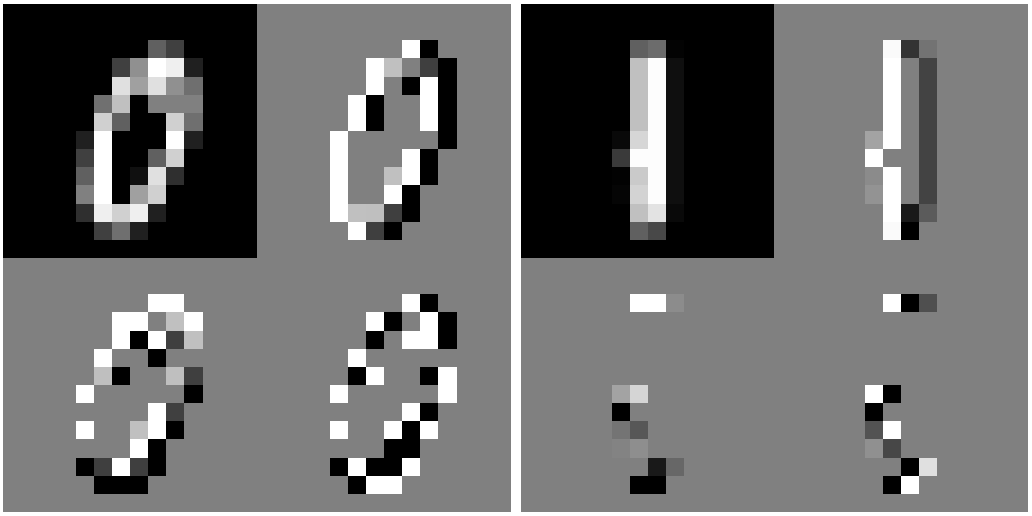
Figure 7.3.3: Comparison of DCT-based compression and Haar Wavelet compression on the cameraman image at a compression ratio of 32:1.

Transformation of an MNIST "0" and "1" digit. Notice the "1" has strong vertical details but very few horizontal details, while the zero has both vertical and horizontal details. We can use this kind of information to distinguish between the two digits.

Let $H(i,j)$ denote the horizontal detail coefficients and $V(i,j)$ the vertical, where $i,j = 1,\ldots,14$ (the images are 28×28) for a given MNIST digit. Now, the signs (e.g., positive or negative) of the detail coefficients are irrelevant, since they just indicate whether the detail represents a change from black to white or vice versa. So the first step is to take the absolute value of the detail coefficients $|H(i,j)|$ and $|V(i,j)|$. Second, we don't care where the detail appears in the image (in this simple example), since the digit may be written off-center or slightly differently than other digits in the same class. To remove the information about position, we sum the absolute values of the detail coefficients, which is also called *pooling*. This gives two numbers for every MNIST digit

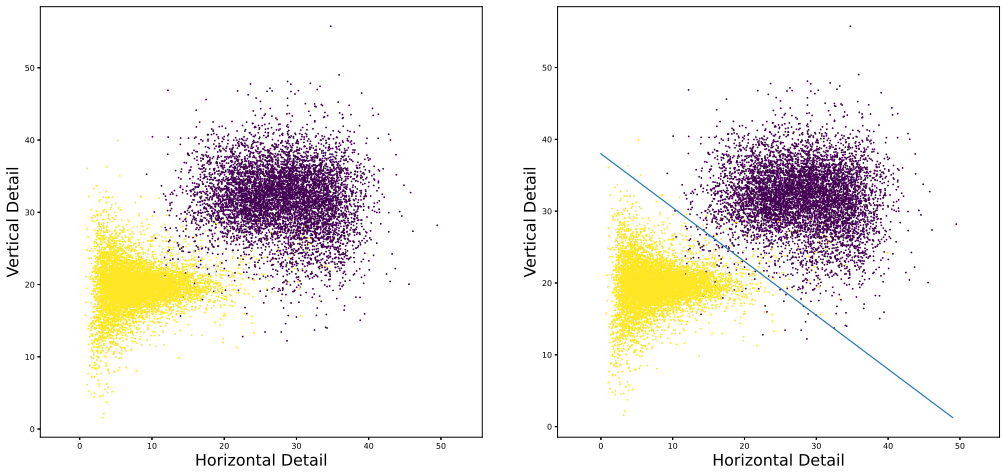
$$v = \sum_{i=1}^{14}\sum_{j=1}^{14} |V(i,j)| \quad \text{and} \quad h = \sum_{i=1}^{14}\sum_{j=1}^{14} |H(i,j)|. \tag{7.4.1}$$

Figure 7.4.2 (a) shows the horizontal and vertical scores h and v for each "0" and "1" MNIST digit. The zeros are colored purple and the ones are



(a) Zero (b) One

Figure 7.4.1: One level Haar Wavelet Transform of MNIST digits.



(a) Horizontal vs Vertical (b) Decision boundary

Figure 7.4.2: Classification of MNIST digits "0" and "1" with a linear decision boundary in the horizontal and vertical detail coordinates. The digit "0" is purple and "1" is yellow. Classification with this decision boundary correctly classifies the digits "0" and "1" with 98.93% accuracy.

colored yellow. We can see that simply choosing a threshold for the amount of

vertical or horizontal detail alone (i.e., attempting to separate the classes with a vertical or horizontal line) will missclassify a good fraction of the images. However, we can clearly see in the plot that the two clusters are nearly linearly separable, which means we can find a line that separates nearly all of the images into their respective classes. Figure 7.4.2 (b) shows one such separating line, which we found by inspection to be

$$0.75h + v = 38.$$

This choice is not unique and others are possible. Thus, we can construct a linear classifier by computing $f = 0.75h + v - 38$, and checking its sign. When $f > 0$ we classify the digit as a "0" and when $f \leq 0$ we classify the digit as "1". This classification achieves 98.98% accuracy.

Remark 7.4.1. There are several key components of the Wavelet-based classification method described above. First, we note that taking the absolute value was essential, since otherwise when we summed the detail coefficients, many positive and negative parts would cancel out. So introducing a nonlinearity is important. Second, the pooling step is also essential, since it introduces *translation invariance* into the Wavelet feature extractor. If a digit is shifted in any direction, the pooled features will be unchanged.

Finally, we point out that the horizontal and vertical detail coefficients are nothing other than convolutions of the original image I with filters ψ_1 and ψ_2, that is $H = I * \psi_1$ and $V = I * \psi_2$, followed by downsampling by a factor of 2 along each axis. The downsampling is also a type of pooling operation, and can be absorbed into the definition of the convolution (so we will omit it below). Thus, our binary classifier has the form

$$y = \text{sign}\left(w_1 \cdot \text{pool}(\sigma(I * \psi_1)) + w_2 \cdot \text{pool}(\sigma(I * \psi_2)) + b\right),$$

where $\sigma(x) = |x|$, $w_1 = 0.75$, $w_2 = 1$, $b = -38$ and the pooling operation "pool" simply sums all the pixel values in an image. This is a very simple example of a *Convolutional Neural Network (CNN)* for image classification.

The network has one convolutional layer and one fully connected layer, with a pooling step in between. The convolutional layer has two channels (i.e., convolution with ψ_1 and ψ_2) followed by a nonlinear *activation function* σ. There results are then pooled to give two outputs x_1 and x_2 of the convolutional layers. These are usually called *features*. The fully connected layer is an arbitrary linear function of x_1 and x_2, i.e., $w_1x_1 + w_2x_2 + b$, followed by another nonlinearity, the "sign" function.

We note that the main (and very important) difference between this example and general CNNs is that the filters ψ_i and weights w_i and b are automatically learned from training data in the CNN setting, instead of being manually hand-tuned like we did here, making CNNs more flexible and widely applicable. It is also common to have many more convolutional filters, convolutional layers, and fully connected layer for more complicated image classification problems[2]

7.5 General discrete Wavelets

We now discuss wavelet transformations in more generality, and make non-recursive definitions of the Wavelet Transformation, which allows for different choices of mother wavelets. Recall from Section 7.1 that the 1D Haar Wavelet Transformation on a signal of length $n = 8$ is represented by the matrix W defined by

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}. \tag{7.5.1}$$

The rows of the matrix W are the basis vectors for the Haar Wavelet Transformation. It turns out vectors can be written as rescaled and shifted versions of a *mother wavelet* ψ given by

$$\psi(t) = \begin{cases} 0, & \text{if } t < 0, \\ -1, & \text{if } 0 \leq t < \frac{1}{2}, \\ 1, & \text{if } \frac{1}{2} \leq t < 1, \\ 0, & \text{if } t \geq 1. \end{cases} \tag{7.5.2}$$

Indeed, we consider the rescaled wavelets $\psi_{j,k} \in L^2(\mathbb{Z}_8)$ given by

$$\psi_{j,k}(\ell) = \psi\left(2^{-j}\ell - k\right). \tag{7.5.3}$$

[2]On MNIST, good results (i.e., above 99%) for classification of all digits can be achieved with two convolutional layers followed by two fully connected layers).

The rows of the Haar Wavelet Transformation (except the first) are given exactly by the $\psi_{j,k}$ for $j = 1, 2, 3$ and $k = 0, \ldots, 2^{3-j} - 1$.

Exercise 7.5.1. Show The last 4 rows of W are exactly $\psi_{1,0}, \psi_{1,1}$, $\psi_{1,2}$ and $\psi_{1,3}$, the third and forth rows of W are $\psi_{2,0}$ and $\psi_{2,1}$, and the second row of W is $\psi_{3,0}$. △

The first row—the approximation coefficient—is obtained by another function called the *scaling function*, which in this case is

$$\varphi(t) = \begin{cases} 0, & \text{if } t < 0, \\ 1, & \text{if } 0 \leq t < 1, \\ 0, & \text{if } t \geq 1. \end{cases} \tag{7.5.4}$$

The first row of W is the rescaled scaling function $\varphi_{3,0}(\ell) = \varphi(2^{-3}\ell)$. We note that the mother wavelet ψ and the rescaling function φ are precisely the filters used to obtain the approximation and detail coefficients in the recursive definition of the Haar Wavelet Transformation given in Section 7.1.

For a general signal of length n that is a power of 2, the Haar Wavelet basis is given by the collection of functions $\psi_{j,k}$ with $j = 1, 2, \ldots, \log_2(n)$ and $k = 0, 1, 2, \ldots, 2^{\log_2(n)-j} - 1$, and the coarsest scale approximation coefficient $\varphi_{\log_2(n),0}$.

The Haar basis is an orthogonal basis.

Lemma 7.5.2. *For any (j_1, k_1) and (j_2, k_2), not identically equal, we have*

$$\sum_{\ell \in \mathbb{Z}} \psi_{j_1,k_1}(\ell)\psi_{j_2,k_2}(\ell) = 0.$$

Exercise 7.5.3. Proof Lemma 7.5.2. △

While the Haar Wavelet basis is orthogonal, the vectors $\psi_{j,k}$ do not have unit length, so it is not an orthonormal basis. For this reason, it is also common to choose a different normalization of the wavelets, given by

$$\psi_{j,k} = 2^{-\frac{j}{2}}\psi(2^{-j}\ell - k).$$

This ensures that $\|\psi_{j,k}\| = 1$ for all j, k, so the Haar Wavelet basis is an orthonormal basis.

This general, and non-recursive, definition of the Haar Wavelet makes it possible to construct other types of wavelets by choosing other mother wavelets ψ. The Haar Wavelets are all piecewise constant functions, leading to blocky

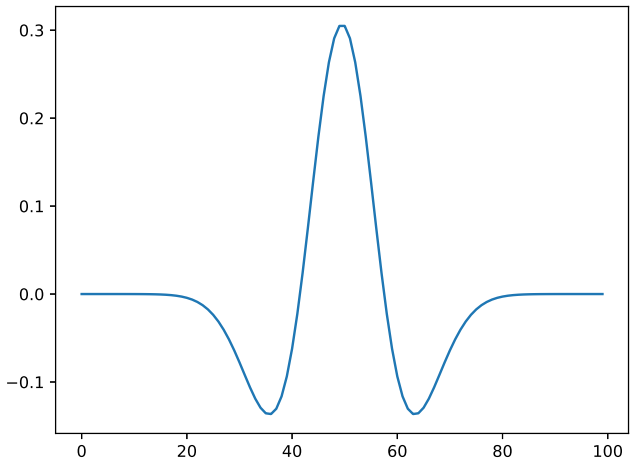


Figure 7.5.1: Plot of the Ricker Wavelet.

piecewise constant images in the compression and denoising contexts. Thus, it is often useful to consider other types of wavelets consisting of continuous mother wavelets, which lead to smoother reconstructions when thresholding away detail coefficients. One common choice is the Ricker Wavelet

$$\psi(t) = \frac{2}{\sqrt{3\sigma}\pi^{1/4}}\left(1 - \left(\frac{t}{\sigma}\right)^2\right) e^{-\frac{t^2}{2\sigma^2}}, \tag{7.5.5}$$

where σ is the bandwidth. We show the Ricker Wavelet in Figure 7.5.1.

The Ricker wavelet, and other constructed for continuous analysis, are not compactly supported and so they must be truncated for discrete analysis. The first compactly supported wavelets satisfying the orthogonality condition in Lemma 7.5.2 are due to Ingrid Daubechies and are called Daubechies wavelets. The 1st order Daubechies wavelet is the Haar Wavelet, and the higher order versions have additional regularity.

Chapter 8

Machine Learning

Machine learning refers to a class of algorithms that learn to complete tasks (like image classification or face recognition) by example, and are not explicitly programmed. For example, to perform handwritten digit recognition with a machine learning algorithm, one would provide many examples (sometimes hundreds or thousands) of images of handwritten digits and their correct numerical labels, and the algorithm learns a general rule to label new instances in a similar way. In this section, we give a brief introduction to machine learning, and cover some basic and more advanced algorithms, including deep neural networks.

For a python notebook with a basic introduction to some simple machine learning algorithms in the sklearn package, see .ipynb.

8.1 Introduction

Let $x_1, x_2, \ldots, x_m \in \mathbb{R}^n$ be a collection of datapoints, and let $y_1, y_2, \ldots, y_m \in \mathbb{R}^k$ be the corresponding labels. For example, in image classification, each x_i represents all the pixel values in a particular image, and the label y_i would indicate the class to which the image belongs (e.g., a dog, cat, backpack, etc.). For other problems like automatic image annotation, the label y_i encodes the caption for image x_i, using a word to vector encoding. There is generally not much loss in generality in assuming our datapoints and labels live in Euclidean space ($\mathbb{R}^n$ and $\mathbb{R}^k$, respectively), since more abstract data is normally embedded in Euclidean space before applying machine learning algorithms. Normally the labels y_i are chosen from the one-hot vectors $e_1, e_2, \ldots, e_k$, which are just the standard basis vectors in $\mathbb{R}^k$, with $e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^k$ having a 1 in only the i^{th} entry. The one-hot vector e_i will normally represent the

i^{th} class, out of k classes in total. There are three general fields within machine learning: (1) fully supervised, (2) semi-supervised, and (3) unsupervised learning.

8.1.1 Fully supervised learning

In fully supervised learning, the algorithm uses the *training data* pairs given by $(x_1, y_1), \ldots, (x_m, y_m)$ to learn a mapping

$$f : \mathbb{R}^n \to \mathbb{R}^k \tag{8.1.1}$$

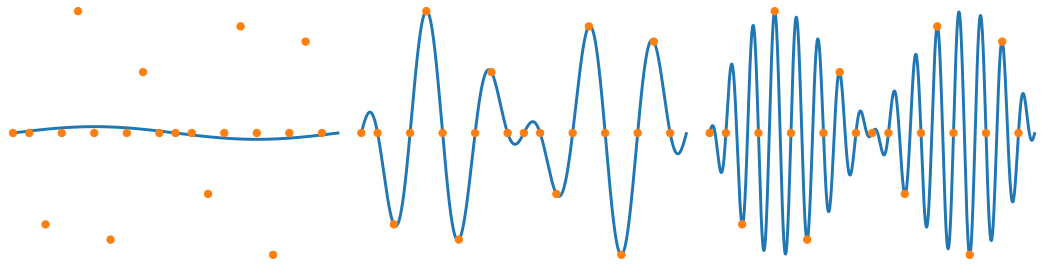
that generalizes the rule $f(x_i) = y_i$. Clearly there are many choices for the function f, so the learned function is far from unique. In practice, f is normally chosen from a class of parameterized functions $f(x; \omega)$, where $\omega \in \mathbb{R}^N$ are the parameters. For example, f could be a linear function $f(x; \omega) = x \cdot \omega$, or f could be the output of a neural network, where ω contains all the weights in each neuron in the network. The learning is acheived by minimizing a loss function of the form

$$\mathcal{L}(\omega) = \frac{1}{m} \sum_{i=1}^{m} \ell(f(x_i; \omega), y_i), \tag{8.1.2}$$

where $\ell : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ is a loss function. By minimizing $\mathcal{L}$, we are attempting to tune the weights ω so that $f(x_i; \omega) = y_i$ for all i, or make their values as close as possible. Possible choices for the loss ℓ are the L^2 loss $\ell(x, y) = |x - y|^2$, or the cross-entropy loss

$$\ell(x, y) = -\sum_{j=1}^{k} x(j) \log(y(j)).$$

While the immediate goal of training is to minimize the loss $\mathcal{L}$, the real objective is to learn a function f that correctly classifies new datapoints that have not been seen and are not included in the training data. *Generalization error* refers to the ability of an algorithm to accurately predict labels for new previously unseen data. A model with small generalization error is said to generalize well. Generally speaking, if the parametrization of f has too few degrees of freedom, then f may not fit the training data well, which is called *underfitting*, and thus will not generalize. If there are too many degrees of freedom then f may *overfit* the training data, which is another cause of poor generalization. The goal is to find a function f that *correctly fits* the training data, in the sense that it gives the simplest explanation for the observed trends,



(a) Underfitting (b) Correct fit (?) (c) Overfitting

Figure 8.1.1: Example of underfitting, correct fitting, and overfitting. The decision about what type of fit is correct, and what constitutes an overfit or underfit, is context dependent. In a setting where some of the datapoints are expected to be noisy, the fit in (b) may in fact be an overfit, and (a) may be preferable.

and is most likely to generalize. Figure 8.1.1 shows examples of underfitting, overfitting, and a correct fit for some training data (the orange points). It is often the case that what constitutes a correct fit is context dependent, and in the setting of high noise levels, the underfitting example in Figure 8.1.1 (a) could be interpreted as a correct (or close to correct) fit.

In practice, we measure generalization error by splitting our dataset into training and testing subsets. The model is trained using the training data, and then evaluated for generalization on the held out test data. If the model performs well on the testing data (or gives similar performance as it did on the training data), then the model can reasonably be expected to generalize to new data. If the testing accuracy is much lower than the training accuracy, then it is likely that the learned function f has *overfit* the training data and will not generalize well to new unseen data. Sometimes another validation dataset is needed to tune parameters in the algorithm, so it is also common to split the dataset three-ways, into training, validation, and testing.

In theoretical machine learning, generalization error is defined as the difference between the empirical loss (8.1.2) after training, and the expected value of the loss given a new data point (x, y) drawn from the same distribution as the training data. We assume the training data $(x_1, y_1), \ldots, (x_m, y_m)$ are independent and identically distributed random variables sampled from a probability distribution μ on $\mathbb{R}^n \times \mathbb{R}^k$. We let $\hat{\omega}$ denote a minimizer of the loss in (8.1.2), so $x \mapsto f(x; \hat{\omega}) =: \hat{f}(x)$ is the learned function. The *generalization error* is

defined as

$$\mathcal{G}(\hat{f}) = \int_{\mathbb{R}^n} \ell(\hat{f}(x), y) d\mu(x, y) - \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{f}(x_i), y_i). \tag{8.1.3}$$

The goal in learning is to find $\hat{f}$ that minimizes the generalization error $\mathcal{G}$, which ensures that the performance on a new random datapoint with the same statistics as the training data is similar to the training loss that was minimized during training. There is a tremendous amount of theoretical work in machine learning that aims to bound the generalization error $\mathcal{G}$ for different machine learning models. Each machine learning model is essentially a family of parameterized functions $\mathcal{F}$, from which $\hat{f}$ is chosen by minimizing the loss (8.1.2). One way to bound the generalization error is to prove uniform bounds over $f \in \mathcal{F}$, that is we use the estimate

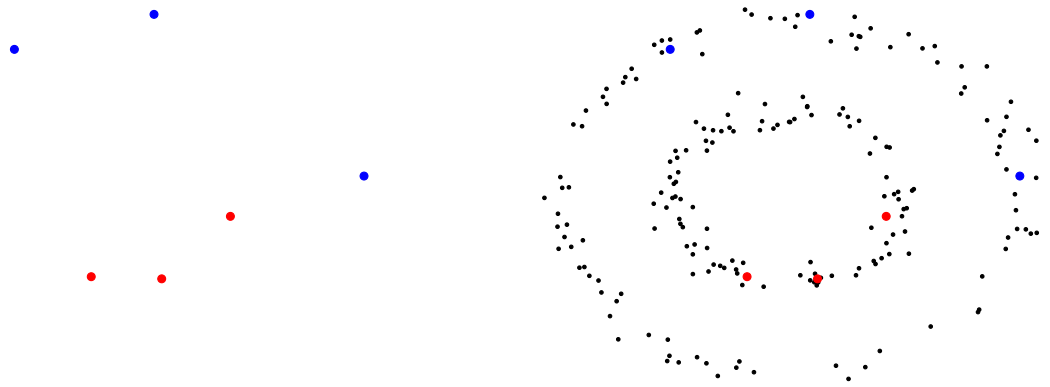
$$\mathcal{G}(\hat{f}) \leq \sup_{f \in \mathcal{F}} \mathcal{G}(f).$$

If we can estimate the supremum on the right hand side, then we can control the generalization error. This is called the *hypothesis space complexity approach*, since bounds on $\sup_{f \in \mathcal{F}} \mathcal{G}(f)$ involve measuring the size of the hypothesis space $\mathcal{F}$ in an appropriate way (e.g., Radamacher complexity, VC dimension, etc.). For more information we refer the reader to [14].

8.1.2 Semi-supervised learning

Fully supervised learning typically requires an abundance of labeled training data to learn from. In many applications (e.g., medical image analysis), labeled training examples are costly to obtain and it is desirable to have algorithms that can achieve good performance with far fewer labeled examples than are required in fully-supervised learning. Semi-supervised learning uses both labeled and unlabeled data to obtain higher performance at lower labeling rates. In this setting, we still have training data $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, but the number of training points m may be small. Additionally, we assume we have access to a large amount of unlabeled data $x_{m+1}, x_{m+2}, \ldots, x_N$, where $N \gg m$. The goal is to use the additional unlabeled data to train a better classifier with limited labeled data. In many applications, like image classification or speech recognition, it is essentially free to get access to large amounts of unlabeled data (e.g., images and audio samples), so we may as well try to make use of this data.

To see why unlabeled data may be useful in classification, consider the datapoints in Figure 8.1.2 (a). If we only use these three red and three blue



(a) 3 labels per class

(b) Labeled and unlabeled data

Figure 8.1.2: Example showing how the unlabeled data (the black points) can be useful for training a classifier. Without the unlabeled data, one cannot see the natural geometry and cluster structure in the dataset.

points to train a classifier, in the fully supervised setting, then we have very little information with which to train the classifier and are unlikely to generalize well. If, on the other hand, we have access to the black points in Figure 8.1.2 (b), i.e., the unlabeled data, then we can use this to inform our classifier, which in this case would split the inner circle of data from the outer one.

Semi-supervised learning comes in 2 variations. The first is the *inductive* setting, where one still learns a general rule $u : \mathbb{R}^n \to \mathbb{R}^k$ that aims to generalize the training data, while using properties of the unlabeled data. The second is the *transductive* setting, where we only learn labels for the additional unlabeled datapoints $x_{m+1}, \ldots, x_N$, i.e., the black points in Figure 8.1.2 (b). The transductive setting does not learn a general rule, and the classifier cannot be immediately applied to new data without retraining (or using some heuristic, like choosing the label of the closest labeled datapoint). One way to encode the structure of the unlabeled data is to build a graph over the data, as we did for spectral clustering in Section 4.2. We will explore this further when we discuss graph-based learning in Section 8.2. For more details on semi-supervised learning, we refer the reader to [5].

8.1.3 Unsupervised learning

Unsupervised learning algorithms use only the unlabeled data $x_1, x_2, \ldots, x_m$ for learning. Common tasks include clustering (like k-means clustering from Section 4.1 and spectral clustering from Section 4.2), as well as dimension

reduction algorithms, like principal component analysis (PCA) described in Chapter 3 and the spectral and t-SNE embeddings described in Section 8.3.2.

8.2 Graph-based semi-supervised learning

Python Notebook: .ipynb

A common way to use the unlabeled data in semi-supervised learning is to build a graph over the data (e.g., in image classification), or use an existing graph structure in the data (e.g., classification of webpages). Graph-based semi-supervised learning has proven to be very effective at utilizing unlabeled data for classification.

In this section, we assume our dataset is endowed with a weight matrix W, as in Section 4.2. The weight matrix W is an $m \times m$ matrix whose (i, j) entry encodes the similarity between datapoints i and j (our dataset consists of m points). Recall from Section 4.2 that the weight matrix has all nonnegative entries, and $W(i, j)$ is large when datapoints i and j are similar, and small or zero otherwise. We assume in this section that the weight matrix is symmetric $W = W^T$, though there are important applications of non-symmetric adjacency matrices. When building a graph over a dataset, we can use the Gaussian weights described in Section 4.2 (see (4.2.1)), or often times we build a k-nearest neighbor graph, where each datapoint is connected to its k-nearest Euclidean neighbors, with weights that are similar to the Gaussian weights. Figure 8.2.1 shows an example of a k-nearest neighbor graph over a synthetic dataset.

In graph-based semi-supervised learning, we are given a small amount of labeled data on the graph. Let $I_m = \{1, 2, \ldots, m\}$ denote the indices of all our datapoints. We assume there is a subset of the nodes $\Gamma \subset I_m$ that are assign ed label vectors from the one-hot vectors

$$S_k = \{e_1, e_2, \ldots, e_k\}.$$

We can treat the labels as a function $g : \Gamma \to S_k$, where $g(i)$ is the label of node $i \in \Gamma$. Recall that if the i^{th} node belongs to the j^{th} class then we set $g(i) = e_j$. The task in graph-based semi-supervised learning is to extend these labels from the subset Γ to the rest of the graph is a meaningful way. Of course, there are infinitely many ways the labels can be extended, so we must place some conditions or assumptions on the learned labels.

It is common in practice to take the *semi-supervised smoothness assumption*, which stipulates that the learned labels should vary as smoothly as pos-

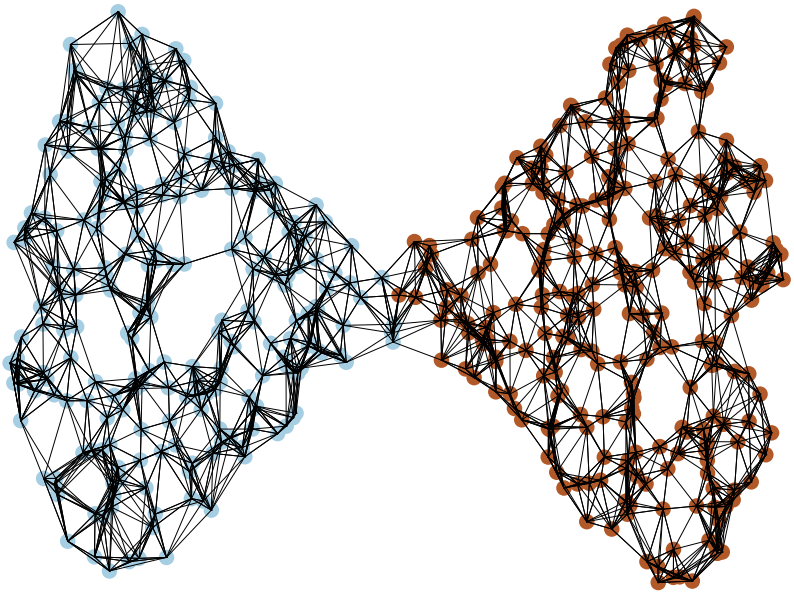


Figure 8.2.1: An example of a k-nearest neighbor graph.

sible, and in particular, should not change rapidly within high density regions of the graph, which are likely to be clusters with the same label. Laplacian regularized learning imposes the semi-supervised smoothness assumption by minimizing the function

$$E(u) = \frac{1}{4}\sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)\|u(i) - u(j)\|^2 \tag{8.2.1}$$

over labeling functions $u : I_m \to \mathbb{R}^k$, subject to $u = g$ on Γ, that is, that the known labels are correct. After minimizing E, the label for i is determined by the largest component of $u(i)$, that is

$$\text{Label of } i = \arg\max_{1\le j\le k}\{u(i)\cdot e_j\}.$$

The function E is the same type of smoothness functional we encountered in Section 4.2 when discussing spectral clustering, and is called a regularizer. Minimizing E ensures the label function u varies smoothly over the graph, enforcing the semi-supervised smoothness assumption. The reader should notice the similarities with the Tikhonov and Total Variation denoising discussed, for example, in Section 6.7.1 (in particular, the energy E is similar to the one

used for image inpainting). The Laplacian learning technique was first introduced in [20], and has been widely used since in machine learning. It is also sometimes called *label propagation*.

To minimize E we use gradient descent. There are several ways to compute the gradient; we will follow the functional analysis approach using the Gateaux derivative (or directional derivative) discussed in Remark 6.6.4. For this, we need to define an inner product. Here, we will use the inner product for $u, v : I_m \to \mathbb{R}^k$ defined by

$$\langle u, v\rangle_d = \sum_{i=1}^{m} d(i)u(i)^T v(i), \tag{8.2.2}$$

where $d : I_m \to \mathbb{R}$ are the degrees, given by $d(i) = \sum_{j=1}^{m} W(i,j)$. The induced norm is

$$\|u\|_d^2 = \langle u, u\rangle_d = \sum_{i=1}^{m} d(i)\|u(i)\|^2.$$

Of course, other choices of inner product are possible, and it is perhaps more natural to omit the degree. We will see the usefulness of including the degree below.

We recall from Remark 6.6.4 that the gradient $\nabla E(u)$ of E at u is defined by the identity

$$\frac{d}{dt}\Big|_{t=0} E(u + tv) = \langle \nabla E(u), v\rangle_d.$$

The idea is that we compute the directional derivative on the left hand side, and then identify the gradient by rewriting the result in the form on the right above. Since the gradient depends on the choice of inner product, it is sometimes common to denote the gradient as $\nabla_d E(u)$, but we will not do so here. We leave the computation of the directional derivative to an exercise.

Exercise 8.2.1. Show that

$$\frac{d}{dt}\Big|_{t=0} E(u + tv) = \sum_{i=1}^{m}\sum_{j=1}^{m} W(i,j)(u(i) - u(j))^T v(i).$$

△

Given the result of Exercise 8.2.1, we define the graph Laplacian as

$$Lu(i) = \sum_{j=1}^{m} W(i,j)(u(i) - u(j)). \tag{8.2.3}$$

The matrix of the Laplacian as a linear operator is exactly the same as the graph Laplacian introduced earlier in Section 4.2 in (4.2.6). The results of Exercise 8.2.1 can be written as

$$\frac{d}{dt}\Big|_{t=0} E(u+tv) = \sum_{i=1}^{m} (Lu(i))^T v(i) = \langle d^{-1}Lu, v\rangle.$$

Notice that the d^{-1} appeared due to the definition of the inner product (8.2.2) involving the degree function d. Therefore, the gradient of E is

$$\nabla E(u) = d^{-1}Lu. \tag{8.2.4}$$

Using gradient descent to minimize E amounts to the iteration

$$u_{t+1} = u_t - dt\nabla E(u_t) \tag{8.2.5}$$

for a time step dt. At each iteration we set the known label values $u_{k+1} = g$ on Γ, and we initialize at $u_0(j) = g(j)$ for $j \in \Gamma$ and $u_0(j) = 0$ otherwise. Now, we cannot use a Von Neumann analysis to choose a stable time step, as we did in Section 6.6.2, since we do not have the Fourier Transform in this setting. We instead use a maximum principle argument to determine stability. Due to our definition of the inner product (8.2.2) involving the degree, it turns out that $dt \leq 1$ results in a stable scheme.

Lemma 8.2.2. *If $0 < dt \leq 1$ then for all $t \geq 1$ and $1 \leq i \leq m$ we have*

$$\|u_t(i)\| \leq \max_{1\leq i\leq m} \|u_0(i)\|. \tag{8.2.6}$$

Proof. We write out gradient descent (8.2.5) as

$$\begin{aligned} u_{t+1}(i) &= u_t(i) - \frac{dt}{d(i)} \sum_{j=1}^{m} W(i,j)(u_t(i) - u_t(j)) \\ &= u_t(i) - \frac{dt}{d(i)} \sum_{j=1}^{m} W(i,j)u_t(i) + \frac{dt}{d(i)} \sum_{j=1}^{m} W(i,j)u_t(j) \\ &= (1-dt)u_t(i) + \frac{dt}{d(i)} \sum_{j=1}^{m} W(i,j)u_t(j). \end{aligned}$$

Taking the norm on both sides and using the triangle inequality (2.1.4) we have

$$\begin{aligned}\|u_{t+1}(i)\| &\leq \|(1-dt)u_t(i)\| + \left\|\frac{dt}{d(i)}\sum_{j=1}^{m} W(i,j)u_t(j)\right\| \\ &\leq |1-dt|\|u_t(i)\| + \frac{dt}{d(i)}\sum_{j=1}^{m} W(i,j)\|u_t(j)\| \\ &\leq \left(\max_{1\leq j\leq m}\|u_t(j)\|\right)\left(|1-dt| + \frac{dt}{d(i)}\sum_{j=1}^{m} W(i,j)\right) \\ &\leq \left(\max_{1\leq j\leq m}\|u_t(j)\|\right)(|1-dt|+dt)\,.\end{aligned}$$

In order to ensure the amplification factor is less than or equal to 1 at each step, we need

$$|1-dt| + dt \leq 1.$$

This is equivalent to

$$-(1-dt) \leq 1-dt \leq 1-dt.$$

The upper inequality is trivial, and the lower one yields $0 \leq 2(1-dt)$ or $dt \leq 1$. In this case we have

$$\|u_{t+1}(i)\| \leq \max_{1\leq j\leq m}\|u_t(j)\|.$$

Continuing by induction we arrive at (8.2.6), which completes the proof. □

Remark 8.2.3. We may wish to go beyond stability and instead prove convergence of the iterations as $k \to \infty$ to a solution of the equation $\nabla E = 0$, that is

$$\begin{cases} Lu(i) = 0, & \text{if } i \in I_m \setminus \Gamma \\ u(i) = g(i), & \text{otherwise.} \end{cases} \tag{8.2.7}$$

Proving convergence is more involved, since it depends on other properties of the graph that we have not discussed. If the graph is connected, then the solution of (8.2.7) is unique and gradient descent converges to this solution. However, when the graph is not connected, the equation (8.2.7) may not have a unique solution for gradient descent to converge to (it may have infinitely many solutions). Gradient descent can still be shown to converge, but the limit is dependent on the initial condition u_0.

10	20	40	80	160
85.4 (4.4)	91.7 (1.2)	93.4 (0.5)	94.3 (0.3)	94.8 (0.1)

Table 8.2.1: Laplace learning on MNIST with 10, 20, 40, 80, and 160 labels per class. The average (standard deviation) classification accuracy over 100 trials is shown.

To see that the solution of (8.2.7) exists and is unique when the graph is connected, recall that by the Rank-Nullity Theorem we only need to show that solutions are unique. Take two solutions u, v of (8.2.7) and subtract them $w = u - v$. Then w satisfies $Lw = 0$ on $I_m \setminus \Gamma$ and $w = 0$ on Γ. We then compute

$$\begin{aligned} 0 = \sum_{i=1}^{m} w(i)^T L w(i) &= \sum_{i=1}^{m} w(i)^T \sum_{j=1}^{m} W(i,j)(w(i) - w(j)) \\ &= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j) \|w(i) - w(j)\|^2, \end{aligned}$$

where the last line follows a similar argument as Exercise 8.2.1. It follows that $w = 0$, and so solutions are unique.

We now give a short example application to the classification of MNIST digits. Here, we use a $k = 10$ nearest neighbor graph, in the same was as in Section 4.2.2, using Euclidean distance between pixel values. We experimented with different label rates, randomly choosing 10, 20, 40, 80, and 160 labels per class (equivalent to 100, 200, 400, 800, and 1600 labels). For each label rate, we ran 100 trials randomizing which images are labeled. Table 8.2 shows the results of the experiment. We can see the method gives good accuracy results even with only a hanful of training examples, showing the power of semi-supervised learning.

8.3 Graph-based embeddings

Python Notebook: .ipynb

Another application of graph-based learning is dimension reduction, which is used to simplify high dimensional data by embedding it into a much lower dimensional Euclidean space while preserving important properties of the

dataset. We are again in the graph-based setting in this section and we assume we have an $m \times m$ weight matrix W representing our dataset.

8.3.1 Spectral embedding

One of the oldest and most widely used embeddings is the spectral embedding, which uses the eigenvectors of the graph Laplacian matrix $L = D - W$, where we recall from Sections 4.2 and 8.2 that D is the diagonal matrix with degrees $d(i) = \sum_{j=1}^{m} W(i,j)$ on the diagonal. Spectral embeddings are the foundation of the dimension reduction techniques in diffusion maps [6], Laplacian eigenmaps [1], and spectral clustering [16].

Let $v_1, v_2, v_3, \ldots$ be the normalized eigenvectors of L, in order of increasing eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \cdots$. The spectral embedding corresponding to L is the map $\Phi : I_m \to \mathbb{R}^k$ (recall $I_m = \{1, 2, \ldots, m\}$ are the indices of our datapoints) given by

$$\Phi(i) = (v_1(i), v_2(i), \ldots, v_k(i)). \tag{8.3.1}$$

Since the first eigenvector v_1 is the trivial constant eigenvector, it is also common to omit this to obtain the embedding

$$\Phi(i) = (v_2(i), v_3(i), \ldots, v_{k+1}(i)).$$

There are other normalizations of the graph Laplacian that are commonly used, such as the symmetric normalization $L = D^{-1/2}(D - W)D^{-1/2}$, and the spectral embedding for a normalized Laplacian is defined analogously. Figure 8.3.1 shows a spectral embedding of the MNIST digits 0, 1, and 2 using both the unnormalized and normalized graph Laplacians into $k = 2$ dimensions. In this case, we omitted v_1. Both normalizations achieve good separation between the classes in the embedding.

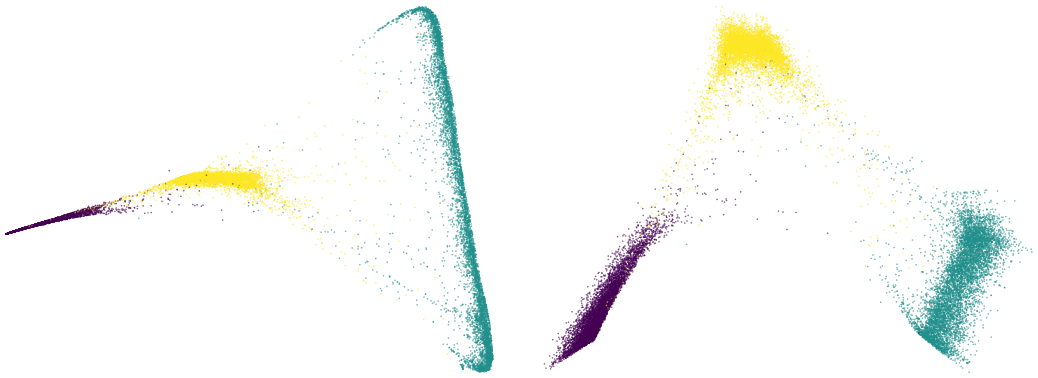
The intuition behind the spectral embedding is encapsulated in the following simple result.

Proposition 8.3.1. *If $A \subset I_m$ is a disconnected component of the graph, which means that $W(i,j) = 0$ for all $i \in A$ and $j \in I_m \setminus A$, then the indicator function of A, denoted u_A, satisfies*

$$Lu_A = (D - W)u_A = 0.$$

Proof. The indicator function u_A satisfies $u_A(i) = 1$ if $i \in A$ and $u_A(i) = 0$ otherwise. For $i \in A$ we compute

$$Lu_A(i) = \sum_{j=1}^{m} W(i,j)(u_A(i) - u_A(j)) = \sum_{j \in A} W(i,j)(u_A(i) - u_A(j)) = 0,$$



(a) Unnormalized (b) Normalized

Figure 8.3.1: Example of spectral embeddings of the 0, 1, and 2 digits of the MNIST dataset using the unnormalized $L = D - W$ and symmetric normalized $L = D^{-1/2}(D - W)D^{-1/2}$ graph Laplacians.

since A is a disconnected component of I_m. Similarly, if $i \notin A$, then

$$Lu_A(i) = \sum_{j=1}^{m} W(i,j)(u_A(i) - u_A(j)) = \sum_{j \notin A} W(i,j)(u_A(i) - u_A(j)) = 0,$$

since $u_A(i) = u_A(j) = 0$ in the sum above. This completes the proof. □

Remark 8.3.2. Proposition 8.3.1 shows that the indicator functions of disconnected components of the graph are eigenfunctions of the graph Laplacian L with smallest eigenvalue $\lambda = 0$. In fact, the multiplicity of the zero eigenvalue is exactly the number of disconnected components of the graph. If the first k eigenvectors $v_1, v_2, \ldots, v_k$ are indicator functions of disconnected components, then the spectral embedding (8.3.1) boils down to $\Phi(i) = e_j$ when i belongs to the j^{th} cluster. So the spectral embedding maps the clusters to the one hot vectors representing them. Of course, in practice data is not perfectly clustered already, and a graph may have no disconnected components. Furthermore, when we compute the eigenvectors v_i, we may get any arbitrary rotation of the eigenvectors; eigenvectors are by no means unique when the eigenmode has multiplicity greater than 1. Spectral embeddings work because they are stable under perturbations of the ideal setting, and give approximately ideal embeddings for clusters that are nearly separated.

8.3.2 t-SNE embedding

We now briefly discuss another graph-based embedding called t-distributed stochastic neighbor embedding (t-SNE), which is a method for visualizing high dimensional data originally proposed by van der Maaten and Hinton [18]. In the past few years t-SNE has become extremely popular in the natural sciences (e.g., gene analysis and math biology) for visualizing data and discovering cluster structure. The mathematical theory behind t-SNE is currently far from complete.

The starting point for t-SNE is an $m \times m$ weight matrix W. As usual, the weight matrix is assumed to be nonnegative, and $W(i,j)$ denotes the similarity between nodes i and j. We do not assume that W is symmetric; it may be the case that W represents a k-nearest neighbor graph, where $W(i,j) = 1$ if j is one of the k nearest neighbors of i. We assume the diagonals of W vanish, so $W(i,i) = 0$. We then construct a matrix P by normalizing and symmetrizing W as follows

$$P = \frac{1}{2m}(D^{-1}W + W^T D^{-1}), \tag{8.3.2}$$

where D is the diagonal matrix of degrees $d(i) = \sum_{j=1}^{m} W(i,j)$. We note that the sum of all entries in P is one, that is $\mathbf{1}^T P \mathbf{1} = 1$. That is, we can think of P as a probability distribution. To see this we note that the diagonal of D is exactly $W\mathbf{1}$, and thus $D^{-1}W\mathbf{1} = \mathbf{1}$ and hence $\mathbf{1}^T W^T D^{-1} = \mathbf{1}^T$. Therefore

$$\mathbf{1}^T P \mathbf{1} = \frac{1}{2m}(\mathbf{1}^T D^{-1} W \mathbf{1} + \mathbf{1}^T W^T D^{-1} \mathbf{1}) = \frac{1}{2m}(\mathbf{1}^T \mathbf{1} + \mathbf{1}^T \mathbf{1}) = 1.$$

Note also that since $W(i,i) = 0$ we have $P(i,i) = 0$. We also have that P is symmetric, so $P(i,j) = P(j,i)$.

t-SNE aims to find embedded points $y_1, y_2, \ldots, y_m \in \mathbb{R}^k$, where usually $k = 2$ or $k = 3$, so that the similarity between y_i and y_j matches $P(i,j)$ as closely as possible. The similarity matrix for the y_i, denoted Q, is the $m \times m$ matrix defined by

$$Q(i,j) = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{\ell \neq s}(1 + \|y_s - y_\ell\|^2)^{-1}}. \tag{8.3.3}$$

The sum in the denominator is over all $\ell = 1, \ldots, m$ and $s = 1, \ldots, m$ with $\ell \neq s$. Notice the entries of the matrix Q also sum to 1, so it can be interpreted as a probability distribution as well. Here, we also set $Q(i,i) = 0$.

t-SNE chooses the points y_i to minimize the distance between the probability distributions P and Q. In particular, t-SNE minimizes the Kullback-Leibler

divergence between P and Q, given by

$$E(y_1, y_2, \ldots, y_k) = D(P\|Q) := \sum_{i \neq j} P(i,j) \log\left(\frac{P(i,j)}{Q(i,j)}\right). \tag{8.3.4}$$

Again, the sum is over all $i = 1, \ldots, n$ and $j = 1, \ldots n$ with $i \neq j$. Notice that the Kullback-Leibler divergence is zero when $P = Q$. We also note that since $\lim_{x \to 0^+} x \log x = 0$, we interpret the term in the sum as zero whenever $P(i,j) = 0$, regardless of the value of $Q(i,j)$. If $Q(i,j) = 0$ and $P(i,j) \neq 0$, then the Kullback-Leibler divergence is infinite. The Kullback-Leibler divergence is always nonnegative, though this is not directly clear from the definition (it is due to Gibbs' inequality).

Let us say a few words about the choice of the Kullback-Leibler divergence, in place of some other distance between probability distributions. In fact, since $D(P\|Q)$ is not symmetric in P and Q it is even natural to wonder whether the symmetrized Kullback-Leibler divergence $D(P\|Q) + D(Q\|P)$ would be more suitable. The choice of the non-symmetric Kullback-Leibler is actually quite intentional in t-SNE. The Kullback-Leibler divergence places a strong emphasis on ensuring that $Q(i,j)$ is positive and matches $P(i,j)$ well whenever $P(i,j) \gg 0$. That is t-SNE emphasizes preserving local structure from the high dimensional space in the embedded space. Of course we cannot preserve all structures in the embedding, since there is far more complexity in high dimensions and some information must be lost in the embedding. For example, in $d = 10$ dimensional Euclidean space we can place 11 points that are mutually equidistant, which is impossible in any lower dimension, and this structure would be removed by the embedding. Notice in the Kullback-Leibler divergence there is little to no penalty placed on discrepancy between $P(i,j)$ and $Q(i,j)$ when $P(i,j) = 0$ or $P(i,j) \ll 1$. Thus, t-SNE does not seek to ensure that points far apart in the original high dimensional space remain far apart in the embedding. This is the relaxation that allows the embedding to give meaningful and useful results, even though two or three dimensional space cannot capture all of the details and nuances of high dimensional data.

t-SNE uses gradient descent to minimize E, starting from a random initial configuration. The only place the y_i appear is through Q, so we may as well simplify the energy to read

$$E(y_1, y_2, \ldots, y_k) = -\sum_{i \neq j} P(i,j) \log Q(i,j),$$

at least for the purposes of computing ∇E. By the definition of Q in (8.3.3)

we can split E into two terms

(8.3.5)

$$E(y_1, y_2, \ldots, y_k) = \sum_{i \neq j} P(i,j) \log(1 + \|y_i - y_j\|^2) + \log\left(\sum_{i \neq j} (1 + \|y_i - y_j\|^2)^{-1}\right).$$

Note we used that $\sum_{i \neq j} P(i,j) = 1$ above. The gradient of E has a component in the direction of each y_i, which we denote by $\nabla_{y_i} E$. We compute the gradient ∇_{y_ℓ} of both terms above separately. Throughout the computation let δ_{ij} denote the Kronecker delta, satisfying $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise. We now compute, via the chain rule, that

$$\begin{aligned}
&\nabla_{y_\ell} \sum_{i \neq j} P(i,j) \log(1 + \|y_i - y_j\|^2) \\
&\quad = \sum_{i \neq j} P(i,j) \nabla_{y_\ell} \log(1 + \|y_i - y_j\|^2) \\
&\quad = \sum_{i \neq j} P(i,j)(1 + \|y_i - y_j\|^2)^{-1} \nabla_{y_\ell} \|y_i - y_j\|^2.
\end{aligned}$$

If $\ell = i$ then by (2.4.2) we have

$$\nabla_{y_\ell} \|y_i - y_j\|^2 = 2(y_i - y_j).$$

Similarly, if $\ell = j$ then

$$\nabla_{y_\ell} \|y_i - y_j\|^2 = \nabla_{y_\ell} \|y_j - y_i\|^2 = 2(y_j - y_i).$$

If $\ell \neq i$ and $\ell \neq j$, then $\nabla_{y_\ell} \|y_i - y_j\|^2 = 0$. Therefore we have

$$\begin{aligned}
&\nabla_{y_\ell} \sum_{i,j:i \neq j} P(i,j) \log(1 + \|y_i - y_j\|^2) \\
&\quad = 2 \sum_{i,j:i \neq j} P(i,j)(1 + \|y_i - y_j\|^2)^{-1} (\delta_{\ell i} - \delta_{\ell j})(y_i - y_j) \\
&\quad = 2 \sum_{j:j \neq \ell} P(\ell, j)(1 + \|y_\ell - y_j\|^2)^{-1} (y_\ell - y_j) \\
&\qquad\quad - 2 \sum_{i:i \neq \ell} P(i, \ell)(1 + \|y_i - y_\ell\|^2)^{-1} (y_i - y_\ell) \\
&\quad = 4 \sum_{j:j \neq \ell} P(\ell, j)(1 + \|y_\ell - y_j\|^2)^{-1} (y_\ell - y_j).
\end{aligned}$$

Note that we are indexing in the sums above which variables are summed over. In the first lines, the sums are over both i and j, while in the remaining lines the sums are over a single index, either i or j, but not both. Letting Z denote the normalization factor $Z = \sum_{i \neq j}(1 + \|y_i - y_j\|^2)^{-1}$ we have

$$(1 + \|y_\ell - y_j\|^2)^{-1} = ZQ(\ell, j). \tag{8.3.6}$$

This gives a simpler form for the gradient of the first term, namely

$$\nabla_{y_\ell} \sum_{i,j:i \neq j} P(i,j) \log(1 + \|y_i - y_j\|^2) = 4Z \sum_{j:j \neq \ell} P(\ell, j)Q(\ell, j)(y_\ell - y_j). \tag{8.3.7}$$

For the gradient of the second term in (8.3.5), we simply compute

$$\begin{aligned}
&\nabla_{y_\ell} \log \left(\sum_{i \neq j} (1 + \|y_i - y_j\|^2)^{-1} \right) \\
&\quad = Z^{-1} \sum_{i \neq j} \nabla_{y_\ell} (1 + \|y_i - y_j\|^2)^{-1} \\
&\quad = -Z^{-1} \sum_{i \neq j} (1 + \|y_i - y_j\|^2)^{-2} \nabla_{y_\ell} \|y_i - y_j\|^2 \\
&\quad = -2Z^{-1} \sum_{i \neq j} (1 + \|y_i - y_j\|^2)^{-2} (\delta_{\ell i} - \delta_{\ell j})(y_i - y_j) \\
&\quad = -2Z \sum_{i \neq j} Q(i,j)^2 (\delta_{\ell i} - \delta_{\ell j})(y_i - y_j),
\end{aligned}$$

where we used the identity (8.3.6) in the final line. Using an argument similar to the first part, evaluating the Kronecker deltas, we arrive at the gradient for the second piece as

$$\nabla_{y_\ell} \log \left(\sum_{i \neq j} (1 + \|y_i - y_j\|^2)^{-1} \right) = -4Z \sum_{j:j \neq \ell} Q(\ell, j)^2 (y_\ell - y_j), \tag{8.3.8}$$

Combining (8.3.7) and (8.3.8), and replacing ℓ with i, we have

$$\nabla_{y_i} E = 4Z \sum_{j:j \neq i} P(i,j)Q(i,j)(y_i - y_j) - 4Z \sum_{j:j \neq i} Q(i,j)^2 (y_i - y_j). \tag{8.3.9}$$

The first term in the gradient is an attraction term, which pulls the points y_i and y_j together when their weights $P(i,j)$ and $Q(i,j)$ are similar and large.

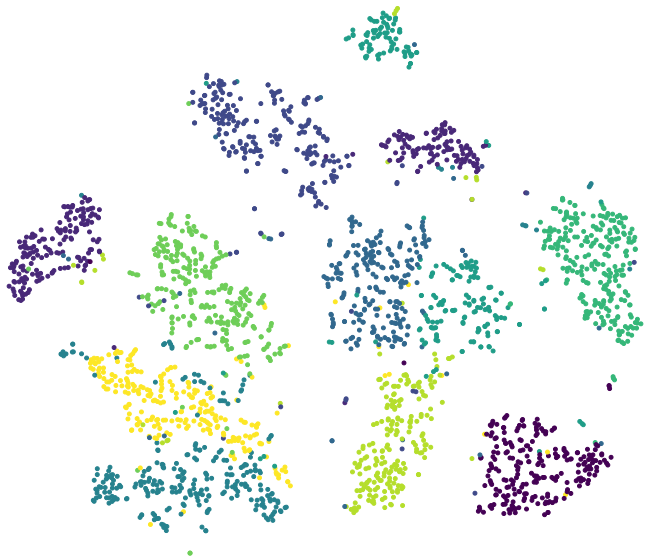


Figure 8.3.2: A t-SNE embedding of 2500 images from the MNIST dataset, with colors corresponding to the digit labels of each image.

The second term is a repulsion term that attempts to spread out nearby points. The terms in the gradient can be combined to simplify its form as

$$\nabla_{y_i} E = 4Z \sum_{j:j\neq i} (P(i,j) - Q(i,j))Q(i,j)(y_i - y_j),$$

but the separation in terms of attraction and repulsion terms is convenient.

The t-SNE energy E is minimized by gradient descent

$$y_i^{k+1} = y^k - h\nabla_{y_i} E(y_1^k, y_2^k, \ldots, y_m^k),$$

where $h > 0$ is the time step. We show in Figure 8.3.2 an example of a t-SNE embedding of 2500 images from the MNIST dataset. We can see that the embedding preserves the structure (e.g., the classes) in the dataset.

For moderate and larger numbers of points m, gradient descent is very slow to converge, so it was proposed in [18] to exaggerate the attractive forces at the beginning of gradient descent, called *early exaggeration*. For the first few hundred iterations of gradient descent, the gradient $\nabla_{y_i} E$ is replaced by

$$\nabla_{y_i} E = 4Z\alpha \sum_{j:j\neq i} P(i,j)Q(i,j)(y_i - y_j) - 4Z \sum_{j:j\neq i} Q(i,j)^2(y_i - y_j),$$

where $\alpha > 1$ is the early exaggeration amplification factor, often chosen around $\alpha = 10$. The early exaggeration period strongly favors attraction forces, and

(a) After early exaggeration (b) Final embedding

Figure 8.3.3: An example of the t-SNE embedding after the early exaggeration phase and the final embedded, for a small version of MNIST with only 500 images from the digits 0, 1, 2, and 3.

quickly begins to form clusters in the embedded space. The later phase of gradient descent, after early exaggeration, restores the repulsive forces and spreads out the clusters more evenly. Figure 8.3.3 shows the results of a t-SNE MNIST embedding of 500 images of the digits 0, 1, 2, adn 3 at the end of early exaggeration, and at the end of the t-SNE embedding procedure.

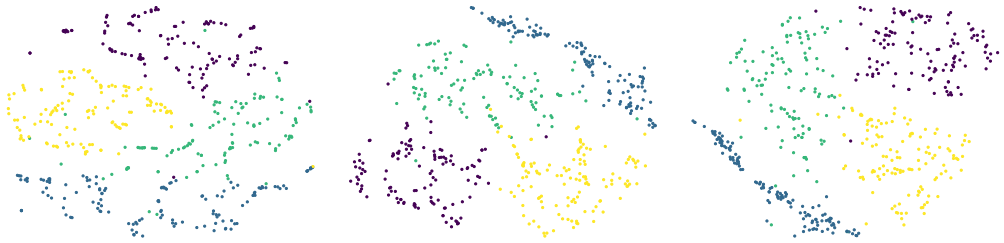
Up to now, we have not discussed how to construct the weight matrix W for t-SNE, and have assumed it is given (recall P is constructed from W in (8.3.2)). It turns out this is important for achieving useful results with t-SNE. The construction used in [18] has the form

$$W(i,j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right),$$

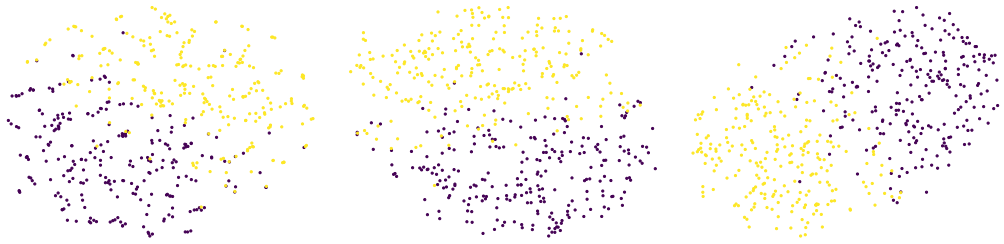
where σ_i is tuned independently for each x_i, in a similar way as a k-nearest neighbor graph. The value of σ_i is tuned to a specified *perplexity* level, usually in the range 5 to 50. The perplexity of the i^{th} row of $W(i,j)$ is $2^{H(i)}$, where

$$H(i) = -\sum_{j=1}^{m} p(j)\log p(j)$$

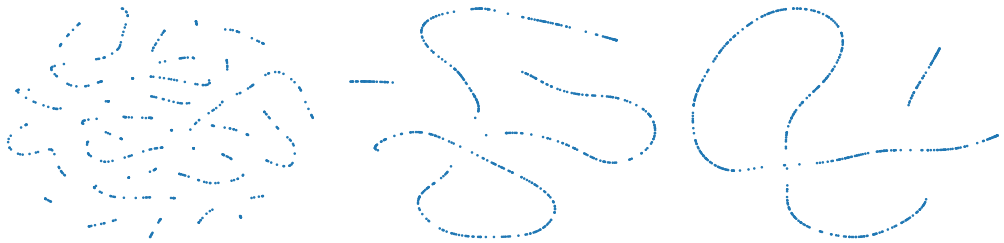
is the entropy of the distribution p given by $p(j) = W(i,j)/\sum_{k=1}^{m} W(i,k)$. The value of σ_i is determined so that the perplexity $2^{H(i)}$ equals a desired user-specified value. The entropy H measures how well-spread out a probability distribution is. A uniform distribution has high entropy and a dirac



(a) MNIST



(b) Mixture of Gaussians

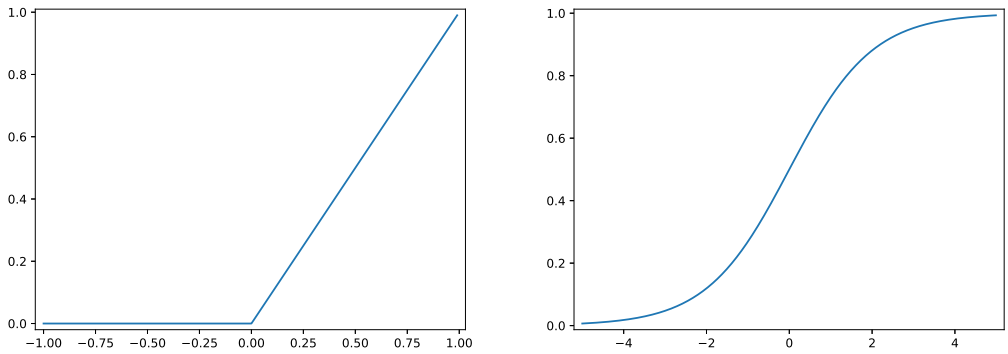


(c) Parabolic curve

Figure 8.3.4: t-SNE embeddings of a subset of MNIST, a mixture of two Gaussians in 10 dimensions, and a parabolic curve in 5 dimensions. The perplexity values from left to right are 5, 30 and 50. The mixture of Gaussians are two standard normal distributions translated a distance of $\sqrt{10}$ apart in 10 dimensional Euclidean space, while the parabolic curve is defined by $x_5 = x_4 = x_3 = x_2 = x_1^2$ for $x_1 \in [0, 1]$.

distribution has low entropy. In particular, the entropy $H(i)$ is monotonically increasing with σ_i, and so a bisection search can be used to find σ_i. Larger values of σ_i are roughly equivalent to using a larger neighborhood in the graph construction.

To give an idea of how t-SNE works on other datasets, we show in Figure 8.3.4 the results of applying t-SNE to a mixture of Gaussians distribution, with two standard normal point clouds in 10 dimensional space separated by a distance of $\sqrt{10}$, and a parabolic curve $x_5 = x_4 = x_3 = x_2 = x_1^2$ for $x_1 \in [0, 1]$



(a) ReLU (b) Sigmoid

Figure 8.4.1: Plots of the ReLU and Sigmoid activation functions. Both activation functions have the behavior that they give zero, or close to zero, responses when the input is below a certain threshold, and give positive responses above.

in 5 dimensional Euclidean space. Clearly the embedding does not preserve properties of the curve, like its connectivity or curvature information.

8.4 Neural networks

This section is a very brief introduction to artificial neural networks and deep learning. We will cover fully connected networks, convolutional neural networks, and some basic theory including the back propagation equations for computing gradients, and universal function approximation theorems. For a more in depth view of deep learning we refer the reader to [11].

8.4.1 Fully connected networks

Artificial neural networks, which we will call neural networks from now on, are parameterized functions made up of simple building blocks: linear functions and simple nonlinearities. The basic building block is a neuron

$$f(x) = \sigma(\omega^T x + b),$$

which is a function $f : \mathbb{R}^n \to \mathbb{R}$ consisting of a linear function $x \mapsto \omega^T x + b$ (here, $\omega \in \mathbb{R}^n$ is the weight and $b \in \mathbb{R}$ is the bias), composed with a nonlinear activation function $\sigma : \mathbb{R} \to \mathbb{R}$. A common choice for the activation function is the rectified linear unit (ReLU)

$$\sigma(t) = \max\{t, 0\}. \tag{8.4.1}$$

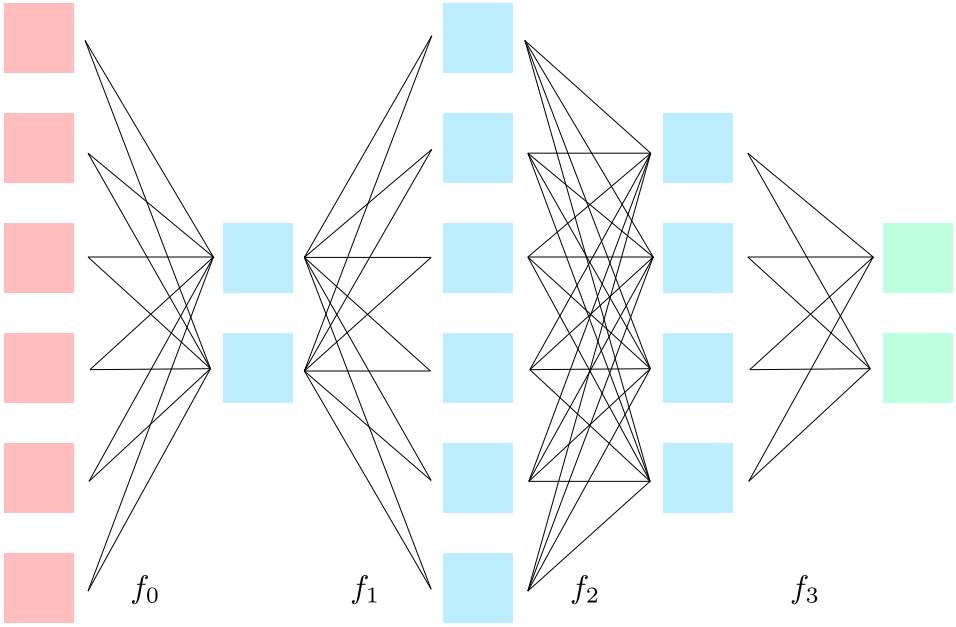


Figure 8.4.2: An example of a fully connected neural network with three hidden layers. The blue nodes are the hidden layers, the red is the input, and the green is the output. The hidden layers have width $n_1 = 2$, $n_2 = 6$, and $n_3 = 4$ and the number of input variables is $n_0 = 6$.

Another popular choice is the *sigmoid* activation

$$\sigma(t) = \frac{1}{1 + e^{-t}}. \tag{8.4.2}$$

Figure 8.4.1 shows both activation functions. The key property of an activation functions is that the response is zero, or close to zero, for inputs below a certain threshold, and then positive (i.e., *activated*) above a threshold. This model for a neuron is very loosely based on neurons in human and animal brains, which are activated only when their input rises to a certain threshold, but the analogy should not be taken further than this. Let us briefly mention that the advantage of the sigmoid activation is that it is continuously differentiable (in fact, smooth), while ReLU activations are Lipschitz continuous but not differentiable at $t = 0$. On the other hand, the ReLU activations are 1-homogeneous, meaning $\sigma(at) = a\sigma(t)$ for $a > 0$.

A fully connected neural network is an interconnection of many neurons organized into a number of layers, so that each neuron in the k^{th} layer takes as its inputs all the outputs of neurons from the $(k-1)^{\text{st}}$ layer. Figure 8.4.2 shows a toy example of a fully connected neural network. The internal layers

(the blue ones in the figure) are called *hidden layers* and the units in these layers called *hidden units*. The red layer corresponds to the input variables (in $\mathbb{R}^6$ here) and the green layer the output (in $\mathbb{R}^2$ here). A neural network is trained for a particular task by adjusting all the weights and biases in achieve correct behavior. As we shall see, the structure of a neural network makes it extremely flexible in its ability to approximate a wide range of functions with ease, which is one main reason for their success.

In more compact notation, we can write a fully connected neural network with L layers recursively as

$$f_k = \sigma_k(W_k f_{k-1} + b_k), \quad k = 1, \dots, L, \tag{8.4.3}$$

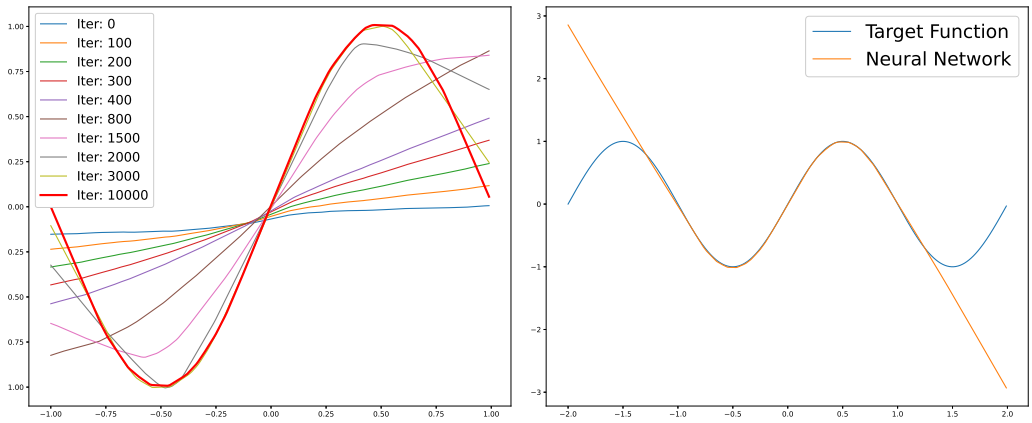
where $f_0 \in \mathbb{R}^{n_0}$ is the input to the network, $f_k \in \mathbb{R}^{n_k}$ for $k = 1, \dots, L-1$ are the values of the network at the hidden layers, f_L is the output of the neural network, and n_k is the number of hidden nodes in the k^{th} layer. Notice we are including all the neurons for the k^{th} layer in the compact and vectorized notation in (8.4.3). The weights $W_k \in \mathbb{R}^{n_k \times n_{k-1}}$ and biases $b_k \in \mathbb{R}^{n_k}$ are the learnable parameters in the neural network. The functions $\sigma_k : \mathbb{R} \to \mathbb{R}$ are the activation functions for each layer, which we allow to be different. When applying σ_k to a vector $x \in \mathbb{R}^m$, we apply σ_k componentwise to x. We note that without a *nonlinear* activation function σ_k, a neural network would simply be a linear function from $\mathbb{R}^{n_0}$ to $\mathbb{R}^{n_L}$ (being a composition of linear functions).

The output of the neural network $f_L \in \mathbb{R}^{n_L}$ is typically fed into a loss function $\mathcal{L} : \mathbb{R}^{n_L} \to \mathbb{R}$ which measures the performance of the network for the given learning task. The value of the output f_L clearly depends on the weights W_k, biases b_k, and the input to the network f_0. We write $f_L(x)$ to denote the value of the output of the network f_L given the input is $f_0 = x$. Neural networks are trained by minimizing the loss $\mathcal{L}(x_L)$ with respect to the choices of the weights W_k and biases b_k. Typically the loss has the form

$$\mathcal{L}(W_1, b_1, \dots, W_L, b_L) = \sum_{i=1}^{m} \ell(f_L(x_i), y_i), \tag{8.4.4}$$

where (x_i, y_i) for $i = 1, \dots, m$ are the training data.

The loss is normally minimized by some version of gradient descent. Let $\frac{\partial \mathcal{L}}{\partial W_k}$ and $\frac{\partial \mathcal{L}}{\partial b_k}$ denote the gradients of $\mathcal{L}$ with respect to W_k and b_k, respectively. The gradient $\frac{\partial \mathcal{L}}{\partial W_k}$ is the $n_k \times n_{k-1}$ matrix whose (i, j) entry is the partial derivative of $\mathcal{L}$ in $W_k(i, j)$. Likewise, the gradient $\frac{\partial \mathcal{L}}{\partial b_k} \in \mathbb{R}^{n_k}$ is the vector whose i^{th} entry is the partial derivative of $\mathcal{L}$ in $b_k(i)$. Gradient descent for minimizing $\mathcal{L}$ corresponds to updating the weights W_k and biases b_k according



(a) Training iterations (b) Neural network vs target

Figure 8.4.3: A toy example of fitting the function $\sin(\pi x)$ with a 2-layer neural network with 100 hidden nodes. In (a) we show the intermediate results over 10000 steps of gradient descent on the L^1 loss, and in (b) we show the final trained network compared to the target function $\sin(\pi x)$. The L^1 loss is restricted to the interval $[-1, 1]$ and the neural network extends linearly outside of this domain (due to the choice of ReLU activations).

to

$$W_k^{j+1} = W_k^j - \alpha \frac{\partial \mathcal{L}}{\partial W_k} \quad \text{and} \quad b_k^{j+1} = b_k^j - \alpha \frac{\partial \mathcal{L}}{\partial b_k}, \tag{8.4.5}$$

where $\alpha > 0$ is the time step, also called the *learning rate*. Each step of gradient descent improves the performance of the network for the task at hand, which is interpreted as *learning*. We note that each gradient depends on the current values W_k^j and b_k^j, that is

$$\frac{\partial \mathcal{L}}{\partial W_k} = \frac{\partial \mathcal{L}}{\partial W_k}(W_1^j, b_1^j, \ldots, W_L^j, b_L^j) \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b_k} = \frac{\partial \mathcal{L}}{\partial b_k}(W_1^j, b_1^j, \ldots, W_L^j, b_L^j).$$

We suppress this dependence for notational convenience.

As a toy example we consider training a 2-layer neural network to approximate the function $\sin(\pi x)$. We use the L^1 loss

$$\mathcal{L}(f_L) = \sum_{i=1}^{m} |f_L(x_i) - \sin(\pi x_i)|$$

for evenly spaced points $-1 = x_1 \leq x_2 \leq \cdots \leq x_m = 1$. We chose $m = 200$ and used a 2-layer neural network with 100 hidden nodes and a ReLU activation

function. We ran gradient descent for 10000 iterations. Figure 8.4.3 (a) shows the intermediate steps of gradient descent and Figure 8.4.3 (b) shows the final neural network function compared to the target function $\sin(\pi x)$. Since the network is only trained on the interval $[-1, 1]$, the values do not agree with the target function outside this interval. The choice of ReLU activations causes the network to extend the function linearly outside the training domain. We also mention that the weights in the network were initialized randomly, which leads to faster convergence than zero intialization.

For modern machine learning problems with very large training sets, it is sometimes impractical to compute the full gradients $\frac{\partial \mathcal{L}}{\partial W_k}$ and $\frac{\partial \mathcal{L}}{\partial b_k}$, since the loss (8.4.4) involves *all* of the training data, which may not even fit into memory all at once. *Stochastic gradient descent (SGD)* alleviates this concern by computing the gradient over a random subset of the training data. That is, we compute the gradients in W_k and b_k of the loss

$$\widetilde{\mathcal{L}}(W_1, b_1, \ldots, W_L, b_L) = \sum_{i \in I} \ell(f_L(x_i), y_i),$$

where $I \subset \{1, 2, \ldots, n\}$ is a random subset, called a *mini-batch*. The mini-batch changes at each iteration of SGD. One pass over all the mini-batches in the dataset is called an *epoch*, and training usually proceeds for some number of ephochs, say 100.

Various other trickes are used in the optimization, and there are by now too many to properly describe. One of the more important ones is *momentum*, which modifies the gradient descent step (8.4.5) to be

$$W_k^{j+1} = W_k^j - \alpha \frac{\partial \mathcal{L}}{\partial W_k} + \beta(W_k^j - W_k^{j-1}),$$

and

$$b_k^{j+1} = b_k^j - \alpha \frac{\partial \mathcal{L}}{\partial b_k} + \beta(b_k^j - b_k^{j-1}),$$

where $\beta \in [0, 1]$ is the momentum parameter. Momentum can help to speed up convergence of gradient descent. Chapter 9 gives a rigorous analysis of gradient descent, including momentum descent and SGD.

8.4.2 Back propagation

Since the neural network is an L-fold composition of functions, the gradients can be computed with the chain rule. However, computing each gradient separately is wasteful and expensive. It turns out there are relationships between

gradients (also following from the chain rule), that can be used to efficiently compute all gradients $\frac{\partial \mathcal{L}}{\partial W_k}$ and $\frac{\partial \mathcal{L}}{\partial b_k}$ for $k = 1, \dots, L$ in one sweep of the neural network. This is known as *back propagation*. While all types of neural networks admit a type of back propagation for computing gradients, we will proceed with the analysis for the case of a fully connected neural network, as introduced in Section 8.4.1. The extension to other types of networks is straightforward.

We need some additional notation before describing the back propagation equations. Recall the fully connected neural network recursion equation (8.4.3). For notational simplicity, we will write

$$z_k = W_k f_{k-1} + b_k, \tag{8.4.6}$$

so that (8.4.3) becomes $f_k = \sigma_k(z_k)$. Let $\frac{\partial \mathcal{L}}{\partial z_k} \in \mathbb{R}^{n_k}$ denote the gradient of $\mathcal{L}$ with respect to z_k. Recall we treat all vectors as column vectors. We also let D_k be the diagonal $n_k \times n_k$ matrix with diagonal entries given by the vector $\sigma_k'(z_k)$. That is

$$D_k = \operatorname{diag}(\sigma_k'(z_k)).$$

The following back propagation result allows us to relate the gradients of L at one layer with gradients at layer $k-1$ to gradients at layer k.

Theorem 8.4.1 (Back propagation)**.** *For $k = 2, \dots, L$ we have*

$$\frac{\partial \mathcal{L}}{\partial z_{k-1}} = D_{k-1} W_k^T \frac{\partial \mathcal{L}}{\partial z_k}, \tag{8.4.7}$$

$$\frac{\partial \mathcal{L}}{\partial W_k} = \frac{\partial \mathcal{L}}{\partial z_k} f_{k-1}^T, \quad \textit{and} \quad \frac{\partial \mathcal{L}}{\partial b_k} = \frac{\partial \mathcal{L}}{\partial z_k}. \tag{8.4.8}$$

Remark 8.4.2. Theorem 8.4.1 suggests an algorithm, called *back propagation*, for efficiently computing the gradients $\frac{\partial \mathcal{L}}{\partial W_k}$ and $\frac{\partial \mathcal{L}}{\partial b_k}$. We first compute $\frac{\partial \mathcal{L}}{\partial z_L}$ based on the form of the loss function $\mathcal{L} = \mathcal{L}(\sigma_L(z_L))$ (note that σ_L is often the identity, in which case $\mathcal{L} = \mathcal{L}(z_L)$). Then we compute the gradients $\frac{\partial \mathcal{L}}{\partial z_k}$ for $k = L-1, L-2, \dots, 2, 1$ using the recursion (8.4.7). Note that the form of the recursion (8.4.7) requires we propagate backwards through the network from the last layer L to the first layer, hence the name *back propagation* (in general $D_{k-1} W_k^T$ is not invertible—it is usually not even square!—so we cannot write $\frac{\partial \mathcal{L}}{\partial z_k}$ in terms of $\frac{\partial \mathcal{L}}{\partial z_{k-1}}$ and are forced to propagate backwards). After computing $\frac{\partial \mathcal{L}}{\partial z_k}$ for all $k = 1, \dots, L$, we easily compute $\frac{\partial \mathcal{L}}{\partial W_k}$ and $\frac{\partial \mathcal{L}}{\partial b_k}$ using the identities in (8.4.8).

Proof of Theorem 8.4.1. We first establish (8.4.8). To do this, we need to compute the $(i,j)^{\text{th}}$ entry of the matrix $\frac{\partial \mathcal{L}}{\partial W_k}$, which is the partial derivative $\frac{\partial \mathcal{L}}{\partial W_k(i,j)}$. We note that in (8.4.6), the entry $W_k(i,j)$ appears only in the component $z_k(i)$, in which case we have

$$z_k(i) = \sum_{\ell=1}^{n_{k-1}} W_k(i,\ell) f_{k-1}(\ell).$$

Thus, by the chain rule we have

$$\frac{\partial \mathcal{L}}{\partial W_k(i,j)} = \frac{\partial \mathcal{L}}{\partial z_k(i)} \frac{\partial z_k(i)}{\partial W_k(i,j)} = \frac{\partial \mathcal{L}}{\partial z_k(i)} f_{k-1}(j).$$

This establishes (8.4.8).

Now, in terms of z_k, the neural network recursion (8.4.3) can be written as

$$z_k = W_k f_{k-1} + b_k = W_k \sigma_{k-1}(z_{k-1}) + b_k.$$

Writing this out in coordinates we have

$$z_k(i) = \sum_{\ell=1}^{n_{k-1}} W_k(i,\ell) \sigma_{k-1}(z_{k-1}(\ell)) + b_k(i).$$

Differentiating in $z_{k-1}(j)$ we have

$$\frac{\partial z_k(i)}{\partial z_{k-1}(j)} = W_k(i,j) \sigma'_{k-1}(z_{k-1}(j)).$$

We now use the chain rule to compute

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z_{k-1}(j)} &= \sum_{i=1}^{n_k} \frac{\partial \mathcal{L}}{\partial z_k(i)} \frac{\partial z_k(i)}{\partial z_{k-1}(j)} \\
&= \sum_{i=1}^{n_k} \frac{\partial \mathcal{L}}{\partial z_k(i)} W_k(i,j) \sigma'_{k-1}(z_{k-1}(j)) \\
&= \left(W_k^T \frac{\partial \mathcal{L}}{\partial z_k} \right)(j) \sigma'_{k-1}(z_{k-1}(j)) \\
&= \left(D_{k-1} W_k^T \frac{\partial \mathcal{L}}{\partial z_k} \right)(j),
\end{aligned}$$

which completes the proof. □

Exercise 8.4.3. The traditional neural network architecture

$$f_k = \sigma_k(W_k f_{k-1} + b_k), \quad k = 1, \ldots, L,$$

often yields worse performance for deeper networks with more layers compared to shallower networks. The main issue it that training is difficult, due to vanishing gradients or gradient blowup (where the gradients either become very small and training does not progress, or become very large and training is unstable). This is not so surprising; consider the case where $\sigma_k(t) = t$ is the identity and the biases $b_k = 0$ vanish. Then

$$f_L(x) = W_L W_{L-1} \cdots W_2 W_1 x.$$

The L-fold product is very sensitive to the spectral norms of the matrices; when the eigenvalues are larger than one in magnitude it blows up exponentially, while when they are less than one it decay to zero exponentially.

The *Residual Neural Network (ResNet)* architecture [12] is a recent development in deep learning that solves this problem by changing the architecture to

$$f_k = f_{k-1} + W_{k,1}\sigma_k(W_{k,2} f_{k-1} + b_k), \quad k = 1, \ldots, L. \tag{8.4.9}$$

The idea is to have each layer learn the *residual* $f_k - f_{k-1}$, which allows the network to easily skip layers, by setting $f_k = f_{k-1}$. Thus, a deeper network with ResNet architecture should not perform worse than a shallower network. The ResNet architecture should remind you of the discretization of an ordinary differential equation (ODE), and many recent works have exploited this to explain the stability of ResNet.

Each ResNet layer has two weight matrices $W_{k,1}$ and $W_{k,2}$ and a bias b_k. Derive the back propagation equations to compute $\frac{\partial \mathcal{L}}{\partial W_{k,1}}$, $\frac{\partial \mathcal{L}}{\partial W_{k,2}}$, and $\frac{\partial \mathcal{L}}{\partial b_k}$ for ResNet. You should closely follow Theorem 8.4.1 with appropriate changes for ResNet. You can assume that all layers have the same number of hidden units so that f_k and f_{k-1} have the same dimensions. △

8.4.3 Classification with neural networks

Python Notebook: .ipynb

We briefly discuss the application of fully connected neural networks to classification problems. For a k-class classification problem, the output of the neural network $f_L(x)$ has k components, so $f_L(x) \in \mathbb{R}^k$ for x the input of the

network. In other words, the number of nodes in the last layer is $n_L = k$. Recall that our label vectors are given as one hot vectors $e_1, \ldots, e_k$ in $\mathbb{R}^k$ (i.e., the standard basis vectors), where e_i represents the i^{th} class. The classification of x is taken to be the largest component of $f_L(x)$; in particular, we do not need f_L to exactly fit the one-hot label vectors, and just need the largest components to be correct, which is an easier task.

In order to normalize the output of the network, normally $f_L(x)$ is fed into a soft-max function. The output of the neural network $f_L(x)$ applied to a training dataset of m data points $x_1, x_2, \ldots, x_m$ consists of m vectors each of length k (for SGD, m is the number of training data points in the current mini-batch). Let $z_1, \ldots, z_m \in \mathbb{R}^k$ denote these output vectors, so $z_i = f_L(x_i)$. The soft-max function converts these into probability vectors $p_1, \ldots, p_m \in \mathbb{R}^k$ given by

$$p_i(j) := \frac{e^{z_i(j)}}{\sum_{q=1}^k e^{z_i(q)}}.$$

The loss used for classification is normally the negative log likelihood loss. Letting $y_1, \ldots, y_m \in \mathbb{R}^k$ denote the one hot vectors representing the classes of the training data, the negative log likelihood loss is

$$\mathcal{L}(f_L) = -\sum_{i=1}^m y_i^T \log(p_i). \tag{8.4.10}$$

Letting $\ell_i \in \{1, \ldots, k\}$ denote the class of node i, which is just the position of the 1 in the one-hot vector y_i, we can write the loss as

$$\begin{aligned}
\mathcal{L}(f_L) &= -\sum_{i=1}^m \log(p_i(\ell_i)) \\
&= -\sum_{i=1}^m \log\left(\frac{e^{z_i(\ell_i)}}{\sum_{j=1}^k e^{z_i(j)}}\right) \\
&= -\sum_{i=1}^m z_i(\ell_i) + \sum_{i=1}^m \log\left(\sum_{j=1}^k e^{z_i(j)}\right) \\
&= -\sum_{i=1}^m z_i \cdot y_i + \sum_{i=1}^m \log\left(\sum_{j=1}^k e^{z_i(j)}\right) \\
&= -\sum_{i=1}^m f_L(x_i)^T y_i + \sum_{i=1}^m \log\left(\sum_{j=1}^k e^{f_L(x_i)^T e_j}\right).
\end{aligned}$$

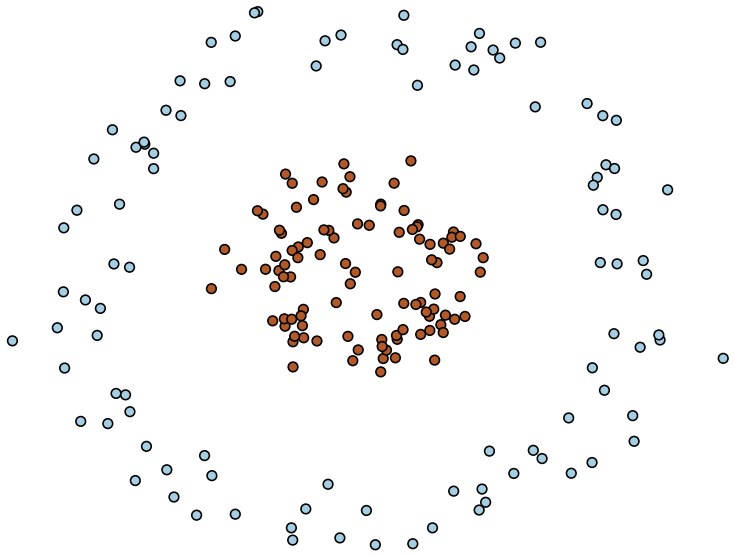
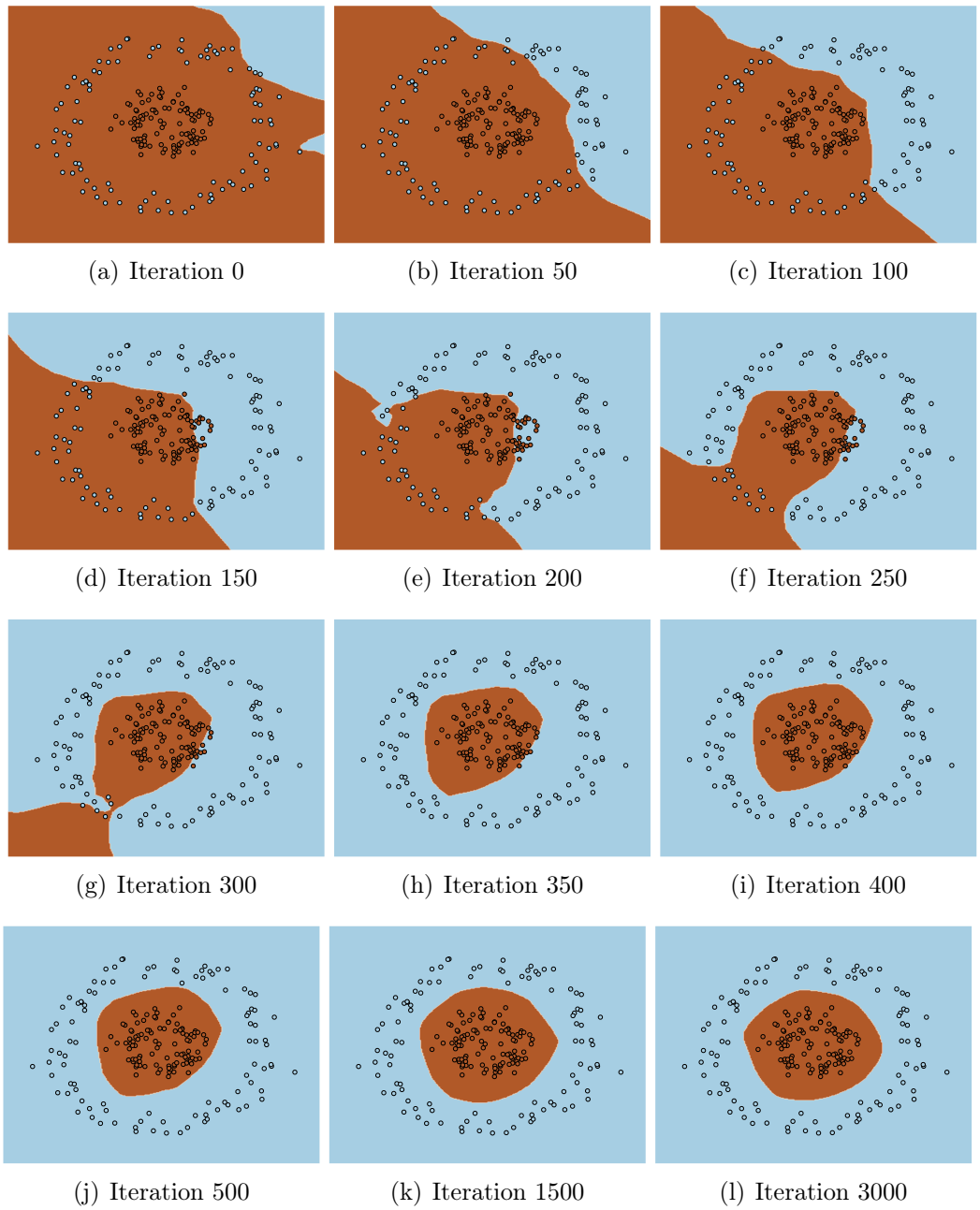


Figure 8.4.4: Synthetic data on rings consisting of two classes.

The loss thus breaks into two terms. Minimizing the first term attempts to maximize $f_L(x_i)^T y_i$, which attempts to make the output of the network $f_L(x_i)$ align as closely as possible with the one-hot label vector y_i. Minimizing the second term acts to normalize the loss to ensure it is bounded below and cannot be minimized simply by scaling f_L. We also note that the soft-max and negative log-likelihood are chosen to be compatible, so that the log and exponential cancel out. In practice, the computation is done using the last line above, leading to better numerical stability.

For our first test, we consider a toy synthetic classification problem with the data given in Figure 8.4.4. The data consists of two classes that are not linearly separable. We used a 2-layer neural network with 100 hidden nodes and trained using 3000 iterations of full batch gradient descent with the negative log likelihood loss. We show in Figure 8.4.5 the evolution of the decision boundary over the training iterations, showing how it adapts to the training data. The final decision boundary attains a good margin between the two classes.

For our next test, we consider classification of MNIST digits. For a simple first experiment, we used a 2-layer neural network with 10 hidden nodes. After 1000 iterations of full batch gradient descent the testing accuracy was around 93% and did not change much with further training. One might naively think that the 10 hidden layers $w_0, \ldots, w_9$ would learn the 10 MNIST digits $0, 1, \ldots, 9$. However, in Figure 8.4.6 we show the weights in the hidden lay-



(a) Iteration 0 (b) Iteration 50 (c) Iteration 100

(d) Iteration 150 (e) Iteration 200 (f) Iteration 250

(g) Iteration 300 (h) Iteration 350 (i) Iteration 400

(j) Iteration 500 (k) Iteration 1500 (l) Iteration 3000

Figure 8.4.5: Example of the evolving decision boundary during training of a 2-layer neural network to classify the data given in Figure 8.4.4.

ers as 2D images. None of them bear any resemblance to an MNIST digit. Instead, the network appears to be learning to look for pixels in particular

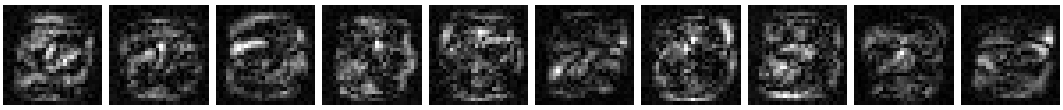


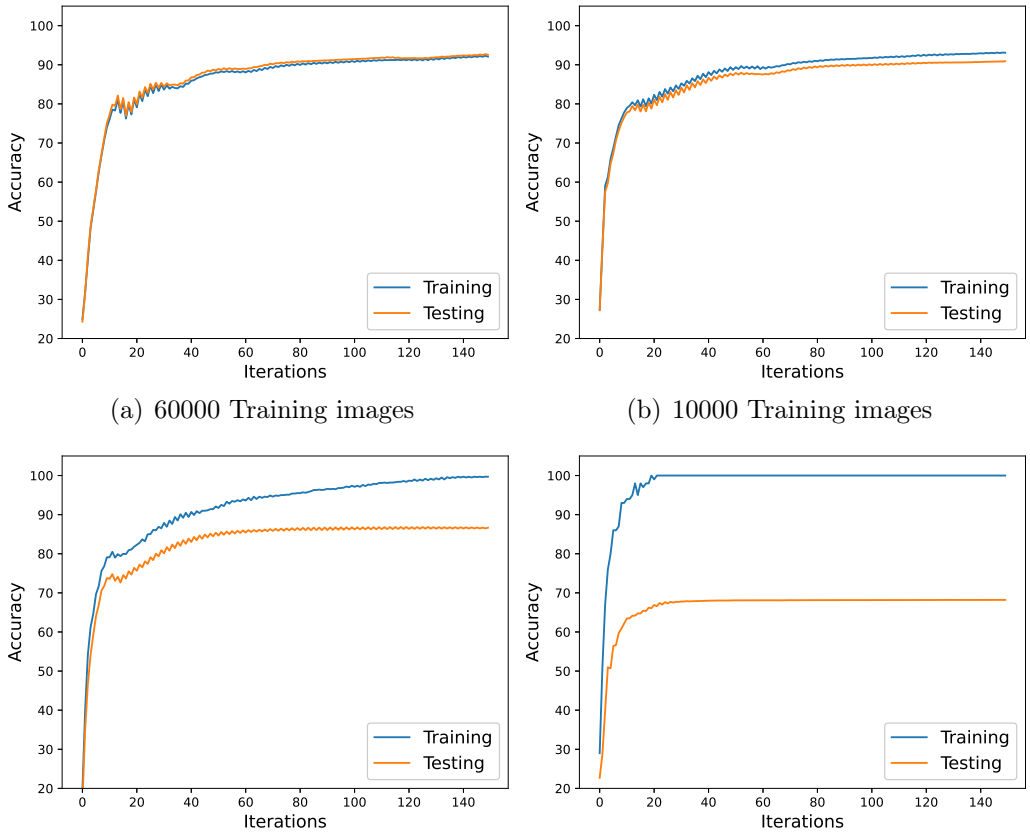
Figure 8.4.6: The 10 hidden nodes for MNIST classification. While one might naively think that each hidden layer would resemble one of the MNIST digit classes, the reality is far different, leading to issues with interpretability of the classifier.

locations in each image to determine the class. This leads to difficulties with interpretability of the neural network classifier, and in this case is a form of severe overfitting. For example, if we shift the images slightly then the pixels for each digit will be in new locations and the classifier will perform poorly. As a test, we shifted the images in the test set to the right, and the testing accuracy decreased from 93% to 87% with a single pixel shift, and down to 62% with a 2-pixel shift! Convolutional neural networks (CNN), introduced in Section 8.4.5 use a multi-resolution analysis and pooling, which introduces translation invariance and can remedy this problem to some degree (though CNNs are not invariant to rotations or scalings).

Of course, one can get better results by using more hidden nodes (though not better in terms of the overfitting phenomenon described above). Using a 2-layer network with 32 hidden layers trained for 1000 iterations of full batch gradient descent yields around 97% accuracy for both training and testing, indicating there is no overfitting. In Figure 8.4.7 (a) we show a plot of how the training and testing accuracy evolve over training. If we decrease the amount of training data, this moderately sized network is easily able to overfit the training data. In Figure 8.4.7 (b), (c), and (d) we show the same results for 10000, 1000, and 100 training images. For 1000 and 100 training images, the training accuracy hits 100% fairly quickly and the testing accuracy levels off at a much smaller value, indicating the network is overfitting the training data. In general, it is difficult to train classifiers with a limited amount of training data.

8.4.4 Universal approximation

One reason for the general effectiveness of neural networks is their ability to approximate any continuous function to arbitrary accuracy (with a sufficient number of units). This is called *universal approximation*. We will cover the basic theory for functions of a scalar variable (i.e., one dimension), in order to easily illustrate the main ideas.



(a) 60000 Training images

(b) 10000 Training images

(c) 1000 Training images

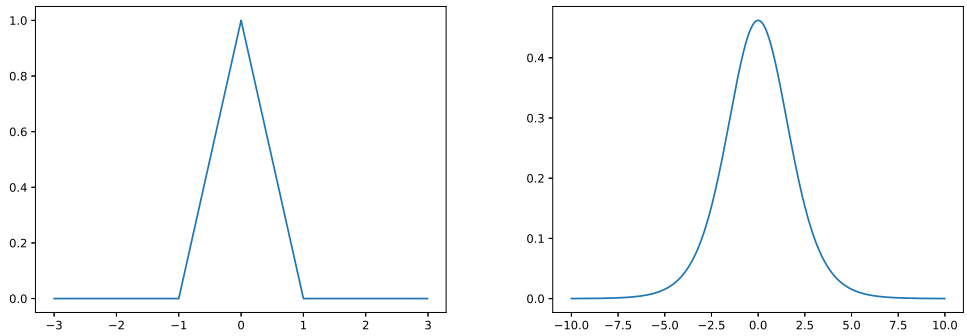
(d) 100 Training images

Figure 8.4.7: Plots of the training and testing accuracy during training on the MNIST dataset with different training set sizes. For 60000 training images, the training and testing accuracy are very similar, indicating there is no overfitting. We see a small amount of overfitting with 10000 training images, and much more with 1000 and 100 training images. The results change very little beyond the 140 iterations shown in the figures.

We will show that a 2-layer neural network with ReLU activations and N hidden units, given by

$$f_N(x) = \sum_{i=1}^{N} a_i(w_i x + b_i)_+, \tag{8.4.11}$$

can approximate any continuous function $u : \mathbb{R} \to \mathbb{R}$, given a sufficient number of parameters n. The tunable weights in the network are $a_1, a_2, \ldots, a_N \in \mathbb{R}$, $b_1, b_2, \ldots, b_N \in \mathbb{R}$, and $w_1, w_2, \ldots, w_N \in \mathbb{R}$, and $a_+ = \max\{a, 0\}$ is the ReLU



(a) ReLU bump (b) Sigmoid bump

Figure 8.4.8: Examples of bump functions constructed from 2-layer neural networks with ReLU and sigmoid activations. The sigmoid bump is constructed differently than the ReLU bump in Theorem 8.4.4 and requires only 2 hidden nodes, namely $g(x) = \sigma(x+1) - \sigma(x-1)$, since the sigmoid activation, see Figure 8.4.1, is bounded and does not grow linearly like ReLU.

activation.

We say a function $u : \mathbb{R} \to \mathbb{R}$ is *Lipschitz continuous* if there exists $C > 0$ such that

$$|u(x) - u(y)| \leq C|x - y|. \tag{8.4.12}$$

The smallest such constant is called the *Lipschitz constant* of u and denoted

$$\mathrm{Lip}(u) = \sup_{\substack{x,y\in\mathbb{R} \\ x\neq y}} \frac{|u(x) - u(y)|}{|x - y|}. \tag{8.4.13}$$

We now show that 2-layer ReLU networks can linearly interpolate functions from samples, leading to a universal approximation result with quantitative estimates on the number of neurons required.

Theorem 8.4.4. *Let $\varepsilon > 0$, let $u : \mathbb{R} \to \mathbb{R}$ be Lipschitz continuous, and let $R > 0$. There exists a 2-layer ReLU neural network $f_N(x)$ of the form (8.4.11) with $N = 6(R\,\mathrm{Lip}(u)\varepsilon^{-1} + 1)$ hidden nodes such that*

$$\max_{-R\leq x\leq R} |f_N(x) - u(x)| \leq \varepsilon. \tag{8.4.14}$$

Furthermore, if u' is Lipschitz continuous then we need only

$$N = 6(R\sqrt{\mathrm{Lip}(u')\varepsilon^{-1}} + 1) \tag{8.4.15}$$

hidden nodes.

Proof. Consider the bump function

$$g(x) = (x+1)_+ - 2x_+ + (x-1)_+,$$

which is depicted in Figure 8.4.8 (a). The bump function g is exactly a 2-layer ReLU neural network with 3 hidden nodes. It has 9 parameters in total, since each node has 3 parameters. Furthermore, we can obtain any dilation, shift and scaling of g with a 2-layer network with 3 hidden nodes. Indeed, for any $a, b, c \in \mathbb{R}$ we have

$$\begin{aligned} ag(b(x-c)) &= ag(bx-bc) \\ &= a(bx-bc+1)_+ - 2a(bx-bc)_+ + a(bx-bc-1)_+, \end{aligned}$$

which is again a 2-layer neural network with 3 hidden nodes. Furthermore, it is clear that a 2-layer network can form any linear combination of dilated and shifted bump functions of the form

$$\sum_{i=1}^{m} a_i g(b_i(x-c_i)). \tag{8.4.16}$$

Such a function is a 2-layer neural network with $3m$ hidden nodes and $9m$ parameters. The rest of the proof shows how to approximate u by a superposition of bump functions of the form (8.4.16).

Let $h > 0$, to be determined later, and define $x_i = hi$ for $i \in \mathbb{Z}$. We define the function

$$f_N(x) = \sum_{i=-m}^{m} u(x_i) g(h^{-1}(x - x_i)),$$

where $m \in \mathbb{N}$ is the least integer greater than or equal to $h^{-1}R$, so $m \leq h^{-1}R + 1$. This ensures that $x_{-m} \leq R$ and $x_m \geq R$. Note that this is a 2-layer neural network with $N = 3(2m+1) \leq 6(h^{-1}R+1)$ hidden nodes. Since $g(x) = 0$ for $|x| \geq 1$ we have

$$g(h^{-1}(x_j - x_i)) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise.} \end{cases}$$

Therefore the function f_N exactly interpolates u at the points x_i; that is

$$f_N(x_i) = u(x_i) \quad \text{for } i = -m, \ldots, m. \tag{8.4.17}$$

Furthermore, the function f_N is piecewise linear, and the only points of non-differentiability are the x_i. So on the interval $[x_i, x_{i+1}]$ the function f_N is a

linear function that interpolates $u(x_i)$ and $u(x_{i+1})$. There is only one such linear interpolation, and it is given by

$$f_N(x) = \left(1 - \frac{x - x_i}{h}\right) u(x_i) + \left(\frac{x - x_i}{h}\right) u(x_{i+1}) \quad \text{for } x_i \le x \le x_{i+1}. \tag{8.4.18}$$

Since u is Lipschitz we have for $x_i \le x \le x_{i+1}$ that

$$|u(x_i) - u(x)| \le \mathrm{Lip}(u)|x - x_i| \le \mathrm{Lip}(u)h$$

and similarly $|u(x_{i+1}) - u(x)| \le \mathrm{Lip}(u)h$. Therefore

$$\begin{aligned}
|f_N(x) - u(x)| &= \left| \left(1 - \frac{x - x_i}{h}\right) u(x_i) + \left(\frac{x - x_i}{h}\right) u(x_{i+1}) - u(x) \right| \\
&= \left| \left(1 - \frac{x - x_i}{h}\right) (u(x_i) - u(x)) + \left(\frac{x - x_i}{h}\right) (u(x_{i+1}) - u(x)) \right| \\
&\le \left(1 - \frac{x - x_i}{h}\right) |u(x_i) - u(x)| + \left(\frac{x - x_i}{h}\right) |u(x_{i+1}) - u(x)| \\
&\le \left(1 - \frac{x - x_i}{h} + \frac{x - x_i}{h}\right) \mathrm{Lip}(u)h \\
&\le \mathrm{Lip}(u)h.
\end{aligned}$$

Choosing $h = \mathrm{Lip}(u)^{-1}\varepsilon$ completes the proof of (8.4.14) when u is Lipschitz continuous.

If u' is Lipschitz continuous, then we rewrite (8.4.18) as

$$f_N(x) = m_i(x - x_i) + u(x_i),$$

where

$$m_i = \frac{u(x_{i+1}) - u(x_i)}{h}.$$

Since u' is Lipschitz, u is everywhere differentiable. By the mean value theorem there exists $x_* \in [x_i, x_{i+1}]$ such that $u'(x_*) = m_i$. We now compute

$$\begin{aligned}
f_N(x) - u(x) &= m_i(x - x_i) + u(x_i) - u(x) \\
&= m_i(x - x_*) + m_i(x_* - x_i) + u(x_i) - u(x) \\
&= u(x_i) - u(x_*) - u'(x_*)(x_i - x_*) \\
&\qquad + u(x_*) + u'(x_*)(x - x_*) - u(x).
\end{aligned} \tag{8.4.19}$$

Now, we recall that by Taylor expansion we have

$$|u(x) - u(x_*) - u'(x_*)(x - x_*)| \le \frac{1}{2}\,\mathrm{Lip}(u')|x - x_*|^2. \tag{8.4.20}$$

Indeed, to see this we simply integrate

$$\begin{aligned} u(x) - u(x_*) &= \int_{x_*}^{x} u'(t)\,dt \\ &= \int_{x_*}^{x} u'(x_*)\,dt + \int_{x_*}^{x} (u'(t) - u'(x_*))\,dt \\ &= u'(x_*)(x - x_*) + \int_{x_*}^{x} (u'(t) - u'(x_*))\,dt. \end{aligned}$$

For the rest of the argument assume $x > x_*$, the other case is is similar. Then we have

$$\begin{aligned} |u(x) - u(x_*) - u'(x_*)(x - x_*)| &\leq \left| \int_{x_*}^{x} (u'(t) - u'(x_*))\,dt \right| \\ &\leq \int_{x_*}^{x} |u'(t) - u'(x_*)|\,dt \\ &\leq \mathrm{Lip}(u') \int_{x_*}^{x} t - x_*\,dt \\ &= \frac{1}{2}\,\mathrm{Lip}(u')|x - x_*|^2, \end{aligned}$$

which establises the Taylor expansion claim.

Applying the Taylor expansion (8.4.20) in (8.4.19) we obtain

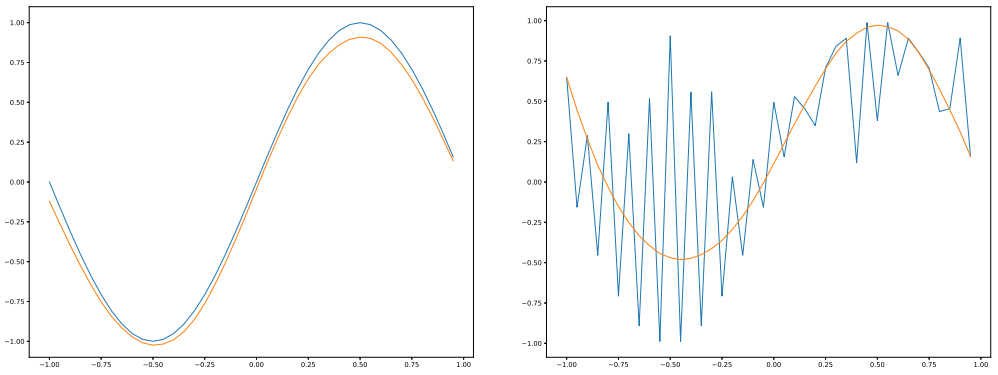
$$|f_N(x) - u(x)| \leq \mathrm{Lip}(u')h^2.$$

Setting $h = \sqrt{\mathrm{Lip}(u)^{-1}\varepsilon}$ completes the proof of (8.4.15) in the case that u' is Lipschitz continuous. □

Remark 8.4.5. The proof of Theorem 8.4.4 shows that the function

$$f_N(x) = \sum_{i=-m}^{m} y_i g(h^{-1}(x - x_i)), \tag{8.4.21}$$

exactly fits the data (x_i, y_i), that is $f_N(x_i) = y_i$. Thus, a 2-layer neural network with $O(N)$ hidden nodes can fit N datapoints exactly, even if that data has no structure and is, say, random noise (at least in dimension $n = 1$). This means that neural networks have the capacity and potential to memorize labels, especially due to the fact that modern deep learning operates in the severely overparameterized regime where there are far more parameters than training datapoints.



(a) Smooth labels (b) Noisy labels

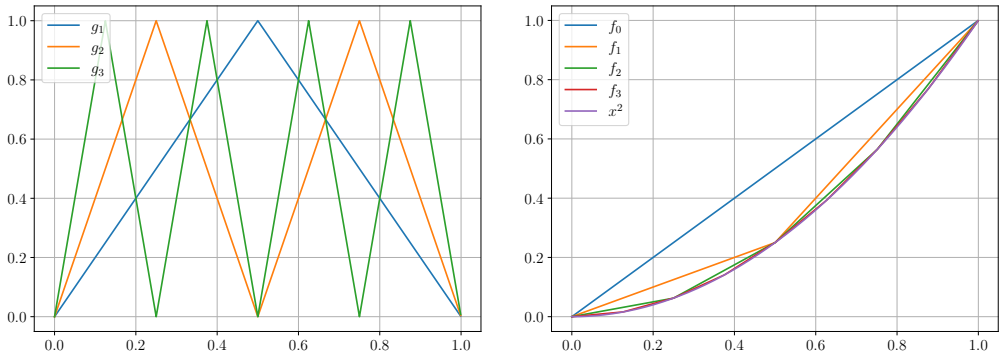
Figure 8.4.9: Example of a network fitting smooth and noisy data. The network has 10000 hidden nodes and only 40 training points. After 10000 iterations of full batch gradient descent, the network does not fit the noise in (b), even though a network with only 40 nodes could fit it perfectly.

Nonetheless, neural networks generally do not overfit, and the reasons for this are still unresolved and somewhat mysterious. In Figure 8.4.9 we compare a neural network trained on a smooth labeling function and a noisy one. We used a 2-layer network with ReLU activations and 10000 hidden nodes, while only using 40 training datapoints and the L^1 loss. The network could have chosen a function that exactly fits the noise in Figure 8.4.9 (b). However, the network chooses instead to ignore the noise, indicating that overfitting and generalization in deep learning are closely connected to the optimzation algorithms used to train deep neural networks.

Remark 8.4.6. In n dimensions, where $n \geq 2$, the bump function construction used here requires a network with $n+1$ layers, but can otherwise be made to work. More sophisticated proof techniques can be used to show that 2-layers is sufficient to approximate any continuous function on $\mathbb{R}^n$.

We now turn to the problem of approximating polynomials. It turns out that deep ReLU networks are very efficient at polynomial approximation. We start with approximating $f(x) = x^2$. We define

$$g(x) = \begin{cases} 2x, & \text{if } 0 \leq x \leq \frac{1}{2} \\ 2 - 2x, & \text{, if } \frac{1}{2} \leq x \leq 1, \end{cases}$$



(a) g_i (b) f_i

Figure 8.4.10: A depicition of the functions f_i and g_i used in approximating the parabola x^2.

and the m-fold composition

$$g_m = \underbrace{g \circ g \circ \cdots \circ g}_{m \text{ times}}.$$

The function $g = g_1$ is a bump function, like we used in Theorem 8.4.4, and can be constructed with a 2-layer ReLU network with 3 hidden nodes. Since g_m is simply the m-fold composition of g, we can implement g_m with an $m+1$ layers (m-hidden layer) ReLU network with $O(m)$ nodes.

The function g_m is a sawtooth function with 2^{m-1} teeth. Figure 8.4.10 shows g_1, g_2 and g_3. To see this, we note that $g_{m+1}(x) = g(g_m(x))$ and so

$$g'_m(x) = g'(g_{m-1}(x))g'_{m-1}(x).$$

Since $g'_1(x) = \pm 2$ we see that $g'_m(x) = \pm 2^m$, and furthermore

$$g'_m(x) = \begin{cases} 2g'_{m-1}(x), & \text{if } 0 \le g_{m-1}(x) \le \frac{1}{2} \\ -2g'_{m-1}(x), & \text{if } \frac{1}{2} \le g_{m-1}(x) \le 1. \end{cases}$$

Thus, going from g_{m-1} to g_m, we are flipping the upper half of all the teeth downwards, and scaling the teeth by a factor of 2, which doubles the number of teeth. From this, we see that

$$g'_m(x) = \begin{cases} 2^m, & \text{if } \frac{2j}{2^m} \le x \le \frac{2j+1}{2^m} \\ -2^m, & \text{if } \frac{2j+1}{2^m} \le x \le \frac{2j+2}{2^m}, \end{cases} \tag{8.4.22}$$

where j is an integer $j = 0, 1, \ldots, 2^{m-1} - 1$.

Let $f_m(x)$ be the piecewise linear approximation of the parabola $f(x) = x^2$ with 2^m pieces, so that $f_m\left(\frac{k}{2^m}\right) = \left(\frac{k}{2^m}\right)^2$ for $k = 0, \ldots, m$. The functions f_0, f_1, f_2 and f_3 are depicted in Figure 8.4.10. By arguments similar to those used in Theorem 8.4.4 we have that

$$|f_m(x) - x^2| \leq 2^{-2m}. \tag{8.4.23}$$

The following lemma gives a surprising relationship between f_m and g_m, and shows that f_m can be computed very efficiently by deep ReLU networks.

Lemma 8.4.7. *For any $0 \leq x \leq 1$ and $m \geq 1$ we have*

$$f_m(x) = x - \sum_{k=1}^{m} \frac{g_k(x)}{2^{2k}}. \tag{8.4.24}$$

Proof. We first claim that

$$f_m - f_{m-1} = -\frac{g_m}{2^{2m}}. \tag{8.4.25}$$

To see this, we consider how the slope f'_m differs from f'_{m-1}. Consider the j^{th} piecwise linear interval for f_{m-1}, which is $[2^{-(m-1)}j, 2^{-(m-1)}(j+1)]$. On this interval we have

$$f'_{m-1}(x) = \frac{\left(2^{-(m-1)}(j+1)\right)^2 - \left(2^{-(m-1)}j\right)^2}{2^{-(m-1)}} = 2^{-(m-1)}(2j+1).$$

The function f_m has twice as many pieces, each with half the width, and splts this inverval in half. On the first half of the interval $[2^{-(m-1)}j, 2^{-(m-1)}(j+\frac{1}{2})]$ f_m has slope

$$f'_m(x) = \frac{\left(2^{-(m-1)}(j+\frac{1}{2})\right)^2 - \left(2^{-(m-1)}j\right)^2}{2^{-m}} = 2^{-(m-1)}(2j+\tfrac{1}{2}).$$

Therefore, on the first half of the interval we have

$$f'_m(x) - f'_{m-1}(x) = 2^{-(m-1)}(\tfrac{1}{2} - 1) = -2^{-m}.$$

A very similar argument shows that on the second half of the interval $[2^{-(m-1)}(j+\frac{1}{2}), 2^{-(m-1)}(j+1)]$ we have

$$f'_m(x) - f'_{m-1}(x) = 2^{-m}.$$

The somewhat surprising conclusion is that the difference in slopes is independent of the interval j we are considering. It follows that

$$f'_m(x) - f'_{m-1}(x) = -\frac{g'_m(x)}{2^{2m}}.$$

Integrating both sides from $x = 0$ to $x = t$ and using that $f_m(0) = 0 = g_m(0)$ we have

$$f_m(t) - f_{m-1}(t) = -\frac{g_m(t)}{2^{2m}}$$

for any $0 \leq t \leq 1$. This establishes the claim (8.4.25).

Summming (8.4.25) over $m = 1$ to $m = k$, the left hand side telescopes to give

$$f_k(x) - f_0(x) = -\sum_{m=1}^{k} \frac{g_m(x)}{2^{2m}}.$$

The proof is completed by noting that $f_0(x) = x$. □

By Lemma 8.4.7, we see that the function f_m can be expressed as a linear combination of $g_1, \ldots, g_m$. Since g_1 can be implemnted as a 2-layer ReLU network, the m-fold composition g_m can be computed using a network with $O(m)$ layers and $O(m)$ hidden nodes. In fact, this network will compute all the intermediate g_i as well, and by using skip connections to a final layer, we can implement f_m with a ReLU network with $O(m)$ layers and nodes. Recalling the approximation error in (8.4.23), we see that a ReLU network with $O(m)$ nodes can approximate x^2 to accuacy 2^{-2m}. If we choose an accuracy $\varepsilon = 2^{-2m}$, then we find that $m = \frac{1}{2}\log_2(\varepsilon^{-1})$. This discussion is summarized in the following result.

Theorem 8.4.8. *For any $\varepsilon > 0$ there exists a ReLU network f with $O(\log(\varepsilon^{-1}))$ layers and nodes such that*

$$|f(x) - x^2| \leq \varepsilon \quad \textit{for } 0 \leq x \leq 1.$$

Theorem 8.4.8 is much sharper than Theorem 8.4.4, which required $O(\varepsilon^{-1/2})$ nodes for the same approximation.

Once we can approximation x^2, we can immediately extend these results to polynomials. Note that

$$xy = \frac{1}{4}(x+y)^2 - \frac{1}{4}(x-y)^2.$$

Therefore, we can compute the multiplication xy via linear functions composed with x^2. This allows us to show that there exists a ReLU network with

$O(2\log(\varepsilon^{-1}))$ that approximates the multiplicaiton xy to accuracy ε. This allow us to implement x^3 to accuracy ε with a ReLU network with $O(3\log(\varepsilon^{-1}))$ nodes, and in general x^k can be implemented with $O(k\log(\varepsilon^{-1}))$. In fact, we have the following theorem.

Theorem 8.4.9. *Let g be a polynomial of degree k. For any $\varepsilon > 0$ there exists a ReLU network f with $O(k\log(\varepsilon^{-1}))$ hidden nodes such that*

$$|f(x) - g(x)| \leq \varepsilon \quad \textit{for } 0 \leq x \leq 1.$$

These improved rates can be extended to, for example, real analytic functions, which are well approximated globally by polynomials, and to smooth functions, which are well-approximated locally by polynomials.

Exercise 8.4.10. Extend the argument in Theorem 8.4.4 to a general activation function σ. What conditions should σ satisfy to make the same argument work? The bump function construction will be different for bounded activations, like the sigmoid activation depicted in Figure 8.4.1. Figure 8.4.8 (b) shows the sigmoid bump function $g(x) = \sigma(x+1) - \sigma(x-1)$. △

8.4.5 Convolutional Neural Networks

Python Notebook: .ipynb

This section offers a brief introduction to Convolutional Neural Networks (CNN). We refer the reader to [11] for more details. CNNs are the most powerful machine learning tools for image processing and computer vision. They can be interpreted as special cases of the fully connected neural networks discussed in Section 8.4.1 that are adapted to image process by introducing *locality* and *translation invariance* into the network. The main operation is the convolution of a $(2N+1) \times (2N+1)$ matrix W and an image I, given by

$$(W * I)(i,j) = \sum_{p,q=-N}^{N} W(N+1+p, N+1+q) I(i+p, j+q).$$

The convolution replaces the linear mapping in the fully connected neuron (8.4.3). The output of the convolution $W * I$ is an image as well, normally taken to be slightly smaller so that $(i+p, j+q)$ is within the image domain for all $-N \leq p, q \leq N$. For example, if $N = 1$ then we are working with 3×3 filters, and a 28×28 pixel MNIST image would be reduced to 26×26 pixels after the convolution. The convolution is the same type of operation we saw in

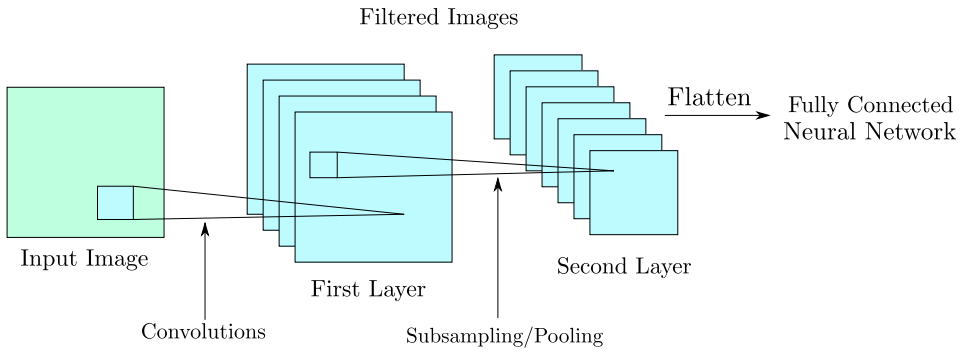


Figure 8.4.11: An example of a typical Convolutional Neural Network (CNN) architecture.

Chapter 6 with the Discrete Fourier Transform and Chapter 7 in the context of the Haar Wavelet. It is the most basic image processing operation, and different types of filters can look for edges or other types of geometric shapes locally in the image.

The filters are restricted to be local, in that they are normally 3×3 or slightly larger (say, 7×7 for higher resolution images). Each filter looks for a particular local feature in the image. The convolution operation applies the filter at every location (i, j) in the image I, which is to say the filter is *translation invariant*. The convolution will detect features wherever they lie in the image.

It is important to note that a single filter W has very few parameters (i.e., a 3×3 filter has 9 parameters), and is applied to every pixel in the image. If there are n pixels where it is possible to apply the filter, without overlapping the boundary of the image, then the output of this single filter is the same size as n fully connected neurons (8.4.3). Thus, by forcing locality and translation invariance, CNNs use drastically fewer parameters compared to fully connected neural networks. This makes it more difficult to overfit with CNNs and allows for faster training, which are some of their main advantages.

Just like with fully connected networks, CNNs use activation functions, normally ReLU, after each layer, and can stack many layers deep. Now, while the convolution is translation invariant in the sense that the same filters are applied to every point in the image, the filtered image $W * I$ is indeed different when the image I is translated (the filtered image is translated as well). To realize translation invariance in CNNs, a subsampling technique called *pooling* is used. Pooling subsamples the filtered images between two layers, usually using max-pooling or average-pooling.

For example, max-pooling by 2 in each direction corresponds to splitting

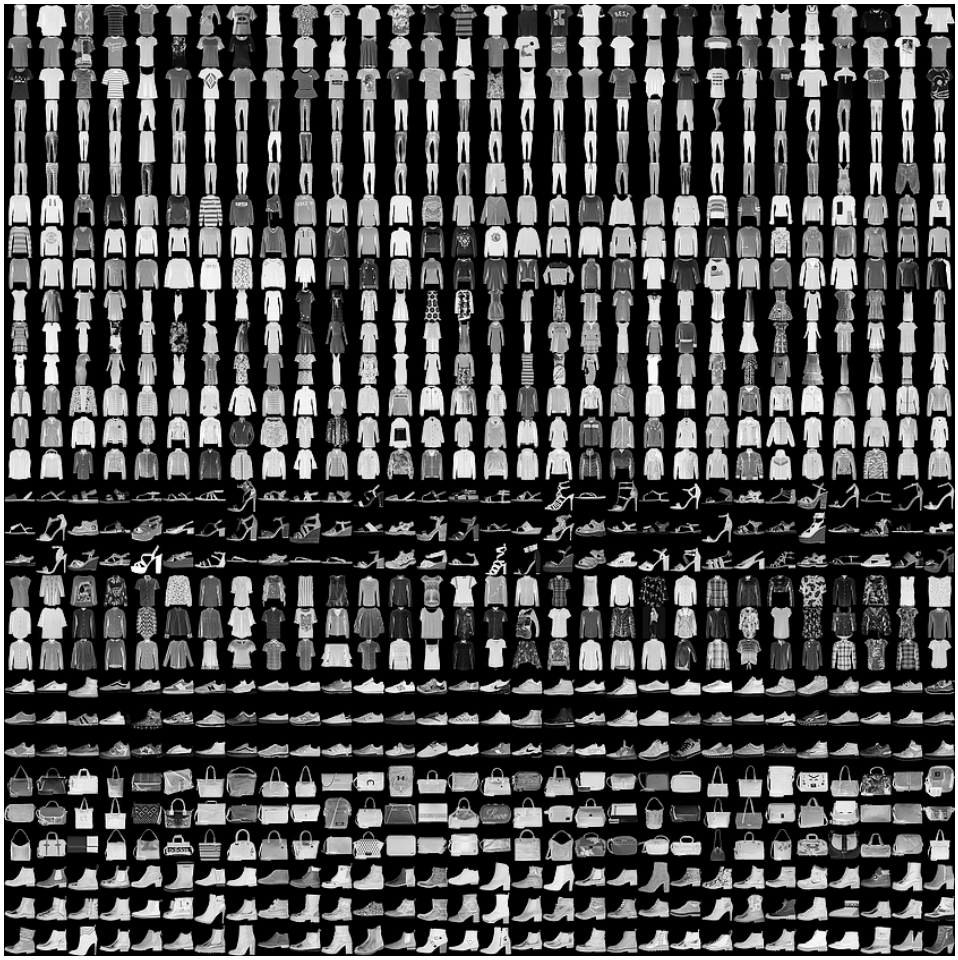


Figure 8.4.12: Example of some images from the FashionMNIST dataset. Each image is a 28×28 pixel image of an item of clothing from a fashion catalog.

the image into 2×2 pixel blocks, and replacing each block with a single pixel taking the maximum pixel value over the block (or average for average-pooling). It is immediately clear that max pooling makes the subsampled image invariant to small shifts. Pooling also allows future layers to detect more global features within the image, yielding a multi-scale image analysis, similar to Wavelets (see Chapter 7).

Figure 8.4.11 shows an example of a 2-layer CNN architecture. As in the figure, for image classification, the output of the convolutional part of the network is normally fed into a fully connected network, which uses the features extracted by the convolutional part to make classification predictions.

We ran an experiment testing a simple convolutional neural network for

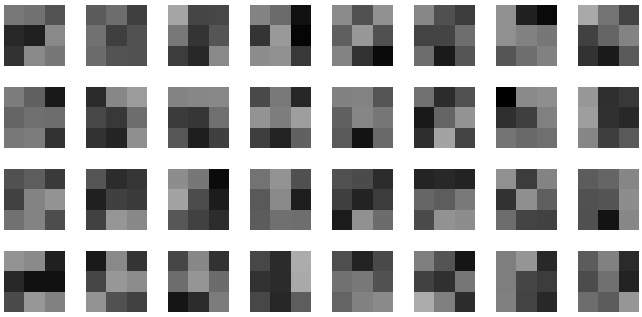


Figure 8.4.13: The 32 3×3 filters from the first layer of the convolutional neural network for classifying MNIST digits.

classification of the MNIST and FashionMNIST image datasets. FashionMNIST is a drop-in replacement for MNIST, except here the classes are different items of clothing with pictures taken from a fashion catalog. Figure 8.4.12 shows an example of some of the FashionMNIST images. The FashionMNIST dataset is more challenging to classify, compared to MNIST, but is still a toy problem for modern machine learning. We used a 4 layer neural network where the first two layers are convolutional layers, and the last two are fully connected layers, and all activations are ReLU. The first layer has 32 channels (which means 32 convolutional filters W), while the second layer has 64 channels. The filter sizes are all 3×3, meaning the first layer reduces the 28×28 pixel MNIST images down to 26×26, and the second layer reduces them further to 24×24. After the second layer there is a max-pooling by 2 operation, reducing the images to 12×12. These 64 images of size 12×12 are then flattened into a long array of length $64 \times 12 \times 12 = 9216$, and this is interpreted as the features of the image, which are then fed into the fully connected part of the network. The hidden layer has 128 nodes, and the output has 10 components, since there are 10 classes.

We used stochastic gradient descent (SGD) with batch size 64 and ran the training for 14 epochs (passes over the whole training set) with a learning rate of $\alpha = 1$ that is reduced by a multiplicative factor of 0.7 each epoch. We used the full 60000 training set, and tested on the held out 10000 images from the testing set. For MNIST we obtained 99.04% testing accuracy and on FashionMNIST we obtained 92.55% accuracy.

It is interesting to view the learned weights and filter responses. Figure 8.4.13 shows the 32 filters (size 3×3) from the first layer. Figure 8.4.14 shows the 32 corresponding filtered images from the first layer, for a single MNIST 2 digit. Figure 8.4.15 shows the outputs of the second layer, which can be

Figure 8.4.14: The 32 channels of output from the first convolutional layer acting on an image of a 2.

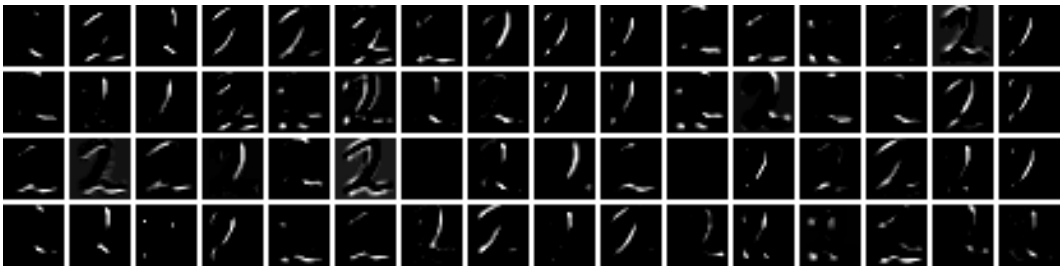


Figure 8.4.15: The 64 channels of output from the second convolutional layer acting on an image of a 2. Notice the channels appear to be detecting edges in various directions.

interpreted as looking for edges in various directions in the image.

We also experimented with the same pixel shifting test, to examine the translation invariance of CNNs. On MNIST a single pixel shift reduced accuracy to 98.07%, a 2 pixel shift to 92.06%, and a 3 pixel shift to 75%. For FashionMNIST a single pixel shift reduced accuracy to 89.73%, a 2 pixel shift to 75.94%, and a 3-pixel shift gives 47.38%. We can see there is more robustness to shifting images, but only by a small amount. Other special techniques, like data augmentation, are required to train a network that is invariant to larger shifts.

Chapter 9

Optimization

Optimization—finding minimizers of functions—is one of the most important problems in science and engineering. The most common optimization methods, especially in machine learning, are first order methods that use only the gradient ∇f of the function f to be minimized. For problems that are smaller in size, so that the Hessian $\nabla^2 f$ can be computed and inverted, Newton's method is a second order optimization algorithm that can achieve much faster quadratic convergence rates. This chapter is mostly focused on theoretical convergence results for gradient descent, including stochastic gradient descent, with an additional section that briefly covers Newton's method.

9.1 Gradient descent

Python Notebook: .ipynb

Gradient descent aims to minimize an objective function $f : \mathbb{R}^n \to \mathbb{R}$ via the iterative procedure

$$x_{k+1} = x_k - \alpha \nabla f(x_k). \tag{9.1.1}$$

The parameter $\alpha > 0$ is the time step (often called the *learning rate* when using gradient descent to train machine learning algorithms).

Exercise 9.1.1. Fix x and define

$$T(y) = f(x) + \nabla f(x)^T (y - x) + \frac{1}{2\alpha} \|y - x\|^2. \tag{9.1.2}$$

Show that T is minimized by

$$y = x - \alpha \nabla f(x).$$

Thus, the gradient descent iteration (9.1.1) can be viewed as repeatedly minimizing the approximation (9.1.2), centered at $x = x_k$, until convergence. $\triangle$

In this section, we give a basic analysis of the convergence of gradient descent. Throughout this section we assume the objective function $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth function that admits a global minimizer $x_* \in \mathbb{R}^n$. That is $f(x_*) \leq f(x)$ for all $x \in \mathbb{R}^n$. For the moment, the minimizer may not be unique, and the function f need not be convex. We denote the optimal value of f by $f_* := f(x_*)$.

We will use the notions of Lipschitzness of ∇f and $\nabla^2 f$, introduced in Section 2.5, and notions of convexity (and strong convexity) introduced in Section 2.6. The reader not familiar with these notions should review the relevant sections of Chapter 2.

9.1.1 The sublinear rate

Our first result is a general convergence result for gradient descent.

Theorem 9.1.2. *Assume ∇f is L-Lipschitz and that $\alpha \leq \frac{1}{L}$. Then for any integer $t \geq 1$ we have*

$$\min_{0 \leq k \leq t} \|\nabla f(x_k)\|^2 \leq \frac{2(f(x_0) - f_*)}{\alpha t}. \tag{9.1.3}$$

Proof. The proof is split into two parts.

1. We first show that

$$f(x_{k+1}) \leq f(x_k) - \frac{\alpha}{2}\|\nabla f(x_k)\|^2 \tag{9.1.4}$$

provided $\alpha \leq \frac{1}{L}$. To see this, we use a first order Taylor expansion of f (see Theorem 2.5.3) to obtain

$$f(y) \leq f(x) + \nabla f(x)^T(y - x) + \frac{L}{2}\|x - y\|^2 \tag{9.1.5}$$

for all $x, y \in \mathbb{R}^n$. We use this to deduce

$$\begin{aligned} f(x_{k+1}) &\leq f(x_k) + \nabla f(x_k)^T(x_{k+1} - x_k) + \frac{L}{2}\|x_{k+1} - x_k\|^2 \\ &= f(x_k) - \alpha \nabla f(x_k)^T \nabla f(x_k) + \frac{\alpha^2 L}{2}\|\nabla f(x_k)\|^2 \\ &= f(x_k) - \left(\alpha - \frac{\alpha^2 L}{2}\right)\|\nabla f(x_k)\|^2. \end{aligned}$$

Notice we used the gradient descent iteration (9.1.1) in the second line above. This shows that gradient descent decreases the energy f provided $\frac{\alpha L}{2} \leq 1$, or $\alpha \leq \frac{2}{L}$. To find the optimal α—the value that gives the largest decrease in f—we maximize the quantity $\alpha - \frac{\alpha^2 L}{2}$ over α, yielding $\alpha = \frac{1}{L}$. Therefore, there is no loss in optimality by making the restriction $\alpha \leq \frac{1}{L}$. Using this restriction yields $\alpha - \frac{\alpha^2 L}{2} \geq \frac{\alpha}{2}$, which establishes (9.1.4).

2. We now rearrange (9.1.4) and sum over k to obtain

$$\frac{\alpha}{2} \sum_{k=0}^{t} \|\nabla f(x_k)\|^2 \leq \sum_{k=0}^{t} (f(x_k) - f(x_{k+1})) = f(x_0) - f(x_{t+1}),$$

since the sum in the middle term above is telescoping. We use the lower bound

$$\sum_{k=0}^{t} \|\nabla f(x_k)\|^2 \geq (t+1) \min_{0 \leq k \leq t} \|\nabla f(x_k)\|^2$$

to obtain

$$\min_{0 \leq k \leq t} \|\nabla f(x_k)\|^2 \leq \frac{2(f(x_0) - f(x_{t+1}))}{\alpha(t+1)}.$$

The proof is completed by noting that $f_* \leq f(x_{t+1})$ and replacing $t+1$ with t. □

Theorem 9.1.2 shows that after t steps of gradient descent, we are guaranteed to find a point x_k, for some $0 \leq k \leq t$, for which $\|\nabla f(x_k)\|^2 = O\left(\frac{1}{t}\right)$. It is important to point out that k may not be equal to t in general, i.e., x_k may not be the most recent gradient descent iterate. The convergence rate $O\left(\frac{1}{t}\right)$ is very slow and is referred to as *sublinear* convergence. Nevertheless, using very few assumptions on f, Theorem 9.1.2 shows that gradient descent converges to a critical point of f in the sense that

$$\lim_{t \to \infty} \min_{0 \leq k \leq t} \|\nabla f(x_k)\|^2 = 0.$$

Since we made no assumptions about f, aside from Lipschitz continuity, there may be critical points that are not global minima of f (e.g., local minima, saddle points, or maxima). In particular, Theorem 9.1.2 does not show that gradient descent converges to a minimizer of f.

To show that gradient descent converges to a global minimizer of f, we need to place an additional assumption on f to ensure that all critical points are minimizers. The simplest such assumption is *convexity*. Section 2.6 reviews some basic theory of convex functions, giving several different equivalent definitions of convexity. For our purposes in this section, the most convenient

definition of convexity is that f lies above its tangent planes (see Theorem 2.6.5 (iii)). That is, a convex function f satisfies

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) \tag{9.1.6}$$

for all $x, y \in \mathbb{R}^n$.

Our next result can be viewed as the extension of Theorem 9.1.2 to convex functions.

Theorem 9.1.3. *Assume f is convex, ∇f is L-Lipschitz, and take $\alpha \leq \frac{1}{L}$. Then for any integer $t \geq 1$ we have*

$$f(x_t) - f_* \leq \frac{\|x_0 - x_*\|^2}{2\alpha t}, \tag{9.1.7}$$

where x_* is any minimizer of f.

Proof. We start with the energy decreasing inequality (9.1.4) from Theorem 9.1.2, which requires the restriction $\alpha \leq \frac{1}{L}$. Let $x_* \in \mathbb{R}^n$ be a minimizer of f, so $f(x_*) = f_*$. Since f is convex, we can use (9.1.6) with $y = x_*$ and $x = x_k$ to obtain

$$f(x_*) \geq f(x_k) + \nabla f(x_k)^T (x_* - x_k).$$

Rearranging we have

$$f(x_k) \leq f(x_*) + \nabla f(x_k)^T (x_k - x_*).$$

Inserting this into (9.1.4) we have

$$\begin{aligned} f(x_{k+1}) &\leq f(x_k) - \frac{\alpha}{2} \|\nabla f(x_k)\|^2 \\ &\leq f(x_*) + \nabla f(x_k)^T (x_k - x_*) - \frac{\alpha}{2} \|\nabla f(x_k)\|^2 \\ &= f_* + \frac{1}{2\alpha} \left(2\alpha \nabla f(x_k)^T (x_k - x_*) - \alpha^2 \|\nabla f(x_k)\|^2 \right) \\ &= f_* + \frac{1}{2\alpha} \left(\|x_k - x_*\|^2 - \|x_k - x_* - \alpha \nabla f(x_k)\|^2 \right) \\ &= f_* + \frac{1}{2\alpha} \left(\|x_k - x_*\|^2 - \|x_{k+1} - x_*\|^2 \right). \end{aligned} \tag{9.1.8}$$

We now subtract f_* from both sides and sum over k to obtain

$$\begin{aligned} \sum_{k=0}^{t-1} (f(x_{k+1}) - f_*) &\leq \frac{1}{2\alpha} \sum_{k=0}^{t-1} \left(\|x_k - x_*\|^2 - \|x_{k+1} - x_*\|^2 \right) \\ &= \frac{1}{2\alpha} \left(\|x_0 - x_*\|^2 - \|x_t - x_*\|^2 \right) \\ &\leq \frac{\|x_0 - x_*\|^2}{2\alpha}, \end{aligned}$$

where we used that the sum on the right hand side of the first line is telescoping. By (9.1.4), the values of f are decreasing with gradient descent, so we have that

$$\sum_{k=0}^{t-1} (f(x_{k+1}) - f_*) \geq t(f(x_t) - f_*).$$

Inserting this above completes the proof. □

Theorem 9.1.3 shows that gradient descent on a convex function f converges to the minimum value f_* at a rate of $O\left(\frac{1}{t}\right)$ after t iterations. This sublinear convergence rate is very slow. In order to minimize f to within $\varepsilon > 0$ of the optimal value f_* requires $t = O(\varepsilon^{-1})$ steps. The reason for this slow convergence is that a general convex function f may be arbitrarily flat near a minimum x_*, meaning that the gradient ∇f is very small and gradient descent proceeds slowly towards x_*.

9.1.2 Linear convergence with the PL inequality

To obtain a better convergence rate, we need to make an additional assumption about how flat f can be at minima. A standard assumption to make is strong convexity. We recall from Section 2.6 that f is *μ-strongly convex* if

$$f(y) \geq f(x) + \nabla f(x)^T(y - x) + \frac{\mu}{2}\|x - y\|^2 \tag{9.1.9}$$

for all $x, y \in \mathbb{R}^n$. Other equivalent definitions of strong convexity are given in Theorem 2.6.5, but this one is most useful in the proofs in this section. It is important to point out that if f is μ-strongly convex and ∇f is L-Lipschitz, then we must have $\mu \leq L$. Indeed, if we compare (9.1.5) and (9.1.9) we obtain

$$\frac{L}{2}\|x - y\|^2 \geq \frac{\mu}{2}\|x - y\|^2$$

for all $x, y \in \mathbb{R}^n$, and so $L \geq \mu$.

Notice that if we take $x = x_*$ in (9.1.9) then $\nabla f(x_*) = 0$ and we get

$$f(y) \geq f_* + \frac{\mu}{2}\|y - x_*\|^2. \tag{9.1.10}$$

This shows that x_* is the unique minimizer of f, since $f(y) > f_*$ for all $y \neq x_*$. This also shows that f grows at least quadratically, like $\frac{\mu}{2}\|x - x_*\|^2$, near the minimizer x_*.

We now give a very useful consequence of strong convexity. If f is μ-strongly convex, then we can minimize both sides of (9.1.9) over $y \in \mathbb{R}^n$ to find that

$$f_* = \min_{y\in\mathbb{R}^n} f(y) \geq f(x) + \min_{y\in\mathbb{R}^n} \left\{ \nabla f(x)^T(y-x) + \frac{\mu}{2}\|x-y\|^2 \right\}.$$

The minimum on the right hand side is attained (by differentiating in y) at y satisfying $\nabla f(x) + \mu(y-x) = 0$, or $y - x = -\frac{1}{\mu}\nabla f(x)$. Substituting this above we find that

$$f_* \geq f(x) - \frac{1}{\mu}\|\nabla f(x)\|^2 + \frac{1}{2\mu}\|\nabla f(x)\|^2$$

Simplifying we obtain

$$\frac{1}{2}\|\nabla f(x)\|^2 \geq \mu(f(x) - f_*) \tag{9.1.11}$$

for all $x \in \mathbb{R}^n$. Eq. (9.1.11) is known as the Polyak-Lojasiewicz (PL) inequality. Every μ-strongly convex function satisfies the PL inequality, but the converse is not true. In fact, there are non-convex functions that satisfy the PL inequality (9.1.11).

Exercise 9.1.4. Show that the function $f(x) = x^2 + 3\sin^2(x)$ satisfies the PL inequality (9.1.11) with $\mu = \frac{1}{32}$, but f is not convex. [Hint: Use the equivalent definition of convexity that $f''(x) \geq 0$.] △

Finally, let us also note that if we combine the PL inequality (9.1.11) with (9.1.10) (which requires strong convexity) we obtain

$$\frac{1}{2}\|\nabla f(x)\|^2 \geq \mu(f(x) - f_*) \geq \frac{\mu^2}{2}\|x - x_*\|^2.$$

In other words, we have the bound

$$\|x - x_*\| \leq \frac{1}{\mu}\|\nabla f(x)\|. \tag{9.1.12}$$

Thus, for a strongly convex function, the distance to the minimizer is controlled by the norm of the gradient vector.

The next result shows that gradient descent converges at a *linear rate* provided f satisfies the PL inequality. This includes strongly convex functions as a special case, but also covers many nonconvex functions.

Theorem 9.1.5. *Assume f satisfies the PL inequality (9.1.11), ∇f is L-Lipschitz, and take $\alpha \leq \frac{1}{L}$. Then for any integer $t \geq 0$ we have*

$$f(x_t) - f_* \leq (1 - \alpha\mu)^t (f(x_0) - f_*). \tag{9.1.13}$$

Proof. We start with (9.1.4) from Theorem 9.1.2 and apply the PL inequality (9.1.11) to obtain

$$f(x_{k+1}) \leq f(x_k) - \frac{\alpha}{2}\|\nabla f(x_k)\|^2 \leq f(x_k) - \alpha\mu(f(x_k) - f_*),$$

provided $\alpha \leq \frac{1}{L}$. Subtracting f_* from both sides and rearranging we find that

$$f(x_{k+1}) - f_* \leq (1 - \alpha\mu)(f(x_k) - f_*).$$

Note that since $\alpha \leq \frac{1}{L}$ and $\mu \leq L$, we have $\alpha \leq \frac{1}{\mu}$, and so $1 - \alpha\mu \geq 0$. Applying this recursively completes the proof. □

The convergence rate in Theorem 9.1.5 is called *linear* because the error $f(x_t) - f_*$ decreases by a constant factor $1 - \alpha\mu$ at each iteration. Plotting the error $f(x_k) - f_*$ on a log-log scale would show a linear relationship with slope $\log(1 - \alpha\mu)$. This linear convergence rate is much faster than the sublinear $O\left(\frac{1}{t}\right)$ rate we obtained for convex functions in Section 9.1.1. Indeed, for the sublinear rate, we need $t = O(\varepsilon^{-1})$ iterations to find x_k with $f(x_k) - f_* \leq \varepsilon$, while for the linear rate we require only $t = O(\log(\varepsilon^{-1}))$ iterations.

Remark 9.1.6. It is also natural to ask how quickly x_k is converging to x_*. For this, we require strong convexity. If f is μ-strongly convex, we can use (9.1.10) with $y = x_t$ and Theorem 9.1.5 to find that

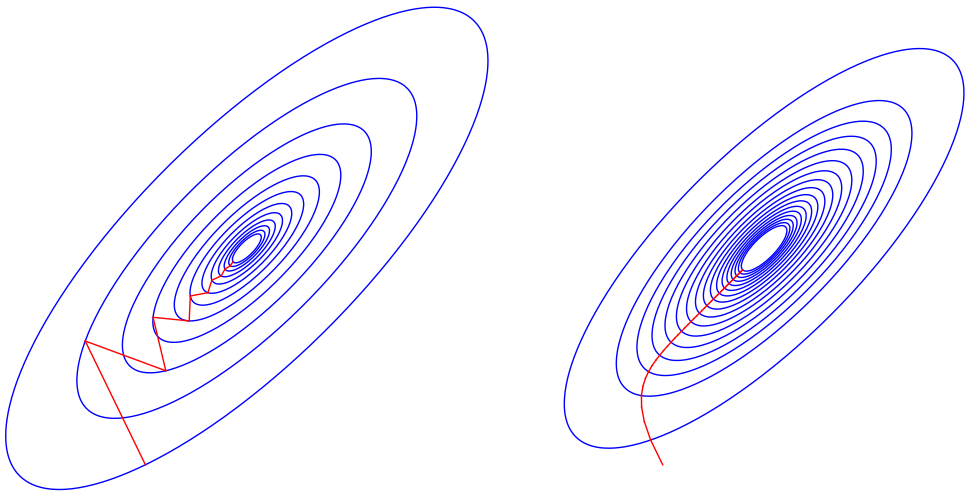
$$\frac{\mu}{2}\|x_t - x_*\|^2 \leq f(x_t) - f_* \leq (1 - \alpha\mu)^t (f(x_0) - f_*).$$

Thus, in the squared Euclidean norm, x_t converges to x_* at the same linear convergence rate as in Theorem 9.1.5, provided f is μ-strongly convex.

9.1.3 Momentum descent

Python Notebook: .ipynb

Even in the strongly convex setting (or, under the weaker PL inequality (9.1.11)), where gradient descent converges linearly, the convergence still slows down considerably near the minimizer. This is due to the fact that the gradient ∇f vanishes at the minimizer, and is small nearby, which slows down the progress of gradient descent. To compensate, one can try to take larger time



(a) $\alpha = 0.05$, 12 steps (b) $\alpha = 0.01$, 50 steps

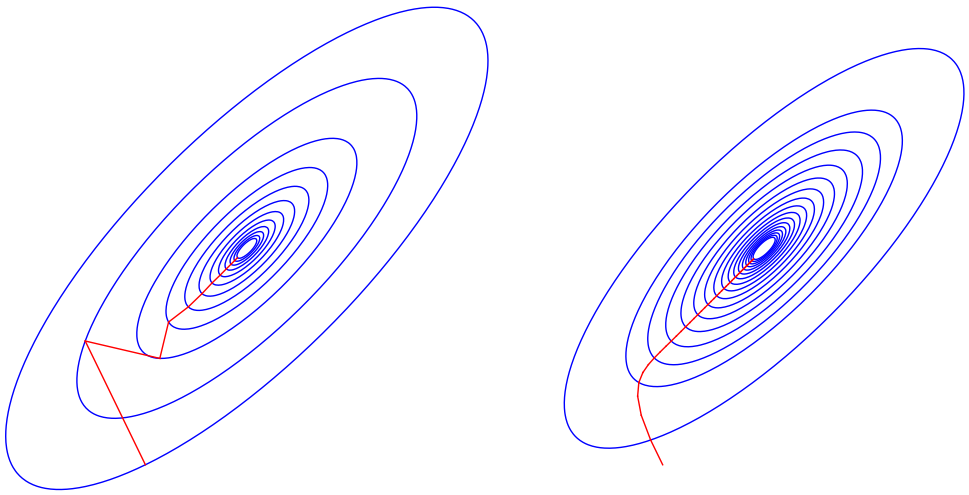
Figure 9.1.1: Gradient descent on a parabolic function with different choices of time steps. For larger time steps the iterations bounce back and forth, limiting progress towards the minimizer, while for smaller time steps the descent path is more direct.

steps in gradient descent, but this often leads to a bouncing effect, where the iterations bounce back and forth across the energy landscape, which also limits progress towards the minimizer. Figure 9.1.1 (a) shows an example of the bouncing effect for large time steps. The function f is given by

$$f(x, y) = (x + y)^2 + 8(x - y)^2,$$

which is strongly convex with a global minimum at $x = y = 0$. The bouncing effect is caused by placing too much trust in the gradient direction, which in general does *not* point towards the minimizer, and cannot be trusted outside of a small local neighborhood. For smaller time steps, as in Figure 9.1.1 (b), we do not see this effect, but we require more steps to reach a similar energy level.

The convergence of gradient descent can be accelerated by utilizing *momentum* in the descent. Momentum methods are loosely based on the idea of rolling a ball with some positive mass down the energy landscape, in the presence of friction forces to slow down the ball. Momentum can build up speed over time, provided the descent directions are similar over many steps, leading to faster convergence near the minimizer. When the descent directions change rapidly over each step, like in the bouncing effect in Figure 9.1.1 (a),



(a) $\alpha = 0.05, \beta = 0.1$, 12 steps (b) $\alpha = 0.01, \beta = 0.25$, 50 steps

Figure 9.1.2: Heavy ball method for different choices of time step and momentum parameter. Momentum acts to average out the descent direction in time, limiting the bouncing effect for larger time steps. Momentum builds up speed and makes more progress towards the minimizer in the same number of steps as gradient descent (see Figure 9.1.1).

momentum acts to average out the descent directions over time and reduces the amount of bouncing and backtracking, leading again to faster convergence.

One of the oldest momentum based methods is the heavy ball method of Polyak [17]. The heavy ball method iterates

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \beta(x_k - x_{k-1}), \tag{9.1.14}$$

where α is the time step and $\beta \in [0, 1]$ is the momentum parameter, where $x_1 = x_0$. Figure 9.1.2 (a) shows how momentum corrects the bouncing effect from Figure 9.1.1 (a) at large time steps. Figure 9.1.2 (b) shows how momentum makes more progress towards the minimizer in the same number of steps, even in the absence of the bouncing effect.

To explain the analogy to a rolling ball, it is useful to consider the continuum version of gradient descent and the heavy ball method. For gradient descent (9.1.1), we can rewrite the equation as

$$\frac{x_{k+1} - x_k}{\alpha} = -\nabla f(x_k).$$

By assuming $x_k = x(\alpha t)$ for a smooth curve $x(t)$, we find that the left hand side is merely a forward differences approximation for $x'(t)$, and so gradient

descent is equivalent in the continuum to the ordinary differential equation (ODE)

$$x'(t) = -\nabla f(x(t)). \tag{9.1.15}$$

Remark 9.1.7. The continuum perspective often makes the convergence analysis easier, since we can use calculus and the chain rule. Suppose $x(t)$ solves (9.1.15) with $x(0) = x_0 \in \mathbb{R}^n$ and assume f is μ-strongly convex. Let $x_* \in \mathbb{R}^n$ denote the unique minimizer of f. We first note that we can rearrange (9.1.15) to read

$$\frac{d}{dt}(x(t) - x_*) = -(\nabla f(x(t)) - \nabla f(x_*)),$$

since $\nabla f(x_*) = 0$ and $\frac{d}{dt}x_* = 0$. We now take the dot product of both sides of the equation above with $x(t) - x_*$ to obtain

$$(x(t) - x_*)^T \frac{d}{dt}(x(t) - x_*) = -(\nabla f(x(t)) - \nabla f(x_*))^T (x(t) - x_*). \tag{9.1.16}$$

By the chain rule, the left hand side is precisely

$$(x(t) - x_*)^T \frac{d}{dt}(x(t) - x_*) = \frac{d}{dt}\frac{1}{2}\|x(t) - x_*\|^2.$$

By the strong convexity of f (see Theorem 2.6.5 (iv)) we have

$$(\nabla f(x(t)) - \nabla f(x_*))^T (x(t) - x_*) \geq \mu \|x(t) - x_*\|^2.$$

Inserting these observations into (9.1.16) we have

$$\frac{d}{dt}\|x(t) - x_*\|^2 \leq -2\mu \|x(t) - x_*\|^2. \tag{9.1.17}$$

It follows that (see Exercise 9.1.8)

$$\|x(t) - x_*\|^2 \leq \|x_0 - x_*\|^2 e^{-2\mu t}. \tag{9.1.18}$$

This is the continuum equivalent of the linear convergence rate for gradient descent provided in the discrete setting by Theorem 9.1.3.

Exercise 9.1.8. Suppose that $e(t) \geq 0$ satisfies $e'(t) \leq ae(t)$ for $a \in \mathbb{R}$. Show that $e(t) \leq e(0)e^{at}$. [Hint: Show that $\frac{d}{dt}\log(e(t)) \leq a$ and integrate both sides from 0 to t. Then exponentiate both sides.] △

We can rearrange the heavy ball method iteration (9.1.14) in a similar way as we did for gradient descent to obtain

$$\frac{x_{k+1} - 2x_k + x_{k-1}}{\alpha} + \frac{1-\beta}{\alpha}(x_k - x_{k-1}) = -\nabla f(x_k). \tag{9.1.19}$$

The first term looks like a finite approximation for the second derivative $x''(t)$, while the second term is a backward difference approximation of the first derivative $x'(t)$. The following exercise clarifies this.

Exercise 9.1.9. Show that

$$\frac{x(t) - x(t-h)}{h} = x'(t) + O(h),$$

and

$$\frac{x(t+h) - 2x(t) + x(t-h)}{h^2} = x''(t) + O(h^2)$$

for a smooth curve $x(t)$. To do this, use the Taylor expansions

$$x(t \pm h) = x(t) \pm x'(t)h + \frac{h^2}{2}x''(t) \pm \frac{h^3}{6}x'''(t) + O(h^4). \qquad \triangle$$

Due to Exercise 9.1.9, we have to think of α as the square root of the time step, so $\alpha = \sqrt{h}$ for a time step h, and we take x_k to be a discretization of a smooth curve $x(t)$ with step size h, so $x_k = x(ht)$. In this case (9.1.19) can be viewed as a discretization of the ODE

$$x''(t) + ax'(t) = -\nabla f(x(t)), \tag{9.1.20}$$

where $a = \frac{1-\beta}{\sqrt{\alpha}}$. This ODE corresponds to Newton's law of motion for a body under the forcing of $-\nabla f$ and friction coefficient a. We can see that $\beta \in [0, 1]$ is required for positivity of the friction coefficient, which ensures the system will dissipate energy and slow down over time. For $\beta > 1$, the friction coefficient becomes negative, which is nonphysical and leads to growth of the total energy, and possibly nonconvergence. Of course, one can always take $\beta < 0$, but this will just lead to an excessive amount of friction, slowing down the progress towards the minimizer and often giving worse performance compared to gradient descent. Also, notice that the form of the friction coefficient $a = \frac{1-\beta}{\sqrt{\alpha}}$ suggests that we can choose β as a function of α to keep the amount of friction fixed. That is, we can choose $\beta = 1 - a\sqrt{\alpha}$. This keeps the amount of friction in the system fixed as α is changed.

The analysis of the heavy ball method is more involved, compared to gradient descent. It turns out it is simpler to study the continuum heavy ball

ODE (9.1.20). We provide a result on this first, before turning our analysis to the discrete scheme. The following result is the analog of the result proved in Remark 9.1.7 for gradient descent.

Theorem 9.1.10. *Suppose $x(t)$ solves (9.1.20) with $x(0) = x_0 \in \mathbb{R}^n$, $x'(0) = 0$, and assume f is L-Lipschitz and μ-strongly convex. Let $x_* \in \mathbb{R}^n$ denote the unique minimizer of f. Then we have*

$$\|x(t) - x_*\|^2 \leq \frac{1}{3\mu}\left(3L + 2a^2\right)\|x_0 - x_*\|^2 \exp\left(-\frac{2\mu a t}{3L + 2a^2}\right). \tag{9.1.21}$$

Remark 9.1.11. By Remark 9.1.7 and Theorem 9.1.10, both gradient descent and the heavy ball method converge at the exponential rate e^{-ct} in the continuum, though for different constants $c > 0$. It is not clear from these results why the heavy ball method can converge more quickly; indeed, neither method necessarily converges to x_* faster in the continuum. The difference in the two methods only appears upon discretizing the methods with the backward Euler scheme. The heavy ball method involves a second derivative in time, which allows for a much larger time step (learning rate) α in the discretization, allowing the discrete scheme to converge faster to the minimizer x_*. We prove this in a special case in the discrete setting later, in Theorem 9.1.13.

Proof of Theorem 9.1.10. Let $y(t) = x(t) - x_*$ and note that y satisfies

$$y''(t) + ay'(t) = -\nabla f(x(t)), \tag{9.1.22}$$

since $y'(t) = x'(t)$ and $y''(t) = x''(t)$. Define the energy

$$e(t) = \frac{3}{2}\|y'(t)\|^2 + 3(f(x(t)) - f_*) + \frac{a^2}{2}\|y(t)\|^2 + ay(t)^T y'(t), \tag{9.1.23}$$

where $f_* = f(x_*)$. By strong convexity of f (see (9.1.10)) we have

$$\begin{aligned} e(t) &\geq \frac{3}{2}\|y'(t)\|^2 + \frac{3\mu}{2}\|x(t) - x_*\|^2 + \frac{a^2}{2}\|y(t)\|^2 + ay(t)^T y'(t) \\ &= \frac{3}{2}\|y'(t)\|^2 + \frac{3\mu}{2}\|y(t)\|^2 + \frac{1}{2}\left(\|ay(t) + y'(t)\|^2 - \|y'(t)\|^2\right) \\ &= \|y'(t)\|^2 + \frac{3\mu}{2}\|y(t)\|^2 + \frac{1}{2}\|ay(t) + y'(t)\|^2 \\ &\geq \frac{3\mu}{2}\|y(t)\|^2. \end{aligned}$$

Therefore, $e(t) \geq 0$ and in particular

$$\|x(t) - x_*\|^2 = \|y(t)\|^2 \leq \frac{2}{3\mu}e(t). \tag{9.1.24}$$

The rest of the proof will focus on showing that $e(t)$ decays to zero exponentially fast.

We differentiate $e(t)$ and use (9.1.22) and the identities $x' = y'$ and $x'' = y''$ to compute

$$\begin{aligned}
e'(t) &= 3y'(t)^T y''(t) + 3\nabla f(x(t))^T x'(t) + a^2 y(t)^T y'(t) + a\|y'(t)\|^2 + ay(t)^T y''(t) \\
&= 3y'(t)^T (y''(t) + \nabla f(x(t))) + ay(t)^T (y''(t) + ay'(t)) + a\|y'(t)\|^2 \\
&= -3a\|y'(t)\|^2 - a\nabla f(x(t))^T y(t) + a\|y'(t)\|^2 \\
&= -2a\|y'(t)\|^2 - a(\nabla f(x(t)) - \nabla f(x_*))^T (x(t) - x_*) \\
&\leq -2a\|y'(t)\|^2 - a\mu\|x(t) - x_*\|^2 \\
&= -a(\mu\|y(t)\|^2 + 2\|y'(t)\|^2),
\end{aligned}$$

where we note that we again used strong convexity of f (see Theorem 2.6.5 (iv)) in the last inequality (in addition to $\nabla f(x_*) = 0$).

By the Cauchy-Schwarz inequality and $ab \leq \frac{1}{2}a^2 + \frac{1}{2}b^2$ we have

$$ay(t)^T y'(t) \leq \|ay(t)\|\|y'(t)\| \leq \frac{a^2}{2}\|y(t)\|^2 + \frac{1}{2}\|y'(t)\|^2.$$

Since ∇f is L Lipschitz we have

$$f(x(t)) - f_* \leq \frac{L}{2}\|x(t) - x_*\|^2 = \frac{L}{2}\|y(t)\|^2,$$

and so

$$\begin{aligned}
e(t) &\leq \frac{3}{2}\|y'(t)\|^2 + \frac{3L}{2}\|y(t)\|^2 + \frac{a^2}{2}\|y(t)\|^2 + \frac{a^2}{2}\|y(t)\|^2 + \frac{1}{2}\|y'(t)\|^2 \\
&= \left(\frac{3L}{2} + a^2\right)\|y(t)\|^2 + 2\|y'(t)\|^2 \\
&= \left(\frac{3L + 2a^2}{2\mu}\right)\mu\|y(t)\|^2 + 2\|y'(t)\|^2 \\
&\leq \left(\frac{3L + 2a^2}{2\mu}\right)\left(\mu\|y(t)\|^2 + 2\|y'(t)\|^2\right),
\end{aligned}$$

where the last inequality follows from the fact that $L \geq \mu$ and so

$$\frac{3L + 2a^2}{2\mu} \geq \frac{3L}{2\mu} \geq \frac{3}{2} \geq 1.$$

Therefore we have

$$\mu\|y(t)\|^2 + 2\|y'(t)\|^2 \geq \frac{2\mu e(t)}{3L + 2a^2}.$$

Substituting this into the upper bound on $e'(t)$ deduced above, we obtain

$$e'(t) \leq -\frac{2\mu a e(t)}{3L + 2a^2}.$$

Since $e(t) \geq 0$ it follows from Exercise 9.1.8 that

$$e(t) \leq e(0) \exp\left(-\frac{2\mu a t}{3L + 2a^2}\right).$$

Combining this with (9.1.24) yields

$$\|x(t) - x_*\|^2 \leq \frac{2e(0)}{3\mu} \exp\left(-\frac{2\mu a t}{3L + 2a^2}\right). \tag{9.1.25}$$

The proof is completed by using that $y'(0) = 0$ in the upper bound on $e(t)$ above to deduce that

$$e(0) \leq \left(\frac{3L}{2} + a^2\right) \|y(0)\|^2 = \left(\frac{3L}{2} + a^2\right) \|x_0 - x_*\|^2,$$

and inserting this into (9.1.25). □

The analysis of the discrete heavy ball method is more challenging. To simlify the setting, we will analyze the discrete method in the special case of solving the linear system

$$Ax = b, \tag{9.1.26}$$

where A is an $n \times n$ positive definite and symmetric matrix (e.g., a discrete Laplacian). We can solve this equation by minimizing

$$f(x) = \frac{1}{2} x^T A x - x^T b.$$

In this case, the gradient of f is $\nabla f(x) = Ax - b$ and the Hessian is $\nabla^2 f(x) = A$ (see Section 2.4). Since A is symmetric and real-valued, the normalized eigenvectors $v_1, v_2, \ldots, v_n$ of A form an orthonormal basis for $\mathbb{R}^n$, and the corresponding eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ are all strictly positive. We denote the smallest eigenvalue by $\mu = \lambda_1$, since this is the same as the μ in the definition of strong convexity (9.1.9), or the PL inequality (9.1.11). The Lipschitz constant of the gradient L is the largest eigenvalue $L = \lambda_n$.

Let $x_* \in \mathbb{R}^n$ denote the solution of (9.1.26). Writing x_* in the basis of eigenvectors of A we have

$$x_* = \sum_{i=1}^{n} (x_*^T v_i) v_i.$$

Applying A on both sides we have

$$Ax_* = \sum_{i=1}^{n} \lambda_i (x_*^T v_i) v_i.$$

Since $b = Ax_*$ we can write b in the same basis to find that

$$\sum_{i=1}^{n} (b^T v_i) v_i = b = Ax_* = \sum_{i=1}^{n} \lambda_i (x_*^T v_i) v_i.$$

Equating coefficients on both sides we have

$$x_*^T v_i = \lambda_i^{-1} b^T v_i.$$

Therefore, in the basis $v_1, \ldots, v_n$, we can write x_* as

$$x_* = \sum_{i=1}^{n} \lambda_i^{-1} (b^T v_i) v_i. \tag{9.1.27}$$

Our first result is a warm-up analysis of gradient descent in this special case.

Theorem 9.1.12. *Suppose x_k satisfies*

$$x_{k+1} = x_k - \alpha(Ax_k - b) \tag{9.1.28}$$

for all $k \geq 1$, and assume $\alpha \leq \frac{1}{L}$. Then we have

$$(1 - \alpha L)^k \leq \frac{\|x_k - x_*\|}{\|x_0 - x_*\|} \leq (1 - \alpha\mu)^k. \tag{9.1.29}$$

Proof. Let's write (9.1.28) for k and $k-1$ for convenience

$$x_k = x_{k-1} - \alpha(Ax_{k-1} - b).$$

Taking the dot product with v_i on both sides yields

$$\begin{aligned} x_k^T v_i &= x_{k-1}^T v_i - \alpha(x_{k-1}^T A v_i - b^T v_i) \\ &= x_{k-1}^T v_i - \alpha\lambda_i x_{k-1}^T v_i + \alpha b^T v_i \\ &= x_{k-1}^T v_i (1 - \alpha\lambda_i) + \alpha b^T v_i. \end{aligned}$$

Repeating the same computation for $x_{k-1}^T v_i$ and substituting above yields

$$x_k^T v_i = x_{k-2}^T v_i (1 - \alpha\lambda_i)^2 + \alpha b^T v_i (1 - \alpha\lambda_i) + \alpha b^T v_i.$$

Repeating recursively we obtain

$$x_k^T v_i = x_0^T v_i (1 - \alpha\lambda_i)^k + \alpha b^T v_i \sum_{j=0}^{k-1} (1 - \alpha\lambda_i)^j.$$

The sum on the right hand side is a geometric series, which sums to

$$\sum_{j=0}^{k-1} (1 - \alpha\lambda_i)^j = \frac{1 - (1 - \alpha\lambda_i)^k}{\alpha\lambda_i}.$$

Substituting above we obtain

$$\begin{aligned} x_k^T v_i &= x_0^T v_i (1 - \alpha\lambda_i)^k + \lambda_i^{-1} b^T v_i - \lambda_i^{-1} b^T v_i (1 - \alpha\lambda_i)^k \\ &= (x_0 - \lambda_i^{-1} b)^T v_i (1 - \alpha\lambda_i)^k + \lambda_i^{-1} b^T v_i. \end{aligned}$$

Therefore

$$\begin{aligned} x_k &= \sum_{i=1}^n (x_k^T v_i) v_i \\ &= \sum_{i=1}^n (x_0 - \lambda_i^{-1} b)^T v_i (1 - \alpha\lambda_i)^k v_i + \sum_{i=1}^n \lambda_i^{-1} (b^T v_i) v_i \\ &= \sum_{i=1}^n (x_0 - \lambda_i^{-1} b)^T v_i (1 - \alpha\lambda_i)^k v_i + x_*, \end{aligned}$$

and so

$$x_k - x_* = \sum_{i=1}^n (x_0 - \lambda_i^{-1} b)^T v_i (1 - \alpha\lambda_i)^k v_i.$$

Since $v_1, \ldots, v_n$ is an orthonormal basis for $\mathbb{R}^n$ we have

$$\|x_k - x_*\|^2 = \sum_{i=1}^n \left[(x_0 - \lambda_i^{-1} b)^T v_i \right]^2 (1 - \alpha\lambda_i)^{2k}.$$

The proof is completed by noting that

$$\sum_{i=1}^n \left[(x_0 - \lambda_i^{-1} b)^T v_i \right]^2 = \|x_0 - x_*\|^2$$

and

$$(1-\alpha L)^{2k} \leq (1-\alpha\lambda_i)^{2k} \leq (1-\alpha\mu)^{2k},$$

provided $\alpha \leq \frac{1}{L}$. □

Theorem 9.1.12 shows that gradient descent for solving $Ax = b$ converges at a linear rate, which is at least $1-\alpha\mu$ and at most $1-\alpha L$. In particular, the rate cannot be faster than linear.

We now extend this analysis to the heavy ball method.

Theorem 9.1.13. *Suppose x_k satisfies*

$$x_{k+1} = x_k - \alpha(Ax_k - b) + \beta(x_k - x_{k-1}) \tag{9.1.30}$$

for all $k \geq 2$ and $x_1 = x_0$. Let $\alpha \leq \frac{1}{L}$ and assume

$$(1-\sqrt{\alpha\mu})^2 \leq \beta \leq 1. \tag{9.1.31}$$

Then for all $k \geq 2$ we have

$$\|x_k - x_*\|^2 + \|x_{k+1} - x_*\|^2 \leq 2\beta^k \|x_0 - x_*\|^2. \tag{9.1.32}$$

Remark 9.1.14. Theorem 9.1.13 suggests that the optimal choice for β is $\beta = (1-\sqrt{\alpha\mu})^2$. In this case, we can take square roots on both sides of (9.1.32) and drop the $k+1$ term to obtain

$$\|x_k - x_*\| \leq \sqrt{2}(1-\sqrt{\alpha\mu})^k \|x_0 - x_*\|.$$

Comparing this to the rate for gradient descent, Eq. (9.1.28) from Theorem 9.1.12, we see that the heavy ball method has a faster linear convergence rate of $1-\sqrt{\alpha\mu}$, compared to $1-\alpha\mu$ for gradient descent. We also note that by taking the largest allowable time step of $\alpha = \frac{1}{L}$ in both theorems, we get convergence rates of $1-\kappa$ for gradient descent and $1-\sqrt{\kappa}$ for the heavy ball method, where $\kappa = \frac{\mu}{L}$ is the condition number of the matrix A. For a poorly conditioned matrix, where κ is very small, the heavy ball method gives a substantial improvement in the convergence rate.

We also mention that Theorem 9.1.13 can be extended to smooth functions f that are μ-strongly convex [17].

Proof of Theorem 9.1.13. We take the dot product with v_i on both sides of (9.1.30) to find that

$$\begin{aligned} x_{k+1}^T v_i &= x_k^T v_i - \alpha(x_k^T A v_i - b^T v_i) + \beta(x_k^T v_i - x_{k-1}^T v_i) \\ &= x_k^T v_i - \alpha(\lambda_i x_k^T v_i - b^T v_i) + \beta(x_k^T v_i - x_{k-1}^T v_i). \end{aligned}$$

For notational convenience, we write $c_k = x_k^T v_i$ and $b_i = \alpha b^T v_i$, so that the iteration above can be written as

$$c_{k+1} = (1 + \beta - \alpha\lambda_i)c_k - \beta c_{k-1} + b_i.$$

Since this iteration involves the last two iterates, we need to write the iteration in the form

$$\begin{bmatrix} c_{k+1} \\ c_k \end{bmatrix} = \underbrace{\begin{bmatrix} 1 + \beta - \alpha\lambda_i & -\beta \\ 1 & 0 \end{bmatrix}}_{B} \begin{bmatrix} c_k \\ c_{k-1} \end{bmatrix} + \begin{bmatrix} b_i \\ 0 \end{bmatrix}. \tag{9.1.33}$$

Iterating this equality we find that

$$\begin{bmatrix} c_{k+1} \\ c_k \end{bmatrix} = B^k \begin{bmatrix} c_1 \\ c_0 \end{bmatrix} + \sum_{j=0}^{k-1} B^j \begin{bmatrix} b_i \\ 0 \end{bmatrix}. \tag{9.1.34}$$

To complete the proof, we need to compute the eigenvalues of B. The characteristic polynomial for the matrix B is

$$p_i(\lambda) = \lambda^2 - (1 + \beta - \alpha\lambda_i)\lambda + \beta.$$

The roots of the characteristic polynomial (i.e., the eigenvalues of B) are

$$\lambda = \frac{1}{2}\left(1 + \beta - \alpha\lambda_i \pm \sqrt{(1 + \beta - \alpha\lambda_i)^2 - 4\beta}\right).$$

The discriminant $(1 + \beta - \alpha\lambda_i)^2 - 4\beta$ is negative precisely when

$$\beta \geq \left(1 - \sqrt{\alpha\lambda_i}\right)^2,$$

which is guaranteed by the assumptions $\beta \geq \left(1 - \sqrt{\alpha\mu}\right)^2$ and $\alpha \leq \frac{1}{L}$. In this case, both roots of p_i are complex-valued and have magnitudes

$$|\lambda|^2 = \frac{1}{4}\left((1 + \beta - \alpha\lambda_i)^2 + 4\beta - (1 + \beta - \alpha\lambda_i)^2\right) = \beta.$$

In particular, $\lambda = 1$ is not an eigenvalue of B, and so $I - B$ is invertible and we can use the geometric series formula

$$\sum_{j=0}^{k-1} B^j = (I - B)^{-1} - B^k(I - B)^{-1}.$$

A direct computation shows that

$$(I-B)^{-1}\begin{bmatrix} b_i \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha^{-1}\lambda_i^{-1} b_i \\ \alpha^{-1}\lambda_i^{-1} b_i \end{bmatrix} \begin{bmatrix} x_*^T v_i \\ x_*^T v_i \end{bmatrix}$$

and so

$$\sum_{j=0}^{k-1} B^j \begin{bmatrix} b_i \\ 0 \end{bmatrix} = (I - B^k) \begin{bmatrix} x_*^T v_i \\ x_*^T v_i \end{bmatrix}.$$

It also follows that the spectral norm of B is bounded by $\sqrt{\beta}$ and so $\|B^k x\|^2 \leq \beta^k \|x\|^2$ for any vector x. Applying these observations in (9.1.34) we have

$$\begin{aligned}
(x_k^T v_i - x_*^T v_i)^2 + (x_{k+1}^T v_i - x_*^T v_i)^2 &= \left\| \begin{bmatrix} x_{k+1}^T v_i - x_*^T v_i \\ x_k^T v_i - x_*^T v_i \end{bmatrix} \right\|^2 \\
&= \left\| \begin{bmatrix} c_{k+1} - x_*^T v_i \\ c_k - x_*^T v_i \end{bmatrix} \right\|^2 \\
&= \left\| B^k \begin{bmatrix} c_1 - x_*^T v_i \\ c_0 - x_*^T v_i \end{bmatrix} \right\|^2 \\
&\leq \beta^k \left\| \begin{bmatrix} c_1 - x_*^T v_i \\ c_0 - x_*^T v_i \end{bmatrix} \right\|^2 \\
&= \beta^k \left\| \begin{bmatrix} x_0^T v_i - x_*^T v_i \\ x_0^T v_i - x_*^T v_i \end{bmatrix} \right\|^2 \\
&= 2\beta^k (x_0^T v_i - x_*^T v_i)^2.
\end{aligned}$$

Summing over i we have

$$\|x_k - x_*\|^2 + \|x_{k+1} - x_*\|^2 \leq 2\beta^k \|x_0 - x_*\|^2,$$

which completes the proof. □

9.1.4 Nesterov's Accelerated Gradient Descent

Python Notebook: .ipynb

In the case where f is convex, but not strongly convex, it is still possible to apply momentum methods to obtain faster convergence compared to gradient descent using *Nesterov Accelerated Gradient Descent* method [15]. To describe one variation of Nesterov's method, we set $\lambda_0 = 0$ and define λ_k by

$$\lambda_k = \frac{1 + \sqrt{1 + 4\lambda_{k-1}^2}}{2}. \tag{9.1.35}$$

Nesterov's accelerated gradient descent method then corresponds to the iteration scheme

$$\begin{cases} y_{k+1} = x_k - \alpha \nabla f(x_k) \\ x_{k+1} = y_{k+1} + \dfrac{\lambda_k - 1}{\lambda_{k+1}}(y_{k+1} - y_k), \end{cases} \tag{9.1.36}$$

with initial guess $x_1 = y_1$. Nesterov's method has two steps. First there is a standard gradient descent step, and then there is an extraplolation beyond this with a momentum type term. It is useful to view them in the opposite order, though. For example, combining the two steps together yields

$$\begin{aligned} y_{k+1} &= x_k - \alpha \nabla f(x_k) \\ &= y_k + \beta_k(y_k - y_{k-1}) - \alpha \nabla f(y_k + \beta_k(y_k - y_{k-1})), \end{aligned}$$

where $\beta_k = \frac{\lambda_k - 1}{\lambda_{k+1}}$. Replacing y_k with x_k for comparison, we have

$$x_{k+1} = x_k + \beta_k(x_k - x_{k-1}) - \alpha \nabla f(x_k + \beta_k(x_k - x_{k-1})).$$

We compare this to the heavy ball method (9.1.14), which reads

$$x_{k+1} = x_k + \beta(x_k - x_{k-1}) - \alpha \nabla f(x_k).$$

Aside from the fact that the momentum parameter β is not constant in Nesterov's method, the main difference is that Nesterov can be viewed as taking the momentum step first, and then evaluating the gradient at the extrapolated point $x_k + \beta_k(x_k - x_{k-1})$, instead of at x_k. This allows for a correction in case the momentum step moves in the wrong direction.

As we shall see below (see Theorem 9.1.17), Nesterov's accelerated gradient descent method converges at a rate of $O\left(\frac{1}{t^2}\right)$ for convex functions, which is clear improvement over the $O\left(\frac{1}{t}\right)$ rate for gradient descent (see Theorem 9.1.2). It turns out the $O\left(\frac{1}{t^2}\right)$ rate is optimal for minimizing convex functions with first order (gradient-based) methods. Figure 9.1.3 shows a comparison of gradient descent, the heavy ball method, and Nesterov's accelerated gradient descent for minimizing a quadratic function.

Before proving the $O\left(\frac{1}{t^2}\right)$ convergence rate for Nesterov's method, we make a few observations about λ_k. First, note that since $\lambda_0 = 0$ we have

$$\lambda_1 = \frac{1 + 1}{2} = 0.$$

We also note that the choice of λ_k in (9.1.35) was made so that the following holds:

$$\lambda_{k-1}^2 = \lambda_k^2 - \lambda_k = \lambda_k(\lambda_k - 1). \tag{9.1.37}$$

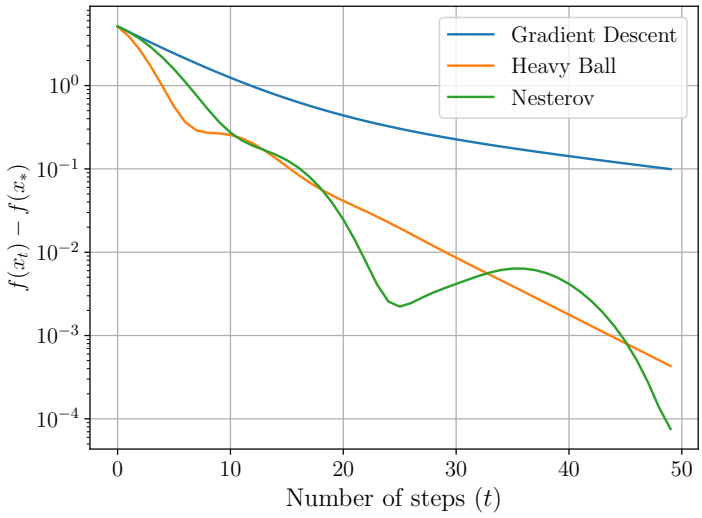


Figure 9.1.3: Comparison of gradient descent, the heavy ball method, and Nesterov acceleration for minimizing a quadratic function. We set $\alpha = 0.01$, $\beta = 0.7$, and took 50 steps for each method.

We also note that λ_k grows linearly in k, and in fact we have the following elementary result.

Proposition 9.1.15. *For all $k \geq 1$ we have*

$$\frac{k}{2} \leq \lambda_k \leq \frac{k}{2} + \frac{1}{4}(3 + \log(k)). \tag{9.1.38}$$

Remark 9.1.16. Proposition 9.1.15 says that $\lambda_k \sim \frac{k}{2}$ as $k \to \infty$, which simply means that

$$\lim_{k\to\infty} \frac{\lambda_k}{\frac{k}{2}} = 1.$$

Using this asymptotic, the momentum coefficient in Nesterov is asympototic to

$$\frac{\lambda_k - 1}{\lambda_{k+1}} \sim \frac{\frac{k}{2} - 1}{\frac{k+1}{2}} = \frac{k-2}{k+1}.$$

It is common in Nesterov acceleration to replace the moment update step in (9.1.36) with the asymptotic equivalent

$$x_{k+1} = y_{k+1} + \frac{k-2}{k+1}(y_{k+1} - y_k), \tag{9.1.39}$$

since the method is simpler to implement and interpret. The same $O\left(\frac{1}{t^2}\right)$ convergence rate that we prove in Theorem 9.1.17 can be established with a similar proof for the alternative momentum update (9.1.39).

Proof of Proposition 9.1.15. We first show that

$$\frac{k}{2} \leq \lambda_k \leq k. \tag{9.1.40}$$

To see this, we note that

$$\lambda_k \geq \frac{1+\sqrt{4\lambda_{k-1}^2}}{2} = \frac{1}{2} + \lambda_{k-1},$$

and using $\sqrt{a^2+b^2} \leq a+b$ we have

$$\lambda_k \leq \frac{1+1+2\lambda_{k-1}}{2} = 1 + \lambda_{k-1}.$$

Then (9.1.40) follows by induction.

We now make the bound more precise, by noting that

$$\lambda_k = \frac{1+\sqrt{1+4\lambda_{k-1}^2}}{2} = \frac{1}{2} + \lambda_{k-1}\sqrt{1+\frac{1}{4\lambda_{k-1}^2}}$$

for $k \geq 2$ so that $\lambda_{k-1} > 0$. Since $f(x) = \sqrt{1+x}$ is a concave function, we have

$$\sqrt{1+x} = f(x) \leq f(0) + f'(0)x = 1 + \frac{1}{2}x.$$

Substituting this above yields

$$\lambda_k \leq \frac{1}{2} + \lambda_{k-1}\left(1 + \frac{1}{8\lambda_{k-1}^2}\right) = \frac{1}{2} + \lambda_{k-1} + \frac{1}{8\lambda_{k-1}}.$$

Since $\lambda_{k-1} \geq \frac{k-1}{2}$ we see that

$$\lambda_k - \lambda_{k-1} \leq \frac{1}{2} + \frac{1}{4(k-1)}$$

for $k \geq 2$. Summing from $k=2$ to $k=t$ we have

$$\lambda_t - 1 \leq \frac{1}{2}(t-1) + \frac{1}{4}\sum_{k=2}^{t}\frac{1}{k-1}.$$

Therefore

$$\lambda_t \leq \frac{t}{2} + \frac{1}{2} + \frac{1}{4}\sum_{k=2}^{t} \frac{1}{k-1}.$$

We finally note that

$$\sum_{k=2}^{t} \frac{1}{k-1} \leq 1 + \int_2^t \frac{1}{x-1}\, dx = 1 + \log(t-1) - \log(1) \leq 1 + \log(t).$$

Substituting this above compeletes the proof for $t \geq 2$. For $t = 1$ we can easily check the inequality holds since $\lambda_1 = 1$. □

We now turn to the proof of the $O\left(\frac{1}{t^2}\right)$ convergence rate for Nesterov's accelerated gradient descent method.

Theorem 9.1.17. *Assume f is convex and ∇f is L-Lipschitz. If $\alpha \leq \frac{1}{L}$ then Nesterov's accelerated gradient descent satisfies*

$$f(y_t) - f(x_*) \leq \frac{2\|x_1 - x_*\|^2}{\alpha(t-1)^2}. \tag{9.1.41}$$

Proof. As in the proof of Theorem 9.1.2, we start with the inequality (9.1.4), which in this case reads

$$f(y_{k+1}) \leq f(x_k) - \frac{1}{2\alpha}\|y_{k+1} - x_k\|^2, \tag{9.1.42}$$

and holds provided $\alpha \leq \frac{1}{L}$. Since f is convex, we can use (9.1.6) with $x = x_k$ to obtain

$$f(y) \geq f(x_k) + \nabla f(x_k)^T(y - x_k) = f(x_k) - \frac{1}{\alpha}(y_{k+1} - x_k)^T(y - x_k)$$

for any $y \in \mathbb{R}^n$. Rearranging we have

$$f(x_k) \leq f(y) - \frac{1}{\alpha}(y_{k+1} - x_k)^T(x_k - y).$$

Inserting this into (9.1.42) we have

$$f(y_{k+1}) - f(y) \leq -\frac{1}{2\alpha}\left(\|y_{k+1} - x_k\|^2 + 2(y_{k+1} - x_k)^T(x_k - y)\right) \tag{9.1.43}$$

for any $y \in \mathbb{R}^n$. We now use (9.1.43) with $y = y_k$, and multiply both sides by $\lambda_k - 1$ to obtain

$$\begin{aligned}(\lambda_k - 1)(f(y_{k+1}) - f(y_k)) \\ \leq -\frac{\lambda_k - 1}{2\alpha}\left(\|y_{k+1} - x_k\|^2 + 2(y_{k+1} - x_k)^T(x_k - y_k)\right).\end{aligned} \tag{9.1.44}$$

We also set $y = x_*$ in (9.1.43) to obtain

$$f(y_{k+1}) - f(x_*) \leq -\frac{1}{2\alpha}\left(\|y_{k+1} - x_k\|^2 + 2(y_{k+1} - x_k)^T(x_k - x_*)\right). \tag{9.1.45}$$

We now carefully add (9.1.44) and (9.1.45). Let us set $\delta_k = f(y_k) - f(x_*)$. Then when we add the left hands of the two equations we obtain

$$\lambda_k f(y_{k+1}) - (\lambda_k - 1)f(y_k) - f(x_*) = \lambda_k \delta_{k+1} - (\lambda_k - 1)\delta_k.$$

Proceeding to add the right hand sides as well, we obtain

$$\begin{aligned}
&\lambda_k \delta_{k+1} - (\lambda_k - 1)\delta_k \\
&\qquad \leq -\frac{\lambda_k}{2\alpha}\|y_{k+1} - x_k\|^2 - \frac{1}{\alpha}(y_{k+1} - x_k)^T(\lambda_k x_k - (\lambda_k - 1)y_k - x_*).
\end{aligned}$$

We now multiply by λ_k on both sides above and use (9.1.37) to obtain

$$\begin{aligned}
&\lambda_k^2 \delta_{k+1} - \lambda_{k-1}^2 \delta_k \\
&\leq -\frac{1}{2\alpha}\left(\|\lambda_k(y_{k+1} - x_k)\|^2 + 2\lambda_k(y_{k+1} - x_k)^T(\lambda_k x_k - (\lambda_k - 1)y_k - x_*)\right) \\
&= -\frac{1}{2\alpha}\Big(\|\lambda_k(y_{k+1} - x_k) + \lambda_k x_k - (\lambda_k - 1)y_k - x_*\|^2 \\
&\qquad\qquad - \|\lambda_k x_k - (\lambda_k - 1)y_k - x_*\|^2\Big) \\
&= -\frac{1}{2\alpha}\left(\|\lambda_k y_{k+1} - (\lambda_k - 1)y_k - x_*\|^2 - \|\lambda_k x_k - (\lambda_k - 1)y_k - x_*\|^2\right) \\
&= -\frac{1}{2\alpha}\left(\|(\lambda_k - 1)(y_{k+1} - y_k) + y_{k+1} - x_*\|^2 - \|\lambda_k x_k - (\lambda_k - 1)y_k - x_*\|^2\right) \\
&= -\frac{1}{2\alpha}\left(\|\lambda_{k+1}(x_{k+1} - y_{k+1}) + y_{k+1} - x_*\|^2 - \|\lambda_k x_k - (\lambda_k - 1)y_k - x_*\|^2\right) \\
&= -\frac{1}{2\alpha}\left(\|\lambda_{k+1}x_{k+1} - (\lambda_{k+1} - 1)y_{k+1} - x_*\|^2 - \|\lambda_k x_k - (\lambda_k - 1)y_k - x_*\|^2\right),
\end{aligned}$$

where we used the Nesterov acclerated gradient descent update (9.1.36) for x_{k+1} in the last line. Both sides above are telescoping sums. Summing from $k = 1$ to $t - 1$ and using $\lambda_0 = 0$ and $\lambda_1 = 1$ we obtain

$$\lambda_{t-1}^2(f(y_t) - f(x_*)) = \lambda_{t-1}^2 \delta_t \leq \frac{1}{2\alpha}\|x_1 - x_*\|^2.$$

Dividing by λ_{t-1}^2 and applying (9.1.40) completes the proof. □

Exercise 9.1.18. Assume f is convex, and let $x(t)$ satisfy the gradient descent ODE

$$x'(t) = -\nabla f(x(t))$$

with $x(0) = x_0$. Show that

$$f(x(t)) - f(x_*) \leq \frac{\|x_0 - x_*\|^2}{2t}$$

where $x_* \in \mathbb{R}^n$ is any minimizer of f. Hint: Define the energy

$$e(t) = t(f(x(t)) - f(x_*)) + \frac{1}{2}\|x(t) - x_*\|^2$$

and show that $e'(t) \leq 0$ so that $e(t) \leq e(0)$. △

Exercise 9.1.19. Consider the version of Nesterov's accelerated gradient descent in the form

$$\begin{cases} y_{k+1} = x_k - \alpha \nabla f(x_k) \\ x_{k+1} = y_{k+1} + \dfrac{k-2}{k+1}(y_{k+1} - y_k). \end{cases} \tag{9.1.46}$$

(i) Show that Nesterov's accelerated gradient descent given in (9.1.46) satisfies

$$\frac{x_{k+1} - 2x_k + x_{k-1}}{\alpha} + a_k \frac{x_k - x_{k-1}}{\sqrt{\alpha}} = -\nabla f(x_k) + \frac{k-2}{k+1}(\nabla f(x_{k-1}) - \nabla f(x_k)),$$

where $a_k = \frac{3}{\sqrt{\alpha}(k+1)}$.

(ii) Assume $x_k = x(\sqrt{\alpha}k)$ is the discretization of a smooth curve $x(t)$ for $t \geq 0$. Explain why it follows from part (i) that when we send $\alpha \to 0$ we obtain that x solves the ordinary differential equation (ODE)

$$x''(t) + \frac{3}{t}x'(t) = -\nabla f(x(t)). \tag{9.1.47}$$

This ODE is sometimes called *continuous time Nesterov*. From the continuum point of view, we can see that Nesterov's method is simlar to the heavy ball method with time-dependent friction $a(t) = \frac{3}{t}$. The friction starts off very large, but vanishes as $t \to \infty$, which allows for accelerated convergence for non-strongly convex functions that may be extremely flat near their minima.

(iii) Assume f is convex and let $x(t)$ satisfy (9.1.47) with $x(0) = x_0$ and $x'(0) = 0$. Show that

$$f(x(t)) - f(x_*) \leq \frac{2\|x_0 - x_*\|^2}{t^2},$$

where $x_* \in \mathbb{R}^n$ is any minimizer of f. Hint: Define the energy functional

$$e(t) = t^2(f(x(t)) - f(x_*)) + 2\left\|x(t) + \frac{t}{2}x'(t) - x_*\right\|^2,$$

and show that $e'(t) \leq 0$ for all $t \geq 0$, so that $e(t) \leq e(0)$, from which the result follows. After differentiating $e(t)$, you will find the inequality below useful, which follows from convexity of f.

$$f(x(t)) - f(x_*) \leq \nabla f(x(t))^T(x(t) - x_*). \qquad \triangle$$

9.1.5 Stochastic gradient descent

Python Notebook: .ipynb

In machine learning, we often are tasked with optimizing functions of the form

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x). \tag{9.1.48}$$

For example, when training neural networks, x represents the trainable weights in the network and $f_i(x) = \ell(g_L(\widetilde{x}_i; x), \widetilde{y}_i)$ where $(\widetilde{x}_i, \widetilde{y}_i)$ are training data, g_L is an L-layer neural network, and ℓ is the loss function.

It can sometimes be computationally burdensome, especially in deep learning where n can be very large, to evaluate the full gradient

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x), \tag{9.1.49}$$

in order to take steps of gradient descent

$$x_{k+1} = x_k - \alpha \nabla f(x_k).$$

In order to accelerate computations, we can approximate the gradient of f by the gradient of any of the f_i. This yields the *stochastic gradient descent* (SGD) algorithm

$$x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k), \tag{9.1.50}$$

where i_k is an index in the range $\{1, \dots, n\}$ chosen uniformly at random at each step. Note we have also allowed the learning rate (time step) α_k to vary with k as well; we shall see the importance of this later on.

Remark 9.1.20. The version of SGD described above has a *batch size* of 1. In general, we may choose a batch size $b \geq 1$ and use the *mini-batch SGD* optimization method

$$x_{k+1} = x_k - \frac{\alpha_k}{b} \sum_{j \in I_k} \nabla f_j(x_k),$$

where $I_k \subset \{1, \dots, n\}$ is a set of b indices indicating the mini-batch and is randomly chosen at each iteration. While we choose to study $b = 1$ in this section, to simplify the exposition, all results extend, with minor adjusments, to mini-batch SGD with $b \geq 2$.

This section requires some familiarity with probability, in particular, with properties of expectation. Since the coordinate i_k is chosen uniformly at random, the expectation of $\nabla f_{i_k}(x_k)$ conditioned on x_k,[1] denoted $\mathbb{E}_k \nabla f_{i_k}(x_k)$ is simply

$$\mathbb{E}_k \nabla f_{i_k}(x_k) = \frac{1}{n} \sum_{i=1}^{n} \nabla_i f(x_k) = \nabla f(x_k). \tag{9.1.51}$$

We thus have that ∇f_{i_k} is an *unbiased* estimator of ∇f. Let us write

$$\xi_k = \nabla f_{i_k}(x_k) - \nabla f(x_k)$$

so that SGD (9.1.50) has the form

$$x_{k+1} = x_k - \alpha_k(\nabla f(x_k) + \xi_k). \tag{9.1.52}$$

Thus, we can view SGD as a noisy version of gradient descnet, corrupted by the noise vector ξ_k. Note that by (9.1.51) we have $\mathbb{E}_k \xi_k = 0$, so the noise vector has mean zero.

In order to establish convergence of SGD, we need to assume the *variance* of the noise vector ξ_k is bounded. In particular, we assume there exists $\sigma \geq 0$ such that

$$\mathbb{E}_k \|\xi_k\|^2 \leq \sigma^2. \tag{9.1.53}$$

[1]For the reader not familiar with probability, the conditional expectation $\mathbb{E}_k \nabla f_{i_k}(x_k)$ is the expectation of $\nabla f_{i_k}(x_k)$ over just the choice of i_k. In particular, we are (not yet) considering the expectation over the previous choices $i_{k-1}, i_{k-2}, \dots, i_1$. To be precise, the conditional expectation $\mathbb{E}_k \nabla f_{i_k}(x_k)$ can be defined by the first equality in (9.1.51), and should be considered as is itself a random variable, since it depends on x_k.

To give some more insight into this condition, let us expand $\mathbb{E}_k\|\xi_k\|^2$ as

$$\begin{aligned}
\mathbb{E}_k\|\xi_k\|^2 &= \mathbb{E}_k\|\nabla f_{i_k}(x_k) - \nabla f(x_k)\|^2 \\
&= \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x_k) - \nabla f(x_k)\|^2 \\
&= \frac{1}{n}\sum_{i=1}^n \left(\|\nabla f_i(x_k)\|^2 - 2\nabla f(x_k)^T\nabla f_i(x_k) + \|\nabla f(x_k)\|^2\right) \\
&= \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x_k)\|^2 - 2\nabla f(x_k)^T\frac{1}{n}\sum_{i=1}^n \nabla f_i(x_k) + \|\nabla f(x_k)\|^2 \\
&= \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x_k)\|^2 - 2\nabla f(x_k)^T\mathbb{E}_k\nabla f_{i_k}(x_k) + \|\nabla f(x_k)\|^2 \\
&= \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x_k)\|^2 - 2\nabla f(x_k)^T\nabla f(x_k) + \|\nabla f(x_k)\|^2 \\
&= \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x_k)\|^2 - \|\nabla f(x_k)\|^2.
\end{aligned}$$

Therefore, the condition (9.1.53) is equivalent to assuming that

$$\frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x)\|^2 \leq \|\nabla f(x)\|^2 + \sigma^2 \tag{9.1.54}$$

for all $x \in \mathbb{R}^n$. We can also write this in terms of conditional expectation, since we have

$$\mathbb{E}_k\|\nabla f_{i_k}(x_k)\|^2 = \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x_k)\|^2 \leq \|\nabla f(x_k)\|^2 + \sigma^2. \tag{9.1.55}$$

As usual, we always assume f is bounded below and attains its minimum value, which we denote by $f_* = \min_{\mathbb{R}^n} f$.

We are now equipped to prove convergence of SGD.

Theorem 9.1.21. *Assume f is L-Lipschitz. Let x_k satisfy (9.1.50) and assume that $\mathbb{E}_k\xi_k = 0$ and $\mathbb{E}_k\|\xi_k\|^2 \leq \sigma^2$ for $\sigma \geq 0$. If $\alpha_k \leq \frac{1}{L}$ then we have*

$$\min_{0\leq k\leq t-1} \mathbb{E}\|\nabla f(x_k)\|^2 \leq \frac{2(f(x_0) - f_*)}{\sum_{k=0}^{t-1}\alpha_k} + L\sigma^2\frac{\sum_{k=0}^{t-1}\alpha_k^2}{\sum_{k=0}^{t-1}\alpha_k}. \tag{9.1.56}$$

In addition, if f is convex, then we have

$$\min_{1\leq k\leq t} (\mathbb{E}f(x_k) - f_*) \leq \frac{\|x_0 - x_*\|^2}{2\sum_{k=0}^{t-1}\alpha_k} + \frac{L\sigma^2\sum_{k=0}^{t-1}\alpha_k^2}{2\sum_{k=0}^{t-1}\alpha_k}. \tag{9.1.57}$$

Proof. The proof of (9.1.56) follows Theorem 9.1.2 closely, with modifications to account for stochasticity. Since f is L-Lipschitz we have

$$\begin{aligned} f(x_{k+1}) &= f(x_k - \alpha_k \nabla f_{i_k}(x_k)) \\ &\leq f(x_k) - \alpha_k \nabla f(x_k)^T \nabla f_{i_k}(x_k) + \frac{L\alpha_k^2}{2} \|\nabla f_{i_k}(x_k)\|^2. \end{aligned}$$

Taking the conditional expectation $\mathbb{E}_k$ on both sides and using (9.1.51) and (9.1.55) we obtain

$$\mathbb{E}_k f(x_{k+1}) \leq f(x_k) - \alpha_k \|\nabla f(x_k)\|^2 + \frac{L\alpha_k^2}{2} \|\nabla f(x_k)\|^2 + \frac{L\alpha_k^2 \sigma^2}{2}.$$

Using the assumption that $\alpha_k \leq \frac{1}{L}$ we have $\frac{L\alpha_k^2}{2} \leq \frac{\alpha_k}{2}$ and so

$$\mathbb{E}_k f(x_{k+1}) \leq f(x_k) - \frac{\alpha_k}{2} \|\nabla f(x_k)\|^2 + \frac{L\alpha_k^2 \sigma^2}{2}. \tag{9.1.58}$$

Rearranging this we have

$$\alpha_k \|\nabla f(x_k)\|^2 \leq 2(f(x_k) - \mathbb{E}_k f(x_{k+1})) + L\alpha_k^2 \sigma^2.$$

We now take the expectation on both sides over all the previous choices of $i_0, i_1, \ldots, i_{k-1}$ during SGD. This expectation is denoted simply by $\mathbb{E}$, and we recall the law of iterated expectations

$$\mathbb{E}\mathbb{E}_k f(x_{k+1}) = \mathbb{E} f(x_{k+1}).$$

This yields

$$\alpha_k \mathbb{E} \|\nabla f(x_k)\|^2 \leq 2\mathbb{E}(f(x_k) - f(x_{k+1})) + L\alpha_k^2 \sigma^2.$$

We now sum both sides above over $k = 0, \ldots, t-1$ to obtain

$$\sum_{k=0}^{t-1} \alpha_k \mathbb{E} \|\nabla f(x_k)\|^2 \leq 2 \sum_{k=0}^{t-1} \mathbb{E}(f(x_k) - f(x_{k+1})) + L\sigma^2 \sum_{k=0}^{t-1} \alpha_k^2.$$

The second summation above is a telescoping sum, which yields

$$\sum_{k=0}^{t-1} \alpha_k \mathbb{E} \|\nabla f(x_k)\|^2 \leq 2(f(x_0) - \mathbb{E} f(x_t)) + L\sigma^2 \sum_{k=0}^{t-1} \alpha_k^2. \tag{9.1.59}$$

We now lower bound the left hand side as follows:

$$\sum_{k=0}^{t-1} \alpha_k \mathbb{E}\|\nabla f(x_k)\|^2 \geq \left(\sum_{k=0}^{t-1} \alpha_k\right) \min_{0\leq k\leq t-1} \mathbb{E}\|\nabla f(x_k)\|^2.$$

Dividing both sides of (9.1.59) by $\sum_{k=0}^{t-1} \alpha_k$ and using the lower bound above, yields

$$\min_{0\leq k\leq t-1} \mathbb{E}\|\nabla f(x_k)\|^2 \leq \frac{2(f(x_0) - \mathbb{E}f(x_t))}{\sum_{k=0}^{t-1} \alpha_k} + L\sigma^2 \frac{\sum_{k=0}^{t-1} \alpha_k^2}{\sum_{k=0}^{t-1} \alpha_k}.$$

The proof of (9.1.56) is completed by using bound $\mathbb{E}f(x_t) \geq \min_{\mathbb{R}^n} f = f_*$.

To prove (9.1.57) when f is convex, we follow the proof of Theorem 9.1.3 closely. Let $x_* \in \mathbb{R}^n$ be any minimizer of f, so that $f_* = f(x_*)$. Applying the argument in Eq. (9.1.8) from Theorem 9.1.3 to the bound in Eq. (9.1.58) yields

$$\mathbb{E}_k f(x_{k+1}) \leq f_* + \frac{1}{2\alpha_k}\left(\|x_k - x_*\|^2 - \|x_{k+1} - x_*\|^2\right) + \frac{L\alpha_k^2\sigma^2}{2}.$$

We now rearrange the equation and take expectations on both sides to obtain

$$\alpha_k(\mathbb{E}f(x_{k+1}) - f_*) \leq \frac{1}{2}\mathbb{E}\left(\|x_k - x_*\|^2 - \|x_{k+1} - x_*\|^2\right) + \frac{L\alpha_k^2\sigma^2}{2}.$$

Summing over k and using that the second sum is telescoping, we have

$$\begin{aligned}\sum_{k=0}^{t-1} \alpha_k(\mathbb{E}f(x_{k+1}) - f_*) &\leq \frac{1}{2}\left(\|x_0 - x_*\|^2 - \mathbb{E}\|x_t - x_*\|^2\right) + \frac{L\sigma^2}{2}\sum_{k=0}^{t-1} \alpha_k^2 \\ &\leq \frac{1}{2}\|x_0 - x_*\|^2 + \frac{L\sigma^2}{2}\sum_{k=0}^{t-1} \alpha_k^2.\end{aligned}$$

Dividing by $\sum_{k=0}^{t-1} \alpha_k$ and lower bounding the left hand side as in the first part proof yields (9.1.57). □

Theorem 9.1.21 shows that the convergence rate for SGD with $\sigma > 0$ depends on the additional quantity

$$A_t := \frac{\sum_{k=0}^{t-1} \alpha_k^2}{\sum_{k=0}^{t-1} \alpha_k}. \tag{9.1.60}$$

We must therefore choose the learning rate α_k at each step so that $A_t \to 0$ as quickly as possible. The following result characterizes common choices for α_k.

Proposition 9.1.22. *The following hold.*

(i) *For a consant learning rate $\alpha_k = \alpha$ we have $A_t = \alpha$ and SGD does not converge.*

(ii) *For the choice $\alpha_k = \alpha/(k+1)^p$ for $0 < p < \frac{1}{2}$ we have*

$$A_t \leq 2^{1-2p}\alpha \left(\frac{1-p}{1-2p}\right) t^{-p}.$$

(iii) *For the choice $\alpha_k = \alpha/\sqrt{k+1}$ we have*

$$A_t \leq (1-p)\alpha \log(t+1)t^{-\frac{1}{2}}.$$

(iv) *For the choice $\alpha_k = \alpha/(k+1)^p$ for $0 < p < \frac{1}{2}$ we have*

$$A_t \leq \alpha \left(\frac{1-p}{2p-1}\right) t^{p-1}.$$

(v) *For the choice $\alpha_k = \alpha/(k+1)$ we have*

$$A_t \leq \frac{\alpha}{(2p-1)(1+\log(t))}.$$

Proof. The proof of (i) is straightforward.

For (ii) and (iii), we note that for any $0 < p < 1$ we have

$$\sum_{k=0}^{t-1} \alpha_k = \alpha \sum_{k=0}^{t-1} \frac{1}{(k+1)^p} \geq \alpha \int_0^t \frac{1}{x^p}\, dx = \frac{\alpha t^{1-p}}{1-p},$$

while for $p = 1$ we have

$$\sum_{k=0}^{t-1} \alpha_k = \alpha \sum_{k=0}^{t-1} \frac{1}{k+1} \geq \alpha \left(1 + \int_1^t \frac{1}{x}\, dx\right) = \alpha(1+\log(t)).$$

For any $0 < p < \frac{1}{2}$ we have

$$\begin{aligned}
\sum_{k=0}^{t-1} \alpha_k^2 &= \alpha^2 \sum_{k=0}^{t-1} \frac{1}{(k+1)^{2p}} \\
&\leq \alpha^2 \int_0^t \frac{1}{(x+1)^{2p}}\, dx \\
&\leq \frac{\alpha^2 (t+1)^{1-2p}}{1-2p} \\
&\leq \frac{2^{1-2p}\alpha^2 t^{1-2p}}{1-2p}.
\end{aligned}$$

This establishes (ii).

For (iii), if $p = \frac{1}{2}$ we have

$$\sum_{k=0}^{t-1} \alpha_k^2 = \alpha^2 \sum_{k=0}^{t-1} \frac{1}{k+1} \leq \alpha^2 \int_0^t \frac{1}{x+1}\, dx \leq \alpha^2 \log(t+1).$$

For (iv) and (v) we take $\frac{1}{2} < p \leq 1$ and compute

$$\sum_{k=0}^{t-1} \alpha_k^2 = \alpha^2 \sum_{k=0}^{t-1} \frac{1}{(k+1)^{2p}} \leq \alpha^2 \int_0^t \frac{1}{(x+1)^{2p}}\, dx \leq \frac{\alpha^2}{2p-1}.$$

This completes the proof. □

Proposition 9.1.22 shows that the optimal power p is $p = \frac{1}{2}$, and up to logarithmic factors, the convergence rate is $A_t = O(t^{-\frac{1}{2}})$. The proposition only considers $0 < p \leq 1$; for $p > 1$ both sums $\sum_{k=0}^{t-1} \alpha_k$ and $\sum_{k=0}^{t-1} \alpha_k^2$ are convergent, and so A_t does not converge to zero, and hence SGD does not converge.

We finally turn to the analysis of SGD for f that are μ-strongly convex.

Theorem 9.1.23. *Assume f is L-Lipschitz and μ-strongly convex. Let x_k satisfy (9.1.50) and assume that $\mathbb{E}_k \xi_k = 0$ and $\mathbb{E}_k \|\xi_k\|^2 \leq \sigma^2$ for $\sigma \geq 0$. For the choice of $\alpha_k = \frac{1}{\mu k + L}$ we have for $k \geq 2$ that*

$$\mathbb{E} f(x_k) - f_* \leq \frac{1}{k+\beta} \left(\beta(f(x_0) - f_*) + \frac{3L\sigma^2}{2\mu^2} \log(k+\beta) \right),$$

where $\beta = \frac{L}{\mu} - 1 \geq 0$.

Proof. We start from (9.1.58) in Theorem 9.1.21, which reads

$$\mathbb{E}_k f(x_{k+1}) \leq f(x_k) - \frac{\alpha_k}{2} \|\nabla f(x_k)\|^2 + \frac{L\alpha_k^2 \sigma^2}{2}.$$

Applying the PL-inequality (9.1.11), which holds for μ-strongly convex functions, we have

$$\mathbb{E}_k f(x_{k+1}) \leq f(x_k) - \alpha_k \mu (f(x_k) - f_*) + \frac{L\alpha_k^2 \sigma^2}{2}.$$

Subtract f_* from both sides and rearrange to obtain

$$\mathbb{E}_k f(x_{k+1}) - f_* \leq (1 - \alpha_k \mu)(f(x_k) - f_*) + \frac{L\alpha_k^2 \sigma^2}{2}.$$

Taking expectations on both sides and setting $e_k = \mathbb{E}(f(x_k) - f_*)$ yields

$$e_{k+1} \leq (1 - \alpha_k \mu) e_k + \frac{L\alpha_k^2 \sigma^2}{2}.$$

We now make the choice $\alpha_k = \frac{1}{\mu k + L}$, which satisfies $\alpha_k \leq \frac{1}{L}$ for all k, and

$$e_{k+1} \leq \left(1 - \frac{1}{k+1+\beta}\right) e_k + \frac{L\sigma^2}{2\mu^2(k+1+\beta)^2},$$

where $\beta = \frac{L}{\mu} - 1 \geq 0$. Iterating we have

$$\begin{aligned} e_{k+1} &\leq \left(\frac{k+\beta}{k+1+\beta}\right)\left(\left(\frac{k-1+\beta}{k+\beta}\right) e_{k-1} + \frac{L\sigma^2}{2\mu^2(k+\beta)^2}\right) + \frac{L\sigma^2}{2\mu^2(k+1+\beta)^2} \\ &= \left(\frac{k-1+\beta}{k+1+\beta}\right) e_{k-1} + \frac{L\sigma^2}{2\mu^2(k+1+\beta)}\left(\frac{1}{k+\beta} + \frac{1}{k+1+\beta}\right). \end{aligned}$$

Continuing by induction we find that

$$e_{k+1} \leq \frac{\beta}{k+1+\beta} e_0 + \frac{L\sigma^2}{2\mu^2(k+1+\beta)} \sum_{j=1}^{k+1} \frac{1}{j+\beta}.$$

The series above can be bounded by

$$\sum_{j=1}^{k+1} \frac{1}{j+\beta} \leq \frac{1}{1+\beta} + \int_1^{k+1} \frac{1}{x+\beta}\, dx \leq 1 + \log(k+1+\beta).$$

This yields

$$e_{k+1} \leq \frac{1}{k+1+\beta}\left(\beta e_0 + \frac{L\sigma^2}{2\mu^2}\left(1 + \log(k+1+\beta)\right)\right).$$

The proof is completed by using the bound

$$1 \leq 2\log(2) \leq 2\log(k+1+\beta)$$

for $k \geq 1$ and then replacing $k+1$ with k. □

Theorem 9.1.23 shows that strong convexity improves the $O(k^{-\frac{1}{2}})$ convergence rate of SGD to $O(k^{-1})$ up to log factors, but this rate is still sublinear, and in particular, SGD does not enjoy the linear convergence rates for strongly convex function that we saw in our analysis of gradient descent.

9.2 Newton's method

Python Notebook: .ipynb

Recalling Exercise 9.1.1, we can interpret of each step of gradient descent as minimizing

$$T(y) = f(x) + \nabla f(x)^T(y-x) + \frac{1}{2\alpha}\|y-x\|^2.$$

We can view this as minimizing the linear approximation of f, with a penalty term to ensure the minimum exists and the step sizes are small. Since T may not approximate f all that well away from the point x, gradient descent requires a small step size α to ensure convergence, which leads to slow convergence with at most linear rates.

A natural way to improve upon gradient descent is to replace the first order Taylor expansion in $T(y)$ with a second order Taylor expansion of f. Recalling Theorem 2.5.7 and omitting the Taylor remainder term, the second order quadratic approximation of f given by Taylor expansion is

$$L(y) = f(x) + \nabla f(x)^T(y-x) + \frac{1}{2}(y-x)^T\nabla^2 f(x)(y-x). \tag{9.2.1}$$

The function L is the quadratic function that best approximates f near x. Each step of Newton's method for optimization is based on minimizing L. Taking the gradient in y we have

$$\nabla L(y) = \nabla f(x) + \nabla^2 f(x)(y-x).$$

Setting $\nabla L(y) = 0$ yields

$$y = x - [\nabla^2 f(x)]^{-1}\nabla f(x), \tag{9.2.2}$$

provided $\nabla^2 f(x)$ is an invertible matrix. Newton's method iterates (9.2.2) until convergence, that is, we generate a sequence of points $x_0, x_1, x_2, \ldots$ satisfying

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1}\nabla f(x_k) \tag{9.2.3}$$

for $k \geq 1$. In this section we analyze the convergence rate of Newton's method.

To prove convergence of Newton's method, we assume that f is μ-strongly convex for $\mu > 0$. This implies that the Hessian matrix $\nabla^2 f(x)$ is positive definite and its smallest eigenvalue is at least μ. In particular, $\nabla^2 f(x)$ is invertible, and the operator (or spectral) norm of $\nabla^2 f(x)^{-1}$ is $1/\mu$. This implies that

$$\|\nabla^2 f(x)^{-1}y\| \leq \frac{1}{\mu}\|y\|, \tag{9.2.4}$$

for any $x, y \in \mathbb{R}^n$.

Theorem 9.2.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$. Assume that f is μ-strongly convex, $\nabla^2 f$ is L-Lipschitz and*

$$\beta := \frac{L}{2\mu^2}\|\nabla f(x_0)\| < 1.$$

Then Newton's method converges as $k \to \infty$ to the unique minimizer of f, and furthermore for any $k \geq 0$ we have

$$\|\nabla f(x_k)\| \leq \frac{2\mu^2}{L}\beta^{2^k}. \tag{9.2.5}$$

Proof. We define

$$e(k) = \frac{L}{2\mu^2}\|\nabla f(x_k)\|.$$

The overall strategy of the proof will be to show that

$$e(k+1) \leq e(k)^2. \tag{9.2.6}$$

This is exactly quadratic convergence, provided $\beta = e(0) < 1$. To prove (9.2.6), we first note that the Newton iteration (9.2.3) satisfies

$$0 = \nabla f(x_k) + \nabla^2 f(x_k)(x_{k+1} - x_k).$$

Therefore, by the Taylor expansion in Theorem 2.5.9 and (9.2.4) we have

$$\begin{aligned}
e(k+1) &= \frac{L}{2\mu^2}\|\nabla f(x_{k+1})\| \\
&= \frac{L}{2\mu^2}\|\nabla f(x_{k+1}) - \nabla f(x_k) - \nabla^2 f(x_k)(x_{k+1} - x_k)\| \\
&\leq \frac{L}{2\mu^2} \cdot \frac{L}{2}\|x_{k+1} - x_k\|^2 \\
&= \frac{L^2}{4\mu^2}\|[\nabla^2 f(x_k)]^{-1}\nabla f(x_k)\|^2 \\
&\leq \frac{L^2}{4\mu^2} \cdot \frac{1}{\mu^2}\|\nabla f(x_k)\|^2 \\
&= e(k)^2.
\end{aligned}$$

This establishes (9.2.6).

Iterating (9.2.6) we obtain

$$e(k) \leq e(0)^{2^k} = \beta^{2^k},$$

which completes the proof. □

Remark 9.2.2. The condition $\beta < 1$ in Theorem 9.2.1 essentially states that we must initialize Newton's method sufficiently close to the minimizer in order to guarante convergence. If the initial guess x_0 not sufficiently close to x_*, it is possible Newton's method will not converge. In practice, Newton's method is usually modified with a time step in the form

$$x_{k+1} = x_k - \alpha[\nabla^2 f(x_k)]^{-1}\nabla f(x_k).$$

The selection of the time step α is often done with a backtracking line search, which attemps to find the best α to give the largest decrease in the objective function f. With the backtracking line search, Newton's method is provably convergent from any initial guess x_0, except that the method may take many steps before it enters the *quadratic convergence* regime where $\beta < 1$.

Remark 9.2.3. As before, we can convert Theorem 9.2.1 into a rate for $\|x_k - x_*\|$ by using (9.1.12) to obtain

$$\|x_k - x_*\| \leq \frac{2\mu}{L}\beta^{2^k}.$$

Bibliography

[1] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[2] M. Benyamin, J. Calder, G. Sundaramoorthi, and A. Yezzi. Accelerated variational PDEs for efficient solution of regularized inversion problems. *Journal of Mathematical Imaging and Vision*, 62(1):10–36, 2020.

[3] J. Bruna and S. Mallat. Invariant scattering convolution networks. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1872–1886, 2013.

[4] A. Chambolle and T. Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of mathematical imaging and vision*, 40(1):120–145, 2011.

[5] O. Chapelle, B. Scholkopf, and A. Zien. *Semi-supervised learning*. MIT, 2006.

[6] R. R. Coifman and S. Lafon. Diffusion maps. *Applied and computational harmonic analysis*, 21(1):5–30, 2006.

[7] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[8] D. L. Donoho and P. B. Stark. Uncertainty principles and signal recovery. *SIAM Journal on Applied Mathematics*, 49(3):906–931, 1989.

[9] D. F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[10] T. Goldstein and S. Osher. The split Bregman method for L1-regularized problems. *SIAM journal on imaging sciences*, 2(2):323–343, 2009.

[11] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] S. G. Johnson and M. Frigo. A modified split-radix FFT with fewer arithmetic operations. *IEEE Transactions on Signal Processing*, 55(1):111–119, 2006.

[14] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[15] Y. E. Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[16] A. Y. Ng, M. I. Jordan, Y. Weiss, et al. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 2:849–856, 2002.

[17] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[18] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[19] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[20] X. Zhu, Z. Ghahramani, and J. D. Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pages 912–919, 2003.