# Convolution

March 22, 2020

### 0.0.1 Setting up

[23]:
```
from bokeh.plotting import figure
from bokeh.io import output_notebook, show,curdoc
from bokeh.themes import Theme
import numpy as np
output_notebook()
theme=Theme(json={})
curdoc().theme=theme
```

## 0.1 Convolution

Convolution is a fundamental operation that appears in many different contexts in both pure and applied settings in mathematics.

**Polynomials: a simple case** Let's look at a simple case first. Consider the space of real valued polynomial functions $\mathcal{P}$.

If $f \in \mathcal{P}$, we have

$$f(z) = \sum_{i=0}^{\infty} a_i z^i$$

where $a_i = 0$ for $i$ sufficiently large. There is an obvious linear map

$$a : \mathcal{P} \to c_{00}(\mathbf{R})$$

where $c_{00}(\mathbf{R})$ is the space of sequences that are eventually zero.

For each integer $i$, we have a projection map $a_i : c_{00}(\mathbf{R}) \to \mathbf{R}$ so that $a_i(f)$ is the coefficient of $z^i$ for $f \in \mathcal{P}$.

Since $\mathcal{P}$ is a ring under addition and multiplication of functions, these operations must be reflected on the other side of the map $a$.

Clearly we can add sequences in $c_{00}(\mathbf{R})$ componentwise, and since addition of polynomials is also componentwise we have:

$$a(f+g) = a(f) + a(g)$$

What about multiplication? We know that

$$a_n(fg) = \sum_{r+s=n} a_r(f)a_s(g) = \sum_{r=0}^{n} a_r(f)a_{n-r}(g) = \sum_{r=0}^{\infty} a_r(f)a_{n-r}(g)$$

where we adopt the convention that $a_i(f) = 0$ if $i < 0$.

Given $(a_0, a_1, \ldots)$ and $(b_0, b_1, \ldots)$ in $c_{00}(\mathbf{R})$, define

$$a * b = (c_0, c_1, \ldots)$$

where

$$c_n = \sum_{j=0}^{\infty} a_j b_{n-j}$$

with the same convention that $a_i = b_i = 0$ if $i < 0$.

This operation is called *convolution*; together with addition it makes the vector space $c_{00}(\mathbf{R})$ into a commutative ring with identity element $(1, 0, 0, \ldots)$. Furthermore by construction we have

$$a(fg) = a(f) * a(g).$$

**Some generalizations - infinite series** There are many generalizations of this.

- we may consider infinite series, not just polynomials. If $a$ and $b$ are such series, we can set $c = (c_r)$ where $c_r = \sum_{r=0}^{\infty} a_r b_{n-r}$ and as usual set $a_i = b_i = 0$ if $i < 0$. Then this sum only involves finitely many terms.
- we can consider infinite series that have a given radius convergence, or that have non-zero radius of convergence. Since the product of convergent series is convergent and the coefficients are computed as if the series were (infinite) polynomials, we can extend convolution to this vector space.

**Fourier Series** A function $f(z)$ in $L^2(S^1)$ has a Fourier Series $\sum_{n=0}^{\infty} a_n e^{2\pi i n z}$ where

$$a_n = \int_{S^1} f(z) e^{-2\pi i n z} dz.$$

These coefficients can be assembled to give a linear map

$$F : L^2(S^1) \to \ell^2(\mathbf{Z})$$

where $\ell^2(\mathbf{Z})$ is the space of functions $h : \mathbf{Z} \to \mathbf{C}$ where $\sum_{n=-\infty}^{\infty} \|h(n)\|^2 < \infty$.

Then $F(fg) = F(f) * F(g)$. Here, the sum giving the Fourier coefficient of the product

$$c(n) = \sum_{r=-\infty}^{\infty} a(r)b(n-r)$$

is infinite and you need the $L^2$ condition to show that it converges.

**Proposition:** The coefficients of the Fourier series of the product of periodic functions is the convolution of the original functions' Fourier coefficients.

The convolution of two functions $f$ and $g$ on the real line is given by the integral

$$(f * g)(\theta) = \int f(\phi)g(\theta - \phi)d\phi.$$

This is a continuous version of the fourier series situation, and you can show that, with the right convergence conditions, that **the fourier transform of the product of functions is the convolution of their fourier transforms.**

**Inverse transform** One last remark. Suppose that $f$ and $g$ are functions on $S^1$. Their convolution is given by the integral

$$(f * g)(z) = \int_{S^1} f(\theta)g(z - \theta)d\theta.$$

The fourier coefficients of the convolution are:

$$a_n(f * g) = \int_{S^1} \int_{S^1} f(\theta)g(z - \theta)dz \; e^{-2\pi i n\theta} d\theta.$$

Make the substitution $\theta = z - u$ and the integral splits up to give

$$a_n(f * g) = a_n(f)a_n(g).$$

So the **Fourier transform of the convolution is the pointwise product of the fourier transforms.**

This holds in the case of the Fourier transform on $\mathbf{R}$ as well, with appropriate convergence assumptions.

## 0.2 An example

Suppose that $f(x)$ is a function on the real line, and $g(x)$ is a square bump

$$g(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

Then

$$(f * g)(y) = \int f(x)g(y - x)dx = \int_{x=y-1}^{y} f(x)dx.$$

In other words the value of $f * g$ at $y$ is the average value of $f$ over the preceeding unit interval – $f * g$ is a "moving average" of $f$.

Let's see how convolution acts like a filtering operation. First we construct a "noisy" sine wave in python and plot it.

[31]:
```
x = np.linspace(-20,20,1000)
y = np.sin(x)+np.random.normal(0,.3,size=x.shape[0])
```

[32]:
```
F=figure()
F.line(x,y,legend_label='noisy')
show(F)
```

The `np.linspace(a,b,n)` function constructs an array $(x_i)$ where $x_0 = a$, $\Delta = (b-a)/n$, and

$$x_i = x_0 + i\Delta$$

for $i = 0, \ldots, n-1$. We can approximate the convolution of $f$ and $g$ by the Riemann sum

$$(f * g)(x_j) = \sum_{i=0}^{n-1} f(x_i) g(x_j - x_i)\Delta = \sum_{i=0}^{n-1} f(x_i) g((j-i)\Delta)\Delta$$

provided we assume that $f$ is zero outside of the interval from $a$ to $b$. So if we have the array of values $f(x_i)$ and we can compute $g$, we can approximate the convolution as a finite sum.

**Exercise:** Look at the documentation for the numpy convolve function. What is the significance of the 'mode' options 'valid', 'same', and 'full'?

In the code below, I choose the option 'same' which means that the arrays are padded with zeros so that the convolution has the same length as the original arrays.

[34]:
```
f = np.vectorize(lambda x: 1 if x>=0 and x<=1 else 0)
g=f(x)
# 1/sum(g) is delta for the riemann sum
fstarg = np.convolve(y,g,'same')/sum(g)
```

[35]:
```
F = figure()
F.line(x,y,legend_label='noisy')
F.line(x,fstarg,color='green',legend_label='smoothed',line_width=3)
F.legend.background_fill_alpha=0
F.legend.click_policy='hide'
F.title.text="A noisy signal and convolution with a symmetric square wave"
show(F)
```

The convolution smooths out the noisy sine wave, but also introduces a small phase shift. (Why?)

### 0.3 Convolution in two dimensions

Just like in one dimension, convolution is a type of filtering operation in two dimensions. The most direct generalization of the example at the beginning of this note would be to consider a function $f(x, y)$ of two variables (our "image"), and take $g(x, y)$ to be the function taking the value 1 on a square centered at the origin and zero elsewhere. Then the convolution $f * g$ would make a new image whose value at a point is the average of the values at the nearby points in a square region.

Let's grab a few MNIST images and do some experiments.

[26]:
```
import pandas as pd
from scipy.signal import convolve2d
from bokeh.layouts import row, column
theme = Theme(json={'attrs':{'Figure':{'toolbar_location':None},'Grid':
 ↪{'visible': False}, 'Axis':{'visible':False}}})
curdoc().theme=theme
images = pd.read_csv('../data/MNIST/train.csv',nrows=50)
```

Let's also make three "stop functions", one of which is just a 3x3 bump, and the other two are vertical and horizontal lines.

[27]:
```
K = np.array([[1,1,1],[1,1,1],[1,1,1]])
KV = np.array([[0,1,0],[0,1,0],[0,1,0]])
KH = np.array([[0,0,0],[1,1,1],[0,0,0]])
```

[28]:
```
F = figure(width=300,height=300)
pic = images.iloc[1,1:].values.reshape(28,28)
F.image(image=[pic],x=0,y=0,dw=1,dh=1,palette='Spectral11')
F.title.text='An MNIST 0 (unfiltered)'
F1 = figure(width=300,height=300)
F1.image(image=[convolve2d(pic,K)/9],x=0,y=0,dw=1,dh=1,palette='Spectral11')
F1.title.text="Averaging Filter"
F2 = figure(width=300,height=300)
F2.image(image=[convolve2d(pic,KV)/3],x=0,y=0,dw=1,dh=1,palette='Spectral11')
F2.title.text="Vertical line filter"
F3 = figure(width=300,height=300)
F3.image(image=[convolve2d(pic,KH)/3],x=0,y=0,dw=1,dh=1,palette='Spectral11')
F3.title.text="Horizontal line filter"
show(column(row(F,F1),row(F2,F3)))
```

# 1 Color Images

We'll take a pokemon image which is an RGBA image – meaning it has 4 channels, one each for Red, Green, Blue, and Alpha (or transparency). Since the image is 120x120, it gives an array of dimensions 120x120x4.

[29]:
```
import imageio
image = imageio.imread('../data/pokemon/delcatty.png')

from scipy.ndimage.filters import convolve
from scipy.ndimage.filters import gaussian_filter
K = np.array([[1,1,1],[1,1,1],[1,1,1]])
H = np.stack([K,K,K,K])
H=H.transpose(1,2,0)/H.sum()
filtered=convolve(image,H)


F = figure(width=300,height=300)
F.image_rgba(image=[image[::-1,::-1,:]],x=0,y=0,dw=1,dh=1)
F.title.text = "original pokemon"
F2 = figure(width=300,height=300)
F2.image_rgba(image=[filtered[::-1,::-1,:]],x=0,y=0,dw=1,dh=1,)
F2.title.text="pokemon with an averaging filter"
F.xgrid.visible=False
F.ygrid.visible=False
F2.xgrid.visible=False
F2.ygrid.visible=False
show(row(F,F2))
```

A gaussian filter means that instead of convolving our image with a "step function" we convolve with a Gaussian. This gives a weighted average for the pixel value at each point. The variance of the Gaussian affects the range over which the averaging is conducted. In the example below, we image our image sits in an infinite plane of zeros for purposes of computing the convolution.

[30]:
```
def filter_image():
    ims = []
    for sigma in np.arange(0,5,1):
        F = figure(width=200,height=200)
        F.image_rgba(image=[gaussian_filter(image[::-1,::-1,:
↪],sigma=sigma,mode='constant',cval=0.0)],x=0,y=0,dw=1,dh=1)
        F.title.text="Gaussian Filter sigma={:.2f}".format(sigma)
        ims.append(F)

    return ims
ims=filter_image()
show(row(ims))
```

## 1.1 Padding and stride

**Padding:** One question with filtering (or convolution) is how to handle boundary cases. Let's suppose for the moment that we are dealing with a discrete convolution kernel, like our 3x3 examples earlier, rather than a continuous one like a Gaussian. What happens when our 3x3 kernel hits an edge of our matrix?

1. Only consider situations where the kernel fits entirely within the field of the image. If the image is NxM, and the kernel is HxW, then you can fit M-W+1 copies of the kernel in each row and N-H+1 copies of the kernel in each column. So the result of the convolution would have size (N-H+1)x(M-W+1).

2. Pad the image (on each side, or top and bottom or both) by zeros – in other words, allow the kernel to overlap the edge of the image, but assume the image is zeros outside of the original bounds. If we add p columns of zeros to each side of the image, then our padded image has M+2p columns and so we can fit M-W+2p+1 copies of the kernel horizontally; and similarly N-H+2p-1 vertically. **Note**: Commonly we can choose W=H, both odd, and then p=(W-1)/2, and we end up with our convolution producing output of the same size as the original image.

**Stride:** What if we move our filter by more than one step each time? The step size is called the *stride* s. In this case the output size is going to be (M-W+2p)/s+1.

### 1.1.1 Discrete Fourier Transform

Let $G = \mathbf{Z}/\mathbf{NZ}$. The *characters* of $G$ are the homomorphisms $G \to \mathbf{C}^*$ where $\mathbf{C}^*$ is the multiplicative group of nonzero complex numbers. In other words, a character of $G$ is a function $f : G \to \mathbf{C}^*$ with $f(x + y) = f(x)f(y)$.

For $j = 0, \ldots, N - 1$, the function

$$f_j(x) = e^{2\pi ijx/N}$$

is a character – it is well defined because of you replace $x$ by $x + N$ the value $f_j(x)$ doesn't change. If $h$ is any function from $G \to \mathbf{C}$, then there is a unique expansion

$$h(x) = \sum_{i=0}^{N} \hat{h}(j) f_j(x).$$

The $\hat{h}(j)$ are called the Fourier coefficients of $h$. To prove this, introduce the Hermitian inner product:

$$< h, k >= \sum_{i=0}^{N-1} h(x)\overline{k}(x)$$

On the one hand the vector space of functions on $G$ is $N$ dimensional; on the other hand you can show that the $N$ functions $f_j$ form an orthogonal basis of the space.

The Fourier coefficients are the projection of the function $h$ onto the corresponding orthonormal basis:

$$\hat{h}(j) = \frac{< h, f_{-j} >}{N}$$

### 1.1.2 Fast Multiplication

Suppose you want to multiply two polynomials of degree $d-1$. If $f(x) = \sum_{i=0}^{d-1} a(i)x^i$ and $g(x) = \sum_{i=0}^{d-1} b(i)x^i$ then the coefficients $c(k)$ of the product are given by

$$c(k) = \sum_{i+j=k} a(i)b(j) = \sum_{i=0}^{2d} a(i)b(k-i)$$

for $k = 0, \ldots, 2d$. Notice that the sum on the right is the convolution of the two functions $a$ and $b$; so thinking of these as functions on the positive integers we see that the $k^{th}$ coefficient of $fg$ is $(a * b)(k)$. Also notice this requires $O(d^2)$ multiplications.

Let's think of $a$, $b$, and $c$ as functions $\mathbf{Z/2dZ} \to \mathbf{C}$.

We can: - compute the Fourier Transforms $\hat{a}$ and $\hat{b}$ of the functions $a$ and $b$.
- compute the pointwise product $\hat{a}\hat{b} = \hat{c}$. - compute the inverse Fourier transform of $\hat{c}$ to recover the function $c$.

This is only a good idea if it is more efficient then the direct multiplication. However, there is an algorithm known as the *Fast Fourier Transform* which computes $\hat{a}$ and $\hat{b}$, and then $\hat{\hat{c}}$, in time $O(d \log d)$, which is better than the $O(d^2)$ of the original method. For $d$ large this gives a much more efficient way to multiply polynomials.

Applied to integers (which are sort of polynomials in powers of 10, or powers of 2) one gets a fast algorithm for multiplying large integers.

[ ]: