Data Science Workshops

Jean Feydy

2017–2018

Practical Information

Statement of intent This semester, Gabriel Peyré will introduce you to major *theoretical* landmarks in today's data sciences: wavelet frames, sparsity priors, deep neural networks and optimal transport.

To supplement these lectures in a way that is most profitable for you, I will strive to teach you how to *implement* mathematical ideas in the most *efficient* way. For better or worse, the ability to design prototypes *quickly* is a blind spot in the license curriculum at the ENS... while being a crucial skill required for any career in applied maths. Fortunately, it's never too late to learn: let's add another string to your bow!

How sessions will pan out I am fully aware that while some students are already fluent in Matlab+Scipy, most of you have barely gone any further than the standard prépa program as far as programming is concerned. In order to let everyone go at its own pace, sessions will keep an informal structure. After a brief recap on Gabriel's lecture, I will introduce you to a handful of numerical tours (`www.numerical-tours.com/`), before answering your questions face-to-face on an individual basis.

If it takes you the whole session to complete the tours, that's great: you definitely learnt something! Otherwise, further readings will be provided and I will stay available to help you out on your projects.

Sessions take place in room Henri Cartan on Tuesdays, from 10.15 a.m. to 12.15 a.m.
A weekly schedule can be found here : `www.math.ens.fr/enseignement/agendas/week.php`.

Contact :

- Mail : `jean.feydy@ens.fr`.
- Office : under the Math Department's glass roof.
- Webpage : `www.math.ens.fr/~feydy/Teaching/index.html`

Part I

Introduction to signal processing

SESSION 1 Entropic Coding

Using `numpy`, `scipy` and `matplotlib`

As of 2017, two main computing frameworks are used by the signal processing community: Matlab and `python+scipy`. Predominant in engineering companies, the former provides stability, consistency and an inch-perfect documentation; `python`'s strong points are its versatility and the compatibility with most cutting edge machine learning libraries.

As you are from the "python generation" as far as the prépa system is concerned, we will leave Matlab aside and focus entirely on `python`, `numpy` (array manipulation), `scipy` (math routines), `matplotlib` (graphical display) and `pytorch` (autodiff and GPU) implementations.

Get a functional workstation To take part in the workshops, you will need to install `python3`, `scipy`, `jupyter` and `pytorch`. If you know how to use your distribution's package system, fine-tuning the details and so on, good for you. Otherwise, the simplest (but heavy!) solution is to use the Anaconda distribution, available here:

`docs.continuum.io/anaconda/install/`.

Then, proceed to the installation of Gabriel's *Numerical Tours*:

`github.com/gpeyre/numerical-tours/archive/master.zip`.

Later on, when you have time, don't forget to install `pytorch`:

`pytorch.org/get-started/locally/`.

Crash-course in `numpy` Unfortunately, the prépa's program leaves aside all the matrix-manipulation syntax that you'll have to get familiar with in the coming weeks. Before getting to the data science, we first have to get used to it!

To get started, go into your `numerical-tours-master/python` directory and type "`jupyter notebook`" in a terminal (hopefully, this works). Then, enjoy the official tutorial:

`docs.scipy.org/doc/numpy/user/quickstart.html`,

and remember that StackOverflow's your best buddy ;-)

For your convenience, here is an adaptation of Julian Gaal's python cheat sheets, available on GitHub:

`github.com/juliangaal/python-cheat-sheet/tree/master/NumPy`,

Basics

Data science algorithms rely on numerical `arrays` which are typically provided by the `numpy` library – later on, we shall replace them with `pytorch`'s `tensors` that support automatic differentiation. The major difference between `lists` and `arrays` is functionality and speed. `lists` give you basic operation, but `numpy` adds FFTs, convolutions, fast searching, basic statistics, linear algebra, histograms, etc.

`axis 0` always refers to row `axis 1` always refers to column

Operator	Description
`np.array([1,2,3])`	1d array
`np.array([(1,2,3),(4,5,6)])`	2d array
`np.arange(start,stop,step)`	range array
`np.linspace(0,2,9)`	Evenly spaced values in array of length ...
`np.zeros((1,2))`	Create and array filled with zeros
`np.ones((1,2))`	Creates an array filled with ones
`np.random.random((5,5))`	Creates random array
`np.empty((2,2))`	Creates an empty array

```
# 1 dimensional
x = np.array([1,2,3])
# 2 dimensional
y = np.array([(1,2,3),(4,5,6)])

x = np.arange(3)
>>> array([0, 1, 2])

y = np.arange(3.0)
>>> array([ 0.,  1.,  2.])

x = np.arange(3,7)
>>> array([3, 4, 5, 6])

y = np.arange(3,7,2)
>>> array([3, 5])
```

Array properties, copying and sorting

Operator	Description
`array.shape`	Dimensions (Rows,Columns)
`len(array)`	Length of Array
`array.ndim`	Number of Array Dimensions
`array.size`	Number of Array Elements

Operator	Description
`array.dtype`	Data Type
`array.astype(type)`	Converts to Data Type
`type(array)`	Type of Array
`np.copy(array)`	Creates copy of array
`other = array.copy()`	Creates deep copy of array
`array.sort()`	Sorts an array
`array.sort(axis=0)`	Sorts axis of array

```
# Sort in ascending order
y = np.array([10, 9, 8, 7, 6, 5, 4, 3, 2, 1])
y.sort()
print(y)
>>> [ 1  2  3  4  5  6  7  8  9  10]
```

Array manipulation routines, maths

Adding or removing elements, combining arrays

Operator	Description
`np.append(a,b)`	Append items to array
`np.insert(array, 1, 2, axis)`	Insert items into array at axis 0 or 1
`np.resize((2,4))`	Resize array to shape(2,4)
`np.delete(array,1,axis)`	Deletes items from array
`np.concatenate((a,b),axis=0)`	Concatenates 2 arrays, adds to end
`np.vstack((a,b))`	Stack array row-wise
`np.hstack((a,b))`	Stack array column wise

```
# Append items to array
a = np.array([(1, 2, 3),(4, 5, 6)])
b = np.append(a, [(7, 8, 9)])
print(b)
>>> [1 2 3 4 5 6 7 8 9]

# Remove index 2 from previous array
print(np.delete(b, 2))
>>> [1 2 4 5 6 7 8 9]

a = np.array([1, 3, 5])
b = np.array([2, 4, 6])
# Stack two arrays row-wise
print(np.vstack((a,b)))
>>> [[1 3 5]
```

```
     [2 4 6]]
# Stack two arrays column-wise
print(np.hstack((a,b)))
>>> [1 3 5 2 4 6]
```

Splitting arrays and more

Operator	Description
`numpy.split()`	
`np.array_split(array, 3)`	Split an array in sub-arrays of (nearly) identical size
`numpy.hsplit(array, 3)`	Split the array horizontally at 3rd index
`other = ndarray.flatten()`	Flattens a 2d array to 1d
`array = np.transpose(other) array.T`	Transpose array

```
# Split array into groups of ~3
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print(np.array_split(a, 3))
>>> [array([1, 2, 3]), array([4, 5, 6]), array([7, 8])]
```

Operations, comparison. All of these work element-wise:

Operator	Description
`np.add(x,y)x + y`	Addition
`np.substract(x,y)x - y`	Subtraction
`np.divide(x,y)x / y`	Division
`np.multiply(x,y)x @ y`	Multiplication
`np.sqrt(x)`	Square Root
`np.sin(x)`	Element-wise sine
`np.cos(x)`	Element-wise cosine
`np.log(x)`	Element-wise natural log
`np.dot(x,y)`	Dot product
`==`	Equal
`!=`	Not equal
`<, <=`	Smaller than, or equal
`>, >=`	Greater than, or equal
`<=`	Smaller than, or equal

```
# If a 1d array is added to a 2d array (or the other way), NumPy
# chooses the array with smaller dimension and adds it to the one
# with bigger dimension
a = np.array([1, 2, 3])
```

```
b = np.array([(1, 2, 3), (4, 5, 6)])
print(np.add(a, b))
>>> [[2 4 6]
     [5 7 9]]
# Using comparison operators will create boolean NumPy arrays
z = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
c = z < 6
print(c)
>>> [ True  True  True  True  True False False False False False]
```

Basic Statistics

Operator	Description
`np.mean(array)`	Mean
`np.median(array)`	Median
`array.corrcoef()`	Correlation Coefficient
`np.std(array)`	Standard Deviation
`array.sum()`	Array-wise sum
`array.min()`	Array-wise minimum value
`array.max(axis=0)`	Maximum value of specified axis
`array.cumsum(axis=0)`	Cumulative sum of specified axis

```
# Statistics of an array
a = np.array([1, 1, 2, 5, 8, 10, 11, 12])

# Standard deviation
print(np.std(a))
>>> 4.2938910093294167

# Median
print(np.median(a))
>>> 6.5
```

Slicing and Subsetting

Operator	Description
`array[i]`	1d array at index i
`array[i,j]`	2d array at index[i][j]
`array[i<4]`	Boolean Indexing, see Tricks
`array[0:3]`	Select items of index 0, 1 and 2
`array[0:2,1]`	Select items of rows 0 and 1 at column 1
`array[:1]`	Select items of row 0 (equals array[0:1, :])
`array[1:2, :]`	Select items of row 1

Operator	Description
`array[ : :-1]`	Reverses `array`

```
b = np.array([(1, 2, 3), (4, 5, 6)])

# The index *before* the comma refers to *rows*,
# the index *after* the comma refers to *columms*
print(b[0:1, 2])
>>> [3]

print(b[:len(b), 2])
>>> [3 6]
```

```
print(b[0, :])
>>> [1 2 3]

print(b[0, 2:])
>>> [3]

print(b[:, 0])
>>> [1 4]

c = np.array([(1, 2, 3), (4, 5, 6)])
d = c[1:2, 0:2]
print(d)
>>> [[4 5]]
# Index trick when working with two np-arrays
a = np.array([1,2,3,6,1,4,1])
b = np.array([5,6,7,8,3,1,2])

# Only saves a at index where b == 1
other_a = a[b == 1]
#Saves every spot in a except at index where b != 1
other_other_a = a[b != 1]

x = np.array([4,6,8,1,2,6,9])
y = x > 5
print(x[y])
>>> [6 8 6 9]

# Even shorter
x = np.array([1, 2, 3, 4, 4, 35, 212, 5, 5, 6])
print(x[x < 5])
>>> [1 2 3 4 4]
```

Matplotlib

Creating and saving plots

Operator	Description
`fig = plt.figure()`	a container that contains all plot elements

`fig.add_axes(); a = fig.add_subplot(222)`
`fig, b = plt.subplots(nrows=3, nclos=2)`
`ax = plt.subplots(2, 2)`
	Initializes subplot on a grid system row-col-num
	Adds subplot
	Creates subplot

`plt.savefig('pic.png')`	Saves plot/figure to image

After configuring your plot, you must use `plt.show()` to make it visible

Plotting data

Operator	Description
`lines = plt.plot(x,y)`	Plot data connected by lines
`plt.scatter(x,y)`	Creates a scatterplot, unconnected data points
`plt.bar(xvalue, data , width, color...)`	simple vertical bar chart
`plt.barh(yvalue, data, width, color...)`	simple horizontal bar
`plt.hist(x, y)`	Plots a histogram
`plt.boxplot(x,y)`	Box and Whisker plot
`plt.violinplot(x, y)`	Creates violin plot
`ax.fill(x, y, color='...')`	Fill area under/between plots
`ax.fill_between(x,y,color='...')`	Fill area under/between plots
`fig, ax = plt.subplots()`	
`im = ax.imshow(img, cmap, vmin...)`	Colormapped or RGB arrays

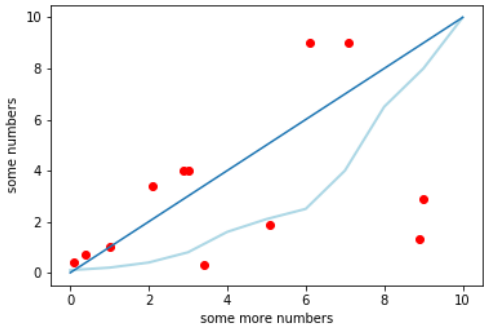
Customization: color, markers, lines, text

Operator	Description
`plt.plot(x, y, color='lightblue', alpha = 0.4)`	colors plot to color blue
`plt.colorbar(mappable, orientation='horizontal')`	`mappable`: the image, contourset etc.
`plt.plot(x, y, marker='*')`	adds * for every data point
`plt.scatter(x, y, marker='.')`	adds . for every data point
`plt.plot(x, y, linewidth=2)`	Set line width
`plt.plot(x, y, ls='solid')`	Set linestyle, `ls` can be ommitted, see 2 below
`plt.plot(x, y, ls='--')`	Set linestyle, `ls` can be ommitted, see below
`plt.plot(x,y,'--', x**2, y**2, '-.')`	Lines are dashed and dash-dot.
`plt.text(1, 1,'Example Text',style='italic')`	Places text at coordinates 1/1
`ax.annotate('some annotation', xy=(10, 10))`	Annotate the point with coordinates `xy`
`plt.title(r'$delta_i=20$', fontsize=10)`	Mathtext

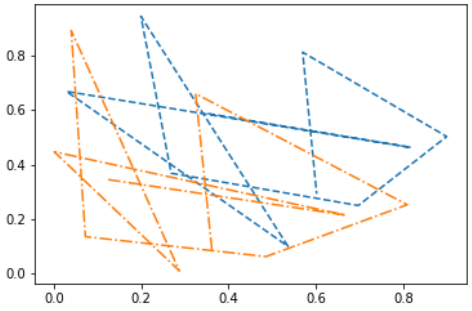
Limits, legend, layout

Operator	Description
`plt.xlim(0, 7)`	Set x-axis to display 0 - 7
`plt.ylim(-0.5, 9)`	Set y-axis to display -0.5 - 9
`ax.set(xlim=[0, 7], ylim=[-0.5, 9])`	Set limits
`plt.margins(x=1.0, y=1.0)`	Set margins: add padding to a plot
`plt.axis('equal')`	Set the aspect ratio of the plot to 1
`plt.title('just a title')`	Set title of plot
`plt.xlabel('...')`	Set label next to x-axis
`plt.ylabel('...')`	Set label next to y-axis
`ax.set(title='axis', ylabel='...', xlabel='...')`	Set title and axis labels
`ax.legend(loc='best')`	No overlapping plot elements
`plt.xticks(x, labels, rotation='vertical')`	Set ticks, example
`ax.xaxis.set(ticks=range(1,3), ticklabels=[-12,"foo"])`	Set x-ticks
`ax.tick_params(axis='y', direction='inout', length=10)`	Make y-ticks longer and go in and out

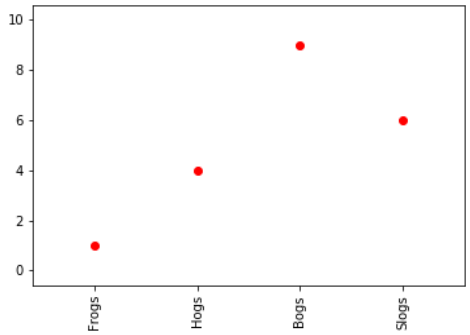
```
import matplotlib.pyplot as plt

x = [1, 2.1, 0.4, 8.9, 7.1, 0.1, 3, 5.1, 6.1, 3.4, 2.9, 9]
y = [1, 3.4, 0.7, 1.3, 9, 0.4, 4, 1.9, 9, 0.3, 4.0, 2.9]
plt.scatter(x,y, color='red')

w = [0.1, 0.2, 0.4, 0.8, 1.6, 2.1, 2.5, 4, 6.5, 8, 10]
z = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
plt.plot(z, w, color='lightblue', linewidth=2)

c = [0,1,2,3,4, 5, 6, 7, 8, 9, 10]
plt.plot(c)

plt.ylabel('some numbers')
plt.xlabel('some more numbers')
plt.show()
```



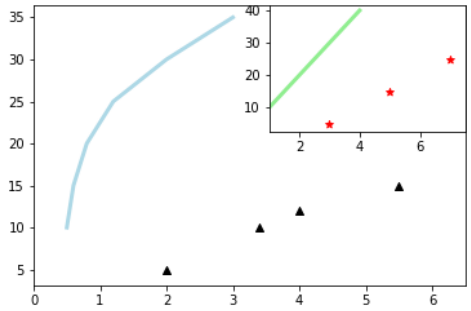

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.rand(10)
y = np.random.rand(10)

plt.plot(x,y,'--', x**2, y**2,'-.')
plt.savefig('lines.png')
plt.show()
```



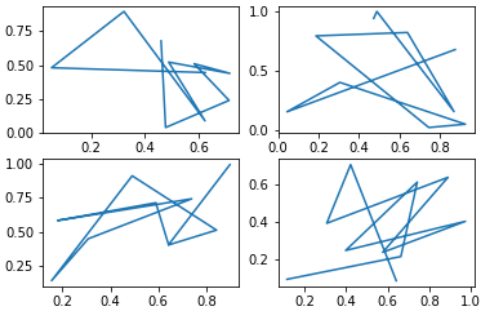
```
import matplotlib.pyplot as plt


x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
# You can specify a rotation for the tick labels in degrees or with keywords.
plt.xticks(x, labels, rotation='vertical')
# Pad margins so that markers don't get clipped by the axes
plt.margins(0.2)
plt.savefig('ticks.png')
plt.show()
```



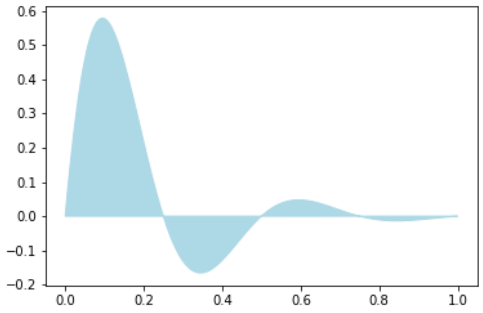

```
import matplotlib.pyplot as plt

x = [0.5, 0.6, 0.8, 1.2, 2.0, 3.0]
y = [10, 15, 20, 25, 30, 35]
z = [1, 2, 3, 4]
w = [10, 20, 30, 40]

fig = plt.figure()
ax =  fig.add_subplot(111)
ax.plot(x, y, color='lightblue', linewidth=3)
ax.scatter([2,3.4,4, 5.5],
               [5,10,12, 15],
               color='black',
               marker='^')
ax.set_xlim(0, 6.5)

ax2 =  fig.add_subplot(222)
ax2.plot(z, w, color='lightgreen', linewidth=3)
ax2.scatter([3,5,7],
               [5,15,25],
               color='red',
               marker='*')
ax2.set_xlim(1, 7.5)

plt.savefig('mediumplot.png')
plt.show()
```



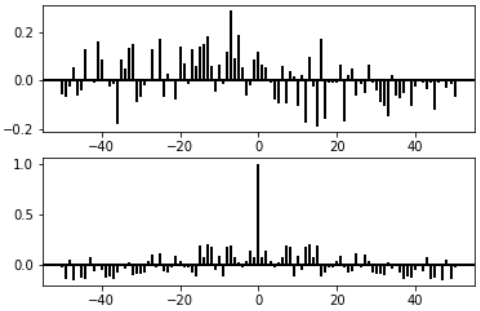
```
import numpy as np
import matplotlib.pyplot as plt

# First way #

x = np.random.rand(10)
y = np.random.rand(10)

figure1 = plt.plot(x,y)

# Second way #

x1 = np.random.rand(10)
x2 = np.random.rand(10)
x3 = np.random.rand(10)
x4 = np.random.rand(10)
y1 = np.random.rand(10)
y2 = np.random.rand(10)
y3 = np.random.rand(10)
y4 = np.random.rand(10)

figure2, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
ax1.plot(x1,y1)
ax2.plot(x2,y2)
ax3.plot(x3,y3)
ax4.plot(x4,y4)

plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 500)
y = np.sin(4 * np.pi * x) * np.exp(-5 * x)

fig, ax = plt.subplots()

ax.fill(x, y, color='lightblue')
plt.show()
```




```
import matplotlib.pyplot as plt
import numpy as np


np.random.seed(0)

x, y = np.random.randn(2, 100)
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.xcorr(x, y, usevlines=True, maxlags=50, normed=True, lw=2)
ax1.grid(True)
ax1.axhline(0, color='black', lw=2)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
ax2.grid(True)
ax2.axhline(0, color='black', lw=2)

plt.show()
```


0.2
0.0
−0.2
−40
−20
0
20
40
1.0
0.5
0.0
−40
−20
0
20
40

Entropy, Huffman code

Yesterday's lesson. In his lecture, Gabriel Peyré introduced the *encoding problem* in its simplest form. We first assume that a memoryless machine generates a stream of characters $X_1, X_2, X_3, ...$ by drawing the X_i's *independently* according to a known probability law (p_i), our *prior* information. Then, as we strive to convert this sequence into a binary stream using a *finite dictionary* – aka. hash map – we make sure that the resulting stream can be decoded (unambiguously) on-the-fly, thus enforcing a *prefix code* constraint.

Crucially, Shannon exhibited a lower bound on the average length of the binary output:

$$\text{Average code length } \geqslant \mathbb{E}_p\left[\log_2 \tfrac{1}{p}\right] = \sum_i p_i \log_2 \tfrac{1}{p_i}, \tag{1.1}$$

and we call this number the *entropy* of the probability distribution (p_i).

Exercise 1: Can you prove this result?
(Hint: How does the "prefix code" constraint influence the distribution of code lengths? Wouldn't it be easier to work with non-integer code lengths?)

Using Huffman's algorithm, we can define a family of dictionaries on *words* (bags of characters) with compression ratios that converge to the limit set by the entropy of the generative distribution. The Shannon bound is thus *sharp* as far as hash tables and asymptotical performances are concerned.

Today's session Today, we'll go through the **Entropic Coding and Compression** numerical tour, implementing Huffman's algorithm along the way. Go into your `numerical-tours-master/python`, open a terminal and type "`jupyter notebook`". Then, in your web browser, open the file `coding_2_entropic.ipynb`. Thanks to the terrific job of Gabriel Peyré, Laurent Condat and Pierre Stock, you should be good to go :-)

If you still have some time left, why wouldn't you read Chris Olah's blogpost on information theory,

`colah.github.io/posts/2015-09-Visual-Information/` ?

He explains the entropy bound much better than I do! Otherwise, you may also install the `pytorch` python library, and get your hands on the official tutorial:

`pytorch.org/tutorials/beginner/pytorch_with_examples.html`.

Summary Applied mathematicians build models. These can look good, canonical or whatever, but *always* rely on assumptions made on the underlying practical problem. Hence, at the end of each session, I will ask you to write a brief summary answering the following questions:

Data model	How do I describe the data given as input?
Objective	What am I trying to achieve?
Practical constraints	What is the range of tools available to solve my problem?
Prior information	How do I encode the prior knowledge on the data?
Mathematical result	What is the theorem that helps me to achieve non-trivial performance?
Pros and cons	In which use cases would my theory be relevant?

Solution

Exercise 1: Let's denote by $\mathcal{X} = \{x_1, \dots, x_I\}$ the alphabet of size I in which the input stream is encoded. We're looking for *dictionaries* $f : \mathcal{X} \to \{0,1\}^{(\mathbb{N})}$ mapping letters to finite binary words such that the output stream $f(X_1)f(X_2)\cdots$ is unambiguously decodable on-the-fly. This is equivalent to saying that f is a *prefix code*:

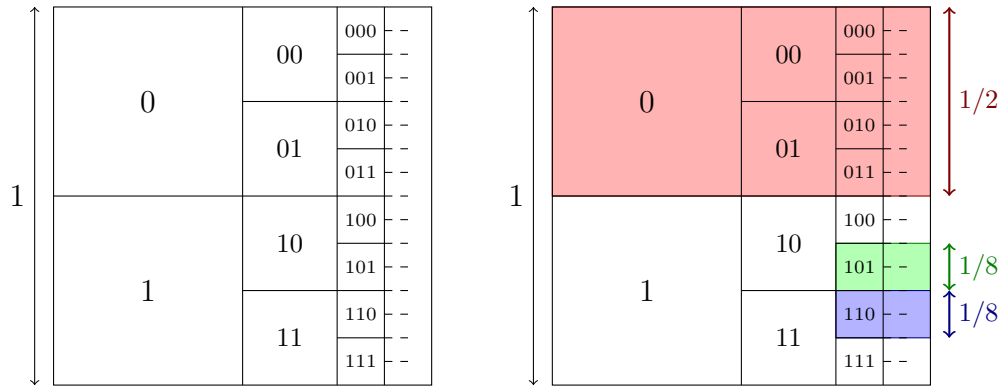
$$\forall\, i, j, \quad f(x_i) \text{ is a prefix of } f(x_j) \iff i = j. \tag{1.2}$$

We must show that there is a lower bound on the compression ratio: if $\ell(f(x_i))$ is the length of the binary word associated to x_i, we have

$$\mathbb{E}_{X\sim p}\left[\ell(f(X))\right] = \sum_{i=1}^{I} p_i \ell(f(x_i)) \geqslant \sum_{i=1}^{I} p_i \log_2 \tfrac{1}{p_i} = \mathbb{E}_p\left[\log_2 \tfrac{1}{p}\right]. \tag{1.3}$$

Prefix code constraint Thankfully, the prefix code constraint can be expressed analytically in terms of the code lengths $\ell(f(x_i))$:

$$\sum_i 2^{-\ell(f(x_i))} \leqslant 1. \tag{1.4}$$



(a) The space of binary codes.

(b) Shadows cast by an encoding dictionary $f = \{x_1 : 0, x_2 : 101, x_2 : 110\}$.

Figure 1.1: Graphical demonstration of the fact that a prefix code $f : \mathcal{X} \to \{0,1\}^{(\mathbb{N})}$ satisfies (1.4).

Constrained minimization Once we have understood how to formulate the prefix code constraint analytically, proving the Shannon bound is surprisingly simple. Right now, we're trying to find a lower bound on

$$B_{\mathbb{N}^\star} = \min\left\{ \sum_{i=1}^{I} p_i l_i \;\middle|\; l_i \in \mathbb{N}^\star,\ \sum_{i=1}^{I} \frac{1}{2^{l_i}} \leqslant 1 \right\}. \tag{1.5}$$

But this quantity is obviously greater than or equal to

$$B_{\mathbb{R}_+^\star} = \inf\left\{ \sum_{i=1}^{I} p_i l_i \;\middle|\; l_i \in \mathbb{R}_+^\star,\ \sum_{i=1}^{I} \frac{1}{2^{l_i}} \leqslant 1 \right\}, \tag{1.6}$$

which is itself equal to

$$B^{=}_{\mathbb{R}^{\star}_{+}} \;=\; \inf\left\{ \sum_{i=1}^{I} p_i l_i \;\middle|\; l_i \in \mathbb{R}^{\star}_{+},\; \sum_{i=1}^{I} \frac{1}{2^{l_i}} = 1 \right\}, \tag{1.7}$$

as you can always reduce the l_i's to *saturate* the constraint. Now, to compute $B^{=}_{\mathbb{R}^{\star}_{+}}$, notice that it is the minimum of the positive linear function

$$g : l \mapsto \sum_{i=1}^{I} p_i l_i \tag{1.8}$$

on the level set $\{h(l) = 1\}$, where $h : l \mapsto \sum_{i=1}^{I} \frac{1}{2^{l_i}}$. Thanks to the theory of Lagrange multipliers – which relies on the implicit function theorem – we know that at the optimum l^{opt}, there exists a scalar λ such that

$$\nabla g(l^{\mathrm{opt}}) \;=\; -\lambda \nabla h(l^{\mathrm{opt}}) \tag{1.9}$$

$$\text{i.e.} \qquad \forall\, i, \quad p_i \;=\; +\lambda \frac{\log 2}{2^{l_i^{\mathrm{opt}}}} \tag{1.10}$$

$$\text{i.e.} \qquad \forall\, i, \quad l_i^{\mathrm{opt}} \;=\; \log_2 \tfrac{1}{p_i} \;+\; \log_2(\lambda \log 2). \tag{1.11}$$

Now, the constraint $\sum_i p_i = 1$ gives us that

$$\sum_i p_i \;=\; 1 \;=\; \lambda \log(2) \sum_i \frac{1}{2^{l_i^{\mathrm{opt}}}}, \tag{1.12}$$

i.e. $\lambda = \frac{1}{\log(2)}$. Hence, we get that $l_i^{\mathrm{opt}} = \log_2 \frac{1}{p_i}$ and

$$B_{\mathbb{N}^{\star}} \;\geqslant\; B^{=}_{\mathbb{R}^{\star}_{+}} \;=\; \sum_i p_i \log_2 \frac{1}{p_i}. \tag{1.13}$$

All-in-all, up to standard computations and a clever diagram, the Shannon entropy bound is thus nothing but **an integer-valued minimization problem whose optimal value is bounded below by its real-valued counterpart**.

Summary Here is what you should keep in mind, as far as entropic encoding is concerned:

Data model	A stream of letters drawn from the alphabet $\mathcal{X}$ is generated using an IID process.
Objective	Compress the input stream into a binary stream, with maximal compression ratio. The output stream should be unambiguously decodable on-the-fly.
Practical constraints	Use dictionaries, mapping one symbol at a time from $\mathcal{X}$ to $\{0,1\}^{(\mathbb{N})}$.
Prior information	Fixed probability law (p_i) on the alphabet $\mathcal{X}$.
Mathematical result	Shannon entropy bound (cf. Lagrange multipliers) + Huffman's code.
Pros and cons	Baseline encoding method, assuming independence between letters in the input stream and the use of simplistic encoding/decoding devices.

SESSION 2 Linear signal processing

The bidimensional Discrete Fourier Transform

Working on 2D images can be much more exciting than plotting 1D signals! In order to keep your spirits up, let's introduce the 2D Discrete Fourier Transform and work on images for the remainder of this session.

If u is an M-by-N array, we define its 2D Discrete Fourier transform on $(\mathbb{Z}/M\mathbb{Z}) \times (\mathbb{Z}/N\mathbb{Z})$ by

$$\widehat{u}(\omega_1, \omega_2) \;=\; \langle e_{\omega_1,\omega_2},\, u\rangle \;=\; \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} u(m,n) e^{-2i\pi\left(\frac{\omega_1 m}{M} + \frac{\omega_2 n}{N}\right)} \tag{2.1}$$

where the base vector $e_{\omega_1,\omega_2} = e_{\omega_1} \otimes e_{\omega_2}$ is the complex-valued "stripes image" associated to the wave vector (ω_1, ω_2) in the periodic Fourier domain $(\mathbb{Z}/M\mathbb{Z}) \times (\mathbb{Z}/N\mathbb{Z})$ – often identified with its zero-centered period $[-M//2, +M//2[\times[-N//2, +N//2[$ – given by

$$e_{\omega_1,\omega_2}(m,n) \;=\; e^{+2i\pi\left(\frac{\omega_1 m}{M} + \frac{\omega_2 n}{N}\right)}. \tag{2.2}$$

Exercise 1: Can you prove a reconstruction formula for the bidimensional Discrete Fourier Transform? Why can we seamlessly implement a 2D-FFT algorithm?

Remember that working with discrete Fourier transforms is akin to assuming that your signal is periodic!

Fourier interpolation

Exercise 2: Implement the zero-padding algorithm introduced by Gabriel yesterday. Discuss. Fill a standard "summary form" as follows:

Data model
Objective
Practical constraints
Prior information
Mathematical result
Pros and cons

Sub-pixel image processing

Translating an image by an integer number of pixels is easy...
But what about going right by 1/3 of pixel?

Exercise 3: Given a signal u of size N with Fourier transform $\widehat{u}$ and a translation vector $z \in \mathbb{R}$, we define the signal $T_z u$ by its Fourier transform:

$$\widehat{T_z u}(\omega) = \begin{cases} e^{-\frac{2i\pi}{N}\omega z}\,\widehat{u}(\omega) & \text{if } -N/2 < \omega < N/2 \\ 0 & \text{if } \omega = N/2 \end{cases}. \tag{2.3}$$

Why is it a sensible choice for the translated-by-z version of u? Why do we have to set $\widehat{T_z u}(N/2)$ to zero? Implement this newly defined "Fourier translation" operator on 2D images.

For those of you who are interested in generalized standard operations on images, I recommend the MVA M2 notes of the "Sub-pixel image processing" course by Lionel Moisan:

`www.math-info.univ-paris5.fr/~moisan/mva/mva_moisan_2012.pdf`.

In section 2.1.6, theorem 3, you will find a reasonable set of axioms on translation operators which are only satisfied by the "Fourier translation" defined above.

Linear image denoising

In your `numerical-tours-master/python` folder, open the file

`denoisingsimp_2b_linear_image.ipynb`

and go through the notebook.

Exercise 4: In which sense is the Wiener filter optimal? Prove it.
How can you interpret its behavior?
Theoretically, the Wiener filter relies on an oracle giving the spectral energy density of the signal. In practice, how do you think it is used?

Summary Can you fill a "summary note" on Wiener filtering?

Data model
Objective
Practical constraints
Prior information
Mathematical result
Pros and cons

Solution

Exercise 1: First, let's show that $(e_{\omega_1,\omega_2})_{(\omega_1,\omega_2)\in(\mathbb{Z}/M\mathbb{Z})\times(\mathbb{Z}/N\mathbb{Z})}$ is an orthonormal basis of $L^2((\mathbb{Z}/M\mathbb{Z})\times(\mathbb{Z}/N\mathbb{Z})\to\mathbb{C})$, up to a constant multiplicative factor. For any choice of indices $(\omega_1,\omega_2),(\omega_1',\omega_2')\in(\mathbb{Z}/M\mathbb{Z})\times(\mathbb{Z}/N\mathbb{Z})$, we have

$$\begin{aligned}\left\langle e_{\omega_1,\omega_2}, e_{\omega_1',\omega_2'}\right\rangle &= \sum_{m,n}\overline{e_{\omega_1}(m)e_{\omega_2}(n)}\cdot e_{\omega_1'}(m)e_{\omega_2'}(n) &(2.4)\\ &= \left\langle e_{\omega_1}, e_{\omega_1'}\right\rangle\cdot\left\langle e_{\omega_2}, e_{\omega_2'}\right\rangle &(2.5)\\ &= MN\delta_{\omega_1,\omega_1'}\delta_{\omega_2,\omega_2'}, &(2.6)\end{aligned}$$

thanks to the summation formula for geometric series. Therefore, the 2D discrete harmonics matrix F (whose columns are given by the harmonics (e_{ω_1,ω_2})) is such that

$$\frac{1}{MN}FF^\star = \mathrm{Id}_{M\times N}, \tag{2.7}$$

where $F^\star$ is the Hermitian conjugate of F, i.e. the DFT operator. This means that for any image u of size M-by-N, we have

$$\begin{aligned}u &= \frac{1}{MN}FF^\star u &(2.8)\\ &= \frac{1}{MN}F\,\widehat{u} &(2.9)\\ &= \frac{1}{MN}\sum_{\omega_1,\omega_2}\widehat{u}(\omega_1,\omega_2)\,e_{\omega_1,\omega_2}. &(2.10)\end{aligned}$$

This is nothing but the standard reconstruction formula for an orthonormal basis, scaled by a normalization factor $1/MN$.

2D-FFT Now, in order to compute $\widehat{u}$ efficiently, one simply has to notice that

$$\begin{aligned}\widehat{u}(\omega_1,\omega_2) &= \sum_{m,n}e^{-2i\pi\left(\frac{\omega_1 m}{M}+\frac{\omega_2 n}{N}\right)}u(m,n) &(2.11)\\ &= \sum_n e^{-2i\pi\frac{\omega_2 n}{N}}\cdot\sum_m e^{-2i\pi\frac{\omega_1 m}{M}}u(m,n) &(2.12)\\ &= \sum_n e^{-2i\pi\frac{\omega_2 n}{N}}\cdot\widehat{u(\cdot,n)}(\omega_1). &(2.13)\end{aligned}$$

That is, one can compute a 2D FFT of u by:

1. Computing the 1D-FFT of every line, storing the results in place.
2. Applying a 1D-FFT to every column of the result above.

All of this at an overall cost of $O(N\cdot M\log(M)+M\cdot N\log(N))$, that is, a $O(MN\log(MN))$ cost instead of the naive $O(M^2N^2)$ one.

Exercise 2: An efficient implementation of the zero-padding algorithm is given below:

```python
from __future__ import division
from nt_toolbox.general import *
from nt_toolbox.signal import *
from scipy.misc import imsave
import numpy as np

def fourier_interpolate(u, H, W) :
    "Interpolates the image u to a shape [u.height+2H, u.width+2W]."
    fu  = fftshift(fft2(u))                          # Center the zero frequency
    pfu = np.pad(fu, [(H,H), (W,W)], 'constant')  # Pad the FFT array
    # We have to balance the "1/MN" factor of the ifft !
    pfu = pfu * (u.shape[0]+2*H)*(u.shape[1]+2*W) / (u.shape[0]*u.shape[1])
    pu  = real(ifft2(ifftshift(pfu)))                # Don't forget the 'real'.
    return pu

# Load and interpolate
name = 'nt_toolbox/data/letter-z.bmp'
x0 = load_image(name) # 70x70
px0 = fourier_interpolate(x0, 140,140) # -->  x5 height, x5 width

# Plot and save
# px0 may be out of the [0,255] range !
sx0 = np.minimum(np.maximum(255*px0, 0), 255).astype(np.uint8)
# Save as an RGB image :
imsave('output/zero_padding.png', np.stack([sx0,sx0,sx0], axis=2))

imageplot(x0)
imageplot(sx0)
```

(a) Source image. (b) After Fourier interpolation.

Figure 2.1: Output of the zero padding interpolation algorithm on a non-smooth signal.

Why doesn'it "work" ? Because the Fourier zero-padding algorithm relies on a lowpass assumption, which is not satisfied as soon as the underlying continuous image presents a *sharp edge*.

Understanding subsampling, the Fourier way To understand this, let's rewrite clearly what is the action of the *zero-padding* algorithm in the Fourier domain. Hereafter, I shall write the main ideas in a cavalier way, slightly abusing the Dirac comb notation – this is the way everyone in the signal processing community remembers those results. All of this can be rigorously proven by doing the calculations on finite numbers of periods, working on finite Fourier series before taking the limit.

Assume that we work with a continuous periodic signal $u(x)$, where $x \in \mathbb{R}/2\pi\mathbb{Z}$, and denote by

$$\text{Ш}_T = \sum_{n\in\mathbb{Z}} \delta_{nT} \tag{2.14}$$

the dirac comb of period T, be it in the spatial or spectral domain. Now, thanks to the Nyquist-Shannon lemma about the Dirichlet kernel ($\widehat{\text{Ш}_T} = \text{Ш}_{2\pi/T}$), and as we know that

$$u \star \frac{1}{\#\mathbb{N}}\text{Ш}_{2\pi} \;=\; u, \qquad \text{we can show that} \qquad \widehat{u} \cdot \frac{1}{\#\mathbb{N}}\text{Ш}_1 \;=\; \widehat{u}, \tag{2.15}$$

i.e. $\widehat{u}$ is a measure supported by $\mathbb{Z}$:

$$\widehat{u} \;=\; \sum_{\omega\in\mathbb{Z}} \widehat{u}(\omega)\delta_\omega, \tag{2.16}$$

where the left-hand hat denotes the *continuous* Fourier transform, and the right-hand one, the coefficient of the Fourier series of u computed on a single period. Now, taking an N-sampling of u is akin to considering the sampled signal $\text{Ш}_{2\pi/N} \cdot u$, whose Fourier transform $\text{Ш}_N \star \widehat{u}$ is an N-periodized version of $\widehat{u}$ – if you're a careful reader, you'll notice the abuse of notations here ;-)

The subampling problem Now, question is: from the N-sampled signal $\text{Ш}_{2\pi/N} \cdot u$, can we reconstruct the M-sampled version $\text{Ш}_{2\pi/M} \cdot u$, where $M > N$?

Equivalently, can we reconstruct $\text{Ш}_M \star \widehat{u}$ from $\text{Ш}_N \star \widehat{u}$?

Spectral folding This is only possible with some kind of prior telling you how to reconstruct M Fourier coefficients from the spectral signal of size N you have at hand.

Related to a smoothness prior, **the main hypothesis here** will be that

$$\forall\, \omega \in \mathbb{Z}, \quad |\omega| \geqslant N/2 \implies \widehat{u}(\omega) \;=\; 0. \tag{2.17}$$

As a consequence, we then know that **the convolution $\text{Ш}_N \star \widehat{u}$ is a trivial copy-paste operation**, (we say that there is *no spectral folding*), and we can simply obtain $\text{Ш}_M \star \widehat{u}$ by zero-padding the centered array used to store a period of $\text{Ш}_N \star \widehat{u}$.

What happens if the lowpass assumption is not satisfied Now, imagine that the support of $\widehat{u}$ is not contained within $]\!]-N/2, +N/2[\![$. Then, the N-sampling operation $\widehat{u} \mapsto \text{Ш}_N \star \widehat{u}$ mixes up high frequencies with lower ones – this is exactly what happens when you watch videos of spinning car wheels. Applying our flawed "lowpass prior", we mistakenly confuse those high frequencies of $\widehat{u}$ with lower ones: **as the contribution of $x \mapsto e^{i(\omega+N)x}$ is indistinguishable from that of $x \mapsto e^{i\omega x}$ on the sampling set** $[\![0, 2\pi/N, \ldots, 2\pi(N-1)/N]\!]$, we will wrongly interpolate all the higher harmonics of u. This mistakes results in the ringing artifacts of Figure 2.1.

Summary Here is what you should remember as far as *zero-padding* is concerned:

Data model	A periodic, continuously supported image u is supposed to be known through a sampling bitmap, a `.png` file for instance. The underlying interpolation assumption is that the spectral support of $\widehat{u}$ is narrow enough for the Nyquist-Shannon theorem to hold.
Objective	Get a finer subsampling of the continuous signal u.
Practical constraints	It's not what I would call a "practical" algorithm...
Prior information	The `.png` file, and a blind faith in your lowpass assumption.
Mathematical result	Nyquist-Shannon theorem on dirac combs: if the spectral support of u is narrow enough, no spectral folding occurs.
Pros and cons	The algorithm is easy to implement... But its baseline hypothesis is more-or-less never satisfied. (A single edge in your signal will result in dreadful ringing artifacts!) More often than not, you'd be better off using a simplistic (bi)linear or spline interpolation. However, this algorithm relies on a fruitful idea: the use of a (mathematically grounded) *transform* to perform signal processing tasks.

Exercise 3: An implementation of the sub-pixel translation algorithm is given on the next page.

Just like in Exercise 2, we assume here that u has narrow spectral support. That is (let's write everything in 1D, as it's easier to read),

$$\forall\ x \in \mathbb{R}/N\mathbb{Z},\ u(x) \ = \ \frac{1}{N} \sum_{|\omega|<N/2} \widehat{u}(\omega)\, e^{2i\pi \frac{\omega x}{N}}. \tag{2.18}$$

Under this very strict assumption, it's pretty straightforward to check that

$$u(x-t) \ = \ \frac{1}{N} \sum_{|\omega|<N/2} \widehat{u}(\omega)\, e^{-2i\pi \frac{\omega t}{N}}\, e^{2i\pi \frac{\omega x}{N}} \tag{2.19}$$

$$\text{i.e.} \qquad \widehat{T_t u}(\omega) \ = \ \begin{cases} e^{-\frac{2i\pi}{N}\omega t}\, \widehat{u}(\omega) & \text{if } -N/2 < \omega < N/2 \\ 0 & \text{if } \omega = N/2 \end{cases}. \tag{2.20}$$

Note that assuming some kind of transitivity of our translation operator, there is no other choice but to kill the $\omega = N/2$ component, when N is even. Indeed, imagine having to translate the corresponding "checkerboard" harmonic $e_{N/2} = (+1, -1, +1, -1, \cdots, +1, -1)$: the only sensible choice for $T_{1/2}(+1, -1, +1, -1, \cdots, +1, -1)$ would be $(0, 0, 0, 0, \cdots, 0, 0)$. But then, we would have

$$T_{-1/2} T_{+1/2} e_{N/2} \ = \ T_{-1/2}(0, \cdots, 0) \ = \ (0, \cdots, 0) \ \neq \ e_{N/2}. \tag{2.21}$$

Summary Overall, the Fourier translation has the same flaws as the zero-padding interpolation of Exercise 2: it is only meaningful if there is no sharp edge in the input signal. A single sharp feature, combined with the nonlocality of the Fourier Transform, results in perceptually awful results at a log-linear cost.

As linear and spline interpolations provide robustness at a discount linear cost, we understand why "Fourier translations" are seldom used in practice.

```python
def fourier_translate(u, X, Y) :
    "Translates the periodic image u using a lowpass assumption."
    (M,N)   = u.shape

    # Coordinates in the DFT domain
    omega_1 = np.hstack( (np.arange(0,M//2), np.arange(-M//2,0))).reshape((M,1))
    omega_2 = np.hstack( (np.arange(0,N//2), np.arange(-N//2,0))).reshape((1,N))

    # We have to put to zero the "checkerboard patterns" frequencies,
    # as there is no way to translate them properly of 1/2 pixel
    # without getting confused with constant images.
    phase_1 = omega_1 / M ; phase_1[-M//2,0] = 0
    phase_2 = omega_2 / N ; phase_2[0,-N//2] = 0

    # phase shifts:
    phase_1 = phase_1 * Y ; phase_2 = phase_2 * X

    # Apply the above to the input data u
    fu = fft2(u)
    ftu = fu * np.exp(-2*np.pi*1j*(phase_1+phase_2)) # column+line = full matrix
    tu = real(ifft2(ftu))
    return tu

# Load and interpolate
name = 'nt_toolbox/data/letter-z.bmp'
x0 = load_image(name) # 70x70
tx0 = fourier_translate(x0, .5, 0)
```

(a) Translation by half a pixel to the right. (b) Translation by a vector $(5.2, 10.3)$.

Figure 2.2: The Fourier-domain translation delocalizes edges.

Exercise 4: As always, we'll write equations in 1D to save space and improve readability. Assume that a complex vector X of size N is corrupted by a complex random Gaussian (white) noise B of mean 0 and covariance $\sigma^2 \mathrm{I}_N$. That is, the coordinates of B are independent and follow the same law $\mathcal{N}(0, \sigma^2)$.

Now, the noisy signal is given as

$$Y \;=\; X + B \tag{2.22}$$

and we're looking for *the best convolution filter, on average, with respect to the squared L^2 norm.* That is, the optimal filter h_{opt} (of size N):

$$h_{\text{opt}} \;=\; \arg\min_h \mathbb{E}_B \left[\|h \star Y - X\|_2^2 \right]. \tag{2.23}$$

Isometries and isotropic noise Now, what's great about Fourier transform is that it is an isometry (up to a constant multiplicative factor) which diagonalizes convolution operators. If we denote by $F^\star$ the DFT operator (see Exercise 1), we remark that

$$\begin{aligned}
\mathbb{E}\left[\|h \star Y - X\|_2^2 \right] &= \mathbb{E}\left[\|h \star (X + B) - X\|_2^2 \right] && (2.24)\\
&= \mathbb{E}\left[\|F^\star \left[h \star (X + B) \right] - F^\star X\|_2^2 \right] \cdot \frac{1}{N} && (2.25)\\
&= \mathbb{E}\left[\left\| \widehat{h} \cdot (\widehat{X} + F^\star B) - \widehat{X} \right\|_2^2 \right] \cdot \frac{1}{N} && (2.26)
\end{aligned}$$

Now, because $F^\star F = N \mathrm{I}_N$, one can show that $F^\star B$ is also a Gaussian white noise, such that its components $C(\omega)$ are independent and follow a normal law $\mathcal{N}(0, N\sigma^2)$. This means that we can separate the multivariate optimization on $\widehat{h}$ into N separate minimizations on the coefficients $\widehat{h}(\omega)$:

$$\arg\min_h \mathbb{E}\left[\|h \star Y - X\|_2^2 \right] \;=\; \arg\min_h \sum_{\omega \in \mathbb{Z}/N\mathbb{Z}} \mathbb{E}\left[|\widehat{h}(\omega) \cdot (\widehat{X}(\omega) + C(\omega)) - \widehat{X}(\omega)|^2 \right] \tag{2.27}$$

Eventually, as $C(\omega)$ is a random variable whose mean is equal to zero (plus the fact that all the other terms are deterministic), we can write

$$\begin{aligned}
\mathbb{E}\left[|\widehat{h}(\omega) \cdot (\widehat{X}(\omega) + C(\omega)) - \widehat{X}(\omega)|^2 \right] &= |\widehat{h}(\omega) - 1|^2 \cdot |\widehat{X}(\omega)|^2 \;+\; |\widehat{h}(\omega)|^2 \cdot N\sigma^2 && (2.28)\\
&= |\widehat{h}(\omega)|^2 \cdot \left(|\widehat{X}(\omega)|^2 + N\sigma^2 \right) \;-\; 2\,\mathrm{Re}(\widehat{h}(\omega)) |\widehat{X}(\omega)|^2 && (2.29)\\
&+ \; |\widehat{X}(\omega)|^2 && (2.30)
\end{aligned}$$

which is minimized in the variable $\widehat{h}(\omega)$ if and only if

$$\widehat{h}(\omega) \;=\; \frac{\frac{1}{N}|\widehat{X}(\omega)|^2}{\frac{1}{N}|\widehat{X}(\omega)|^2 + \sigma^2}. \tag{2.31}$$

Note that the result still holds if you're looking at a real-valued vector X corrupted by a real-valued Gaussian white noise B.

Interpretation The Wiener filter thus scales every harmonic of the signal with respect to the expected "SNR", or signal-to-noise ratio $|\widehat{X}(\omega)|^2/N\sigma^2$. If it is high enough, $\widehat{h}(\omega) \simeq 1$: we rely on the signal, even though it might be a bit noisy. However, if it is too low, the conservative bet of Wiener is to shrink the corresponding frequency, thus preventing the inclusion of noise into the output signal.

As convolution filters act on Fourier frequencies independently, this behavior makes a lot of sense. In practice, one computes typical power spectrums $\frac{1}{N}|\widehat{X}(\omega)|^2$ on training images, and use this prior knowledge to implement an efficient linear denoising on similar images. As natural images tend to have a spectral energy which is heavily concentrated in the lowest frequencies, the Wiener filter can achieve decent performances by "smoothing out" the noise texture.

Summary Here is what you should remember about Wiener filtering:

Data model	A signal X is corrupted by a Gaussian white (i.e. pixel-wise) noise.
Objective	Find a denoised image which is as close to the original X as possible, in the average-L^2 sense.
Practical constraints	Use a fixed convolution filter.
Prior information	The spectral power function of X.
Mathematical result	Gaussian white noise is "compatible" with the Fourier transform, which also diagonalizes convolution operators.
Pros and cons	Easy to implement, but tends to smooth edges out.

SESSION 3 Multiscale analysis using wavelets

Computing a multiscale decomposition, the Fourier way

Given a signal u_0 of size N_0, how can we compute a relevant *multiscale decomposition*? In practice, we're looking for a linear operator

$$\begin{array}{rccc} W & : & \mathbb{C}^{N_0} & \to & \mathbb{C}^{N_1} \times \cdots \times \mathbb{C}^{N_n} \\ & & u_0 & \mapsto & (u_1, \ldots, u_n) \end{array} \tag{3.1}$$

which should be an *isometry*, that is,

$$\forall\ u \in \mathbb{C}^{N_0},\ \ \|u\|_2\ =\ \|Wu\|_2\,, \tag{3.2}$$

and be such that factor-2 sub- and supsamplings of u corresponds to left and right shifts in the multiscale transform n-uple – at least informally. We'll say that u_1 (resp. u_n) encodes the lowest (resp. largest) scales of u.

To build such an operator, one can simply iterate in the Fourier domain a highpass-lowpass routine

$$\begin{array}{rccc} W_{g,h} & : & \mathbb{C}^{N_0} & \to & \mathbb{C}^{N_0} \times \mathbb{C}^{N_0} \\ & & u & \mapsto & (u_{\text{high}}, u_{\text{low}}) = (u \star g, u \star h) \end{array} \tag{3.3}$$

on the successive "second coordinates" or "low frequencies", and choose

$$u_1\ =\ u_0 \star g_1, \qquad\qquad u_2\ =\ (u_0 \star h_1) \star g_2, \qquad\qquad \cdots \tag{3.4}$$

$$u_{n-1}\ =\ (u_0 \star h_1 \star \cdots \star h_{n-2}) \star g_{n-1}, \qquad u_n\ =\ u_0 \star h_1 \star \cdots \star h_{n-1}. \tag{3.5}$$

where the scaled filters are given through their Fourier transforms:

$$\widehat{g_k}(\omega)\ =\ \widehat{g}(2^k\,\omega), \qquad\qquad \widehat{h_k}(\omega)\ =\ \widehat{h}(2^k\,\omega). \tag{3.6}$$

Exercise 1: Show that the above operator W – or equivalently, $W_{g,h}$ – is an isometry if and only if we have

$$\forall\ \omega \in \mathbb{R}, \quad |\widehat{g}(\omega)|^2\ +\ |\widehat{h}(\omega)|^2\ =\ 1. \tag{3.7}$$

A typical strategy is then to choose a nonnegative radial lowpass formula $\widehat{h}(|\omega|) \in \mathbb{R}_+$, and take its companion highpass filter

$$\widehat{g}(|\omega|)\ =\ \sqrt{1 - \widehat{h}(|\omega|)^2}. \tag{3.8}$$

Using those continuous formulas on Discrete Fourier Transforms will be akin to making the assumption that our sampled signal u satisfies the hypothesis of the Nyquist-Shannon theorem.

Exercise 2: Implement the Shannon, Meyer and Simoncelli multiscale decompositions on 1D and 2D signals; according to the terminology of the paper *Steerable Pyramids and Tight Wavelet Frames in* $L^2(R^d)$, by Unser, Chenouard and Van De Ville (2011), they respectively correspond to

$$\widehat{h}_{\text{shannon}}(\omega) = \begin{cases} 1 & \text{if } |\omega| < \pi \\ 0 & \text{if } \pi \leqslant |\omega| \end{cases} \tag{3.9}$$

$$\widehat{h}_{\text{meyer}}(\omega) = \begin{cases} 1 & \text{if } |\omega| \leqslant \pi/2 \\ \cos(|\omega| - \frac{\pi}{2}) & \text{if } \pi/2 \leqslant |\omega| \leqslant \pi \\ 0 & \text{if } \pi \leqslant |\omega| \end{cases} \tag{3.10}$$

$$\widehat{h}_{\text{simoncelli}}(\omega) = \begin{cases} 1 & \text{if } |\omega| \leqslant \pi/2 \\ \cos\left(\frac{\pi}{2}\log_2(\frac{2|\omega|}{\pi})\right) & \text{if } \pi/2 \leqslant |\omega| \leqslant \pi \\ 0 & \text{if } \pi \leqslant |\omega| \end{cases} \tag{3.11}$$

with the convention that the sampled signal u is 1-sampled, i.e. the axis of an `fftshifted` version of $\widehat{u}$ are mapped onto $[-\pi, \pi[$. You may want to plot these curves before diving into your `python` code! Discuss.

Computing a multiscale decomposition, the Mallat way

Now, let's implement the Fast Wavelet Transform of Stéphane Mallat, which was explained by Gabriel in the previous lecture. Combined with the lecture notes, his numerical tour covers pretty much everything you need to know on the topic. So, within Jupyter, open the file called

`wavelet_4_daubechies2d.ipynb`.

To get interaction with your images, and thus become able to zoom in,

simply replace the magic line `%matplotlib inline` with `%matplotlib notebook`.

Beware: Due to changes in `python`'s default behavior, you may encounter a few `TypeErrors` when trying to use sliced indexing. This is because `Jmax = np.log2(n)-1 = 17.0` is a float and not an integer, which confuses the indexing routines... As a workaround, in your notebook and in the `nt_toolbox/signal.py` file,

simply replace `arange(Jmax,Jmin-1,-1)` with `arange(int(Jmax),int(Jmin)-1,-1)`.

Exercise 3: Go through the numerical tour, and answer questions 1 and 3 at the end of the notebook:

1. Compare the approximation obtained using wavelet with different number of vanishing moments. Remember that Daubechies filters are given in Gabriel's notes.
2. Implement a 2-D separable wavelet transform. (No correction will be given next week, as it's just more of the same...)
3. Display a 2-D wavelet by applying the backward transform to a Dirac (i.e. all zeros excepted a single 1 at a well-chosen position).

Exercise 4: In your opinion, how will the multiscale decompositions of Exercise 2 (basically, the Stationary Wavelet Transform) and that of Exercise 3 (the standard Fast Wavelet Transform) compare with each other and the Discrete Fourier Transform, when applied to various signal processing tasks such as compression and denoising ?

Solution

Exercise 1: First, let's note that a few simple choices of g and h do the trick:

– If $(g,h) = (\delta_0, 0)$, i.e. $(\widehat{g},\widehat{h}) = (\mathbf{1}, 0)$, then $W_{g,h}(u) = (u,0)$ and

$$\forall\, u,\ \ \|u\|_2^2 \ =\ \|u\|_2^2 \,+\, 0^2 \ =\ \|(u,0)\|_2^2 \ =\ \|W_{g,h}(u)\|_2^2\,. \tag{3.12}$$

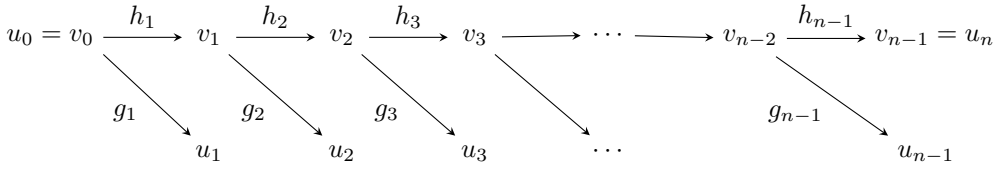
– If $(g,h) = (\frac{1}{\sqrt{2}}\delta_0, \frac{1}{\sqrt{2}}\delta_0)$, i.e. $(\widehat{g},\widehat{h}) = (\frac{1}{\sqrt{2}}\mathbf{1}, \frac{1}{\sqrt{2}}\mathbf{1})$, then $W_{g,h}(u) = (\frac{u}{\sqrt{2}}, \frac{u}{\sqrt{2}})$ and

$$\forall\, u,\ \ \|u\|_2^2 \ =\ \frac{1}{2}\|u\|_2^2 \,+\, \frac{1}{2}\|u\|_2^2 \ =\ \left\|\left(\frac{u}{\sqrt{2}}, \frac{u}{\sqrt{2}}\right)\right\|_2^2 \ =\ \|W_{g,h}(u)\|_2^2\,. \tag{3.13}$$

In the general case, denoting $\mathcal{F}$ the Fourier transform operator, we have

$$\begin{aligned}
W_{g,h} \text{ is an isometry} &\iff \forall\, u, && \|u\|_2^2 \ =\ \|W_{g,h}(u)\|_2^2 && (3.14)\\
&\iff \forall\, u, && \|u\|_2^2 \ =\ \|g\star u\|_2^2 \,+\, \|h\star u\|_2^2 && (3.15)\\
&\iff \forall\, u, && \|\mathcal{F}(u)\|_2^2 \ =\ \|\mathcal{F}(g\star u)\|_2^2 \,+\, \|\mathcal{F}(h\star u)\|_2^2 && (3.16)\\
&\iff \forall\, u, && \|\widehat{u}\|_2^2 \ =\ \|\widehat{g}\cdot\widehat{u}\|_2^2 \,+\, \left\|\widehat{h}\cdot\widehat{u}\right\|_2^2 && (3.17)\\
&\iff \forall\, u, && \int_\omega |\widehat{u}(\omega)|^2 \mathrm{d}\omega \ =\ \int_\omega |\widehat{u}(\omega)|^2\cdot\Big(|\widehat{g}(\omega)|^2 \,+\, |\widehat{h}(\omega)|^2\Big)\,\mathrm{d}\omega && (3.18)\\
&\iff \forall\, u, && 1 \ =\ |\widehat{g}(\omega)|^2 \,+\, |\widehat{h}(\omega)|^2 && (3.19)\\
&\iff \forall\, i, && 1 \ =\ |\widehat{g}_i(\omega)|^2 \,+\, |\widehat{h}_i(\omega)|^2 && (3.20)\\
&\iff \forall\, i, && W_{g_i,h_i} \text{ is an isometry.} && (3.21)
\end{aligned}$$

Now, assuming that the above condition is satisfied, let's show that W is an isometry. Consider the following diagram:



At every step, we have

$$(u_i, v_i) \ =\ W_{g_i,h_i}(v_{i-1}), \qquad \text{so that} \qquad \|v_{i-1}\|_2^2 \ =\ \|u_i\|_2^2 + \|v_i\|_2^2\,. \tag{3.22}$$

Eventually, a recurrence shows that W is an isometry, that is,

$$\forall\, u,\ \ \|u_0\|_2^2 \ =\ \|u_1\|_2^2 + \|u_2\|_2^2 + \cdots + \|u_n\|_2^2\,. \tag{3.23}$$

Exercise 2: An implementation in the Fourier domain is given below.

```
from __future__ import division
from nt_toolbox.general import *
from nt_toolbox.signal import *
from scipy.misc import imsave
import numpy as np
%pylab inline
%matplotlib notebook

# Define the three lowpass functions which act pointwise on arrays "AbsOm",
# filled with the absolute value of pulsation Omega in [-pi, pi[
# (standard convention in continuous signal analysis)
def shannon(AbsOm) :
    fh = np.zeros(AbsOm.shape)
    fh[AbsOm < np.pi] = 1
    return fh

def meyer(AbsOm) :
    fh      = np.zeros(AbsOm.shape)
    indices = np.logical_and(np.pi/2 <= AbsOm, AbsOm <= np.pi)
    om      = AbsOm[indices]
    fh[indices] = np.cos(om - np.pi/2)
    fh[AbsOm <= np.pi/2] = 1
    return fh

def simoncelli(AbsOm) :
    fh      = np.zeros(AbsOm.shape)
    indices = np.logical_and(np.pi/2 <= AbsOm, AbsOm <= np.pi)
    #print(indices)
    om      = AbsOm[indices]
    fh[indices] = np.cos(np.pi/2 * np.log2(2*om/np.pi))
    fh[AbsOm <= np.pi/2] = 1
    return fh

def fWgh(j, fu, name) :
    """
    If fu is the ffshifted fourier transform of a signal u,
    return a highpass-lowpass couple of fftshifted fourier transforms.
    """
    if   len(fu.shape) == 1 : # 1D signal
        AbsOm = np.abs(np.linspace(-np.pi, np.pi, fu.shape[0]+1))[:-1]
    elif len(fu.shape) == 2 : # 2D signal
        Omega_1 = np.linspace(-np.pi, np.pi, fu.shape[0]+1)[:-1].reshape(
                                                            (fu.shape[0],1))
        Omega_2 = np.linspace(-np.pi, np.pi, fu.shape[1]+1)[:-1].reshape(
                                                            (1,fu.shape[1]))
        AbsOm   = np.sqrt( Omega_1**2 + Omega_2**2 )
    methods = {'shannon':shannon, 'meyer':meyer, 'simoncelli':simoncelli}
    fh = methods[name](AbsOm * 2**j)
    fg = np.sqrt( 1 - fh**2 )
    return (fg * fu, fh * fu)
```

```
def multiscale(u, scales, name) :
    if   len(u.shape) == 1 :  # 1D signal
        fu = fft(u)
    elif len(u.shape) == 2 :  # 2D signal
        fu = fft2(u)
    fv = fftshift(fu) # v0 = u0
    funs = []         # list of [u1,...,un]
    for i in range(scales-1) :
        (fv_high, fv_low) = fWgh(i, fv, name)
        funs.append(fv_high)
        fv = fv_low
    funs.append(fv)   # un = vn
    if len(u.shape) == 1 :
        uns = [real( ifft(ifftshift(fun))) for fun in funs]
    elif len(u.shape) == 2 :
        uns = [real(ifft2(ifftshift(fun))) for fun in funs]
    return uns

# Load and interpolate
name = 'nt_toolbox/data/lena_cropped_64.png'
I0 = load_image(name) # 64x64

for filtername in ['shannon', 'meyer', 'simoncelli'] :
    Ins = multiscale(I0, scales = 7, name = filtername)
    pIns = [In + .5 for In in Ins] # Set 0 = gray color for all scales...
    pIns[-1] = Ins[-1]             # Except the lowpass one.

    # Plot and save
    for (n, In) in enumerate(pIns):
        sIn = np.minimum(np.maximum(255*In, 0), 255).astype(np.uint8)
        imsave('output/'+filtername+'_'+str(n)+'.png',
               np.stack([sIn,sIn,sIn], axis=2))
```

Results On the next page, we give the multiscale decompositions corresponding to this cropped version of Lena:

Exercise 3: You can append this solution at the end of the numerical tour.

```
from scipy.misc import imsave
from scipy.signal import daub

daubechies = [np.hstack(([0.],daub(p))) for p in range(1, 11)]
daubechies = [h/norm(h) for h in daubechies]
name = 'nt_toolbox/data/lena_cropped_256.png'
f = load_image(name) ; n = f.shape[0]
for (p,h) in enumerate(daubechies) :
    p = p+1
    print('Daubechies ' + str(p) + ': ' + str(h))
    fW = perform_wavortho_transf(f,Jmin,+1,h)
    # Linear representation ----------------------------------------
    eta = 4
    fWLin = zeros((n,n))
    fWLin[:int(n/eta):,:int(n/eta):] = fW[:int(n/eta):,:int(n/eta):]
    fLin = perform_wavortho_transf(fWLin,Jmin,-1,h)

    print('Linear     SNR : ' + str(snr(f,fLin)))
    save = np.minimum(np.maximum(255*fLin, 0), 255).astype(np.uint8)
    imsave('output/linear_daubechies_'+str(p)+'.png',
           np.stack([save,save,save], axis=2))
    # Nonlinear representation -------------------------------------
    # Simply thresholding above a common value is not a fair comparison
    # in terms of compression efficiency. Instead, we'll save "the best 10%"
    # and compare the resulting images, both perceptually and using SNR.
    fW_flat = fW.ravel()
    indices = np.argsort(abs(fW_flat))
    fW_flat[indices[:int(np.ceil(.95*len(indices)))]] = 0
    fWT = fW_flat.reshape(fW.shape)
    fnl = perform_wavortho_transf(fWT,Jmin,-1,h)

    print('Non-Linear SNR : ' + str(snr(f,fnl)))
    save = np.minimum(np.maximum(255*fnl, 0), 255).astype(np.uint8)
    imsave('output/nonlinear_daubechies_'+str(p)+'.png',
```

Figure 3.1: A 64-by-64 cropped patch from the Lena test image.

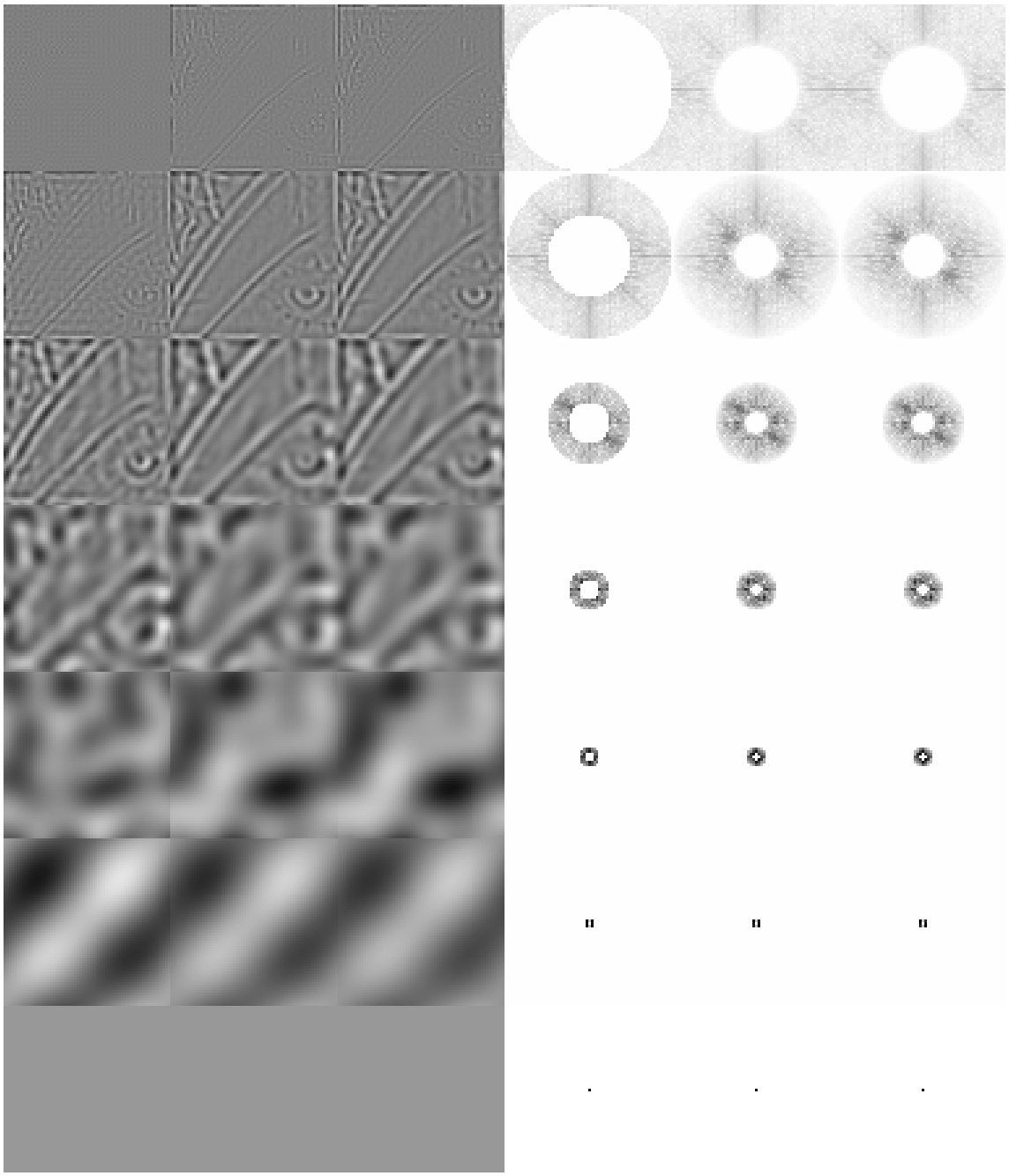


Figure 3.2: Three multiscale decompositions of the same cropped image, in the spatial (left) and spectral (right, square root of the norm of the Fourier coefficient) domains. High scales are above and low scale below; the first, second and third columns are computed using Shannon, Meyer and Simoncelli filters, respectively.
Meyer and Simoncelli decompositions are pretty similar, but the Shannon formula seems to introduce *ringing* artifacts. This is a consequence of the the discontinuities of the bandpass filters: as Shannon bandpass functions are heavy-tailed and delocalized in the spatial domain, sharp edges contaminate the whole image plane.

```
            np.stack([save,save,save], axis=2))
    # Visualize the wavelets ------------------------------------------
    fWav = 0 * fWT ; fWav[13,13] = 10 # Play around with those values !
    Wav = perform_wavortho_transf(fWav,Jmin,-1,h)

    save = np.minimum(np.maximum(255*(Wav+.5), 0), 255).astype(np.uint8)
    imsave('output/daub_'+str(p)+'.png', np.stack([save,save,save], axis=2))
```

Exercise 4: The second type of multiscale decomposition, computed by the Fast Wavelet Transform, offers better performances in compression tasks as it is quite memory-savvy: large scales are optimally subsampled, so that the whole pyramid has the same memory footprint as the original image.
However, this compact representation is not translation-invariant: if you translate a simple image (say, a sampled wavelet) by one pixel to the right, its Wavelet transform will be completely modified as lower scales take up the burden of the representation of a large wavelet "out of the sampling grid". Meanwhile, a simplistic Bank of Filters "Fourier" decomposition would be... translated. Accordingly, robust denoising and analysis schemes will favor redundant representations such as the one computed in Exercises 1 and 2.

Figure 3.3: Non-linear approximations of a cropped Lena image, keeping only the largest five-percent coefficients of a 2D wavelet decomposition. Here, we're using Daubechies filters with $p = 1$ (top left) to $p = 8$ (bottom right) vanishing moments.
As evidenced here, having many vanishing moments allows one to compress the smooth parts of the signal more efficiently. But with larger filter sizes, large "lone" wavelet coefficients can give birth to unpleasant visual artifacts, especially next to the borders of smooth regions. In this case, I would say that the filter with three vanishing moments offers the best trade-off.

Figure 3.4: A typical 2D-Daubechies wavelet on the diagonal (at position 14-14 in the dyadic decomposition), where the number of vanishing moments goes from $p = 1$ (top left) to $p = 8$ (bottom right).

SESSION 4 Nonlinear approximations

Image approximation with orthogonal bases

Gabriel's numerical tour makes for a perfect workshop session in itself: within Jupyter, open the file called

`coding_1_approximation.ipynb`

and complete Exercises 1 to 12.

To get interaction with your images, and thus become able to zoom in,

simply replace the magic line `%matplotlib inline` with `%matplotlib notebook`.

Beware: Due to changes in `python`'s default behavior, you may encounter a few `TypeErrors` when trying to use sliced indexing. This is because `Jmax = np.log2(n)-1 = 17.0` is a float and not an integer, which confuses the indexing routines... As a workaround, in your notebook and in the `nt_toolbox/signal.py` file,

simply replace `arange(Jmax,Jmin-1,-1)` with `arange(int(Jmax),int(Jmin)-1,-1)`.

Solution

All the solutions are located in your `nt_solutions/coding_1_approximation/` folder.

SESSION 5 Compression and denoising

Image denoising with orthogonal bases

Gabriel's numerical tour makes for a perfect workshop session in itself: within Jupyter, complete the notebook called

`denoisingwav_2_wavelet_2d.ipynb`.

To get interaction with your images, and thus become able to zoom in,

simply replace the magic line `%matplotlib inline` with `%matplotlib notebook`.

Beware: Due to changes in `python`'s default behavior, you may encounter a few `TypeErrors` when trying to use sliced indexing. This is because `Jmax = np.log2(n)-1 = 17.0` is a float and not an integer, which confuses the indexing routines... As a workaround, in your notebook and in the `nt_toolbox/signal.py` file,

simply replace `arange(Jmax,Jmin-1,-1)` with `arange(int(Jmax),int(Jmin)-1,-1)`.

Solution

The solution of the mini-exercise is located in the solution folder `nt_solutions/denoisingwav_2_wavelet_2d/`.

Exercise 1:

```
ms   = array([1,2,3,4,8,16,32])
snrs = []
for m in ms:
    print('m = ' + str(m))
    [dY,dX] = meshgrid(arange(0,m),arange(0,m))
    delta = concatenate( (dX.reshape(m*m,1), dY.reshape(m*m,1)), axis=1)
    fTI = zeros([n,n])
    T = 3*sigma
    for i in arange(0,m*m):
        fS = circshift(f,delta[i,:])
        a = perform_wavortho_transf(fS,Jmin,1,h)
        aT = thresh_hard(a,T)
        fS = perform_wavortho_transf(aT,Jmin,-1,h)
        fS = circshift(fS,-delta[i,:])
        fTI = i/(i+1.0)*fTI + 1.0/(i+1)*fS
    snrs.append(snr(f0,fTI))
```

```
snrs = array(snrs)
plot(ms, snrs)
```

Exercise 2:

```
m = 8 # Reasonably good, and we don't want to wait forever...

Tlist = linspace(.8,4.5,25)*sigma
err_soft = zeros([len(Tlist),1])
err_hard = zeros([len(Tlist),1])
for (j,T) in enumerate(Tlist):

    [dY,dX] = meshgrid(arange(0,m),arange(0,m))
    delta = concatenate( (dX.reshape(m*m,1), dY.reshape(m*m,1)), axis=1)

    # Cycle spinning
    fTI_hard = zeros([n,n]) ; fTI_soft = zeros([n,n])
    for i in arange(0,m*m):
        fS = circshift(f,delta[i,:])
        a = perform_wavortho_transf(fS,Jmin,1,h)
        # Hard thresholding :
        aT = thresh_hard(a,T)
        fS = perform_wavortho_transf(aT,Jmin,-1,h)
        fS = circshift(fS,-delta[i,:])
        fTI_hard = i/(i+1.0)*fTI_hard + 1.0/(i+1)*fS
        # Soft thresholding :
        aT = thresh_soft(a,T)
        aT[:2^Jmin:,:2^Jmin:] = a[:2^Jmin:,:2^Jmin:]
        fS = perform_wavortho_transf(aT,Jmin,-1,h)
        fS = circshift(fS,-delta[i,:])
        fTI_soft = i/(i+1.0)*fTI_soft + 1.0/(i+1)*fS
    err_hard[j] = snr(f0,fTI_hard)
    err_soft[j] = snr(f0,fTI_soft)

h1, = plot(Tlist/sigma,err_hard)
h2, = plot(Tlist/sigma,err_soft)
axis('tight')
legend([h1,h2], ['Hard', 'Soft']);
```

Exercise 3: There's a whole numerical tour dedicated to block thresholding! In your `python` folder, it's the file called

`denoisingwav_4_block.ipynb`.

Part II

Inverse problems and sparsity

SESSION 6 Variational methods for inverse problems

Image deconvolution using a Sobolev or TV regularization

Gabriel's numerical tour makes for a perfect workshop session in itself: within Jupyter, open the file called

`inverse_2_deconvolution_variational.ipynb`

and complete Exercises 1 to 4.

To get interaction with your images, and thus become able to zoom in,

simply replace the magic line `%matplotlib inline` with `%matplotlib notebook`.

To get figures in the current cell, you can then add a few `figure();` statements before the `imageplots(...)`.

Make sure that you understand why :

1. The L^2 and Sobolev deconvolutions can be computed using closed-form solutions in the Fourier domain.
2. The use of the squared L^2 distance as a dissimilarity criterion between y and Φf is related to a Gaussian white noise model. (Hint: think about the Maximum A Posteriori statistical framework)
3. The L^2-gradient of the TV functional $\frac{1}{2}\|y - h \star f\|_2^2 + \lambda \sum_x \sqrt{\|\nabla f(x)\|_2^2 + \varepsilon^2}$ is given by

$$\tilde{h} \star (h \star f - y) - \lambda \operatorname{div}\left(\frac{\nabla f}{\sqrt{\|\nabla f\|_2^2 + \varepsilon^2}} \right). \tag{6.1}$$

Solution

All the solutions are located in your

`nt_solutions/inverse_2_deconvolution_variational.ipynb/`

folder.

SESSION 7 Sparse regularization

How should we think about gradient descent?

Most signal processing tasks (denoising, deconvolution, inpainting...) can be expressed as the maximization of a score; or conversely, as the *minimization of an energy*. If X is the space of signals (say, $\mathbb{R}^n$ for n sufficiently large) and d is the data at hand, we're looking for a signal x^* which is optimal in the sense of a rating formula $E_d : X \to \mathbb{R}$, that is,

$$x^* \quad = \quad \arg\min_{x \in X} E_d(x). \tag{7.1}$$

Sometimes, E_d is so simple that finding x^* from the data d is trivial. However, when one digs a little bit deeper, difficult problems arise quickly to the attention of the data scientist. Supervised machine learning is about *learning*, more or less by heart, the decision rule $d \mapsto x^*$. If the amount of data is sufficient, and if implicit (Convolutional Neural Nets, ...) or explicit (kernels, ...) constraints allow us to break the curse of dimensionality for our problem, this is a sensible strategy.

Order 1 minimization schemes In most practical cases, however, bypassing the minimization step is not feasible. It is therefore crucial to know how to find a convincing (local) minimum of a generic functional E. A standard approach is to use *gradient descent*: that is, if E is sufficiently smooth, to iterate the gradient step

$$x_{n+1} \quad \leftarrow \quad x_n \; - \; \tau \nabla E(x_n), \tag{7.2}$$

where τ is a sufficiently small step size. If E is smooth enough, and if τ is small enough, this algorithm is known to converge to a local optimum. But beyond the theorems, how should one think about the gradient descent step?

Directional derivatives: the naive approach In calculus class, we've all learned that, when the signal space X is a vector space $\mathbb{R}^n$,

$$\nabla E(x) \; = \; \begin{pmatrix} \frac{\partial E}{\partial x_1}(x) \\ \vdots \\ \frac{\partial E}{\partial x_n}(x) \end{pmatrix} \; \simeq \; \frac{1}{\delta t} \begin{pmatrix} E(x + \delta t \cdot (1, 0, \ldots, 0)) - E(x) \\ \vdots \\ E(x + \delta t \cdot (0, \ldots, 0, 1)) - E(x) \end{pmatrix}. \tag{7.3}$$

Gradient descent then gets the intuitive explanation of "we decrease the i^{th} coordinate of x if it is correlated with an increase of E". This way of seeing gradient descent is pretty easy to explain to undergrad students, but it has two major flaws:

1. It relies on an arbitrary coordinate system, which may or may not make sense: after all, why should images always be understood as pixel bitmaps?
2. It ingrains in the student's mind the **huge misconception** that "computing a gradient" = "computing n directional derivatives" = "costs about $n + 1$ evaluations of E", in terms of computing time. Just to make it clear: this is **not** true. Gradients are **much cheaper** to compute than you think they are.

Duality maps: paving the way for backpropagation A pretty general definition of the gradient, if E is defined from a source Hilbert space $(X,\langle\cdot,\cdot\rangle_X)$ to a target one, $(Y,\langle\cdot,\cdot\rangle_Y)$, is to see it as the **adjoint of the differential**. That is, the unique linear map $\partial_x E = (\mathrm{d}_x E)^*$ such that for any covector a in Y^*,

$$\langle a, E(x+\delta x) - E(x)\rangle_Y \;=\; \langle a, \mathrm{d}_x E\cdot\delta_x\rangle_Y + o(\delta x) \;=\; \langle \partial_x E\cdot a, \delta x\rangle_X + o(\delta x). \tag{7.4}$$

In the workshop session 12, we will see how this point of view allows one to compute gradients of arbitrary symbolic functionals E, at a cost which is equivalent to that of a handful of evaluations of E... *Whatever the dimension n of the signal space X may be.*

Using $X=\mathbb{R}^n$ and $Y=\mathbb{R}$ endowed with their canonical euclidean metrics allow us to fall back on the previous naive definition: this Hilbertian point of view has made clear the fact that *a gradient is always given with respect to a metric*, as the "usual" gradient is nothing but the canonical "L^2" one.

Proximal operators But, really, if we are striving to minimize a functional E iteratively... Couldn't we try to use a "genuine" local minimization step such as

$$x_{n+1} \;\leftarrow\; \arg\min_{\|t-x_n\|_2^2\leqslant\tau} E(t) \quad ? \tag{7.5}$$

In general, **no**. Most minimization tasks are difficult because of the "unkindness" of the energy formula, not because of the unboundedness of the minimization domain. But if calculus courses have taught us one thing, it is that hard constraints are meant to be softened: we may as well use

$$x_{n+1} \;\leftarrow\; \arg\min_t E(t) + \frac{1}{2\tau}\|t-x_n\|_2^2\,. \tag{7.6}$$

Crucial to most sparsity-enforcing algorithms, this update rule is known as the L^2-Proximal gradient of E at x_n. If E is differentiable at the local optimum $t^*=x_{n+1}$, one gets that

$$x_{n+1} \;\leftarrow\; x_n - \tau\nabla E(x_{n+1}). \tag{7.7}$$

A proximal step is thus nothing but an ideal, implicit gradient descent step. The L^1 norm is then remarkably simple in the following sense: its proximal operator is given by coordinate-wise *soft* thresholding.

Inpainting using a wavelet sparsity prior

Within Jupyter, open the file called

`inverse_5_inpainting_sparsity.ipynb`

and complete Exercises 1 to 5. You should now be able to see that this workshop is nothing but an exercise in energy minimization, using a clever "gradient-like" descent scheme – just like most of the sessions shared with the MVA M2 program, which are related to sparsity and compressed sensing algorithms.

Beware: Due to changes in `python`'s default behavior, you may encounter a few errors when trying to use sliced indexing. As a workaround, in your notebook and in the `nt_toolbox/perform_wavelet_transf.py` file, replace `end/2` with `end//2` at lines 155-156.

Solutions: located as usual in your `nt_solutions/inverse_5_inpainting_sparsity/` folder.

SESSION 8 TV algorithms through convex duality

How do we solve minimization problems?

In the last few sessions, we have put a strong focus on solving *variational* problems that can be written as minimizations of *energies*

$$E_d(x) \;=\; \underbrace{\mathrm{Att}(d,x)}_{\text{Data attachment term}} \;+\; \underbrace{\mathrm{Reg}(x),}_{\text{Regularization term}} \tag{8.1}$$

where d is the data at hand and x is a model to optimize in a signal space X:

$$\text{Sobolev denoising:}\quad E_y^{H^1}(x) \;=\; \frac{1}{2}\|y-x\|_2^2 \;+\; \lambda\cdot\|\nabla x\|_{2,2}^2 \tag{8.2}$$

$$\text{Regularized TV denoising:}\quad E_y^{\varepsilon}(x) \;=\; \frac{1}{2}\|y-x\|_2^2 \;+\; \lambda\cdot\left\|\sqrt{\|\nabla x\|_2^2+\varepsilon^2}\right\|_1 \tag{8.3}$$

$$\text{Wavelet inpainting:}\quad E_y^{\Psi}(x) \;=\; \frac{1}{2}\|y-\Phi\Psi\, x\|_2^2 + \lambda\cdot\|x\|_1 \tag{8.4}$$

$$\text{TV denoising:}\quad E_y^{\mathrm{TV}}(x) \;=\; \frac{1}{2}\|y-x\|_2^2 \;+\; \lambda\cdot\|\nabla x\|_{1,2} \tag{8.5}$$

Gradient descent. As we have seen already, Sobolev denoising can be solved analytically in the Fourier domain. Regularized TV denoising is tackled using L^2-gradient descent, and the sparsity inducing L^1-prior of the wavelet inpainting algorithm is simple enough to be handled by *proximal* (i.e. implicit) gradient descent. But how can we solve the TV denoising problem, which relies on a regularizer that is neither differentiable nor simple?

Fenchel to the rescue. Thankfully, the energy E_y^{TV} is **convex** and somewhat related to well-known algebraic functions. The theory of **convex duality** will thus allow us to replace the *primal* problem of "minimizing E_d over the vector space X" by an equivalent *dual* problem over the space X^*. In this precise case, calculations will pan out very well as the differentiation operator ∇ goes "from the regularizer to the data attachment term": we will end up with a dual energy that is easy to minimize using the proximal gradient technique.

Understanding convex analysis. Now, where does this "duality" trick comes from? As a complete reference on the subject, I would strongly recommend the lectures notes of Anne Sabourin and Pascal Bianchi on *Convex Analysis*, available at the following address:

`www.lix.polytechnique.fr/bigdata/mathbigdata/wp-content/uploads/2014/10/Lnotes_CvxAn_FullEn.pdf`.

They go straight to the point, and are perfectly suited to the background and mindset of a French graduate student. Keep in mind though, that the scope of this *Data Science* class is pretty wide and we do not expect students to assimilate whole new theories every week... In today's session, we will thus focus on understanding **why the formulas written in the notebook are essentially correct**, without trying to prove them in the most general cases.

Forward-Backward method on the dual problem

Within Jupyter, open and go through the notebook called

`optim_3_condat_fb_duality.ipynb`.

To make sure that you understand what is going on, please have a look at the following exercises:

Exercise 1: Show that $f : \mathbb{R}^d \to \overline{\mathbb{R}}$ is convex, proper and lower semi-continuous if and only if its epigraph is a closed, convex, nonempty domain $\mathrm{epi}(f)$ of $\mathbb{R}^{d+1}$ such that

$$\forall\ x \in \mathbb{R}^d, \exists\ y \in \mathbb{R}, (x, y) \notin \mathrm{epi}(f). \tag{8.6}$$

Show that this is equivalent to saying that $\mathrm{epi}(f)$ is the intersection of the half-spaces defined by its supporting hyperplanes.

Exercise 2: If $f : \mathbb{R}^d \to \overline{\mathbb{R}}$ is proper and convex, we define

$$f^* :\ u \in (\mathbb{R}^d)^*\ \mapsto\ \sup_{x \in \mathbb{R}^d} \langle u, x\rangle - f(x). \tag{8.7}$$

How would you "compute" graphically f^*, the Fenchel transform or *dual* of f? Show that

$$\forall\ u \in (\mathbb{R}^d)^*, \forall\ x \in \mathbb{R}^d, \langle u, x\rangle\ \leqslant\ f^*(u) + f(x), \qquad \text{(Young inequality)} \tag{8.8}$$

with equality if and only if $u \in \partial f(x)$. How do you interpret f^* in terms of supporting hyperplanes? Show that f^* is always l.s.c. and convex. We will now identify $(\mathbb{R}^d)^*$ with $\mathbb{R}^d$ through the canonical Riesz isomorphism.

Exercise 3: A few computations:

1. Let $f : x \mapsto \frac{1}{2}\|Ax - b\|_2^2$ with $A \in \mathbb{R}^{N \times N}$ an invertible matrix. Show that

$$f^*(u)\ =\ \frac{1}{2}\big\|(A^*)^{-1}u + b\big\|_2^2\ -\ \frac{1}{2}\|b\|_2^2. \tag{8.9}$$

   (Hint: remember that $u \in \partial f(x)$ iff $f^*(u) = \langle u, x\rangle - f(x)$.)

2. Let $f : x \mapsto \|x\|$ be a norm on $\mathbb{R}^d$; as usual, it defines a dual norm $\|\cdot\|_*$ on $(\mathbb{R}^d)^*$. Show that the convex dual f^* of f is in fact the indicator of the dual unit ball

$$f^* = \iota_{\|\cdot\|_* \leqslant 1} : u \mapsto \begin{cases} 0 & \text{if } \|u\|_* \leqslant 1 \\ +\infty & \text{otherwise} \end{cases}. \tag{8.10}$$

   What is the proximal operator of f^*?

Exercise 4: Fenchel-Rockafellar theorem. Let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ be two smooth and strictly convex functions which go to $+\infty$ when $|x| \to \infty$. Note that $f + g$ can be understood as the difference between f and $h = -g$, which is concave. Show that

$$\min_{x \in \mathbb{R}} f(x) + g(x)\ =\ \min_{x \in \mathbb{R}} f(x) - h(x)\ =\ -\min_{u \in \mathbb{R}^*} f^*(u) - h^*(u)\ =\ -\min_{u \in \mathbb{R}^*} f^*(u) + g^*(-u), \tag{8.11}$$

and that there exists a point x^* and a slope u^* realizing the shared optimum such that

$$f'(x^*)\ =\ u^*\ =\ h'(x^*). \tag{8.12}$$

(Hint: Show that the left-hand minimum is necessarily reached at a point x where $f'(x) = h'(x)$. Then, show that the right-hand minimum is necessarily reached for a slope u such that there exists $x \in \mathbb{R}$, $u = f'(x) = h'(x)$.)

Remark that if L is an invertible linear operator, then $(g \circ L)^*(u) = g^*((L^*)^{-1}u)$: the Fenchel-Rockafellar formula should now make sense.

Solution

Exercise 1: The fact that

$$f \text{ is proper} \iff \text{epi}(f) \text{ is not empty, and } \forall\, x \in \mathbb{R}^d, \exists\, y \in \mathbb{R}, (x,y) \notin \text{epi}(f), \tag{8.13}$$

$$f \text{ is convex} \iff \text{epi}(f) \text{ is convex}, \tag{8.14}$$

is a direct consequence of the definitions. Let's show that

$$f \text{ is l.s.c.} \iff \text{epi}(f) \text{ is closed.} \tag{8.15}$$

Direct implication: Assume that f is l.s.c., and take a sequence $\big((x_n, y_n)\big)_{n\in\mathbb{N}}$ of points in $\text{epi}(f)$ which converges towards $(x_\infty, y_\infty) \in \mathbb{R}^d \times \mathbb{R}$. Then, notice that

$$\forall\, n \in \mathbb{N},\ \ y_n \geqslant f(x_n) \qquad \text{and} \qquad y_n \longrightarrow y_\infty,\ x_n \longrightarrow x_\infty. \tag{8.16}$$

Hence, as f is l.s.c.,

$$y_\infty \geqslant \liminf y_n \geqslant \liminf f(x_n) \geqslant f(x_\infty), \quad \text{so that } (x_\infty, y_\infty) \in \text{epi}(f). \tag{8.17}$$

The epigraph of f is thus a closed set.

Conversely: Assume that f is *not* l.s.c.: There exists $x_n \to x_\infty$ such that $\liminf f(x_n) < f(x_\infty)$. If we extract a subsequence $x_{\sigma(n)}$ such that $f(x_{\sigma(n)}) \to \liminf f(x_n)$, we can then remark that $(x_{\sigma(n), f(x_{\sigma(n)})})$ is a sequence of $\text{epi}(f)$ which converges towards a point $(x_\infty, \liminf f(x_n))$ which is not in $\text{epi}(f)$.

Relationship with supporting hyperplanes: For any "slope" covector $u \in (\mathbb{R}^d)^*$, we define

$$b_u = \max\big\{b \in \mathbb{R},\ \ \forall\, x \in \mathbb{R}^d,\ f(x) \geqslant \langle u, x\rangle + b\big\} \tag{8.18}$$

$$= \sup\big\{f(x) - \langle u, x\rangle,\ \ x \in \mathbb{R}^d\big\}. \tag{8.19}$$

The supporting hyperplane of f with slope u is no one but the graph of the function $x \mapsto \langle u, x\rangle + b_u$ which defines a half-space

$$H_u = \big\{(x,y) \in \mathbb{R}^d \times \mathbb{R},\ \ y \geqslant \langle u, x\rangle + b_u\big\}. \tag{8.20}$$

According to the definition of b_u, we know that

$$\text{epi}(f) \subset \bigcap_{u\in(\mathbb{R}^d)^*} H_u. \tag{8.21}$$

Conversely, assume that $(x,y) \notin \text{epi}(f)$, i.e. $y < f(x)$. Since $\text{epi}(f)$ is convex, closed and non-empty, we know that there exists a unique **orthogonal projection** (x_p, y_p) of (x,y) onto epi](f) – the unique minimizer of the strictly convex function $(s,t) \mapsto \|(s,t) - (x,y)\|^2$ on the set $\text{epi}(f)$. Since the latter is **closed**, we also know that $(x_p, y_p) \neq (x,y)$.

Then, if we note $(v,w) = (x_p - x, y_p - y)$, the convexity of $\text{epi}(f)$ ensures that

$$\left\{(s,t) \in \mathbb{R}^d \times \mathbb{R},\ \langle (v,w),\, (s,t)\rangle = \underbrace{\big\langle (v,w),\, \tfrac{1}{2}(x+a, y+b)\big\rangle}_{c}\right\} \tag{8.22}$$

$$= \left\{(s,t) \in \mathbb{R}^d \times \mathbb{R},\quad w\,t = c - \langle v, s\rangle\right\} \tag{8.23}$$

is a **strictly separating hyperplane** between $\text{epi}(f)$ and (x,y).

Then, there are two cases. If $w \neq 0$, we can go and conclude that since

$$\left\{(x,y) \in \mathbb{R}^d \times \mathbb{R}, \quad y \geqslant \left\langle -\frac{v}{w}, x \right\rangle + \frac{c}{w}\right\} \tag{8.24}$$

separates $\mathrm{epi}(f)$ and (x,y), so must the optimal hyperplane $H_{-v/w}$. Hence, $(x,y) \notin \bigcap H_u$.

Otherwise, if $w = 0$, this means that the separating hyperplane defined eq. (8.23) is **vertical**, with $\mathrm{epi}(f)$ purely on one side of the separation. Since $\mathrm{epi}(f)$ is convex and points upwards, we can then use eq. (8.6) to find a "non-vertical" separating slope, which can be encoded as a supporting half-space H_u, as we link up with the previous argument.

N.B.: This result is nothing but the classical characterization of closed convex sets in Banach spaces (consequence of the Hahn-Banach theorem), in the special setting of **optimization** where there is one privileged "vertical" direction. This anisotropy burdens a bit the proofs ($w \neq 0$, etc...): that's life.

Exercise 2: Surprise surprise, the "optimal" offset coefficient b_u defined in the correction of Exercise 1 actually's got a name: it is no one but $-f^*(u)$. Here is the sketch you should have in mind when thinking about the Fenchel transform:

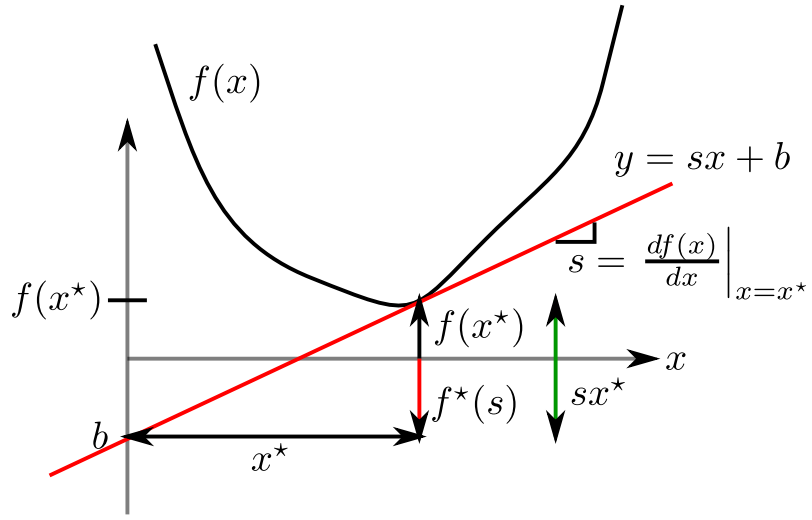


Figure 8.1: Graphical computation of the Fenchel transform $f^*(s)$ – with $u = s$, here. Image taken from the very nice website `www.onmyphd.com/?p=legendre.fenchel.transform`.

The Young inequality is a direct consequence of the definition of the Fenchel transform as a supremum; then, we know that for all $u \in (\mathbb{R}^d)^*$, $x \in \mathbb{R}^d$, we have

$$\langle u, x\rangle - f(x) = f^*(u) \iff \forall\, y \in \mathbb{R}^d,\ \langle u, x\rangle - f(x) \geqslant \langle u, y\rangle - f(y) \tag{8.25}$$

$$\iff \forall\, y \in \mathbb{R}^d,\ f(y) \geqslant f(x) + \langle u, y - x\rangle \tag{8.26}$$

$$\iff u \text{ is a subdifferential of } f \text{ at point } x. \tag{8.27}$$

Furthermore, as a supremum of l.s.c. and convex functions of u (the linear forms $u \mapsto \langle u, x\rangle - f(x)$), f^* is also l.s.c. and convex: with respect to epigraphs, **the supremum acts like the intersection of sets** and closedness, convexity are two properties which are stable by this operator.

Exercise 3: For $x_0 \in \mathbb{R}^d$, the gradient of the quadratic function $f : x \mapsto \frac{1}{2}\|Ax - b\|^2$ is given by

$$\partial_x f(x_0) \;=\; A^\star\,(Ax_0 - b) \;=\; A^\star A x_0 - A^\star b. \tag{8.28}$$

Since $\partial_x f$ is an invertible mapping from $\mathbb{R}^N$ to $\mathbb{R}^N$, the convex dual of f is easy to compute using the equality case in Young's inequality: if $u \in (\mathbb{R}^d)^*$ is written as $A^\star\,(Ax - b)$, we know that

$$\begin{aligned}
f^*(u) \;&=\; \langle u, x\rangle &&- \tfrac{1}{2}\|Ax - b\|_2^2 && (8.29)\\
&=\; \left\langle u, A^{-1}((A^*)^{-1}u + b)\right\rangle &&- \tfrac{1}{2}\left\|(A^*)^{-1}u\right\|_2^2 && (8.30)\\
&=\; \left\langle (A^*)^{-1}u, (A^*)^{-1}u + b\right\rangle &&- \tfrac{1}{2}\left\|(A^*)^{-1}u\right\|_2^2 && (8.31)\\
&=\; \tfrac{1}{2}\left\|(A^*)^{-1}u + b\right\|_2^2 &&- \tfrac{1}{2}\|b\|_2^2\,. && (8.32)
\end{aligned}$$

Question 2: The norm's convex dual is defined as the supremum

$$f^*(u) \;=\; \sup_{x\in\mathbb{R}^d} \langle u, x\rangle \;- \|x\|\,. \tag{8.33}$$

Note that we do not ask for the norm here to be the L^2 norm associated with the canonical scalar product. To compute $f^*(u)$, we must distinguish two cases:

1. If $\|u\|_* \leqslant 1$, we know that the value of $f^*(u)$ is bounded above by zero.
   Since $\langle u, 0\rangle - \|0\| = 0$, this maximal value is reached and $f^*(u) = 0$.
2. If $\|u\|_* > 1$, then by definition of the supremum, there exists a vector x_0 such that $\langle u, x_0\rangle > \|x_0\|$.
   Taking $x = \lambda\, x_0$ with $\lambda \to +\infty$ in the conjugate's definition, we show that $f^*(u) = +\infty$.

The convex dual of f is thus no one but the indicator $\iota_{\|\cdot\|_* \leqslant 1}$ of the dual unit ball associated to f. Since

$$\operatorname{prox}_{f^*}(u) \;=\; \arg\min_{v\in(\mathbb{R}^d)^*} \iota_{\|\cdot\|_*\leqslant 1}(v) + \tfrac{1}{2}\|u - v\|_2^2 \;=\; \arg\min_{\|v\|_*\leqslant 1} \|u - v\|_2^2\,, \tag{8.34}$$

we see that the proximal operator of f^* (which is always defined with respect to a descent metric, just like a gradient – here, the L^2 norm) is the **orthogonal projector onto the unit ball associated to the dual norm $\|\cdot\|_*$ of $f = \|\cdot\|$.**

Exercise 4: Since $f + g$ is a strictly convex function defined on the convex set $\mathbb{R}$, it reaches its minimum at exactly one point $x^* \in \mathbb{R}$. Then, since $f + g$ is smooth, we know that

$$f'(x^*) \;+\; g'(x^*) \;=\; 0 \qquad\qquad \text{i.e.} \qquad\qquad f'(x^*) \;=\; h'(x^*). \tag{8.35}$$

We must now show that this shared value of the derivative, that we denote u^*, is precisely the solution of the dual optimization problem $\min_{u\in\mathbb{R}^*} f^*(u) - h^*(u)$. **To keep track of all the variables involved, please make a sketch with, say, two parabolas f and h!**

Since f and h are both smooth, strictly convex/concave and reach their respective minimum/maximum, their derivatives f' and h' are increasing/decreasing functions from $\mathbb{R}$ to open

intervals I_f and I_h with a non empty intersection (containing, at the very least, a neighborhood of the origin 0). We can thus define the inverse bijections

$$(f')^{-1} : I_f \to \mathbb{R} \qquad \text{and} \qquad (h')^{-1} : I_h \to \mathbb{R} \tag{8.36}$$

which are respectively increasing and decreasing functions of the dual variable u. Then, according to Exercise 2, we know that

$$f^* \;=\; f \circ (f')^{-1} \qquad \text{and} \qquad h^* \;=\; h \circ (h')^{-1}, \tag{8.37}$$

so that

$$f^* - h^* : u \mapsto \begin{cases} f \circ (f')^{-1}(u) + h \circ (h')^{-1}(u) & \text{if } u \in I_f \cap I_h \\ +\infty & \text{otherwise} \end{cases}. \tag{8.38}$$

Remark that $u^* \in I_f \cap I_h$.
Let's take $u \in I_f \cap I_h$ which is not equal to u^*, and let's show that

$$(f^* - h^*)(u) \;>\; (f^* - h^*)(u^*). \tag{8.39}$$

We denote $x_1 = (f')^{-1}(u)$ and $x_2 = (h')^{-1}(u)$ the abscissa involved in the computation of $(f^* - h^*)(u)$; since $(f')^{-1}$ and $(h')^{-1}$ are increasing/decreasing, u^* is the only point such that "$x_1 = x_2$". Hence, since $u \neq u^*$, we know that $x_1 \neq x_2$.

According to Exercise 2, we can compute

$$f^*(u) \;=\; \langle u, x_1 \rangle - f(x_1) \qquad \text{and} \qquad h^*(u) \;=\; \langle u, x_2 \rangle - h(x_2). \tag{8.40}$$

That is, as we know that $u = f'(x_1) = h'(x_2)$,

$$(f^* - h^*)(u) \;=\; h(x_2) - f(x_1) \;-\; \langle u, x_2 - x_1 \rangle \tag{8.41}$$

$$\text{and} \quad (f^* - h^*)(u^*) \;=\; h(x^*) - f(x^*) \;+\; \langle u^*, x^* - x^* \rangle\,. \tag{8.42}$$

Therefore, we have

$$(f^* - h^*)(u) - (f^* - h^*)(u^*) \;=\; f(x^*) - f(x_1) + h(x_2) - h(x^*) + \langle u, x_1 - x_2 \rangle \tag{8.43}$$

$$= \underbrace{f(x^*) - f(x_1) + \langle u, x_1 - x^* \rangle}_{> 0 \text{ since } f \text{ is stricly convex}} + \underbrace{h(x_2) - h(x^*) + \langle u, x^* - x_2 \rangle}_{> 0 \text{ since } h \text{ is stricly concave}}. \tag{8.44}$$

Hence why

$$(f^* - h^*)(u) \;>\; (f^* - h^*)(u^*). \tag{8.45}$$

Part III

Machine-learning and optimal transport

SESSION 9 Introduction to machine learning

How do we train algorithms?

Vocabulary. We consider the general setting of **supervised learning:** given a set of observations (x_i, y_i), we would like to learn a model f_w, parametrized by a vector w, such that $f_w(x_i) \approx y_i$: the x_i's are the *data points*, and the y_i's are the *labels*. If the y_i's were continuous function values, we would be tackling a **regression** problem. In this workshop session, however, we're interested in a *discrete* **classification** problem where the y_i's are labels living in an arbitrary set {blue, yellow, red} identified for mathematical convenience with the integer set $\{1, 2, 3\}$ - or $\{0, 1, 2\}$ to fit `python`'s indexing conventions.

Notations. In the equations written below, $x_i \in \mathbb{R}^2$ and $y_i \in \{1, \dots, K\} = \{1, 2, 3\}$ will denote generic data points and labels; x and y will be generic variables of deterministic functions, whereas the capital letters X and Y denote the observed training samples $(x_i)_{1 \leqslant i \leqslant I}$ and $(y_i)_{1 \leqslant i \leqslant I}$.

A simple probabilistic setting. In this workshop, we will assume that **the x_i's are independent** identically distributed variables, **with y_i following a law g_{x_i} that we wish to regress**. At the very least, we will strive to compute the maximum likelihood estimator

$$\arg\max_y g_{x_i}(y), \qquad \text{approximated by the parametric function} \qquad f_w(x_i) \tag{9.1}$$

and then assume a "signal+noise" generative process: $y_i \sim g_{x_i} \simeq \mathcal{N}(f_w(x_i), \sigma^2)$ for instance.

A deterministic optimization problem. We propose to learn the parameters w of the model through the **minimization of a loss function** of the form

$$\text{Cost}_{X,Y}(w) \;=\; \sum_{i=1}^{I} \text{Att}(f_w(x_i), y_i) + \text{Reg}(w). \tag{9.2}$$

The optimal set of parameters minimizes the sum of a **data attachment term** $\sum_i \text{Att}(f_w(x_i), y_i)$ and of an **explicit regularization term** $\text{Reg}(w)$. Choosing relevant formulas for "Att" and "Reg" is a critical modelling step: this introductory session is dedicated to the review of a handful of popular strategies on a toy dataset. Remark that our problem is equivalent to the maximization of

$$\exp\big(-\text{Cost}_{X,Y}(w)\big) \;=\; \exp\big(-\text{Reg}(w)\big) \cdot \prod_i \exp\big(-\text{Att}(f_w(x_i), y_i)\big) \tag{9.3}$$

which can be written, *up to a multiplicative renormalization constant*, as a conditional likelihood

$$\mathbb{P}(w, Y \mid X) \;=\; \mathbb{P}(w) \;\cdot\; \prod_i \mathbb{P}(y_i \mid w, x_i). \tag{9.4}$$

Quadratic costs and Gaussian models. If both Reg and Att are **quadratic** functions, minimizing Cost is therefore akin to computing a maximum a posteriori estimation of w for a **Gaussian generative model**, which explains the empirical data distribution as follows:

- The input data X is a deterministic input.
- The model's parameters w are drawn independently of X, according to a fixed Gaussian distribution.
- The "clean" values $f_w(x_i)$ are computed **independently from each other**, before corruption by an additive Gaussian noise results in the observed values y_i.

A simplistic linear regression

The simplest model of them all is to view y_i as a real-valued function of x_i, and to fit a linear model of parameters $W \in \mathbb{R}^2, b \in \mathbb{R}$ by using

$$f_{W,b}(x) \;=\; \langle W, x \rangle + b \tag{9.5}$$

with no regularization and an L^2 attachment term. That is, we minimize

$$\text{Cost}_{X,Y}(W,b) \;=\; \frac{1}{2I} \sum_i \|f_{W,b}(x_i) - y_i\|^2. \tag{9.6}$$

This minimization problem can be solved explicitely using the pseudoinverse... But for the sake of pedagogy, note that we could also use gradient descent on the vector $w = (W[1], W[2], b)$! As displayed in Figure 9.1, the results of this classification method are pretty poor. This is not surprising: as our regression model treats the class labelling $x \mapsto y(x)$ as a real-valued function, it induces an unbearable bias with respect to the classes' ordering.

Training a softmax linear classifier

Embedding linear scores in a space of probability measures. To alleviate this problem, we propose to use a softmax linear classifier that allows us to handle **labels** as discrete variables. We still compute linear scores, but now interpret them as the unnormalized log-probabilities of each class, **without relying on any arbitrary ordering of the labels**.

If K is the number of class, the trainable parameters of our model are a weight matrix W of size $K \times D$ (in the math convention, where vectors are columns... and not lines, as in `python`), and an offset vector b of size K. The linear scores $s \in \mathbb{R}^K$ at a generic location $x \in \mathbb{R}^D$ are computed as

$$s \;=\; Wx + b \tag{9.7}$$

and we decide to **interpret those generic "class scores" as un-normalized log probabilities**. That is, given $s = (s[1], \dots, s[K])$ a given score vector, we assume that

$$P_s[k] \;=\; \mathbb{P}(\text{class} = k \mid s) \;=\; \frac{\exp(s[k])}{\sum_{j=1}^K \exp(s[j])}. \tag{9.8}$$

The equation above allows us to interpret $f_w(x_i)$ as a probability measure on the discrete space of labels $\{1, 2, \dots, K\}$. The exponential in the formula is chosen for convenience, as it allows us to represent a wide range of positive probabilities with scores that are relatively close to each other.

This trick allows us to pass from scalar-valued scores to *probabilities* on the set of labels. It is clever, as we can now use standard pseudo-distances between probability measures, which do not rely on any parametrization of the set of labels.

Using a standard (pseudo-)metric structure on the space of probability measures. Now, what kind of data attachment term $\text{Att}(f_w(x_i), y_i)$ could we use to compare the "generated" $f_w(x_i) \simeq P_{Wx_i+b}$ with the "deterministic" empiric law y_i, identified with the measure δ_{y_i} such that

$$\mathbb{P}_{\delta_{y_i}}(\text{class} = k) = \begin{cases} 1 & \text{if } k = y_i \\ 0 & \text{otherwise} \end{cases} \quad ? \tag{9.9}$$

Information Theory to the rescue. Since $\{1, 2, \ldots, K\}$ is a label space deprived of any relevant metric structure, a good choice is the cross-entropy or **Kullback-Leibler divergence**

$$\mathrm{KL}(\delta_y \,||\, P_s) \;=\; \int \log\left(\frac{\mathrm{d}\delta_y}{\mathrm{d}P_s}\right) \mathrm{d}\delta_y \;=\; -\log(P_s[y]) \;=\; -\log\left(\frac{\exp(s[y])}{\sum_{j=1}^K \exp(s[j])}\right) \tag{9.10}$$

since δ_y is completely localized on y. Remember from the course on information theory and entropic coding that the KL-divergence between two **probability** measures μ and ν is given by

$$\mathrm{KL}(\mu \,||\, \nu) \;=\; \text{«}\; \underbrace{\int \log\left(\frac{1}{\mathrm{d}\nu}\right) \mathrm{d}\mu}_{\text{Perf. of a } \nu\text{-code on } \mu} \;-\; \underbrace{\int \log\left(\frac{1}{\mathrm{d}\mu}\right) \mathrm{d}\mu}_{\text{Perf. of an optimal } \mu\text{-code on } \mu} \;\text{»} \;=\; \int \log\left(\frac{\mathrm{d}\mu}{\mathrm{d}\nu}\right) \mathrm{d}\mu \tag{9.11}$$

and measures "how well a ν-code can be used to encode μ". It induces a data attachment formula which is not symmetric with respect to the variables δ_y and P_s, but is **nonnegative**, **differentiable**, and **null iff** $P_s = \delta_y$. On top of this, we propose to use an L^2 quadratic regularization on W: denoting $s_i = Wx_i + b$, the final objective function can thus be written as

$$\mathrm{Cost}_{X,Y}(W, b) \;=\; -\frac{1}{I}\sum_i \log\left(\frac{\exp(s_i[y_i])}{\sum_j \exp(s_i[j])}\right) + \frac{\varepsilon}{2}\,\mathrm{trace}(W^T W). \tag{9.12}$$

Remember: The method presented above, kown as **(Linear) Logistic regression** (because it is linear regression on the log-probabilities), is the baseline tool for **classification**. It treats class values as *labels*, and does not rely on any (arbitrary) embedding of the set of labels in a metric space.

Training a linear SoftMargin-SVM

Another popular strategy: using a Support Vector Machine loss. In its simplest form, the mechanism that computes the scores vector s is linear, just as in the softmax classifier above:

$$s \;=\; Wx + b. \tag{9.13}$$

However, instead of using a probabilistic interpretation, we focus on **geometry** and interpret the collection of argmax level sets in $\mathbb{R}^2$

$$\{s[1] \geqslant s[2] \text{ and } s[3]\}, \quad \{s[2] \geqslant s[1] \text{ and } s[3]\}, \quad \{s[3] \geqslant s[1] \text{ and } s[2]\} \tag{9.14}$$

as **a partition whose margins should separate the point clouds as well as possible**. More precisely, we wish to minimize

$$\mathrm{Cost}_{X,Y}(W, b) \;=\; \sum_{i=1}^{I}\sum_{k=1}^{K} 1_{k\neq y_i} \cdot \big(s_i[k] - (s_i[y_i] - \Delta)\big)^+ + \frac{\varepsilon}{2}\,\mathrm{trace}(W^T W) \tag{9.15}$$

where $s_i = Wx_i + b$, and where $z^+ = \max(0, z)$ denotes the positive part of a real number z. This formula penalizes W and b if they assign to "wrong" classes $k \neq y_i$ a score $s_i[k]$ which is nearly or clearly superior to that of the "correct" class $s_i[y_i]$, with a detection margin parameter Δ.

Training SVMs with different kernels

Introducing nonlinear features. Both methods presented above look pretty basic, as they rely on **linear scores**. Thankfully, we can tweak those linear algorithms to tackle nonlinear problems using the **kernel trick**. The idea is to embed the data in a higher dimensional feature space, where a linear classifier may achieve better performance. Surprisingly, there exists an efficient way to work **implicitly** with high-dimensional embeddings. Indeed, since the classification algorithms presented above are **purely geometric**, they can be expressed in terms of the **Gram matrix** of the point cloud

$$G_{i,j} \;=\; \langle x_i, y_j \rangle. \tag{9.16}$$

Consequently, to work with an embedding φ, there's no need to compute the high-dimensional vectors $\varphi(x)$: having access to the **Kernel matrix**

$$K_{i,j} \;=\; \langle \varphi(x_i), \varphi(y_j) \rangle \;=\; k(x_i, y_j) \tag{9.17}$$

is enough. Then, **Mercer's theorem** gives necessary and sufficient conditions on the *kernel function* k to be able to interpret the Kernel matrix as a "mapped" Gram matrix. In the simple case of translation invariant kernels $k(x_i, y_j) = k(x_i - y_j)$, these conditions read as

$$\forall\, \omega \in \mathbb{R}^D, \quad \widehat{k}(\omega) \;>\; 0, \tag{9.18}$$

that is, "the Fourier transform of k should be real and positive". **Taking all of this into account, a standard approach to classification is to use SVM or logistic regression (implemented in terms of the Gram matrix) with a custom *kernel* chosen with respect to the data.**

Remember: As shown in Figure 9.1, the RBF (Gaussian) kernel SVM pretty much solves toy examples for a wide range of values of the parameters. Being a robust, well-understood and efficient tool, it is now a standard baseline in the machine learning industry: a popular "next step" after linear methods.

Training a 2-layer perceptron

The Neural Networks fantasy. In the previous sections, we've seen how to train probabilistic models and soft-margin classifiers. A third popular strategy is to try to learn the optimal program for a given classification task. Instead of optimizing in, say, the well-delimited space of decision hyperplanes, **the ambition is to search our decision rule $f_w : x_i \mapsto f_w(x_i) \simeq y_i$ in a much wider (and *wilder*) space of computer programs**.

Making a naive **tree search** in the space of Turing machines / λ-expressions is obviously intractable: it is a **combinatorial** problem with crazy high complexity - without even speaking of the classical CS paradoxes! In order to link up with efficient optimization routines, it is thus necessary to restrict ourselves to **continuously parametrized** and finite operations: given elementary computational bricks

$$F^i_{w_i} : \mathbb{R}^{N_{i-1}} \mapsto \mathbb{R}^{N_i} \tag{9.19}$$

and a fixed **program depth** D, we'll look for the optimal parameters $w = (w_1, \dots, w_D)$ of a classification program encoded as a composition

$$f_w \;=\; F^D_{w_D} \circ \cdots \circ F^2_{w_2} \circ F^1_{w_1}. \tag{9.20}$$

Which kind of elementary block should we use ? From a computational point of view, the simplest choice would be to use linear operators (**affine**, really). That is, to define $w_i = (W_i, b_i) \in \mathbb{R}^{N_i \times N_{i-1}} \times \mathbb{R}^{N_i}$ and use

$$F^i_{w_i} : x \mapsto W_i x + b_i. \tag{9.21}$$

Introducing non-linearities. Problem is: the composition of two affine functions is still an affine function... If we concatenate linear operators, increasing the program depth D doesn't help us to **explore a wider set of programs**. In order to build richer models, we thus have to "break" the linearity of our atomic operators, and use

$$F^i_{w_i} : x \mapsto \sigma(W_i x + b_i), \tag{9.22}$$

where σ is a pointwise non-linear function. A popular choice is to use the cheap **REctified Linear Unit**, or positive part

$$\sigma : x \mapsto x^+ = \max(0, x). \tag{9.23}$$

The elementary operators $F^i_{W_i,b_i}$ defined above are both generic and easy to work with. We haven't made any sparsity assumption on the matrix W_i: *a priori*, the output coordinates depend of *all* the input coordinates. In the literature, these elementary bricks are thus known as **fully connected** layers.

Putting all of this into practice. The next workshop session will be dedicated to this "algorithmic" stance on machine learning: we will investigate its potential for image processing applications alongside its limits. For today, let's just see "how it's done" on a simplistic 2D-dataset. We implement a **gradient descent optimization** on the parameters of a 2-layer perceptron, that is, of a non-linear operator

$$f_w \;=\; F^2_{W_2,b_2} \circ F^1_{W_1,b_1} \tag{9.24}$$

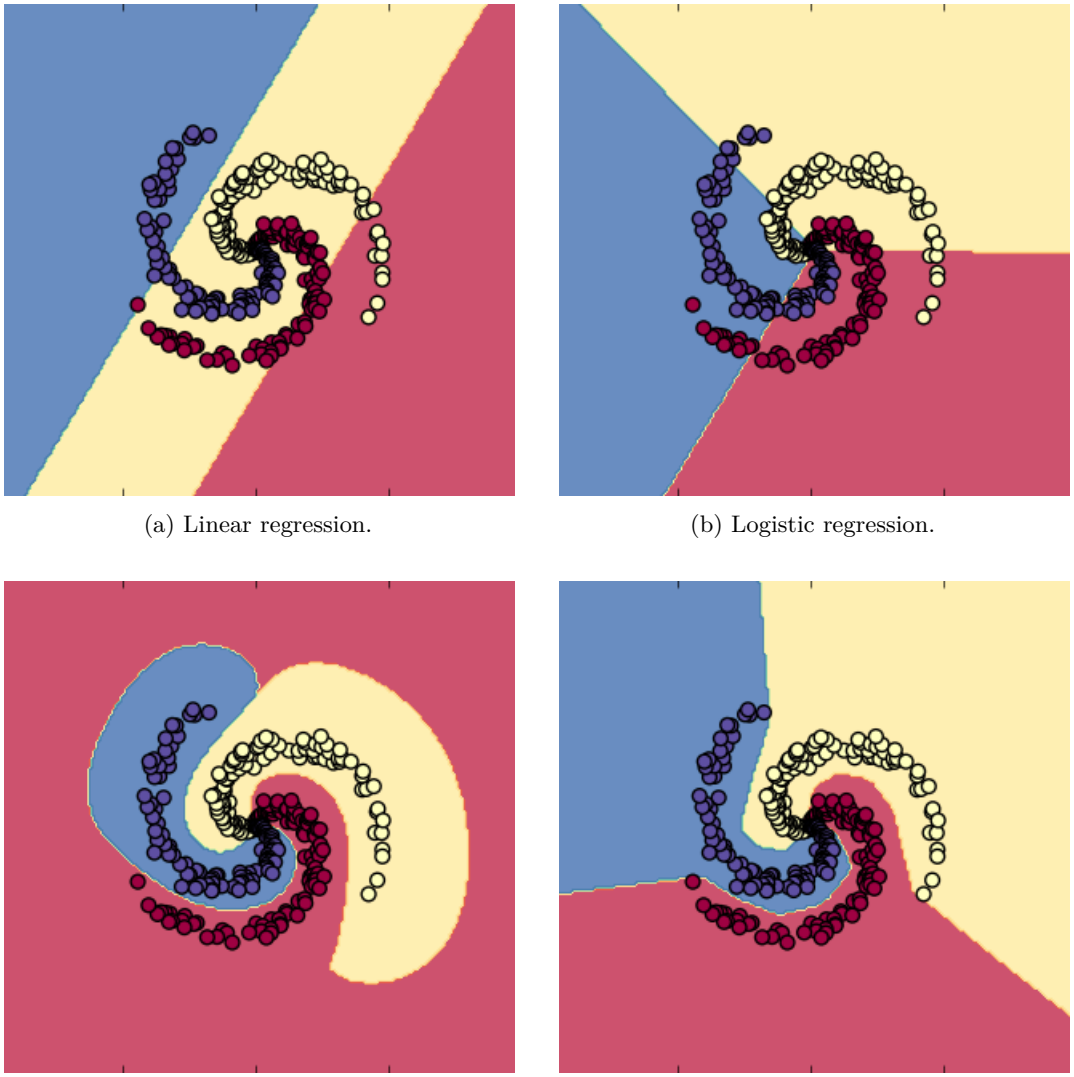
where

$$F^1_{W_1,b_1} : x \in \mathbb{R}^2 \mapsto (W_1 x + b_1)^+ \in \mathbb{R}^H \quad \text{and} \quad F^2_{W_2,b_2} : x \in \mathbb{R}^H \mapsto (W_2 x + b_2) \in \mathbb{R}^K. \tag{9.25}$$

Non-linear generation of probabilistic scores. Here, $H = 100$ is the number of "hidden units" in the intermediate vector space $\mathbb{R}^{N_1}$, and $K = 3$ is the number of classes. Since $f_w(x_i) \in \mathbb{R}^K$ can be interpreted as a *scores vector*, we'll use the "softmax+KL" discrepancy discussed in the section on Logistic regression, and interpret the output of our model as a vector of *un-normalized log-probabilities*. All in all, denoting $s_i = f_w(x_i)$, we strive to minimize

$$\mathrm{Cost}_{X,Y}(W_1, b_1, W_2, b_2) \;=\; -\frac{1}{I}\sum_i \log\left(\frac{\exp(s_i[y_i])}{\sum_j \exp(s_i[j])}\right) + \frac{\varepsilon}{2}\,\mathrm{trace}(W_1^T W_1 + W_2^T W_2); \tag{9.26}$$

A marketable theory. Since the 60's (one layer) and 80's (multilayer), these algorithmic models are known as **multilayer perceptrons**. A name which is not very relatable... As these stacked data flows are (very) loosely related to structures present in the cortex of mammals, tunable programs $f_w = F^D \circ \cdots \circ F^1$ are now colloquially referred to as **neural networks**. In the next workshop session, we will present their extensions to higher dimensional problems (esp. image classification), recently marketed as **deep learning** in the mass media.



(a) Linear regression.

(b) Logistic regression.

(c) Kernel-SVM (RBF kernel).

(d) 2-layer perceptron.

Figure 9.1: Eventual decision rules presented in the notebook. The synthetic dataset (made out of three classes Blue-Yellow-Red) is taken from Andrej Karpathy's CS231n class, `cs231n.github.io/neural-networks-case-study/`, alongside the examples (b) and (d).

(a) Straightforward **linear regression** performs extremely poorly, as it handles the labels 0 (blue), 1 (yellow) and 2 (red) as values of a linear function to regress.
(b) Logistic regression embeds (linear) features into a space of probability measures, before using the Kullback-Leibler divergence as a standard data attachment formula. The eventual decision rule is therefore **invariant wrt. the set of labels**, and does not depend on any arbitrary ordering.
(c) The kernel trick allows us to use non-linear features in a geometric algorithm, be it a logistic regression or a support vector machine.
(d) Perceptrons strive to find the correct non-linear scores function in a continuous space of programs, implicitely encoded in the structure of a "neural" network.

Conclusion

Overview. Today, we presented a handful of strategies to fit **trainable models** f_w to a supervised dataset (x_i, y_i). Interestingly, even though these algorithms are close to each other from a practical point of view (a one-layer perceptron is basically logistic regression...), they can be interpreted from very different viewpoints! In just one workshop session, we introduced:

- **Probabilistic** modeling: Logistic regression, Gaussian mixture models...
- **Geometric** classification: SVMs, k-means++...
- **Algorithmic** regression: Perceptrons, CNNs, RNNs...

Model evalutation. In the notebook, we asserted visually the relevance of our classifiers. But as soon as the dimension of our data points exceeds 3, this becomes intractable! The usual safeguard here is to split the dataset into a **training** set, used by the optimization loop, and a **test set** used to compute a final classification score. If both subsets are independent (and if one does not fine-tune the hyper-parameters too much), this protocol allows us to get an **independent** measure of the quality of our trained model.

How can machine learning algorithms generalize on unobserved data? The data scientist's fiercest enemy is **overfit**: when a model has **memorized** the training set well, without actually **learning** any generalizable decision rule. This can for instance happen if the trained model f_w converges towards the "hash table" function

$$f_w \longrightarrow \sum_{i \in \text{Training}} y_i \cdot 1_{\{x_i\}}. \tag{9.27}$$

Putting the right amount of prior into a model. To prevent this from happening, the key ingredient is the **regularization prior** encoded in our algorithm. It should be strong enough to filter out the acquisition noise in the dataset, and small enough to let us take advantage of every bit of relevant information present in the training set.

Balancing this regularization strength is easier said than done. Think, for instance, of linear models: As the classifier is constrained to being linear, we may naively assume that it cannot learn a gibberish decision rule... But unfortunately, this intuition turns out to be wrong if the dimension of the input vector x is too large! On images for instance, a linear classifier can easily overfit on the value of a single pixel – or any other meaningless decision rule – if it happens to "separate" the training set well enough.

Quantifying the strength of the regularization prior (compared to, say, the signal to noise ratio or the number of training samples) is a difficult task, and still an open problem for neural networks. In the algorithms that we presented today, the regularization is encoded in the following elements:

- **Linear** logistic regression and SVMs: dimensionality of the input space + explicit L^2 regularization prior on the linear scores matrix W. The latter may seem innocuous; but remember that if the canonical basis used to encode your data does not make sense with respect to your problem, neither does the L^2 norm! One could, at the very least, think of replacing it with another euclidean metric on the space of matrices.
- **Kernel** methods: intrinsic dimensionality of the dataset with respect to the kernel used. For instance, a large kernel width **blurs** the dataset and reduces the risk of overfit.
- **Neural networks**: implicit combination of the model's **architecture** + the L^2 (stochastic) gradient descent scheme used to optimize its weights.

SESSION 10 Automatic Differentiation

Automatic differentiation for applied mathematicians

In recent years, a considerable effort has been put into the development of modern scientific computing libraries, also known as *autodiff* frameworks. As of 2017, the most famous in the data science community are `Theano` (RIP), `TensorFlow` and `PyTorch`. Now, relative to the curriculum of math graduate students, those computational tools rely on *low-level* software engineering... But they can be so useful! Indeed, they now allow researchers to compute efficiently, on GPUs, the derivatives of any symbolic computational graph written in `python`.

To illustrate the internal design of these recent libraries, we now dedicate a whole workshop session to the step-by-step computation and differentiation of the classical *kernel norm*:

$$H(q,p) \;=\; \frac{1}{2}\sum_{i,j=1}^{N} k(q^i-q^j)\,\big\langle p^i,p^j\big\rangle_{\mathbb{R}^D} \tag{10.1}$$

$$\;=\; \frac{1}{2}\langle p, K_{q,q}p\rangle_{\mathbb{R}^{ND}} \qquad \text{with} \qquad (K_{q,q})_{i,j} \;=\; k(q^i-q^j), \tag{10.2}$$

where (q^i) and (p^i) are encoded as N-by-D float arrays. By the end of this session, hopefully, you should have a clear understanding of what is (and what is *not*) possible with an automatic differentiation framework.

N.B.: Those pages have been adapted from the documentation of the KeOps library, that I am currently writing with Benjamin Charlier and Joan Alexis Glaunès – `www.kernel-operations.io`.

Backpropagation 101

Finite differences are not the solution. Let $F:\mathbb{R}^n\to\mathbb{R}$ be a differentiable function defined as a symbolic computer program. Can we compute *efficiently* the (value,gradient) pair $(F(x_0),\partial_x F(x_0))$ at any given location $x_0\in\mathbb{R}^n$?

A naive approach, the so-called *finite differences* scheme, would be to use a Taylor expansion of F around x_0 and write, for δt sufficiently small,

$$\begin{pmatrix}\partial_{x^1}F(x_0)\\ \partial_{x^2}F(x_0)\\ \vdots\\ \partial_{x^n}F(x_0)\end{pmatrix} \;\simeq\; \frac{1}{\delta t}\begin{pmatrix}F(x_0+\delta t\cdot(1,0,0,\dots,0))-F(x_0)\\ F(x_0+\delta t\cdot(0,1,0,\dots,0))-F(x_0)\\ \vdots\\ F(x_0+\delta t\cdot(0,0,0,\dots,1))-F(x_0)\end{pmatrix}. \tag{10.3}$$

This idea is simple to implement. But it also requires $n+1$ evaluations of the function F to compute a *single* gradient vector! As soon as the dimension of the input space exceeds 10-100, this is not tractable: Just like inverting a full matrix A is not a sensible way of solving the linear system "$Ax=b$", one should not use finite differences – or any equivalent *forward* scheme – to compute a gradient.

Gradients between Hilbert spaces. Thankfully, there exists an efficient way of computing the gradients of real-valued functions: the reverse accumulation scheme. Relying on a *backward* pass through the computational graph, this useful algorithm has recently been popularized under the name of "*backpropagation*" and lies at the core of every Deep Learning framework. To understand it, consider the following definition of (generalized) gradients between Hilbert spaces:

Définition 10.1. Let $(X, \langle \cdot, \cdot \rangle_X)$ and $(Y, \langle \cdot, \cdot \rangle_Y)$ be two Hilbert spaces, and let $F : X \to Y$ be a continuously differentiable function between them.
Let also $x_0 \in X$ be an input position and $\alpha \in Y^*$ be a linear form on Y, which we identify with a vector $a \in Y$ through the Riesz theorem.
Then, for all increment $\delta x \in X$, we have

$$\begin{aligned} \langle \alpha, F(x_0 + \delta x) \rangle &= \langle \alpha, F(x_0) \rangle + \langle \ \alpha, \ \mathrm{d}_x F(x_0) \cdot \delta x \ \rangle + o(\|\delta x\|) && (10.4) \\ &= \langle \alpha, F(x_0) \rangle + \langle \ (\mathrm{d}_x F)^*(x_0) \cdot \alpha, \ \delta x \ \rangle + o(\|\delta x\|) && (10.5) \\ &= \langle a, F(x_0) \rangle_Y + \langle \ \partial_x F(x_0) \cdot a, \ \delta x \ \rangle_X + o(\|\delta x\|), && (10.6) \end{aligned}$$

where we identify the adjoint of the differential $(\mathrm{d}_x F)^*(x_0) : Y^* \to X^*$ with a continuous linear map $\boldsymbol{\partial_x F(x_0) : Y \to X}$ through the Riesz theorem. We say that the latter is the **generalized gradient** of F at x_0, with respect to the Hilbertian structures of X and Y.

If X and Y are respectively equal to $\mathbb{R}^n$ and $\mathbb{R}$ endowed with their canonical L^2-Euclidean structures, the matrix of $\partial_x F(x_0)$ in the canonical basis is no one but the vector $\nabla_x F(x_0)$ of directional derivatives.

Chain rule for gradients. This Hilbertian definition of the gradient has two major advantages over the "vector of derivatives" one. First, it stresses the fact that a gradient is an object which is defined with respect to a *metric structure*, not a basis. As data scientists often work with spaces of signals on which the L^2 metric makes very little sense, this is important.

Second, it allows us to *compose* gradients without reserve. Indeed, if X, Y, Z are three Hilbert spaces, and if $F = H \circ G$ with $G : X \to Y$ and $H : Y \to Z$, then for all $x_0 \in X$, the chain rule asserts that

$$\mathrm{d}_x F(x_0) = \mathrm{d}_y H(G(x_0)) \circ \mathrm{d}_x G(x_0), \tag{10.7}$$

so that

$$\begin{aligned} [\mathrm{d}_x F(x_0)]^* &= [\mathrm{d}_x G(x_0)]^* \circ [\mathrm{d}_y H(G(x_0))]^* && (10.8) \\ \text{i.e.} \qquad \partial_x F(x_0) &= \partial_x G(x_0) \ \circ \ \partial_y H(G(x_0)). && (10.9) \end{aligned}$$

Backpropagation. Suppose that the function of interest $F : \mathbb{R}^n \to \mathbb{R}$ is defined as a composition $F = F_p \circ \cdots \circ F_2 \circ F_1$ of elementary functions $F_i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ where $N_0 = n$ and $N_p = 1$:

$$\mathbb{R}^n = \mathbb{R}^{N_0} \xrightarrow{F_1} \mathbb{R}^{N_1} \xrightarrow{F_2} \mathbb{R}^{N_2} \xrightarrow{\cdots} \cdots \xrightarrow{F_p} \mathbb{R}^{N_p} = \mathbb{R}$$

To keep the notations simple, we will assume that all the input and output spaces $\mathbb{R}^{N_i}$ are endowed with their canonical L^2-Euclidean metrics. Remember that we are interested in computing, at an arbitrary location $x_0 \in \mathbb{R}^n$, the gradient

$$\partial_x F(x_0) : \mathbb{R} \to \mathbb{R}^n, \tag{10.10}$$

which is a linear map that is entirely determined by the value of the "gradient vector"

$$\begin{aligned} \partial_x F(x_0)\cdot 1 &= \partial_x F_1(x_0)\circ \partial_x F_2(F_1(x_0))\circ\cdots\circ \partial_x F_p(\,F_{p-1}(\cdots(F_1(x_0)))\,)\cdot 1 && (10.11)\\ &= \partial_x F_1(x_0)\circ \partial_x F_2(\quad x_1\quad)\circ\cdots\circ \partial_x F_p(\qquad x_{p-1}\qquad)\cdot 1 && (10.12)\end{aligned}$$

where the $x_i = F_i\circ\cdots\circ F_1(x)$ are nothing but the intermediate results in the computation of $x_p = F(x_0)$.

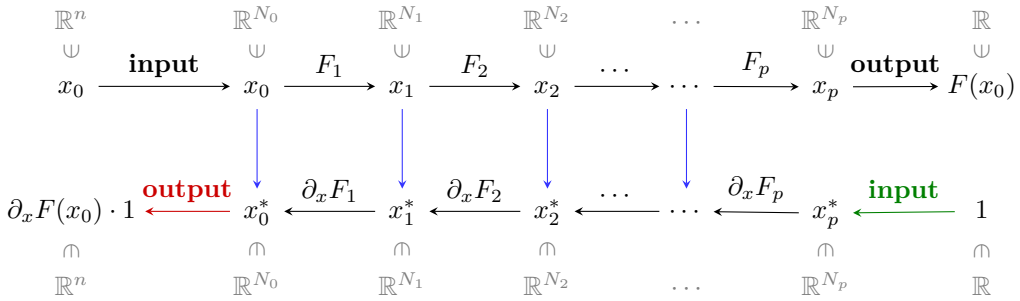
We then assume that the *forward* and *backward* operators

$$\begin{aligned} F_i \;:\; & \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i} \\ & x \mapsto F_i(x) \end{aligned} \qquad (10.13)$$

and

$$\begin{aligned} \partial_x F_i \;:\; & \mathbb{R}^{N_{i-1}}\times\mathbb{R}^{N_i} \to \mathbb{R}^{N_{i-1}} \\ & (x_0, a) \mapsto \partial_x F_i(x_0)\cdot a \end{aligned} \qquad (10.14)$$

are known and **encoded as computer programs**. According to Eq. (10.12), it is thus possible to compute both $F(x_0)$ and $\partial_x F(x_0)$ with a forward-backward pass through the following diagram:



The *backpropagation* algorithm can be cut in two steps that correspond to the two lines of the above diagram:

1. Starting from $x_0 \in \mathbb{R}^n = \mathbb{R}^{N_0}$, compute and **store in memory** the successive vectors $x_i \in \mathbb{R}^{N_i}$. The last one, $x_p \in \mathbb{R}$, is equal to the value of the objective $F(x_0)$.
2. Starting from the canonical value of $x_p^* = 1 \in \mathbb{R}$, compute the successive *dual* vectors

$$x_i^* = \partial_x F_{i+1}(x_i)\cdot x_{i+1}^*. \qquad (10.15)$$

   The last one, $x_0^* \in \mathbb{R}^n$, is equal to the gradient $\nabla F(x_0) = \partial_x F(x_0)\cdot 1$.

Implementation and performances. The generalization of this procedure to any acyclic "forward" computational graph is straightforward. Hence, provided that the forward and backward operators of Eq. (10.13-10.14) are pre-implemented, we can compute *automatically* the gradient of any symbolic procedure that is written as a succession of elementary vector operations, the F_i's.

Consequently, Deep Learning libraries rely on three core modules: a `numpy`-like set of low-level GPU routines; a high-level graph manipulation API; a comprehensive list of operations (forward and backward) provided to end-users.

Crucially, the *backwards* of the usual operations are seldom more costly than 4-5 applications of the corresponding *forward* operators. Ergo, if one has enough memory available to store the intermediate results during the forward pass, **the backpropatation algorithm is an automatic and time-effective way of computing arbitrary gradients**.

Autodiff is easy to use

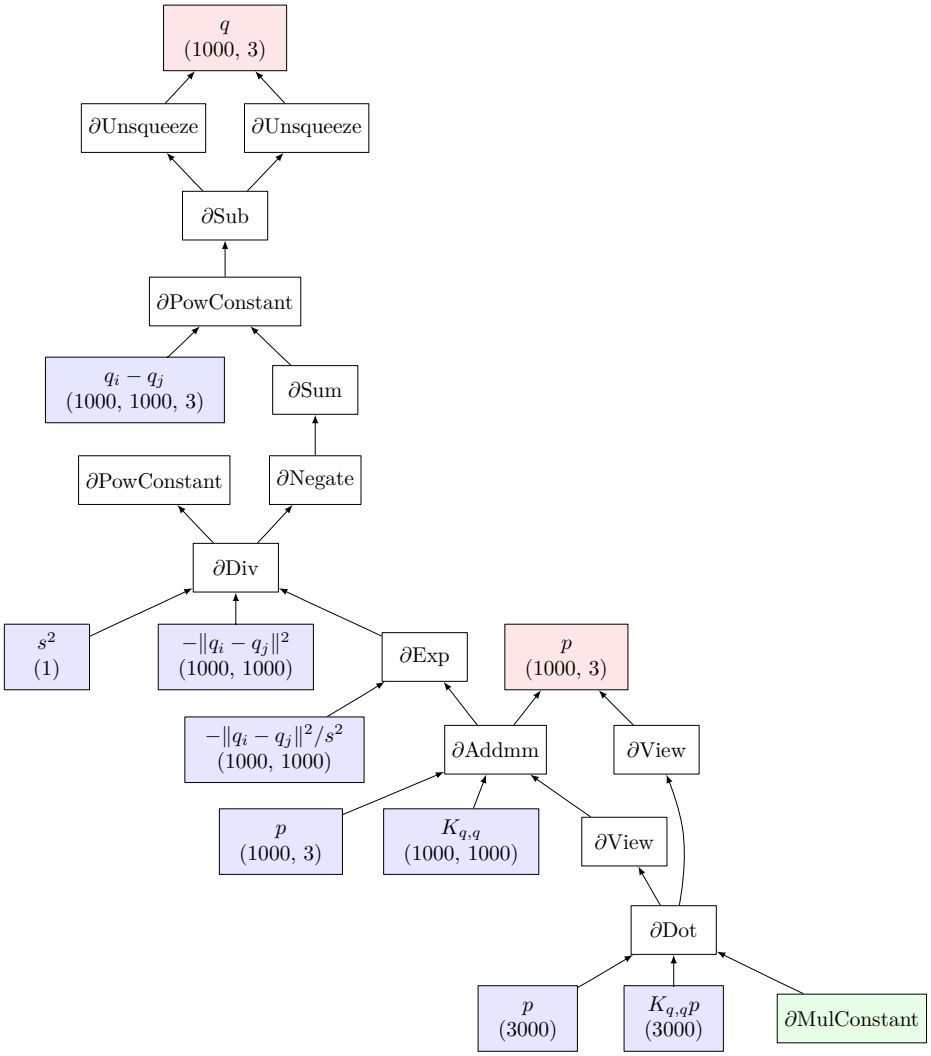
A minimal working example. Let us illustrate the underlying mechanics of `PyTorch` – a Deep Learning library – in a simple case: the computation of the kernel squared norm defined Eq. (10.1-10.2) with a *Gaussian kernel* of deviation $s > 0$:

$$k \quad : \quad \begin{array}{ccc} \mathbb{R}^D \times \mathbb{R}^D & \to & \mathbb{R} \\ (x,y) & \mapsto & e^{-\|x-y\|_2^2 \,/\, s^2} \end{array} \cdot \tag{10.16}$$

```python
import torch                    # GPU + autodiff library
from torchviz import make_dot   # See github.com/szagoruyko/pytorchviz

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu else torch.FloatTensor
# Under the hood, this flag determines the backend used for
# forward and backward operations, which have all been
# implemented both in pure CPU and in GPU (CUDA) code.

# Step 1: Define numerical tensors (from scratch or numpy) --------
N = 1000; D = 3 ; # Work with clouds of 1,000 points in 3D
# Generate arbitrary arrays on the host (CPU) or device (GPU):
q = torch.linspace( 0, 5, N*D ).type(dtype).view(N,D)
p = torch.linspace( 3, 6, N*D ).type(dtype).view(N,D)
s = torch.Tensor(     [2.5]   ).type(dtype)

# Step 2: Tell PyTorch to keep track of q and p's children --------
# In this demo, we won't try to fine tune the deformation model,
# so we do not need any derivative with respect to s:
q.requires_grad = True
p.requires_grad = True

# Step 3: Actual computations -------------------------------------
# Every PyTorch instruction is executed on-the-fly, but the graph
# API 'torch.autograd' keeps track of the operations and stores in
# memory the intermediate results needed for the backward pass.
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix multiplication. (N,N)@(N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p):
H    = .5 * torch.dot( p.view(-1), v.view(-1) ) # .5*<p,v>

# Display -- see next figure.
print(H); make_dot(H, {'q':q, 'p':p, 's':s}).render(view=True)
```

For the sake of completeness, we provide here a full, verbose working example: its header is bound to get deprecated sooner or later. But the core of the procedure, the lines related to the kernel norm formula, those are here to stay. From a mathematical point of view, these symbolic python instructions define a *computational graph* that can be used to differentiate $H(q, p)$ with respect to q and p.

Encoding a formula in the computer's memory In the figure below, we display the computational history of the variable `'H'` as it is understood by `PyTorch` – it is stored in the `'H.grad_fn'` attribute: have a look! This acyclic graph is the exact equivalent of the second "backward" line of the diagram presented page 69: Every white node stands for a backward operator $\partial_x F_i : (x_i, x_{i+1}^*) \mapsto x_i^*$. The green leave is the first covariable $x_p^* \in \mathbb{R}$, the "gradient with respect to the output" which is initialized to 1; the red leaves are the covariables x_0^* in which the gradients are to be accumulated; and the blue ones are the *stored* values x_i computed during the forward pass.



Computing gradients for free. Thanks to the groundwork done by `PyTorch`'s developers, computing gradients with `Python` is *that* simple:

```
# N.B.: Higher-order derivatives are also supported:
# just use the create_graph=True optional argument of 'grad'
grad_q, grad_p = torch.autograd.grad( H, [q,p] )
print(grad_q.shape, grad_p.shape)
```

As we will show in the next few sessions, this new feature provided by autodiff frameworks is a game changer for applied mathematicians. Before going any further, I thus recommend to go through the `PyTorch` tutorial available at the following address:

`pytorch.org/tutorials/beginner/pytorch_with_examples.html`

Bonus track: linking custom CUDA routines with `PyTorch`

As an appendix to this session dedicated to automatic differentiation, let me now describe the proper way of extending the library by linking custom CUDA routines to `PyTorch` symbolic instructions. This will allow us to dive into the framework's internal behaviors and paradigms, and thus get a clear understanding of its limits.

Why Memory usage. Out of the box, `PyTorch` is a fantastic library. But in the code written above, we can still point out the extravagant memory usage required for the computation of the velocity field `v = K(q,q) @ p`: to differentiate "PowConstant", "Div", "Exp" and "Addmm", the backpropagation algorithm has to store in memory full N-by-N intermediate results: $q_i - q_j$, $-\|q_i - q_j\|^2$, etc.

Therefore, the native `PyTorch` implementation of page 72 is intractable as soon as the number of points N exceeds the **square root of the GPU memory** – that is, about $50,000$ for a recent piece of hardware.

Towards more flexibility. To break this ceiling, one can wrap the critical computation of `'v'` into a generic and memory efficient operator: the `KernelProduct` object, which implements the kernel convolution formula. As `KernelProduct` takes as input two point clouds and one momentum field – of respective shapes `(N,D)`, `(M,D)` and `(M,E)` – to output a momentum field of shape `(N,E)`, the `torch.autograd` module will never need to store full `(N,M)` arrays in the GPU memory. This way of doing bypasses the built-in `PyTorch` operators to rely on a finely crafted CUDA memory management scheme.

Static autodiff. Remember. The legacy `Theano` (2008-2017) library divided in *three steps* the translation of a `python` symbolic script into an efficient GPU routine. First, the `python` programmer declared a whole computational graph at once, without any actual *computation* taking place. Then, a graph optimizer pruned out unused nodes, merged subgraphs, etc., and automatically generated a C/CUDA program. The latter was then compiled using `gcc`, and the resulting executable was linked to a wrapper `python` function.

This way of doing made differentiation look easy: `'.grad'` was just another symbolic node in a purely abstract computational graph. Unfortunately, it also induced two adverse side effects: a lengthy compilation time at the start of every single script; the inability to implement dynamic flow control (`if-then-else` structures, etc.), which make the operations applied to arrays depend on their actual *values*.

The dynamic workflow. The `PyTorch` library has recently been introduced to cover those deficiencies. Unlike the static Declaration-Optimization-Compilation frameworks, it implements a *dynamic* workflow which can be summarized as follows:

1. `Variables` are seen as graph objects, wrapped around `int/float` arrays.
2. Instructions are executed on-the-fly, using pre-compiled CPU or GPU routines in the back end. The result of any such operation is a new `Variable` wrapped around an "output" array, with a "graph history" attribute keeping track of all the operations that are needed to compute the output.
3. As differentiation upsets the whole "graph history" of input `Variables`, it is handled by a specific bunch of instructions. If a variable `H` was computed using two user-defined variables `q` and `p`, the most math-like way of applying a backpropagation pass on the graph history of `H` is to write:

   ```
   [dq,dp] = torch.autograd.grad( H, [q,p], g, create_graph=True)
   ```

   where `g` is the "ìnput" gradient x_p^* with respect to `H`, initialized by default to `1` if `H` is scalar. This special instruction outputs two variables `dq` and `dp` whose numerical values were computed as output of the *backpropagation* algorithm.

Computing second derivatives. The default behaviour of `autograd.grad` is to output `Variable` objects with a blank history: if one simply needs to do gradient descent on `H`, keeping track of the computational history of the gradients is basically useless. However, differentiating functions such as the Hamiltonian *a second time* is crucial in many applications – for instance, shape analysis.

This is made possible by the `create_graph` flag. When it is set to the value `True` – as in the above instruction – `PyTorch` considers that the diagram displayed in page 69 is a new "*forward*" program which takes as input the vector x_0 and the covector x_p^*, to output the backpropagated gradient x_0^*.

This means that the `autograd.grad` instruction can be differentiated once again, at the condition that the order-0 and order-1 operators F_i and $\partial_x F_i$ defined Eq. (10.13-10.14) are available as computer programs... **As well as their own gradients**. In practice, this means that `PyTorch` must know how to compute the "gradients of gradients" operators:

$$\begin{array}{rcccc} \partial_{x_0}(\partial_x F_i(x_0)\cdot a) & : & \mathbb{R}^{N_{i-1}}\times\mathbb{R}^{N_i}\times\mathbb{R}^{N_{i-1}} & \to & \mathbb{R}^{\mathbf{N_{i-1}}} \\ & & (\boldsymbol{x_0},\boldsymbol{a},\boldsymbol{e}) & \mapsto & \partial_{x_0}(\partial_x F_i(x_0)\cdot a)(\mathbf{x_0},\mathbf{a})\cdot\mathbf{e} \end{array} \tag{10.17}$$

$$\begin{array}{rcccc} \partial_{a}\ (\partial_x F_i(x_0)\cdot a) & : & \mathbb{R}^{N_{i-1}}\times\mathbb{R}^{N_i}\times\mathbb{R}^{N_{i-1}} & \to & \mathbb{R}^{\mathbf{N_i}} \\ & & (\mathbf{x_0},\mathbf{a},\mathbf{e}) & \mapsto & \partial_{a}(\partial_x F_i(x_0)\cdot a)(\mathbf{x_0},\mathbf{a})\cdot\mathbf{e} \end{array} \tag{10.18}$$

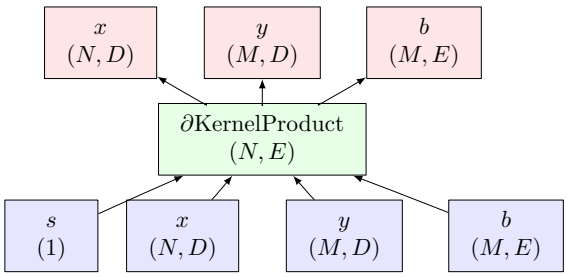
The convolution operator. The simplest way of doing so is to define operators of "order 1" such as `KernelProductGrad_x`, and explicitly use them in the `backward` method of our order 0 operator, `KernelProduct`.

As `PyTorch` is not yet fully documented, we provide below the meaningful elements of syntax that can allow one to implement a twice-differentiable CUDA-based operator. This may help other researchers to get their own non-standard ideas to work on real data. Going further, one can *bootstrap* these routines and define infinitely differentiable programs, as documented in our reference code:

`https://www.kernel-operations.io`.

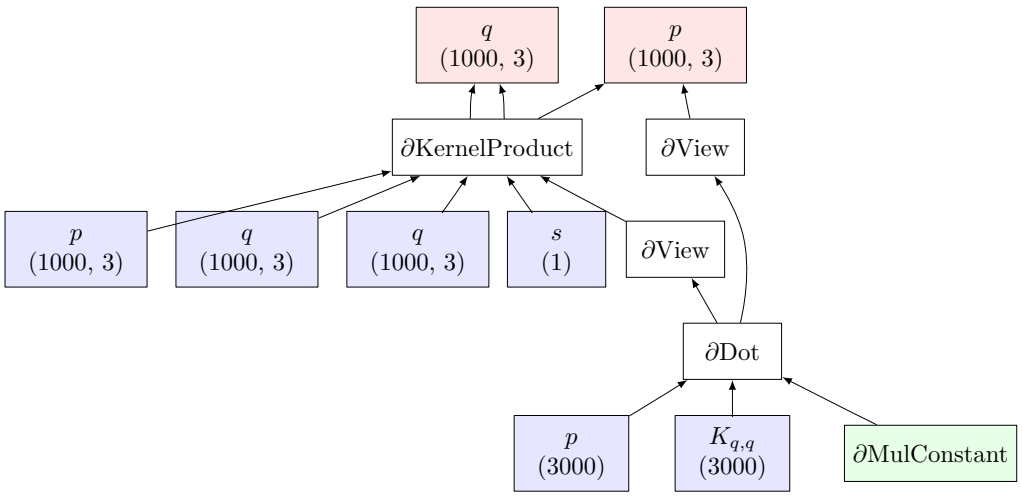
```
import torch
class KernelProduct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, s, x, y, b, kernel_type):
        # save everything to compute the gradient
        ctx.save_for_backward( s, x, y, b )
        # init gamma, the output of the convolution K_xy @ b
        gamma  = torch.zeros( x.size()[0] * b.size()[1] ).type(dtype)
        # Inplace CUDA routine on the raw float arrays,
        # loaded from .dll/.so files by the "ctypes" module
        cudaconv.cuda_conv(    x.numpy(), y.numpy(), b.numpy(),
                           gamma.numpy(), s.numpy(),
                           kernel = kernel_type)
        gamma  = gamma.view( x.size()[0], b.size()[1] )
        return gamma

    @staticmethod
    def backward(ctx, a):
        (ss, xx, yy, bb) = ctx.saved_variables
        # In order to get second derivatives, we encapsulated the
        # cudagradconv.cuda_gradconv routine in another
        # torch.autograd.Function object:
        kernelproductgrad_x = KernelProductGrad_x().apply

        # Call the CUDA routines
        # ...
        grad_x = kernelproductgrad_x( ... )
        # ...
        return (grad_s, grad_x, grad_y, grad_b, None)

class KernelProductGrad_x(torch.autograd.Function):
    @staticmethod
    def forward(ctx, s, a, x, y, b, kernel_type):
        # Save for Backward + Call the CUDA routines
        # ...
        return grad_x

    @staticmethod
    def backward(ctx, e):
        # Call the CUDA routines
        # ...
        return (grad_xs, grad_xa, grad_xx, grad_xy, grad_xb, None)
```

The resulting object can now be used seamlessly in a `PyTorch` computation, taking as input a kernel size, a kernel type (such as `"gaussian"` or `"energy"`) and three tensors. `KernelProduct` is as easy to use as a built-in operator and stands for the following piece of graph:



Computing a kernel product efficiently. This `python` object can be used to compute "kernel", "currents", "varifolds" or "Optimal Transport" discrepancies between shapes. Crucially, it can also be used in the declaration of the Hamiltonian: in the `PyTorch` example showcased page 72, one simply has to replace the lines 25-35 with the code shown below. The resulting computational graph is then mathematically equivalent to that of page 71, but doesn't store any large matrix in memory.

```
# Compute the kernel convolution
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p):
H = .5 * torch.dot( p.view(-1), v.view(-1) ) # .5*<p,v>
```



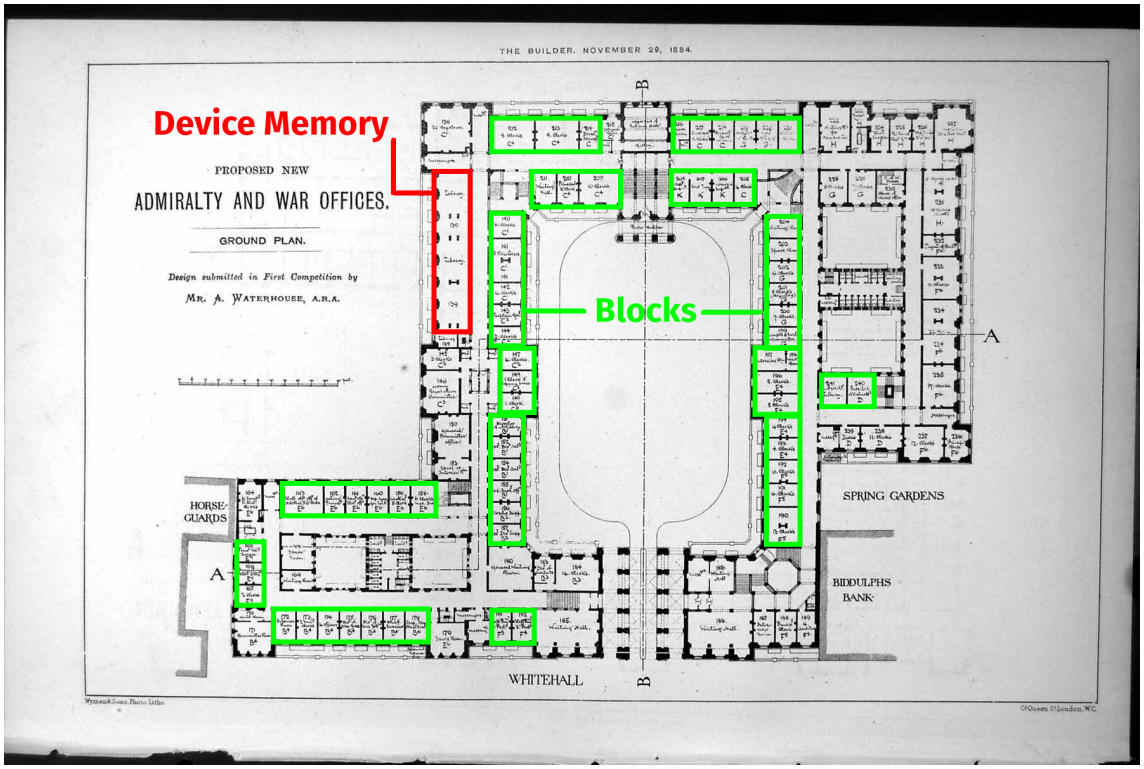


Figure 10.1: GPU architecture.

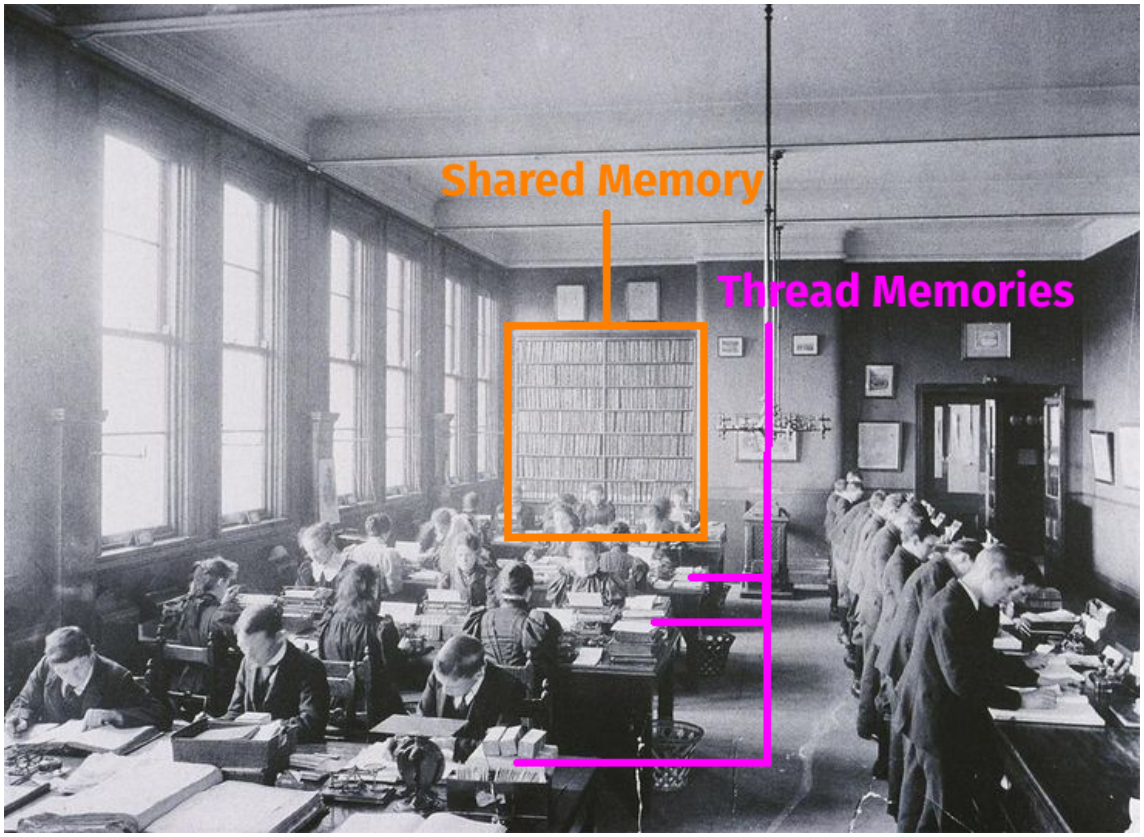


Figure 10.2: Inside a computational block.

SESSION 11 Introduction to deep-learning

Deep Learning 101

In today's workshop, we're going to classify images using neural networks and nonlinear image transforms. The emphasis will be put on models' **architectures**, with the actual training and visualization code kept into routines such as `evaluate_model`, located in the `model_utils` file.

De-mystifying neural networks. The main purpose of these sandbox notebooks is to let you play around with models, "neurons" and convolution filters. As you get to see how these things work under the hood, you should hopefully understand:

- Just how much of a game changer the **Autodiff+GPU** combo can be: today in computer vision (and natural language processing), tomorrow in your own research field, whatever it is.
- How **Convolutional Neural Networks** can be linked, (at the very least) from an algorithmic perspective, to **Wavelet Transforms**.
- That the current "Artifical Intelligence" hype around image processing algorithms does not come from scientists. Using momentum-based gradient descent (i.e. letting a heavy ball roll on a hyper-surface of potential) to fine tune the parameters of a wavelet-like transform can help you to extract the most relevant features in your signal - which is an **incredibly useful pre-processing step** with tons of industrial applications. But it can not, in any way, model **thought**. As far as the modelling of cognitive processes is concerned, we're still very much in the stone age.

Anyway, let's get started. If you haven't done it already, please go through the `PyTorch` syntax tutorial available at the following address:

`pytorch.org/tutorials/beginner/pytorch_with_examples.html`

Gradient descent: the stealth regularization prior

In the Deep Learning literature, researchers tend to use (stochastic/momentum-based) gradient descent as their go-to optimization procedure: this algorithm is both simple to implement and efficient in practice. But is it as innocuous/trivial as it seems to be? **No, it isn't.**

The gradient depends on your underlying metric. As we've seen in the previous workshop sessions, the gradient is fundamentally a **metric** object. If $f : X \to Y$ is a differentiable function between two Euclidean/Hilbert spaces, it is defined as the **adjoint of the differential, seen through the Riesz isomorphism**; that is, as the unique application $\partial_x f(x_0) : Y \to X$ such that

$$\forall\, b \in Y, \forall\, \delta x \in X, \quad \langle f(x_0 + \delta x)\,,\, b\rangle_Y \;=\; \langle f(x_0)\,,\, b\rangle_Y \;+\; \langle \delta x\,,\, \partial_x f(x_0).b\rangle_X \;+\; o(\delta x). \tag{11.1}$$

If $f : \mathbb{R}^N \to \mathbb{R}$ is a cost function, `PyTorch` computes its "algorithmic" gradient

$$\partial_x^{L^2} f(x_0) : \mathbb{R} \to \mathbb{R}^N \;\simeq\; (\partial_{x[1]} f(x_0), \cdots, \partial_{x[N]} f(x_0))^T \tag{11.2}$$

which is nothing but the L^2-gradient associated to the canonical L^2-norm on $X = \mathbb{R}^N$:

$$\forall\, x \in \mathbb{R}^N, \quad \langle x, x\rangle_X \;=\; \sum_{i=1}^{N} x[i]^2. \tag{11.3}$$

More often than not, this choice is a good one. But you should never forget how much it depends on the way you encoded your data vector x: **with respect to your real-life problem, the L^2 gradient is just as (ir)relevant as the L^2 unit ball**. Does it make sense? Great. Otherwise, you should be *really* careful about the bias, the **implicit** regularization prior you're introducing in your algorithm.

To illustrate this, consider the minimization of a blur-distance function

$$f_{k,y} : x \in \mathbb{R}^N \mapsto \|k \star x - y\|_2^2, \tag{11.4}$$

where $y \in \mathbb{R}^N$ is a target signal and k is a 1D-convolution kernel. If the Fourier transform of k is never equal to zero, the associated convolution operator is invertible; in this case (say, if k is Gaussian), $f_{k,y}$ is a **positive definite quadratic form** defined on $\mathbb{R}^N$, whose unique minimum is the well-defined signal

$$x^* \;=\; k^{(-1)} \star y. \tag{11.5}$$

Naive theoretical prediction. The strictly convex function $f_{k,y}$ is as simple as it gets, in a finite-dimensional space: Therefore, a standard gradient descent (with a small stepsize) should quickly converge to x^*, the unique signal such that $k \star x^* = y$. Let's see how this works in practice.

After a sharp decline, the algorithm seems to get stuck and converge... **towards a value which is not that of the unique critical point of our function**, $f_{k,y}(x^*) = 0$.

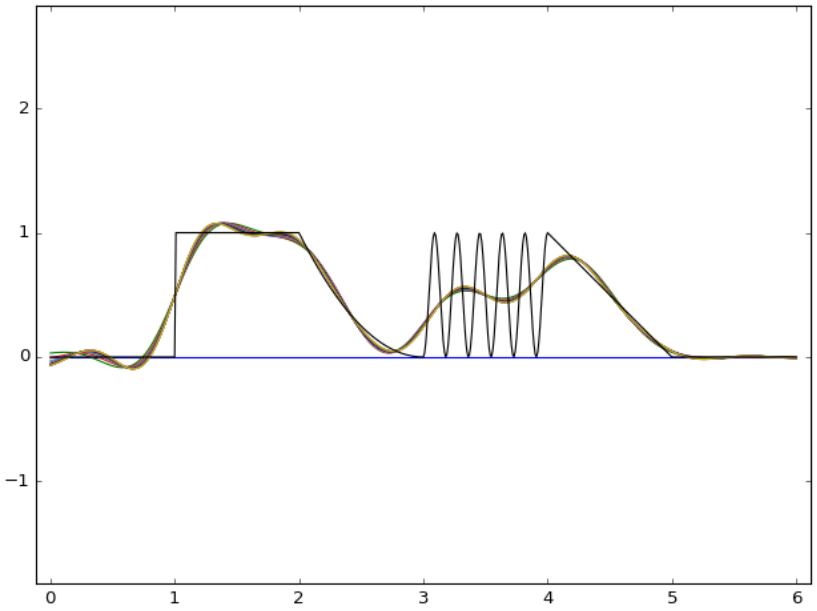


Figure 11.1: As we try to solve the deconvolution problem with the black curve standing for y, starting from the null blue curve, our **unconstrained** L^2 gradient descent scheme gets stuck in a position which is as close as possible to the target y, while sharing the same (nil) high frequency content as the initial guess.

Computing the L^2 gradient. To understand the "smoothing" behavior of the L^2 gradient descent in this particular example, we have to open the black-box and actually compute $\partial_x^{L^2} f_{k,y}(x_0)$ at an arbitrary location x_0. By definition, it is the unique vector of $X = \mathbb{R}^N$ such that, for all $\delta x \in X$ and $b \in \mathbb{R}$,

$$\langle\ f_{k,y}(x_0 + \delta x)\ ,\ b\ \rangle_{\mathbb{R}}\ =\ \langle\ f_{k,y}(x_0)\ ,\ b\ \rangle_{\mathbb{R}}\ +\ \langle\ \delta x\ ,\ \partial_x f_{k,y}(x_0).b\ \rangle_2\ +\ o(\|\delta x\|_2). \tag{11.6}$$

Since we know that

$$f_{k,y}(x_0 + \delta x)\ =\ \langle\ (k \star x_0 - y)\ +\ k \star \delta x\ ,\ (k \star x_0 - y)\ +\ k \star \delta x\ \rangle_2, \tag{11.7}$$

we have

$$\partial_x^{L^2} f_{k,y}(x_0)\ =\ 2\ \tilde{k} \star (k \star x_0 - y), \tag{11.8}$$

where $\tilde{k}$ is the mirrored symmetric of k, the unique filter such that "$\tilde{k} \star \cdot$" is the L^2-adjoint of "$k \star \cdot$". If k is a centered Gaussian kernel, one simply has $\tilde{k} = k$.

Poorly conditioned operators. Hence, computing the L^2 gradient of $f_{k,y}$ involves a **smoothing** of the difference vector $k \star x_0 - y$. As this operation all but kills the high frequency components, our gradient descent scheme **has no way to generate high frequencies in x, nevermind $k \star x$.** In practice, using the L^2 gradient to minimize $f_{k,y}$ is thus akin to enforcing a **soft restriction**: that of only looking for vectors x which have "the same high frequency content" as the initial guess $x(\text{it} = 0)$.

Using arbitrary descent metrics. We won't discuss this topic any further. But please note that if $A : \mathbb{R}^N \to \mathbb{R}^N$ is an invertible linear matrix of size N-by-N, then AA^T is a symmetric positive definite matrix which defines a metric on $\mathbb{R}^N$:

$$\forall\, x \in \mathbb{R}^N,\ \ \langle x, x \rangle_{AA^T}\ =\ \langle x, AA^T x \rangle_2. \tag{11.9}$$

Then, if $f : \mathbb{R}^N \to \mathbb{R}$ is a differentiable function, **the minimization of f with an AA^T-gradient is equivalent to the minimization of $f \circ A^{-1}$ with an L^2 gradient**. Using `PyTorch`, we can thus use arbitrary descent metrics by introducing hidden changes of variables.

An image classification problem

These preliminaries in mind, we can now use a "neural" network to tackle our first **image classification problem**. Unlike what was presented in the previous workshop session, we won't add an explicit regularizer to our cross-entropy loss: **we rely entirely on the implicit regularization provided by the L^2 descent scheme**, and hope for the best.

Loading the dataset. Our task: classifying into ten groups the images from the "FashionMNIST" dataset of whom you can get an overview at the following address: `github.com/zalandoresearch/fashion-mnist`. A sample is displayed below:

Tackling the problem using a multilayer perceptron

Neural nets reminder. Just as in the previous workshop session, we can try to "learn" a regression rule by optimizing the weights of a fully connected network. Formally, we're trying to optimize the weights W_1, b_1 and W_2, b_2 of a couple of operators (F_1, F_2) given by

$$F_1(x) = \quad \text{Relu}\ (W_1 \cdot x + b_1) \tag{11.10}$$
$$F_2(x) = \text{argmax}(W_2 \cdot x + b_2) \tag{11.11}$$

where $\text{Relu} : x \mapsto \max(0, x)$ is applied pointwise, and argmax goes from $\mathbb{R}^{10}$ to $\{1, 2, \cdots, 10\}$. The global model $F = F_2 \circ F_1$ is the concatenation of those two operators, and we wish to find F such that $F(x_i)$ is as often as possible equal to the label y_i on the test set. That is, we wish to minimize

$$\sum_i 1_{F(x_i) \neq y_i} \tag{11.12}$$

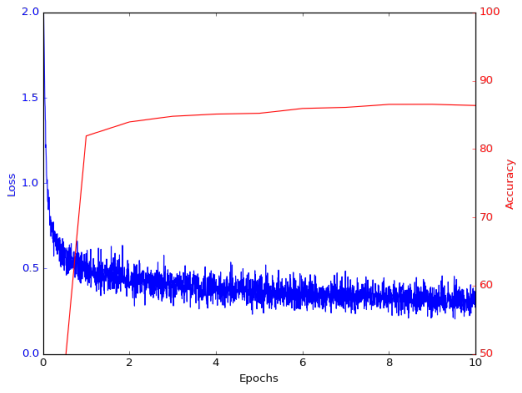
on the test set, having only tuned the parameters on the "training" set.

The logistic embedding. Because the argmax operator is piecewise constant, its gradient is pretty uninformative and cannot drive an optimization routine. Hence, we replace it with a softmax operator

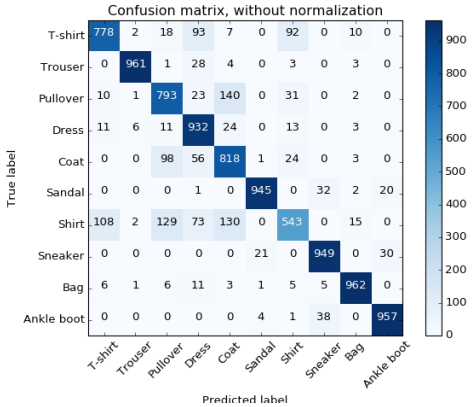
$$\text{Softmax} : x \in \mathbb{R}^{10} \mapsto \left(\frac{e^{x_i}}{\sum_j e^{x_j}} \right)_i \in \mathbb{R}^{10} \tag{11.13}$$

and strive to minimize the cross-entropy loss seen in the previous workshop session (Softmax Logistic regression).

Results. The performances on the test dataset are surprisingly good: most of the confusion comes from classes which are "close" to each other such as "Shirts" and "T-shirts". Even a human operator could get somewhat confused isn't it?



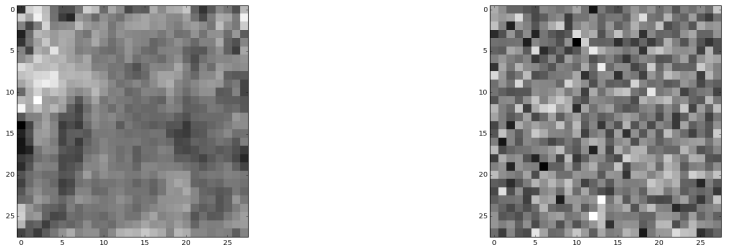
(a) Tracking the training process.



(b) Final confusion matrix.

Figure 11.2: Training of a two-layer perceptron on the FashionMNIST dataset.

Visualizing our classification program. To understand the decision rule, we generate by gradient ascent on the score (from a user-defined starting point) an image that our network classifies as "100% a T-shirt":



(a) Starting from a zero image. (b) Starting from a noisy image.

Figure 11.3: Images labelled as T-shirts with full confidence by our trained 2-layer perceptron. We can more or less see the shape of a shirt in (a)... However, keep in mind that our model **has no prior knowledge of what a natural image should look like**. Hence, it does not distinguish (a) from (b), and classifies both as T-shirts.

Tuning our network's architecture. To gain a few percents, one can finely tune the data flow and create more and more complicated models. In the notebook, we present a model (freely available on GitHub) that was written with this dataset in mind and indeed performs slightly better on the MNIST dataset.

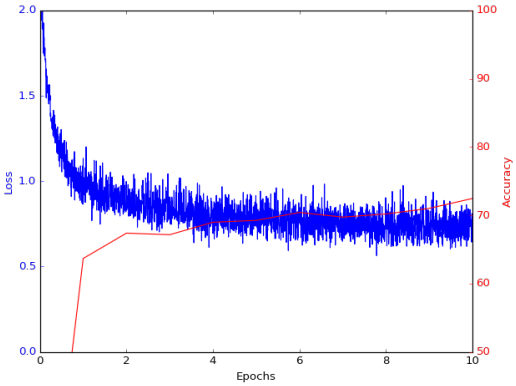
Remember: Thanks to the flexibility of Autodiff libraries, we can now implement and optimize the parameters of **any model that fits in memory**. Given this astounding freedom, the real question thus becomes: how should we design our image processing programs?

A more realistic dataset

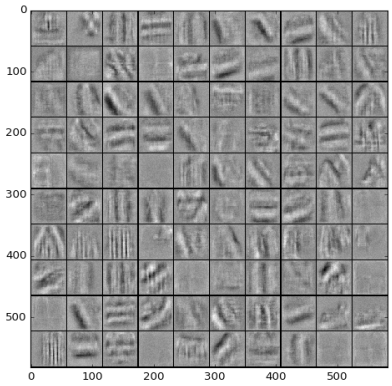
Perceptrons are fine for images that are perfectly centered as they can, for instance, learn that "sandal" images will never present white pixels in the top left corner. But all of this sounds a bit too easy, isn't it? We now want to focus on a more realistic classification problem: a FashionMNIST dataset in which pieces of clothing are **not perfectly centered**. This is relevant as in practice, the correct segmentation of image parts is at least as difficult as the classification of normalized and centered images...

Loading the dataset - and applying random translations. In the notebook, we take advantage of a nice `PyTorch` syntax to apply randomized transformations every time an image is loaded. A random sample is displayed below:

First try : a good old two-layer perceptron. Can the fully connected perceptron perform well in this "un-sanitized" environment? Well, not really. As shown below, test set accuracy is considerably lower than in the centered case.



(a) Monitoring the learning process.



(b) Visualizing the first hidden layer.

Figure 11.4: Training a two-layer perceptron on the "un-centered" FashionMNIST dataset.
(a) Notice the drop in performance, compared with Figure 11.2.
(b) Every little square corresponds to a "neuron" of our perceptron's first operator – really, these are linear forms on the space of images. Quite remarkably, the first operator of our model has converged towards a DiscreteCosinus-like Transform, made out of stripes! This makes sense, as a linear and translation-invariant operator is necessarily diagonal in the Fourier basis: even though it relies on non-linearities, our network has converged towards a kind of "spectral" translation-invariant classifier.

Enforcing translation invariance with convolutions

Expecting a generic perceptron to give perfect results was too optimistic: as this model is theoretically able to emulate *any* classifier, it is prone to overfit the training data. In a sense, we already encountered such a problem in the Wavelet Thresholding Numerical Tour:

`nbviewer.jupyter.org/github/gpeyre/numerical-tours/blob/master/python/denoisingwav_2_wavelet_2d.ipynb`

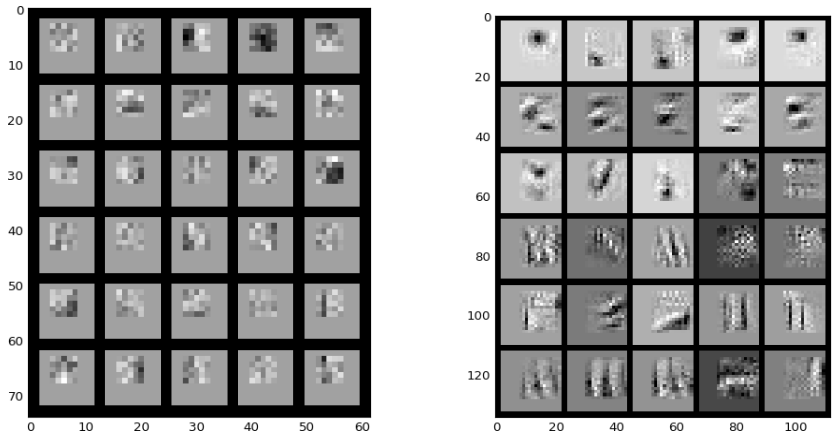
The translation invariance prior. In this previous workshop session, replacing the orthogonal wavelet transform with a **translation-invariant transform** (using cycle-spinning or the *algorithme à trous*) dramatically increased the robustness of wavelet-based denoising algorithms. Likewise, enforcing **translation invariance in perceptrons** will be a crucial step in the design of trainable operators for image processing. This prior is easy to enforce: we know that a translation-invariant linear operator can necessarily be represented as a **convolution operator**. It is thus natural to replace the **generic linear operators** known as "Fully connected layers" by their translation invariant counterparts, encoded as **sets of convolution filters**.

The multiscale prior. Furthermore, on top of translation invariance, we also know (or assume...) that natural images have a **multiscale structure**: there are relevant features at every scale, which are built as combinations of smaller details (edges $\rightarrow$ eyes $\rightarrow$ faces). In practice, this means that we should prefer architectures with **deep** cascades of **small** convolution filters. Just like in a wavelet transform!

Designing trainable transforms for image processing. A consensus emerged in the last few years: when designing a "neural network" (i.e. a trainable transform) for image classification tasks, we should typically restrict ourselves to a cascade of:

- Convolution operators such as `nn.Conv2d`.
- Su(b-p)sampling ("(un)pooling") operators such as `F.max_pool2d`.
- Pointwise operations such as `F.relu`.
- Utility operators such as batch normalization or dropout layers, that we won't detail here.

By enforcing a **relevant prior knowledge** into an algorithm-fitting method, we are able to achieve much better results. Please play around with the visualization routines provided in the notebook!



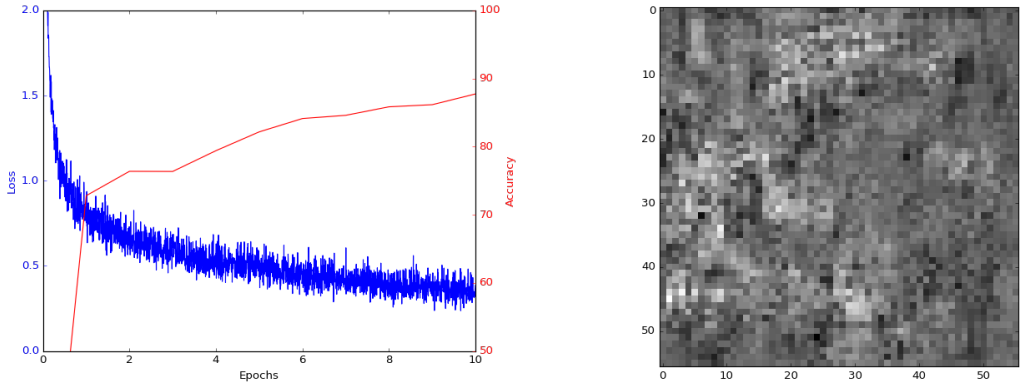
(a) First convolution layer. (b) Second convolution layer.

Figure 11.5: Visualizing some "neurons" of a Convolutional Neural Network trained on the uncentered FashionMNIST dataset. These images were designed to trigger a strong response after one (a) or two (b) convolution+Relu operations. As we dive into deeper layers, these images have a larger support and tend to "represent" more complex features.

Trying with a deeper net

Why do we speak about "deep learning"? As everyone who actually trained these kinds of model can tell: **Deeper is better**. That is, stacking layers empirically increases accuracy and reduces overfit. As of 2017, this regularizing effect of "deep" networks is not understood – some people try to explain it through statistical physics analogies, but I don't know what it's worth. This hasn't prevented the GPU arms race to take place: researchers now routinely work with tens or hundreds of stacked convolution layers in a single model.

Deep learning: is it lasagna cooking? To illustrate on your machines this "copy-paste" philosophy which produces excellent results in most computer vision tasks, a simple four-layer architecture is presented in the notebook. Please have a look!



(a) Monitoring the training process.

(b) This is definitely a shirt!

Figure 11.6: Training a 4-layer CNN on the un-centered FashionMNIST dataset.
(a) The training process is computationaly intensive, but tends to converge towards a better classifier. The general trend is that deeper CNNs generalize best outside of their training sets.
(b) Keep in mind, though, that good performances on a given task do not magically transfer into a general **understanding** of the underlying real-world phenomenon. High-dimensional, generic machine learning algorithms tend to be very **brittle**: they cannot be expected to provide meaningful answers outside of their comfort zone, the " "convex hull" " of their training dataset.

Going further

Getting back to earth. At this point, one could be tempted to see CNNs as programs which intelligently extract the best out of any large enough database. Indeed, fascinated by the beautiful "DeepDreams" or "DeepArt" visualizations that were recently advertised in the media, people tend to fall into the **"Pygmalion trap"** of attributing to good-looking results human-like qualities. As it is a hot "newspapers" topic at the moment, here are a few papers and links I would recommend on the subject:

- CNNs do **not** see the world with human eyes: *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images*, Nguyen, Nosinski, Clune, 2015 : `www.evolvingai.org/fooling`
- **Prior to any training**, the implicit image regularizing prior encoded in a convolutional architecture + L^2 gradient descent scheme is already **extremely strong**: *Deep Image Prior*, Ulyanov, Vendaldi, Lempitsky, 2017 : `dmitryulyanov.github.io/deep_image_prior`
- The theory of non-linear wavelet transforms (aka. **scattering** operators) can be studied as a first mathematical model of deep convolutional networks: *Invariant Scattering Convolution Networks*, Bruna, Mallat, 2012 : `arxiv.org/abs/1203.1513`;
  Deep Haar Scattering Networks, Cheng, Chen, Mallat, 2015 : `arxiv.org/abs/1509.09187`.

Orthogonal references. In this course, Gabriel and myself presented to you a brief introduction to Data Science and Deep Learning from a **geometric**, **analytical** point of view. But you shouldn't take our word on it! To help you see this moving field **without mathematically-tinted glasses**, here is a list of websites which are definitely worth reading:

- The course notes of Andrej Karpathy from Stanford, `cs231n.github.io` . This is the reference introduction to the subject from the **computer science** point of view. His post on Recurrent Neural Networks (which are to hidden Markov models what CNNs are to wavelet transforms) can also help you to get a grasp of the expressivity (or lack thereof) of the models currently used in natural language processing: `karpathy.github.io/2015/05/21/rnn-effectiveness`
- The blog of Chris Olah (+ everything on Distill), `colah.github.io`. In my opinion, his statements and visualizations are sometimes a bit over-optimistic, brushing things under the carpet... But he has put *a lot* of high-quality work into producing **accessible** material on the subject. Have a look!
- The blog of Ferenc Huszár, `www.inference.vc`, with a refreshing **Bayesian + Information Theoretic** point of view on pretty much everything.

What should you take back from all this?

In my opinion, these are three main points you should remember from today's session:

The generic Neural Network problem is irrelevant. You cannot just stack up a bunch of generic linear operators and hope for the best: **if a model can express *everything*, it can also overfit *anything***. To **put prior** into their algorithms, researchers restrict themselves to carefully chosen **architectures** (submanifolds in the space of programs/transforms, if you want) and use optimizing techniques which empirically provide good generalization properties outside of the training set (i.e. little overfit).

Using an "isotropic" L^2 gradient to optimize a neural network is **very significant**, as it heavily implies that the coordinates used to represent the model (i.e. the neural weights) should be "decorrelated" and "of the same scale". In general, gradient descent **does not** converge towards the global optimum of the cost functional, or even towards a genuine critical point.
(N.B.: Heavy ball, BFGS or other order-1 minimization methods don't really change this, just like they can't automagically recover the "high frequency" dimensions that one loses when working with blurred signals.)

CNNs are great, but definitely not "intelligent". They should not be expected to perform well on non-image/audio data. You'll get a much clearer understanding of their limits if you see them as **"beefed up" wavelet transforms**.

As of today, neural networks have proved their worth in "only" two fields: **signal processing** (where CNNs succeed wavelet transforms) and **natural language processing** (where RNNs succeed hidden markov models). As the "neural net" versions of the classical models **share their algorithmic structure with their predecessors**, they are subject to the same kind of pitfalls (e.g. the structural inability of RNNs to generate purposeful sentences and paragraphs).

You can now agressively optimize the parameters of your favorite data flow. If we leave the pseudo-physiological justifications aside, what really sticks out of the current research on neural networks is the development of **automatic differentiation** frameworks. In the last few years, the MILA, Google and Facebook (mainly), have put in a considerable engineering effort to develop easy-to-use and scalable development toolboxes such as `TensorFlow` and `PyTorch`.

Now, this may seem surprising... but in my opinion, **this low-level work is the most far-reaching component of the current research effort on neural networks**. A genuine revolution for many applied fields, and not just computer vision!

Going beyond formulas. Until very recently, the only way to improve existent algorithms was to **think** about your problem very hard, try a few long shot ideas and hopefully come up with better results. But these efficient autodiff libraries have opened up a new path: the **large-scale tuning** of the thousands of parameters defined by your abstract formula/theory/computational graph. This means that as researchers, we now have the tools to leave the reassuring shores of "fully understood programs" and actually venture towards the wild world of real, non-mathematically formulated problems.

A new challenge for applied mathematicians. Understanding which parts of our "mathematical" data flows are **crucial** (translation invariance, multiscale priors...), and which part can be **"freely optimized"** (the actual filter coefficients...) is one of the major challenges that awaits researchers in the coming years.

If I had to make a far-fetched *architectural* analogy, I'd say that traditional mathematical theories are comparable to stone, steel and wood: highly structured materials that can produce lasting monuments, but require a skilled workforce and have their intrinsic limitations. Pure data on the other hand is a bit like concrete: an amorph mass which was very hard to deploy in large scale applications... Until the development of prestressed concrete in the 1920's!

Finding the right balance between expert prior knowledge and acquired datasets. The development of Python+Autodiff+GPU frameworks, which provide researchers with a simple way of **leveraging supervised datasets for their own specialized workflows**, has the potential to be a turning point in the history of many applied maths fields. As we head towards hybrid, "mathematically structured" + "data driven" models, we may be able to engage more easily with our colleagues from other fields, on top of seeing a new class of "meta" problems arise.

Now, I may be wrong... But whatever the outcome of the journey, there's exciting times ahead!

Bonus tracks

Due to the training time of the models, I highly doubt you'll be able to read those lines before the end of the workshop session... But, just in case, please find below a few extensions, with some sandbox code provided in the notebooks.

When CNNs meet wavelets. Constraining a fully-connected neural networks to be (quasi) translation-invariant, we ended up with a data flow that iterates convolutions with small filters, pointwise nonlinearities and subsampling operations. This very much looks like a non-linear wavelet transform (with complex norms at every scale), where **filters are optimized with respect to a given task instead of being chosen for their mathematical properties**. Thanks to the current hardware (GPUs) and software (efficient and easy-to-use autodiffs libraries) revolution, this kind of fine-tuning of the models' parameters (say, filter coefficients) is now a realistic task. But do we really need to optimize every single filter from scratch ?

Not always. In some cases indeed, the first filters of a typical CNN converge to wavelet-like features, which makes sense even from a physiological point of view: vision relies on edge detectors, which quotient out local illumination changes. Hence, we can "help" the training of our CNN by replacing its first layers with a hard-coded wavelet-like operator, the *scattering transform*. To toy around this idea, please use the code provided in the notebook. Beware: the underlying routines have not been optimized on CPU, and can thus be veeeeeery slow if you don't own an Nvidia GPU...

Transfer learning: using a pre-trained transform. Convolutional Neural Networks are nothing but finely tuned non-linear transforms. So why should we retrain them every time? In practice, most researchers and engineers contend themselves with a standard pre-trained network, and build custom applications on top! In this community, the re-use of neural weights is known as **Transfer learning**. You can read more about it at the following address:

`cs231n.github.io/transfer-learning`

Then, if your computer is fast enough, you can try to play around in the sandbox notebook cells :-)

SESSION 12 Introduction to optimal transport

Information Geometry and Statistical Manifolds

Let μ and ν be two *probability* measures on a reference space X. In the previous workshop sessions, we presented the **Kullback-Leibler** divergence

$$\mathrm{KL}(\mu \,||\, \nu) \;=\; \begin{cases} \int \log\left(\frac{\mathrm{d}\mu}{\mathrm{d}\nu}\right) \mathrm{d}\mu & \text{if } \mu \text{ is absolutely continuous wrt. } \nu \\ +\infty & \text{otherwise} \end{cases} \tag{12.1}$$

and shown that it can be used as a convenient discrepancy formula between the two "histograms" μ and ν. It is nonnegative, and null iff $\mu = \nu$, but is *not* symmetric: μ should be understood as an "observed distribution", and ν as a prior "model".

In his lecture dedicated to Optimal Transport, Gabriel also introduced the **Wasserstein** distance between any two (nonnegative) probability measures μ and ν defined on a **metric** space (X, d_X):

$$\mathrm{d}_W^2(\mu,\nu) \;=\; \min_{\mu \xrightarrow{\Gamma} \nu} \int_{X\times X} \mathrm{d}_X^2(x,y)\,\mathrm{d}\Gamma(x,y) \tag{12.2}$$

where admissible **transport plans** Γ are defined as nonnegative measures on the product $X \times X$ such that (with slightly abusive notations):

$$\int_X \mathrm{d}\Gamma(x,y)\,\mathrm{d}y \;=\; \mathrm{d}\mu(x) \qquad \text{and} \qquad \int_X \Gamma(x,y)\,\mathrm{d}x \;=\; \mathrm{d}\nu(y). \tag{12.3}$$

Since these two quantities are often used as "standard" data attachment terms between measures, and since both of them have strong theoretical backing, one question absolutely needs to be answered: **How do these two *discrepancies* compare with each other?** To find out, let's investigate their properties in the simple case of **Gaussian densities on the real line**.

Gaussian distributions. On the metric space $(\mathbb{R}, |\cdot|)$ endowed with the standard Lebesgue measure, one can define the **Normal Gaussian distribution** $\mathcal{N}(0,1)$ by

$$\mathcal{N}(0,1)(A) \;=\; \frac{1}{\sqrt{2\pi}} \int_A \exp(-x^2/2)\,\mathrm{d}x \quad \text{for any Lebesgue-measurable set } A. \tag{12.4}$$

Then, we define the Gaussian distribution $\mathcal{N}(\mu,\sigma^2)$ as the image of the normal measure $\mathcal{N}(0,1)$ through the *affine* reparametrization

$$f_{\mu,\sigma} : x \in \mathbb{R} \mapsto \sigma x + \mu, \tag{12.5}$$

where $\mu \in \mathbb{R}$ is the *mean* and $\sigma^2 \geqslant 0$ the *variance* of our measure. The well-known formula is that for any deviation $\sigma > 0$, one has:

$$\mathcal{N}(\mu,\sigma^2)(A) \;=\; \frac{1}{\sigma\sqrt{2\pi}} \int_A \exp(-(x-\mu)^2 \,/\, 2\,\sigma^2)\,\mathrm{d}x \quad \text{for any Lebesgue-measurable set } A. \tag{12.6}$$

Statistical manifolds. A powerful viewpoint to analyze statistical problems is to see the parameters θ of modeled distributions $p(\,\cdot\,,\theta)$ as coordinates of a **manifold** of probability distribution; as we endow the latter with the **metric structure induced by a canonical discrepancy** between probability measures, we can leverage our geometric intuition and expect non-trivial results. This is what we're about to do in the simple case of *Gaussian* models, as we link up with the course of "Statistics" provided by the the maths department!

In the following exercises, the parameter $\theta = (\mu, \sigma) \in \mathbb{R} \times \mathbb{R}_+$ is a point in the upper half-plane, and we generically denote

$$G_i \;=\; \mathcal{N}(\mu_i, \sigma_i^2) \;\in\; \mathrm{Prob}(\mathbb{R}). \tag{12.7}$$

Exercise 1: Fisher-Rao metric on the space of Gaussian distributions

1. Assuming that σ_1 and σ_2 are positive, show that

$$\mathrm{KL}(G_1 \,||\, G_2) \;=\; \frac{1}{2}\left[\log\left(\frac{\sigma_2^2}{\sigma_1^2}\right) - 1 + \left(\frac{\sigma_1}{\sigma_2}\right)^2 + \frac{(\mu_1-\mu_2)^2}{\sigma_2^2}\right]. \tag{12.8}$$

   In the general multidimensional case, where $\mu_i \in \mathbb{R}^d$ and $\Sigma_i \sim \sigma_i^2$ is a symmetric definite positive d-by-d matrix, this formula becomes:

$$\mathrm{KL}(G_1 \,||\, G_2) \;=\; \frac{1}{2}\left[\log\left(\frac{|\Sigma_2|}{|\Sigma_1|}\right) - d + \mathrm{tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_1-\mu_2)^\mathsf{T}\Sigma_2^{-1}(\mu_1-\mu_2)\right]. \tag{12.9}$$

2. The formula displayed above is not symmetric and thus cannot be used to induce a *distance* on the space of parameters – never mind a Riemannian one. However, there exists a way to "fix" the theory by defining an *infinitesimal* Riemannian metric as follow. Let (μ_0, σ_0) be a set of parameters, and let $(\delta\mu, \delta\sigma)$ be a small admissible deviation associated to a probability measure $G_{0+\delta} = \mathcal{N}(\mu_0 + \delta\mu, (\sigma_0 + \delta\sigma)^2)$. Show that

$$\mathrm{KL}(G_{0+\delta} \,||\, G_0) \;=\; \frac{\frac{1}{2}(\delta\mu)^2 + (\delta\sigma)^2}{\sigma_0^2} \;+\; o((\delta\mu, \delta\sigma)^2). \tag{12.10}$$

3. Endow the upper half-plane of parameters (μ, σ) with the **Fisher-Rao (Riemannian) information metric** defined above. Up to a change of variable $\mu \mapsto \mu/\sqrt{2}$, please recognize the **Poincaré hyperbolic metric**. Can you reach the boundary of this statistical manifold?

Exercise 2: Wasserstein metric on the space of Gaussian distributions
In the exercise above, we endowed the upper half-plane with the unique (pull-back) Riemannian metric $g_{\mathrm{FR}} : \mathbb{R} \times \mathbb{R}_+ \to S_2^{++}(\mathbb{R})$ such that

$$\mathcal{N} : \quad (\mu, \sigma) \in (\mathbb{R} \times \mathbb{R}_+, g_{\mathrm{FR}}) \quad \mapsto \quad \mathcal{N}(\mu, \sigma^2) \in (\mathrm{Prob}^\infty_{>\mathrm{Leb}}(\mathbb{R}), \text{Fisher-Rao}) \tag{12.11}$$

is a Riemannian isometry onto its image, where $\mathrm{Prob}^\infty_{>\mathrm{Leb}}(\mathbb{R})$ is the set of probability measures with smooth, positive density with respect to the Lebesgue measure on $\mathbb{R}$. Making a similar analysis, show that the "pull-back" of the **Wasserstein** metric on the set of probability measures by the embedding "Gaussian" map $\mathcal{N}$ is no one but the **Euclidean metric on** $\mathbb{R} \times \mathbb{R}_+$.
(Hint: In dimension 1, an optimal transport plan for the Wasserstein metric is monotonous. The optimal mapping from G_1 to G_2 is thus explicitly given by $f_2 \circ f_1^{-1}$.)

Exercise 3: How do you interpret the geometry of these two statistical manifolds? One can say that the Fisher-Rao metric defines a canonical *vertical* distance between densities, which is independent of the underlying parametrization or *system of coordinates*. On the other hand, the Wasserstein problem defines a canonical *horizontal* distance between measures, which does not model any kind of mass creation/destruction.
In which case would you recommend to use one over another?

As of 2017, the study of these metric on spaces of measures is a very active research topic. To go further, I would recommend the following paper: *An Interpolating Distance Between Optimal Transport and Fisher–Rao Metrics*, Chizat, Peyré, Schmitzer, Vialard (2016).

Exercise 4: How do we **compute**, in practice, the Wasserstein distance between any two discrete measures? To discover the baseline "simplex" algorithm, please go through the dedicated Numerical Tour: `optimaltransp_1_linprog.ipynb`.
Please note that thanks to the work of Jean-Charles Gilbert, the *French* Wikipedia articles on linear optimization are a great read!

Solution

Exercise 1: Using a change of variables $f^{-1}_{\mu_1,\sigma_1} : y \mapsto \frac{y-\mu_1}{\sigma_1} = x$, we find

$$\begin{aligned}
\mathrm{KL}(G_1 \,||\, G_2) &= \int \log\left(\frac{\mathrm{d}G_1}{\mathrm{d}G_2}(y)\right) \mathrm{d}G_1(y) && (12.12)\\
&= \int \log\left(\frac{\mathrm{d}G_1/\mathrm{d}\lambda(x)}{\mathrm{d}G_2/\mathrm{d}\lambda(x)}\right) \mathrm{d}\mathcal{N}(0,1)(x) && (12.13)\\
&= \int \log\left(\frac{\frac{1}{\sigma_1\sqrt{2\pi}}\exp(-x^2/2)}{\frac{1}{\sigma_2\sqrt{2\pi}}\exp(-(\sigma_1 x+\mu_1-\mu_2)^2/2\sigma_2^2)}\right) \mathrm{d}\mathcal{N}(0,1)(x) && (12.14)\\
&= \int \log\left(\frac{\sigma_2}{\sigma_1}\right) - \frac{x^2}{2} + \frac{\sigma_1^2}{2\sigma_2^2}x^2 + 2\,\frac{\sigma^1}{2\sigma_2^2}(\mu_1-\mu_2)x + \frac{(\mu_1-\mu_2)^2}{2\sigma_2^2}\,\mathrm{d}\mathcal{N}(0,1)(x) && (12.15)\\
&= \log\left(\frac{\sigma_2}{\sigma_1}\right) - \frac{1}{2} + \frac{\sigma_1^2}{2\sigma_2^2} + \frac{1}{2}\frac{(\mu_1-\mu_2)^2}{\sigma_2^2} && (12.16)\\
&= \frac{1}{2}\left[\log\left(\frac{\sigma_2^2}{\sigma_1^2}\right) - 1 + \left(\frac{\sigma_1}{\sigma_2}\right)^2 + \frac{(\mu_1-\mu_2)^2}{\sigma_2^2}\right], && (12.17)
\end{aligned}$$

since we have

$$\underbrace{\int 1 \cdot \mathrm{d}\mathcal{N}(0,1)(x)}_{\text{total mass}} = 1, \qquad \underbrace{\int x \cdot \mathrm{d}\mathcal{N}(0,1)(x)}_{\text{mean}} = 0, \qquad \underbrace{\int x^2 \cdot \mathrm{d}\mathcal{N}(0,1)(x)}_{\text{variance}} = 1. \tag{12.18}$$

Question 2: Using a Taylor expansion of $\log(1+x)$, we get

$$\begin{aligned}
\mathrm{KL}(G_{0+\delta} \,||\, G_0) &= -\log\left(1+\frac{\delta\sigma}{\sigma_0}\right) + \frac{1}{2}\left[-1 + \left(1+\frac{\delta\sigma}{\sigma_0}\right)^2 + \frac{\delta\mu^2}{\sigma_0^2}\right] && (12.19)\\
&= -\left(\frac{\delta\sigma}{\sigma_0} - \frac{1}{2}\left(\frac{\delta\sigma}{\sigma_0}\right)^2\right) + \frac{1}{2}\left[2\,\frac{\delta\sigma}{\sigma_0} + \left(\frac{\delta\sigma}{\sigma_0}\right)^2 + \frac{\delta\mu^2}{\sigma_0^2}\right] + o((\delta\mu,\delta\sigma)^2) && (12.20)\\
&= \frac{\frac{1}{2}(\delta\mu)^2 + (\delta\sigma)^2}{\sigma_0^2} + o((\delta\mu,\delta\sigma)^2) && (12.21)
\end{aligned}$$

Question 3: Denoting $\nu = \mu/\sqrt{2}$, we get

$$\mathrm{KL}(G_{0+\delta} \,||\, G_0) = \frac{(\delta\nu)^2 + (\delta\sigma)^2}{\sigma_0^2}, \tag{12.22}$$

which is the canonical expression of the Poincaré metric on the upper half-plane. For complete reference on the subject, I recommend the nice introductory paper *Hyperbolic Geometry* by J.W. Cannon et al. (1997). This standard model of hyperbolic geometry is in the baggage of most mathematicians; but for the sake of completeness, let's show that **its boundary cannot be reached**.

Starting from an arbitrary location, say $A = (0,1)$, we wish to see if a path of **finite length** can allow us to get out of the domain $\mathbb{R} \times \mathbb{R}_+^\star$. According to the reduction principle – local orthogonal decompositions between vertical (useful) and horizontal (needless) displacements – we know that the "shortest paths out" are the vertical lines (upward and downward) parametrized, for instance, by the paths

$$\gamma : t \in [0,1] \mapsto (0, 1-t) \qquad \text{and} \qquad \kappa : t \in [0,+\infty] \mapsto (0, 1+t). \tag{12.23}$$

But then, computing the Riemannian lengths, we get

$$\ell(\gamma) = \int_0^1 \|\dot\gamma(t)\|_{\gamma(t)}\,\mathrm{d}t = \int_0^1 \|(0,-1)\|_{(0,1-t)}\,\mathrm{d}t \tag{12.24}$$

$$= \int_0^1 \sqrt{\frac{\frac{1}{2}\cdot 0^2 + (-1)^2}{(1-t)^2}}\,\mathrm{d}t = \int_0^1 \frac{1}{1-t}\,\mathrm{d}t = +\infty, \tag{12.25}$$

$$\ell(\kappa) = \int_0^{+\infty} \|\dot\kappa(t)\|_{\kappa(t)}\,\mathrm{d}t = \int_0^{+\infty} \|(0,+1)\|_{(0,1+t)}\,\mathrm{d}t \tag{12.26}$$

$$= \int_0^{+\infty} \sqrt{\frac{\frac{1}{2}\cdot 0^2 + (+1)^2}{(1+t)^2}}\,\mathrm{d}t = \int_0^{+\infty} \frac{1}{1+t}\,\mathrm{d}t = +\infty. \tag{12.27}$$

Hence, **the space of Gaussian laws endowed with the Fisher-Rao metric is geodesically complete**.

Exercise 2: The only non-decreasing mapping from G_1 to G_2 is given by $f_2 \circ f_1^{-1}$. According to Brenier's characterization of Optimal Transport plans as gradients of convex potential fields, it is thus the "Optimal Transport" mapping, as the transport plan Γ is formally defined as

$$\Gamma(A) = \int_{\mathbb{R}} 1_A\left(t, f_2 \circ f_1^{-1}(t)\right)\,\mathrm{d}G_1(t) \tag{12.28}$$

$$= \int_{\mathbb{R}} 1_A\left(f_1(t), f_2(t)\right)\,\mathrm{d}\mathcal{N}(0,1)(t) \quad \text{for any Borelian set } A \subset \mathbb{R}\times\mathbb{R}. \tag{12.29}$$

The Wasserstein distance between G_1 and G_2 can thus be computed as

$$\mathrm{d}_W^2(G_1,G_2) = \mathbb{E}_{(x,y)\sim\Gamma}\left[(y-x)^2\right] = \int_{\mathbb{R}} \left[f_2(t)-f_1(t)\right]^2\,\mathrm{d}\mathcal{N}(0,1)(t) \tag{12.30}$$

$$= \int_{\mathbb{R}} \left[(\sigma_2 t+\mu_2)-(\sigma_2 t+\mu_2)\right]^2\,\mathrm{d}\mathcal{N}(0,1)(t) \tag{12.31}$$

$$= (\sigma_2-\sigma_1)^2\cdot\int t^2\cdot\,\mathrm{d}\mathcal{N}(0,1)(t) + (\mu_2-\mu_1)^2\cdot\int 1\cdot\,\mathrm{d}\mathcal{N}(0,1)(t) \tag{12.32}$$

$$+ 2\,(\sigma_2-\sigma_1)\,(\mu_2-\mu_1)\cdot\int t\cdot\,\mathrm{d}\mathcal{N}(0,1)(t) \tag{12.33}$$

$$= (\sigma_2-\sigma_1)^2 + (\mu_2-\mu_1)^2. \tag{12.34}$$

The geometry here is so simple that there is no need to speak of Riemannian infinitesimal metrics: in every possible sense, **the space of Gaussian laws endowed with the Wasserstein distance is isometric to the upper half-plane, endowed with its canonical Euclidean structure**.

Exercise 3: We're in luck! Gabriel released a nice illustration on the subject via his Twitter account `@gabrielpeyre` (which is great, by the way: have a look!) :

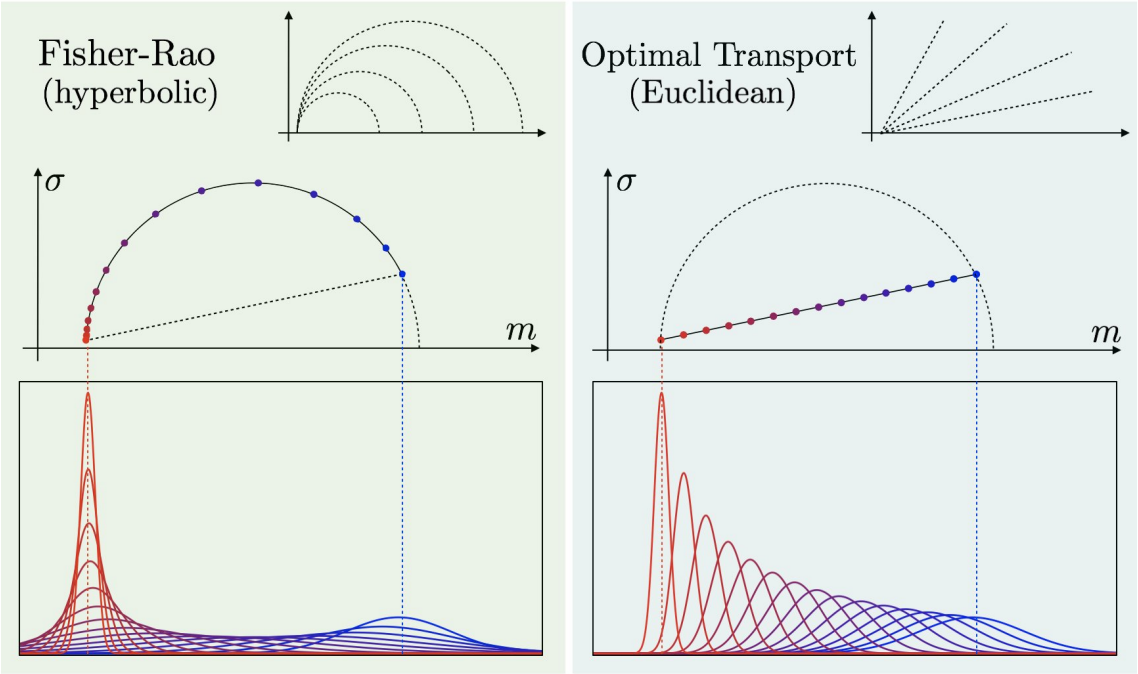


Figure 12.1: Geodesics between two Gaussian laws, at constant speed. Note that Gabriel cheated a little bit here, as he should have written "$m/\sqrt{2}$" instead of "m" to legend the Fisher-Rao plot's horizontal axis... But the nice color scheme more than makes up for it ;-)

From a practical point a view, the **major difference** between both metrics lies in the way they handle **degenerate** distributions on the horizontal axis $\sigma = 0$: while the Fisher-Rao structure pushes diracs at infinity, the Wasserstein model handles them just like other Gaussian laws... at the cost of **losing geodesic completeness**.

There is no clean way to extend a curve of Gaussian laws that reaches a dirac. In your modelling process, you should thus ask yourself if these degenerate distributions are worth the trouble of working with boundary manifolds (OT), or if you can afford to "throw them out" of your model and stay in a geodesically complete setting (Fisher-Rao) – which considerably simplifies the implementation and study of classical algorithms such as **gradient descent**, for optimization.

SESSION 13 Distances between measures

In this workshop session, we showcase the properties of several geometric divergences defined on the space of probability measures:

- **Kernel Norms** (aka. Maximum Mean Discrepancies),
- **Maximum Likelihoods of Gaussian Mixture Models** (aka. sum-Hausdorff distances),
- **Optimal Transport costs** (aka. Wasserstein or Earth-Mover's distances).

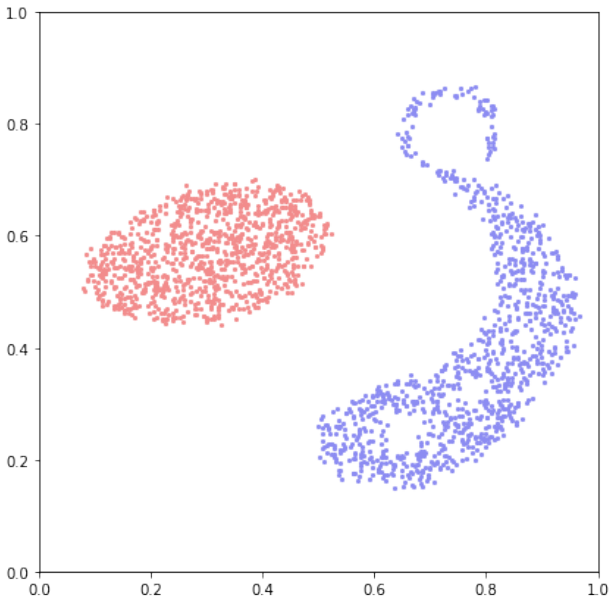
To keep things simple and allow us to assess graphically the performances of our methods, we will work with measures α and β sampled on the unit square:

$$\alpha \;=\; \sum_{i=1}^{\mathrm{N}} \alpha_i \delta_{x_i}, \qquad\qquad \beta \;=\; \sum_{j=1}^{\mathrm{M}} \beta_j \delta_{y_j}, \tag{13.1}$$

where α_i, β_j are **positive weights** associated to the **samples** x_i and y_j in $\mathbb{R}^2$. In this notebook, we will focus on the case where α and β are **probability measures**:

$$\sum_{i=1}^{\mathrm{N}} \alpha_i \;=\; 1 \;=\; \sum_{j=1}^{\mathrm{M}} \beta_j. \tag{13.2}$$

In [3]:
```
# α and β are sampled from two png densities
α_i, x_i = draw_samples("data/density_a.png", NPOINTS, dtype)
β_j, y_j = draw_samples("data/density_b.png", NPOINTS, dtype)
```



Gradient flows. This notebook is all about studying *Cost* functions that have distance-like properties on the space of probability measures. A simple way of highlighting the geometry induced by such functionals is to follow their **Wasserstein gradient flows**, i.e. to integrate the ODE

$$\dot{x}_i(t) \;=\; -\tfrac{1}{\alpha_i}\,\nabla_{x_i}\mathrm{Cost}\big(\sum_i \alpha_i\delta_{x_i(t)},\beta\big) \tag{13.3}$$

starting from an initial condition $x_i(t=0)=x_i$, performing a weighted gradient descent on the function

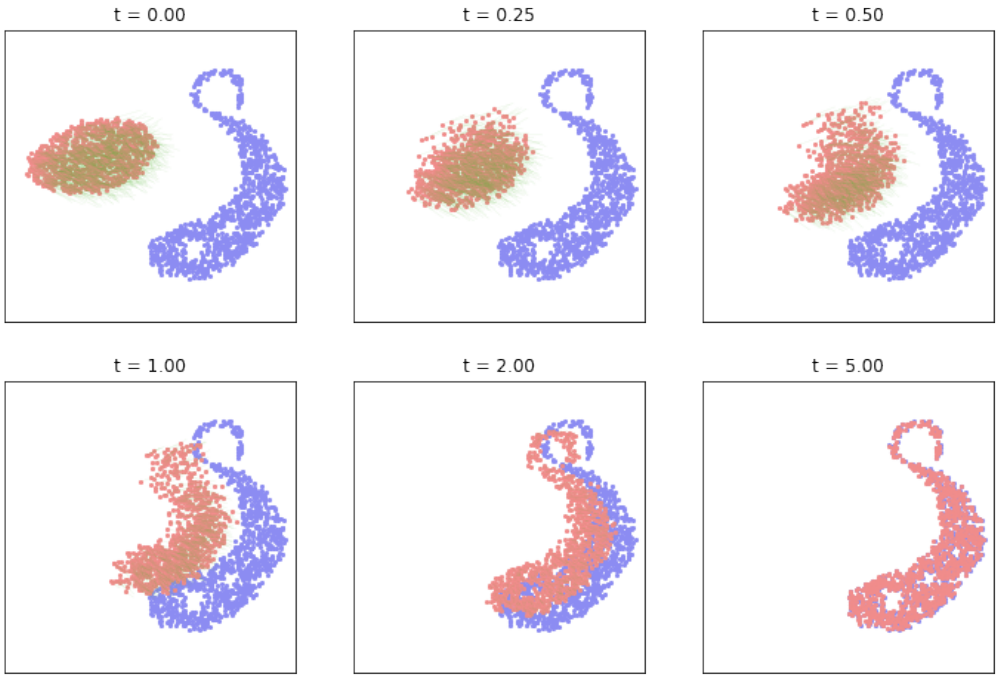
$$\mathrm{Cost}_\beta \;:\; (x_i)\in\mathbb{R}^{\mathrm{N}\cdot d} \;\mapsto\; \mathrm{Cost}\big(\sum_i \alpha_i\delta_{x_i},\beta\big). \tag{13.4}$$

```
In [5]: def gradient_flow(α_i, x_i, β_j, y_j, cost, lr=.05) :
            """
            Flows along the gradient of the cost function, using a simple Euler scheme.

            Parameters
            ----------
                α_i : (N,1) torch tensor
                    weights of the source measure
                x_i : (N,2) torch tensor
                    samples of the source measure
                β_j : (M,1) torch tensor
                    weights of the target measure
                y_j : (M,2) torch tensor
                    samples of the target measure
                cost : (α_i,x_i,β_j,y_j) -> torch float number,
                    real-valued function
                lr : float, default = .05
                    learning rate, i.e. time step
            """

            # Parameters for the gradient descent
            Nsteps = int(5/lr)+1
            t_plot      = np.linspace(-0.1, 1.1, 1000)[:,np.newaxis]
            display_its = [int(t/lr) for t in [0, .25, .50, 1., 2., 5.]]

            # Make sure that we won't modify the input measures
            α_i, x_i, β_j, y_j = α_i.clone(), x_i.clone(), \
                                 β_j.clone(), y_j.clone()

            # We're going to perform gradient descent on Cost(Alpha, Beta)
            # wrt. the positions x_i of the diracs masses that make up Alpha:
            x_i.requires_grad_(True)

            plt.figure(figsize=(12,8)) ; k = 1
            for i in range(Nsteps): # Euler scheme ===============
                # Compute cost and gradient
                loss = cost(α_i, x_i, β_j, y_j)
                [g]  = torch.autograd.grad(loss, [x_i])
                # in-place modification of the tensor's values
                x_i.data -= lr * (g / α_i)
```

This evolution can be understood as an ideal, *model-free* machine learning problem where a source distribution α_t is iteratively fitted towards a target (empirical) distribution.

Let us now display the evolution associated to the quadratic spring energy between **labeled** point clouds.

```
In [6]: def L2_cost(α_i, x_i, β_j, y_j) :
            """
            Simplistic L2 cost (aka. spring energy) between sampled point clouds,
            assuming a pairwise correspondence between x_i[k] and y_j[k].
            """
            return .5*(α_i*((x_i-y_j)**2).sum(1,keepdim=True)).sum()

        gradient_flow(α_i, x_i, β_j, y_j, L2_cost)
```



It works! Now, let's move on to costs that are well-defined between **unlabeled** point clouds with, possibly, different weights and numbers of samples.

A computational building block: the kernel product

Most standard costs between sampled measures can be computed using a **kernel product operator**

$$\mathrm{KP}: \big((x_i),(y_j),(\beta_j)\big) \in \mathbb{R}^{\mathrm{N}\cdot d} \times \mathbb{R}^{\mathrm{M}\cdot d} \times \mathbb{R}^{\mathrm{M}\cdot 1} \quad \mapsto \quad \Big(\sum_j k(x_i - y_j)\, \beta_j \Big)_i \in \mathbb{R}^{\mathrm{N}\cdot 1}$$

where $k : \mathbb{R}^d \to \mathbb{R}$ is a convolution kernel. Mathematically, this operation is known as a **discrete convolution**: Indeed, if $\beta = \sum_j \beta_j \delta_{y_j}$ is a discrete measure, the convolution product $k \star \beta$ is a function defined on $\mathbb{R}^d$ by

$$\big(k \star \beta\big)(x) \;=\; \sum_j k(x - y_j)\, \beta_j,$$

so that computing the kernel product $\mathrm{KP}\big((x_i), (y_j), (\beta_j)\big)$ is equivalent to computing and sampling $k \star \beta$ on the point cloud (x_i).

```
In [7]: def KP(x,y,β_j, kernel = "gaussian", s = 1.) :
            """
            Computes K(x_i,y_j) @ β_j = \sum_j k(x_i-y_j) * β_j
            where k is a kernel function (say, a Gaussian) of deviation s.
            """
            x_i = x[:,None,:]  # Shape (N,d) -> Shape (N,1,d)
            y_j = y[None,:,:]  # Shape (M,d) -> Shape (1,M,d)
            xmy = x_i - y_j    # (N,M,d) matrix, xmy[i,j,k] = (x_i[k]-y_j[k])
            if   kernel == "gaussian" : K = torch.exp( - (xmy**2).sum(2) / (2*(s**2)) )
            elif kernel == "laplace"  : K = torch.exp( - xmy.norm(dim=2) / s )
            elif kernel == "energy"   : K = - xmy.norm(dim=2)
            return K @ β_j.view(-1,1) # Matrix-vector product
```

Using a kernel norm

Total Variation: a first dual norm. Now, which cost function $\mathrm{Cost}(\alpha_t, \beta)$ are we going to choose to drive our simple optimization routine? Given two measures α and β on $\mathbb{R}^d$, one of the simplest distance that can be defined is the Total Variation

$$\mathrm{d_{TV}}(\alpha, \beta) \;=\; \|\alpha - \beta\|_\infty^\star \;=\; \sup_{\|f\|_\infty \leqslant 1} \int f \mathrm{d}\alpha - \int f \mathrm{d}\beta,$$

using the dual norm on $L^\infty(\mathbb{R}^d, \mathbb{R})$. Unfortunately, this formula is not suited *at all* to sampled, discrete probability measures with non-overlapping support: If $\alpha = \sum_i \alpha_i\, \delta_{x_i}$ and $\beta = \sum_j \beta_j\, \delta_{y_j}$ with $\{x_i, \dots\} \cap \{y_j, \dots\} = \emptyset$, one can simply choose a function f such that

$$\forall\, i,\; f(x_i) \;=\; +1 \quad \text{and} \quad \forall\, j,\; f(y_j) \;=\; -1$$

to show that

$$\mathrm{d_{TV}}(\alpha, \beta) \;=\; |\alpha| + |\beta| \;=\; 2 \quad \text{as soon as } \mathrm{supp}(\alpha) \text{ and } \mathrm{supp}(\beta) \text{ do not overlap.}$$

The gradient of the Total Variation distance between two sampled measures is thus completely uninformative, being zero for almost all configurations.

Smoothing measures to create overlap. How can we fix this problem? An idea would be to choose a **blurring function** g, and compare the blurred functions $g \star \alpha$ and $g \star \beta$ by using, say, an L^2 norm:

$$\mathrm{d}(\alpha,\beta) \;=\; \|g\star(\alpha-\beta)\|_2^2 \;=\; \langle g\star(\alpha-\beta), g\star(\alpha-\beta)\rangle_2.$$

But then, if we define $k=\tilde{g}\star g$, where $\tilde{g}=g\circ(x\mapsto -x)$ is the mirrored blurring function, one gets

$$\mathrm{d}_k(\alpha,\beta) \;=\; \langle g\star(\alpha-\beta), g\star(\alpha-\beta)\rangle_2 \;=\; \langle \alpha-\beta, k\star(\alpha-\beta)\rangle \;=\; \|\alpha-\beta\|_k^2.$$

Assuming a few properties on k (detailed below), d_k is the quadratic norm associated with the k-scalar product between measures:

$$\langle \alpha,\beta\rangle_k \;=\; \langle \alpha, k\star\beta\rangle.$$

More specifically,

$$\left\langle \sum_i \alpha_i\,\delta_{x_i}, \sum_j \beta_j\,\delta_{y_j}\right\rangle_k \;=\; \left\langle \sum_i \alpha_i\,\delta_{x_i}, \sum_j \beta_j\,\big(k\star\delta_{y_j}\big)\right\rangle \tag{13.5}$$

$$= \left\langle \sum_i \alpha_i\,\delta_{x_i}, \sum_j \beta_j\,k(\,\cdot\,-y_j)\right\rangle \;=\; \sum_{i,j} k(x_i-y_j)\,\alpha_i\beta_j. \tag{13.6}$$

```
In [8]: # PyTorch syntax for the L2 scalar product...
        def scal(α, f) :
            return torch.dot(α.view(-1), f.view(-1))

        def kernel_scalar_product(α_i, x_i, β_j, y_j, mode = "gaussian", s = 1.) :
            Kxy_β = KP(x_i,y_j,β_j,mode,s)
            return scal( α_i, Kxy_β )
```

Having defined the scalar product, we then simply develop by bilinearity:

$$\tfrac{1}{2}\|\alpha-\beta\|_k^2 \;=\; \tfrac{1}{2}\langle\alpha,\alpha\rangle_k \;-\; \langle\alpha,\beta\rangle_k \;+\; \tfrac{1}{2}\langle\beta,\beta\rangle_k.$$

```
In [9]: def kernel_distance(mode = "gaussian", s = 1.) :
            def cost(α_i, x_i, β_j, y_j) :
                D2 =   (.5*kernel_scalar_product(α_i, x_i, α_i, x_i, mode,s) \
                       +.5*kernel_scalar_product(β_j, y_j, β_j, y_j, mode,s) \
                       -   kernel_scalar_product(α_i, x_i, β_j, y_j, mode,s) )
                return D2
            return cost
```

This formula looks good: points **interact** with each other as soon as $k(x_i,y_j)$ is non-negligible. But if we want to get a genuine norm between measures, which hypotheses should we make on k?

This question was studied by mathematicians from the first half of the 20th century who developed the theory of Reproducing Kernel Hilbert Spaces - RKHS. In our specific translation-invariant case (in which we "hardcode" convolutions), the results can be summed up as follow:

– Principled kernel norms are the ones associated to **kernel functions** k whose Fourier transform is *real-valued* and *positive* - think, Gaussian kernels:

$$\forall\,\omega\in\mathbb{R}^d,\;\; \widehat{k}(\omega) > 0.$$

– For any such kernel function, there exists a unique blurring kernel function g such that $g \star g = k$: Simply choose

$$\widehat{g}(\omega) \;=\; \sqrt{\widehat{k}(\omega)}.$$

– These kernels define a **Hilbert norm** on a subset of $L^2(\mathbb{R}^d)$:

$$\|f\|_V^2 \;=\; \int_{\omega\in\mathbb{R}^d} \frac{|\widehat{f}(\omega)|^2}{\widehat{k}(\omega)}\,\mathrm{d}\omega \;=\; \langle k^{(-1)} \star f\,,\, f\rangle$$

where $k^{(-1)}$ is the deconvolution kernel associated to k. If we define

$$V \;=\; \big\{f \in L^2(\mathbb{R}^d),\ \|f\|_V < \infty\big\},$$

then $(V, \|\cdot\|_V)$ is a Hilbert space of functions endowed with the scalar product

$$\langle f\,,\, g\rangle_V \;=\; \int_{\omega\in\mathbb{R}^d} \frac{\overline{\widehat{f}(\omega)}\,\widehat{g}(\omega)}{\widehat{k}(\omega)}\,\mathrm{d}\omega \;=\; \langle k^{(-1)} \star f\,,\, g\rangle.$$

– **We focus on kernel functions such that for all points $x \in \mathbb{R}^d$, the evaluation at point x is a continuous linear form on V.** That is,

$$\delta_x : f \in (V, \|\cdot\|_V) \mapsto f(x) \in (\mathbb{R}, |\cdot|)$$

is well-defined and continuous. A sufficient condition for this is to ask that $\widehat{k} \in L^1(\mathbb{R}^d)$ and continuous. Then, we show that the Riesz theorem identifies δ_x with the continuous function $k \star \delta_x : y \mapsto k(y - x)$:

$$\forall\, f \in V,\quad f(x) \;=\; \langle \delta_x\,,\, f\rangle \;=\; \langle k \star \delta_x\,,\, f\rangle_V.$$

– **Finite sampled measures can thus be identified with linear forms on V. The k-norm is nothing but the dual norm of $\|\cdot\|_V$:**

$$\forall\, \alpha \in V^\star,\ \|\alpha\|_k \;=\; \sqrt{\langle \alpha\,,\, k \star \alpha\rangle} \;=\; \sup_{\|f\|_V = 1} \langle \alpha\,,\, f\rangle.$$

All-in-all, **just like the TV distance, the kernel distance can be seen as the dual of a norm on a space of functions**. Whereas TV was associated to the infinity norm $\|\cdot\|_\infty$ on $L^\infty(\mathbb{R}^d)$, the kernel formulas are linked to Sobolev-like norms $\|\cdot\|_V$ on spaces of k-smooth functions, denoted by the letter V.

Exercise 1: Using the method of Lagrange multipliers (aka. *théorème des extrema liés* in the French curriculum), show the last equality above (kernel norms are dual norms on Hilbert spaces of functions).

Solution 1: We are optimizing the linear form $f \mapsto \langle \alpha, f\rangle$ on the unit V-sphere, which is a level set of the function

$$R(f) \;=\; \|f\|_V^2 \;=\; \langle f, k^{(-1)} \star f \rangle, \quad \text{with gradient} \quad \nabla R(f) \;=\; 2 \cdot k^{(-1)} \star f. \tag{13.7}$$

At the optimum, we thus get a constant $\lambda \in \mathbb{R}$ such that

$$\alpha \;=\; 2\lambda \cdot k^{(-1)} \star f \qquad \text{i.e.} \qquad f \;=\; \underbrace{\tfrac{1}{2\lambda}}_{\mu}\, k \star \alpha. \tag{13.8}$$

Then, the equation "$\langle f, k^{(-1)} \star f \rangle = 1$" gives $\mu = 1/\sqrt{\langle \alpha, k \star \alpha \rangle}$ and finally

$$\langle \alpha, f \rangle \;=\; \mu \, \langle \alpha, k \star \alpha \rangle \;=\; \sqrt{\langle \alpha, k \star \alpha \rangle}. \tag{13.9}$$

Exercise 2: Why can we say that RKHS generalize high-order Sobolev spaces? In dimension 1, what is the functional space associated to the Laplace kernel

$$k(x,y) \;=\; e^{-\|x-y\|} \;\; ? \tag{13.10}$$

Solution 2: H^s Sobolev norms are defined through

$$\begin{aligned}
\|f\|_{H^s}^2 \;&=\; \|f\|_{L^2}^2 \;+\; \|f'\|_{L^2}^2 \;+\; \cdots \;+\; \|f^{(s)}\|_{L^2}^2 && (13.11)\\
&=\; \|\widehat{f}\|_{L^2}^2 \;+\; \|\widehat{f'}\|_{L^2}^2 \;+\; \cdots \;+\; \|\widehat{f^{(s)}}\|_{L^2}^2 && (13.12)\\
&=\; \int_\omega (1 + |\omega|^2 + \cdots + |\omega|^{2s})\, |\widehat{f}(\omega)|^2 \, \mathrm{d}\omega && (13.13)\\
&=\; \langle f, k_s^{(-1)} \star f \rangle, && (13.14)
\end{aligned}$$

with $\widehat{k_s}(\omega) \;=\; 1/(1 + |\omega|^2 + \cdots + |\omega|^{2s})$. Kernel norms allow us to generalize this construction to arbitrary (non-rational) spectral profiles, such as that of the Gaussian kernel. Going further, we could even consider kernels which are **not translation-invariant**, leaving the comfort of Fourier analysis to handle realistic, inhomogeneous situations.

On a side note: in dimension 1, since the Fourier transform of $x \mapsto e^{-|x|}$ is given by $\omega \mapsto 1/(1+\omega^2)$ up to a constant multiplicative factor, we can identify the RKHS associated to this kernel with the classic Sobolev space H^{-1}, dual of the space H^1 of square-integrable functions with square-integrable derivative.

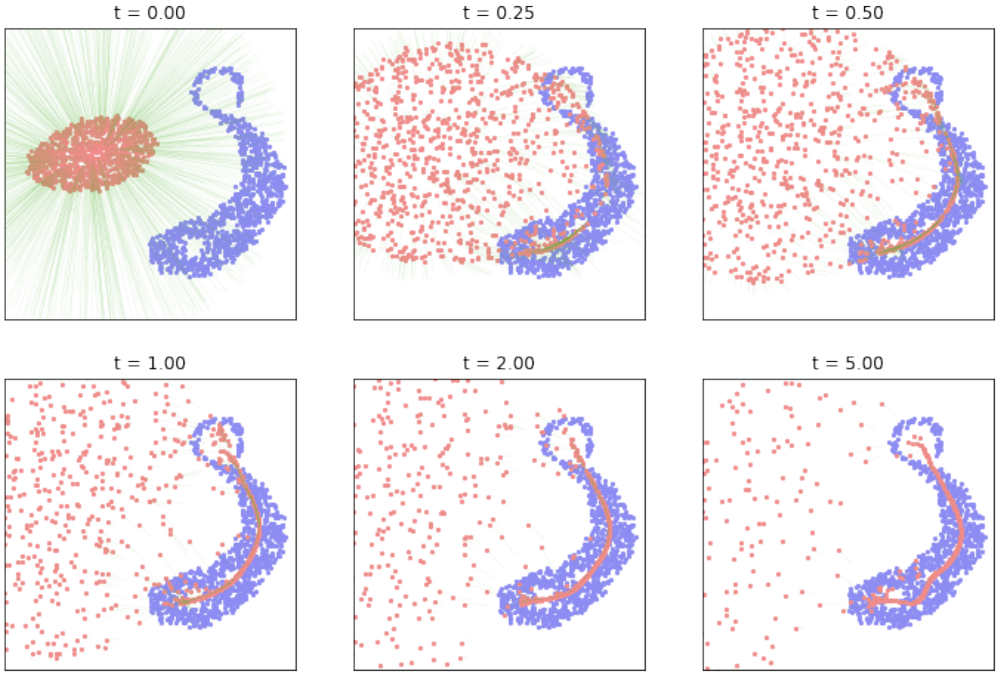
Exercise 3: What can you say about the Energy Distance kernel

$$k(x,y) \;=\; -\|x-y\| \;\; ? \tag{13.15}$$

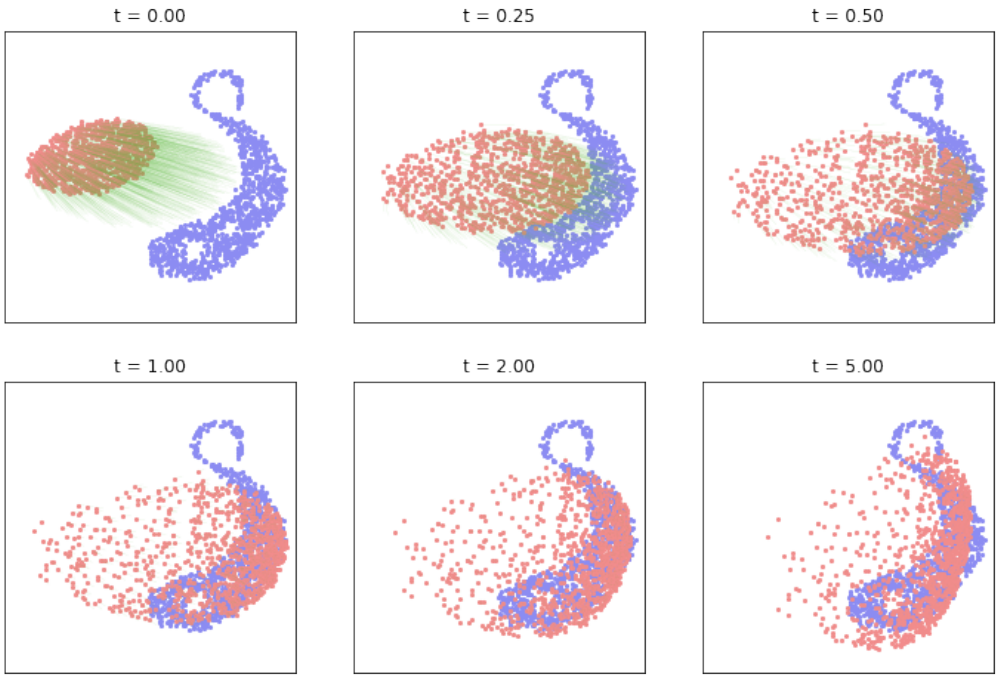
Does it satisfy the hypotheses above?

Solution 3: In dimension 1, the Fourier transform of $x \mapsto -|x|$ is given by an improper integral, $\omega \mapsto 1/\omega^2$. Consequently, it lies a bit outside of the simple theory of **positive definite kernels**: we can only say that it defines a **conditionally** positive definite kernel, and a meaningful norm between measures which have **the same mass** - thus avoiding the problem of evaluating the Fourier transform of $k \star (\alpha - \beta)$ at $\omega = 0$.

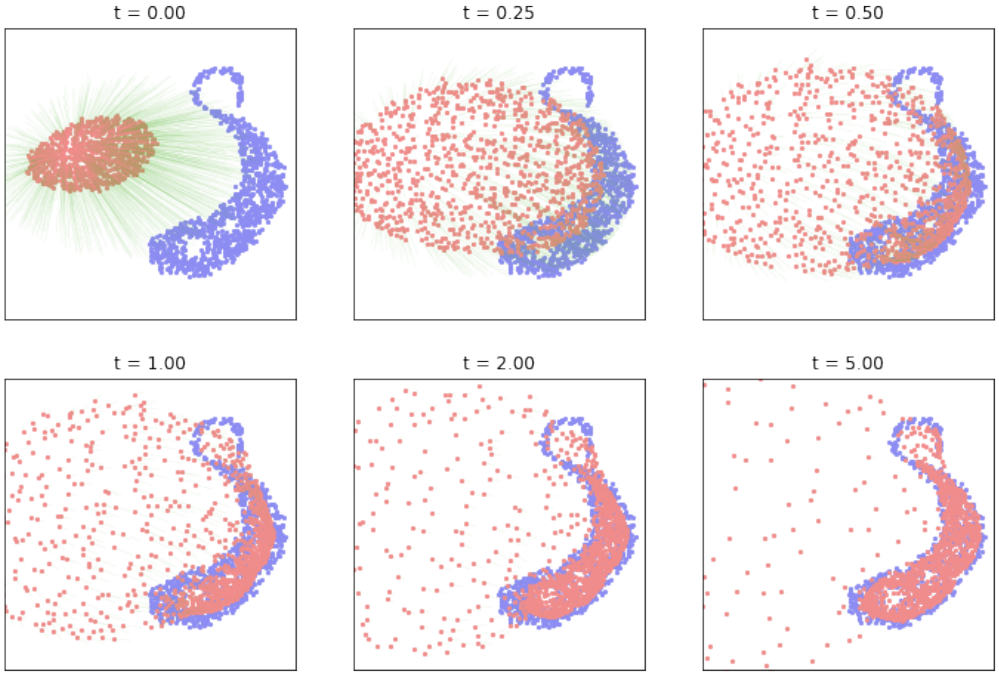
```
In [10]: gradient_flow(α_i, x_i, β_j, y_j, kernel_distance("gaussian", s=.1) )
```



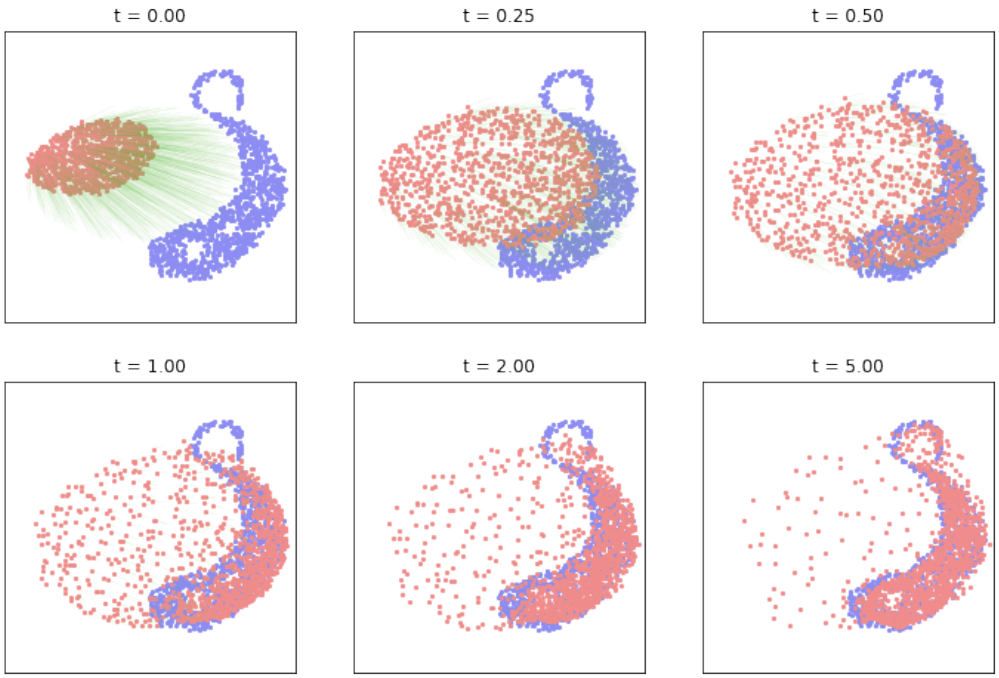

```
In [11]: gradient_flow(α_i, x_i, β_j, y_j, kernel_distance("gaussian", s=.5) )
```



```
In [12]: gradient_flow(α_i, x_i, β_j, y_j, kernel_distance("laplace", s=.5) )
```




```
In [13]: gradient_flow(α_i, x_i, β_j, y_j, kernel_distance("energy") )
```



Exercise 4: Compare the behaviours of these kernel norms for different formulas and scales. Discuss.

Solution 4: Two parameters influence the final results: - the spectral bandwidth of the kernel function k; if k is smooth, $\widehat{k}(\omega)$ converges towards 0 at infinity and the kernel norm becomes blind to high frequencies; we only register α *roughly* onto β - the spatial support of the kernel function; is k is too narrow, the x_i stop interacting with the y_j's and simply spread out to minimize the auto-correlation term $\langle \alpha, k \star \alpha \rangle$.

Noticeably, we observe a **screening effect**: some particles feel a very low gradient and only converge slooooowly towards β. This is best explained with the vocabulary of electrostatics: here, the x_i's are particles with charge $+\alpha_i$, the y_j's have a negative charge $-\beta_j$, the kernel function represents the interaction potential and the kernel norm is the total energy of the system.

The total force on a given particle x_i is then given as the sum of a **repulsion term** from the other x_i's, and the **attraction** towards the y_j's. Since particles on the left end of α are repulsed nearly as much as they are attracted, they only move slowly towards the target.

In practice, to match sampled measures, we tend to choose kernel functions with: - A null derivative at zero, and a large enough "blurring radius" to prevent overfit on the precise sampled locations of diracs. - A heavy tail, to prevent isolated parts of α_t and β from being "forgotten" by the gradient.

Using a Maximum-likelihood estimator

In the previous section, we've seen how to compare measures by seeing them as linear forms on a Sobolev-like space of functions. An other idea would be to **see the measure β as the realization of an i.i.d. sampling according to the measure α, with likelihood**

$$\text{Likelihood}_\alpha(\beta) \;=\; \text{Likelihood}_\alpha\Big(\sum_j \beta_j \delta_{y_j}\Big) \;=\; \prod_j \text{Likelihood}_\alpha(y_j)^{\beta_j}\,.$$

But which value could we attribute to the "likelihood of drawing y, given the measure α"? Since α is discrete, supported by the x_i's, interpreting it as a density wouldn't be practical at all... Thankfully, there's a simple solution: we could **convolve** α with a simple density function $k > 0$ of mass 1 - say, a Gaussian - and thus end up with a probability measure $k \star \alpha$ which is absolutely continous wrt. the Lebesgue measure, with density

$$\text{Likelihood}_{k\star\alpha}(y) \;=\; \sum_i k(y - x_i)\,\alpha_i > 0 \quad \text{for all } y \in \mathbb{R}^d.$$

From a probabilistic point of view, using this density function as a "model" is equivalent to assuming that the random variable y is generated as a sum $x + w$, where x and w are two independent variables of laws equal to α and $k \cdot \text{Lebesgue}(\mathbb{R}^d)$. If k is a Gaussian function, we speak of **Gaussian Mixture Models**.

Given α, β and a symmetric kernel function k, we can then choose to **maximize the likelihood**

$$\text{Likelihood}_{k\star\alpha}(\beta) \;=\; \prod_j \big(\sum_i k(x_i - y_j)\,\alpha_i\big)^{\beta_j},$$

i.e. to **minimize the negative log-likelihood**

$$\mathrm{d}_{\mathrm{ML},k}(\alpha,\beta) \;=\; -\sum_j \log\big(\sum_i k(x_i - y_j)\,\alpha_i\big)\,\beta_j.$$

Information theoretic interpretation. Before going any further, we wish to stress the link between maximum-likelihood estimators and the Kullback-Leibler divergence. In fact, **if we assume that a measure $\beta_{\mathbf{gen}}$ is absolutely continuous wrt. the Lebesgue measure λ,** then

$$\begin{aligned}\mathrm{H}(\beta_{\mathrm{gen}}\,|\,\alpha) \;&=\; \int \log\left(\frac{\mathrm{d}\beta_{\mathrm{gen}}}{\mathrm{d}\alpha}\right)\mathrm{d}\beta_{\mathrm{gen}} \;=\; \int \log\left(\frac{\mathrm{d}\beta_{\mathrm{gen}}/\mathrm{d}\lambda}{\mathrm{d}\alpha/\mathrm{d}\lambda}\right)\mathrm{d}\beta_{\mathrm{gen}} && (13.16)\\ &=\; \int \log\left(\frac{\mathrm{d}\beta_{\mathrm{gen}}}{\mathrm{d}\lambda}\right)\mathrm{d}\beta_{\mathrm{gen}} \;-\; \int \log\left(\frac{\mathrm{d}\alpha}{\mathrm{d}\lambda}\right)\mathrm{d}\beta_{\mathrm{gen}} && (13.17)\end{aligned}$$

$$\text{i.e.}\quad \mathrm{H}(\beta_{\mathrm{gen}}\,|\,\alpha) \;=\; \mathrm{H}(\beta_{\mathrm{gen}}\,|\,\lambda) \;-\; \int \log\left(\frac{\mathrm{d}\alpha}{\mathrm{d}\lambda}\right)\mathrm{d}\beta_{\mathrm{gen}} \qquad (13.18)$$

$$\text{so that}\quad -\int \log\left(\frac{\mathrm{d}\alpha}{\mathrm{d}\lambda}\right)\mathrm{d}\beta_{\mathrm{gen}} \;=\; \mathrm{H}(\beta_{\mathrm{gen}}\,|\,\alpha) - \mathrm{H}(\beta_{\mathrm{gen}}\,|\,\lambda). \qquad (13.19)$$

Hence, as the sampled measure β weakly converges towards a measure β_{gen},

$$d_{\mathrm{ML},k}(\alpha,\beta) \;\longrightarrow\; \mathrm{H}(\beta_{\mathrm{gen}}\,|\,k\star\alpha) \;-\; \mathrm{H}(\beta_{\mathrm{gen}}\,|\,\lambda).$$

As a function of α, this formula is minimized if and only if $\quad k\star\alpha = \beta_{\mathrm{gen}}$.

Practical implementation. As noted by the careful reader, the maximum-likelihood cost $\mathrm{d}_{\mathrm{ML},k}(\alpha,\beta)$ can be computed as the scalar product between the vector of weights (β_j) and the pointwise logarithm of the **Kernel Product** $\mathrm{KP}\big((y_j),(x_i),(\alpha_i)\big)$ - up to a negative sign. So, is using our `KP` routine a sensible thing to do? **No, it isn't.**

Indeed, if a point y_j is far away from the support $\{x_i,\dots\}$ of the measure α, $\sum_i k(x_i - y_j)\,\alpha_i$ can be prohibitively small. Just remember how fast a Gaussian kernel decreases to zero! If this sum's order of magnitude is close to the floating point precision (for `float32` encoding, around $10^{-7} \simeq e^{-4^2}$), applying to it a logarithmic function is just asking for trouble.

Additive v. Multiplicative formulas. In the previous section, we defined the **kernel distance** d_k and never encountered any accuracy problem. This is because, as far as **sums** are concerned, small "kernel's tail" values can be safely discarded - providing a reasonable balance in the weights' distribution. However, when using maximum likelihood estimation, all the values are **multiplicated with each other**: the smaller ones cannot be "neglected" anymore, as they very much determine the magnitude of the whole product. In the log-domain, near-zero values of the density $\big(k\star\alpha\big)(y_j)$ have a large influence on the final result!

Numerical stabilization. We now understand the importance of **magnitude-independent schemes** as far as multiplicative formulas are concerned. Programs which do not spiral out of control when applied to values of the order of 10^{-100}. How do we achieve such robustness? For arbitrary expressions, the only solution may be to increase the memory footprint of floating-point numbers...

But in this specific "Kernel Product" case, a simple trick will do wonders: using a **robust log-sum-exp expression**. Let's write

$$U_i \;=\; \log(\alpha_i), \qquad C_{i,j} \;=\; \log\big(k(x_i - y_j)\big) \quad \text{(given as a stable explicit formula)}.$$

Then, the log-term in the ML distance can be written as

$$\log\big(k \star \alpha\big)(y_j) \;=\; \log\bigg(\sum_i k(x_i - y_j)\,\alpha_i\bigg) \;=\; \log\bigg(\sum_i \exp\big(C_{i,j} + U_i\big)\bigg).$$

This expression lets us see that **the order of magnitude of $\big(k \star \alpha\big)(y_j)$ can be factored out easily**. Simply compute

$$M_j \;=\; \max_i C_{i,j} + U_i, \quad \text{and remark that} \quad \log\big(k \star \alpha\big)(y_j) \;=\; M_j + \log\bigg(\sum_i \exp\big(C_{i,j} + U_i - M_j\big)\bigg).$$

As the major exponent has been pulled out of the sum, we have effectively solved our accuracy problems. In practice, we can simply use the `.logsumexp()` reduction provided by recent versions of the PyTorch library.

```
In [14]: def KP_log(x,y,β_j_log, p = 2, blur = 1.) :
             x_i = x[:,None,:]  # Shape (N,d) -> Shape (N,1,d)
             y_j = y[None,:,:]  # Shape (M,d) -> Shape (1,M,d)
             xmy = x_i - y_j    # (N,M,d) matrix, xmy[i,j,k] = (x_i[k]-y_j[k])
             if   p==2 : C =  - (xmy**2).sum(2) / (2*(blur**2))
             elif p==1 : C =  - xmy.norm(dim=2) / blur
             return (blur**p)*(C + β_j_log.view(1,-1)).logsumexp(1,keepdim=True)

In [15]: def kernel_neglog_likelihood(p=2, blur = 1.) :
             def cost(α_i, x_i, β_j, y_j) :
                 loglikelihoods = KP_log(y_j, x_i, α_i.log(), p, blur)
                 dAB            = -scal(β_j, loglikelihoods)
                 return dAB
             return cost
```

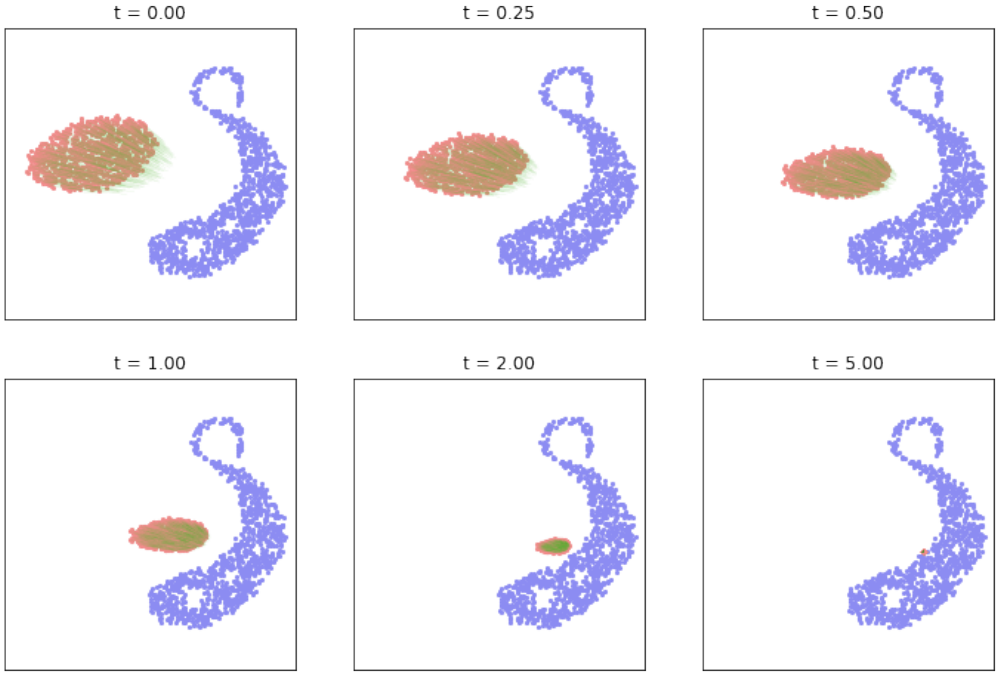
Exercise 5: Why did we put a multiplicative factor `blur**p` in the definition of `KP_log`? What influence does it have on the gradient flow?

Solution 5: Denoting the `blur` by the standard letter σ, we've defined `KP_log` as a sampler of a function f such that

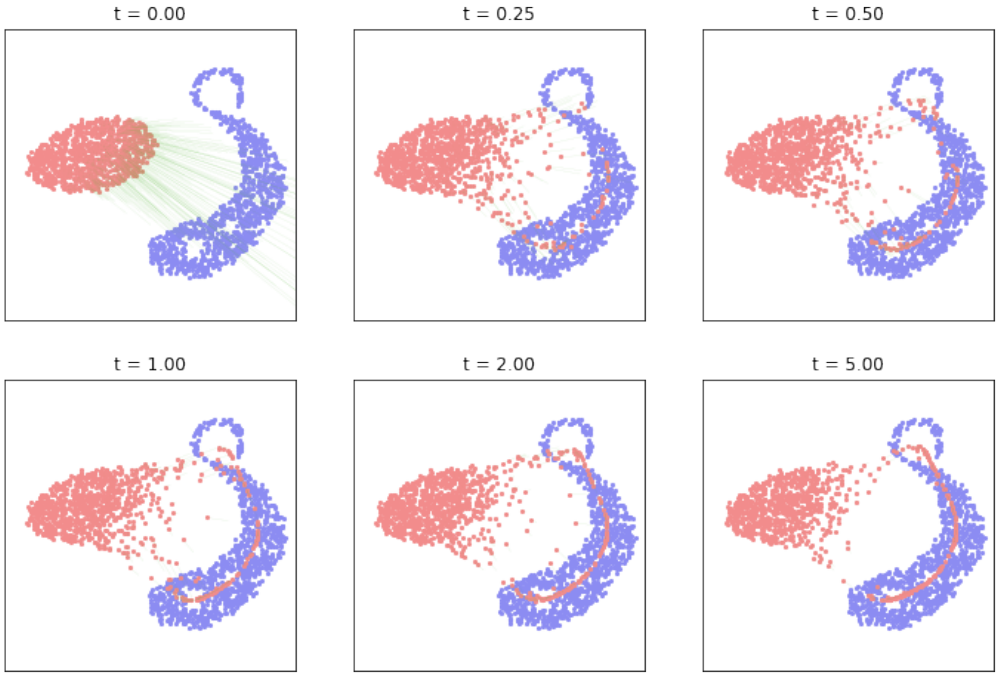
$$f(x) \;=\; \sigma^p \,\log \int_y \exp\!\big(-\tfrac{\|x-y\|^p}{p\,\sigma^p}\big)\,\mathrm{d}\beta(y) \tag{13.20}$$

Since β is a probability measure, $f(x)$ thus scales as $\frac{1}{p}\|x-y\|^p$ away from $y \sim \beta$; its gradient will then scale nicely for all values of σ, and we will get comparable flow dynamics.

```
In [16]: gradient_flow(α_i, x_i, β_j, y_j, kernel_neglog_likelihood(p=2, blur=.5) )
```




```
In [17]: gradient_flow(α_i, x_i, β_j, y_j, kernel_neglog_likelihood(p=2, blur=.1) )
```



Exercise 6: How would you describe the behaviour of this *Cost* functional? In the blur $\to 0$ limit, which simple formula do you recognize? What about the blur $\to +\infty$ limit? Can you explain the **mode collapse** observed for large values of the blurring parameter?

Solution 6: As implemented above,

$$\mathrm{d}_{\mathrm{ML},k}(\alpha,\beta) \;=\; \langle\, \beta(y)\,,\, -\sigma^p \log \int \exp\big(-\tfrac{\|x-y\|^p}{p\sigma^p}\big)\,\mathrm{d}\alpha(x)\,\rangle, \tag{13.21}$$

which can be rewritten as

$$\langle\, \beta(y)\,,\, \min_{x\sim\alpha,\sigma^p} \tfrac{1}{p}\|x-y\|^p\,\rangle, \tag{13.22}$$

where the SoftMin operator $\min_\varepsilon$ is defined through

$$\min_{x\sim\alpha,\varepsilon} \varphi(x) \;=\; -\varepsilon \log \int \exp\big(-\varphi(x)/\varepsilon\big)\mathrm{d}\alpha(x) \tag{13.23}$$

$$\xrightarrow{\epsilon\to\,0} \quad \min_{x\in\mathrm{supp}(\alpha)} \varphi(x) \tag{13.24}$$

$$\xrightarrow{\epsilon\to+\infty} \quad \int \varphi\,\mathrm{d}\alpha. \tag{13.25}$$

As we recognize a **smooth interpolation between the min and the sum reduction**, we can now make sense of the behavior of the GMM-MaxLikelihood functional:

1. When $\sigma \to 0$,

$$\mathrm{d}_{\mathrm{ML},k}(\alpha,\beta) \;\to\; \tfrac{1}{p}\langle\, \beta(y)\,,\, \min_{i=1..\mathrm{N}} \|y-x_i\|^p\,\rangle \tag{13.26}$$

   which can be understood as a **sum-Hausdorff** loss that is only interested in putting **some** points x_i in the neighborhood of β.
2. When $\sigma \to +\infty$,

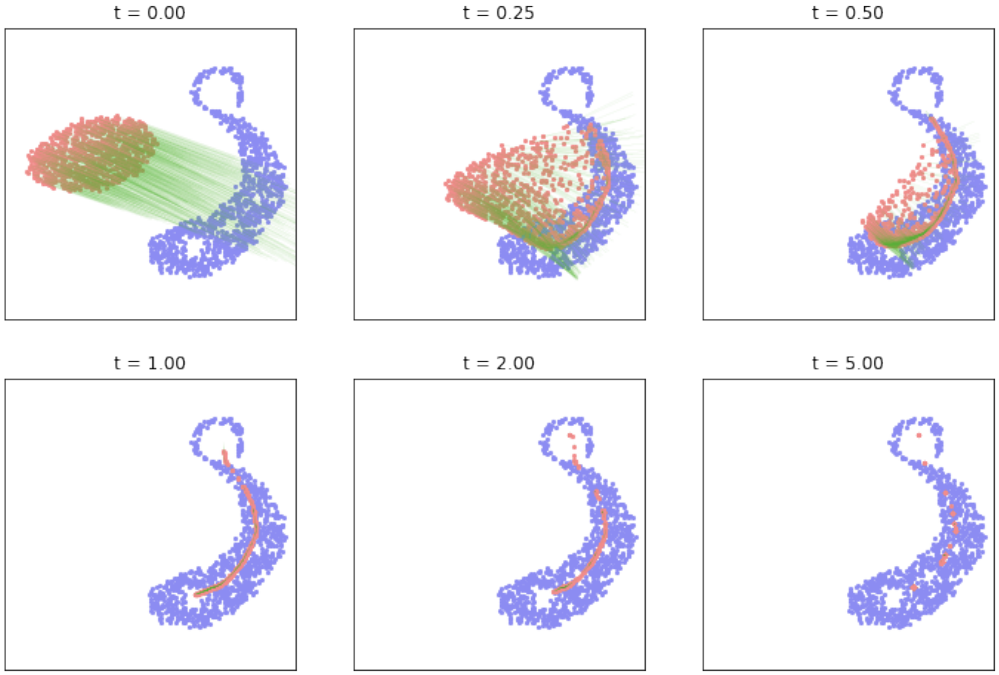
$$\mathrm{d}_{\mathrm{ML},k}(\alpha,\beta) \;\to\; \iint \tfrac{1}{p}\|y-x\|^p\,\mathrm{d}\alpha(x)\,\mathrm{d}\beta(y) \;=\; \langle\, \beta\,,\, \tfrac{1}{p}\|\,\cdot\,\|^p \star \alpha\,\rangle, \tag{13.27}$$

   which is minimized when α is a **Dirac atom located at the median (p=1) or mean (p=2) value of the target** β.

Exercise 7: One could be tempted to symmetrize the maximum-likelihood cost, as implemented below. Discuss.

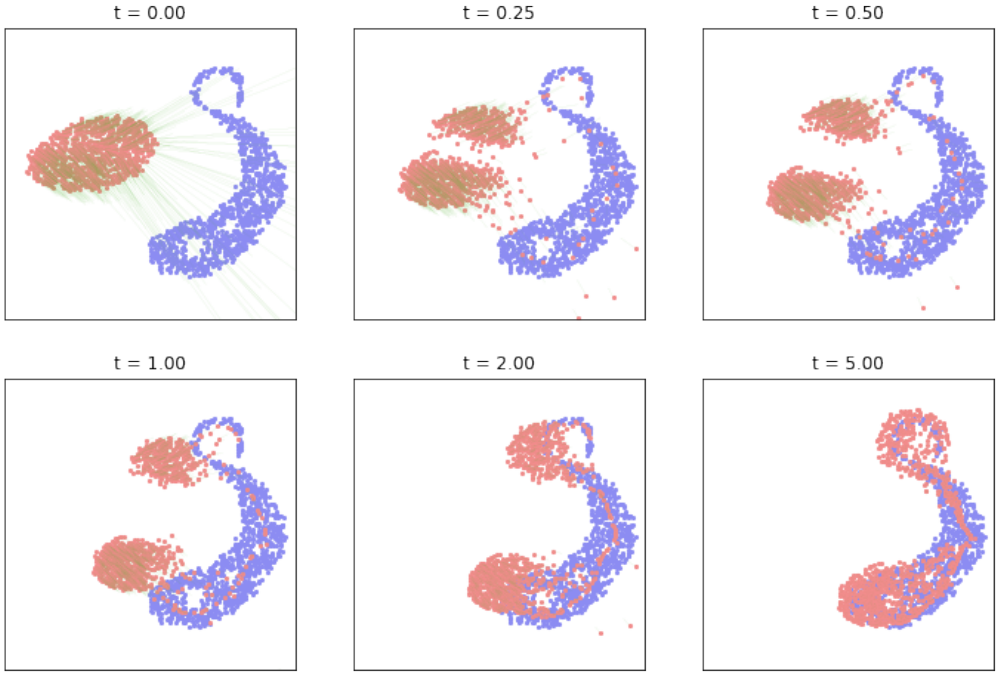
```
In [18]: def kernel_sym_neglog_likelihood(p=2, blur=1) :
             def cost(α_i, x_i, β_j, y_j) :
                 a_j = -KP_log(y_j, x_i, α_i.log(), p, blur)
                 b_i = -KP_log(x_i, y_j, β_j.log(), p, blur)
                 return scal(α_i,b_i) + scal(β_j, a_j)
             return cost
```

```
In [19]: gradient_flow(α_i, x_i, β_j, y_j, kernel_sym_neglog_likelihood(p=1, blur=.1) )
```

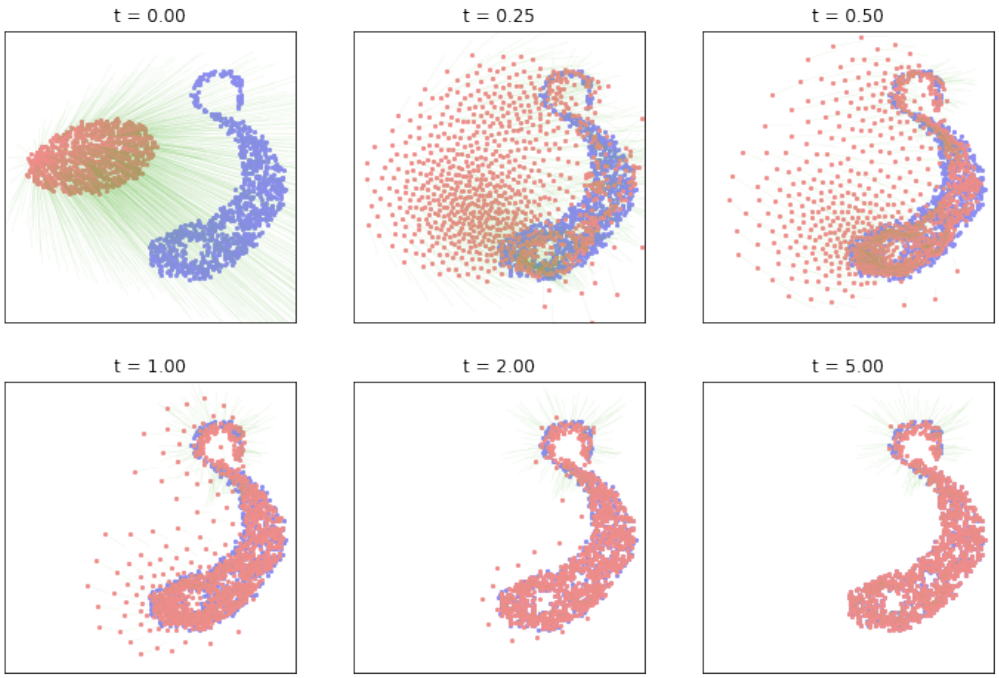



```
In [20]: def kernel_full_neglog_likelihood(p=2, blur=1) :
             def cost(α_i, x_i, β_j, y_j) :
                 a_i = -KP_log(x_i, x_i, α_i.log(), p, blur)
                 a_j = -KP_log(y_j, x_i, α_i.log(), p, blur)
                 b_i = -KP_log(x_i, y_j, β_j.log(), p, blur)
                 b_j = -KP_log(y_j, y_j, β_j.log(), p, blur)

                 return scal(α_i, b_i-a_i) + scal(β_j, a_j-b_j)
             return cost
```

```
In [21]: gradient_flow(α_i, x_i, β_j, y_j, kernel_full_neglog_likelihood(p=2, blur=.05) )
```




```
In [22]: gradient_flow(α_i, x_i, β_j, y_j, kernel_full_neglog_likelihood(p=1, blur=.05) )
```



Solution 7: Both "fixes" rely on smooth distance fields

$$a(y) \;=\; \min_{x\sim\alpha,\sigma^p} \tfrac{1}{p}\|x-y\|^p, \qquad\qquad b(x) \;=\; \min_{y\sim\beta,\sigma^p} \tfrac{1}{p}\|x-y\|^p. \tag{13.28}$$

The first formula,

$$\mathrm{d}_{\text{ML-sym}}(\alpha,\beta) \;=\; \langle\, \alpha\,,\, b\,\rangle \;+\; \langle\, \beta\,,\, a\,\rangle, \tag{13.29}$$

is the sum of terms that mean that "the x_i's should be close to β" and "the y_j's should be close to α"... but stills suffers from mode collapse. The second fix,

$$\mathrm{d}_{\text{ML-full}}(\alpha,\beta) \;=\; \langle\, \alpha-\beta\,,\, b-a\,\rangle \;=\; \langle\, \alpha-\beta\,,\, \log\frac{k\star\alpha}{k\star\beta}\,\rangle, \tag{13.30}$$

with $k(x) = \exp(-\|x\|^p/p\sigma^p)$ is a better try as it mimicks the quadratic-like formula of kernel norms.

Unfortunately though, it can be shown simply that this "log-kernel", Hausdorff-like formula does **not** define a positive definite divergence between measures. Generically, there exists a measure $\alpha \neq \beta$ such that

$$\mathrm{d}_{\text{ML-full}}(\alpha,\beta) < \mathrm{d}_{\text{ML-full}}(\beta,\beta) = 0. \tag{13.31}$$

Using an Optimal Transport distance

In the previous two sections, we've seen how to compute **kernel distances**, which are the duals of Sobolev-like norms on space of functionals, as well as **Maximum Likelihood scores** for Gaussian-Laplace Mixture Models, which can be understood as soft generalizations of the integrated Hausdorff/Chamfer distance.

Last but not least, we now show how to compute **Optimal Transport** plans efficiently, ending up on Wasserstein-like distances between unlabeled measures.

Getting used to Optimal Transport. The modern OT theory relies on a few objects and problems that we now briefly recall. For a complete reference on the subject, you may find useful Filippo Santambrogio's *Optimal Transport for Applied Mathematicians* (2015) or Peyré-Cuturi's *Computational Optimal Transport* (2017), depending on your background.

Kantorovitch problem. Given $\alpha = \sum_i \alpha_i\,\delta_{x_i}$ and $\beta = \sum_j \beta_j\,\delta_{y_j}$ we wish to find a **Transport Plan** π (a measure on the product $\{x_i,\dots\}\times\{y_j,\dots\}$, encoded as an N-by-M matrix $(\pi(x_i \leftrightarrow y_j)) = (\pi_{i,j})$) which is a solution of the following optimization problem:

$$\text{minimize} \quad \langle \pi, C\rangle \;=\; \sum_{i,j} \pi_{i,j} C_{i,j}$$

$$\text{subject to:} \quad \forall\, i,j, \quad \pi_{i,j} \geqslant 0, \quad \sum_j \pi_{i,j} \;=\; \alpha_i, \quad \sum_i \pi_{i,j} \;=\; \beta_j,$$

where the **Cost matrix** $C_{i,j} = c(x_i, y_j)$ encodes the cost of moving a unit mass from point x_i to point y_j.

Wasserstein distance. If one uses $c(x_i, y_j) = \|x_i - y_j\|^2$, the optimal value of the above problem is called the **Wasserstein distance** $\mathrm{d}_{\mathrm{Wass}}(\alpha, \beta)$ between measures α and β. Its theoretical properties are plentiful... But can we compute it efficiently? **In the general high-dimensional case: no, we can't.** Indeed, the Kantorovitch problem above is a textbook **Linear optimization problem**, combinatorial by nature. Even though the simplex algorithm or other classical routines output exact solutions, they do so at a prohibitive cost: at least cubic wrt. the number of samples.

Entropic regularization. Thankfully, we can however **compute approximate transport plans at a much lower cost**. Given a small regularization parameter ε, the idea is to add an **entropic barrier** to the Linear Kantorovitch problem and solve

$$\begin{aligned}\text{minimize} \quad \langle \pi, C \rangle + \varepsilon \mathrm{KL}(\pi, \alpha \otimes \beta) \;&=\; \sum_{i,j} \pi_{i,j} C_{i,j} + \varepsilon \sum_{i,j} \big[\pi_{i,j} \,\log \frac{\pi_{i,j}}{\alpha_i \beta_j} \;-\; \pi_{i,j} \;+\; \alpha_i \beta_j \big] \\ \text{subject to:} \quad \forall\, i,j, \quad \pi_{i,j} \geqslant 0, \quad \sum_j \pi_{i,j} \;&=\; \alpha_i, \quad \sum_i \pi_{i,j} \;=\; \beta_j.\end{aligned}$$

An important property of the $x \mapsto x \log x - x + 1$ function is that it has **a $-\infty$ derivative at location $x = 0$**. Since the main objective function $\langle \pi, C \rangle$ is linear wrt. the $\pi_{i,j}$'s, this implies that the optimal value of the regularized problem is attained **in the relative interior of the simplex, defined by the constraints:**

$$\pi \;>\; 0, \quad \pi 1 \;=\; \alpha, \quad \pi^T 1 \;=\; \beta.$$

Hence, **the optimum is necessarily reached at a critical point of our constrained problem**. At the optimum $\pi^\star$, the gradient of the objective can thus be written as a linear combination of the equality constraints' gradients:

$$\exists\, (f_i^\star) \in \mathbb{R}^{\mathrm{N}}, \; (g_j^\star) \in \mathbb{R}^{\mathrm{M}}, \quad \forall\, i,j, \quad C_{i,j} + \varepsilon \,\log \frac{\pi^\star_{i,j}}{\alpha_i \beta_j} \;=\; f_i^\star + g_j^\star$$

where $f_i^\star$ is the coefficient associated to the constraint $\sum_j \pi_{i,j} = \alpha_i$, as $g_j^\star$ is linked to $\sum_i \pi_{i,j} = \beta_j$.

All in all, we see that the optimal transport plan $(\pi^\star_{i,j}) \in \mathbb{R}_+^{\mathrm{N} \times \mathrm{M}}$ is characterized by **a single pair of vectors** $(f^\star, g^\star) \in \mathbb{R}^{\mathbf{N}+\mathbf{M}}$:

$$\forall\, i,j, \qquad \log \frac{\pi^\star_{i,j}}{\alpha_i \beta_j} \;=\; (f_i^\star + g_j^\star - C_{i,j})/\varepsilon$$

$$\text{i.e.} \quad \pi^\star \;=\; \mathrm{diag}(\alpha_i U_i)\, K_{i,j}\, \mathrm{diag}(V_j \beta_j)$$

$$\text{with} \quad U_i \;=\; \exp(f_i^\star/\varepsilon), \quad V_j \;=\; \exp(g_j^\star/\varepsilon), \quad K_{i,j} \;=\; \exp(-C_{i,j}/\varepsilon).$$

Consequences of this "critical point equation" are twofold:

– The **dimension of the space in which we should search the optimal transport plan is greatly reduced**, jumping from $(\mathrm{M} \times \mathrm{N})$ to $(\mathrm{M} + \mathrm{N})$ - the adjoint variables associated to the equality constraints. Furthermore, the optimal value of the cost can be computed using this **cheap formula**:

$$\mathrm{OT}_\varepsilon(\alpha, \beta) \;=\; \langle \pi^\star, C \rangle + \varepsilon \mathrm{KL}(\pi^\star, \alpha \otimes \beta) \;=\; \sum_{i,j} \pi^\star_{i,j} \left(f^\star_i + g^\star_j \right) \;=\; \langle \alpha, f^\star \rangle + \langle \beta, g^\star \rangle. \tag{13.32}$$

– The optimal transport plan can be expressed as **the positive scaling of a positive kernel matrix** K. But in the meantime, it should also satisfy the two marginal constraints which can be written in terms of (U_i) and (V_j):

$$U_i \;=\; \frac{1}{(K(V\beta))_i}, \qquad V_j \;=\; \frac{1}{(K^T(U\alpha))_j}.$$

As was remarked by a long trail of authors (from Schrödinger's original work, to economy and statistical physics in the 60-80-90-00's, to object recognition in the 90-00's and more recently in the machine learning literature) this reformulation of (entropic regularized) Optimal Transport can be linked to the **Sinkhorn Theorem**: It admits a unique solution $(U, V) \in \mathbb{R}^{\mathrm{N}+\mathrm{M}}_{>0}$, which can be approached iteratively by applying the steps

$$U^{(0)} = (1, \dots, 1),\; V^{(0)} = (1, \dots, 1), \qquad V^{(n+1)} \;=\; \frac{1}{K^T(U^{(n)}\alpha)}, \qquad U^{(n+1)} \;=\; \frac{1}{K(V^{(n+1)}\beta)}.$$

These are nothing but coordinate ascent steps on the **dual maximization problem**:

$$\mathrm{OT}_\varepsilon(\alpha, \beta) \;=\; \max_{f,g} \; \langle \alpha, f \rangle \;+\; \langle \beta, g \rangle \;-\; \varepsilon \left\langle \alpha \otimes \beta, e^{(f \oplus g - C)/\varepsilon} - 1 \right\rangle. \tag{13.33}$$

Hence, **one can solve the regularized Optimal Transport problem by iterating kernel products (aka. discrete convolutions) and pointwise divisions, on variables which have the same memory footprint as the input measures!**

Sinkhorn algorithm in the log domain

Is the scheme presented above stable enough? No, it isn't. Indeed, as discussed in the section dedicated to Maximum likelihood estimators, **if we use kernels in multiplicative formulas, we should favor log-domain implementations**.

```
In [23]: D = lambda x : x.detach() # use the formula at convergence for the gradient

         def ot_reg(p = 2, blur = .05, scaling=.5 ) :
             def cost(α_i, x_i, β_j, y_j) :
                 # ε-scaling heuristic (aka. simulated annealing):
                 # let ε decrease across iterations, from 1 (=diameter) to the target value
```

```
        scales = [ tensor([np.exp(e)]) for e in
                   np.arange(1, np.log(blur), np.log(scaling)) ] + [blur]

        # Solve the OT_ε(α,β) problem
        f_i, g_j = torch.zeros_like(α_i), torch.zeros_like(β_j)
        for scale in scales :
            g_j = -KP_log(y_j, D(x_i), D(f_i/scale**p + α_i.log()), p=p, blur=scale)
            f_i = -KP_log(x_i, D(y_j), D(g_j/scale**p + β_j.log()), p=p, blur=scale)

        # Return the dual cost OT_ε(α,β), assuming convergence in the Sinkhorn loop
        return scal(α_i, f_i) + scal(β_j, g_j)
    return cost
```

Exercise 8: Explain why the implementation above is **correct** and **numerically stable**.

Solution 8: On the dual variables $f^{(n)} = \varepsilon \log U^{(n)}$ and $g^{(n)} = \varepsilon \log V^{(n)}$, the Sinkhorn iterations read

$$
\begin{aligned}
g_j^{(n+1)} &= -\varepsilon \log K^T (U^{(n)} \alpha) && (13.34)\\
&= -\varepsilon \log \sum_{i=1}^{\mathrm{N}} \exp\big(-\|x_i - y_j\|^p / p\varepsilon + f_i^{(n)}/\varepsilon + \log \alpha_i \big) && (13.35)\\
g_j^{(n+1)} &= \min_{x \sim \alpha, \varepsilon} \big[\tfrac{1}{p} \|y_j - x\|^p - f^{(n)}(x) \big], && (13.36)\\
& && (13.37)\\
f_i^{(n+1)} &= -\varepsilon \log K (V^{(n+1)} \beta) && (13.38)\\
&= -\varepsilon \log \sum_{j=1}^{\mathrm{M}} \exp\big(-\|x_i - y_j\|^p / p\varepsilon + g_j^{(n+1)}/\varepsilon + \log \beta_i \big) && (13.39)\\
f_i^{(n+1)} &= \min_{y \sim \beta, \varepsilon} \big[\tfrac{1}{p} \|x_i - y\|^p - g^{(n+1)}(y) \big], && (13.40)
\end{aligned}
$$

which is what is implemented here with $\varepsilon = \sigma^p$. Crucially, if the log-sum-exp reduction is implemented properly, this code won't suffer from numerical overflows even if $U^{(n)} = e^{\pm 100}$...

Exercise 9: Link this implementation with the log-likelihood cost presented in the previous section. Intuitively, could you explain the behaviour of this algorithm? Why is it much **faster** than the "standard" Sinkhorn algorithm, with a fixed value of ε? How could you **improve** it further?

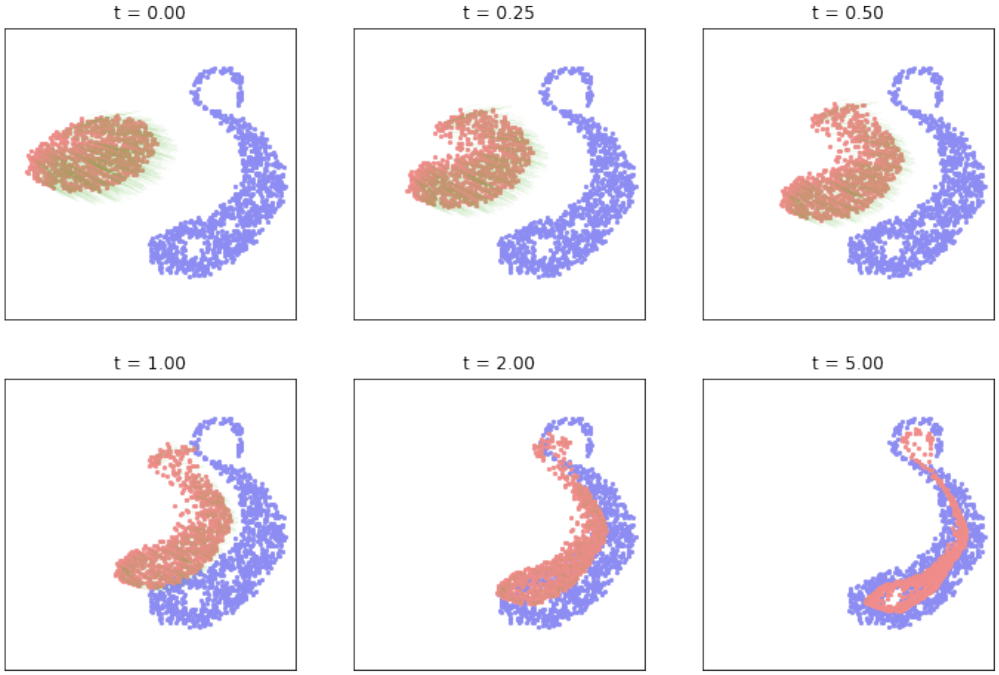
Solution 9: As detailed in the previous answer, the Sinkhorn iterations can now be understood with quantities that are homogeneous to the cost $\frac{1}{p}\|x-y\|^p$: the **prices** f and g. The $\min_\varepsilon$ updates now resemble closely those of standard combinatorial methods such as the Auction algorithm, and can be studied accordingly: see *Kosowsky and Yuille (1993)* and *Schmitzer (2016)* for reference.

Two intuitions arise from this analysis:

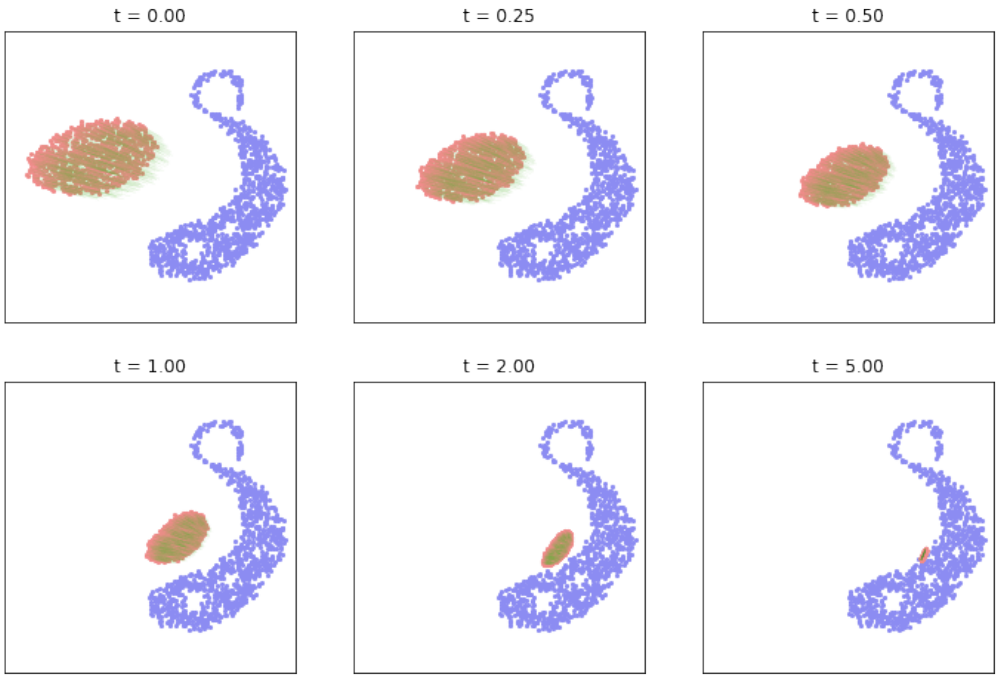
1. Before reaching convergence, Sinkhorn updates typically make steps of size ε in the maximization of the dual cost. By using larger values of ε in the first few iterations, we make larger strides and reach quickly the fine-tuning, end-game regime.
2. Optimal Transport is fundamentally a **multiscale problem**: a rough transport plan computed at a coarse scale can always be refined into a finer correspondence. This is what the ε-scaling heuristic is all about, as we lower the amount of blur (or *temperature*) from one iteration to the other.

To improve this algorithm further, we could remark that the updates at any iteration are typically "ε-smooth". In the first few iterations, we could thus work on *subsampled measures* and develop a fully-fledged multiscale algorithm.

```
In [24]: gradient_flow(α_i, x_i, β_j, y_j, ot_reg(p=2, blur=.05) )
```




```
In [25]: gradient_flow(α_i, x_i, β_j, y_j, ot_reg(p=2, blur=.25) )
```



Exercise 10: In the experience above, can you explain the **entropic bias**, which pushes α_t away from β, onto a medial-axis like measure with a narrow support?

Solution 10: We know that $\mathrm{OT}_\varepsilon(\alpha,\beta)$ is (roughly) equal to the transport cost associated to a **fuzzy** transport plan

$$\pi^\star \;=\; \exp\tfrac{1}{\varepsilon}(f^\star \oplus g^\star - C)\cdot \alpha\otimes\beta, \tag{13.41}$$

which typically links any point x_i to an ε-ball of points y_j in β. As we minimize the sum of (squared) lengths associated to this fuzzy "system of springs", points x_i tend to converge towards the **median** (if p=1) or **mean** (if p=2) value of their ε-mates, which often lies deep **inside** the convex hull of β's support.

To solve this problem, an idea is to define the **Sinkhorn divergence** in a way that mimicks the bilinear expansion of squared Euclidean norms:

$$\mathrm{S}_\varepsilon(\alpha,\beta) \;=\; \mathrm{OT}_\varepsilon(\alpha,\beta) \;-\; \tfrac{1}{2}\mathrm{OT}_\varepsilon(\alpha,\alpha) \;-\; \tfrac{1}{2}\mathrm{OT}_\varepsilon(\beta,\beta). \tag{13.42}$$

Most interestingly, we then get that **Sinkhorn divergences interpolate between Optimal Transport and kernel norms**:

$$\mathrm{OT}_C(\alpha,\beta) \;\xleftarrow{0\leftarrow\varepsilon}\; \mathrm{S}_\varepsilon(\alpha,\beta) \;\xrightarrow{\varepsilon\to+\infty}\; \tfrac{1}{2}\|\alpha-\beta\|^2_{-C}. \tag{13.43}$$

In 2018, it was shown that under mild assumptions, S_ε defines a symmetric, **positive-definite** divergence which is **convex** with respect to each variable and metrizes the convergence in law. In particular, the entropic bias is removed and our gradient flow converges towards β, up to the high-frequency components lost when seeing both measures through the blurring convolution kernel

$$k_\varepsilon(x,y) \;=\; e^{-C(x,y)/\varepsilon}. \tag{13.44}$$

```
In [26]: def sinkhorn_divergence(p = 2, blur = .05, scaling=.5 ) :
             def cost(α_i, x_i, β_j, y_j) :
                 # ε-scaling heuristic (aka. simulated annealing):
                 # let ε decrease across iterations, from 1 (=diameter) to the target value
                 scales = [ tensor([np.exp(e)]) for e in
                            np.arange(0, np.log(blur), np.log(scaling)) ] + [blur]

                 # 1) Solve the OT_ε(α,β) problem
                 f_i, g_j = torch.zeros_like(α_i), torch.zeros_like(β_j)
                 for scale in scales :
                     g_j = - KP_log(y_j, D(x_i), D(f_i/scale**p + α_i.log()), p=p, blur=scale)
                     f_i = - KP_log(x_i, D(y_j), D(g_j/scale**p + β_j.log()), p=p, blur=scale)

                 # 2) Solve the OT_ε(α,α) and OT_ε(β,β) problems
                 scales_sym = [scale]*3 # Symmetric updates converge very quickly
                 g_i, f_j = torch.zeros_like(α_i), torch.zeros_like(β_j)
                 for scale in scales_sym :
                     g_i=.5*(g_i - KP_log(x_i, x_i, g_i/scale**p + α_i.log(), p=p, blur=scale))
```

```
            f_j=.5*(f_j - KP_log(y_j, y_j, f_j/scale**p + β_j.log(), p=p, blur=scale))
        # Final step, to get a nice gradient in the backprop pass:
        g_i = - KP_log(x_i, D(x_i), D(g_i/scale**p + α_i.log()), p=p, blur=scale)
        f_j = - KP_log(y_j, D(y_j), D(f_j/scale**p + β_j.log()), p=p, blur=scale)

        # Return the "dual" cost :
        # S_ε(α,β) =        OT_ε(α,β)       - ½OT_ε(α,α) - ½OT_ε(β,β)
        #          = (<α,f_αβ>+<β,g_αβ>) -  <α,g_αα> - <β,f_ββ>
        return scal(α_i, f_i - g_i) + scal(β_j, g_j - f_j)
    return cost
```

Exercise 11: Explain why the implementation above is correct.

Solution 11: We've already detailed the step 1 ($\mathrm{OT}_\varepsilon(\alpha,\beta)$ problem) and now focus on the symmetric case of the $\alpha \leftrightarrow \alpha$ problem ($\beta \leftrightarrow \beta$ can be handled identically). We know that

$$\mathrm{OT}_\varepsilon(\alpha,\alpha) \;=\; \max_{f,g} \langle \alpha, f+g\rangle \;-\; \varepsilon \langle \alpha\otimes\alpha, e^{(f\oplus g - C)/\varepsilon} - 1\rangle \tag{13.45}$$

$$\;=\; 2\max_{g} \langle \alpha, g\rangle \;-\; \tfrac{\varepsilon}{2} \langle \alpha\otimes\alpha, e^{(g\oplus g - C)/\varepsilon} - 1\rangle, \tag{13.46}$$

because the (f,g) problem is concave and symmetric with respect to a permutation of the dual potentials: there exists a solution $(f=g,g)$ on the diagonal of the space $\mathbb{R}^\mathrm{N}\times\mathbb{R}^\mathrm{N}$ of dual pairs.

How can we find such a solution efficiently? Given a current estimate $(g^{(n)}, g^{(n)})$, we know that defining

$$\overline{g}_i^{(n+1)} \;=\; \min_{x\sim\alpha,\varepsilon} \left[\tfrac{1}{p}\|x_i - x\|^p - g_i^{(n)}\right] \tag{13.47}$$

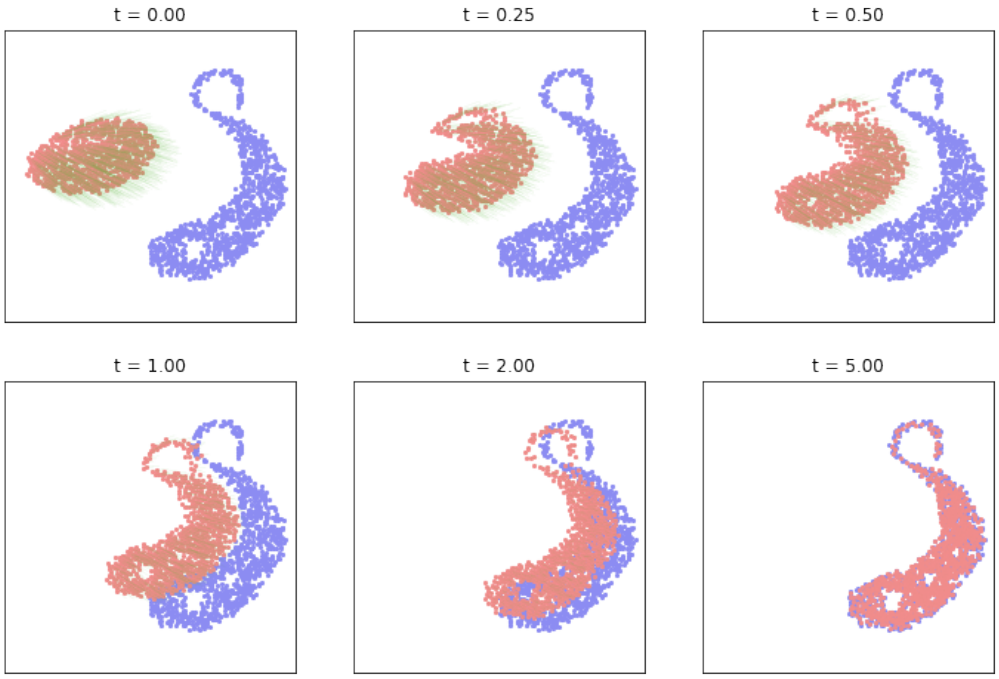
and jumping to $(g^{(n)}, \overline{g}^{(n+1)})$ or $(\overline{g}^{(n+1)}, g^{(n)})$ would bring us closer to the optimum, as this standard Sinkhorn update is a coordinate ascent step on the dual problem.

Going further, **averaging** these two updates by setting

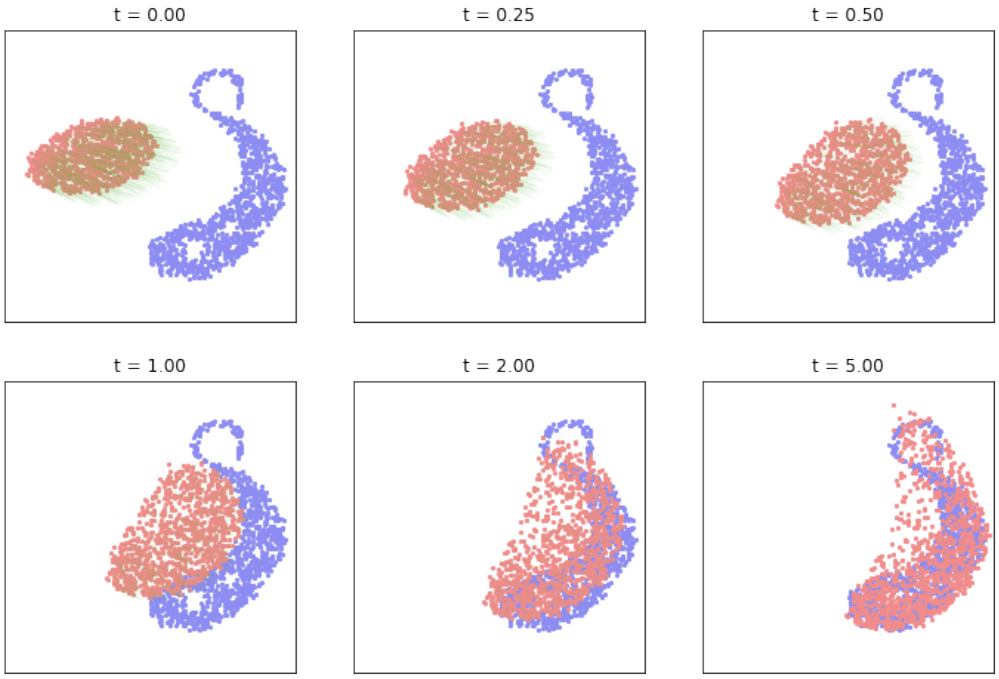
$$g^{(n+1)} \;=\; \tfrac{1}{2}(g^{(n)} + \overline{g}^{(n+1)}) \tag{13.48}$$

and jumping to $(g^{(n+1)}, g^{(n+1)})$ is an even better idea: thanks to the **concavity** of the dual objective, we know that this competitor is at least as good as $(g^{(n)}, \overline{g}^{(n+1)})$ and $(\overline{g}^{(n+1)}, g^{(n)})$... And at the same time, it belongs to the diagonal of the space of dual pairs, where the global optimum is known to lie. Empirically, we always converge to a good enough solution in three or four steps.

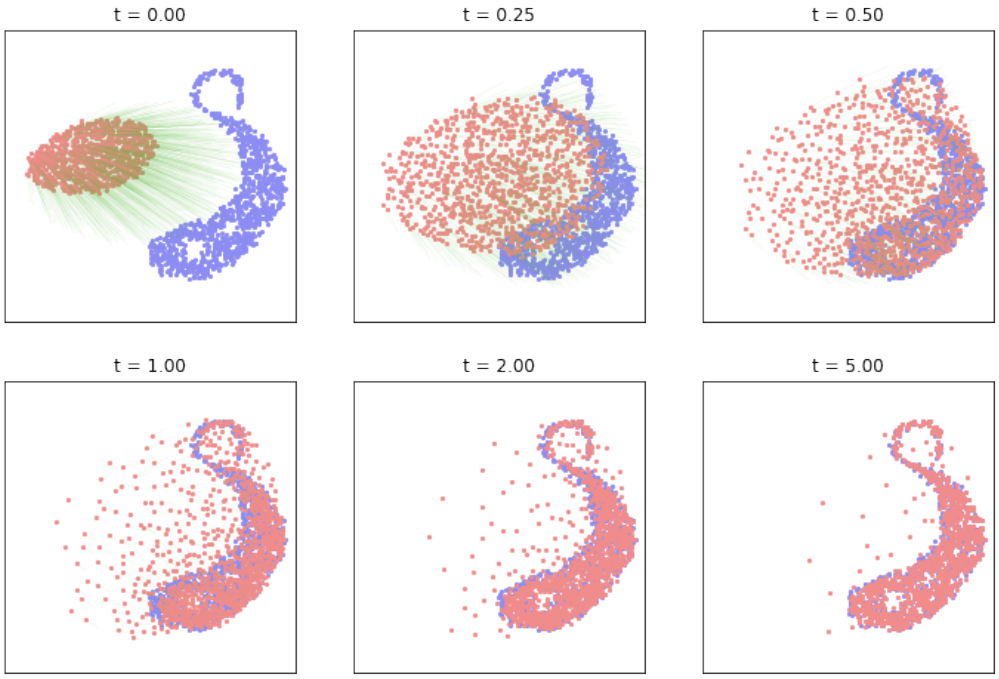
```
In [27]: gradient_flow(α_i, x_i, β_j, y_j, sinkhorn_divergence(p=2, blur=.01) )
```




```
In [28]: gradient_flow(α_i, x_i, β_j, y_j, sinkhorn_divergence(p=2, blur=.2) )
```



```
In [29]: gradient_flow(α_i, x_i, β_j, y_j, sinkhorn_divergence(p=1, blur=.2) )
```



Exercise 12: Discuss the behaviour of S_ε for varying values of the parameters.

Solution 12: When ε is small, we retrieve the behavior of the "true" Wasserstein distance at an affordable cost. However, as ε grows, S_ε behaves more and more like the kernel norm $\frac{1}{2}\|\alpha-\beta\|^2_{-\frac{1}{p}\|\cdot\|^p}$:

- if p=2, $\|\alpha-\beta\|^2_{-\|\cdot\|^2/2} = \frac{1}{2}\|\text{mean}(\alpha)-\text{mean}(\beta)\|_2^2$: the divergence becomes blind to fine details and only registers the first moments with each other.
- if p=1, $\frac{1}{2}\|\alpha-\beta\|^2_{-\|\cdot\|}$ is the **Energy distance**, a kernel norm that was studied in the first section and presents **screening artifacts**.

Fortunately, in all cases, the **entropic bias** is alleviated and we do not observe any **mode collapse**.

Conclusion

In this notebook, we presented three major families of "distance" costs between probability measures:

- Kernel distances (also known as *Maximum Mean Discrepancies*), which descend from dual norms on **Sobolev-like spaces of functions**.
- Empirical log-likelihoods of mixture models, which are smooth generalizations of **Hausdorff-like distances**.

– Sinkhorn divergences, designed as cheap approximations of Optimal Transport costs; they can be linked to dual norms on **Lipschitz-like spaces of functions**.

Interestingly, the three of them use **the same atomic operation: the Kernel Product**, possibly implemented in the log-domain. As it is a GPU-friendly operation, using these formulas results in scalable algorithm: use a vanilla **PyTorch** implementation for clouds of <2,000 samples, and the powerful **KeOps** library for larger (10,000-1,000,000) problems.

But which formula should we use in practical applications? This is the main question, that can only be answered with respect to specific applications and datasets.

A rule of thumb: kernel distances and log-likelihoods are both cheap, and differ in the way they handle **outliers** and isolated x_i's or y_j's. With kernel distances, they are nearly *forgotten* as the tail of the kernel function k goes to zero; in the likelihood case, though, since $-\log k(x) \to +\infty$ when x grows, outliers tend to have a large influence on the overall cost and its gradient. More recently introduced, Sinkhorn distances tend to be both interpretable and robust... At a higher computational cost (multiscale tree-based approaches are super-efficient in dimensions 2 or 3, but break down for high-dimensional problems). Computing a full transport plan to simply get a *gradient* is overkill for most applications, and future works will certainly focus on cheap approximations that interpolate between OT and simpler theories.

To get your own intuition on the subject, feel free to re-run this notebook with measures sampled from your own sketches!

SESSION 14 — Going further

A quick overview. In the past few months, Gabriel and myself guided you through a mathematical tour of data science, starting from standard wavelet analysis to end up on prospective Wasserstein distances. We taught you how to define and implement multiscale representations suited to signal processing tasks; you should now have 1001 methods in mind to perform gradient descent on non-smooth objectives; and, most importantly, **you now have a clear understanding of a typical *applied maths* research field**.

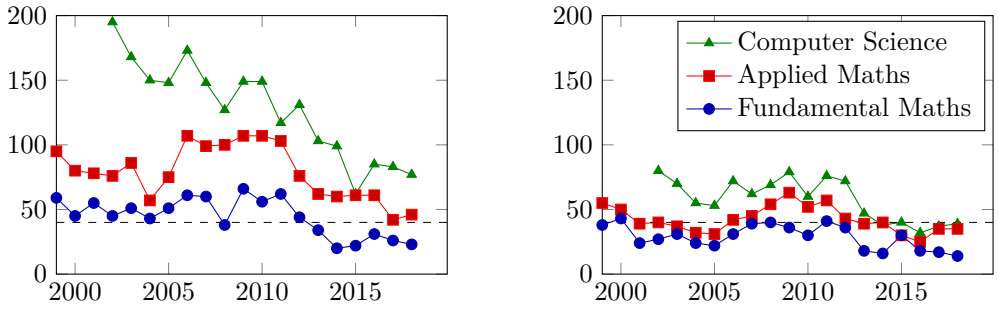
Contrary to a common prejudice in the French prépa system, "real-life problem" does not rhyme with "simplistic ideas"! Today, there are *tons* of unanswered *mathematically interesting* questions related to applied problems, be it in analysis, geometry, probabilities or even algebra. Modern computational tools relieve mathematicians from the burden of low-level implementation, and allow us to focus on what we do best: building up significant abstractions.

The cold truth about academic life. As you are about to become professional scientists, you guys must know where you're heading to. In the deceiving comfort of the "rue d'Ulm", this is somewhat easy to ignore... But pursuing a career in fundamental mathematics is very much akin to the struggle of becoming a professional *musician*.

In academia, whichever field you look at, no more than a **handful** of people per year get the chance to pick their dream job, with access to widespread recognition and motivated pupils. Meanwhile, the owerwhelming majority of scholars find their joy in a confidential practice of research, as they put bread on the table by lecturing mediocre, uninterested students.

Opportunities for applied mathematicians. This way of life is eminently respectable... But doesn't suit everybody. Providentially, applied maths offer you a **wide range** of possible career paths to consider. First of all, as evidenced by the charts below, **getting an academic position is comparatively easier in applied fields**. You simply get more fundings if you can argue that your work will be to the taxpayer's benefit.

Furthermore, applied mathematicians get access to a host of **fullfilling research positions in the industry**. In exchange for the fact that *they* choose what topic you're working on (like an advisor, really), companies are eager to offer you a dramatic pay increase and top-notch research facilities. All of this may seem remote... But at some point, having these options at hand may dispense you from having to choose between your family and your career.



(a) Maîtres de conférence.

(b) Professors.

Figure 14.1: University positions available in France, depending on the field. One should also take into account the CNRS (all fields) and INRIA (applied maths + CS only), which offer about 20 "Chargé de recherche" and 10 "Directeur de recherche" positions per year each.
Data available on the reference website, `postes.smai.emath.fr/2018/OUTILS/bilans.php`.

Choosing your Master 2 program. If you're so passionate about pure maths that you think they're worth putting your career perspectives in jeopardy, that's great; listen to your heart. But otherwise, for your own sake, **consider giving a chance to applied M2 programs**. Off the top of my head, in the Paris region, you have access to the **MSV** (biology), the **MASH** (machine learning and social sciences), the **MVA** (machine learning and image processing)...
So do not hesitate, have a look at their webpages :-)

A selection of lectures from the MVA program. To help you to find relevant topics, here is a shortlist of MVA courses which may suit your mathematical taste: I ranked them by difficulty, ranging from ($\star$) "easy enough" to ($\star\star\star$) "just as challenging as regular DMA courses".

If you liked Gabriel's course, why wouldn't you try to attend one of them in the coming semester? Obviously, you could also try out the CS and engineering-oriented classes!
Further information can be found on the program's website:

`math.ens-paris-saclay.fr/version-francaise/formations/master-mva/contenus-/master-mva-cours-2017-2018-161721.kjsp`

October-January:

- $\star$ **Introduction to medical image analysis.** Get familiar with the specific problems encountered in the medical imaging industry, and see how mathematicians can help to tackle such life-impacting problems.
- $\star$ **Sub-pixel image processing.** Become an FFT guru!
- $\star\star$ **Topological data analysis for imaging and machine learning.** Extract relevant information from noisy datasets by computing invariants defined in the course of algebraic topology.
- $\star\star$ **Sparsity and compressed sensing.** Become an L^1-norm guru...
  But you know that already ;-)
- $\star\star\star$ **Computational statistics.** Discover the mathematical results behind widely used statistical algorithms such as stochastic gradient descent, EM, Markov Chain Monte Carlo samplers and approximated Bayesian computing methods.
- $\star\star\star$ **Mathematical methods for neurosciences.** In-depth study of biological neurons, using both deterministic (bifurcations) and stochastic (Itô calculus) theories of dynamical systems. Fascinating!

January-April:

- $\star$ **Statistical computing on manifolds and data assimilation.** From medical images to anatomical and physiological models. Very similar to the "Introduction to medical image analysis" course, albeit focused on geometrical problems.
- $\star\star$ **Longitudinal data analysis.** How do we study time-dependent series of structured data points? A textbook problem is that of predicting the evolution of Alzheimer's disease, from a few brain MRI scans.
- $\star\star$ **Deformable models and geodesic methods for image analysis.** Learn to segment image regions – Gabriel's course in the second semester.
- $\star\star\star$ **Inverse problems in imagery.** How do we reconstruct geological profiles from seismic data? The lecturer was in charge of the "Stochastic Processes" course at the ENS a few years ago.
- $\star\star\star$ **Geometry and shapes spaces.** Study groups of diffeomorphisms of the ambient space as infinite-dimensional Riemannian manifolds, and use the Hamiltonian characterization of geodesics to perform non-linear analysis on populations of shapes – brain and heart MRI scans, mostly. A personal favorite, as the lecturer is no one but my (fantastic) PhD advisor!

Contacts. Being ENS students, you have plenty of opportunities to go and speak with experienced researchers. Don't forget that it's part of your professors' duties... And that they love it! People at the DMA will always have time to answer your questions regarding orientation and internships: as a starting point, I would advise you to meet **Amandine Véber** (biology), **Bertrand Maury** (flow modelling), **Jean-Philippe Vert** (genomic studies), **Gabriel Peyré** (compressed sensing, optimal transport) and **myself** (medical imaging + student's tips).

Whatever your future orientation, I wish you all the best!