

COMPÉTITION KAGGLE

RSNA Pneumonia Detection Challenge

Can you build an algorithm that automatically detects potential pneumonia cases ?

EMNA JAÏEM

INGÉNIEURE DE RECHERCHE EN MATHÉMATIQUES APPLIQUÉES
ET DATA SCIENTIST

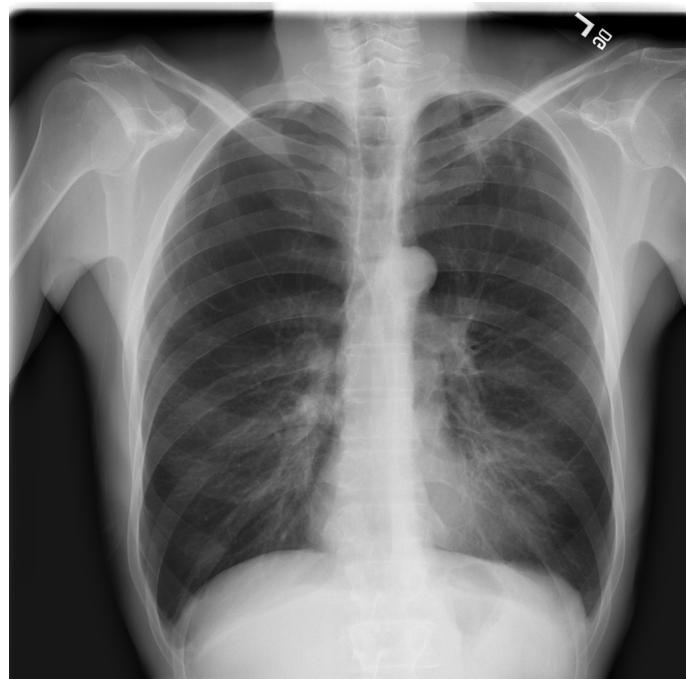


Table des matières

1	Introduction	4
2	Analyse des données	5
2.1	Identification des variables	5
2.2	Analyse univariée	14
2.2.1	Analyse de la variable x	15
2.2.2	Analyse de la variable y	16
2.2.3	Analyse de la variable $width$	18
2.2.4	Analyse de la variable $height$	20
2.2.5	Analyse de la variable $Target$	21
2.2.6	Analyse de la variable $class$	21
2.2.7	Analyse de la variable Age	22
2.2.8	Analyse de la variable $Sexe$	23
2.2.9	Analyse de la variable $Position$	24
2.3	Analyse multivariée	25
2.3.1	Combinaisons de type continu/continu	25
2.3.2	Combinaisons de type catégoriel/catégoriel	28
2.3.3	Combinaisons de type catégoriel/continu	31
2.3.4	Corrélation et conclusions	33
2.4	Valeurs manquantes et aberrantes	36
2.4.1	Traitements des valeurs manquantes	36
2.4.2	Traitements des valeurs aberrantes	37
2.5	Extraction de caractéristiques (<i>Features Engineering</i>)	37
3	Modèles simples de classification d'images	39
3.1	Réseaux de neurones profonds	39
3.2	Réseau de neurones convolutif (CNN)	41
3.2.1	Opérations de convolution	42
	Couche de convolution	42
	Pooling	45
	Padding	46
	Redimensionnement (Flattened)	47
3.2.2	Techniques d'amélioration	47
	Prétraitement des données d'entrée	47
	Méthode de dégradation des pondérations	48
	Décrochage	48
	Augmentation des données	48
	<i>Batch normalization</i>	50
	<i>Transfer Learning</i>	51
3.2.3	Architectures classiques de CNN	51
	VGG	51
	DenseNet	55
	CheXNet	55
3.3	Implémentation et analyse des différentes architectures	56
3.3.1	Data generator	56

TABLE DES MATIÈRES

3.3.2	Diagnostic du modèle d'apprentissage	61
3.3.3	Résultats	62

1 Introduction

Le "RSNA Pneumonia Detection Challenge" est un challenge lancé en 2018 par la société nord-Américaine de radiologie (RSNA) sur la plateforme Kaggle. L'objectif de ce challenge est de construire un algorithme permettant de détecter la présence de pneumonie dans des radiographies pulmonaires numérisées. Il s'agit, en effet, d'un enjeu important, rien qu'aux États-Unis, plusieurs centaines de milliers de personnes sont admises aux urgences à cause d'une pneumonie et plusieurs dizaines de milliers d'entre elles en meurent chaque année [Rui and Kang, 2015; dea, 2017].

La pneumonie est une infection des poumons provoquée généralement par un virus ou une bactérie. Les patients souffrant d'une pneumonie présentent le plus souvent de la toux accompagnée de fièvre, de crachats, de difficultés respiratoires ou de douleurs thoraciques. En milieu hospitalier, l'agent pathogène à l'origine de la pneumonie est diagnostiqué à partir de tests sanguins, de radiographie du thorax et de bien d'autres tests. Néanmoins, en pratique, le rôle principal de la radiographie est de venir confirmer un diagnostic préétabli afin de mettre en place une stratégie de traitement. La radiographie permet également de suivre l'évolution des effets du traitement [Nambu et al., 2014].

Malgré les connaissances approfondies des pneumonies, le diagnostic de celles-ci à travers des radiographies n'en reste pas moins une tâche relativement compliquée qui nécessite l'œil expert de médecin hautement qualifié. En effet, une raison à cela peut venir de la qualité parfois insuffisante des radiographies, comme par exemple, un mauvais positionnement du patient [Kelly, 2012]. Une raison supplémentaire à la difficulté de détection des pneumonies est due à la présence d'autres pathologies. Les pneumonies sont détectables par la présence d'une ou plusieurs zones opaques appelées opacités. Néanmoins, d'autres pathologies provoquent également des opacités comme les œdèmes pulmonaires, les saignements, les cancers et bien d'autres. Le but de ce challenge a donc été d'inviter la communauté de l'intelligence artificielle à collaborer afin d'améliorer la qualité des diagnostics.

Ce rapport s'organise autour de trois axes. Le premier, et non des moindres concerne l'analyse de la base de données. L'objectif de cette étape est de préparer les données d'apprentissage pour notre modèle. La qualité du modèle dépend fortement de cette étape. La seconde partie de ce manuscrit concerne le *data generator*, c'est à dire, préparer les outils permettant au modèle d'accéder judicieusement aux données d'apprentissage. Ensuite, nous présentons les réseaux de neurones convolutifs (CNN). Dans cette partie, nous détaillons les différentes étapes de ces réseaux et nous discuterons des stratégies concernant différentes architectures que peut avoir notre modèle.

2 Analyse des données

L'analyse des données (en anglais : *Exploration Data Analysis* (EDA)) est une famille de méthodes statistiques dont l'objectif est de comprendre, nettoyer et préparer les données pour la modélisation. Cette étape est relativement importante et ne doit pas être négligée, la qualité des entrées détermine la qualité des sorties. En apprentissage profond, les étapes d'EDA sont bien définies, les voici :

- Identification des variables
- Analyse univariée
- Analyse multivariée
- Traitement des valeurs manquantes
- Traitement des valeurs aberrantes
- Transformation des variables
- Création de variables

Avant de proposer un modèle raffiné, certaines étapes peuvent être répétées plusieurs fois comme, par exemple, les étapes 4 à 7.

2.1 Identification des variables

La première tâche est de définir les variables d'entrée et de sortie, d'identifier le type de chaque variable (continue ou catégorielle) et de les préparer pour les étapes suivantes. Notre étude s'intéressera plus particulièrement aux fichiers *stage_2_train_labels.csv*, *stage_2_detailed_class_info.csv* et au dossier *stage_2_train_images* contenant une liste de fichiers au format *.dcm*. Dans un premier temps, intéressons-nous au premier fichier.

```
1 import os
2 import numpy as np
3 import pandas as pd
4
5 path = '../input/rsna-pneumonia-detection-challenge/'
6
7 train = pd.read_csv(path+'stage_2_train_labels.csv')
8 print(train.head())
9 print(train.info())
```

	patientId	x	y	width	height	Target
0	0004cfa...6bddd6	NaN	NaN	NaN	NaN	0
1	00313ee...3241cd	NaN	NaN	NaN	NaN	0
2	00322d4...e640eb	NaN	NaN	NaN	NaN	0
3	003d8fa...8495c5	NaN	NaN	NaN	NaN	0
4	0043651...9b9ab4	264.0	152.0	213.0	379.0	1

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30227 entries, 0 to 30226
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype 

```

2. ANALYSE DES DONNÉES

```
0   patientId    30227 non-null  object
1   x             9555  non-null  float64
2   y             9555  non-null  float64
3   width         9555  non-null  float64
4   height        9555  non-null  float64
5   Target        30227 non-null  int64
dtypes: float64(4), int64(1), object(1)
memory usage: 1.4+ MB
None
```

Le fichier possède six colonnes et 30227 lignes. Il contient une colonne *patientId* de type *object* contenant un label permettant d'identifier le patient, ce label pointe sur le nom de la radiographie du patient. La colonne *Target* vaut 0 si le patient n'est pas atteint de pneumonie et 1 dans le cas contraire, il s'agit de la colonne de sortie. Dans le cas où le patient est atteint de pneumonie, un encadrement de l'opacité est décrit via les colonnes *x*, *y*, *width* et *height*. Puisqu'un patient peut avoir plusieurs opacités, il peut apparaître plusieurs fois dans le fichier.

Maintenant, regardons en détail le second fichier.

```
11 detailed = pd.read_csv(path+'stage_2_detailed_class_info.csv')
12 print(detailed.head())
13 print(detailed.info())
```

	patientId	class
0	0004cfa...63a80b6bdd6	No Lung Opacity / Not Normal
1	00313ee...c148ed3241cd	No Lung Opacity / Not Normal
2	00322d4...b6754be640eb	No Lung Opacity / Not Normal
3	003d8fa...ac657f8495c5	Normal
4	0043651...de91049b9ab4	Lung Opacity

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30227 entries, 0 to 30226
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype  
 0   patientId   30227 non-null   object 
 1   class        30227 non-null   object 
dtypes: object(2)
memory usage: 472.4+ KB
None
```

Celui-ci est composé de 30227 lignes et de 2 colonnes. La première colonne *patientId* correspond à l'identifiant des patients et la deuxième apporte une information supplémentaire sur l'état du patient. Dans cette colonne, trois choix sont possibles. La première *Normal*, signifie que la radiographie est normale, sans opacité. La seconde

2. ANALYSE DES DONNÉES

Lung Opacity signifie que des zones opaques dues à une pneumonie sont visibles sur la radiographie. Et finalement, *No Lung Opacity / Not Normal* signifie que la radiographie n'est pas normale, c'est-à-dire qu'il y a bien des zones d'opacités, mais que ces zones ne sont pas causées par une pneumonie. La prise en compte de ces informations n'est pas imposée par les règles du challenge. Néanmoins, elles pourraient permettre d'améliorer le modèle, voire de remplacer la variable de sortie et il serait déraisonnable de les écarter dès maintenant.

Pour faciliter la manipulation des données, les deux bases de données sont concaténées dans une table *df* à la manière d'une jointure SQL sur la colonne *patientId* :

```
15 df = pd.merge(left = train, right = detailed, how = 'left', on = 'patientId').drop_duplicates()  
16 del train, detailed  
17 print(df.head())  
18 print(df.info())
```

	patientId	x	...	Target	class
0	000...dd6	NaN	...	0	No Lung Opacity / Not Normal
1	003...1cd	NaN	...	0	No Lung Opacity / Not Normal
2	003...0eb	NaN	...	0	No Lung Opacity / Not Normal
3	003...5c5	NaN	...	0	Normal
4	004...ab4	264.0	...	1	Lung Opacity


```
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 30227 entries, 0 to 37627  
Data columns (total 7 columns):  
 #   Column      Non-Null Count  Dtype     
 --  --          --          --         
 0   patientId  30227 non-null   object    
 1   x           9555 non-null   float64  
 2   y           9555 non-null   float64  
 3   width       9555 non-null   float64  
 4   height      9555 non-null   float64  
 5   Target       30227 non-null   int64    
 6   class        30227 non-null   object    
dtypes: float64(4), int64(1), object(2)  
memory usage: 1.8+ MB  
None
```

À ce stade, nous pouvons faire plusieurs petites vérifications sur les données. Nous pouvons commencer par vérifier que la classe que nous venons d'ajouter est conforme au reste des datas. Pour cela, nous pouvons utiliser une simple fonction de comptage :

```
20 print(df[df['class'] == 'Normal']['Target'].value_counts())
```

2. ANALYSE DES DONNÉES

```
0      8851  
Name: Target , dtype: int64
```

```
22 print(df[df['class'] == 'No Lung Opacity / Not Normal']['Target'].  
        value_counts())
```

```
0      11821  
Name: Target , dtype: int64
```

```
24 print(df[df['class'] == 'Lung Opacity']['Target'].value_counts())
```

```
1      9555  
Name: Target , dtype: int64
```

Les données sont cohérentes puisque les labels *Normal* et *No Lung Opacity / Not Normal* sont bien associés à un *Target* de 0 (pas de pneumonie) et le label *Lung Opacity* est associé à un *Target* de 1 (présence de pneumonie).

Nous pouvons également vérifier que les patients n'ayant pas de pneumonies possèdent une seule entrée dans la table. Calculons, tout d'abord, le nombre total de patients :

```
26 print('Nombre de patients :', len(df['patientId'].drop_duplicates()))
```

```
Nombre de patients : 26684
```

Nous retrouvons bien ici le nombre de fichiers contenus dans le dossier contenant les fichiers DICOM. Nous cherchons maintenant à savoir combien de fois un patient apparaît dans le tableau :

```
28 print(df['patientId'].value_counts().head())
```

76f71a93-8105-4c79-a010-0cf86f0061a	4
349f10b4-dc3e-4f3f-b2e4-a5b81448ce87	4
8dc8e54b-5b05-4dac-80b9-fa48878621e2	4
7d674c82-5501-4730-92c5-d241fd6911e7	4
32408669-c137-4e8d-bd62-fe8345b40e73	4

```
Name: patientId , dtype: int64
```

Un même patient apparaît donc bien plusieurs fois et pour vérifier qu'il a bien plusieurs opacités, nous pouvons imprimer le détail des lignes où il apparaît. Par exemple, pour le premier patient possédant 4 lignes dans la table :

```
30 print(df[df['patientId']=='76f71a93-8105-4c79-a010-0cf86f0061a'])
```

2. ANALYSE DES DONNÉES

	patientId	x	y	width	height	T		class
14312	76...061a	619.0	488.0	127.0	155.0	1	Lung	Opac
14316	76...061a	673.0	303.0	106.0	145.0	1	Lung	Opac
14320	76...061a	314.0	461.0	91.0	128.0	1	Lung	Opac
14324	76...061a	348.0	303.0	81.0	91.0	1	Lung	Opac

Ce patient possède effectivement 4 opacités différentes. Regardons maintenant combien de patients apparaissent dans la table une seule fois, deux fois, etc.

```
32 print(df['patientId'].value_counts().value_counts())
1    23286
2     3266
3      119
4       13
Name: patientId , dtype: int64
```

La majorité des patients apparaissent une seule fois dans la table. En sommant la deuxième colonne nous retrouvons bien le nombre de patients total, à savoir 26684. Nous pouvons finalement vérifier que les patients n'ayant pas de pneumonies ont une seule entrée dans la table.

```
34 print(df[df['class']!='Lung Opacity']['patientId'].value_counts().
         value_counts())
1    20672
Name: patientId , dtype: int64
```

Les patients n'ayant pas de pneumonie apparaissent bien une seule fois dans la table. Enfin, nous pouvons terminer ces vérifications en regardant si un patient possède bien une seule classe, c'est à dire, qu'un patient ne peut pas être atteint à la fois d'une pneumonie et d'autres pathologies (bien qu'en réalité ce doit être possible). Pour cela, on regroupe les patients par classe puis on utilise deux fois la fonction *nunique*, qui retourne le nombre de valeurs distinctes.

```
36 print(df.groupby('patientId')['class'].nunique().nunique())
1
```

Chaque patient appartient à une et une seule classe. Ceci termine nos vérifications sur notre table *df*.

Intéressons-nous maintenant au dossier contenant les fichiers au format *.dcm* aussi appelé format *DICOM*¹ (Digital Imaging and COmmunications in Medicine). Ce format est utilisé en médecine pour enregistrer des images numérisées issues d'IRM, d'échographies ou autres. Ce qui différencie les DICOM des formats classiques de

1. <https://www.dicomstandard.org/>

2. ANALYSE DES DONNÉES

stockage d'images est qu'ils peuvent contenir des informations supplémentaires sur le patient et sur la radiographie.

Le dossier contient 26684 fichiers, ce qui correspond bien au nombre de patients. Pour ouvrir un fichier au format DICOM, nous utilisons la bibliothèque *pydicom*, par exemple :

```
38 import pydicom
39 Id = df['patientId'][0]
40 dicomFile = path + 'stage_2_train_images/%s.dcm' % Id
41 dicomData = pydicom.read_file(dicomFile)
42 print(type(dicomData))
43 print(dicomData)
```

	<class 'pydicom.dataset.FileDataset'>
(0008,0005)	Specific Character Set CS: 'ISO_IR 100'
(0008,0016)	SOP Class UID UI: Sec. Capt. Img Stor.
(0008,0018)	SOP Instance UID UI: 1.2...485.775526
(0008,0020)	Study Date DA: '19010101'
(0008,0030)	Study Time TM: '000000.00'
(0008,0050)	Accession Number SH: ''
(0008,0060)	Modality CS: 'CR'
(0008,0064)	Conversion Type CS: 'WSD'
(0008,0090)	Ref. Physician's Name PN: ''
(0008,103e)	Series Description LO: 'view: PA'
(0010,0010)	Patient's Name PN: '0004c...bddd6'
(0010,0020)	Patient ID LO: '0004c...bddd6'
(0010,0030)	Patient's Birth Date DA: ''
(0010,0040)	Patient's Sex CS: 'F'
(0010,1010)	Patient's Age AS: '51'
(0018,0015)	Body Part Examined CS: 'CHEST'
(0018,5101)	View Position CS: 'PA'
(0020,000d)	Study Instance UID UI: 1.2...85.775525
(0020,000e)	Series Instance UID UI: 1.2...85.775524
(0020,0010)	Study ID SH: ''
(0020,0011)	Series Number IS: "1"
(0020,0013)	Instance Number IS: "1"
(0020,0020)	Patient Orientation CS: ''
(0028,0002)	Samples per Pixel US: 1
(0028,0004)	Photometric Interp. CS: 'MONOCHROME2'
(0028,0010)	Rows US: 1024
(0028,0011)	Columns US: 1024
(0028,0030)	Pixel Spacing DS: [0.143, 0.143]
(0028,0100)	Bits Allocated US: 8
(0028,0101)	Bits Stored US: 8
(0028,0102)	High Bit US: 7
(0028,0103)	Pixel Representation US: 0
(0028,2110)	Lossy Img Compr. CS: '01'

2. ANALYSE DES DONNÉES

```
(0028,2114) Lossy Img Compr. Meth. CS: 'ISO_10918_1'  
(7fe0,0010) Pixel Data OB: Array of 142006 elmts
```

Le fichier DICOM importé est stocké dans une instance d'une classe de la bibliothèque *pydicom*. Cette classe est une collection de dictionnaires contenant chacune des informations sur le patient. Parmi les différentes informations contenues dans le DICOM, certaines me semblent particulièrement intéressantes comme l'âge du patient, son sexe ainsi que sa position. L'image numérisée est contenue dans un tableau associé à la clé *Pixel Data*. Examinons ce tableau :

```
45 dicomImage = dicomData.pixel_array  
46 print(type(dicomImage))  
47 print(dicomImage.dtype)  
48 print(dicomImage.shape)
```

```
<class 'numpy.ndarray'>  
uint8  
(1024, 1024)
```

Le tableau est converti en un tableau *numpy*, de taille 1024×1024 dont chaque pixel est codé sur 8 bits (=1 octet = 256 couleurs), il s'agit donc d'une image en noir et blanc. Pour visualiser une image, nous pouvons utiliser la bibliothèque *matplotlib* :

```
50 import matplotlib.pyplot as plt  
51 fig, ax = plt.subplots()  
52 ax.imshow(dicomImage, cmap='gray')  
53 plt.axis('off')  
54 plt.show()
```



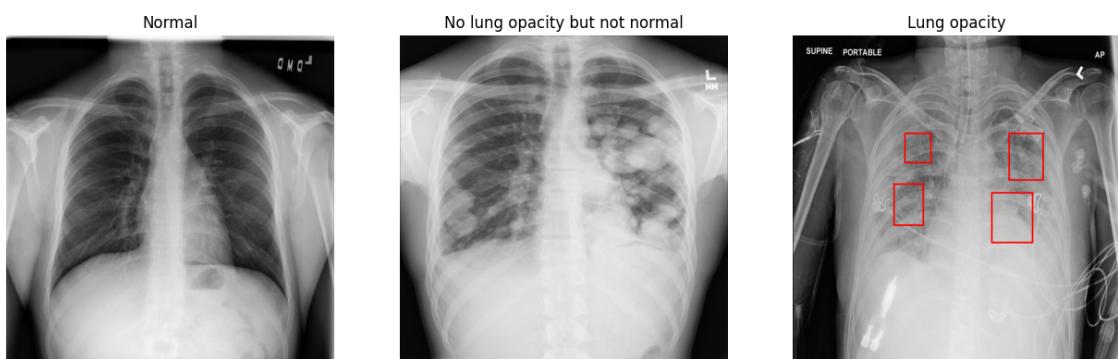
Les DICOM sont connectés à notre table par le *patientId*, ainsi il est possible d'inclure des informations contenues dans la table comme la classe à une image, ou même les boîtes d'encadrement des opacités en cas de pneumonie. Tout d'abord, on crée une fonction qui à partir d'un identifiant de patient et d'une instance de la classe *Axes* retourne une image DICOM :

2. ANALYSE DES DONNÉES

```
56 def plotDICOM(patientId, ax):
57     dicomFile = path + 'stage_2_train_images/' + patientId + '.dcm'
58     dicomData = pydicom.read_file(dicomFile)
59     dicomImage = dicomData.pixel_array
60     if df[df['patientId']==patientId]['Target'].values[0] == 1:
61         dicomImage = np.stack([dicomImage] * 3, axis=2)
62         for index, row in df[df['patientId']==patientId].iterrows():
63             epaisseur = 5
64             x1 = int(row['x'])
65             x2 = int(x1 + row['width'])
66             y1 = int(row['y'])
67             y2 = int(y1 + row['height'])
68             dicomImage[y1:y1 + epaisseur, x1:x2] = [255,0,0]
69             dicomImage[y2:y2 + epaisseur, x1:x2] = [255,0,0]
70             dicomImage[y1:y2, x1:x1 + epaisseur] = [255,0,0]
71             dicomImage[y1:y2+ epaisseur, x2:x2 + epaisseur] =
72                 [255,0,0]
73             ax.imshow( dicomImage,cmap='gray')
74             ax.axis('off')
75             return ax
```

Utilisons cette fonction pour représenter la radiographie d'un patient appartenant à chacune des 3 classes :

```
76 fig, ax = plt.subplots(nrows=1,ncols=3)
77 ax[0] = plotDICOM('003d8fa0-6bf1-40ed-b54c-ac657f8495c5',ax[0])
78 ax[0].set_title('Normal')
79 ax[1] = plotDICOM('00322d4d-1c29-4943-afc9-b6754be640eb',ax[1])
80 ax[1].set_title('No lung opacity but not normal')
81 ax[2] = plotDICOM('1bf08f3b-a273-4f51-bafa-b55ada2c23b5',ax[2])
82 ax[2].set_title('Lung opacity')
83 plt.show()
```



Pour finir, nous allons intégrer l'âge, le sexe et la position des patients contenus dans les fichiers DICOM dans notre table.

2. ANALYSE DES DONNÉES

```
85 age = []
86 sexe = []
87 pos = []
88 for index, row in df.iterrows():
89     Id = df['patientId'][index]
90     dicomFile = path + 'stage_2_train_images/%s.dcm' % Id
91     dicomData = pydicom.read_file(dicomFile)
92     age.append(dicomData['PatientAge'].value)
93     sexe.append(dicomData['PatientSex'].value)
94     pos.append(dicomData['ViewPosition'].value)
95
96 df['Age'] = age
97 df['Sexe'] = sexe
98 df['Position'] = pos
```

Remarque : Nous aurions pu à ce stade remplacer notre table par un dictionnaire afin d'optimiser le stockage. En effet, les dictionnaires éviteraient de stocker plusieurs lignes pour un même patient. Néanmoins, les dictionnaires ne sont pas adaptés aux bibliothèques graphiques classiques. En revanche, il faudra être vigilant quant à l'utilisation des données de la table afin de tenir compte des répétitions de lignes qui pourraient fausser certaines études.

Vérifions le contenu de cette table :

```
100 print(df.head())
101 print(df.info())
```

	patientId	x	...	class	Age	Sexe	Position
0	00...d6	NaN	...	No Lung	51	F	PA
1	00...cd	NaN	...	No Lung	48	F	PA
2	00...eb	NaN	...	No Lung	19	M	AP
3	00...c5	NaN	...	Normal	28	M	PA
4	00...b4	264.0	...	Lung Opac	32	F	AP

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30227 entries, 0 to 30226
Data columns (total 10 columns):
 # Column Non-Null Count Dtype
 -- -- -- --
 0 patientId 30227 non-null object
 1 x 9555 non-null float64
 2 y 9555 non-null float64
 3 width 9555 non-null float64
 4 height 9555 non-null float64
 5 Target 30227 non-null int64
 6 class 30227 non-null object
 7 Age 30227 non-null int64
 8 Sexe 30227 non-null object

2. ANALYSE DES DONNÉES

```
9    Position    30227 non-null object
dtypes: float64(4), int64(2), object(4)
memory usage: 2.3+ MB
None
```

Maintenant que les données des fichiers DICOM sont entrées dans notre table, nous pouvons vérifier aisément le nombre de valeurs manquantes :

```
103 print('Nombre de valeurs NaN de la colonne Age :', len(df) - df['Age']
      ].count())
104 print('Nombre de valeurs NaN de la colonne Sexe :', len(df) - df['
      Sexe'].count())
105 print('Nombre de valeurs NaN de la colonne Position :', len(df) - df
      ['Position'].count())
```

```
Nombre de valeurs NaN de la colonne Age : 0
Nombre de valeurs NaN de la colonne Sexe : 0
Nombre de valeurs NaN de la colonne Position : 0
```

Bien heureusement pour nous, il n'y a pas de valeurs manquantes. Nous pouvons également vérifier le nombre de valeurs différentes apparaissant dans chacune de ces colonnes :

```
107 print('Nombre de valeurs differentes de la colonne Age :', df['Age'].nunique())
108 print('Nombre de valeurs differentes de la colonne Sexe :', df['Sexe'].nunique())
109 print('Nombre de valeurs differentes de la colonne Position :', df['Position'].nunique())
```

```
Nombre de valeurs differentes de la colonne Age : 97
Nombre de valeurs differentes de la colonne Sexe : 2
Nombre de valeurs differentes de la colonne Position : 2
```

Il y a 97 valeurs différentes dans la colonne *Age*, ce qui donne un large panel. Et comme nous pouvions nous y attendre, seulement 2 valeurs pour les colonnes *Sexe* ('M' ou 'F') et *Position* ('PA' ou 'AP'). Ceci en termine avec l'identification et la préparation des variables.

2.2 Analyse univariée

Cette première étape d'analyse consiste à explorer les variables une à une. Selon la nature de la variable, les méthodes sont différentes. Premièrement, dans le cas de variables continues, on cherche à comprendre la tendance centrale (moyenne, médiane, mode ...) ainsi que la variabilité des variables (variance, écart-type, écart interquartile ...). Cela permet de mettre en évidence les valeurs manquantes ou aberrantes. Pour représenter ces indicateurs, des histogrammes et des boites à moustaches sont

2. ANALYSE DES DONNÉES

de bons outils. Deuxièmement, pour des variables catégorielles, on effectue généralement une analyse fréquentielle pour comprendre la distribution de chaque catégorie. Les graphiques à barres sont de bons outils pour visualiser cette distribution.

Dans notre étude, nous avons identifié 9 variables, à savoir x , y , $width$, $height$, $Target$, $class$, Age , $Sexe$ et $Position$. Pour chacune d'entre elles, nous allons faire une analyse univariée selon sa nature. Néanmoins, comme nous l'avions précisé dans la section précédente, nous devons être vigilant à ne pas utiliser de l'information dupliquée qui fausserait les résultats. Pour cette raison, nous allons créer une table temporaire dans laquelle les lignes dont le $patientId$ est dupliqué sont supprimées :

```
111 dfNoDup = df.drop_duplicates('patientId', keep='first')
```

2.2.1 Analyse de la variable x

La variable x représente l'abscisse du coin supérieur gauche de la boîte d'encadrement d'une opacité, il s'agit d'une variable continue. Représentons cette variable sur un histogramme et sur une boite à moustache. Mais avant cela, nous rappelons qu'une boite à moustache est représentée par une boite dont les dimensions correspondent aux quartiles Q_1 (valeur qui est supérieure ou égale à au moins un quart des données ordonnées) et Q_3 (valeur qui est supérieure ou égale à au moins trois quarts des données ordonnées). La médiane est représentée à l'intérieur de cette boite. Les "moustaches" sont calculées à partir de l'écart interquartile ($Q_3 - Q_1$) par les formules :

$$m_g = \max(\min(X), Q_1 - 1.5 * (Q_3 - Q_1))$$

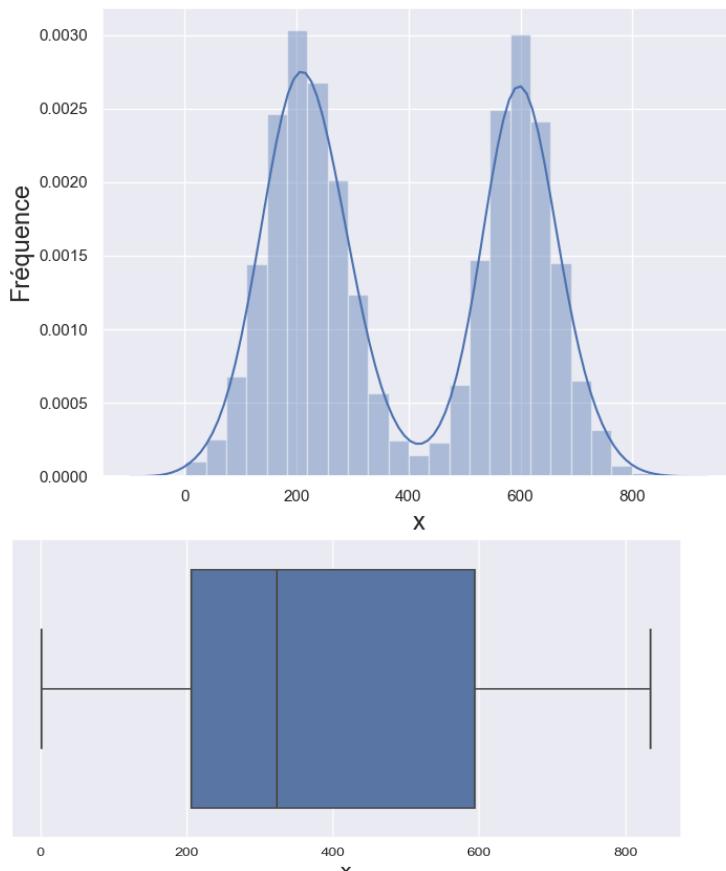
pour la moustache gauche et

$$m_d = \min(\max(X), Q_3 + 1.5 * (Q_3 - Q_1))$$

pour la moustache droite et où X représente les données d'entrée. Ces moustaches permettent de mettre en évidence des valeurs aberrantes.

```
113 # Analyse de la variable x
114 import seaborn as sns
115 plt.figure(figsize=(7.8,5.8))
116 sns.set()
117 s = sns.distplot(df['x'])
118 s.set_xlabel('x', fontsize=17)
119 s.set_ylabel('Fréquence', fontsize=17)
120 plt.show()
121
122 plt.figure(figsize=(9,5))
123 s = sns.boxplot( x="x", data=df)
124 s.set_xlabel('x', fontsize=17)
125 plt.show()
```

2. ANALYSE DES DONNÉES



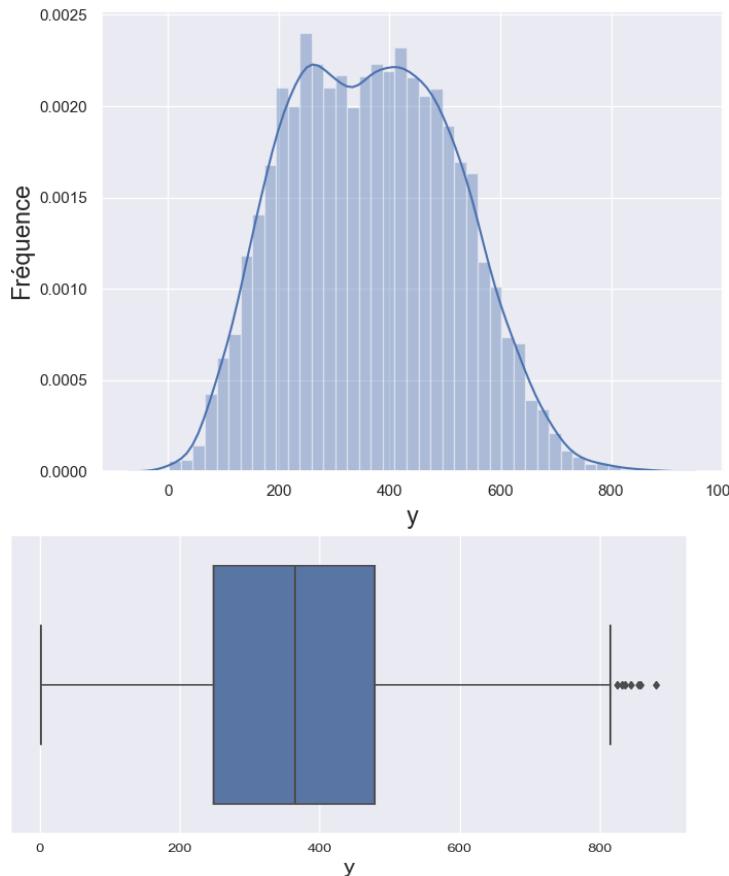
Il semble que la distribution de la variable x suit la somme de deux lois normales, symétriques d'axe $x = 400$. Néanmoins, nous pouvons voir que la médiane n'est pas centrée en 400 et se situe plutôt vers 320, par conséquent, une légère majorité des boites d'encadrement ont leurs coins supérieurs gauches dans la partie gauche de l'image.

2.2.2 Analyse de la variable y

La variable y représente l'ordonnée du coin supérieur gauche de la boîte d'encadrement d'une opacité, il s'agit également d'une variable continue. Comme la variable x , nous représentons cette variable sur un histogramme et une boîte à moustache :

```
127 # Analyse de la variable y
128 plt.figure(figsize=(7.8,5.8))
129 sns.set()
130 s = sns.distplot(df['y'])
131 s.set_xlabel('y', fontsize=17)
132 s.set_ylabel('Fréquence', fontsize=17)
133 plt.show()
134
135 plt.figure(figsize=(9,5))
136 s = sns.boxplot( x="y", data=df)
137 s.set_xlabel('y', fontsize=17)
138 plt.show()
```

2. ANALYSE DES DONNÉES



À l'instar de la variable précédente, la distribution de la variable y semble être la somme de deux lois normales, symétriques par rapport à l'axe $y = 340$. Contrairement à la variable x , la médiane est parfaitement centrée par rapport aux quartiles Q_1 et Q_3 . Notons tout de même les quelques valeurs situées à l'extérieur de la boîte à moustache. Puisque la distribution de la variable y ne semble pas aléatoire, nous pouvons considérer ces valeurs comme aberrantes. De plus, avec une valeur de plus de 800, cela signifie que l'opacité se situe sur la partie basse de l'image. Regardons, par exemple, combien de patients ont une opacité dont le coin supérieur gauche de la boîte d'encadrement à une hauteur supérieure à 850 :

```

140 patientId = df[df['y'] > 850][['patientId', 'y']].reset_index(drop=True)
141 print(df[df['y'] > 850][['patientId', 'y']])

```

	patientId	y
0	32408669-c137-4e8d-bd62-fe8345b40e73	856.0
1	72dc0bc2-8c6a-4214-b06f-862e9e8444bd	859.0
2	e821a416-aed0-411d-ab8a-7813185384b9	881.0

Trois patients ont une opacité située sur le bas de l'image, regardons ce que donnent les radiographies de ces patients :

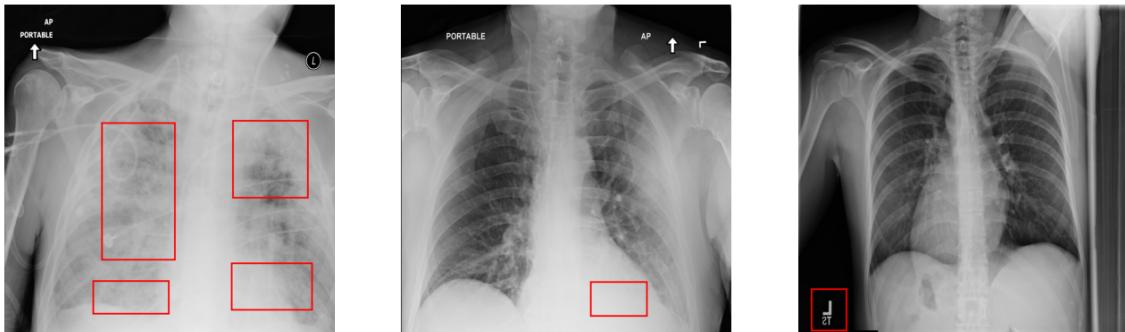
```

143 fig, ax = plt.subplots(nrows=1, ncols=len(patientId))
144 for index, row in (patientId).iterrows():

```

2. ANALYSE DES DONNÉES

```
145     ax[index] = plotDICOM(row['patientId'], ax[index])
146 plt.show()
```



Le premier patient possède 4 opacités, dont deux sur la partie inférieure de la radiographie. Le second patient, quant à lui, a une opacité située au niveau du bassin, par conséquent, il est extrêmement difficile de visualiser l'opacité. Cette radiographie pourrait donc être considérée comme une aberration. L'œil expert d'un radiologue pourrait trancher la question. Dans la dernière radiographie, l'opacité est située non pas dans les poumons, mais sur un logo, il s'agit clairement d'une aberration qu'il faudra traiter.

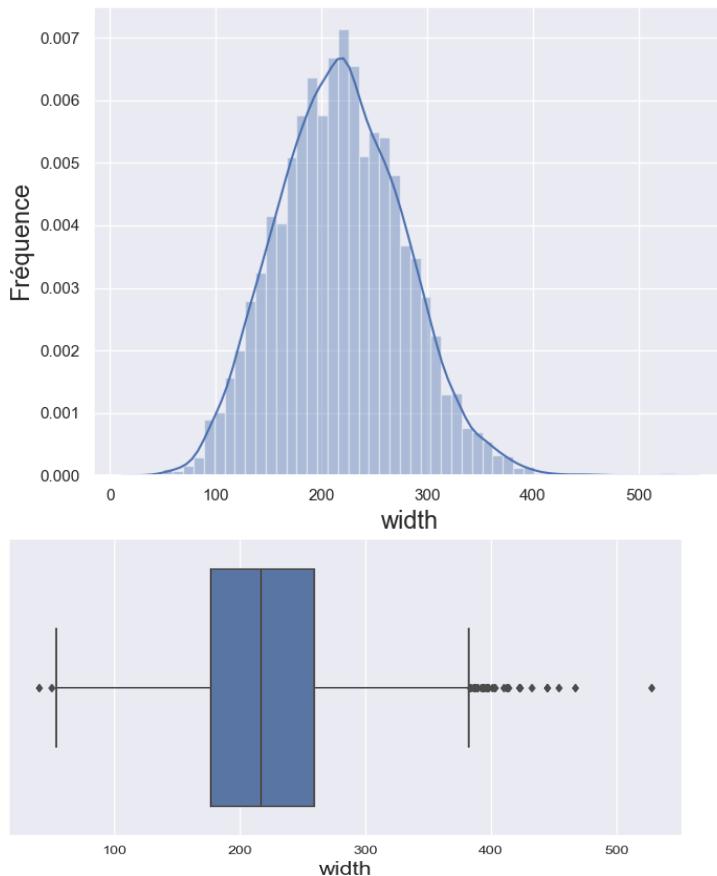
2.2.3 Analyse de la variable *width*

La variable *width* représente la largeur d'une boîte d'encadrement d'une opacité. Comme les variables précédentes, c'est une variable continue.

```
148 # Analyse de la variable width
149 plt.figure(figsize=(7.8,5.8))
150 sns.set()
151 s = sns.distplot(df['width'])
152 s.set_xlabel('width', fontsize=17)
153 s.set_ylabel('Fréquence', fontsize=17)
154 plt.show()

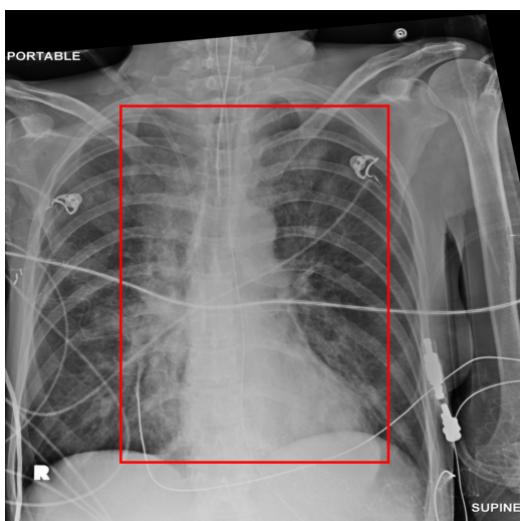
155
156 plt.figure(figsize=(9,5))
157 s = sns.boxplot( x="width", data=df)
158 s.set_xlabel('width', fontsize=17)
159 plt.show()
```

2. ANALYSE DES DONNÉES



D'après l'histogramme, la variable suit une loi de distribution normale centrée autour de 220. Nous pouvons observer plusieurs valeurs situées à l'extérieur de la moustache droite, dont une particulièrement éloignée. Visualisons la radiographie correspondante à cette valeur :

```
161 fig, ax = plt.subplots()  
162 patientId = df[df['width'] > 500][['patientId']]  
163 ax = plotDICOM(patientId.iloc[0]['patientId'], ax)  
164 plt.show()
```



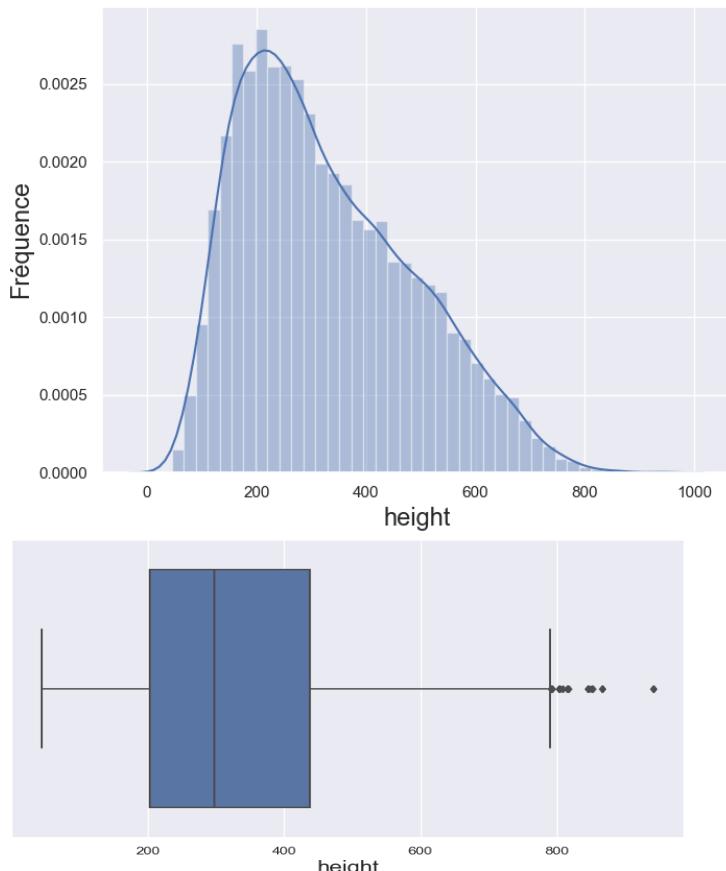
2. ANALYSE DES DONNÉES

N'étant pas spécialiste, il m'est difficile de juger si cette valeur est aberrante ou non.

2.2.4 Analyse de la variable *height*

Après avoir analysé la largeur des boîtes d'encadrement des opacités, nous nous intéressons à la hauteur au travers de la variable *height* de type continu.

```
166 # Analyse de la variable height
167 plt.figure(figsize=(7.8,5.8))
168 sns.set()
169 s = sns.distplot(df['height'])
170 s.set_xlabel('height', fontsize=17)
171 s.set_ylabel('Fréquence', fontsize=17)
172 plt.show()
173
174 plt.figure(figsize=(9,5.5))
175 s = sns.boxplot( x="height", data=df)
176 s.set_xlabel('height', fontsize=17)
177 plt.show()
```



Il est difficile de juger ici que la loi de distribution de la variable *height* suit une loi normale, après avoir atteint un pic autour d'une hauteur de taille 200, la distribution décroît lentement. Ce phénomène est également visible dans la boîte à moustache, la médiane étant plus proche du quartile Q_1 . Les valeurs situées à l'extérieur de la

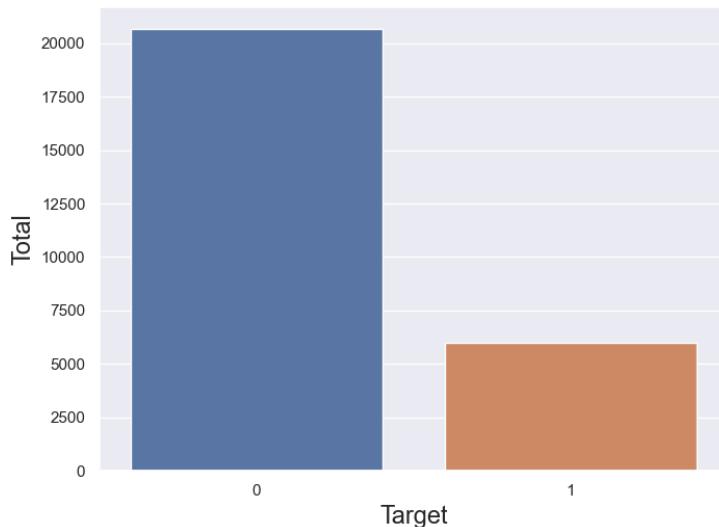
2. ANALYSE DES DONNÉES

moustache droite sont également causées par cette décroissante lente.

2.2.5 Analyse de la variable *Target*

Contrairement aux variables précédentes, la variable *Target* indiquant si le patient est atteint d'une pneumonie ou non est de type catégoriel. Un graphique à barres permet de visualiser chaque catégorie :

```
179 # Analyse de la variable Target
180 plt.figure(figsize=(7.8,5.8))
181 s = sns.countplot(x='Target', data=dfNoDup)
182 s.set_xlabel('Target', fontsize=17)
183 s.set_ylabel('Total', fontsize=17)
184 plt.show()
```



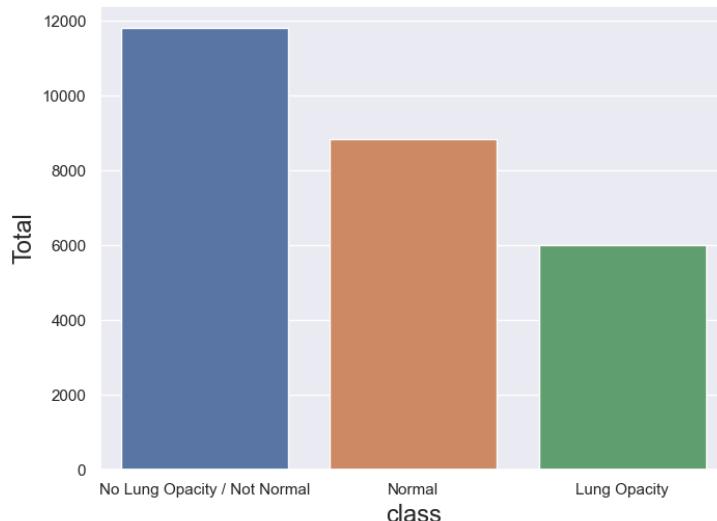
Parmi l'ensemble des données, les patients ayant une pneumonie sont environ trois fois moins nombreux.

2.2.6 Analyse de la variable *class*

La variable *class* est catégorielle :

```
186 # Analyse de la variable class
187 plt.figure(figsize=(7.8,5.8))
188 s = sns.countplot(x='class', data=dfNoDup)
189 s.set_xlabel('class', fontsize=17)
190 s.set_ylabel('Total', fontsize=17)
191 plt.show()
```

2. ANALYSE DES DONNÉES



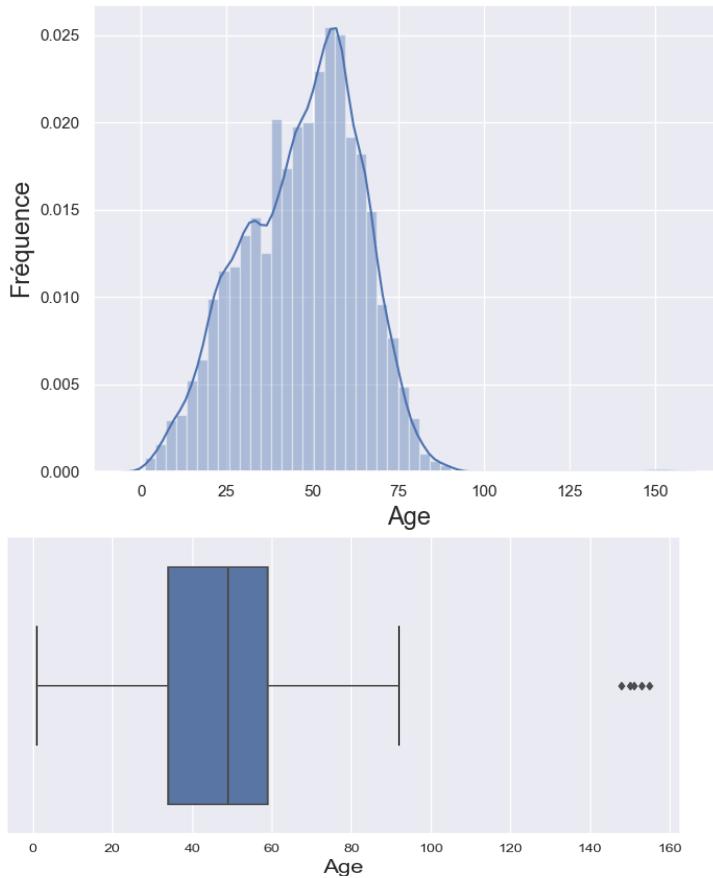
La classe la plus représentée avec un nombre légèrement inférieur à 12000 est la classe où les patients possèdent une pathologie différente de la pneumonie. Les patients ayant une radiographie normale sont environ 8800. La dernière catégorie au nombre de 6000 correspond aux patients atteints d'une pneumonie.

2.2.7 Analyse de la variable *Age*

La variable *Age* est de type continu.

```
193 # Analyse de la variable Age
194 plt.figure(figsize=(7.8,5.8))
195 sns.set()
196 s = sns.distplot(dfNoDup['Age'])
197 s.set_xlabel('Age', fontsize=17)
198 s.set_ylabel('Fréquence', fontsize=17)
199 plt.show()
200
201 plt.figure(figsize=(9,5))
202 s = sns.boxplot( x="Age", data=dfNoDup)
203 s.set_xlabel('Age', fontsize=17)
204 plt.show()
```

2. ANALYSE DES DONNÉES



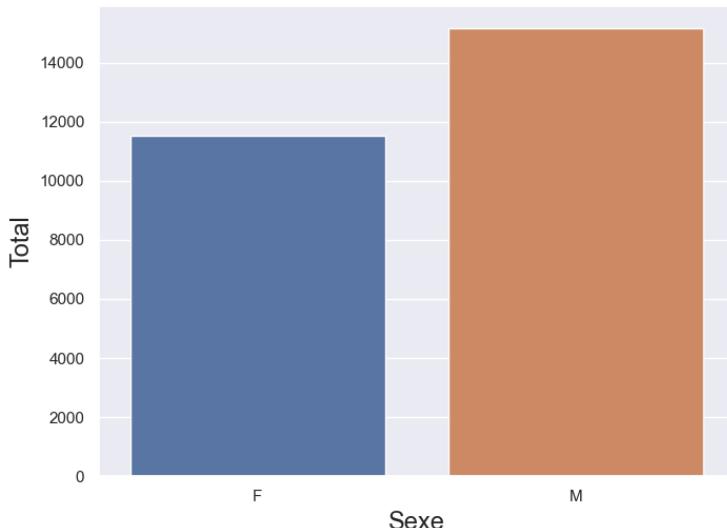
Nous pouvons observer que la majorité des patients sont âgés de plus de 50 ans. Environ 25% des patients ont moins de 30 ans et environ 25% ont plus de 60 ans. Nous observons 5 valeurs aberrantes indiquant un âge supérieur à 140 ans.

2.2.8 Analyse de la variable *Sexe*

La variable *Sexe* est de type catégorie.

```
206 # Analyse de la variable Sexe
207 plt.figure(figsize=(7.8,5.8))
208 s = sns.countplot(x='Sexe', data=dfNoDup)
209 s.set_xlabel('Sexe', fontsize=17)
210 s.set_ylabel('Total', fontsize=17)
211 plt.show()
```

2. ANALYSE DES DONNÉES

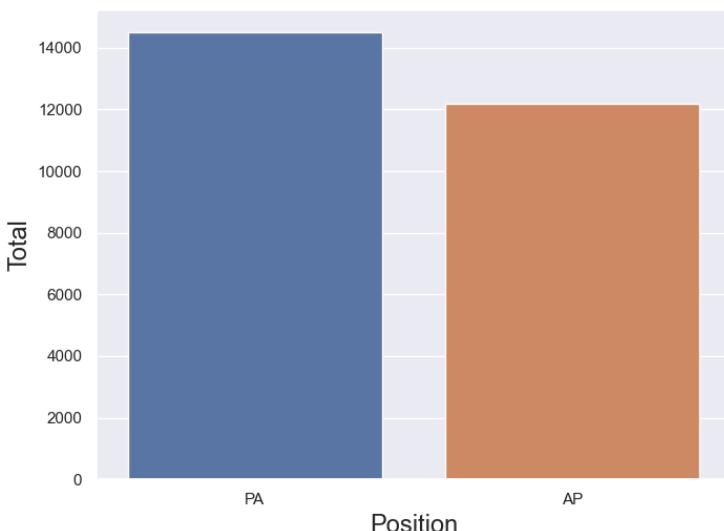


On observe une majorité d'hommes avec un total d'environ 15000. Les femmes sont représentées par un peu moins de 12000 patientes.

2.2.9 Analyse de la variable *Position*

La variable *Position* représente la position dans laquelle le patient était au moment de la radiographie. Deux positions sont possibles, la position antéro-postérieure (AP) et la position postéro-antérieur (PA). Il s'agit donc d'une variable catégorielle.

```
213 # Analyse de la variable Position
214 plt.figure(figsize=(7.8,5.8))
215 s = sns.countplot(x='Position', data=dfNoDup)
216 s.set_xlabel('Position', fontsize=17)
217 s.set_ylabel('Total', fontsize=17)
218 plt.show()
```



On observe des valeurs assez proches l'une de l'autre avec environ 14000 radiographies effectuées en position PA et 12000 en position AP.

2.3 Analyse multivariée

L'analyse multivariée permet de mettre en évidence des relations entre des combinaisons de variables. Généralement, l'EDA se contente de l'analyse bivariée qui met en évidence la relation entre deux variables. L'analyse bivariée peut se faire pour toute combinaison de variables, à savoir, continue/continue, catégorielle/catégorielle ou catégorielle/continue. Néanmoins, selon la combinaison les méthodes et outils de visualisation sont différents.

Pour les combinaisons de type continu/continu, construire un diagramme de dispersion est une façon astucieuse à découvrir la relation entre les deux variables. Le motif du nuage de points indique la façon dont les variables interagissent entre elles. La relation peut être linéaire ou non linéaire. Néanmoins, un graphique n'indique pas l'intensité de la relation entre elles. Pour cela, nous pouvons calculer la corrélation par la formule :

$$\text{Cor}(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}$$

La corrélation varie entre -1 et +1, d'une corrélation linéaire négative parfaite à une corrélation linéaire positive parfaite en passant par 0 (aucune corrélation).

Pour des combinaisons de variable catégorielle/catégorielle, les histogrammes à valeurs superposées ou empilées sont de bons outils d'analyse. Ils permettent de visualiser le nombre ou le pourcentage d'observations disponibles dans chaque combinaison de catégories de lignes et de colonnes. Si cela n'est pas suffisant, on peut utiliser le test du χ^2 afin de vérifier l'absence de relation entre les deux variables.

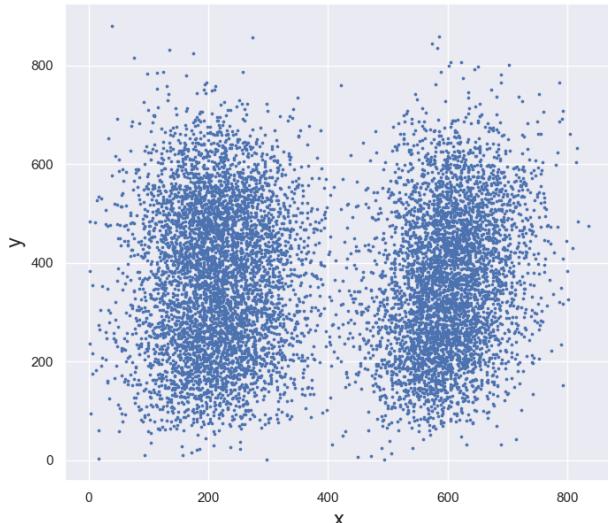
Enfin, pour les combinaisons catégorielles/continues, nous pouvons tracer des boîtes à moustaches ou des *violin plot* pour chaque niveau de variables catégorielles. Pour examiner la signification statistique, nous pouvons effectuer un test Z ou un test T. Nous allons maintenant faire l'analyse de variables de notre table. Je ne montrerais qu'une partie des résultats, ceux qui me semble les plus pertinents.

2.3.1 Combinaisons de type continu/continu

Intéressons-nous dans un premier temps à la relation entre la variable x et la variable y :

```
220 # x-y
221 plt.figure(figsize=(8,7))
222 sns.set()
223 s = sns.scatterplot(x="x", y="y", linewidth=0, s=7, data=df)
224 s.set_xlabel('x', fontsize=17)
225 s.set_ylabel('y', fontsize=17)
226 plt.show()
```

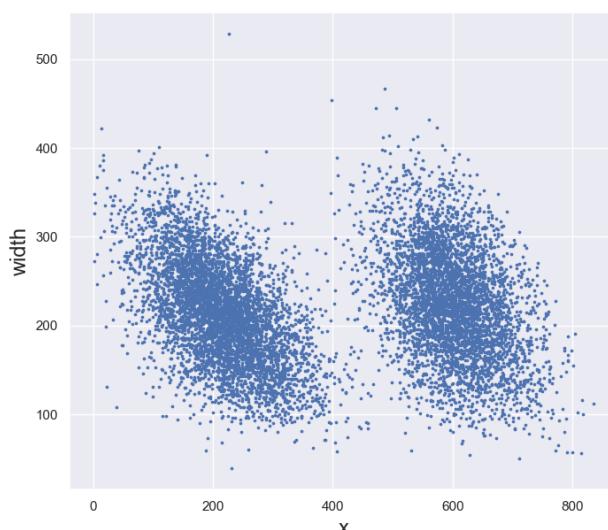
2. ANALYSE DES DONNÉES



Nous pouvons observer deux clusters correspondant aux deux poumons. Notons qu'il sera difficile de trouver une corrélation entre la variable x et les autres variables. En effet, la recherche d'une corrélation entre 2 variables a pour objectif de mettre en évidence une relation linéaire ou non linéaire, donc il s'agit d'une relation continue. Or si l'on imagine une projection des poumons sur l'axe des abscisses, on obtient deux domaines indépendants non continus. Pour cette raison, il sera impossible de trouver une corrélation entre la variable x et les autres, néanmoins cette analyse bivariée permet de traquer les valeurs aberrantes.

La relation entre la variable x et la variable $width$ est représentée par :

```
228 # x-width
229 plt.figure(figsize=(8,7))
230 sns.set()
231 s = sns.scatterplot(x="x", y="width", linewidth=0, s=7, data=df)
232 s.set_xlabel('x', fontsize=17)
233 s.set_ylabel('width', fontsize=17)
234 plt.show()
```



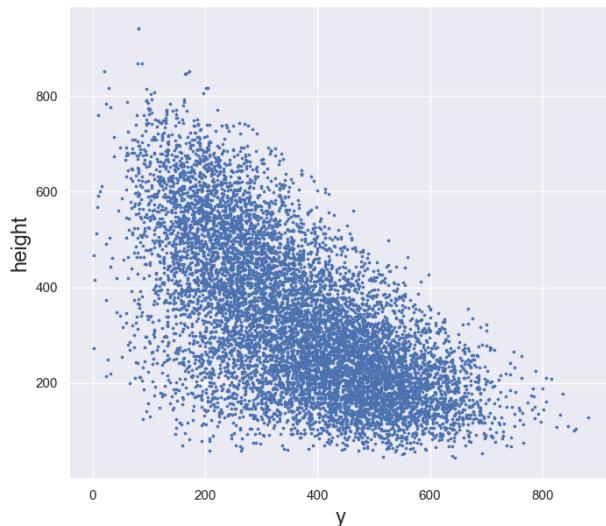
Ici aussi, nous pouvons observer deux clusters. Ce graphique est clairement utile

2. ANALYSE DES DONNÉES

pour mettre en évidence certaines valeurs aberrantes.

Regardons maintenant la relation entre les variables y et $height$:

```
236 # y-height
237 plt.figure(figsize=(8,7))
238 sns.set()
239 s = sns.scatterplot(x="y", y="height", linewidth=0, s=7, data=df)
240 s.set_xlabel('y', fontsize=17)
241 s.set_ylabel('height', fontsize=17)
242 plt.show()
```

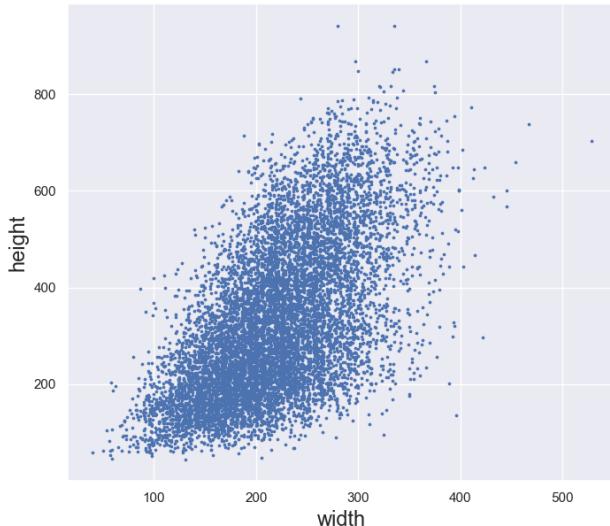


Ici, nous pouvons observer une bonne corrélation linéaire entre ces deux variables. Contrairement à la variable x , la variable y n'a qu'un seul domaine de définition. Pour s'en convaincre, il suffit d'imaginer une projection des poumons sur l'axe des ordonnées.

Pour finir, intéressons-nous à la relation entre la largeur et la hauteur des boîtes d'encadrement des opacités :

```
244 # width-height
245 plt.figure(figsize=(8,7))
246 sns.set()
247 s = sns.scatterplot(x="width", y="height", linewidth=0, s=7, data=df)
248 s.set_xlabel('width', fontsize=17)
249 s.set_ylabel('height', fontsize=17)
250 plt.show()
```

2. ANALYSE DES DONNÉES



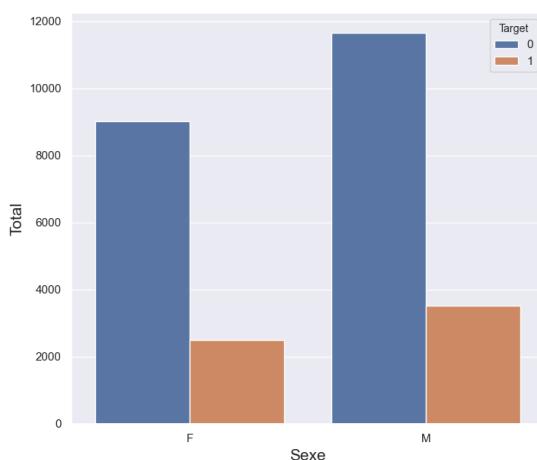
Ici aussi, on peut observer une certaine corrélation linéaire entre les deux variables.

2.3.2 Combinaisons de type catégoriel/catégoriel

Nous allons maintenant porter notre attention sur les combinaisons de variables catégorielles/catégorielles. Notons que pour cette analyse, nous travaillons sur la table dont les lignes où un patient apparaît plusieurs fois ont été supprimées. De plus, nous n'allons pas présenté les résultats pour la relation entre les variables *Target* et *class* puisqu'il est évident qu'il y a une forte corrélation.

Commençons par analyser la relation entre les variables *Sexe* et *Target* :

```
252 # sexe-target
253 plt.figure(figsize=(8,7))
254 sns.set()
255 s = sns.countplot(x = 'Sexe', hue = 'Target', data = dfNoDup);
256 s.set_xlabel('Sexe', fontsize=15)
257 s.set_ylabel('Total', fontsize=15)
258 plt.show()
```

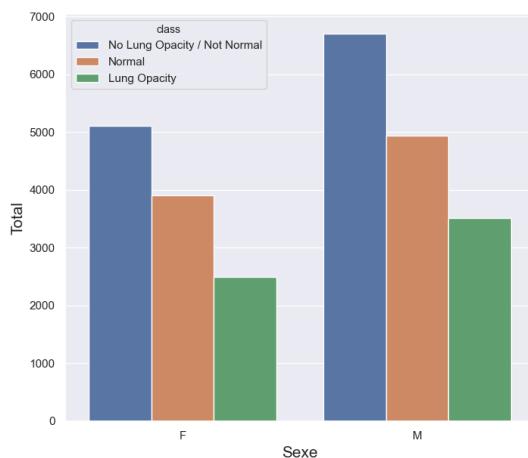


Quelque soit le sexe, le ratio entre le nombre de personnes ayant une pneumonie et celles qui n'ont pas est presque identique, ce qui signifie qu'il y a peu de corrélations entre ces variables. On retrouvera ce même comportement dans le graphique suivant

2. ANALYSE DES DONNÉES

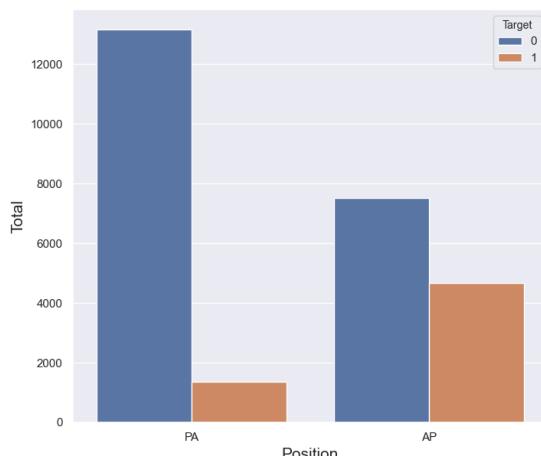
où cette fois nous avons représenté le nombre d'hommes et de femmes pour chaque *class*.

```
260 # sexe-class
261 plt.figure(figsize=(8,7))
262 sns.set()
263 s = sns.countplot(x = 'Sexe', hue = 'class', data = dfNoDup);
264 s.set_xlabel('Sexe', fontsize=15)
265 s.set_ylabel('Total', fontsize=15)
266 plt.show()
```



Intéressons maintenant à l'analyse de la corrélation entre la variable *Position* et *Target* :

```
268 # position-target
269 plt.figure(figsize=(8,7))
270 sns.set()
271 s = sns.countplot(x = 'Position', hue = 'Target', data = dfNoDup);
272 s.set_xlabel('Position', fontsize=15)
273 s.set_ylabel('Total', fontsize=15)
274 plt.show()
```

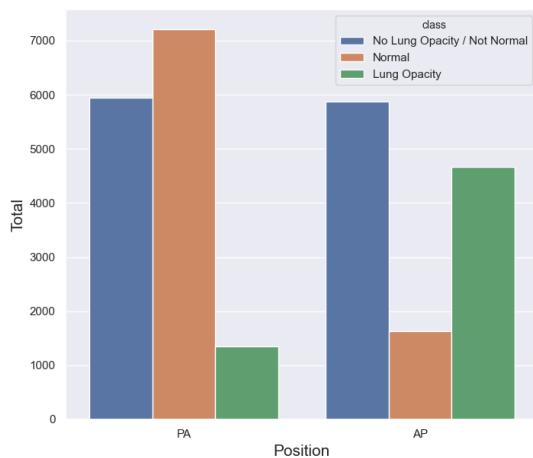


Ici nous pouvons observer que le ratio entre les patients atteints d'une pneumonie

2. ANALYSE DES DONNÉES

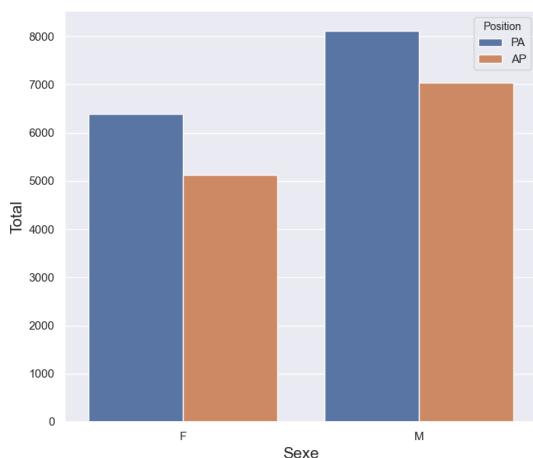
ou non n'est pas le même selon la position. Ce qui laisse à penser une corrélation entre ces variables. Le graphique suivant aboutit aux mêmes constatations.

```
276 # position-class
277 plt.figure(figsize=(8,7))
278 sns.set()
279 s = sns.countplot(x = 'Position', hue = 'class', data = dfNoDup);
280 s.set_xlabel('Position', fontsize=15)
281 s.set_ylabel('Total', fontsize=15)
282 plt.show()
```



Pour en terminer avec ce type de combinaison, nous regardons la relation entre les variables *Sexe* et *Position* :

```
284 # sexe-position
285 plt.figure(figsize=(8,7))
286 sns.set()
287 s = sns.countplot(x = 'Sexe', hue = 'Position', data = dfNoDup);
288 s.set_xlabel('Sexe', fontsize=15)
289 s.set_ylabel('Total', fontsize=15)
290 plt.show()
```



Quel que soit le sexe, le ratio est quasiment le même quelle que soit la position. Il y

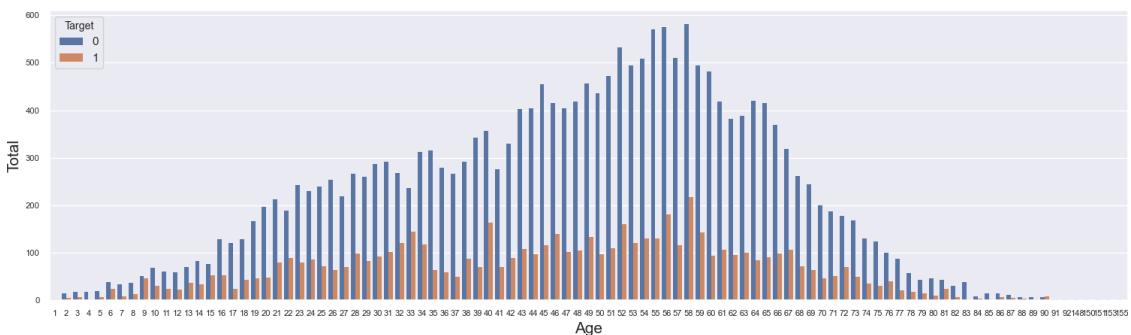
2. ANALYSE DES DONNÉES

a par conséquent une très faible corrélation entre ces variables.

2.3.3 Combinaisons de type catégoriel/continu

Dans cette partie, nous allons principalement regarder la relation entre l'âge et les autres variables catégorielles. Premièrement, regardons la relation entre l'âge et la variable *Target* :

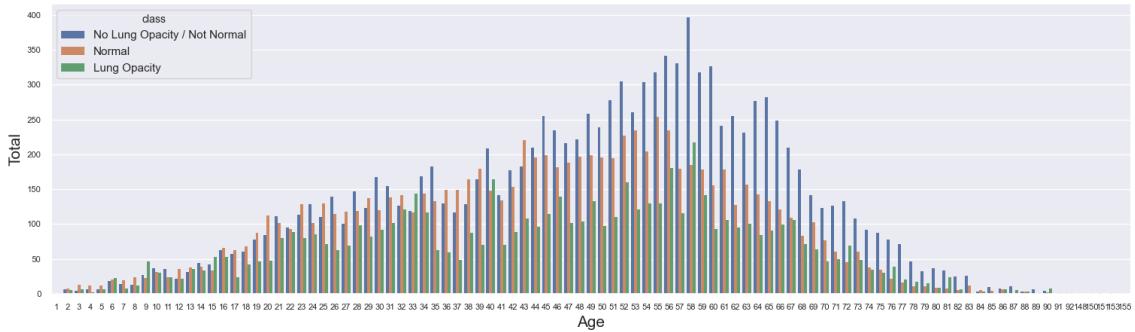
```
292 # age-target
293 plt.figure(figsize=(25,5))
294 sns.set()
295 s = sns.countplot(x = 'Age', hue = 'Target', linewidth=0,data =
296     dfNoDup);
297 s.set_xlabel('Age',fontsize=15)
298 s.set_ylabel('Total',fontsize=15)
299 s.tick_params(labelsize=8)
300 plt.show()
```



Les amplitudes du graphique pour des patients atteints d'une pneumonie ont la même forme que les amplitudes pour des patients n'ayant pas de pneumonie, à un coefficient près bien sûr. Il y a peu de corrélation entre ces deux variables, et par conséquent, il y a également peu de corrélation entre l'âge et la variable *class* :

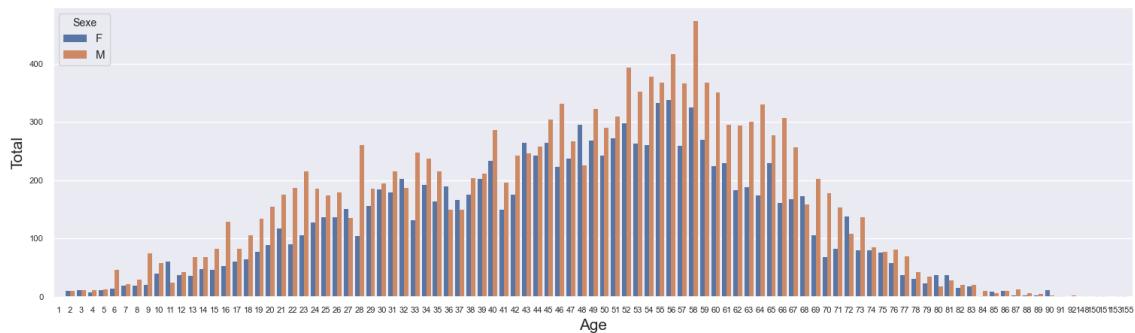
```
301 # age-class
302 plt.figure(figsize=(25,5))
303 sns.set()
304 s = sns.countplot(x = 'Age', hue = 'class', linewidth=0,data =
305     dfNoDup);
306 s.set_xlabel('Age',fontsize=15)
307 s.set_ylabel('Total',fontsize=15)
308 s.tick_params(labelsize=8)
309 plt.show()
```

2. ANALYSE DES DONNÉES



Regardons s'il existe une corrélation entre l'âge et le sexe des patients :

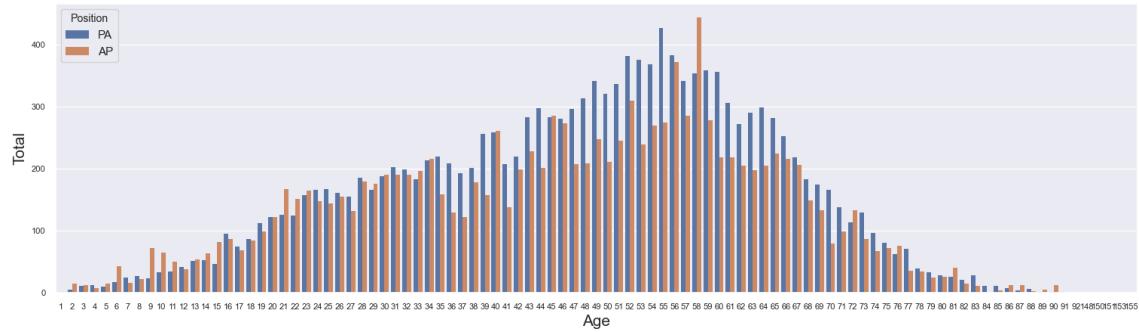
```
310 # age-sexe
311 plt.figure(figsize=(25,5))
312 sns.set()
313 s = sns.countplot(x = 'Age', hue = 'Sexe', linewidth=0,data = dfNoDup);
314 s.set_xlabel('Age',fontsize=15)
315 s.set_ylabel('Total',fontsize=15)
316 s.tick_params(labelsize=8)
317 plt.show()
```



Comme nous pouvions nous y attendre, il y a peu de corrélations entre ces variables.
Regardons pour terminer, la relation entre l'âge et la position :

```
319 # age-position
320 plt.figure(figsize=(25,5))
321 sns.set()
322 s = sns.countplot(x = 'Age', hue = 'Position', linewidth=0,data =
323     dfNoDup);
324 s.set_xlabel('Age',fontsize=15)
325 s.set_ylabel('Total',fontsize=15)
326 s.tick_params(labelsize=8)
327 plt.show()
```

2. ANALYSE DES DONNÉES



Ici aussi, nous pouvons observer qu'il y a peu de corrélations entre ces variables.

2.3.4 Corrélation et conclusions

Nous avons mis en évidence certaines corrélations pouvant exister. Pour mesurer l'intensité de ces relations, nous allons nous intéresser aux calculs des corrélations pour chacune des variables. La bibliothèque *Pandas* dispose d'une méthode permettant de calculer les corrélations sur une table. Néanmoins, celle-ci s'utilise pour des variables numériques. Notre table nécessite donc un traitement pour convertir les données catégorielles de type *object* en valeur numérique. Mais sur quelle table allons-nous travailler ? La table contenant toutes les informations ou la table dont les lignes avec un même identifiant de patient ont été supprimées ? La réponse est les deux.

En effet, la table initiale est adaptée pour toutes les corrélations où apparaissent les variables *x*, *y*, *width*, *height*. À l'inverse, la table simplifiée est adaptée pour toutes les corrélations où ces mêmes variables ne sont pas impliquées. Nous allons par conséquent, calculer la corrélation pour les deux tables et les fusionner en ne gardant que la corrélation exacte :

```
328 df2 = df.copy()
329 df2['class'] = df2['class'].map({'Normal': 0, 'No Lung Opacity / Not
      Normal': 1, 'Lung Opacity': 2})
330 df2['Sexe'] = df2['Sexe'].map({'F': 0, 'M': 1})
331 df2['Position'] = df2['Position'].map({'PA': 0, 'AP': 1})
332 print(df2.head())
333
334 dfNoDup['class'] = dfNoDup['class'].map({'Normal': 0, 'No Lung
      Opacity / Not Normal': 1, 'Lung Opacity': 2})
335 dfNoDup['Sexe'] = dfNoDup['Sexe'].map({'F': 0, 'M': 1})
336 dfNoDup['Position'] = dfNoDup['Position'].map({'PA': 0, 'AP': 1})
337
338 corr = df2.corr()
339 corr2 = dfNoDup.corr()
340 corr.iloc[4:9,4:9] = corr2.iloc[4:9,4:9]
341 print(corr)
```

2. ANALYSE DES DONNÉES

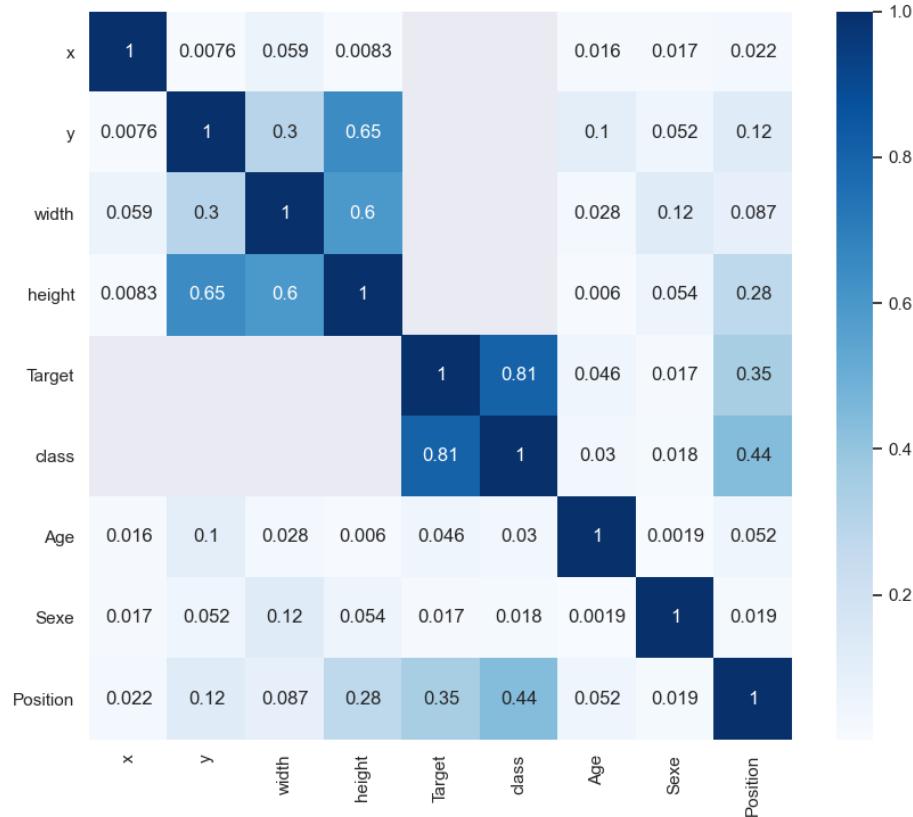
	patientId	x	y	width	height	Target	class	Age	Sexe	Position
0	000...dd6	NaN	NaN	NaN	NaN	0	1	51	0	0
1	003...1 cd	NaN	NaN	NaN	NaN	0	1	48	0	0
2	003...0 eb	NaN	NaN	NaN	NaN	0	1	19	1	1
3	003...5 c5	NaN	NaN	NaN	NaN	0	0	28	1	0
4	004...ab4	264.0	152.0	213.0	379.0	1	2	32	0	1

	x	y	width	height	Target	class	Age	Sexe	Position
x	1.00000	0.00764	-0.05866	0.00825	NaN	NaN	-0.016081	-0.017267	0.022248
y	0.00760	1.00000	-0.29989	-0.64536	NaN	NaN	0.104195	0.052201	-0.124721
width	-0.05866	-0.29989	1.00000	0.59746	NaN	NaN	0.027666	0.123515	0.087427
height	0.00825	-0.64536	0.59746	1.00000	NaN	NaN	-0.006048	0.054157	0.275490
Target	NaN	NaN	NaN	NaN	1.00000	0.807715	-0.046437	0.016851	0.346032
class	NaN	NaN	NaN	NaN	0.80771	1.000000	0.029643	0.018187	0.440231
Age	-0.01608	0.10419	0.02766	-0.00604	-0.04643	0.029643	1.000000	-0.001923	-0.051792
Sexe	-0.01726	0.05220	0.12351	0.05415	0.01685	0.018187	-0.001923	1.000000	0.019354
Position	0.02224	-0.12472	0.08742	0.27549	0.34603	0.440231	-0.051792	0.019354	1.000000

2. ANALYSE DES DONNÉES

Pour mieux visualiser les corrélations importantes, nous allons représenter cette table sur un graphique. Puisque l'objectif est de simplement visualiser l'intensité des relations entre les variables, nous prenons la valeur absolue.

```
343 sns.set()
344 s = sns.heatmap(abs(corr), cmap='Blues', annot=True)
345 plt.show()
```



Quelques conclusions, par rapport à ces données :

- La variable x n'étant pas continue, elle n'est pas corrélée aux autres variables. Néanmoins, on a l'intuition que cette variable peut avoir un impact plus conséquent au même titre que la variable y . On pourrait alors envisager de transformer la variable via une projection, ou de la décomposer entre le poumon droit et le poumon gauche.
- La variable y est fortement corrélée à la hauteur, comme nous avions pu l'observer. De plus, on peut observer que la variable y a un coefficient de corrélation non négligeable avec la largeur, l'âge et la position.
- La largeur et la hauteur des boîtes d'encadrement des opacités ont un coefficient de corrélation conséquent.
- Nous ne l'avions pas représenté graphiquement, mais nous pouvons observer une importante corrélation entre les variables $Target$ et $class$; ce qui est logique puisque la $class$ est une sous-catégorie de $Target$.
- Nous avions pressenti un lien entre la variable $Target$ (et *a fortiori* la variable $class$) et la variable $Position$. Ici, nous pouvons observer qu'il y a effectivement

2. ANALYSE DES DONNÉES

une bonne corrélation entre ces variables. Nous pouvons nous interroger quant à la nature de cette relation. Est-ce que la pneumonie est plus détectable dans une des deux positions ? Ou alors, s'il y a suspicion de pneumonie, y-a-t-il un protocole médical favorisant l'une des deux positions ? Un expert du domaine pourrait sûrement apporter une réponse. On pourrait voir un élément de réponse en observant une corrélation non négligeable entre la position et la hauteur des boîtes d'encadrement des opacités.

- L'âge et le sexe des patients ont peu de corrélation avec la variable *Target*. Par conséquent, ces variables ne devraient pas être prises en compte dans le modèle, du moins sans autre traitement de notre part.

2.4 Valeurs manquantes et aberrantes

En data science, les bases de données contiennent souvent des anomalies. Cela peut être dû à la présence de valeurs aberrantes et/ou manquantes. Ces valeurs peuvent nuire aux performances du modèle prédictif en entraînant d'une part des estimations extrêmement fausses et d'autre part une phase d'entraînement plus longue. Les étapes de visualisation et du traitement de ces valeurs s'avèrent donc cruciales avant toutes autres études. Dans cette partie, nous nous focaliserons sur ces deux types de valeurs.

2.4.1 Traitement des valeurs manquantes

Les données d'apprentissage ont un impact important sur la qualité du modèle. Des données manquantes peuvent conduire à un modèle biaisé et par conséquent mener à une mauvaise classification. Le traitement des valeurs manquantes n'est pas une approche systématique, elle dépend généralement de plusieurs facteurs, comme le type des données, le problème, etc.

Il existe plusieurs méthodes pour les traiter. Elles se basent généralement sur le caractère aléatoire des données manquantes. Lorsque les données manquantes sont complètement aléatoires, l'une de ces méthodes consiste tout simplement à la suppression de ces cas. Cette technique présente l'avantage d'être relativement simple à mettre en place. Néanmoins, elle peut réduire la qualité du modèle lorsque les données d'apprentissage sont peu nombreuses, ou que le nombre de valeurs manquantes représente un pourcentage non négligeable des données d'apprentissage. Une autre technique consiste à supprimer les données manquantes en se basant sur les résultats d'analyse statistique, similaire à la matrice de corrélation. Cette technique a l'avantage d'être moins drastique que la première.

En revanche, si nous avons de nombreuses valeurs manquantes, une technique consiste à remplacer les valeurs manquantes par des valeurs estimées comme la médiane, la moyenne ou encore par des valeurs issues d'un modèle prédictif. On pourrait dans cette situation, ajouter une colonne supplémentaire indiquant si la variable a été estimée ou non.

Fort heureusement, notre base de données ne comporte pas de valeurs manquantes, et ne nécessite donc pas de traitement particulier.

2.4.2 Traitement des valeurs aberrantes

Aucune définition mathématique n'existe pour une valeur aberrante (outlier en anglais). Cependant, une valeur aberrante peut être définie comme une observation ou valeur qui est très éloignée des autres observations/valeurs dans un même échantillon. L'étude des valeurs aberrantes comporte deux étapes. La première étape de visualisation a pour objectif de les identifier. Quant à la deuxième étape, il s'agit de comprendre les causes d'apparition de ces valeurs dans le jeu de données pour qu'ensuite choisir les méthodes de traitement.

Les valeurs aberrantes sont couramment étudiées par les analystes et les scientifiques des données, car elles nécessitent une attention particulière. L'une des meilleures méthodes graphiques pour les visualiser est les boîtes à moustaches pour l'analyse univariée et les diagrammes de dispersion pour l'analyse multivariée ou encore les méthodes de projection.

Les causes d'apparition des valeurs aberrantes sont multiples. Elles peuvent être dues à une erreur de saisie des données lors de la collecte et/ ou de l'enregistrement. Elles peuvent aussi indiquer une erreur expérimentale ou une erreur de mesure due à un défaut dans l'appareil de mesure ou encore un arrondi des valeurs obtenues. Une autre cause d'apparition de valeurs aberrantes est l'erreur d'échantillonnage. Par exemple, dans notre cas, des radiologies reliées à une maladie autre que la pneumonie pourraient exister dans notre base de données.

Bien que la compréhension des causes d'apparition des valeurs aberrantes peut souvent aider à la gestion de ces valeurs, le choix de méthodes de traitement de ces valeurs dans un jeu de données reste une question délicate. En effet, il n'y a pas d'approche systématique pour les gérer. Si le modèle prédictif est sensible aux outliers et que le but est d'avoir un modèle optimal, il est judicieux dans ce cas de supprimer ces valeurs aberrantes. Une analyse des valeurs aberrantes éliminées est souhaitable pour mieux comprendre les causes d'apparition de ces valeurs . En effet, la suppression de ces valeurs aberrantes ne doit pas être une approche systématique avant d'avoir compris leurs sens d'apparition dans le jeu de données. Si par contre, le but du modèle est de détecter les comportements anormaux, dans ce cas il est important de conserver ces valeurs aberrantes, car l'algorithme d'apprentissage devra les observer. Dans d'autres situations, il suffit de remplacer (corriger) ces valeurs par d'autres, voire les traiter séparément.

Concernant notre jeu de données, nous avons pu mettre en évidence grâce à l'analyse univariée et bivariée un certain nombre de valeurs aberrantes. Néanmoins, comme nous venons de l'expliquer, avant de traiter une valeur aberrante il est nécessaire, voire obligatoire, de comprendre l'origine de ces valeurs. L'avis d'un spécialiste (dans notre cas un radiologue) pourrait nous aider à mettre en place une stratégie adéquate à la gestion de ces valeurs. Je n'effectuerai donc aucun traitement de ces valeurs, mais on pourrait envisager une réflexion afin de comprendre les causes et d'améliorer les données d'apprentissage.

2.5 Extraction de caractéristiques (*Features Engineering*)

L'extraction des caractéristiques appelée encore découverte de caractéristiques est souvent la phase la plus complexe de développement de tout projet de machine learn-

2. ANALYSE DES DONNÉES

ning ou deep learning. Elle se décompose en deux étapes, à savoir, la transformation de variables et la création de variables.

Les variables de la base de données sont généralement de natures différentes et ont des comportements différents. Le but de la transformation de variable est avant tout d'homogénéiser ces variables. Pour cela, plusieurs transformations existent. Par exemple, pour harmoniser l'amplitude des variables, nous pouvons normer chacune des variables de manière à ce que toutes les valeurs soient comprises entre 0 et 1. Un autre exemple, il est préférable d'utiliser des variables dont la distribution est symétrique, par conséquent si ce n'est pas le cas, nous pouvons appliquer une transformation non linéaire afin de les symétriser. Par exemple, si une distribution est asymétrique à droite, nous pouvons appliquer une transformation logarithmique ou avec une racine carrée et pour une distribution asymétrique gauche nous pouvons simplement prendre le carré, le cube de la variable ou lui appliquer une fonction exponentielle. Pour notre base de données il est difficile de transformer les variables. En effet, les variables x , y , $width$, $height$ bien que continues, ne sont définies que lorsque la variable *Target* vaut 1, ce qui signifie qu'elles sont également catégorielles. Il est donc difficile de les traiter et même d'envisager de les intégrer aux modèles en tant que métadonnées. Un bon candidat aurait été l'âge, cependant, nous avons vu qu'il n'y a pas de corrélations entre l'âge et la variable *Target*.

La création de variables consiste à créer de nouvelles variables plus pertinentes afin d'améliorer la qualité du modèle. Généralement, les variables peuvent être créées à partir d'autres variables existantes. Par exemple, pour notre étude, nous pouvons créer une variable correspondant au centre des boîtes d'encadrement des opacités. Néanmoins, plusieurs analyses m'ont permis de voir que ces variables n'apportent pas d'informations supplémentaires.

3 Modèles simples de classification d'images

Dans la section précédente, nous avons mené une étude sur les données dont nous disposons. Nous avons mis en évidence l'importance de traiter certaines d'entre elles pour améliorer le modèle d'apprentissage. Nous avons vu également que pour améliorer la qualité du modèle, il faudrait prendre en considération certaines variables comme la position du patient. Néanmoins, il n'est pas judicieux de modéliser le problème avec un modèle complexe directement. En effet, l'ajustement d'un modèle dépend d'un grand nombre de paramètres, cela nécessite plusieurs tentatives de modélisation avant de trouver un bon jeu de paramètres. En réduisant ces paramètres, on accélère la démarche et cela permet d'avoir une meilleure intuition quant à l'approche à adopter sur un modèle complexe.

Dans cette partie, nous allons nous concentrer sur un modèle basique de classification (sans les boites d'encadrement) avec pour seule entrée les images pixelisées. Pour le choix des labels, deux choix s'offrent à nous, à savoir, la variable *Target* indiquant si le patient est atteint d'une pneumonie ou non ou la variable *class* indiquant si le patient n'a pas de pathologie, s'il a une pathologie différente d'une pneumonie ou s'il a une pneumonie. Nous allons utiliser la variable *class* puisqu'elle contient plus d'informations, mais également pour être cohérent avec notre volonté de prendre en compte les différentes catégories depuis le début de cette étude.

Nous avons défini les données d'entrée, les labels de sortie, il nous reste la question du modèle. Il existe de nombreuses techniques d'apprentissage automatique pour modéliser les problèmes de classification par exemple, les machines à vecteur de support, les arbres de décisions, les classificateurs linéaires, les classificateurs Bayésiens, etc. Néanmoins, des progrès récents ont permis de développer des techniques ultra-performantes d'apprentissage en profondeur : les réseaux de neurones profonds. Ils se sont imposés ces dernières années de manière impressionnante dans plusieurs champs de recherche, par exemple en reconnaissance faciale, en synthèse vocale, en traduction et bien d'autres.

3.1 Réseaux de neurones profonds

Avant de présenter les différents types de réseaux de neurones, commençons par définir tout d'abord le modèle de base : "le neurone" également appelé "perceptron". Un neurone dont la conception s'inspire du fonctionnement des neurones biologiques est un modèle mathématique qui prend en entrée une ou plusieurs variables et retourne une seule valeur numérique. Le neurone est défini ainsi :

$$y = f(\sum_{i=1}^n w_i x_i + b)$$

avec x_i une entrée ou la sortie d'un autre neurone et w_i un poids représentant l'importance de x_i , b est un biais permettant d'ajuster le modèle et f une fonction non linéaire appelée *fonction d'activation*. Cette fonction est généralement dans le cas de classification une fonction *sigmoid*, une tangente hyperbolique, une fonction *ReLU* (*Rectifier Linear Unit*) ou encore une fonction *softmax*. Cette fonction est généralement représentée sous forme de diagramme comme illustré sur la figure 1, ce qui permet de détailler les différentes opérations.

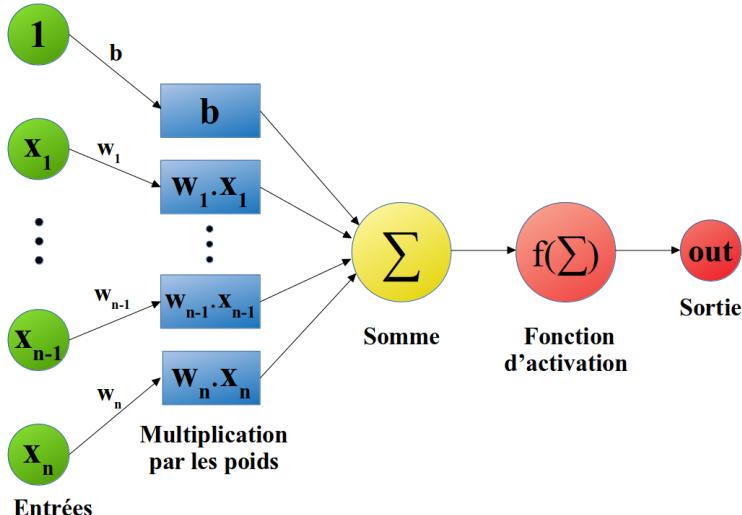


FIGURE 1 – Neurone artificiel (Perceptron)

Un réseau de neurones est composé de plusieurs neurones, généralement répartis en plusieurs couches connectées entre elles. En effet, un réseau de neurones à une couche ne peut classer que des données linéairement séparables. Les couches supplémentaires entre la couche d'entrée et la couche de sortie sont appelées couches cachées (*hidden layers*) dont chaque connexion possède son propre poids w . L'appellation réseau de neurones profond signifie simplement que le réseau est représenté par un grand nombre de couches. Dans un réseau séquentiel, l'information ne peut pas passer deux fois par le même neurone (pas de boucles) et elle circule dans une seule direction, de l'entrée vers la sortie. Il existe néanmoins, d'autres réseaux que les réseaux à couches séquentielles. Par exemple, nous pouvons également choisir de les organiser avec deux couches d'entrée, deux couches de sortie et des couches cachées interconnectées de manière aléatoire.

Il existe un grand nombre de réseaux de neurones profonds qui diffèrent par leurs architectures, c'est-à-dire la façon dont les neurones sont organisés dans le réseau, et parfois même, par la manière dont ils sont entraînés. Parmi ces architectures, on retrouve les perceptrons multicouches, les réseaux de neurones convolutifs, les réseaux de neurones récurrents ou encore les auto-encodeurs. Généralement, pour la classification d'images, on utilise des réseaux de neurones convolutifs particulièrement bien adaptés car malheureusement tous ne le sont pas. Par exemple, pour les perceptrons multicouches, chaque neurone d'une couche est connecté à l'ensemble des neurones de la couche précédente, on dit que les couches sont entièrement connectées (*fully connected*). Dans le cas d'une image, les données de la couche d'entrée représentent chaque pixel de l'image ce qui est extrêmement coûteux. En effet, prenons l'exemple d'une image en niveau de gris de taille 512×512 , soit 262.144 pixels au total et *a fortiori* 262.144 neurones par couches. Et puisque chaque neurone est connecté aux neurones de la couche précédente, on a $262.144^2 = 68.719.476.736$ connexions. Par conséquent, il faut être prudent sur le choix du réseau de neurones à utiliser si on veut modéliser un problème de classification d'images.

3.2 Réseau de neurones convolutif (CNN)

Les réseaux de neurones convolutifs aussi appelés CNN (pour Convolutional Neural Network) ou ConvNet sont un type des réseaux de neurones particulièrement bien adapté à la classification des images. Ils ont de larges applications autour de l'image ou de la vidéo dont fait partie la reconnaissance faciale ou encore la classification d'images. Le but principal d'un réseau CNN est d'extraire des caractéristiques d'une image donnée.

L'architecture des CNN (figure 2) se décompose en trois parties bien distinctes. La première partie : l'entrée, une image (dans notre cas un DICOM) est fournie sous la forme d'une matrice de pixels. Elle a 2 dimensions pour une image en niveaux de gris. La deuxième partie d'un réseau CNN fait la particularité de ce type de neurones puisqu'il fonctionne comme un extracteur de caractéristiques des images (*features* en anglais). L'idée est d'utiliser un ensemble d'outils mathématiques afin de réduire la taille de l'image et de ne conserver que les caractéristiques importantes. Parmi ces méthodes, la plus importante est le produit de convolution 2d (aussi appelé filtre). Plusieurs filtres sont testés avec des valeurs différentes. D'une étape à la suivante, le réseau apprend à reconnaître les éléments caractéristiques d'une image et les meilleurs filtres sont retenus. Une fois un élément appris à un endroit de l'image, le réseau sera capable de le reconnaître dans n'importe quelles autres images. Enfin, la dernière partie comprend un ensemble de couches entièrement connectées. Le but étant de classer l'image en renvoyant une probabilité d'appartenance aux différentes catégories.

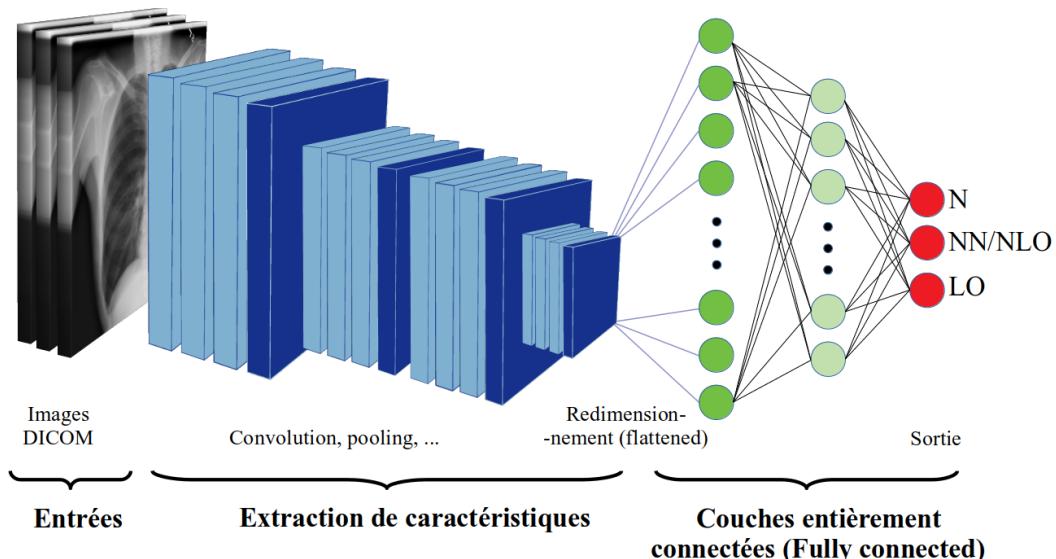


FIGURE 2 – Réseau neuronal convolutif

Leur principe est assez facile à comprendre comme nous venons de l'expliquer. Néanmoins, il n'existe pas de règles quant au choix de l'architecture, autrement dit le nombre de paramètres à choisir et mettre en place (nombre de filtres, taille de filtres, déplacement des filtres, choix de pooling, nombre de couches de neurones, nombre de neurones par couche, etc). Dans le domaine de l'imagerie, de nombreuses études

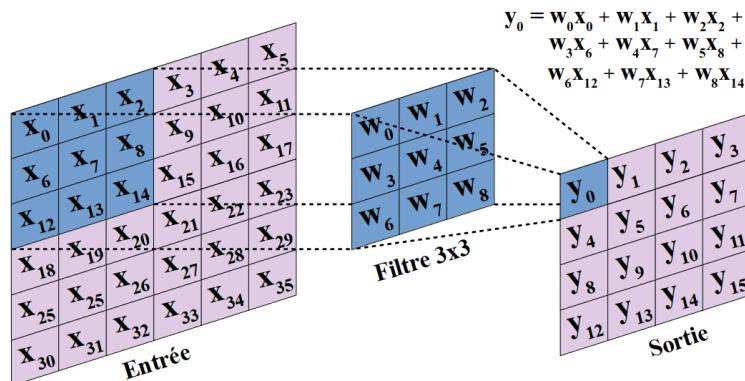
3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

ont démontré l'efficacité de certaines architectures que nous allons présenter dans ce qui suit. Mais avant cela, nous allons voir certaines opérations de convolution couramment utilisées dans les CNN ainsi que quelques techniques d'amélioration du modèle.

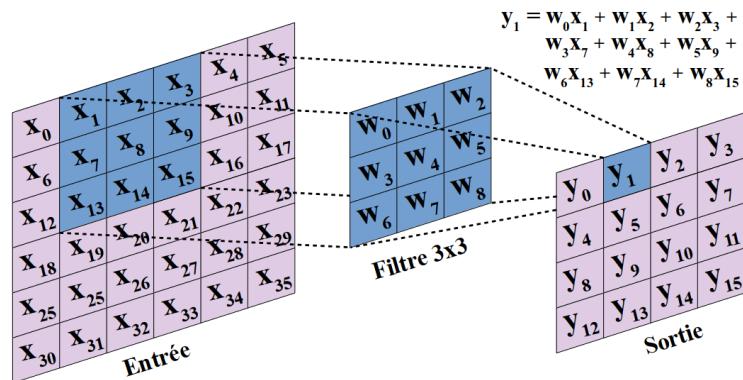
3.2.1 Opérations de convolution

Couche de convolution

La couche de convolution est l'élément le plus important d'un CNN. Il se compose d'un ensemble de filtres (également appelés noyaux ou extracteurs de caractéristiques), où chaque filtre est appliqué dans toutes les zones des données d'entrée. Un filtre est défini par un ensemble de poids apprenants. La couche d'entrée peut s'agir aussi bien de la première couche du réseau (donc chaque neurone d'entrée représente l'intensité de couleur d'un pixel) que de n'importe quelle couche du réseau. La figure 3 illustre très bien un produit de convolution entre des données d'entrées stockées dans une matrice de taille 6x6 et un filtre 3x3.



(a) Calcul de y_0



(b) Calcul de y_1

FIGURE 3 – Produit de convolution

Tout d'abord, nous appliquerons le filtre dans le coin supérieur gauche de l'image. Chaque neurone d'entrée est associé à un poids unique du filtre. La sortie du filtre

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

est une somme pondérée de ses entrées. Son but est de mettre en évidence une caractéristique spécifique dans l'entrée, par exemple, un bord ou une ligne. Pour calculer chaque autre neurone, il suffit de glisser le filtre sur l'image d'entrée, mais les poids eux ne changent pas. En réduisant le nombre de poids, nous réduisons l'espace mémoire. Le filtre met en évidence des caractéristiques spécifiques. En partageant les poids, nous garantissons que le filtre sera en mesure de localiser la même caractéristique à un autre endroit de l'image. Notons que le glissement n'est pas forcément de 1 pixel, il peut avoir un pas supérieur.

Dans une couche de convolution, la valeur d'activation du neurone est définie de la même manière que la valeur d'activation du perceptron, cependant, le neurone ne prend en entrée qu'un nombre limité de neurones. Cela s'oppose à une couche entièrement connectée, où l'entrée provient de tous les neurones.

Nous allons maintenant voir à travers un exemple comment les filtres permettent de mettre en évidence des caractéristiques. Implémentons une opération de convolution en appliquant quelques filtres sur une image. Notons que l'image est une image couleur et possède donc trois couches en profondeur. Dans ce cas, le filtre est un tenseur dont la profondeur est 3, ce qui assure d'obtenir une matrice 2D. Tout d'abord, importons les librairies et l'image test :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.image as img
4 from scipy.signal import convolve
5
6 # Importation de l'image
7 image = img.imread('doc.png')
8 print("Type de l'image : ", type(image))
9 print("Taille de l'image : ", image.shape)
```

Type de l'image : <class 'numpy.ndarray'>
Taille de l'image : (512, 512, 3)

Pour automatiser la démarche, nous créons une fonction qui prend en entrée un filtre et effectue un produit de convolution sur notre image. On utilise pour cela la fonction *convolve* de la librairie *scipy*. On aurait tout aussi bien pu programmer notre propre produit de convolution puisqu'il s'agit simplement de 4 boucles imbriquées. Notons également que l'on effectue le produit sur chacune des couches RGB.

```
11 # Fonction effectuant un produit de convolution entre un filtre et
12 # une image RGB
13 def convProduct(filtre) :
14     rgb = []
15     for channel in range(3):
16         c = convolve(image[:, :, channel], filtre, mode='valid')
17         rgb.append(c)
18
19     img2 = np.dstack((rgb[0], rgb[1], rgb[2]))
```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```
19     return img2
```

Maintenant, nous implémentons 5 filtres de natures différentes, à savoir, un filtre de flou gaussien, un filtre d'amélioration de la netteté, un filtre de repoussage et deux filtres de détection des bords.

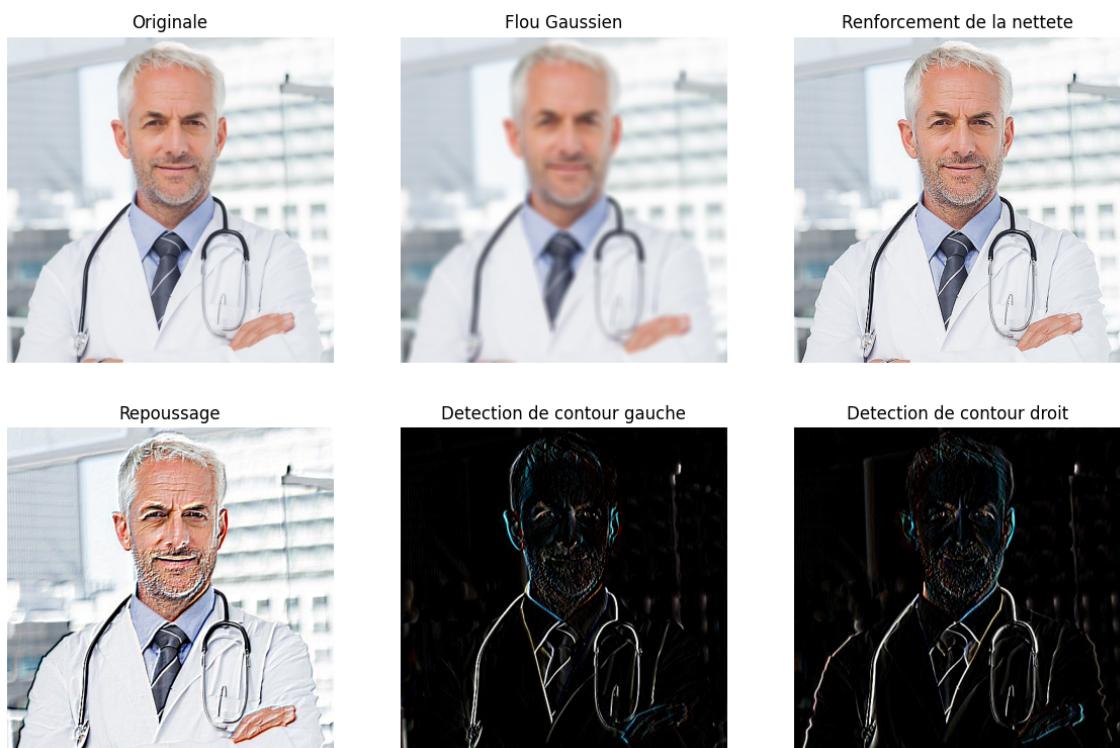
```
21 # Filtre Gaussien 10x10
22 f0 = np.full([10, 10], 1. / 100)
23 image0 = convProduct(f0)
24
25 # Amelioration de la nettete
26 f1 = np.array([
27     [0, -1, 0],
28     [-1, 5, -1],
29     [0, -1, 0]
30 ])
31 image1 = convProduct(f1)
32
33 # Repoussage - effet de perspective
34 f2 = np.array([
35     [-2, -1, 0],
36     [-1, 1, 1],
37     [0, 1, 2]
38 ])
39 image2 = convProduct(f2)
40
41 # Detection des bords gauches
42 f3 = np.array([
43     [-1, 0, 1],
44     [-2, 0, 2],
45     [-1, 0, 1]
46 ])
47 image3 = convProduct(f3)
48
49 # Detection des bords droits
50 f4 = np.array([
51     [1, 0, -1],
52     [2, 0, -2],
53     [1, 0, -1]
54 ])
55 image4 = convProduct(f4)
```

On termine par afficher les images résultantes :

```
49 # Affichage
50 fig, ax = plt.subplots(nrows=2, ncols=3)
51 ax[0,0].imshow(image)
52 ax[0,0].axis('off')
53 ax[0,0].set_title('Originale')
54
55 ax[0,1].imshow(image0)
56 ax[0,1].axis('off')
57 ax[0,1].set_title('Flou Gaussien')
58
59 ax[0,2].imshow(image1)
```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```
60 ax[0,2].axis('off')
61 ax[0,2].set_title('Renforcement de la nettete')
62
63 ax[1,0].imshow(image2)
64 ax[1,0].axis('off')
65 ax[1,0].set_title('Repoussage')
66
67 ax[1,1].imshow(image3)
68 ax[1,1].axis('off')
69 ax[1,1].set_title('Detection de contour gauche')
70
71 ax[1,2].imshow(image4)
72 ax[1,2].axis('off')
73 ax[1,2].set_title('Detection de contour droit')
74 plt.show()
```



Dans cet exemple, nous avons utilisé des filtres avec des poids codés en dur pour visualiser le fonctionnement de l'opération de convolution dans les réseaux de neurones. En réalité, les poids du filtre seront appris lors de l'apprentissage du réseau. Il suffit de définir l'architecture du réseau, comme le nombre de couches convolutives ou la taille des filtres. Le réseau apprend les caractéristiques, mises en évidence par chaque filtre pendant l'apprentissage.

Pooling

Un *pooling* est une opération visant à "mettre en commun" un certain nombre de données d'entrées. L'image d'entrée est divisée en carré de n pixels de chaque côté ne chevauchant pas. Par exemple, pour un *pooling* de taille 2x2 on a généralement un

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

pas de 2, pour un *pooling* de taille 3x3 un pas de 3, etc. Chaque carré comprenant plusieurs pixels est soumis à une fonction d'agrégation (maximum ou moyenne) permettant de réduire le carré à un seul pixel. Les *pooling* ne modifient pas la profondeur du volume, car la fonction d'agrégation est effectuée indépendamment sur chaque tranche. Un exemple de *pooling* est représenté sur la figure 4. Un *pooling* de taille 2x2 avec un pas de 2 est appliqué sur une matrice 4x4.

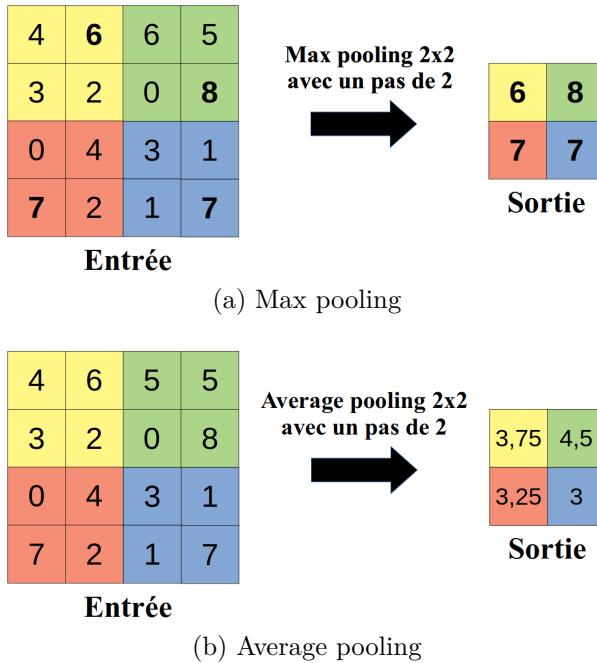


FIGURE 4 – Exemple de *pooling*

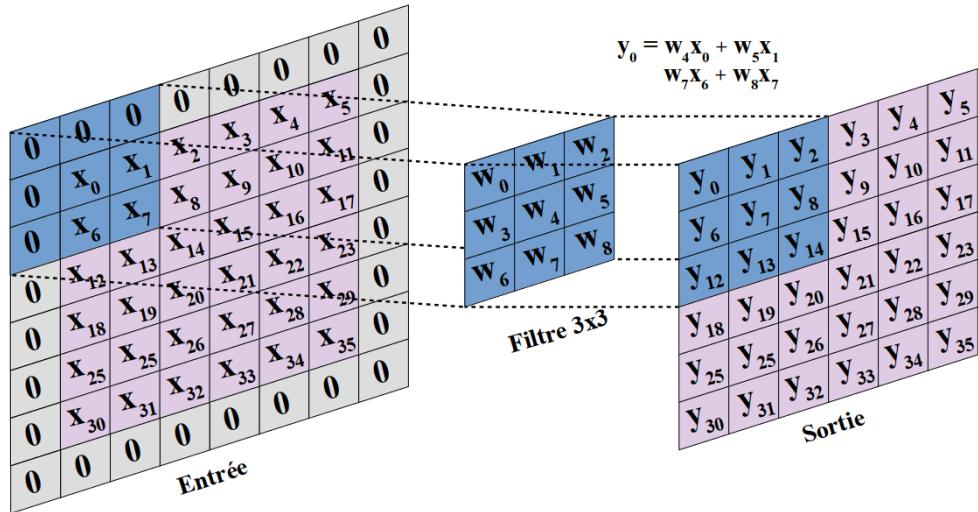
La première étape consiste à diviser la matrice en carré de taille 2x2. Puis la seconde étape consiste à appliquer une fonction d'agrégation à chaque carré. Pour une fonction d'agrégation de type *maximum*, seule la valeur maximale de chaque carré est conservée. Dans le cas d'une fonction d'agrégation de type *moyenne*, la moyenne des valeurs de chaque carré est alors retournée.

Les couches de *pooling* n'ont pas de poids contrairement aux filtres de convolution.

Padding

Les opérations de convolution vues jusqu'à présent ont généré une sortie plus petite que l'entrée. Mais, en pratique, il est souvent préférable de contrôler la taille de la sortie. Pour cela, une technique appelée *padding* consiste à ajouter temporairement des pixels de valeur 0 sur le contour de l'image. La façon la plus courante d'utiliser le *padding* est de l'associer à un produit de convolution afin de produire une sortie avec les mêmes dimensions que l'entrée. Dans la figure 5, nous pouvons voir un exemple de *padding*.

Les pixels gris de valeur 0 représentent le *padding*. L'entrée et la sortie ont les mêmes dimensions (pixels mauves). Il s'agit de la manière la plus courante d'utiliser le *padding*. Les pixels du contour ont une valeur de 0 afin de ne pas affecter le résultat du produit de convolution.


 FIGURE 5 – Exemple de *padding*

Redimensionnement (Flattened)

Nous venons de voir les opérations de convolution classiques existant dans toutes les architectures de CNN. Ces opérations sont appliquées sur plusieurs couches, de plus, plusieurs produits de convolution sont utilisés sur une même couche. On obtient alors un tenseur dont la profondeur est égale au nombre de filtres utilisés. Après la dernière opération de convolution, le tenseur obtenu est redimensionné sous forme de vecteur, cette opération est couramment nommée *flatten*. Le vecteur est ensuite utilisé comme une entrée d'un réseau de neurones entièrement connecté (*fully connected*).

3.2.2 Techniques d'amélioration

Après avoir vu les opérations courantes utilisées dans les CNN, nous allons voir maintenant des techniques d'amélioration de modèle.

Prétraitement des données d'entrée

Les images ont des valeurs de pixels comprises entre 0 et 255, ce qui correspond numériquement à un intervalle important. Par conséquent, lorsque sur une même image ou bien même d'une image à l'autre, les valeurs des pixels sont relativement différentes, le modèle peut ne pas considérer les zones où la valeur des pixels est plus faible. Ceci est encore plus vrai si l'on considère une image couleur RGB, l'information contenue dans une couche de couleur peut être masquée par les deux autres.

Pour remédier à cela, un prétraitement, une normalisation des images est nécessaire. En pratique, deux techniques sont utilisées. Premièrement, la mise à l'échelle est une technique de normalisation des données via la formule :

$$\mathbf{x} = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}$$

En pratique, on divise simplement la valeur des pixels par 255.

La deuxième technique est de calculer la cote Z des données, *standard score* en anglais. Elle se calcule de la même manière que la variable centrée réduite :

$$\mathbf{x} = \frac{\mathbf{x} - \mu}{\sigma}$$

où μ représente la moyenne de \mathbf{x} et σ son écart-type.

Méthode de dégradation des pondérations

Pour éviter les problèmes de sur-apprentissage, des techniques dites de régularisation sont généralement introduites dans le modèle. L'une d'elles est la méthode de dégradation des pondérations ou *weight decay* en anglais. Il a été démontré dans [Krogh and Hertz, 1992] qu'en ajoutant un terme de pénalité dans la fonction de coût, cela force la diminution des poids, ce qui donne un réseau moins spécialisé et évite le sur-apprentissage.

Décrochage

Le décrochage ou abandon (*dropout* en anglais) est également une technique de régularisation évitant les problèmes de sur-apprentissage. Elle consiste à abandonner temporairement des neurones de manière aléatoire [Srivastava et al., 2014]. Le décrochage peut être appliqué après une opération de convolution (produit, pooling, ...) ou après une couche dense. Un exemple de décrochage est représenté dans la figure 6 pour un réseau entièrement connecté.

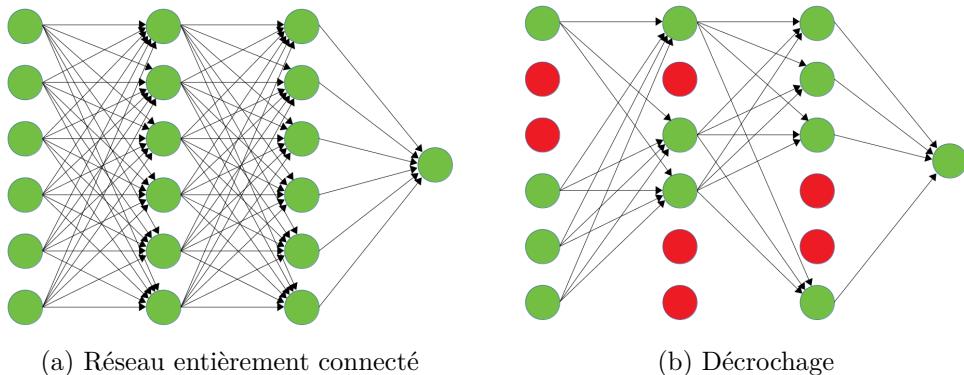


FIGURE 6 – Exemple de décrochage

Augmentation des données

L'augmentation des données est l'une des techniques de régularisation les plus efficaces. Si les données d'entraînement sont trop petites, le réseau peut sur-apprendre. L'augmentation des données permet de contrer cela en augmentant artificiellement la taille des données d'apprentissage. Pour cela, nous pouvons appliquer des augmentations aléatoires aux images, avant de les utiliser pour l'apprentissage. Les étiquettes resteront les mêmes. Parmi les augmentations d'image les plus populaires, on trouve

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

les rotations, les symétries, les étirements, les homothéties, les translations, et bien d'autres. Reprenons l'exemple précédent pour représenter quelques transformations.

```
76 # Rotation
77 from scipy.ndimage import rotate
78 rgb = []
79 imgR = rotate(image[:, :, 0], 30, reshape=False)
80 rgb.append(imgR)
81 imgG = rotate(image[:, :, 1], 30, reshape=False)
82 rgb.append(imgG)
83 imgB = rotate(image[:, :, 2], 30, reshape=False)
84 rgb.append(imgB)
85 image0 = np.dstack((rgb[0], rgb[1], rgb[2]))
86
87 # Symetrie
88 rgb = []
89 imgR = np.fliplr(image[:, :, 0])
90 rgb.append(imgR)
91 imgG = np.fliplr(image[:, :, 1])
92 rgb.append(imgG)
93 imgB = np.fliplr(image[:, :, 2])
94 rgb.append(imgB)
95 image1 = np.dstack((rgb[0], rgb[1], rgb[2]))
96
97
98 # Recadrage
99 lx = image.shape[0]
100 ly = image.shape[1]
101
102 rgb = []
103 imgR = image[lx // 4: - lx // 4, ly // 4: - ly // 4, 0]
104 imgG = image[lx // 4: - lx // 4, ly // 4: - ly // 4, 1]
105 imgB = image[lx // 4: - lx // 4, ly // 4: - ly // 4, 2]
106 rgb.append(imgR)
107 rgb.append(imgG)
108 rgb.append(imgB)
109 image2 = np.dstack((rgb[0], rgb[1], rgb[2]))
110
111
112 # Affichage
113 fig, ax = plt.subplots(nrows=1, ncols=4)
114 ax[0].imshow(image)
115 ax[0].axis('off')
116 ax[0].set_title('Originale')
117
118 ax[1].imshow(image0)
```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```

119 ax[1].axis('off')
120 ax[1].set_title('Rotation')
121
122 ax[2].imshow(image1)
123 ax[2].axis('off')
124 ax[2].set_title('Symetrie')
125
126 ax[3].imshow(image2)
127 ax[3].axis('off')
128 ax[3].set_title('Recadrage')
129
130 plt.show()

```



Batch normalization

Dans le prétraitement des données, nous avons expliqué pourquoi la normalisation des données est importante. La normalisation des mini-lots (*Batch normalization*) fournit un moyen d'appliquer un traitement des données, similaire à la cote Z, pour les couches cachées du réseau [Ioffe and Szegedy, 2015]. En notant x_i , $i = 1, 2, \dots, m$ les valeurs de sorties des neurones de la couche précédente sur un mini-lot de taille m . Le *batch normalization* normalise les sorties de la couche cachée pour chaque mini-lot (d'où le nom) de telle manière qu'il maintient sa valeur d'activation moyenne proche de 0 et son écart-type proche de 1.

$$\tilde{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

avec ε une valeur négligeable,

$$\mu = \frac{1}{m} \sum_{i=1}^m x_i$$

et

$$\sigma = \sqrt{\frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2}.$$

La sortie de la couche du *batch normalization* vaut :

$$y_i = \gamma \tilde{x}_i + \beta.$$

Sous *Keras*, un mini-lot comprend 4 paramètres, 2 entraînables γ et β et 2 non entraînables μ et σ .

Transfer Learning

L'entraînement consiste à optimiser les coefficients du réseau pour minimiser l'erreur de classification en sortie. En pratique, entraîner un réseau de neurones convolutifs est très coûteux et peut prendre beaucoup de temps pour les meilleurs CNN. En effet, plus les couches s'empilent plus le nombre de convolutions et de paramètres à optimiser est élevé. Une stratégie efficace pour contourner ce problème et ne pas créer un réseau CNN de A à Z, est d'appliquer le *Transfer Learning* (ou apprentissage par transfert). L'idée de base de cette technique consiste à réutiliser des réseaux CNN préentraînés sur d'autres problèmes ; autrement dit utiliser les connaissances acquises par un réseau de neurones existant qui classifie bien une large collection d'images afin de résoudre un autre plus au moins similaire d'où le nom *transfer learning*.

3.2.3 Architectures classiques de CNN

Les réseaux de neurones convolutifs sont le type de réseau le plus adapté à la tâche de classification des images. Leur principe est assez facile à comprendre comme nous venons de l'expliquer. Néanmoins, il n'existe pas de règles quant au choix de l'architecture, autrement dit le nombre de paramètres à choisir et mettre en place (nombre de filtres, taille de filtres, déplacement des filtres, choix de pooling, nombre de couches de neurones, nombre de neurones par couche, etc.). Dans le domaine de l'imagerie médicale, de nombreuses études ont démontré l'efficacité de certaines architectures que nous allons présenter dans ce qui suit.

VGG

Dans cette partie, nous nous focalisons sur VGG (Visual Geometry Group), une architecture d'un CNN, proposée par Simonyan et Zisserman [Simonyan and Zisserman, 2015]. Plus profonde que l'architecture AlexNet, VGG est constituée de 16 couches pour VGG-16 et 19 pour VGG-19 comme illustré sur la figure 7 avec une succession d'un empilement de couches de convolution suivie d'un max-pooling. En effet, Simonyan et Zisserman ont remarqué qu'une couche convolutionnelle avec un filtre de grande taille peut être remplacée par un empilement de deux couches convolutionnelles ou plus avec des filtres plus petits. Cet empilement a plusieurs avantages. D'une part, le nombre de poids à optimiser de cet empilement est plus petit que celui d'une couche convolutionnelle avec un filtre relativement large. D'autre part, cet empilement rend la fonction de décision plus discriminante due aux fonctions de rectification non linéaire incorporées après chaque couche convolutionnelle [Simonyan and Zisserman, 2015]. VGG est ainsi basée sur des noyaux de convolution de dimension fixe égale à 3×3 contrairement à ceux d'AlexNet avec des noyaux différents. Le "stride" est fixé à 1 pixel, quant au "padding" est tel que la résolution spatiale est préservée après l'opération de convolution. Le max-pooling est effectué sur une fenêtre de 2×2 pixel avec un stride 2. En 5 étapes, l'image passe de 224×224 pixels à 7×7 pixels. Cet empilement de couches de convolution avec du max-pooling est suivi de 3 couches entièrement connectées (FC) ; ce qui fait un total de presque de 134 millions de paramètres à optimiser. Malgré la robustesse et l'efficacité du VGG prouvées sur plusieurs bases de données, son nombre de paramètres constitue son

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

VGG16	VGG19
conv 3x3, 64	conv 3x3, 64
conv 3x3, 64	conv 3x3, 64
max pool	
conv 3x3, 128	conv 3x3, 128
conv 3x3, 128	conv 3x3, 128
max pool	
conv 3x3, 256	conv 3x3, 256
conv 3x3, 256	conv 3x3, 256
conv 3x3, 256	conv 3x3, 256
	conv 3x3, 256
max pool	
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
	conv 3x3, 512
max pool	
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
	conv 3x3, 512
max pool	
fc-4096	
fc-4096	
fc-1000	
softmax	

FIGURE 7 – Architecture VGG-16 et VGG-19 [Vasilev et al., 2019].

défaut en matière de temps de calcul et espace mémoire.

```

132 # model
133 my_VGG16 = Sequential() # Creation d'un reseau de neurones
134
135 # 1ere couche de convolution, suivie d'une couche ReLU
136 my_VGG16.add(Conv2D(64, (3, 3), input_shape=(img_dims, img_dims, 1),
137 padding='same', activation='relu'))
138 # 2eme couche de convolution, suivie d'une couche ReLU
139 my_VGG16.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
140 # 1ere couche de pooling
141 my_VGG16.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
142
143 # 3eme couche de convolution, suivie d'une couche ReLU
144 my_VGG16.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
145 # 4eme couche de convolution, suivie d'une couche ReLU
146 my_VGG16.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
147 # 2eme couche de pooling

```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```

148 my_VGG16.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
149
150 # 5eme couche de convolution, suivie d'une couche ReLU
151 my_VGG16.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
152 # 6eme couche de convolution, suivie d'une couche ReLU
153 my_VGG16.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
154 # 7eme couche de convolution, suivie d'une couche ReLU
155 my_VGG16.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
156 # 3eme couche de pooling
157 my_VGG16.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
158
159 # 5eme couche de convolution, suivie d'une couche ReLU
160 my_VGG16.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
161 # 6eme couche de convolution, suivie d'une couche ReLU
162 my_VGG16.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
163 # 7eme couche de convolution, suivie d'une couche ReLU
164 my_VGG16.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
165 # 3eme couche de pooling
166 my_VGG16.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
167
168 # 5eme couche de convolution, suivie d'une couche ReLU
169 my_VGG16.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
170 # 6eme couche de convolution, suivie d'une couche ReLU
171 my_VGG16.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
172 # 7eme couche de convolution, suivie d'une couche ReLU
173 my_VGG16.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
174 # 3eme couche de pooling
175 my_VGG16.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
176
177 my_VGG16.add(Flatten()) # Conversion des matrices en vecteur 1D
178
179 # 1ere couche fully-connected, suivie d'une couche ReLU
180 my_VGG16.add(Dense(4096, activation='relu'))
181
182 # 2eme couche fully-connected, suivie d'une couche ReLU
183 my_VGG16.add(Dense(4096, activation='relu'))
184
185 # Couche fully-connected qui permet de classifier
186 my_VGG16.add(Dense(2, activation='sigmoid')) # 2 classes donc 2 et
187 # sigmoid
188 my_VGG16.summary()

```

Layer (type)	Output Shape	Param #
conv2d_14 (Conv2D)	(None, 224, 224, 64)	640

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

conv2d_15 (Conv2D)	(None, 224, 224, 64)	36928
max_pooling2d_6 (MaxPooling2D)	(None, 112, 112, 64)	0
conv2d_16 (Conv2D)	(None, 112, 112, 128)	73856
conv2d_17 (Conv2D)	(None, 112, 112, 128)	147584
max_pooling2d_7 (MaxPooling2D)	(None, 56, 56, 128)	0
conv2d_18 (Conv2D)	(None, 56, 56, 256)	295168
conv2d_19 (Conv2D)	(None, 56, 56, 256)	590080
conv2d_20 (Conv2D)	(None, 56, 56, 256)	590080
max_pooling2d_8 (MaxPooling2D)	(None, 28, 28, 256)	0
conv2d_21 (Conv2D)	(None, 28, 28, 512)	1180160
conv2d_22 (Conv2D)	(None, 28, 28, 512)	2359808
conv2d_23 (Conv2D)	(None, 28, 28, 512)	2359808
max_pooling2d_9 (MaxPooling2D)	(None, 14, 14, 512)	0
conv2d_24 (Conv2D)	(None, 14, 14, 512)	2359808
conv2d_25 (Conv2D)	(None, 14, 14, 512)	2359808
conv2d_26 (Conv2D)	(None, 14, 14, 512)	2359808
max_pooling2d_10 (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_2 (Flatten)	(None, 25088)	0
dense_4 (Dense)	(None, 4096)	102764544
dense_5 (Dense)	(None, 4096)	16781312
dense_6 (Dense)	(None, 2)	8194
Total params: 134,267,586		
Trainable params: 134,267,586		
Non-trainable params: 0		

DenseNet

DenseNet est une architecture proposée par Huang et al. [Huang et al., 2017] comportant des connexions denses. Autrement dit, elle est construite de telle sorte que les résultats des couches inférieures sont transmis à toutes les couches supérieures c.-à-d. les plus profondes du réseau. En effet, l'architecture est divisée en plusieurs blocs denses. Dans ces derniers, chaque couche de convolution prend en entrée les caractéristiques extraites par toutes les couches précédentes créant ainsi des sauts de connexion entre les couches internes des blocs comme illustré dans la figure 8.

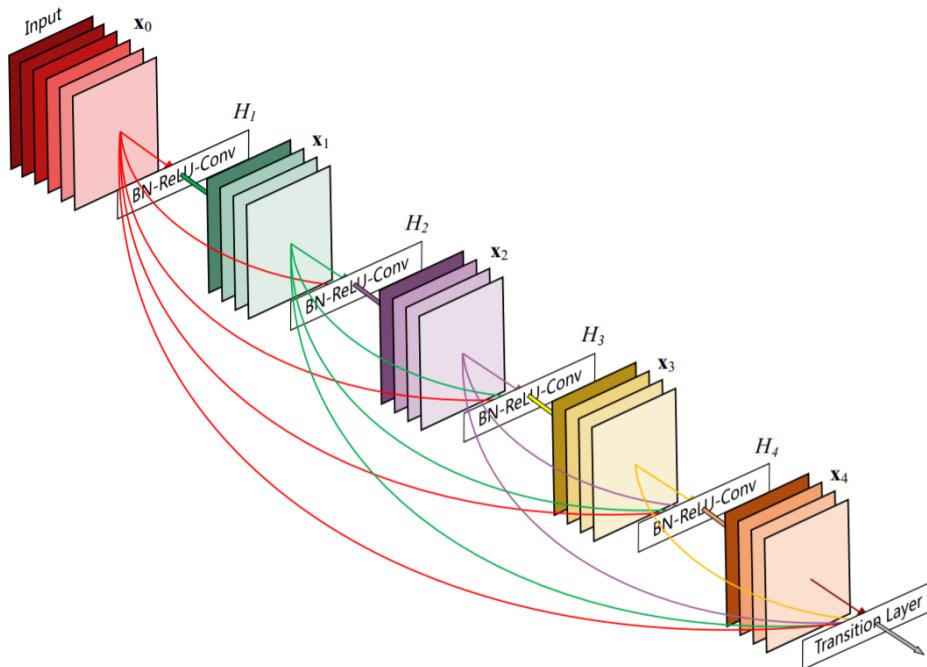


FIGURE 8 – Bloc dense à 5 couches [Huang et al., 2017].

Ces blocs denses sont connectés entre eux par des convolutions et du pooling, appelée couche de transition. Les avantages des architectures de denseNet sont nombreux. D'une part, les connexions denses permettent au gradient de se propager immédiatement des couches supérieures aux couches inférieures. D'autre part, le nombre de paramètres est faible étant donné que l'architecture est divisée en plusieurs blocs. Néanmoins, ces architectures demandent un espace mémoire important pour stocker les activations et gradients intermédiaires.

CheXNet

CheXNet est un réseau de neurones convolutif dense à 121 couches (DenseNet) créé pour analyser les résultats de radiographies thoraciques frontales. En effet, il prend en entrée une radiographie du thorax et renvoie la probabilité de pneumonie avec une carte thermique des régions les plus révélatrices de la pathologie et ainsi les plus pertinentes pour la classification du diagnostic. En 2017 [Rajpurkar et al., 2017], CheXNet était la meilleure approche disponible pour le diagnostic de pneumonie. En effet, ce réseau de neurones a été validé sur la base de données *Chest X-ray 14* contenant plus de 100000 radiographies de patients souffrant de 14 maladies thoraciques différentes, y compris la pneumonie. Pour assurer l'entraînement d'un tel réseau aussi profond, les auteurs [Rajpurkar et al., 2017] ont utilisé les techniques de "dense connections" et "batch normalization", un processus

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

éprouvé pour accroître la vitesse et l'exactitude. Par ailleurs, les paramètres (poids) du modèle ont été initialisés avec les poids d'un modèle préentraîné sur la base de données ImageNet avec des "mini-batches" égales à 16. La dernière couche entièrement connectée est remplacée par une couche à une seule sortie à laquelle une fonction non linéaire "sigmoid" est appliquée. L'image d'entrée est de taille 224×224 pixels comme c'était le cas pour VGG. Quant à la fonction de perte ; celle qui quantifie l'écart entre les prédictions du modèle et les observations réelles du jeu de données ; elle a été pondérée par des poids judicieusement choisis. En effet, ces poids sont calculés par rapport à la répartition des exemples positifs / négatifs de chaque classe.

Étant donné que cette architecture était utilisée pour la détection des régions de pneumonie, elle sera testée dans la partie suivante dédiée à la détection des boîtes d'encadrement des opacités.

3.3 Implémentation et analyse des différentes architectures

Les calculs présentés dans cette partie ont été réalisés sur un ordinateur portable avec un processeur *IntelCore i7-9750H CPU 2.60GHz×12* et de carte graphique *GeForce GTX 1660 Ti/PCIe/SSE2*. De plus, suite à l'installation de *Tensorflow-2.1.0* et *Cuda-10.2.0*, les calculs ont bénéficié d'une accélération sur carte graphique. À titre informatif, un calcul réalisé sur 25 epochs prend en moyenne 6h de calcul.

3.3.1 Data generator

L'utilisation d'un générateur de données pour charger les données est cruciale lors de l'entraînement d'un modèle d'apprentissage profond, car ayant un ensemble de données important cela nécessite un grand espace mémoire.

Par exemple, pour le chargement des images, souvent la librairie *Numpy* est utilisée pour lire l'ensemble des images stockées et les transformer en tenseur. Ce tenseur est alors conservé en mémoire et est utilisé entièrement ou partiellement lors de l'apprentissage. Cependant lorsque la base de données est importante, plusieurs problèmes se posent et notamment le manque d'espace mémoire.

Pour surmonter ce problème, les data ingénieurs ont généralement recours à des *data generator* qui permettent de lire un petit nombre fini de fichiers de manière optimale et de ne les conserver que temporairement en mémoire. Nous nous concentrerons dans cette partie, sur la construction d'un générateur de données pour charger et traiter les images en utilisant la bibliothèque *Keras* et/ou *tensorflow*. Les avantages des générateurs sont multiples. En effet, ils traitent les données au fur et à mesure et permettent donc de gérer des bases de données bien plus volumineuses.

Nous allons donc créer un générateur d'une part pour la lecture des images au format DICOM et d'autre part pour la création d'un jeu de données pour l'entraînement et d'un jeu de données pour la validation. Commençons par appeler les librairies que nous utiliserons :

```
1 # -*- coding: utf-8 -*-
2
3 #from my_classes import DataGenerator
4 import os
5 import random
6 import numpy as np
```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```
7 import pandas as pd
8 import pydicom
9 import matplotlib.pyplot as plt
10 from tensorflow.keras.models import Sequential
11 from tensorflow.keras.layers import Dense, Flatten, Dropout
12 from tensorflow.keras.layers import Conv2D, MaxPool2D
13 from tensorflow.keras import optimizers
14 import keras
15 import tensorflow as tf
16 from tensorflow import keras
17 from skimage.transform import resize
```

Comme nous l'avons déjà précisé, les calculs bénéficient d'une accélération GPU. Le code suivant permet de paramétriser le calcul sur GPU.

```
19 gpus = tf.config.experimental.list_physical_devices('GPU')
20 if gpus:
21     try:
22         for gpu in gpus:
23             tf.config.experimental.set_memory_growth(gpu, True)
24         logical_gpus = tf.config.experimental.list_logical_devices('GPU')
25         print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical"
26               "GPUs")
27     except RuntimeError as e:
28         print(e)
```

Nous chargeons ensuite les fichiers au format CSV.

```
29 # =====
30 # Chargement des fichiers CSV
31 # =====
32 path = '../BaseDeDonnee/rsna-pneumonia-detection-challenge/'
33 dataP = pd.read_csv(path+"stage_2_train_labels.csv")
34 dataP2 = pd.read_csv(path+"stage_2_detailed_class_info.csv")
```

Dans la première section de ce rapport, pour l'analyse des données, nous avons utilisé une structure de type DataFrame de Pandas pour manipuler les données. Néanmoins, nous avons vu que des patients pouvaient apparaître plusieurs fois s'ils possédaient plusieurs opacités. Pour éviter cela, nous allons travailler ici avec un dictionnaire dont la clé correspondant à l'identifiant des patients. La valeur renverra un second dictionnaire dans lequel nous stockerons le chemin menant au fichier DICOM ainsi que le label.

```
36 # =====
37 # Dictionnaire contenant l'ID des patients, et les labels de sortie
38 # =====
39 # Fonction définissant le dictionnaire
40 def parse_data(dataP,dataP2):
41
42     parsed = {}
```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```
43     for n, row in dataP.iterrows():
44
45         pid = row['patientId']
46
47         # --- Vérification que le patient n'est pas déjà pris en compte
48         if pid not in parsed:
49
50             # Selon la class ajoute cree un label
51             if dataP2.iloc[n]['class'] == "Normal":
52                 ctgr = 0
53             elif dataP2.iloc[n]['class'] == "No Lung Opacity / Not Normal":
54                 :
55                 ctgr = 1
56             else :
57                 ctgr = 2
58
59             parsed[pid] = {
60                 'dicom': 'stage_2_train_images/%s.dcm' % pid,
61                 'label': ctgr}
62
63
64     # Creation du dictionnaire
65     parsed = parse_data(dataP,dataP2)
66     print("Taille du dictionnaire",len(parsed))
```

Taille du dictionnaire 26684

Nous obtenons finalement 26684 clés dans le dictionnaire, ce qui correspondant effectivement au nombre total de radiographies. Nous allons maintenant créer le jeu des données d'entraînement et de validation. Pour cela nous allons utiliser une technique permettant de regrouper les clés d'un dictionnaire par valeur, ici par *class*.

```
68     # =====
69     # Creation du jeu d'entraînement et de validation
70     # =====
71     from collections import defaultdict
72     res = defaultdict(list)
73     for key, val in sorted(parsed.items()):
74         res[val['label']].append(key)
75
76     print('Nombre de patients sans pathologie :', len(res[0]))
77     print('Nombre de patients avec pathologie mais sans pneumonie :', len(res[1]))
78     print('Nombre de patients avec pneumonie :', len(res[2]))
```

Nombre de patients sans pathologie : 8851
Nombre de patients avec pathologie mais sans pneumonie : 11821
Nombre de patients avec pneumonie : 6012

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

Comme nous pouvons le voir, les classes ne sont pas équilibrées avec pratiquement deux fois moins de patients ayant une pneumonie que de patients ayant une pathologie autre que la pneumonie. Pour que l'entraînement du modèle se passe dans les meilleures conditions, il est recommandé d'utiliser un jeu de données équilibré. C'est pourquoi nous allons conserver que 6012 patients de chaque catégorie. Il existe néanmoins une autre technique permettant de conserver l'ensemble des données, en pondérant la fonction de coût par des poids dépendant de la fréquence d'apparition de chaque classe. Nous envisageons de tester cette approche pour l'élaboration du modèle complexe.

Généralement, les données sont réparties à 80%-20% entre un jeu de données d'apprentissage et de validation :

```
80 n_total_samples_cat = min(len(res[0]),len(res[1]),len(res[2]))
81 n_valid_samples_cat = int(n_total_samples_cat*0.2)
82 n_train_samples_cat = n_total_samples_cat - n_valid_samples_cat
83 print( 'Taille de la dataset par categorie :', n_total_samples_cat )
84 print( 'Taille du validation set par categorie :', n_valid_samples_cat )
85 print( 'Taille du training set par categorie :', n_train_samples_cat )
```

```
Taille de la dataset par categorie : 6012
Taille du validation set par categorie : 1202
Taille du training set par categorie : 4810
```

On peut maintenant concaténer les données de chaque catégorie pour créer les jeux de données finaux :

```
87 n_total_samples = n_total_samples_cat*3
88 n_valid_samples = n_valid_samples_cat*3
89 n_train_samples = n_train_samples_cat*3
90
91 train_filenames = res[0][n_valid_samples_cat:n_total_samples_cat]
92 train_filenames[n_train_samples_cat:2*n_train_samples_cat] = res[1][
    n_valid_samples_cat:n_total_samples_cat]
93 train_filenames[2*n_train_samples_cat:3*n_train_samples_cat] = res[2][
    n_valid_samples_cat:n_total_samples_cat]
94
95 valid_filenames = res[0][:n_valid_samples_cat]
96 valid_filenames[n_valid_samples_cat:2*n_valid_samples_cat] = res[1][:[
    n_valid_samples_cat]
97 valid_filenames[2*n_valid_samples_cat:3*n_valid_samples_cat] = res[2][:
    n_valid_samples_cat]
98
99 print("Taille du jeu d'entraînement :",len(train_filenames))
100 print("Taille du jeu de validation :",len(valid_filenames))
```

```
Taille du jeu d'entraînement : 14430
Taille du jeu de validation : 3606
```

Nous allons maintenant créer la classe qui nous servira de *data generator*. Cette classe hérite de la classe *keras.utils.Sequence*. Nous devons principalement définir les tailles des mini-lots, la lecture des images DICOM et des labels associés.

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```
102 # =====
103 # Classe permettant la creation du datagenerator
104 # =====
105 class DataGenerator(keras.utils.Sequence):
106
107     def __init__(self, folder, filenames, pneumonia_locations=None,
108                  batch_size=16, image_size=256, n_channels=1,
109                  n_classes=3, shuffle=True, augment=False, predict=False):
110         'Initialisation'
111         self.folder = folder
112         self.filenames = filenames
113         self.pneumonia_locations = pneumonia_locations
114         self.batch_size = batch_size
115         self.image_size = image_size
116         self.n_channels = n_channels
117         self.n_classes = n_classes
118         self.shuffle = shuffle
119         self.augment = augment
120         self.predict = predict
121         self.on_epoch_end()
122
123
124     def __len__():
125         'Calcul le nombre de mini-lots par epoch'
126         return int(len(self.filenames) / self.batch_size)
127
128
129     def __getitem__(self, index):
130         'Lecture des donnees'
131
132         # Selectionne un nombre de fichier
133         filenames = self.filenames[index*self.batch_size:(index+1)*self.
134                                     batch_size]
135
136
137         # ouverture des images et des labels
138         items = [self.__data_generation(filename) for filename in filenames
139                  ]
140
141
142         # decoupage et creation des entrees et label
143         X, y = zip(*items)
144         X = np.array(X)
145         y = np.array(y)
146
147
148         # On melange les datas apres chaque epoch
149         def on_epoch_end():
150             'Mise a jour apres chaque epoch'
```

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

```
148     if self.shuffle:
149         random.shuffle(self.filenames)
150
151
152     def __data_generation(self, filename):
153
154         # chargement de l'image
155         img = pydicom.dcmread(os.path.join(self.folder, filename+'.dcm')).pixel_array
156
157         if filename in self.pneumonia_locations:
158             # chargement du label
159             y = self.pneumonia_locations[filename]['label']
160
161         # redimensionnement de l'image
162         X = resize(img, (img_dims, img_dims, 1))
163
164     return X, keras.utils.to_categorical(y, num_classes=self.n_classes)
```

Finalement, nous pouvons utiliser la classe que l'on vient de définir pour créer des générateurs des jeux de données d'apprentissage et de validation.

```
166 # =====
167 # Creation des generators pour les jeux d'apprentissage et de validation
168 # =====
169 img_dims = 224
170 folder = path+'stage_2_train_images'
171 training_generator = DataGenerator(folder, train_filenames,
172                                     pneumonia_locations=parsed, batch_size=8, image_size=img_dims, augment=True)
172 validation_generator = DataGenerator(folder, valid_filenames,
173                                     pneumonia_locations=parsed, batch_size=8, image_size=img_dims, shuffle=False)
```

Désormais, le modèle que l'on définira pourra accéder aux données de manière optimale.

3.3.2 Diagnostic du modèle d'apprentissage

Le but d'apprentissage automatique est de conclure à un modèle qui représente au mieux les données d'entrée tout en étant bien adapté à de nouvelles données (capacité de généralisation). Ce processus d'apprentissage d'un réseau de neurones consiste à ajuster et optimiser les paramètres (poids) de chaque couche en minimisant l'erreur de prédiction. Cette erreur est calculée entre les vraies valeurs et les valeurs prédictives du modèle par le biais d'une fonction de coût, appelée aussi fonction de perte (*loss* en anglais). En effet, cette fonction est une indicatrice de l'erreur totale du modèle sur l'ensemble du jeu de données d'entraînement que l'on va chercher à minimiser et une indicatrice sur la capacité de généralisation du modèle sur l'ensemble de validation. Ainsi, représenter l'évolution de cette fonction *loss* à chaque *epoch* du CNN pour l'ensemble d'entraînement et de validation constitue un bon diagnostic du modèle d'apprentissage. Le coût d'adaptation des paramètres appris du réseau est alors présenté à chaque *epoch* du CNN pour atteindre la

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

prédiction souhaitée.

Il existe plusieurs fonctions coût en fonction de la tâche à réaliser. Dans notre cas de classification, la vraie valeur (vérité) y se présente sous la forme d'un *encodage one-hot*. Autrement dit, pour un problème à n classes, y est encodé sous la forme $(0, \dots, 0, 1, 0, \dots, 0)$, c.-à-d. un vecteur dont toutes les composantes sont nulles à l'exception de la $k^{\text{ème}}$ (classe d'appartenance de y). Dans ce cas, la fonction d'entropie croisée

$$H(z, y) = - \sum_{i=1}^n y_i \log(z_i),$$

est préférée pour la minimisation. Dans ce cas, les activations en sortie du réseau z_i ont été passés dans une fonction softmax

$$z_i = \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)},$$

qui n'est rien d'autre que la généralisation de la sigmoïde (fonction utilisée pour la classification binaire) au cas multiclass.

Il est important à noter que la convergence de l'algorithme de descente de gradient (utilisé pour la minimisation de la fonction coût) n'est jamais garantie. D'une part, elle n'est garantie que dans le cas d'une fonction coût convexe, ce qui est rarement le cas pour les réseaux de neurones profonds. Et d'autre part, plusieurs paramètres peuvent influencer cette convergence et donc la minimisation de cette fonction *loss*. En effet, le pas de descente appelé aussi le taux d'apprentissage (*learning rate* en anglais) joue un rôle important dans l'optimisation des réseaux profonds puisqu'il contrôle l'amplitude des mises à jour des paramètres (poids) à optimiser du modèle. Ainsi, un *learning rate* très élevé entraîne souvent une convergence instable ; alors qu'un *learning rate* faible ralentit la convergence et la bloque souvent dans des minimums locaux. Il faut donc trouver le bon compromis. En effet, il est recommandé de démarrer avec un *learning rate* élevé pour accélérer la descente de gradient et de le réduire par la suite. D'autres études se sont concentrées sur des *learning rate* adaptatifs ; autrement dit ils s'ajustent automatiquement dans l'apprentissage [D. Zeiler, 2012].

Un deuxième point crucial qui peut aussi influencer la minimisation de la fonction coût est l'initialisation des poids du modèle. Cette initialisation peut être effectuée avec des valeurs aléatoires limitant l'explosion des gradients.

Tous ces hyperparamètres et bien d'autres déjà discutés dans la partie 3.2.2 influencent la minimisation de la fonction coût et plus généralement la performance des modèles entraînés.

Il est important de noter que la descente de gradient minimise l'erreur sur le jeu d'entraînement, mais l'erreur réellement intéressante est celle commise sur le jeu de validation permettant de juger la capacité de généralisation du modèle.

3.3.3 Résultats

Nous avons entraîné notre base de données sur les architectures VGG16, VGG19, DenseNet121 et DenseNet169. Ce qui nous intéresse ici, c'est principalement d'observer comment se comporte l'entraînement du modèle sur les différentes architectures. L'idée est avant tout de se familiariser avec notre modèle, de voir le comportement de certains paramètres (Learning rate, régularisation, weight decay,...), et ceci afin d'accumuler un maximum de connaissances pour l'élaboration du modèle final. Ainsi nous ne regarderons, que l'évolution des courbes de convergence de la précision (*Accuracy*) et des pertes (*Loss*). Pour

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

chaque architecture, nous avons testé plusieurs paramètres comme le *learning rate*, l'initialisation des poids, le décrochage (*dropout*), et plusieurs autres. Néanmoins, nous ne présenterons ici que les résultats les plus pertinents. Dans la majorité des cas, les résultats étaient relativement similaires.

Les paramètres du modèle par défaut que nous avons utilisé dans nos modèles sont les suivants :

- Taille de l'image après redimensionnement : 224x224
- Batch size : 8
- Initialisation des poids : par défaut (*glorot_uniform*)
- Dropout : sans (sauf s'il fait parti de l'architecture)
- Epoch : 25
- Optimizer : *RMSprop*
- Learning rate : 1e-4
- weight decay : sans

Pour toutes les architectures considérées, nous avons imposé le nombre maximum d'epochs à 25, néanmoins, la plupart du temps, nous stoppons manuellement la convergence dès lors qu'un phénomène de sous-apprentissage ou sur-apprentissage apparaît. Ceci se manifeste généralement par une valeur de fonction de perte (loss) qui décroît sur l'ensemble d'apprentissage et au contraire qui augmente sur l'ensemble de validation.

Nous avons appliqué l'architecture VGG-16 pour la base de données du RSNA pour la détection de pneumonie. Nous avons suivi la démarche de Simonyan et Zisserman en redimensionnant les images DICOM à 224×224 pixels (dimension fixée de l'image d'entrée pour ConvNets) par le biais de la fonction *resize* (dans *def __data_generation*). Nous avons tracé l'*accuracy* et le *loss* dans la figure 9. Nous pouvons voir que les courbes de *validation set* suivent la même tendance que les courbes du *training set*. De plus, l'*accuracy* et le *training accuracy* augmentent au fur et à mesure que le nombre d'*epochs* augmente et que le *loss* et le *validation loss* eux diminuent, ce qui est plutôt correct. Néanmoins, on peut apercevoir une instabilité dans la convergence des courbes de validation, il faudrait lancer l'apprentissage sur plusieurs centaines d'*epochs* pour vérifier cette tendance. De plus, des techniques d'amélioration sont à prévoir pour une meilleure convergence.

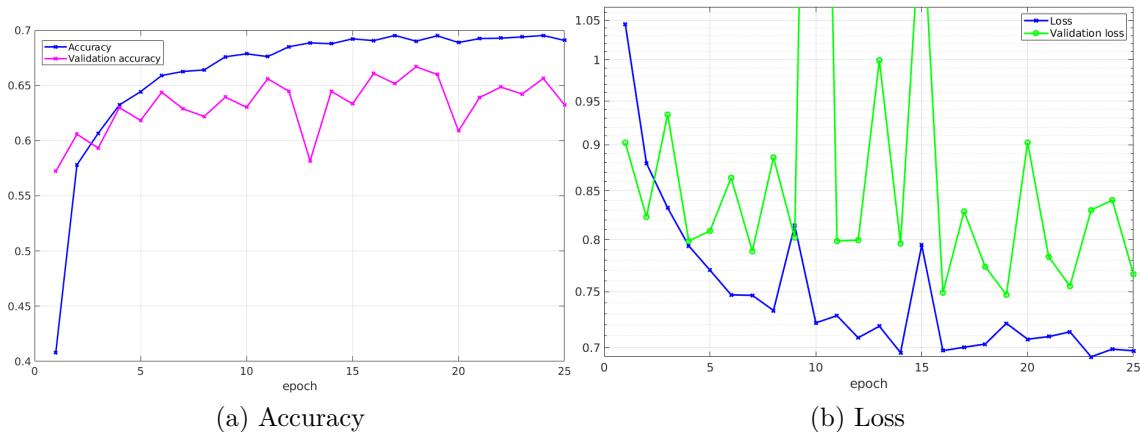


FIGURE 9 – Résultats de convergence obtenu pour VGG16

En ce qui concerne l'architecture VGG19, nous présentons ici plusieurs résultats. Tout

3. MODÈLES SIMPLES DE CLASSIFICATION D'IMAGES

d'abord, pour le test1 nous avons utilisé les paramètres par défaut présentés ci-dessus. Nous observons dans la figure 10 qu'il n'y a aucune évolution pour l'*accuracy / validation accuracy* et pour le *loss / validation loss*. Ainsi, nous avons changé d'une part l'initialisation des poids pour le test2 et test3 avec *he_normal* et d'autre part, nous avons ajouté un décrochage de 20% pour le test3. Ces changements ont abouti à de meilleures convergences pour les deux courbes *accuracy / loss*. Néanmoins, nous remarquons le même phénomène d'instabilité que nous avons observé sur les résultats pour l'architecture VGG16.

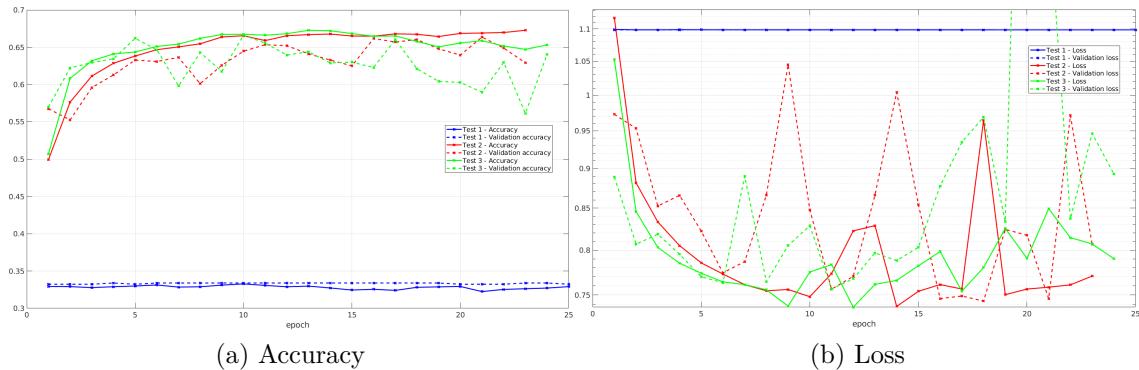


FIGURE 10 – Résultats de convergence obtenu pour VGG19

Pour les architectures DenseNet121 (Figure 11) et DenseNet169 (Figure 12), un phénomène de sur-apprentissage apparaît. En effet, la valeur de la fonction de perte *loss* décroît sur l'ensemble d'apprentissage et au contraire croît sur l'ensemble de validation. Ce phénomène de sur-apprentissage (*overfitting*) se confirme également sur les courbes d'*accuracy*.

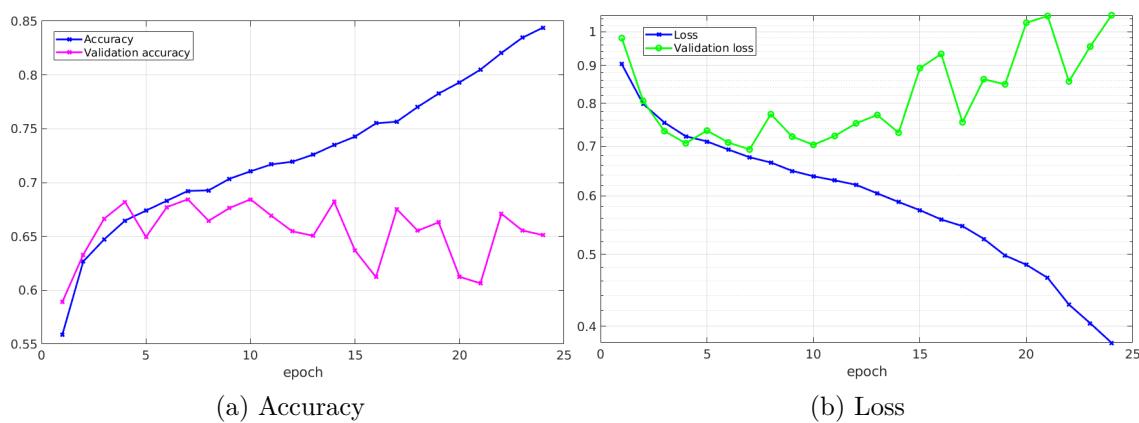


FIGURE 11 – Résultats de convergence obtenu pour DenseNet121

Il s'agit ici des résultats les plus marquants que j'ai obtenus pour classifier les images DICOM selon leurs classes sur des architectures simples.

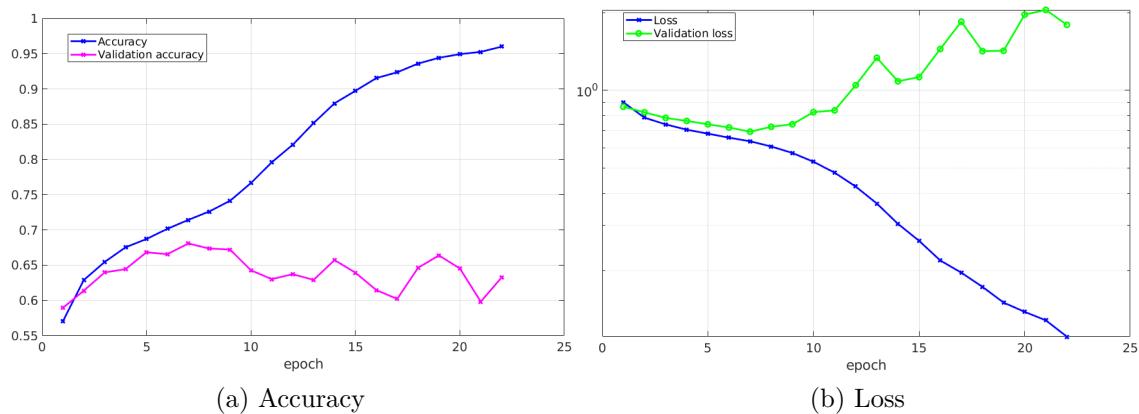


FIGURE 12 – Résultats de convergence obtenu pour DenseNet169

Références

- (2017). Deaths : Final data for 2015. *National Vital Statistics Reports*, 66(6). Supplemental Tables. Tables I-21, I-22. (Cité à la page 4)
- D. Zeiler, M. (2012). Adadelta : An adaptive learning rate method. *Dans : arXiv : 1212.5701 [cs]*. (Cité à la page 62)
- Huang, G., Liu, Z., Maaten, L. V. D., and Q. Weinberger, K. (2017). Densely connected convolutional networks. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI*, pages 2261–2269. (Cité à la page 55, 55)
- Ioffe, S. and Szegedy, C. (2015). Batch normalization : Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv :1502.03167*. (Cité à la page 50)
- Kelly, B. (2012). The chest radiograph. *The Ulster Medical Journal*, 81(3) :143. (Cité à la page 4)
- Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957. (Cité à la page 48)
- Nambu, A., Ozawa, K., Kobayashi, N., and Tago, M. (2014). Imaging of community-acquired pneumonia : Roles of imaging examinations, imaging diagnosis of specific pathogens and discrimination from noninfectious diseases. *World Journal of Radiology*, 6(10) :779–793. (Cité à la page 4)
- Rajpurkar, P., Irvin, J., Zhu, K., Yang, B., Mehta, H., Duan, T., Ding, D., Bagul, A., L. Ball, R., Langlotz, C., Shpanskaya, K., Lungren, M., and Y. Ng, A. (2017). Chexnet : Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv :1711.05225v3 [cs.CV] 25 Dec 2017*, page 1–7. (Cité à la page 55, 55)
- Rui, P. and Kang, K. (2015). *National Ambulatory Medical Care Survey : 2015 Emergency Department Summary Tables*. Table 27. (Cité à la page 4)
- Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *Dans : Proceedings of International Conference on Learning Representations*. (Cité à la page 51, 51)
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout : a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1) :1929–1958. (Cité à la page 48)
- Vasilev, I., Slater, D., Spacagna, G., Roelants, P., and Zocca, V. (2019). *Python Deep Learning Second Edition*. Packt Publishing Ltd, Livery Place 35 Livery Street Birmingham B3 2PB, UK. (Cité à la page 52)