

# Rapport d'étude sur l'intelligence artificielle

Déterminer si un poumon est sain, ou atteint  
d'une maladie virale ou bactérienne

Valentin NOEL, Jonathan KHALIFA

Fabrice Sumsa, Bastien ANGLES et Kévan SADEGHI

Mai 2021

# Table des matières

<b>Table des matières</b>	<b>2</b>
<b>Introduction</b>	<b>4</b>
But	4
Description du projet	4
<b>Les différents algorithmes de détection</b>	<b>5</b>
Les réseaux de neurones	5
<i>K-Nearest Neighbor algorithm (KNN)</i>	5
<i>Artificial neural network (ANN)</i>	5
<i>Réseaux de Neurones à Convolution (CNN)</i>	5
Les arbres décisionnels	5
<i>Les Random Forests</i>	6
<i>Les Boosted Trees</i>	6
Nos choix finaux	6
<b>Théorie et Algorithmes</b>	<b>7</b>
Principes généraux	7
<i>Overfitting</i>	7
<i>Underfitting</i>	7
<i>Bias</i>	7
<i>Variance</i>	7
<i>Descente de gradient</i>	7
<i>Fonction d'activation</i>	8
<i>Un neurone</i>	8
Hyperparamètres	8
Préparation des données	9
<i>Premier aperçu de notre dataset</i>	9
<i>Nettoyage des données</i>	9
<i>Redimensionner les images</i>	10

<i>Gérer la colorimétrie</i>	10
<i>Délimitation des zones de recherches</i>	11
<i>Rééquilibrage Test / Train</i>	12
<b>Résultats</b>	12
<i>Data Augmentation</i>	13
Réseau neuronal à convolution	14
<i>Fonctionnement</i>	14
<i>Architecture du modèle</i>	16
<i>Choix techniques</i>	17
Boosted Tree	21
<i>Fonctionnement</i>	21
<i>Architecture du modèle</i>	27
Choix techniques	28
<b>Résultats</b>	32
Réseaux de Neurones à Convolution (CNN)	32
Boosted Trees	34
<b>Discussions</b>	36
<b>Conclusion</b>	37
<b>Sources et Références</b>	38

# Introduction

## But

Le but de ce projet dans le cadre d'une initiation à l'IA est de pouvoir déterminer à l'aide d'un algorithme d'intelligence artificielle, si la radiographie d'un poumon est sain, ou s'il est atteint d'une pneumonie, et si tel est le cas, déterminer la nature de cette maladie, qu'elle soit virale ou bactérienne.

## Description du projet

Pour ce premier projet en IA il nous a été fourni un jeu de données constitué de 5856 images. Ces images sont des radiographies thoraciques provenant d'un hôpital. Il nous faudra donc trouver quel type de modèle algorithmique est propice à la classification d'images.

Ce même jeu de données est brut et n'a donc pas été préparé pour de l'entraînement sur algorithmes. Il fera donc partie de nos tâches de le nettoyer avant de pouvoir l'exploiter.

Le modèle que nous auront choisi devra donc s'entraîner sur ce jeu de données pour apporter ensuite des résultats concluants en matière de prédictions sur des images qu'il n'a jamais vu.

Le projet consiste donc à découvrir l'importance de trois étapes clés lors de la résolution d'un problème grâce à l'utilisation de l'IA, à savoir : Le choix du modèle algorithmique, la préparation des données ainsi que l'interprétation des résultats.

## Les différents algorithmes de détection

Pour mener ce projet à bien et essayer d'avoir des résultats le plus précis possible, il nous a fallu faire un choix quant à l'algorithme que notre IA allait utiliser. Cependant, il existe de nombreux types d'algorithmes différents qui ont tous leurs avantages et leurs inconvénients, ceux-ci catégorisés dans plusieurs familles différentes. Pour essayer d'en apprendre plus sur l'IA en générale, et ne pas avoir une source de résultats uniques, nous avons décidé d'utiliser deux algorithmes qui proviennent de deux familles différentes : Les réseaux de neurones et les arbres de décision.

Voici une petite description de plusieurs modèles que nous avons analysés.

### Les réseaux de neurones

#### K-Nearest Neighbor algorithm (KNN)

La "méthode des k plus proches voisins" est un algorithme de classification ou de régression permettant l'analyse d'images représentées dans l'espace (distance entre deux points d'une image par exemple). L'avantage de cet algorithme est sa capacité à pouvoir traiter de multiples classes (A.K.A: Chats, Chiens, Lapins, etc...).

Cependant, son plus grand point faible est le fait d'utiliser l'entièreté du jeu de données d'apprentissage pour faire une classification, ce qui peut mener à des erreurs de prédictions notamment dû à un jeu de données relativement petit.

#### Artificial neural network (ANN)

Le réseau ANN est le plus simple réseau de neurone existant, il est utilisé pour faire de la reconnaissance faciale, et plus généralement pour de la Computer Vision.

Cependant, ce genre de réseau implique des problèmes liés aux performances hardware de la machine qui exécute l'apprentissage, ce qui peut impliquer des comportements inattendus dans l'apprentissage de celui-ci.

#### Réseaux de Neurones à Convolution (CNN)

Les réseaux de neurones à convolutions ont été créés sur la base de principes physiques et biologiques, humain et animal. En effet, les neurones de ce réseau sont créés de telle sorte à ce qu'il correspondent à des régions qui se chevauchent lors du pavage du champ visuel, une mécanique empruntée aux cerveaux humain et animal.

Ce type de réseaux de neurones est utilisé pour la reconnaissance d'images, de vidéos, mais également pour les systèmes de recommandation et le traitement du langage naturel.

### Les arbres décisionnels

L'arbre de décision est un simple diagramme de prise de décision. L'idée est de traverser un unique arbre de décision avec plusieurs conditions pour atteindre une solution. Cette approche, bien que basique et ne pouvant pas être appliquée à des cas concrets, reste la fondation des Random Forest et des Boosted Trees.

Les Random Forests et les Boosted Tree sont des algorithmes de machine learning performants pour tout ce qui est arbre de décision. Ils sont principalement utilisés pour des problèmes de régression ou de classification. Bien que ce soit tout deux des arbres de décisions, ils ne sont pas créés de la même manière, et ont chacun leurs avantages et désavantages.

### **Les Random Forests**

Les Random Forests utilisent un très grand nombre d'arbres, où tous les résultats de chaque arbre sont combinés, souvent par probabilité, pour fournir une probabilité unique.

### **Les Boosted Trees**

Les Boosted Trees se composent aussi d'arbres de décision, mais contrairement aux Random Forests, les arbres sont générés de manière séquentielle. Chaque arbre utilise les résultats de l'arbre précédent pour corriger ses erreurs et créer un arbre encore plus optimisé que le précédent.

## **Nos choix finaux**

Au départ, nous avions envisagé d'utiliser un réseau de neurones artificiel (ANN). Cependant, nous avons vite vu que nous allions nous retrouver avec une quantité énorme de paramètres et donc un temps d'apprentissage trop long. Au vu des observations précédentes, nous avons décidé d'utiliser un Réseau de Neurones à Convolution (CNN) qui constitue une approche incontournable dans la classification d'images.

De l'autre côté, nous avons utilisé un algorithme de Boosted Tree pour mettre en opposition les résultats du CNN et tenter une approche avec des arbres décisionnels.

# Théorie et Algorithmes

Dans cette partie, nous allons expliquer comment fonctionnent les différents algorithmes que nous avons choisis, la théorie qui les régit, de manière à comprendre et interpréter plus facilement les résultats par la suite.

## Principes généraux

Avant de se lancer dans l'explication du fonctionnement des algorithmes, nous allons rapidement faire une petite explication des différents termes qui leur sont communs, ainsi qu'une présentation des modifications que l'on a faites au dataset pour les rendre uniformes.

### Overfitting

Il s'agit d'un problème qui est souvent rencontré en machine learning. Il survient lorsque notre modèle essaie de trop coller aux données d'entraînements quitte à négliger la prédiction finale pour se rapprocher aux maximum de connaissances du modèle.

### Underfitting

Il s'agit d'un autre problème qui est également rencontré en machine learning, à l'instar de l'overfitting, un modèle souffrant d'underfitting, aura tendance à ne pas assez apprendre de son jeu de données, quitte à donner des prédictions totalement fausse aléatoirement.

### Bias

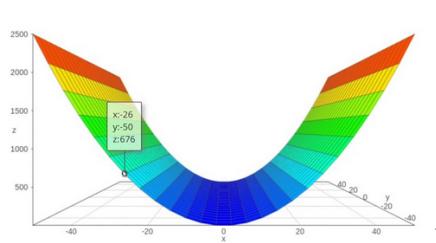
Il s'agit là d'une démarche où un procédé engendre des erreurs dans les résultats d'une étude. Plus le bias est élevé, plus le modèle risque de souffrir d'underfitting.

### Variance

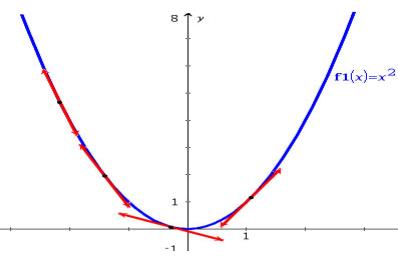
La variance est une mesure de la dispersion des valeurs d'un échantillon ou d'une distribution de probabilité, cela nous servira à prédire si les résultats proposés sont éloignés ou non de la réalité, plus la variance est grande, moins les prédictions sont justes. Plus la variance est élevée, plus le modèle risque de souffrir d'overfitting.

### Descente de gradient

La Descente de Gradient est un algorithme d'optimisation qui permet de trouver le minimum de n'importe quelle fonction convexe en convergeant progressivement vers celui-ci. Une fonction convexe est une fonction dont l'allure ressemble à celle d'une belle vallée avec au centre un minimum global.



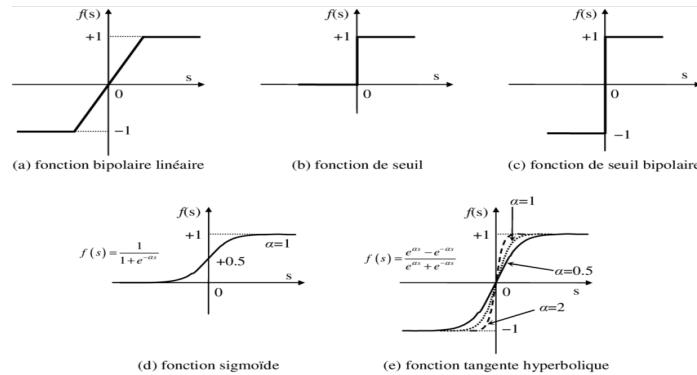
(Exemple de recherche d'un minimum)



(Exemple de fonction convexe)

## Fonction d'activation

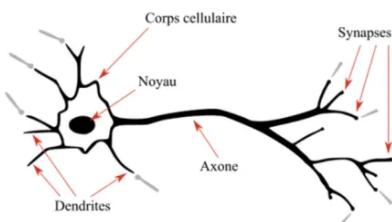
Il s'agit d'une fonction mathématique appliquée à un signal en sortie d'un neurone artificiel. Le terme de "fonction d'activation" vient de l'équivalent biologique : "potentiel d'activation". Selon un seuil d'activation, qui, une fois atteint, entraîne une réponse du neurone.



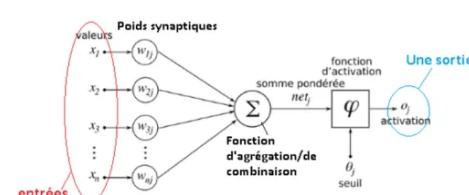
(exemple de différentes fonctions d'activation)

## Un neurone

Aussi appelé un neurone formel en informatique et en statistiques, il s'agit d'une représentation mathématique et informatique d'un neurone biologique. Il reproduit certaines caractéristiques biologiques, en particulier les dendrites, axones et synapses, au moyen de fonctions et de valeurs numériques.



NEURONE BIOLOGIQUE



NEURONE ARTIFICIEL

(source : <https://deeplearning.fr/cours-theoriques-deep-learning/fonctionnement-du-neurone-articiel/>)

## Hyperparamètres

C'est une multitude de paramètres dont les valeurs sont utilisées pour contrôler le processus d'apprentissage.

Exemple d'hyper-paramètres: Learning Rate, Batch Size, Nombre de neurones d'entrées dans le réseaux, Nombre de neurones cachées dans le réseaux, etc...

## Préparation des données

La première grande étape à laquelle nous avons été confrontés consiste à rendre le jeu de données que l'on nous a fourni exploitable. Une fois le dataset nettoyé, il pourra être utilisé et réutilisé par tous nos modèles.

### Premier aperçu de notre dataset

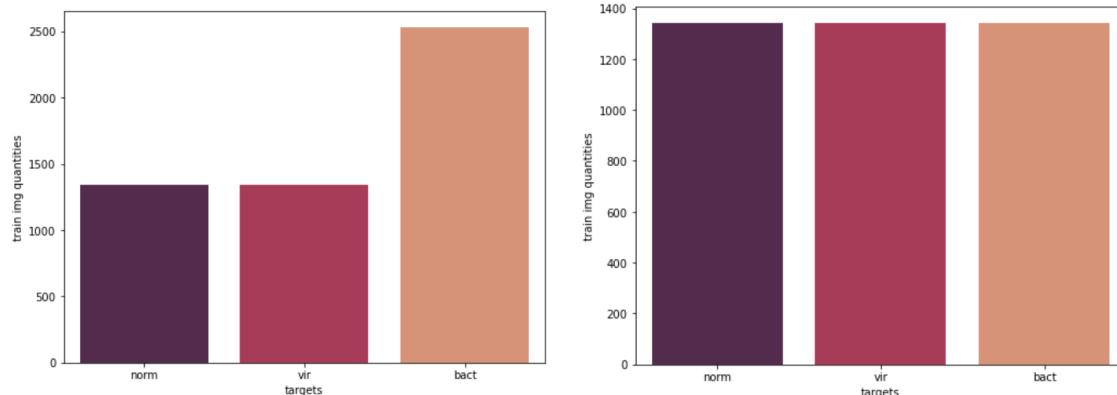
Le dataset qui nous a été fourni est une collection de 5856 images représentant des radiographies thoraciques. Le dataset a été divisé en 3 sous parties : TRAIN , TEST et VAL. Chaque sous partie contient des dossiers libellés NORMAL et PNEUMONIA.

Nous remarquons que les images situées dans les dossiers PNEUMONIA présentent systématiquement le mot « bacteria » ou « virus » dans le nom du fichier. Ceci va nous permettre de distinguer au final 3 targets qui seront Normal, Virus et Bacteria et nous permettront d'entraîner nos modèles sur ces trois critères.

### Nettoyage des données

Le nettoyage des données est une étape qui peut sembler longue, mais elle est indispensable pour nous permettre de faire des entraînements dans les meilleures conditions possibles.

Tout d'abord nous devons nous assurer que la répartition des images au sein de nos 3 targets soit homogène. Un déséquilibre pourrait désavantager l'une ou l'autre target au profit d'un sur-apprentissage de la target dominante.

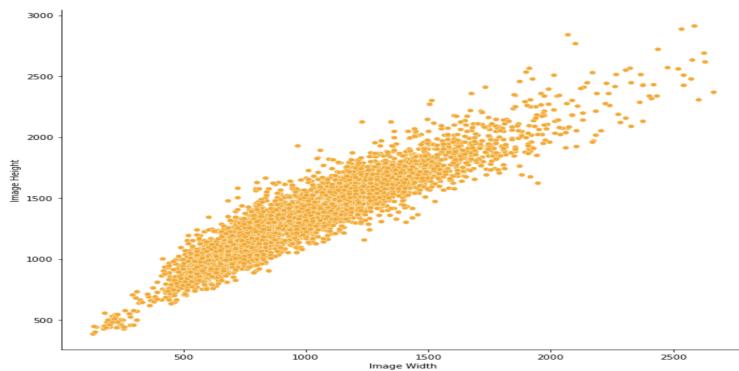


Dans notre cas, nous distinguons de fortes disparités dans les dossiers TRAIN et TEST. Le graph de gauche correspondant au dossier TRAIN illustre une dominance d'images Bacteria.

Nous avons donc rééquilibré les proportions (graph de droite), mais non sans amputer près de 1'500 images. Il s'agit là certes d'une perte importante d'informations, mais nous y remédierons plus tard grâce à de la data augmentation.

## Redimensionner les images

Dans le but d'entraîner notre modèle, nous devons également nous assurer que toutes les images ont les mêmes dimensions. Là encore, nous avons des surprises comme le témoigne le graph ci dessous :



Nous avons donc redimensionné toutes nos images au format 64x64 pixels. Ce choix fut un bon compromis entre un temps d'entraînement limité par la puissance de calcul de nos machines et un résultat obtenu en matière de précision.

A noter cependant que d'après cette publication scientifique parue sur ce site de publications radiographiques (<https://pubs.rsna.org/doi/full/10.1148/ryai.2019190015>), les meilleurs résultats pour de l'imagerie médicale sont obtenus avec des images entre 256 et 320 pixels de côté.

## Gérer la colorimétrie

Nos images devront toutes avoir le même nombre de canaux colorimétriques. Nous les convertirons toutes sur 3 canaux, R,G et B. Pourquoi RGB et non pas en nuances de gris nous direz-vous?

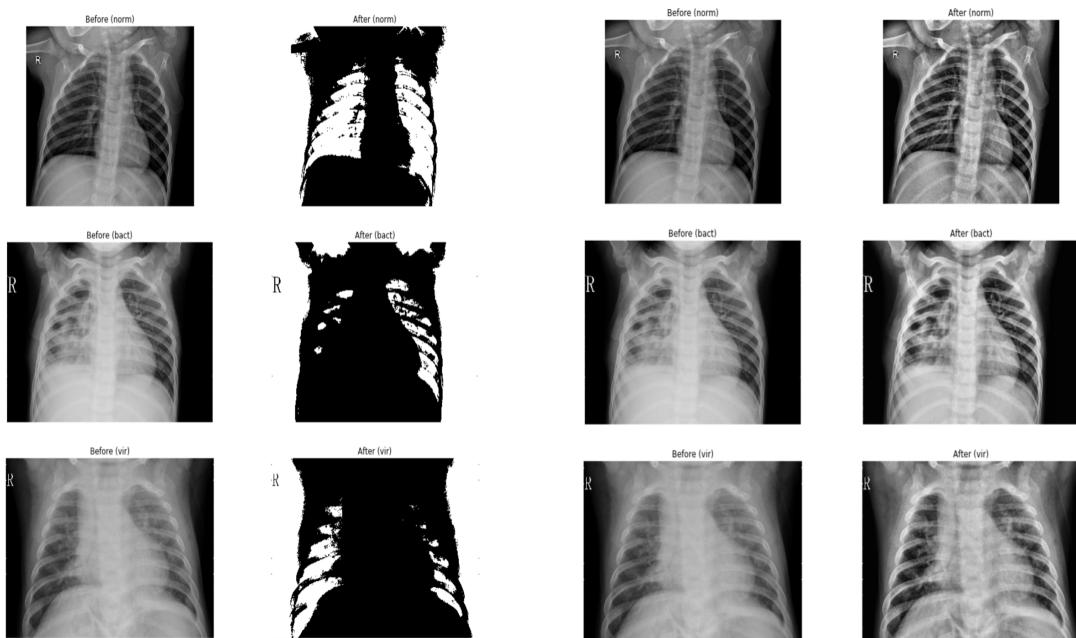
Chaque pixel de nos images correspond à une valeur comprise entre 0 et 255. Nous normalisons tous nos pixels pour obtenir des valeurs comprises entre 0 et 1 en divisant la valeur de chaque pixel par la valeur maximale, soit 255. Cette opération réduira la complexité du problème que notre modèle essaye de solutionner. Par conséquent, le modèle peut s'entraîner plus rapidement et potentiellement avoir une meilleure précision.

### Délimitation des zones de recherches

Pour aller un peu plus loin dans la préparation de nos données, nous avons voulu enrichir nos images en leur appliquant des traitements. Le but étant toujours d'optimiser l'apprentissage de notre modèle. Pour se faire, avons explorés deux approches:

La première consiste à segmenter les images afin de délimiter les pixels de part et d'autre d'une valeur seuil. Le but étant de séparer clairement les poumons sains des poumons malades. Cette technique nous a permis d'obtenir de très bons résultats lorsque notre modèle ne se contentait que de deux targets (normal et pneumonia). En revanche, la segmentation sur un modèle à trois targets n'a pas été convaincante. Le modèle ne parvenait pas à différencier les bactéries des virus. (voir Figure 1)

La seconde approche fut celle que nous avons gardé. Celle-ci consistait à optimiser l'équilibre des contrastes afin d'éliminer le bruit et rendre les images plus détaillées. Nous avons utilisé l'algorithme CLAHE (Contrast-Limited Adaptive Histogram Equalization) présent dans la librairie OpenCV. Les résultats ont été concluants. (voir Figure 2)



(fig. 1 : première approche)

(fig.2 : deuxième approche)

### Rééquilibrage Test / Train

L'étape suivante consiste à rééquilibrer la proportion d'images TEST et TRAIN. Pour se faire, nous avons redistribué équitablement nos images en trois blocs. Chaque bloc correspond à un tiers du dataset à l'exception de nos 16 images du dossier VAL. Chaque bloc possède le même nombre d'images NORM, VIR, et BACT. De cette façon nous pourrons faire lors de l'entraînement un k-fold cross-validation. Il suffira de concaténer deux des trois blocs pour former les données de TRAIN, et le bloc restant sera les données de TEST. Soit une proportion de 66% TRAIN, 33% TEST et 1%VAL.

Ce split est cohérent pour un bon entraînement de notre modèle.

Pour qu'un algorithme puisse s'entraîner sur nos images, nous devons les libellés. De par le nom de chaque fichier d'image nous allons pouvoir libeller chaque image en fonction des trois critères que nous recherchons : norm, bact, vir.

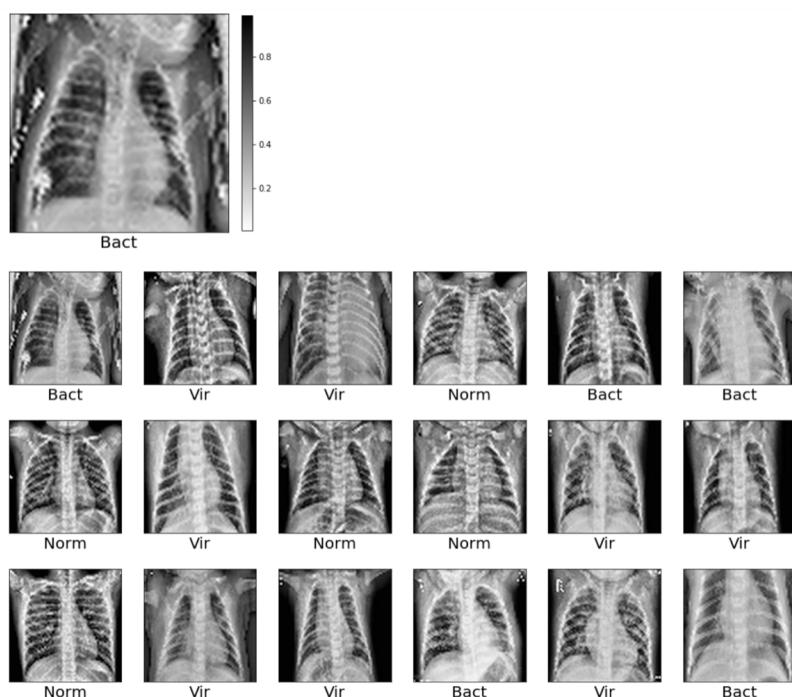
Le libellé est sous un format de matrice plus communément appelé one-hot encoding. Dans notre cas il se présente comme ceci :

- [ 1, 0, 0 ] pour un patient sain
- [ 0, 1, 0 ] pour un patient souffrant de pneumonie virale
- [ 0, 0, 1 ] pour un patient souffrant de pneumonie bactérienne

Chaque indice correspond donc à un target. A ce stade nous avons des images de même forme (shape), normalisées, enrichies par CLAHE, et en quantités similaires parmi les trois targets.

### Résultats

Une fois tous ces changements apportés à nos données, nous obtenons des images comme celles de l'échantillon ci-dessous.



## Data Augmentation

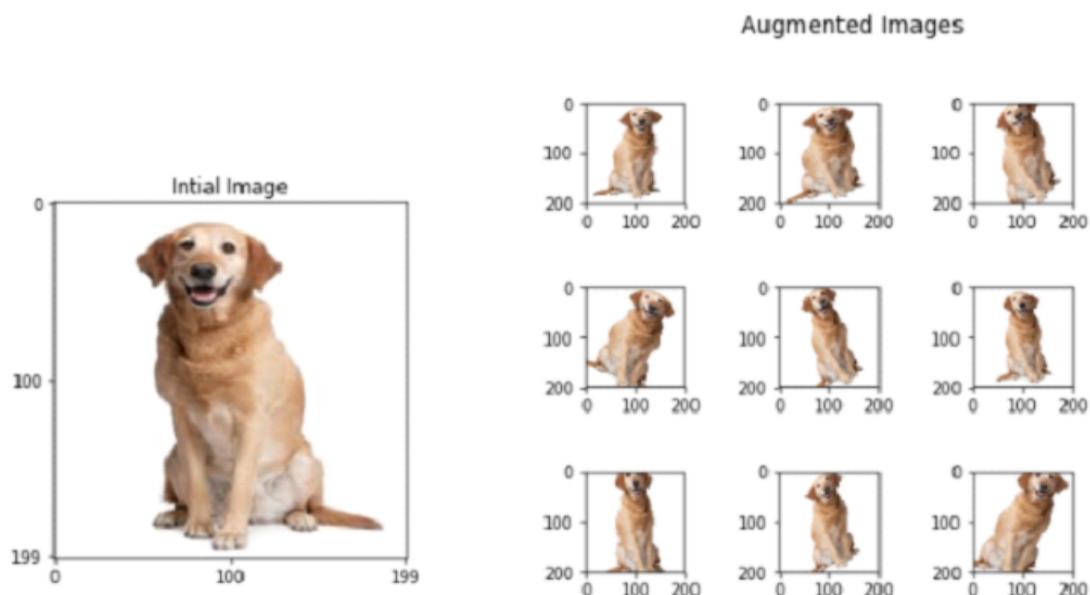
La dernière étape de la préparation de nos données sera de pallier au nombre insuffisant d'images présentes dans le dataset.

Suite au rééquilibrage des proportions correspondant à chaque target, nous avons réduit d'environ 25% le volume notre dataset initial. Nous craignons n'avoir plus assez d'images pour correctement entraîner notre modèle et risque de faire de l'overfitting trop facilement.

Pour éviter ce désagrément nous avons artificiellement générée de nouvelles images à partir de celles déjà présentes dans notre dataset. Les images artificielles présentent une série de modifications légères sans pour autant les dénaturer et conserveront le même label.

C'est via le module Image Data Generator de Keras que nous y parviendront.

Les modifications apportées sont diverses comme une rotation, inversion, amincir ou étirer sur un axe, assombrir, zoomer etc... Illustration à titre d'exemple ci-dessous:



## Réseau neuronal à convolution

### Fonctionnement

Pour ce projet, le modèle que nous avons retenu est un réseau neuronal de convolution. Les modèles CNN sont populaires pour faire de la classification ou reconnaissance d'images pour leur excellente précision. Nous comparerons les résultats de notre CNN avec le modèle Boosted Tree dans la partie discussions afin de confirmer notre choix.

Le concept de la convolution consiste en une petite matrice qui se déplace pour être appliquée à une image entière. Chaque case dans la matrice correspond un poids qu'elle appliquera sur le pixel où elle se trouve. Si l'on additionne les produits de chaque poids avec la valeur de leurs pixels alors on obtient une valeur de sortie. Après avoir parcouru toute l'image, on obtient une nouvelle image de résolution inférieure.

Une couche de convolution est créée lorsque nous appliquons plusieurs matrices aux images d'entrée. Les valeurs des poids seront mis à jour par rétropropagation du gradient et la couche sera entraînée à déterminer les meilleures valeurs de poids de filtre. Illustration ci-dessous :

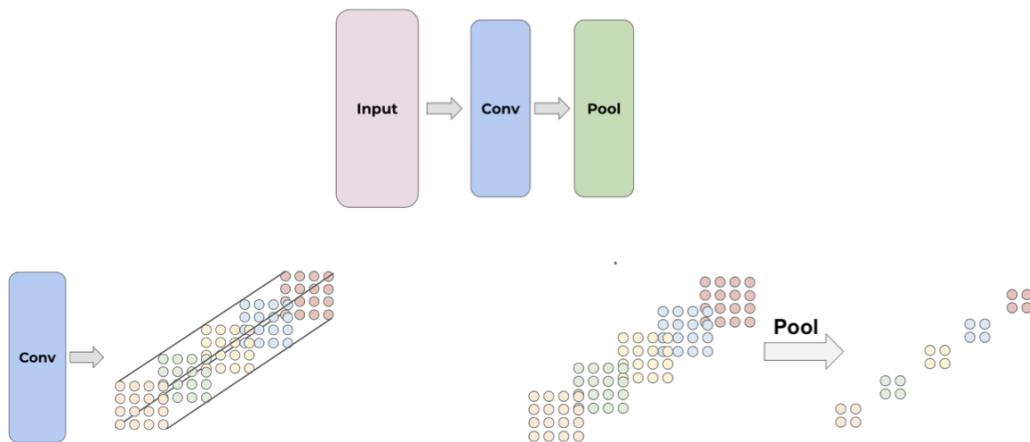


Dans Tensor Flow nos couches de convolution auront pour paramètres le nombre de nos matrices sous le nom de « filters », les dimensions de chaque matrice sous « kernel\_size », puis le pas de déplacement sous « stride ». Nous n'appliquerons pas de bord « padding » à notre image car le bord de nos radiographies ne comportent pas d'éléments utiles à la résolution de notre problème.

Nos couches de convolution seront aussi dotées d'une fonction d'activation, ce qui apportera de la non-linéarité dans notre réseau. Sans fonction d'activation, notre CNN ne serait finalement qu'une régression linéaire. Ici nous avons utilisé la fonction d'activation Relu, populaire dans les CNN.

La couche de convolution va nous permettre de réduire le nombre de paramètres grâce à la connectivité locale. Contrairement à une couche dense, tous les neurones ne sont pas entièrement connectés et seront connectés à un sous-ensemble de neurones locaux dans la couche suivante.

Ensuite nous avons les couches de pooling qui acceptent les couches convolutives comme entrée et vont à nouveau réduire considérablement le nombre de paramètres. Le fonctionnement d'une couche de pooling est assez similaire au filtre de la convolution. Une matrice généralement de 2 par 2 va parcourir le tensor issu de la couche de convolution et à chaque déplacement va recueillir la valeur de pixel la plus grande.

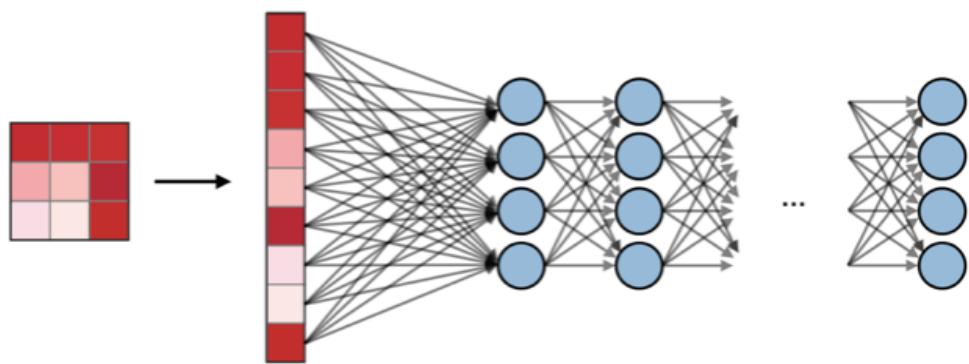


Nous utiliserons une autre technique courante qui s'appelle le dropout. Relativement simple et très efficace pour éviter le sur-apprentissage ou plutôt l'overfitting. Le principe du dropout est de désactiver aléatoirement un pourcentage de neurones. A chaque epoch, la combinaison de neurones sur lequel le modèle s'entraîne sera différente. A savoir, une epoch est une itération d'apprentissage sur le jeu de données complet.

Notre CNN devra ensuite procéder à un flatten. Le but étant de transformer le résultat du dernier pooling en vecteur à une dimension.

Ce nouveau vecteur pourra ensuite être passé aux couches denses de notre réseau de neurones.

Nos couches denses où chaque neurone est connecté avec tous les neurones de la couche suivante. Ceci va accroître considérablement le nombre de paramètres. Là aussi nous appliquons la fonction d'activation Relu.



La dernière couche dense communément appelée output layer aura comme fonction d'activation Softmax car nous avons 3 targets différents. L'output sera dans notre cas un vecteur à une dimensions au format [ x, y, z ]. Chaque index correspond à un target. La valeur à chaque index correspond à une probabilité. Softmax va arrondir les probabilités pour obtenir 3 cas possibles :

- [ 1, 0, 0 ] pour un patient sain

- [ 0, 1, 0 ] pour un patient souffrant de pneumonie virale
- [ 0, 0, 1 ] pour un patient souffrant de pneumonie bactérienne

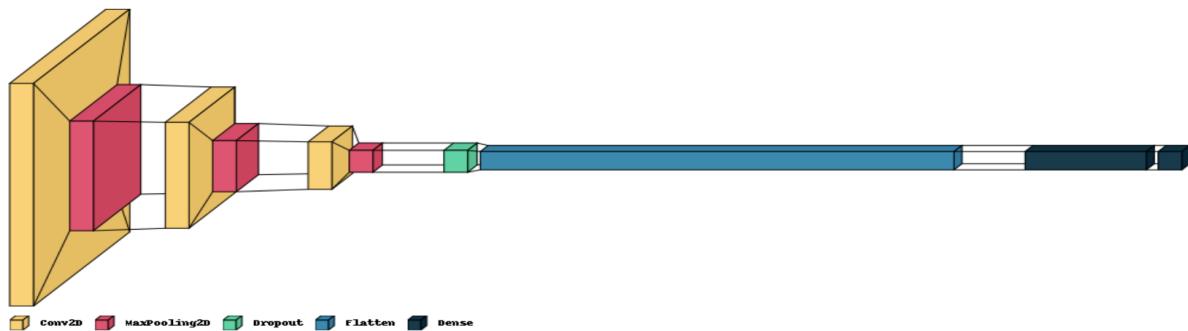
Enfin un optimiseur sera nécessaire pour finaliser notre CNN.

L'optimiseur a pour but d'optimiser le modèle en modifiant ses poids le plus efficacement possible. La fonction de perte avec une descente de gradient va permettre au modèle de progressivement trouver les poids les plus adaptés. Nous avons obtenu de bons résultats avec deux optimisateurs. Adam qui est réputé pour sa rapidité d'entraînement et son efficacité. Puis Nadam, une variante de Adam, qui bénéficie d'un accélérateur de gradient au nom de Nesterov momentum.

### Architecture du modèle

L'architecture que nous utiliserons :

- Nous avons en entrée des images de 64x64x3
- Une première couche de convolution à 32 filtres, et un kernel de 4x4
- Une première couche de pooling avec un kernel de 2x2
- Une seconde couche de convolution à 64 filtres, et un kernel de 2x2
- Une seconde couche de pooling avec un kernel de 2x2
- Une troisième couche de convolution à 128 filtres, et un kernel de 2x2
- Une troisième couche de pooling avec un kernel de 2x2
- Un dropout de 50%
- Un flatten
- Une couche dense de 1024 neurones
- Une couche d'output de 3 neurones



Cette architecture à la fois ni trop simple ni trop complexe nous a permis de maintenir un nombre de paramètres raisonnable, comme le montre l'image ci-dessous :

Layer (type)	Output Shape	Param #
conv2d_22 (Conv2D)	(None, 61, 61, 32)	1568
max_pooling2d_22 (MaxPooling)	(None, 30, 30, 32)	0
spacing_dummy_layer_30 (Spac)	(None, 30, 30, 32)	0
conv2d_23 (Conv2D)	(None, 29, 29, 64)	8256
max_pooling2d_23 (MaxPooling)	(None, 14, 14, 64)	0
spacing_dummy_layer_31 (Spac)	(None, 14, 14, 64)	0
lastConv (Conv2D)	(None, 13, 13, 128)	32896
1 (MaxPooling2D)	(None, 6, 6, 128)	0
spacing_dummy_layer_32 (Spac)	(None, 6, 6, 128)	0
2 (Dropout)	(None, 6, 6, 128)	0
3 (Flatten)	(None, 4608)	0
spacing_dummy_layer_33 (Spac)	(None, 4608)	0
4 (Dense)	(None, 1024)	4719616
7 (Dense)	(None, 3)	3075

Total params: 4,765,411  
Trainable params: 4,765,411  
Non-trainable params: 0

## Choix techniques

### Environnement

Nous utiliserons Jupyter Notebook pour écrire notre code, l'exécuter, afficher les résultats et exporter son contenu. Cet environnement de programmation basé sur le web a une prise en main facile et intuitive. Le projet tient sur un seul fichier et est facilement exportable.

### Librairies

Côté librairies IA nous utilisons principalement Tensor Flow. Soutenu par Google, cette librairie aux ressources débordantes est incontestablement la plus utilisée et se prête parfaitement à notre exercice.

Côté graphs : Seaborn plutôt que matplotlib pour la qualité visuelle des graphiques. Nous avons également implémenter visualkeras, une bibliothèque permettant de représenter fidèlement le modèle entraîné directement depuis le programme.

### Preprocessing

La dimension de 64x64x3 pour l'ensemble de nos images. Semble être le seuil pour obtenir des résultats corrects malgré la perte de détail dans les images. Ceci nous a permis d'adapter le temps d'apprentissage à la puissance de nos machines. D'après l'étude <https://pubs.rsna.org/doi/full/10.1148/ryai.2019190015>, la taille d'images pour obtenir un ratio apprentissage/temps d'apprentissage serait de 256x256.

Le train test split, consiste en la proportion d'images destinées au dataset TRAIN et au dataset TEST. Le ratio se situe généralement entre 75/25 et 60/40 pour ce type d'exercice. Nous choisirons  $\frac{2}{3}$  pour TRAIN et  $\frac{1}{3}$  pour TEST.

La normalisation de nos pixels sur une échelle entre 0 et 1 afin d'alléger les calculs et de gagner en temps d'entraînement.

Équilibrer les quantités d'images de chaque target et donc avoir autant de NORM que de VIR que de BACT nous a permis d'optimiser l'apprentissage en évitant que le modèle ne s'entraîne trop sur une target ou pas assez sur une autre.

Améliorer les contrastes de nos images à l'aide de traitements. Nous utiliserons l'algorithme CLAHE ( contrast limited adaptive histogram equalization) pour amplifier les contrastes dans nos images afin de faire ressortir des poumons les zones malades. Nous constaterons un gain en accuracy à la suite de ce traitement appliqué à notre dataset.

Le one-hot encoding lorsque nous avons libellé les images pour garder un cohérence avec l'output softmax de notre modèle.

### Model

Nous avons choisi de faire une architecture ni trop simple ni trop complexe pour rester en cohérence avec la taille des images que nous voulons lui donner, le temps d'apprentissage et les résultats souhaités. La première couche de convolution avec un kernel de grande taille nous permet de recueillir les gros features de l'image, puis les convolutions suivantes de récupérer les détails plus précis.

Les couches de maxpooling intercalées entre chaque convolution vont permettre de réduire la taille des images en gardant les pixels aux valeurs les plus élevées afin de faire ressortir les éléments importants pour la prochaine convolution.

Le dropout sera un atout pour lutter contre l'overfitting. Dans notre cas, il va aléatoirement désactiver 50% des neurones au passage de chaque batch. Ceci nous aidera à poursuivre un entraînement long sur 100 epochs.

La fonction d'activation RelU est appliquée à toutes nos couches à l'exception de la couche de sortie. Relu est la fonction d'activation non linéaire par excellence et la plus utilisée en CNN.

La fonction d'activation softmax quant à elle va nous permettre de convertir à 1 la probabilité la plus élevée parmi nos trois targets et les deux autres à 0. Indispensable pour classifier des images à plus de 2 targets. Le format devient identique à celui des labels de nos images encodées avec one-hot.

Le learning rate(LR) à 0.0002 est le fruit de fine-tuning et a été ajusté au cours de tests successifs. Un learning rate trop élevé implique du bruit et un overfitting alors qu'un LR trop bas aurait pour conséquence un underfitting.

La métrique AUC en plus de l'accuracy, va nous permettre de juger de la performance de notre modèle et sera accompagnée de la courbe ROC qui donne le taux de vrais positifs en fonction du taux de faux positifs.

### Callbacks

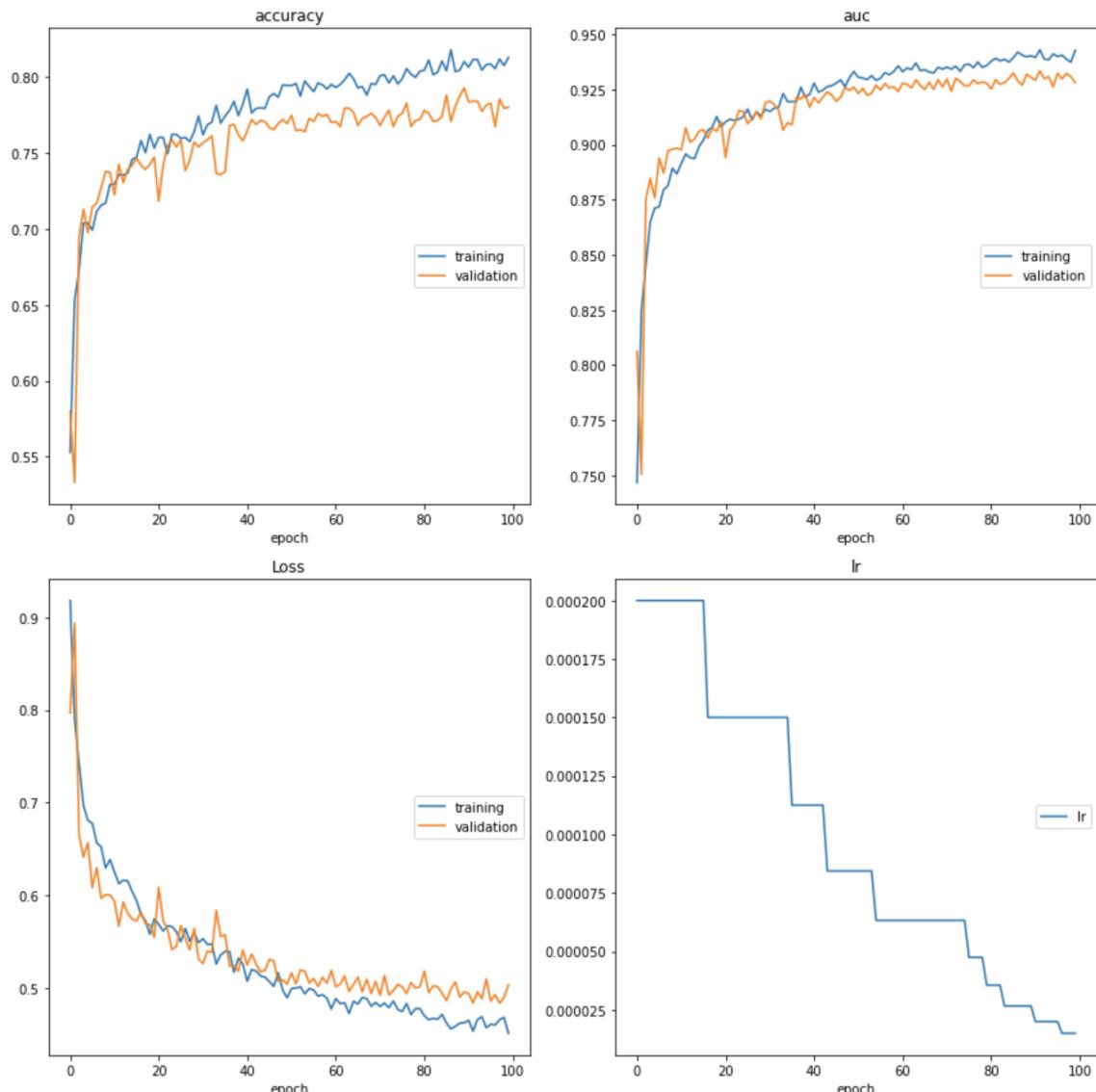
ModelCheckpoint va nous permettre de sauvegarder dans un fichier les poids à l'issue de notre entraînement. L'avantage est de pouvoir s'en servir à volonté et de pouvoir utiliser d'autres sauvegardes sans avoir à entraîner à nouveau le modèle.

Earlystopping va permettre au modèle d'arrêter son entraînement s'il commence à faire de l'overfitting et donc éviter d'endommager son entraînement. La patience de 15 signifie qu'il mettra fin à l'entraînement au bout de 15 epochs consécutifs overfittés. Ce chiffre est cohérent avec le nombre d'epochs que nous voulons entraîner.

ReduceLROnPlateau va permettre d'ajuster automatiquement le learning rate dès lors que le modèle commence à overfitter. De cette façon, nous pourrons maîtriser et retarder le phénomène d'overfitting. Dans notre cas, le LR diminue de 75% à chaque fois que le modèle overfit durant 4 epochs consécutives.

PlotLossKeras va nous permettre de visualiser en temps réel l'avancée de notre apprentissage. Nous aurons 4 graphiques qui se mettront à jour à la fin de chaque epoch. On y verra la courbe ROC, l'accuracy, loss, et LR. On pourra suivre l'entraînement et l'arrêter en cours si besoin afin d'y ajuster des hyperparamètres.

illustration de PlotLossKeras appliqué à notre modèle:



## Params

Le Batch size correspond au nombre d'images qui seront utilisées pour parcourir le modèle au cours de chaque réajustement des poids. Nous utiliserons un batch size de 8 pour gagner en précision et réduire le bruit lors de l'entraînement au détriment de sa durée.

Les Epochs correspondent au passage complet de toutes les batchs dans le modèle. Ici nous avons fixé 100 epochs pour entraîner notre modèle. Soit suffisamment pour que la courbe d'accuracy, d'auc, et de loss s'aplanissent suffisamment pour en déduire que le modèle a appris correctement.

## Data augmentation

Le dataset ne dépassant pas les 6000 images nous semblait trop limité pour un bon apprentissage. Nous avons choisi d'augmenter artificiellement le nombre d'images de notre dataset pour à la fois gagner en accuracy et lutter contre l'overfitting.

## K-fold cross validation

Ce principe de validation croisé va nous permettre de faire trois entraînements à la suite. Chaque entraînement utilisant un dataset TRAIN et TEST différent grâce aux blocs modulables que nous avons créés au préalable. illustration ci-dessous :

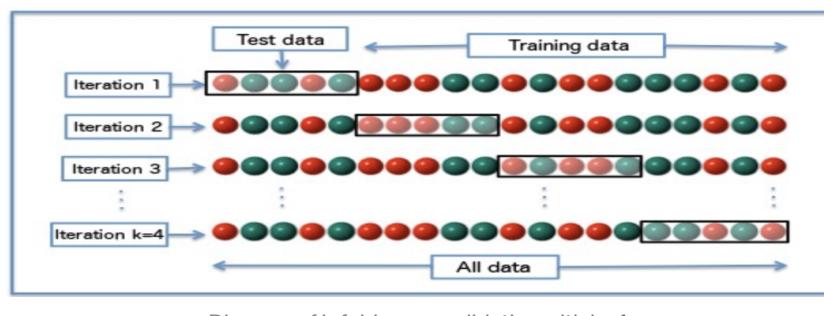
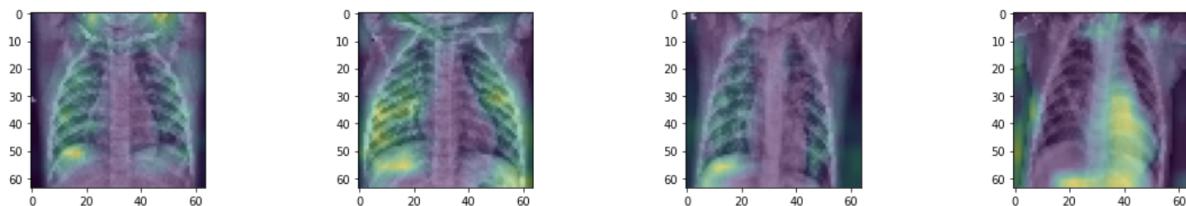


Diagram of k-fold cross-validation with  $k=4$ .

Cette méthode nous permet de vérifier la cohérence et la généralisation des résultats obtenus lors des trois entraînements. De fortes disparités de résultats mettrait en évidence une anomalie dans le modèle qui aurait alors passé inaperçu si nous n'avions pas fait de validation croisée.

## Grad cam

Le grad cam nous permet de s'assurer que le modèle s'entraîne sur les bonnes parties de l'image. Il ajoute une carte thermique sur l'image et donne un côté très visuel de l'apprentissage du modèle sur chaque image.



## Boosted Tree

### Fonctionnement

Les Boosted Trees peuvent résoudre des problèmes de régression ou de classification. Dans notre cas, nous devons classer des images dans trois catégories : sain, virus, bactérie. L'explication qui va suivre s'attardera donc sur le fonctionnement des Boosted Trees pour une classification. Certains calculs différents pour un problème de régression.

Pour aborder le fonctionnement de cet algorithme nous allons utiliser un exemple volontairement simple avec une classification : sain ou malade.

Feature 1	Feature 2	Feature 3	Feature 4	Diagnostic
23	145	234	56	Healthy
123	5	32	164	Sick
156	134	240	123	Healthy
92	6	10	154	Sick
45	2	86	149	Sick

L'algorithme du Boosted Tree commence par construire une **feuille unique**. Cette feuille représente la prédiction initiale pour toutes les entrées de notre dataset. Cette valeur est déterminée à partir du **log(odds)**. Nous ne détaillerons pas ici le calcul de cette expression.

Dans le cas de notre jeu d'entraînement, nous avons **3** personnes malades et **2** patients sains. Ainsi, la prédiction qu'une personne soit malade dans cette échantillon équivaut à :

$$\log\left(\frac{3}{2}\right) \approx 0.40$$

Ce résultat constitue notre **prédiction initiale** et devient donc la valeur de notre première feuille. La meilleure manière d'utiliser cette prédiction est de la convertir en **probabilité**. Pour ce faire, nous utilisons la fonction logistique.

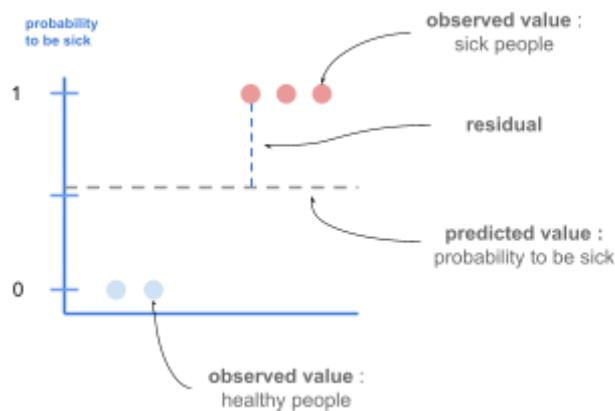
$$p = \frac{e^{\log(\text{odds})}}{1 + e^{\log(\text{odds})}}$$

Cette formule nous donne une probabilité d'être malade proche de 0.52. Dans la mesure où la probabilité d'être malade est supérieure à 0.50, nous décidons arbitrairement de déterminer que toutes les personnes de cet exemple sont malades. Cependant, cette première observation est plutôt mauvaise puisque deux des personnes sont en bonne santé.

Nous pouvons mesurer la qualité de la prédiction initiale en calculant les **pseudo residuals**, c'est-à-dire la différence entre les valeurs observées et les valeurs prédictes.

$$\text{residual} = \text{observed} - \text{predicted}$$

Nous pouvons représenter graphiquement le **residual**.

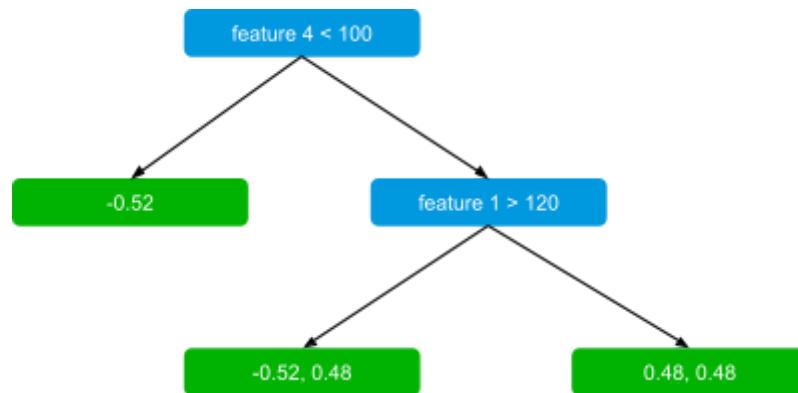


Par exemple, pour la première personne malade, le residual équivaut à  $(1 - 0.52) = 0.48$ . De la même manière, on peut maintenant calculer le residual pour toutes les autres entrées de la dataset.

Feature 1	Feature 2	Feature 3	Feature 4	Diagnostic	Residual
23	145	234	56	Healthy	-0.52
123	5	32	164	Sick	0.48
156	134	240	123	Healthy	-0.52
92	6	10	154	Sick	0.48
45	2	86	149	Sick	0.48

Avec ces résiduels, nous pouvons maintenant calculer de nouvelles probabilités pour chaque élément de notre dataset. Pour ce faire, nous allons construire un arbre, avec plusieurs feuilles cette fois-ci, dans le but de prédire les prochains résiduels. En d'autres termes, chaque feuille contiendra une prédiction qui sera la valeur d'un residual.

Notez que cet arbre reste très simple dans le cas de notre exemple, nous limiterons le nombre de feuilles à 3. En pratique, on fixe souvent une profondeur d'arbres entre 8 et 32. Cette valeur peut être déterminée à l'aide d'un **hyperparamètre**.

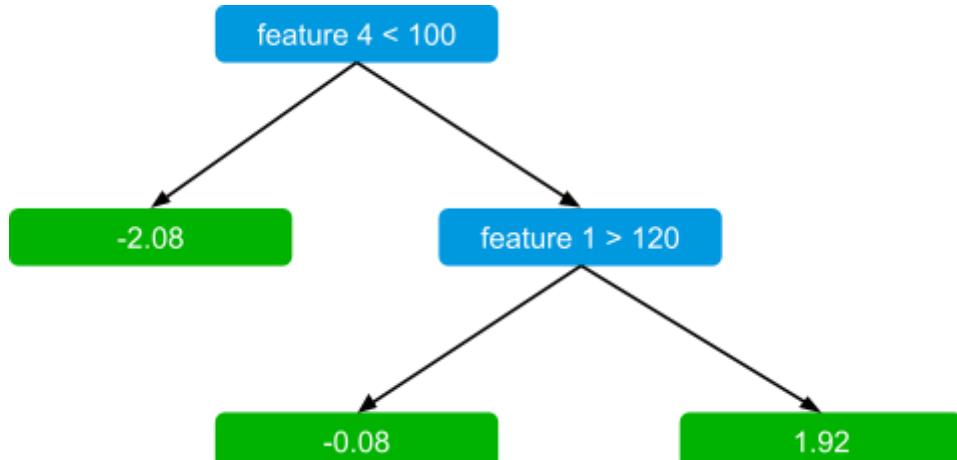


Quand nous utilisons l'algorithme des Boosted Trees pour un problème de classification, et contrairement à la régression, nous ne pouvons pas utiliser directement les valeurs des feuilles, nous devons convertir ces **résiduels** en **prédition**, autrement dit en **log(odd)**. Pour ce faire, nous utilisons la formule suivante sur chacune des feuilles.

$$prediction = \frac{\sum Residual}{\sum [Previous Probability_i \times (1 - Previous Probability_i)]}$$

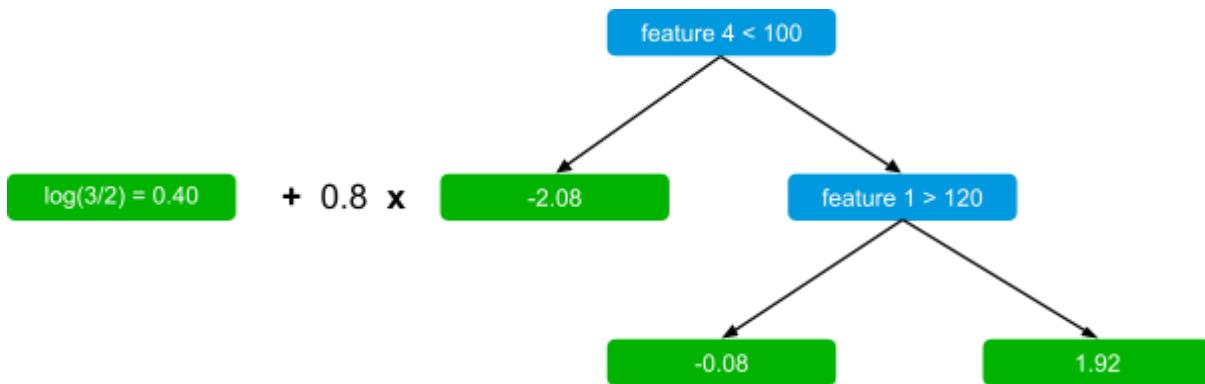
Nous obtenons alors une prédition unique pour chacune des feuilles de notre arbre. Pour rappel, comme il s'agit du premier arbre, la prédition précédente est commune à tout le dataset et vaut 0,48. Si l'on prends comme exemple la feuille en bas à droite, nous obtenons le calcul :

$$\frac{-0.52 + 0.48}{0.48 * (1 - 0.48) + 0.48 * (1 - 0.48)} = -0.08\dots$$



Avec ces nouvelles valeurs, nous pouvons maintenant déterminer de nouvelles prédictions pour chaque entrée de notre dataset. Pour ce faire, on additionne la prédition précédente et la nouvelle prédition. Pour limiter l'**overfitting**, Boosted Tree utilise un **learning rate** pour échelonner la prédition du nouvel arbre. Sans ce **learning rate**, nous risquerions d'avoir un biais faible, mais une variance très élevée.

Dans notre exemple, le learning rate est volontairement important. Cependant, en réalité, le learning rate sera plus proche de 0.1.



Maintenant, il suffit de déterminer la feuille de l'arbre qui correspond à chaque élément de notre ensemble de données.

Ensuite, pour obtenir une nouvelle prédition logarithmique pour chaque patient, nous effectuons le calcul suivant :

$$\log(\text{odds}) = \text{initial prediction} + (\text{learning rate} \times \text{leaf value})$$

Enfin, nous convertissons cette prédition en une probabilité avec la même formule que précédemment :

$$p = \frac{e^{\log(\text{odds})}}{1 + e^{\log(\text{odds})}}$$

Feature 1	Feature 2	Feature 3	Feature 4	Diagnostic	Old prob.	New prob.
23	145	234	56	Healthy	0.52	0.22
123	5	32	164	Sick	0.52	0.58
156	134	240	123	Healthy	0.52	0.58
92	6	10	154	Sick	0.52	0.87
45	2	86	149	Sick	0.52	0.87

Auparavant, nous n'avions qu'une seule prédition pour l'ensemble des données. Avec cette première itération, nous constatons que les probabilités sont meilleures qu'auparavant. Pour rappel, une probabilité proche de 1 prédit un patient malade, tandis qu'une probabilité proche de 0 prédit un patient sain.

Nous remarquons que la probabilité du premier patient a diminué et est plus proche de 0, ce qui suggère qu'il s'agit d'un patient sain. **Bonne prédition!**

Nous remarquons également que les probabilités du quatrième et du dernier patient ont fortement augmenté et sont proches de 0,9, ce qui suggère que ces patients sont malades. **Bonne prédition encore une fois!**

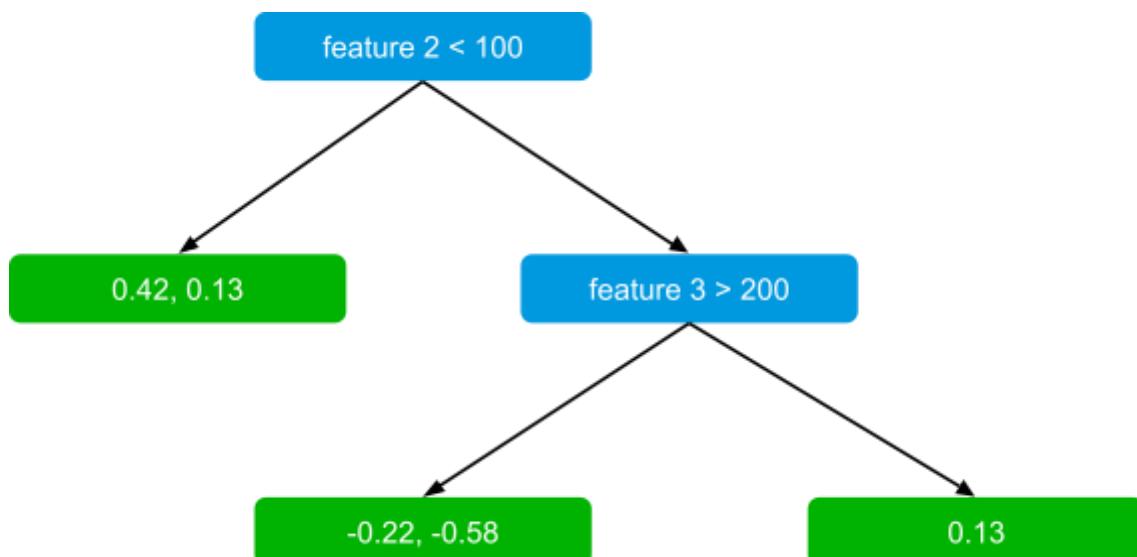
Seules les probabilités du deuxième et du troisième patient n'ont pas beaucoup changé et restent proches de 0,5. Il semble encore difficile de faire une prédition à leur sujet. Pour affiner cette recherche, l'algorithme va créer un nouvel arbre.

Avant de créer un nouvel arbre, nous devons recalculer les residuals pour chaque patient à partir de leur nouvelle probabilité. Pour ce faire, nous utilisons la même formule que précédemment.

$$\text{residual} = \text{observed} - \text{predicted}$$

Feature 1	Feature 2	Feature 3	Feature 4	Diagnostic	New prob.	Residual
23	145	234	56	Healthy	0.22	-0.22
123	5	32	164	Sick	0.58	0.42
156	134	240	123	Healthy	0.58	-0.58
92	6	10	154	Sick	0.87	0.13
45	154	86	149	Sick	0.87	0.13

Nous pouvons maintenant créer un nouvel arbre sur le même principe que précédemment. L'algorithme du Boost Tree va définir aléatoirement de nouvelles features à observer pour cet arbre en particulier.



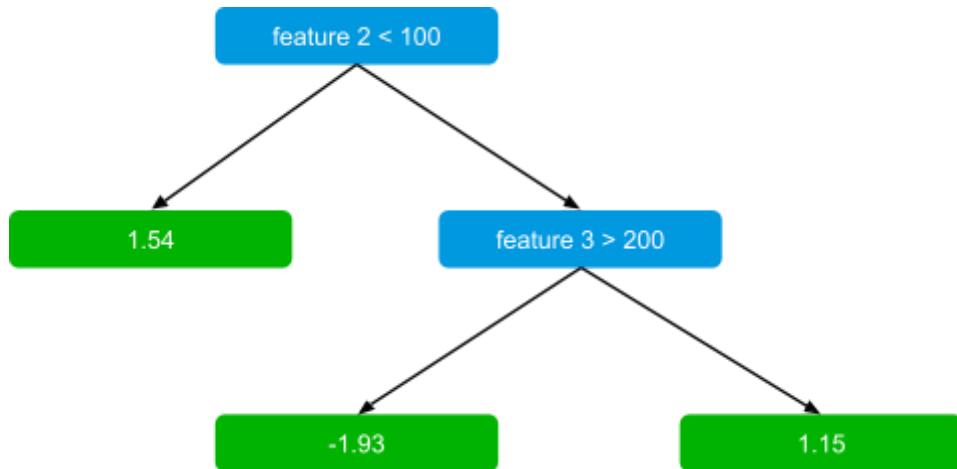
Pour ce nouvel arbre, nous avons classé le résiduel de chaque individu dans une feuille. Pour obtenir une nouvelle prédition logarithmique, nous devons appliquer la transformation précédente et nous obtenons ce nouvel arbre de prédition.

$$prediction = \frac{\sum Residual}{\sum [Previous Probability_i \times (1 - Previous Probability_i)]}$$

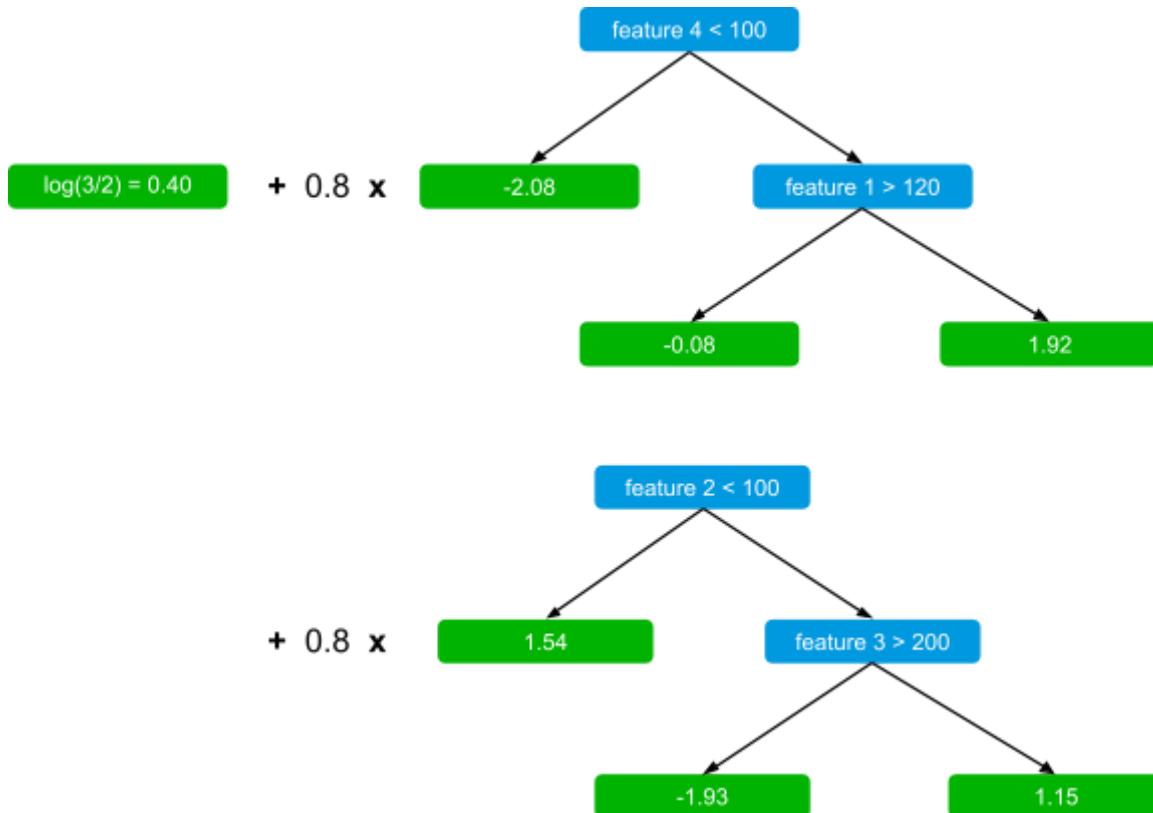
Si on reprends l'exemple de la feuille en bas à droite, qui possède les premier et troisième patients, nous avons donc le calcul suivant :

$$\frac{-0.22 - 0.58}{0.22 * (1 - 0.22) + 0.58 * (1 - 0.58)} = -1.9267...$$

Lorsque l'on applique ceci à toutes les feuilles, cela nous donne le schéma suivant :



Maintenant, nous pouvons combiner tous les arbres pour obtenir de nouvelles probabilités pour tous les patients de notre jeu de données et classer les personnes malades et les personnes saines.



Comme précédemment, nous prenons chaque patient et, en fonction des conditions, nous le classons dans la bonne feuille de chaque arbre. Ensuite, nous pouvons déterminer une nouvelle prédition pour chaque entrée du jeu de données.

Par exemple, la nouvelle prédition pour le premier patient serait :

$$prediction_1 = 0.40 + (0.8 \times -2.08) + (0.8 \times -1.93) = -2.81$$

Comme nous devons convertir ce log(odds) en une probabilité, nous utilisons à nouveau la fonction logistique.

$$p = \frac{e^{\log(\text{odds})}}{1 + e^{\log(\text{odds})}} = \frac{e^{-2.81}}{1 + e^{-2.81}} = 0.06$$

Nous répétons ce processus pour toutes les entrées du jeu de données et nous obtenons de nouvelles probabilités pour chacun d'entre eux.

Feature 1	Feature 2	Feature 3	Feature 4	Diagnostic	Old prob.	New prob.
23	145	234	56	Healthy	0.22	0.06
123	5	32	164	Sick	0.58	0.77
156	134	240	123	Healthy	0.58	0.22
92	6	10	154	Sick	0.87	0.95
45	2	86	149	Sick	0.87	0.95

Avec cette nouvelle itération, on se rend compte que les probabilités se rapprochent du résultat attendu. Chaque nouvel arbre affinera un peu plus les probabilités. Ce processus se répète jusqu'à ce que nous ayons atteint le nombre maximum d'arbres spécifiés, ou que les residual deviennent insignifiants.

Habituellement, l'algorithme des Boosted Trees utilise des arbres d'une profondeur de 8 à 32 niveaux. Dans notre cas, les probabilités s'améliorent rapidement puisque le jeu de données est très petit, les caractéristiques sélectionnées pour les arbres sont pertinentes et surtout notre learning rate est énorme.

### Architecture du modèle

Les Boosted Trees, basés sur une séquence d'arbre décisionnel, demeurent une structure relativement simple.

Dans notre cas, nous utiliserons :

- Une séquence composée de 100 estimateurs, soit 100 arbres.
- Des arbres d'une profondeur maximale de 4.

## Choix techniques

### Environnement

Comme pour le modèle précédent, nous utiliserons Jupyter Notebook pour écrire notre code, l'exécuter, afficher les résultats et exporter son contenu. Bien évidemment, il sera nécessaire d'installer toutes les dépendances Python nécessaires. Pour ce faire, nous avons utilisé Conda qui permet de créer des environnements Python spécifiques à un projet. Ainsi, l'environnement Python du système n'est pas surchargé par de nombreuses librairies inutiles d'un projet à l'autre.

### Librairies

Pour cette algorithme de Boosted Trees, nous avons d'abord essayé de l'implémenter avec Tensor Flow. L'idée était de pouvoir réutiliser nos connaissances assimilées pendant la création du CNN. Cependant, la manipulation et les concepts des Boosted Trees sont complètement différents des réseaux de neurones. Malgré tout, nous avons essayé d'implémenter un premier algorithme avec Tensor Flow mais les résultats n'étaient pas satisfaisants.

Après plusieurs recherches, nous nous sommes logiquement dirigés vers la librairie XGBoost qui est l'une des plus utilisées pour modéliser des algorithmes Boosted Trees. Elle est facile à utiliser et les ressources associées sont nombreuses.

Côté graphiques, nous allons utiliser Seaborn comme précédemment.

### Preprocessing

Nous avons décidé d'aborder une approche complètement différente des réseaux de neurones pour mettre en exergue les résultats obtenus précédemment. Dans le but de conserver des résultats cohérents, le travail de preprocessing effectué pour cet algorithme est quasiment identique au modèle précédent. Le but étant de travailler sur un jeu de données similaire.

La dimension des images demeure 64 pixels x 64 pixels. Chaque pixel de l'image étant modélisé en RGB soit une matrice de 3. Pour les besoins du traitement par XGBoost, ces images seront "étalées" - flatten. Ainsi, chaque image sera représentée par une matrice de dimension 1 par 12 288, le résultat de 64 x 64 x 3 où chaque valeur varie entre 0 et 255.

Pour faciliter et accélérer le calcul, nous normalisons les données des matrices pour n'avoir plus que des valeurs entre 0 et 1.

Comme précédemment, nous avons manuellement lisser les quantités d'images de nos 3 classes. Ainsi, le nombre d'images est identique que ce soit les labels NORM, VIR et BACT. Cette manipulation évite un entraînement excessif pour une classe en particulier.

L'algorithme intègre également CLAHE ( Contrast Limited Adaptive Histogram Equalization) pour amplifier les contrastes des images afin de faire ressortir des poumons les zones malades. Cette manipulation permet des gains de performances.

Contrairement aux réseaux de neurones, nous n'utilisons pas de One Hot Encoding. XGBoost permet difficilement son utilisation et dans le cas des arbres décisionnels le gain n'est pas significatif. Nos classes seront simplement représentées par des entiers : 0 pour une patient sain, 1 pour le virus et 2 pour la bactérie.

## Model

Comme présenté ci-dessus, notre modèle est basé sur un algorithme de Boosted Tree. Le but était d'explorer une nouvelle perspective que les réseaux neuronaux. Les arbres décisionnels sont une autre famille d'algorithmes majoritairement utilisée en machine learning. Dans cette perspective, nous avions donc le choix d'implémenter une Random Forest ou des Boosted Trees. Notre choix s'est porté sur les Boosted Trees au vu des performances élogieuses à son égard. La perspective d'utiliser XGBoost a confirmé ce choix.

XGBoost rend très accessible l'implémentation d'un modèle Boosted Trees. Le modèle peut être créé en deux lignes : une ligne pour définir le problème abordé, une régression ou une classification et une ligne pour entraîner le modèle. Dans notre cas, nous utilisons la méthode XGBClassifier pour résoudre notre problème de classification. Toute la difficulté et l'efficacité de notre modèle vont résider dans l'ajustement des hyper-paramètres. Ces valeurs vont notamment déterminer le nombre d'arbres et leur profondeur maximale, qui sont deux paramètres majeurs dans ce type d'algorithme.

## L'ajustement des hyper-paramètres

En plus de la préparation en amont du jeu de données, l'efficacité de notre algorithme réside également dans le bon ajustement des hyper-paramètres. Il est donc important d'identifier les valeurs les plus performantes.

Trois paramètres vont être essentiels dans notre algorithme : le **max\_depth** qui correspond à la profondeur maximale de nos arbres, le **number\_estimator** qui définit le nombre d'arbres et enfin le **learning\_rate** qui est un coefficient appliqué entre chaque étape pour prévenir de l'overfitting. XGBoost permet de régler de nombreux autres paramètres, plus secondaires, dont la plupart sont déjà réglés correctement par défaut.

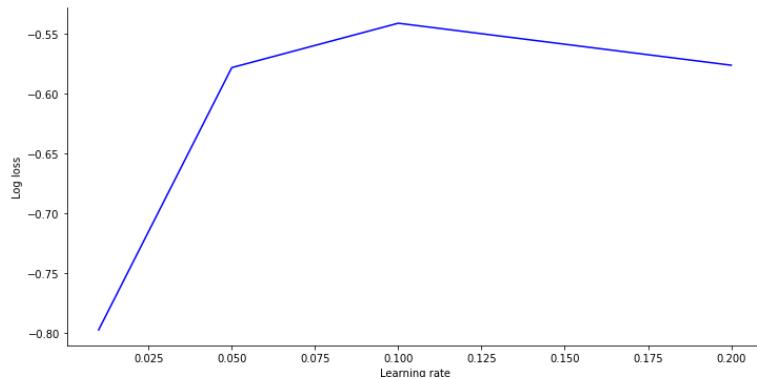
Il existe trois grandes méthodes pour régler des hyper-paramètres correctement. Ces méthodes s'appliquent à XGBoost mais à toutes autres librairies de Machine Learning.

- **Grid Search.** Cette méthode consiste à sélectionner une série de valeurs et à les comparer selon des indicateurs pour en tirer la plus efficace.
- **Random Search.** Cette approche est similaire à la précédente, mais il s'agit maintenant de piocher des valeurs au hasard dans un intervalle donné.
- **Bayesian optimization algorithms.** Ces algorithmes vont deviner un ensemble d'hyper-paramètres en fonction des résultats des tests précédents. L'algorithme SMAC permet ce genre d'optimisation.

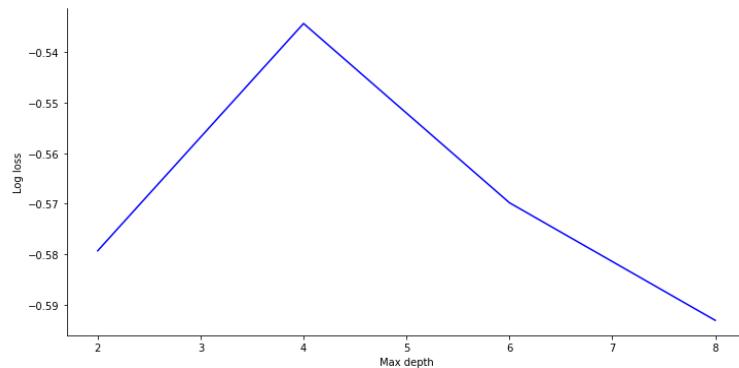
Dans notre cas, nous utilisons la première méthode de la **Grid Search avec la librairie GridSearchCV**. Pour des raisons matérielles, nous ne pouvons que tester un nombre restreint de valeurs. Notre optimisation consiste à définir 4 valeurs pour chaque hyper-paramètre et à observer leur résultat. Les résultats sont déterminés par la valeur moyenne du log-loss. Le log-loss indique la pertinence d'une prédiction par rapport à la valeur attendue. Plus l'indicateur est proche de 0, plus la prédiction est proche de la réalité.

Ces valeurs ne sont pas choisies arbitrairement, elles s'appuient sur les travaux réalisés par [Machine Learning Mastery](#).

Pour commencer, nous allons étudier 4 valeurs potentielles pour le learning rate : 0.01, 0.05, 0.1, 0.2. Les résultats suivants nous montrent que le learning rate est optimal autour de 0.1 puisque le log-loss est le plus proche de 0. On a tendance à penser que plus le learning rate sera faible, plus l'algorithme sera précis. Or, nous constatons que pour un learning rate de 0.01, les résultats sont moins bons que pour une valeur plus grande, 0.2 par exemple. Au vu de ces résultats, nous choisirons une valeur de 0.1 pour notre learning rate.

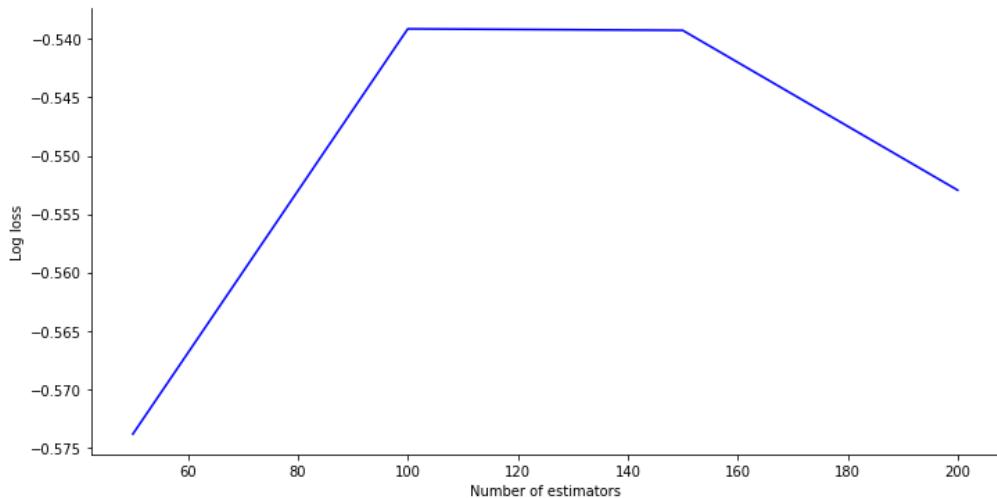


Nous poursuivons sur l'optimisation de l'hyper-paramètre max\_depth. Ce paramètre définit la profondeur maximale de nos arbres. Nous avons pensé, à tort, que plus les arbres étaient profonds, plus l'algorithme était précis. Ajouter de la profondeur à nos arbres ne fait qu'augmenter le risque d'overfitting. En effet, l'algorithme finit par se concentrer sur des observations sans importance. La valeur habituellement recommandée pour ce paramètre varie suivant le problème rencontré. Dans notre cas, la valeur optimale semble être 4. En effet, le log-loss est à son apogée et la valeur décroît drastiquement lorsque la profondeur est augmentée. Au vu de ces observations, nous choisirons une valeur de 4 pour notre max\_depth.



Enfin, le dernier principal paramètre à définir est le `n_estimators`, qui correspond au nombre d'arbres qui seront utilisés par notre algorithme. Comme pour le `max_depth` précédent, nous pensions qu'il était préférable de favoriser la profondeur des arbres au nombre d'arbres. Nous verrons dans la section résultats que l'inverse est préférable. Pour l'optimisation de cette variable, nous sélectionnons 4 valeurs à nouveau: 50, 100, 150 et 200.

Nous constatons que le log-loss est important si le nombre d'arbres est trop faible. En revanche, augmenter le nombre d'arbres ne réduit pas davantage le log-loss voire produit l'effet inverse. Suite à ces résultats, nous conserverons un `n_estimators` de 100.



Cette analyse reste à nuancer. En effet, nous ne comparons que 4 situations à chaque fois pour sélectionner la meilleure. Il est fort probable qu'il existe une configuration plus performante dans un intervalle entre de 2 valeurs.

Nous pouvons maintenant construire un modèle optimisé de Boosted Tree et comparer les résultats avec le CNN.

## Résultats

### Réseaux de Neurones à Convolution (CNN)

A l'issue de notre triple entraînement grâce au k-fold cross-validation, nous obtenons les résultats suivants.

	Perte	Auc	Précision
Entraînement 1	0.48443037271499634	0.93672114610672	0.8071236610412598
Entraînement 2	0.5010800361633301	0.931718111038208	0.7786720395088196
Entraînement 3	0.5289214849472046	0.9286917448043823	0.7836021780967712

La précision correspond à la proportion de bonnes prédictions par rapport à toutes les prédictions. Nous obtenons 79% sur nos trois targets.

L'auc (aire sous la courbe ROC) montre que nous avons 93% de chances qu'un positif soit vrai. On peut interpréter l'AUC comme une mesure de la probabilité pour que le modèle classe un exemple positif aléatoire au-dessus d'un exemple négatif aléatoire.

Les résultats sont homogènes et ne présentent pas d'anomalies.

Regardons plus en détail les résultats du troisième entraînement avec le rapport de classification suivant :

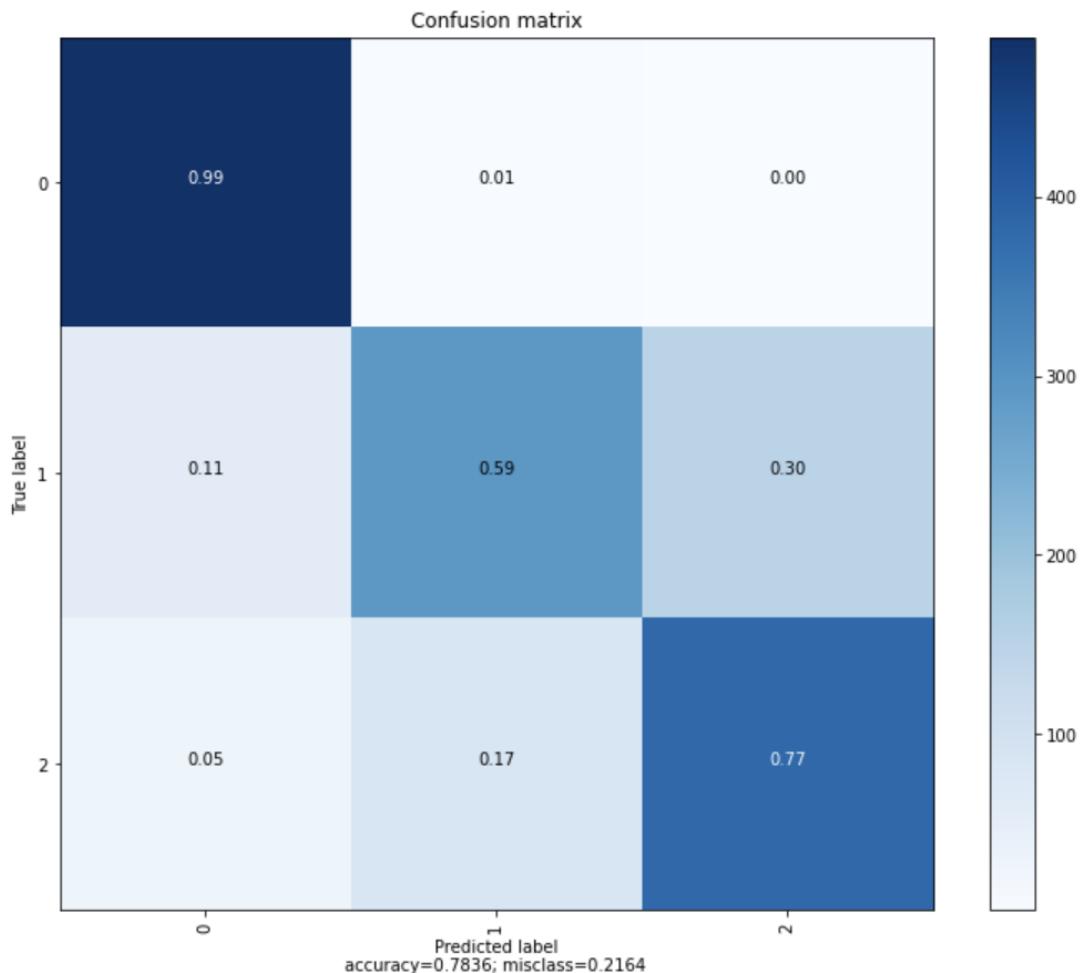
	precision	recall	f1-score	support
Bact	0.72	0.77	0.75	496
Norm	0.85	0.99	0.92	496
Vir	0.77	0.59	0.67	496
accuracy			0.78	1488
macro avg	0.78	0.78	0.78	1488
weighted avg	0.78	0.78	0.78	1488

La précision correspond au nombre de fois où la classe a été correctement attribuée par rapport au nombre total de prédictions appartenant à cette classe.

Le recall correspond au nombre de fois où la classe a été correctement attribuée par rapport au nombre total d'images appartenant à cette classe. C'est la métrique que nous utiliserons dans la matrice de confusion plus bas.

F1 combine précision et recall, et ne prend pas en compte les vrai négatifs. Ceci permet d'éviter qu'une classe qui serait moins représentée dans le dataset vienne fausser la précision du modèle. Ce qui n'est pas le cas dans notre dataset car nos classes sont parfaitement équilibrées.

Pour chercher à mieux comprendre où le modèle s'est trompé nous avons dressé une matrice de confusions comme ci-dessous :



Ici, les labels 0,1,2 correspondent respectivement à NORM,VIR,BACT.

A en juger de cette matrice, nous pouvons dire sans problème affirmer que le modèle prédit avec excellence un patient sain avec 99% de prédictions correctes.

Malheureusement, lorsqu'il s'agit d'un patient malade, le modèle rencontre un peu moins de réussites. 77% de réussite à prédire une pneumonie bactérienne, et seulement 59% lorsqu'il s'agit de la variante virale.

Cependant, on peut noter que, malgré que le taux de réussite ne soit pas au même niveau que pour les cas sains, le modèle a plutôt tendance à continuer à prédire que le résultat reste malade et non pas annoncer un faux négatif (11% de faux négatif en cas de pneumonie virale, et seulement 5% pour la version bactérienne).

## Boosted Trees

Les résultats présentés ci-dessous mettent en parallèle notre premier modèle de Boosted Trees issu de nos hypothèses de départ et le second modèle qui fait suite à notre recherche d'optimisation des hyper-paramètres.

	Précision	Recall	Accuracy
Modèle sans optimisation des hyper-paramètres	0.71	0.70	0.70
Modèle avec optimisation des hyper-paramètres	0.78	0.75	0.75

Nous constatons que l'accuracy a augmenté dans le second modèle. Cette observation met en avant l'importance de bien configurer les hyper-paramètres. Le premier modèle se basait sur une stratégie d'arbres profonds mais peu nombreux tandis que le second se compose d'arbres plus petits mais plus nombreux. Dans l'ensemble de nos recherches, nous avons constaté que cette seconde approche était largement plébiscitée, dû au fait qu'un arbre trop profond entraînera rapidement de l'overfitting.

Comme le second modèle est plus performant, nous concentrerons nos analyses sur ses résultats.

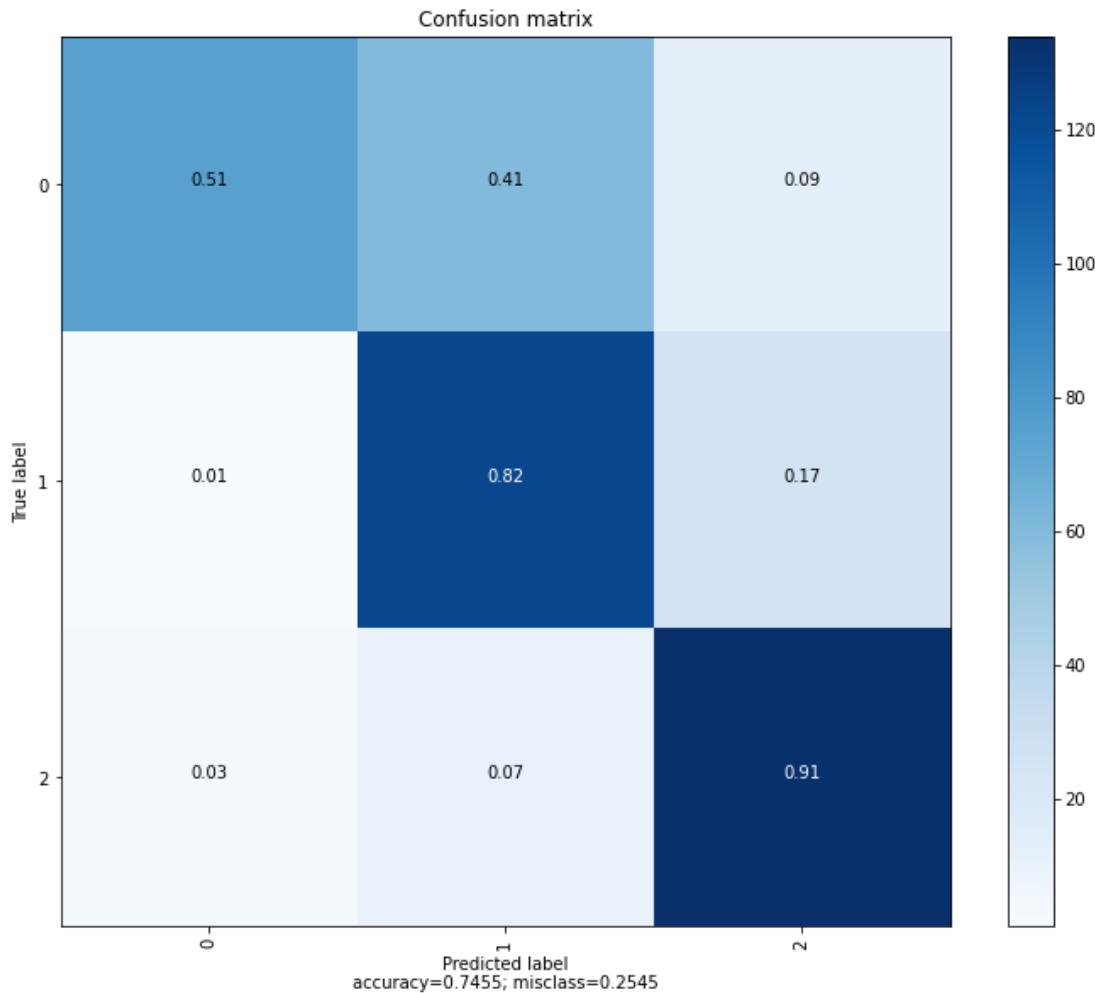
L'accuracy de 75% du modèle suggère que les trois quart de nos prédictions sont justes, toutes classes confondues. Bien qu'inférieure, cette performance se rapproche des résultats obtenus par le réseau de neurones. Cependant, en observant plus en détail la précision et le recall de chaque classe, nous constatons immédiatement le point faible de notre modèle. Pour rappel, la précision se concentre sur les faux positifs : ce sont les images qui ont été assignées dans une classe qui n'est pas la leur. Le recall se concentre sur les faux négatifs : ce sont les images qui n'ont pas été retenues mais qui appartiennent pourtant à la classe.

Dans notre cas d'étude, il est intéressant d'analyser le recall. En effet, il est important de se concentrer sur les malades qui n'ont pas été correctement diagnostiqués, les faux négatifs. Un recall proche de 1 suggère que peu de cas sont ignorés.

	precision	recall	f1-score	support
Bact	0.78	0.91	0.84	148
Norm	0.94	0.51	0.66	148
Vir	0.64	0.82	0.72	148
accuracy			0.75	444
macro avg	0.78	0.75	0.74	444
weighted avg	0.78	0.75	0.74	444

On peut constater que le recall pour la classe NORM est très faible. On peut en déduire que le modèle se trompe près d'une fois sur deux lorsqu'il s'agit d'identifier un patient sain d'un patient malade. En revanche, son analyse des classes BACT et VIR est très bonne. Il affiche un taux de réussite de 91% pour les pneumonie bactérienne et un taux de réussite de 82% pour

les pneumonie virale. La matrice de confusion va nous permettre d'identifier la répartition de ces prédictions.



Comme précédemment, les labels 0, 1, 2 correspondent respectivement aux classes NORM, VIR, BACT. Cette matrice de confusion met en lumière l'échec de l'algorithme pour différencier un patient sain d'un malade. Il faut considérer que l'algorithme aura tendance à désigner une personne saine comme malade. En revanche, la réciproque n'est pas vrai. En effet, l'algorithme identifie presque parfaitement un malade. Cependant, dans le cas d'une pneumonie virale, il aura parfois du mal à diagnostiquer la pathologie, virale ou bactérienne. L'identification des cas de pneumonie bactérienne est, quant à elle, excellente.

## Discussions

Nous pouvons conclure que notre modèle CNN remplit bien son rôle pour prédire nos trois targets. Il se démarque de la classification entre personnes saines et personnes malades où il excelle. En revanche, lorsqu'il s'agit de différencier une pneumonie virale d'une bactérienne, il est un peu moins performant et aura tendance à confondre les deux pathologies.

Nous pensons que ceci est dû à la faible dimension de nos images. Les images ne sont plus suffisamment détaillées, ce qui impacte le modèle lorsqu'il tente de trouver les features sur lesquelles il pourra alors s'entraîner à différencier les deux types de pneumonies.

D'un autre côté, le Boosted Tree affiche également des résultats très satisfaisants. En effet, on constate que le Boosted Tree est performant pour classer les pneumonies, notamment les bactériennes. En revanche, il a beaucoup de mal à faire la différence entre un patient sain et un malade, contrairement au CNN. Il était intéressant d'aborder un algorithme totalement différent : d'un côté les réseaux de neurones et de l'autre les arbres décisionnels. On s'aperçoit notamment que chacun à ses forces et faiblesses.

Les Boosted Trees ne sont qu'une facette des arbres décisionnels, il pourrait aussi être intéressant d'aborder le problème avec un Random Forest. En effet, l'avantage des Boosted Tree, qui est d'améliorer sans cesse les arbres en fonction des erreurs précédentes, peut être aussi un inconvénient dans certains cas. L'apprentissage est transmis d'arbre en arbre. Ainsi, l'erreur dans un arbre amont et potentiellement diffusée dans les suivants. Cette observation est d'autant plus vraie dans un jeu de données altérées comme le nôtre où les caractéristiques clés sont difficiles à déceler. Le Random Forest n'est pas soumis à ce risque de contamination puisque tous les arbres sont indépendants.

On s'aperçoit que le "bottleneck" dans cet exercice se situe dans la puissance de calcul mis à disposition pour l'entraînement des modèles. Les images peuvent être trop réduites, au détriment des performances du modèle, ou des images à la bonne taille mais au détriment du nombre d'itération et donc d'un entraînement qui n'arrivera pas à maturité.

A la suite des résultats de nos deux modèles, un nouvel axe de recherches s'ouvre à nous. Explorer de quelle façon nous pourrions combiner les points forts de chaque modèle, sur le modèle de l'**Ensemble Learning** par exemple. Le CNN excelle sur la classification entre patients sains et patients malades, puis le Boosted Tree pour classifier les deux types de pneumonie lorsqu'il s'agit d'un patient malade. Nos modèles seraient parfaitement complémentaires et nous pourrions s'imaginer à de brillants résultats. Mais à ce jour, si nous devions choisir un seul modèle pour résoudre notre classification, nous choisirions le CNN qui offre un résultat plus homogène.

## Conclusion

A l'issue de cet exercice nous concluons que le modèle le plus adapté à la classification d'images est le CNN. Cependant, cette observation reste à nuancer puisque nous n'avons abordé qu'une petite partie du machine learning. Toutefois, les résultats que nous avons obtenus dépassent nos attentes en début de projet.

Il a été intéressant de se former à deux approches distinctes. Il est vrai que dans le machine learning, les réseaux de neurones sont souvent mis en avant. Néanmoins, nous avons été surpris par les algorithmes de decision trees qui peuvent s'avérer tout aussi performants et même, à première vue, plus facilement abordables.

Nous avons constaté qu'il n'existe pas d'algorithme miracle pour répondre à un problème. En effet, chaque algorithme semble plus ou moins performant selon le problème abordé. Néanmoins, la découverte de l'Ensemble Learning, nous a permis d'observer que les faiblesses d'un modèle peuvent être comblées par les qualités d'un autre. La rigueur et la configuration des modèles jouent également une part non négligeable dans le processus.

N'ayant jamais fait d'IA auparavant, cette entrée en matière fut un excellent challenge. Nous avons dû faire énormément de recherches, autant sur le modèle à choisir que sur des détails tels que le fine-tuning des hyperparamètres.

Le choix du sujet n'est certainement pas anodin et rejoint les problèmes sanitaires actuels. Nos recherches nous ont donné une idée plus précise sur la diversité et l'omniprésence grandissante de l'IA dans le monde que nous vivons.

Impatients de découvrir les futurs projets IA qui nous seront attribués, ce premier projet conforte entièrement notre choix de spécialité.

Nous avons pris conscience qu'il est facile de mettre en œuvre un modèle IA, mais qu'il peut être très compliqué de l'affiner à la perfection. La sensibilité des hyperparamètres rend l'opération délicate. Un changement a priori anodin peut avoir un fort impact sur le résultat final. Trouver l'équilibre entre le bias et la variance tout en maintenant un modèle suffisamment complexe pour prétendre à un apprentissage qualitatif fut difficile. A ce jour, au bout de 100 epochs, notre modèle continue à faire un léger overfitting qui pourrait être amélioré. La puissance de calcul de nos machines a été un frein à notre recherche. Nous explorerons d'autres alternatives pour les projets suivants et pour ne pas avoir à brider notre modèle.

## Sources et Références

General :

<https://www.lovelyanalytics.com/2020/05/26/accuracy-recall-precision/>

<https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9>

<https://medium.com/swlh/image-classification-using-machine-learning-and-deep-learning-2b18bfe4693f>

CNN :

<https://pubs.rsna.org/doi/full/10.1148/ryai.2019190015>

<https://bdtechtalks.com/2020/03/02/geoffrey-hinton-convnets-cnn-limits/>

<https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a>

<https://deeplearning.fr/cours-theoriques-deep-learning/fonctionnement-du-neurone-artificiel/>

<https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5>

<https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-gradcam-554a85dd4e48>

<https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/>

<https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035>

Boosted Tree :

<https://docs.conda.io/en/latest/>

<https://towardsdatascience.com/machine-learning-part-18-boosting-algorithms-gradient-boosting-in-python-ef5ae6965be4>

<https://xgboost.readthedocs.io/en/latest/tutorials/model.html>

<https://medium.com/@aravanshad/gradient-boosting-versus-random-forest-cfa3fa8f0d80>

<https://www.datasciencecentral.com/profiles/blogs/decision-tree-vs-random-forest-vs-boosted-trees-explained>

<https://stats.stackexchange.com/questions/173390/gradient-boosting-tree-vs-random-forest>

<https://machinelearningmastery.com/tune-number-size-decision-trees-xgboost-python/>