Rapport

Zoidberg2.0

From X-ray images, an A.I that can detect pneumonia

Résumé

Ce projet a pour objectif la classification d'images radiographiques pour la détection de pneumonie (sain, virus et bactérie). Pour répondre à cette problématique, nous utilisons un modèle de réseau de convolution. Au vu de la faible quantité d'images dont nous disposons ($\sim$ 6000 images), nous avons décidé d'utiliser les méthodes de *transfer learning*. Ainsi notre modèle est basé sur *EfficientNet-B0* qui a été pré-entrainé sur le dataset ImageNet. Nous obtenons un coefficient de corrélation de Matthew à 0.75 sur notre ensemble de test. Celui-ci monte à 0.93 sur la classification binaire : sain versus pneumonie.

Sections

A. FrereJacques, M. Dupont, X. Nouaille, G. Collin, P.A. Bolteau

1 Introduction

1.1 Contexte & Objectifs

La détection des pathologies pulmonaires représente un défi majeur pour les professionnels de santé. Les médecins ont le plus souvent recours à la radiographie thoracique pour établir un diagnostic. Cependant, son interprétation peut être difficile et nécessite souvent une expertise considérable. Dans ce contexte, notre objectif est d'améliorer l'interprétation des radiographies en proposant une solution basée sur le machine learning. Nous devons ainsi développer un modèle de classification d'images de poumons.

Nous avons à notre disposition une base de données d'environs $\sim$ 6000 images classées en trois catégories : personne saine, pneumonie virale et pneumonie bactérienne. Pour atteindre notre objectif de classification, nous avons opté pour l'utilisation de réseaux de convolutions, une approche qui a montré de très bons résultats dans la classification d'images en extrayant les caractéristiques significatives des images à l'aide de filtres. Nous utiliserons Google Colab comme environnement de développement et travaillerons avec des bibliothèques populaires telles que *TensorFlow* [1] et *Keras* [2] pour développer notre modèle de classification.

1.2 Organisation du projet

Nous avons organisé notre projet autour de trois parties majeures, qui sont essentielles pour tout projet de machine learning : l'analyse des données, la comparaison des différents modèles en vue de sélectionner celui qui obtient les meilleures performances, et enfin l'optimisation du modèle sélectionné.
En plus des tâches techniques, nous avons également effectué diverses recherches, notamment sur les différents modèles potentiels, afin d'enrichir nos connaissances et d'obtenir les meilleurs performances. De plus, nous avons consacré du temps à la production de la documentation, qui joue un rôle essentiel dans la communication des résultats et des méthodes utilisées.

Au sein de notre équipe composée de 5 membres, nous avons réparti les tâches de la manière suivante :

- *A. FrereJacques* et *G. Collin* ont été responsables de l'analyse et de la préparation des données.
- *M. Dupont* et *X. Nouaille* se sont chargés de la production de la documentation ainsi que de la recherche sur les différents modèles potentiels.
- *P.A. Bolteau* s'est focalisé sur la comparaison des modèles et l'optimisation du modèle final.

Cette répartition nous a permis d'optimiser notre efficacité et de tirer parti des compétences spécifiques de chaque membre de l'équipe pour mener à bien toutes les facettes du projet.

La suite du rapport est structurée autour des 3 axes : analyse des données, choix du modèle et optimisation de celui-ci. Cependant, avant d'aborder ces axes, nous avons inclus deux parties [1] sur les connaissances théoriques nécessaires pour ce projet.

2 Le Machine Learning en bref

Avant de rentrer dans les détails de ce projet, nous présentons une rapide introduction au machine learning. Le but est avant tout de fournir les définitions utiles et de donner une compréhension intuitive sur ce sujet. Ainsi de nombreux aspects ne seront pas abordés (notamment l'existence de solutions et leur convergence).

1. Les sections annotées d'un * présentent des concepts mathématiques avancés.

2.1 Machine Learning : Kezako ?

Dans plusieurs domaines dans lesquelles l'informatique est utilisé, il peut être utile que l'on ne décrive pas explicitement les règles d'un algorithme mais plutôt que l'ordinateur les apprennent par lui-même. C'est exactement ce que l'on cherche à faire quand on parle de **Machine Learning** (ou apprentissage automatisé en français).
La définition de T. Mitchell fournie une explication plus pratique de ce domaine :

> *"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*
>
> Tom Mitchell, 1997

Illustrons ça tout de suite avec un des exemples le plus populaire : *La classification des emails en spam ou non-spam.*
Pour déterminer si un email est un spam, on pourrait tenter de définir des règles explicites sur le titre du mail, une blacklist de mots, etc. Cependant, l'algorithme serait excessivement grand si l'on veut qu'il soit efficace, et sa maintenance serait très complexe au fur et à mesure que de nouveaux spams soient créés. C'est là que l'on se rend compte qu'il nous faut passer par le machine learning. En reprenant la définition de T. Mitchell, nous aurions :

— la tâche T : classifier un email en spam ou non-spam
— l'expérience E : examiner une banque de données d'un ensemble d'emails étiquetés comme spam ou non-spam
— la performance P : la fraction d'emails correctement classés comme spam par l'algorithme.

Il existe de nombreux modèles de machine learning que l'on peut classer en sous-domaines présentés dans la Figure 1.

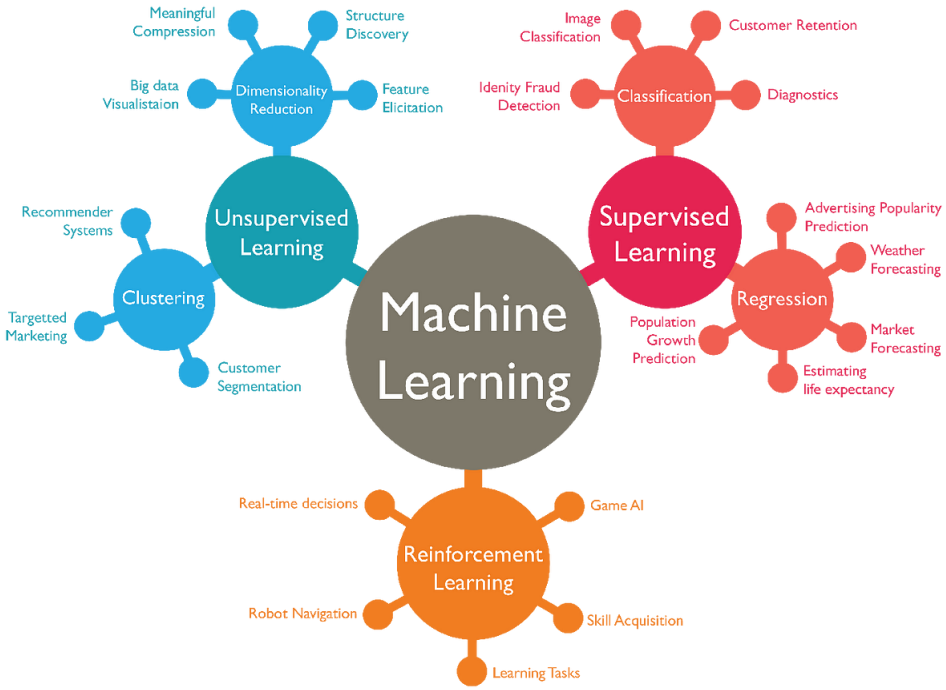


FIGURE 1. Cartographie non exhaustive des modèles de machine learning

Comme énoncé précédemment, dans ce projet, nous devons réaliser une classification en 3 classes. Nous allons donc nous restreindre à l'apprentissage supervisé, et en particulier à la classification [2].

2. A noté une équivalence entre classification et régression. En effet, la plupart des modèles s'adaptent très bien aux deux problématiques (SVM, decision tree, etc.).

2.2 De l'apprentissage à la prédiction

Essayons de formaliser la problématique de l'apprentissage supervisé :
A partir d'observations données $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y},\ 1 \leq i \leq m$ de *entées*[3]/*sorties* (appelées **données d'entraînements**), notre but est de prédire un nouveau label $y \in \mathcal{Y}$ à partir d'une nouvelle donnée $x \in \mathcal{X}$ (ces nouvelles données étant appelées **données de test**).
La prédiction n'est autre qu'une fonction dépendant de paramètres θ appelé **fonction d'hypothèse** :

$$h_\theta : \mathcal{X} \to \mathcal{Y}$$

Pour Zoidberd2.0, $\mathcal{X}$ est l'ensemble des images et $\mathcal{Y} = \{$ *normal*, *virus*, *bacteria* $\}$.
La forme de h_θ dépend du modèle choisie[4] : cela peut-être un simple polynôme dans le cas de la régression linéaire ou une fonction plus complexe comme ... un réseau de neurones (détaillé à la section 3). En revanche, θ est justement ce que l'on cherche à trouver via l'apprentissage.

En résumé, on cherche à trouver un algorithme $\mathcal{A}$ tel que $\mathcal{A}$ prend en entrée la distribution $\mathcal{D}(x_i)$ et nous renvoie θ tout en cherchant à ce que $h_\theta(x_i)$ soit le plus proche de y_i.

2.3 Risque & Fonction de perte*

Le coeur de l'apprentissage supervisé réside dans le fait de mesurer à quel point la valeur prédite d'une donnée d'entraînement est éloignée de sa valeur réelle. Pour quantifier cet écart, on définit la **fonction de perte** (loss function en anglais) :

$$\begin{aligned} \mathcal{L} : \mathcal{Y} \times \mathcal{Y} &\to \mathbb{R}^+ \\ (\overline{y}\,,\, y) &\mapsto \mathcal{L}(\overline{y}, y) \end{aligned}$$

avec $\overline{y_i} = h_\theta(x_i)$. $\mathcal{L}(\overline{y}, y)$ tend vers zéro si la prédiction est proche de la valeur réelle et tend vers l'infini dans le cas contraire.

On cherche un algorithme qui va minimiser cette écart entre "prédit" et "réelle" sur l'ensemble des données d'entraînement. C'est pourquoi on va regarde la moyenne de la fonction de perte sur l'ensemble des données. Cette moyenne étant appelée **risque** :

$$\mathcal{R}_\mathcal{A}[\mathcal{D}(x_i)] \coloneqq \mathbb{E}\Big[\mathcal{L}(h_\theta(x_i), y_i)\Big] = \frac{1}{m}\sum_i^m \mathcal{L}(h_\theta(x_i), y_i)$$

Ainsi, l'algorithme $\mathcal{A}$ cherche $\tilde{\theta}$ tel que :

$$\tilde{\theta} = \underset{\theta}{\operatorname{argmin}}\ \mathcal{R}_\mathcal{A}[\mathcal{D}(x_i)]$$

La définition de ce risque[5] permet de transposer notre problématique d'apprentissage en un problème d'optimisation. En effet, tout en gardant une très grande capacité de généralisation, nous avons pu réduire notre problème d'apprentissage au simple fait de minimiser une fonction.

Il existe de plusieurs techniques d'optimisation (comme l'équation normale) mais la plus utilisée est le **Gradient Descent** et ses dérivés[6] :

---

3. En générale, $x_i \in \mathbb{R}^n$ i.e. x_i est un vecteur de taille n où n correspond au nombre de *features* (variables caractéristiques de la donnée d'entrée). Dans notre projet, n correspond au nombre de pixels dans les images.

4. La fonction d'hypothèse ne peut pas prendre n'importe quelle forme : elle doit être un *concept que l'on peut apprendre* et mesurable par la VC dimension. [3] [4]

5. En réalité, le risque $\mathcal{R}_\mathcal{A}$ tend vers une valeur optimale et ce que l'on cherche à minimiser est $\mathcal{R}_\mathcal{A} - \mathcal{R}^*$ où $\mathcal{R}^*$ est le risque de Bayes. Pour plus de détails, on se référa à l'excellent cours de Francis Bach [5].

6. Il existe de nombreuses variantes du Gradient Descent que nous n'aborderons pas ici comme le Gradient Descent Stochastique ou Adam.

L'algorithme de Gradient Descent [a]

1. On initialise θ aléatoirement.
2. On met à jour θ pour réduire $J(\theta)$ jusqu'à ce qu'on atteigne un minimum :

```
repeat until convergence {
```

$$\theta_j := \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta) \quad (\text{pour } j = 0, \dots, n)$$

```
}
```

où $J(\theta) = \mathcal{R}_{\mathcal{A}}[\mathcal{D}(x_i)]$ et η est le taux d'apprentissage (learning rate en anglais) qui contrôle la taille des pas que l'algorithme prend à chaque étape.

a. Le Gradient Descent est développé en annexe A.1.

Cette partie étant très théorique, on propose un exemple de classification avec la régression logistique en annexe A.2.

2.4 Performances

Comme la tâche principale du machine learning consiste à sélectionner un modèle et à l'entraîner sur des données, les deux choses qui peuvent mal tourner sont soit un "mauvais modèle", soit de "mauvaises données".

2.4.1 Validation

Avant de regarder les problèmes énoncés, voyons comment peut-on mesurer si un modèle performe bien ou non.
La première des choses est de séparer notre base de donnée en deux (usuellement en 80%/20%) :

— données d'entraînement : données sur lesquelles le modèle apprend

— données de test : données sur lesquel le modèle final est testé. Ces données ne devront être **utilisées qu'à la toute fin** pour voir si le modèle performera bien une fois déployé.

Lorsque l'on entraîne un modèle, on cherche également à trouver ces meilleurs *hyper-paramètres* [7]. Pour ce faire, nous utilisons la validation :

— ensemble de validation : on redivise notre ensemble d'entraînement en 80%/20% et on test ces hyper-paramètres sur ces 20%

— cross-validation : on répète plusieurs fois le découpage entraînement/validation de manière aléatoire (souvent préféré pour maximiser les données d'entraînement et réduire la variabilité de la procédure de validation).

2.4.2 Gérer les données

On peut maintenant regarder les problématiques face aux mauvaises données :

1. *Manque de données* : C'est probablement le problème le plus courant. La plupart des modèles nécessitent des milliers de données (voire des millions pour les réseaux de neurones).
2. *Données non représentative* : Les données d'entraînement doivent être les plus représentatives des cas que nous souhaitons généraliser.
3. *Mauvaise qualité des données* : Trop d'erreurs, de bruits ou d'outliers dans nos données créeront inexorablement un biais dans notre modèle.
4. *Features non pertinentes* : Le système ne peut apprendre que si les features fournies permettent de caractériser l'output attendu. Pour Zoidberg2.0, la date de création des images ne nous permettrait en rien de classifier nos images.

7. Les hyper-paramètres du modèle sont des paramètres qui ne sont pas appris par le modèle lui-même, mais qui doivent être choisis avant l'entraînement pour déterminer la structure et le comportement du modèle. Exemples : le learning rate dans le Gradient Descent, le nombre de neurones dans un réseau, etc.

2.4.3 Underfitting & Overfitting

Côté modèle, les erreurs viennent du dilemme *biais* / *variance*.

1. *Biais* : Il y a un biais lorsque les hypothèses de départ sont erronées. Cela se produit lorsque le modèle est trop simple par rapport aux données fournies. Le modèle aura une grande erreur sur les données d'entraînement. On parle alors de sous-apprentissage (*underfitting* en anglais).
2. *Variance* : Une grande variance du modèle correspond à sa grande sensibilité aux légères variations des données d'entraînement. Le modèle va très bien apprendre mais il aura des erreurs importantes sur de nouvelles données. On parle alors de sur-apprentissage (*overfitting* en anglais).

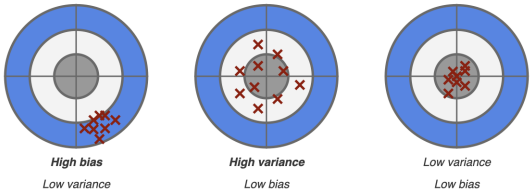


FIGURE 2. Exemple du dilemme Biais / Variance sur une cible

Dans les deux cas, il existe plusieurs solutions possibles en fonction du modèle utilisé. Voici quelques-unes des solutions possibles :

— Underfitting :
  1. choisir un modèle plus complexe avec plus de paramètres
  2. utiliser des caractéristiques (features) plus adaptées pour le modèle
  3. réduire les contraintes imposées au modèle.

— Overfitting :
  1. choisir un modèle plus simple avec moins de paramètres
  2. Restreindre notre modèle avec la régularisation
  3. ajouter plus de données au modèle pour augmenter la variété des exemples
  4. arrêter l'apprentissage du modèle dès qu'il ne progresse plus pour éviter d'apprendre trop spécifiquement les exemples d'entraînement.

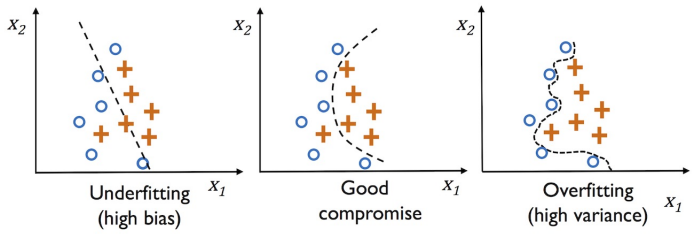


FIGURE 3. Underfitting et Overfitting dans le cas d'une classification

3 Plongée dans le Deep Learning

Maintenant que les bases du machine learning ont été présentées, on peut aborder une sous-branche de celui-ci : le **deep learning**. Ce domaine respecte les principes vus précédemment, mais il se caractérise par l'utilisation de réseaux de neurones artificiels comme fonction d'hypothèse. L'utilisation de neurones présente des avantages dans plusieurs domaines, car cela permet de traiter de très grandes quantités de données et de capturer des structures complexes en leur sein.
Comme discuté pour la régression logistique en annexe A.2, dans le machine learning classique, les modèles

de classification sont linéaires [8]. Mais si les données ne sont pas linéairement séparables, il faut rajouter des features (en combinant celles que nous avons à disposition). Cela entraîne un ralentissement inévitable de ces modèles. Les réseaux de neurones arrivent à gérer ce type de problèmes en combinant plusieurs modèles très simples (un modèle équivalent à un neurone).

3.1 Un neurone

Un neurone est une unité de base dans les réseaux de neurones. Il prend en entrée plusieurs features, telles que la couleur ou la taille d'un objet, et renvoie une seule valeur. Cette valeur peut être utilisée pour prédire une catégorie ou une valeur numérique. Le neurone fait passer ces features par 2 étapes :

1. Transformation linéaire : le neurone associe à chaque caractéristique un poids qui reflète son importance dans le calcul de la valeur de sortie. Ces poids θ sont multipliés par les features et additionnés pour produire une valeur intermédiaire.
2. Transformation non-linéaire : la valeur intermédiaire est passée à travers une fonction non-linéaire, appelée fonction d'activation (notée σ), qui produit la valeur de sortie finale. Cette étape est nécessaire car les relations entre les caractéristiques d'entrée ne sont souvent pas linéaires.

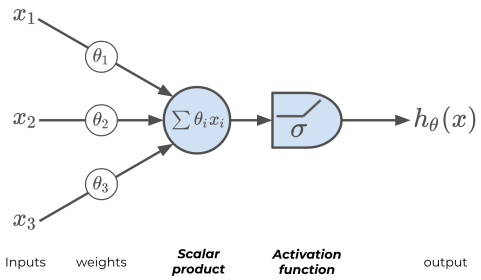


FIGURE 4. Exemple d'un neurone avec 3 entrées et la fonction ReLu(x) comme fonction d'activation

Au final, si $x \in \mathbb{R}^n$ est le vecteur d'entrée, la sortie d'un neurone est :

$$h_\theta(x) = \sigma(\theta.x + b)$$

où b est un biais rajouté à chaque neurone [9]. Dans le cas où σ est la fonction sigmoïde, un neurone équivaut à la régression logistique.

Il existe plusieurs fonctions d'activation possibles. Voici les plus utilisées :

— la fonction *sigmoïde* : $S(x) = \frac{1}{1+e^{-}x}$ qui est utilisée notamment dans la régression logistique. Elle restreint la sortie entre $[0, 1]$.

— la fonction *tangente hyperbolique* : $\tanh(x) = \frac{e^x+e^{-x}}{e^x-e^{-x}}$. Elle restreint la sortie entre $[-1, 1]$.

— la fonction *Rectified Linear Unit* : $\text{ReLu}(x) = \max(x, 0)$ et ses dérivées. Elle est très en vogue actuellement du faite que sa sortie soit dans $\mathbb{R}^+$.

3.2 Des neurones

Dans un réseau de neurones artificiels, on va simplement utiliser un ensemble de neurones que l'on organise en **couches**. Les neurones de la couche l utilisent les sorties des neurones de la couche $l-1$ comme entrées. Dans chaque couche l de neurones, nous utilisons généralement la même fonction d'activation σ_l. Dans notre exemple, nous examinons la couche $l = 1$, qui est la couche la plus basse du réseau de neurones. Nous notons $\theta_{i,j}^{(1)}$ le poids de l'input x_j pour le neurone i dans la couche $l = 1$. Ainsi, cette couche a comme sortie :

$$a^{(1)} = \sigma_1(\Theta^{(1)}.x + B)$$

8. Ils séparent les classes en traçant une droite, pas une courbe.

9. Le biais est une valeur ajoutée à un neurone pour compenser les différences dans les données d'entrée.

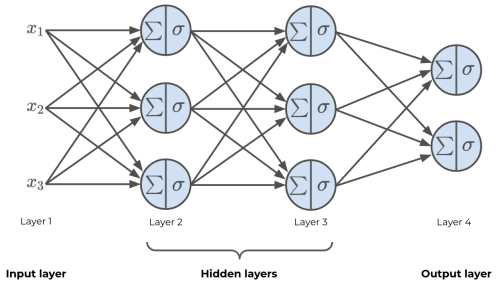


FIGURE 5. Réseau de neurones entièrement connectés avec 4 couches et une sortie en 2 neurones

où $a^{(1)}$ est un vecteur de dimension 3. Ce vecteur sera ensuite utilisé comme entrée pour la couche $l = 2$. La matrice de poids θ est devenue une matrice $\Theta^{(1)}$ de dimensions 3×3 [10] dont les éléments sont $\theta^{(1)}_{i,j}$.

Nous pouvons également remarquer que notre réseau se termine avec 2 neurones en sortie. Le nombre de neurones en sortie d'un réseau de neurones dépendra du nombre de classes du problème. Nous aurons un neurone en sortie pour chaque classe. Chaque neurone en sortie renverra un score de probabilité qui représente la probabilité d'appartenance de l'input à cette classe. La somme des scores de tous les neurones de sortie doit être égale à 1, car l'input doit appartenir à l'une des classes. La classe prédite sera celle ayant le score de probabilité le plus élevé [11].

Enfin, lorsqu'un neurone dans une couche est connecté à tous les neurones de la couche précédente, nous parlons de couche entièrement connectée. Ce type de couche est couramment utilisé dans les réseaux de neurones pour permettre aux neurones de recevoir des informations de toutes les entrées précédentes.

3.3 Entraîner un réseau de neurones*

L'entraînement des réseaux de neurones a été l'un des défis majeurs en deep learning au XX^e siècle. En 1986, D. Rumelhart, G. Hinton et R. Williams ont proposé une solution appelée **backpropagation** [6], qui est maintenant largement utilisée.

La backpropagation n'est pas un algorithme d'optimisation en soi, mais plutôt une méthode utilisée pour calculer les gradients du risque par rapport aux poids du réseau de neurones. Ces gradients sont ensuite utilisés par des algorithmes d'optimisation tels que le Gradient Descent pour ajuster les poids du réseau, afin de minimiser le risque.

L'algorithme de Backpropagation

1. On initialise l'ensemble des poids $\Theta^{(k)}$ aléatoirement.
2. On passe les inputs à la couche d'entrée jusqu'à celle de sortie pour nous donner $\overline{y}$. On mesure, ensuite, le risque du modèle. C'est la **forward pass**.
3. On calcule ensuite le gradient descent avec la **chain rule**.
4. Le gradient est propagé en arrière à travers le réseau pour mettre à jour les poids. La mise à jour est celle du gradient descent :

$$\Theta^{(k)} = \Theta^{(k)} - \eta \frac{\partial J}{\partial \Theta^{(k)}}$$

C'est la **backward pass**.

10. Pour chaque couche l, $\Theta^{(l)} \in \mathcal{M}_{n_l \times n_{l-1}}(\mathbb{R})$ i.e c'est une matrice avec n_l lignes (nombre de neurones de la couche l) et n_{l-1} colonnes (nombre de neurones de la couche $l-1$).

11. C'est pourquoi nous utilisons généralement en sortie du réseau de neurones, une fonction d'activation spéciale appelée **softmax** qui normalise les scores de probabilité en une distribution de probabilité sur toutes les classes.

En reprenant notre exemple de réseau, on a (en omettant le biais) :

$$a^{(1)} = \sigma_1(\Theta^{(1)}.x) \tag{1}$$
$$a^{(2)} = \sigma_2(\Theta^{(2)}.a^{(1)}) \tag{2}$$
$$a^{(3)} = \sigma_3(\Theta^{(3)}.a^{(2)}) \tag{3}$$

avec $h_\Theta(x) = a^{(3)}$. La chain rule nous permet d'obtenir la dérivée du risque J par rapport à n'importe quel paramètre. Exemple avec $\Theta^{(2)}$, en définissant $z^{(l)} = \Theta^{(l)}.a^{(l-1)}$:

$$\frac{\partial J}{\partial \Theta^{(2)}} = \frac{\partial J}{\partial a^{(3)}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial \Theta^{(2)}} \tag{4}$$
$$= \frac{\partial J}{\partial a^{(3)}} \cdot \sigma_3' \cdot \Theta^{(3)} \cdot \sigma_2' \cdot a^{(1)} \tag{5}$$

Les termes σ_3' et σ_2' sont simplement les dérivées des fonctions d'activation de la couche 2 et 3, les termes $\Theta^{(3)}$ et $a^{(1)}$ sont calculés lors de la forward pass. Il ne reste que le terme $\partial J/\partial a^{(3)}$ qui est facile à calculer.

3.4 Les réseaux de convolution

La dernière partie présente la structure de réseau que nous allons utiliser dans ce projet : **les réseaux de convolution**. Ce type de réseau est particulièrement performant dans la reconnaissance d'image. Sa performance vient du faite qu'il :

— permette de réduire le nombre de paramètres et donc le temps d'entraînement [12] en réduisant le nombre de connection d'un neurone et en partageant les mêmes paramètres au sein d'une couche.

— capture des structures de plus en plus complexes. Leur première couche va capturer des structures très simples dans les images, puis la couche suivante va agréger ces structures pour en produire de plus complexe et ainsi de suite.

Pour cela, il s'appuie sur deux opérations : la *convolution* & le *pooling*. Un réseau de convolution est présenté à la figure 6.

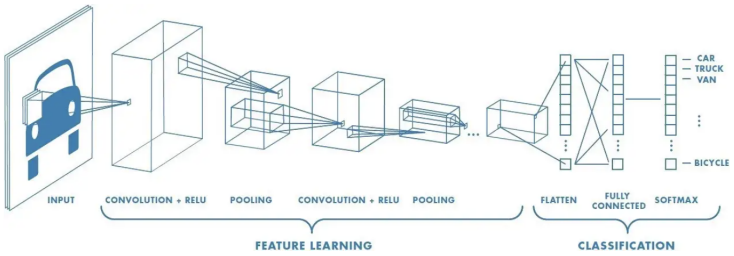


FIGURE 6. Réseau de convolution typique : la 1re partie contient successivement des couches de convolution et de pooling. La 2e partie est un réseau entièrement connecté (vu précédemment).

3.4.1 Couche de convolution

Une couche de convolution contient les mêmes éléments qu'une couche entièrement connectée : une transformation linéaire et une transformation non linéaire (dans ce cas-ci, la fonction d'activation est ReLu). Cependant, la différence réside dans la transformation linéaire, le produit scalaire vu dans la section précédente est remplacé par ... la **convolution** !

Dans une couche de convolution, chaque neurone est connecté à une petite partie de la couche précédente appelée *champ récepteur*. Cela signifie que chaque neurone a un nombre limité de paramètres, déterminé par la taille du champ récepteur. Ces paramètres sont regroupés dans une matrice que l'on appelle *kernel*. Ce kernel est le même pour tous les neurones de la couche, ce qui permet de réduire le nombre de paramètres à apprendre. En partageant les mêmes poids, le réseau de neurones peut détecter

12. Imaginons une image de 100×100 pixels (soit 10 000 features) et une 1re couche entièrement connectée contenant 1 000 neurones, cela représenterait 10 000 000 de paramètres rien que pour cette couche !

des motifs similaires à différentes positions de l'entrée sans avoir à apprendre de nouveaux poids pour chaque position. Cette propriété est particulièrement utile pour la détection de motifs spatiaux, tels que les bords, les coins ou les textures, dans une image.
Quand on calcul le produit du champ récepteur avec le kernel, on obtient la sortie d'un neurone de convolution. L'ensemble des sorties d'une couche de convolution est appelé *feature map* (cf Figure 7).

On peut utilisé plusieurs kernels par couche de convolution, ce qui donnera plusieurs feature maps par couche, ces feature maps étant les inputs de la couche suivante.

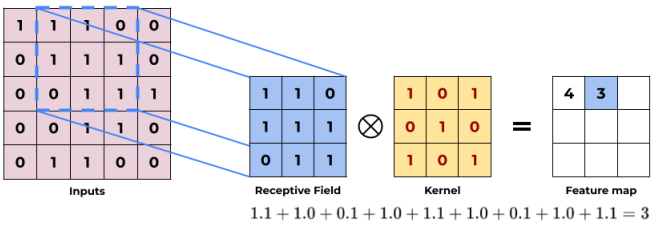


FIGURE 7. Exemple de convolution avec un champ récepteur de taille 3×3.

3.4.2 Couche de pooling

La couche de pooling est plus simple. Son but est d'agréger les données pour réduire le nombre de paramètres. Comme dans la couche de convolution, chaque neurone de la couche de pooling est connectée aux neurones de la couche précédente, présent dans son champ récepteur. Un neurone de pooling va alors agréger les inputs de son champ récepteur via une fonction tel que :

— la fonction *max* : renvoie l'input le plus élevé

— la fonction *avg* : renvoie la valeur moyenne des inputs.

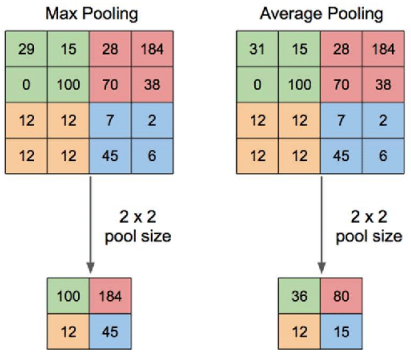


FIGURE 8. Exemple de pooling où chaque champ récepteur est associé à la couleur de son neurone de pooling

4 Préparation des données

Après avoir exploré les concepts préalables requis, nous sommes maintenant en mesure d'examiner les données fournies et d'identifier les préparations nécessaires.

4.1 Analyse des données

Nous avons à notre disposition un dataset labelisé comprenant un total de 5856 images au format *JPEG*. Ces images ont déjà été organisées en trois dossiers distincts : `train` (entraînement), `test` (test) et `val` (validation). Chaque image représente une radiographie des poumons d'un patient.

Notre objectif est de détecter et classer les images dans l'une des trois catégories suivantes : *bacteria* (pneumonie bactérienne), *normal* (personne saine) ou *virus* (pneumonie virale). Chaque classe représente un état spécifique des poumons, permettant ainsi d'identifier les éventuelles infections bactériennes ou virales.

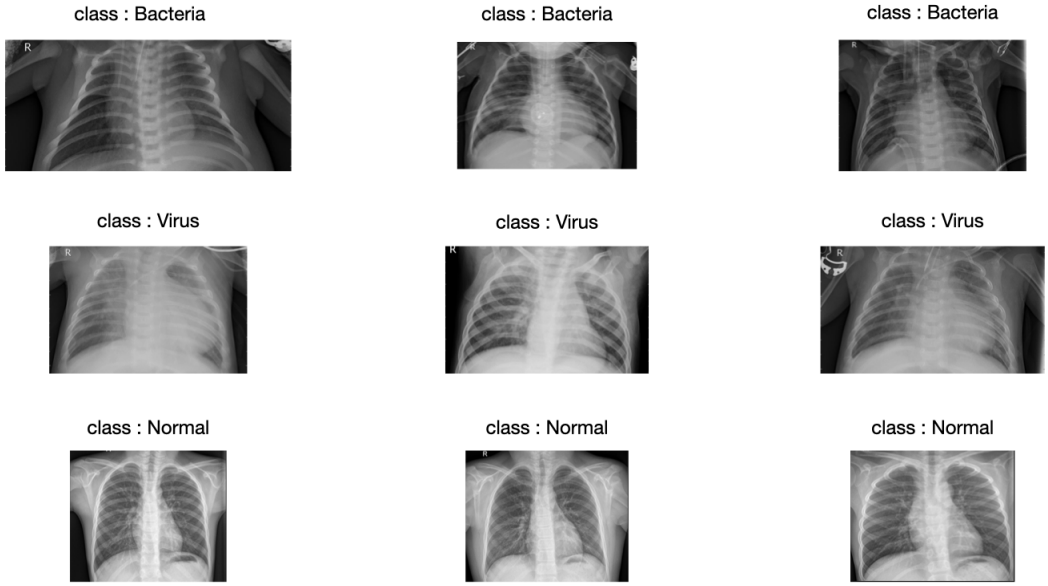


FIGURE 9. Exemple d'images dans notre dataset

Lorsqu'on examine le nombre d'images par classe, tel qu'illustré dans la figure 10, on remarque un déséquilibre entre les classes. En effet, il y a deux fois plus d'images classées comme *bacteria* que d'images classées *normal* ou *virus*. Cette disparité peut entraîner un sur-apprentissage du modèle sur la classe *bacteria* au détriment des autres classes.

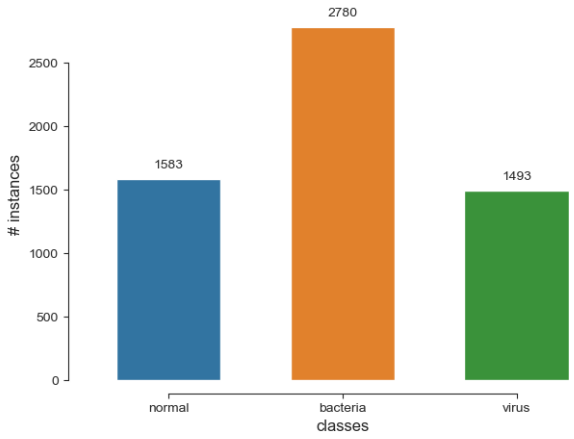


FIGURE 10. Distribution des données par classes sur l'ensemble de notre dataset.

Pour remédier à cette problématique, il existe plusieurs solutions envisageables. Dans notre cas, nous avons opté pour l'ajout de poids dans la fonction de perte, en assignant un poids plus faible à la classe contenant le plus grand nombre d'images :

- classe *bacteria* : poids = 0.70
- classe *normal* : poids = 1.24
- classe *virus* : poids = 1.32

De plus, il faudra faire attention aux métriques qui sont "insensibles" aux classes déséquilibrées comme l'accuracy et préférer des métriques tel que le F1-score ou le coefficient de corrélation de Matthew.

Afin de pouvoir entraîner notre modèle, il est essentiel que chaque image présente la même taille et

le même nombre de canaux (par exemple, noir et blanc, RGB, etc.). Par conséquent, nous devons maintenant examiner les métadonnées de ces images.

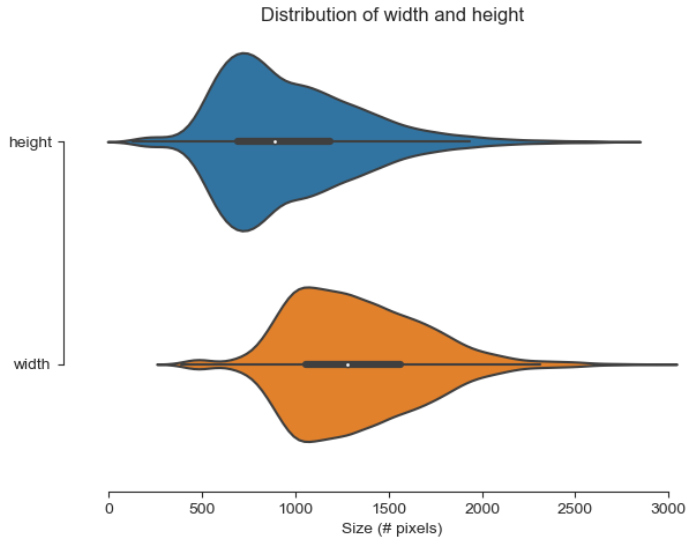


FIGURE 11. Distribution (kernel density) des images : en *bleue* selon leur hauteur, en *orange* selon leur largeur. Taille donnée en nombre de pixels.

Bien que toutes les images soient en noir et blanc, elles diffèrent toutes en termes de taille. Au vu de la distribution de leur taille 11, on peut raisonnablement les redimensionner en 512×512 sans risquer de perdre trop d'information [13].

4.2 Préparation de données

Maintenant que nous avons une meilleure compréhension de nos données, il est temps de les préparer. Dans notre cas, qui concerne des images, la préparation des données est assez simple : nous devons les *redimensionner* et les *normaliser*.

En ce qui concerne le redimensionnement, nous devons nous assurer de ne pas dépasser la taille de 512×512, ce qui est une limite importante. Les modèles que nous allons utiliser ont généralement besoin d'images de taille 224×224, ce qui est bien inférieur à cette limite.
La normalisation est souvent nécessaire pour éviter les problèmes liés à l'échelle des variables. Lorsque les variables d'entrée présentent des ordres de grandeur différents, cela peut entraîner des problèmes lors de l'entraînement du modèle. De plus, cela permet d'accélérer la convergence du modèle [14].

Augmentation des données : Étant donné le nombre limité de données disponibles et la complexité des modèles utilisés, il existe un risque élevé de sur-apprentissage (overfitting) [15]. Une des façons de gérer ce problème consiste à augmenter "virtuellement" le nombre de données en appliquant aléatoirement des transformations aux données avant l'entraînement. Nous appliquerons les 3 transformations suivantes :

1. Zoom : zoom ou dézoome l'image de maximum 10%.
2. Translation : décale l'image verticalement et horizontalement de maximum 10%.
3. Rotation : tourne l'image de maximum 10% d'une rotation complète.

Ces 3 transformations ont été choisis car elles conserves la structure des images en évitant un maximum la perte d'un information. Un exemple sur une image est présenté dans la figure 12.

13. Le risque vient principalement de sur-dimensionner des images (augmenter la taille) ce qui peut occasionner un "flou" dans l'image.

14. Nous normaliserons les données pour tout nos modèles excepté EfficientNet qui est construit sans que nous ayons besoin de le faire.

15. Nous avons effectivement des problèmes d'overfitting, comme présenté dans la section 6.3.

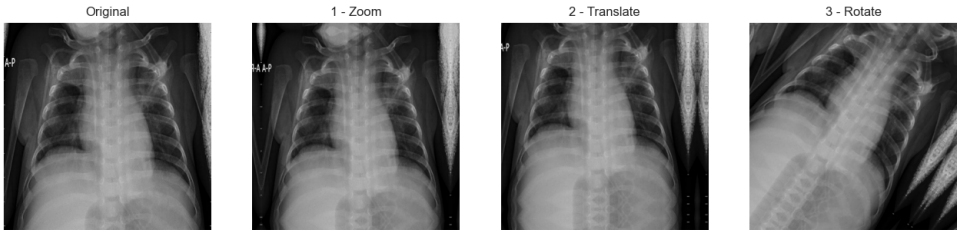


FIGURE 12. Exemple des transformations utilisées pour l'augmentation des données.

5 Modèles utilisés

La construction d'un modèle complet basé sur un réseau de convolution peut être long et complexe. C'est pourquoi, nous allons utiliser le **transfer learning**. Le but est de réutiliser un modèle ou une partie d'un modèle déjà construit et entraîné par un tiers. Il nous faut alors choisir quel modèle utiliser.
Au vu de la littérature, nous avons restreint notre choix à 5 modèles que nous allons comparer : *VGG*, *ResNet*, *Inception-ResNet*, *Xception*, *EfficientNet*.
A noter que chacun de ces modèles présentent différents versions de plus en plus complexes (réseau plus profond). Comme nous voulons éviter l'overfitting, nous utiliserons à chaque fois les versions les moins profondes.

Dans cette partie, nous présentons succinctement ces modèles.

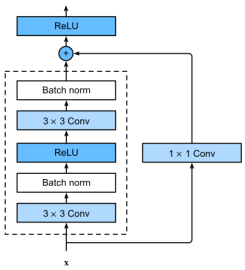
5.1 VGG (*VGG-16*)

Le modèle VGG-16 [7] est un réseau de convolution qui a été proposée par l'équipe Visual Geometry Group (VGG) de l'Université d'Oxford. L'indice "16" dans son nom fait référence au fait qu'il comporte 16 couches de convolution et de pooling. VGG est un réseau de convolution "classique" comme présenté précédemment avec la succession convolution / pooling.

L'avantages du modèle VGG-16 résident dans sa capacité à extraire des caractéristiques de manière précise et discriminative. Sa profondeur lui permet de mieux capturer les détails et les textures des images, ce qui peut être particulièrement bénéfique pour des tâches de classification d'images complexes. Cependant, la principale limitation du modèle VGG-16 réside dans son coût computationnel élevé en raison de sa profondeur. Il contient un grand nombre de paramètres, ce qui rend son entraînement et son utilisation plus exigeants en termes de ressources.

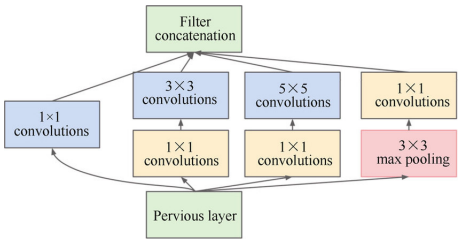
5.2 ResNet (*RestNet-50*)

Le modèle ResNet-50 [8] est une architecture qui appartient à la famille des réseaux résiduels (ResNet). Il a été développé par Microsoft Research. L'indice "50" dans son nom indique qu'il comprend 50 couches, ce qui en fait un modèle profond. La particularité du ResNet-50 réside dans l'utilisation de blocs résiduels. Ces blocs permettent de résoudre le problème de dégradation de la performance des réseaux profonds en introduisant des connexions *skip* (raccourcis) qui sautent certaines couches. Cela permet d'apprendre des représentations résiduelles plutôt que des représentations complètes à chaque couche, facilitant ainsi l'apprentissage en profondeur.



5.3 Inception-Resnet

Le modèle Inception-ResNet [9] est une architecture qui combine les concepts de
l'Inception Network et du ResNet. Cette version a été développée par l'équipe de recherche de Google. En termes de profondeur, l'Inception ResNet est un modèle relativement profond avec plusieurs dizaines de couches. Nous avons vu précédemment le principe de ResNet. Introduisons maintenant l'inception. Les modèles d'inceptions se base sur des cellules d'inception, qui permettent une extraction d'information à différentes échelles spatiales.



Le module d'inception se compose de plusieurs branches parallèles, chacune effectuant une opération de convolution avec des filtres de tailles différentes. Ces branches sont conçues pour capturer des motifs à différentes résolutions spatiales, allant des détails fins aux caractéristiques plus globales.

5.4 Xception

Le modèle Xception [10], également connu sous le nom d'Extreme Inception, est une architecture proposée par François Chollet, le créateur de Keras. Il s'inspire de l'architecture InceptionNet, mais avec une approche différente pour l'extraction des caractéristiques. En termes de profondeur, le modèle Xception est relativement profond avec plusieurs dizaines de couches.

Il utilise une technique appelée "convolution séparable" qui consiste à séparer la convolution standard en deux étapes distinctes : une convolution spatiale et une convolution en profondeur. Le modèle peut ainsi effectuer des opérations de convolution à différentes résolutions spatiales, ce qui lui permet de capturer des motifs à la fois fins et globaux dans les images.

5.5 EfficientNet (*EfficientNet-B0*)

Le modèle EfficientNet-B0 [11] est une architecture qui a été développée pour obtenir un équilibre optimal entre la précision de classification et l'efficacité computationnelle. Il fait partie de la famille des modèles EfficientNet, qui se caractérisent par leur approche d'optimisation de la taille du modèle en ajustant simultanément la profondeur, la largeur et la résolution des réseaux.

EfficientNet-B0 est relativement moins profond par rapport à d'autres modèles tels que ResNet ou InceptionNet. Cependant, grâce à une architecture spécialement conçue, il parvient à obtenir de bonnes performances tout en réduisant le nombre total de paramètres du modèle, ce qui permet une meilleure efficacité computationnelle.

6 Résultats obtenus

6.1 Choix des métriques

Pour assurer le succès de notre projet, il est essentiel de choisir les métriques les plus appropriées. Pour ce faire, nous allons compiler une liste des métriques les plus couramment utilisées et les comparer en fonctions des besoins de notre projet.

Commençons par rappeler les spécificités du projet. Nous avons un modèle supervisé de classification d'images avec 3 classes (multi-classes). On dispose de peu de données d'entraînements ($\sim$ 6000) et les classes ne sont pas bien balancées.
De plus, d'un point de vue métier, on se trouve dans le secteur médicale. On peut donc supposer qu'il faut mettre avant tout l'accent sur la détection de la pneumonie, car elle permettra un traitement rapide et efficace des patients atteints de cette maladie potentiellement mortelle. Bien que la distinction entre les pneumonies virales et bactériennes, ainsi que les faux positifs [16], soient également des considérations importantes, elles ne doivent pas être prioritaires par rapport à la détection initiale de la pneumonie.

Il existe de nombreuses métriques pour évaluer un modèle de classification. La plupart sont basées sur la matrice de confusion d'une classification binaire représenté sur la figure 13. Pour généraliser un modèle binaire à un modèle multi-classe, on utilise souvent le principe *OvR* (i.e One vs Rest) : chaque classe est comparée à l'ensemble de toutes les autres classes prises ensemble [17].

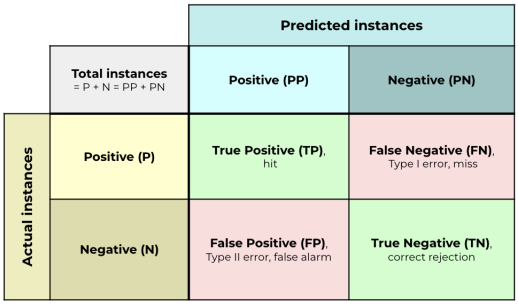


FIGURE 13. Matrice de confusion dans le cas d'une classification binaire

Regardons les métriques principales référencé dans le tableau ci-dessus. En gardant en tête le principe *OvR*, on ne les définie que dans un cadre de classification binaire dans le tableau 1.
Comme les classes ne sont pas bien balancées, les métriques *Accuracy* et *AUC ROC* ne peuvent être trompeuses et sont à délaisser. En effet, en reprenant la formule de l'accuracy, si une classe est très minoritaire dans l'ensemble de données, le modèle peut prédire systématiquement la classe majoritaire et atteindre une accuracy élevée, même s'il fait de très mauvaises prédictions pour la classe minoritaire. Pour AUC ROC, c'est le même principe. Si la classe majoritaire est bien séparée de la classe minoritaire, la courbe ROC peut atteindre une valeur élevée même si le modèle ne fait pas de bonnes prédictions pour la classe minoritaire.
Les métriques *Precision* et *Recall* pourront nous données de bonnes indications sur la façon de le modèle performe. Mais les classes n'étant pas bien balancées, on utilisera la moyenne pondérée pour avoir leur valeurs globales sur l'ensemble des données.
Nous n'utiliserons pas le *F1-score* comme métrique globale. En effet, comme il ne se base que sur la précision et le rappel, il n'évalue que les "true positives" et délaisse les "true négatives". Ceci peut conduire à des biais si l'on utilise le F1-score pour résumer la performance du modèle.
Le *Kappa score* et le *Matthews correlation coefficient* sont tout deux utiles dans notre cas. Cependant, comme l'on souhaite mettre un coût plus important sur la non-détection de la pneumonie que sur le reste des erreurs, la symétrie du coefficient de Matthew en fait la métrique la plus intéressante.

16. Personnes saines qui sont classifiées comme personnes atteintes de la maladie.

17. Il existe aussi *OvO* (i.e One vs One) : On compare chaque couple de classes. Cependant *OvR* est souvent préféré car plus rapide et plus stable.

Métrique	Formule	Interprétation
Accuracy	$\frac{TP+TN}{P+N}$	Proportion de prédictions correctes (instances correctement classifiées)
Precision	TP/PP	Proportion des prédictions positives réellement positives
Recall	TP/P	Proportion des instances positives correctement classifiées
F1-score	$2 \times \frac{Precision \times Recall}{Precision + Recall}$	Moyenne harmonique de la précision et du rappel (cherche à maximiser la précision et le rappel)
AUC ROC	Aire sous la courbe ROC	Capacité du modèle à distinguer les classes. ROC : rappel en fonction de 1 - la spécificité (TN/N)
Kappa Score	$\frac{p_o - p_e}{1 - p_e}$	Mesure de la concordance entre les prédictions et les observations. p_o : la proportion d'accord observé, p_e : la proportion d'accord attendu par hasard.
Matthews correlation coefficient	$\frac{TP \times TN - FP \times FN}{\sqrt{PP \times P \times N \times PN}}$	Coefficient de corrélation entre les prédictions et les observations

Table 1. Comparaison des métriques pour la classification binaire

Pour résumer, on utilisera le coefficient de Matthew (noté MCC par la suite) comme métrique globale. La précision et le rappel seront calculés pour chaque classe et globalement pour mieux comprendre les erreurs du modèles. On calculera également $\text{FPR}_{\text{normal}} = \text{FP}_{\text{normal}}/\text{N}_{\text{normal}}$ qui nous donnera le ratio de pneumonies non détectées.

6.2 Comparaison des modèles

Pour comparer la performance de nos modèles, nous utilisons une petite partie de nos données : 20% de l'ensemble d'entraînement (soit $\sim$ 1000 images) et de l'ensemble de validation. Cela nous permet de sauver du temps et d'économiser des ressources.
Chaque modèle étant déjà pré-entraîné sur le dataset d'ImageNet, pour les comparer, nous entraînerons que les couches hautes [18] sur 30 epochs.
De plus, comme discuté dans la section précédente, nous utiliserons uniquement le coefficient de Matthew comme métrique de comparaison.

Les méta-résultats obtenus sont présentés dans le tableau 2 suivant.

Modèle	Taille des paramètres	Temps d'entraînement
VGG-16	373.5Mb	135 s
ResNet-50	126.1Mb	116 s
Inception-ResNet	243.8Mb	160 s
Xception	115.2Mb	128 s
EfficientNet-B0	46.2Mb	118 s

Table 2. Méta-résulats des différents modèles.

On observe que le modèle le plus léger, de loin, et l'un des plus rapide est EfficientNet. VGG au contraire est extrêmement lourd (le fichier contenant ces paramètres est 10 fois plus lourd que celui pour EfficientNet). Passons maintenant au MCC calculé sur l'ensemble de validation `val_MCC` présenté dans la figure 14 ci-dessous.

18. Les couches hautes correspondent aux couches à la fin du modèle après celles de convolution et de pooling.

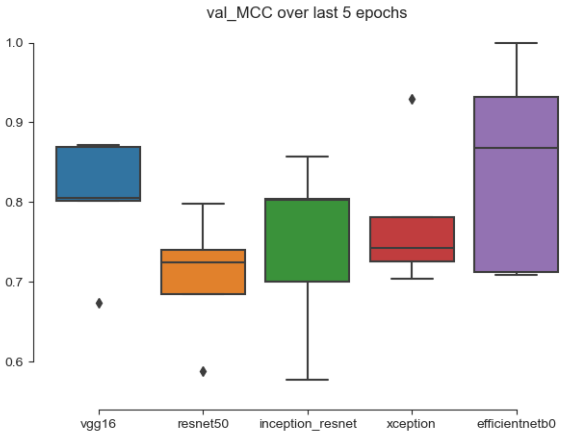


FIGURE 14. Comparaison du MCC des modèles sur l'ensemble de validation sur leur 5 dernières epochs.

Le MCC est relativement bon pour l'ensemble des modèles. Toutefois, VGG et EfficientNet semble être les plus prometteurs.

On finit par regarder les courbes d'apprentissage pour chaque modèle dans la figure 15. Nous observons une grande volatilité sur la validation (dû au peu de données présentes) mais pas d'overfitting apparent.

Au vu de l'ensemble de ces résultats, notre choix se porte sur **EfficientNet-B0** qui sera performant et lèger (très utile pour les interfaces que nous utilisons i.e Colab et nos ordinateurs personnels).

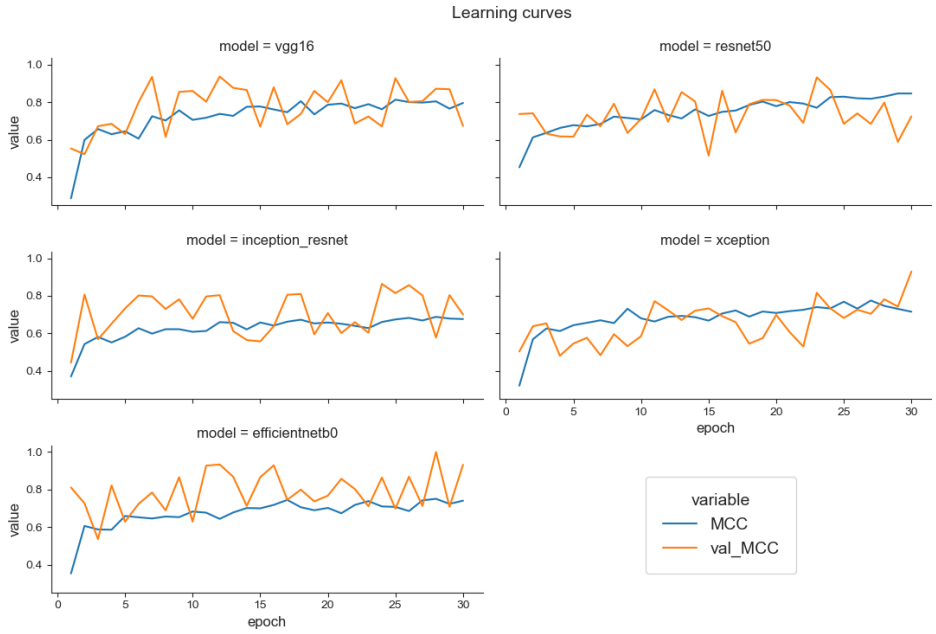


FIGURE 15. Courbes d'apprentissage pour chaque modèle.

6.3 Évaluation de la performance finale

Une fois le modèle choisi, il nous faut l'entraîner sur l'ensemble d'entraînement complet.
En reprenant, notre modèle de base EfficientNet-B0 sans l'augmentation des données nous obtenons les courbes d'apprentissage sur la figure 16.
Le modèle semble extrêmement bien performer : la valeur du MCC est très proche de 1 mais avec beaucoup d'overfitting (i.e un écart de 0.2 entre `MCC` et `val_MCC`) ! Il nous faut donc gérer ce problème.

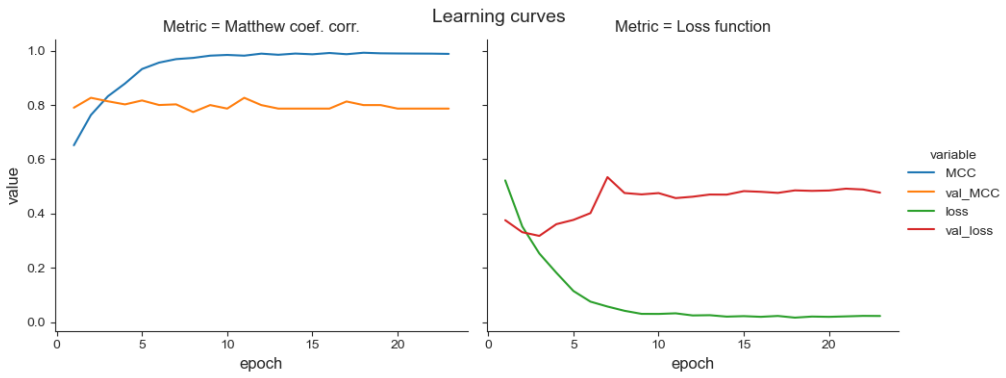


FIGURE 16. Courbe d'apprentissage du modèle EfficientNet-B0 avant les techniques d'overfitting.

Pour se faire, nous utilisons 3 techniques :

- L'augmentation des données comme discuté précédemment.
- L'abandon (dropout) : sur les couches hautes de notre modèle, on désactive aléatoirement un certain pourcentage de neurones pendant l'entraînement, ce qui réduit la dépendance entre les neurones.
- L_1-régularisation : sur les couches hautes de notre modèle, on ajouter une pénalité à la fonction de perte en proportion de la somme des valeurs absolues des poids du modèle.

Ce qui nous donne les courbes d'apprentissage de la figure 17.

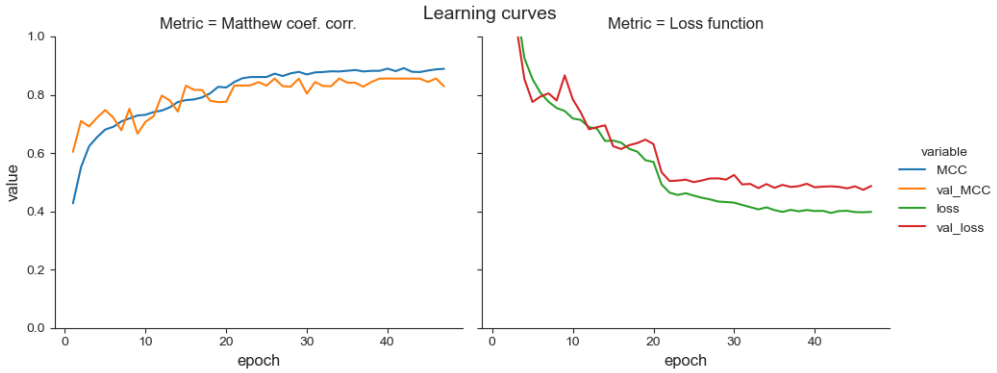


FIGURE 17. Courbe d'apprentissage du modèle EfficientNet-B0 après les techniques d'overfitting.

Nous avons toujours un léger problème d'overfitting mais le modèle semble mieux se comporter. On peut donc regarder les résultats finaux sur l'ensemble de test.

Sur cet ensemble nous obtenons les métriques globales :

Precision	**Recall**	**$\mathbf{FPR_{normal}}$**	**MCC**
0.84	0.83	0.03	0.75

et pour chaque classe :

Classe	**Precision**	**Recall**	**MCC**
bacteria	0.90	0.74	0.70
normal	0.91	0.99	0.93
virus	0.69	0.82	0.63

Pour plus de détails, nous présentons également la matrice de confusion 18 sur l'ensemble des classes.
Le problème d'overfitting est toujours présent vu la valeur plus faible du MCC que nous obtenons sur l'ensemble de test. Toutefois, cette valeur reste relativement bonne.
Le $\text{FPR}_{\text{normal}}$ et la matrice de confusion nous montre que nous sommes très performant dans la détection

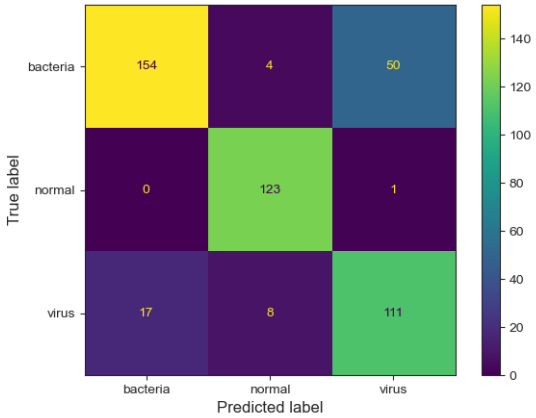


FIGURE 18. Matrice de confusion du modèle sur l'ensemble de test.

de la pneumonie mais c'est la distinction virus / bactérie qui nous pénalise. Avec une précision sur la classe *virus* à 0.69 et 0.90 pour la classe *bacteria*, on voit clairement que l'axe d'amélioration réside dans une meilleur classification des bactéries entre *bacteria* et *virus*.

Nous proposons 2 solutions pour tenter d'améliorer ce problème :

1. Pré-entrainer le modèle sur un ensemble proche du notre (image de radiographie), contenant beaucoup d'image. Ce genre de dataset est disponible sur internet.
2. Utiliser les techniques d'ensemble learning. Autrement dit, entraîner un modèle spécifiquement pour résoudre ce problème et utiliser les deux modèles ensembles pour faire une prédiction.

Références

[1] G. B. Team, "Tensorflow : An open source machine learning framework for everyone," https://www.tensorflow.org/.

[2] F. Chollet, "Keras," https://keras.io/, 2015.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Learnability and the vapnik-chervonenkis dimension," *J. ACM*, vol. 36, no. 4, p. 929–965, oct 1989. [Online]. Available : https://doi.org/10.1145/76359.76371

[4] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*, 2nd ed. Cambridge, MA : MIT Press, 2018.

[5] F. Bach, "Introduction to machine learning," Cours magistral, 2021, École Normale Supérieur, M2 ICFP. [Online]. Available : https://www.di.ens.fr/~fbach/ML_physique_2021.html

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available : https://doi.org/10.1038/323533a0

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available : https://arxiv.org/abs/1409.1556

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available : http://arxiv.org/abs/1512.03385

[9] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016. [Online]. Available : http://arxiv.org/abs/1602.07261

[10] F. Chollet, "Xception : Deep learning with depthwise separable convolutions," *CoRR*, vol. abs/1610.02357, 2016. [Online]. Available : http://arxiv.org/abs/1610.02357

[11] M. Tan and Q. V. Le, "Efficientnet : Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019. [Online]. Available : http://arxiv.org/abs/1905.11946

[12] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow : concepts, tools, and techniques to build intelligent systems*, 2nd ed. Sebastopol, CA : O'Reilly Media, 2019.

[13] Wikipedia, "Hyperplane separation theorem," 2006, accessed version 23/03/2023. [Online]. Available : https://en.wikipedia.org/wiki/Hyperplane_separation_theorem

A Annexes

A.1 Gradient Descent en détails

On note $J(\theta)$ le risque et θ est vecteur de dimension n. Comme on l'a expliqué dans notre discussion dans la section 2.3, nous souhaitons minimiser le risque au regard des paramètres θ.
Pour ce faire, nous allons revoir ensemble les principes de la dérivation. On commence avec un seul paramètre i.e. $\theta \in \mathbb{R}$

A.1.1 Dimension $n = 1$

Supposons que $J(\theta)$ soit de la forme ci-dessous, que nous prenions deux points (θ_a, J_a), (θ_b, J_b) avec $J_a = J(\theta_a)$, $J_b = J(\theta_b)$, $\theta_b > \theta_a$. Nous voulons savoir si en allant de θ_a à θ_b nous grimpons la courbe de J ou nous la descendons ?

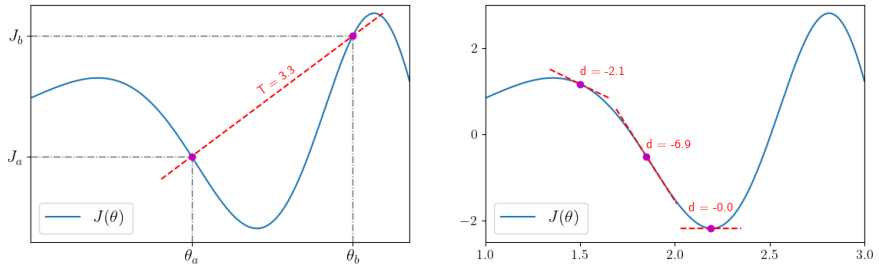


FIGURE 19. Exemple du calcul d'un taux d'accroissement (graphique de gauche) et de la valeur de la dérivée sur plusieurs points (graphique de droite).

Pour répondre à cette question, nous pouvons tracer une droite entre ces deux points et observer si la pente de cette droite est positive ou négative. Cette pente est appelée le taux d'accroissement et est défini comme suit :

$$T = \frac{J_b - J_a}{\theta_b - \theta_a}$$

Si T est positif, cela signifie que nous avons grimpé, alors que si T est négatif, cela indique que nous avons descendu (exactement comme pour le dénivelé d'une route). Cependant, le problème ici est que nous avons ignoré tout ce qui s'est passé entre θ_a et θ_b. Il serait donc plus pratique de savoir si, à chaque point de la courbe, nous montons ou descendons lorsque nous nous déplaçons vers la gauche ou vers la droite.
Pour cela, on peut réduire l'écart entre θ_a et θ_b de sorte qu'il soit à peine discernable (pour qu'il n'y ai pas de variation entre ces deux points).
Redéfinissons $\theta_b = \theta_a + d\theta$ où $d\theta$ est une distance infinitésimal entre θ_a et θ_b, nous avons :

$$T = \frac{J(\theta_a + d\theta) - J(\theta_a)}{d\theta} \coloneqq \frac{dJ}{d\theta}(\theta_a)$$

Nous venons de définir la dérivée. Pour reformuler, avec $d\theta$ très petit, c'est comme si nous avions suffisamment zoomé pour que la courbe ressemble à une droite. Si la pente de cette droite est positive, alors en nous déplaçant vers la droite, on monte. Si on se déplace vers la gauche, on descend et on attendra un minimum [19] (inversement si la dérivé est négative). Le Gradient Descent utilise justement ce concept :

1. On commence par se placer aléatoirement sur la courbe à θ^0
2. On calcul la dérivée de J à θ^0. Si elle est positive alors il faut reculer (i.e. $\theta^1 < \theta^0$), sinon il faut avancer (i.e. $\theta^0 < \theta^1$)

19. Il peut exister plusieurs minimums pour une fonction, nous nous cherchons le minimum global, c'est-à-dire le plus "bas". Si le modèle admet un unique minimum, on dit que le modèle est convexe.

3. Ce qui nous donne la mise à jour :

$$\theta^{k+1} = \theta^k - \eta \frac{dJ}{d\theta}(\theta^k)$$

où le learning rate η définit simplement si le pas entre θ^{k+1} et θ^k est plus ou moins grand.

4. Quand on atteint le minimum, la pente devient nulle (en effet au minimum, on stagne : $J(\theta + d\theta) = J(\theta)$). Ainsi, $\theta^{k+1} = \theta^k$ et l'on dit que l'on a convergé.

A.1.2 Dimension $n > 1$

Le cas pour $\theta \in \mathbb{R}^n$ avec $n > 1$ est très similaire. Restreignons nous à $n = 2 : \theta = (\theta_1, \theta_2)^T \implies J(\theta) = J(\theta_1, \theta_2)$. Savoir comment varie J, revient à savoir comment varie J selon θ_1 **et** selon θ_2. Autrement dit, la variation de J, noté dJ comme précédemment, est la somme des variation de J selon les deux axes θ_1, θ_2 :

$$dJ = \frac{\partial J}{\partial \theta_1} d\theta_1 + \frac{\partial J}{\partial \theta_2} d\theta_2$$

Les ∂ ("d rond") signifient dérivées partielles. La dérivée partielle $\frac{\partial J}{\partial \theta_1}$ correspond simplement à la dérivée de J par rapport à θ_1 en laissant θ_2 constant [20]. Autrement dit, on regarde comment varie J quand on se déplace sur l'axe de θ_1, sans se déplacer sur l'axe θ_2.
C'est précisement ce qui nous intéresse pour le Gradient Descent. On souhaite mettre à jour θ_1 pour atteindre le minimum selon son axe puis faire de même avec θ_2. D'où la mise à jour :

$$\theta_i^{k+1} = \theta_i^k - \eta \frac{\partial J}{\partial \theta_i}(\theta_i^k) \quad \text{avec} \quad i = 1, 2$$

A.2 Régression Logistique

Illustrons la section 2.3 avec le "hello world" de la classification : la régression logistique. Commençons par poser le problème :
Supposons un jeux de données très simple avec 2 features et 2 classes présenté dans la figure 20.

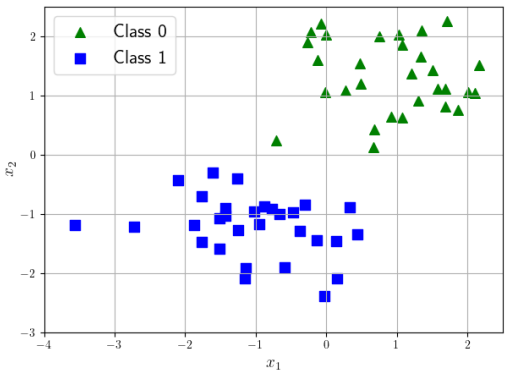


FIGURE 20. Exemple de données 2D linéairement séparable

La première étape est de regarder les données. Ce que nous voyons tout de suite c'est que les données de chaque classes sont bien distinctes et qu'il est possible de les séparer linéairement (i.e. on peut tracer une droite [21] pour séparer les 2 classes). Ce problème est donc idéal pour la régression logistique, qui est une classification linéaire [22].

20. Le vecteur dont l'élément selon l'axe i est la dérivée partielle selon θ_i de J est appelé **gradient** de J d'où le nom "Gradient Descent". Il est généralement noté avec un nabla : $\nabla_\theta J = (\partial_{\theta_1} J, \partial_{\theta_2} J)^T$

21. Dans un cas général où il y a $n \geq 1$ features, on ne parle pas de droite mais d'hyperplan. Pour $n = 1$, la séparation se fait au niveau d'un point, pour $n = 2$, d'une droite, pour $n = 3$, d'un plan, etc. Pour plus de détails, on peut se référer l'article sur Wikipédia [13]

22. Si les données ne sont pas linéairement séparables, on peut quand même utiliser la régression logistique via le rajout de dimensions. On peut donner au modèle de nouvelles features ou des combinaisons des premières tel que $x_3 = x_1^2$, $x_4 = x_1.x_2$, etc.

Pour rappel, une droite correspond à la représentation graphique d'une fonction linéaire f. Ainsi dans notre exemple, on a :

$$f(x_1) = \alpha x_1 + \beta \quad \text{avec} \quad f(x_1) = x_2$$

On peut transformer cela pour nous donner l'équation de la droite :

$$\theta_1.x_1 + \theta_2.x_2 + \theta_3.1 = 0$$

Si l'on prend un point (x'_1, x'_2) et qu'il vérifie l'équation précédente alors ce point appartient la droite. Si $\theta_1.x'_1 + \theta_2.x'_2 + \theta_3.1 < 0$, le point est "en-dessous" de la droite et "au-dessus" si le résultat est strictement positif. On peut donc utiliser cette équation pour obtenir notre classification. Il nous faut juste trouver les bons paramètres $(\theta_1, \theta_2, \theta_3)$ pour que la droite soit située entre les 2 classes.

On peut noter qu'il y a $2 + 1$ paramètres. Si l'on note cette équation d'un point de vue vectoriel, on a :

$$\theta.x = 0 \quad \text{avec} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

La composante 1 dans x est appelé *bias term*. Ainsi, on passe simplement de $x \in \mathbb{R}^n$ à $x \in \mathbb{R}^{n+1}$.

Avec ce qu'on vient de voir, on pourrait utiliser la fonction $g_\theta(x) = \theta.x$ comme fonction d'hypothèse mais cette fonction peut prendre n'importe quelle valeur entre $-\infty$ et $+\infty$ alors que l'on a seulement deux classes (une de valeur 0 et l'autre de valeur 1). C'est pourquoi, on enveloppe cette fonction avec la fonction sigmoïde. C'est fonction nous renvoie uniquement des valeurs entre 0 et 1.

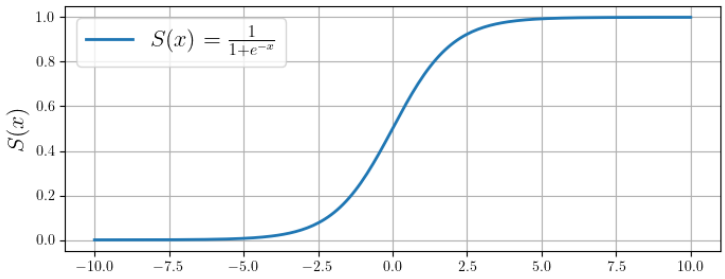


FIGURE 21. Représentation graphique de la fonction sigmoïde

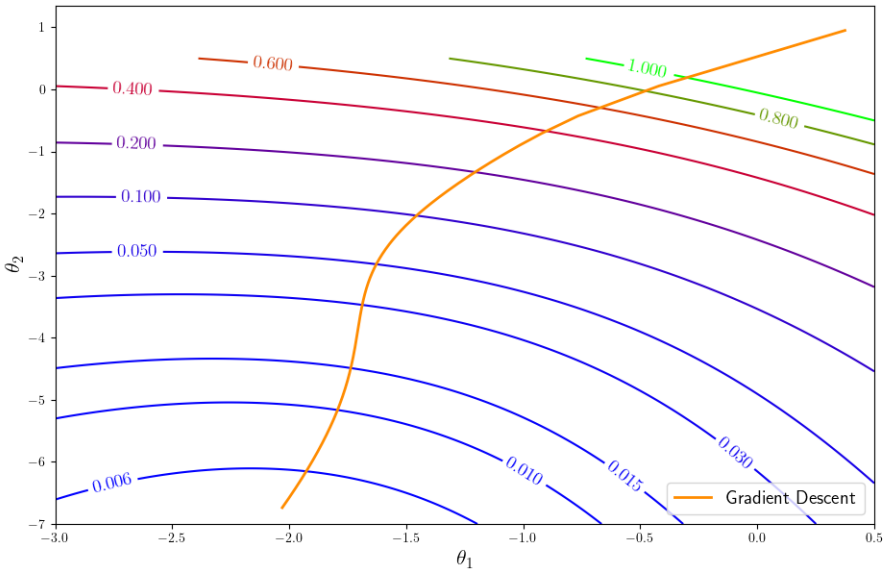
Ce qui nous donne la fonction d'hypothèse :

$$h_\theta(x) = \frac{1}{1 + e^{-\theta.x}}$$

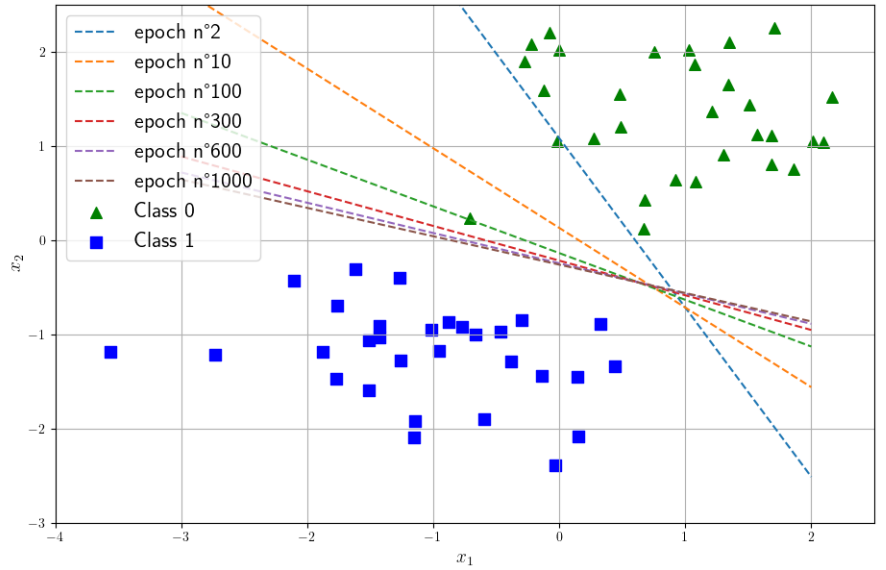
Dans le cas d'une classification, la fonction de perte la plus utilisée est l'entropie de Shanon [4] [12] qui donne le risque suivant :

$$J(\theta) = \frac{1}{m} \sum_i^m -y_i \log(h_\theta(x_i)) + (1 - y_i) \log(1 - h_\theta(x_i))$$

On peut maintenant minimiser ce risque avec le gradient Descent pour obtenir $\tilde{\theta}$.
La droite séparant les deux classes (appelé frontière de décision) obtenue après différentes mises à jour de θ est présentée dans la figure 22.



(A) Gradient Descent sur $J(\theta)$ par rapport à θ_1 et θ_2



(B) Evolution des frontières de décision après plusieurs mises à jour de θ

FIGURE 22. Gradient Descent illustré sur notre jeu de donnée (1 epoch correspond à une mise à jour de θ)