

T-DEV-810

ZOIDBERG2.0

Group N°5

1. Préambule	4
1.1. Contexte du projet	4
1.2. Jeux de données	4
1.3. Outils	5
2. Recherches	6
2.1. Analyse d'images et CNN (Convolutional Neural Networks)	6
2.2. Problématiques rencontrées	7
2.2.1. Taille et équilibrage du dataset	7
2.2.2. Overfitting	10
2.2.3. Entrainement	11
2.2.4. Métriques	13
2.3. Template commun	14
3. Expérimentation	17
3.1. Hyperparamètres	17
3.1.1. Époques	17
3.1.2. Batch size	18
3.1.3. Nombre de KFold	20
3.2. Normalisation	22
3.2.1. Taille de l'image	22
3.2.2. Équilibre du dataset	24
3.3. Augmentation d'image	26
3.3.1. Techniques d'augmentation de données	26
3.3.2. Modes opératoires	26
3.3.3. Padding	28
3.3.4. Zoom	28
3.3.5. Sheer	29
3.3.6. Crop	29
3.3.7. Masque (In painting)	30
3.3.8. Résultats	31
3.4. CNN custom	31
3.4.1. Variation du nombre de filtres	31
3.4.2. Variation de la taille du filtre (kernel size)	33
3.4.3. Variation du nombre de couches de convolution	34
3.4.4. Max Pooling et Average Pooling	37
3.4.5. Variation du dropout	38
3.4.6. Optimizer (autre que 'adam')	39
3.5. Classifieur / Modèle connus	40
3.5.1. RandomForest	41
3.5.1.1. Présentation	41
3.5.1.2. Implémentation	42
3.5.1.3. Résultats	43
3.5.2. AlexNET	44
3.5.2.1. Présentation	44
3.5.2.2. Implémentation	45

3.5.2.3. Résultats	46
3.5.3. ResNET	48
3.5.3.1. Présentation	48
3.5.3.2. Implémentation	49
3.5.3.3. Résultats	50
3.5.4. Transfer Learning et ChestX-ray8	52
4. Élaboration du modèle final	53
5. Classification multiclasse	55
5.1. Changements apportés	55
5.2. Observations	57
6. Conclusion	60

1. Préambule

1.1. Contexte du projet

Le projet s'inscrit dans le cadre d'une détection de pneumonie chez des patients à partir de radios des poumons. Il est effectué par notre groupe constitué de 5 membres :

- Elliott CLAVIER
- Clément MATHÉ
- Nathan PIGNON
- Paul RIPAULT
- Maxime MARTIN

Il s'agit pour nous du premier projet de spécialité sur le sujet de l'intelligence artificielle, et a donc vocation à nous faire pratiquer et découvrir ce domaine dans un cas concret.

Afin de réussir à proposer la meilleure solution possible, nous devons donc passer par une grande phase de recherches, avant d'élaborer une stratégie pour pouvoir étudier et améliorer nos résultats dans le cadre d'une démarche qui se veut scientifique. Notre démarche a mis en jeu plusieurs concepts, outils et méthodologies du *Machine Learning* et parfois plus précisément du *Deep Learning*.

1.2. Jeux de données

Sont mis à notre disposition trois datasets différents :

- Un jeu d'entraînement, servant à entraîner notre modèle en ajustant les paramètres de ce dernier afin de minimiser l'erreur sur les données d'entraînement.
- Un jeu de validation, qui est utilisé pour évaluer les performances du modèle pendant la phase d'entraînement et pour ajuster les hyperparamètres.
- Enfin, le jeu de test est utilisé pour évaluer les performances finales du modèle après l'entraînement. Ce jeu de données est réservé à cette étape pour éviter le biais d'avoir utilisé les données de test pendant l'entraînement ou la validation.

Chacun des jeux de données est constitué d'un ensemble d'images de radios des poumons. Certaines de ces radios présentent des poumons de patients indemnes et d'autres de patients atteints de pneumonie virale ou bactérienne.

Concrètement, d'un point de vue de la donnée, une image est un tableau de pixels. Dans notre cas, ces images sont encodées en nuances de gris, on a donc un tableau contenant un seul tableau de pixel. Une image de radio aura la forme $(H, W, 1)$, avec H la hauteur de l'image, W la largeur et 1 pour indiquer que nous travaillons sur un seul canal de couleur: les nuances de gris (on aurait eu 3 si nos images étaient encodées en RGB). **La forme de notre jeu de données est donc $(N, H, W, 1)$, avec N le nombre d'images de notre jeu de données.**

L'objectif est de construire un environnement qui permettent au mieux de prédire sur une image de radio si le patient présente un cas de pneumonie ou non. L'extension de cas d'étude serait de pouvoir

prédir si le patient présente un cas de pneumonie virale, bactérienne ou si simplement le patient n'est pas malade.

1.3. Outils

Nous utilisons plusieurs outils de calculs, de visualisation et de création de modèles d'apprentissage automatique, embarqués dans un environnement Anaconda:

- **Python** : langage de programmation utilisé pour le développement du projet.
- **Jupyter Notebook** : environnement de développement interactif permettant de créer et d'exécuter du code Python ainsi que d'afficher des résultats sous forme de graphiques ou de texte.
- **NumPy** : bibliothèque Python pour les calculs numériques et la manipulation de données.
- **Matplotlib** : bibliothèque Python pour la création de graphiques en 2D.
- **Seaborn** : bibliothèque Python pour la création de graphiques en 2D avec une interface plus haut niveau que Matplotlib.
- **Scikit-learn** : bibliothèque Python pour l'apprentissage automatique, incluant des outils pour la classification, la régression, le clustering, la sélection de modèles, la préparation de données, etc.
- **TensorFlow** : bibliothèque Python pour le développement et le déploiement de modèles d'apprentissage automatique à grande échelle.
- **Keras** : bibliothèque Python pour la création de réseaux de neurones profonds.
- **OpenCV** : bibliothèque Python pour le traitement d'images et de vidéos.
- **DataSpell** : IDE de JetBrains utilisé lors du projet
- **Cuda ToolKit** : permet l'exécution de calculs intensifs sur les GPU

2. Recherches

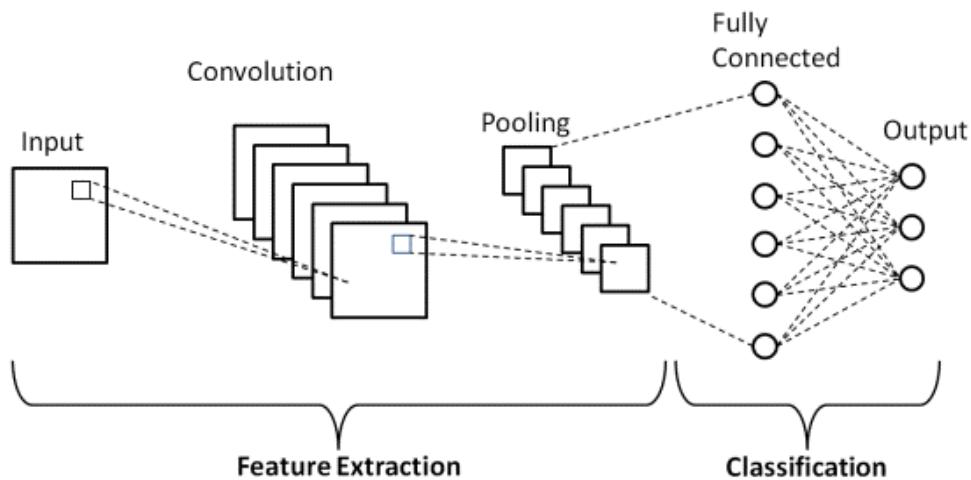
Ce projet étant notre premier dans le domaine de l'élaboration d'un modèle de reconnaissance d'images, une démarche de recherche a dû être mise en œuvre par l'équipe. Nous nous sommes tout d'abord tournés vers les réseaux de neurones et plus précisément vers les réseaux de neurones à convolutions, qui sont aujourd'hui la meilleure solution pour la classification d'images.

2.1. Analyse d'images et CNN (Convolutional Neural Networks)

Le *deep learning* est une branche de l'intelligence artificielle qui permet d'apprendre à partir de données. Il s'agit d'une méthode d'apprentissage automatique qui utilise des réseaux de neurones artificiels pour découvrir des caractéristiques dans les données et effectuer des prédictions ou des classifications. Les réseaux de neurones sont des modèles mathématiques inspirés du cerveau humain, qui sont conçus pour apprendre à partir de données en ajustant leurs paramètres pour minimiser une fonction de coût. Au sens du deep learning, on trouve un type de réseau de neurones qui va nous intéresser dans le cadre du projet : le CNN.

Un CNN, ou Convolutional Neural Network, est une architecture de réseau de neurones particulièrement adaptée à la reconnaissance d'images. Il est construit à partir de différentes couches, chacune remplissant un rôle spécifique dans le traitement des images.

Keras et Tensorflow sont deux des bibliothèques les plus utilisées pour construire des CNN en Python. Keras est une bibliothèque d'apprentissage en profondeur haut niveau qui permet de construire des modèles de manière rapide et efficace. Tensorflow, quant à lui, est une bibliothèque d'apprentissage en profondeur plus bas niveau qui offre un contrôle plus fin sur le modèle et le processus d'apprentissage.

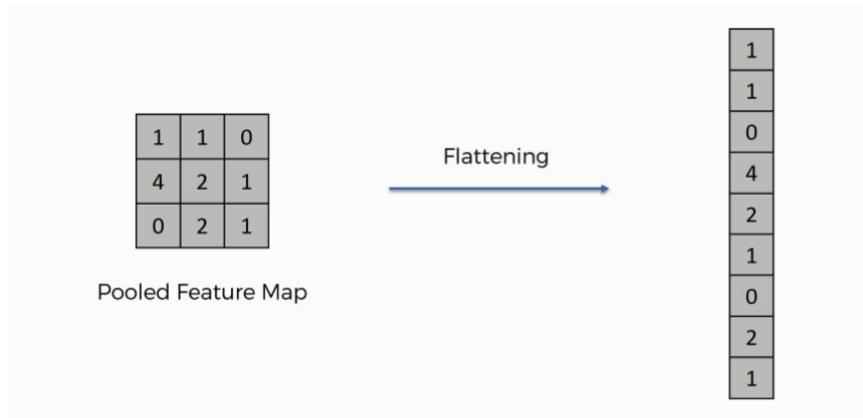


Le processus de construction d'un CNN commence par la création d'un modèle **séquentiel**. Cette architecture permet de définir les différentes couches qui composent le réseau. Les **couches de convolution** sont les plus importantes et leur fonctionnement est basé sur l'utilisation de **filtres**. Ces

filtres sont des matrices de poids qui sont appliquées à l'image d'entrée afin d'extraire des caractéristiques. **La taille de kernel**, qui correspond à la taille du filtre, est un hyperparamètre important à régler pour obtenir les meilleures performances.

Après chaque couche de convolution, il est généralement recommandé d'ajouter une couche de **pooling** pour réduire la taille de la représentation de l'image tout en conservant les caractéristiques importantes. Le pooling peut être effectué selon différents critères, tels que la **moyenne** ou la **valeur maximale**.

Le CNN est ensuite composé de plusieurs couches cachées, qui permettent de relier les caractéristiques extraites aux différentes classes d'images. Ces couches peuvent être constituées de différentes fonctions d'activation, telles que *ReLU* ou *sigmoid*.



La liaison entre les couches de convolutions et les couches de neurones cachées se fait grâce au **flattening** qui permet d'aplatir un tenseur (un vecteur comme une image par exemple) en une seule dimension, sans quoi les couches cachées ne pourraient pas manipuler la donnée. En effet, dans le cas d'un CNN les données d'entrées sont des images en 3D (longueur, largeur, canaux de couleurs).

Enfin, le modèle doit être compilé avant d'être entraîné. Cela implique de définir une fonction de perte (*loss function*) qui mesure l'écart entre les prédictions du modèle et les véritables étiquettes d'images, une fonction d'optimisation qui ajuste les poids du modèle pendant l'apprentissage, et une ou plusieurs métriques qui évaluent les performances du modèle.

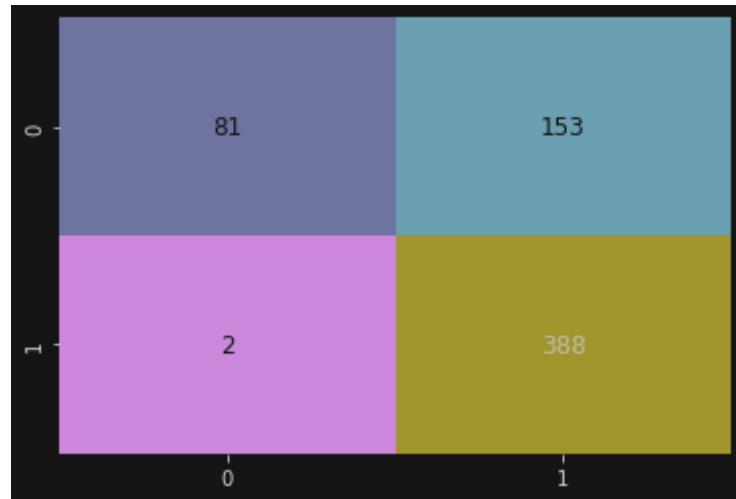
2.2. Problématiques rencontrées

Lors de la création de notre premier CNN, plusieurs problématiques ont fait surface. Ces problématiques étaient liées à un manque de compréhension des différentes techniques utilisées pour entraîner et prédire un modèle, mais aussi à la spécificité du jeu de données qui présente des particularités pouvant biaiser les choix faits par notre modèle.

2.2.1. Taille et équilibrage du dataset

En partant de la situation initiale, l'entraînement du CNN avait tendance à prédire ses résultats en faveur du label "Pneumonie". Pourtant, les données étaient normalisées: on transforme les pixels des images d'une échelle de 0 à 255 vers une échelle de 0 à 1. La normalisation permet de réduire la variance des données en les mettant à la même échelle, afin de faciliter l'apprentissage du modèle et

d'améliorer sa performance. Les images étaient également toutes redimensionnées sous format 224x224.

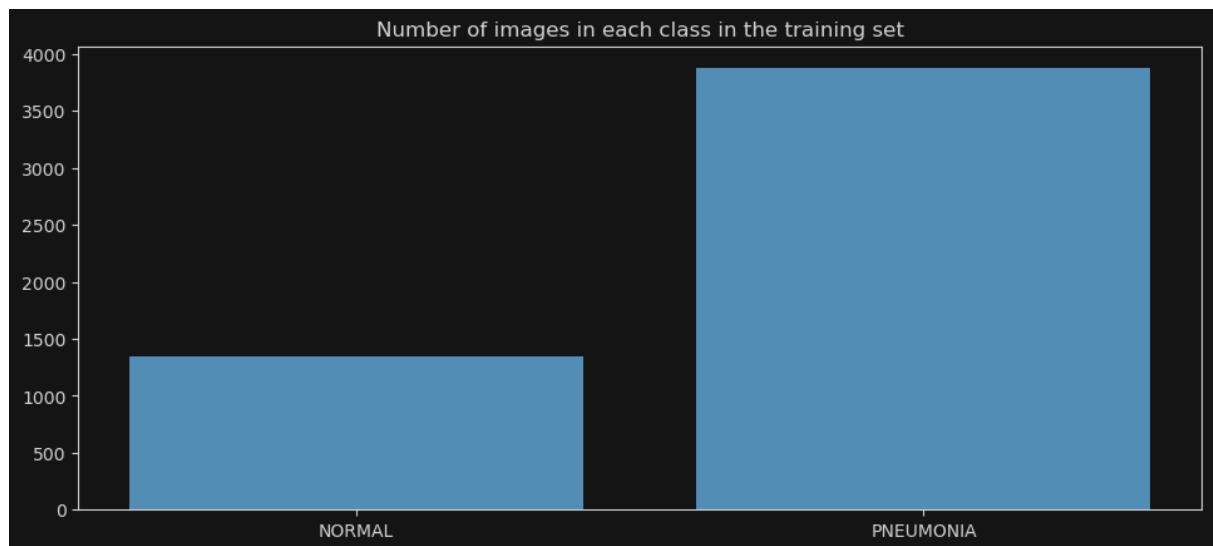


Matrice de confusion qui fait suite aux premiers entraînements sur le jeu de données

Si 0 est le label “Normale” et 1 le label “Pneumonie”, et si l’abscisse représente les valeurs prédites et l’ordonnée représente les valeurs attendues, alors on remarque un nombre de faux positifs très important, pour un nombre suspectif de faux négatifs. **Un faux positif** correspond à une radio normale qui a été prédite comme présentant une pneumonie. **Un faux négatif** correspond à une radio d’un patient présentant une pneumonie mais que le modèle aurait reconnu comme indemne.

Si un faux positif reste moins dangereux qu’un faux négatif (on préfère prédire la présence de la pneumonie plutôt que l’ignorer), le taux de faux positif reste anormalement élevé et donne un score de **précision de 75%** avec un **recall de 35%** pour le label normal, ce qui est très faible.

Des recherches ont permis de montrer que par défaut, les jeux de données étaient fortement déséquilibrés et que ce déséquilibre influence la manière dont le modèle va apprendre.

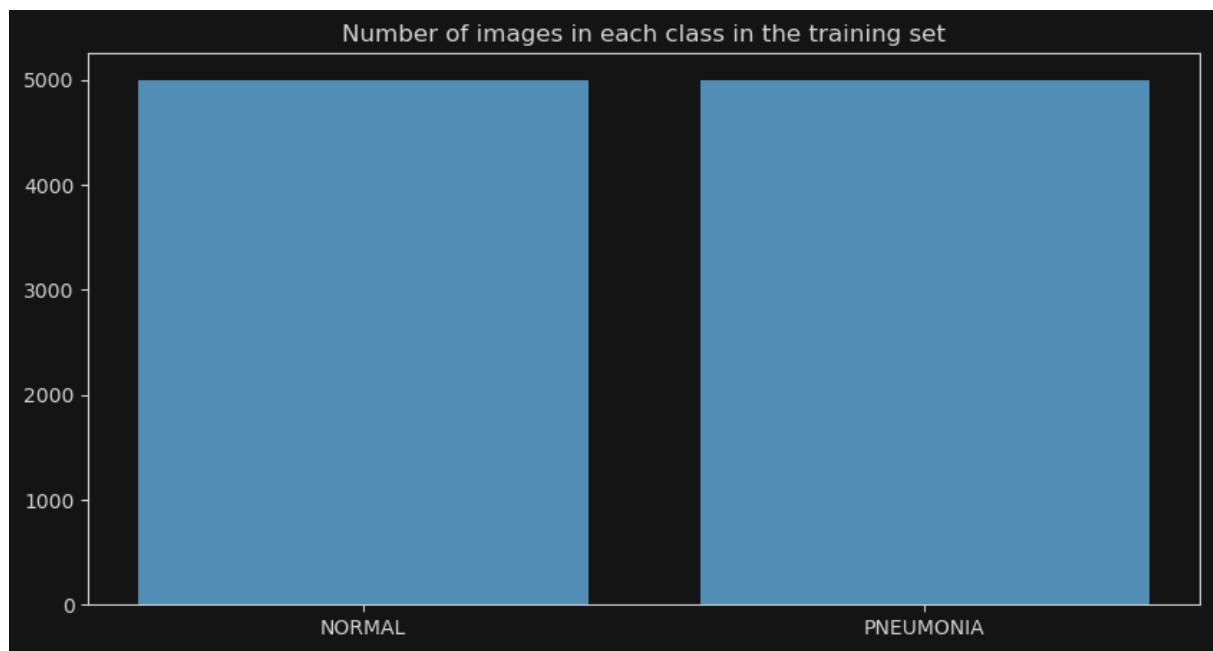


Après avoir détecté ce déséquilibre et avoir compris l'importance d'avoir un jeu de données équilibré par classe en entrée de l'entraînement du modèle, nous nous sommes penchés vers **l'augmentation de données**.



Dans l'exemple ci-dessus, des images ont été inversées horizontalement et verticalement. A partir d'une image, on a générée deux nouvelles images.

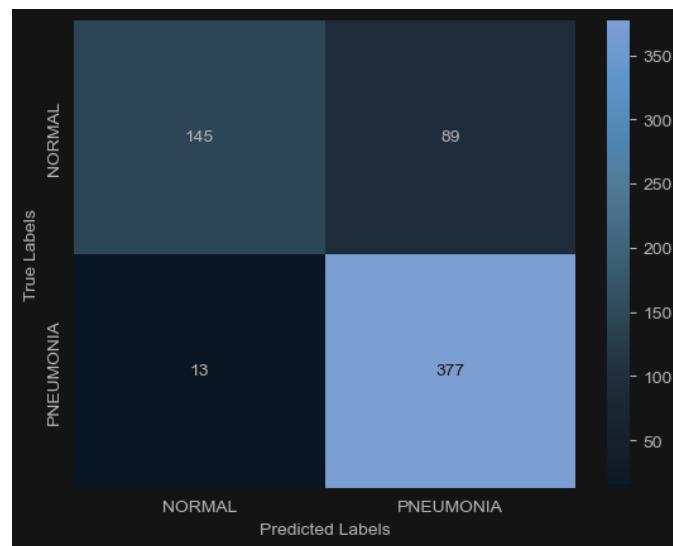
L'augmentation des données est une technique qui permet de multiplier le nombre d'images à notre disposition en générant de nouvelles images à partir du jeu de données existant. Cette augmentation est réalisée en appliquant des transformations aléatoires aux images existantes, telles que des rotations, des zooms, des translations, des retournements et des changements de luminosité. En conséquence, le modèle voit des images similaires à celles qu'il a déjà vu, mais avec des différences subtiles, ce qui lui permet d'apprendre à reconnaître les objets indépendamment de leur position, leur orientation ou leur luminosité.



En utilisant l'augmentation d'images, on a alors générée 5000 nouvelles images de radios de patients indemnes et 5000 nouvelles images de patients atteints de pneumonie. Pour notre cas l'augmentation d'image a permis d'équilibrer la présence des deux **labels** dans notre jeu de données, mais aussi de régler un deuxième problème récurrent des réseaux de neurones: l'*overfitting*.

2.2.2. Overfitting

Après avoir équilibré le jeu de données d'entraînement, le modèle a permis d'obtenir les résultats de prédictions suivants:



Matrice de confusion obtenue après équilibrage du jeu de données d'entraînement

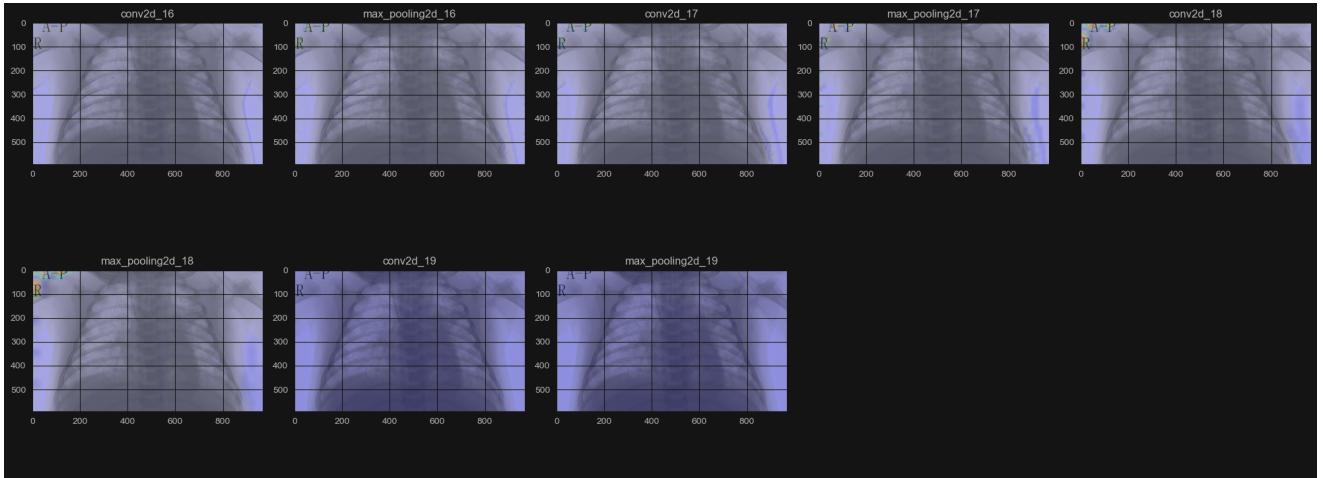
On observe des résultats plus cohérents avec **un score de précision de 84%** et **un recall de 62% pour la classe “Normale”**, c'est-à-dire une amélioration du nombre de faux positifs obtenus.

Puis c'est alors posé la question de savoir sur quelles parties de l'image le réseau de neurones s'est basé pour réaliser ses prédictions. Pour se faire, il existe une technique de visualisation des choix du réseaux de neurones à convolutions sous forme de cartes de chaleur appelée *Grad-CAM*.

Le *Grad-CAM* (*Gradient-weighted Class Activation Mapping*) est une technique de visualisation qui permet de localiser les régions d'une image qui ont contribué le plus à la décision d'un modèle de réseau de neurones convolutif (Convolutional Neural Network ou CNN).

Plus précisément, le *Grad-CAM* calcule un score de classe pour chaque pixel de l'image en utilisant les gradients de sortie de la dernière couche de convolution du modèle par rapport à la sortie de la couche de classification. Ces scores de classe sont ensuite combinés en utilisant une pondération pour produire une carte de chaleur (heatmap) qui indique les régions de l'image qui ont le plus contribué à la décision de classification.

L'utilisation de cette technique nous a alors permis de savoir si notre modèle faisait des prédictions justifiées sur des critères valables des radios passées en entrée. On a alors affiché une carte de chaleur par couche de convolution ou pooling; le résultat est le suivant:



On peut observer plus particulièrement sur les couches *max_pooling_17*, *conv2d_18* et *max_pooling_18* que notre réseaux de neurones semblent orienter ses décisions sur les lettres présentes en haut à gauche de chacune des radios, et qui indiquent le vrai résultat de la radio étiqueté par l'entité médicale. Notre réseau de neurones a choisi la solution de facilité et s'est adapté à notre jeu de données qui comporte des images présentant toujours ces lettres, c'est l'*overfitting*.

L'overfitting, ou surapprentissage, se produit lorsqu'un modèle d'apprentissage automatique est trop complexe par rapport aux données qu'il doit apprendre. Cela se traduit par une performance élevée sur les données d'entraînement, mais une performance médiocre sur les données de test ou de validation. En effet, on peut opposer le score de précision sur le jeu de données de test de 84% contre **le score de précision à l'entraînement de presque 99%**.

Pour régler ce problème de surapprentissage, plusieurs solutions ont été apporté:

- des paramètres d'augmentation des images avec des portées légèrement plus importantes pour éviter la présence trop récurrente de *pattern*.
- l'utilisation de ***drop out*** sur les couches de neurones cachées, qui permet de désactiver aléatoirement un certain pourcentage de neurones pendant l'entraînement.

2.2.3. Entrainement

Afin de faire valider la cohérence de notre modèle, nous avons ensuite mis en place de la *cross validation*.

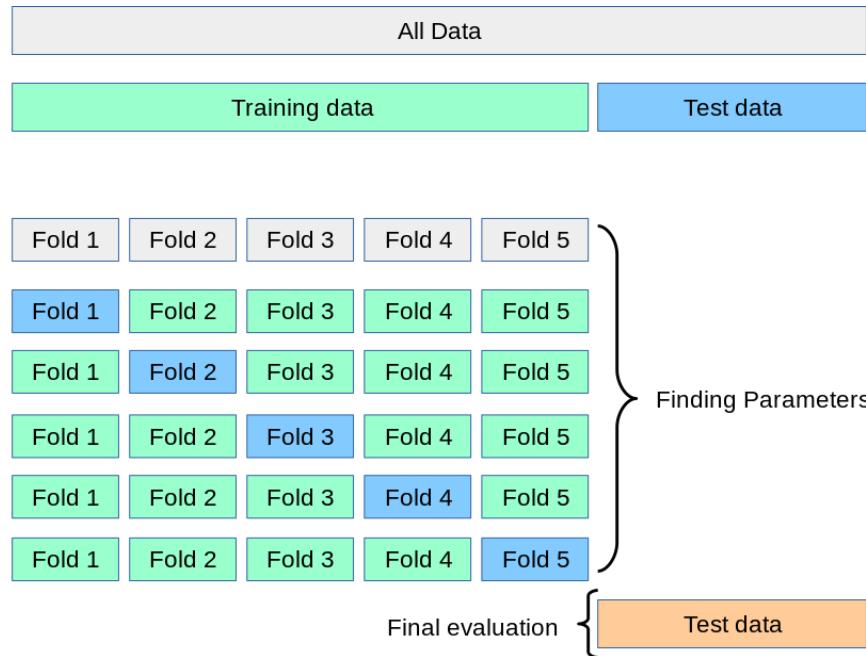


Schéma explicitant le fonctionnement de la cross validation

La cross-validation est une méthodologie consistant à diviser l'ensemble de données en plusieurs sous-ensembles appelés **folds**. Le modèle est alors entraîné sur plusieurs combinaisons d'ensembles de *folds*, où chaque fold sert à la fois d'ensemble de test et d'ensemble de validation. Pour chaque combinaison de *folds*, le modèle est entraîné sur les *folds* d'entraînement et évalué sur les *folds* de validation, afin de mesurer la performance moyenne du modèle sur les différentes combinaisons de *folds*.

Cela permet d'évaluer la performance du modèle de manière plus robuste, car elle permet d'utiliser l'ensemble de données disponible de manière plus efficace et de détecter les problèmes de sur apprentissage (*overfitting*) ou de sous apprentissage (*underfitting*) en comparant les scores d'entraînement et de validation obtenu par *folds*.

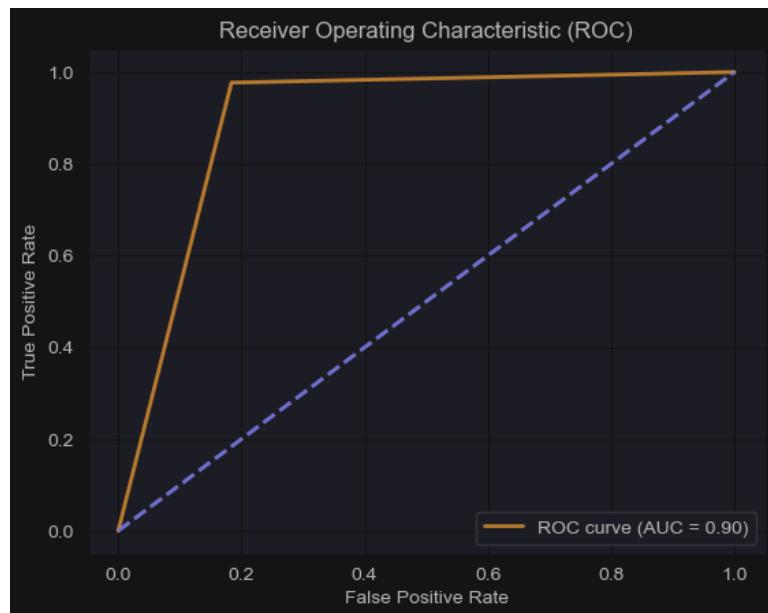
Un manque de compréhension de la méthodologie nous a conduit à une erreur dans sa mise en place. En effet, après *cross validation*, nous choisissons le modèle ayant obtenu le meilleur score de précision sur le jeu de données d'entraînement et de validation, puis nous le soumettons au jeu de données de test pour réaliser des prédictions. Comme le schéma ci-dessus l'indique, la cross validation sert seulement à valider la cohérence du modèle. Suite à une cross validation jugée positive grâce à des métriques, il faut ensuite réentraîner le modèle avec **l'ensemble du jeu de données d'entraînement** et avec **les mêmes hyperparamètres**. Une simple correction a permis de respecter les règles de cette méthodologie, mais aussi d'exploiter au maximum l'ensemble du jeu de données d'entraînement pour obtenir un modèle performant.

2.2.4. Métriques



Visualisation de l'évolution de la précision et de la perte lors de l'évaluation du modèle lors de l'entraînement par le jeu de donnée de données de validation à chaque époque

Comme nous l'avons vu précédemment, nous avons progressivement mis en jeu plusieurs métriques pour déterminer la performance de notre modèle. D'abord les scores de précisions, de *recall* et le *f1 score*, puis l'utilisation du *Grad-CAM* et enfin la *cross validation* qui permet d'obtenir des scores sur un jeu de données de validation. Il est important de noter que nous avons par la suite ajouté une visualisation de ces métriques.



Exemple d'une courbe ROC avec une AUC de 0.90

Nous avons également ajouté une nouvelle métrique: **l'AUC qui découle de la courbe ROC. La courbe ROC** (Receiver Operating Characteristic) est une représentation graphique de la performance d'un

modèle de classification binaire. Elle est tracée en faisant varier le seuil de décision du modèle et en calculant le taux de vrais positifs et le taux de faux positifs pour chaque seuil de décision.

L'aire sous la courbe ROC (**AUC**, pour Area Under the Curve) est une mesure numérique de la performance du modèle, qui représente la probabilité que le modèle classe une observation positive au hasard plus haut que celle négative. Plus l'AUC est proche de 1, meilleure est la performance du modèle, tandis qu'une AUC proche de 0,5 indique une performance aléatoire.

2.3. Template commun

Après avoir réglé l'ensemble des problématiques rencontrées lors de l'entraînement et le test de notre modèle, nous avons regroupé les différentes étapes permettant d'obtenir un résultat complet pour notre cas d'étude. Parmi ces étapes, on retrouve le chargement, l'augmentation, l'équilibre et la normalisation des jeux de données d'entraînement, de test et de validation. On retrouve également les étapes de création du modèle, de cross validation, d'entraînement et d'évaluation du modèle avec le calcul de l'ensemble des métriques qui nous ont été utiles lors de la phase de recherche.

Notre CNN possède l'architecture suivante:

```
model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(3,3), input_shape=(img_width, img_height, 1),
activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())

model.add(Dense(256, activation="relu"))
model.add(Dropout(0.5))

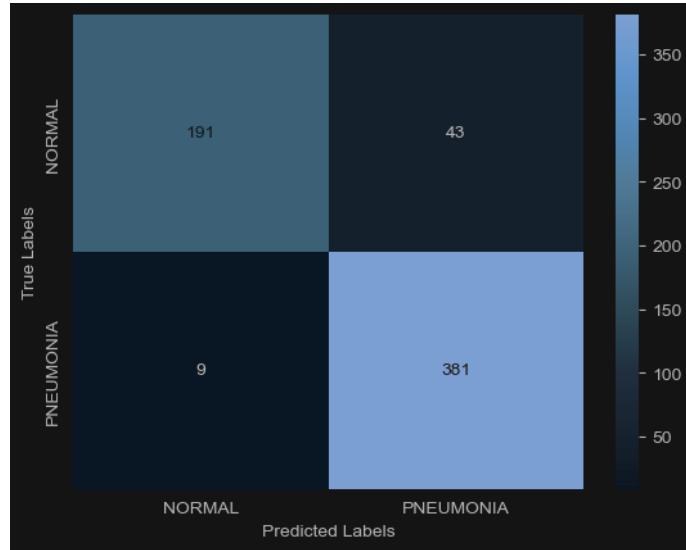
model.add(Dense(128, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(1, activation="sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam", metrics=metrics)
```

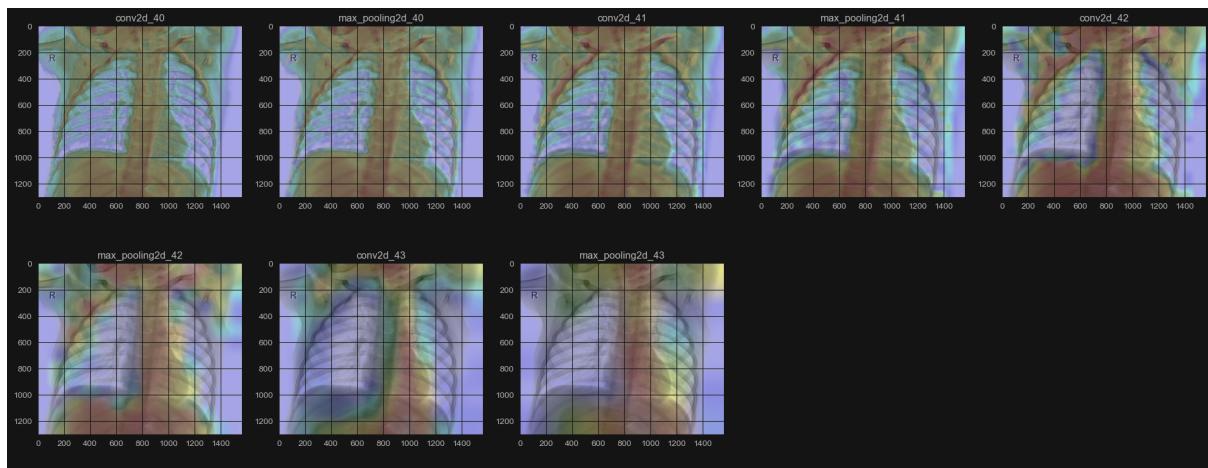
Les paramètres et hyperparamètres principaux sont les suivants:

- **taille d'image** de 128 par 128
- **batch size** de 64
- **nombre d'époques** de 7

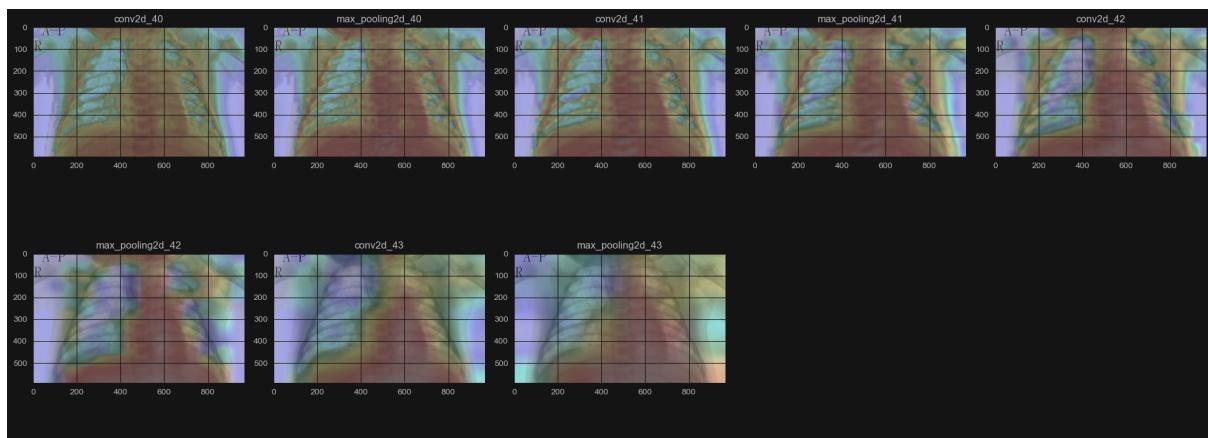


Matrice de confusion obtenue sur les données du jeu de test suite à l'entraînement du modèle de notre template

Les résultats suite à l'évaluation du modèle entraîné indique un score de précision de **92%**, avec un nombre **très faible de faux négatifs**.



Heatmaps obtenues par notre modèle sur une image de test d'un patient indemne



Heatmaps obtenues par notre modèle sur une image de test d'un patient atteint de pneumonie

Les heatmaps obtenues suite à l'évaluation du modèle sur le jeu de données de test indiquent également que les critères sélectionnés par le modèle pour réaliser ses prédictions semblent valables et ne présentent pas de piste permettant d'indiquer de *l'overfitting* comme expérimenter plus tôt.

3. Expérimentation

Une fois que nous avons achevé la phase de recherche et la conception de notre template commun, nous sommes maintenant prêts à passer à la partie expérimentale de notre projet. Cette étape vise à affiner notre modèle initial en modifiant certains paramètres afin de voir si nous pouvons améliorer les performances de notre algorithme.

Pour cela, nous avons mis en place un document Excel contenant tous les leviers sur lesquels nous pouvons jouer. Le but premier étant d'optimiser notre modèle, cette phase d'expérimentation nous permet également de comprendre un peu plus comment les différents paramètres influencent la capacité de notre modèle à détecter avec précision la présence de pneumonie.

3.1. Hyperparamètres

Les hyperparamètres sont des paramètres qui ne sont pas appris par le modèle lui-même, mais qui doivent-être fixés avant l'entraînement. Les performances du modèle dépendent en partie des valeurs des hyperparamètres choisis, et trouver les valeurs optimales peut faire une grande différence dans les performances du modèle.

3.1.1. Époques

Le nombre d'époques idéal est important pour déterminer le moment où le modèle de réseau de neurones atteint sa meilleure performance en termes de précision et de capacité à généraliser sur de nouvelles données. Si le nombre d'époques est trop faible, le modèle peut ne pas être suffisamment entraîné et manquer des informations importantes dans les données, conduisant à une performance médiocre sur les données de test. En revanche, si le nombre d'époques est trop élevé, le modèle risque de sur-apprendre (overfitting) les données d'entraînement, ce qui peut entraîner une performance dégradée sur les données de test.

Dans le contexte de nos expérimentations, nous avons opté pour l'utilisation de *GridSearchCV* de la librairie **Scikit-Learn**. Cet objet nous permet de renseigner une liste de valeurs d'époques à tester, afin d'obtenir à la suite de l'entraînement le meilleur d'entre eux. Pour choisir le meilleur, on comparera les valeurs de *mean_score* (moyenne des scores issus de la cross validation) et de *std_score* (mesure de l'écart-type des performances de validation croisée).

```
epochs = [5, 10, 20]

# Wrap your create_model function with KerasClassifier
wrapped_model = KerasClassifier(build_fn=create_model, verbose=1)

# Define the range of epochs
param_grid = {
    'epochs': epochs,
}

# Initialize the GridSearchCV object with your wrapped_model, param_grid, and the
desired number of folds
```

```
grid_search = GridSearchCV(estimator=wrapped_model, param_grid=param_grid, cv=n_splits,  
return_train_score=True)
```

Un premier entraînement a donc comparé les valeurs de 5, 10 et 20 époques, avant de conclure à un meilleur résultat **pour la valeur de 10**.

```
0.9294102191925049, 0.02870926863569606 with: {'epochs': 5}  
0.9301089882850647, 0.026130167427028204 with: {'epochs': 10}  
0.9251182079315186, 0.017580123675320067 with: {'epochs': 20}
```

À ce stade, afin d'améliorer la valeur optimale du nombre d'époques, nous sommes partis du résultat précédent pour affiner la recherche à l'aide de la GridSearchCV. Cette fois, nous cherchons le meilleur nombre d'époques entre 7, 10 et 13.

```
epochs = [7, 10, 13]
```

Le résultat obtenu indique que le nombre optimal serait de 13 époques :

```
Best : 0.9306090831756592, using {'epochs': 13}  
0.9237203955650329, 0.0168399786093972 with: {'epochs': 7}  
0.9082426190376282, 0.036496024431784006 with: {'epochs': 10}  
0.9306090831756592, 0.021853183450978794 with: {'epochs': 13}
```

3.1.2. Batch size

Le *batch size* correspond au nombre d'exemples que le modèle va utiliser pour calculer une approximation de la fonction de coût (ou perte) et mettre à jour les poids du réseau de neurones.

Ainsi, un **batch size** plus élevé signifie que le modèle mettra à jour les poids moins fréquemment, ce qui peut rendre l'entraînement plus rapide, mais également moins précis. À l'inverse, un **batch size** plus faible signifie que le modèle mettra à jour les poids plus fréquemment, ce qui peut rendre l'entraînement plus lent, mais également plus précis.

Il est donc important de chercher la valeur idéale du *batch size* pour chaque modèle que l'on entraîne. En effet, un *batch size* trop petit peut entraîner une convergence lente ou un surapprentissage, tandis qu'un *batch size* trop grand peut entraîner une convergence rapide, mais avec un risque de ne pas généraliser correctement à de nouveaux exemples.

La recherche de la valeur optimale du *batch size* est donc un processus important pour s'assurer que le modèle converge rapidement et avec précision, tout en généralisant bien aux nouvelles données.

Pour s'y employer, nous allons d'abord entraîner notre modèle avec une valeur de *batch size* significativement plus faible qu'avec nos paramètres communs, puis nous ferons l'inverse pour observer l'impact de l'évolution de la valeur de cet hyperparamètre.

Ainsi donc, nous passons d'un *batch size* de 64 à 16, puis à 128 pour un second entraînement. La mesure de la précision des modèles ainsi entraînés donnent les valeurs suivantes, respectivement : 0,91 et 0,89 pour 16 de *batch size* puis 128. On n'observe donc dans le premier cas pas d'impact significatif sur le résultat, au contraire du deuxième qui voit la précision de son modèle diminuer.

Pour approfondir ces résultats, nous décidons une nouvelle fois de recourir à la GridSearchCV, mais cette fois en cherchant la combinaison optimale de valeur de *batch_size* et d'époques :

```
# Define the range of epochs and batch sizes
epochs = [5, 10, 15]
batch_size = [16, 32, 64]

param_grid = {
    'epochs': epochs,
    'batch_size': batch_size
}

datagen = ImageDataGenerator()

# Wrap your create_model function with KerasClassifier
wrapped_model = KerasClassifier(build_fn=create_model, verbose=1)

# Initialize the GridSearchCV object with your wrapped_model, param_grid, and the
# desired number of folds
grid_search = GridSearchCV(estimator=wrapped_model, param_grid=param_grid, cv=n_splits,
                           return_train_score=True)
```

Parmis toutes les combinaisons testées, on se base sur le *mean_score* :

```
Best score: 0.93 using 10 epochs and 16 batch size
Accuracy: 0.89 with: {'batch_size': 16, 'epochs': 5}
Accuracy: 0.93 with: {'batch_size': 16, 'epochs': 10}
Accuracy: 0.93 with: {'batch_size': 16, 'epochs': 15}
Accuracy: 0.89 with: {'batch_size': 32, 'epochs': 5}
Accuracy: 0.9 with: {'batch_size': 32, 'epochs': 10}
Accuracy: 0.91 with: {'batch_size': 32, 'epochs': 15}
Accuracy: 0.92 with: {'batch_size': 64, 'epochs': 5}
Accuracy: 0.93 with: {'batch_size': 64, 'epochs': 10}
Accuracy: 0.91 with: {'batch_size': 64, 'epochs': 15}
```

Une combinaison optimale de 10 époques et 16 *batch_size* semble se démarquer, mais certaines autres combinaisons semblent être équivalentes dans les résultats obtenus :



Précision obtenue sur le jeu de validation par combinaison de batch_size et d'époques

3.1.3. Nombre de KFold

Nous utilisons la validation croisée pour évaluer la performance du modèle sur des données pendant l'entraînement, ce qui permet de mieux estimer sa capacité à généraliser sur de nouvelles données.

Les k-folds sont simplement une variation de la validation croisée où le jeu de données est divisé en k parties égales. Le modèle est alors entraîné k fois, chacune des k parties étant utilisée une fois comme validation set et k-1 fois comme training set. La performance du modèle est finalement calculée comme la moyenne des performances obtenues sur les k-itérations.

Il est important de choisir le nombre optimal de folds pour au moins deux raisons :

- Si le nombre de folds est trop faible, cela peut conduire à une estimation biaisée de la performance du modèle, car il sera évalué sur un nombre trop restreint de données.
- Si le nombre de folds est trop élevé, cela peut rendre l'entraînement du modèle très coûteux en temps et en ressources, car il devra être entraîné k fois au lieu d'une seule fois.

Afin d'étudier la valeur la plus appropriée dans le cadre de notre projet, nous avons entrepris d'entraîner notre modèle commun plusieurs fois, avec un nombre de folds différents à chaque fois. Nous avons ensuite focalisé notre attention sur l'évolution des métriques d'accuracy et de loss entre les folds et de leur cohérence, le temps d'entraînement, mais aussi sur l'AUC (Area Under the Curve).

Avec un nombre de folds définis à 10 pour le modèle commun, nous avions donc obtenu un score AUC de 0.90, l'entraînement a duré et cette courbe semble indiquer une cohérence des résultats obtenus entre chaque folds, pour les métriques d'accuracy et de loss.

Ces résultats préliminaires vont donc servir de comparatif dans notre recherche du nombre optimal de folds.

Nous entraînons le même modèle, mais cette fois avec 5 folds, puis 15 et 20 :

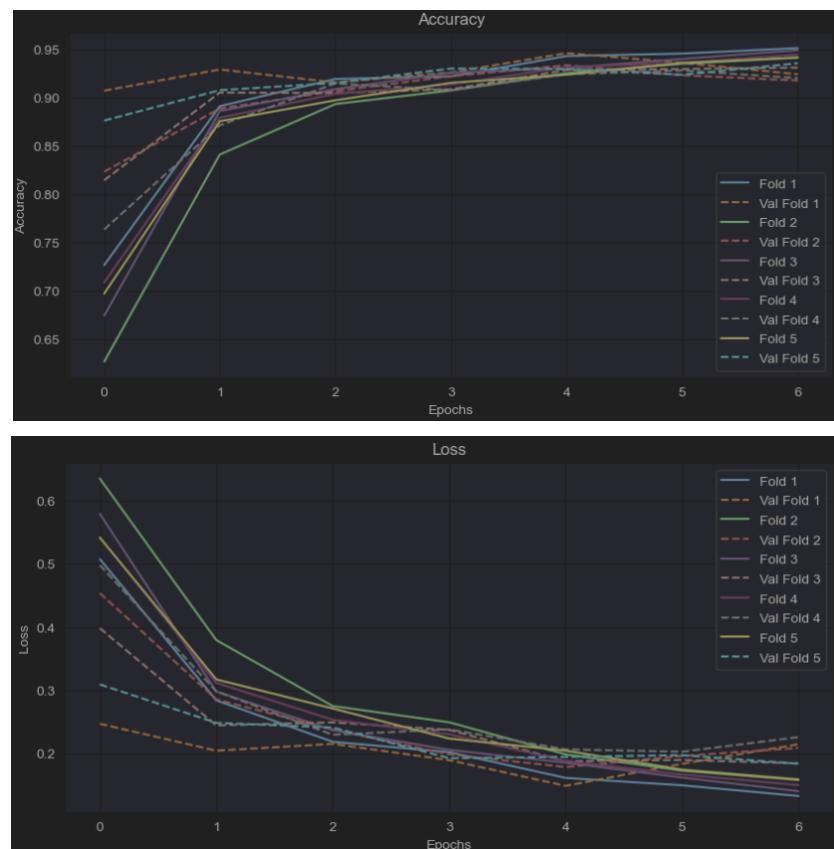
```
# Define the number of folds for cross-validation
n_splits = 5

# Initialize KFold with the number of folds
kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
```

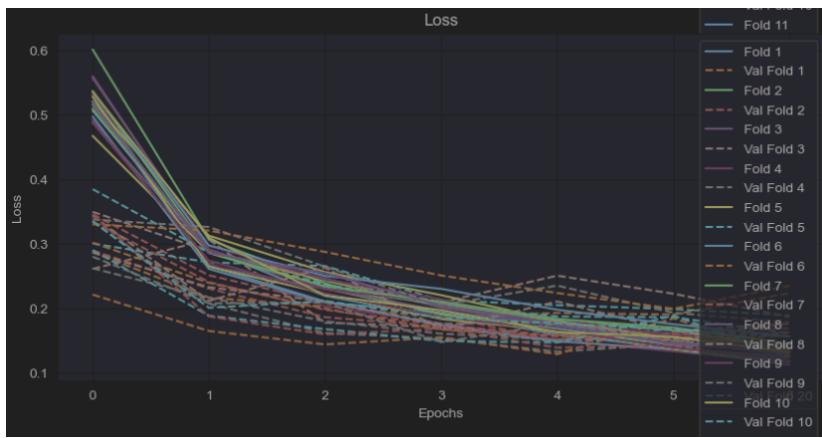
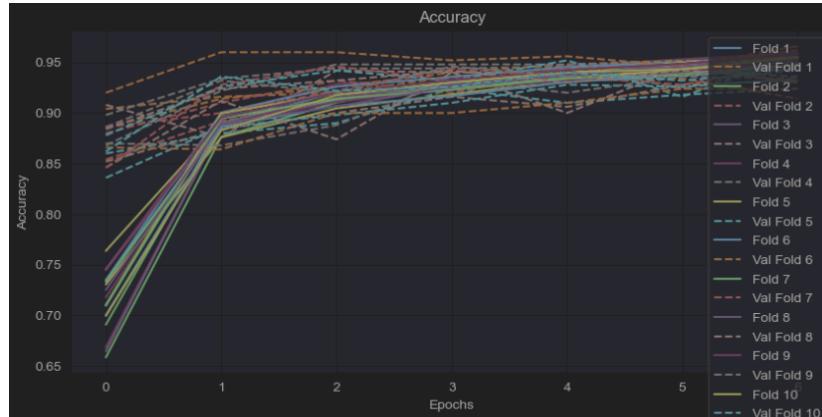
Puis nous obtenons les résultats suivants :

Nombre de folds	5	10	15	20
AUC	0.91	0.90	0.90	0.89
Temps d'entraînement	2mn 39s	6mn 54s	10mn 29s	14mn 05s

Nous traçons également pour chaque folds la courbe des valeurs de précision et de coût en fonction des époques de l'entraînement, pour vérifier la cohérence entre ces derniers. Nous obtenons des courbes satisfaisantes comme celle-ci dans tous les cas.



Excepté lorsque nous utilisons 20 folds, nous obtenons alors plus de variance entre ces derniers, comme le montre ce dernier graphique, par l'écart entre les différentes courbes.



La conclusion de cette recherche est donc qu'un faible nombre de splits n'altère pas, au contraire, les performances de notre modèle pendant l'entraînement, contrairement aux nombres de splits plus élevés qui alourdissent l'entraînement, altèrent les performances à son issue et livre des incohérences de résultats entre les folds.

3.2. Normalisation

La normalisation est une étape importante dans la création d'un modèle CNN. Elle vise à rendre les données plus facilement exploitable par le modèle et à accélérer l'apprentissage.

L'idée principale derrière la normalisation est de mettre toutes les caractéristiques (ou pixels) des images sur une échelle commune. Cela permet d'éviter que certaines caractéristiques aient des valeurs très différentes des autres, ce qui pourrait perturber le processus d'apprentissage. En d'autres termes, la normalisation aide le modèle à mieux comprendre et généraliser les motifs présents dans les données.

3.2.1. Taille de l'image

Les images étant le centre du projet, nous avons donc essayé de jouer sur la taille de ces dernières afin de voir si nous pouvions améliorer les résultats de notre modèle. Nous avons donc essayé trois tailles d'images différentes, 32x32, 64x64 et 128x128.

Au-delà d'une taille d'image de 128x128 cela devenait compliqué pour certains PC de faire tourner le modèle, c'est pourquoi nous avons définis une limite qui est de 128 pixels.

Les étapes pour pouvoir tester plusieurs tailles d'images étaient simples, nous allons prendre l'exemple d'une taille d'image de 32x32. Pour pouvoir tester cette taille d'image, il suffisait simplement de changer les valeurs des variables "img_width" et "img_height".

```
# Define the size of the images you want to resize your images to
img_width, img_height = 32, 32 # Function to load and resize images from a folder
```

Une fois cette modification faite, dans la plupart des cas, il ne restait plus qu'à lancer le modèle. Cependant pour les images de taille 32x32, nous avons fait face à un problème.

En effet, lors de chaque passage de l'image dans une couche de convolution, sa taille est divisée par 2, et lors du passage dans la dernière couche de convolution, la taille d'image était négative et provoque alors logiquement une erreur.

Pour palier à ce problème, nous avons décidé, pour les images ayant des tailles trop petites pour passer toutes les couches sans problèmes, de supprimer la dernière couche de convolution.

Voici la couche que nous avons dû supprimer pour ce test :

```
model.add(Conv2D(filters=256, kernel_size=(3, 3),
activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Nous sommes désormais prêts à exécuter notre nouveau template de test, et une fois l'exécution terminée nous comparons les matrices de convolutions obtenues, ainsi que l'accuracy.

Toujours pour rester dans notre exemple avec les images de taille 32x32, voici les résultats obtenus :



Matrice de confusion

On peut constater ici que les résultats sont moins bons que ceux de notre modèle de base, nous avons beaucoup plus de faux négatifs (**27 ici contre 9 dans notre modèle de base**) et moins de vrais positifs (**363 ici contre 381 dans notre modèle de base**).

	precision	recall	f1-score	support
NORMAL	0.88	0.88	0.88	234
PNEUMONIA	0.93	0.93	0.93	390
accuracy			0.91	624
macro avg	0.91	0.91	0.91	624
weighted avg	0.91	0.91	0.91	624

En ce qui concerne l'accuracy, nous pouvons remarquer qu'elle est acceptable mais tout de même légèrement inférieure à celle de notre modèle de base (**0.91 contre 0.92**).

Après comparaison entre les différents templates, nous avons pu en conclure qu'une taille d'image plus grande conduit à une meilleure précision au détriment d'un temps d'entraînement plus important.

3.2.2. Équilibre du dataset

Il était intéressant pour nous d'essayer d'évaluer l'impact que l'équilibre du jeu de données d'entrainements avait sur les performances de notre réseau de neurones. Différentes combinaisons de nombres d'images de radiographies "NORMAL" et "PNEUMONIA" ont été testées.

Prenons l'exemple d'un jeu de données constitué de 3000 images "NORMAL" et de 5000 images "PNEUMONIA". Pour pouvoir réaliser cela, il a fallu modifier notre fonction de chargement d'images pour notre jeu de données d'entraînement.

```
def load_images_from_folder(folder, limit_per_subfolder=None):
    X = []
    y = []
    for subfolder in subfolders:
        folder_path = os.path.join(folder, subfolder)
        if limit_per_subfolder:
            # Get amount of random filenames from the folder
            if subfolder == 'NORMAL':
                limit = min(limit_per_subfolder,
len(os.listdir(folder_path)))
            else:
                limit = min(5000, len(os.listdir(folder_path)))
            filenames = np.random.choice(os.listdir(folder_path),
limit, replace=False)
        else:
```

On peut voir que dans cette fonction, nous passons 2 paramètres, le premier qui est le dossier (différent si c'est pour un jeu de données d'entraînement, de test ...) et le deuxième (paramètre optionnel) qui est une limite d'images par sous-dossiers.

Dans cette fonction, que nous avons adapté à nos besoins, nous pouvons remarquer que si le paramètre de limite d'images est renseigné, alors nous allons utiliser cette limite pour le sous-dossier "NORMAL" qui va contenir les images "NORMAL", pour les autres sous-dossier, nous laissons la limite à 5000.

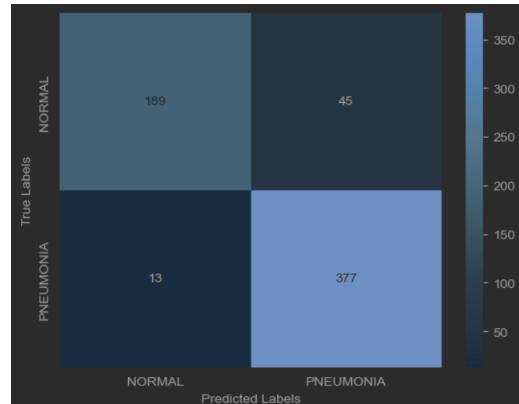
Il nous reste plus qu'à ajouter cette limite de 3000 images lors de l'appel à cette fonction pour le jeu de données d'entraînement pour réaliser le déséquilibre recherché.

```
min_images = 3000
X_train, y_train = load_images_from_folder(train_folder,
min_images)
```



Nous sommes désormais prêts à exécuter notre nouveau template de test, et une fois l'exécution terminée nous comparons les matrices de convolutions obtenues, ainsi que l'accuracy.

Toujours pour rester dans notre exemple avec un nombre d'images de 3000 pour les images "NORMAL" et 5000 pour les images "PNEUMONIA" (toujours dans notre jeu de données d'entraînement), voici les résultats obtenus :



Les résultats sont légèrement moins bons que ceux de notre modèle de base mais restent acceptables, nous avons beaucoup plus de faux négatifs (**13 ici contre 9 dans notre modèle de base**) et moins de vrais positifs (**377 ici contre 381 dans notre modèle de base**).

	precision	recall	f1-score	support
NORMAL	0.94	0.81	0.87	234
PNEUMONIA	0.89	0.97	0.93	390
accuracy			0.91	624
macro avg	0.91	0.89	0.90	624
weighted avg	0.91	0.91	0.91	624

Nous pouvons constater ici que l'accuracy est acceptable mais tout de même légèrement inférieur à celle de notre modèle de base (**0.91 contre 0.92**).

Les tests ont permis de constater qu'un jeu de données d'entraînement équilibré est important pour obtenir de meilleures performances avec le modèle CNN. Le déséquilibre du jeu de données en faveur d'une classe conduit à des performances moins bonnes, avec plus de faux négatifs et moins de vrais positifs. Avoir suffisamment d'images des deux classes est essentiel pour que le modèle apprenne à les reconnaître correctement. Il est donc recommandé de garder le jeu de données d'entraînement équilibré pour obtenir de meilleurs résultats.

3.3. Augmentation d'image

L'augmentation de données est une technique couramment utilisée pour améliorer la performance des modèles de machine learning. L'idée est de créer un ensemble d'entraînement plus large et plus varié à partir des données existantes en appliquant des transformations aléatoires.

Cette pratique est reconnue pour permettre au modèle d'apprendre à mieux généraliser avec de nouvelles images. Nous avons donc supposé que l'augmentation de données pourrait aider à améliorer les performances de notre modèle et nous avons souhaité valider cette information. Nous avons aussi cherché à découvrir les paramètres les plus impactant pour notre modèle et s'il était possible d'améliorer ses performances grâce à l'augmentation de données.

3.3.1. Techniques d'augmentation de données

Pour réaliser notre augmentation de données, nous avons utilisé à la fois des outils prêts à l'emploi fournis par la bibliothèque Keras et des techniques personnalisées. Compte tenu de la forme rectangulaire de nos images et de la présence d'informations non pertinentes, notamment des lettres à la périphérie des images, nous avons envisagé plusieurs techniques d'augmentation de données, dont la distorsion de ratio, le padding, le rognage, le masquage, le zoom, le cisaillement, la rotation, etc.

Pour le padding, le rognage et le masquage, nous avons utilisé des fonctions personnalisées. Nous avons reproduit le jeu de données existant en appliquant ces modifications avant d'augmenter l'ensemble des images.

Pour la distorsion de ratio, le zoom et le cisaillement, nous avons utilisé la classe “*ImageDataGenerator*” de Keras. Elle nous a permis de définir précisément les paramètres de chaque technique d'augmentation et de les appliquer de manière aléatoire à nos images lors de l'entraînement, économisant ainsi un temps précieux.

3.3.2. Modes opératoires

Nous avons dressé une liste (non exhaustive) des paramètres que nous voulions tester. Puis nous avons imaginé un mode opératoire.

Tout d'abord nous avons défini deux templates de base :

- Un premier nommé **aug_control_v2** et qui comprenait déjà les paramètres jugés optimaux via *ImageDataGenerator*

```
datagen = ImageDataGenerator(  
    rotation_range=20,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest'  
)
```

Ses principales métriques indiquaient :

- o **AUC de 0.90**
- o Une matrice de confusion de la forme [[191, 43], [9, 391]]
- o Heatmaps qui fixent les zones pertinentes

- Un second, intitulé **aug_control_v1**, et qui ne comportait aucun paramètre lié à *ImageDataGenerator* hormis le flip horizontal.

```
datagen = ImageDataGenerator(  
    rotation_range=0,  
    width_shift_range=0,  
    height_shift_range=0,  
    shear_range=0,  
    zoom_range=0,  
    horizontal_flip=True,  
    fill_mode='nearest',  
)
```

Les principales métriques indiquaient :

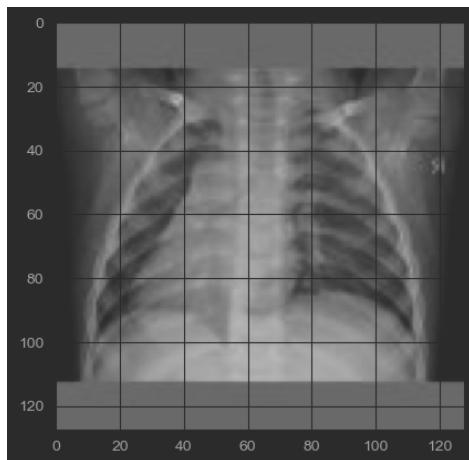
- o **AUC de 70**
- o Une matrice de confusion de la forme [[93, 141], [2, 388]]
- o Heatmaps qui fixent en partie sur les informations non pertinentes (lettres dans les coins)

Par défaut, ces deux templates utilisent l'option “horizontal_flip” et la distorsion de ratio. Pour examiner l'efficacité de différents paramètres d'augmentation, nous avons mené des tests dans deux scénarios distincts :

- Dans le premier scénario, marqué par **v1.xx**, nous avons comparé les résultats de nos tests aux résultats obtenus avec le modèle de référence **aug_control_v1**. Ce scénario a permis d'évaluer l'impact de chaque paramètre d'augmentation individuellement.
- Dans le deuxième scénario, marqué par **v2.xx**, nous avons comparé les résultats de nos tests aux résultats obtenus avec le modèle de référence **aug_control_v2**. Ce scénario était centré sur l'évaluation de l'impact combiné de différents paramètres d'augmentation.

L'étude du scénario V2 est particulièrement pertinente pour notre objectif, car elle permet de comprendre comment les différentes techniques d'augmentation de données peuvent interagir entre elles et influencer conjointement les performances du modèle.

3.3.3. Padding



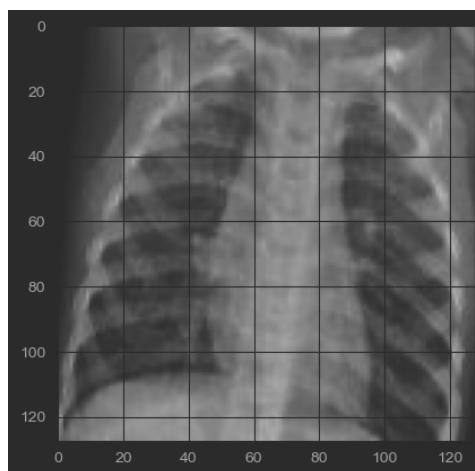
Le padding consiste à ajouter des pixels supplémentaires autour des bords d'une image. Lors de nos expérimentations, nous avons ajouté du padding avant le traitement de l'image par Keras sur des images rectangulaires pour les rendre carrées, ce qui a eu pour effet de conserver le ratio original.

- Dans la version V1, la performance du modèle semble s'être améliorée, bien que la heatmap montre, à l'instar du template, que le modèle ne se focalise pas toujours sur les zones pertinentes.

- En revanche, pour la version V2, le padding a dégradé les performances du modèle, avec une augmentation du nombre de faux positifs.

En conclusion, l'ajout du padding et la conservation du ratio original n'ont pas amélioré les performances du modèle en V2, seulement en V1. Par conséquent, nous n'avons pas retenu ce paramètre.

3.3.4. Zoom



Le zoom permet de modifier l'échelle d'une image, afin de permettre au modèle d'apprendre à reconnaître les caractéristiques des objets indépendamment de leur taille dans l'image

Durant les expérimentations, différents niveaux de zoom de 0 à 0.8 ont été appliqués depuis le centre des images.

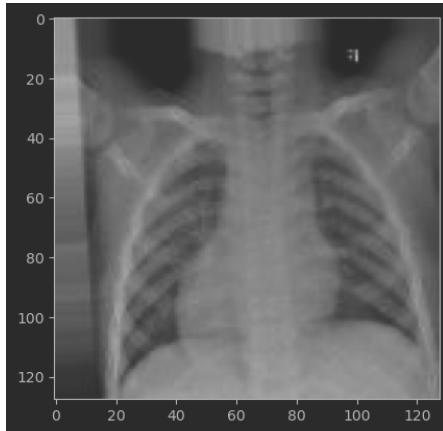
- En V1, le template ne bénéficie à la base d'aucun zoom. De manière générale, les effets du zoom se sont avérés très intéressants et ont permis d'améliorer de manière très significative les performances du modèle. Avec un zoom à 0.4, l'AUC est passé de 70 à 92 avec une heatmap normale, et des faux positifs et négatifs de 23 et 25. En augmentant au-delà, les performances diminuent.

- en V2 :
 - Zoom de 0 : n'a pas eu d'impact important. Une dégradation mineure des performances est constatée

- Zoom de 0.4 : L'AUC recule d'un point. A l'instar du zoom 0, les performances commencent à se dégrader avec plus de faux et de vrais négatifs.
- Zoom de 0.6 : Similaire à 0.4
- Zoom de 0.8 : La perte de performance se marque et les faux négatifs doublent en nombre.

En conclusion, le zoom, à lui tout seul, semble pouvoir permettre au modèle d'apprendre à mieux généraliser. **Une des raisons qui pourrait expliquer ce phénomène est peut-être le fait qu'il supprime des informations non pertinentes et concentre l'image sur les éléments pertinents.** Cependant, lorsqu'il est associé à d'autres paramètres, le zoom ne semble pas avoir autant d'impact. **La valeur idéale du paramètre semble être 0.2.**

3.3.5. Sheer

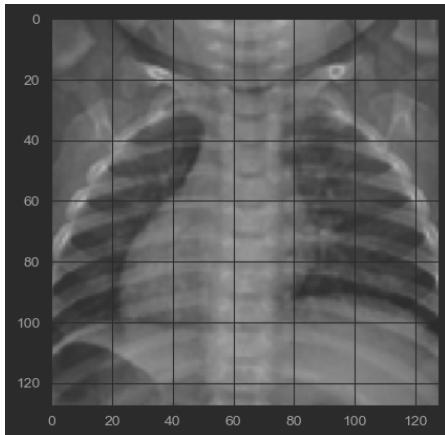


Le cisaillement est une transformation qui modifie la géométrie d'une image en déplaçant ses points dans une direction donnée, donnant l'impression que l'image est "poussée". Nous avons appliqué différents niveaux de cisaillement, de 0 à 0.6, à nos modèles :

- En V1, l'ajout de cisaillement a légèrement amélioré les performances, avec un nombre de faux négatifs inférieur (AUC à 76). Cependant, cette amélioration diminue avec un niveau de cisaillement supérieur à 0.4. Malgré cela, le cisaillement seul n'a pas réussi à empêcher les modèles de se concentrer sur des informations non pertinentes
- En V2
 - Shear de 0 : La suppression de ce paramètre n'a pas eu d'impact significatif sur les performances du modèle.
 - Shear de 0.4 : Une augmentation légère de ce paramètre n'a pas eu d'impact notable sur les performances du modèle.
 - Shear de 0.6 : L'AUC est passé de 90 à 92, accompagné d'une légère diminution des faux positifs et d'une légère augmentation des faux négatifs.

En conclusion, le cisaillement, surtout à 0.6, améliore les performances en V2 mais est moins efficace en V1. Ces résultats suggèrent que le cisaillement mérite une exploration plus approfondie, spécifiquement en combinaison avec d'autres techniques d'augmentation des données.

3.3.6. Crop



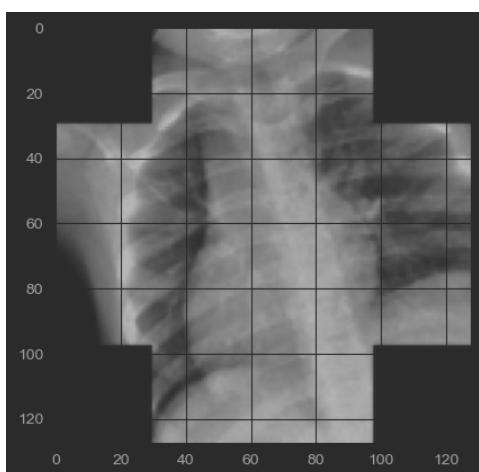
Le cropping est une technique d'augmentation des données qui consiste à couper une portion de l'image. Cela peut aider à focaliser l'attention du modèle sur les parties les plus importantes de l'image. Initialement, un script a été utilisé pour recadrer les images, passant d'un ratio rectangulaire à un ratio carré. Ensuite, les images ont été traitées par le module de data augmentation de Keras.

Les expérimentations en V1 et V2 ont produit des résultats différents :

- En V1, l'AUC est passée de 70 à 81 points. La matrice de confusion montre une diminution significative des faux positifs, cependant, le modèle continue de se focaliser en partie sur des informations non pertinentes.
- En V2, l'AUC a diminué, passant de 90 à 83 points. La matrice de confusion montre une augmentation générale des erreurs, en particulier des faux négatifs. De plus, la heatmap révèle un modèle moins sensible aux zones pertinentes de l'image.

En conclusion, le recadrage a amélioré les performances en V1, mais a dégradé celles en V2, suggérant que son utilisation doit être soigneusement calibrée. De plus, cette technique à elle seule ne semble pas suffire pour empêcher le modèle de se concentrer sur des informations non pertinentes.

3.3.7. Masking



Le "masking" consiste à masquer certaines parties de l'image pour réduire le bruit et les détails non pertinents, aidant le modèle à se concentrer sur les éléments essentiels. Cette technique a été appliquée en pré-traitement sur les images, avant la data augmentation de Keras. Nous avons testé différentes tailles de masques, supposant qu'elles pourraient optimiser la focalisation du modèle sur les éléments essentiels de l'image.

- En V1, une seule taille de masque a été testée. Elle a permis d'augmenter les performances de 10 points avec un faible nombre de faux négatifs. La heatmap révèle que le modèle se concentre mieux sur les zones pertinentes, bien que cela ne soit pas encore parfait.

- En V2, plusieurs tailles de masques (de 35 px à 90px) ont été testées. Les premières itérations ont montré des performances légèrement inférieures au template V2. Cependant, lors de la dernière itération, les performances ont chuté considérablement, suggérant que cette approche pourrait ne pas être optimale pour ce type de jeu de données.

En conclusion, bien que le "*masking*" ait montré des résultats prometteurs en V1, son efficacité en V2 est discutable, nécessitant des investigations supplémentaires pour déterminer son utilité réelle dans l'augmentation de données

3.3.8. Résultats

En synthèse, l'efficacité des techniques d'augmentation de données, telles que le zoom, le padding, le cisaillement, le recadrage et le "*masking*", varie en fonction du contexte. Le zoom et le cisaillement ont montré des améliorations prometteuses des performances du modèle, tandis que le padding et le recadrage ont produit des résultats mixtes, améliorant les performances dans certains cas (V1) mais pas dans d'autres (V2). Le "*masking*" a montré des résultats encourageants en V1, mais sa pertinence en V2 reste à déterminer.

Ces résultats soulignent l'importance de l'évaluation rigoureuse et de l'adaptation judicieuse des techniques d'augmentation en fonction du contexte spécifique du jeu de données et du modèle. Ils rappellent également la nécessité de surveiller les biais potentiels introduits par l'augmentation de données.

3.4. CNN custom

Lors de notre apprentissage sur l'utilisation des réseaux de neurones, plusieurs notions théoriques nous ont été présentées. Cependant, nous n'avons pas forcément pu voir en pratique leur impact sur les résultats de l'entraînement de notre modèle. C'est pourquoi nous avons joué sur les différents paramètres de notre architecture pour mieux comprendre les impacts de chacun sur les scores finaux, mais aussi pour trouver d'éventuelles optimisations permettant d'obtenir une meilleure précision finale.

3.4.1. Variation du nombre de filtres

Un des premiers paramètres sur lequel il est possible de jouer dans l'architecture de notre réseau de neurones est le nombre de filtres par couches de convolutions. Pour cette expérimentation, nous avons joué sur un nombre relativement faible et un nombre relativement haut par rapport aux filtres de notre *template*. On a alors établi 4 tests:

- réduire le nombre de filtres à un extrême bas sans changer le nombre de neurones par couches cachées
- réduire le nombre de filtres à un extrême bas
- réduire le nombre de filtres à un extrême haut sans changer le nombre de neurones par couches cachées
- réduire le nombre de filtres à un extrême haut

Par exemple, voici l'implémentation pour le test de l'extrême bas avec modification des couches cachées:

```
model = Sequential()

model.add(Conv2D(filters=2, kernel_size=(3,3), input_shape=(img_width, img_height, 1),
activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=4, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=8, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=16, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())

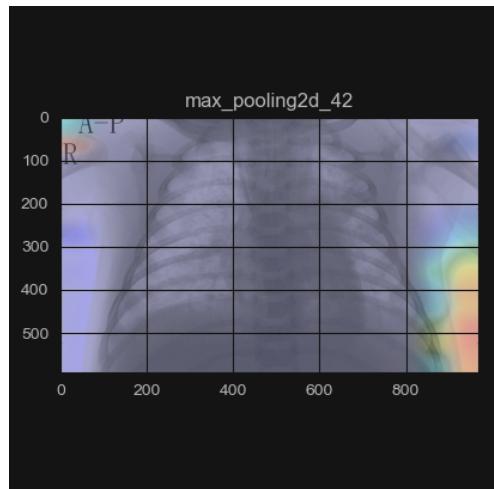
model.add(Dense(16, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(8, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(1, activation="sigmoid"))

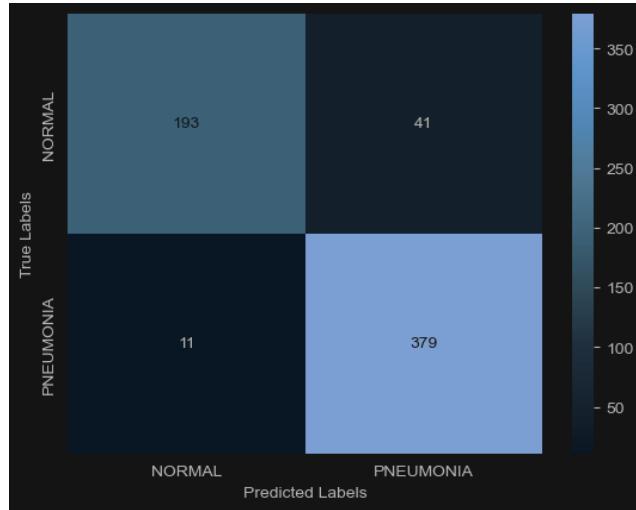
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=metrics)
```

Pour cette implémentation, on compte 10937 paramètres au total contre 2,7M. Logiquement, cette baisse de paramètre se fait ressentir dans les résultats: le score de précision a chuté **88%** et les cartes de chaleur montrent des zones d'intérêts qui ne sont pas censées permettre de se prononcer.



Exemple de heatmap extraite d'une prédiction du modèle avec peu de filtres et de neurones sur une image de test

Au contraire, le test qui ne limite seulement que le nombre de filtres sans toucher aux couches cachées présente des résultats très satisfaisants. Le modèle possède alors 182297 paramètres. On obtient la matrice de confusion suivante:



Matrice de confusion obtenue sur les données du jeu de test suite à l'entraînement de notre modèle avec un nombre de filtres limités

L'entraînement du modèle permet d'obtenir un score de précision équivalent au *template* (92%) avec une tendance au *recall* avantageant les faux positifs (*recall* plus important la prédiction "Pneumonie"). De plus, les cartes de chaleurs montrent des détections de caractéristiques cohérentes par le réseau de neurones à convolutions: **les résultats du test sont favorables**. Le gain de cette baisse de filtre est **la réduction du temps d'entraînement**, qui peut avoir un impact temporel et monétaire dans un cas plus concret. **Cela montre aussi que la complexité d'un modèle n'est pas proportionnelle à sa performance.**

Le cas inverse de test, à savoir l'augmentation des filtres par couches sans modifier les couches cachées montrent des résultats similaires d'un point de vue performance, mais avec une augmentation du temps d'entraînement. On aura alors tendance à réduire le nombre de filtres pour notre cas, plutôt que de l'augmenter.

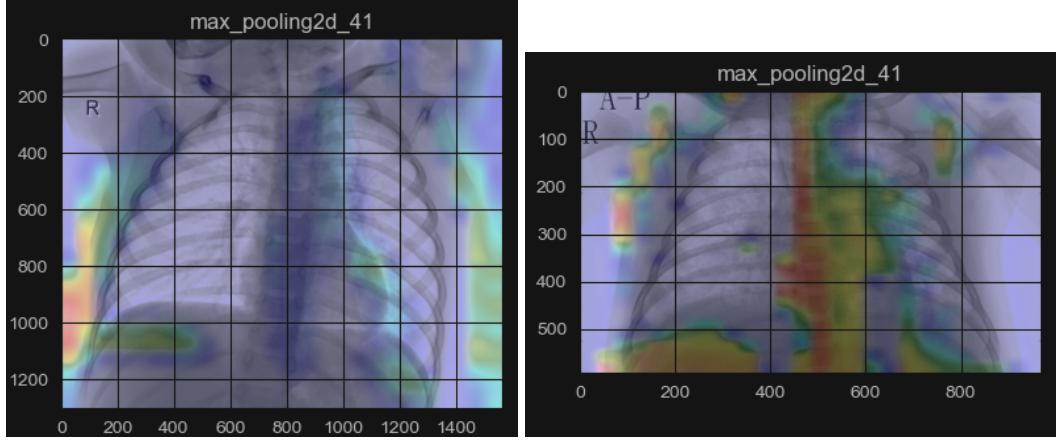
Parallèlement au test d'altération des couches de convolutions et des couches cachées vers le bas, le test d'altération vers le haut à les mêmes conséquences sur l'évaluation: une perte de précision et des choix de caractéristiques non pertinents faits par le réseau de neurones. **On peut alors aussi conclure que les couches cachées semblent faire varier plus largement les résultats que le nombre de filtres en lui-même.**

3.4.2. Variation de la taille du filtre (kernel size)

La taille du filtre est un hyperparamètre sur lequel il est également possible d'influer. Dans notre cas, nous avons décidé de mener deux expériences:

- augmenter la taille des filtres de la première couche de convolution
- augmenter la taille des filtres de toutes les couches de convolution

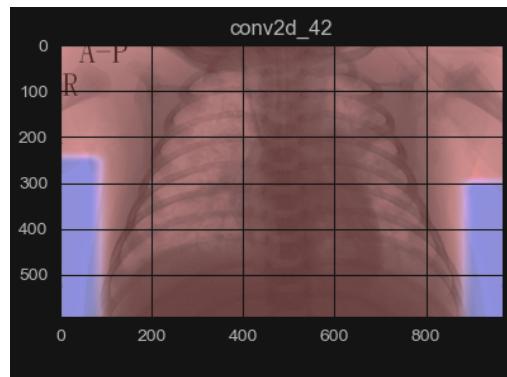
Dans le cas d'une augmentation partagée de la taille du filtre sur les différentes couches de convolutions (taille de 16 par 16), on remarque que le changement empêche le réseaux de neurones d'extraire des caractéristiques totalement pertinentes sur les images proposées. **Même si le score de précision est de 93% et l'AUC à 0.92**, les cartes de chaleurs montrent des faiblesses sur le choix des zones d'intérêt. Paradoxalement, ce n'est pas le cas de toutes les cartes de chaleur:



A gauche, une heatmap réalisée sur une image de radio normale. A droite, une image réalisée sur une radio d'un patient atteint de pneumonie.

En fonction de ces cartes de chaleur, on pourrait alors en déduire que l'augmentation de la taille du filtre pourrait avoir des effets positifs sur la détection des caractéristiques.

De manière radicalement opposée, l'augmentation de la taille du filtre pour la première couche de convolution (taille de 32 par 32) a eu un effet catastrophique sur la performance du modèle. Le score de précision a chuté à **73%** et les cartes de chaleur sont sans appel:



Heatmap réalisée sur une image de radio d'un patient atteint de pneumonie par le modèle ayant une première couche de convolution avec une taille de filtre extrêmement haute.

Si une critique pouvait être apportée à ce test, il semblerait qu'une taille de filtre légèrement plus grande dès la première couche de convolution permettrait d'extraire plus facilement certaines caractéristiques. Ce comportement a été observé lors de tests qui ont suivi et semble également être validé par le test d'augmentation de la taille de filtre partagée sur plusieurs couches de convolutions. Le problème de nos tests sur la taille des filtres est d'avoir pris des valeurs trop élevées.

3.4.3. Variation du nombre de couches de convolution

En parallèle de l'augmentation du nombre de filtre par couches de convolution, nous avons testé d'augmenter et de réduire le nombre de couches de convolutions dans notre architecture. Cela est très similaire aux tests de modification du nombre de filtres par couches, car nous modifions la complexité du modèle et son nombre de paramètres.

En termes d'implémentation, dans le cas d'une réduction du nombre de couche, on a:

```
model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(3,3), input_shape=(img_width, img_height, 1),
activation="relu"))
model.add(MaxPooling2D(pool_size=(4,4)))

model.add(Conv2D(filters=256, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(4,4)))

model.add(Flatten())

model.add(Dense(256, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(128, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(1, activation="sigmoid"))

# compile the model
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=metrics)
```

Pour ce cas précis, on observe une baisse de précision (90%) avec une augmentation du nombre de faux négatifs: **les résultats ne sont pas satisfaisants mais restent acceptables.**

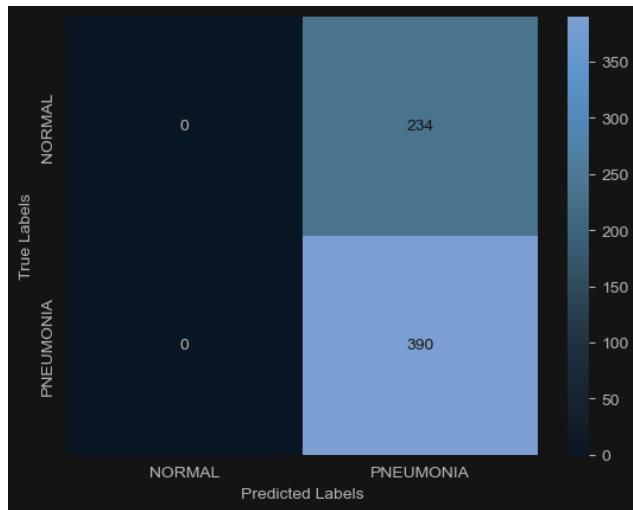
L'ajout de couche de convolutions a eu un effet de bord massif, en suivant cette implémentation:

```
model.add(Conv2D(filters=32, kernel_size=(3,3), input_shape=(img_width, img_height, 1),
activation="relu"))
model.add(Conv2D(filters=64, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), activation="relu"))
model.add(Conv2D(filters=64, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
```



Comme en témoigne la matrice de confusion ci-dessus, le modèle a été incapable de réaliser des prédictions valables. Il semblerait donc y avoir des règles à respecter lors de la répartition des couches de convolutions et de leur nombre de filtres. **Comme en témoigne le notebook correspondant**, les cartes de chaleur montrent qu'il n'y a presque aucune caractéristique relevée par le réseau de neurones pour faire ses prédictions.

Lors du dernier test, nous avons tenté d'améliorer notre architecture en ajoutant une seule couche de convolution avec une couche de *max pooling*, et une couche de neurones cachée le tout suivant une logique d'augmentation progressive du nombre de filtres. L'implémentation est la suivante:

```

model.add(Conv2D(filters=32, kernel_size=(3,3), input_shape=(img_width, img_height, 1),
activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())

model.add(Dense(512, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(256, activation="relu"))
model.add(Dropout(0.5))

model.add(Dense(128, activation="relu"))
model.add(Dropout(0.5))

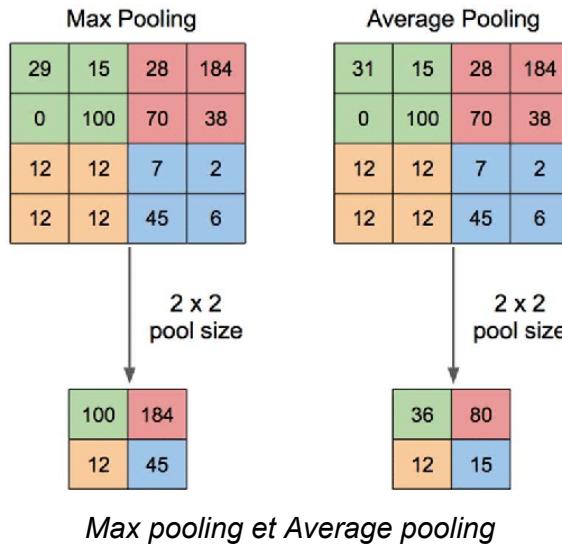
model.add(Dense(1, activation="sigmoid"))

```

Malgré que les cartes de chaleur montrent des caractéristiques plus que pertinentes, la précision a chuté à 90% contre 92%, prouvant encore qu'il est important de mesurer la complexité de son architecture en fonction de son cas d'étude.

3.4.4. Max Pooling et Average Pooling

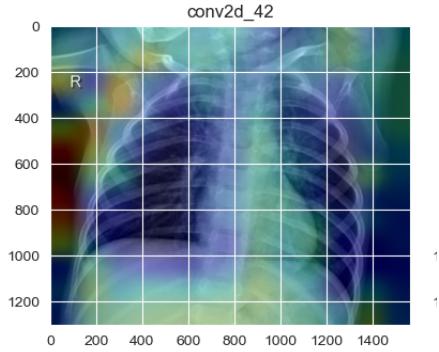
Au sein de notre architecture, nous avons par défaut utilisé le *max pooling*, c'est-à-dire le pooling utilisé le plus communément. Le but du test suivant est de voir l'impact de l'utilisation du *average pooling* à la place du *max pooling*.



D'un point de vue détection de caractéristique, l'*average pooling* a fait basculer une partie des prédictions vers le pattern de lettre ([voir partie 2.2.2](#)). Le changement a également eu une incidence sur la précision, avec une chute à 90%, qui serait acceptable si la détection de caractéristique n'était pas impactée.

En reprenant le *max pooling*, nous avons également testé l'impact de la taille du *pooling* en l'augmentant sur chacune des couches de convolutions; nous avions alors deux couches de convolutions avec deux couches de max pooling (réduction du nombre de couche liée à l'augmentation de la taille du *pooling*):

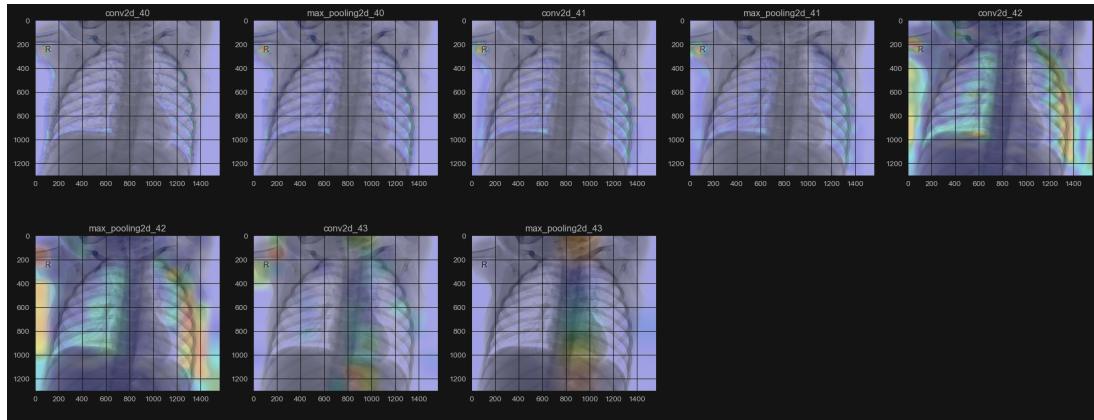
```
model.add(MaxPooling2D(pool_size=(8,8)))
```



De manière analogue, la détection des caractéristiques pertinentes semble avoir été brouillé. Les deux tests n'ont pas été concluants mais montrent que le type et la taille du pooling ont un impact concret sur la détection des caractéristiques de nos images et peuvent avoir un intérêt à être modifié selon la taille des images en entrée.

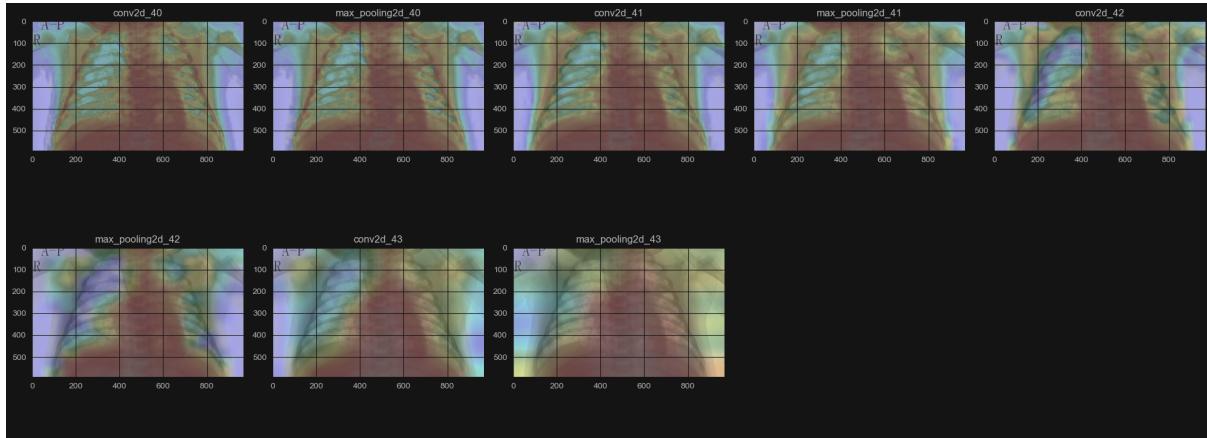
3.4.5. Variation du dropout

En réponse à *l'overfitting* de nos entraînements sur notre modèle, nous avions appliquer un *drop out* sur les couches cachées. Nous avons alors réalisé 3 tests pour tester l'impact réel du *drop out* sur la performance de notre modèle: avec un ***drop out* élevé (0.9 par couche)**, **bas (0.1 par couche)** et **sans *drop out***.



Heatmaps issues de l'évaluation du modèle sans drop out

L'expérimentation sans *drop out* a eu les effets attendus: la non détection de caractéristiques pertinentes sur les images qui s'apparentent à de *l'overfitting*. **L'augmentation du *drop out* a eu le même effet. Il est à noter que les deux évaluations présentaient une précision de 93%.**



Heatmaps issues de l'évaluation du modèle avec un drop out très bas

Contrairement aux deux premiers tests, la réduction du drop out a permis de stabiliser et légèrement augmenter le score de précision tout en permettant au réseau de neurones de prendre des décisions cohérentes.

Le drop out a donc bel et bien une incidence sur l'effet d'overfitting, mais il est à utiliser prudemment pour éviter de rendre le drop out totalement inefficace en trompant les résultats.

3.4.6. Optimizer (autre que 'adam')

Parmi les derniers leviers pouvant être modifiés, il est possible de jouer sur l'optimiseur du modèle. L'optimiseur est une fonction mathématique utilisée dans les algorithmes d'apprentissage automatique pour ajuster les paramètres d'un modèle de manière à minimiser la différence entre les prédictions du modèle et les valeurs réelles des données d'entraînement; c'est ce dernier qui ajuste les poids de manière itérative. Durant quelques tests, nous avons testé les optimiseurs **Nadam**, **RMSProp** et **SDG**, tous les trois proposés par Keras.

Optimiseur	Score de précision	Heatmaps cohérentes
Adam (par défaut)	0.92	OUI
SGD	0.89	Partiellement
Nadam	0.92	OUI
RMSProp	0.93	OUI

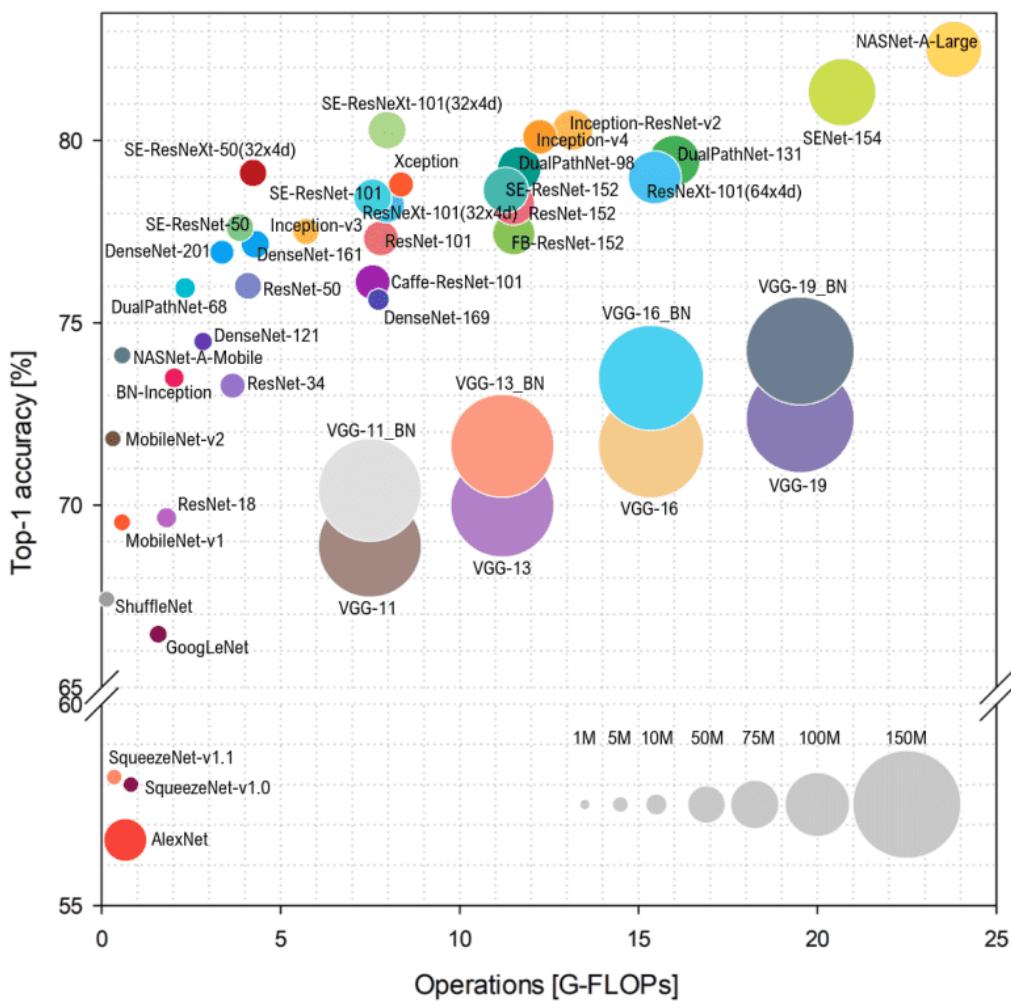
Là où SDG a fait chuter la précision à 89% et a créé une faille dans la recherche de caractéristiques pertinentes sur les images, Nadam a gardé des résultats en phase avec le template. A noter que Nadam est une extension d'Adam, ce qui pourrait expliquer l'impact très faible du changement. En revanche, RMSProp a permis d'augmenter la précision à 93%, ce qui est considérable.

Ces tests permettent de montrer qu'il est important de tester plusieurs fonctions d'optimisation car il est possible d'augmenter plus ou moins la performance de son modèle.

3.5. Classifieur / Modèle connus

En parallèle des différents essais avec plusieurs paramètres sur notre réseau de neurones à convolutions, il est intéressant d'appliquer le même jeu de données en entrée à des classificateurs déjà connus, que ce soit des algorithmes de *Machine Learning* ou bien des architectures de réseaux de neurones. Le but est de pouvoir juger de la pertinence d'algorithme et d'architecture plus ou moins complexes par rapport à notre problématique afin de démontrer que la complexité n'est pas automatiquement liée à une bonne performance sur un jeu de données précis.

Pour ce faire, nous avons testé l'algorithme de *Machine Learning RandomForest*, avant de tester ensuite l'implémentation des architectures de réseaux de neurones à convolutions **AlexNet** et **ResNet50**.



L'intérêt de tester deux architectures de CNN différentes s'explique par ce diagramme; sur ce dernier, on retrouve la complexité du modèle (en G-FLOPs) en abscisse et la précision obtenue sur le jeu de données **ImageNet** en ordonnée. La taille des cercles représentant chacune des architectures correspond au nombre de paramètres de ces architectures. On remarque qu'AlexNet et ResNet50 sont deux architectures moyennement complexes en comparaison à d'autres architectures, avec un nombre de paramètres moyen mais des précisions sur le jeu de données **ImageNet** très différentes.

Il est intéressant d'étudier si ces architectures seraient de manière équivalente performante sur notre jeu de données relativement limitées par rapport au jeu de données **ImageNet**, qui comprend actuellement plus de 14 millions d'images.

A noter qu'il a été question de réaliser des tests sur notre jeu de données avec l'architecture VGG16 ou VGG19, implémentées par Keras. Ces tests n'ont pas été concluants car le modèle final bloquait à une précision de 50%. Ce problème est sûrement lié à une anomalie dans notre implémentation ou dans les données passées en entrée. Toutefois, l'implémentation de ResNet50 montre qu'un plus grand nombre de paramètres ou une complexité plus importante du modèle n'aurait pas automatiquement permis d'obtenir de meilleurs résultats.

3.5.1. RandomForest

3.5.1.1. Présentation

De manière simplifiée, le Random Forest ou forêt aléatoire est un algorithme de prédiction créé en 1995 par Ho, puis formellement proposé par les scientifiques Adele Cutler et Leo Breiman en 2001. Il combine les notions de sous-espaces aléatoires et de *bagging*.

Le Random Forest est composé de plusieurs arbres de décision, entraînés de manière indépendante sur des sous-ensembles du jeu de données d'apprentissage (méthode de *bagging* permettant de réduire la variance et le surapprentissage).

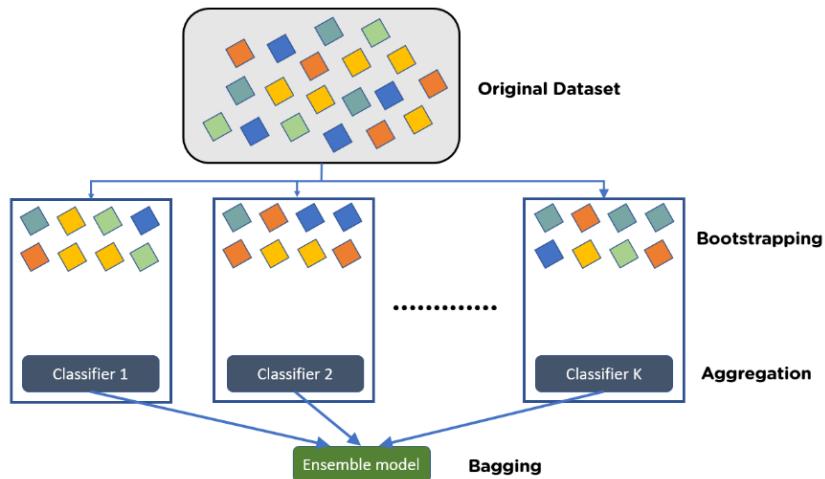


Schéma expliquant le principe de "Bootstrap aggregating" ou bagging

Chacun des arbres produit une estimation, et la combinaison des résultats permet d'obtenir la prédiction finale qui se traduit par une variance réduite. En somme, il s'agit de s'inspirer de différents avis, traitant un même problème, pour mieux l'appréhender. En quelque sorte, on calcule la moyenne des avis de chacun des arbres. Chaque modèle est distribué de façon aléatoire en sous-ensembles d'arbres décisionnels.

Random Forest Simplified

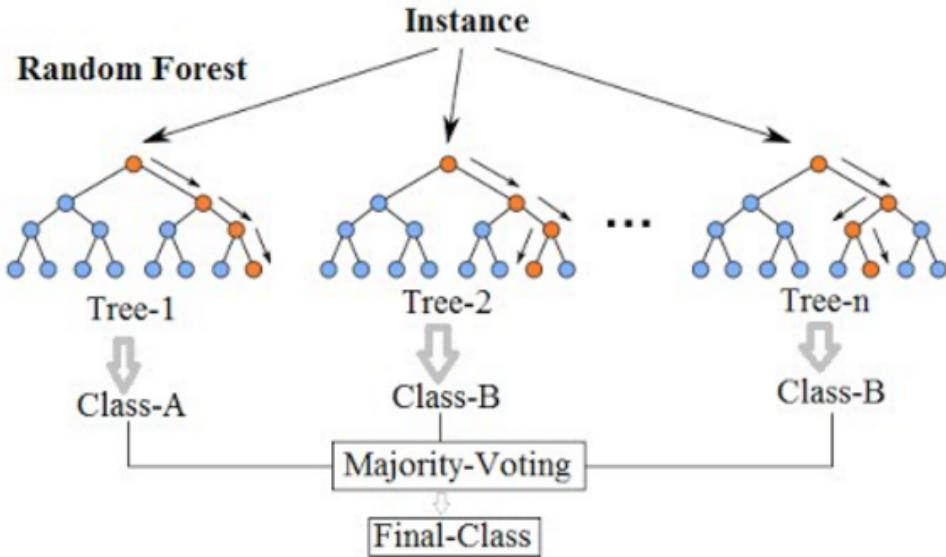


Schéma montrant la manière dont Random Forest rend sa prédiction selon une donnée d'entrée

3.5.1.2. Implémentation

Dans un cas de classification, Random Forest s'implémente facilement grâce à **Scikit-Learn** et sa classe **RandomForestClassifier**:

```
from sklearn.ensemble import RandomForestClassifier  
  
# Define the model architecture  
def create_model():  
    return RandomForestClassifier(random_state=42)
```

Il est possible de paramétriser (*tuning*) l'algorithme grâce à plusieurs paramètres, qui vont notamment déterminer la manière dont les arbres de décisions seront construits et comment l'estimation de chaque arbre sera calculée. On peut combiner l'ensemble de ces paramètres avec l'outil *GridSearchCV* de Scikit-Learn pour tester aléatoirement plusieurs combinaisons. Ici, on teste 10 combinaisons différentes:

```

from sklearn.model_selection import RandomizedSearchCV
params = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 15, 20],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 5, 10, 15],
    'max_features': ['log2']
}

nb_folds = 10

model = create_model()
estimators = RandomizedSearchCV(model, param_distributions=params, n_iter=nb_folds,
cv=nb_folds, verbose=3, random_state=42)
estimators.fit(X_train_temp, y_train)

```

3.5.1.3. Résultats

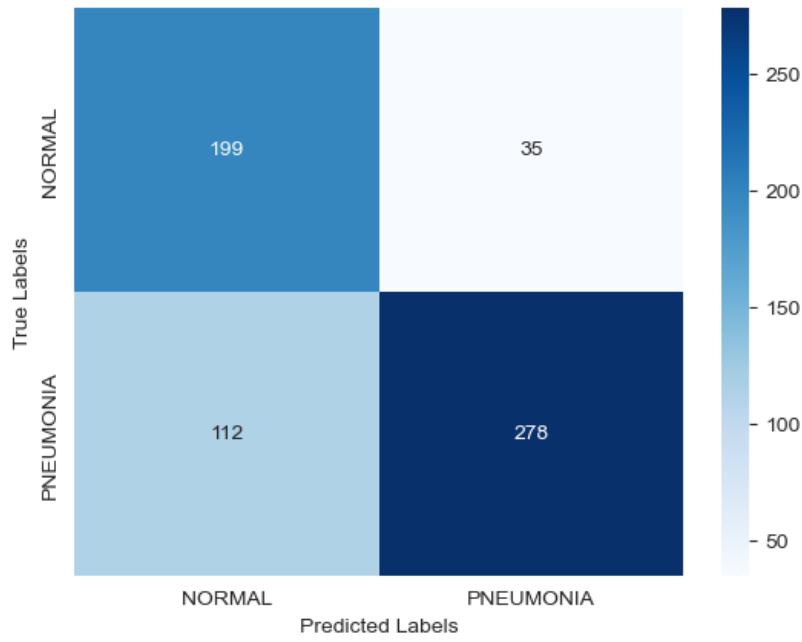
Une fois la meilleure combinaison de paramètre trouvée, on peut ensuite entraîner une instance de classifieur sur le jeu de données d'entraînement:

```

model = RandomForestClassifier(
    n_estimators=estimators.best_params_['n_estimators'],
    max_depth=estimators.best_params_['max_depth'],
    min_samples_split=estimators.best_params_['min_samples_split'],
    min_samples_leaf=estimators.best_params_['min_samples_leaf'],
    max_features=estimators.best_params_['max_features'],
    random_state=42
)
model.fit(X_train_temp, y_train)

```

On réalise ensuite des prédictions sur le jeu de données de test. Les résultats des prédictions sont les suivants:



Matrice de confusion obtenue sur les données du jeu de test suite à l'entraînement du classifieur RandomForest avec les meilleurs paramètres trouvés

Cette matrice de confusion présente un nombre très important et supérieur de faux négatifs par rapport aux faux positifs. Ces résultats se traduisent aussi par une précision obtenue de 76%.

Les tests ont démontré que le Random Forest permet d'obtenir une valeur de précision acceptable assez rapidement, ce qui permet de déterminer une valeur basse qui devient le seuil inférieur toléré lors de la recherche d'une solution plus adaptée. Bien que le Random Forest puisse offrir un temps d'entraînement plus court, il ne parvient pas à rivaliser avec notre CNN en termes de précision, ce qui montre bien l'importance des CNN et plus généralement des réseaux de neurones dans le monde de l'Intelligence Artificielle

3.5.2. AlexNET

3.5.2.1. Présentation

AlexNet est un réseau de neurones convolutifs profonds qui a été présenté en 2012 par Alex Krizhevsky, Ilya Sutskever et Geoffrey Hinton. Il a remporté le concours **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** en 2012, ce qui a marqué une étape importante dans l'histoire de l'apprentissage profond.

En effet, lors de l'édition 2010, les meilleurs scores obtenus à ce concours étaient 47.1% pour la métrique *top-1 error* et 28.2% pour la métrique *top-5 error*. Le *top-1 error* permet de mesurer la précision avec laquelle le modèle prédit exactement le même label que le label cible; le *top-5 error* indique si le label cible fait partie des 5 premières prédictions du modèle. AlexNet a battu le précédent score en établissant un *top-1 error* 37.5% et un *top-5 error* de 17.0%.

Avant AlexNet, les réseaux de neurones profonds étaient considérés comme difficiles à entraîner et à optimiser, et étaient généralement limités à des architectures plus simples. Cependant, plusieurs innovations ont permis d'améliorer les performances des réseaux de neurones profonds:

- La fonction d'activation ReLU (Rectified Linear Unit) qui a permis d'accélérer l'apprentissage en évitant les problèmes de saturation des fonctions d'activation précédemment utilisées.
- L'entraînement parallèle sur plusieurs cartes graphiques (GPU), qui a permis d'accélérer considérablement le temps d'entraînement.

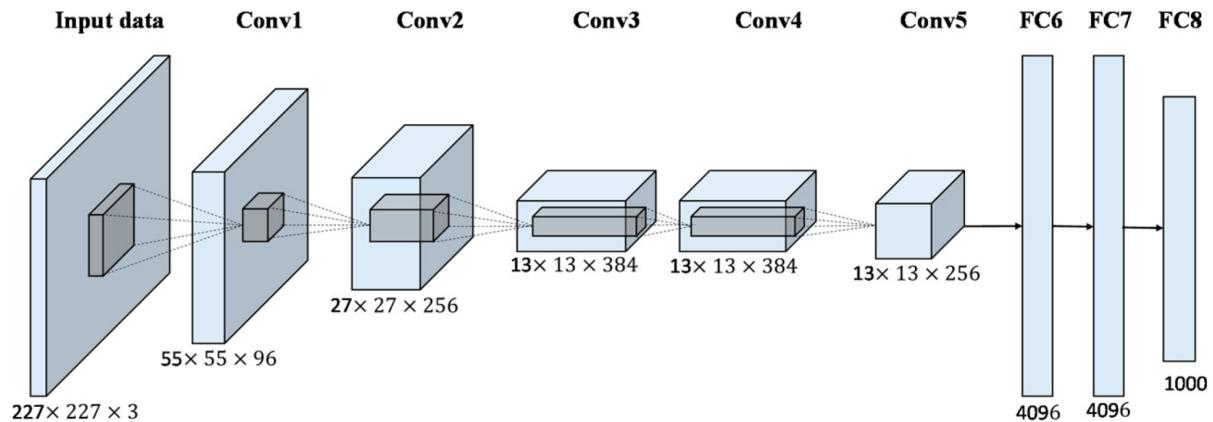


Schéma de l'architecture d'AlexNet avec 8 couches au total

L'architecture d'AlexNet se compose de 5 couches de convolution parfois suivies de pooling et de 3 couches de neurones cachés.

Historiquement, AlexNet a été une avancée majeure dans le domaine du *deep learning* en montrant que les réseaux de neurones profonds peuvent être entraînés avec succès pour des tâches de classification d'images à grande échelle. Il a introduit de nouvelles techniques qui ont depuis été largement adoptées et a ouvert la voie à des progrès considérables dans de nombreux domaines de l'intelligence artificielle. AlexNet est considéré comme l'un des papiers scientifiques les plus influents dans le domaine de la vision par ordinateur (*Computer Vision*); à l'heure actuelle, l'article d'AlexNet a été cité plus de 132 000 fois.

3.5.2.2. Implémentation

Afin d'implémenter AlexNet, il a été nécessaire d'écrire de zéro l'architecture du modèle à l'aide des outils fournis par Keras:

```
model = Sequential()

# 1st Convolutional Layer
model.add(Conv2D(filters=96, input_shape=(img_width, img_height, 3),
                 kernel_size=(11,11), strides=(4,4), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'))

# 2nd Convolutional Layer
model.add(Conv2D(filters=256, kernel_size=(11,11), strides=(1,1), padding='valid',
                 activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid'))

# 3rd Convolutional Layer
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
                 activation='relu'))
```

```

# 4th Convolutional Layer
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
activation='relu'))

# 5th Convolutional Layer
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid'))

# Flatten
model.add(Flatten())
# 1st Fully Connected Layer
model.add(Dense(4096, input_shape=(img_width * img_height * 3,), activation='relu'))
model.add(Dropout(0.4))

# 2nd Fully Connected Layer
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.4))

# 3rd Fully Connected Layer
model.add(Dense(1000, activation='relu'))
model.add(Dropout(0.4))

# Output Layer
model.add(Dense(1, activation='sigmoid'))

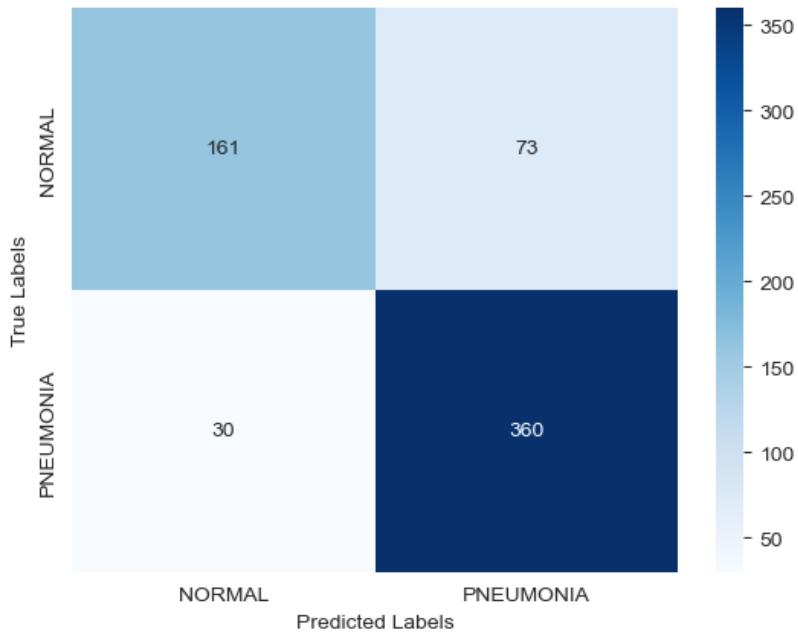
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=metrics)

```

On note dans notre implémentation l'ajout d'une dernière couche cachée avec un neurone afin d'adapter la sortie à notre cas de classification binaire. Cela implique aussi l'utilisation de la fonction de coût *Binary Cross Entropy*.

3.5.2.3. Résultats

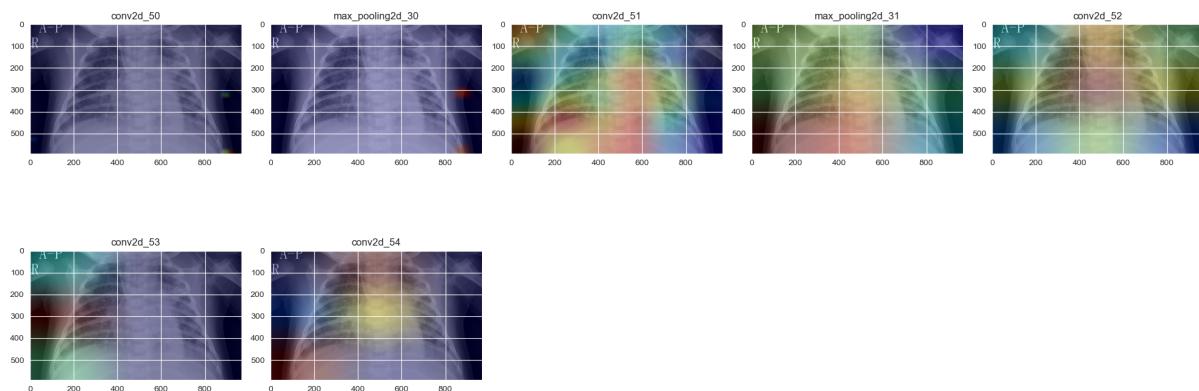
A la suite de l'entraînement du modèle, les prédictions sur le jeu de données de test ont permis d'obtenir les résultats suivants:



Matrice de confusion obtenue sur les données du jeu de test suite à l'entraînement du modèle basé sur l'architecture d'AlexNet

Cette matrice de confusion présente un nombre faux positifs supérieurs à celui des faux négatifs, se traduisant par un *recall* de 69% pour le label “radio normale”. Au total, on observe une précision de 83%.

En comparaison avec Random Forest, la supériorité des faux positifs sur les faux négatifs est rassurante (on préférera détecter une pneumonie plutôt que l'ignorer). En revanche, la précision obtenue est nettement inférieure à notre modèle fait main qui obtient un score de précision de 92%. Cela peut s'expliquer par le fait qu'AlexNet est un modèle plus complexe et moins spécifique à notre cas; ce dernier ayant été testé sur le jeu de données **ImageNet**.



Extrait de heatmaps générés par couches par le modèle AlexNet sur une image de test passée en entrée annotée “Pneumonie”

De plus, on remarque grâce à des heatmaps que le modèle semble baser ces choix sur des éléments qui ne permettent pas forcément de déterminer si le patient est malade ou non.

Néanmoins, AlexNet reste beaucoup plus puissant que RandomForest, ce qui montre l'intérêt du *deep learning* dans les cas de classification par rapport aux algorithmes de Machine Learning.

3.5.3. ResNET

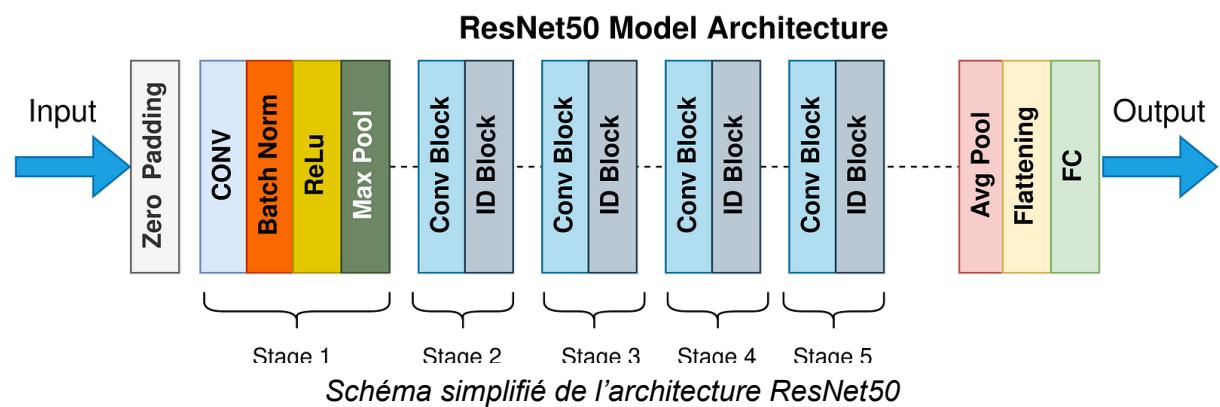
3.5.3.1. Présentation

ResNet, ou Residual Network, est un type de réseau de neurones profonds qui a été introduit en 2015 par des chercheurs de Microsoft Research dans leur article "*Deep Residual Learning for Image Recognition*".

Avant l'introduction de ResNet, l'augmentation de la profondeur des réseaux de neurones était souvent limitée par le phénomène de la "*vanishing gradient*", qui rendait l'apprentissage plus difficile à mesure que le nombre de couches augmentait. ResNet a résolu ce problème en utilisant des connexions résiduelles (ou *shortcut connections*), qui permettent à l'information de sauter des couches dans le réseau. En utilisant ces connexions résiduelles, ResNet a pu entraîner avec succès des réseaux de plus de 100 couches, ce qui était auparavant considéré comme difficile à réaliser.

ResNet50 est une variante spécifique de ResNet qui a 50 couches. Il a été utilisé pour remporter la compétition de classification d'images sur **ImageNet** en 2015 avec une précision de 96,5%. Cette architecture a révolutionné le domaine de la vision par ordinateur en améliorant considérablement la précision de la classification d'images.

A noter que par la suite, nous allons utiliser l'implémentation de ResNet50 par la librairie Keras, qui indique sur son site avoir obtenu un score *top-1 accuracy* de 74,9% et un score *top-5 accuracy* de 92,1% sur le jeu de données **ImageNet**, ce qui est nettement supérieur aux scores d'AlexNet.



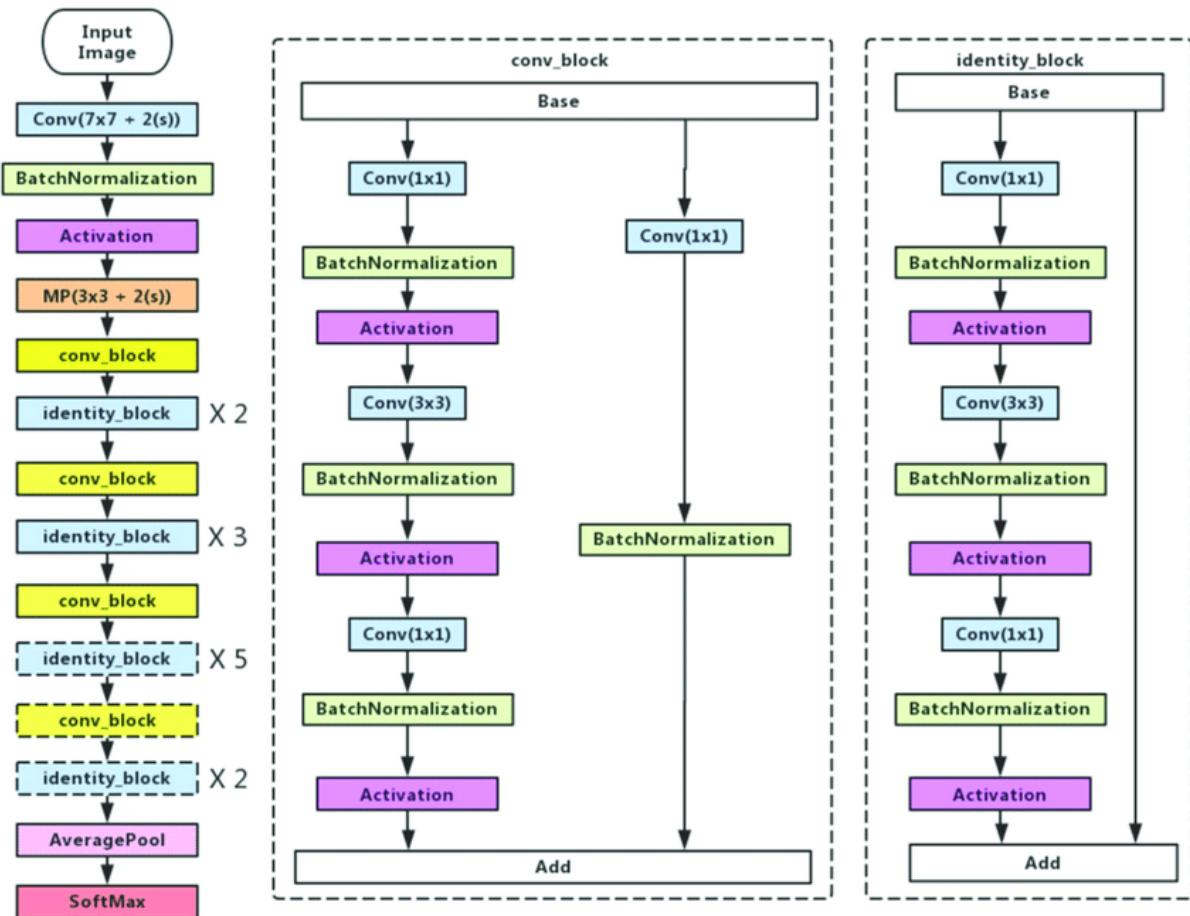


Schéma détaillé de l'architecture ResNet50, avec une vision sur les blocs de connexions résiduelles

L'architecture de ResNet50 se compose de plusieurs blocs de connexions résiduelles, chacun contenant plusieurs couches de convolutions, de normalisation par lots et d'activation ReLU. La sortie de chaque bloc est additionnée à la sortie d'une connexion résiduelle pour former l'entrée du bloc suivant. Cette structure permet de réduire la quantité d'information perdue lors de la propagation avant dans le réseau, en aidant les gradients à se propager plus facilement à travers les couches.

3.5.3.2. Implémentation

Comme explicité précédemment, l'implémentation de ResNet50 peut se faire facilement grâce à Keras:

```
resnet_model = ResNet50(
    include_top=False,
    weights=None,
    input_tensor=None,
    input_shape=(img_width, img_height, 3),
    pooling='max',
    classes=1
)

x = resnet_model.output
x = Flatten()(x)
x = Dense(256, activation='relu', name='custom_dense_1')(x)
```

```

x = Dropout(0.5)(x)
x = Dense(128, activation='relu', name='custom_dense_2')(x)
x = Dropout(0.5)(x)
predictions = Dense(1, activation='sigmoid', name='custom_dense_3')(x)

# Create the model
model = Model(inputs=resnet_model.input, outputs=predictions)

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=metrics)

```

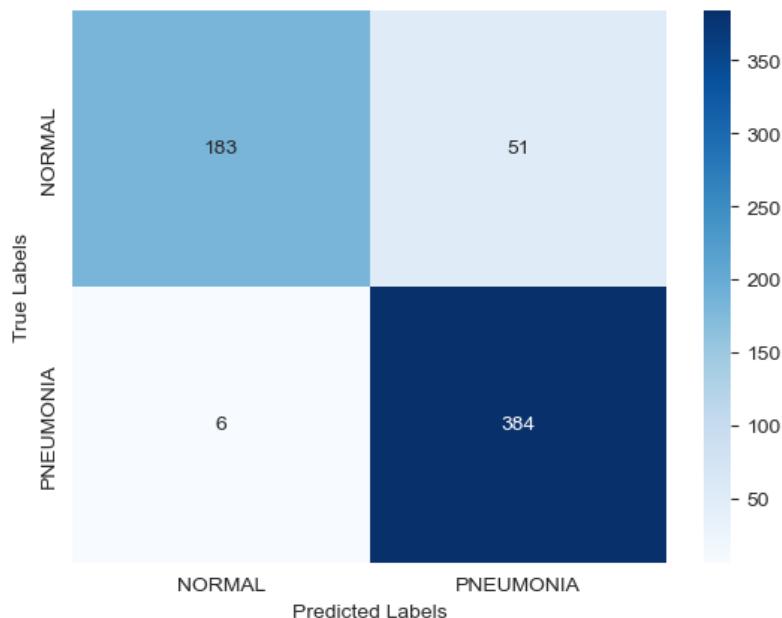
Dans cette implémentation, plusieurs choses sont à noter. D'abord, on crée le modèle ResNet50 sans poids pré entraînés (comme sur **ImageNet** par exemple); en effet, le but de l'expérimentation est de tester l'architecture en l'entraînant uniquement sur notre jeu de données.

On instancie également le modèle sans couches de neurones cachées (paramètre *include_top* à False) afin de pouvoir adapter paramétriquement la sortie de la classification en mode binaire.

Enfin, on ajoute notre propre de couches de neurones à l'architecture ResNet50 pour pouvoir obtenir une sortie de classification binaire.

3.5.3.3. Résultats

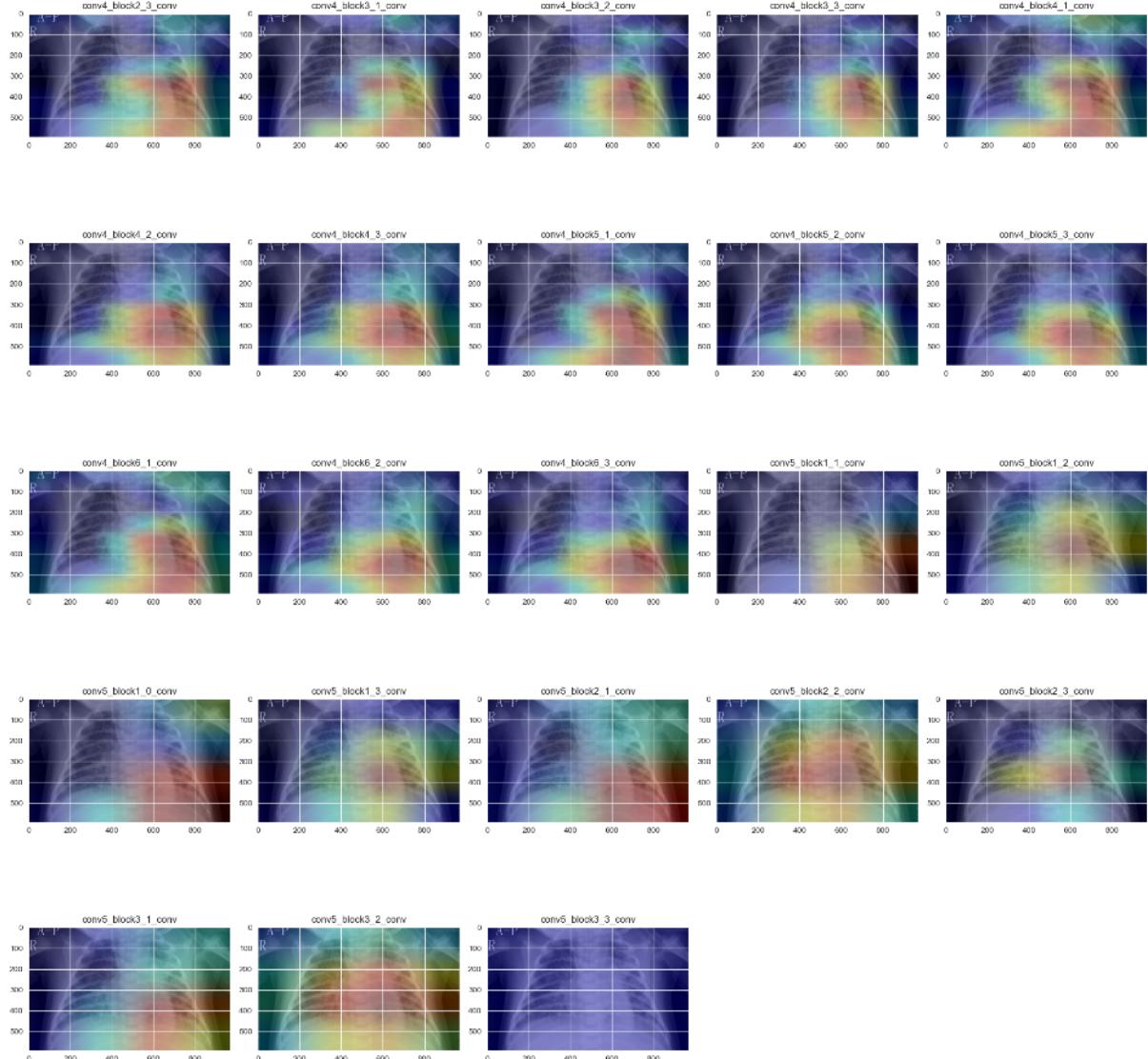
Après avoir entraîné notre modèle, on réalise des prédictions sur le jeu de données de test:



Matrice de confusion obtenue sur les données du jeu de test suite à l'entraînement du modèle basé sur l'architecture ResNet50

Cette matrice de confusion présente un nombre faux positifs plus faible que dans les résultats d'AlexNet et un nombre de faux négatifs quasi-nul. Au total, on observe une précision de 91%, ce qui se rapproche de notre modèle.

On observe une augmentation très intéressante de la précision par rapport à notre implémentation d'AlexNet. Par ailleurs, on obtient également un score de précision quasi-équivalent à notre modèle fait main, et ce, **sans pousser la recherche de paramètres optimaux**.

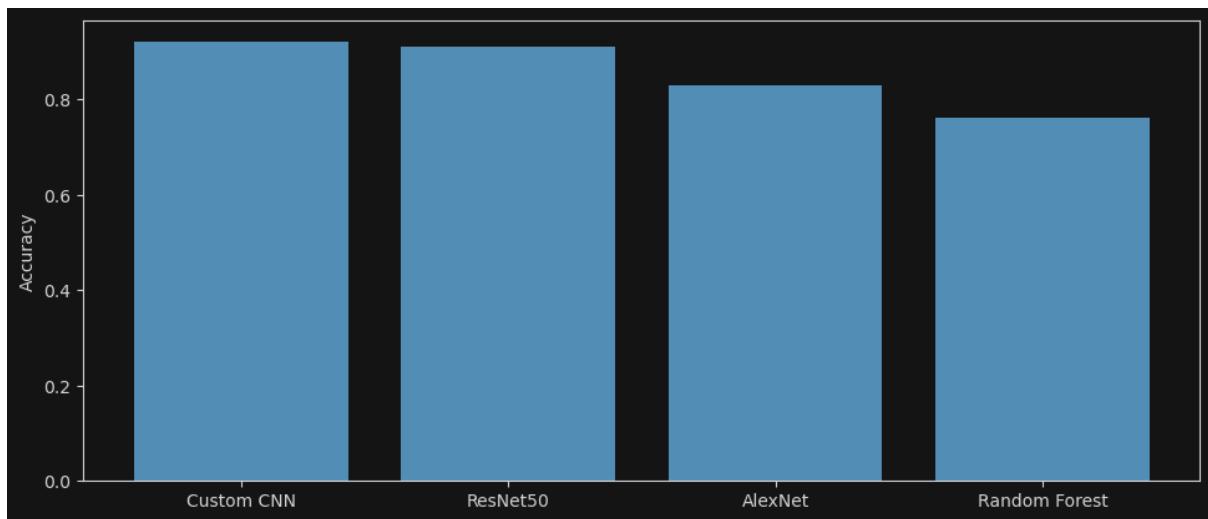


Extrait de heatmaps générés par couches par le modèle ResNet50 sur une image de test passée en entrée annotée “Pneumonie”

Il est aussi intéressant de noter que le modèle semble faire ses prédictions sur des éléments concrets des radios passées en entrée, ce qui semblait plus discutable lors de la visualisation des heatmaps générées par AlexNet.

Même si ResNet50 donne des résultats solides pour une complexité moyenne, le temps d'entraînement est largement supérieur à notre modèle fait main. Tous ces éléments montrent que le modèle ResNet50 est adapté à une classification à multiples labels, là où notre modèle custom aura plus de difficultés dans un cas de classification multiple.

En conclusion de toutes ces expérimentations sur différents modèles, on obtient les scores de précisions suivants:



3.5.4. Transfer Learning et ChestX-ray8

Une ouverture possible à l'amélioration de nos résultats et notamment l'application d'une classification à multiple labels se trouve dans deux éléments appéles *Transfer Learning* et *ChestX-ray8*.

Le *Transfer Learning* est une technique qui consiste à utiliser des modèles pré-entraînés sur de grandes quantités de données pour résoudre des tâches spécifiques. Au lieu de créer un modèle à partir de zéro, le *Transfer Learning* permet d'utiliser des connaissances acquises lors de l'entraînement sur des tâches similaires pour améliorer les performances sur une nouvelle tâche. Par exemple, nous avons décidé dans notre cas d'étude d'utiliser l'architecture ResNet50 sans poids pré entraînés, mais il est possible de charger les poids entraînés sur des jeux de données déjà établis, comme le jeu de données **ImageNet**. Dans notre cas spécifique, il existe un jeu de données qui semble correspondre à notre besoin, appelé *ChestX-ray8*.

Le jeu de données *ChestX-ray8* est un ensemble de données d'images médicales qui est souvent utilisé pour entraîner des modèles de détection de pathologies pulmonaires à partir de radiographies thoraciques. Le jeu de données contient environ 108 000 images de radiographies thoraciques de 32 717 patients, étiquetées avec huit types de pathologies différentes. Ces pathologies incluent la pneumonie, la tuberculose, la fibrose pulmonaire et d'autres maladies pulmonaires courantes. Le jeu de données est devenu une référence pour le développement et l'évaluation de modèles de détection de pathologies pulmonaires à partir de radiographies thoraciques utilisant le *Transfer Learning*.

Il serait donc pertinent d'explorer les prédictions pouvant être faites par un modèle connu comme ResNet50 qui aurait été pré entraîné sur ce jeu de données en amont.

4. Élaboration du modèle final

Afin d'élaborer une réponse finale au problème proposé dans le cadre du projet, nous allons donc maintenant nous appuyer largement sur les expérimentations effectuées auparavant. Ces dernières nous ont permis dans certains cas de chercher des valeurs de paramètres optimales, de nous pencher sur des métriques essentielles dans l'élaboration et l'amélioration du modèle, ou encore de soulever des problématiques essentielles dans l'obtention de résultats satisfaisants.

L'objectif est donc de proposer notre meilleure solution, mais avec la difficulté de combiner des valeurs optimales de paramètres obtenues lors d'expérimentations indépendantes les unes des autres, alors que certains d'entre eux influencent largement d'autres paramètres.

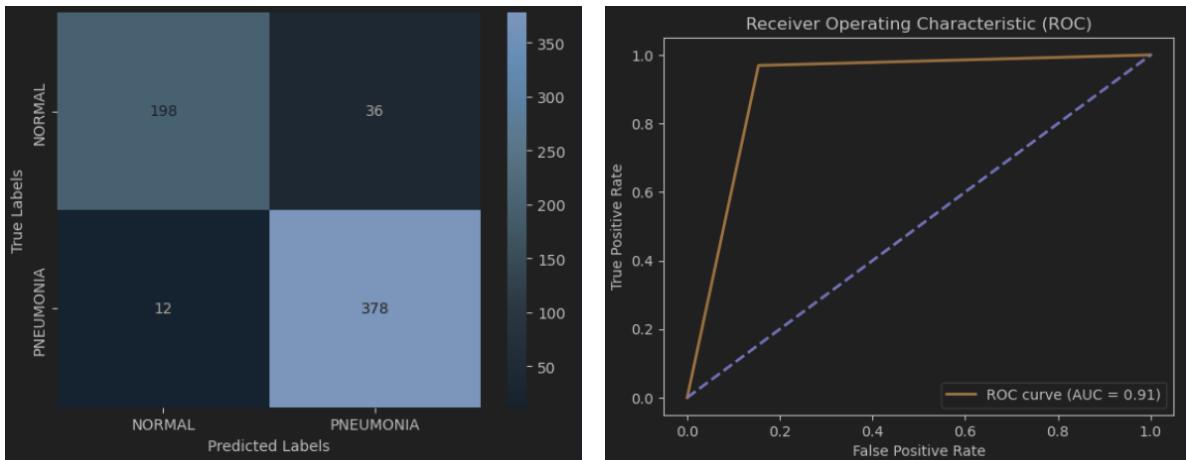
Voici donc les modifications effectuées :

- Nombre de folds: 15 (contre 10 précédemment)
- Epochs: 10 (contre 7 précédemment)
- Batch Size: 16 (contre 64)
- Drop out sur couches cachées: 0.1 (contre 0.5)
- Optimizer: RMSprop (contre "adam" précédemment)

À la suite de ce travail, nous pouvons relever les résultats suivants relevés sur le modèle final :

La matrice de confusion indique la présence de 12 cas prédits correspondant à des faux négatifs, contre 9 lors de la version antérieure du modèle, et 36 faux positifs contre 43 pour la version antérieure. On note une légère amélioration sur le nombre de faux positifs, mais une très légère régression sur les faux négatifs.

La courbe ROC reste très similaire à celle du modèle précédent, tout comme le score AUC qui passe de 0.90 à 0.91 pour la version finale. Cela représente une évolution marginale, mais qui reste néanmoins dans le sens d'une amélioration.



Matrice de confusion

ROC

On obtient exactement les mêmes scores de précision, *recall* et *f1 score*. Les modifications permettaient de manière individuelle d'atteindre une précision de **0.93**. On peut conclure que l'accuracy maximale de notre modèle customisé se situe entre **92% et 93%** avec un nombre de faux négatifs inférieur au nombre de faux positifs (12 contre 36).

	precision	recall	f1-score	support
NORMAL	0.94	0.85	0.89	234
PNEUMONIA	0.91	0.97	0.94	390
accuracy			0.92	624
macro avg	0.93	0.91	0.92	624
weighted avg	0.92	0.92	0.92	624

Rapport de classification pour le jeu de données de test

5. Classification multiclasse

5.1. Changements apportés

Une des ouvertures possibles du sujet était de transformer notre cas de classification binaire en classification multiclasse, en classifiant les radios “normales”, présentant une pneumonie virale ou présentant une pneumonie bactérienne. Nous avons alors testé si notre architecture était capable dans une moindre mesure de répondre à ce cas de classification. Afin de transformer notre *template* de classification binaire en classification multiclasse, plusieurs modifications sont nécessaires à différents niveaux.

id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

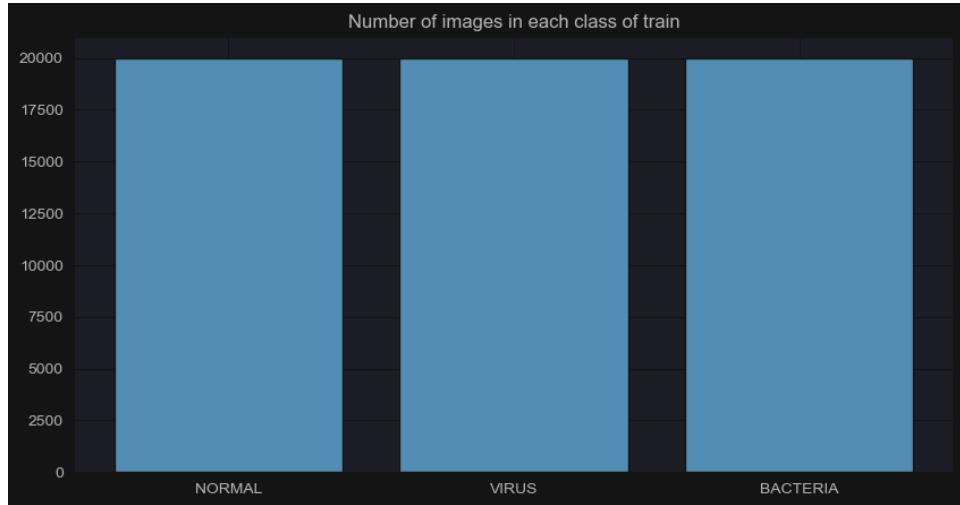
id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

Schéma expliquant l'effet du One Hot Encoding

Tout d'abord, dans l'étape de préparation et normalisation des données, on utilise une technique appelée *One Hot Encoding* sur nos labels; c'est une technique de prétraitement des données qui permet de représenter des catégories sous forme de vecteurs binaires pour les utiliser dans des algorithmes de machine learning.

Supposons que nous ayons une variable catégorielle avec k catégories différentes. Nous allons créer k variables binaires distinctes, une pour chaque catégorie. Chaque variable binaire aura une valeur de 1 si l'exemple de données appartient à la catégorie correspondante, et une valeur de 0 sinon (comme montré sur le schéma).

Nous avons ensuite séparé l'ensemble des images de pneumonies en deux sous-dossiers pour chaque jeu de données: un dossier regroupant les pneumonies virales et un dossier regroupant les pneumonies bactériennes.



Au fur et à mesure des expérimentations, nous avons également augmenté le nombre d'images augmentées, de 5000 à 20000 pour tenter d'aider le réseau de neurones à mieux différencier les différentes classes.

Une fois avoir adapté les étapes de préparation de données, il est essentiel de modifier la configuration de l'architecture de notre modèle. La dernière implémentation testée est décrite ci-dessous.

```
model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(7,7), activation="relu"))
model.add(MaxPooling2D(pool_size=(4,4)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(5,5), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())

model.add(Dense(256, activation="relu"))
model.add(Dropout(0.4))

model.add(Dense(128, activation="relu"))
model.add(Dropout(0.4))

model.add(Dense(len(subfolders), activation="softmax"))

# compile the model
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=metrics)
```

On remarque plusieurs changements cruciaux:

- la dernière couche de neurones a désormais **autant de neurones que de classes à prédire** et sa fonction d'activation est passée de **sigmoid** à **softmax**
- la fonction de coût est passée de **binary_crossentropy** à **categorical_crossentropy**

Au fur et à mesure des tests, d'autres modifications ont été apportées:

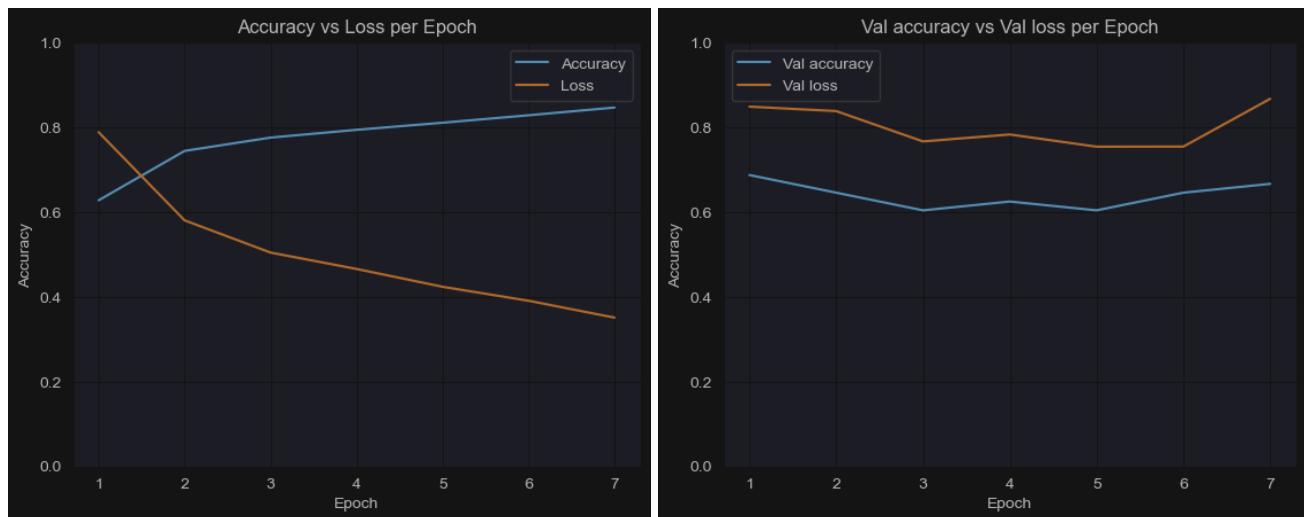
- un changement de la taille des filtres pour les deux premières couches de convolutions (respectivement 7 par 7 et 5 par 5)
- un changement de la taille du *pooling* de la première couche de *pooling* (4 par 4)
- Une baisse légère du *drop out* (0.5 vers 0.4)

Parmi les hyperparamètres restants, seul le *batch size* s'est vu être modifié (de 64 à 192), le nombre d'époques restant inchangé (7).

5.2. Observations

Après plusieurs entraînements et évaluations, quelques comportements ont été observés. Ces comportements empêchent de valider notre modèle et sont preuves d'anomalies.

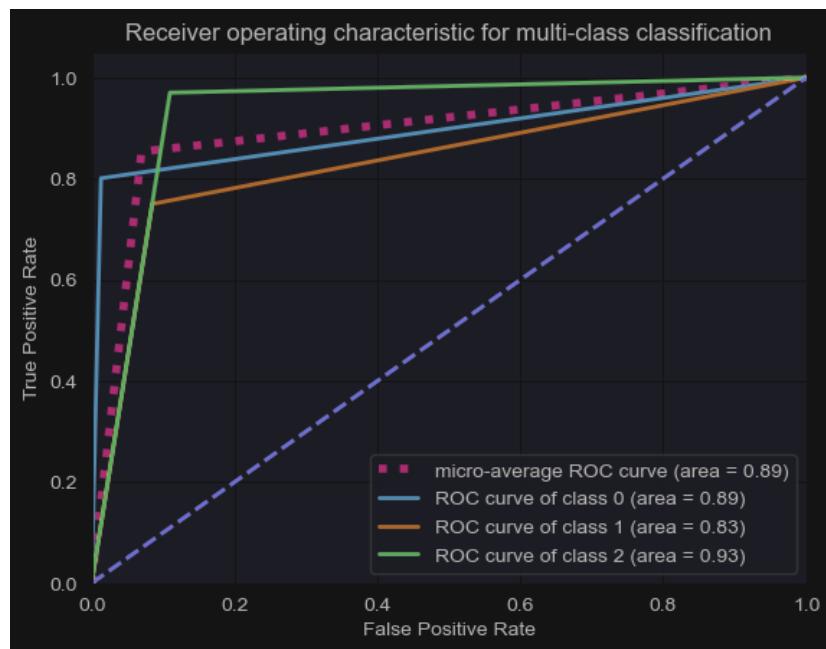
Tout d'abord, lors de l'entraînement du modèle, il semblerait que la précision et le coût sur le jeu de données de validation soit indépendants de la précision et du coût sur le jeu d'entraînement.



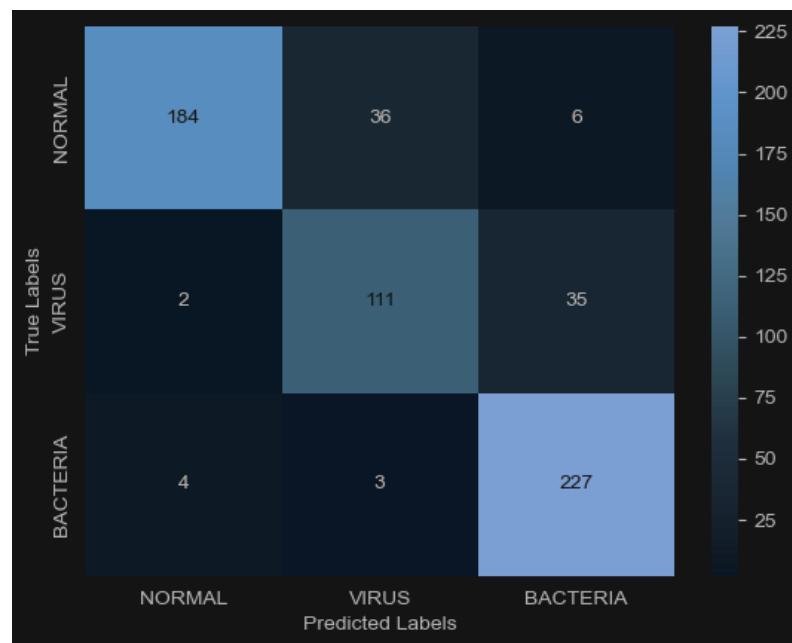
A gauche, les courbes de précision et de coût sur le jeu d'entraînement par époque. A droite, les courbes de précision et de coût sur le jeu de validation par époque.

Nous avons alors remplacé le jeu de données de validation par le jeu de données de test pour faire la validation, et ce comportement n'était pas observé. **Une hypothèse serait alors que l'incident provient des images de validation; on remarque d'ailleurs qu'il n'y a aucune image de pneumonies virales dans ce dernier et qu'il a fallu en transférer manuellement depuis le jeu de données de test.**

Un autre problème identifié est la précision obtenue pour la classe “Pneumonie virale”, qui est largement inférieure aux deux autres classes et qui fait alors baisser la précision de notre modèle.

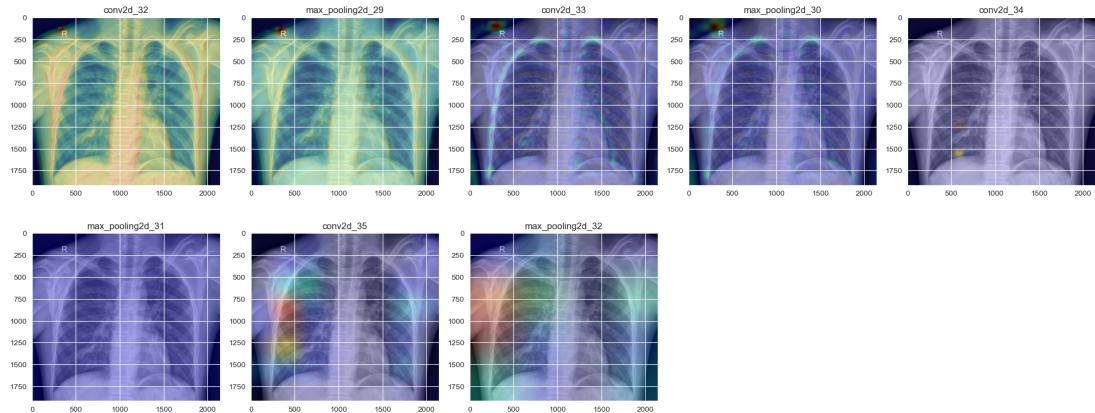


Selon la courbe ROC pour la classe 1 (“Pneumonie virale”), on remarque que la valeur de l’AUC est largement inférieure aux deux autres classes



La matrice de confusion montre que beaucoup d’erreurs de prédictions gravitent autour de la classe “Pneumonie virale”

Au final, le modèle propose une précision de 89%, mais une précision pour la classe “Pneumonie virale” de seulement 74%. De plus les heatmaps montrent des résultats moyennement cohérents, notamment pour la classification des radios annotées “normales”:



Heatmaps produites par notre modèle pour une image de test annotée “normale”

Néanmoins, dans le cas d'une classification multiclasse, il est possible d'observer quelle classe est prédict en majorité, mais aussi quelle classe est prédict en deuxième option. Pour 3 images de classes distinctes, on a observé ces résultats de prédictions:

Classe	Première prédition	Seconde prédition	Prédictions proches ?
Normale	Normale	Virale	Non (562 et -397)
Virale	Bactérienne	Virale	Oui (144 et -15)
Bactérienne	Bactérienne	Normale	Oui (33 et -16)

On remarque que la prédition de la radio annotée “normale” s'est faite très largement.

En revanche, la prédition de l'image annotée “virale” a échoué car le modèle n'a prédict la bonne classe qu'en seconde position. **Néanmoins, les valeurs de prédition sont relativement proches, ce qui laisse à penser que le modèle confond les pneumonies bactériennes et virales, mais tend vers la prédition des pneumonies bactériennes.**

Enfin, on remarque aussi que l'image annotée “bactérienne” a été correctement prédict, mais que la seconde prédition (dite “normale”) est très proche en terme de valeur, ce qui montre une deuxième faiblesse de notre classification. **Notre modèle sait prendre des décisions tranchées pour les radios dites “normales”, mais pas celles présentant une pneumonie.**

Cet ensemble d'anomalies dans la classification empêche de valider la solidité de notre modèle. Une des possibilités envisagées pour régler les problèmes rencontrés serait d'utiliser un modèle pré entraîné sur le jeu de données ChestX-ray8 et très efficace dans la reconnaissance d'image, car notre architecture ne semble pas assez poussée et montre des limites pour la classification de pneumonies virales ou bactériennes. **L'utilisation d'un modèle pré entraîné pourrait permettre de déterminer si le problème vient des données, de leur préparation ou bien du modèle utilisé.**

6. Conclusion

Pour conclure, il est important de prendre du recul sur le travail de recherches et d'expérimentation réalisé à l'issue du projet. Nous pouvons noter quelques points forts de notre approche :

- La quantité de pratique et de recherches effectuées ont permis de réaliser une entrée en matière dans le domaine de l'intelligence artificielle, par la réalisation d'une solution au problème énoncé par le sujet du projet.
- Une mise en commun d'un ensemble de paramètres pour la réalisation du modèle a permis de disposer d'une base comparative pour l'évolution de ces derniers, réalisable par tous les membres du groupe.
- Le découpage des différents types de paramètres ajustables a permis d'étudier les résultats qu'ils engendraient de manière indépendante, et ainsi d'étudier leur impact, apporter des points de vigilance et parfois même de trouver leurs valeurs optimales.
- L'utilisation de solutions se basant sur différents modèles connus / classifieurs, ou même avec des CNN customisés, nous a permis de prendre du recul et de conforter nos choix les plus essentiels dans la réalisation de notre modèle final.

Nous pouvons également noter la présence de certains points à améliorer :

- La modification de différents paramètres indépendamment des autres peut présenter des limites si certains doivent se voir combiner avec d'autres paramètres afin de présenter ses vrais effets.
- Dans le contexte de la découverte de ce domaine, d'autres métriques ou représentations graphiques pourraient venir à manquer pour compléter l'étude du sujet.
- Le modèle final obtenu présente des indicateurs de performance satisfaisants dans l'absolu, mais présente seulement une légère amélioration par rapport à notre modèle de base.

Notre solution répond donc au problème posé par le sujet du projet, même si ses résultats sont améliorables. L'intérêt dans cette étude réside dans le cheminement qui nous a conduit à ce modèle final.