Rapport projet Kaggle : "Chest X-ray (Pneumonia) - CNN & Transfer Learning"

Jamay Théo & Goulancourt Rebecca
M2BI

I. Introduction

Le but de ce projet est de pouvoir déterminer si des patients sont atteints ou non de la pneumonie avec pour seule information le scanner de leur poumon sans avoir à demander un avis médical. Pour cela nous allons coder un script python qui va lire les images et émettre une conclusion quant à la santé des patients : sain ou atteints de pneumonie. Ce script sera écrit sur la base du "Deep Learning", méthode consistant à créer un réseau neuronal convolutif (CNN) qui prendra en input des images de scanner et pourra ainsi prédire si ces images appartiennent à des sujets malades.

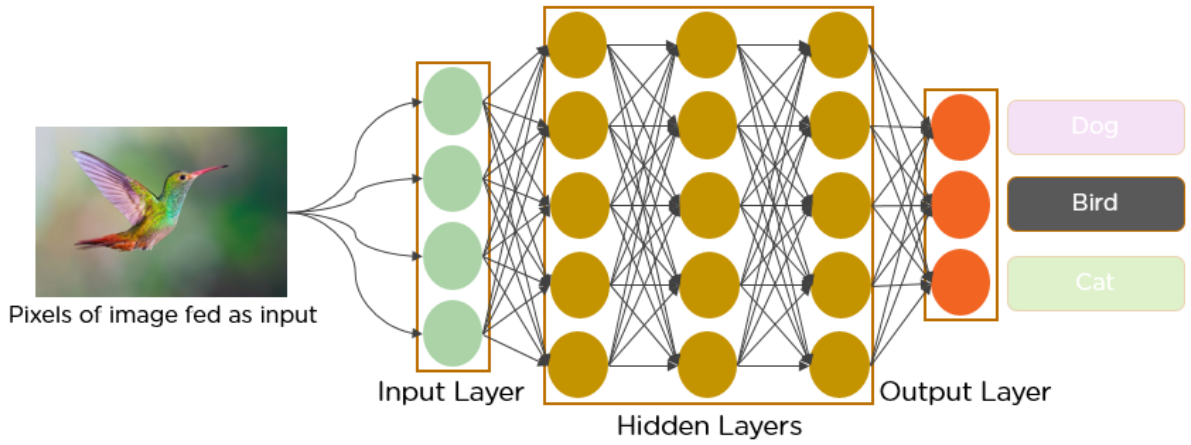


Figure 1 : Schéma simplifié d'un réseau neuronal convolutif (CNN).

Un CNN est constitué d'une ou plusieurs couches de neurones (ou layers) qui récupèrent de l'information des neurones précédents pour en déduire une nouvelle information. Le but d'un réseau neuronal est d'apprendre sur des données existantes pour pouvoir effectuer des prédictions sur d'autres jeux de données. Ainsi les résultats obtenus seront dépendant de la qualité du modèle de prédiction et de l'apprentissage effectué sur un jeu de données initiales. Les prédictions peuvent aller de la classification à la régression, la classification consistant à avoir des prédictions de type binaire ou multiple comme par exemple déterminer quelle espèce d'animal est présent sur la photo tandis que la régression peut permettre par exemple

une reconnaissance faciale. Dans notre cas, il est nécessaire de faire de la classification car on souhaite déterminer si des individus sont atteints de pneumonie ou non. Le CNN apprend en ajustant les poids qui sont des facteurs multiplicatifs représentant l'importance du travail de chaque noyau. Les poids relient chaque neurone entre eux, c'est par là que passe l'information. Ainsi toutes les couches sont connectées à la couche suivante ou précédente, à l'exception des premières et dernières couches. Chacune des couches reçoit donc une somme de données en input qu'il transforme et renvoie sous forme d'output à la couche suivante. La dernière couche est appelée la couche "*fully-connected*" ou "*Dense Layer*" car elle est connectée à toutes les couches et permet de donner le résultat de la classification.

II. Matériel et méthodes

On dispose d'un jeu de données organisé de la manière suivante :

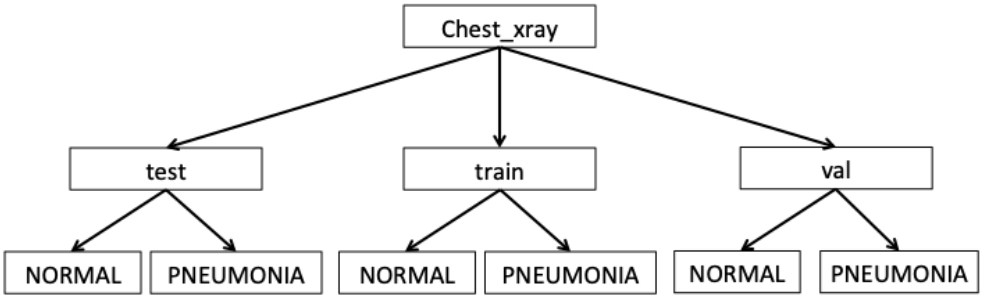
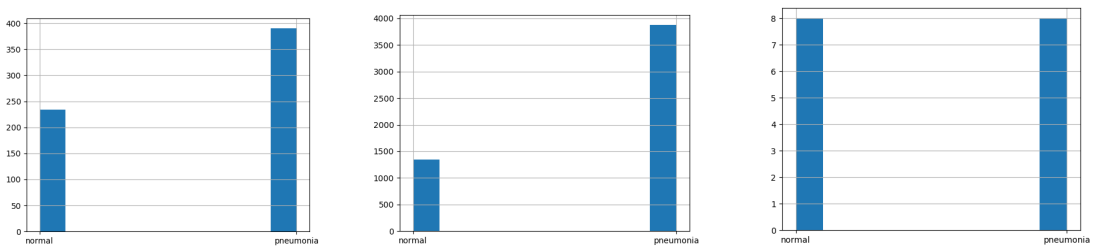


Figure 2 : Organisation des données.

Ainsi chaque dossier contient des images de scanner de patients. Pour chaque dossier test, train et val (jeux de données de test, d'apprentissage et de validation) on dispose donc d'images de patients sains et malades. Les inputs et output de notre modèle de CNN seront donc des variables qualitatives binaire. Nous commençons par visualiser la répartition de nos données. Le code ayant servi à produire ces histogrammes est donné en annexe.



(a) (b) (c)

Figure 3 : histogramme de la répartition des données dans chacun des jeux test (a), train (b) et validation (c).

On constate que pour les dossiers train et test il y a une proportion plus importante d'image de pneumonie que d'images de poumons normaux. Ce déséquilibre n'est pas un problème car il est compensé par le fait que nous ayons quand même un grand nombre d'images de poumons normaux (supérieur à 1000). Ensuite, on constate que le dossier de validation contient seulement 16 images ce qui est très peu comparé aux plus de 5000 images contenues dans le dossier train. Premièrement, nous avons testé différents notebook disponibles sur kaggle pour voir les différentes approches qui ont été utilisées afin de résoudre la problématique. Nous nous sommes rendu compte lors de nos tests qu'il y avait trop peu d'images dans le dossier validation pour avoir des performances sur le modèle cohérentes. Nous avons donc cherché à séparer nous même nos données de validation sans passer par le découpage prédéfini dans les dossiers. Dans un premier temps, nous avons essayé de lire nos images via le package cv2 et ensuite de les convertir en tableau numpy. Nous avions une liste X_train contenant nos images converties en tableau numpy et une liste y_train contenant les labels 0 si l'image correspond à un poumon normal et 1 s'il était atteints de pneumonie.

Seulement pour créer une dataset de validation issue de X et y train nous avons utilisé la fonction train_test_split (package sklearn) qui provoquait le plantage de nos ordinateurs personnels. Nous avons donc tenté une autre approche en concaténant les images contenues dans les dossiers train, test, val sous forme de liste pour ensuite les transformer en dataframes. En parallèle de la rédaction de ce script nous avons découvert la fonction *flow_from_directory()* issue du module ImageDataGenerator. Cette fonction, lorsqu'elle est appelée, parcourt un chemin précis et retourne une dataset d'images répartis en deux sous-répertoires étiquetés 0 et 1. Utiliser *flow_from_directory* associé à ImageDataGenerator nous a permis de normaliser nos images et de faire de la data augmentation assez simplement. De plus on a pu générer rapidement et simplement un sous ensemble de validation issus de notre ensemble d'images train via le paramètre validation_split. Cet aspect pratique et ces paramètres assez intuitifs nous ont convaincu de privilégier cette approche plutôt que nos précédentes tentatives avec train_test_split et la concaténation en dataframe.

Nous avons ensuite cherché à faire de l'augmentation de données (data augmentation) via ImageDataGenerator. Cette étape consiste à générer de nouvelles données à partir d'images déjà existantes de notre jeu de données en modifiant certains paramètres tels que la rotation de l'image (rotation_range = 20), le zoom (zoom_range = 0.1), ... Nous avons testé différents paramètres et évalué l'impact de ces paramètres sur les performances de notre

modèle. Nous avons par exemple constaté que le paramètre brightness_range semblait diminuer les performances de notre modèle.

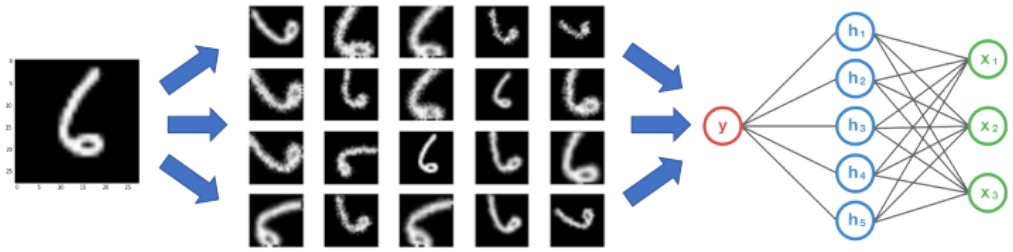


Figure 3 : Schéma explicatif de la DataAugmentation. Une image est prise et modifiée pour créer de nouvelles images puis ajoutées aux données totales.

Nous avons ensuite construit un modèle de type CNN pour prédire si une image correspond à un poumon malade ou normal. Les données en input subiront en premier lieu des étapes de convolution, c'est-à-dire que les images subiront des modifications en appliquant des filtres de pixel, cela correspond en réalité à une "couche" neuronale. Cette étape est faite à l'aide de la fonction .Conv2D() qui va en réalité produire une matrice appelée *feature map* issue d'un filtre (ou kernel, ou noyaux du réseau) représentant une matrice de dimensions NxN parcourant l'image.

Les fonctions d'activation sont spécifiées dans la couche convolutionnelle (*.Conv2D(activation = 'relu' ou 'softmax')*) et permettent au neurone de transformer l'output à donner à la prochaine couche pour avoir une nouvelle approche des données. En effet un neurone effectue une somme des informations qu'il récupère et grâce à une fonction d'activation il calcule et transmet une nouvelle information au neurone suivant. Les fonctions d'activations utilisées ici seront les fonctions ReLu (Rectified Linear Unit) et softmax. Elles servent à modifier les données de manière non linéaire afin de modifier la perspective des données vue par le modèle, chaque couche possédant une fonction d'activation propre. Toutes nos couches auront pour fonction d'activation ReLu à l'exception de la Dense Layer. En effet la fonction ReLu va effectuer un filtre des output de chacunes des couches du réseau de neurones car elle rend x si $x > 0$ donc elle ne laisse passer que les valeurs positives à la couche suivante, c'est pour cela qu'elle n'est utilisée que dans les couches intermédiaires. Pour ce qui est de la dernière couche, la fonction softmax est utilisée car elle permet de résoudre des problèmes de classification, ce qui est notre cas. En réalité elle donne en sortie

une somme des valeurs reçue de la couche précédente, c'est une probabilité donc la somme finale sera égale à 1 car softmax attribue à chaque classe une probabilité.

(a) (b)

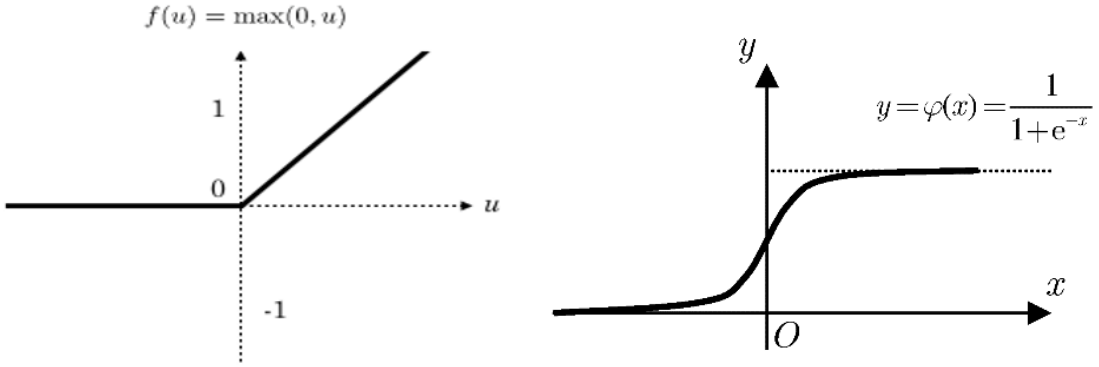


Figure 3 : Aperçu des fonctions ReLu (a) et Softmax (b)

Nous avons testé de manière empirique de nombreux paramètres différents pour notre modèle. Ainsi nous avons commencé par créer un modèle le plus simple possible pour ensuite monter progressivement en complexité. Nous avons cependant rencontré un problème de temps sur nos machines personnelles même avec un faible nombre paramètres l'entraînement de nos modèles prenait au minimum 20 minutes

Nous avons testé la "*Batch Normalization*" des données qui permet de normaliser les inputs et de les mettre sur une même échelle pour améliorer la vitesse et les performances du modèle. Cette BatchNormalization est un type de normalisation effectuée entre chaque layer et non directement sur les données brutes afin d'améliorer la vitesse d'apprentissage. Les données sont normalisées tel que :

$$z^N = \frac{(z - m_z)}{s_z}$$

Figure 4 : Formule de calcul l'output après Batch Normalisation; avec z^N l'output après Batch Normalisation, m_z la moyenne de la sortie du neurone et s_z sa variance.

Nous avons également testé le Maxpooling, cela consiste à réduire les dimensions des données d'image en diminuant le nombre de pixels de l'output correspondant à la couche précédente. Ce filtre est composé d'une dimension précise qui se déplace sur les pixels de l'image. Il récupère alors un bloc d'information de la même dimension faisant partie de l'image et garde la valeur maximale de ce bloc, ce processus est réitéré jusqu'à avoir traversé toute l'image et résulte en un nouvel output (ou image) à donner à la couche convolutionelle

suivante. Le Maxpooling est appliqué aux données avec la fonction .MaxPooling2D qui va effectuer un maxpooling de dimensions 2x2 (pool_size = (2, 2)), la valeur par défaut de 1 du stride a été laissée ce qui signifie que la matrice de pooling avance d'un pixel à la fois. Nous avons utilisé padding = "same" afin que les output produits aient la même dimension que les données prises en entrée.

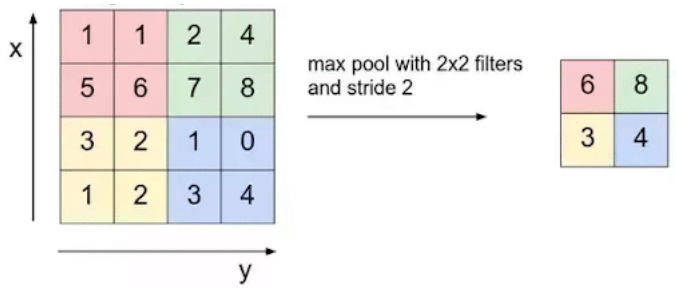


Figure 5 : Schéma explicatif du Maxpooling pour une poolsize 2x2. Uniquement les valeurs maximales de chaque bloc de couleurs sont gardées.

Enfin nous avons essayé différentes valeurs de dropout. Ce dropout permet d'éviter le surapprentissage des données lors de l'entraînement du modèle. En réalité le dropout "désactive" certains neurones aléatoirement à chaque époque/epoch (1 époque = 1 cycle où le modèle s'est entraîné sur la dataset entière) ainsi que les connexions le liant aux autres neurones. Ainsi, si nous avons un dropout de 0.5 cela signifie qu'à chaque époque 50% des neurones et leur connexions seront désactivés aléatoirement.

Nous avons ensuite testé différents paramètres pour compiler notre modèle. Nous avons testé l'utilisation d'un learning rate de 10^{-3} et de 10^{-5}. Nous avons essayé les optimizer Adam, Adamax et enfin Adagrad. Ensuite pour l'entraînement de notre modèle nous avons fixé le nombre d'époques à 50 et nous avons défini un callback (*EarlyStopping*). Le callback permet de réduire le surapprentissage en arrêtant l'apprentissage du modèle plus tôt lorsque la valeur surveillée ne s'améliore pas. En effet, trop d'époques peuvent mener au surapprentissage tandis que trop peu d'époque peuvent mener à un sous-apprentissage. Le callback permet également un gain de temps important en évitant de tourner sur 50 époques inutilement. Nous avons testé le callback avec différentes valeurs de patience (3,5 et 10) et nous avons finalement fixé la patience à 3 ce qui signifie que le modèle arrêtera d'apprendre après 3 époques sans amélioration de ses performances en validation.

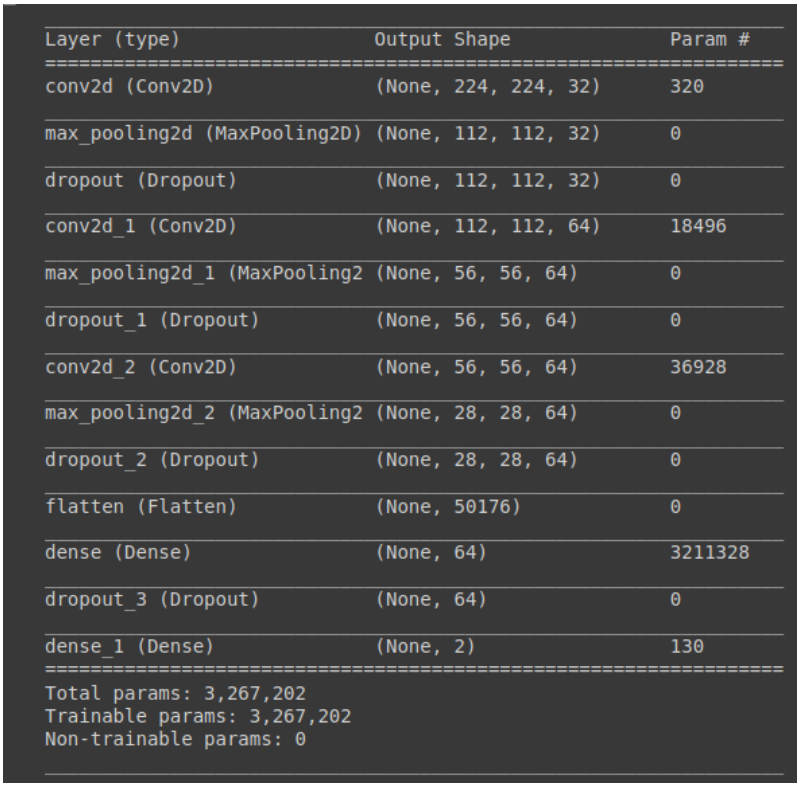
À la suite de cela nous évaluons les performances de notre modèle à l'aide de la loss et de l'accuracy issue du jeu de données test, obtenues grâce à la fonction .evaluate() de

Tensorflow. De manière générale, la loss définit la performance du modèle et peut être calculée sur les données de train, test et validation. Elle permet d'avoir un aperçu de la qualité du CNN, en effet plus la loss est faible plus le modèle sera performant dans sa prédiction. Elle est caractérisée par une somme des erreurs faites sur chaque échantillon, c'est en quelque sorte un degré d'erreur. Quant à l'accuracy, elle correspond au nombre de bonnes prédictions que le modèle a effectué sur l'échantillon après comparaison avec les résultats attendus. Un pourcentage d'accuracy est ensuite calculé. C'est-à-dire que sur 100 images, si 99 images sont correctement prédites alors le taux d'accuracy est de 99%.

Le but serait alors de modifier les paramètres pour in fine avoir les meilleures performances possibles, visualisables avec une loss très faible et une accuracy élevée. En complément, nous avons essayé une méthode appelée "*Transfer Learning*". C'est une méthode d'apprentissage qui consiste à récupérer la valeur des poids d'un bon modèle qui a déjà appris sur un autre jeu de données. Cela permet donc d'obtenir des paramètres de CNN correct afin de partir à priori sur un modèle qui prédit correctement les données.

III. Résultats & Discussion

Notre modèle de CNN a donc appris sur des images de poumons. On peut voir ci-dessous le résumé du réseau que nous avons créé. Nous avons choisit d'utiliser trois couches Conv2D et 2 couches dense avec des étapes des Maxpooling et Dropout.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 224, 224, 32)	320
max_pooling2d (MaxPooling2D)	(None, 112, 112, 32)	0
dropout (Dropout)	(None, 112, 112, 32)	0
conv2d_1 (Conv2D)	(None, 112, 112, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 56, 56, 64)	0
dropout_1 (Dropout)	(None, 56, 56, 64)	0
conv2d_2 (Conv2D)	(None, 56, 56, 64)	36928
max_pooling2d_2 (MaxPooling2	(None, 28, 28, 64)	0
dropout_2 (Dropout)	(None, 28, 28, 64)	0
flatten (Flatten)	(None, 50176)	0
dense (Dense)	(None, 64)	3211328
dropout_3 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 2)	130

Total params: 3,267,202
Trainable params: 3,267,202
Non-trainable params: 0

Figure 7 : Descriptif du réseau construit.

On constate l'impact du MaxPooling sur les output avec une réduction en taille d'un facteur 2. En effet, le nombre de pixel diminue de 224 à 112 (224/2), puis de 112 à 56 (112/2) jusqu'à atteindre une taille de 28x28 pixels. L'étape de flatten permet le passage de 2D à 1D. À la fin du processus la dernière couche nous donne un output de taille 2 car nous avons 2 classes à prédire : normal ou pneumonia. Quant aux paramètres ils représentent le nombre de paramètres entraînés au cours de l'étape convolutive. Aucuns paramètres ne sont entraînés durant le MaxPooling ou de dropout car ces étapes ne nécessitent pas d'apprentissage, de même pour la couche de flatten. Théoriquement, il serait idéal d'avoir le moins de paramètres entrainables possibles pour améliorer les performances et la vitesse d'exécution du modèle. Cependant du fait que nous utilisons un générateur via la fonction *flow_from_directory* cela ralentit l'apprentissage de notre modèle. Ainsi même en passant de ~700 000 paramètres à plus de 3 millions une époque prend ~78 sec pour se terminer sur Google Colab (1 GPU). Initialement nous faisions nos tests sur nos machines personnels et nous ne pensions pas que c'était *flow_from_directory* qui ralentissait l'entraînement de notre modèle mais bien la faible puissance de calcul de nos machines.

```
Epoch 1/50
147/147 [==============================] - 78s 511ms/step - loss: 0.5152 - accuracy: 0.7638 - val_loss: 0.3878 - val_accuracy: 0.8081
Epoch 2/50
147/147 [==============================] - 75s 511ms/step - loss: 0.2998 - accuracy: 0.8733 - val_loss: 0.4118 - val_accuracy: 0.8061
Epoch 3/50
147/147 [==============================] - 76s 516ms/step - loss: 0.2617 - accuracy: 0.8907 - val_loss: 0.2531 - val_accuracy: 0.8829
```

```
Epoch 8/50
147/147 [==============================] - 76s 514ms/step - loss: 0.2071 - accuracy: 0.9182 - val_loss: 0.2281 - val_accuracy: 0.9079
Epoch 9/50
147/147 [==============================] - 76s 515ms/step - loss: 0.2028 - accuracy: 0.9216 - val_loss: 0.2383 - val_accuracy: 0.8944
```

Figure 8 : Apprentissage du modèle sur le jeu train. Les trois premières et deux dernières époque sont représentées. Le temps de calcul pour l'époque est visible ainsi que les valeurs calculées de loss et d'accuracy sur les dataset de train et validation.

Nous pouvons observer en premier lieu que l'apprentissage du modèle s'arrête au bout de la neuvième époque ce qui est dû à l'EarlyStopping. Pour avoir une meilleure visualisation de l'apprentissage, nous avons représenté l'évolution de la loss et de l'accuracy sous forme d'un graphe :

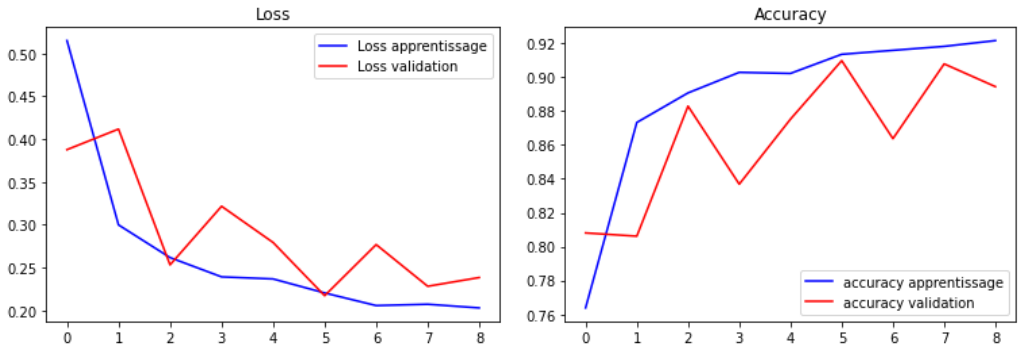


Figure 9 : Graphe représentant la loss et accuracy sur les échantillons d'apprentissage et de validation au cours des époques.

On remarque que la loss sur l'échantillon d'apprentissage possède une courbe caractéristique d'un bon apprentissage car le modèle minimise les erreurs (loss) au cours des époques. Toutefois sur la dataset de validation on observe une diminution progressive ponctuée par des augmentations.

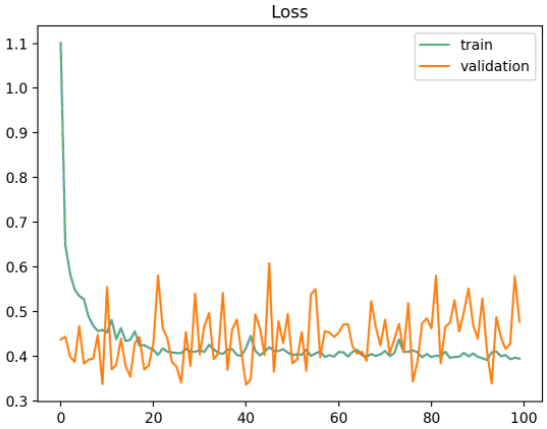


Figure 10 : Graphe exemple représentant la loss de validation et de train en fonction des époques

La figure 10 est un exemple vu en cours qui montre l'impact d'un jeu de validation trop faible sur la courbe de validation loss. On retrouve la même allure en dent de scie sur nos graphes d'accuraccy et de loss, ainsi cela pourrait être le signe qu'il n'y a pas suffisamment d'images dans notre validation. Il pourrait également s'agir d'un déséquilibre entre le nombre d'images de poumons normaux et le nombre d'images de pneumonie dans le jeu de validation.

Pour ce qui est de la figure 9 on constate que l'accuracy tend vers 92% pour le jeu d'apprentissage et on retrouve une courbe en dent de scie pour le jeu de validation. Cela nous confirme dans notre hypothèse qu'il faudrait augmenter la taille du jeu de validation. Enfin nous avons évalué notre modèle sur la jeu de test à l'aide de la fonction .evaluate().

```
624/624 [==============================] - 8s 12ms/step - loss: 0.3213 - accuracy: 0.8638
Test Loss : 0.32
Test Accuracy : 86.38
```

Figure 11 : Résultats des performances obtenues sur la dataset de test..

On observe à l'aide de la figure 11 une loss de 0.32 et une accuracy de 86.38% sur les données de test. Cette loss reste semblable à celle obtenue sur l'échantillon de validation. Une accuracy de 86% signifie que le modèle a pu prédire correctement 86% des images de la dataset test. Nous avons réalisé plusieurs tests et nous constatons que l'accuracy pour le jeu test de notre modèle varie entre ~81% et ~87%. Cette variation peut s'expliquer par un nombre d'époques pas suffisamment important pour que notre modèle apprenne correctement. On peut aussi émettre l'hypothèse que la composition en images de notre jeu de validation est trop variable d'une exécution à l'autre. Ainsi un jeu de validation homogène donne un meilleur apprentissage qu'un jeu de validation hétérogène (ici nous avons beaucoup plus d'image de pneumonie que d'image normale).

IV. Conclusion

Nous avons construit un modèle CNN avec une performance sur le jeu test autour des ~ 85%. Nous nous sommes rendu compte lors des tests sur Google Collab que la fonction *flow_from_directory* ralentissait probablement l'apprentissage de notre modèle. Ensuite le paramètre validation_split bien que plus performant comparé à l'utilisation des 16 images contenues dans le dossier de validation donne des résultats variables. Nous nous posons donc

la question s'il aurait peut être été plus satisfaisant de rester sur une approche classique avec X_train, y_train, X_val, y_val et X_test, y_test.

Malgré ces défauts la fonction *flow_from_directory* nous a permis d'obtenir de bonnes performances et cela même sur des machines avec peu de puissance de calcul. Ensuite cela reste une fonction simple à prendre en main et avec un choix de paramètres complets.

ANNEXES :

Le code de notre modèle est disponible sur Github:
https://github.com/TheoJamay/chest_xray.git

Le code ayant servi aux premières réflexions du projet et au tout premier modèle construit est disponible sur Github à l'adresse suivante : https://github.com/Rebbekkah/Kaggle.git