Réseau de neurones appliqué à des images de radiographie pulmonaire (cas de Pneumonie)

Introduction :

Le but de ce projet est de réaliser une classification, entre les individus sains et les individus atteints de pneumonie, à partir d'images de radiographie pulmonaire de patients. Pour cela, nous avons construit un réseau de neurones, composé de plusieurs couches de convolution, que nous avons entraîné pour distinguer les patients sains des patients malades en utilisant une partie des images qui nous a été fourni. Cette méthode nous a permis d'avoir une classification automatisée des individus sains et malades très fiable à partir d'un jeu d'images restreint.

Matériel et Méthodes :

Matériels :

Le jeu de données fourni est composé de 3 séries d'images de radiographie pulmonaire. La première série, qui sera donnée à notre réseau pour s'entraîner, contient 5 216 images dont 1 341 proviennent de patients sains et 3 875 de patients malades. La seconde, qui contient 624 images dont 234 cas "normal" et 390 cas "pneumonie", sera utilisée afin de tester notre modèle une fois ce dernier entraîné. Enfin, la dernière série contenant 16 images, n'a pas été utilisée de part sont trop faible nombre d'images. Nous avons préféré utiliser 10% du jeu d'entraînement comme jeu de validation.

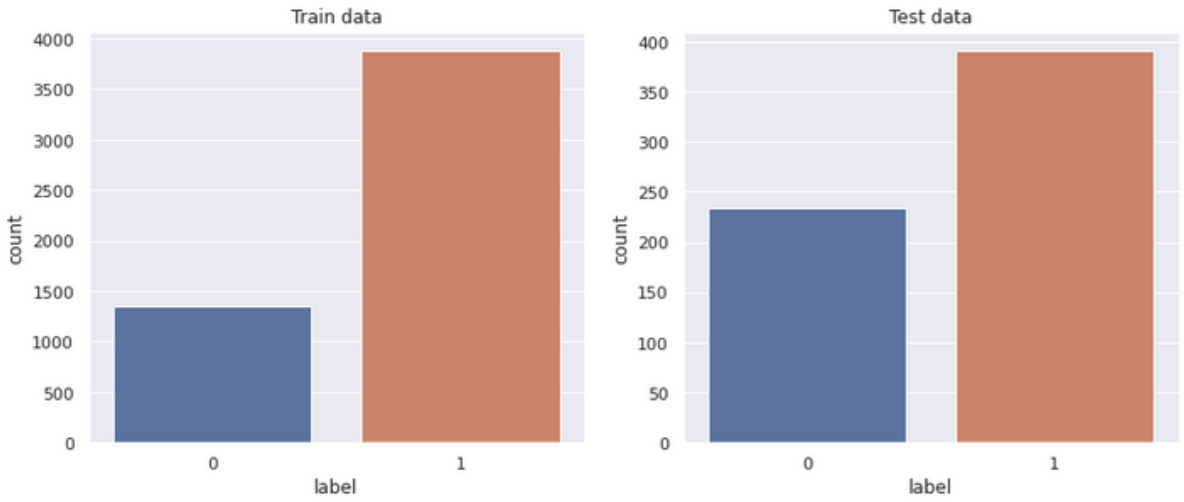

Figure 1 : *Répartition des différents cas au sein du jeu de données (0 : cas "normal", 1 : cas "pneumonie")*

Figure 2 : *Images de radiographie pulmonaire (noir et blanc)*

Les images n'étant pas toutes formatées avec les mêmes dimensions, chacune a été redimensionnée de façon à ce que toutes fassent au final 150x150 (longueur X largeur). Les images sont visualisées en noir et blanc mais possèdent 3 cannaux de couleur, Pour simplifier nous avons reformatée les images avec 1 seul canal (150,150,1). De plus, l'intensité de chaque pixels à été normalisée entre 0 et 1.

Etant donné que nous n'avons que deux classes à prédire, nos classes seront en format binaire (0 : Normal, 1 : Pneumonie), cela signifie que toute nos couches de sortie seront des couches Dense avec 1 neurone, utiliserons une sigmoïde comme fonction d'activation et qu'on utilisera la fonction loss "binary crossentropy" durant l'apprentissage.

Méthodes :

Pour réaliser notre apprentissage nous avons utilisé le Module Keras de la librairie Tensorflow (version 2.6.0) sous Python et utiliser le service google colab pour gérer l'environnement de travail et avoir accès à des temps de calculs avec GPU.

Pour débuter ce projet nous avons utilisé des structures de réseau conçu et mis à disposition sur la plateforme Kaggle (Figure 3 et 4) .

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 32)	320
max_pooling2d (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_1 (Conv2D)	(None, 72, 72, 32)	9248
max_pooling2d_1 (MaxPooling2	(None, 36, 36, 32)	0
flatten (Flatten)	(None, 41472)	0
dense (Dense)	(None, 32)	1327136
dense_1 (Dense)	(None, 1)	33

Total params: 1,336,737
Trainable params: 1,336,737
Non-trainable params: 0

Figure 3 : *Modèle_simple*

Layer (type)	Output Shape	Param #
input_23 (InputLayer)	[(None, 150, 150, 1)]	0
conv2d_110 (Conv2D)	(None, 144, 144, 16)	800
conv2d_111 (Conv2D)	(None, 138, 138, 32)	25120
batch_normalization_83 (Batc	(None, 138, 138, 32)	128
max_pooling2d_83 (MaxPooling	(None, 69, 69, 32)	0
conv2d_112 (Conv2D)	(None, 65, 65, 32)	25632
conv2d_113 (Conv2D)	(None, 61, 61, 64)	51264
batch_normalization_84 (Batc	(None, 61, 61, 64)	256
max_pooling2d_84 (MaxPooling	(None, 30, 30, 64)	0
conv2d_114 (Conv2D)	(None, 28, 28, 64)	36928
conv2d_115 (Conv2D)	(None, 26, 26, 128)	73856
batch_normalization_85 (Batc	(None, 26, 26, 128)	512
max_pooling2d_85 (MaxPooling	(None, 13, 13, 128)	0
flatten_21 (Flatten)	(None, 21632)	0
dense_42 (Dense)	(None, 32)	692256
dropout_21 (Dropout)	(None, 32)	0
dense_43 (Dense)	(None, 1)	33

Total params: 906,785
Trainable params: 906,337
Non-trainable params: 448

Figure 4: *Modèle_DK*

Model_simple : Ce modèle se compose de 2 couches de Convolution suivi chacune par un MaxPooling. Ce bloc est ensuite suivi d'une couche Flatten pour aplatir les données suivi d'une première couche Dense de 32 neurones avec une activation relu puis d'une couche de sortie. Les deux Couche de Convolution ont comme paramètres : filtre = 32, kernel size = (3,3), activation = relu.

Model_DK : ce modèle est très similaire au modèle présent dans le notebook (X-Ray Pneumonia - CNN, Tensorflow 2.0, Keras [94%]) et a été obtenu après plusieurs tests empiriques. Il se compose principalement de 3 blocs contenant chacun 2 couches de convolution, où le nombre de neurones est doublé toute les 2 couches de convolution successives (bloc 1: 16,32 ; bloc 2: 32,64; bloc 3: 64,128), la taille des kernel diminue dans chaque bloc (bloc 1: (7,7) ; bloc 2: (5,5); bloc 3: (3;3)), la fonction d'activation est une relu. Ces deux couches sont suivies d'un BatchNormalization et d'un MaxPooling.

Résultats :

Après une première série de tests réalisés sur le modèle simple, nous nous sommes rendu compte que tous nos modèles rencontraient un problème de surapprentissage et n'était pas capable de prédire la classe des images. Nous avons donc réalisé une étape de data augmentation qui consiste à appliquer des modifications aléatoires des images au sein du jeu d'apprentissage et du jeu de validation. De plus de part le léger déséquilibre quant à la répartitions des classes (Figure 1) on a décidé de fixer

les valeurs initiale des poids en utilisant la formule suivante : $w_0 = \frac{1}{n_0} \times \frac{n_{total}}{2}$; $w_1 = \frac{1}{n_1} \times \frac{n_{total}}{2}$ ce qui donne respectivement 1.932 et 0.674.

Après avoir testé plusieurs structures de réseaux de façon empirique à partir du modèle_simple et du modèle_DK (nombre de convolution, batchnormalization ou dropout…), nous avons obtenu un modèle résultant de la simplification du modèle_DK (Figure 5). De plus les tests avec early stopping avaient montré qu'un apprentissage excédant les 12/13 époques n'étaient pas nécessaire. Pour gagner du temps, nous nous sommes restreints à un apprentissage sur 10 époques sur tous les tests suivants.

Layer (type)	Output Shape	Param #
input_14 (InputLayer)	[(None, 150, 150, 1)]	0
conv2d_52 (Conv2D)	(None, 148, 148, 16)	160
batch_normalization_52 (Batc	(None, 148, 148, 16)	64
max_pooling2d_52 (MaxPooling	(None, 74, 74, 16)	0
conv2d_53 (Conv2D)	(None, 72, 72, 32)	4640
batch_normalization_53 (Batc	(None, 72, 72, 32)	128
max_pooling2d_53 (MaxPooling	(None, 36, 36, 32)	0
conv2d_54 (Conv2D)	(None, 34, 34, 64)	18496
batch_normalization_54 (Batc	(None, 34, 34, 64)	256
max_pooling2d_54 (MaxPooling	(None, 17, 17, 64)	0
conv2d_55 (Conv2D)	(None, 15, 15, 128)	73856
batch_normalization_55 (Batc	(None, 15, 15, 128)	512
max_pooling2d_55 (MaxPooling	(None, 7, 7, 128)	0
flatten_13 (Flatten)	(None, 6272)	0
dense_26 (Dense)	(None, 32)	200736
dropout_13 (Dropout)	(None, 32)	0
dense_27 (Dense)	(None, 1)	33

Total params: 298,881
Trainable params: 298,401
Non-trainable params: 480

Figure 4: *Modèle_DK*

Modèle_TF : il se compose de 4 bloc constitué de:

- Un couche de Convolution, avec des kernel size = (3,3) et une fonction d'activaton relu
- Une couche de Batchnormalization (nos test montrait que ça fonctionnait mieux qu'avec dropout ou aucun des deux)
- Une couche de MaxPooling et un pool size de (2,2)

s'ensuit une couche Flatten pour réduire à une dimension nos données, une couche Dense avec 32 neurones et une relu comme fonction d'activation, un Dropout de 0.2 et enfin la couche de Sortie.
Le nombre de filtres des Convolutions commencent à 16 dans le bloc n°1 et est doublé à chaque nouveau bloc.

Ajustement de la Data augmentation et amélioration du modèle :

A l'aide du modèle_TF nous avons testé les effets de différents paramètres présents dans la fonction ImageDataGenerator (fonction utilisée pour réaliser la data augmentation).

Premièrement nous avons réalisé un apprentissage sans data augmentation sur les jeux d'entraînement et de validation. Les résultats furent assez satisfaisants, la val loss a diminué progressivement avant de stagner durant les 4 dernières époques, la val accuracy a elle augmenté puis globalement stagné durant les 4 dernières époques. L'accuracy sur le jeu de test était elle de 78% (bien meilleur que les anciens test sur le modèle_simple).

Deuxièmement nous avons utilisé l'argument rotation_range pour appliquer une rotation aléatoire entre -10 et +10°, on peut voir l'effet de la rotation sur la figure 6.

Figure 6 : *Image de radiographie avec une rotation aléatoire*

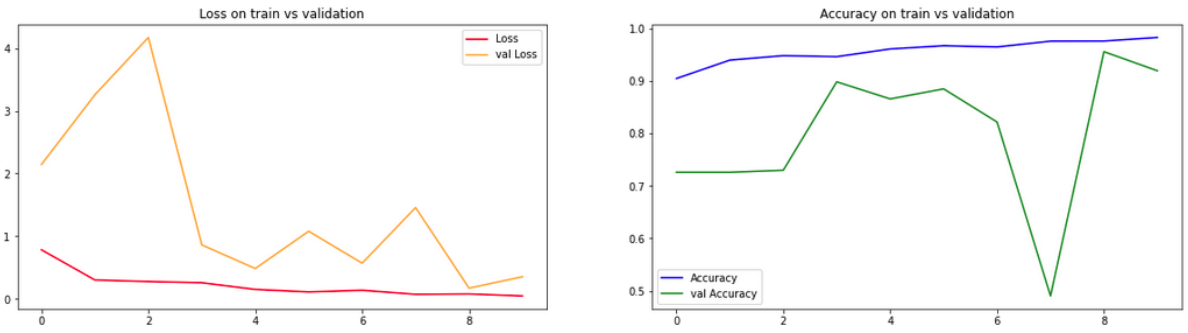

Figure 7: *Evolution de la valeur de Loss et d'Accuracy au fil des époques avec rotation aléatoire*

On constate que la val loss diminue globalement au cours des époques mais varie d'une époque à l'autre. Du côté de la val accuracy hormis une grosse chute que nous ne serions expliquée, l'accuracy à globalement augmenté. De plus, l'accuracy sur le test était de 87%, ce qui représente une nette hausse par rapport au résultat obtenu sans data augmentation.

Troisièmement nous avons cette fois ci essayé les paramètre "width_shift_range" et "height_shift_range", qui applique respectivement sur l'axe horizontal et l'axe

vertical. Les résultats n'ont pas été concluants, l'accuracy obtenu sur le jeu de test a été de 76%, soit moins bien que sans modification sur les images.

Quatrièmement, nous avons testé le paramètre "zoom_range" qui applique un zoom aléatoire de 10% dans notre cas, son effet est montré sur la Figure 8. Comme pour la rotation les résultats ont été concluants avec une test accuracy de 87%. De plus, on voit sur la Figure 9 que la val loss à bien diminué au cours de l'apprentissage avant de rester stable sur une valeur assez proche de la loss du jeu d'entraînement.

Figure 8 : *Image de radiographie avec un zoom aléatoire*

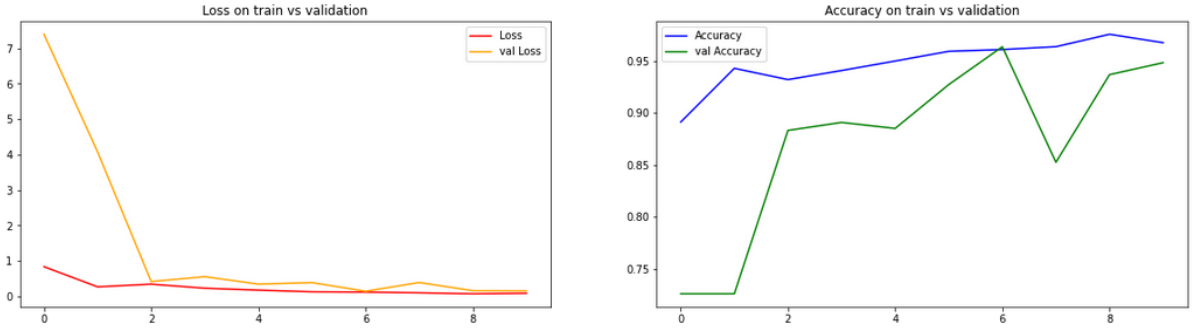

Figure 9: *Evolution de la valeur de Loss et d'Accuracy au fil des époques avec un zoom aléatoire*

Cinquièmement, nous avons fait varier la luminosité avec l'argument "brightness_range", l'exemple de son effet est montré sur la Figure 10 où l'on peut voir la même image avec différentes luminosité. Tout comme pour les décalages horizontaux et verticaux, les résultats n'ont pas été concluants. Nous avons obtenu cette fois-ci une accuracy sur le jeu de 62.5%, une valeur égale à la proportion de pneumonie dans le jeu test. Le modèle à sur appris et ne peux pas classer les images correctement et attribue la classe 1 à toutes les images.

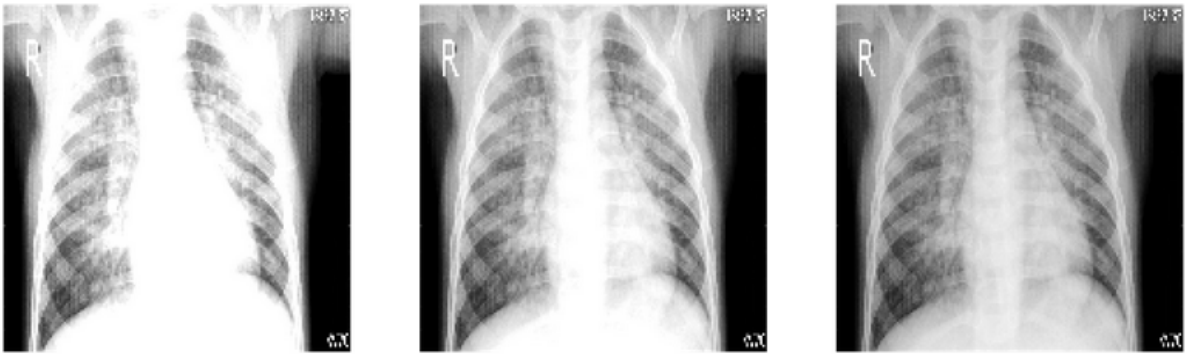

Figure 10 : *Image de radiographie avec modification aléatoire de la luminosité sur un intervalle [-50%; +50%]*

Sixièmement, On a testé l'argument "shear_range" qui applique une déformation sur une range de 10° dans notre cas. Son effet est visible sur la Figure 11. Comme pour la rotation et le zoom les résultats ont été concluants, avec une val loss qui diminue tout au fil des époques. La val accuracy a augmenté tout au fil des époques également. De plus, l'accuracy obtenu sur le jeu de test à été de 85%.

Pour finir ces test sur la data augmentation nous on avons conserver les paramètres ayant donné des résultats satisfaisant c'est-à-dire : rotation_range, zoom_range et shear_range.

L'effet de ces 3 paramètres est visible sur la Figure 13. Les résultats sont très satisfaisants, en effet comme le montre la Figure 14, la val loss à bien diminuée atteignant des valeurs aussi faible que la loss du jeu d'entraînement avant de légèrement augmenter. La val accuracy augmente elle tout au fil des époques. De plus, l'accuracy obtenu en test était de 91.5% ce qui est un très bon résultat.

Figure 11 : *Image de radiographie avec déformation aléatoire*

Pour tester la data augmentation comme précisé au préalable nous avions fixé l'initialisation des poids. On a cependant décidé de ré effectuer un test sur l'initialisation des poids avec le modèle_TF et la data augmentation bien établie.

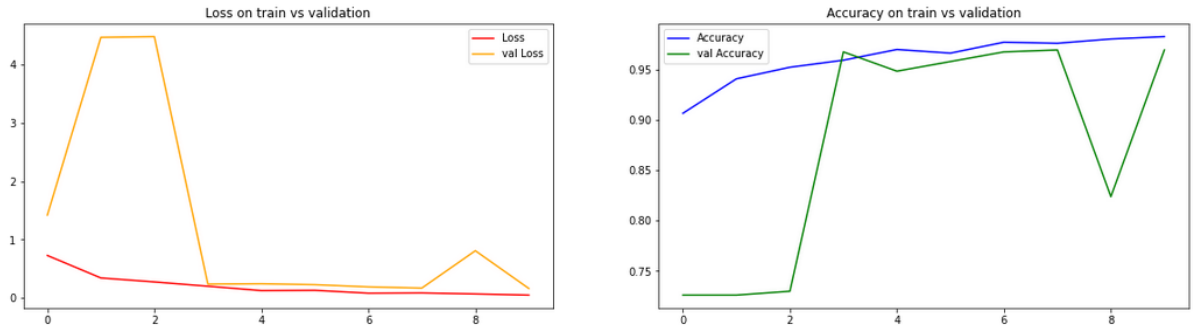

Figure 12: *Evolution de la valeur de Loss et d'Accuracy au fil des époques avec une déformation aléatoire*

Figure 13 : *Image de radiographie avec rotation , déformation et zoom aléatoire*

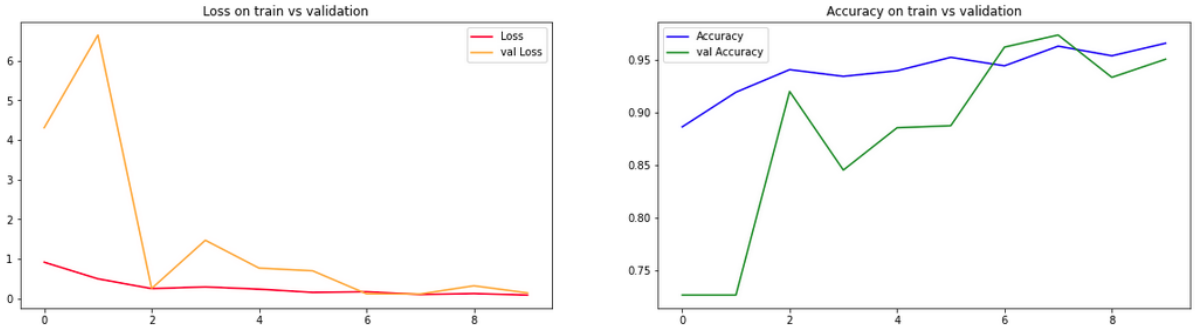

Figure 14: *Evolution de la valeur de Loss et d'Accuracy au fil des époques avec une déformation, rotation et un zoom aléatoire*

Comparaison de différentes initialisation des poids :

Pour ces tests nous avons donc utilisé comme référence les résultats obtenus précédemment avec le modèle_TF et la data augmentation finale, lors de cet apprentissage les poids avait initialisé aux valeurs 1.932 pour la classe 0 et 0.674 pour la classe 1 ce qui correspond à notre méthode 1.

méthode 2 : Cette fois-ci les poids aux ont été initialisés avec les valeurs suivantes : 0.74 pour la classe 0 et 0.26 pour la classe 1, ce qui correspond respectivement à la proportion de classe 1 et de classe 0 dans le jeu d'entraînement. Ces poids de départs offre des performances proche de la méthode précédente, nous avons obtenu 90.3% pour l'accuracy sur le jeu test avec des évolutions de loss et d'accuracy similaire à ce que l'on peut voir sur la Figure 14 (avec cependant plus de variations sur la val accuracy).

méthode 3 : Cette fois-ci nous avons laissé l'initialisation aléatoire des poids. Les résultats ont également été très bons, l'évolution des loss et des accuracy pour cette méthode est très similaire à ce qui est représenté sur la Figure 14. De plus, l'accuracy sur le jeu de test à été de 88%.

Ce que l'on peut déduire de ces tests c'est que le déséquilibre de proportion entre les classes étant faible (3 pneumonie pour 1 normal), l'initialisation des poids permet d'augmenter légèrement les performances du modèle_TF. Cependant ce n'est pas un paramètre crucial. Pour les prochain tests nous allons donc conserver la méthode pour l'initialisation des poids.

Intervalle de Confiance sur l'accuracy test :

Pour effectuer une data augmentation nous utilisons la fonction "ImageDataGenerator" de Keras qui créé un générateur d'image. Les modifications réalisées par ce générateur sont aléatoires et dépendent donc d'une seed. Cette seed peut être fixée manuellement ou bien choisie aléatoirement lors de la création de l'objet ImageDataGenerator. Depuis que nous avions créé le générateur appliquant à la fois une rotation, un zoom et une déformation, nous utilisions le même générateur et donc la même seed. Cependant si nous relancions la création du générateur la seed serait différente et donc les modification apporté aux images

le serait également, nous aurions donc un jeu de donnée légèrement différent. Nous voulons donc voir si le modèle_TF est sensible ou non à cette variation. Pour cela nous avons conservé l'accuracy test obtenu en amont (91.5%) et avons lancé 10 fois l'apprentissage en utilisant un nouveau générateur (les paramètres sont inchangés pour chaque itération) à chaque itération. Cela nous a permit d'obtenir un intervalle de confiance à 95% de $90.4 \pm 1.9\%$. Le modèle_TF avec la méthode 1 pour initialiser les poids et les bon paramètres de data augmentation est donc suffisamment robuste pour ne pas dépendre de la seed choisi lors de la création de l'objet ImageDataGenerator.

Variations du Learning rate :

Afin de voir l'influence du Learning rate (LR) sur les performances de notre réseau, nous avons fait varier celui-ci entre 10e-1 et 10e-5, sachant qu'il est par défaut réglé

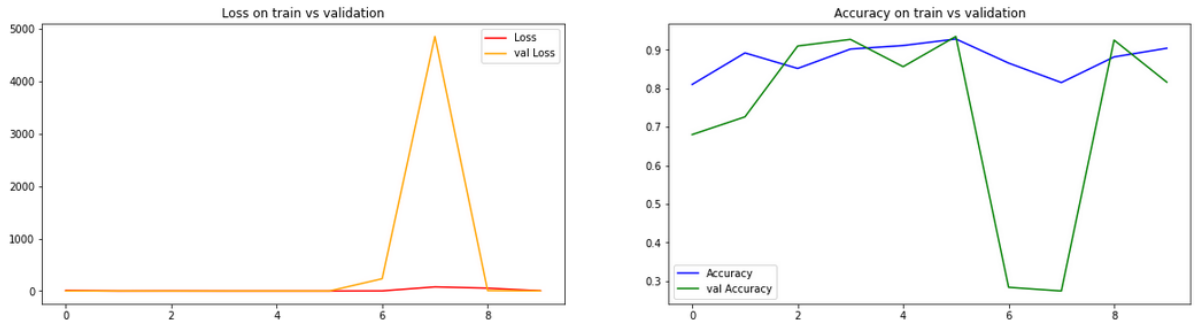

Figure 15: *Evolution de la valeur de Loss et d'Accuracy au fil des époques, pour un Learning rate de 10e-1*

à 10e-3 (voir figure 14). Nous passons donc d'abord le LR à 10e-1 pour commencer.

Nous pouvons voir ici que le pas attribué est beaucoup trop grand, ce qui conduit à une forte variation dans l'apprentissage de notre réseau. Cela influence bien sûr négativement la performance lors du test, qui descend à ~81% de bonnes prédictions. Nous diminuons ensuite ce LR à 10e-2.

Là encore, nous nous retrouvons avec un pas trop petit qui influence négativement l'apprentissage, et fait tomber le pourcentage de bonnes prédictions à 70% lors du test. Nous regardons après ce qu'il se passe quand le LR est inférieur à la valeur par défaut, en prenant une valeur de 10e-4.

Cette fois-ci également, on voit que les valeurs de Loss et Accuracy varient beaucoup mais sur des échelles plus petites, ce qui signifie que le pas est devenu trop petit et peine à se stabiliser. La performance est cependant meilleure qu'avec les valeurs précédentes de LR, avec 87%, mais reste inférieure à celle avec le LR par défaut. Nous avons pour finir testé avec un LR de 10e-5, mais ce pas était tellement faible que le réseau n'arrivait finalement plus à apprendre et attribuait lors du test la classe "pneumonie" à toutes les images sans distinction.

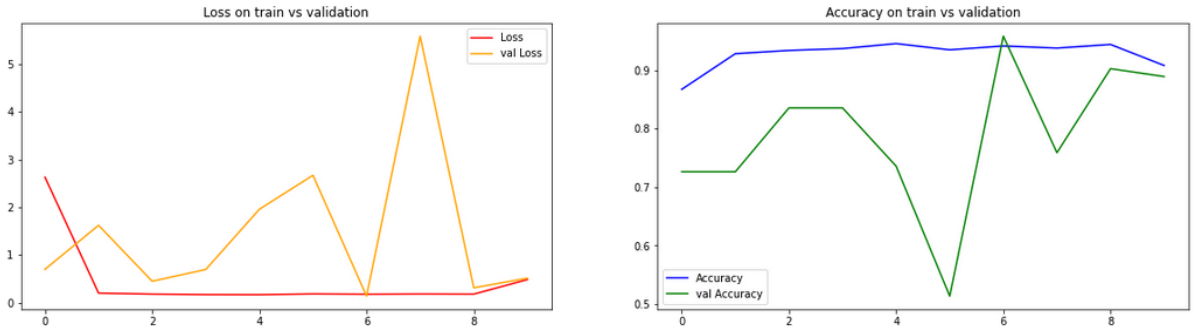

Figure 16: *Evolution de la valeur de Loss et d'Accuracy au fil des époques, pour un Learning rate de 10e-2*

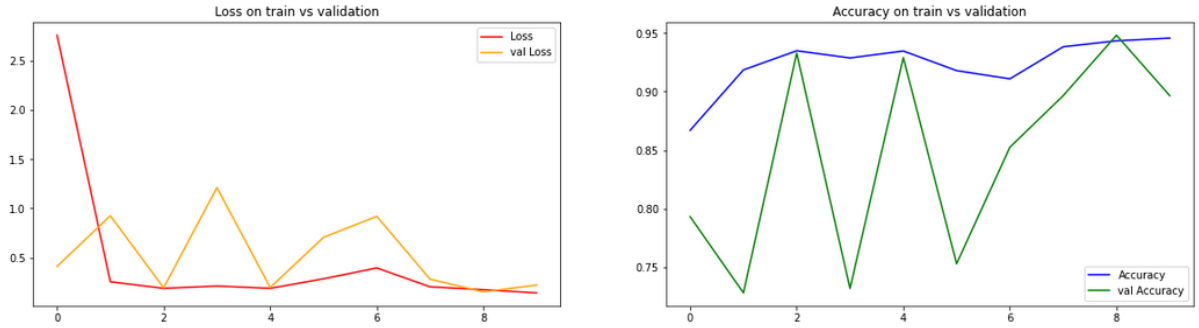

Figure 17: *Evolution de la valeur de Loss et d'Accuracy au fil des époques, pour un Learning rate de 10e-4*

Modification de la structure du modèle :

Avant de concevoir le modèle_TF, on avait repris de notebook disponible sur Kaggle le modèle_simple et conçu le modèle_DK. En réalisant des tests en modifiant ces 2 modèles on avait finalement obtenu le modèle_TF. Ce modèle est assez proche du modèle_DK. La différence entre les 2 est que dans un bloc, le modèle_TF ne contient qu'une couche de Convolution tandis que le modèle_DK contient deux couches. De plus, les kernel size du modèle_TF sont fixés à (3,3) tandis qu'au sein du modèle_DK le kernel size du premier bloc est de (7,7) puis la taille est diminuée

de 2 à chaque nouveau bloc. Nous avons donc voulu tester des modèles intermédiaires.

Deux couches de Convolution par bloc : Nous avons testé les performances d'un modèle nommé modèle_DC qui contient deux couches de convolution par bloc. Avec ce modèle les évolutions de la loss en fonction des époques était similaire au modèle_TF et DK. L'accuracy sur le jeu de test était elle de 84.6% inférieur au modèle_TF et au modèle_DK.

Variation de la taille des kernels : Nous avons ensuite essayé de faire varier la taille des kernels comme pour le modèle_DK. Cependant on a obtenu un sur apprentissage qui a empêché la prédiction des classes du jeu test.

Augmentation de la taille des kernels : Plutôt que de faire varier la taille des kernels au fil des blocs, nous les avons fixés à une taille de (7,7) au lieu de (3,3) dans le modèle_TF. La val loss a globalement diminué au fil des époques et la val accuracy à globalement augmenté malgré une chute en milieu d'apprentissage. De plus, on a obtenu une accuracy sur le test de 86.8% soit une valeur inférieur au modèle_TF et proche du modèle_DK.

Ces modèles n'ont pas pu obtenir de meilleure performance que le modèle_TF mais utiliserons leur performance pour une faire une comparaison globale.

Transfer Learning :

Nous avons, ensuite, comparé ces résultats obtenus avec ceux de réseaux réalisés à partir de Transfer Learning. Pour cela, différents modèles ont été sélectionnés : VGG16, ResNet50V2, InceptionV3 et DenseNet201. Tous ont reçu les poids pré-entraînés de "imagenet", sans être ré-entraînés avec notre jeu d'entraînement (seul les dernières couches avant la sortie sont entraînées par notre jeu). Les poids de "imagenet" ont été obtenue par apprentissage sur des images en couleurs. Nous avons donc dû remettre artificiellement nos images au format de couleur "RGB", ce qui ne les remet pas véritablement en couleur, afin de pouvoir réaliser nos différents Transfer Learning. Globalement, chacun de ces réseaux obtenus par Transfer Learning ont eu entre 85% et 88% de bonnes prédictions sur notre jeu de test. Ces réseaux nous fournissent, donc, des résultats de test équivalents à notre réseau de

simples convolutions. Cela démontre que pour un problème de classification simple, comme celui-ci, un simple réseau de neurones composés de quelques couches de convolution est suffisant pour avoir des résultats fiables.

Comparaison des performances des différents modèles obtenus :

Toutes ces expérimentations ont permis de construire plusieurs modèles de réseaux, pouvant chacun atteindre au moins 85% de bonnes prédictions lors du test de façon régulière.

	Accuracy test	Sensiti.(%)	Specifi.(%)	Number of Parameters
Model_TF	0.924	93.076923	91.452991	298 881
Model_DC	0.846	99.230769	60.256410	396 657
Model_K1	0.868	98.717949	67.094017	565 761
Model_DK	0.873	98.717949	68.376068	906 785
TL_VGG	0.877	87.435897	88.034188	14 722 881
TL_ResNet50V2	0.888	93.333333	81.196581	23 616 001
TL_InceptionV3	0.859	95.384615	70.085470	21 821 217
TL_DenseNet201	0.881	93.846154	78.632479	18 352 705

Figure 18 : *Tableau récapitulatif des performances sur le jeu de test pour chaque modèle de réseau sélectionné (les noms de modèle commençant par "TL" signifie que le modèle utilise le Transfer Learning)*

Nous pouvons voir que, globalement, tous les modèles possèdent une valeur d'Accuracy proche. Néanmoins, nous pouvons dire que les modèles utilisant le Transfer Learning renvoient à des prédictions de meilleures qualités. En effet, bien que les modèles sans Transfer Learning possèdent une sensibilité élevée, et prédisent donc bien les cas "pneumonie", leur capacité à prédire correctement les cas "normal" reste assez faible (spécificité < 70%). Ce qui n'est pas le cas avec les Transfer Learning, qui possèdent de très bonnes sensibilités sans pour autant avoir une trop mauvaise spécificité.

Pour autant, le meilleur modèle parmi cette sélection reste le modèle avec le moins de paramètre, dépassant les 90% en sensibilité comme en spécificité. Il permet ainsi d'avoir beaucoup de précision dans la classification de nos cas "normal" et "pneumonie".

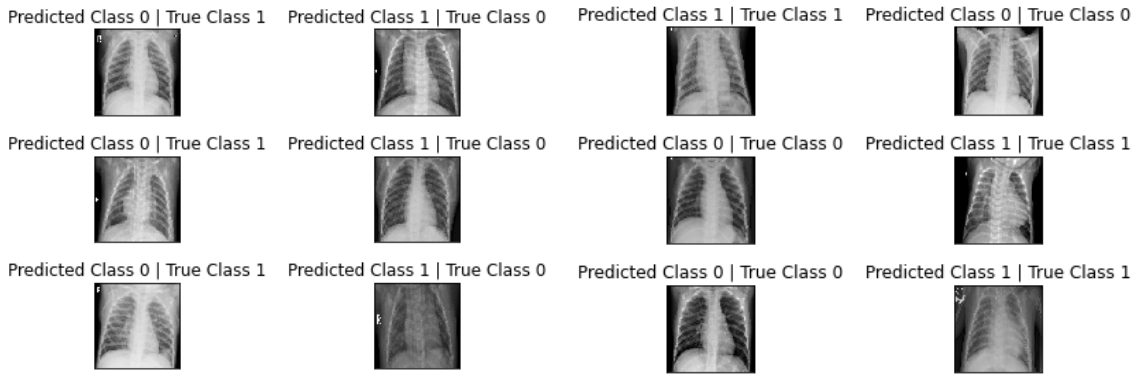

Figure 19 : *Échantillon des classes prédites en fonction des images fournit à notre modèle*

La variabilité des prédictions de ce modèle semble essentiellement résulter du fait que certaines images peuvent être de mauvaise qualité, ce qui empêche notre réseau de reconnaître correctement la classe des images qu'on lui donne en entrée.

Discussion :

Bien que notre modèle ait pu dépasser la barre symbolique des 90% de bonnes prédictions sur le jeu de test, les meilleurs réseaux de neurones observés pour la classification de ce genre d'image peuvent eux atteindre les 95%. En d'autres termes, il serait normalement possible d'améliorer encore un peu plus notre modèle de réseau, bien que ses performances dépassent déjà celles de la majorité de ces autres réseaux. Il serait par exemple possible de tester encore d'autres formes de modèles, ou d'utiliser d'autres paramètres pour la Data augmentation afin de tenter d'arriver au 95%. Avec plus de temps, il serait également possible d'améliorer ce réseau pour qu'il puisse différencier les pneumonies bactériennes des pneumonies virales, simplement en exécutant le data processing nécessaire pour avoir 3 classes (normal, bactérie, virus) avant d'entraîner le réseau pour ce cas.

Conclusion :

Nous avons pu répondre au problème posé pour ce sujet en construisant un réseau de neurones entraîné capable de classifier correctement les images de radiographie pulmonaire de patients, selon s'ils sont malades ou non. Même s'il reste possible d'atteindre de meilleures performances, celui-ci reste cependant l'un des meilleurs qu'il est possible d'avoir pour ce genre de cas. D'autant plus, ce réseau se démarque des autres de part à la fois sa simplicité et sa concision dans la forme du modèle utilisé, qui garantit de bonnes prédictions au moins 9 fois sur 10.