# Rapport Kaggle : Chest X-Ray Images (Pneumonia)

Dai yuping, Ren yani, Eid Rita

## 1. Introduction

La pneumonie a causé de nombreux décès dans le monde entier, et il est difficile de détecter de nombreuses maladies pulmonaires telles que l'atélectasie, la cardiomégalie, le cancer du poumon, etc. L'intelligence artificielle (IA) a le potentiel de révolutionner le diagnostic et la gestion des maladies en effectuant une classification difficile pour les experts humains et en examinant rapidement de nombreuses d'images.
Dans notre projet, on va commencer à créer un modèle de classification d'images radiographiques pour prédire si une radiographie montre la présence d'une pneumonie ou non. Notre projet présente principalement les étapes suivantes : chargement des données d'image, création et entraînement d'un réseau de neurones de convolution, affinement et régularisation du modèle et la prédiction des résultats.

## 2. Matériels et Méthodes

### 2.1 Le jeu de données

Le jeu de données de classification de pneumonie d'origine dans Kaggle est divisé en deux sous-ensembles: train et test. Au début, on cherche à retrouver les classes sur lesquelles on va faire l'apprentissage. Soit dans le groupe train on a remarqué les 2 classes voulues 'Normal' ou 'Pneumonia' . Nous avons 1349 images qui appartiennent à la classe 'Normal' et 3883 images qui appartiennent à la classe 'Pneumonia'.
On remarque que le pourcentage des échantillons 'Pneumonia' est plus important que celui des échantillons 'Normal'. Il est donc important de bien diviser nos groupes quand on veut faire notre groupe de validation. En effet, nous avons échantillonné au hasard 20% des données du sous jeu validation, en nous assurant d'obtenir autant d'images de classes "Normal" que de "Pneumonia" pour constituer notre jeu de validation. Cela est utile pour avoir une validation pertinente et non biaisée par la taille des données en entrée.
Finalement, nous obtenons pour le groupe train 4185 images appartenant à deux classes (Normal : 826, Pneumonia : 3360), le groupe test contient 624 fichiers images appartenant à deux classes (Normal : 234, Pneumonia : 390 ) et le groupe validation contient 1047 fichiers images appartenant à deux classes (Normal : 523, Pneumonia : 523 ).
On remarque en plus que dans la classe 'Pneumonia ', nous possédons 2 sous classes soit si la pneumonie est bactérienne ou virale. Grâce à cette information nous pouvons encore plus approfondir notre travail en divisant en plus le jeu de données en 3 classes. On a alors la classe 'Normal'(1349), la classe 'Pneumonie Bactérienne' (2538) et ' Pneumonie Virale'(1345). Vu que nous avons le même problème avec la classe 'Pneumonie Bactérienne' qui a plus d'échantillons que les 2 autres, on va utiliser le système pour avoir la validation. soit on échantillonne 20% du train en mettant une valeur d'individu égale pour chaque classe.
Finalement, nous obtenons pour le groupe train 4182 images appartenant à trois classes (Normal : 999, Pneumonie Bactérienne : 2118, Pneumonie Virale :995 ), le groupe test contient 624 fichiers images appartenant à trois classes (Normal : 234, Pneumonie

Bactérienne: 242, Pneumonie Virale :148 ) et le groupe validation contient 1050 fichiers images appartenant à trois classes (Normal : 350, Pneumonie Bactérienne : 350, Pneumonie Virale :350 ).

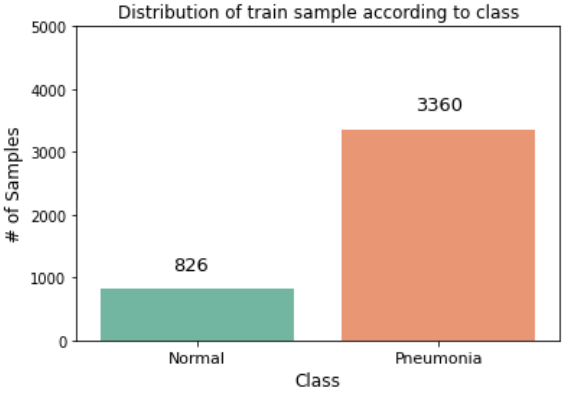



Fig1 : Distribution de l'échantillon de train selon la classe

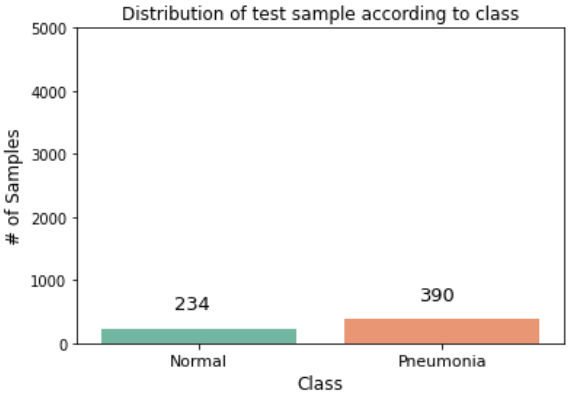



Fig2 : Distribution de l'échantillon de test selon la classe

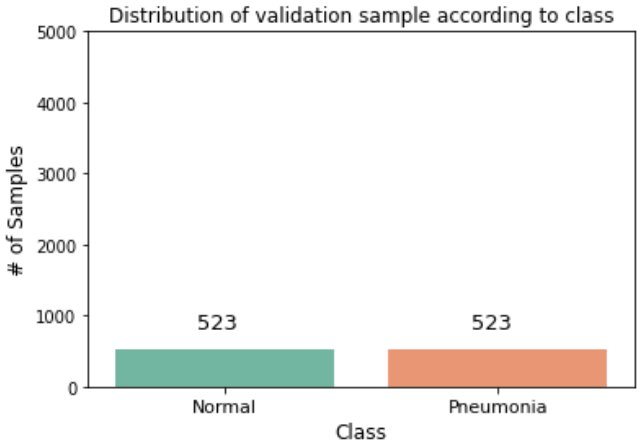



Fig3 : Distribution de l'échantillon de validation selon la classe

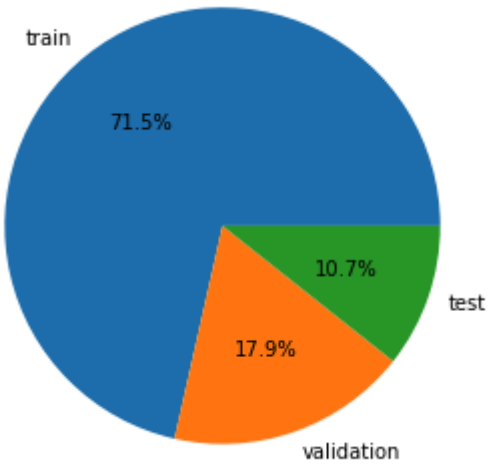



Fig4 : Distribution de l'échantillon selon les groupes

Le diagramme représente le pourcentage de trois sous-ensembles:  train, validation et test.

## 2.2 Traitement des données d'image

Avant de créer notre modèle, on doit au début traiter les images. On utilise deux data frames individuelles pour le jeu de données train et le jeu de données test. On a donc un dataframe train avec le path de l'image et la classe à laquelle elle appartient. La classe '0' correspond à 'Normal' et la classe '1' correspond à 'Pneumonia'. Et pour la division à 3 classe: la classe '0' correspond à 'Normal' et la classe '1' correspond à 'Pneumonie bactérienne' et le classe '2' correspond à 'Pneumonie virale'. Nous utilisons le data frame train pour échantillonner notre jeu d'apprentissage et jeu de validation.
Ensuite nous avons lue les images avec la fonction cv2.imread(), et nous avons redimensionner les images à 224*224 pixels pour obtenir une taille homogène des images. Pour cela nous utilisons la fonction resize(). Nous convertissons les images couleurs en noirs et blancs aussi grâce à l'aide de la bibliothèque cv2.
Nous redimensionnons également la valeur des pixels de chaque image dans un intervalle [0,1] en divisant par 255.
Puis on divise le groupe train pour avoir la validation selon les critères que nous avons préalablement définis. En plus nous preinicialisons les poids afin de plus mettre en valeur la classe 'Normal'.
Ensuite nous sauvegardons les images en numpy array qui va être notre x_train et et en utilisant la fonction to categorical, on récupère le y_train. on a fait de même pour le test et la validation.
Pour ajouter des modifications au niveau de notre jeu de données, nous réalisons une étape de "data augmentation". Cette technique consiste à appliquer plusieurs transformations à partir d'une image existante afin de générer plusieurs exemplaires différents. Ces transformations peuvent être des rotations, des agrandissements ou autres qui permettent d'obtenir un modèle plus robuste et de meilleures performances sur de nouvelles données. Pour cela, nous utilisons la méthode "ImageDataGenerator" implémentée dans la bibliothèque Keras. Cette méthode permet l'augmentation du nombre d'images en temps réel pendant l'apprentissage du modèle.
Pour pouvoir donner nos images en entrée dans les réseaux déjà entraîné tel que VGG16, nous avons redimensionner les images avec le fonction flow_from_dataframe , qui permet de charger des données via un dataframe Pandas. .flow méthode pour construire nos générateurs de train, de validation et de test. le paramètre class_mode(obligatoire, défini par défaut sur categorical) - Quelques options parmi lesquelles choisir (binaire, brut, entrée, etc.) indiquent à votre générateur le type d'étiquette cible à attendre. Donc Le fonction permet de lire et modifier des données plus simple.

## 2.3 Création du réseau

Une fois que le jeu de données est prêt, il faut entraîner le réseau de neurones. Comme nos données sont des images, nous décidons de créer un réseau de neurones convolutifs puis un réseau avec du transfert learning. Chaque neurone d'une couche convolutive vont apprendre les caractéristiques d'une image. L'entraînement consiste à faire varier les millions de paramètres du réseau jusqu'à ce que sa sortie soit la plus proche possible du résultat attendu.

1. Réseaux CNN

En entrée, un CNN prend des tenseurs de forme (image_height, image_width, color_channels), en ignorant la taille du lot. Ici, la valeur du channel est de 1, signifiant que les images sont en noirs et blancs. Si ce n'est pas le cas et que les images sont en couleur, le parametre color_channels prend la valeur de 3 . Dans notre example, nous avons configuré notre CNN pour traiter les entrées de forme (IMG_SIZE = 224, IMG_SIZE = 224, 1), qui est le format des images dans notre x_train. Nous avons dû le faire en passant l'argument input_shape à notre première couche.
Après la couche input nous avons ajouté des couches Conv2D, MaxPooling2D, Dropout, BatchNormalization selon les modeles. Nous pouvons voir que la sortie de chaque couche Conv2D et MaxPooling2D est un tenseur de forme 3D (hauteur, largeur, canaux).

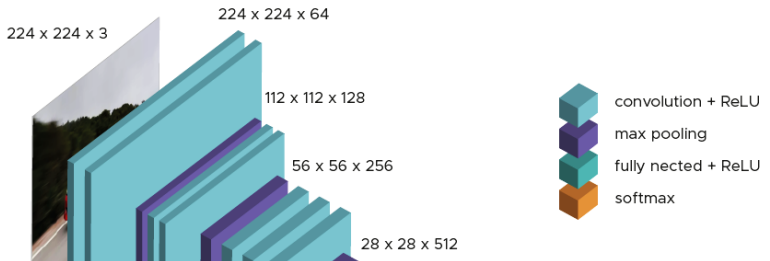
Les dimensions de largeur et de hauteur ont tendance à diminuer au fur et à mesure que vous avancez dans le réseau. Le nombre de canaux de sortie pour chaque couche Conv2D est contrôlé par le premier argument soit le filters . En règle générale, à mesure que la largeur et la hauteur diminuent, vous pouvez vous permettre (calculé) d'ajouter plus de canaux de sortie dans chaque couche Conv2D.

Nous avons essayé de créer un modèle CNN pour essayer de prédire deux classes (Normal et Pneumonie) et également trois classes (Normal, Pneumonie Bactérienne, Pneumonie Virale).
Au niveau de la compilation du modèle on utilise pour les 2 classes les paramètres suivants: une loss 'binary_crossentropy', un optimiseur 'Adam' (sur lequel on fait varier la learning rate )et une matrice 'binary_accuracy'.
Au niveau de la compilation du modèle on utilise pour les 3 classes les paramètres suivants: une loss 'categorical_crossentropy', un optimiseur 'Adam' (sur lequel on fait varier la learning rate )et une matrice 'catagorical_accuracy'.
Pour obtenir notre modèle CNN final, nous avons testé différents paramètres individuels sur nos modèles. Le premier hyperparamètre testé est le taux d'apprentissage (learning rate), nous avons ensuite fait varier le nombre de batch.Dans un second temps, nous avons essayé de faire de la data augmentation sur les données. Nous avons aussi utilisé les fonctions 'early stopping' et 'plateau' si nécessaire au niveau de notre modèle. Enfin, nous avons augmenté le nombre de couches.

Pour chacun des modèles nous retenons, le résultat du loss et de l'accuracy, le nombre d'epoch et le nombre de paramètres utilisés. Grâce à ces résultats on déterminera la pertinence du modèle.

2. Transfer Learning
    a. VGG

Nous avons également essayé d'appliquer un modèle pré-entraîné sur nos données afin de comparer la structure du modèle et les performances. Parmi les modèles existant, nous avons testé VGG16. C'est un réseau composé de 13 couches convolutionnel, 4 couches MaxPooling, 2 couches dense et la couche de sortie est contrôlée par la fonction softmax.



Source : DataScientest, url : https://datascientest.com/quest-ce-que-le-modele-vgg

Ce réseau a été entraîné sur le jeu de données ‘ImageNet’ qui est composé d’environ 14 millions d’images en couleurs.

Nous avons également testé le réseau pré-entraîné ResNet, qui a également été entraîné sur le jeu de données ‘ImageNet’.

b. ResNets

Contrairement aux réseaux de neurones convolutifs qui ont une architecture linéaire (un empilement de couches dont chaque sortie est uniquement connectée à la couche suivante), Afin de résoudre le problème du gradient de disparition / explosion, cette architecture a introduit le concept appelé Réseau résiduel. Dans ce réseau, nous utilisons une technique appelée sauter les connexions". La connexion saute la formation de quelques couches et se connecte directement à la sortie. ResNet propose deux mapping : l'un est le mapping d'identité x, et l'autre est le mapping résiduel F(x), donc la sortie finale est y=F(x)+x

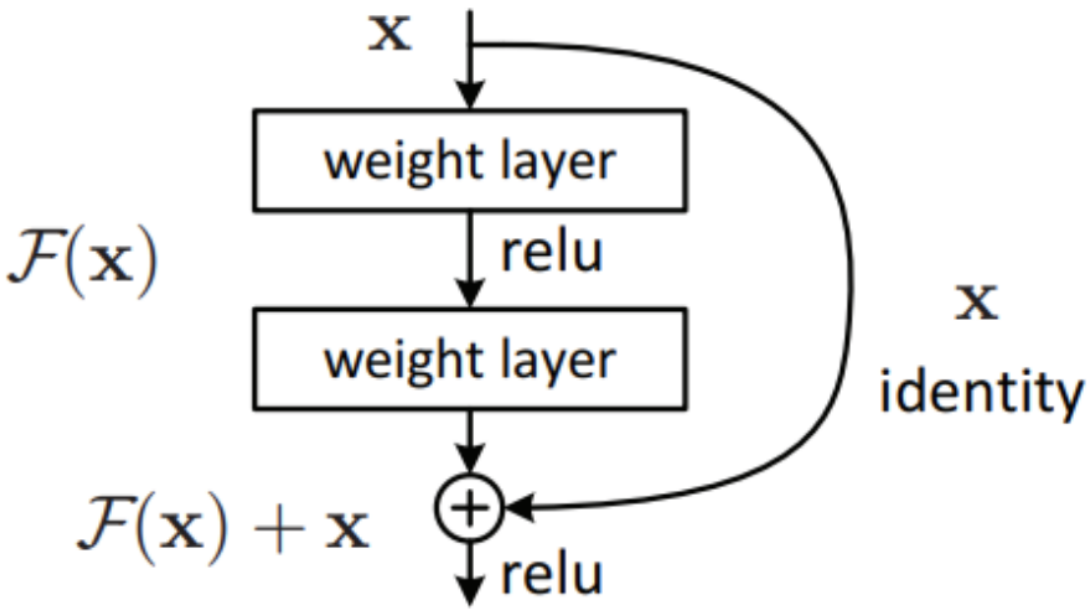



fig 4 schéma de structure Resnet
En utilisant l’API Tensorflow et Keras, nous pouvons concevoir l’architecture ResNet (y compris les blocs résiduels) à partir de zéro. Pour les réseaux plus profonds,

nous utilisons la structure de goulot d'étranglement, en utilisant d'abord une convolution 1x1 pour la réduction de dimensionnalité, puis une convolution 3x3, et enfin en utilisant la dimensionnalité 1x1 pour restaurer la dimension d'origine.Cela équivaut à réduire le nombre de paramètres pour le même nombre de couches, afin de pouvoir l'étendre à des modèles plus profonds.Parmi les modèles existant, nous avons testé le modèle ResNets50, ResNets101 et ResNets152 pour 50, 101 et 152 couches, et non seulement il n'a pas eu de problèmes de dégradation, mais aussi le taux d'erreur a été considérablement réduit et la complexité de calcul est devenue très faible.

# 3. Résultats

Pour évaluer les performances du modèle, nous nous sommes basés sur la différence entre la classe prédite par le modèle et la classe attendue sur le jeu de données test (loss) ainsi que sur la capacité du modèle à bien prédire (accuracy). Les performances sur le jeu de validation nous aident à régler les hyperparamètres pour optimiser les modèles à apprendre davantage.

## 3.1 Modèle de prédictions CNN à 2 classes

Pour la prédiction de deux classes (Normal et Pneumonie), sur le modèle le plus simple composé d'une seule couche convolutive (8 filtres), d'une couche maxPooling, d'une couche Flatten et d'une couche de sortie.Nous obtenons de meilleures performances à mesure qu'on diminue la valeur du taux d'apprentissage. En effet, avec un taux d'apprentissage de 0.1, la valeur loss du test est très élevée, le modèle n'apprend pas bien. Cependant un taux d'apprentissage trop faible (0.000001) fait diminuer les performances. Un taux d'apprentissage de l'ordre 10^-4 donne de meilleurs résultats. De plus, L'augmentation de la batch-size améliore les performances.

Le test des différents hyperparamètres nous a permis d'obtenir un modèle simple qui minimise l'erreur d'apprentissage. En appliquant le modèle sur les données issues du traitement avec la data augmentation, nous obtenons de moins bonnes performances.

L'ajout de couches Drop out qui réinitialise aléatoirement les poids des neurones à 0 et l'ajout d'une couche de Batch normalisation avant la couche de classification améliore les performances, on obtient une perte d'environ 0.38 et une précision proche de 0.84. L'ajout de deux couches Dense améliore encore plus le modèle, on obtient une perte d'environ 0.32 et une précision de 0.9. Cependant ajouter encore plus de couches n'améliore plus le modèle.

Ensuite, l'ajout d'une 3ème couche convolutive n'améliore pas réellement le modèle, mais l'association de deux couches convolutives avec une couche dense donne l'une des meilleures performances.

Le nombre d'épochs est toujours de 10.

| Modèle | Data Augmentation | Nb param. | Learning Rate | Batch-size | Test Loss | Test Acc | Val Loss | Val Acc |
|---|---|---|---|---|---|---|---|---|
| 1conv | non | 96,969 | 0.1 | 32 | 9.9960 | 0.74 | 3.0595 | 0.85 |
| 1conv | non | 96,0969 | 0.00001 | 32 | 0.5157 | 0.81 | 0.1581 | 0.94 |
| 1conv | non | 96,0969 | 0.000001 | 32 | 0.4913 | 0.74 | 0.5682 | 0.70 |
| 1conv | non | 96,069 | 0.00001 | 16 | 0.6361 | 0.79 | 0.1887 | 0.93 |
| 1conv2D, 2dense | non | 1,549161 | 0.000028 | 32 | 0.3197 | 0.90 | 0.1994 | 0.97 |
| 2 conv2D, 1dense | non | 1,087,393 | 0.000028 | 32 | 0.2900 | 0.90 | 0.1502 | 0.96 |
| 2conv2D, 1dense | oui | 1,087,393 | 0.000028 | 32 | 81.041 | 0.37 | 58.683 | 0.52 |
| 3conv2D, 1dense | non | 727,841 | 0.000028 | 32 | 0.3047 | 0.87 | 0.1391 | 0.91 |

Parmi les tests des différents modèles, nous observons que lorsque les hyperparamètres sont optimisés, les modèles présentant des performances les plus élevées sont constitués de peu de couches.

### 3.2 Modèle de prédiction CNN à 3 classes:

Pour la prédiction de trois classes (Normal, Pneumonie Bactérienne, Pneumonie Virale):

Nous avons au début tester sur un modèle simple avec une couche de convolution 2D (filters =8) et une couche maxPooling. On a utilisé la couche Flatten et la couche Dense de sortie. C'est le modèle 1 dans le tableau.

Nous arrivons à dégrader les performances à mesurer quand on diminue la valeur du taux d'apprentissage. En effet, avec un taux d'apprentissage de 0.001, la valeur loss du test augmente le modèle n'apprend pas bien malgré le fait que la courbe de la loss est très satisfaisante(voir annexe). C'est le modèle 3. Cependant grâce à la data augmentation qui est le modèle 2 on arrive a diminuer la valeur du loss mais l'accuracy reste peu acceptable comparé au autre.

En appliquant le modèle sur les données issues du traitement avec la data augmentation, nous obtenons de bonnes performances (voir annexe pour les courbe). Mais en utilisant une couche Batch Normalisation nous obtenons de moins bonnes performances. C'est le modèle 4. En effet la courbe de la loss au niveau de la validation est très éloignée de celle du train, on a un sur-apprentissage qui n'est pas applicable comme modèle(voir annexe).

Ensuite, l'ajout d'une couche de convolutif et d'une couche MaxPool n'améliore pas réellement le modèle (modèle 5), même si on cherche les paramètres des filtres de 16 à 8 dans la couche de convolution les résultats ne s'améliorent pas (modèle 7).
Mais avec la data augmentation on obtient des résultats très satisfaisant avec 10 epochs et peu de paramètres et avec une courbe de loss acceptable. C'est le modèle 8.
Mais l'association d'une couche Batch Normalisation et avec la data augmentation , soit le modèle 6 on obtient les meilleurs résultats. Mais il est bien de noter que ces résultats sont obtenus après 20 epochs et que la courbe de la loss varie beaucoup trop(voir annexe).

| modèle | paramètres | epochs | data augmentation | learning rate | loss test | accuracy test | confusion matrix |
|---|---|---|---|---|---|---|---|
| modèle 1 | 295 787 | 10 | non | 0.001 | 0.553 | 0.824 | ([[170, 19, 45],<br>[ 2, 230, 10],<br>[ 2, 32, 114]]) |
| modèle 2 | 295 787 | 10 | oui | 0.001 | 0.517 | 0.812 | ([[178, 16, 40],<br>[ 10, 218, 14],<br>[ 4, 33, 111]]) |
| modèle 3 | 295 787 | 10 | oui | 0.00001 | 0.655 | 0.777 | ([[212, 8, 14],<br>[ 16, 200, 26],<br>[ 52, 23, 73]]) |
| modèle 4 | 690 059 | 10 | non | 0.001 | 4.434 | 0.647 | ([[109, 30, 95],<br>[ 3, 199, 40],<br>[ 0, 52, 96]]) |
| modèle 5 | 141 219 | 10 | non | 0.001 | 0.865 | 0.731 | ([[124, 30, 80],<br>[ 1, 222, 19],<br>[ 0, 38, 110]]) |
| modèle 6 | 327 843 | 20 | oui | 0.001 | 0.409 | 0.871 | ([[219, 3, 12],<br>[ 23, 202, 17],<br>[ 7, 18, 123]]) |
| modèle 7 | 70 651 | 10 | non | 0.001 | 0.945 | 0.713 | ([[122, 56, 56],<br>[ 1, 217, 24],<br>[ 1, 41, 106]]) |
| modèle 8 | 141 219 | 10 | oui | 0.001 | 0.484 | 0.815 | ([[198, 11, 25],<br>[ 20, 184, 38],<br>[ 8, 13, 127]]) |

3.3 Modèle de prédiction à l'aide du transfert learning : modèle VGG16 N à 2 classes

| modèle | Nb param. | epochs | Test Loss | Test Acc | Val Loss | Val Acc |
|---|---|---|---|---|---|---|
| VGG16 | 14,780,481 | 15 | 0.24 | 0.8 | 0.1955 | 0.9241 |

3.4 Modèle de prédiction à l'aide du transfert learning : modèle ResNet N à 2 classes

| modèle | Nb param. | epochs | Test Loss | Test Acc | Val Loss | Val Acc |
|---|---|---|---|---|---|---|
| ResNet50 | 39,305,089 | 15 | 0.607 | 0.671 | 0.311 | 0.824 |
| ResNet101 | 42,920,577 | 15 | 0.74 | 0.375 | 0.781 | 0.257 |
| ResNet152 | 58,594,049 | 17 | 0.217 | 0.923 | 0.152 | 0.928 |

Nous avons vu comment ResNet a résolu le problème d'optimisation / dégradation avec un réseau plus profond en ignorant certaines couches en utilisant des connexions de saut ou des blocs résiduels. ResNet a réussi à augmenter la profondeur et à ne pas nuire aux performances du modèle, les performances sont devenues meilleures.

## 4. Discussion :

Nous avons remarqué que selon la construction du jeu de données, l'utilisation de la data augmentation n'améliore pas toujours le modèle à mieux apprendre, et peuvent à l'inverse faire diminuer les performances des modèles. C'est le cas quand on a 2 classes. Mais dans le cas où on a 3 classes, la data augmentation est très bénéfique pour l'apprentissage et améliore les résultats.

Nous avons également remarqué qu'un modèle simple peut permettre de donner des performances similaires avec moins de paramètres par rapport à un modèle plus complexe. Cela est généralement vrai pour les 2 classes.

En plus concernant le learning rate, nous observons qu'un taux d'apprentissage trop élevé ou encore trop faible rendent le modèle moins bon. En effet, lorsqu'elle est grande, le modèle n'apprend pas bien, la valeur de la loss est très élevée. Lorsqu'elle est trop petite, le modèle prend plus de temps à apprendre et peut être coincé dans un minimum local.

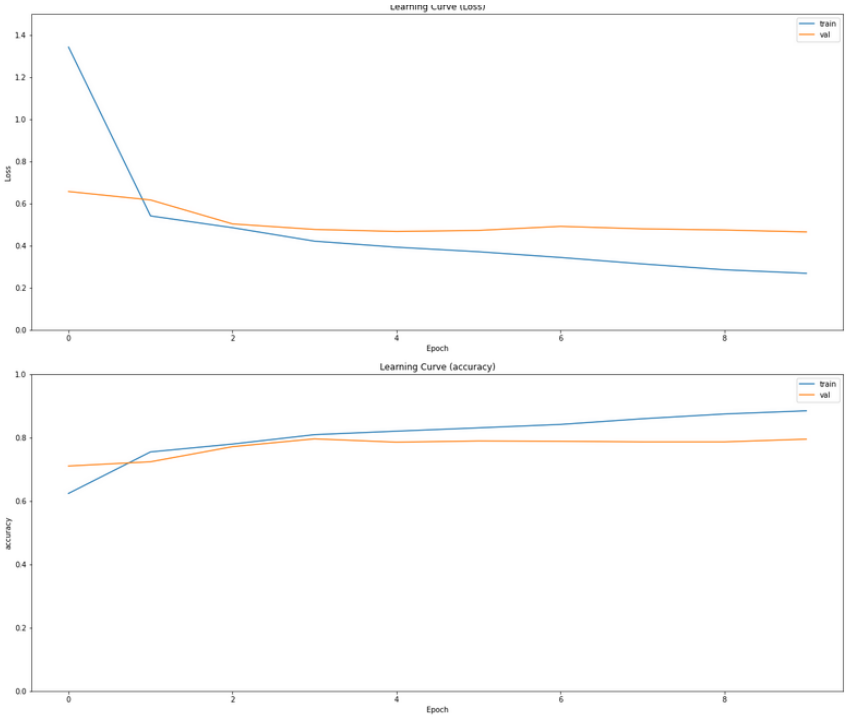
# 5. Conclusion

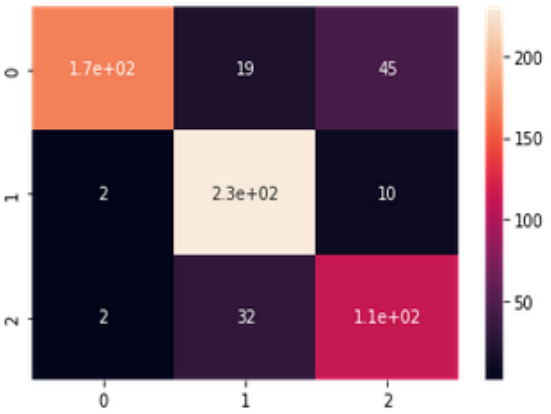
Nous avons traité le jeu de données en utilisant plusieurs modèles et en modifiant plusieurs paramètres. Nous avons prouvé l'utilité de ces paramètres que ce soit avec ou sans la data augmentation, le learning rate, le nombre d'époch et le nombre de Batch. Dans la perspective d'améliorer notre protocole d'analyse, nous pouvons évaluer notre modèle sur différents jeux de données test. En effet, dans le cas présent nous évaluons nos modèles sur un unique jeu test.  En plus, il est intéressant de continuer sur le transfert learning et voir les résultats obtenus avec 2 classes.

# 6. Annexe

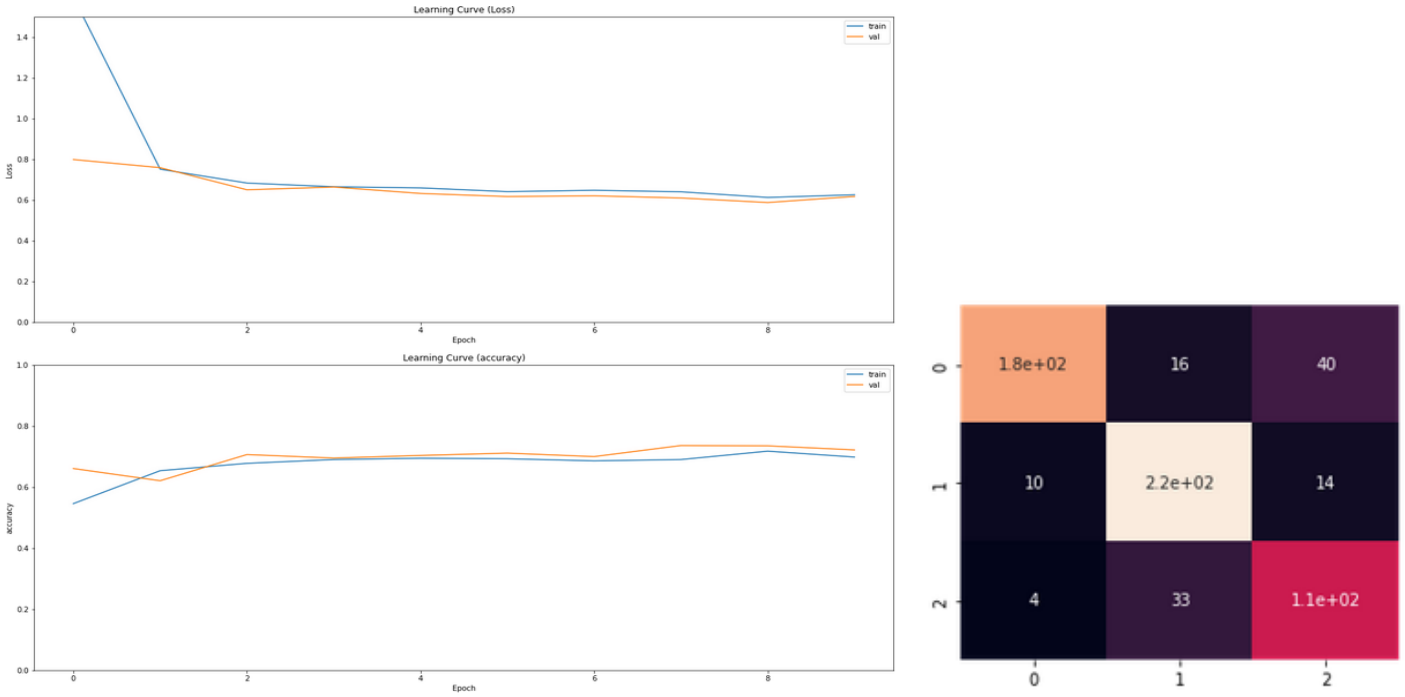
On a la courbe blue qui est le train et la orange qui représente la validation

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 1)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 222, 222, 8)       80
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 8)       0
_________________________________________________________________
flatten (Flatten)            (None, 98568)             0
_________________________________________________________________
dense (Dense)                (None, 3)                 295707
=================================================================
Total params: 295,787
Trainable params: 295,787
Non-trainable params: 0
_________________________________________________________________
```

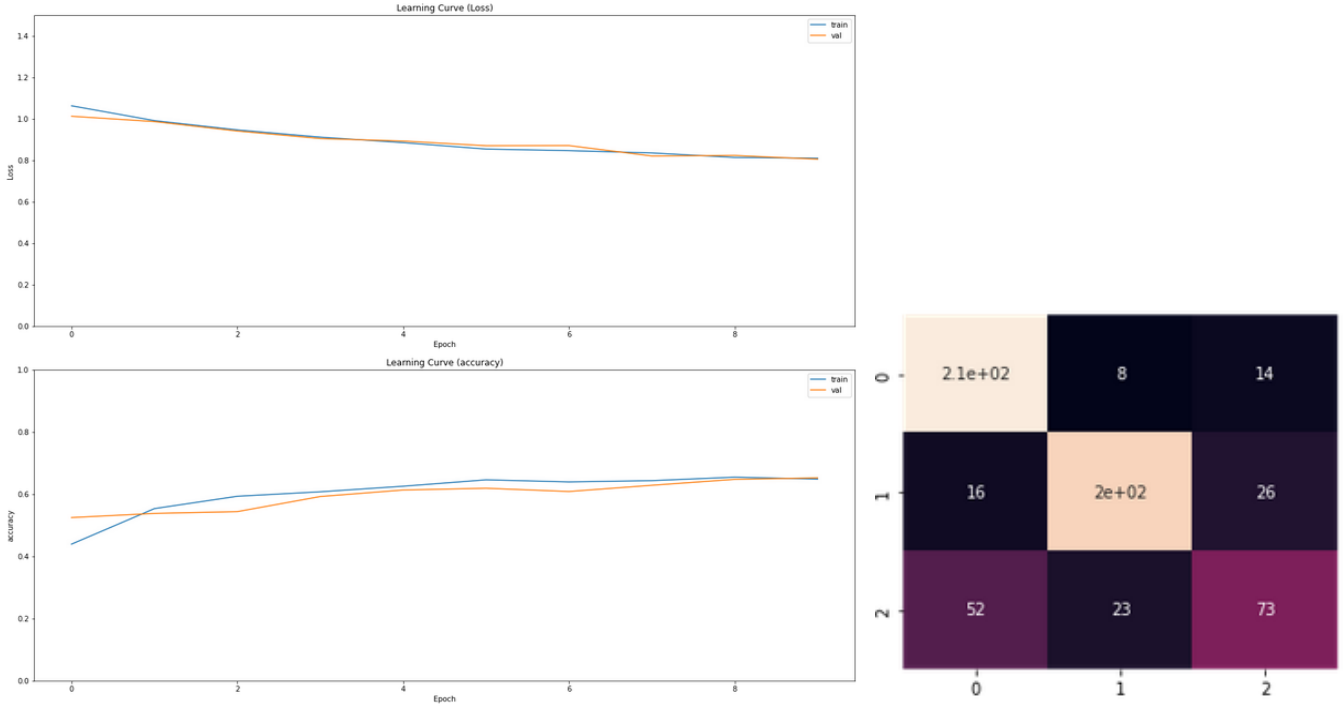



figA: modèle, courbe et heatmap du modèle 1 pour le jeu de données avec 3 classes.

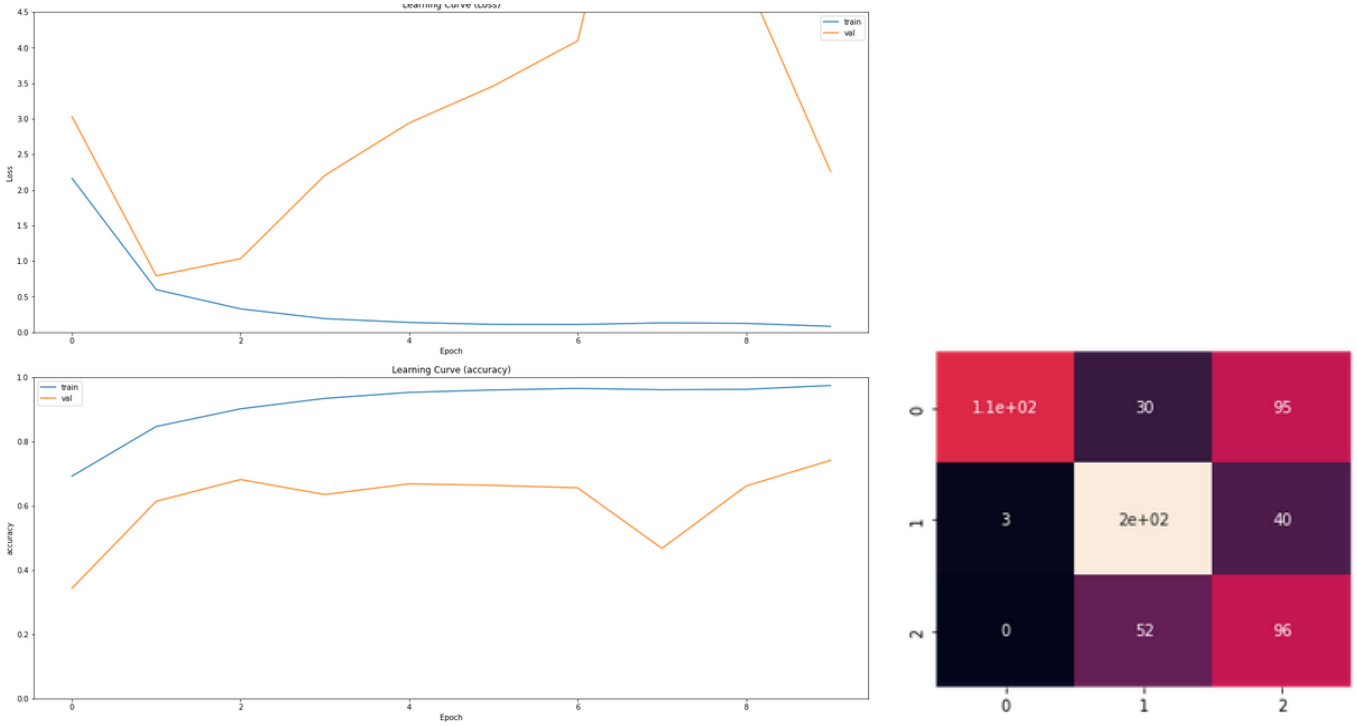



figB: courbe et heatmap du modèle 2 pour le jeu de données avec 3 classes. Ce modèle est mais que le modèle 1 mais avec de la data augmentation




figC: courbe et heatmap du modèle 3 pour le jeu de données avec 3 classes. Ce modèle est mais que le modèle 1 mais avec le changement du learning rate.

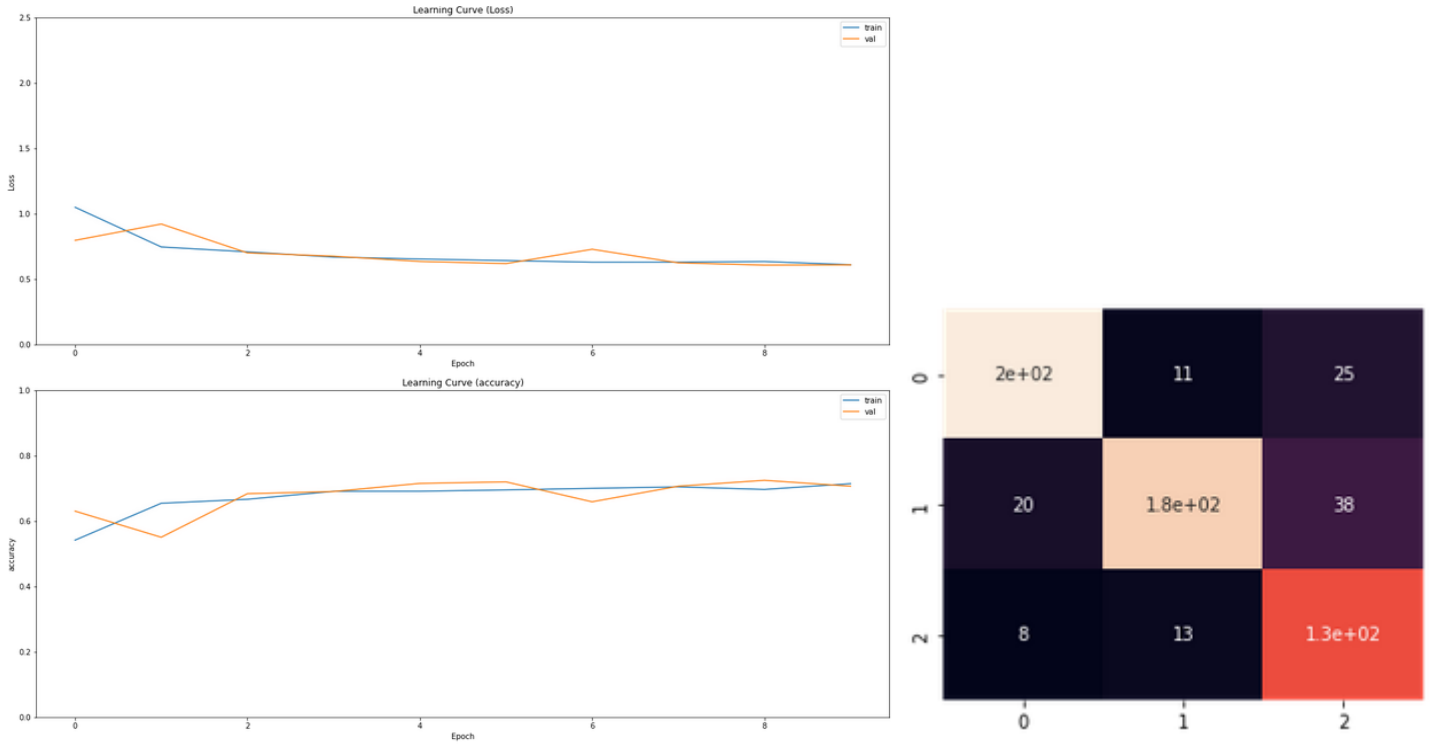
```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 1)]     0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 222, 222, 8)       80
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 8)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 98568)             0
_________________________________________________________________
batch_normalization (BatchNo (None, 98568)             394272
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 295707
=================================================================
Total params: 690,059
Trainable params: 492,923
Non-trainable params: 197,136
_________________________________________________________________
```




figD: modèle, courbe et heatmap du modèle 4 pour le jeu de données avec 3 classes.

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 1)]     0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 222, 222, 8)       80
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 8)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 109, 109, 16)      1168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 54, 54, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 46656)             0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 139971
=================================================================
Total params: 141,219
Trainable params: 141,219
Non-trainable params: 0
```

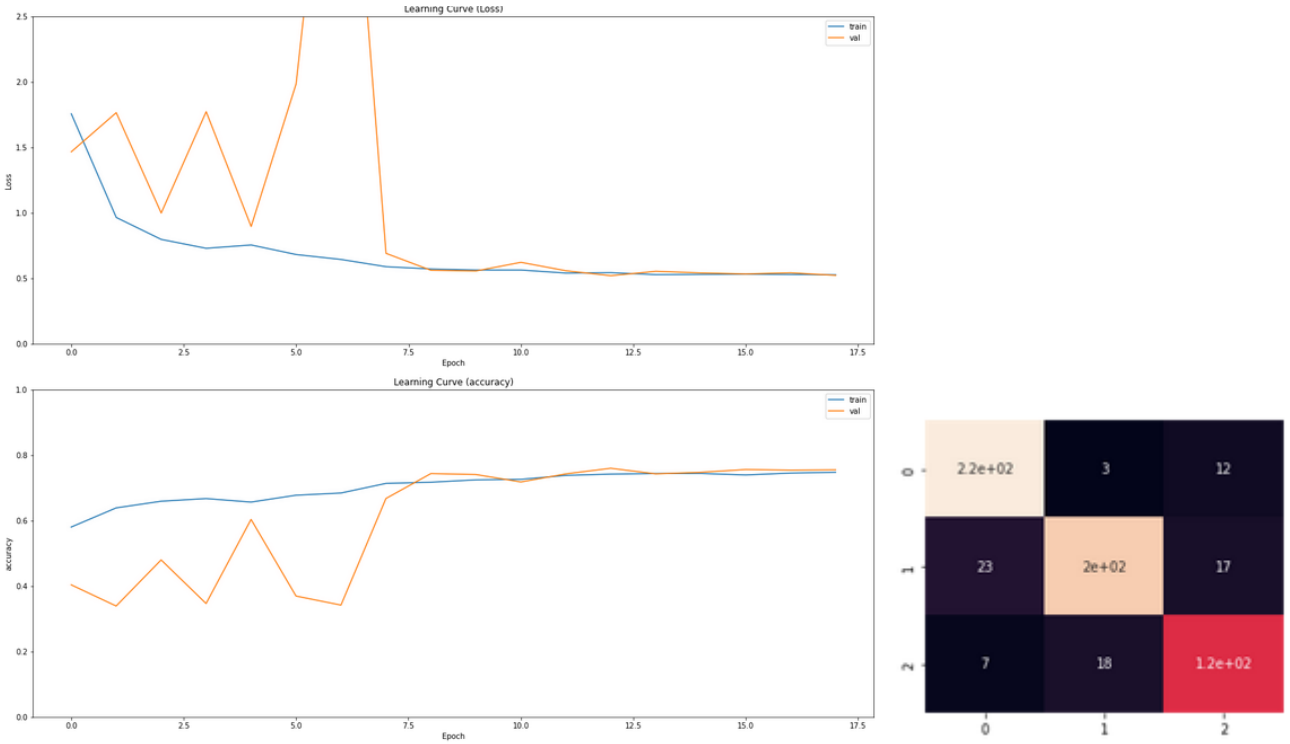
figE.1: modèle 5 pour le jeu de données avec 3 classes.




figE.2: courbe et heatmap du modèle 5 pour le jeu de données avec 3 classes.

```
Model: "model_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, 224, 224, 1)]     0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 222, 222, 8)       80
_________________________________________________________________
max_pooling2d_32 (MaxPooling (None, 111, 111, 8)       0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 109, 109, 16)      1168
_________________________________________________________________
max_pooling2d_33 (MaxPooling (None, 54, 54, 16)        0
_________________________________________________________________
flatten_18 (Flatten)         (None, 46656)             0
_________________________________________________________________
batch_normalization_2 (Batch (None, 46656)             186624
_________________________________________________________________
dense_30 (Dense)             (None, 3)                 139971
=================================================================
Total params: 327,843
Trainable params: 234,531
Non-trainable params: 93,312
```

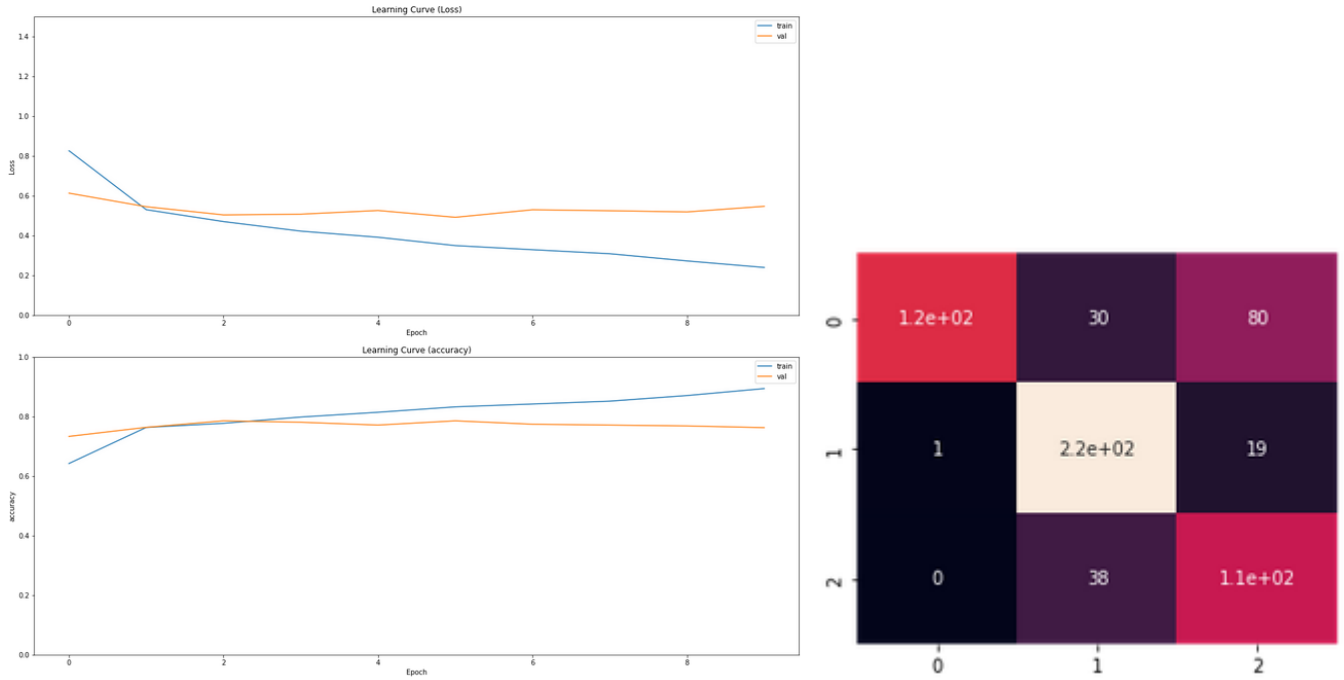
figF.1: modèle 6 pour le jeu de données avec 3 classes.



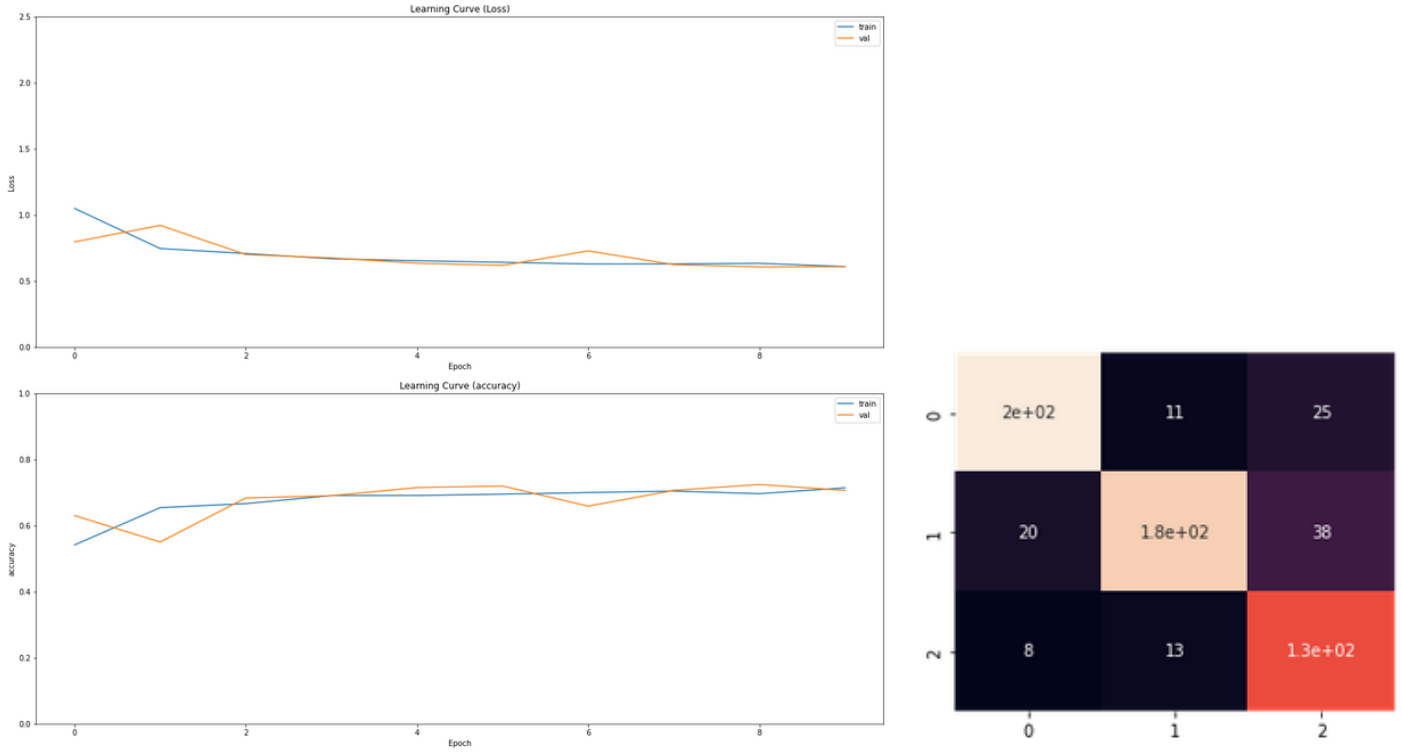

figF.2: courbe et heatmap du modèle 6 pour le jeu de données avec 3 classes.

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 224, 224, 1)]     0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 222, 222, 8)       80
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 111, 111, 8)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 109, 109, 8)       584
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 54, 54, 8)         0
_________________________________________________________________
flatten_5 (Flatten)          (None, 23328)             0
_________________________________________________________________
dense_8 (Dense)              (None, 3)                 69987
=================================================================
Total params: 70,651
Trainable params: 70,651
Non-trainable params: 0
_________________________________________________________________
```

figG.1: modèle 7 pour le jeu de données avec 3 classes.



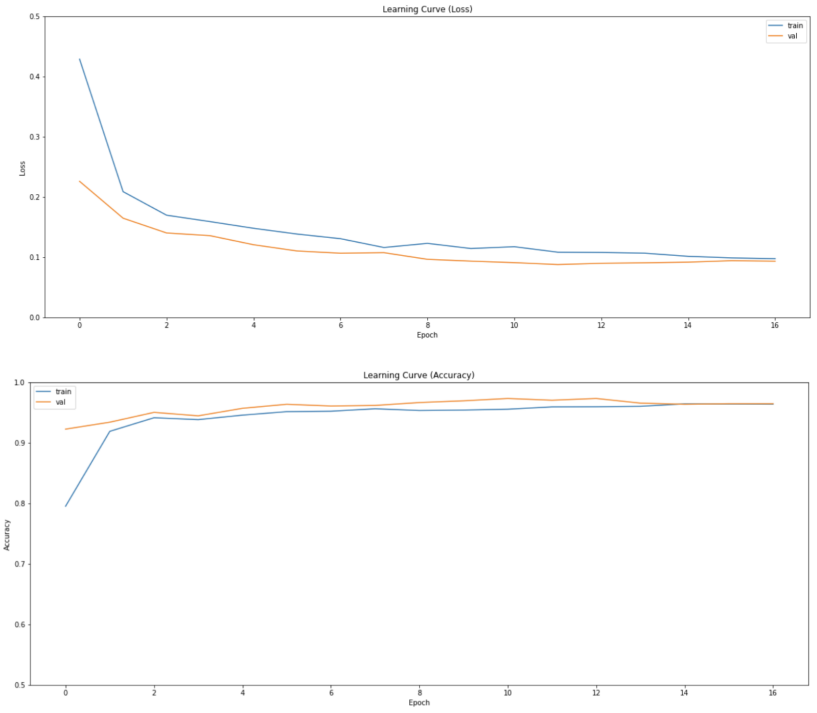

figG.2: courbe et heatmap du modèle 7 pour le jeu de données avec 3 classes.




figH: courbe et heatmap du modèle 8 pour le jeu de données avec 3 classes. C'est le modele 5 avec de la data augmentation.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
resnet152v2 (Functional)     (None, 7, 7, 2048)        58331648
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 128)               262272
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 58,594,049
Trainable params: 262,401
Non-trainable params: 58,331,648
_________________________________________________________________
```

figl_1: modèle Resnet152




figl_2: courbe du modèle Resnet152 pour le jeu de données avec 2 classes.