

Rapport Projet Deep Learning : Airplanes Image Classification



Faical Toubali Hadaoui
Mehdi Sensali
Ignacio Lucas Oros Campo
Thibaud Merieux
Cédric Martin

20 Mars 2021

Contents

1 Description du sujet	4
2 Constitution de la base de données	6
2.1 Méthodologie suivie	6
2.1.1 Acquérir les données	6
2.1.2 Annoter les données	6
2.1.3 Partitionnement des images pour l'entraînement, les tests et la validation	6
2.2 Script de chargement des données	6
3 Liens	7
4 Partie 2 : Description de la résolution	8
4.1 Version 1 : Classification binaire : Commercial/Militaire	8
4.1.1 Réseau utilisé : AlexNet simplifié	8
4.1.2 Methodologie suivie :	9
4.2 Version 2 : Classification multi-classes : 10 labels différents correspondant aux différents modèles d'avions militaires/commerciaux que contient notre base de données :	9
5 Partie 3 : Analyse de résultats	15
5.1 Version 1 : Classification binaire : Commercial/Militaire	15
5.1.1 Mise en oeuvre 1:	15
5.1.2 Mise en oeuvre 2:	17
5.1.3 Mise en oeuvre 3:	18
5.1.4 Conclusion sur la version 1:	19
5.2 Version 2 : Classification multi-classes : 10 labels différents correspondant aux différents modèles d'avions militaires/commerciaux que contient notre base de données :	20
5.2.1 Mise en oeuvre 1:	21
5.2.2 Mise en oeuvre 2:	22
5.2.3 Mise en oeuvre 3:	23
5.2.4 Mise en oeuvre 4:	25
5.2.5 Mise en oeuvre 5:	26
5.2.6 Mise en oeuvre 6:	27
6 Conclusion	29

List of Figures

1	Dassault Rafale	4
2	Lockheed Martin F-35 Lightning II	4
3	Eurofighter Typhoon	4
4	A400M Atlas Airbus	4
5	Soukhoi Su-57	4
6	Airbus A380	5
7	Airbus Beluga	5
8	Boeing 747	5
9	Boeing 777	5
10	Dassault Falcon 8X	5
11	Version 1 : les paramètres du réseau	8
12	Méthodologie suivie : Josh Tobin	9
13	Version 2 : Labels des avions	10
14	Version 2 : Premier réseau de neurones	11
15	Version 2 : réseaux de neurones pré-entraînées dans la documentation officielle de Keras	12
16	Version 2 : VGG-16 pré-entraîné	13
17	Version 2 : Classifieur extracteur de caractéristiques	13
18	Version 2 : Modèle entraîné en gelant les poids du VGG-16	14
19	Version 2 : Hyperparamètres du réseau en mise en oeuvre 4	14
20	Version 2 : Modèle entraîné en dégelant les poids du VGG-16	14
21	Version 2 : Hyperparamètres du réseau en mise en oeuvre 5	15
22	Version 1 : Analyse des résultats de la mise en oeuvre 1	16
23	Version 1 : Analyse des résultats de la mise en oeuvre 2	17
24	Version 1 : évolutions des 5 dernières epochs de la mise en oeuvre 2	17
25	Version 1 : Analyse des résultats de la mise en oeuvre 3	18
26	Version 1 : évolutions des 5 dernières epochs de la mise en oeuvre 3	18
27	Version 1 : les métriques du modèle en mise en oeuvre 3	19
28	Version 1 : Matrice de confusion de la mise en oeuvre 3	19
29	Version 2 : Analyse des résultats de la mise en oeuvre 3	21
30	Version 2 : Adresser un sur-apprentissage par la méthodologie de Josh	22
31	Version 2 : Analyse des résultats de la mise en oeuvre 2	22
32	Version 2 : évolutions des 5 dernières epochs de la mise en oeuvre 3	23
33	Version 2 : Analyse des résultats de la mise en oeuvre 3	23
34	Version 2 : les métriques du modèle en mise en oeuvre 3	24
35	Version 2 : Matrice de confusion de la mise en oeuvre 3	24
36	Version 2 : Analyse des résultats de la mise en oeuvre 4	25
37	Version 2 : Analyse des résultats de la mise en oeuvre 5	26
38	Version 2 : Derniers 5 Epochs de la mise en oeuvre 5	26
39	Version 2 : Analyse des résultats de la mise en oeuvre 6	27
40	Version 2 : Derniers 5 Epochs de la mise en oeuvre 6	27
41	Version 2 : les métriques du modèle en mise en oeuvre 6	28
42	Version 2 : Matrice de confusion de la mise en oeuvre 6	28

1 Description du sujet

Dans le cadre de ce projet nous nous proposons de réaliser un classifieur d'images d'avions. Dans une première version nous allons séparer les avions commerciaux et les avions militaires et puis dans une seconde version nous allons distinguer entre 10 différentes classes (5 pour chaque catégorie).

Les modèles d'avion qui constitueront les classes seront les suivants :

Avions militaires



Figure 1: Dassault Rafale



Figure 2: Lockheed Martin F-35 Lightning II



Figure 3: Eurofighter Typhoon



Figure 4: A400M Atlas Airbus



Figure 5: Soukhoi Su-57

Avions commerciaux



Figure 6: Airbus A380



Figure 7: Airbus Beluga



Figure 8: Boeing 747



Figure 9: Boeing 777



Figure 10: Dassault Falcon 8X

2 Constitution de la base de données

2.1 Méthodologie suivie

2.1.1 Acquérir les données

Pour acquérir les données nous avons recherché successivement dans le moteur de recherche les différents modèles d'avion constituant les classes de notre sujet. Ensuite nous avons extrait tous les URL chargés de google images sous la forme d'un fichier texte grâce à un code javascript introduit sur la console du navigateur. Ce fichier texte contenant les URL est à son tour passé en paramètre à un script Python qui télécharge chaque photo dont l'URL est listé et les garde toutes dans un dossier. Finalement il faut aller dans le dossier des images téléchargés et faire un filtrage des images qui sont valides à la main ainsi que supprimer le maximum de doublons possibles manuellement.

2.1.2 Annoter les données

Nous avons décidé d'utiliser un script écrit en python pour annoter les images téléchargées. En effet même si avec une base de donnée de 4000 images ce travail pourrait être fait à la main, un tel travail serait tédieux et long à faire donc dans l'idée d'optimiser le processus. L'utilisation d'un script est l'option la plus rapide. Une fois l'annotation réalisée, nous vérifions rapidement que le label associé à chaque photo est bien cohérent.

2.1.3 Partitionnement des images pour l'entraînement, les tests et la validation

La proportion choisie pour partitionner le dataset est la suivante : 70% des images seront dédiées à l'entraînement, 15% des images seront dédiées à la vérification et 15 % au test.

2.2 Script de chargement des données

Pour charger les images que notre base de données contient, nous avons utilisé le script Notebook déposé sur Moodle en l'adaptant à notre base de données d'images **Dataset** qui contient des sous-dossiers correspondant aux types d'avions (**Commercial / Militaire**) . Chaque sous-dossier comporte lui même les modèles d'avions. Le sous-dossier **Militaire** contient les sous-dossiers correspondant aux différents modèles d'avions :

- Dassault Rafale
- Lockheed Martin F-35 Lightning II
- Eurofighter Typhoon
- A400M d'Airbus
- Soukhoi Su-57

et Le sous-dossier **Commercial** contient les sous-dossiers correspondant aux différents modèles d'avions :

- Airbus A380
- Airbus Beluga
- Boeing 747
- Boeing 777

- Dassault Falcon 8X

Pour héberger notre base de données nous avons choisi GitHub, ainsi un Git clone sera très rapide depuis Google Colab.

Dans la **version 1**, nous avons divisé la base de données en 2 dossiers : Commercial et Militaire en revanche, dans la **version 2** nous avons réparti les données en 10 classes qui correspondent chacune à un modèle d'avion.

3 Liens

Lien vers les images de la base de données sur github

https://github.com/faicaltoubali/Airplanes_Image_Classification

4 Partie 2 : Description de la résolution

4.1 Version 1 : Classification binaire : Commercial/Militaire

4.1.1 Réseau utilisé : AlexNet simplifié

Pour cette version simple de notre sujet, nous avons utilisé le réseau de neurones AlexNet simplifié utilisé en TP3.

Ce réseau se compose de 6 couches comptant la couche de sortie. Chaque couche est constituée d'une convolution et d'un max-pooling.

- Couche 1 : 32 filtres de convolution + fonction d'activation ReLu
- Couche 2 : 64 filtres de convolution + fonction d'activation ReLu
- Couche 3 : 96 filtres de convolution + fonction d'activation ReLu
- Couche 4 : 128 filtres de convolution + fonction d'activation ReLu
- Couche 5 : Couche dense de 512 neurones + fonction d'activation ReLu
- Couche de sortie : 1 seul neurone avec une fonction d'activation sigmoid puisqu'on est dans le cas d'un problème binaire

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_12 (Conv2D)	(None, 150, 150, 32)	896
<hr/>		
max_pooling2d_12 (MaxPooling)	(None, 75, 75, 32)	0
<hr/>		
conv2d_13 (Conv2D)	(None, 75, 75, 64)	18496
<hr/>		
max_pooling2d_13 (MaxPooling)	(None, 37, 37, 64)	0
<hr/>		
conv2d_14 (Conv2D)	(None, 37, 37, 96)	55392
<hr/>		
max_pooling2d_14 (MaxPooling)	(None, 18, 18, 96)	0
<hr/>		
conv2d_15 (Conv2D)	(None, 18, 18, 128)	110720
<hr/>		
max_pooling2d_15 (MaxPooling)	(None, 9, 9, 128)	0
<hr/>		
flatten_3 (Flatten)	(None, 10368)	0
<hr/>		
dense_4 (Dense)	(None, 512)	5308928
<hr/>		
dense_5 (Dense)	(None, 1)	513
<hr/>		
Total params: 5,494,945		
Trainable params: 5,494,945		
Non-trainable params: 0		

Figure 11: Version 1 : les paramètres du réseau

Le réseau a comme nombre total de paramètres : 5,494,945 qui doivent tous être entraînés. Il a pour fonction de coût (fonction objectif) l'Entropie croisée avec un optimiseur Adam qui est généralement plus stable et donne de meilleurs résultats. Le taux d'apprentissage est fixé à 3×10^{-4} .

4.1.2 Méthodologie suivie :

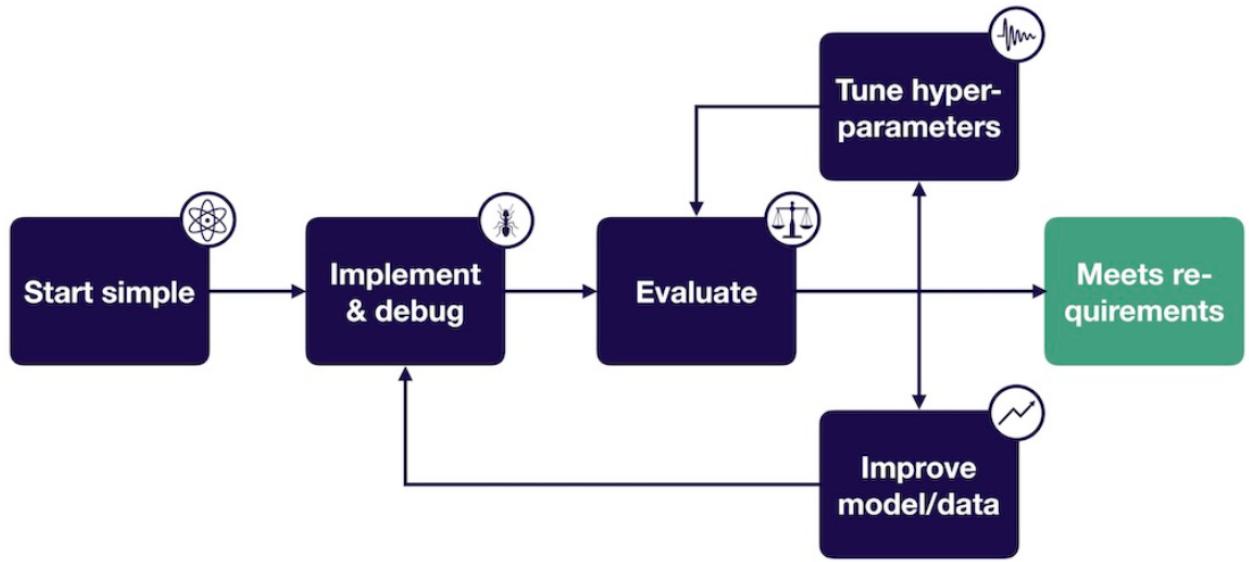


Figure 12: Méthodologie suivie : Josh Tobin

Pour résoudre la version du problème binaire, nous avons opté pour la méthodologie définie par Josh Tobin pour le Troubleshooting des réseaux de Deep Learning. En effet, cette méthodologie est bien structurée et bien siblée permettant de débugger, d'évaluer, puis de boucler en vue d'améliorer le modèle ou les hyperparamètres jusqu'à avoir les résultats attendus.

Nous allons alors démarrer simplement en premier lieu en utilisant des paramètres par défauts et puis nous allons augmenter la capacité de notre réseau jusqu'à avoir des résultats satisfaisants.

- Mise en oeuvre 1 : 10 epochs, sans augmentation de données, sans régularisation. Analyse de résultats en [5.1.1](#)
- Mise en oeuvre 2 : 10 epochs, avec augmentation de données, sans régularisation. Analyse de résultats en [5.1.2](#)
- Mise en oeuvre 3 : 40 epochs, avec augmentation de données, sans régularisation. Analyse de résultats en [5.1.3](#)

4.2 Version 2 : Classification multi-classes : 10 labels différents correspondant aux différents modèles d'avions militaires/commerciaux que contient notre base de données :

Dans cette partie, nous avons labelisé chacun des avions d'un label de la forme suivante : type/modele. En l'occurrence, un rafale sera labelisé par le label suivant : Military/Rafale et un A380 sera labelisé par le label suivant : Commercial/Airbus_A380. A chaque label on associera un entier.

- 0 : Commercial/Airbus_A380
- 1 : Commercial/Airbus_Beluga

- 2 : Commercial/Boeing_747
- 3 : Commercial/Boeing_777
- 4 : Commercial/Falcon_8x
- 5 : Military/A400M
- 6 : Military/Eurofighter_Typhoon
- 7 : Military/Rafale
- 8 : Military/F-35
- 9 : Military/Su-57

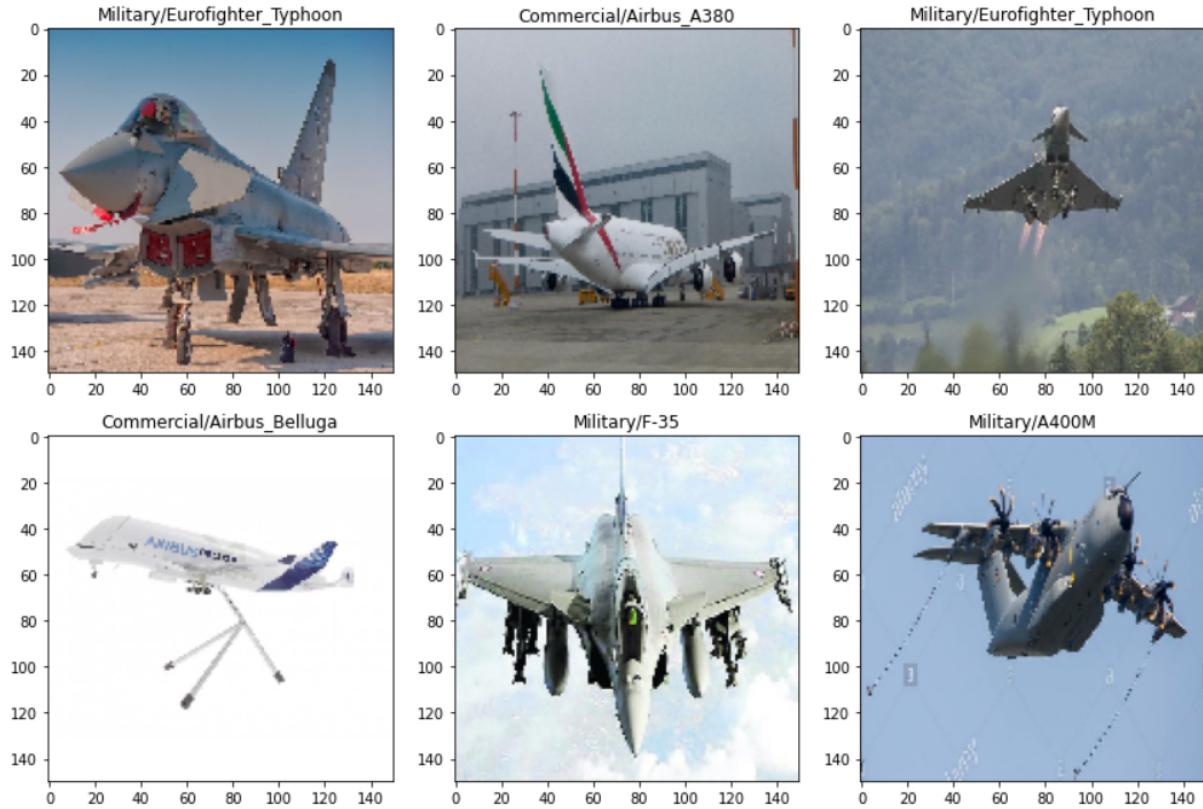


Figure 13: Version 2 : Labels des avions

Ainsi, nous aurons 10 labels associés à nos 10 différents modèles d'avions. Le générateur d'images d'entraînement génère un ensemble de données dont le nombre est *batch_size* et dont la dimension est *image_size*, par contre un label sera ici sous forme d'un tableau de taille 10 qui contient l'entier 1 dans la position correspondante au label de l'avion. Le label d'une Rafale dans Dataframe généré sera le tableau suivant : [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

Pour commencer en douceur, nous avons choisi un réseau de neurones dont les couches cachées et les hyperparamètres sont les mêmes que celui qu'on a utilisé en version 1 sauf que dans ce cas la couche de

sortie sera d'une dimension de 10 puisqu'on est dans un problème de 10 classes où il faut prédire un tableau de dix entiers.

Model: "sequential_10"		
Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 150, 150, 32)	896
max_pooling2d_12 (MaxPooling)	(None, 75, 75, 32)	0
conv2d_13 (Conv2D)	(None, 75, 75, 64)	18496
max_pooling2d_13 (MaxPooling)	(None, 37, 37, 64)	0
conv2d_14 (Conv2D)	(None, 37, 37, 96)	55392
max_pooling2d_14 (MaxPooling)	(None, 18, 18, 96)	0
conv2d_15 (Conv2D)	(None, 18, 18, 128)	110720
max_pooling2d_15 (MaxPooling)	(None, 9, 9, 128)	0
flatten_4 (Flatten)	(None, 10368)	0
dense_20 (Dense)	(None, 512)	5308928
dense_21 (Dense)	(None, 10)	5130
<hr/>		
Total params: 5,499,562		
Trainable params: 5,499,562		
Non-trainable params: 0		

Figure 14: Version 2 : Premier réseau de neurones

Comme dans la version 1, nous avons opté pour la méthodologie définie par Josh Tobin pour le Troubleshooting du réseau de neurones.

- Mise en oeuvre 1 : 40 epochs + sans augmentation de données + avec une simple régularisation Weight Decay de Lasso $l1$. Analyse de résultats : [5.2.1](#)
- Mise en oeuvre 2 : 40 epochs + avec augmentation de données + avec une simple régularisation Weight Decay de Lasso $l1$. Analyse de résultats : [5.2.2](#)
- Mise en oeuvre 3 : 80 epochs + avec augmentation de données + avec augmentation de la capacité du réseau + avec une simple régularisation Weight Decay de Lasso $l1$. Analyse de résultats : [5.2.3](#)

D'après les résultats de la mise en oeuvre 3, il parrait que les couches du premier réseau de neurones convolutif n'ont pas apprit suffisamment de filtres afin de détecter les caractéristiques propres à chaque avion. Nous avons donc opté à utiliser une méthode largement utilisé dans l'entraînement de réseau de neurones connu par sa puissance à généraliser aux mieux les caractéristiques. Il s'agit de l'apprentissage par transfert (Learning Transfert) avec des réseaux de neurones pré-entraînées.

Les modèles pré-entraînés sont utilisés des trois manières courantes suivantes lors de la création de nouveaux modèles ou de leur réutilisation : Extraction des caractéristiques d'un réseau de neurones pré-entraîné, Utilisation des caractéristiques d'une réseau de neurones entraînées sur les données augmenté et finalement le Fine Tuning qui consiste à 'dégeler' les neurones bloqués pour faire évoluer le modèle en totalité.

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-
EfficientNetB0	29 MB	-	-	5,330,571	-
EfficientNetB1	31 MB	-	-	7,856,239	-
EfficientNetB2	36 MB	-	-	9,177,569	-
EfficientNetB3	48 MB	-	-	12,320,535	-
EfficientNetB4	75 MB	-	-	19,466,823	-

Figure 15: Version 2 : réseaux de neurones pré-entraînées dans la documentation officielle de Keras

Dans notre cas, nous utiliserons le fameux modèle pré-entraîné de l'université Oxford VGG-16 qui est spécialisé dans la construction de réseaux convolutif spécifique à la reconnaissance visuelle. Nous allons essayé aussi le ResNet50. Ils sont tous les deux entraînés sur une grande base de données (ImageNet) de l'ordre de millions d'images.

Dans la mise en oeuvre 4, nous allons construire un classifieur qui va tirer profit des caractéristiques du réseau du VGG-16 entraîné sur une grande base de données même si cette dernière ne contient pas des images similaires à notre propre base de données.

```

Model: "vgg16"
=====
Layer (type)          Output Shape         Param #
=====
input_4 (InputLayer)   [(None, 150, 150, 3)]   0
block1_conv1 (Conv2D)  (None, 150, 150, 64)    1792
block1_conv2 (Conv2D)  (None, 150, 150, 64)    36928
block1_pool (MaxPooling2D) (None, 75, 75, 64)   0
block2_conv1 (Conv2D)  (None, 75, 75, 128)    73856
block2_conv2 (Conv2D)  (None, 75, 75, 128)    147584
block2_pool (MaxPooling2D) (None, 37, 37, 128)  0
block3_conv1 (Conv2D)  (None, 37, 37, 256)   295168
block3_conv2 (Conv2D)  (None, 37, 37, 256)   590080
block3_conv3 (Conv2D)  (None, 37, 37, 256)   590080
block3_pool (MaxPooling2D) (None, 18, 18, 256)  0
block4_conv1 (Conv2D)  (None, 18, 18, 512)   1180160
block4_conv2 (Conv2D)  (None, 18, 18, 512)   2359808
block4_conv3 (Conv2D)  (None, 18, 18, 512)   2359808
block4_pool (MaxPooling2D) (None, 9, 9, 512)   0
block5_conv1 (Conv2D)  (None, 9, 9, 512)   2359808
block5_conv2 (Conv2D)  (None, 9, 9, 512)   2359808
block5_conv3 (Conv2D)  (None, 9, 9, 512)   2359808
block5_pool (MaxPooling2D) (None, 4, 4, 512)   0
=====
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0

```

Figure 16: Version 2 : VGG-16 pré-entraîné

```

Model: "sequential"
=====
Layer (type)          Output Shape         Param #
=====
dense (Dense)         (None, 256)        2097408
dropout (Dropout)     (None, 256)        0
dense_1 (Dense)       (None, 10)         2570
=====
Total params: 2,099,978
Trainable params: 2,099,978
Non-trainable params: 0

```

Figure 17: Version 2 : Classifieur extracteur de caractéristiques

- Mise en oeuvre 4 : Transfer Learning : VGG-16 comme extracteur de caractéristiques pour qu'il soit utilisable dans le transfert learning [5.2.4](#)

```
Model: "sequential_9"
-----  

Layer (type)          Output Shape       Param #
-----  

vgg16 (Functional)    (None, 4, 4, 512)   14714688  

-----  

flatten_9 (Flatten)   (None, 8192)        0  

-----  

dense_20 (Dense)      (None, 256)         2097408  

-----  

dense_21 (Dense)      (None, 10)          2570  

-----  

Total params: 16,814,666  

Trainable params: 2,099,978  

Non-trainable params: 14,714,688
```

Figure 18: Version 2 : Modèle entraîné en gelant les poids du VGG-16

```
1 model2.compile(loss='categorical_crossentropy',
2                  optimizer=optimizers.Adam(lr=1e-4),
3                  metrics=['acc'])
4
5 history = model2.fit(
6     train_generator_augmented2,
7     epochs=50,
8     validation_data=validation_generator2)
```

Figure 19: Version 2 : Hyperparamètres du réseau en mise en oeuvre 4

- Mise en oeuvre 5 : Transfer Learning : VGG-16 avec augmentation de base de données pour entraîner le réseau attaché au VGG-16 afin que l'on puisse effectuer le fine tuning [5.2.5](#)

```
2 conv_base.trainable = True
3 model2.summary()

Model: "sequential_9"
-----  

Layer (type)          Output Shape       Param #
-----  

vgg16 (Functional)    (None, 4, 4, 512)   14714688  

-----  

flatten_9 (Flatten)   (None, 8192)        0  

-----  

dense_20 (Dense)      (None, 256)         2097408  

-----  

dense_21 (Dense)      (None, 10)          2570  

-----  

Total params: 16,814,666  

Trainable params: 16,814,666  

Non-trainable params: 0
```

Figure 20: Version 2 : Modèle entraîné en dégelant les poids du VGG-16

```
1 model2.compile(loss='categorical_crossentropy',
2                  optimizer=optimizers.Adam(lr=1e-5),
3                  metrics=['acc'])
4
5 history = model2.fit(
6     train_generator_augmented2,
7     epochs=40,
8     validation_data=validation_generator2)
```

Figure 21: Version 2 : Hyperparamètres du réseau en mise en oeuvre 5

- Mise en oeuvre 6 : Fine Tuning : VGG-16 avec augmentation de base de données afin d'offrir un maximum de performances [5.2.6](#)

5 Partie 3 : Analyse de résultats

5.1 Version 1 : Classification binaire : Commercial/Militaire

5.1.1 Mise en oeuvre 1:

La mise en oeuvre 1 correspond à un entrainement avec 10 Epochs sans régularisation et sans augmentation de base de données. Avec des paramètres par défauts définis dans la présentation du réseau en partie 2.

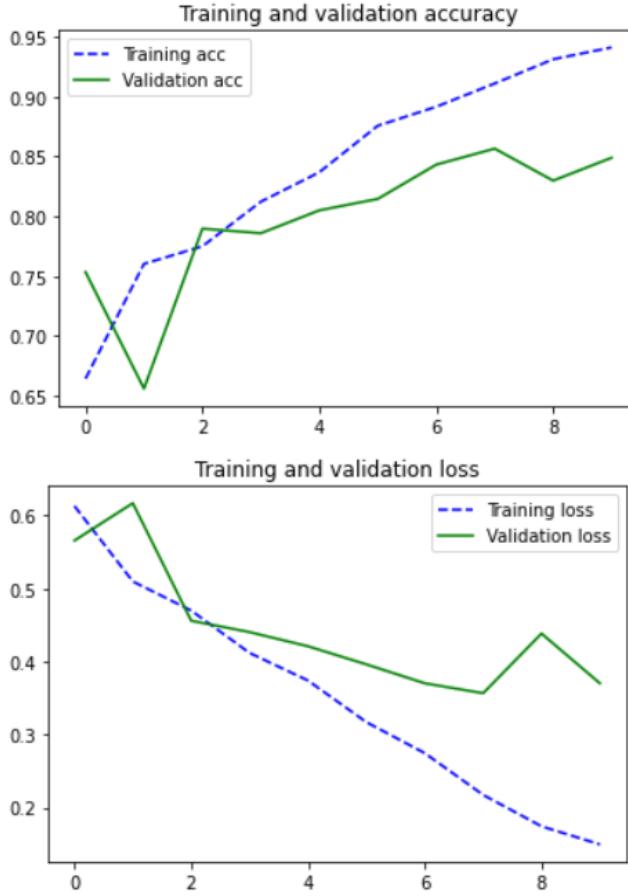


Figure 22: Version 1 : Analyse des résultats de la mise en oeuvre 1

Nous constatons bien évidemment un sur-apprentissage (Over-fitting). Le problème provient probablement de la capacité de la base de données qui est très petite par rapport à la complexité du réseau de neurones. En effet, le réseau que nous avons utilisé contient 5 millions de paramètres à entraîner et cela avec une base de données de l'ordre de 4000 images. Donc nous pouvons procéder de la manière suivante :

- Ajouter mécaniquement des données : Pratiquement ce n'est pas possible car nous avons essayé de collecter le maximum d'images pertinentes d'avions que nous pouvions à partir de plusieurs moteurs de recherche.
- Augmenter les données en exploitant la base de données élaborée.
- Régularisation.

Nous allons choisir l'approche de l'augmentation de base de données. C'est une pratique rusée et pertinente puisque les réseaux de neurones n'ont pas la capacité à détecter avec précision les formes d'images, ainsi, en appliquant des transformations d'images (translations, rotations, ajout d'effets ...) sur une même image, les réseaux de neurones apprennent à chaque fois.

5.1.2 Mise en oeuvre 2:

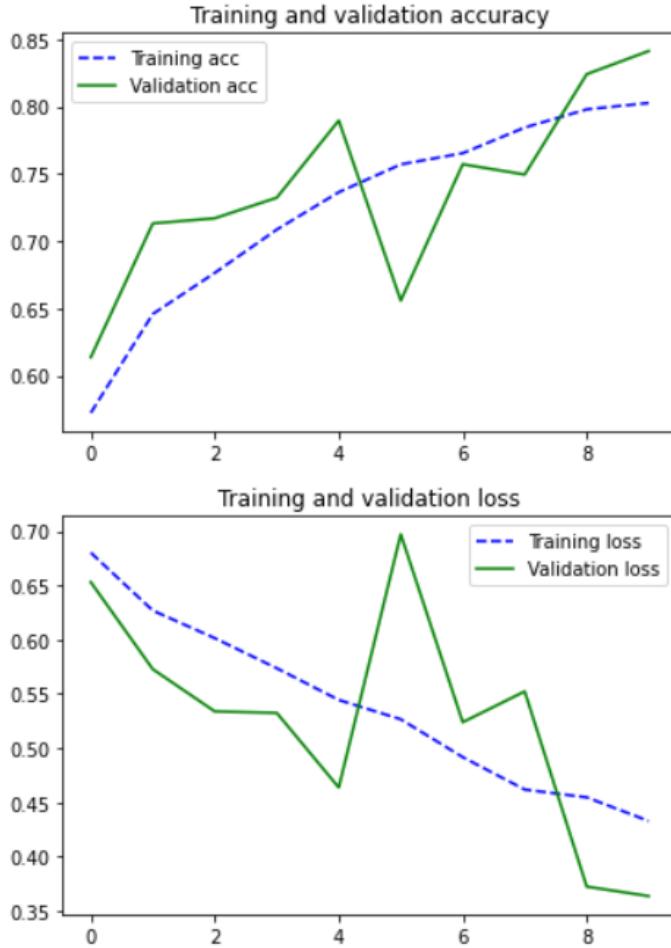


Figure 23: Version 1 : Analyse des résultats de la mise en oeuvre 2

```

Epoch 5/10
158/158 [=====] - 68s 431ms/step - loss: 0.5577 - acc: 0.7349 - val_loss: 0.4637 - val_acc: 0.7897
Epoch 6/10
158/158 [=====] - 66s 419ms/step - loss: 0.5443 - acc: 0.7390 - val_loss: 0.6972 - val_acc: 0.6558
Epoch 7/10
158/158 [=====] - 69s 436ms/step - loss: 0.4997 - acc: 0.7578 - val_loss: 0.5239 - val_acc: 0.7572
Epoch 8/10
158/158 [=====] - 67s 426ms/step - loss: 0.4844 - acc: 0.7699 - val_loss: 0.5523 - val_acc: 0.7495
Epoch 9/10
158/158 [=====] - 66s 421ms/step - loss: 0.4556 - acc: 0.7996 - val_loss: 0.3723 - val_acc: 0.8241
Epoch 10/10
158/158 [=====] - 66s 420ms/step - loss: 0.4206 - acc: 0.8228 - val_loss: 0.3636 - val_acc: 0.8413

```

Figure 24: Version 1 : évolutions des 5 dernières epochs de la mise en oeuvre 2

Nous constatons que le sur-apprentissage s'est limité par rapport à la mise en oeuvre précédente, toutefois, les résultats demeurent insatisfaisants puisque le risque espéré est très élevé aussi que le risque empirique est supérieure au risque espéré ce qui est considéré comme une sorte de sous-apprentissage, ainsi, nous allons

effectuer une augmentation du nombre d'Epochs pour permettre au réseau de s'entraîner plus et d'analyser par suite le comportement d'entraînement sur grande échelle.

5.1.3 Mise en oeuvre 3:

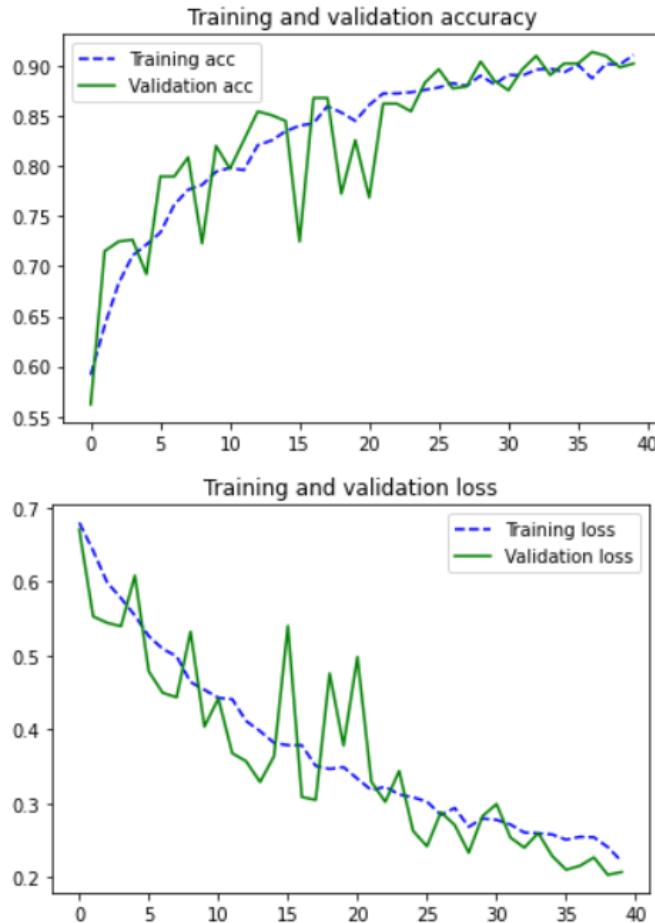


Figure 25: Version 1 : Analyse des résultats de la mise en oeuvre 3

```

Epoch 35/40
158/158 [=====] - 63s 396ms/step - loss: 0.2568 - acc: 0.8966 - val_loss: 0.2289 - val_acc: 0.9025
Epoch 36/40
158/158 [=====] - 63s 396ms/step - loss: 0.2375 - acc: 0.9107 - val_loss: 0.2102 - val_acc: 0.9025
Epoch 37/40
158/158 [=====] - 63s 396ms/step - loss: 0.2447 - acc: 0.8926 - val_loss: 0.2155 - val_acc: 0.9140
Epoch 38/40
158/158 [=====] - 63s 397ms/step - loss: 0.2443 - acc: 0.9067 - val_loss: 0.2269 - val_acc: 0.9101
Epoch 39/40
158/158 [=====] - 65s 409ms/step - loss: 0.2449 - acc: 0.8993 - val_loss: 0.2033 - val_acc: 0.8987
Epoch 40/40
158/158 [=====] - 67s 426ms/step - loss: 0.2200 - acc: 0.9082 - val_loss: 0.2071 - val_acc: 0.9025

```

Figure 26: Version 1 : évolutions des 5 dernières epochs de la mise en oeuvre 3

	precision	recall	f1-score	support
Commercial	0.83	0.86	0.85	299
Military	0.81	0.78	0.79	227
accuracy			0.82	526
macro avg	0.82	0.82	0.82	526
weighted avg	0.82	0.82	0.82	526

Figure 27: Version 1 : les métriques du modèle en mise en oeuvre 3

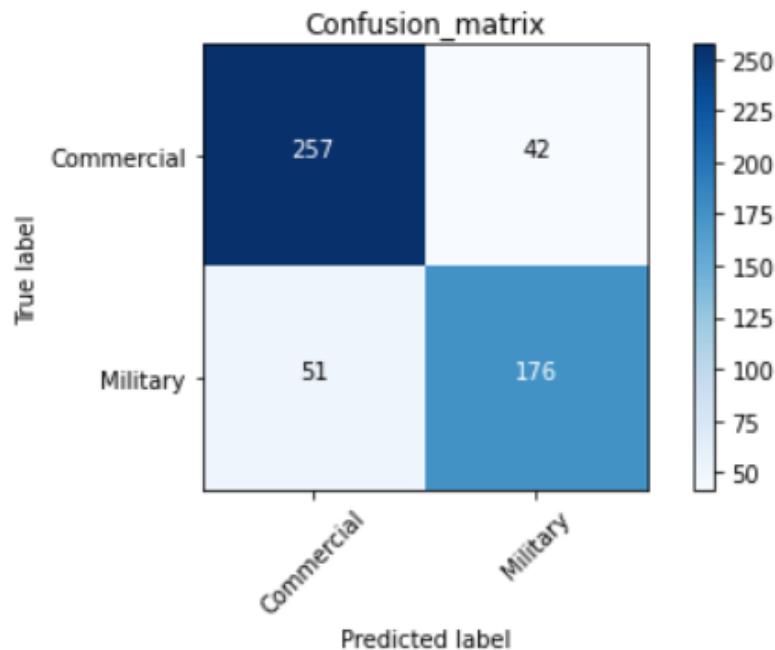


Figure 28: Version 1 : Matrice de confusion de la mise en oeuvre 3

Nous constatons de très bons résultats, le sur-apprentissage est très limité, c'est presque un 'perfect fit' puisque les valeurs du Training Loss et du Validation Loss sont très proches (ils diffèrent d'un léger écart de l'ordre de 0.02). Le modèle apprend très bien puisque le Validation Loss atteint une valeur de 0.20 et la précision résultante est de l'ordre de 90%. La matrice de confusion est de plus à diagonale strictement dominante, ce qui montre que la prédition est très efficace pour cette version.

Les résultats semblent satisfaisants. Toutefois, nous pouvons aller plus loin en appliquant une régularisation adéquate et en enrichissant l'ensemble de validation pour éviter l'instabilité.

5.1.4 Conclusion sur la version 1:

Les résultats apparaissent satisfaisants pour une version aussi simple que la version binaire de notre problème. Nous avons opté d'utiliser le réseau simplifié d'AlexNet car c'est un réseau qui nous est familier

puisque nous l'avons manipulé plusieurs fois dans les séances du TP. Toutefois, nous proposons aussi une autre approche qui apparait plus générale :

Une approche générale avec le réseau LeNet en utilisant de bons paramètres par défauts, à savoir, un optimiseur Adam avec un taux d'apprentissage 3×10^{-4} et sans régularisation et d'appliquer de proche en proche les étapes indiquées par la méthodologie de Josh Tobin en adressant en premier lieu le sous-apprentissage, puis le sur-apprentissage.

Nous pouvons aussi partir loin en utilisant le Transfer Learning et le Fine-Tuning avec le VGG-16. Vu que les résultats précédents sont satisfaisants, nous avons appliqué ceci pour la version 2 qui est plus complexe et nécessite plus de réflexion.

5.2 Version 2 : Classification multi-classes : 10 labels différents correspondant aux différents modèles d'avions militaires/commerciaux que contient notre base de données :

Nous avons chercher dans différentes ressources pour notre problème à 10 classes et nous avons découvert qu'il est d'une complexité importante qui est loin d'être intuitif dans la mesure où les avions commerciaux et militaires dans notre base de données sont très similaires dans leur ensemble qu'il devient parfois même difficile de différencier entre deux avions de même type et de différent modèle à l'oeil nue, cela en dit long pour un réseau artificiel. Prenons l'exemple d'une Rafale et d'une Eurofighter Typhoon ou bien l'exemple d'une Airbus A380 et d'une Boeing 737, par conséquent, l'entraînement d'un réseau de neurones de grande précision sera notre "challenge".

5.2.1 Mise en oeuvre 1:

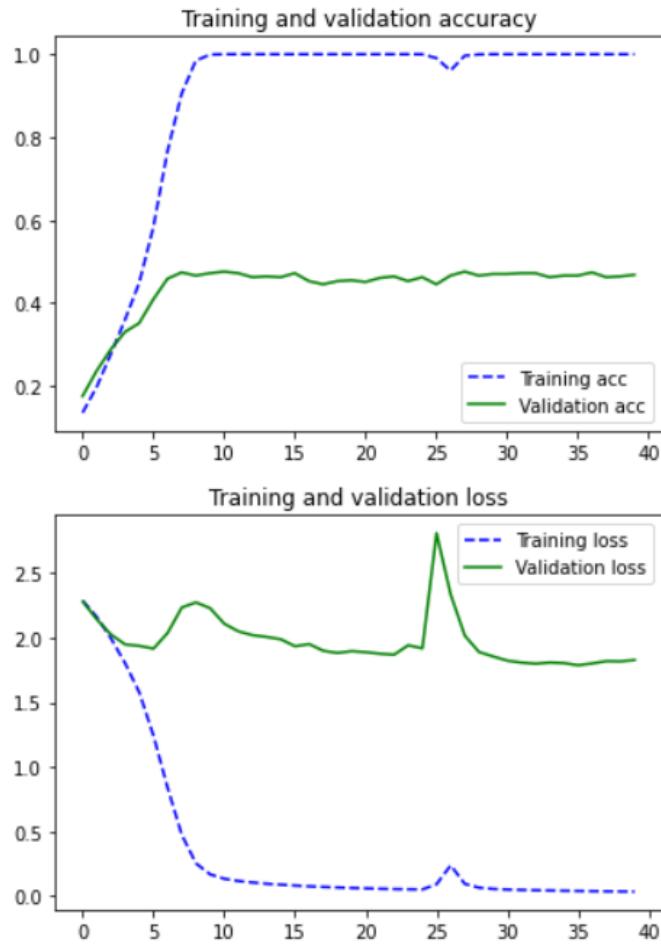


Figure 29: Version 2 : Analyse des résultats de la mise en oeuvre 3

Nous constatons à la volée un sur-apprentissage très fort qui fait mal aux yeux. Néanmoins, il n'était pas surprenant, nous l'attendions. Le problème provient de la capacité de la base de données qui est très faible par rapport à la complexité du réseau de neurones et c'est moins performant que la première version qui partage le dataset d'entraînement seulement sur 2 labels alors qu'ici sur 10 labels.

Nous avons employé une régularisation Weight decay de Lasso $L1$ afin d'éliminer les neurones dont le poids tend vers 0 afin de réduire le sur-apprentissage. Toutefois, cela n'a pas eu un grand effet. Comme l'ajout des données est une contrainte, la première solution consiste à l'augmentation des données comme le montre la figure suivante qui a pour but d'adresser le sur-apprentissage.

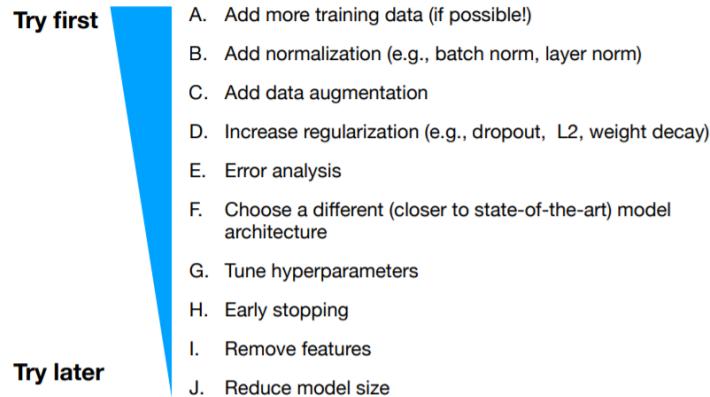


Figure 30: Version 2 : Adresser un sur-apprentissage par la méthodologie de Josh

5.2.2 Mise en oeuvre 2:

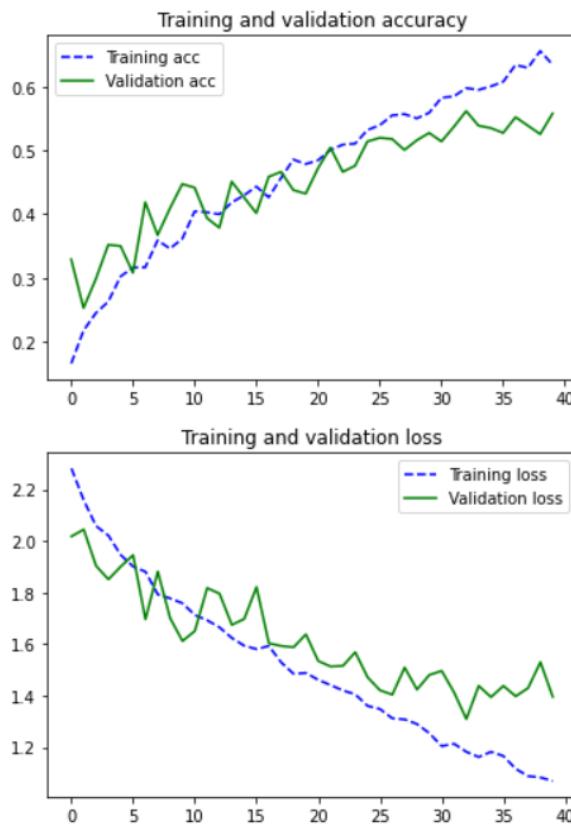


Figure 31: Version 2 : Analyse des résultats de la mise en oeuvre 2

```

Epoch 35/40
158/158 [=====] - 67s 426ms/step - loss: 1.1818 - acc: 0.6010 - val_loss: 1.3940 - val_acc: 0.5354
Epoch 36/40
158/158 [=====] - 67s 423ms/step - loss: 1.1654 - acc: 0.6082 - val_loss: 1.4376 - val_acc: 0.5277
Epoch 37/40
158/158 [=====] - 66s 420ms/step - loss: 1.1155 - acc: 0.6348 - val_loss: 1.3973 - val_acc: 0.5526
Epoch 38/40
158/158 [=====] - 67s 424ms/step - loss: 1.0874 - acc: 0.6300 - val_loss: 1.4291 - val_acc: 0.5392
Epoch 39/40
158/158 [=====] - 67s 424ms/step - loss: 1.0823 - acc: 0.6566 - val_loss: 1.5297 - val_acc: 0.5258
Epoch 40/40
158/158 [=====] - 67s 421ms/step - loss: 1.0687 - acc: 0.6352 - val_loss: 1.3950 - val_acc: 0.5583

```

Figure 32: Version 2 : évolutions des 5 dernières epochs de la mise en oeuvre 3

Effectivement, le sur-apprentissage s'est limité par rapport à la mise en oeuvre précédente,toutefois, nous observons un sous-apprentissage très fort. La validation loss oscille autour de la valeur 1.4 et les valeurs du Training Loss restent environs de 1 après avoir effectué 40 Epochs. Afin de limiter le sous-apprentissage, nous avons décidé d'augmenter la capacité de notre réseau.

- Nous avons utilisé deux layers FC (Fully Connected) de dimension 128 avec une fonction d'activation relu + des couches de Dropout afin de limiter le sur apprentissage
- Nous avons aussi augmenté le nombre d'Epochs en 80 Epochs pour permettre au réseau d'apprendre encore plus

5.2.3 Mise en oeuvre 3:

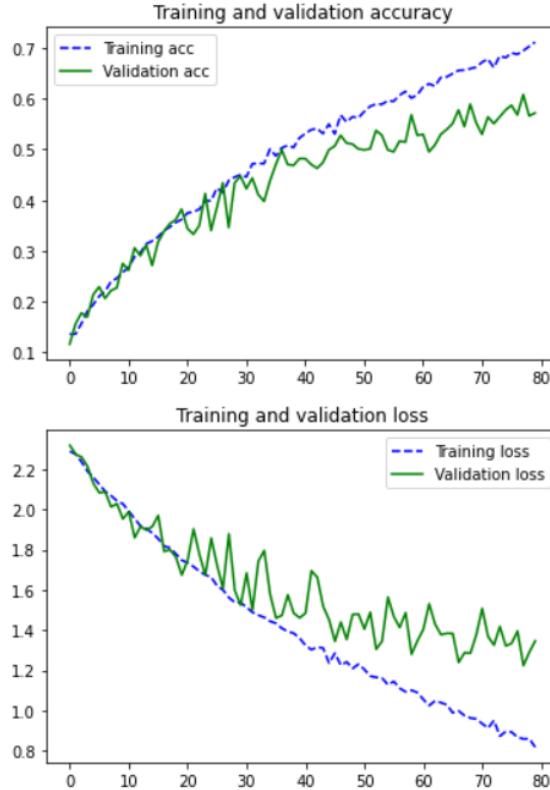


Figure 33: Version 2 : Analyse des résultats de la mise en oeuvre 3

	precision	recall	f1-score	support
Commercial/Airbus_A380	0.33	0.04	0.07	50
Commercial/Airbus_Belluga	0.80	0.10	0.17	42
Commercial/Boeing_747	0.34	0.51	0.41	45
Commercial/Boeing_777	0.27	0.73	0.39	45
Commercial/Falcon_8x	0.41	0.62	0.49	45
Military/A400M	0.85	0.65	0.74	63
Military/Eurofighter_Typhoon	0.46	0.48	0.47	66
Military/Rafale	0.17	0.17	0.17	60
Military/F-35	0.19	0.15	0.17	66
Military/Su-57	0.56	0.34	0.42	44
micro avg	0.38	0.38	0.38	526
macro avg	0.44	0.38	0.35	526
weighted avg	0.43	0.38	0.35	526
samples avg	0.38	0.38	0.38	526

Figure 34: Version 2 : les métriques du modèle en mise en oeuvre 3

```
[[ 2  0 14 25  6  2  1  0  0  0]
 [ 0  4 14 16  3  1  2  1  0  1]
 [ 2  0 23 17  1  0  0  1  0  1]
 [ 0  0  5 33  3  0  1  1  2  0]
 [ 0  0  3  9 28  0  2  2  1  0]
 [ 0  1  1  9  4 41  2  1  4  0]
 [ 1  0  2  2  6  0 32  5 14  4]
 [ 1  0  2  1  8  1 20 10 15  2]
 [ 0  0  2  8  5  3  6 28 10  4]
 [ 0  0  1  3  5  0  3  9  8 15]]
```

Confusion matrix, without normalization

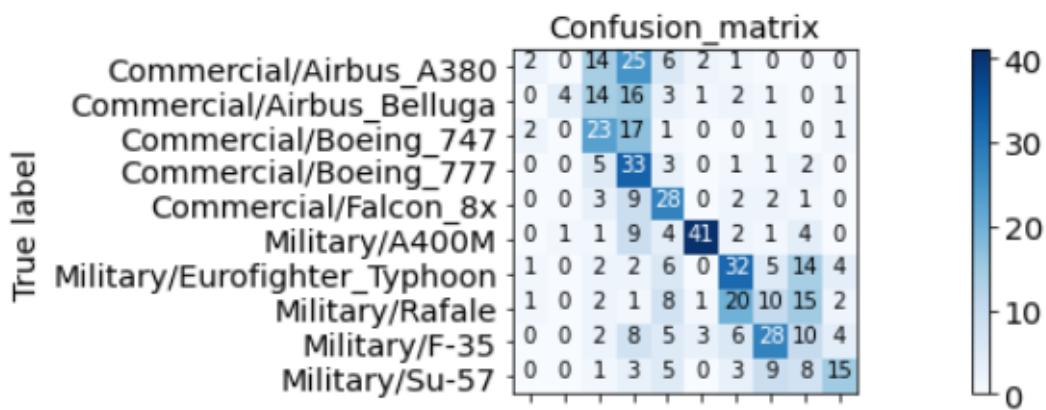


Figure 35: Version 2 : Matrice de confusion de la mise en oeuvre 3

Les performances du réseau de neurones convolutif de base ne sont pas très satisfaisantes. En effet, les précisions des classes sont faibles à part celles de Airbus A400M, Airbus Belluga et Su-57. Ce qui était attendu car ces derniers se distinguent par leurs formes particulières. La matrice de confusion présente une forte corrélation entre quelques classes, par exemple, Airbus A380 est souvent mal reconnu et pris pour Boeing 777. Par conséquent, le modèle est faible et nous devons envisager d'autres solutions.

Remarque : A noter ici dans cette mise en oeuvre que nous avons utilisé deux couches FC de dimension 512 à la place de 128 mais le réseau entraîné n'évoluait presque pas du fait que les poids stagnent.

Il n'existe pas une grande amélioration par rapport à la mise en oeuvre précédente même après augmentation de la capacité du réseau. Nous décidons alors de changer l'architecture du réseau en vue d'obtenir de meilleurs résultats.

5.2.4 Mise en oeuvre 4:

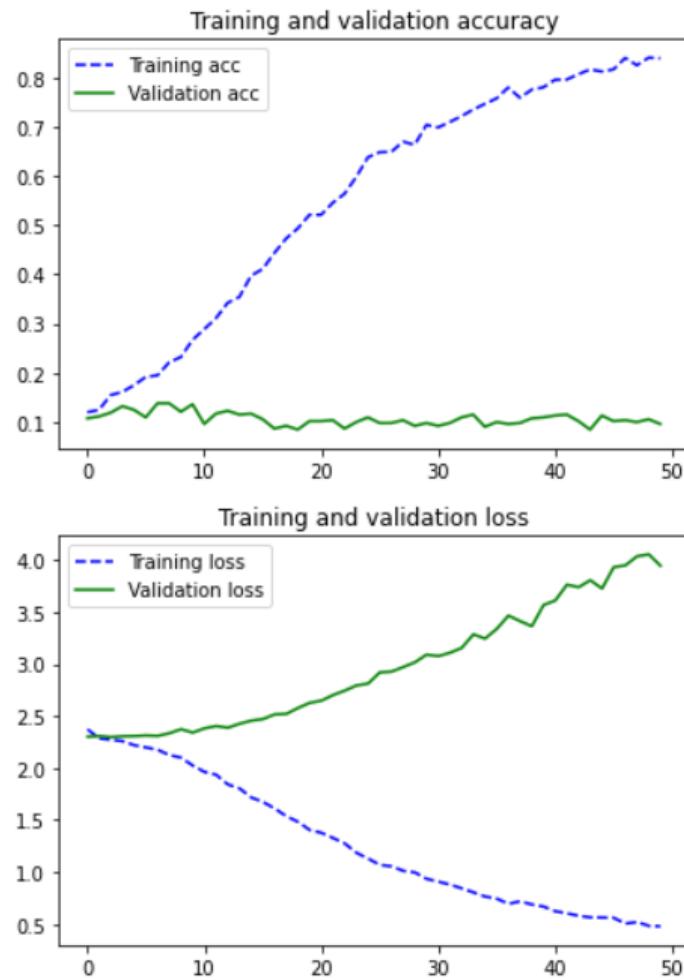


Figure 36: Version 2 : Analyse des résultats de la mise en oeuvre 4

Nous observons un sur-apprentissage, un écart de grand échelle entre le train de modèles et la précision de la validation après la dixième Epoch. La première solution consiste en une augmentation des données. En effet, nous pourrons calculer les caractéristiques de VGG sur les données augmentées, et ainsi classifier ces caractéristiques.

5.2.5 Mise en oeuvre 5:

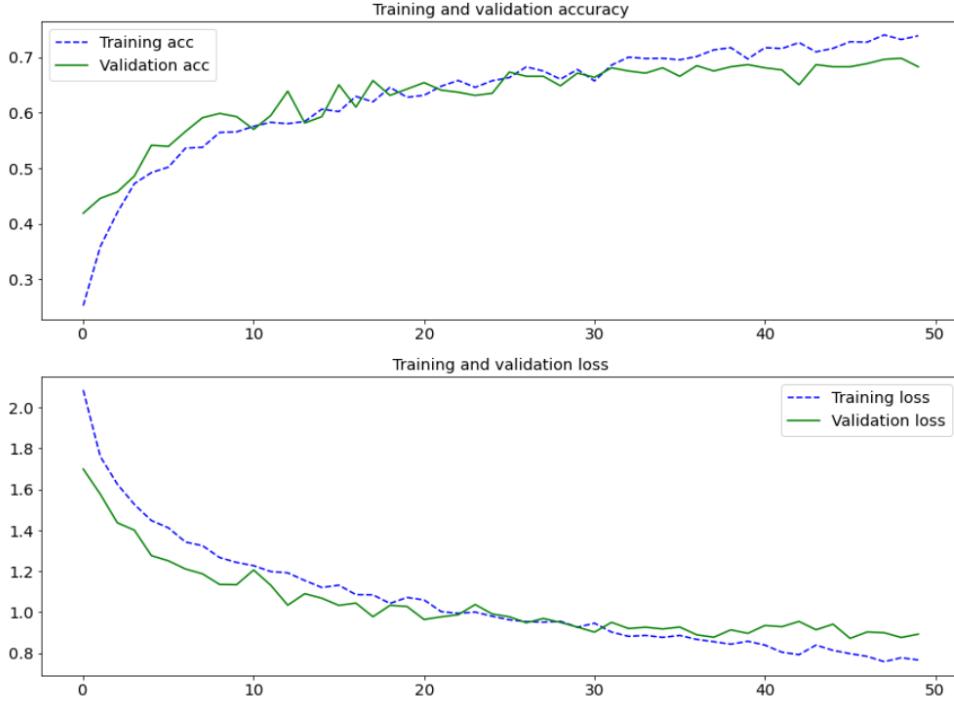


Figure 37: Version 2 : Analyse des résultats de la mise en oeuvre 5

```

Epoch 45/50
158/158 [=====] - 77s 490ms/step - loss: 0.8439 - acc: 0.7082 - val_loss: 0.9419 - val_acc: 0.6826
Epoch 46/50
158/158 [=====] - 78s 498ms/step - loss: 0.7898 - acc: 0.7206 - val_loss: 0.8724 - val_acc: 0.6826
Epoch 47/50
158/158 [=====] - 79s 501ms/step - loss: 0.7286 - acc: 0.7456 - val_loss: 0.9038 - val_acc: 0.6883
Epoch 48/50
158/158 [=====] - 78s 496ms/step - loss: 0.7400 - acc: 0.7398 - val_loss: 0.9001 - val_acc: 0.6960
Epoch 49/50
158/158 [=====] - 78s 496ms/step - loss: 0.7553 - acc: 0.7488 - val_loss: 0.8768 - val_acc: 0.6979
Epoch 50/50
158/158 [=====] - 77s 491ms/step - loss: 0.7325 - acc: 0.7529 - val_loss: 0.8932 - val_acc: 0.6826

```

Figure 38: Version 2 : Derniers 5 Epochs de la mise en oeuvre 5

Nous pouvons voir une grande amélioration par rapport aux mises en oeuvres précédentes, notamment par rapport à notre réseau de neurones convolutif de la mise en oeuvre 3. En effet, celui-ci à une précision de validation de l'ordre de 70% et une validation loss de l'ordre de 0.85 contre une validation loss de l'ordre de 1.4 par rapport au réseau de neurones de base.

Cela montre que l'apprentissage par transfert peut être utile. Bien que le modèle semble être en sous-apprentissage sur les données d'entraînement, nous obtenons toujours une grande précision de validation.

5.2.6 Mise en oeuvre 6:

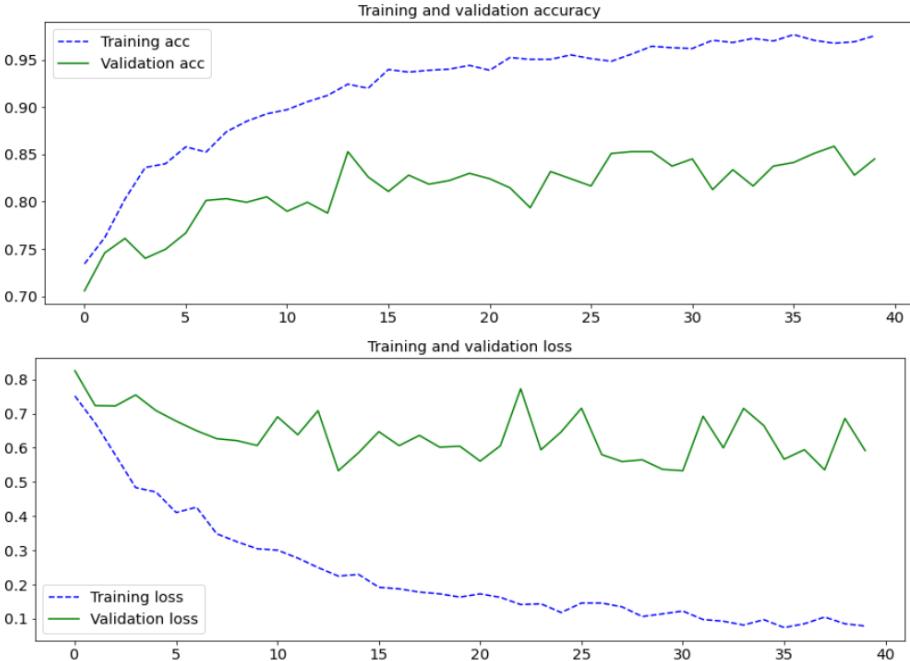


Figure 39: Version 2 : Analyse des résultats de la mise en oeuvre 6

```

Epoch 35/40
158/158 [=====] - 87s 554ms/step - loss: 0.1256 - acc: 0.9635 - val_loss: 0.6644 - val_acc: 0.8375
Epoch 36/40
158/158 [=====] - 87s 553ms/step - loss: 0.0561 - acc: 0.9801 - val_loss: 0.5662 - val_acc: 0.8413
Epoch 37/40
158/158 [=====] - 88s 554ms/step - loss: 0.0683 - acc: 0.9756 - val_loss: 0.5940 - val_acc: 0.8509
Epoch 38/40
158/158 [=====] - 87s 548ms/step - loss: 0.1070 - acc: 0.9690 - val_loss: 0.5349 - val_acc: 0.8585
Epoch 39/40
158/158 [=====] - 86s 545ms/step - loss: 0.0793 - acc: 0.9745 - val_loss: 0.6855 - val_acc: 0.8279
Epoch 40/40
158/158 [=====] - 87s 551ms/step - loss: 0.0719 - acc: 0.9781 - val_loss: 0.5914 - val_acc: 0.8451

```

Figure 40: Version 2 : Derniers 5 Epochs de la mise en oeuvre 6

Vu la difficulté que pose notre base de données en termes de similarité entre les classes, ce résultat est le meilleur qu'on a pu avoir en version 2 . Nous observons une précision de validation de l'ordre de 85% et une validation loss de l'ordre de 0.50 aussi qu'une précision d'entraînement de l'ordre de 98% et un erreur d'entraînement de l'ordre de 0.07. Bien que le modèle présente un sur-apprentissage, il constitue le meilleur résultat.

	precision	recall	f1-score	support
Commercial/Airbus_A380	0.57	0.52	0.54	50
Commercial/Airbus_Belluga	0.85	0.83	0.84	42
Commercial/Boeing_747	0.84	0.58	0.68	45
Commercial/Boeing_777	0.45	0.82	0.58	45
Commercial/Falcon_8x	0.53	0.93	0.67	45
Military/A400M	0.87	0.75	0.80	63
Military/Eurofighter_Typhoon	0.87	0.20	0.32	66
Military/Rafale	0.06	0.03	0.04	60
Military/F-35	0.06	0.08	0.07	66
Military/Su-57	0.49	0.77	0.60	44
micro avg	0.51	0.51	0.51	526
macro avg	0.56	0.55	0.52	526
weighted avg	0.55	0.51	0.49	526
samples avg	0.51	0.51	0.51	526

Figure 41: Version 2 : les métriques du modèle en mise en oeuvre 6

```
[[26 1 4 15 3 1 0 0 0 0]
 [ 0 35 0 4 1 0 0 1 0 1]
 [ 7 0 26 9 1 0 1 0 0 1]
 [ 3 1 1 37 2 0 0 0 0 1]
 [ 0 0 0 3 42 0 0 0 0 0]
 [ 3 2 0 3 5 47 1 0 2 0]
 [ 1 1 0 3 10 1 13 1 27 9]
 [ 1 0 0 1 5 3 0 2 43 5]
 [ 3 0 0 5 6 2 0 27 5 18]
 [ 2 1 0 2 5 0 0 0 0 34]]
```

Confusion matrix, without normalization

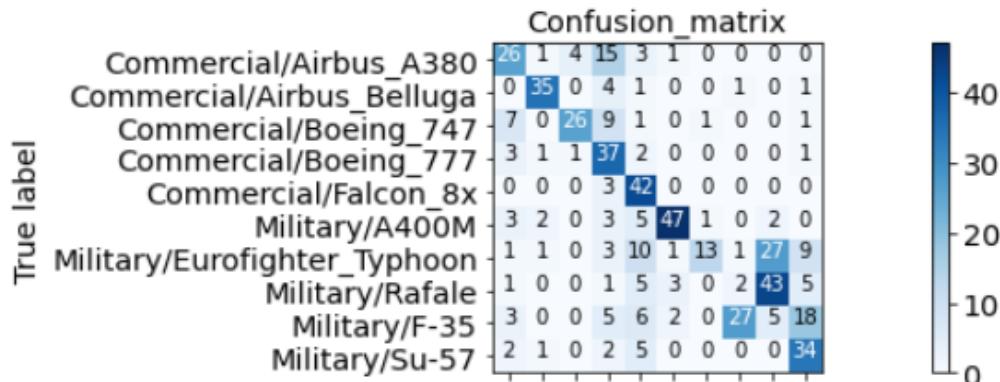


Figure 42: Version 2 : Matrice de confusion de la mise en oeuvre 6

Les métriques du modèle présentent des performances satisfaisantes vu que les précisions se sont améliorées largement et pour toutes les classes par rapport à la mise en oeuvre 3 (celle avec le réseau de neurones convolutif de base). La diagonale de la matrice de confusion est à diagonale strictement dominante, ce qui est très rassurant. Cela prouve l'utilité du transfert learning dans l'amélioration des performances du réseau.

6 Conclusion

Ce projet était l'occasion de manipuler concrètement des réseaux de neurones et de comprendre leur utilité dans le domaine de la classification de plus, nous avons aussi compris la difficulté d'améliorer les performances de celui-ci lorsque la prédiction est une tâche difficile ce qui est le cas dans la version 2 de notre projet. Nous avons employé plusieurs techniques pour diminuer l'overfitting et l'underfitting. Nous avons aussi réussi à optimiser les performances, cependant des améliorations sont possibles notamment au niveau de la base de données car certains modèles d'avion tel que F-35 et Rafale ont été assez souvent confondus d'après la matrice de confusion et cela est à priori dû à leur forte ressemblance. Nous envisageons également pour améliorer les performances du réseau d'essayer d'autres réseaux pré-entraînés qui seront plus complexes tel que Inception pour mieux s'adapter à notre base de données.