CARANGEOT Hugo

CONTE-DEVOLX Titouan

E5DSIA

Rapport de bord du Projet de Classification d'images par IA

Thème : véhicules (vélos, voitures, avions, camions, bateaux, trains, tracteurs)

Lien Google drive avec images et codes

Aperçu rapide des résultats obtenus :

Tableau des résultats* à la précision sur l'ensemble de test pour les différentes méthodes utilisées.

Méthode	Réseau FC	Réseau CNN	Transfer learning
Temps d'entrainement	30 min	1h 30min	30 min
Test accuracy	*40.4%*	87.5%	***98.7%***

**Il faut prendre en compte que ces résultats sont dû à une part d'aléatoire et qu'ils auraient pu être plus ou moins bons puisqu'ils suivent une loi de distribution normale. Cependant, les valeurs ne sont pas aberrantes donc nous les avons conservées.*

Intéressons-nous maintenant à comment nous avons obtenu ces résultats et quel a été notre parcours.

I – Base de données

Le thème que nous avons choisi, mentionné ci-dessus, a été sélectionné de façon arbitraire. Il a été jugé suffisamment simple pour en permettre une approche claire, tout en offrant une certaine complexité pour ne pas être trop élémentaire.

Thème : véhicules (vélos, voitures, avions, camions, bateaux, trains, tracteurs), **7 classes**

Nous avons récupéré les images sur le site flickr.com à l'aide de l'extension chrome « **Imageye** ».

Pour **chaque classe**, nous avons récupéré entre **550** et **800 images**. Nous avons sélectionné volontairement un petit nombre d'images pour **raccourcir le temps d'exécution** même si cela baissera la fiabilité de nos prédictions. De plus, il n'est pas dérangeant que nos classes ne soient pas totalement équilibrées en nombre d'images puisque nous ne cherchions pas à faire un jeu parfait pour « gonfler » nos résultats. Par ailleurs, **une vérification manuelle des images** a été effectuée, mais nous avons conservé toutes les images que nous (humains) considérions comme identifiables. Ainsi, dans la catégorie avion, on a parfois des photos n'ayant que les ailes. A l'inverse, une photo de la catégorie vélo avec une voiture en arrière-plan est supprimée car on ne peut déterminer correctement le véhicule.

Puisque ce sont des images récoltées sur le WEB, il faut les redimensionner pour pouvoir entrainer nos modèles dessus, nous avons donc fait du **pre-processing**. Nous avons choisi la dimension de **224x224x3** (couleurs) puisque c'est un bon compromis entre **qualité et rapidité**. Bien que nous ayons des images noires et blanches dans le lot, nous avons décidé de garder la couleur car cela peut apporter des informations supplémentaires pour distinguer nos véhicules (mais cela n'est pas suffisant, e.g. même si les tracteurs sont souvent rouges ou verts, voir un véhicule de cette couleur ne signifie pas que c'est un tracteur).

Nous avons profité de ce pre-processing pour faire de la **data augmentation** grâce à **tensorflow/keras (python)**. Ainsi des **zooms**, **décalages**, **rotations**, etc, ont été faits pour **doubler** notre quantité d'**images par classe**.

Par la suite, nous avons **compressé** notre **dossier data_processed** avec ses **7 sous-dossiers** (un par classe) contenants les images respectives pour les déposer sur **Google Drive**.

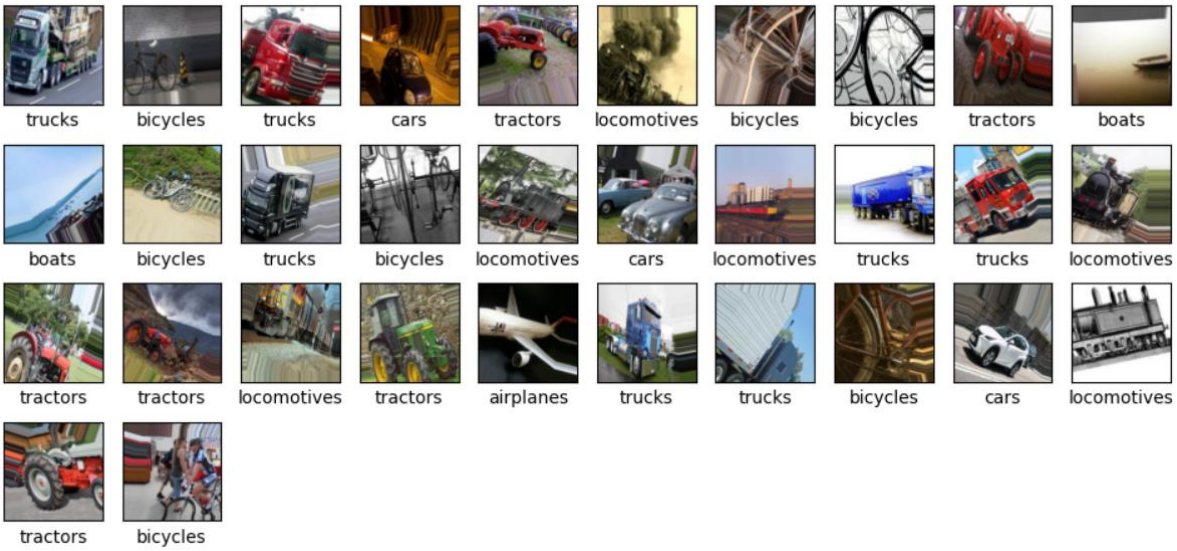
Enfin, nous avons scindés nos sets de données en **train (70%)**, **validation (20%)**, **test (10%)** de manière **aléatoire** et les avons enregistrés en sous-fichiers. Nous n'avons pas chargé en mémoire toutes les images parce que trop nombreuses, elles faisaient planter la RAM de Colab. Le train, sert à entrainer nos modèles, le validation à vérifier si le modèle fonctionne et enfin lorsqu'on a fini d'entraîner nos modèles, on les fait prédire sur test pour voir comment ils se comportent devant de nouvelles données. Cela permet d'**avoir** des valeurs concrètes des **performances** de nos **modèles** et de s'assurer qu'on **over/under fit** pas.

Capture d'écran de la phase de séparation en Set

```
Entraînement : 5939 images, Validation : 1699 images, Test : 851 images
```

Ci-dessous, un extrait aléatoire du set d'entrainement :

Capture d'écran d'une partie aléatoire du set d'entrainement



II – Classification d'images par IA

Pour faciliter les choses, partons du principe que les classes sont **équilibrées**. Étant donné qu'il y a **7 classes**, la probabilité de trouver la bonne classe pour une image donnée sans aucune connaissance est de **100/7 ~ 14.29%**. Nos algorithmes devront donc faire mieux pour dire qu'ils fonctionnent.

De plus, nous avons fait 3 notebooks distincts pour les 3 méthodes, afin qu'elles soient isolées.

1 – Réseau FC (Fully Connected)

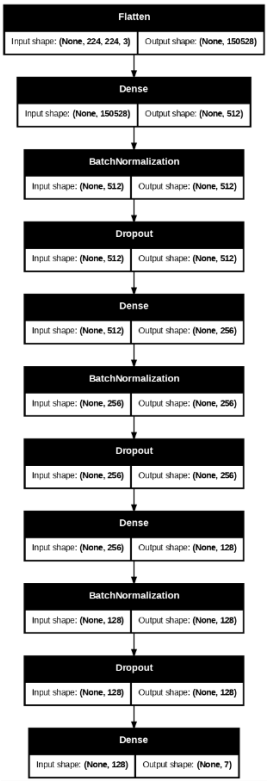
Pour **évaluer** un peu la **complexité** de la tâche qui nous attendait, on **a commencé** par un **réseau FC très simple** avec seulement :

- L'entrée (img en 224x224x3)
- Une couche Flatten (nécessaire pour le FC)
- Une couche de 64 neurones avec la fonction relu en activation
- Une couche de 7 (car 7 classes) neurones avec softmax pour la classification multi-classes

Avec ce réseau nous tournions dans les 14% d'accuracy soit comme l'aléatoire.

Il a donc fallu **complexifier** notre **modèle** et ajouter des techniques pour qu'il progresse.

Schéma de notre second modèle



On voit ici **3 couches** se succéder avec **512, 256, 128 neurones**.
Nous avons augmenté le nombre de neurones sur la première couche pour capturer des relations complexes sur nos images.

Puis nous avons **256**, **128 neurones** pour créer un **réseau de neurones profond** (pour apprendre des choses de plus en plus complexes).

Et les couches supplémentaires de **BatchNormalization** et **Dropout** sont présentes pour :

- Normaliser les activations à chaque étape afin de maintenir la **stabilité** de l'entraînement et **accélérer** le processus d'apprentissage. (batch)
- **Réduire** le risque de **surapprentissage** en régularisant le modèle. (dropout)

Enfin, pour permettre à notre modèle d'apprendre, nous avons avons fixé **100 epochs** avec des **batchs de 32** (il y en a donc 186 par epoch). Cependant, le modèle peut ne plus progresser ou overfit avant les 100 epochs.
C'est pour cela que nous avons utilisé la méthode **EarlyStopping** de **keras**.

```
[67] from tensorflow.keras.callbacks import EarlyStopping

     # Création du callback EarlyStopping
     early_stopping = EarlyStopping(monitor='val_accuracy',
                                    patience=5,
                                    restore_best_weights=True)
```

On lui demande de vérifier **val_accuracy** (précision du modèle sur les données de validation) à la fin de **chaque epoch**. Si au bout de **5 epochs**, la **valeur** ne s'est **pas améliorée**, alors on **restaure** les meilleurs **paramètres** du **modèle** et on arrête l'apprentissage.

*Pourquoi avons-nous choisi **val_accuracy** et non une autre valeur ?*

Nous nous intéressons ici à la capacité du modèle de prédire sur des données sur lesquelles il ne s'est pas entrainé. De plus, ce paramètre permet de tracer l'overfitting et d'interrompre l'entrainement si jamais le modèle commence à trop apprendre.

Capture d'écran du history

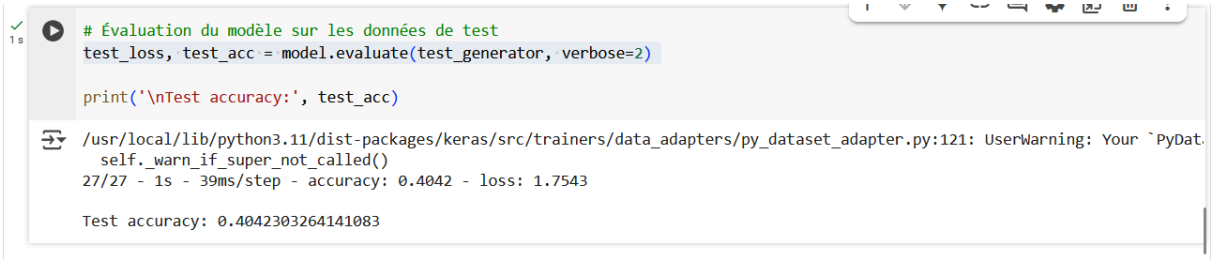
```
Epoch 15/100
186/186 ━━━━━━━━━━━━━━━━━━━━ 95s 513ms/step - accuracy: 0.3411 - loss: 1.8124 - val_accuracy: 0.3773 - val_loss: 1.7778
Epoch 16/100
186/186 ━━━━━━━━━━━━━━━━━━━━ 94s 504ms/step - accuracy: 0.3418 - loss: 1.8104 - val_accuracy: 0.3337 - val_loss: 1.8194
Epoch 17/100
186/186 ━━━━━━━━━━━━━━━━━━━━ 94s 509ms/step - accuracy: 0.3347 - loss: 1.8188 - val_accuracy: 0.3461 - val_loss: 1.8007
Epoch 18/100
186/186 ━━━━━━━━━━━━━━━━━━━━ 93s 501ms/step - accuracy: 0.3500 - loss: 1.8025 - val_accuracy: 0.3214 - val_loss: 1.8324
Epoch 19/100
186/186 ━━━━━━━━━━━━━━━━━━━━ 96s 515ms/step - accuracy: 0.3347 - loss: 1.8107 - val_accuracy: 0.3590 - val_loss: 1.7920
Epoch 20/100
186/186 ━━━━━━━━━━━━━━━━━━━━ 93s 501ms/step - accuracy: 0.3314 - loss: 1.8187 - val_accuracy: 0.3461 - val_loss: 1.8002
```

On s'aperçoit que le modèle s'est arrêté à l'epoch 20 (30min d'entrainement), après 5 epochs sans progresser sur val_accuracy,ce qui confirme que notre early stopping était une bonne idée.

Malheureusement, nous n'avons pas tracé la courbe d'apprentissage pour le réseau FC (elle y est pour les autres modèles). Le modèle a extrêmement progressé sur les 3 premiers tours, passant de 20% à 30% d'accuracy puis a mis 17 tours à converger.

Une fois le modèle entrainé, nous allons vérifier qu'il fonctionne correctement sur le set de test.

Capture d'écran de la précision des prédictions sur le set test



```
# Évaluation du modèle sur les données de test
test_loss, test_acc = model.evaluate(test_generator, verbose=2)

print('\nTest accuracy:', test_acc)
```

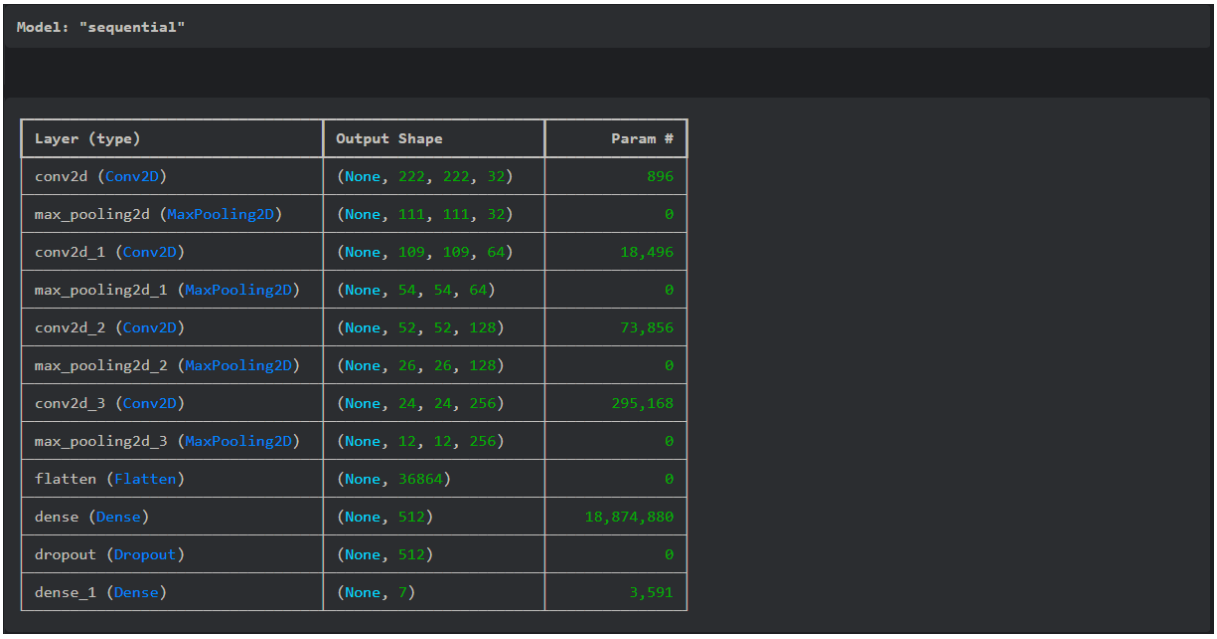
```
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDat
  self._warn_if_super_not_called()
27/27 - 1s - 39ms/step - accuracy: 0.4042 - loss: 1.7543

Test accuracy: 0.4042303264141083
```

Comme vu sur la page de garde, nous obtenons **40.4% d'accuracy** à ce test, nous sommes assez contents du résultat puisque nous pensions avoir moins (dans les 25%). Il faut tout de même prendre en compte que nos sets sont formés de manière aléatoire et ainsi, les 40% d'accuracy auraient pu être 30% ou 50% (en poussant dans l'extrême). Cela suit une **loi normale de distribution** et nous n'avons pas lancé notre code des centaines de fois pour tracer cette loi car cela aurait été trop long. Cependant, nous ne considérons pas notre résultat comme aberrant.

2 – Réseau CNN

Pour ce **réseau**, on a directement fait un **modèle complet et complexe**. On s'est inspiré du modèle de **VGG** puisqu'il est reconnu pour son **efficacité**.

Capture d'écran du model.summary()



Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 128)	0
conv2d_3 (Conv2D)	(None, 24, 24, 256)	295,168
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 256)	0
flatten (Flatten)	(None, 36864)	0
dense (Dense)	(None, 512)	18,874,880
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 7)	3,591

Quelques petites explications :

- Dropout à la même fonction que pour le réseau FC

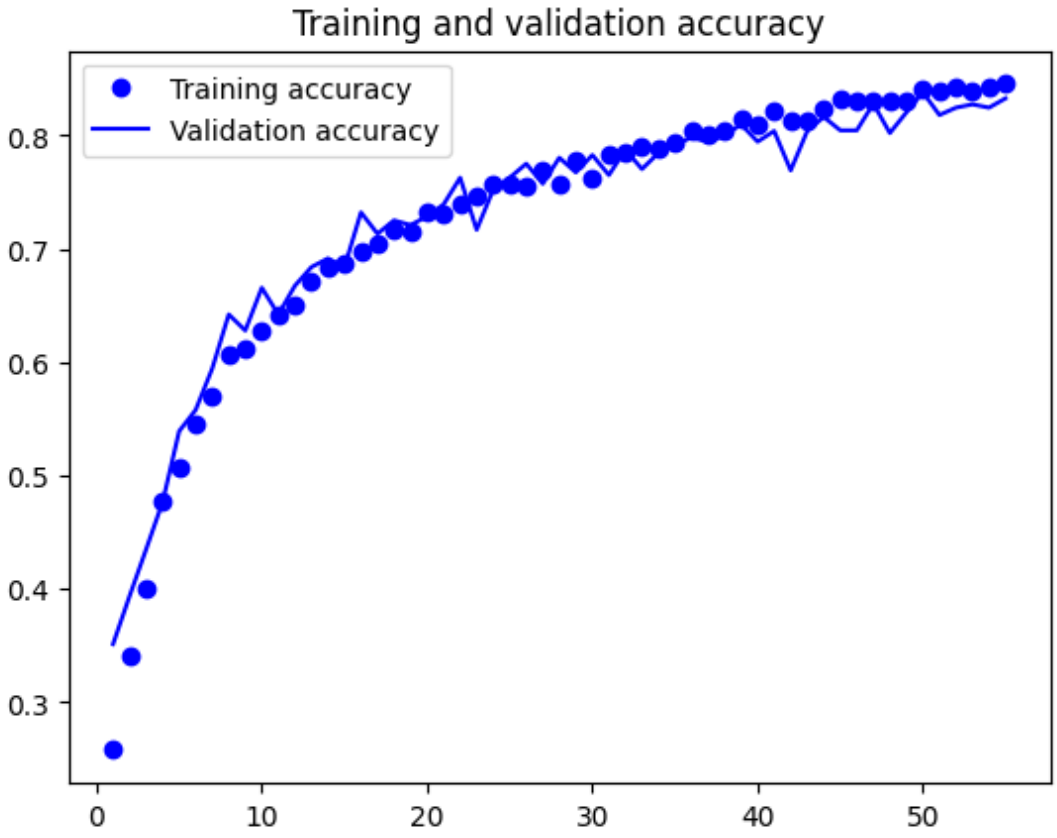
- Convolution et pooling permettent de "voir" les caractéristiques importantes de l'image tout en réduisant sa taille pour accélérer l'apprentissage.

De même, l'utilisation des différentes couches nous donne **un réseau de neurones profond**. Ce n'est pas la même manière d'apprendre ici, elle est beaucoup **plus coûteuse en temps/énergie/ressources**.

Nous avons répété le **earlyStopping** etc de la couche FC, ce qui change ici c'est le modèle en lui-même et cela sera également le cas pour le transfer learning.

Passons donc à **l'entrainement et aux résultats** (voir ci-dessous).

Courbe d'apprentissage du réseau CNN.



Il a fallu **1h36min et 55 epochs** pour que le **réseau converge**.

On peut voir qu'il **progresse très vite** au début **(< 10 epochs**) puis progresse presque de manière **linéaire** jusqu'à **converger**. On s'aperçoit d'ailleurs que notre modèle s'est arrêté puisque la validation accuracy ne progressait plus depuis 5 tours bien que le training accuracy continuait de progresser, ce modèle commençait à overfit. Cela confirme que notre **earlystopping est judicieux**.

Enfin testons notre précision sur le set de test

Capture d'écran de la précision des prédictions sur le set test

```
[ ] # Évaluation du modèle sur les données de test
    test_loss, test_acc = model.evaluate(test_generator, verbose=2)

    print('\nTest accuracy:', test_acc)

    27/27 - 2s - 67ms/step - accuracy: 0.8754 - loss: 0.3727

    Test accuracy: 0.8754406571388245
```

Nous obtenons **87.5% d'accuracy**, ce qui est **excellent** pour notre **réseau CNN**.

3 – Transfer Learning

Pour cette partie, nous n'avons pas grand-chose à dire, le **code commenté** sera plus parlant :

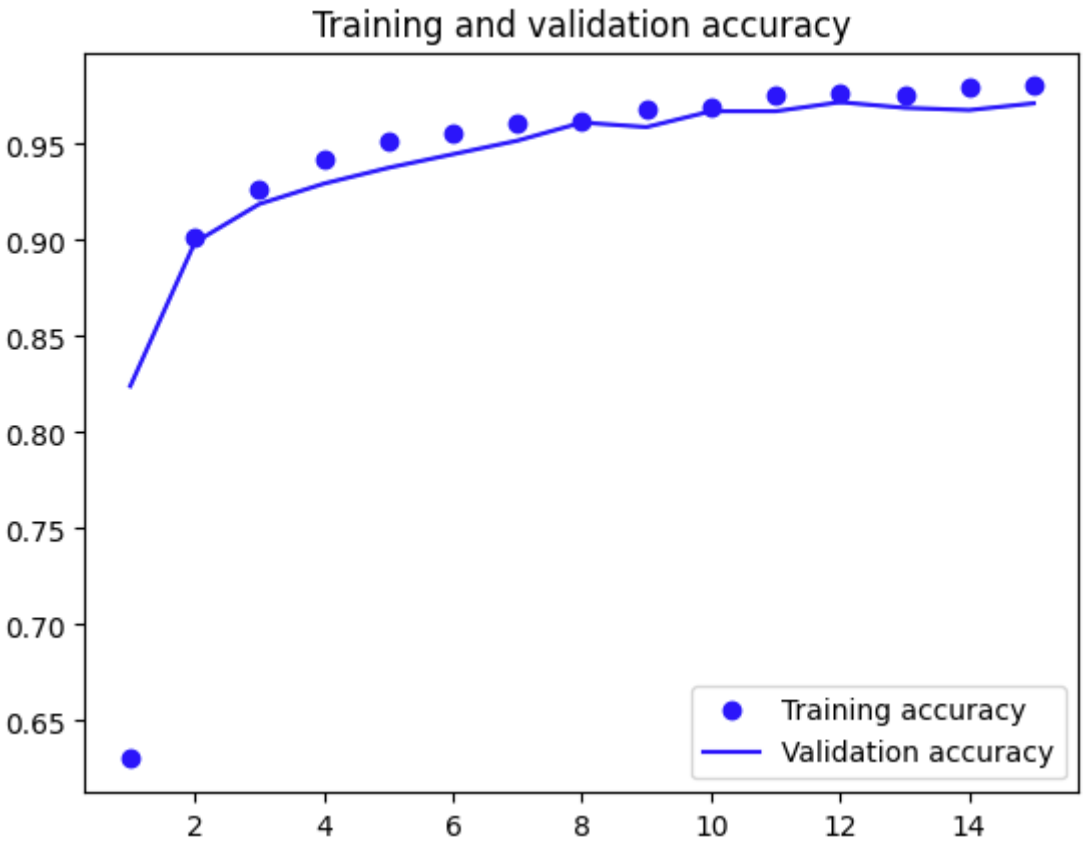
Extrait de code du fine-tuning de mobilenetv2

```
# Charger le modèle MobileNetV2 pré-entraîné sur ImageNet
conv_base = MobileNetV2(weights='imagenet', include_top=False,input_shape=(224, 224, 3))

# Réutilisation des poids des couches CONV pour conserver les features donc les layers ne se réentraine
pas
for layer in conv_base.layers:
    layer.trainable = False

# # Réinitialisation des couches FC pour ajouter un classificateur adapté à notre tâche (7 classes)
x = conv_base.output
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(1024, activation='relu')(x)
predictions = keras.layers.Dense(7, activation='softmax')(x) #classification en 7 classes

# Créer le modèle final
model = Model(inputs=conv_base.input, outputs=predictions)

# Compiler le modèle
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Fine-tuning : "dégel" des couches convolutionnelles pour les réentrainer avec les nouvelles couches
ajoutées
for layer in conv_base.layers:
    layer.trainable = True
```

Ce qu'il faut retenir c'est que nous avons **adapté un modèle très puissant** à **notre problème** en lui **modifiant des couches** pour la classification en **7 catégories**.

Passons directement à la phase d'entrainement. Cette fois, on a mis une **patience de 3** (et non 5) à **earlystopping** car le modèle prenait trop de temps à converger et notre temps de session colab n'était plus suffisant. Voici le résultat de l'entrainement :

Courbe d'apprentissage du réseau CNN.



On s'aperçoit que le modèle a bien **convergé** (**en 30min**) et a **très vite progressé** avant de se **perfectionner** en une **dizaine d'epochs**.

Capture d'écran de la précision des prédictions sur le set test

```
[ ]  #Évaluation du modèle sur les données de test
     test_loss, test_acc = model.evaluate(test_generator, verbose=2)

     print('\nTest accuracy:', test_acc)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py
  self._warn_if_super_not_called()
27/27 - 3s - 103ms/step - accuracy: 0.9871 - loss: 0.0390

Test accuracy: 0.9870740175247192
```

Nous obtenons **98.7% d'accuracy**, c'est un **excellent résultat** et nous sommes contents de notre travail.

III – Conclusion

Tableau des résultats* à la précision sur l'ensemble de test pour les différentes méthodes utilisées.

Méthode	Réseau FC	Réseau CNN	Transfer learning
Temps d'entrainement	30 min	1h 30min	30 min
Test accuracy	*40.4%*	87.5%	*98.7%*

**Il faut prendre en compte que ces résultats sont dû à une part d'aléatoire et qu'ils auraient pu être plus ou moins bons puisqu'ils suivent une loi de distribution normale. Cependant, les valeurs ne sont pas aberrantes donc nous les avons conservées.*

Comme attendu, nos différentes **étapes** sont de **plus en plus performantes**.

Le réseau **FC** est **peu convaincant**, bien que 'fait maison', il nous a pris autant de **temps** à **entrainer** que le **temps** qu'il nous a fallu pour **modifier** le **transfer learning**, or, les **résultats** sont bien **faibles** comparé aux **98.7%**.

Le réseau **CNN** est **assez convaincant** vu le **ratio résultat/ressources** (il n'y a eu qu'une heure et demie d'entrainement sur google colab) et que c'est un modèle 'fait maison'.

Le réseau **Transfer Learning** est excellent et rapide a re-entrainé partiellement, il est sans aucun doute **le meilleur**.

Avec **plus d'images** et de **temps d'entrainement** on sait que l'on aurait pu **avoir** des **résultats** encore **meilleurs** que ceux qu'on a obtenu.

Nous n'avons pas eu trop de mal à mettre en place nos modèles car les cours et nos recherches complémentaires sur des forums nous ont permis de bien comprendre le sujet.

Par ailleurs, il serait grandement intéressant **d'accéder** à **colab pro** (un peu comme à la suite de JETBRAINS) avec le mail étudiant de l'ESIEE. On ne sait pas si cela est possible mais on apprécierait grandement pour accéder aux GPU encore plus puissants et sans limite de temps. Cela a été **une excellente découverte** pour nous.

Merci pour votre lecture, **Titouan** et **Hugo**.