TP Introduction au Deep Learning: Application à la vision par ordinateur

CAIPA PRIETO Julian Andres

École Nationale Supérieure des Arts et Métiers

Computer Vision

Enseignant : Fakhreddine Ababsa

PA10

Campus de Paris, France

2024

Table des matières

Table des figures

Table des codes

Introduction

Dans ce travail pratique, nous allons explorer deux stratégies pour l'entraînement d'un modèle de classification d'images avec un ensemble de données d'apprentissage limité, une situation courante dans les problèmes de vision par ordinateur. La première approche consiste à entraîner un modèle à partir de zéro en utilisant des techniques d'augmentation de données. La seconde approche repose sur l'extraction de caractéristiques à partir d'un réseau neuronal pré-entraîné, en particulier le réseau VGG16, pour améliorer la performance du modèle.

Le but principal de ce travail est de classifier des images de chiens et de chats en utilisant un ensemble de données de 4000 images, dont 2000 images de chaque classe. Ce processus impliquera plusieurs étapes cruciales, telles que le téléchargement et la préparation des données, la construction et l'entraînement de modèles de réseaux de neurones convolutionnels (ConvNet), ainsi que la mise en œuvre de méthodes de réduction de sur-ajustement et d'augmentation des données. Nous comparerons les résultats obtenus avec les deux approches afin de déterminer celle qui donne les meilleures performances.

Objectifs

Dans ce TP, nous examinerons deux stratégies différentes pour résoudre ce problème :

— Entraînement d'un nouveau modèle à partir de zéro avec une augmentation de données.

— Extraction de caractéristiques avec un réseau pré-entraîné.

Contextualisation des réseaux neuronaux et des réseaux convolutionnels

Un réseau de neurones artificiel (ANN) est un modèle d'inspiration biologique, imitant le fonctionnement du cerveau humain à travers des neurones interconnectés. Dans un réseau neuronal typique, il existe trois types de couches (3) :

1. **Couche d'entrée** : C'est la couche où les données brutes, comme les images, sont introduites dans le modèle. Le nombre de neurones dans cette couche correspond au nombre total de caractéristiques dans les données.
2. **Couches cachées** : Chaque neurone dans ces couches effectue une transformation mathématique sur la sortie de la couche précédente, à l'aide de poids et biais, pour en extraire des caractéristiques plus complexes.
3. **Couche de sortie** : Elle applique une fonction d'activation (comme la sigmoïde ou le softmax) pour convertir la sortie en une probabilité ou une classification.

Les réseaux de neurones convolutionnels (CNN) sont une extension des réseaux de neurones artificiels. Ils sont principalement utilisés pour traiter des données structurées sous forme de grilles, comme des images. Leur principale caractéristique réside dans l'utilisation de couches de convolution, où les neurones sont connectés à des sous-régions locales de l'image d'entrée plutôt qu'à l'ensemble de celle-ci. Ces couches permettent d'extraire des caractéristiques locales comme les contours, textures ou formes, puis les couches de pooling réduisent la taille des cartes de caractéristiques pour rendre le modèle plus efficace.

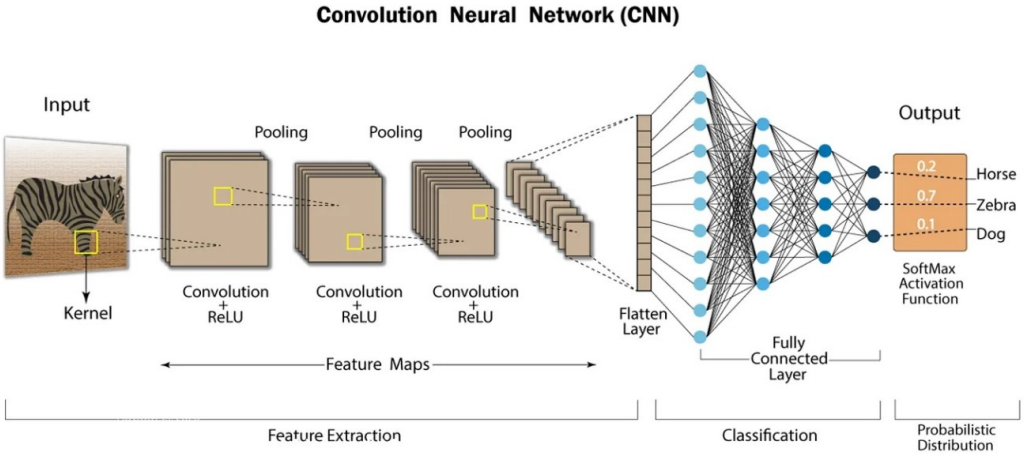

FIGURE 1 – Convolution Neural Network (2).

Dans ce travail, on a utilisé un réseau CNN adapté à un problème de classification d'images (chiens et chats). En plus des couches convolutionnelles, des couches MaxPooling2D ont été ajoutées pour réduire la taille des cartes de caractéristiques sans perdre les informations essentielles, suivies d'une couche Flatten pour préparer les données pour les couches denses et, enfin, une couche de sortie avec activation sigmoïde, nécessaire pour un problème de classification binaire.

Les réseaux CNN sont particulièrement efficaces pour la vision par ordinateur car ils permettent d'extraire automatiquement des caractéristiques des images, simplifiant ainsi le

processus d'entraînement du modèle par rapport à un ANN traditionnel où les caractéristiques doivent être définies manuellement (3).

1 Téléchargement de données

Pour commencer la classification des images de chiens et de chats, le jeu de données a été téléchargé depuis la plateforme Kaggle, à partir du défi "dogs-vs-cats". Ce jeu de données contient 25 000 images de chiens et de chats au format JPEG.

FIGURE 2 – Exemples de l'ensemble de données « dogs-vs-cats ». Les exemples sont hétérogènes en taille, en apparence, etc. (1).

Après le téléchargement, les images ont été décompressées et réorganisées en trois sous-ensembles [1], conformément aux exigences du travail pratique :

- Un ensemble d'entraînement comprenant 1 000 images pour chaque classe (chiens et chats).
- Un ensemble de validation avec 500 images pour chaque classe.
- Un ensemble de test avec 500 images pour chaque classe.

Ce processus a permis d'organiser les données de manière appropriée pour leur utilisation dans l'entraînement du modèle de classification.

1. Bien qu'il soit possible de le faire automatiquement avec du code, nous l'avons fait manuellement pour éviter de compliquer le code.

2 Entraînement d'un ConvNet à partir de zéro

2.1 Architecture du réseau

Tout d'abord, l'architecture du réseau a été conçue, il s'agit d'un réseau de neurones convolutionnel, pour lequel une couche d'entrée, une couche de sortie et plusieurs couches intermédiaires ont été utilisées pour améliorer les performances, en suivant les étapes fournies. Le résumé du réseau est présenté ci-dessous :

Layer (type)	Output Shape	Param #
conv2d_11 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_12 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_12 (Conv2D)	(None, 72, 72, 64)	18,496
max_pooling2d_13 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_13 (Conv2D)	(None, 34, 34, 128)	73,856
max_pooling2d_14 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_14 (Conv2D)	(None, 15, 15, 128)	147,584
max_pooling2d_15 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_3 (Flatten)	(None, 6272)	0
dense_10 (Dense)	(None, 512)	3,211,776
dense_11 (Dense)	(None, 1)	513

FIGURE 3 – Résumé du réseau.

Couche par couche, le rôle de chacune a été analysé. La première couche fonctionne comme une couche d'entrée et est responsable de définir les paramètres des données d'entrée. Dans ce cas, le code a été légèrement modifié pour utiliser la fonction 'Input' de Keras, qui permet de spécifier directement les dimensions des images dans la couche d'entrée, définissant ainsi la ligne de code suivante :

```
1 model.add(Input(shape=(150, 150, 3)))
```

Code 1 – Couche d'entrée.

Les deux couches suivantes sont des couches convolutionnelles. La première est une couche Conv2D, qui applique des filtres (dans ce cas 32 filtres de taille 3x3) pour générer des cartes de caractéristiques des images. Cette couche utilise une activation de type ReLU, qui introduit de la non-linéarité et permet d'apprendre des motifs complexes, tout en convertissant les valeurs négatives en 0, afin d'éviter le problème du "Vanishing Gradient", c'est-à-dire des problèmes liés à la diminution du gradient au cours du processus d'apprentissage.

```
1 model.add(layers.Conv2D(32, (3, 3), activation='relu'))
```

Code 2 – Couche Conv2D initial.

La deuxième couche est une MaxPooling, qui réduit la taille des cartes de caractéristiques de moitié dans chaque dimension (de 150x150 à 75x75), en sélectionnant les valeurs maximales de chaque bloc de 2x2 pixels. Cela vise à réduire la complexité et à rendre le réseau plus efficace, tout en conservant les caractéristiques les plus importantes.

```
model.add(layers.MaxPooling2D(2, 2))
```

Code 3 – Couche MaxPooling initial.

Ensuite, il y a 6 couches supplémentaires, effectuant essentiellement le même processus que les couches Conv2D et MaxPooling, mais en augmentant exponentiellement le nombre de filtres (de base 2, passant de 32 à 64, puis de 64 à 128), ceci afin de détecter davantage de caractéristiques à mesure que le modèle progresse.

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D(2, 2))
```

Code 4 – Couches Conv2D et MaxPooling additionnelles.

Après cela, une couche Flatten est ajoutée, qui prend simplement la sortie tridimensionnelle des couches convolutionnelles et la "aplatie" en un vecteur unidimensionnel. Cela prépare les données à être envoyées vers les couches denses (fully connected), qui traitent la classification.

```
model.add(layers.Flatten())
```

Code 5 – Couche d'aplatissement.

Après l'aplatissement, viennent les couches denses, une de traitement et une de sortie. La première est une couche fully connected avec 512 neurones, chacune étant entièrement connectée à la sortie de la couche précédente (la sortie aplatie des couches convolutionnelles). Une activation est appliquée pour éviter le "vanishing gradient". Tout cela vise à permettre au modèle d'apprendre des combinaisons plus complexes de ces caractéristiques.

```
model.add(layers.Dense(512, activation='relu'))
```

Code 6 – Couche dense de 512 neurones.

Enfin, la couche de sortie est une couche dense avec un seul neurone, en raison de la nature de classification binaire du problème.

```
model.add(layers.Dense(1, activation='relu'))
```

Code 7 – Couche de sortie.

Maintenant, nous avons les lignes de code suivantes, qui compilent le modèle créé afin de le préparer à l'entraînement.

```
model.compile(optimizer=optimizers.RMSprop(learning_rate=1e-4),
loss='binary_crossentropy',
metrics=['acc'])
```

Code 8 – Compilation du réseau.

L'optimiseur RMSprop est utilisé, un classique qui ajuste la taille des pas dans la descente du gradient de manière adaptative. Cela permet d'avoir différentes taux d'apprentissage pour différents paramètres en fonction de la taille du gradient, et c'est pour cette raison que des activations sont ajoutées pour éviter le problème de "vanishing gradient". Le taux d'apprentissage (Learning Rate) contrôle simplement la taille des poids.

Ensuite, la fonction de perte est définie comme étant l'entropie binaire, étant donné que le modèle est binaire. Cela est parfait pour ce type de modèle, car cette fonction mesure la différence entre les probabilités prédites par le modèle et les vraies étiquettes (0 ou 1), avec pour objectif de minimiser cette différence. Enfin, la métrique de "Accuracy" (acc) mesure le pourcentage de prédictions correctes. Comme il s'agit d'un problème de classification binaire, c'est un bon indicateur de performance, car elle compare simplement combien de prédictions étaient correctes par rapport au nombre total de prédictions réalisées.

2.2 Prétraitement des données

Pour préparer les données à l'entraînement du modèle, plusieurs étapes ont été suivies pour les convertir en tenseurs à virgule flottante et les prétraiter correctement avant de les envoyer au réseau. Tout d'abord, les fichiers image ont été lus et convertis du format JPEG en grille de pixels RGB. Ces grilles ont ensuite été transformées en tenseurs numériques, où les valeurs des pixels, initialement comprises entre 0 et 255, ont été normalisées dans l'intervalle [0, 1].

Keras, à travers son module *ImageDataGenerator*, a été utilisé pour automatiser ce processus de prétraitement. Cela a permis de générer des lots d'images prétraitées qui ont ensuite été envoyées au modèle pour l'entraînement et la validation.

Ainsi, avec cela bien compris, nous disposons de trois blocs de code. Le premier bloc normalise les images de sorte que leurs valeurs de pixels soient comprises entre [0,1], ce qui est fondamental pour que le modèle s'entraîne de manière plus stable et plus rapide.

En effet, la plupart des modèles de réseaux neuronaux fonctionnent mieux avec des entrées dans cette plage.

```
train_datagen = ImageDataGenerator(rescale=1. / 255)
test_datagen = ImageDataGenerator(rescale=1. / 255)
```

Code 9 – Normalisation des données.

Le deuxième et le troisième blocs de code génèrent respectivement les données d'entraînement et de validation.

```
train_generator = train_datagen.flow_from_directory(
    'dogs_cats/train',
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')
validation_generator = test_datagen.flow_from_directory(
    'dogs_cats/validation',
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')
```

Code 10 – Génération des données.

On utilise une "Target Size" pour uniformiser la taille des images, quel que soit leur format d'origine. Ensuite, on détermine la taille du lot ou "batch", c'est-à-dire le nombre d'images prises par itération, et enfin, la classification binaire est choisie comme type de classe à prédire.

2.3 Ajustement du modèle

Avec le réseau et les données prêtes, on procède à l'entraînement du réseau avec les données générées, en utilisant le code suivant :

```
history = model.fit(
    train_generator,
    steps_per_epoch=100,
    epochs=10,
    validation_data=validation_generator,
    validation_steps=50)
```

Code 11 – Première entraînement.

En résumé, la fonction ”model.fit()” est celle qui réalise le processus d’entraînement. On lui passe le générateur d’entraînement ainsi que les données de validation (générateur de validation). En plus de cela, on spécifie le nombre d’époques, c’est-à-dire le nombre de fois que le modèle verra l’ensemble complet des données d’entraînement, et les "steps per epoch", qui définissent combien de lots sont analysés à chaque époque. Il est facile de constater que 100 lots de 20 images (”batch size”) donnent les 2000 images d’entraînement. Les "steps" pour la validation fonctionnent de manière similaire : ils indiquent combien de lots de 20 images seront traités, soit 50 dans ce cas, ce qui correspond à 1000 images.

À la fin de l’exécution du code, on obtient le résultat suivant de l’entraînement.

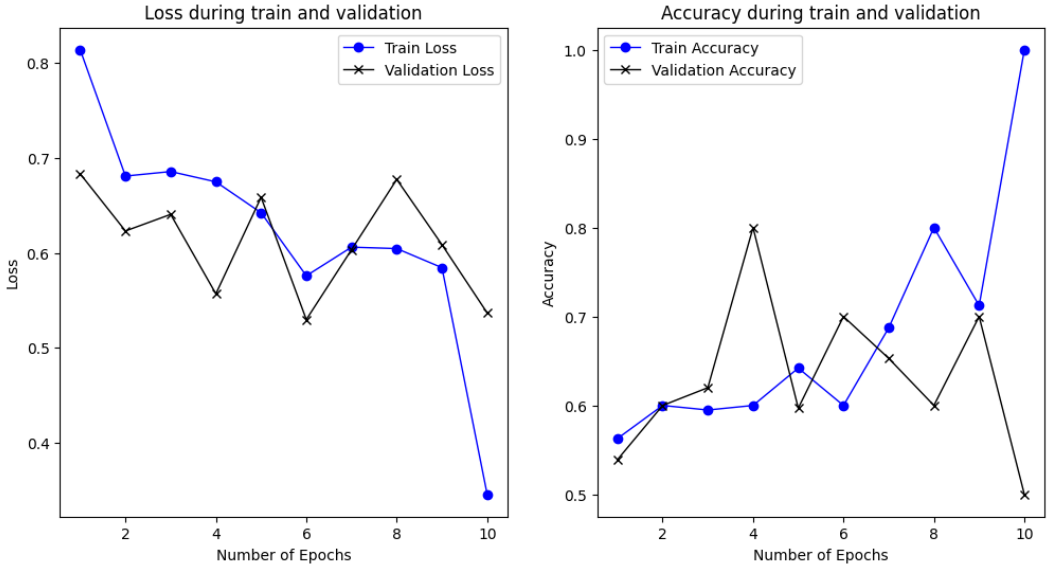

FIGURE 4 – Première entraînement.

D’après les graphiques générés, plusieurs observations peuvent être faites concernant le comportement du modèle en termes de perte (Loss) et de précision (Accuracy) pendant l’entraînement et la validation.

La perte pendant l’entraînement diminue de façon constante au fil des époques, ce qui montre que le modèle apprend efficacement des données d’entraînement. Cependant, la perte pendant la validation est plus erratique. Bien qu’elle diminue initialement, elle fluctue ensuite sans suivre une tendance claire, ce qui pourrait indiquer que le modèle ne généralise pas bien aux données de validation, suggérant un possible surapprentissage (overfitting).

La précision de l’entraînement atteint 100% à la fin, ce qui est un signe évident de surapprentissage. Le modèle semble mémoriser les données d’entraînement sans apprendre des motifs généralisables. En revanche, la précision sur les données de validation fluctue entre 70% et 75% sans amélioration notable, montrant que le modèle n’arrive pas à améliorer ses performances sur des données non vues.

Le modèle montre des signes clairs de surapprentissage. À mesure que la précision sur les données d’entraînement approche les 100%, la précision sur les données de validation ne suit pas la même tendance et fluctue. De plus, la perte sur les données de validation ne

diminue pas de manière cohérente, indiquant que le modèle apprend trop spécifiquement sur l'ensemble d'entraînement sans bien généraliser pour de nouvelles données.

2.3.1 Améliorations proposées

Étant donné que les résultats d'exécution n'ont pas été aussi satisfaisants, quelques améliorations du réseau sont proposées afin de rendre les implémentations plus optimales, rapides et efficaces.

Dernière couche avec activation sigmoïde La fonction d'activation dans la dernière couche d'un réseau neuronal est importante car elle définit le comportement de la sortie du modèle. Par conséquent, l'activation utilisée joue un rôle crucial dans les performances du réseau. Dans ce cas, toutes les couches, y compris la dernière, utilisent une activation ReLU, qui est principalement employée dans les couches cachées car elle aide à résoudre le problème des "vanishing gradients" et améliore les performances dans les réseaux profonds. Cependant, dans le cas d'un réseau binaire, elle n'est pas appropriée pour la couche de sortie, car elle ne restreint pas les sorties à un intervalle spécifique, ce qui pourrait entraîner des valeurs autres que 0 ou 1. Cela pourrait confondre le modèle en termes de classification, étant donné que ReLU supprime uniquement les valeurs négatives en les rendant nulles.

Dans ce cas, l'activation sigmoïde est idéale, car elle génère une sortie binaire avec des valeurs comprises entre 0 et 1, ce qui est plus adapté au problème de classification binaire présenté. Après avoir effectué ce changement, les résultats obtenus sont les suivants :

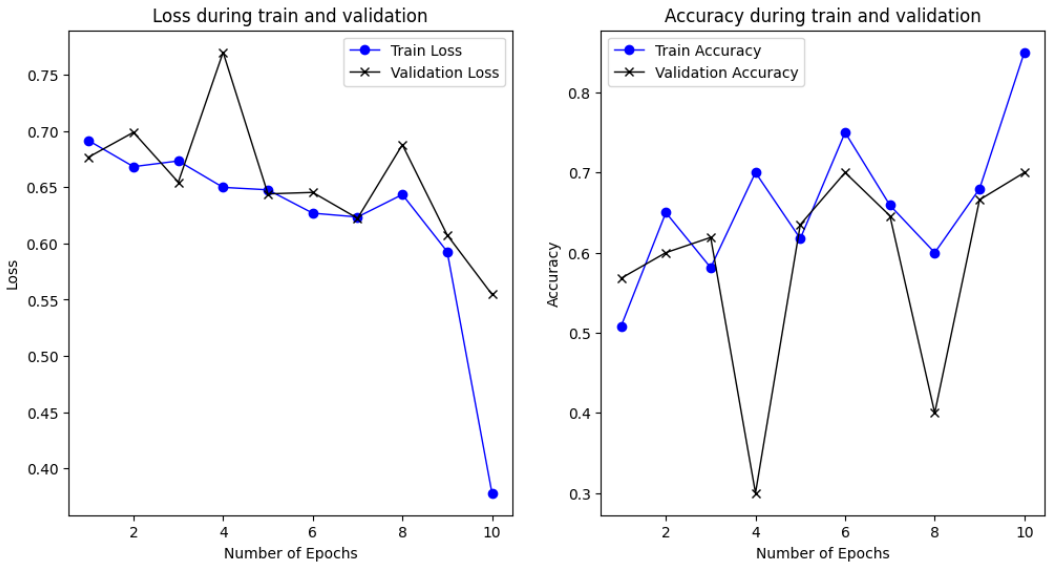

FIGURE 5 – Correction ReLU à Sigmoid dernière couche entraînement.

On peut constater qu'avec l'activation sigmoïde, la courbe s'améliore légèrement dans la courbe de "Accuracy", car la précision, bien qu'elle ait globalement diminué, est plus adéquate sur les données de validation par rapport à la période où l'activation ReLU était utilisée. Dans la courbe de "Loss", l'effet est plus évident, car la perte a globalement diminué et la validation suit mieux l'entraînement.

Couches supplémentaires de Dropout La régularisation est une technique utilisée dans les modèles de réseaux neuronaux pour réduire le risque de surapprentissage (overfitting). Le surapprentissage se produit lorsque le modèle s'ajuste trop bien aux données d'entraînement, mais ne généralise pas bien aux nouvelles données. Pour atténuer ce problème, plusieurs techniques de régularisation peuvent être appliquées, et l'une des plus courantes est l'utilisation du Dropout. Cette technique désactive aléatoirement (c'est-à-dire, met à zéro) une fraction des neurones dans le réseau à chaque étape d'entraînement, forçant ainsi le réseau à être plus robuste et à ne pas dépendre excessivement d'un neurone individuel. Cela permet au modèle d'être plus généralisable à de nouvelles données. À chaque étape d'entraînement, un pourcentage de neurones est "désactivé", et les neurones restants doivent compenser l'absence des neurones désactivés. Pendant la phase d'inférence (évaluation et prédiction), tous les neurones sont actifs, mais leurs poids sont ajustés pour éviter une dépendance excessive à certains sous-groupes spécifiques.

On ajoute des couches de Dropout avec le commande :

```
model.add(Dropout(0.3))
```

Code 12 – Couche de Dropout avec 30% des neurones.

Il est généralement recommandé d'ajouter des couches de Dropout après les couches denses (fully connected) ou après les couches de convolution (mais pas directement après les couches de pooling). Dans ce cas, une première couche de Dropout de 30% des neurones a été insérée juste avant la sortie.

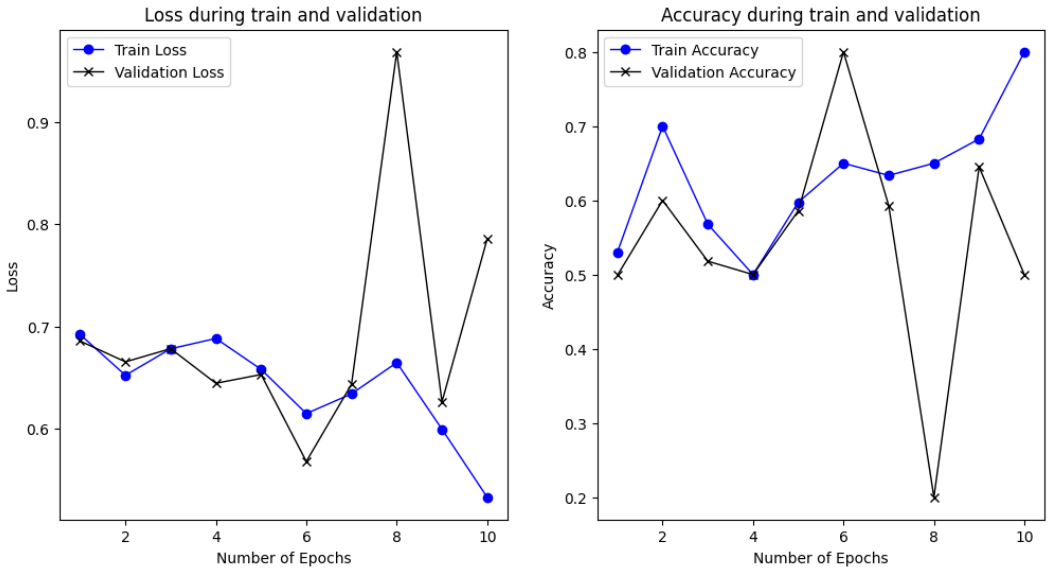

FIGURE 6 – Correction avec Dropout avant de la sortie de 30%.

Par la suite, deux autres couches de Dropout ont été ajoutées, juste après les couches de convolution finales de MaxPooling, avec un taux de 20% des neurones.

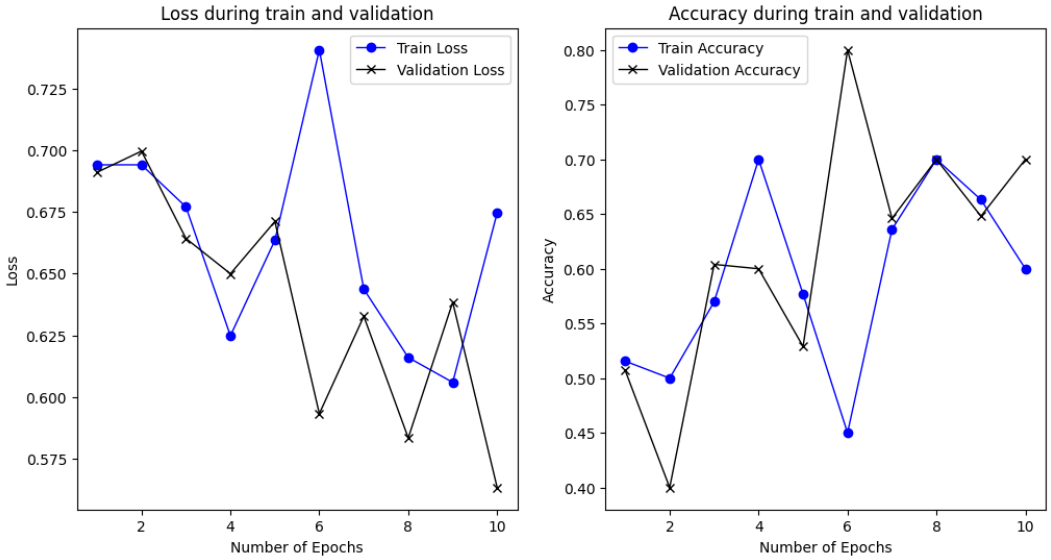

FIGURE 7 – Correction avec Dropout après des couches de MaxPooling de 20%.

À première vue, il peut sembler que l'effet du Dropout n'ait pas été significatif, mais en observant les échelles des graphiques sur les axes de "Loss" et "Accuracy", on remarque qu'après l'introduction des couches de Dropout, le réseau est devenu plus robuste et la validation est désormais plus cohérente avec l'entraînement, suivant ce dernier de manière plus précise et améliorant globalement les deux valeurs. Bien que la précision (”Accuracy”) soit globalement plus faible, la validation suit mieux l'entraînement, ce qui rend le réseau plus intéressant. De plus, les temps d'exécution du réseau ont considérablement diminué, passant de 6 à 8 minutes à environ 1 à 2 minutes, tout en affichant des résultats adéquats, ce qui démontre une optimisation claire du réseau.

Autres modifications supplémentaires : Bien qu'il existe d'autres modifications supplémentaires telles que l'EarlyStopping ou la régularisation des poids L2, cela impliquerait d'augmenter le nombre d'époques (même si le modèle s'arrête automatiquement lorsqu'il ne détecte pas d'amélioration évidente). Par conséquent, ces techniques n'ont pas été implémentées, car le modèle est déjà considérablement performant pour un réseau simple avec si peu de données. Il est envisagé de poursuivre le rapport avec l'augmentation des données.

2.4 Augmentation de données

L'augmentation de données est une technique qui génère de nouvelles échantillons à partir de celles déjà existantes en appliquant des transformations aléatoires telles que la rotation, le zoom, les déplacements et le miroir horizontal, entre autres. Cela est particulièrement utile lorsque l'on dispose d'un petit ensemble de données, car cela permet d'élargir la taille des données d'entraînement et d'aider le modèle à mieux généraliser.

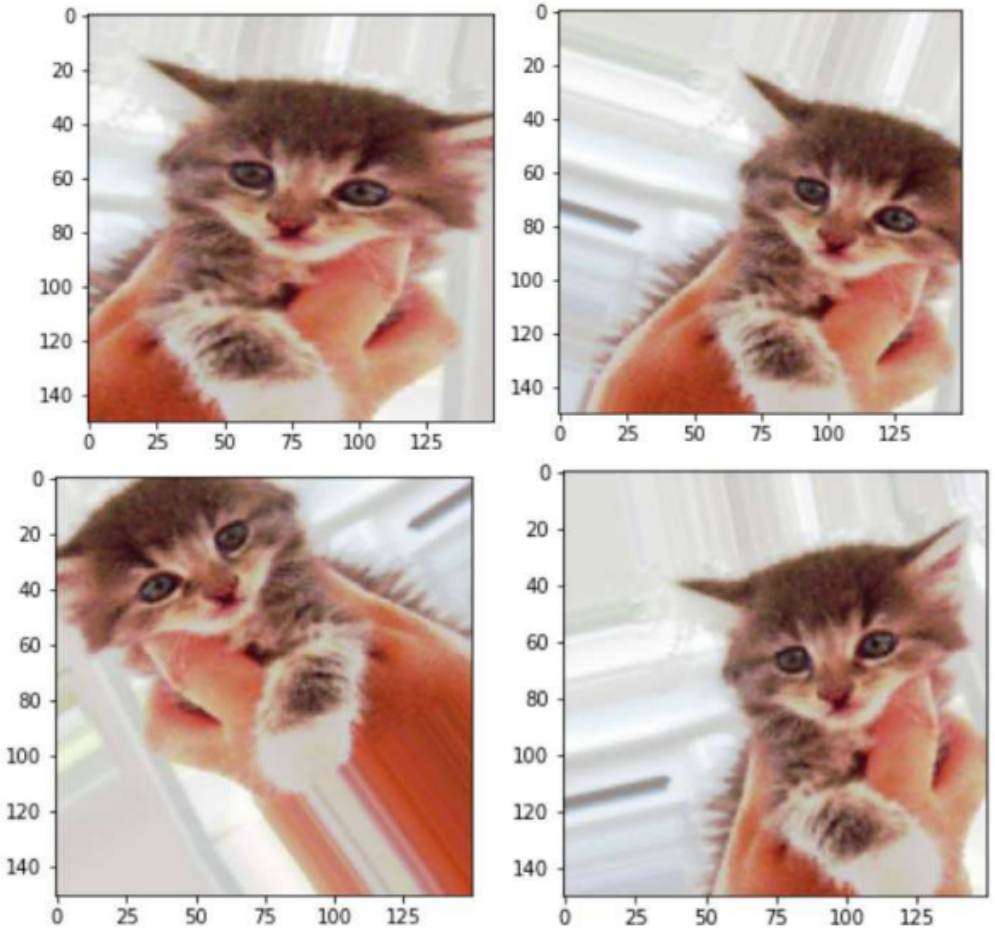

FIGURE 8 – Génération d'images de chats par augmentation aléatoire de données (1).

Il est maintenant important de parler du surapprentissage (overfitting) et de la manière dont l'augmentation de données permet de l'atténuer. Le surapprentissage survient lorsqu'un modèle apprend trop bien les détails et les motifs spécifiques des données d'entraînement, ce qui l'amène à mal se comporter sur de nouvelles données qu'il n'a pas encore vues. Le modèle, au lieu d'apprendre des caractéristiques générales qui sont utiles pour les données non vues, mémorise des particularités des images d'entraînement, ce qui le rend moins performant sur des données qu'il ne connaît pas, entraînant ainsi des échecs lors de la validation.

L'augmentation de données introduit des transformations aléatoires sur les images, créant ainsi de nouvelles versions légèrement différentes des images originales. Bien que ces nouvelles versions représentent toujours les mêmes classes, comme des chiens et des chats, elles présentent des différences visuelles qui obligent le modèle à apprendre des caractéristiques plus générales plutôt que des détails spécifiques.

Avec peu de données, le modèle a tendance à mémoriser ces mêmes données et leurs caractéristiques exactes au cours du processus d'entraînement, un problème dû à un manque de diversité et de quantité de données. Les transformations obligent le modèle à apprendre des caractéristiques invariantes aux positions, rotations, échelles ou toute autre variation

des données, sans changer la classe de l'objet. En augmentant les données, le réseau considère cela comme de nouveaux lots d'images, car ce sont des versions différentes des originales, que le modèle analyse pendant l'entraînement, ce qui a un effet positif en évitant la mémorisation de données spécifiques. Ainsi, un chien reste un chien, peu importe s'il est tourné, déplacé, agrandi, etc. Cela réduit la capacité du modèle à surapprendre sur l'ensemble de données d'entraînement, car il ne verra jamais exactement la même image lors de deux itérations consécutives. Au lieu de cela, il apprendra des motifs généraux qui sont utiles pour faire des prédictions sur des données non vues.

La dynamique est très simple : dans la commande du "ImageDataGenerator" pour les données d'entraînement, des lignes de code sont introduites pour appliquer ces transformations aux données et générer de nouvelles informations. Bien que ce soient toujours les mêmes images à la base, le réseau les interprète comme de nouvelles en raison des transformations, ce qui l'oblige à éviter le surapprentissage ("overfitting"). Le code du générateur de données est présenté ci-dessous, avec une explication de chaque commande de transformation.

```
train_datagen = ImageDataGenerator(
    # Normaliser les images à des valeurs comprises entre 0 et 1.
    rescale=1. / 255,
    # Fait pivoter les images aléatoirement jusqu'à 40 degrés.
    rotation_range=40,
    # Déplacer horizontalement les images jusqu'à 20% de leur largeur.
    width_shift_range=0.2,
    # Déplacer verticalement les images jusqu'à 20% de leur hauteur.
    height_shift_range=0.2,
    # Appliquer des transformations de cisaillement (shear) jusqu'à 20%.
    shear_range=0.2,
    # Effectuer un zoom aléatoirement jusqu'à 20%.
    zoom_range=0.2,
    # Inverser les images horizontalement.
    horizontal_flip=True,
    # Remplir les nouveaux pixels après les transformations en utilisant
    #la valeur du pixel le plus proche.
    fill_mode='nearest'
)
```

Code 13 – Train Generator avec des transformations.

En ce qui concerne les données de validation, elles restent inchangées. Le réseau proposé par le document pour cette section inclut une couche de Dropout de 50 % avant la dernière couche, ce qui est similaire à ce qui a été fait dans la section des améliorations proposées pour le réseau sans augmentation de données (1). Par conséquent, cela a été mis en

œuvre comme indiqué dans le guide, puis des améliorations ont été proposées, dont il sera question plus tard. De plus, le nombre d'époques a été augmenté à 100, ce qui peut sembler un peu excessif, mais ce sera la configuration initiale pour la première expérimentation.

Les résultats suivants sont obtenus :

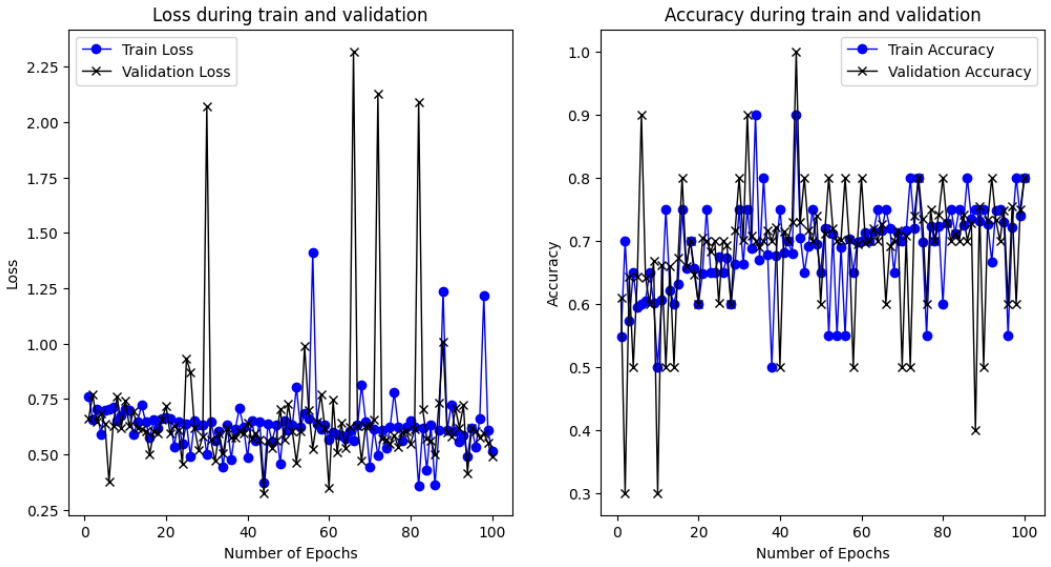

FIGURE 9 – Première entraînement du réseau augmenté.

En analysant les résultats obtenus dans les graphiques de perte (”Loss”) et de précision (”Accuracy”) après l'entraînement du réseau avec la configuration donnée, plusieurs observations peuvent être faites en comparaison avec les résultats précédents (sans l'augmentation de données).

Le comportement de la perte constate que le modèle ne parvient pas à bien se stabiliser. La perte continue de fluctuer au fil des époques, ce qui pourrait indiquer que le modèle éprouve encore des difficultés à généraliser correctement sur les données de validation. Bien que la perte de validation diminue légèrement à certains moments, les oscillations suggèrent que le modèle n'apprend pas de manière cohérente et qu'il pourrait encore rencontrer des problèmes avec les données de validation. Les pics de perte dans la validation (courbe noire) sont des signes d'instabilité. Ils montrent qu'à certains moments, le modèle produit des erreurs beaucoup plus importantes sur certains lots de données de validation, ce qui pourrait indiquer un surajustement aux lots d'entraînement ou un besoin d'une meilleure optimisation.

La précision du modèle continue de montrer des fluctuations importantes au cours des époques, ce qui est un autre indicateur d'une incohérence dans l'apprentissage. Bien que la précision de l'entraînement semble s'améliorer légèrement à certains moments, celle de la validation ne montre pas d'amélioration claire, ce qui pourrait indiquer que le modèle se surajuste aux données d'entraînement sans améliorer sa capacité à généraliser sur les données de validation.

2.4.1 Améliorations proposées

En observant qu'il existe toujours un problème d'instabilité et de possible surajustement dans le réseau, des solutions sont mises en place pour améliorer et optimiser le modèle.

Réseau modifié Le réseau a été modifié en ajoutant des couches supplémentaires après le Dropout, de manière similaire à ce qui a été fait avec le réseau sans augmentation de données, en plus de changer l'activation de la dernière couche pour une activation sigmoïde. Avec ces modifications, dont les effets prévus ont déjà été discutés, on obtient le graphique suivant des résultats :

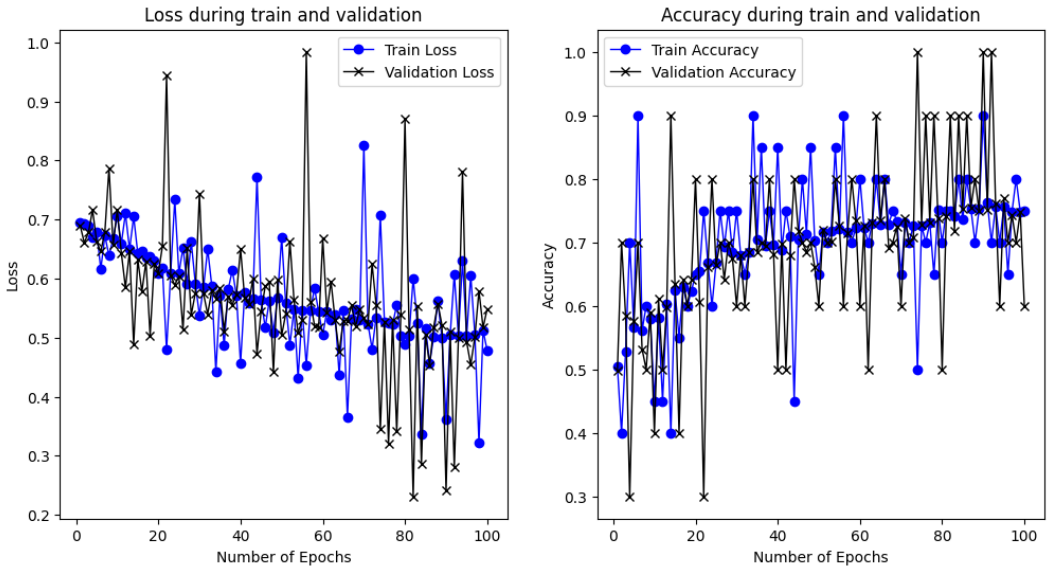

FIGURE 10 – Entraînement du réseau augmenté avec des couches supplémentaires.

Bien que le "Loss" et l'"Accuracy" se soient améliorés globalement, le réseau reste assez instable. Le suivi entre l'entraînement et la validation est bien meilleur qu'avant les modifications du réseau, il est plus robuste, mais le temps d'exécution reste le même en raison du nombre d'époques, et le surentraînement est visible dans les dernières époques, ce qui peut signifier qu'il y en a trop et que le réseau est en train de surapprendre. Cela dit, il est pertinent d'optimiser le nombre d'époques.

Early Stopping et Callback Une méthode courante pour éviter le surentraînement dû à un nombre excessif d'époques est de générer un Early Stopping, un type de Callback, qui arrête le code lorsque cela est nécessaire. Il est d'abord important de comprendre ce que sont les Callbacks, puis d'implémenter l'Early Stopping.

Les Callbacks sont des outils puissants dans Keras qui permettent de contrôler le processus d'entraînement de manière plus dynamique et efficace. Ils aident à améliorer les performances du modèle, à faire des ajustements automatiques pendant l'entraînement, et à optimiser l'utilisation des ressources en arrêtant l'entraînement lorsqu'il n'est plus nécessaire, tout cela en analysant les résultats de manière dynamique à des moments clés de l'exécution, comme à la fin d'une époque dans ce cas.

L'Early Stopping est un Callback qui arrête l'entraînement du modèle lorsque celui-ci ne s'améliore plus. En effet, il arrive que le modèle continue à s'entraîner pendant plusieurs époques supplémentaires, sans pour autant s'améliorer en termes de validation et, au contraire, commence à surapprendre. L'Early Stopping surveille les performances sur les données de validation et interrompt l'entraînement lorsqu'il n'y a plus d'améliorations significatives, ce qui permet d'éviter le surapprentissage et de gagner du temps d'entraînement.

Pour implémenter l'Early Stopping, on commence par configurer le Callback en utilisant la ligne de code suivante :

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
```

Code 14 – Impleméntation Callback pour l'Early Stopping.

Dans cette ligne, on définit la variable Monitor, qui indique ce qui est surveillé, dans ce cas la valeur de la perte (Loss). Ensuite, on définit la patience, c'est-à-dire le nombre d'époques que le modèle attend avec des résultats similaires ; ici, après 10 époques sans amélioration, le modèle est arrêté. Enfin, les poids de l'époque avec la plus faible perte de validation sont stockés et restaurés, ce qui peut être utile pour analyser le réseau.

Ensuite, la ligne de Callbacks est ajoutée à la fonction qui exécute le modèle, en ajoutant la ligne nécessaire dans le modèle, ce qui donne le bloc de code suivant :

```
history = model.fit(
    train_generator,
    steps_per_epoch = 100,
    epochs = 100,
    validation_data = validation_generator,
    validation_steps = 50,
    callbacks=[early_stopping] #Appel du Callback type Early Stopping
)
```

Code 15 – Modèle avec le Callback.

Une question classique est de savoir quand utiliser la variable "Accuracy" pour l'Early Stopping et quand utiliser la "Loss". Bien, "Accuracy" est utilisée lorsque les données sont bien équilibrées et que l'objectif est d'améliorer le pourcentage de prédictions correctes. Surveiller la précision est alors plus utile et aligné avec l'objectif final du projet. En revanche, "Loss" est surveillée lorsqu'on souhaite détecter des améliorations, même lorsque la précision ne change pas beaucoup, car la perte est plus sensible aux petits changements et plus adaptée lorsque le modèle est encore en phase d'apprentissage.

Après avoir appliqué l'Early Stopping, les résultats suivants ont été obtenus :

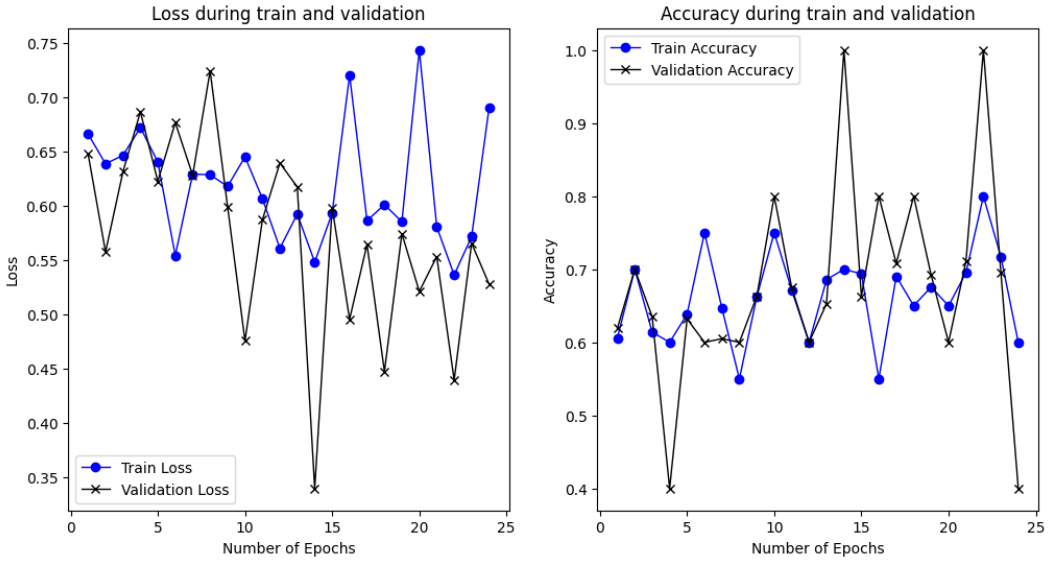

FIGURE 11 – Entraînement du réseau augmenté avec des couches supplémentaires et Early Stopping.

Bien que l'Early Stopping ait rendu le modèle beaucoup plus optimal et efficace, il reste très instable, avec des améliorations globales de l'erreur et de la précision, mais avec des pics instables qui peuvent entraîner des mauvaises interprétations par le réseau et même être un signe de surentraînement.

Learning Rate et LearningRateScheduler Le taux d'apprentissage est un paramètre de la compilation du modèle, déjà vu précédemment, qui est généralement un hyperparamètre contrôlant la taille des pas que l'optimiseur effectue lors de la mise à jour des poids du modèle. En d'autres termes, il détermine dans quelle mesure les poids changent en réponse à l'erreur mesurée à chaque itération de l'entraînement. Des taux très élevés peuvent rendre les modèles rapides mais instables, tandis que des taux très bas rendent les modèles robustes mais lents. Il est donc essentiel de trouver un équilibre optimal. Parfois, le réglage de la Learning Rate peut être un peu complexe, c'est pourquoi Keras propose un autre Callback qui permet de modifier dynamiquement le taux d'apprentissage. Ainsi, lorsqu'un plateau est atteint (un point où le taux d'apprentissage ne change plus), le Callback effectue automatiquement un ajustement en réduisant le taux avec un facteur spécifique. Ce processus se poursuit jusqu'à atteindre un taux minimal défini, et cela est appelé LearningRateScheduler.

Tout comme le Callback d'Early Stopping, ce Callback est implémenté en le configurant et en l'insérant ensuite dans la ligne de commande des Callbacks du modèle, activant ainsi les deux Callbacks simultanément et rendant le modèle beaucoup plus optimal.

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5,
min_lr=1e-6)
```

Code 16 – Impleméntation Callback pour LearningRateScheduler.

Dans ce cas, le LearningRateScheduler reçoit comme paramètres le moniteur, la patience, le facteur de réduction du taux et le taux minimal à atteindre. La modification pour l'ajouter

se fait dans la ligne des Callbacks du modèle.

```
history = model.fit(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50,
    callbacks=[early_stopping, reduce_lr]
    #Appel du Callback type Early Stopping et LearningRateScheduler
)
```

Code 17 – Modèle avec les Callbacks.

Après ces modifications, le modèle donne le résultat suivant :

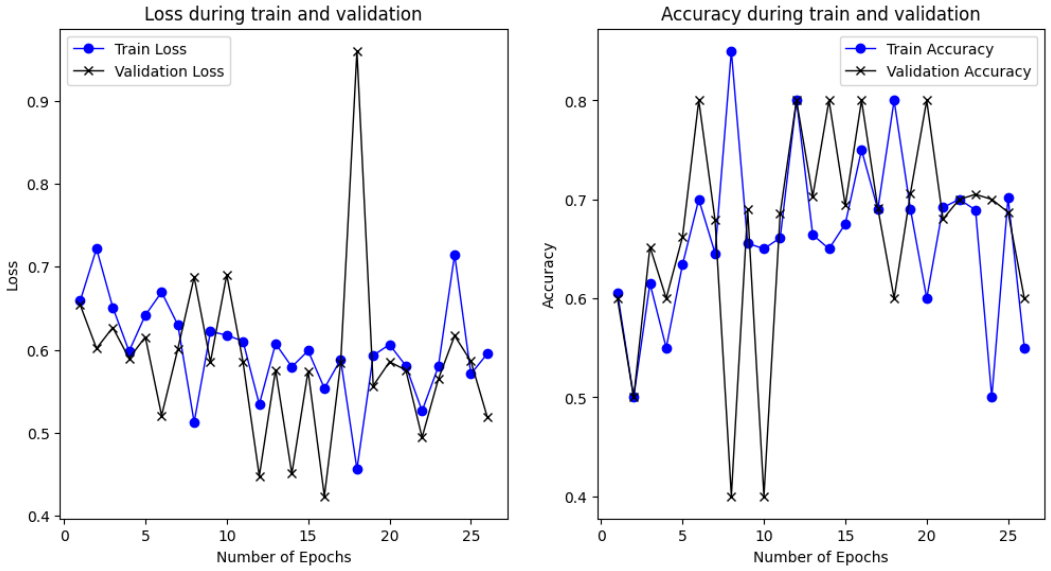

FIGURE 12 – Entraînement du réseau augmenté avec des couches supplémentaires, Early Stopping et LearningRateScheduler.

Bien que la mise en place du taux d'apprentissage dynamique ait été effectuée, le réseau reste assez instable. Globalement, le Loss et l'Accuracy se sont améliorés, et l'entraînement ainsi que la validation sont plus cohérents, ce qui a également amélioré le temps d'entraînement avec des durées inférieures à 2 minutes. Cependant, le modèle reste peu robuste, en raison de l'instabilité observée, qui peut être due à un surajustement ou à un manque de données, rendant ainsi le réseau peu fiable.

Par conséquent, on conclut que le réseau, bien qu'il soit nettement meilleur que l'original, reste très instable, même avec l'augmentation de données, les couches de Dropout, l'activation sigmoïde et les Callbacks. Ainsi, on procède à l'utilisation d'un réseau préentraîné.

3 Utilisation d'un ConvNet pré-entraîné.

Pour cette partie, un grand réseau ConvNet pré-entraîné sur le jeu de données ImageNet a été utilisé. Ce jeu de données contient 1,4 million d'images étiquetées et 1000 classes différentes, y compris des espèces variées de chiens et de chats, ce qui rend ce réseau particulièrement adapté au problème de classification entre chiens et chats. L'utilisation de ce réseau pré-entraîné a permis d'améliorer les performances de la classification grâce à l'apprentissage initial sur un jeu de données beaucoup plus vaste et varié que celui utilisé pour l'entraînement à partir de zéro.

De manière générale, un réseau pré-entraîné est un réseau neuronal qui a déjà été entraîné sur un ensemble de données massif et qui a appris un grand nombre de caractéristiques générales sur les images, comme les contours, les textures, les formes et les objets communs. Son utilisation est très utile car, au lieu d'entraîner un modèle depuis zéro (ce qui pourrait prendre beaucoup de temps et nécessiter un grand ensemble de données), on peut réutiliser ce modèle déjà entraîné et l'adapter au problème spécifique, comme dans ce cas la classification des chiens et des chats. Il existe deux manières d'utiliser les réseaux pré-entraînés :

1. Extraction de caractéristiques :
   - Dans cette approche, les premières couches du réseau pré-entraîné sont figées et seules les couches supérieures (généralement des couches denses ou de classification) sont entraînées.
   - Le réseau pré-entraîné a déjà appris à détecter des caractéristiques générales comme les contours, les formes et les textures, de sorte que ces caractéristiques peuvent être utilisées pour classer les images sans réentraîner tout le modèle.
   - Il suffit d'ajouter une dernière couche dense ou de classification spécifique à la tâche et d'entraîner cette partie, tandis que les couches convolutionnelles pré-entraînées restent figées.
2. Réglage de précision :
   - Dans cette approche, on commence de la même manière que pour l'extraction de caractéristiques (c'est-à-dire en figeant les premières couches).
   - Ensuite, certaines des couches les plus profondes sont "défigées" et ces couches supplémentaires sont entraînées en même temps que les couches supérieures.
   - Cet ajustement fin permet au réseau d'apprendre des caractéristiques plus spécifiques à la tâche en question, tout en conservant les caractéristiques générales des premières couches.

Dans ce cas, on a utilisée utiliser l'extraction de caractéristiques, car elle est idéale pour les petits ensembles de données, souvent plus petits que ceux utilisés pour le réseau pré-entraîné. Le réseau que on a utilisée est le VGG-16, qui présente l'architecture suivante :

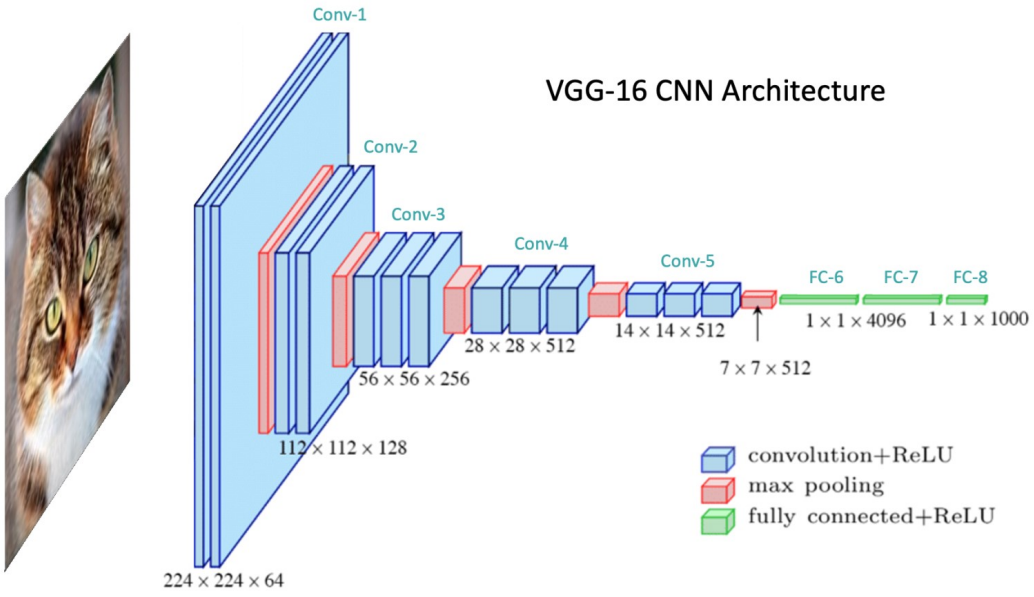

FIGURE 13 – Architecture VGG-16 CNN (4).

3.1 Extraction de caractéristiques

Pour l'utilisation du réseau neuronal préentraîné, nous utilisons la modalité d'extraction de caractéristiques. L'extraction est réalisée de manière très simple avec les lignes de code suivantes :

```
1 from keras.applications.vgg16 import VGG16
2
3 conv_base = VGG16(weights='imagenet', include_top=False,
4 input_shape=(150, 150, 3))
```

Code 18 – Extraction VGG-16.

En essence, le modèle VGG-16 est importé depuis Keras. Ensuite, il est chargé avec sa commande propre, en lui indiquant que les poids utilisés seront ceux préentraînés sur ImageNet, la base de données de 1,4 million d'images réparties en 1000 classes, parfaite pour les réseaux d'images comme dans ce cas. Il est également spécifié de ne pas inclure les couches supérieures denses du modèle, c'est-à-dire les Fully Connected initiales, et enfin, la taille des inputs est définie, de manière similaire à ce qui a été fait avec le réseau sans préentraînement.

Avec tout cela, nous obtenons le résumé suivant du réseau préentraîné, qui montre les couches préchargées et figées.

Layer (type)	Output Shape	Param #
input_layer_16 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1,792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36,928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73,856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147,584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295,168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590,080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590,080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

FIGURE 14 – Résumé du réseau préentraîné.

Comme on peut le voir, le réseau contient plusieurs couches convolutionnelles et le nombre de paramètres est bien supérieur à celui du réseau non entraîné (14,71 millions contre 3,45 millions).

3.2 Extraction de caractéristiques sans augmentations des données et Classifieur

Avec le réseau préentraîné prêt, l'idée générale est d'ajouter un Classifieur au-dessus de la dernière couche, ayant la même forme, et qui est réalisé en ajoutant des couches propres définies selon les objectifs du projet actuel, aboutissant à un Classifieur entièrement connecté.

Pour cela, on exécute d'abord les instances de ImageDataGenerator pour extraire les images sous forme de tableaux *NumPy* avec leurs étiquettes respectives.

Les premières lignes de code sont simples et permettent de spécifier le répertoire où se trouvent les données des images (chiens et chats), de les reconfigurer en binaire et de définir la taille de chaque batch.

```
base_dir = 'dogs_cats'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20
```

Code 19 – Directoires et ImageDataGenerator.

La fonction ”extract_features” est l'élément clé du processus d'extraction de caractéristiques.

```
def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary'
    )
    i = 0
        features_batch = conv_base.predict(input_batch)
        features[i * batch_size:(i + 1) * batch_size] = features_batch
        labels[i * batch_size:(i + 1) * batch_size] = label_batch
        i += 1
        if i * batch_size >= sample_count:
            break

    return features, labels
```

Code 20 – Boucle for pour extraire les caractéristiques.

Tout d'abord, il y a la préaffectation d'espace, où des tableaux de zéros (vides) sont créés pour les caractéristiques à extraire, avec une taille de 4x4x512, correspondant à la sortie de la dernière couche de VGG-16 : des cartes de caractéristiques de taille 4x4 avec 512 filtres. Quant aux étiquettes, elles ne sont que des stockages de valeurs binaires (1 ou 0).

Ensuite, un générateur est créé, très similaire à celui déjà utilisé dans le réseau non entraîné, avec les indications concernant la taille cible (150x150, entrée pour VGG-16), la taille du batch et la classe binaire.

La boucle for se charge simplement de remplir ces tableaux vides, en utilisant des batches de la base de données et en les faisant passer par VGG-16 à l'aide de la fonction *Predict*, qui est l'élément central de la boucle. Enfin, ces tableaux *NumPy*, contenant les résultats de l'interprétation des caractéristiques des images par le réseau préentraîné, sont retournés.

Après cela, la fonction est appelée pour générer l'extraction des caractéristiques avec l'entraînement, la validation et le test.

```
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

Code 21 – Extraction en utilisant "extract_features".

Ensuite, comme le modèle passe aux couches denses, similaire à ce qui s'est passé avec le réseau non entraîné, il est nécessaire d'aplatir ces tableaux obtenus lors de l'extraction des caractéristiques. Pour cela, on utilise la fonction *reshape*, qui transforme les vecteurs en une dimension, en tenant compte des 2000 images pour l'entraînement et des 1000 pour la validation.

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
train_labels = np.reshape(train_labels, (2000 * 1))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
validation_labels = np.reshape(validation_labels, (1000 * 1))
```

Code 22 – Redimmensionement des vecteurs pour l'entrée.

3.3 Génération du modèle dense entièrement connecté ou Classifieur

Après avoir extrait toutes les caractéristiques du VGG-16 et figé ses couches, nous procédons à la génération du modèle dense, qui sera la partie ajoutée et ajustée au modèle spécifique. Le code suivant est utilisé pour cela :

```
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

Code 23 – Modèle dense séquentiel.

Tout d'abord, le modèle est défini comme séquentiel, permettant d'ajouter les couches une par une. Ensuite, on génère les trois couches d'intérêt : une première couche dense avec 256 neurones qui prend les caractéristiques 4x4x512 (8192 sous forme unidimensionnelle) et les réduit à une dimension de 256, avec une activation ReLU. La deuxième est une couche

Dropout de 50%, et la troisième est la couche de sortie avec un neurone et une activation binaire sigmoïde.

Ensuite, la commande standard est utilisée pour compiler le modèle, exactement comme cela a été fait pour le réseau non entraîné, avec un taux d'apprentissage de 2e-5, une perte par entropie croisée binaire et un suivi de la "Accuracy".

```
model.compile(optimizer=optimizers.RMSprop(learning_rate=2e-5),
              loss='binary_crossentropy',
              metrics=['acc'])
```

Code 24 – Compilation du modèle.

Enfin, le modèle est exécuté avec la commande standard "model.fit", sur 30 époques et des batchs de 20 images.

```
history = model.fit(train_features,
                    train_labels,
                    epochs = 30,
                    batch_size = 20,
                    validation_data = (validation_features, validation_labels))
```

Code 25 – Entraînement du modèle.

3.4 Résultats finales

En réalisant cette opération, nous obtenons les résultats suivants :

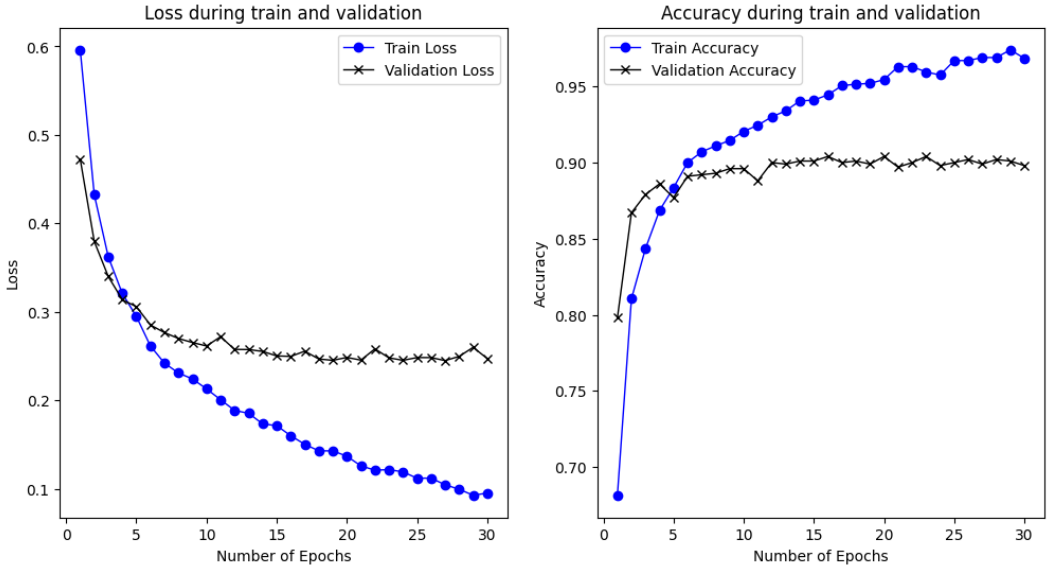

FIGURE 15 – Entraînement du réseau préentraîné.

Après avoir analysé les résultats, il est clairement évident que le réseau préentraîné a montré une nette amélioration par rapport au réseau non entraîné précédemment. Globalement, la ”Loss” s’est améliorée, atteignant pratiquement 0, et la ”Accuracy” s’est améliorée pour atteindre pratiquement 1. Les données de validation suivent maintenant de très près celles de l’entraînement, et le plus important, le modèle est complètement stable, sans pics dangereux ni surapprentissage (overfitting) excessif non souhaité.

3.4.1 Améliorations proposées

Des Callbacks sont implémentés dans le réseau et le nombre d’époques est augmenté, obtenant ainsi le résultat suivant :

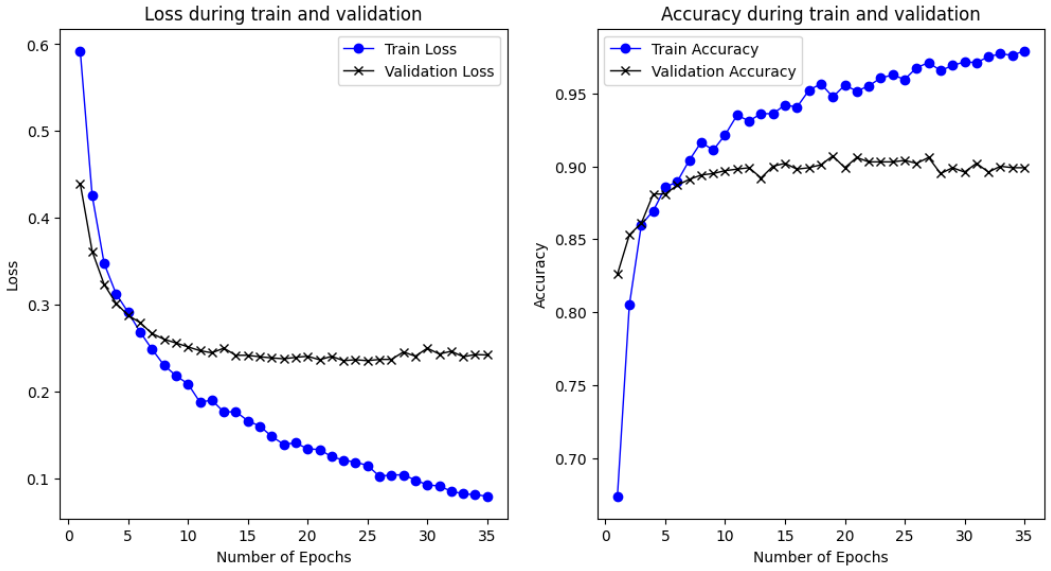

FIGURE 16 – Entraînement du réseau augmenté avec Early Stopping et LearningRateScheduler.

Comme on peut le constater, les résultats ne varient pas beaucoup. Par conséquent, pour optimiser davantage le réseau, il faudrait implémenter d’autres stratégies qui dépassent le cadre de ce rapport, telles que l’augmentation de données, le changement du nombre de neurones ou le Fine-Tuning en dégelant des couches du réseau VGG-16.

4 Conclusions

Le travail pratique a permis d'explorer deux approches principales pour la classification d'images : l'entraînement d'un modèle ConvNet à partir de zéro et l'utilisation d'un réseau pré-entraîné (VGG-16) pour l'extraction de caractéristiques.

L'entraînement du modèle à partir de zéro a montré des résultats initiaux encourageants, mais a révélé des signes de surapprentissage (overfitting) et une instabilité importante, malgré les améliorations telles que l'ajout de Dropout, l'activation sigmoïde et l'augmentation de données. Bien que ces modifications aient permis de stabiliser en partie le réseau et d'améliorer la performance générale, la robustesse du modèle restait insuffisante.

L'utilisation du réseau pré-entraîné VGG-16 a significativement amélioré les performances du modèle, notamment grâce à sa capacité à exploiter des caractéristiques générales préalablement apprises sur un ensemble de données beaucoup plus vaste (ImageNet). L'ajout d'un classifieur dense a permis d'obtenir une meilleure précision et une réduction de la perte, tout en garantissant une stabilité accrue.

Cependant, pour atteindre des résultats optimaux, il reste des pistes d'amélioration comme le fine-tuning des couches congelées du modèle pré-entraîné ou l'ajustement des hyperparamètres. En définitive, l'approche par réseaux pré-entraînés se révèle particulièrement efficace dans des contextes où l'ensemble de données est limité.

Bibliographie

[1] Ababsa, F. Introduction au deep learning : Application à la vision par ordinateur. *TP CNN* (2024).

[2] Haque, N. What is convolutional neural network — cnn (deep learning). *LinkedIn, Microsoft* (2023).

[3] IBM. ¿qué son las redes neuronales ? *Topic : Neural Networks* (2024). Accès :13 de Octobre de 2024.

[4] LLC, B. V. Convolutional neural network (cnn) : A complete guide. *Getting Started with TensorFlow and Keras* (2024). Accès :13 de Octobre de 2024.