



Rapport de Projet

HAI606I : Projet de Programmation 2

”Amusons-nous avec des images”

par

Romain GALLERNE

Baptiste BRONSIN

Morgan NAVEL

Éric GILLES

Lisa SAVY

Encadrant : M. Pascal PONCELET

Responsable du module : Mme Anne-Elisabeth BAERT

Laboratoire d’Informatique, de Robotique et de Microélectronique de Montpellier



Table des matières

1	Introduction	3
2	Classification d'images	4
2.1	Introduction	4
2.2	Les fondamentaux de l'apprentissage profond	4
2.2.1	Rappel sur la classification supervisée	4
2.2.2	Rappels sur les réseaux de neurones	5
2.2.3	Rappel sur les réseaux de neurones convolutifs (CNN)	8
2.2.4	Rappels sur la descente de Gradient	10
2.3	Présentation des données	11
2.4	Premier modèle et résultats	12
2.4.1	Résultats modèle éléphant	13
2.4.2	Résultat modèle tigre	14
2.4.3	Résultat modèle renard	15
2.4.4	Explication des résultats	16
2.5	Des extensions au modèle de base	17
2.5.1	Structure du modèle	17
2.5.2	Modèle plus complexe d'éléphant	18
2.5.3	Modèle plus complexe de tigre	19
2.5.4	Modèle plus complexe de renard	19
2.5.5	Résultats	20
2.6	Conclusion	20
3	Augmenter le nombre d'images	21
3.1	Rappel sur la fonction ImageDataGenerator	21
3.2	Résultats obtenus	21
3.3	Conclusion	22

4 Vers des modèles plus complexes	23
4.1 Rappel sur les modèles plus complexes	23
4.1.1 Réseau neuronal résiduel	24
4.2 Application de Resnet	25
4.3 Conclusion	25
5 Auto-encodeurs	26
5.1 Introduction	26
5.2 Réduction de données	26
5.3 L'utilisation des auto-encodeurs pour débruiter	28
5.4 Conclusion	30
6 Génération d'images	31
6.1 Introduction	31
6.2 Réseaux antagonistes génératifs (GANs)	31
6.3 Ouverture sur la génération d'image	32
7 Organisation du projet	33
8 Conclusion	35
Bibliographie	36

Introduction

L'IA est un domaine qui vise à créer des systèmes informatiques capables de réaliser des tâches qui nécessitent normalement une intelligence humaine, comme la reconnaissance vocale, la traduction automatique, la conduite autonome et bien d'autres encore. Il existe plusieurs catégories d'intelligences artificielles, mais celle qui va ici nous intéresser est l'apprentissage automatique, de son nom anglais "machine learning", qui utilise des algorithmes pour apprendre à partir de données. L'apprentissage profond, ou plus connu sous son nom anglais "deep learning", est une sous-catégorie d'apprentissage qui utilise des réseaux de neurones pour résoudre des problèmes complexes.

L'objectif de ce projet est le développement et la mise en place d'un réseau de neurones que nous améliorerons le plus possible au fil du projet en utilisant de nouvelles méthodes afin d'obtenir le meilleur taux de réussite possible. Nous nous essaierons également à la modification automatique d'images et théoriserons une éventuelle poursuite du projet qui serait la génération d'image. Le code développé a été entièrement réalisé en Python, nous avons travaillé sur le logiciel Google Colaborate. Pour le rapport, il a été réalisé en LaTeX sur le site Overleaf. En effet, le Python est un langage massivement utilisé dans le domaine de l'IA car celui-ci contient énormément de fonctionnalités permettant de mettre en place des réseaux de neurones de façon efficace.

Dans ce rapport, nous allons expliquer ce qu'est l'apprentissage profond ainsi que présenter nos résultats lors du développement de notre propre réseau de neurones de reconnaissance d'image. Nous aborderons les différentes notions auxquelles nous nous sommes confrontés. Nous aborderons aussi brièvement la génération d'image avec l'image data generator.

Classification d'images

2.1 Introduction

2.2 Les fondamentaux de l'apprentissage profond

On se propose ici d'aborder quelques-unes des notions fondamentales de l'apprentissage profond qui seront ensuite évoquées tout au long du présent rapport. Les points abordés ici le sont principalement à but de rappel pour la suite. Le lecteur souhaitant approfondir des notions génériques sur l'intelligence artificiel et l'apprentissage profond pourra consulter les sources suivantes : [6] [9] [4] [8]

2.2.1 Rappel sur la classification supervisée

La classification supervisée est un type de classification où le modèle doit classer des images en différentes catégories (ou classes) qui sont connues au préalable par le modèle, car données par le créateur, d'où le nom "supervisée".

Afin que le modèle puisse classer les images, il est nécessaire que celui-ci apprenne sur des images d'exemple, c'est ce qu'on appellera le **jeu d'apprentissage**. Ainsi, ce jeu sera utilisé comme modèle d'entraînement. On le distingue du **jeu de test** qui sera utilisé afin de tester notre algorithme sur des images qu'il n'a encore jamais rencontrées. On pourra alors mesurer l'**accuracy** de l'algorithme, c'est-à-dire son pourcentage de réussite dans la reconnaissance des images et leur classification.

On pourra d'ailleurs distinguer deux types d'accuracy. L'accuracy d'entraînement, qui est le pourcentage de réussite sur le jeu d'apprentissage en phase de test et l'accuracy de validation qui est le pourcentage de réussite sur le jeu de test. Il est important que ces deux valeurs soient proches, car dans le cas où l'accuracy du jeu d'entraînement dépasserait celle du jeu de test, cela voudrait dire que nous sommes en situation de **sur-apprentissage**,

c'est-à-dire que l'algorithme a appris à reconnaître le jeu d'apprentissage par coeur, ce que nous voulons éviter.

Afin d'obtenir une valeur d'accuracy la moins biaisée possible, on souhaite réaliser différents tests avec divers jeux de tests et d'apprentissage. On séparera donc un unique jeu en deux, un jeu d'apprentissage et un jeu de test qui seront différents à chaque nouvel apprentissage. L'accuracy totale sera la moyenne de l'accuracy de ces différents essais avec tous ces jeux. Afin de simplifier, nous appellerons "fold" ou "k-fold" le nombre d'apprentissages différents servant à réaliser cette moyenne. Le lecteur curieux pourra tout de fois se renseigner davantage sur la notion de fold grâce aux sources en bibliographie [1] [7], mais nous ne les détaillerons pas plus ici.

Dans la suite, nous nous intéresserons principalement aux réseaux de neurones qui sont un type de classificateur pour faire de la classification supervisée.

2.2.2 Rappels sur les réseaux de neurones

Abordons le concept de réseaux de neurones. Les réseaux de neurones, ou réseaux de neurones profonds, utilisés en deep learning sont souvent appelés ainsi en raison de la profondeur de leur architecture, c'est-à-dire le nombre de couches de neurones qu'ils contiennent. Un réseau de neurones peut avoir des dizaines, voire des centaines de couches, chacune contenant des milliers de neurones.

Ces réseaux sont des modèles dit "inspirés du cerveau humain" car ceux-ci sont composés de nombreux noeuds (ici appelés neurones) connectés entre eux. Chacun de ces noeuds représente une unité de calcul qui effectue une opération mathématique sur les entrées qu'il reçoit, avant de transmettre le résultat à d'autres neurones dans le réseau.

Ainsi, lorsque l'on qualifie ces unités de neurones, cela pourrait être considéré comme un abus de langage. En effet, ce ne sont en réalité que des fonctions mathématiques effectuant une opération bien précise, le parallèle fait avec le cerveau humain est plutôt à faire sur la méthode d'apprentissage.

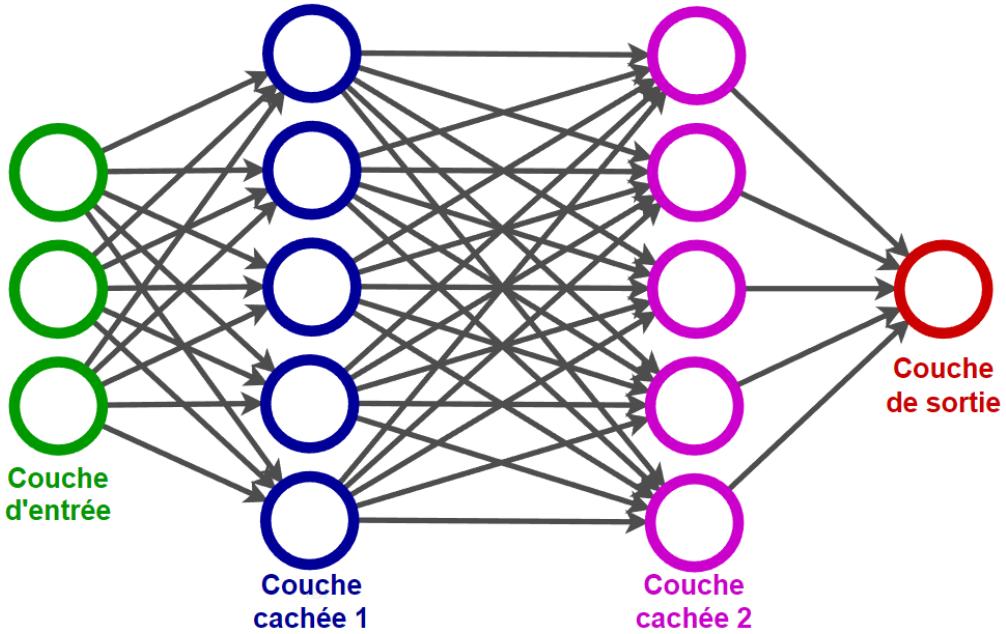


FIGURE 2.1 – Représentation graphique d'un réseau de neurones

Comme on peut le constater dans la figure ci-dessus (Figure 2.1), un réseau de neurones est composé de différentes couches, voici les principales :

- La **couche d'entrée**, ou "input layer" en anglais, est la couche de neurones qui va recevoir les données et les transmettre aux couches suivantes. Elle est l'une des deux couches dites "visibles". Le nombre d'input layer correspond au type de données en entrée.
- Les **couches cachées**, ou "hidden layers", sont une succession pouvant être plus ou moins grande de couches de neurones dites "cachées" ou "invisibles". C'est l'algorithme lui-même qui manipule ces couches. Chacune des couches cachées est composée d'une série de neurones alignés pouvant contenir plusieurs milliers de noeuds.
- Enfin, la **couche de sortie**, ou "output layer", est composée d'un unique noeud, car notre sortie est binaire. Ce noeud va permettre d'extraire la valeur de sortie du réseau pour l'entrée prodiguée. Dans le cas d'une sortie non-binaire, on peut imaginer plusieurs noeuds de sortie. On peut voir la sortie binaire d'une image sur la Figure 2.2.

Tous les neurones sont reliés entre eux et on associe à chacune de ces liaisons des poids aléatoires. À l'issue des premières phases d'apprentissage, le réseau procèdera à une "forward propagation", c'est-à-dire une mise à jour des poids de chaque noeud afin de faire correspondre l'image et le résultat attendue. On utilise ensuite une fonction d'activation pour casser la linéarité, les plus utilisés étant ReLu ou sigmoïd. Celle-ci ont pour objectif de limiter le surapprentissage en évitant que le réseau entier s'adapte à l'image courante. Certains noeuds resteront ainsi inchangés à chaque propagation. On évite ainsi l'apprentissage par coeur du jeu de données et on permet la généralisation du modèle. Enfin, on évalue l'erreur du réseau et on effectue une backpropagation. La backpropagation vise à corriger les erreurs selon l'importance de la contribution de chaque élément, ainsi, plus un élément est impactant dans le résultat, plus celui-ci sera adapté.

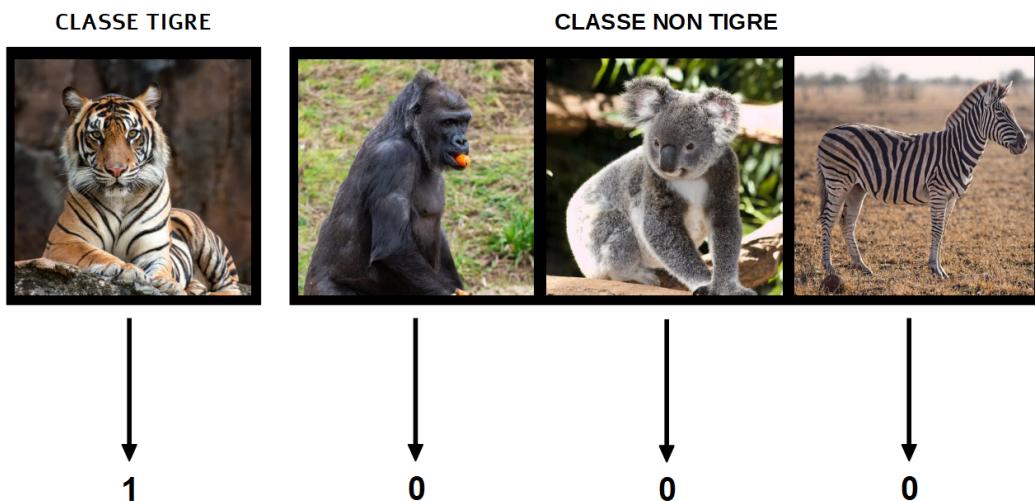


FIGURE 2.2 – Exemple de classification d'image de tigre avec la sortie 1 pour la reconnaissance d'un tigre et la sortie 0 dans le cas inverse.

- **La fonction ReLU (Rectified Linear Unit)** est couramment utilisée dans les réseaux de neurones. Elle a pour but de corriger les valeurs négatives des données d'entrée en les remplaçant par zéro, tout en laissant inchangées les valeurs positives.

Cette correction présente de nombreux avantages. Elle peut accélérer le processus d'apprentissage en réduisant le nombre de neurones inactifs dans le réseau. Elle peut également aider à prévenir le sur-apprentissage en limitant l'impact des valeurs négatives sur les poids du réseau. Enfin, elle peut améliorer la stabilité de la formation en évitant les valeurs négatives exponentielles qui peuvent survenir avec d'autres types de couches de correction.

Nous invitons les lecteurs souhaitant approfondir ces notions à consulter nos sources :
[3] [5]

En somme, la couche de correction ReLU est un élément important des réseaux de neurones convolutifs (CNN), qui peut contribuer à améliorer leur performance et leur stabilité.

2.2.3 Rappel sur les réseaux de neurones convolutifs (CNN)

Une sous-catégorie de réseaux de neurones, appelés les réseaux de neurones convolutifs ou CNN (pour convolutional neural networks en anglais), sont utilisés pour les tâches de reconnaissances.

Dans notre cas, nous allons utiliser ces réseaux de neurones afin de classifier des images. Par exemple, il pourra déterminer si cette image représente un tigre ou un renard.



FIGURE 2.3 – Reconnaissance par réseaux de neurones convolutifs

Les CNN s'appuient sur plusieurs étapes, comme détaillé dans la Figure 2.4, afin d'effectuer la reconnaissance ou non d'une image.

- **La convolution** est la première opération qui permet de détecter les caractéristiques clés de l'image à l'aide de filtres qui sont décalés à travers l'entrée. Le résultat de l'opération est enregistré sous forme d'une matrice. Ce processus mathématique permet ainsi de capturer les relations entre les pixels d'une image.

- **L'opération de Pooling** vise à minimiser la taille des images d'entrée, à alléger la charge de travail en réduisant le nombre de paramètres du réseau et donc la complexité du calcul, tout en préservant les caractéristiques clés de l'image.

- **Le Flattening**, ou mise à plat, est une étape importante dans la construction des réseaux de neurones convolutifs, car il permet de convertir les matrices issues des couches de convolution et de pooling en un vecteur sur une dimension. Lors de ce processus, toutes les caractéristiques extraites de l'image par les couches de convolutions et de pooling sont concaténées.

- **La classification** est la dernière étape pour la reconnaissance d'images avec les réseaux de neurones convolutifs. L'objectif de cette étape est de catégoriser les images entrantes en différentes classes en fonction des caractéristiques extraites des opérations précédentes. Par exemple, pour reconnaître un chat, les caractéristiques, telles qu'avoir quatre pattes ou avoir des poils, seront accentuées par les étapes de convolutions et de pooling et le fait d'avoir des plumes ou une trompe seront atténués. Ces caractéristiques sont apprises au cours d'un processus d'entraînement en utilisant un algorithme d'apprentissage automatique tel que la descente de gradient, permettant ainsi de faire des prédictions pour déterminer la classe à laquelle l'image appartient.

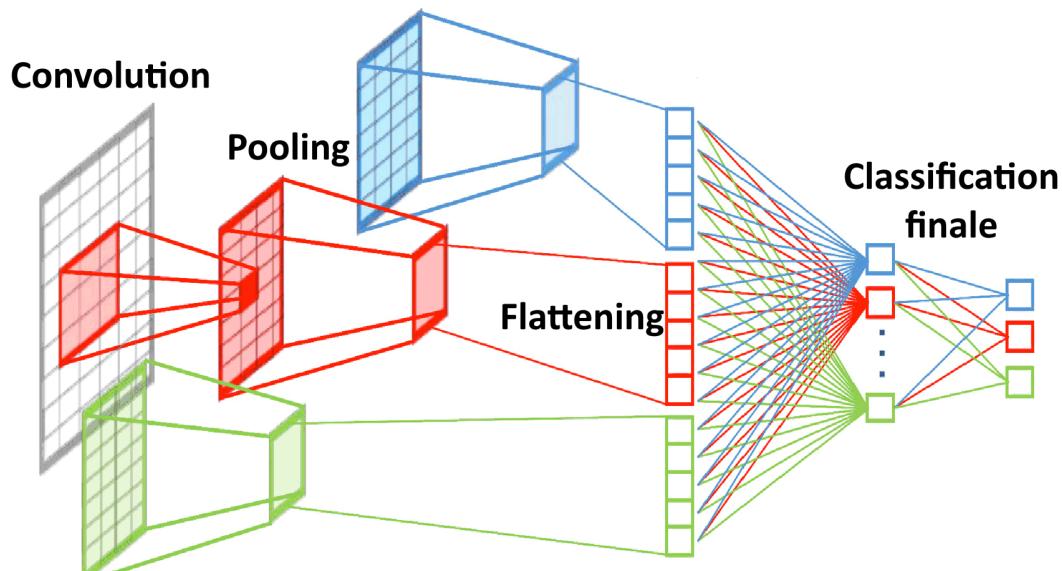


FIGURE 2.4 – Schéma d'un réseau de neurones convolutifs dont les différentes étapes sont détaillées

Image d'origine : Jean-Claude Heudin, Comprendre le deep learning.

2.2.4 Rappels sur la descente de Gradient

Afin que l'algorithme soit capable de s'améliorer tout seul au passage chaque image, celui-ci doit pouvoir se corriger. Cette correction peut être plus ou moins forte. La "force" de cette correction est appelée le pas, ou en anglais "Learning Rate". Ce pas définit à quel point l'algorithme va se corriger fortement lorsque il doit s'adapter à chaque image. Par corriger on entend modifier le poids des différents neurones. L'objectif ici est d'atteindre le taux d'erreurs, appelé erreur quadratique moyenne, le plus bas. Cette erreur quadratique moyenne peut être représentée comme une fonction quadratique 2.5. Le but ici est de se rapprocher du minimum de la fonction, plus on en sera proche, plus l'algorithme sera performant (comprendre ici, aura un meilleur taux de bonne réponse). Le pas définit donc de combien on avance sur l'axe x à chaque correction.

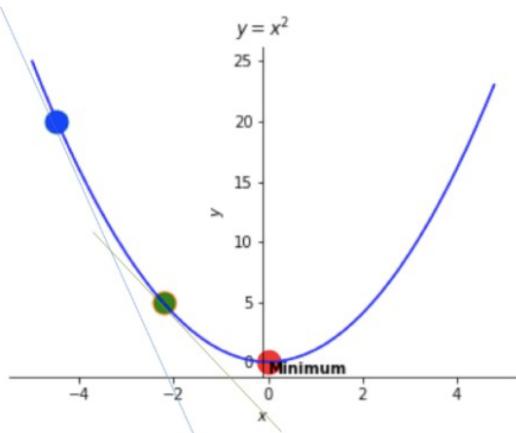


FIGURE 2.5 – Graphique de la Descente de Gradiant, on voit ici indiqué le pas qui correspond à la correction appliquée.

Pour simplifier on a ici choisi de ne montrer que la version quadratique "simple" de la descente de gradient. Pour plus de précisions on pourrait plutôt, dans certain cas, la comparer à un fonction en trois dimensions.

On comprend ainsi bien qu'un pas très grand permet de se rapprocher rapidement du minimum, en revanche, il sera difficile de l'atteindre précisément (car on risque de le dépasser vite).

A l'inverse, un très petit pas permettra de trouver une valeur de x très proche du minimum de la fonction, cependant le temps d'apprentissage en sera décuplé car les corrections seront très minimales (donc très précise).

2.3 Présentation des données

Nous avons fait trois modèles de classification : un pour les éléphants, un pour les tigres et enfin un pour les renards. Pour nos différentes expériences sur nos modèles, nous avons utilisé différents jeux de données. Pour chaque modèle, nous avons un jeu de 200 images dans lequel se trouve différents animaux.

Pour chaque modèle, nous avons comme jeu de données des images de l'animal du modèle, classées dans la classe 0, et des images d'autres animaux pour l'apprentissage, classées dans la classe 1 (voir Figure 2.6). Nos images n'ont pas forcément les mêmes dimensions comme le montre la Figure 2.9 donc la première étape est de mettre toutes les images de la même taille, en 124 pixels par 124 pixels (voir Figure 2.10). Elles sont donc à présent prêtes à être utilisées par notre modèle (voir Figures 2.6, 2.7 et 2.8).

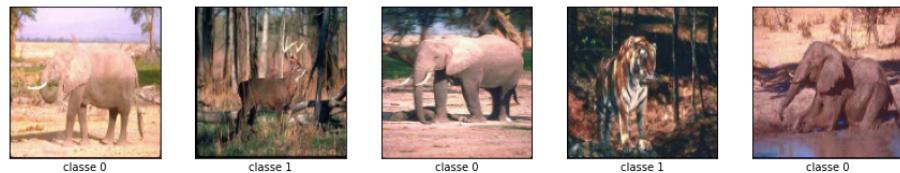


FIGURE 2.6 – Jeu de données pour le modèle Éléphant



FIGURE 2.7 – Jeu de données pour le modèle Tigre



FIGURE 2.8 – Jeu de données pour le modèle Renard



FIGURE 2.9 – Images d’éléphants



FIGURE 2.10 – Images d’éléphants après modification

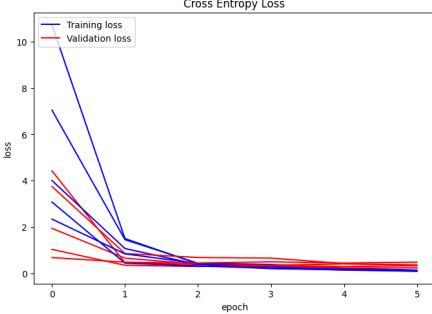
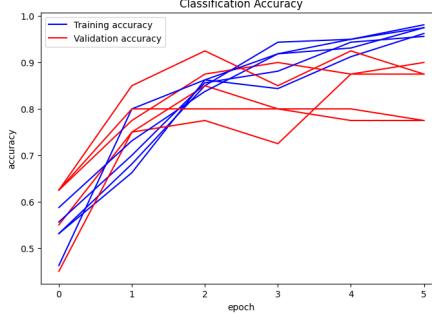
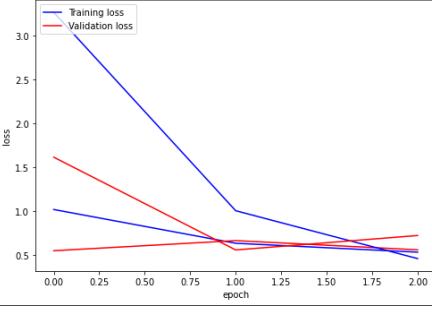
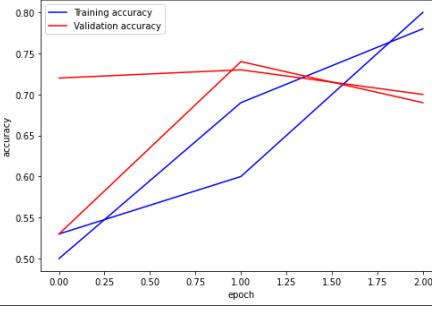
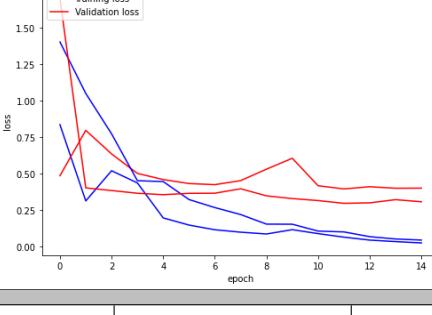
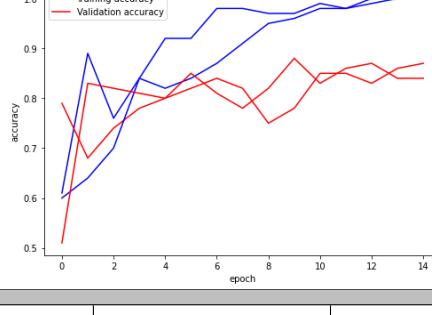
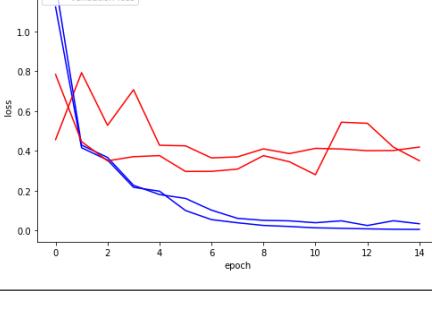
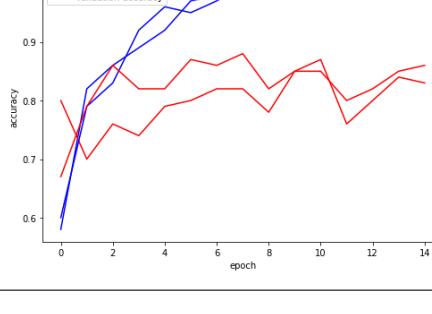
2.4 Premier modèle et résultats

Le premier modèle est le modèle baseline dont la structure reste très simple comme le montre la Figure 2.11.

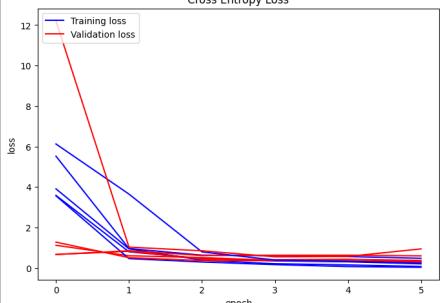
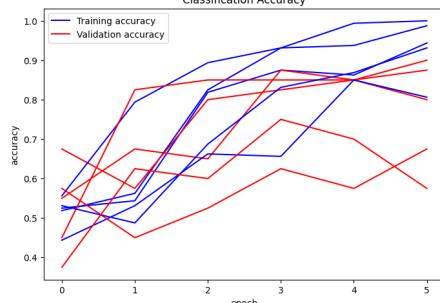
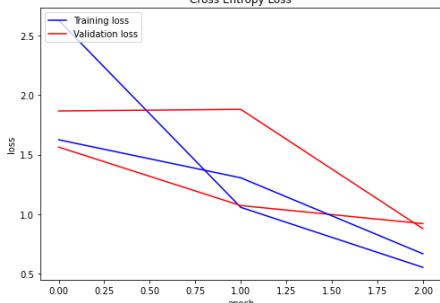
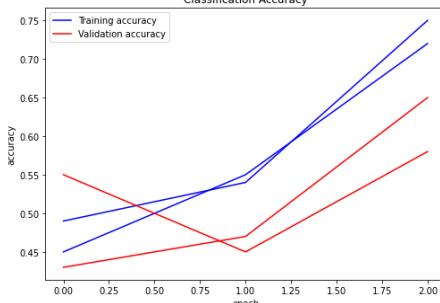
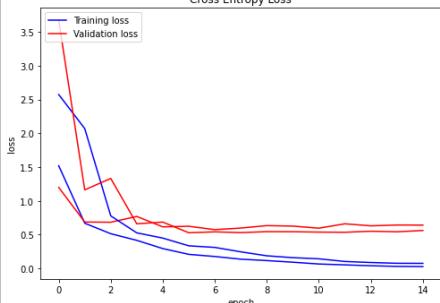
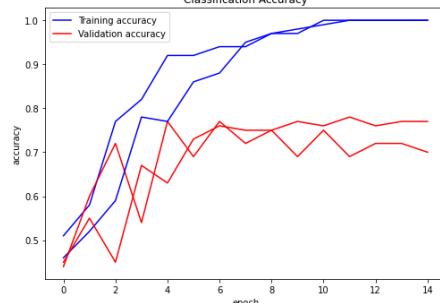
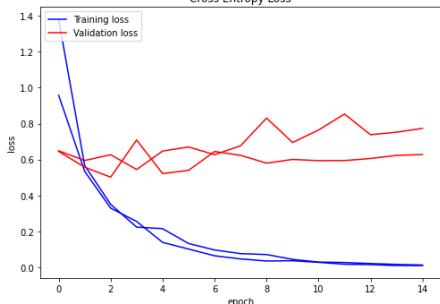
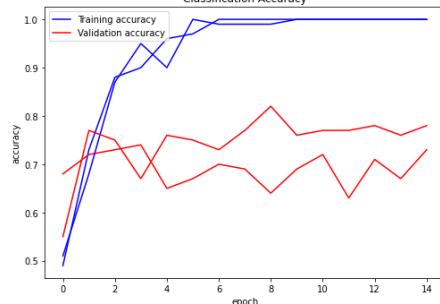
Layer (type)	Output Shape	Param #
<hr/>		
Conv2D_1 (Conv2D)	(None, 124, 124, 32)	416
Maxpooling2D_1 (MaxPooling2D)	(None, 62, 62, 32)	0
flatten (Flatten)	(None, 123008)	0
dense_54 (Dense)	(None, 1)	123009
<hr/>		
Total params: 123,425		
Trainable params: 123,425		
Non-trainable params: 0		

FIGURE 2.11 – Model Baseline

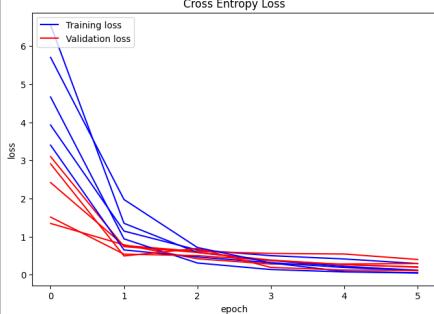
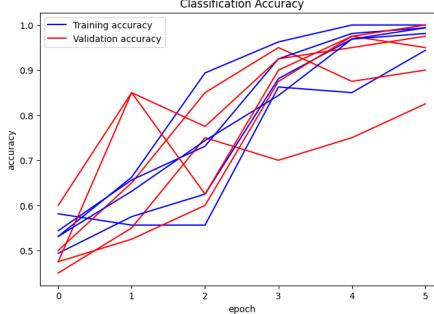
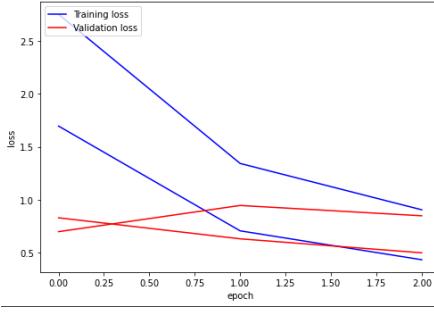
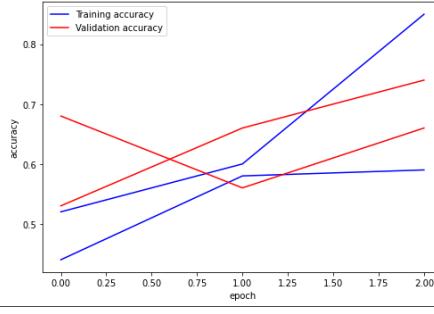
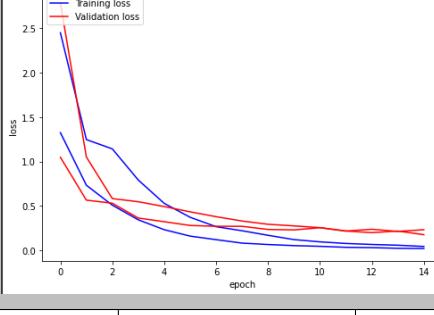
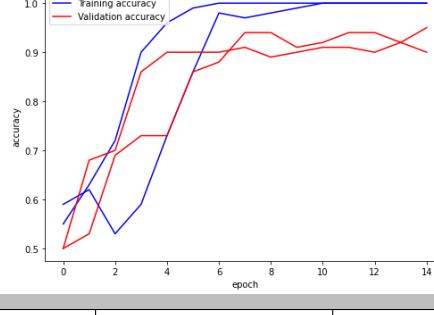
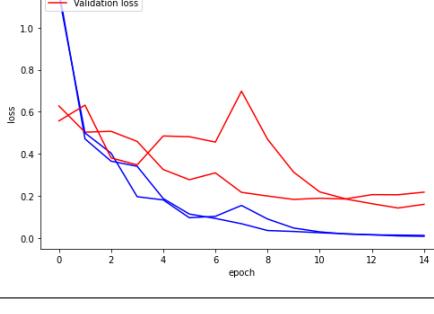
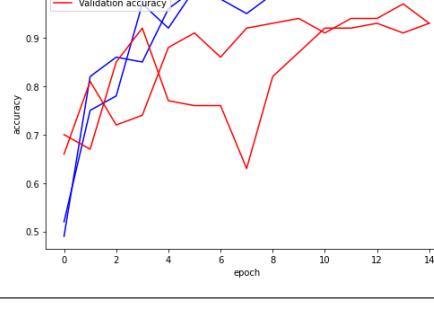
2.4.1 Résultats modèle éléphant

fold	epoch	batchsize	accuracy	écart-type
5	6	16	84.000%	5.385
				
2	3	16	69.000%	0.500
				
2	15	16	85.500%	1.500
				
2	15	8	84.500%	1.500
				

2.4.2 Résultat modèle tigre

fold	epoch	batchsize	accuracy	écart-type
5	6	16	76.500%	12.309
				
2	3	16	61.500%	3.500
				
2	15	16	73.500%	3.500
				
2	15	8	75.500%	2.500
				

2.4.3 Résultat modèle renard

fold	epoch	batchsize	accuracy	écart-type
5	6	16	93.000%	6.205
				
2	3	16	70.000%	4.000
				
2	15	16	92.500%	2.500
				
2	15	8	93.000%	0.000
				

2.4.4 Explication des résultats

Dans un premier temps, nous avons diminué le "fold" afin d'avoir un graphe beaucoup plus clair pour mieux interpréter les résultats. Le fold est maintenant de 2, on obtiendra donc deux courbes qui représentent l'apprentissage du modèle après remise à zéro des poids.

On décide alors de modifier le paramètre "epoch". L'epoch est le paramètre qui donne le nombre de fois que le modèle va apprendre sur les données. On s'attend donc à avoir une précision bien supérieure, sur les trois modèles, on l'augmente donc de 3 à 15. On remarque une différence de 12 à 22% suivant les modèles. Cette différence est principalement liée aux données avec ce changement. En effet, ce changement est très normal, car on laisse plus de temps au modèle pour apprendre à partir de nos données, ce qui a pour effet d'avoir une meilleure connaissance. Cependant, cet effet a un contre coup majeur, le sur-apprentissage des données. En effet, sur le graphe, on peut voir un sur-apprentissage à partir du 4e epoch, la courbe d'erreur continue de baisser pour le jeu d'entraînement, mais stagne pour le jeu de validation, ce qui traduit une forme de sur-apprentissage du modèle sur le jeu de données.

De plus, nous remarquons une inégalité dans les résultats de nos différents modèles. Il semble que le jeu de données des renards est plus pertinent que les deux autres jeux de données puisque nous obtenons des valeurs très intéressantes en ce qui concerne l'accuracy. Ce comportement s'explique par la différence des jeux de données entre les modèles. En effet, le renard obtient des résultats bien meilleurs que les autres, car toutes les images de renard que nous avons, comportent uniquement un renard bien en évidence. Alors que dans le cas des éléphants, on retrouve quelques images en groupe (voir Figure 2.12). En ce qui concerne le tigre, comme son habitat étant les forêts tropicales, les mangroves, les territoires fortement boisés et les marécages, dans plusieurs images, nous avons des tigres cachés par la forêt ou dans les marécages (voir Figure 2.13).



FIGURE 2.12 – Exemple d’images d’éléphant



FIGURE 2.13 – Exemple d’images de tigre

2.5 Des extensions au modèle de base

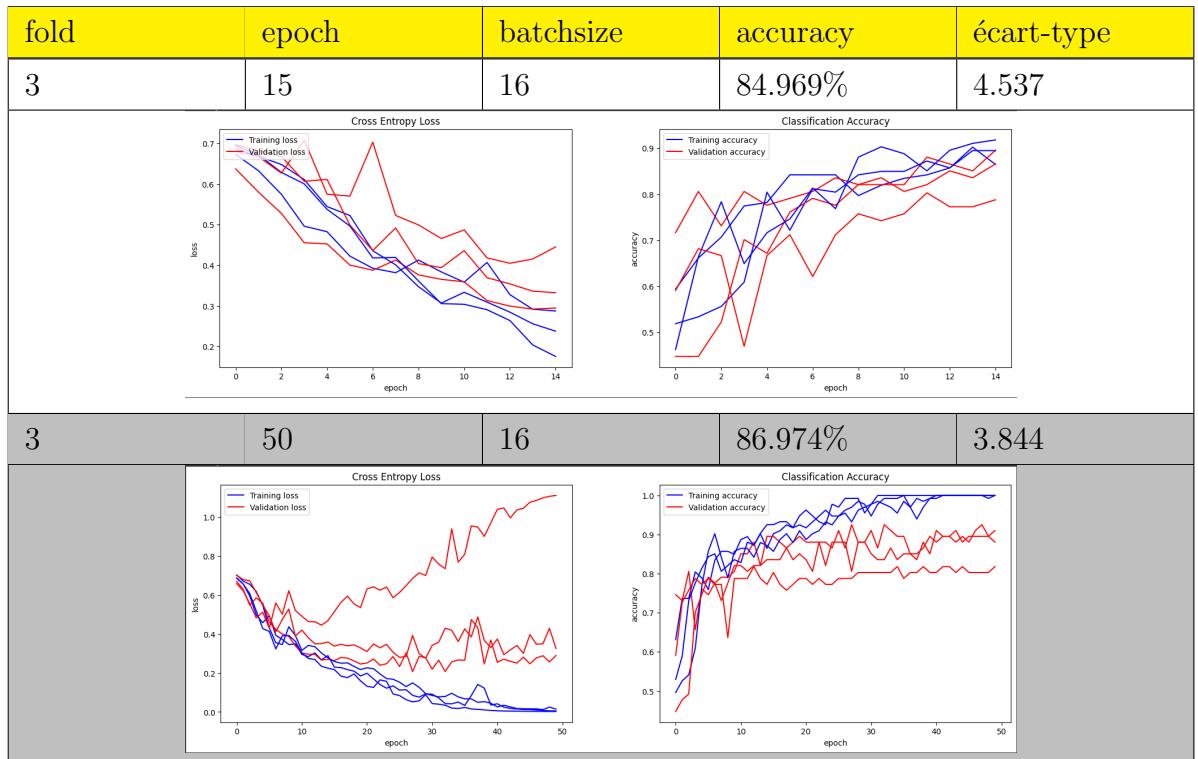
2.5.1 Structure du modèle

Ce modèle est quelque peu différent du modèle baseline présenté précédemment. En effet, ce modèle comporte quatre couches de convolutions où après chaque couche suit un "MaxPooling" comme le montre la figure 2.14.

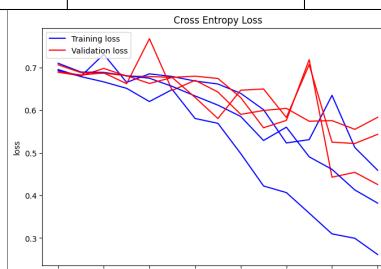
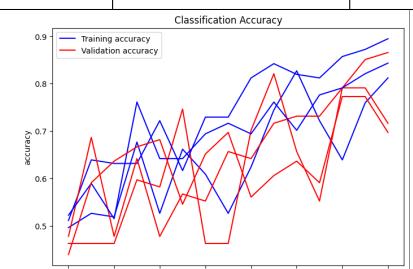
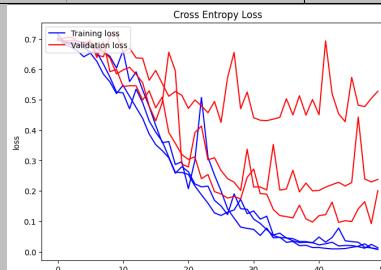
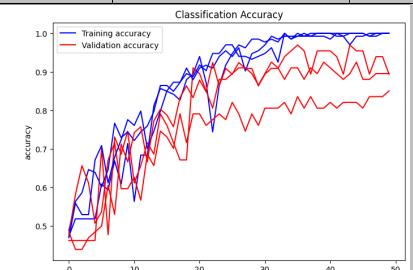
Layer (type)	Output Shape	Param #
Conv2D_1 (Conv2D)	(None, 124, 124, 32)	416
Maxpooling2D_1 (MaxPooling2D)	(None, 62, 62, 32)	0
Conv2D_2 (Conv2D)	(None, 62, 62, 32)	4128
Maxpooling2D_2 (MaxPooling2D)	(None, 31, 31, 32)	0
Conv2D_3 (Conv2D)	(None, 31, 31, 32)	4128
Maxpooling2D_3 (MaxPooling2D)	(None, 15, 15, 32)	0
Conv2D_4 (Conv2D)	(None, 15, 15, 32)	4128
Maxpooling2D_4 (MaxPooling2D)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0
dense_155 (Dense)	(None, 1)	1569
<hr/>		
Total params: 14,369		
Trainable params: 14,369		

FIGURE 2.14 – Modèle baseline avec 4 couches

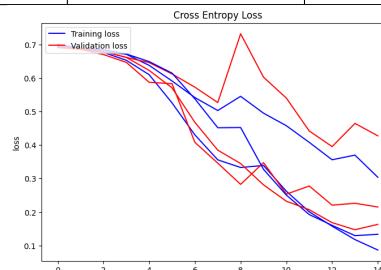
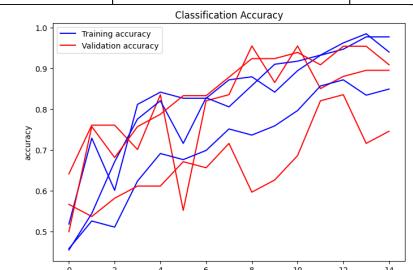
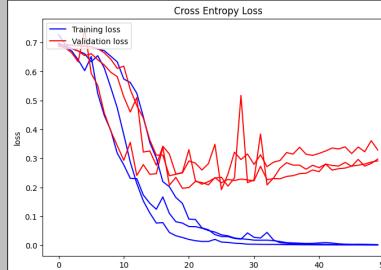
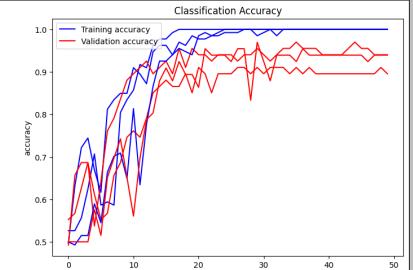
2.5.2 Modèle plus complexe d'éléphant



2.5.3 Modèle plus complexe de tigre

fold	epoch	batchsize	accuracy	écart-type
3	15	16	75.969%	7.536
			 	
3	15	16	88.007%	2.074
			 	

2.5.4 Modèle plus complexe de renard

fold	epoch	batchsize	accuracy	écart-type
3	15	16	85.029%	7.377
			 	
3	50	16	92.507%	2.090
			 	

2.5.5 Résultats

En comparant chaque modèle baseline au modèle plus complexe, on remarque que la complexité de sa structure influence sur sa capacité à apprendre. En effet, on peut voir à travers les courbes que le nouveau modèle tarde le sur-apprentissage. Sur les modèles de base, on constate un sur-apprentissage à partir de la 3e epoch alors que sur le modèle plus complexe, celle-ci a lieu environ vers la 10e epoch suivant les modèles. La différence d'epoch entre les modèles s'explique par la difficulté qu'a le modèle à apprendre les données, donc la qualité de l'image ou encore le contexte de l'image influent sur la complexité d'apprentissage d'un modèle.

2.6 Conclusion

Pour conclure, on comprend que peu importe le modèle, il aura toujours du sur-apprentissage, même si l'on rend le modèle plus complexe, cependant celui-ci le retarde. Il nous suffirait ainsi de ne pas apprendre tout le jeu de données par coeur. Une solution serait d'augmenter le nombre d'images de notre jeu de données.

Augmenter le nombre d'images

3.1 Rappel sur la fonction ImageDataGenerator

Pour augmenter notre jeu de données, nous allons utiliser la fonction `ImageDataGenerator` qui permet de générer de nouvelles images en appliquant des transformations sur les images de notre jeu de données (des rotations, des décalages, des zooms, etc).

3.2 Résultats obtenus



FIGURE 3.1 – Image générée d'un chat (zoom x1.5 avec retournement horizontal



FIGURE 3.2 – Image générée d'un tigre (retournement horizontal)



FIGURE 3.3 – Image générée d'un renard (retournement horizontal et vertical)

On voit que les images sont assez différentes, cependant on remarque que l'on perd de l'information, dans ce cas, on voit que la couleur est différente de l'image de base (voir les Figures 3.1, 3.2 et 3.3).

3.3 Conclusion

Grâce à la fonction `ImageDataGenerator`, il est possible d'augmenter le jeu de données en créant de nouvelles images à partir du modèle initial. Cela permet d'améliorer les performances du modèle en évitant le sur-apprentissage grâce à l'utilisation de transformations sur les images comme la rotation, le zoom ou le décalage. Cependant, il faut faire attention, car l'image peut être déformée après le passage de `ImageDataGenerator` ce qui peut entraîner un mauvais apprentissage du modèle.

Vers des modèles plus complexes

4.1 Rappel sur les modèles plus complexes

Il existe plusieurs modèles plus complexes, qui ont été conçus pour la reconnaissance d'image, l'un des plus connus est le modèle ResNet (Residual Network ou réseau neuronal résiduel en français). Ce modèle est un CNN, son architecture a été conçue pour supporter des centaines, voire des milliers de couches de convolution pour traiter de nombreux paramètres (voir Figure 4.1).

Layer (type)	Output Shape	Param #
<hr/>		
resnet50 (Functional)	(None, 2048)	23587712
dense_12 (Dense)	(None, 512)	1049088
dense_13 (Dense)	(None, 10)	5130
<hr/>		
Total params: 24,641,930		
Trainable params: 24,588,810		
Non-trainable params: 53,120		

FIGURE 4.1 – Image de la structure de ResNet50

4.1.1 Réseau neuronal résiduel

Un réseau neuronal résiduel est un réseau de neurones qui comporte des centaines de couches, mais à la différence des autres réseaux de neurones de convolution vu jusqu'à maintenant, celui-ci peut sauter plusieurs couches en même temps, on appelle ce phénomène "sauts de connexion" ou "raccourcis" (voir Figure 4.2), ceci est possible grâce à des neurones très spéciaux appelés "les neurones amonts". Une matrice supplémentaire est nécessaire pour ces neurones.

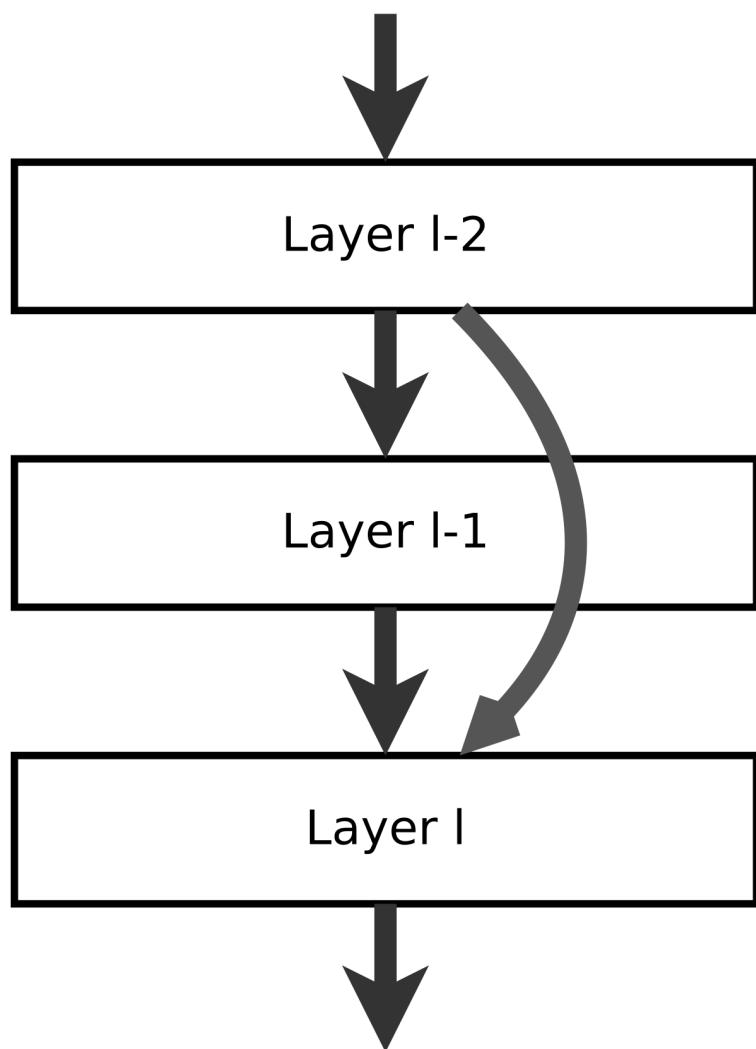


FIGURE 4.2 – Schéma d'un saut de connexion (source wikipedia)

4.2 Application de Resnet

On peut voir que le modèle est très bon, car sur le jeu de validation et le jeu d'apprentissage le modèle affiche des résultats assez bon, 70% (Voir Figure 4.3 et 4.4).

```
Epoch 1/10
118/118 [=====] - 45s 378ms/step - loss: 0.8085 - accuracy: 0.7030 - val_loss: 0.7582 - val_accuracy: 0.7116
Epoch 2/10
118/118 [=====] - 39s 334ms/step - loss: 0.7919 - accuracy: 0.7074 - val_loss: 0.7401 - val_accuracy: 0.7184
Epoch 3/10
118/118 [=====] - 37s 310ms/step - loss: 0.7894 - accuracy: 0.7095 - val_loss: 0.7322 - val_accuracy: 0.7171
Epoch 4/10
118/118 [=====] - 37s 314ms/step - loss: 0.7892 - accuracy: 0.7060 - val_loss: 0.7424 - val_accuracy: 0.7214
Epoch 5/10
118/118 [=====] - 39s 330ms/step - loss: 0.7857 - accuracy: 0.7094 - val_loss: 0.7123 - val_accuracy: 0.7302
Epoch 6/10
118/118 [=====] - 38s 320ms/step - loss: 0.7754 - accuracy: 0.7133 - val_loss: 0.7239 - val_accuracy: 0.7216
Epoch 7/10
118/118 [=====] - 39s 329ms/step - loss: 0.7637 - accuracy: 0.7174 - val_loss: 0.7240 - val_accuracy: 0.7154
Epoch 8/10
118/118 [=====] - 38s 321ms/step - loss: 0.7627 - accuracy: 0.7166 - val_loss: 0.7244 - val_accuracy: 0.7258
Epoch 9/10
118/118 [=====] - 37s 316ms/step - loss: 0.7588 - accuracy: 0.7191 - val_loss: 0.7032 - val_accuracy: 0.7375
Epoch 10/10
118/118 [=====] - 37s 314ms/step - loss: 0.7508 - accuracy: 0.7217 - val_loss: 0.7121 - val_accuracy: 0.7363
```

FIGURE 4.3 – Image des epochs lors de l'apprentissage

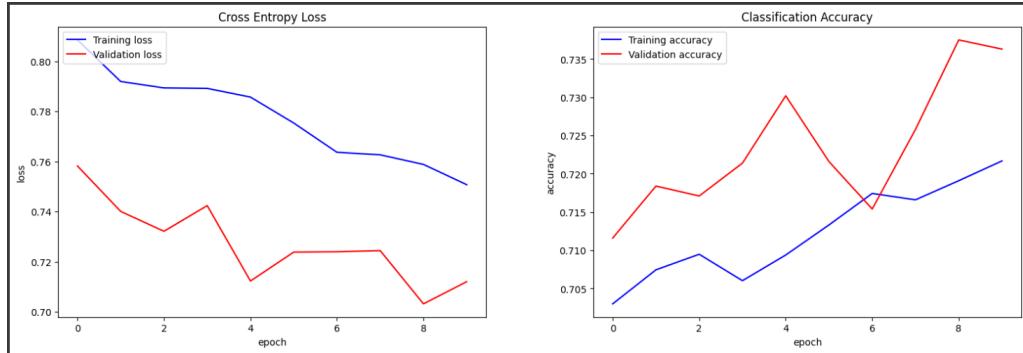


FIGURE 4.4 – Courbe d'apprentissage du modèle ResNet50

4.3 Conclusion

Pour conclure, on a pu voir que d'autres modèles plus complexes existent, ils comportent bien plus de couche que les modèles que nous avions utilisés jusqu'à maintenant (modèle baseline). Ils ont des comportements bien plus complexes, avec notamment dans le cas du modèle ResNet, des sauts de connexion pour sauter des couches.

Auto-encodeurs

5.1 Introduction

Les encodeurs automatiques peuvent être utilisés pour augmenter le jeu de données en générant de nouvelles images à partir d'autres images. Cet algorithme non supervisé est utile lorsque les ensembles de données sont limités ou coûteux à collecter, ou pour améliorer les performances des modèles d'apprentissage automatique en fournissant davantage de données d'apprentissage. Les auto-encodeurs peuvent également être utilisés pour effectuer des transformations d'images, comme restaurer des images dégradées ou transformer une image en une autre, comme c'est le cas avec les GANs (Generative Adversarial Networks).

5.2 Réduction de données

L'encodage et le décodage d'images à l'aide d'un auto-encodeur réduisent la dimensionnalité de celles-ci. En pratique, un encodeur prend une image d'entrée et la convertit en une représentation latente de taille réduite (ou vecteur de caractéristiques) qui capture les informations importantes sur l'image. Le décodeur prend alors cette représentation latente et essaie de la reconstruire en une image similaire à l'entrée d'origine.

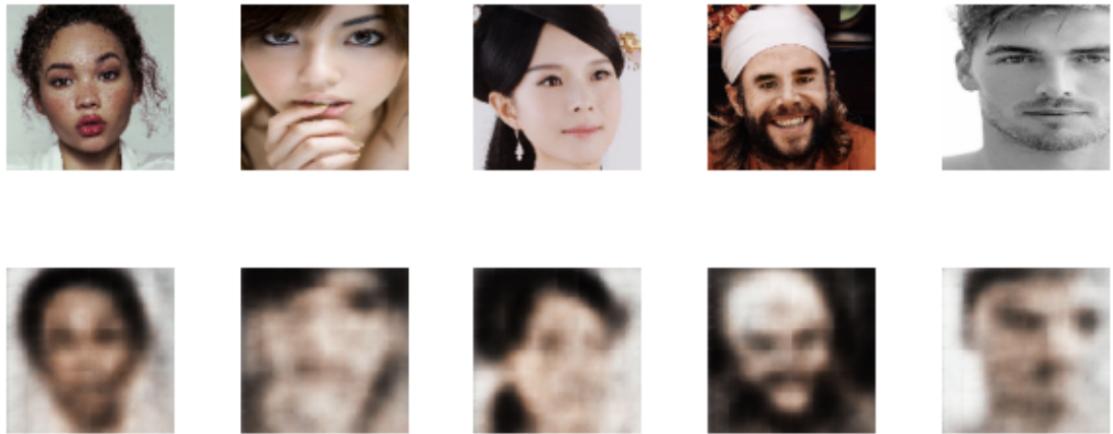


FIGURE 5.1 – Images avant et après réduction par notre auto-encodeur

Dans un premier temps, nos images ont été encodées, c'est-à-dire qu'elles ont été représentées par une série de nombres (un vecteur) que l'on appelle "latent". Puis, elles ont été régénérées sous la forme d'une image couleur. Le résultat de ce processus nous donne une nouvelle image qui ne contient que les caractéristiques importantes de l'image mère. C'est pourquoi nous remarquons sur la Figure 5.1 un aspect "flou" à nos images résultantes.

Les auto-encodeurs sont couramment utilisés pour la compression de données dans l'apprentissage automatique. La compression d'images est l'un des exemples les plus courants d'utilisation d'auto-encodeurs. En fait, les auto-encodeurs peuvent être entraînés pour encoder des images haute résolution en vecteurs dimensionnels plus petits qui représentent l'essence de l'image. Cette compression est utile pour stocker des images avec moins d'espace de stockage ou pour accélérer le calcul du modèle et le processus de formation.

5.3 L'utilisation des auto-encodeurs pour débruiter

Une seconde utilisation des auto-encodeurs peut être pour débruiter une image. Une image bruitée est représentée par du bruit numérique. Nous allons nous pencher sur trois types de bruits différents :

- Une image couleur qui devient blanche et noire
- Une image qui contient des trous (représentés par un rectangle noir)
- Une image qui contient du bruit numérique



FIGURE 5.2 – Image avant le bruitage numérique



FIGURE 5.3 – Images après différents bruitages ('blackwhite', 'mask', 'noise')

L'objectif maintenant est de débruiter ces images dans le but de retrouver l'image originale. Pour ce faire, nous utilisons une nouvelle fois les auto-encodeurs.

Nous entraînons l'auto-encodeur qui utilise un jeu de données comprenant des paires d'images bruitées et non bruitées. Celui-ci doit générer un modèle pour minimiser la différence entre l'image décompressée et l'image originale.

Avec un jeu de données de 20 images et 20 epochs, nous obtenons des résultats tout à fait intéressants puisque nous atteignons près de 80% de précision (voir Figure 5.4).

```
Epoch 16: saving model to ./myweights/AEweights_corrupted.h5
36/36 [=====] - 44s 1s/step - loss: 0.0056 - accuracy: 0.7573 - val_loss: 0.0069 - val_accuracy: 0.8045
Epoch 17/20
36/36 [=====] - ETA: 0s - loss: 0.0057 - accuracy: 0.7614
Epoch 17: saving model to ./myweights/AEweights_corrupted.h5
36/36 [=====] - 46s 1s/step - loss: 0.0057 - accuracy: 0.7614 - val_loss: 0.0064 - val_accuracy: 0.7835
Epoch 18/20
36/36 [=====] - ETA: 0s - loss: 0.0058 - accuracy: 0.7577
Epoch 18: saving model to ./myweights/AEweights_corrupted.h5
36/36 [=====] - 44s 1s/step - loss: 0.0058 - accuracy: 0.7577 - val_loss: 0.0059 - val_accuracy: 0.7912
Epoch 19/20
36/36 [=====] - ETA: 0s - loss: 0.0057 - accuracy: 0.7600
Epoch 19: saving model to ./myweights/AEweights_corrupted.h5
36/36 [=====] - 44s 1s/step - loss: 0.0057 - accuracy: 0.7600 - val_loss: 0.0057 - val_accuracy: 0.7979
Epoch 20/20
36/36 [=====] - ETA: 0s - loss: 0.0056 - accuracy: 0.7637
Epoch 20: saving model to ./myweights/AEweights_corrupted.h5
36/36 [=====] - 44s 1s/step - loss: 0.0056 - accuracy: 0.7637 - val_loss: 0.0057 - val_accuracy: 0.8015
```

FIGURE 5.4 – Image des epochs lors de l'apprentissage de notre modèle de coloration

Utilisons notre image 5.3 en blanc et noir, parvient-on à retrouver l'image couleur initiale de notre Figure 5.2.



FIGURE 5.5 – Images en niveau de gris et générée par notre modèle de coloration

L'image couleur générée à la Figure 5.5 ne semble pas très nette, cependant nous remarquons que le panel de couleur est respecté ! Peut-être obtiendrons-nous une meilleure image si l'on entraînait notre auto-encodeur avec un jeu de données plus conséquent.

5.4 Conclusion

Pour conclure cette section, nous avons vu que les auto-encodeurs pouvaient être utilisés dans la réduction de données (en des vecteurs latents) et pour débruiter des images. La décoloration d'une image couleur est assez simple, mais sa coloration n'est pas intuitive. Il nous faut passer par une intelligence artificielle qui, une fois entraînée sur un grand nombre d'images, peut ainsi essayer de retrouver les couleurs originelles depuis une image en niveau de gris.

De même, le débruitage d'images peut être effectué efficacement à l'aide d'auto-encodeurs, bien que cela puisse être plus complexe en fonction du type de bruit présent dans l'image. En entraînant un auto-encodeur sur un ensemble de données bruitées et non bruitées, il apprend à générer des images débruitées aussi proches que possible de l'image d'origine. Par conséquent, les auto-encodeurs offrent une solution intéressante pour améliorer la qualité des images et faciliter l'analyse des données dans divers domaines.

En somme, les auto-encodeurs sont une technique d'apprentissage automatique puissante qui peut être appliquée dans de nombreux domaines. Ils offrent des possibilités de compression de données, de réduction de dimensionnalité, de génération d'images, de détection d'anomalies et de débruitage d'images. En continuant à explorer les capacités des auto-encodeurs, nous pourrons découvrir de nouvelles façons d'améliorer la compréhension et l'utilisation des données dans divers domaines de la science et de la technologie.

Génération d'images

6.1 Introduction

Les auto-encodeurs peuvent également être utilisés pour générer des images ou détecter des anomalies. Les auto-encodeurs peuvent être chargés de générer de nouvelles images à partir de vecteurs latents aléatoires, ce qui permet la création d'images qui n'étaient pas présentes dans le jeu de données d'origine. De même, les auto-encodeurs peuvent être chargés de détecter les défauts dans les données en comparant la reconstruction d'une partie des données avec l'original. Si la reconstruction est significativement différente de l'original, cela peut indiquer la présence d'un problème avec les données.

6.2 Réseaux antagonistes génératifs (GANs)

Les réseaux antagonistes génératifs ou GANs (Generative Adversarial Network) sont des modèles de génération de données à partir d'une base existante. Ils sont composés de deux réseaux de neurones : un générateur et un discriminant. Le générateur crée de nouvelles données qui sont similaires aux données de la base. Le discriminant évalue la qualité des données pour voir si les données sont originelles ou si elles sont créées par le générateur. Le but des GANs est d'améliorer la reconnaissance du discriminant et de perfectionner le générateur pour avoir des résultats plus réalistes dans la base de données. Les erreurs de reconnaissances du discriminant sont aussi appliquées au générateur pour l'aider à s'améliorer. De plus, il faut faire attention à ce que le discriminant et le générateur aient des performances équivalentes pour un meilleur apprentissage. Cependant, un GAN est un modèle qui peut engendrer des défaillances comme le mode "collapse" ou la disparition du gradient (voir bibliographie : [2]).

6.3 Ouverture sur la génération d'image

Les réseaux antagonistes pourraient nous permettre de générer de nouvelles images semblables aux données présentes dans notre modèle initial, améliorant ainsi notre base d'apprentissage pour permettre une meilleure reconnaissance du sujet du modèle.

Organisation du projet

Tout au long de ce semestre, nous avons suivi des cours en visio-conférence avec notre responsable M. Poncelet. Ces cours nous ont permis d'apprendre les notions essentielles pour réaliser ce projet, ainsi que l'utilisation des Notebooks contenant les bases de code du projet.

Pour ce qui est de la répartition des tâches, nous nous sommes partagés le travail au mieux entre le travail sur le code et l'écriture du rapport.

Nous avons commencé par travailler sur la partie Classification des images avec en premier lieu le modèle baseline. Tous les membres du groupe ont participé à la création du modèle baseline pour que la base du projet soit acquise pour tous. Romain s'est en plus chargé de commencer le rapport en mettant au point tous les rappels de notions nécessaires à la compréhension du lecteur.

Morgan et Eric ont ensuite étendu le modèle de base pendant que Romain et Lisa rédigeaient la partie sur les CNN. Eric aidait également à la rédaction de la partie CNN du rapport.

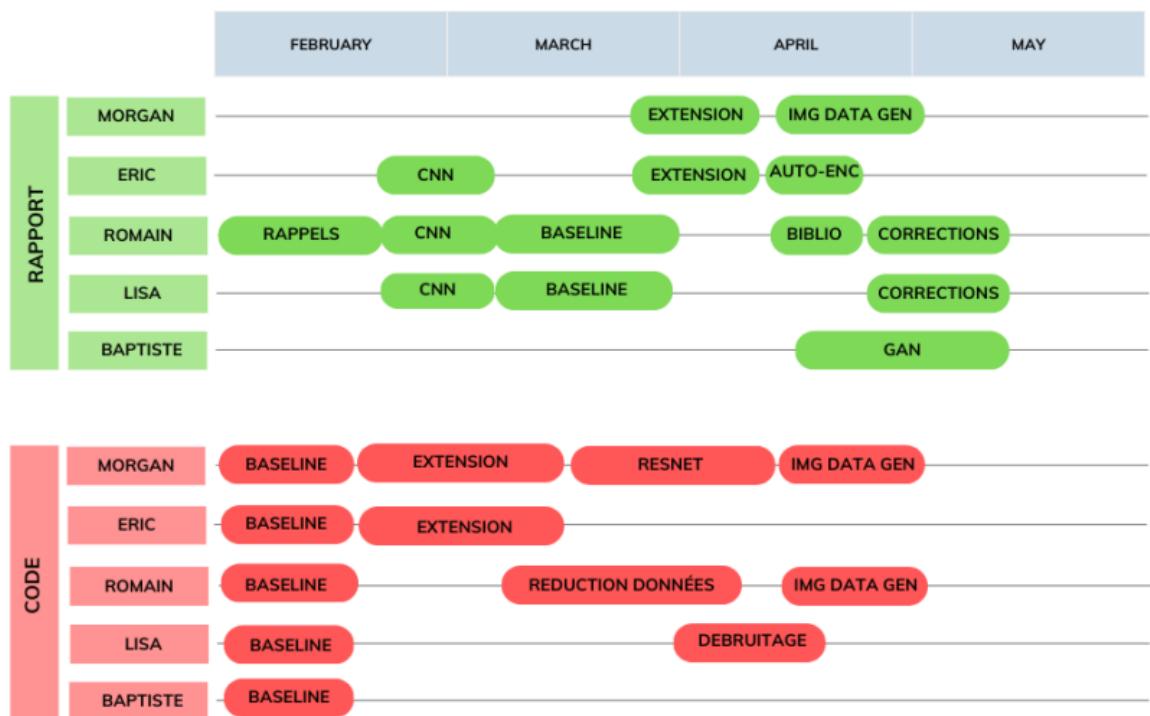
Morgan a, à partir de ce moment là, travaillé principalement sur le code et a aidé tous le monde même s'il a participé principalement au travail sur la partie CNN (ResNet) et extension du jeu de données pour le coté code.

Côté rapport, Lisa et Romain ont écrit la partie du modèle baseline. Romain a également codé la réduction des données sur la partie auto-encodeurs. Lisa travaillera de son côté sur le côté débruitage à l'aide des auto-encodeurs.

Eric et Morgan enchaîneront sur le rapport avec l'explication du code sur l'extension du modèle de base.

Enfin, niveau code, Morgan et Romain finiront le code avec l'extension du jeu de données à l'aide la fonction ImageDataGenerator. Niveau rapport, Morgan s'occupera de l'explication du fonctionnement de l'ImageDataGenerator. Baptiste travaillera sur les GANs même si nous n'aurons pas le temps de les implémenter, Eric sur la partie auto-encodeurs, Romain sur la bibliographie et Lisa s'occupera de la majorité des corrections et retouches, aidée par Romain.

HAI606I : PROJET DE PROGRAMMATION 2 DIAGRAMME DE PLANNIFICATION (MIS À JOUR)



Conclusion

Au cours de ce second semestre de projet, nous avons pu mettre en application les différents concepts et différentes connaissances acquises sur l'IA et sur les techniques utilisées pour la reconnaissance d'images, notamment les réseaux de neurones convolutifs que nous avons longuement explorés. Nous avons pu mettre en place plusieurs réseaux de neurones capables de reconnaître des animaux avec un bon taux de réussite. Pour améliorer encore davantage nos réseaux nous avons du étendre notre modèle de base en le complexifiant. Nous avons également appris à générer de nouvelles images pour notre jeu de données en modifiant les images déjà existante : leur taille, leur forme, leur rotation etc. Tout cela nous a permis d'améliorer encore notre taux de reconnaissance et d'atteindre des taux de précision $> 92\%$. L'entièreté du projet comporte, pour la partie modèle Baseline 624 lignes de code, le modèle complexifié avec CNN en comporte 1417, soit un total de 2041 lignes de codes. Il est cependant à noter que le code a été largement commenté et détaillé directement dans les fichiers sources.

Une poursuite possible de notre travail serait d'utiliser les connaissances et les données préalablement acquises pour générer de toutes nouvelles images. Nous avons pour cela déjà étudié les techniques de génération d'images, notamment les réseaux antagonistes génératifs (GANs).

Bibliographie

- [1] Mauro Castelli, Leonardo Vanneschi, and Álvaro Rubio Largo. Supervised learning : classification. *por Ranganathan, S., M. Grisbskov, K. Nakai y C. Schönbach*, 1 :342–349, 2018.
- [2] Haiyang Chen. Challenges and corresponding solutions of generative adversarial networks (gans) : a survey study. In *Journal of Physics : Conference Series*, volume 1827, page 012066. IOP Publishing, 2021.
- [3] Neha Gupta et al. Artificial neural network. *Network and Complex Systems*, 3(1) :24–28, 2013.
- [4] Ying-jie Liang, Xiao-peng Cui, Xing-hua Xu, and Feng Jiang. A review on deep learning techniques applied to object detection. In *2020 7th International Conference on Information Science and Control Engineering (ICISCE)*, pages 120–124. IEEE, 2020.
- [5] Keiron O’Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv :1511.08458*, 2015.
- [6] Pascal Poncelet. Les réseaux de neurones / descente de gradient / les réseaux de neurones convolutifs.
- [7] Ramalingam Saravanan and Pothula Sujatha. A state of art techniques on machine learning algorithms : a perspective of supervised learning approaches in data classification. In *2018 Second international conference on intelligent computing and control systems (ICICCS)*, pages 945–949. IEEE, 2018.
- [8] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1) :1–48, 2019.
- [9] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, et al. Deep learning for computer vision : A brief review. *Computational intelligence and neuroscience*, 2018, 2018.