



DataScientest • com

Rapport final - 16 juillet 2021

MushPy - Mushroom Recognition



Parcours Bootcamp - Promo mai 2021 - Data Scientist

Github projet: <https://github.com/DataScientest/mushpy>

Sevil CARON

Adrien MOREAU

Fernando GONÇALVES

Thibault KACZMAREK

Sommaire

Sommaire	3
Contexte	5
Data	5
Cadre	5
Exploration des données	6
Nettoyage des données	10
Caractéristiques des images	12
Mesure de la luminosité	14
Fichiers d'intérêt :	14
Visualisation des données	15
Visualisation des familles à l'oeil humain	15
Réduction de dimensions	19
Isomap	19
PCA	21
Modélisation	22
Classification du problème	22
Test des différents modèles	22
Réduction de dimensions	22
Présentation :	22
Résultats obtenus :	23
Fichiers d'intérêt :	23
LeNet	25
Présentation:	25
Résultats obtenus :	27
Fichiers d'intérêt :	27
Modélisation par “transfer learning”	28
Schéma général utilisé :	29
Identification d'un modèle propice	31
VGG16	31
Présentation :	31
Résultats obtenus :	32
Fichiers d'intérêt :	32
VGG19	34
Présentation :	34
Résultats obtenus :	36
Fichiers d'intérêt :	36
ResNet50	37
Présentation :	37
Résultats obtenus :	39

Fichiers d'intérêt :	39
Inceptionv3	40
Présentation :	40
Résultats obtenus :	41
Fichiers d'intérêt :	41
EfficientNetB1	42
Présentation :	42
Résultats obtenus :	43
Fichiers d'intérêt :	43
Optimisation du modèle sélectionné :	45
Fichiers d'intérêt :	45
Synthèse des résultats	52
Difficultés rencontrées lors du projet	53
Réduction du jeu de données : choix du focus (famille / genres)	53
Architecture et ressources informatiques :	58
Problématique des premiers entraînements :	59
Vie des modèles, itérations et gestion des données :	60
Sauvegarde et chargement des modèles :	60
Implémentation de l'algorithme Grad-CAM :	61
Bilan	63
Pistes d'amélioration / perspectives :	64
Jeu de données :	64
Sauvegarde/chargement du modèle :	64
Nombre de familles prédites :	64
Affiner la classification obtenue : prédire le genre	64
Première étape : génération des jeux de données :	65
Seconde étape : générer les nouveaux modèles :	67
Fichiers d'intérêt :	67
Troisième étape : réaliser les prédictions - méthode entonnoir :	70
Fichiers d'intérêt :	70
Possibles applications	73
Bibliographie	75
Articles et Tutoriels	75
Table des figures	77

Contexte

L'objectif de ce projet est de faire de la reconnaissance d'images des champignons à l'aide d'algorithmes de computer vision. Les champignons présentent un grand nombre d'espèces, de genres, de familles, ainsi qu'une grande variété de caractéristiques biologiques et nutritives. Il peut être intéressant de savoir les classifier précisément à l'aide de la computer vision.

Le rapport suivant montre le travail effectué pour atteindre cet objectif, à commencer par la collecte de données, l'exploration et la data visualisation, l'application des différents modèles et finalement les résultats obtenus.

Le nom que nous avons donné au projet est MushPy.

Data

Cadre

Pour mener ce projet à bien, nous allons utiliser le site mushroomobserver¹ qui est une base de données dédiée à l'identification des champignons. Il contient les éléments nécessaires que nous recherchons pour établir notre algorithme de classification, à savoir une grande quantité d'images auxquelles sont associées leurs informations concernant leur taxonomie (entre autres).

Lors de notre première recherche bibliographique, nous avons constaté qu'un ancien projet² avait déjà exploré ce site afin de créer un dataset d'images datant de 2006 à 2016. Il nous a servi de point de départ pour commencer notre exploration des données.

¹ Lien : <https://mushroomobserver.org/>

² Lien : <https://github.com/bechtle/mushroomobser-dataset>

Exploration des données

La première étape du traitement de nos données a été de créer un dataframe à partir des 12 fichiers au format JSON.

Chaque fichier JSON contient des milliers de documents “champignon” avec la structure suivante :

```

"observation": "/observer/show_observation/264880",
"label": "Cortinarius",
"image_id": "699257",
"image_url": "http://mushroomobserver.org/images/320/699257",
"user": "/observer/show_user/6796",
"date": "2016-12-18",
"gbif_info": {
    "status": "ACCEPTED",
    "kingdom": "Fungi",
    "usageKey": 2524960,
    "phylumKey": 34,
    "kingdomKey": 5,
    "family": "Cortinariaceae",
    "familyKey": 4172,
    "confidence": 94,
    "rank": "GENUS",
    "class": "Agaricomycetes",
    "phylum": "Basidiomycota",
    "scientificName": "Cortinarius (Pers.) Gray, 1821",
    "canonicalName": "Cortinarius",
    "synonym": false,
    "matchType": "EXACT",
    "genus": "Cortinarius",
    "orderKey": 1499,
    "order": "Agaricales",
    "genusKey": 2524960,
    "classKey": 186
},
"thumbnail": 1,
"location": "/location/show_location/1883"

```

Figure 01 : Structure des fichiers JSON

Au total, il y a 650.743 documents. Beaucoup sont incomplets (il peut manquer la photo, la famille, le genre, l'espèce etc.).

La clé “image_url” contient le chemin vers l'image, à laquelle il faut ajouter l'extension .jpg³ pour afficher correctement l'image.

Chaque image a été téléchargée avec un script utilisant le module “requests”.

Pour créer le dataframe, nous avons utilisé la fonction `generate_dataframe_from_json`. Il nous a permis de sélectionner les images issues du dataset `clean_dataset`.

³ Lien vers un exemple : <https://images.mushroomobserver.org/320/39.jpg>

La colonne thumbnail de notre dataframe nous permet de sélectionner les images qui appartiennent au dataset *clean_dataset*. Si thumbnail vaut 0, les images appartiennent au dataset *complete_dataset*. Si thumbnail vaut 1, les images appartiennent au dataset *clean_dataset*. Le graphe ci-dessous nous présente le total pour chaque thumbnail:

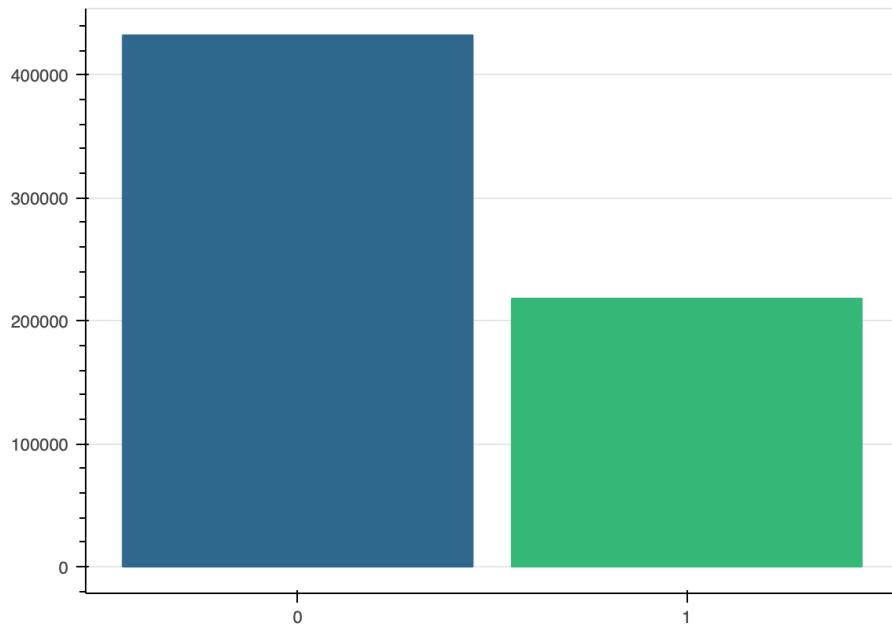


Figure 02 : Distribution des images par thumbnail

On a pu constater que les images qui nous intéressent représentent environ 34% (soit 218 262 images) du nombre d'images décrites dans le dataframe.

Nous le reverrons par la suite, les données sont très mal équilibrées.

À partir de tous les fichiers JSON, voici la répartition pour les royaumes :

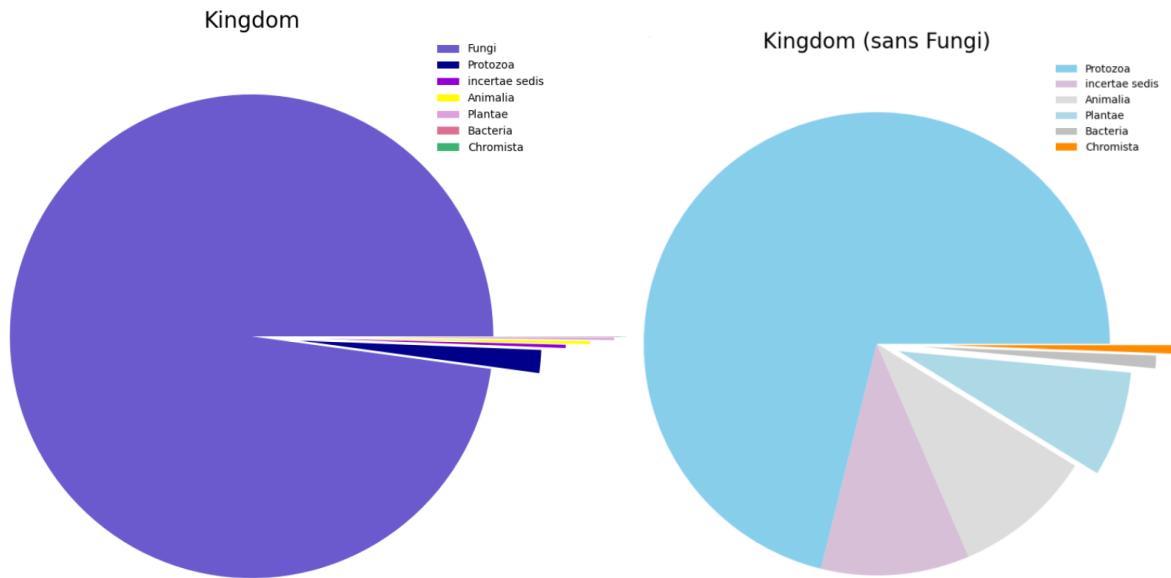


Figure 03 : Distribution des images par royaume

Kingdom	
Fungi	603.391
Protozoa	9.453
incertae sedis	1.368
Animalia	1.293
Plantae	975
Bacteria	117
Chromista	80

Figure 04 : Nombre des images par royaume

Pour les volumétries suivantes, seuls ont été pris en compte les documents issus des fichiers JSON qui possèdent toutes les clés suivantes :

- thumbnail = 1
- family
- genus
- species
- image_url

Voici la répartition que nous obtenons pour les familles :

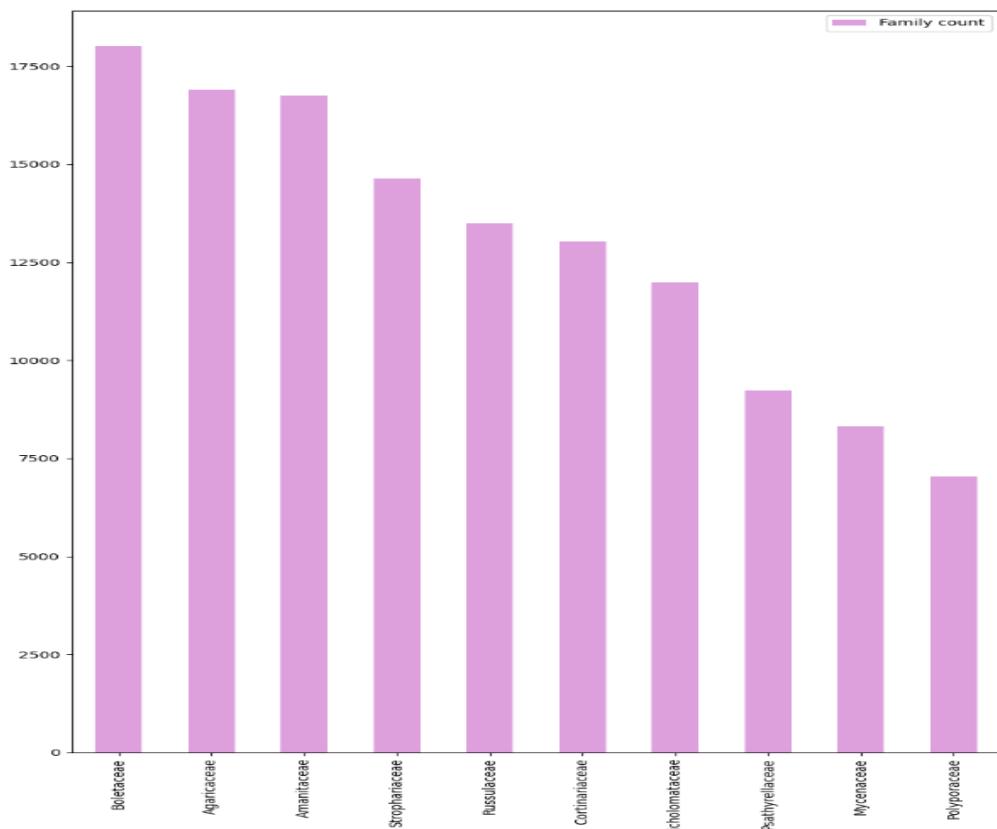


Figure 05 : Répartition des images par famille avec les données sélectionnées

Family	
Boletaceae	18.011
Agaricaceae	16.905
Amanitaceae	16.754
Strophariaceae	14.636
Russulaceae	13.485
Cortinariaceae	13.029
Tricholomataceae	11.978
Psathyrellaceae	9.239
Mycenaceae	8.317
Polyporaceae	7.038

Figure 06 : Nombre d'images par famille avec les valeurs sélectionnées

La répartition pour les genres :

Genus	
Amanita	16.464
Russula	9.008
Cortinarius	8.879
Mycena	6.322
Agaricus	5.627
Inocybe	5.438
Psathyrella	4.922
Boletus	4.591
Entoloma	4.451
Lactarius	3.939
Tricholoma	3.865
Pluteus	3.863
Gymnopilus	3.718
Hygrocybe	3.117
Gymnopus	3.113
Clitocybe	3.024
Suillus	2.952
Pholiota	2.526
Armillaria	2.511

Figure 07 : Nombre d'images par genre avec les données sélectionnées

Nettoyage des données

Le nettoyage des données a été commencé une fois notre décision prise de travailler au niveau taxonomique ‘famille’. Ce niveau nous a paru le plus pertinent pour arriver à une classification performante. Vous pourrez retrouver la démarche qui a mené à cette décision

dans la suite de notre rapport à la partie “Réduction du jeu de données : choix du focus (famille/genre)”.

Une fois notre choix de travailler avec les familles, nous avons gardé les données JSON suivantes pour créer notre dataframe :

- `image_id`,
- `image_url`,
- `family`,
- `filepath`,

Ces variables constituent nos features.

Nous avons retenu les familles avec plus de 3.000 images afin d'avoir suffisamment d'images pour le bon entraînement de nos modèles. En prenant ce critère, nous obtenons l'histogramme suivant :

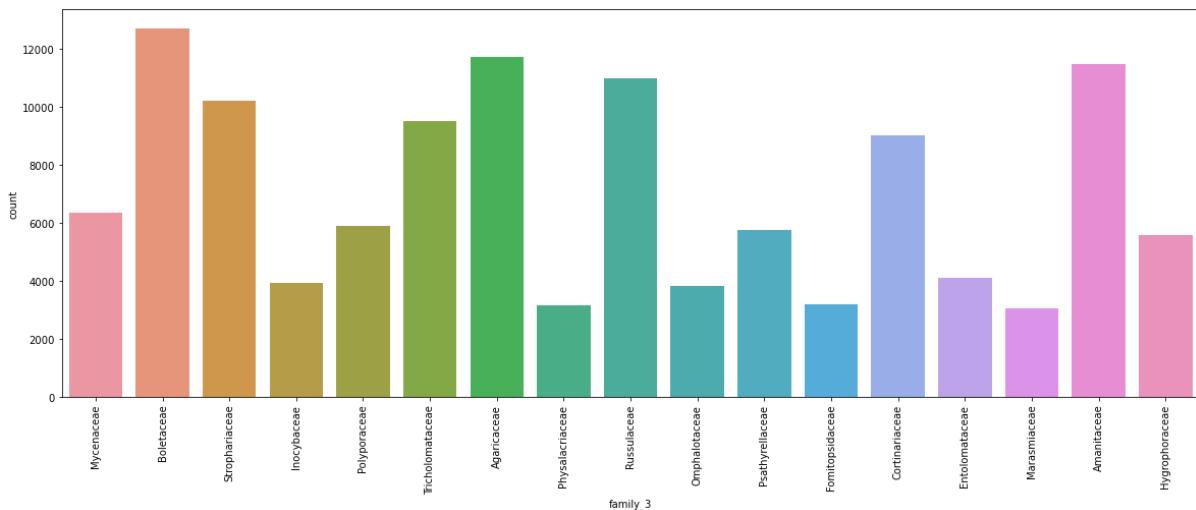


Figure 08 : Nombre d'images par famille avec plus de 3000 images

Ces familles représentent un peu plus de 120.000 images au total. Le dataset obtenu est volumineux et induit des entraînements nécessitant trop de ressources. C'est pourquoi nous avons décidé de le réduire en ne gardant que 5 familles avec un nombre d'images équivalent afin d'avoir un dataset équilibré. Ces 5 familles sont présentées dans l'histogramme ci-dessous et elles feront l'objet de notre étude.

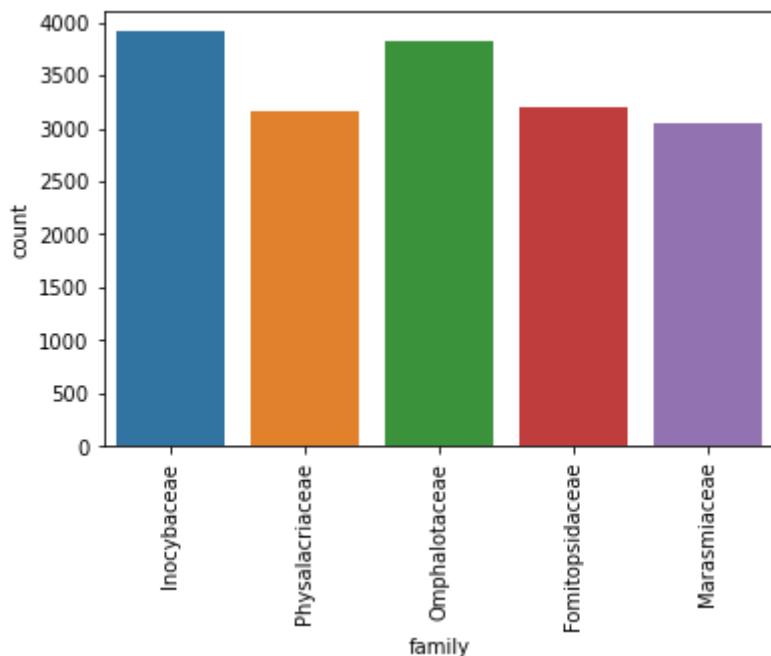


Figure 09 : Les 5 familles retenues pour notre étude

Enfin, notre variable target, au format numérique, est nommée “**label**”. Ses valeurs sont 0, 1, 2, 3 ou 4 en fonction de la famille.

Le nettoyage des données a suivi une méthode classique avec :

- Suppression (*dropna*) des NaN
 - genres NaN
 - espèces NaN
 - familles NaN
- Mise à jour de la famille à partir des ranks plus spécialisés (dans certains cas, l'espèce est renseignée mais pas la famille, le jeu de données permet parfois de retrouver la valeur avec un autre cas)

Enfin la distribution des données nous a contraint à choisir la famille comme niveau taxonomique de classification. Nous approfondirons ce point par la suite.

Caractéristiques des images

Le projet MushPy a pour but de classifier des images de champignons. À partir de cet objectif, les approches possibles sont multiples.

En tant qu'humain, nous avons l'habitude d'ignorer beaucoup d'informations contenues dans une image. Par exemple, lorsque nous souhaitons faire la différence entre 2 objets, nous focalisons notre attention sur ces 2 objets spécifiquement. Peu importe le fond de l'image, sa colorimétrie...

C'est ce que nous espérons de la part de nos algorithmes de classification, mais ce n'est pas automatique.

Parmi les informations que nous avons l'habitude "d'ignorer", il y a la luminosité de l'image. Notre modèle devra faire de même afin d'éviter l'introduction d'un biais.

La classification devra reposer sur les caractéristiques même du champignon et non sur les caractéristiques de l'image.

Nous avons vérifié la luminosité des images. Il est intéressant de noter que la luminosité d'une image n'est pas une notion avec une définition précise. Il existe ainsi de multiples manières de mesurer la luminosité d'une image :

1. Convert image to greyscale, return average pixel brightness.

```
def brightness( im_file ):
    im = Image.open(im_file).convert('L')
    stat = ImageStat.Stat(im)
    return stat.mean[0]
```

2. Convert image to greyscale, return RMS pixel brightness.

```
def brightness( im_file ):
    im = Image.open(im_file).convert('L')
    stat = ImageStat.Stat(im)
    return stat.rms[0]
```

3. Average pixels, then transform to "perceived brightness".

```
def brightness( im_file ):
    im = Image.open(im_file)
    stat = ImageStat.Stat(im)
    r,g,b = stat.mean
    return math.sqrt(0.241*(r**2) + 0.691*(g**2) + 0.068*(b**2))
```

4. RMS of pixels, then transform to "perceived brightness".

```
def brightness( im_file ):
    im = Image.open(im_file)
    stat = ImageStat.Stat(im)
    r,g,b = stat.rms
    return math.sqrt(0.241*(r**2) + 0.691*(g**2) + 0.068*(b**2))
```

5. Calculate "perceived brightness" of pixels, then return average.

```
def brightness( im_file ):
    im = Image.open(im_file)
    stat = ImageStat.Stat(im)
    gs = (math.sqrt(0.241*(r**2) + 0.691*(g**2) + 0.068*(b**2))
          for r,g,b in im.getdata())
    return sum(gs)/stat.count[0]
```

Figure 10 : Approches possibles pour mesurer la luminosité d'une image⁴

Comme nous pouvons le constater, certaines approches se basent sur des mesures concrètes alors que d'autres approches ajoutent des facteurs de modification afin de se rapprocher d'une mesure similaire à la perception humaine.

Nous avons sélectionné l'approche de conversion de l'image en **niveau de gris**, car cela semble une approche facile à mettre en œuvre et avec une forte reproductibilité.

⁴ d'après la solution à une question de stackoverflow.com, disponible via ce [lien](#)

Mesure de la luminosité

Fichiers d'intérêt :

- /EDA/mushpy_study_image_brightness_20210616.ipynb

Dans un premier temps, nous avons réalisé le travail d'étude de luminosité sur l'ensemble des 17 familles initialement sélectionnées, comme le montre la figure ci-dessous:

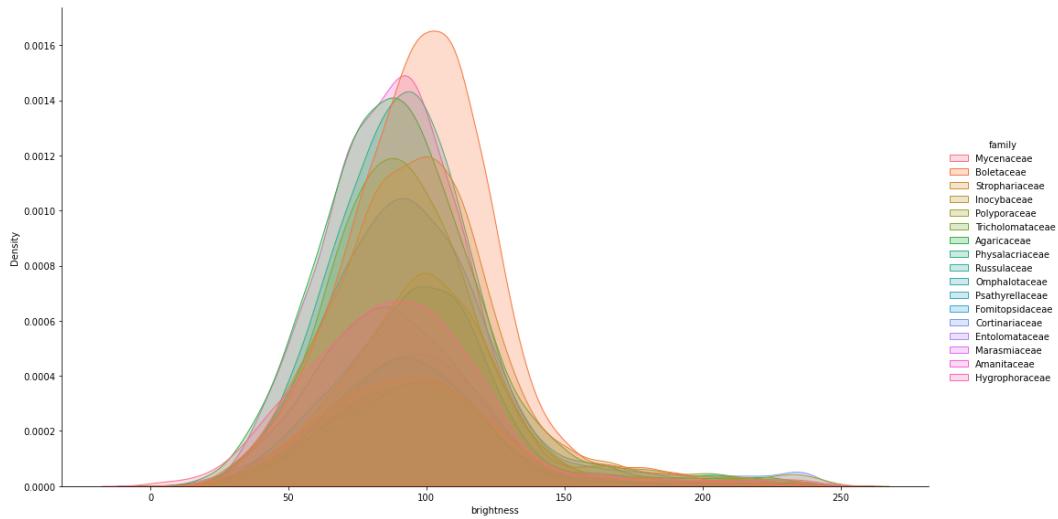


Figure 11 : Distribution de la luminosité des images de 17 familles de champignons contenues dans le jeu de données sélectionné.

Dans un second temps, nous avons réalisé à nouveau cette analyse sur le jeu de données restreint à 5 familles, comme le montre la figure ci-dessous:

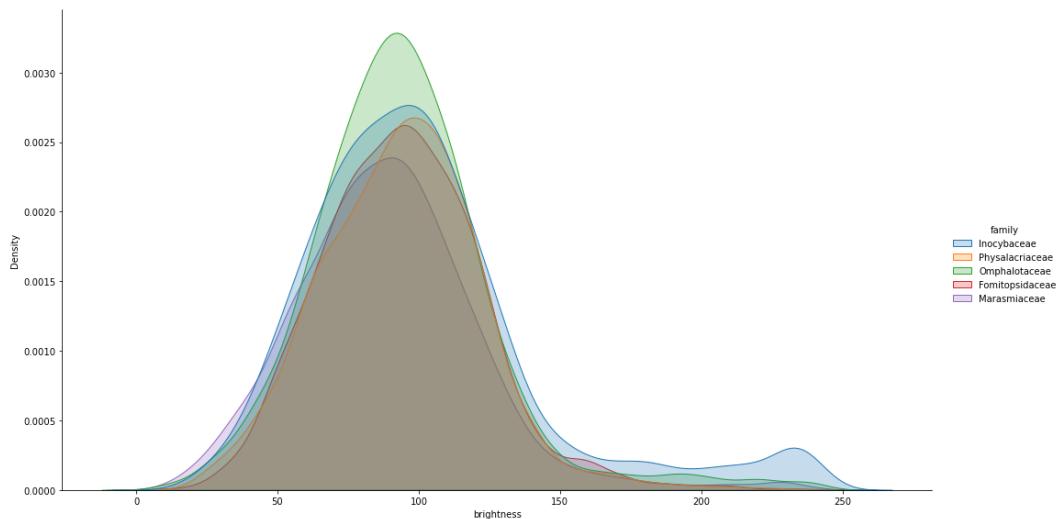


Figure 12 : Distribution de la luminosité des images des 5 familles de champignons contenues dans le jeu de données sélectionné (restreint).

Ces deux visualisations nous indiquent bien que **nous n'avons pas de forte disparité** dans la répartition **de la luminosité** de nos images. Cela nous indique donc clairement que **nous n'aurons pas à travailler ce paramètre** avant de procéder à la modélisation.

Visualisation des données

Visualisation des familles à l'oeil humain

A la suite de la sélection de nos familles, il serait intéressant de prendre un peu de recul sur les données et de prêter attention aux images que nous avons dans nos classes avant de commencer l'élaboration de notre modèle. Ici, le but est de voir si nous pouvons à l'œil voir des différences entre nos classes et si nous-mêmes pouvons classer facilement ces familles.

Ci-dessous, nous allons montrer différentes photos de nos classes prises aléatoirement afin de nous "entraîner" sur nos classes.

Fichier d'intérêt :

- Fichier:
 - EDA/Comparison_of_labels_on_different_pictures_20210711.ipynb

Label 0 :

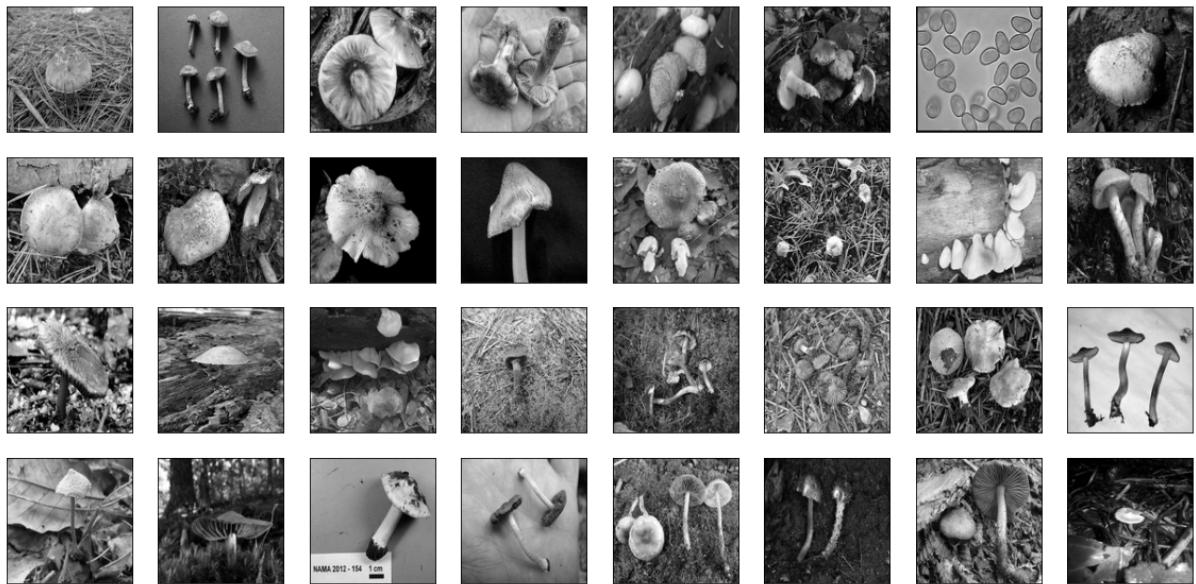


Figure 13 : Exemples d'images ayant le label 0 (*Inocybaceae*)

Label 1:



Figure 14 : Exemples d'images ayant le label 1 (*Omphalotaceae*)

Label 2 :



Figure 15 : Exemples d'images ayant le label 2 (*Fomitopsidaceae*)

Label 3 :



Figure 16 : Exemples d'images ayant le label 3 (*Physalacriaceae*)

Label 4 :Figure 17 : Exemples d'images ayant le label 4 (*Marasmiaceae*)

Comparaison des labels sur quelques images:

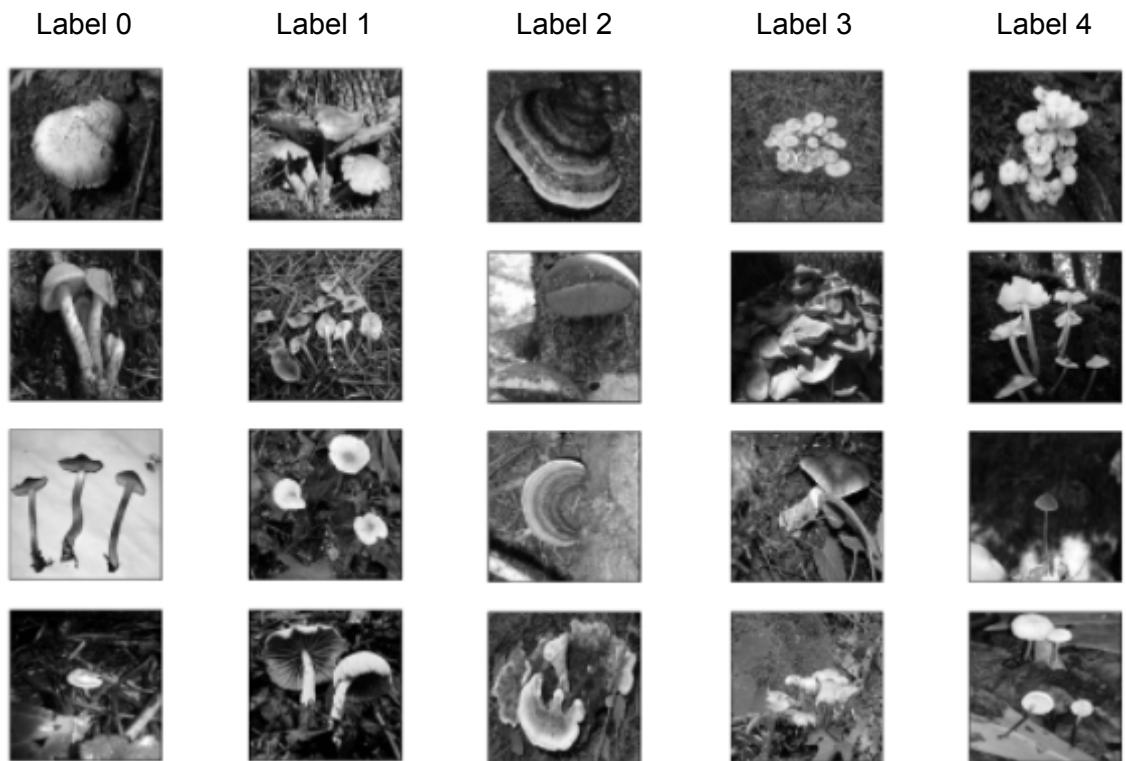


Figure 18 : Images de nos labels pour comparaison

Nous pouvons remarquer que pour un œil non entraîné, la classification est vraiment difficile sauf pour une classe. Le label 2 est lui bien distinct des autres alors que la différence entre les autres classes est vraiment compliquée à réaliser. En toute logique, nous pouvons donc nous attendre à avoir de bonnes prédictions sur le label 2 de nos modèles alors que sur les autres classes, cela pourrait être plus complexe.

Réduction de dimensions

Grâce à l'étude des images que nous venons de faire, nous savons désormais qu'un label se démarque bien des autres. Il est alors intéressant d'appliquer la méthode de réduction de dimensions à nos images puis de les projeter sur un plan. Cela pourrait déjà faire apparaître des clusters de famille et nous aider à sélectionner des features si tel est le cas.

Afin de pouvoir appliquer cette méthode, nous avons réalisé un prétraitement des images en les passant en noir et blanc, ainsi qu'en les redimensionnant toutes en 128x128 pixels. De fait, elles étaient toutes de même couleurs et dimensions lors de l'application de la réduction de dimensions. Sur le plan nous avons projeté à la fois les familles avec différentes couleurs et aussi quelques images, afin de faciliter la lecture de ce plan.

Les méthodes utilisées ont été Isomap et PCA. L'*Isomap* permet d'identifier des corrélations non linéaires entre les variables et la *PCA* permet de faire la même chose pour des corrélations linéaires. En utilisant ces 2 méthodes, nous pouvons ainsi faire apparaître des relations ou non dans notre jeu de données.

Fichier d'intérêt :

- Fichier:
 - EDA/Dimensionality_reduction_visualization_1_8_20210711.ipynb

Isomap

Voici le plan que nous obtenons après la méthode Isomap:

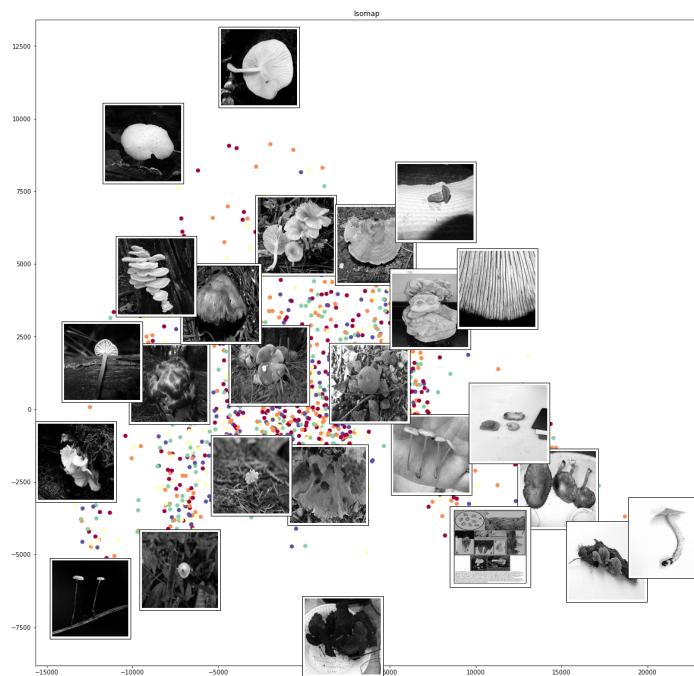


Figure 19 : Projections de nos images sur un plan après Isomap

Nous pouvons remarquer qu'il n'y a pas de clusters qui apparaissent. En effet, les familles sont toutes réparties entre elles, cela se voit très bien grâce à leurs couleurs. Ceci dit, nous pouvons voir grâce aux images qu'il y a ici une répartition de la couleur des images. Celles les plus sombres se situent à gauche du plan et celles les plus claires à sa droite.

PCA

Voici le plan que nous obtenons après la méthode *PCA*:

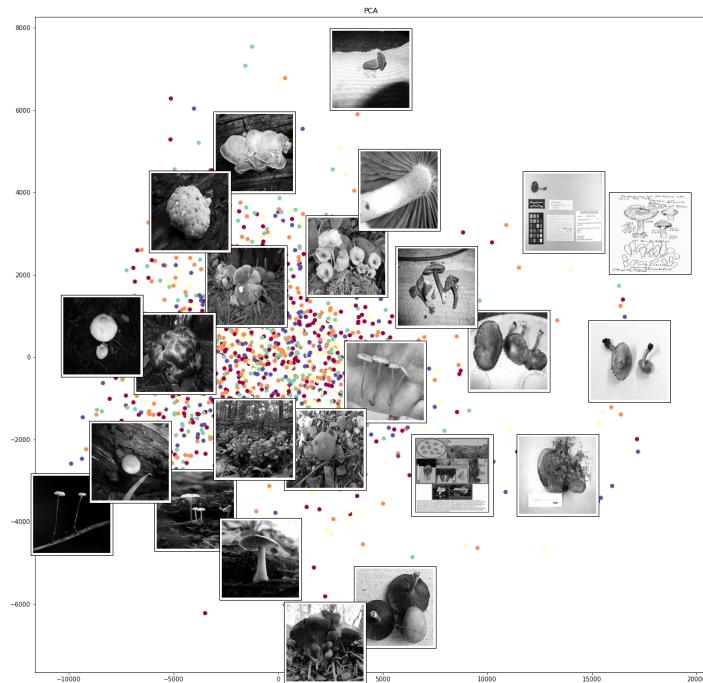


Figure 20 : Projections de nos images sur un plan après PCA

À nouveau nous pouvons remarquer qu'il n'y a pas de clusters qui apparaissent. En effet, les familles sont toutes reparties entre elles une nouvelle fois. Par ailleurs, nous pouvons voir grâce à cette figure, une répartition de la couleur des images. Celles les plus sombres se situent à gauche du plan et celles les plus claires à sa droite. C'est exactement la même chose qu'avec l'*Isomap*.

L'utilisation de ces deux méthodes a porté à notre connaissance que nous ne pouvions pas mettre en évidence de clusters de famille avec une méthode de réduction. Néanmoins, ce résultat est à modérer, car nous avons énormément réduit les dimensions jusqu'à en garder que deux. Ces deux dimensions ne nous paraissent pas suffisantes pour capturer l'essentiel de la structure fondamentale des données, mais en conserver plus nous enlèvera la lisibilité sur un plan ou un espace 3D. C'est pourquoi, dans la suite de notre rapport, nous avons dans un premier temps essayé des modèles de classification classiques basés sur la réduction de dimensions. Ainsi nous avons pu connaître sa pertinence comparée au deep learning.

Modélisation

Classification du problème

Comme mentionné auparavant, le but de ce projet est l'identification des champignons, c'est un problème typique de classification des images qui est généralement abordé par des méthodes de deep learning. Pour atteindre notre objectif, nous avons testé plusieurs modèles qui sont présentés ci-dessous, dont un qui ne relève pas du deep learning.

Les métriques utilisées pour comparer les performances de ces modèles étaient l'accuracy, les rapports de classification, ainsi que les matrices de confusion. En effet l'accuracy, nous permettait de rapidement savoir la performance de notre classification. Les rapports de classification nous permettaient d'avoir une lecture plus fine des performances sur chaque classe. Enfin les matrices de confusion nous aidaient à comprendre les prédictions correctes et incorrectes, mais surtout elles nous donnaient les indices nécessaires pour comprendre les problèmes de nos modèles, tels que l'overfitting par exemple. De fait, elles nous permettaient d'appliquer des actions correctives à nos modèles afin de pouvoir les améliorer.

Test des différents modèles

Dans notre projet, nous avons au total utilisé 7 modèles différents. Nous avons ainsi testé 1 modèle par réduction de dimension, 1 modèle de deep learning simple et 5 modèles par transfer learning.

Cela nous a permis de voir les différentes performances de chaque modèle, ainsi que leurs avantages et inconvénients quant à notre problématique. Ainsi nous avons aussi pu tester toutes les différentes techniques que nous avons apprises au cours de notre formation chez DataScientest lors de notre projet.

Réduction de dimensions

Présentation :

La réduction de dimensions consiste à récupérer des données d'un espace de grande dimension, et à les remplacer par des données dans un espace plus restreint. Le but étant de garder que les dimensions les plus pertinentes à la construction de nos modèles.

Cela nous permet de faciliter la visualisation, réduire les coûts de calculs, de stockage et d'acquisition de données. Par conséquent, nous pouvons entraîner des modèles moins complexes et donc moins chronophages.

Une fois les réductions de dimensions effectuées, nous avons mis en œuvre deux modèles, un *SVC* et un *Random Forest*.

En dépit de la réduction, nous avons déjà pu connaître certaines limites et notamment celle du temps de calcul. En effet, malgré un jeu de données déjà réduit, il nous aura été nécessaire de le réduire encore pour pouvoir utiliser notre modèle avec SVC. Celui-ci était particulièrement long à entraîner, alors que l'usage de Random Forest était plus raisonnable. Ainsi, pour avoir un temps de calcul acceptable (quelques heures au maximum), il a fallu réduire le dataset de 8 fois.

Résultats obtenus :

Fichiers d'intérêt :

- **Fichier du modèle *SVC* et évaluation :**
 - models/iteration2/Dimensionality_reduction_SVC_1_08_20210710.ipynb
- **Fichier du modèle *Random Forest* et évaluation :**
 - models/iteration2/Dimensionality_reduction_random_forest_1_08_20210710.ipynb

Les 2 modèles ont donné des résultats similaires, cependant ceux-ci n'étaient pas bons. Ils avaient une accuracy autour de 20%, soit les probabilités du hasard.

Voici les rapports de classification:

SVC					Random Forest				
	precision	recall	f1-score	support		precision	recall	f1-score	support
0	0.22	0.25	0.23	99	0	0.20	0.40	0.27	94
1	0.23	0.26	0.24	87	1	0.16	0.25	0.20	96
2	0.24	0.19	0.21	79	2	0.17	0.09	0.12	79
3	0.20	0.20	0.20	83	3	0.21	0.04	0.06	81
4	0.17	0.13	0.14	78	4	0.22	0.10	0.14	78
accuracy			0.21	426	accuracy			0.19	428
macro avg	0.21	0.21	0.21	426	macro avg	0.19	0.18	0.16	428
weighted avg	0.21	0.21	0.21	426	weighted avg	0.19	0.19	0.16	428

Figure 21 : Rapports de classification des modèles SVC et Random Forest après réduction des dimensions

Voici les matrices de confusion:

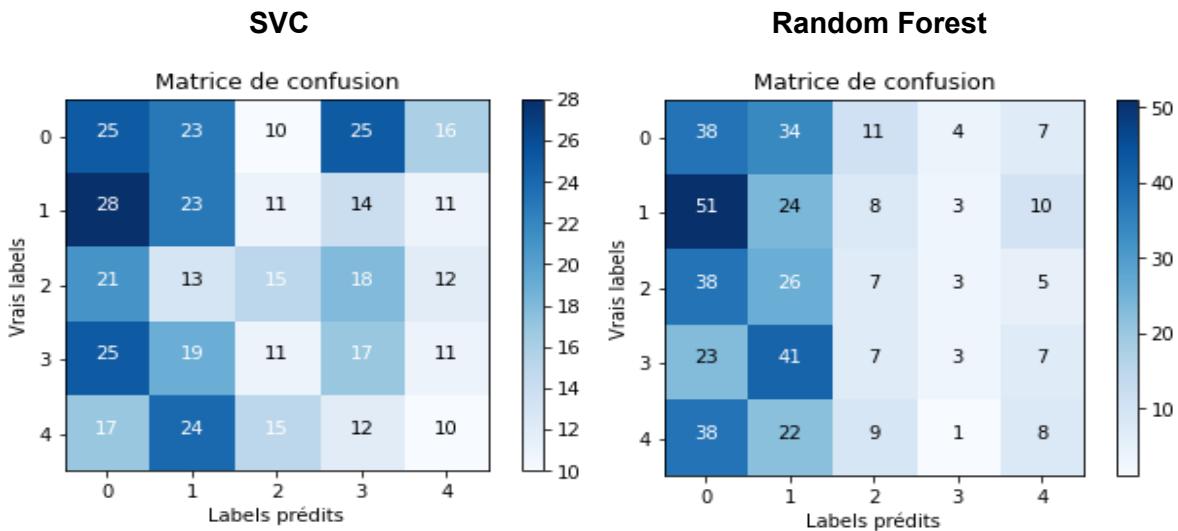


Figure 22 : Matrices de confusion des modèles SVC et Random Forest après réduction des dimensions

Nous pouvons constater les mauvais classements de nos 2 modèles, ainsi qu'un overfitting sur les labels 0 et 1. Pour résoudre ce problème, il conviendrait de réaliser un *GridSearch* afin de trouver de meilleurs paramètres qui permettraient d'améliorer nos résultats. Cependant, cette voie n'a pas été empruntée, car l'amélioration possible était trop faible par rapport à ce que nous souhaitons faire. En effet, nous aurions peut-être obtenu une accuracy aux alentours de 30% et une meilleure répartition des classifications mais

l'objectif d'une classification précise de nos 5 familles n'aurait pas été atteint. C'est pourquoi, nous nous sommes tournés vers le deep learning qui nous promettait de bien meilleurs résultats.

LeNet

Présentation:

L'architecture *LeNet* est introduite par LeCun et al. en 1998 dans cet article : Gradient-Based Learning Applied to Document Recognition. L'implémentation de *LeNet* par les auteurs était principalement utilisée pour l'OCR et la reconnaissance de caractères dans les documents.

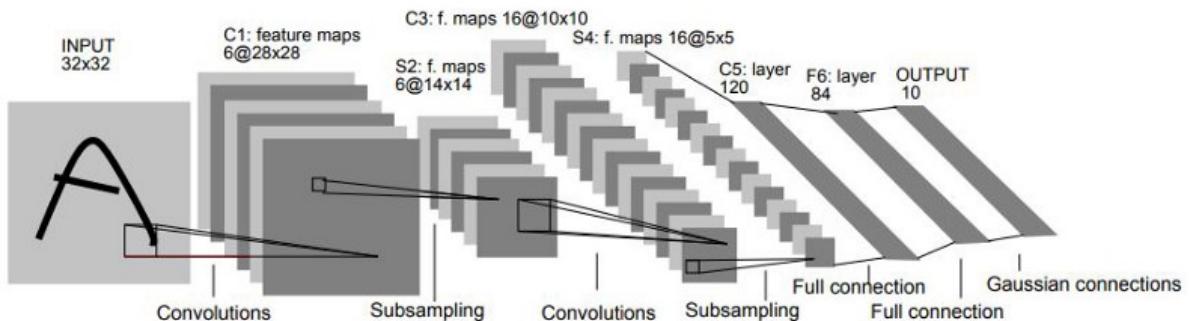


Figure 23 : L'image originale publiée dans [LeCun et al., 1998]

Voici en détails l'architecture *LeNet* que nous avons utilisée pour notre problématique :

1. Initialisation du model *Sequential()*
2. Ajout d'une première couche de convolution (*Conv2D*)
3. Ajout d'une couche *Maxpooling*
4. Ajout de la deuxième couche de convolution
5. Ajout de *Maxpooling*
6. Ajout de *Dropout* fixé à 0.4 pour éviter le surapprentissage
7. Ajout d'une couche *Flatten*
8. Ajout d'une première couche *Dense*
9. Ajout d'une dernière couche *Dense* avec le même nombre de neurones que nos labels (5) et une fonction d'activation *Softmax*

Ce qui nous donne le summary suivant :

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 252, 252, 30)	2280
<hr/>		
max_pooling2d (MaxPooling2D)	(None, 126, 126, 30)	0
<hr/>		
conv2d_1 (Conv2D)	(None, 124, 124, 16)	4336
<hr/>		
max_pooling2d_1 (MaxPooling2D)	(None, 62, 62, 16)	0
<hr/>		
dropout (Dropout)	(None, 62, 62, 16)	0
<hr/>		
flatten (Flatten)	(None, 61504)	0
<hr/>		
dense (Dense)	(None, 128)	7872640
<hr/>		
dense_1 (Dense)	(None, 5)	645
<hr/>		
Total params: 7,879,901		
Trainable params: 7,879,901		
Non-trainable params: 0		

Figure 24 : Summary du modèle *LeNet*

Nous avons utilisé la classe *ImageDataGenerator* pour augmenter le nombre de nos images et éviter le surapprentissage. Nous avons utilisé la méthode *flow_from_dataframe* pour créer nos datasets de train et de test.

Résultats obtenus :

Fichiers d'intérêt :

- Fichier du modèle **LeNet et évaluation** :
 - `models/iteration2/lenet_mushpy.py`

Sur la figure ci-dessous, vous pouvez voir les plots de l'*accuracy* et la fonction de loss pour nos datasets de train et de test :

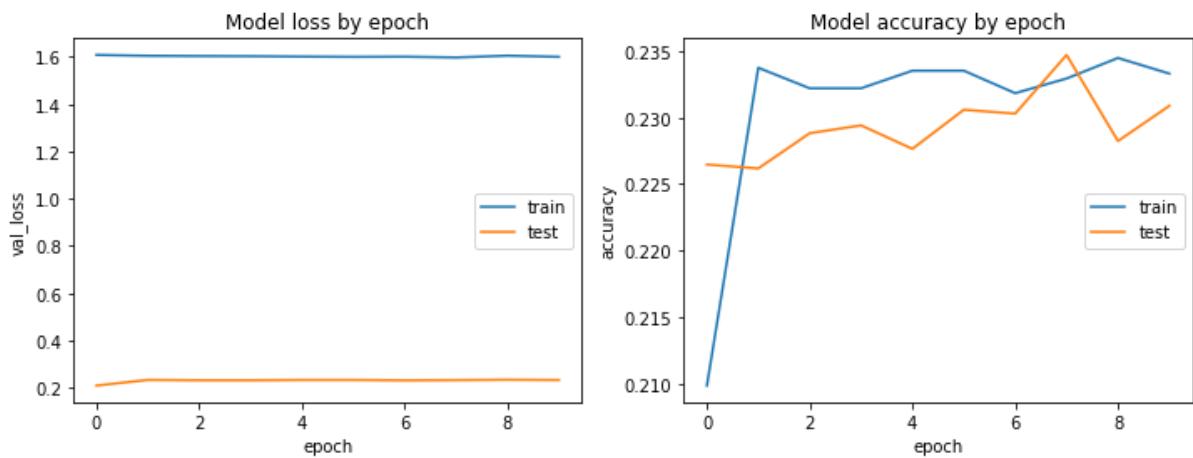


Figure 25: historique d'entraînement du modèle Lenet

Nous pouvons constater que nous n'avons pas une amélioration significative de notre accuracy ni pour notre dataset de train, ni pour celui de test. La fonction de perte ne diminue pas dans les deux cas. Nous pouvons en conclure que notre modèle n'apprend pas.

La matrice de confusion nous montre que le modèle classe la plupart des images en label 0 :

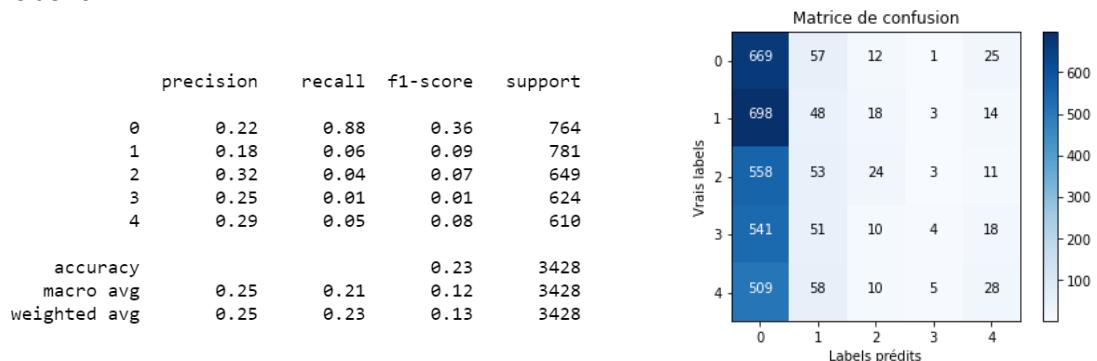


Figure 26: Rapport de classification et matrice de confusion du modèle Lenet

Via notre rapport de classification, nous remarquons des *F-score* très faibles pour tous les labels.

Ces indicateurs nous prouvent que l'architecture *LeNet* n'est pas une architecture adaptée pour notre problématique de classification des champignons. Par conséquent, nous avons privilégié la piste de transfert learning.

Modélisation par “transfer learning”

Le transfer learning, bien qu'étant une approche en plein essor (mis en avant notamment par Andrew Yan-Tak Ng en 2016), est une approche dont les fondements étaient déjà décrits en 1976⁵. Le transfer learning répond donc notamment à deux problématiques grandissantes :

1. Les données disponibles sont grandissantes;
2. Les ressources informatiques sont limitées alors que la complexité des modèles requiert des puissances de calculs toujours plus importantes.

Ainsi, le “*transfer learning*” est une approche qui consiste à transférer à un nouveau modèle, le savoir d'un modèle déjà entraîné. En l'occurrence, il s'agit de figer les poids du premier modèle et de transférer ces poids (issus de la résolution d'un problème précis) afin de répondre à une nouvelle problématique résolue par un nouveau modèle.

De cette manière, le problème à résoudre bénéficie du résultat d'un entraînement préalable pour lequel nous n'aurions eu ni le temps, ni les ressources matérielles.

D'un point de vue pratique, le transfer learning s'implémente de manière relativement simple. Il convient d'importer un modèle pré-entraîné avec ses poids (en excluant les paramètres d'entrée pour pouvoir lui donner nos entrées spécifiques), il faut ensuite figer les poids de ce modèle puis l'inclure dans notre architecture de modèle en tant que modèle d'entrée.

Le transfer learning requiert tout de même deux problématiques proches l'une de l'autre afin de pouvoir s'appliquer.

Dans notre cas, nous sommes face à une problématique de classification d'images. Ainsi, nous nous sommes focalisés sur des modèles ayant été pré-entraînés sur une très large bibliothèque d'image : *ImageNet*.

ImageNet est une collection de données organisées sous l'architecture “WordNet” : une base de données lexicale (initialement développée en anglais). Au sein de *WordNet*, chaque concept de la langue est représenté par un “synset” (il en existe plus de 100 000). L'objectif d'*ImageNet* est donc de proposer un jeu de donné labellisé composé d'au moins 1000 images pour illustrer chaque synset⁶. A ce jour, un jeu de données est particulièrement utilisé dans le

⁵ Stevo. Bozinovski and Ante Fulgosi (1976). "The influence of pattern similarity and transfer learning upon the training of a base perceptron B2." (original in Croatian) Proceedings of Symposium Informatica 3-121-5, Bled

⁶ <https://image-net.org/about.php>

cadre d'une "compétition" visant à évaluer la pertinence et l'efficacité d'un modèle (*ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*). *ImageNet* contient 14 197 122 images, issus de 21 841 synsets. Chaque image a été annotée à la main.

La section suivante vise donc à présenter les modèles que nous avons évalués, puis celui que nous avons retenu et optimisé.

Schéma général utilisé :

Afin d'évaluer la pertinence de différents modèle, nous nous sommes arrêtés sur une base de modèle fixe dans lequel les principales étapes sont :

1. Utilisation du jeu de données
2. Création d'un jeu d'entraînement (80%) et un jeu de test (méthode *train_test_split* avec ajout d'un *random state* pour pouvoir répliquer les résultats).

Pour nos premiers essais (rapidement abandonnés, car ne fonctionnaient pas):

3. Création d'un *Dataset tensorflow* avec redimensionnement des images et pré-processing spécifique au modèle de base.

Pour les itérations suivantes : donc étape 3 ou étape 4 (mais pas les deux)

4. Création d'un *Image Data Generator* avec rotation, décalage et zoom de l'image.
5. Utilisation du *flow_from_dataframe* pour envoyer les images au model.

6. Import du modèle de base : **BaseModel**

7. Construction de notre modèle *Séquentiel* :

```
a. model = Sequential()
b. model.add(BaseModel1)
c. model.add(GlobalAveragePooling2D())
d. model.add(Dense(units = 1024, activation = 'relu'))
e. model.add(Dropout(0.3))
f. model.add(Dense(units = 512, activation = 'relu'))
g. model.add(Dropout(0.5))
h. model.add(Dense(units = 5, activation = 'softmax'))
```

Tout ceci aboutit au "summary" suivant :

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
TRANSFER	Something	Something
LEARNING		
MODEL		
<hr/>		
global_average_pooling2d	(G1 (None, 1280)	0
dense (Dense)	(None, 1024)	1311744
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, Class-Number)	2565
<hr/>		

Figure 27 : Summary type issu du transfer learning

8. Définition de 3 callbacks :

- a. Checkpoint : qui permet d'enregistrer le modèle au fur et à mesure les propriétés du modèle si ces dernières se sont améliorées par rapport au précédent enregistrement;
- b. Ajustement du taux d'apprentissage ("learning rate"). Le learning rate est un paramètre crucial permettant de minimiser adéquatement la fonction de perte. Dans les cas où cette dernière ne diminue plus, le callback diminue ce paramètre afin de permettre un meilleur ajustement de la fonction de perte, en espérant atteindre la convergence et la minimisation de la fonction de perte;
- c. Arrêt prématué ("early stopping"). Dans certains cas, le modèle ne s'améliore plus. Ainsi, à la place de laisser l'entraînement consommer des ressources informatiques inutilement, ce callback permet de stopper l'entraînement et de restaurer les résultats de la meilleure époque.

9. Le modèle est compilé avec les paramètres suivants : *Adam* en tant qu'*optimizer* : la littérature internet semble s'accorder à dire qu'*Adam* représente un *Optimizer*, qui, actuellement, correspond au meilleur point de départ pour les problèmes de classification

; *sparse_categorical_crossentropy* pour la fonction de perte et *accuracy* pour la métrique d'efficacité de notre modèle).

10. Entraînement sur 40 époques.

11. Sauvegarde du modèle.

Bien entendu, cette approche est limitée puisque d'autres types d'architecture auraient pu être envisagés (garder le nombre de neurones constants pour les couches denses, à l'inverse procéder avec un nombre de neurones croissants...). De cette manière nous passons nécessairement à côté potentiellement d'un modèle qui aurait pu mieux fonctionner avec une architecture différente.

Néanmoins, cette approche est importante afin de comparer les modèles entre eux : il est nécessaire de ne faire varier qu'un seul paramètre à la fois.

Il est également possible de constater que pour les premières itérations "sérieuses" nous avons utilisé des étapes de *dropout* très importantes (30 et 50%) afin de lutter contre le sur-apprentissage que nous avons obtenu avec nos tout premiers essais. Ceci est abordé dans le paragraphe des difficultés rencontrées. Il s'agit également d'un élément qui a fait l'objet de test lors de la phase d'optimisation.

Compte tenu de nos différents essais, nous avons choisi de ne présenter ici que la dernière itération nous ayant permis d'identifier le modèle avec lequel nous avons choisi d'aller plus loin.

Identification d'un modèle propice

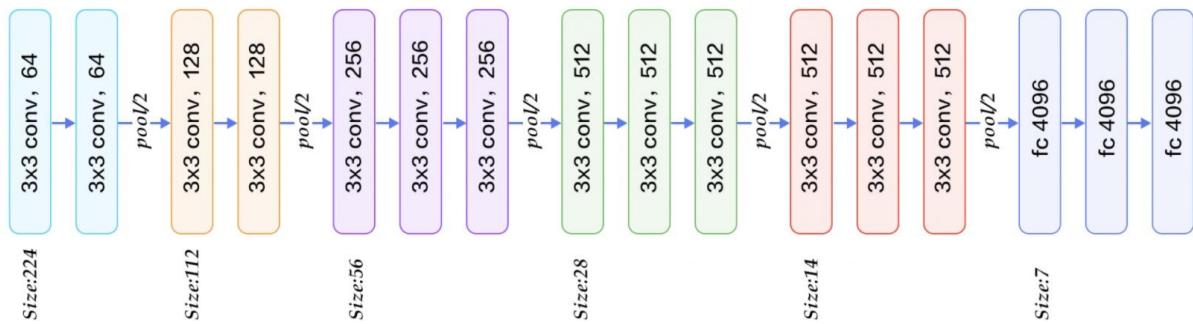
VGG16

Présentation :

*VGG16*⁷ est un réseau neuronal convolutif proposé par K. Simonyan et A. Zisserman de l'université d'Oxford dans un article publié : "Very Deep Convolutional Networks for Large-Scale Image Recognition". Le modèle atteint une top-5 *accuracy* de 92,7% sur le jeu de données d'*ImageNet*, ainsi qu'une erreur de 8,8%.

Ce dernier est nommé *VGG16* car il contient tout simplement 16 couches profondes.

⁷ <https://keras.io/api/applications/vgg/#vgg16-function>

Figure 28: Architecture du modèle VGG16⁸

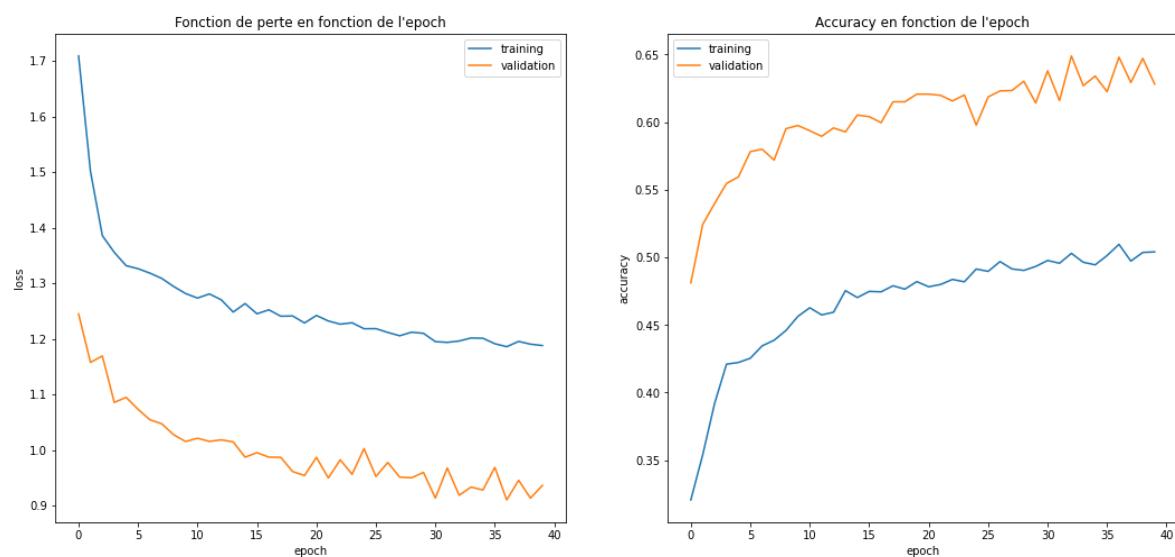
En appliquant l'approche transfer learning, nous avons donc entraîné un modèle dont VGG16 est la base.

Résultats obtenus :

Fichiers d'intérêt :

- **Obtention du modèle :**
 - /models/iteration_1/20210707_model_VGG16-dagenerator_GPU_mac.ipynb
- **Nom du modèle :**
 - model_vgg16_GPU_20210617.h5
- **Evaluation du modèle :**
 - /models/iteration_1/20210707_evaluate_model_VGG16-dagenerator_GPU_mac.ipynb

Dans un premier temps, nous pouvons observer l'évolution de l'entraînement de notre modèle.



⁸ <https://www.datacorner.fr/vgg-transfer-learning/>

Figure 29: historique d'entraînement du modèle VGG16

Il est possible de constater, avec surprise, une meilleure efficacité sur le set de validation (par rapport au set d'entraînement). Une potentielle première explication est le dropout très élevé que nous avons utilisé pour lutter contre le sur-apprentissage (30 puis 50%).

Les rapports de classification et table de confusion sont :

Classification report du modèle VGG16:

	precision	recall	f1-score	support
0	0.36	0.65	0.47	441
1	0.23	0.45	0.30	411
2	0.86	0.65	0.74	863
3	0.48	0.37	0.42	805
4	0.53	0.35	0.42	908
accuracy			0.48	3428
macro avg	0.49	0.49	0.47	3428
weighted avg	0.55	0.48	0.49	3428

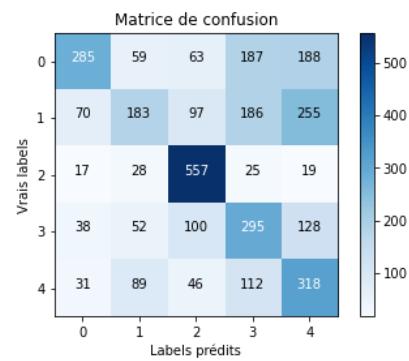


Figure 30: Rapport de classification et matrice de confusion du modèle VGG16

Nous pouvons constater que le modèle ne classifie pas très efficacement nos images, mis à part pour la classe n°2. D'après notre exploration des images, cette classe était la classe la plus "facile" à prédire pour nos regards naïfs de non expérimentés.

Dans le processus, nous avons entraîné puis sauvegardé le modèle (à l'aide de la fonction `model.save("modelXX.h5")`). Afin de rédiger ce rapport, nous avons poussé l'évaluation pour produire les visuels ci-dessus, alors que les courbes de l'évolution de l'entraînement sont obtenues directement à l'issue de l'entraînement avant la sauvegarde du modèle.

Il est donc possible de remarquer que l'accuracy mesurée après chargement du modèle est de 0,48 alors qu'elle était de 0,6281 à l'issue de l'entraînement.

Dernière époque d'entraînement :

```
Epoch 40/40
214/214 [=====] - 619s 3s/step - loss: 1.1877 - acc: 0.5045 -
val_loss: 0.9365 - val_acc: 0.6281
```

Il s'agit à priori d'un problème largement documenté :

<https://github.com/keras-team/keras/issues/4875> (un sujet disponible parmi bien d'autres).

Pour cette raison, pour les modèles subséquents, nous ne présenterons que l'historique d'entraînement que nous avons, par chance enregistré à l'issue des entraînements.

Voici cependant quelques exemples d'images assorties de leurs prédictions comparées aux classes réelles :

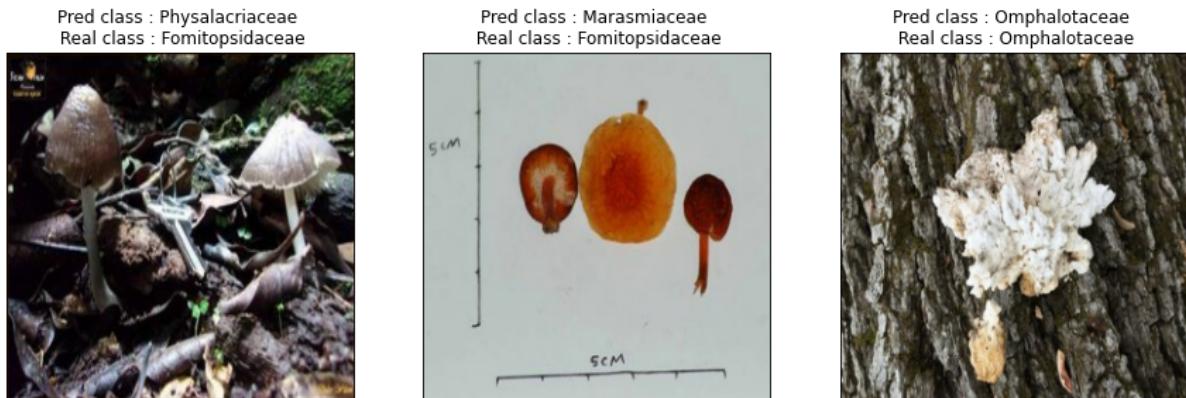


Figure 31: Exemple d'images assorties de leurs prédictions comparées aux classes réelles issu du modèle VGG16

Ce premier exemple d'images nous permet également de soulever une caractéristique / limites de ce jeu de données : en effet comme illustré sur l'image du milieu, certaines images (mal classifiées) sont de très faibles qualités. Pouvons-nous réellement nous attendre à une classification adéquate sur ce genre d'échantillons ?

VGG19

Présentation :

VGG19⁹ est un réseau neuronal convolutif également proposé par K. Simonyan et A. Zisserman de l'université d'Oxford dans le même article qui a présenté le modèle VGG16 ("Very Deep Convolutional Networks for Large-Scale Image Recognition"). Le modèle atteint une *top-5 error-rate* de 9% sur le jeu de données d'*ImageNet*.

Ce dernier est nommé VGG19 car il contient tout simplement 19 couches profondes.

⁹ <https://keras.io/api/applications/vgg/#vgg19-function>

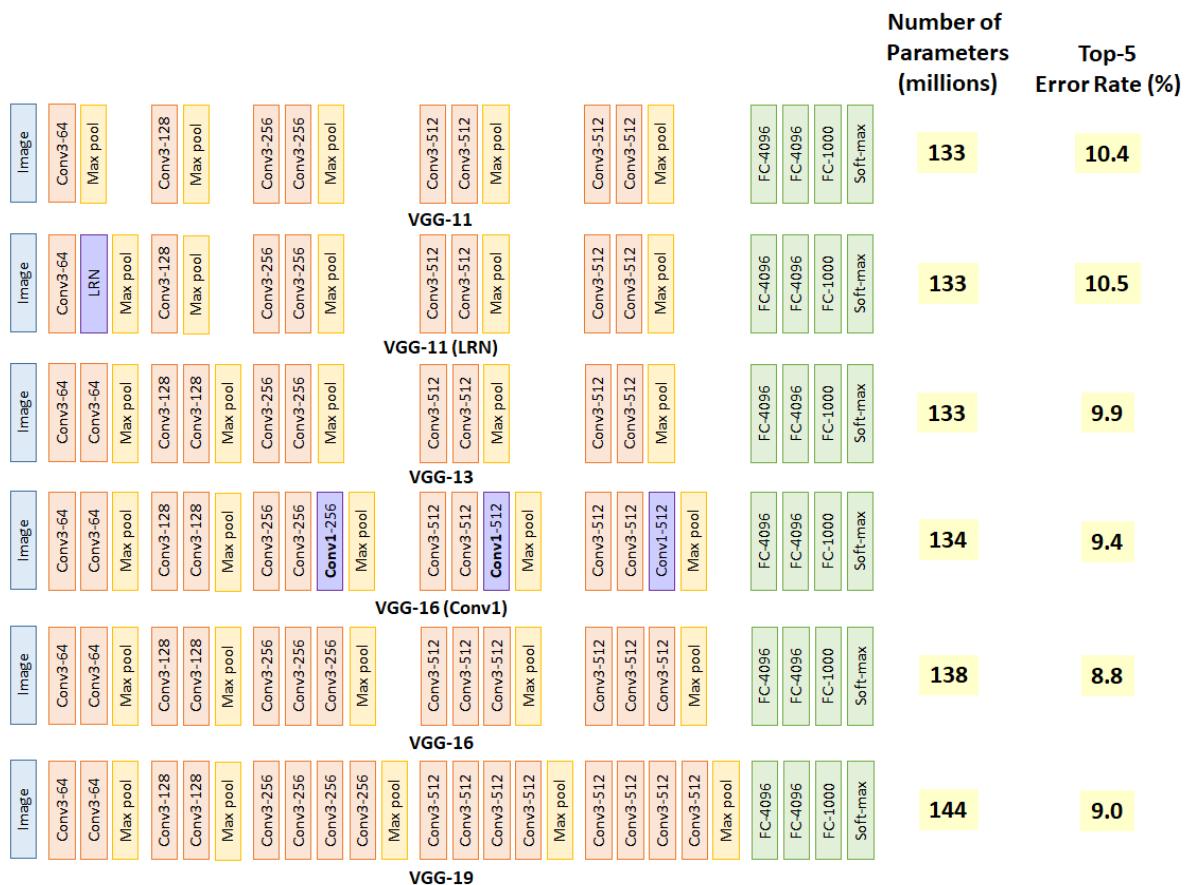


Figure 32: Comparaison entre l'architecture des modèles VGG existants¹⁰.

Comme nous pouvons le constater sur la figure ci-dessus, il n'existe pas que les modèles VGG16 et VGG19. Néanmoins, ces deux modèles sont les plus souvent présentés. Nous pouvons aussi comprendre que les créateurs ont ajouté progressivement de plus en plus de couches à leurs modèles jusqu'à atteindre un plateau d'erreur : ils ont continué à ajouter des couches jusqu'à ce que l'*error-rate* réaugmente sensiblement, signifiant que l'ajout de couches supplémentaires ne permettait pas d'obtenir de gain supplémentaire. Il est aussi possible de constater que ces modèles évaluent un nombre de paramètres colossal de plus de 100 millions !

En appliquant l'approche transfer learning, nous avons donc entraîné un modèle dont VGG19 est la base.

¹⁰

<https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvrc-2014-image-classification-d02355543a11>

Résultats obtenus :

Fichiers d'intérêt :

- **Obtention du modèle :**
 - /models/iteration_1/20210707_model_VGG19-datagenerator_GPU_mac.ipynb
- **Nom du modèle :**
 - model_vgg19_GPU_20210620.h5
- **Evaluation du modèle :**
 - /models/iteration_1/20210707_evaluate_model_VGG19-datagenerator_GPU_mac

Dans un premier temps, nous pouvons observer l'évolution de l'entraînement de notre modèle.

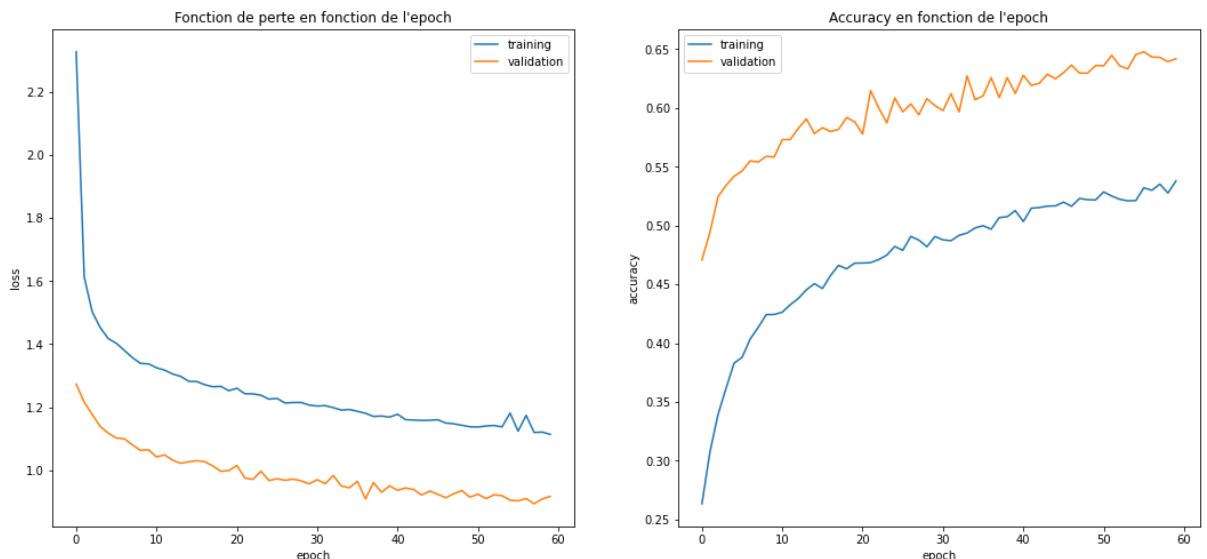


Figure 33: historique d'entraînement du modèle VGG19

Ici également, la fonction de perte reste très élevée et les performances du modèle sont également très modestes : dernière époque d'entraînement :

Epoch 60/60

```
214/214 [=====] - 902s 4s/step - loss: 1.1141 - acc: 0.5383 -
val_loss: 0.9180 - val_acc: 0.6418
```

ResNet50

Présentation :

L'évaluation d'un modèle tel que *ResNet50*¹¹ constitue un test bien plus profond que le simple test d'un modèle supplémentaire. En effet, le modèle *ResNet50* repose sur la mise au point d'une toute nouvelle architecture. Cette dernière a notamment été primée lors de la compétition *ImageNet* en 2015 avec une avance importante : diminution de moitié du taux d'erreur top-5 par rapport aux compétiteurs (top-5 erreur sur le jeu de validation d'*ImageNet* : 5,25%¹²) . Cela témoigne donc d'une réelle percée dans l'architecture des modèles.

Selon les architectures connues avant *ResNet50* : plus un modèle comporte de couches, plus on peut dire que le réseau est profond, meilleures sont les performances. Néanmoins, tel que nous avons pu le constater entre *VGG16* et *VGG19*, à partir d'une certaine profondeur, cette "règle" ne s'applique plus et l'ajout de couches supplémentaires provoque l'effet inverse : une dégradation des performances. Il est alors proposé que dans ces cas, le gradient n'arrive plus à se propager dans les couches les plus profondes.

C'est pour pallier à ces défauts que les inventeurs de l'architecture *ResNet*¹³ ont inventé ce qu'ils ont appelé des connexions par saut, ainsi les différents niveaux sont reliés entre eux. L'optimisation de ce type de connexion serait alors plus facile que l'optimisation de réseaux plus "traditionnels". La notion de bloc résiduel est alors développée : l'entrée x est directement ajoutée à la sortie du réseau. C'est la connexion qui lie l'entrée et la sortie du réseau qui est nommée connexion par saut. Il a été suggéré que ce type d'architecture fonctionne en réalité comme un ensemble de sous-réseaux plus petits, et donc plus facile à entraîner.

¹¹ <https://keras.io/api/applications/resnet/#resnet50-function>

¹² <https://neurohive.io/en/popular-networks/resnet/>

¹³ Publiée : Deep Residual Learning for Image Recognition, He et al 2016

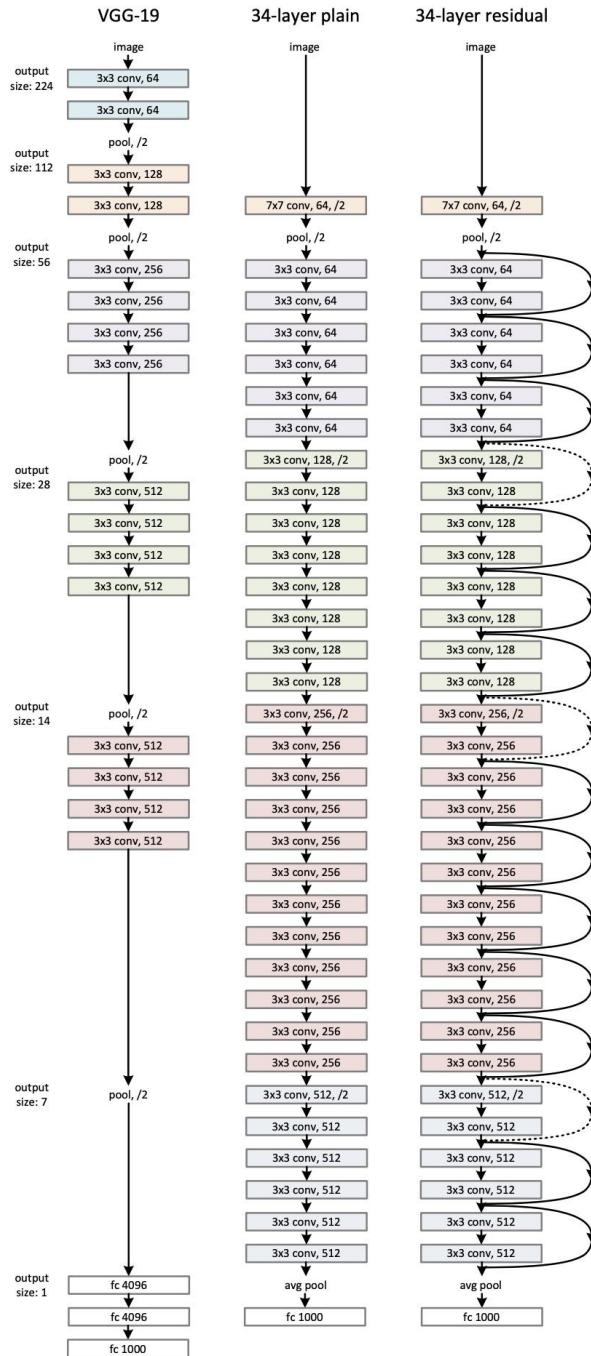


Figure 34: Architecture du modèle *ResNet50* publiée¹⁴ par les inventeurs, en comparaison avec le modèle *VGG19* (à gauche), un modèle “conventionnel” à 34 couches (au centre) et le modèle *ResNet50* (à droite) illustrant les connexions par saut additionnant l’entrée à la sortie du bloc résiduel.

En appliquant l’approche transfer learning, nous avons donc entraîné un modèle dont *ResNet50* est la base.

¹⁴ Deep Residual Learning for Image Recognition, He et al 2016

Résultats obtenus :

Fichiers d'intérêt :

- **Obtention du modèle :**
 - /models/iteration_1/20210707_model_resnet50-datagenerator_GPU_mac.ipynb
- **Nom du modèle :**
 - model_resnet50_GPU_20210617.h5
- **Evaluation du modèle :**
 - /models/iteration_1/20210707_evaluate_model_resnet50-datagenerator_GPU_mac.ipynb

Nous pouvons observer l'évolution de l'entraînement de notre modèle.

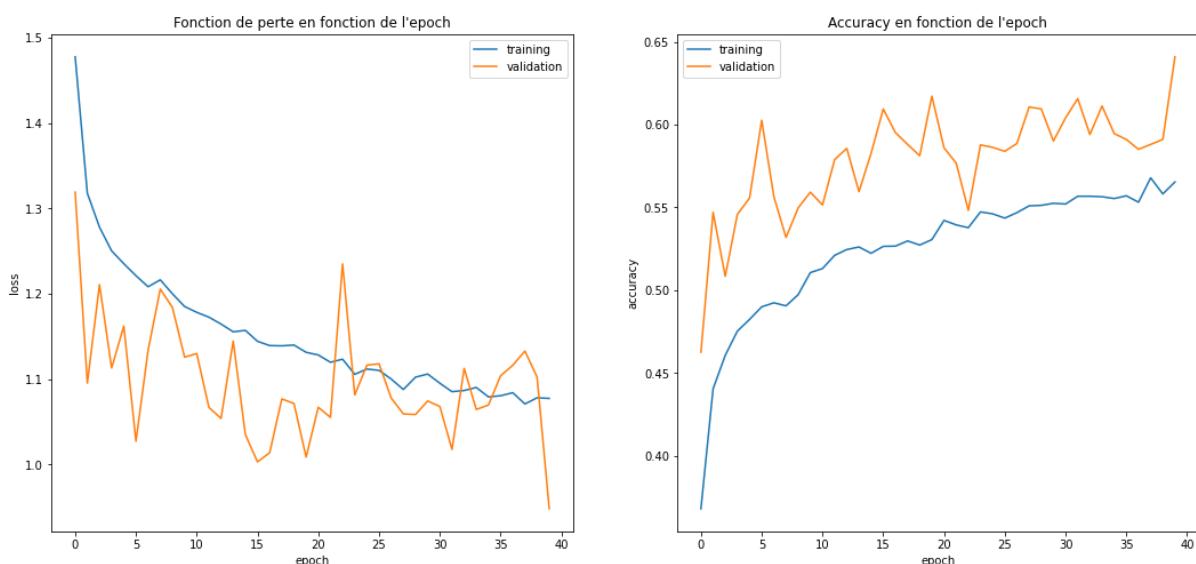


Figure 35: historique d'entraînement du modèle ResNet50

Ici également, la fonction de perte reste très élevée et les performances du modèles sont également très modestes : dernière époque d'entraînement :

```
Epoch 40/40
214/214 [=====] - 565s 3s/step - loss: 1.0768 - acc: 0.5655 -
val_loss: 0.9483 - val_acc: 0.6409
```

Inceptionv3

Présentation :

Le modèle Inception v3¹⁵, présenté lors de la compétition *ImageNet* de 2015 a été publié par Christian Szegedy et collaborateurs en 2016¹⁶. *Inception v3* est un réseau de neurones à convolution “classique” au premier abord en ce sens puisqu'il utilise une succession de couches convolutives et de pooling dans sa phase d'extraction des données et un réseau de neurones dans sa phase de classification. Néanmoins la particularité d'*Inception* réside dans la présence de modules “inception”. Habituellement, plusieurs paramètres peuvent être choisis pour une couche de convolution. En effet, par exemple, il faut déterminer la taille du kernel, qui lui même déterminera les paramètres extraits des images. Des tailles des 5x5, 3x3, 1x1 ou même 11x11 peuvent, par convention, être utilisées. Ainsi, au lieu de devoir faire un choix, les modules inception se composent comme suit: initialement, une première convolution avec un kernel de taille 1x1 est réalisée. À partir de cette convolution, deux nouvelles convolutions successives avec un kernel de taille 5x5 puis de taille 3x3 sont réalisées. En parallèle, à partir de la donnée d'entrée, un *average pooling* est réalisé. Finalement, l'ensemble des données sont regroupées dans une seule matrice par concaténation¹⁷.

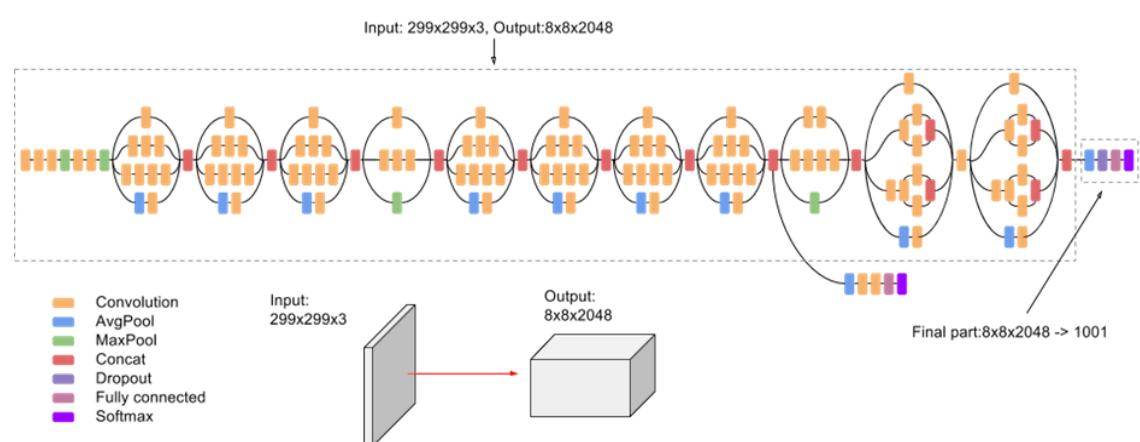


Figure 36: Architecture du modèle Inception v3¹⁸

¹⁵ <https://keras.io/api/applications/inceptionv3/>

¹⁶ Rethinking the Inception Architecture for Computer Vision

¹⁷ <https://steemit.com/fr/@rerere/comment-fonctionne-un-reseau-de-neurones-inception-v3-4>

¹⁸ <https://cloud.google.com/tpu/docs/inception-v3-advanced?hl=fr>

En appliquant l'approche transfer learning, nous avons donc entraîné un modèle dont Inception v3 est la base.

Résultats obtenus :

Fichiers d'intérêt :

- **Obtention du modèle :**
 - /models/iteration_1/20210707_model_inception_v3-datagenerator_GPU_mac.ipynb
- **Nom du modèle :**
 - model_inception_v3_GPU_20210617.h5
- **Evaluation du modèle :**
 - /models/iteration_1/20210707_evaluate_model_incepv3-datagenerator_GPU_mac.ipynb

Nous pouvons observer l'évolution de l'entraînement de notre modèle.

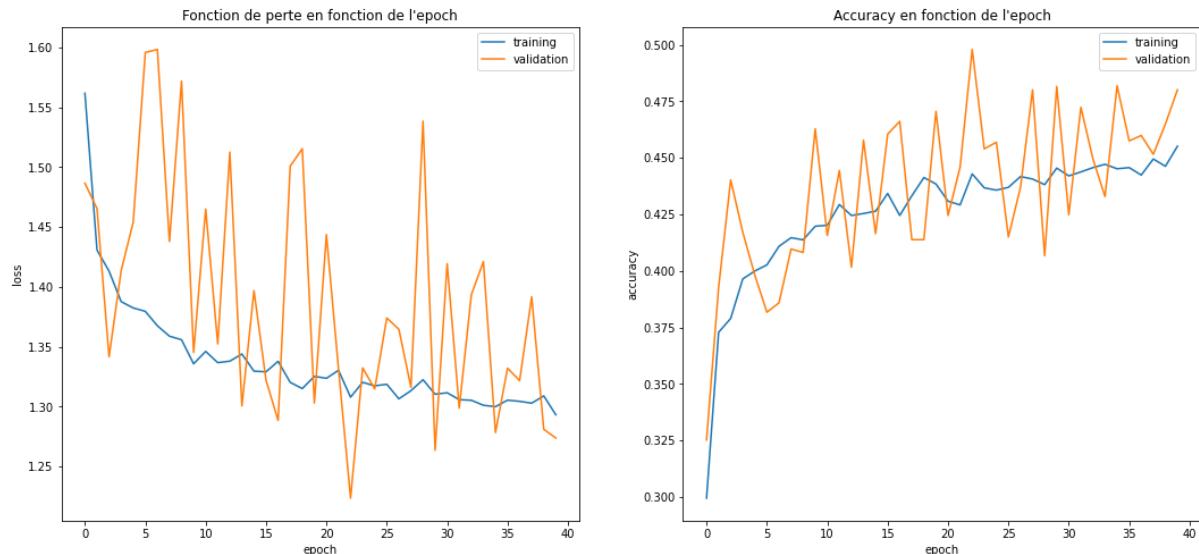


Figure 37: historique d'entraînement du modèle Inception v3

Ici nous pouvons constater que les performances sont probablement les plus mauvaises de nos essais, la fonction de perte reste très élevée et les performances du modèle sont également très modestes. Voici la dernière époque d'entraînement :

```
Epoch 40/40
214/214 [=====] - 271s 1s/step - loss: 1.2934 - acc: 0.4551 -
val_loss: 1.2739 - val_acc: 0.4801
```

Les modèles *ResNet50* et *Inception v3*, sont des modèles ayant démontré d'excellentes performances lors des challenges *ImageNet*.

Dans nos conditions, ces résultats que l'on pourrait qualifier de "mauvais" sur nos données soulèvent donc 2 questions :

1. Ces modèles sont-ils adaptés pour résoudre notre problématique ? La réponse est probablement oui.
2. Avons-nous identifié des paramètres convenables pour tester réellement ces modèles ? Cette fois, la réponse est probablement non. Nous pouvons donc raisonnablement penser qu'une seconde voie d'optimisation serait le travail plus en profondeur de ces 2 modèles dont les performances sont reconnues, notamment dans le cadre de la classification d'images.

EfficientNetB1

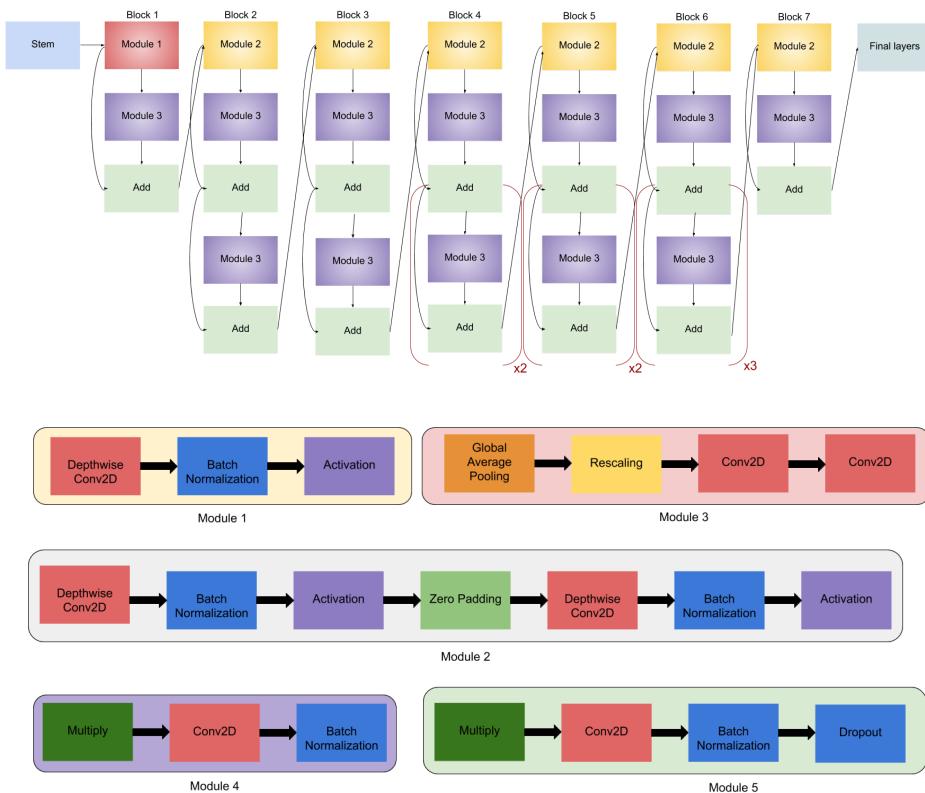
Présentation :

Le modèle *EfficientNetB1*¹⁹ a été développé et publié par M Tan et Q. V. Le²⁰. Les modèles *EfficientNet* sont au nombre de 8, compris entre B0 et B7. Chaque modèle a une itération différente incluant de plus en plus de paramètres. L'objectif de la mise en place des modèles *EfficientNet* a été de travailler sur la mise à l'échelle des modèles convolutifs. Cette mise à l'échelle est habituellement une manière d'accroître les performances des modèles. Néanmoins plusieurs paramètres peuvent être modifiés et les processus menant aux gains de performances sont rarement expliqués. Ainsi, les approches traditionnelles de mise à l'échelle se concentrent sur la **largeur du modèle** (un modèle plus large est supposé capturer des caractéristiques plus fines et être plus simple à entraîner), **sa profondeur** (un modèle plus profond est également supposé être capable de capter plus d'informations) ou encore la **Résolution des images d'entrée** (qui doit aussi permettre de capture des caractéristiques plus fines). Cependant, bien que cette mise à l'échelle s'accompagne en général d'une amélioration d'*accuracy*, elle s'accompagne également d'un coût de calcul plus important (mesuré en *FLOPS* pour "floating point operations per second"). Afin d'améliorer la mise à l'échelle, les auteurs proposent de travailler sur la mise à l'échelle des 3 paramètres simultanément. Cela se révèle très efficace selon les tests effectués tant sur le jeu de données d'*ImageNet* que sur des librairies d'images communément utilisées dans le cadre du transfer learning²¹.

¹⁹ <https://keras.io/api/applications/efficientnet/#efficientnetb1-function>

²⁰ EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, 2019

²¹ EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, 2019

Figure 38: Architecture du modèle EfficientNetB1²².

En appliquant l'approche transfer learning, nous avons donc entraîné un modèle dont EfficientNetB1 est la base.

Résultats obtenus :

Fichiers d'intérêt :

- **Obtention du modèle :**
 - /models/iteration_1/20210707_first_models_efficientnetB1.ipynb
- **Nom du modèle :**
 - checkpoint_effnet/model_efficientNet_20210617.h5
- **Evaluation du modèle :**
 - /models/iteration_1/20210707_evaluate_first_model_effnetb1.ipynb

Pour le modèle *EfficientNetB1*, il s'agit du tout premier modèle que nous avons entraîné avec une architecture de modèle un peu différente (nous sommes passés par la

²²

<https://towardsdatascience.com/complete-architectural-details-of-all-efficientnet-models-5fd5b736142>

fonction `Dataset.from_tensor_slices` pour créer nourrir notre modèle d'images, et nous avons réalisé l'entraînement en 2 temps: 10 époques puis 20 époques)

Voici l'architecture que nous avons utilisé :

Model: "sequential"		
Layer (type)	Output Shape	Param #
efficientnetb1 (Functional)	(None, 8, 8, 1280)	6575239
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense (Dense)	(None, 1024)	13111744
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 5)	2565

Total params: 8,414,348
Trainable params: 1,839,109
Non-trainable params: 6,575,239

Figure 39 : Summary du modèle EfficientNetB1

Nous avons donc obtenu une **accuracy** de 71% sur le set de test :

Epoch 20/20

642s 3s/step - loss: 0.1521 - accuracy: 0.9507 - val_loss: 1.0890
- val_accuracy: 0.7135

Ici, nous pouvons constater que les performances sont les meilleures de nos essais, la fonction de perte a été plutôt bien minimisée et les performances du modèle sont raisonnables.

Cette fois encore, il est intéressant de constater que nous n'arrivons pas à retrouver les mêmes métriques en sortie de modélisation et après loading du modèle :

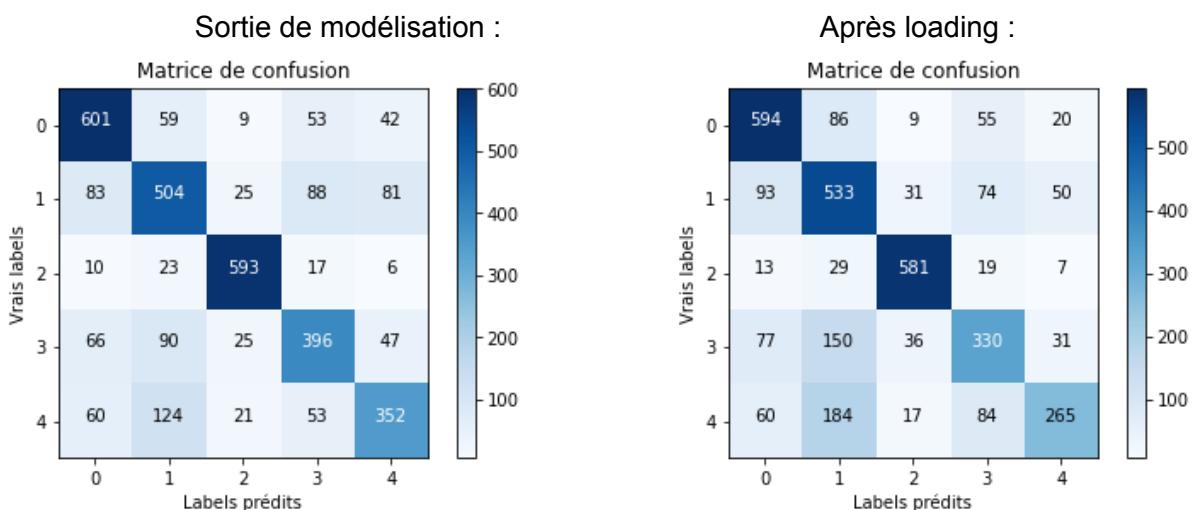


Figure 40 : Matrices de confusion à la sortie du modèle *EfficientNetB1* et après loading

Le rapport de classification indique également une *accuracy* plus basse après loading du modèle:

Classification report du modèle EfficientNetB1:

	precision	recall	f1-score	support
0	0.78	0.71	0.74	837
1	0.68	0.54	0.60	982
2	0.90	0.86	0.88	674
3	0.53	0.59	0.56	562
4	0.43	0.71	0.54	373
accuracy			0.67	3428
macro avg	0.66	0.68	0.66	3428
weighted avg	0.70	0.67	0.68	3428

Figure 41 : Rapport de classification après loading du modèle EfficientNetB1

Néanmoins, cette fois encore, la classe 0 et la classe 2 ont été les classes les plus performantes, ce qui correspond notamment à nos observations à l'œil.

Compte tenu des meilleures performances obtenues avec le modèle *EfficientNetB1*, nous avons choisi de poursuivre les étapes d'optimisation avec ce modèle.

Optimisation du modèle sélectionné :

Fichiers d'intérêt :

- **Obtention du modèle :**
 - /models/iteration_2/20210622_model_effnet-datagenerator_unfreeze_GPU_colab_initial.ipynb
 - /models/iteration_2/20210623_model_effnet-datagenerator_unfreeze_GPU_colab.ipynb
- **Nom du modèle :**
 - model_effnetB1_final_20210624.h5
- **Evaluation du modèle :**
 - /models/iteration_2/20210707_Predict_family_and_get_image_grad-cam.ipynb

Compte tenu des performances du modèle que nous avons construit avec *EfficientNetB1* pour modèle de base (*accuracy* à 71%), nous avons choisi de poursuivre avec ce modèle.

Lors de notre séquence d'entraînement, nous avons été face à 2 comportements :

- 1) Dans un premier cas un sur-apprentissage : le modèle *EfficientNetB1* avait une *accuracy* à 0,95 vs. *val_accuracy* à 0,71 - dans un contexte sans *image datagenerator* mais avec des *dropout* successives à 30 et 50%.
- 2) Dans un second cas nous avons enrayer le surapprentissage en ajoutant une étape *image data generator* en conservant le *dropout*. Nous avons alors souvent constaté de meilleures performances sur le set de test comparé au set d'entraînement.

Ainsi, nous avons choisi de réaliser notre phase d'optimisation de la modélisation en travaillant sur 3 aspects :

1. L'entraînement et le surentraînement : en variant l'*ImageDataGenerator* et les couches de *dropout*.
2. Les performances du test : en permettant l'ajustement de la dernière couche de convolution du modèle de base (*EfficientNetB1*).
3. L'interprétabilité en implémentant un algorithme Grad-Cam.

Afin de trouver un **équilibre entre un entraînement** du modèle performant et éviter le **sur-entraînement**, nous avons opté pour un équilibre mêlant l'implémentation d'un *image data generator* (propriétés : `rotation_range=5, width_shift_range = 0.1, height_shift_range = 0.05, zoom_range = 1.1`) et un ajustement des couches de *dropout* à la sortie des couches *denses*, en les passant de 30 et 50% à 20% chacune. De cette manière nous avons réussi à obtenir un équilibre entre un bon apprentissage tout en limitant le surapprentissage. Voici les performances finales :

```
Epoch 17/30
428/428 [=====] - 321s 749ms/step - loss: 0.6498 - accuracy: 0.7423 - val_loss: 0.6500 - val_accuracy: 0.7754
Restoring model weights from the end of the best epoch.
Epoch 00017: early stopping
```

Nous obtenons donc une *accuracy* et *val accuracy* similaires de 74,2 et **77,5%**.

Jusqu'à présent, nous avions réalisé l'entraînement d'un modèle qui employait la totalité des poids du modèle pré-entraîné *EfficientNetB1*. Afin de le rendre encore un peu plus adapté et spécifique à notre problème précis, nous avons souhaité permettre l'ajustement des poids de la dernière couche du modèle.

Cela "autorise" donc l'entraînement des couches spécifiées (ici : -1, donc la dernière uniquement).

Nous avons donc obtenu le modèle summary suivant :

Model: "sequential"		
Layer (type)	Output Shape	Param #
efficientnetb1 (Functional)	(None, 8, 8, 1280)	6575239
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense (Dense)	(None, 1024)	1311744
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 5)	2565

Total params: 8,414,348
Trainable params: 8,352,293
Non-trainable params: 62,055

Figure 42 : *Summary* du modèle *EfficientNetB1* avec ajustement des poids de la dernière couche du modèle

En établissant une comparaison avec l'architecture utilisée précédemment (partie *EfficientNetB1*) que nous avons maintenant un nombre bien plus important de paramètres que l'on peut entraîner, une diminution concomitante du nombre de paramètres qu'il n'est pas possible d'entraîner, le tout, comme attendu, avec un nombre de paramètres total qui ne change pas.

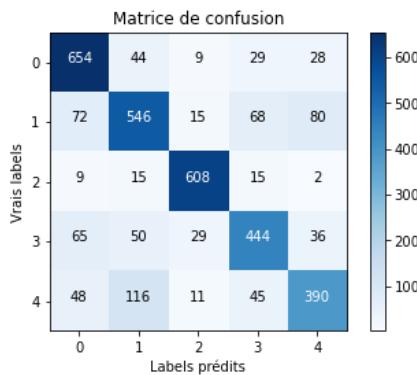
À l'issue de cet entraînement, nous avons donc constaté une amélioration de 6% de l'accuracy sur notre jeu de test, ce qui est une très nette amélioration. Constatant ce processus, nous avons souhaité approfondir la démarche et rendre les 2 dernières couches *optimisables* ("trainable"). Ce processus n'a pas abouti. En effet, les ressources informatiques nécessaires étaient trop importantes pour que nous puissions mener les tests complètement.

Voici donc les résultats après l'entraînement de notre modèle final :

Classification report du modèle EfficientNetB1 Final:

	precision	recall	f1-score	support
0	0.86	0.77	0.81	848
1	0.70	0.71	0.70	771
2	0.94	0.90	0.92	672
3	0.71	0.74	0.72	601
4	0.64	0.73	0.68	536
accuracy			0.77	3428
macro avg	0.77	0.77	0.77	3428
weighted avg	0.78	0.77	0.77	3428

Figure 43 : Rapport de classification et matrice de



confusion du modèle final avec EfficientNetB1 et ajustement des poids de sa dernière couche

Comme depuis le début, nous pouvons constater que les classes 0 et 2 demeurent les plus "faciles" à prédire alors que la classe 4 est souvent confondue avec la classe 1. Il est aussi très intéressant de noter que cette fois, le loading du modèle permet d'obtenir des résultats identiques que lors de l'entraînement. Cela soulève alors la question de la plateforme d'enregistrement. En effet, ce dernier modèle a été entraîné et sauvegardé à l'aide de *Google Colab*. Alors que les autres modèles ont été entraînés et sauvegardés sur un *macbook pro* (et parfois même avec une architecture toute particulière permettant d'utiliser le *GPU AMD Radeon Pro 5500M*).

Il est donc possible de regarder quelques exemples de classification :

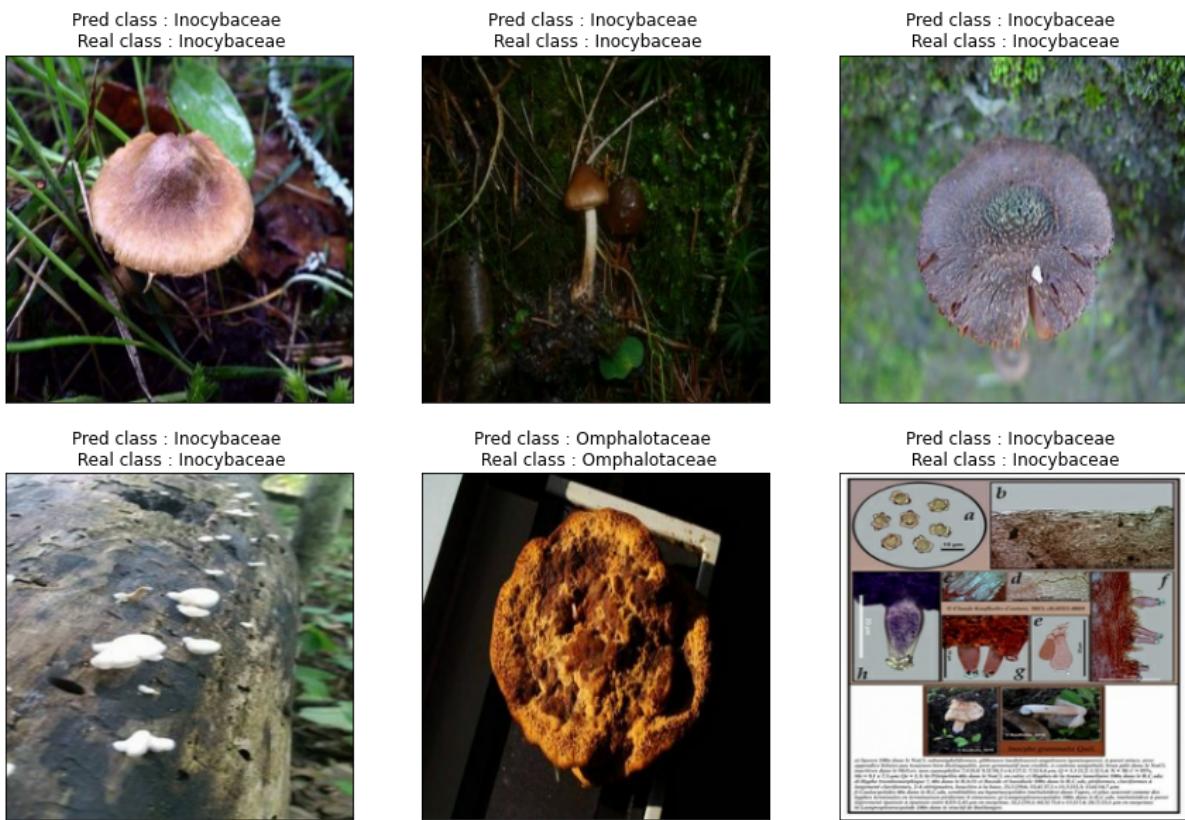


Figure 44: Exemple d'images assorties de leurs prédictions comparées aux classes réelles issu du modèle final

Cette fois encore, cette illustration d'images prises au hasard et classifiées est très intéressante. En effet, nous sommes face à une parfaite illustration de la disparité et de l'imperfection du jeu de données. Comme le souligne l'image en bas à droite, cette dernière ne représente pas un champignon.

Finalement, nous souhaitions pouvoir comprendre sur quelle base notre modèle s'appuyait pour réaliser les classifications.

Pour cela, nous nous sommes tournés vers l'algorithme Grad-CAM. Ce dernier est l'acronyme de Gradient-weighted Class Activation Map développé et publié par Ramprasaath R. Selvaraju en 2017²³. Cette approche provient d'une catégorie plus générale qui consiste à produire des heatmaps représentant les classes d'activation sur les images d'entrée. Une classe activation heatmap est associée à une classe de sortie spécifique. Ces classes sont calculées pour chaque pixel d'une image d'entrée, indiquant l'importance de chaque pixel par rapport à la classe considérée.

²³ Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization, 2017

En d'autres termes, il va être possible d'attribuer à chaque pixel son importance dans le processus de décision permettant d'attribuer la classe à l'objet.

Voici une illustration facilitant la compréhension du fonctionnement de Grad-CAM :

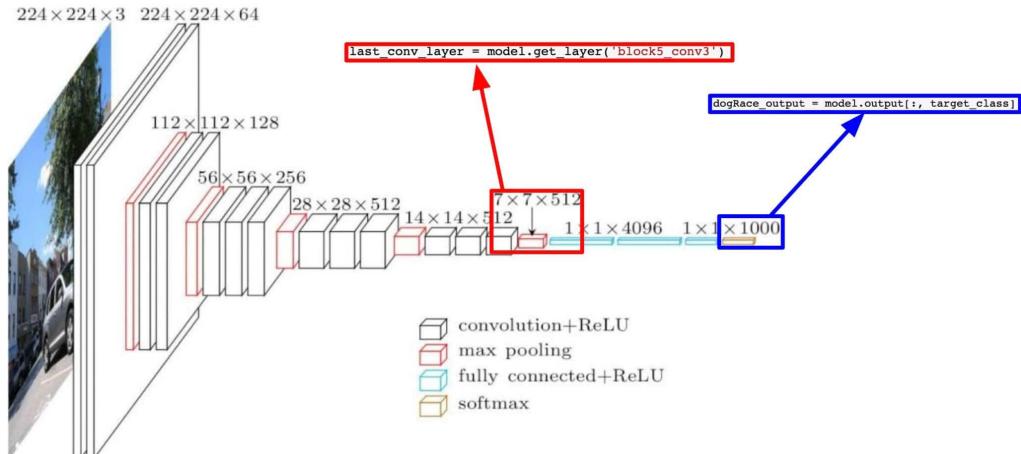


Figure 45: Images d'illustration pour la compréhension du fonctionnement de Grad-CAM

Ainsi, il faut aller chercher l'information de la sortie finale du modèle : quelle classe le modèle prédit-il (cadre bleu sur l'illustration). Une fois cette donnée connue, il faut retourner à la dernière couche de convolution du modèle utilisé (cadre rouge sur l'illustration) puis pour chaque pixel extraire la classe d'activation ayant permis de prédire cette classe spécifiquement. En réalisant cette opération pour chaque pixel de l'image, il est finalement possible de prédire par un jeu de couleur quels pixels ont été importants pour déterminer la classe finale.

Cette méthode se révèle très utile pour comprendre la pertinence du modèle : Est-ce que la prédiction se base sur l'objet d'intérêt ? Ou plutôt sur le fond de l'image ? Sur quoi se basent les prédictions ? Sur quoi reposent les échecs de prédiction ?

Voici donc quelques exemples de résultat du Grad-CAM :

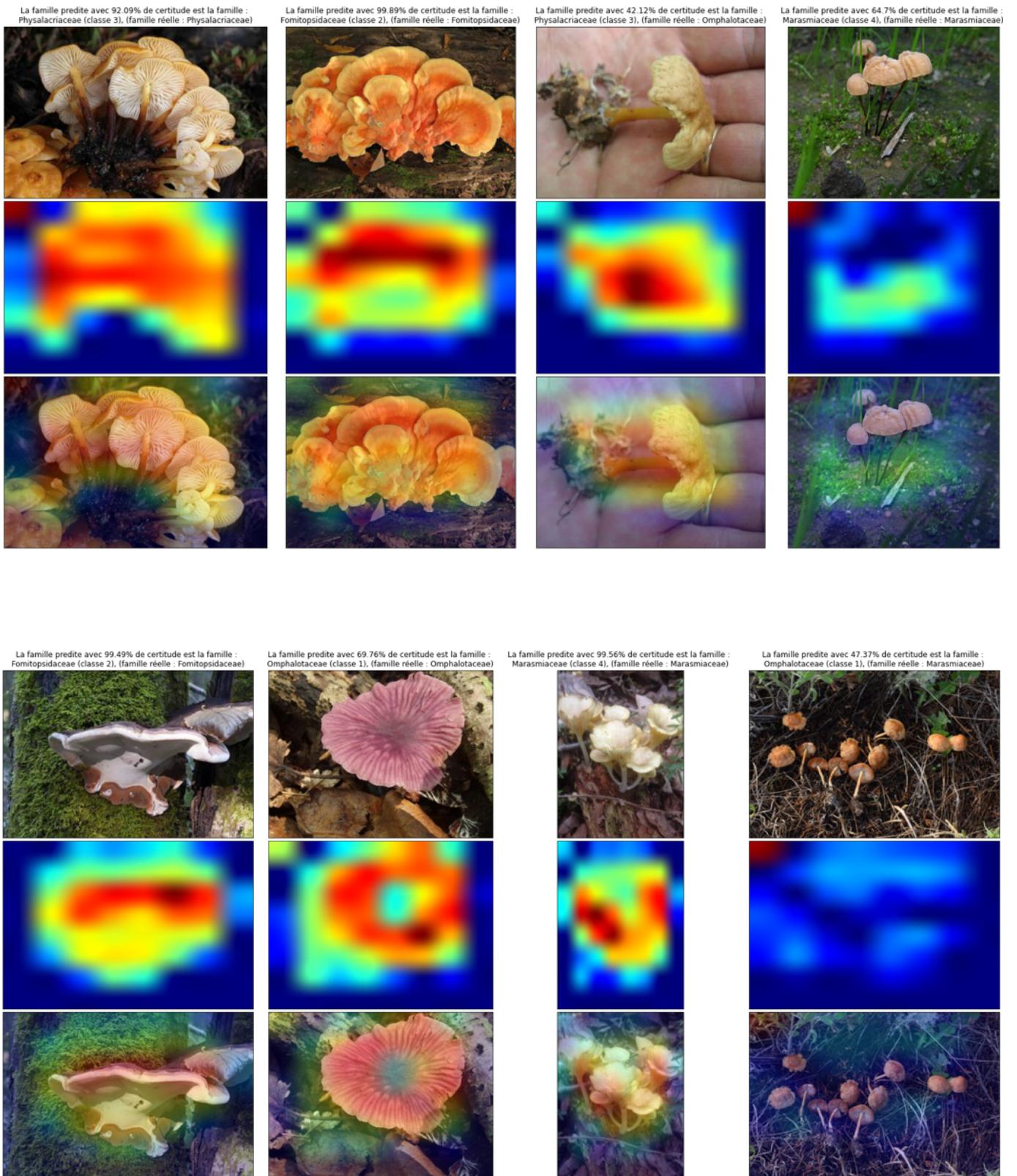


Figure 46: Exemples de résultat du *Grad-CAM* avec leurs prédictions sur la famille

Avec ces 8 exemples, nous pouvons constater avec plaisir que le *Grad-CAM* révèle que notre modèle se base sur des pixels pertinents dans une large proportion des cas. Néanmoins, nous pouvons constater sur la première ligne, 3^{ème} image malgré une carte d'activation pertinente (cible bien le champignon), le modèle fait une erreur de classification. Cela révèle donc une difficulté dans la classification et une confusion entre 2 classes. Il est cependant possible de relativiser cette erreur : on peut constater que le pourcentage de certitude est très bas (42,12%). Cela signifie donc que parmi les 5 probabilités disponibles, la plus haute n'est que de 42%, et donc que les autres probabilités rejetées sont également hautes (il faut partager les ~60% entre les 4 classes restantes). Le modèle a donc hésité ou tout au moins attribué une probabilité importante pour au moins une autre classe.

En poursuivant, toujours sur la première ligne, la 4^{ème} image (à droite), nous obtenons une prédiction correcte (avec cependant une faible certitude), mais une carte d'activation qui révèle que le modèle ne s'est que très peu basé sur le champignon pour réaliser sa prédiction. Dans ce cas, nous constatons également, que c'est le fond de l'image qui a servi à réaliser la prédiction ce que nous souhaiterions vraiment éviter.

Finalement, nous avons aussi un exemple, 2^{ème} ligne, dernière image où le modèle fait une erreur de prédiction, encore une fois avec un faible pourcentage de certitude, le tout combiné avec un très mauvais *Grad-CAM*. La mauvaise prédiction n'est alors pas surprenante.

Synthèse des résultats

D'un point de vue synthétique nous avons choisi de pousser le développement de la modélisation par transfert learning ayant comme base le modèle *EfficientNetB1*. C'est ainsi avec ce modèle que nous avons obtenu les meilleures performances (~77%). Dans le processus, nous avons donc pu constater que les outils tels que le *ImageDataGenerator* et les couches de *Dropout* peuvent être utilisés en synergie mais doivent être utilisés avec parcimonie pour ne pas entraver les bons déroulements de l'entraînement du modèle.

Nous avons donc obtenu un modèle qui n'est pas parfait mais dont les prédictions reposent en grande partie sur des zones pertinentes des images.

Cependant nous avons pu constater que ce modèle était d'une très bonne performance comparé aux autres modèles testés. En effet, les modèles *SVC* et *Random Forest* après une réduction de dimensions, ainsi que l'architecture de deep learning *LeNet* nous ont donné des résultats qui n'étaient guère mieux que le hasard.

Difficultés rencontrées lors du projet

Réduction du jeu de données : choix du focus (famille / genres)

En biologie, une taxonomie est une façon de définir des groupes basés sur des caractéristiques communes, et ces groupes sont organisés de façon hiérarchique, comme le montre la figure ci-dessous:

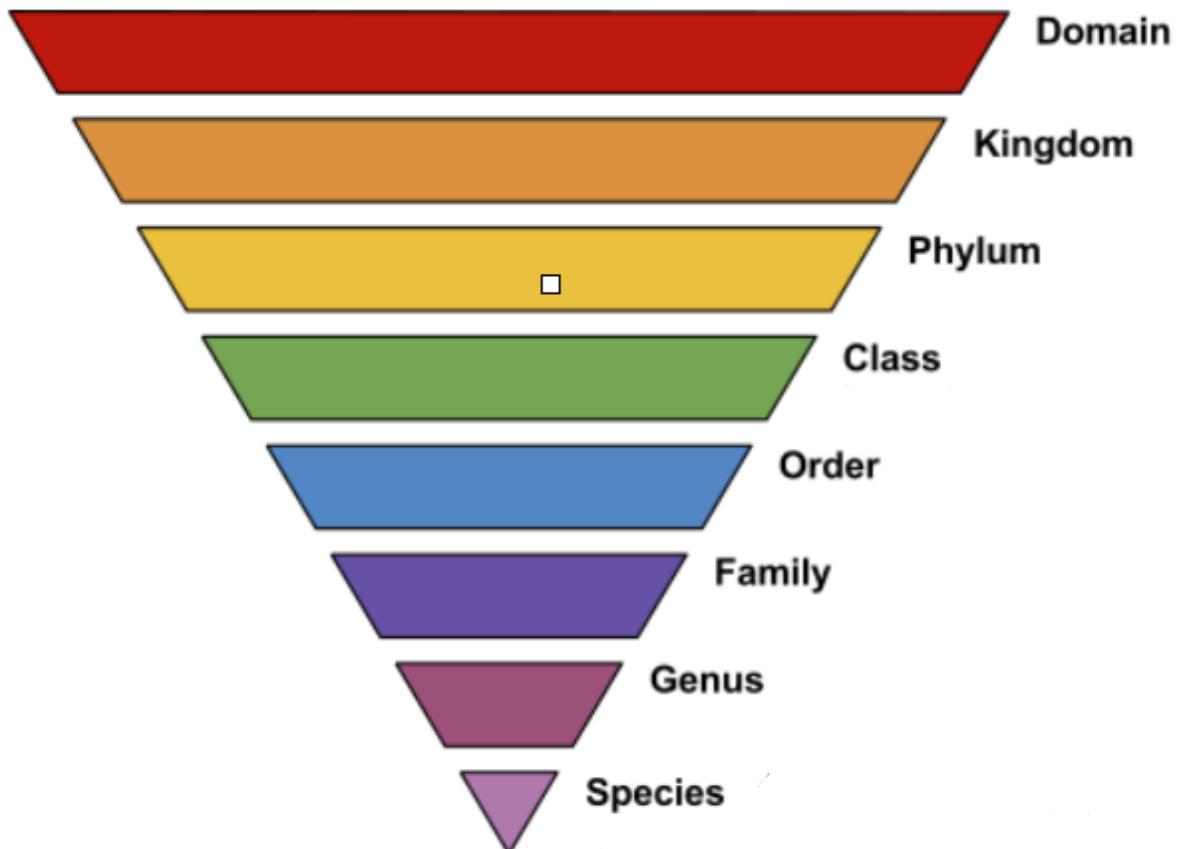


Figure 47 : Illustration des rangs taxonomiques de la classification scientifique du monde vivant²⁴

²⁴

https://upload.wikimedia.org/wikipedia/commons/thumb/7/71/Taxonomic_Rank_Graph.svg/555px-Taxonomic_Rank_Graph.svg.png

Dans notre jeu de données, la variable ‘rank’, déjà présente dans les documents JSON, indique le niveau taxonomique le plus spécialisé de chaque image (ayant atteint le consensus des amateurs sur le site internet). Il arrive aussi que plusieurs niveaux soient renseignés.

Si, par exemple, une image possède un ‘rank’ (ou rang) égal à *kingdom*, cela signifie que l'image pourra être classée, au mieux, avec ce niveau de taxonomie, par exemple *Fungi*.

Les images ne possèdent donc pas toutes le même rang, cela vient filtrer grandement nos données en fonction du niveau de taxonomie choisi.

Dans la partie qui suit, nous allons décrire la démarche que nous avons menée pour choisir à quel rang travailler avec notre jeu de données. En effet, au travers des différents taxons, les données avaient des distributions qui étaient parfois très déséquilibrées. Ces taxons étaient donc à éviter pour notre projet, car nous n'aurions jamais pu réaliser de bonnes performances de classification à leurs niveaux. Ainsi pour réduire notre dataset, la difficulté a été de pouvoir se placer à un rang taxonomique qui est à la fois équilibré et qui possède le plus d'images possible afin de pouvoir réaliser des entraînements de qualité.

Les graphiques ci-dessous donnent une idée de la distribution des images au sein des rangs de notre dataset. Nous choisissons de commencer notre démarche par le haut de la hiérarchie taxonomique. Nous avons déjà vu ce point lors de l'exploration des données, le ‘royaume’ *Fungi* écrase tous les autres par sa volumétrie. Comme *Fungi* est justement le ‘royaume’ des champignons, cela est donc parfaitement cohérent voire rassurant, de savoir que les données portent essentiellement sur des champignons. Ceci dit, en toute logique, il ne peut correspondre au bon taxon pour notre étude.

Alors il nous faut descendre d'un rang, le *phylum*, et commencer notre démarche de recherche d'équilibre dans les données. Le résultat est dans la figure ci-dessous:

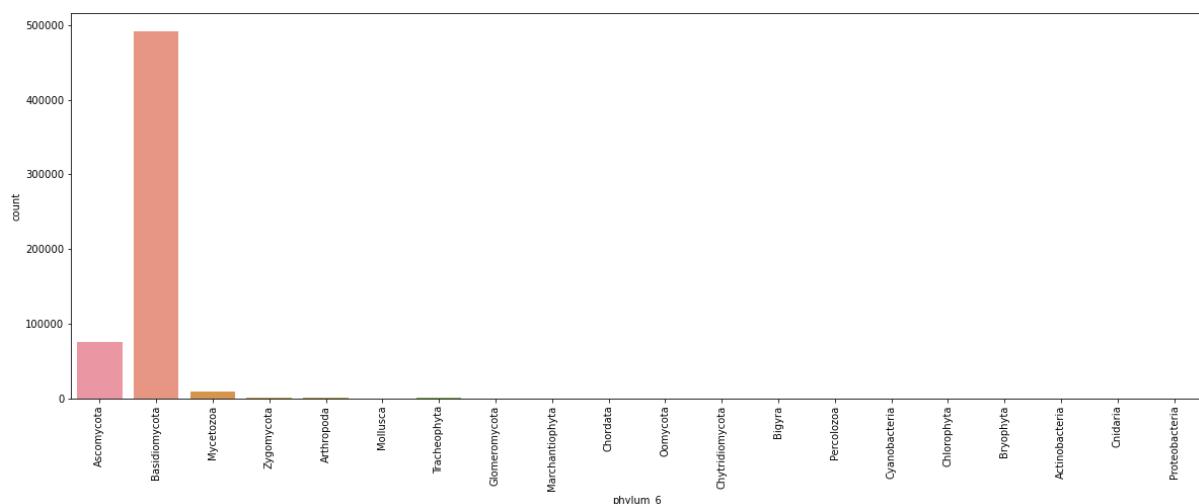


Figure 48 : Distribution des images au rang ‘phylum’

Nous pouvons remarquer que les données sont très déséquilibrées au niveau du taxon *phylum* pour réaliser une classification dans de bonnes conditions. De fait, il ne correspond pas à un bon niveau pour notre étude, sans compter que nous nous situons encore très haut dans la classification.

Il nous faut descendre d'un rang, la ‘classe’, pour voir si nous y trouvons une meilleure distribution.

Nous avons essayé de voir au niveau du taxon ‘classe’ qui se trouve dans le *phylum* le plus représenté, le *Basidiomycota*. Le résultat est dans la figure ci-dessous:

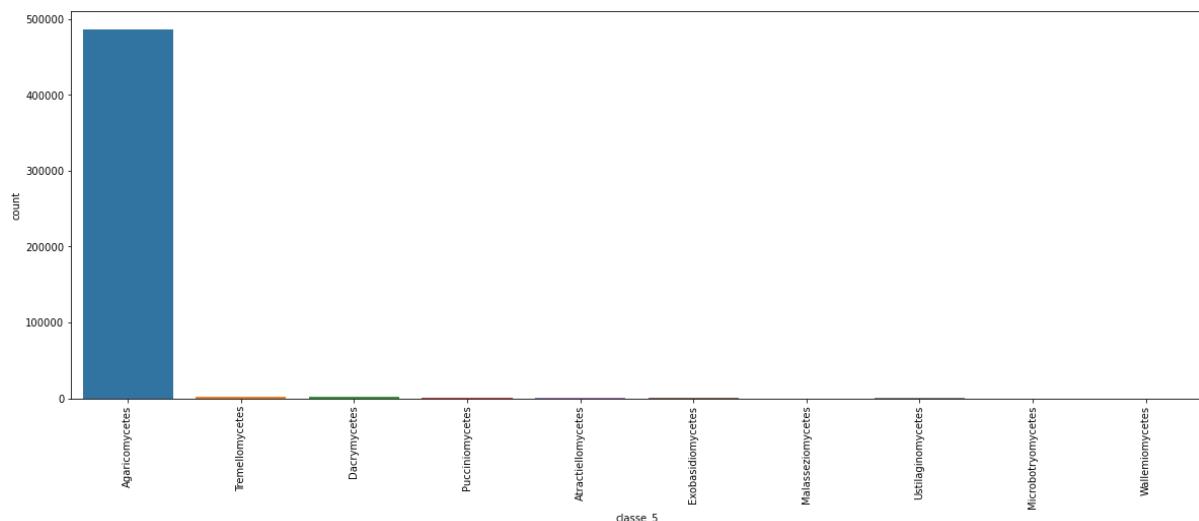


Figure 49 : Distribution des images au rang ‘classe’ dans le *phylum* le plus représenté

Les résultats sont à nouveau très mauvais pour réaliser une classification. La ‘classe’ *Agaricomycetes* est surreprésentée par rapport aux autres et une classification ne peut

fonctionner avec ce type de distribution. De fait, ce taxon ne correspond pas à un bon niveau pour notre étude.

Il nous faut descendre d'un rang, l'ordre, pour voir si nous y trouvons une meilleure distribution. Le résultat est dans la figure ci-dessous:

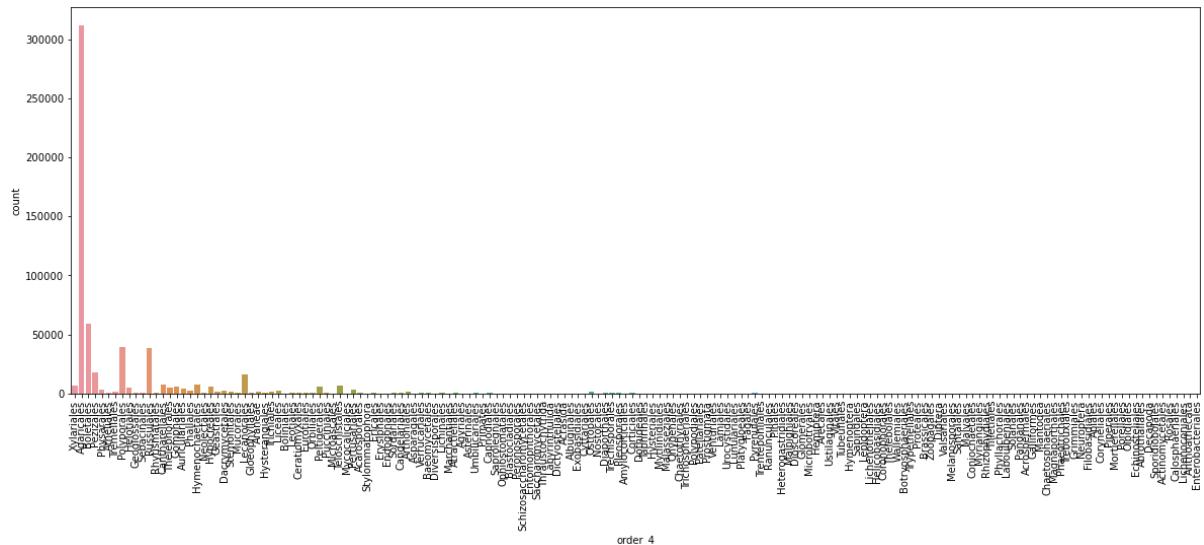


Figure 50 : Distribution des images au rang ‘ordre’

Les résultats sont toujours très mauvais pour réaliser une classification. L'ordre *Agaricales* est surreprésenté par rapport aux autres. De fait, ce taxon ne correspond toujours pas à un bon niveau pour notre étude.

Il nous faut descendre d'un rang, la ‘famille’, pour voir si nous y trouvons une meilleure distribution. Le résultat est dans la figure ci-dessous:

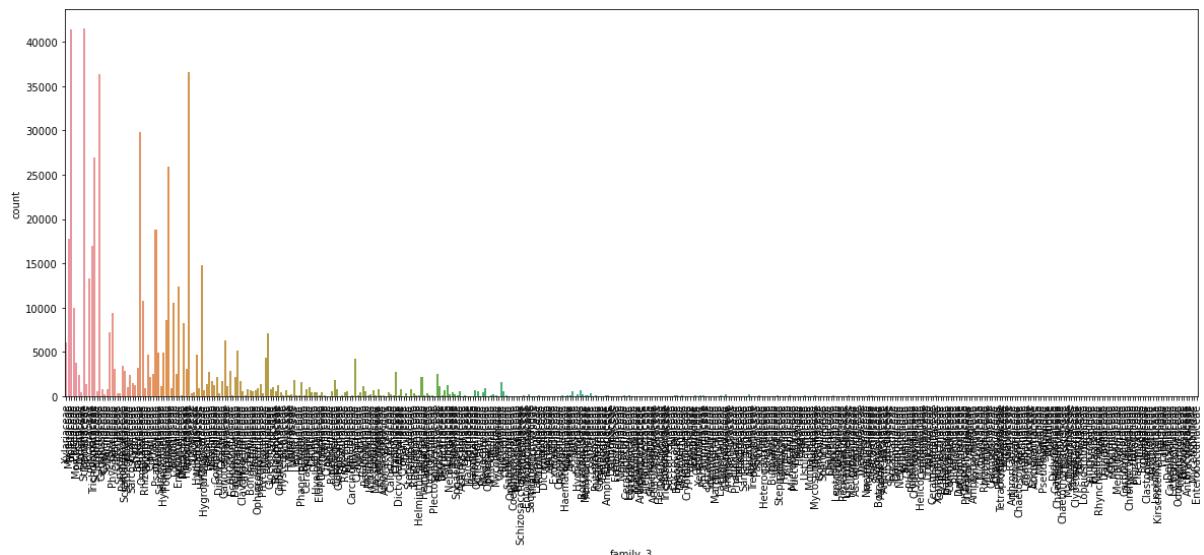


Figure 51 : Distribution des images au rang ‘famille’

Les résultats commencent à être intéressants pour réaliser une classification. Le taxon ‘famille’ est toujours loin d’être parfait mais il possède plusieurs familles qui ont un nombre d’images équivalent. Ces familles pourraient faire l’objet de notre étude. De fait, ce taxon peut correspondre à un bon niveau pour notre étude.

Descendons à nouveau d’un rang, le ‘genre’, pour voir si nous y trouvons encore une meilleure distribution. Le résultat est dans la figure ci-dessous:

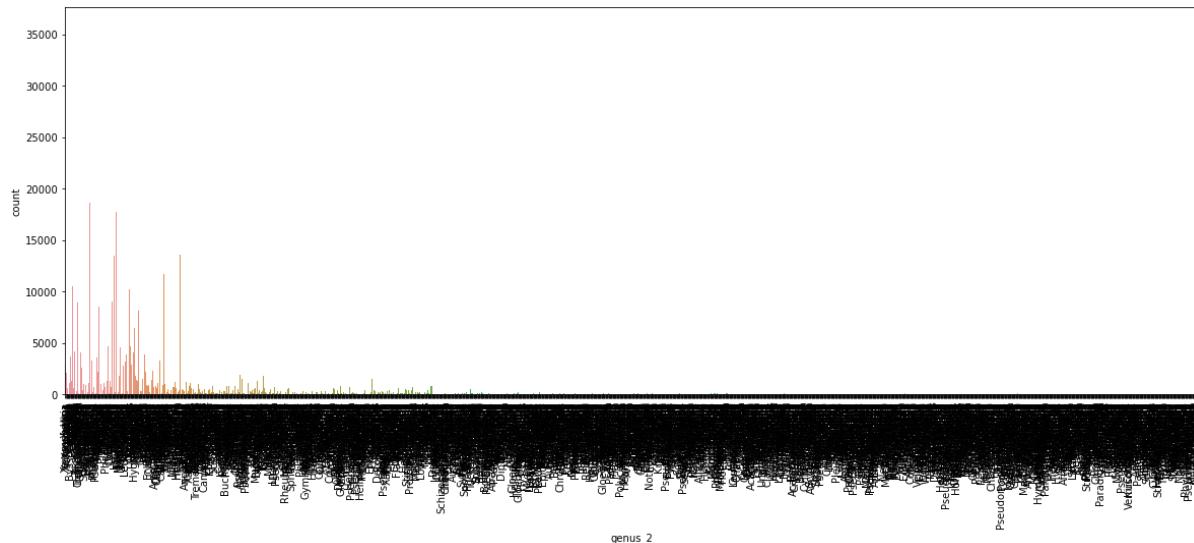


Figure 52 : Distribution des images au rang ‘genre’

Les résultats sont tout aussi intéressants pour réaliser une classification. Le taxon ‘genre’ est tout autant loin d’être parfait que ‘famille’ mais il possède aussi plusieurs genres qui ont un nombre d’images équivalent. Ces genres pourraient aussi faire l’objet de notre étude. De fait, ce taxon peut correspondre à un bon niveau pour notre étude.

Ceci dit, nous pouvons voir que nous arrivons au bout de notre démarche. En effet, le nombre d’images commence à baisser drastiquement. Si nous descendons plus bas dans les rangs taxonomiques nous n’aurons certainement pas suffisamment d’images pour faire nos modèles de classification.

Ainsi notre choix de taxon pour notre projet doit se faire entre les taxons ‘famille’ et ‘genre’. Les deux semblent intéressants et nous décidons de garder le taxon ‘famille’. Celui-ci présente un avantage que le ‘genre’ n’a pas. En effet, vous verrez dans la suite du rapport que nous avons essayé d’aller plus loin que de classer uniquement des familles, et nous avons réussi à classer des familles puis leurs genres. Effectivement, en nous plaçant au niveau de la famille, nous savons que nous avons suffisamment d’images pour classifier à la fois les familles mais aussi les genres. Alors que cela n’aurait pas été possible voire plus difficile, si nous avions voulu faire la même chose en partant du genre.

In fine, le classement sera donc fait par **famille** avec la répartition comme suit dans les rangs taxonomiques:

- 456 familles
- 1930 genres
- 8903 espèces.

Architecture et ressources informatiques :

Nous avons initié ce projet avec beaucoup d'ambitions ! Nous aurions aimé pouvoir tenter une classification sur l'ensemble des 650 000 images disponibles. Après une première estimation, nous nous sommes rapidement aperçus que cela n'était pas envisageable.

Ainsi pour notre première itération, nous avons sélectionné 17 familles différentes (tel que présenté précédemment), ce qui correspondait à un peu plus de 120 000 images. Néanmoins dès le tout premier essai, nous nous sommes confrontés à la problématique des ressources informatiques nécessaires pour traiter ce genre de problème : chaque époque d'entraînement nécessitait plusieurs heures.

Par conséquent, nous avons alors choisi de réduire à nouveau le set de données pour nous concentrer sur 5 familles seulement, ce qui représente tout de même un peu plus de 17 000 images.

Même en ayant réduit le jeu de données, nous avons encore été confrontés à des besoins importants en ressources informatiques le premier constat qui s'est imposé : ce genre de tâches d'entraînement nécessite absolument l'accès à des ressources GPU, sans quoi le code demande énormément de temps d'entraînement. Le gain à l'utilisation d'un GPU est très important.

Lors de la réalisation de ce projet nous nous sommes également confrontés à la diversité des infrastructures informatiques. Cette diversité impacte notamment :

1. Le fait d'être opérationnel rapidement (créer son environnement de travail efficace);
2. Partager efficacement les codes utilisés pour faire tourner les modèles.

Un exemple de la combinaison de ces problèmes de ressource et d'architecture a été rencontré afin de faire tourner les codes sur un Mac afin de bénéficier du GPU AMD Radeon Pro 5500M. En dehors d'une installation compliquée, cela aboutit (à priori) à des problèmes de sauvegarde des modèles lorsqu'ils sont chargés sur des architectures différentes.

Afin de pallier ces soucis, nous nous sommes tournés vers l'utilisation de Google Colab. Néanmoins, cette solution présente également ses défauts qui sont principalement : des limites d'utilisation (puisque l'accès aux ressources GPU est limité), la difficulté à charger les jeux de données (l'accès au fichier contenant les images est très difficile) et des déconnexions très (trop) fréquentes. Dans ce dernier cas, si nous avions oublié d'utiliser le callback "checkpoint", alors tout l'entraînement était complètement perdu et il fallait alors attendre d'avoir à nouveau une disponibilité GPU... soit principalement la nuit. La solution de Google Colab (en tout cas dans sa version gratuite), n'est pas une solution pérenne pour mener à bien un projet de ce type sereinement.

Problématique des premiers entraînements :

Les premiers essais de modélisation que nous avons réalisés se sont soldés par deux caractéristiques similaires : la stagnation et le surapprentissage.

La stagnation correspond au fait que le modèle n'arrive pas à prédire les labels à partir des images du jeu d'entraînement. On remarque cet état lorsque la valeur de la fonction de perte (*loss*) ne diminue plus et en parallèle la métrique d'évaluation (*accuracy* dans notre cas) ne s'améliore pas.

Le surapprentissage quant à lui est un processus associé à l'entraînement de notre modèle: le modèle prédit avec une certaine performance les labels des images du jeu d'entraînement mais prédit beaucoup moins bien les labels des images du jeu de test (images que le modèle n'a encore jamais vu et dont il ne peut pas se servir pour établir les caractéristiques utilisées pour les prédictions). On remarque cet état en observant l'entraînement de notre modèle : lorsque la métrique d'évaluation du modèle (*accuracy* dans notre cas) est bien meilleure pour le jeu d'entraînement que pour le jeu de test (*val_accuracy*).

Il se trouve que ces soucis étaient liés : pour la stagnation, nous avons oublié d'introduire la préparation d'images (*preprocess_input*) notamment pour les modèles *VGG16* et *VGG19*.

Concernant le surapprentissage, il existe différents moyens de lutter contre ce phénomène. Pour nos premiers essais, nous avons choisi d'opter pour l'ajout de couches de *dropout* à la sortie des couches *dense* de classification. Le *dropout* écarte aléatoirement une proportion (choisie par l'utilisateur) des caractéristiques utilisées pour faire la prédiction. On dit que l'on désactive une partie des neurones. Cela permet donc de ne pas se focaliser toujours sur les mêmes caractéristiques. En forçant le modèle à diversifier ses sources d'apprentissage, on réduit l'efficacité d'apprentissage sur le jeu d'apprentissage, mais

surtout on maximise nos chances de rendre notre modèle applicable sur d'autres images : on limite donc le surapprentissage de notre jeu de données.

Vie des modèles, itérations et gestion des données :

Nous avons également rencontré un problème dans la gestion des jeux d'entraînement et de test. Au fur et à mesure de l'avancement du projet, nous avons multiplié les approches et diversifié nos besoins. Nous avons ainsi pu faire deux constats :

1. Nous aurions dû séparer les données de manière plus "officielle" sans refaire à chaque fois l'étape du *train_test_split*. Il ne s'agit pas d'un bénéfice majeur, mais cela permet quand même d'appréhender avec plus de sérénité les différentes itérations en étant certain que les données sont séparées de manière identique.
2. Nous aurions surtout dû réaliser un jeu d'entraînement, un jeu de validation et un dernier jeu de test que le(s) modèle(s) n'auraient jamais vu quelles que soient les expériences et itérations.

En effet, comme nous pouvons le constater dans la partie "perspectives", nous avons souhaité implémenter une seconde étape qualifiée de méthode entonnoir. Elle consiste à prédire le genre du champignon, dans le but d'atteindre une classification plus poussée et précise. Pour cela, il faut avant toute chose avoir prédit la famille d'un champignon, repris la même image et essayé avec un nouveau modèle spécialement entraîné pour cela. Malheureusement, dans la préparation des données nous n'avons pas pris la problématique des données en considération et avons simplement extrait les images pour chaque genre et avons encore réalisé un *train_test_split*.

Ainsi, à la toute fin, lorsque nous évaluons notre enchaînement de modèle, nous reprenons les images test de notre premier modèle sur les familles. À partir de ce moment, nous ne savons pas si l'image test choisie a été utilisée comme image "train" ou image "test" pour entraîner le modèle sur le genre. Si nous avions procédé à la création d'un jeu d'entraînement, de validation et de test, nous aurions pu ne jamais toucher au jeu de test et ne l'utiliser que pour évaluer à la toute fin notre modèle et nos enchaînements de modèles.

Sauvegarde et chargement des modèles :

Dans le processus itératif de test de nos modèles, nous avons entraîné puis sauvegardé les modèles (à l'aide de la fonction *model.save("modelXX.h5")*). Pour le besoin de rédaction de ce rapport, nous avons poussé l'évaluation pour produire les visuels

proposés. D'un côté, les courbes de l'évolution de l'entraînement sont obtenues directement à l'issue de l'entraînement avant la sauvegarde du modèle et représentent donc le modèle tel qu'il a été sauvegardé. D'un autre côté, après chargement, nous avons souvent remarqué que l'*accuracy* mesurée n'était plus la même.

La première étape a été la vérification de notre jeu de test, afin de s'assurer que nous proposions une évaluation de notre modèle avec un jeu de données de test identique au jeu de test utilisé lors de l'entraînement. Malgré cette vérification, nous n'avons pas réglé le souci.

Nous avons donc procédé à une recherche sur internet. Il s'agit à priori d'un problème largement documenté, et nous avons trouvé ici un sujet disponible²⁵ parmi d'autres. Il est également possible que le problème soit issu de l'architecture informatique utilisée pour réaliser les entraînements.

Implémentation de l'algorithme Grad-CAM :

Tel que précédemment présenté, l'algorithme Grad-CAM permet de retrouver pour chaque pixel la classe d'activation ayant permis la classification finale. Afin de remplir cet objectif, il faut accéder à la dernière couche de convolution du modèle utilisé, hors dans notre cas, nous avons eu recours au transfer learning et avions donc un modèle *summary* qui ressemblait à :

Model: "sequential"		
Layer (type)	Output Shape	Param #
efficientnetb1 (Functional)	(None, 8, 8, 1280)	6575239
global_average_pooling2d (Gl)	(None, 1280)	0
dense (Dense)	(None, 1024)	1311744
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 5)	2565

Total params: 8,414,348
Trainable params: 8,352,293
Non-trainable params: 62,055

Figure 39 : Summary du modèle EfficientNetB1

Dans notre cas, la dernière couche de convolution est donc incluse dans le modèle de base, soit *EfficientNetB1* or, le code proposé pour implémenter le Grad-CAM ne prévoit

²⁵ <https://github.com/keras-team/keras/issues/4875>

pas cette possibilité. La recherche des erreurs obtenues sur internet s'est révélée particulièrement difficile et seul un sujet pertinent a finalement été trouvé, ce qui a permis de résoudre le problème

Bilan

Ce projet nous a permis d'aborder l'intégralité du déroulement d'un projet en Data Science. En partant d'une problématique, ici celle de la classification d'images de champignons, nous avons pu ainsi construire notre stratégie pour arriver à notre but. De plus, cela a été l'occasion de mettre en pratique les nouvelles compétences que nous apprenions grâce à la formation chez DataScientest et de développer nos réflexions. En effet, nous avons parfois exploré des pistes qui au premier abord n'étaient pas pertinentes, telles que les réductions de dimensions et l'architecture LeNet. Cependant, cela nous a permis de mettre en pratique nos nouvelles connaissances tout en balisant notre chemin vers la meilleure solution.

En outre, nous retiendrons que chaque étape est importante et que tout au long d'un projet, de l'exploration des données à l'élaboration du modèle final, les compromis sont omniprésents. Effectivement, ce projet nous a permis d'appréhender pour la première fois quelques difficultés rencontrées dans le cadre de modélisations issues de données de la "vie réelle". Nous avons été confrontés à plusieurs problématiques et problèmes allant du choix des données jusqu'à l'implémentation d'un algorithme interprétable pour notre modèle.

À travers cette expérience, nous avons pu identifier le transfer learning comme un processus plus efficace pour résoudre notre problème de classification. Néanmoins, bien que nous ayons abouti à une prédiction correcte dans plus de 75% des cas, la marge de progrès reste importante. Surtout qu'il faut conserver en mémoire que nous nous sommes concentrés uniquement sur une partie réduite de notre jeu de données (seulement 5 familles, ne représentant que 17 000 images sur les 650 000 initialement disponibles), et à une étape de classification relativement élevée dans l'arbre du vivant.

Pistes d'amélioration / perspectives :

Jeu de données :

Une première piste d'amélioration réside dans la qualité des images fournies au modèle, tant pour l'entraînement que pour le test. En effet, comme nous avons pu le constater tout au long de ce rapport certaines images sont de mauvaise qualité ou ne sont pas réellement des photos de champignons. Une première piste d'amélioration (et probablement une piste cruciale) réside dans l'amélioration du jeu de données.

Sauvegarde/chargement du modèle :

La recherche sur internet a permis d'identifier une potentielle solution pour pallier la divergence entre les performances constatées du modèle enregistré et les performances observées après chargement du même modèle. Il semblerait donc qu'une solution soit de réaliser à nouveau les entraînements avec la métrique *sparse_categorical_accuracy*²⁶ pour éviter la divergence entre le modèle sauvegardé et le modèle chargé.

Nombre de familles prédites :

Une autre piste d'amélioration de ce projet serait tout simplement d'inclure les 12 familles que nous avons rapidement exclues. Cela provoquerait un fort allongement du temps d'entraînement du modèle, mais maintenant que nous avons réalisé une première phase d'optimisation cela pourrait être intéressant d'essayer d'élargir notre fenêtre de prédictions.

Affiner la classification obtenue : prédire le genre

Une piste très intéressante d'amélioration réside dans l'objectif d'atteindre une classification plus fine. Initialement, nous avons choisi de nous focaliser sur l'échelle des familles dans la classification, notamment pour des raisons d'équilibre entre le nombre d'images et le nombre de classes que nous souhaitions prédire.

Maintenant, si nous souhaitions descendre un cran plus bas : prédire le genre, nous pouvons nous demander : quelle serait la meilleure approche ?

²⁶ <https://github.com/tensorflow/tensorflow/issues/42459>

Devrions-nous plutôt créer un nouveau modèle avec le même nombre d'images, mais avec plus de classes (ce qui implique nécessairement moins d'images par classe) ? Ou, devrions-nous plutôt opter pour une succession de prédictions :

- 1) Entraîner un premier modèle pour déterminer la famille et par la suite,
- 2) Pour chaque famille, entraîner un modèle spécifique à chaque famille afin de déterminer le genre d'un champignon (issu de cette famille).

Ainsi, la prédiction finale s'effectue en 2 étapes : prédiction à partir du modèle initial (permettant de déterminer la famille du champignon), qui conditionne le choix du second modèle utilisé (afin de prédire le genre du champignon).

Cette seconde approche présente l'avantage de procéder par étape avec chaque modèle possédant un nombre de classes limitées. Cela permet notamment de ne pas augmenter considérablement le nombre de classes à prédire sans augmenter le nombre d'images disponibles. Néanmoins, cette approche présente également le fort inconvénient de cumuler les potentielles erreurs. Par exemple, si l'on se place dans le cadre d'une erreur sur la famille, le genre prédit sera forcément mauvais.

Dans le cadre de ce projet, nous avons testé cette seconde solution que nous avons nommée : méthode entonnoir.

Première étape : génération des jeux de données :

Fichier d'intérêt :

- /EDA/Get_dataset_for genus_mushpy_20210626.ipynb

Afin de générer de nouveaux jeux de données pour entraîner 5 nouveaux modèles, nous nous sommes basés sur le jeu de données initial comprenant les 5 familles. Pour chaque famille, nous nous sommes alors intéressés au genre et avons exploré un peu les données. Puisque certains genres possédaient que très peu d'images, nous avons choisi de ne conserver que les genres possédant plus de 100 images.

En parallèle de ce premier filtre, nous avons tout de même constaté que les genres au sein des familles étaient relativement déséquilibrés tel qu'illustre la figure suivante :

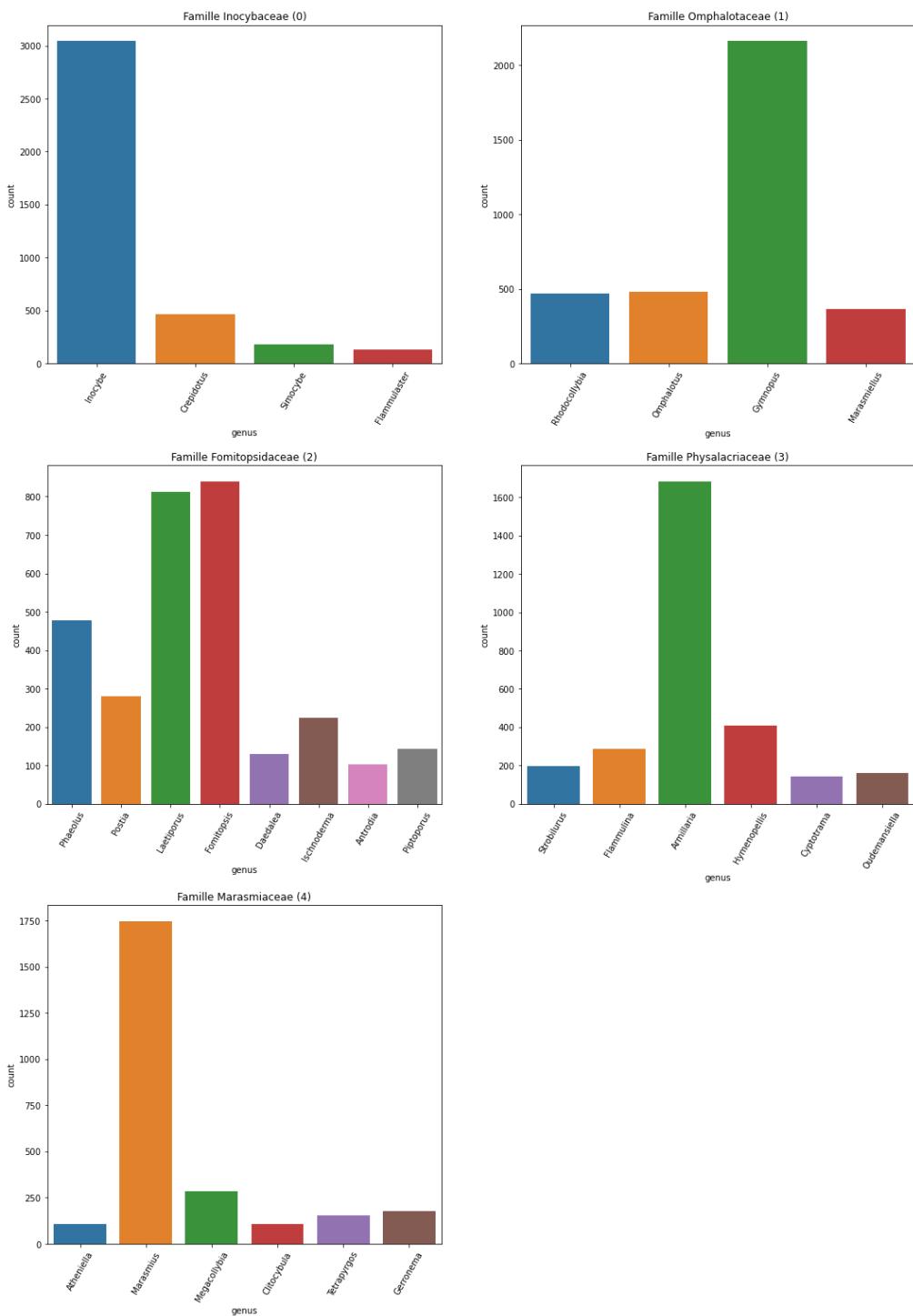


Figure 53 : Répartition des genres pour chaque famille.

Le fait de se baser sur le jeu de données initial complet avec les 5 familles a été une erreur puisque nous aurions dû mettre à côté un jeu de données n'ayant jamais été rencontré par le modèle. Notre méthode n'a pas pris en compte cette caractéristique. Il aurait donc fallu refaire les jeux de données et entraîner à nouveau les modèles.

Seconde étape : générer les nouveaux modèles :

Fichiers d'intérêt :

- **Modélisation :**
 - /models/perspectives_entonnoir/Modeles_to_predict_genus_20210627.ipynb
- **Modèles :**
 - model_label0_effnetB1_fin_20210627.h5
 - model_label1_effnetB1_fin_20210627.h5
 - model_label2_effnetB1_fin_20210627.h5
 - model_label3_effnetB1_fin_20210627.h5
 - model_label4_effnetB1_fin_20210627.h5
- **Evaluation des modèles :**
 - /models/perspectives_entonnoir/20210707_evaluate_models_genus.ipynb

Ainsi en utilisant l'architecture optimisée précédemment nous avons entraîné cinq modèles, un modèle par famille, pour prédire les genres de chacune des 5 familles sur lesquelles nous avons travaillé précédemment.

Lors de l'entraînement, nous avons obtenus les paramètres suivants :

- Famille 0 :

Epoch 28/40

```
62s 646ms/step - loss: 0.2689 - accuracy: 0.9118 - val_loss: 0.2497 - val_accuracy: 0.9280
```

Classification report du modèle Famille 0 :

	precision	recall	f1-score	support
0	0.98	0.97	0.98	599
1	0.91	0.91	0.91	104
2	0.81	0.81	0.81	37
3	0.77	0.92	0.84	25
accuracy			0.95	765
macro avg	0.87	0.90	0.88	765
weighted avg	0.96	0.95	0.95	765

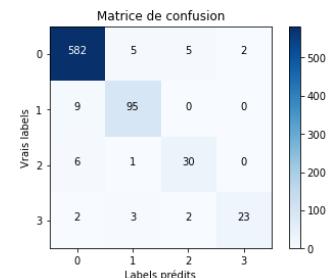


Figure 54 : Rapport de classification et matrice de confusion du modèle Famille

- Famille 1 :

Epoch 34/40

57s 651ms/step - loss: 0.3870 - accuracy: 0.8479 - val_loss: 0.4101 - val_accuracy: 0.8467

Classification report du modèle Famille 1 :

	precision	recall	f1-score	support
0	0.96	0.92	0.94	437
1	0.99	0.96	0.98	102
2	0.72	0.86	0.78	91
3	0.93	0.96	0.94	67
accuracy			0.92	697
macro avg	0.90	0.92	0.91	697
weighted avg	0.93	0.92	0.93	697

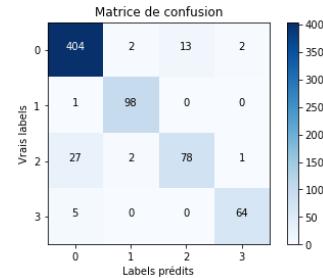


Figure 55 : Rapport de classification et matrice de confusion du modèle Famille 1

- Famille 2 :

Epoch 38/40

49s 643ms/step - loss: 0.5545 - accuracy: 0.8019 - val_loss: 0.5722 - val_accuracy: 0.8299

Classification report du modèle Famille 2 :

	precision	recall	f1-score	support
0	0.93	0.95	0.94	191
1	0.96	0.99	0.97	151
2	0.94	0.91	0.92	87
3	0.93	0.91	0.92	57
4	0.84	0.93	0.88	41
5	0.87	0.67	0.75	30
6	0.85	0.76	0.80	29
7	0.76	0.81	0.79	16
accuracy			0.92	602
macro avg	0.89	0.87	0.87	602
weighted avg	0.92	0.92	0.92	602

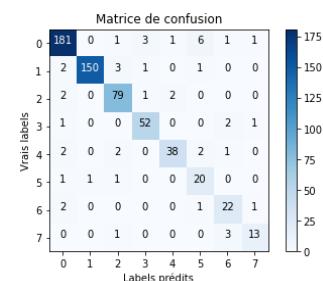


Figure 56 : Rapport de classification et matrice de confusion du modèle Famille 2

- Famille 3 :

Epoch 31/40

```
46s 645ms/step - loss: 0.3608 - accuracy: 0.8689 - val_loss:
0.3864 - val_accuracy: 0.8768
```

Classification report du modèle Famille 3 :				
	precision	recall	f1-score	support
0	0.98	0.98	0.98	342
1	0.93	0.98	0.95	86
2	0.97	0.97	0.97	60
3	0.90	0.85	0.88	33
4	0.90	0.86	0.88	22
5	0.97	1.00	0.98	32
accuracy			0.97	575
macro avg	0.94	0.94	0.94	575
weighted avg	0.97	0.97	0.97	575

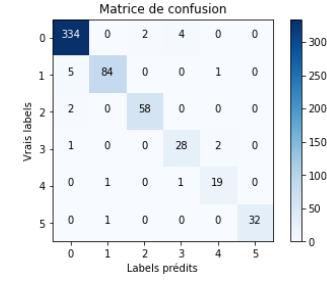


Figure 57 : Rapport de classification et matrice de confusion du modèle Famille 3

- Famille 4 :

Epoch 36/40

```
42s 646ms/step - loss: 0.3573 - accuracy: 0.8775 - val_loss:
0.4472 - val_accuracy: 0.8750
```

Classification report du modèle Famille 4 :				
	precision	recall	f1-score	support
0	0.95	0.97	0.96	339
1	0.98	0.88	0.93	58
2	0.97	0.93	0.95	40
3	0.85	0.88	0.86	40
4	0.80	0.94	0.86	17
5	0.72	0.65	0.68	20
accuracy			0.93	514
macro avg	0.88	0.87	0.87	514
weighted avg	0.93	0.93	0.93	514

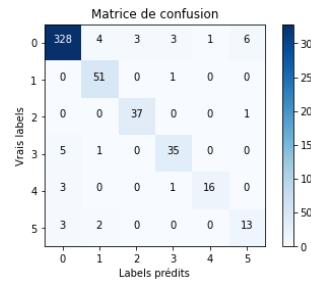


Figure 58 : Rapport de classification et matrice de confusion du modèle Famille 4

Nous pouvons constater que nos modèles ne souffrent pas particulièrement de surapprentissage et en plus que nous bénéficions d'une *accuracy* plutôt bonne, toujours supérieure à 82%. Il faut cependant se remémorer que nous avons des classes plutôt déséquilibrées. Nous avons veillé à utiliser l'*ImageDataGenerator* car il est indiqué dans la littérature que cette méthode permet notamment de contrer ce déséquilibre. Nous pouvons

aussi remarquer que l'efficacité du modèle reste meilleure que le hasard (même avec ce déséquilibre).

Cependant, nous pouvons aussi constater, encore une fois, que l'*accuracy* mesurée après loading du modèle diffère de l'*accuracy* en fin d'entraînement. Cette fois, les modèles ont été entraînés et enregistrés avec *Google Collab*. Cette disparité restera un mystère.

Troisième étape : réaliser les prédictions - méthode entonnoir :

Fichiers d'intérêt :

- /models/iteration_2/20210707_Predict_family_and_genus_and_get_image_grad-cam.ipynb

Une fois que nous avons mis au point nos modèles et évalué leur efficacité de prédiction, il est intéressant de mettre en pratique ces prédictions sur des images de notre jeu de données.

Nous avons donc développé une approche qui permet donc de prédire dans un premier temps la famille du champignon, puis en cascade, en fonction de cette famille de sélectionner un modèle pertinent pour évaluer le genre du champignon.

Attention ! A ce stade comme précédemment indiqué, nous avons un biais important dans les données. Nous ne pouvons pas certifier que pendant le processus d'entraînement, aucun modèle n'aït déjà été confronté à l'image de test choisie (aléatoirement) pour le processus de visualisation. Nous aurions donc 2 possibilités :

1. Extraire les images du `data_test` pour entraîner le modèle "famille" et, de ce jeu d'images, ne conserver que les images qui sont *aussi* présentes dans les `data_test` utilisés pour entraîner les modèles "genres".
2. Continuer le *Web scrapping* sur <http://mushroomobserver.org>. En effet, notre jeu de données initial ne contient que des images entre 2006 et 2017. Il est donc très probable que les familles / genres sélectionnés aient de nouvelles images.

Observons maintenant quelques exemples de classification :

La famille prédite avec 100.0% de certitude est la famille : Physalacriaceae (classe 3), (famille réelle : Physalacriaceae)

Le genre prédit avec 99.81% de certitude est le genre : Armillaria (classe 0), (genre réel : Armillaria)



La famille prédite avec 96.05% de certitude est la famille : Physalacriaceae (classe 3), (famille réelle : Physalacriaceae)

Le genre prédit avec 97.06% de certitude est le genre : Armillaria (classe 0), (genre réel : Armillaria)



La famille prédite avec 99.94% de certitude est la famille : Fomitopsidaceae (classe 2), (famille réelle : Fomitopsidaceae)

Le genre prédit avec 99.59% de certitude est le genre : Laetiporus (classe 1), (genre réel : Laetiporus)



La famille prédite avec 80.44% de certitude est la famille : Marasmiaceae (classe 4), (famille réelle : Marasmiaceae)

Le genre prédit avec 89.65% de certitude est le genre : Marasmius (classe 0), (genre réel : Marasmius)



La famille prédite avec 97.21% de certitude est la famille : Physalacriaceae (classe 3), (famille réelle : Physalacriaceae)

Le genre prédit avec 99.85% de certitude est le genre : Flammulina (classe 2), (genre réel : Flammulina)



La famille prédite avec 73.04% de certitude est la famille : Marasmiaceae (classe 4), (famille réelle : Marasmiaceae)

Le genre prédit avec 58.4% de certitude est le genre : Marasmius (classe 0), (genre réel : Marasmius)



La famille prédite avec 60.38% de certitude est la famille : Inocybaceae (classe 0), (famille réelle : Physalacriaceae)

Le genre prédit avec 99.93% de certitude est le genre : Inocybe (classe 0), (genre réel : Inocybe)



La famille prédite avec 42.67% de certitude est la famille : Inocybaceae (classe 0), (famille réelle : Marasmiaceae)

Le genre prédit avec 97.72% de certitude est le genre : Crepidotus (classe 1), (genre réel : Inocybe)

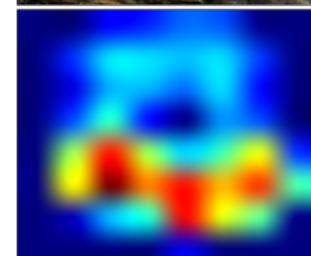
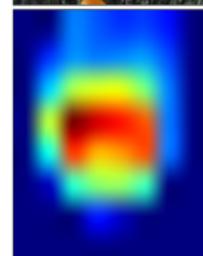


Figure 59: Exemples de résultat du Grad-CAM avec leurs prédictions sur la famille et le genre

Comme précédemment, il est possible de constater que cette approche fonctionne plutôt bien avec des prédictions qui se révèlent souvent justes, mais cette fois avec un Grad-CAM souvent pertinent. Il est aussi important de noter que le Grad-CAM proposé ici repose encore sur le modèle utilisé pour prédire la famille.

Les deux dernières images (2^{ème} ligne, images de droite) démontrent très bien la limite de cette approche : peu importe la "certitude" avec laquelle le 2^{ème} modèle propose une prédition, elle ne pourra jamais être correcte si la première prédition n'est pas juste.

Il est cependant intéressant de noter que le pourcentage de "certitude" pour la famille sur ces deux images est relativement faible. Nous aurions pu commencer par ajouter un seuil : si le pourcentage de certitude est inférieur à 50%, alors ne pas proposer de classification ET, en absence d'une première classification, ne pas proposer de classification pour le genre du champignon. Notons que dans notre situation, seule une image n'aurait pas été classifiée. Il est aussi possible de moduler le seuil : plus il est haut, plus on prend de précautions concernant notre résultat.

Deux concepts s'affrontent alors : une approche conservatrice visant à limiter les erreurs de classification, et une approche "exploratoire" visant à quand même proposer une classification en prenant le risque qu'elle ne soit pas juste.

L'objectif de la classification devrait déterminer l'approche : imaginons que cette classification ait pour objet d'aider un chasseur de champignons amateur à reconnaître les champignons qu'il a pris en photo. Dans ce cas, nous pourrions adopter l'approche exploratoire puisque les conséquences d'une erreur sont très limitées. Comme précédemment, il est possible de constater que cette approche fonctionne plutôt bien avec des prédictions qui se révèlent souvent justes, et plus important, avec un Grad-CAM souvent pertinent. Il est aussi important de noter que le Grad-CAM proposé ici repose encore sur le modèle utilisé pour prédire la famille.

Les deux dernières images (2^{nde} ligne, images de droite) démontrent très bien la limite de cette approche : peu importe la "certitude" avec laquelle le 2nd modèle propose une prédition, elle ne pourra jamais être correcte si la première prédition n'est pas juste.

Il est cependant intéressant de noter que le pourcentage de "certitude" pour la famille sur ces deux images est relativement faible. Nous aurions pu commencer par ajouter un seuil : si le pourcentage de certitude est inférieur à 50% alors ne pas proposer de classification ET, en absence d'une première classification, ne pas proposer de classification pour le genre du champignon. Notons que dans notre situation, seule une image n'aurait pas été classifiée. Il est aussi possible de moduler le seuil : plus il est haut, plus on prend de précautions concernant notre résultat.

Deux concepts s'affrontent alors : une approche conservatrice visant à limiter les erreurs de classification, et une approche "exploratoire" visant à quand même proposer une classification en prenant le risque qu'elle ne soit pas juste.

L'objectif de la classification devrait déterminer l'approche : imaginons que cette classification ait pour objet d'aider un chasseur de champignons amateur à reconnaître les champignons qu'il a pris en photo. Dans ce cas, nous pourrions adopter l'approche exploratoire puisque les conséquences d'une erreur sont très limitées. A l'inverse nous pourrions imaginer combiner notre modèle à d'autres informations afin d'aboutir à la prédiction de la toxicité d'un champignon : prenons en photo notre champignon et attendons la réponse : peut-il être consommé en toute sécurité. Dans ce cas, nous avons besoin d'une approche très conservatrice et d'être sur la classification proposée.

Possibles applications

En l'état actuel les applications de notre modèle sont relativement limitées tant la classification ne concerne qu'une très faible quantité des familles existantes. Néanmoins, en accédant à des ressources informatiques plus conséquentes, il serait possible de développer plus en profondeur cette approche et ainsi élargir le spectre de la prédiction. Comme précédemment, il est possible de constater que cette approche fonctionne plutôt bien avec des prédictions qui se révèlent souvent justes, et plus important, avec un Grad-CAM souvent pertinent. Il est aussi important de noter que le Grad-CAM proposé ici repose encore sur le modèle utilisé pour prédire la famille.

Les deux dernières images (2^{nde} ligne, images de droite) démontrent très bien la limite de cette approche : peu importe la "certitude" avec laquelle le 2nd modèle propose une prédiction, elle ne pourra jamais être correcte si la première prédiction n'est pas juste.

Il est cependant intéressant de noter que le pourcentage de "certitude" pour la famille sur ces deux images est relativement faible. Nous aurions pu commencer par ajouter un seuil : si le pourcentage de certitude est inférieur à 50% alors ne pas proposer de classification ET, en absence d'une première classification, ne pas proposer de classification pour le genre du champignon. Notons que dans notre situation, seule une image n'aurait pas été classifiée. Il est aussi possible de moduler le seuil : plus il est haut, plus on prend de précautions concernant notre résultat.

Deux concepts s'affrontent alors : une approche conservatrice visant à limiter les erreurs de classification, et une approche "exploratoire" visant à quand même proposer une classification en prenant le risque qu'elle ne soit pas juste.

L'objectif de la classification devrait déterminer l'approche : imaginons que cette classification ait pour objet d'aider un chasseur de champignons amateur à reconnaître les champignons qu'il a pris en photo. Dans ce cas, nous pourrions adopter l'approche exploratoire puisque les conséquences d'une erreur sont très limitées. À l'inverse nous pourrions imaginer combiner notre modèle à d'autres informations afin d'aboutir à la prédiction de la toxicité d'un champignon : prenons en photo notre champignon et attendons la réponse : peut-il être consommé en toute sécurité. Dans ce cas, nous avons besoin d'une approche très conservatrice sur la classification proposée.

Bibliographie

Articles et Tutoriels

1. <https://mushroomobserver.org/>
2. <https://github.com/bechtle/mushroomobser-dataset>
3. <https://images.mushroomobserver.org/320/39.jpg>
4. <https://stackoverflow.com/questions/3490727/what-are-some-methods-to-analyze-image-brightness-using-python>
5. Stevo. Bozinovski and Ante Fulgosi (1976). "The influence of pattern similarity and transfer learning upon the training of a base perceptron B2." (original in Croatian) Proceedings of Symposium Informatica 3-121-5, Bled
6. <https://image-net.org/about.php>
7. <https://keras.io/api/applications/vgg/#vgg16-function>
8. <https://www.datacorner.fr/vgg-transfer-learning/>
9. <https://keras.io/api/applications/vgg/#vgg19-function>
10. <https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11>
11. <https://keras.io/api/applications/resnet/#resnet50-function>
12. <https://neurohive.io/en/popular-networks/resnet/>
13. Publiée : Deep Residual Learning for Image Recognition, He et al 2016
14. Deep Residual Learning for Image Recognition, He et al 2016
15. <https://keras.io/api/applications/inceptionv3/>
16. Rethinking the Inception Architecture for Computer Vision
17. <https://steemit.com/fr/@rerere/comment-fonctionne-un-reseau-de-neurones-inception-v3-4>
18. <https://cloud.google.com/tpu/docs/inception-v3-advanced?hl=fr>
19. <https://keras.io/api/applications/efficientnet/#efficientnetb1-function>
20. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, 2019
21. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, 2019
22. <https://towardsdatascience.com/complete-architectural-details-of-all-efficientnet-models-5fd5b736142>
23. Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization, 2017

24. https://upload.wikimedia.org/wikipedia/commons/thumb/7/71/Taxonomic_Rank_Graph.svg/555px-Taxonomic_Rank_Graph.svg.png
25. <https://github.com/keras-team/keras/issues/4875>
26. <https://github.com/tensorflow/tensorflow/issues/42459>

Table des figures

Figure 01 : Structure des fichiers JSON	6
Figure 02 : Distribution des images par thumbnail	7
Figure 03 : Distribution des images par royaume	8
Figure 04 : Nombre des images par royaume	8
Figure 05 : Répartition des images par famille avec les données sélectionnées	9
Figure 06 : Nombre d'images par famille avec les données sélectionnées	9
Figure 07 : Nombre d'images par genre avec les données sélectionnées	10
Figure 08 : Nombre d'images par famille avec plus de 3000 images	11
Figure 09 : Les 5 familles retenues pour notre étude	12
Figure 10 : Approches possibles pour mesurer la luminosité d'une image	13
Figure 11 : Distribution de la luminosité des images de 17 familles de champignons contenues dans le jeu de données sélectionné.	14
Figure 12 : Distribution de la luminosité des images des 5 familles de champignons contenues dans le jeu de données sélectionné (restreint).	14
Figure 13 : Exemples d'images ayant le label 0	16
Figure 14 : Exemples d'images ayant le label 1	16
Figure 15 : Exemples d'images ayant le label 2	17
Figure 16 : Exemples d'images ayant le label 3	17
Figure 17 : Exemples d'images ayant le label 4	18
Figure 18 : Images de nos labels pour comparaison	18
Figure 19 : Projections de nos images sur un plan après Isomap	20
Figure 20 : Projections de nos images sur un plan après PCA	21
Figure 21 : Rapports de classification des modèles SVC et Random Forest après réduction des dimensions	24

Figure 22 : Matrices de confusion des modèles SVC et Random Forest après réduction des dimensions	24
Figure 23 : L'image originale publiée dans [LeCun et al., 1998]	25
Figure 24 : Summary du modèle Lenet	26
Figure 25: historique d'entraînement du modèle Lenet	27
Figure 26: Rapport de classification et matrice de confusion du modèle Lenet	27
Figure 27 : Summary type issu du transfer learning	30
Figure 28: Architecture du modèle VGG16	32
Figure 29: historique d'entraînement du modèle VGG16	32
Figure 30: Rapport de classification et matrice de confusion du modèle VGG16	33
Figure 31: Exemple d'images assorties de leurs prédictions comparées aux classes réelles issu du modèle VGG16	34
Figure 32: Comparaison entre l'architecture des modèles VGG existants	35
Figure 33: historique d'entraînement du modèle VGG19	36
Figure 34: Architecture du modèle ResNet50 publiée par les inventeurs, en comparaison avec le modèle VGG19 (à gauche), un modèle “conventionnel” à 34 couches (au centre) et le modèle ResNet50 (à droite) illustrant les connexions par saut additionnant l'entrée à la sortie du bloc résiduel	38
Figure 35: historique d'entraînement du modèle ResNet50	39
Figure 36: Architecture du modèle Inception v3	40
Figure 37: historique d'entraînement du modèle Inception v3	41
Figure 38: Architecture du modèle EfficientNetB1	43
Figure 39 : Summary du modèle EfficientNetB1	44, 61
Figure 40 : Matrices de confusion à la sortie du model EfficientNetB1 et après loading	44
Figure 41 : Rapport de classification après loading du modèle EfficientNetB1	45
Figure 42 : Summary du modèle EfficientNetB1 avec ajustement des poids de la dernière couche du modèle	47
Figure 43 : Rapport de classification et matrice de confusion du modèle final avec EfficientNetB1 et ajustement des poids de sa dernière couche	48
Figure 44: Exemple d'images assorties de leurs prédictions comparées aux classes réelles issu du modèle final	49
Figure 45: Images d'illustration pour la compréhension du fonctionnement de Grad-CAM	50
Figure 46: Exemples de résultat du Grad-CAM avec leurs prédictions sur la famille	51
Figure 47 : Illustration des rangs taxonomiques de la classification scientifique du monde vivant	53
Figure 48 : Distribution des images au rang 'phylum'	55
<i>Bootcamp DS - mai 2021</i>	78

Figure 49 : Distribution des images au rang ‘classe’ dans le ‘phylum’ le plus représenté	55
Figure 50 : Distribution des images au rang ‘ordre’	56
Figure 51 : Distribution des images au rang ‘famille’	56
Figure 52 : Distribution des images au rang ‘genre’	57
Figure 53 : Répartition des genres pour chaque famille	66
Figure 54 : Rapport de classification et matrice de confusion du modèle Famille 0	67
Figure 55 : Rapport de classification et matrice de confusion du modèle Famille 1	67
Figure 56 : Rapport de classification et matrice de confusion du modèle Famille 2	68
Figure 57 : Rapport de classification et matrice de confusion du modèle Famille 3	68
Figure 58 : Rapport de classification et matrice de confusion du modèle Famille 4	69
Figure 59: Exemples de résultat du Grad-CAM avec leurs prédictions sur la famille et le genre	71