

Rapport technique d'évaluation - 23 Février 2022

Pyck a Champy - Mushroom Recognition

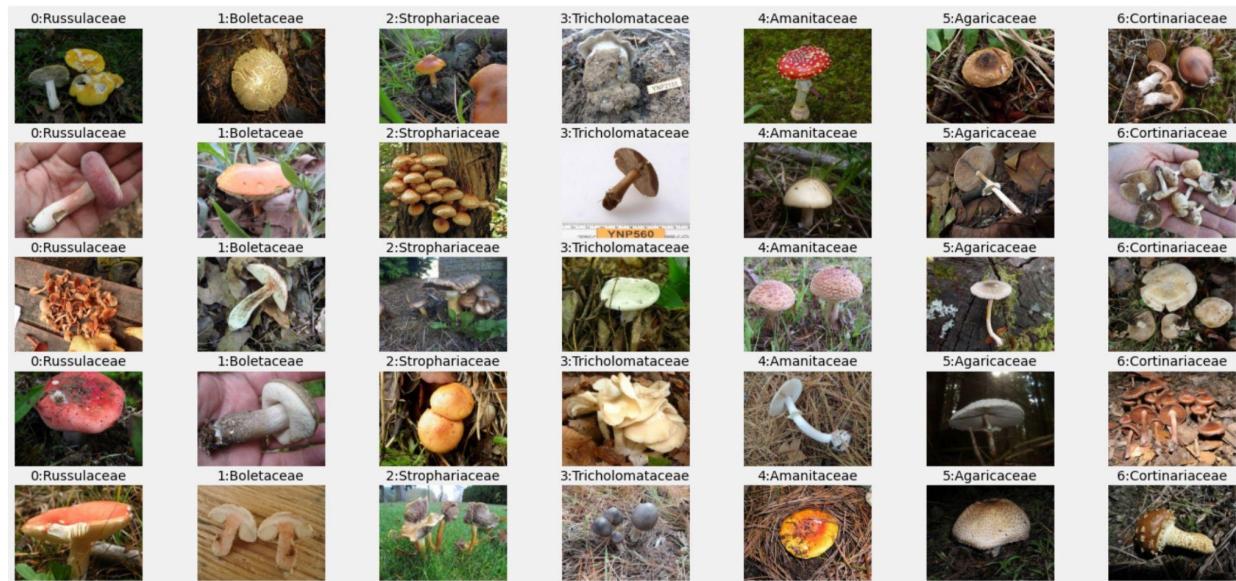


fig1. 7 familles de champignons

Promotion: Data Scientist - Bootcamp - décembre 2021

Participants: Sébastien Thibert, Antoine Poirot-Bourdain et Vincent Siohan

Tuteur: Louis

Notebooks disponibles sur GitHub:

https://github.com/DataScientest-Studio/Pyck_a_Champy

SOMMAIRE

SOMMAIRE	2
CONTEXTE	4
OBJECTIFS	5
DATA	6
Construction de la base de données	6
EXPLORATION DES DONNÉES ET VISUALISATION	9
Taxonomie de la base de donnée	9
Résolution des images	9
Visualisation distribution de différents paramètres	10
Suppression des valeurs manquantes de "family"	14
Suppression des faibles "confidence"	14
Choix du nombre de familles ("family")	14
Visualisation de quelques images	16
Segmentation	16
MODÉLISATION	19
Approche Machine Learning	19
Influence des paramètres de base	19
XGBOOST	20
Approche Deep Learning	21
Comparaison rapide des modèles de base	21
EFB1	23
Approche "unfreeze at start"	25
Approche 'Freeze at start + fine tuning'	27
Approche Hybride: EFB1 + SVC	30
Stacking	31
Autre bibliothèque utilisant la Bayesian optimization	32
Comparaison avec AutoML et précédents travaux	34
INTERPRÉTABILITÉ	36
GradCam	36
Shap	45
DIFFICULTÉS RENCONTRÉES LORS DU PROJET	45
POUR ALLER PLUS LOIN	46
BIBLIOGRAPHIE	47

CONTEXTE

La définition d'un objectif pour ce projet était en fait le premier point à définir pour ce projet. En effet, la reconnaissance d'un champignon à partir d'une image ou photo n'est pas un objectif assez précis. En se basant sur le site "<https://mushroomobserver.org>", on se rend compte qu'il n'existe pas une liste exhaustive de champignons. D'après le site, moins de 5% des espèces de champignons qui existent dans le monde seraient connues.

De plus, la définition même d'un champignon ne semble pas non plus être clairement définie: par exemple, les lichens, les rouilles ou les moisissures peuvent être considérés comme des champignons.

En se basant sur ce site, nous allons restreindre notre projet aux champignons charnus qui sont les champignons (comestibles ou non) qu'on peut trouver dans les forêts.

Néanmoins, l'objectif n'est toujours pas assez précis. Nous allons donc définir un objectif atteignable et assez précis en faisant une première analyse des données disponibles.

Ce projet a donc été traité plutôt d'un point de vue technique.

D'un point de vue économique, si le modèle était plus poussé (beaucoup plus de types de familles reconnues ou en ajoutant une cible comme par exemple si le champignon est comestible ou non) une application sur smartphone pourrait être créée qui reconnaîtrait le champignon qu'on prendrait en photo. Cependant, ce sont des compétences que nous ne possédons pas à ce jour. Par ailleurs, des applications existent déjà et sont disponibles sur l'App Store ou Google Play: "Champignouf", "Champignong pro", "IK-Champi", "Shroomify", etc. Il en existe une dizaine environ. Les plus performantes (comme "Shroomify") peuvent identifier jusqu'à environ 400 champignons communs.

Le modèle de reconnaissance de champignons pourrait aussi être utile d'un point de vue scientifique, tout simplement en indiquant aux chercheurs si le champignon qu'ils prennent en photo est déjà connu ou non. Mais, comme pour le point de vue économique, le modèle devrait pouvoir reconnaître beaucoup plus de caractéristiques.

OBJECTIFS

Ci-dessous les principales étapes du projet:

1. Recherche sur les données disponibles et de quelle façon les récupérer
2. Exploration et visualisation des données récupérées
3. Définition de l'objectif
4. Modélisations simples (1ère itération) avec du machine learning
5. Modélisations simples (2ème itération) avec du deep learning
6. Essais de différentes modélisations plus adaptées à la computer vision
7. Optimisations des modèles les plus performants
8. Interprétation des résultats avec différentes méthodes (*grad-cam*, *SHAP*) et étude de la segmentation (*clustering KMeans* et *MeanShift*)

Pour atteindre ces objectifs, nous nous sommes bien entendu basé sur les cours de DataScientest, dont particulièrement sur le deep learning (*tensorflow*, *keras*).

L'aide de notre tuteur nous a aussi bien aidé en nous permettant de résoudre certains problèmes d'exécution, ainsi qu'en suggérant certaines pistes intéressantes à étudier.

Enfin, l'aide de la communauté web a été indispensable (ex: *stackoverflow*, *kaggle*, *github*, etc.). Voir les liens dans la bibliographie, à la fin du rapport.

DATA

Plusieurs méthodes étaient possibles pour récupérer des données:

- *Web scraping* du site "<https://mushroomobserver.org>": cette méthode n'a pas été explorée principalement à cause du temps qu'il aurait fallu pour récupérer les données (images et informations). En effet, pour des raisons d'accessibilité au site, nous ne pouvons faire que 20 requêtes par minute (voir site "[lien](#)"). Si nous voulions récupérer par exemple 200,000 images, il aurait fallu au moins une semaine entière de temps d'exécution. En février 2022, le site compte plus de 530,000 images.
- Un fichier csv avec les liens de plus de 250,000 images était proposé par le site. Mais, nous aurions été de nouveau été contraint aux nombres de requêtes limités par le site
- À l'aide de notre tuteur, nous avons pu récupérer des images et leurs informations à partir d'un ancien projet *GitHub* ("[lien](#)"). Celui-ci amène au lien suivant qui permet de récupérer les données ("[lien](#)").

Ce jeu de données est constitué de 11 dossiers images (un dossier par année; de 2006 à 2016 inclus) et de 12 dossiers *json*. Les images représentent environ 5 Go et les *json* environ 440 Mo.

Construction de la base de données

La première étape a consisté à mettre toutes les données dans un même fichier csv:

- 6 paramètres pour les images:
 - *file_name*,
 - *file_path*,
 - *resolution*,
 - *file_year*,
 - *image_id*,
 - *format*

- 33 paramètres pour les *json*:

```
Entrée [9]: 1 json_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 650743 entries, 0 to 68567
Data columns (total 33 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   observation      650743 non-null   object  
 1   label             650743 non-null   object  
 2   image_id          650743 non-null   object  
 3   image_url         650743 non-null   object  
 4   user              650743 non-null   object  
 5   date              650743 non-null   object  
 6   thumbnail         650743 non-null   int64  
 7   location          650716 non-null   object  
 8   gbif_info.kingdom 616677 non-null   object  
 9   gbif_info.family   587873 non-null   object  
 10  gbif_info.speciesKey 413033 non-null   float64 
 11  gbif_info.rank    616721 non-null   object  
 12  gbif_info.phylum   612972 non-null   object  
 13  gbif_info.orderKey 608964 non-null   float64 
 14  gbif_info.species  413033 non-null   object  
 15  gbif_info.confidence 650600 non-null   float64 
 16  gbif_info.classKey 610661 non-null   float64 
 17  gbif_info.matchType 650600 non-null   object  
 18  gbif_info.familyKey 587873 non-null   float64 
 19  gbif_info.status    616721 non-null   object  
 20  gbif_info.usageKey  616721 non-null   float64 
 21  gbif_info.kingdomKey 616677 non-null   float64 
 22  gbif_info.genusKey  578871 non-null   float64 
 23  gbif_info.canonicalName 616721 non-null   object  
 24  gbif_info.phylumKey  612972 non-null   float64 
 25  gbif_info.class     610661 non-null   object  
 26  gbif_info.synonym   650600 non-null   object  
 27  gbif_info.scientificName 616721 non-null   object  
 28  gbif_info.genus     578871 non-null   object  
 29  gbif_info.order     608964 non-null   object  
 30  gbif_info.note      32828 non-null    object  
 31  gbif_info           0 non-null      float64 
 32  json_file          650743 non-null   object  
dtypes: float64(10), int64(1), object(22)
memory usage: 168.8+ MB
```

On s'aperçoit qu'il y a beaucoup plus de lignes pour les données *json*.

- En fusionnant les données dans un même fichier "*image_and_json_data.csv*", nous obtenons un *dataframe* de 215,410 lignes et 38 colonnes:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 215410 entries, 0 to 215409
Data columns (total 38 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   file_name        215410 non-null   object  
 1   file_path        215410 non-null   object  
 2   resolution       215410 non-null   object  
 3   file_year        215410 non-null   int64  
 4   image_id         215410 non-null   int64  
 5   format           215410 non-null   object  
 6   observation      215410 non-null   object  
 7   label            215410 non-null   object  
 8   image_url        215410 non-null   object  
 9   user             215410 non-null   object  
 10  date             215410 non-null   object  
 11  thumbnail        215410 non-null   int64  
 12  location          215408 non-null   object  
 13  gbif_info.kingdom 204271 non-null   object  
 14  gbif_info.family 193984 non-null   object  
 15  gbif_info.speciesKey 137145 non-null   float64 
 16  gbif_info.rank    204280 non-null   object  
 17  gbif_info.phylum  203054 non-null   object  
 18  gbif_info.orderKey 201455 non-null   float64 
 19  gbif_info.species 137145 non-null   object  
 20  gbif_info.confidence 215364 non-null   float64 
 21  gbif_info.classKey 202194 non-null   float64 
 22  gbif_info.matchType 215364 non-null   object  
 23  gbif_info.familyKey 193984 non-null   float64 
 24  gbif_info.status   204280 non-null   object  
 25  gbif_info.usageKey 204280 non-null   float64 
 26  gbif_info.kingdomKey 204271 non-null   float64 
 27  gbif_info.genusKey 191145 non-null   float64 
 28  gbif_info.canonicalName 204280 non-null   object  
 29  gbif_info.phylumKey 203054 non-null   float64 
 30  gbif_info.class    202194 non-null   object  
 31  gbif_info.synonym  215364 non-null   object  
 32  gbif_info.scientificName 204280 non-null   object  
 33  gbif_info.genus    191145 non-null   object  
 34  gbif_info.order    201455 non-null   object  
 35  gbif_info.note     10838 non-null   object  
 36  gbif_info          0 non-null     float64 
 37  json_file         215410 non-null   object  
dtypes: float64(10), int64(3), object(25)
memory usage: 62.5+ MB
```

A ce stade, nous pouvons voir qu'il y a beaucoup de valeurs manquantes pour certains paramètres. Mais, avant de les supprimer nous allons explorer et visualiser les données.

Voir *notebook* ([lien](#)) pour la construction des données.

EXPLORATION DES DONNÉES ET VISUALISATION

Taxonomie de la base de donnée

Ci-dessous un exemple de la taxonomie venant des données *json*:

```
{'date': '2006-05-21 07:17:22',
 'gbif_info': {'canonicalName': 'Xerocomells dryophils',
   'class': 'Agaricomycetes',
   'classKey': 186,
   'confidence': 98,
   'family': 'Boletaceae',
   'familyKey': 8789,
   'gens': 'Xerocomells',
   'gensKey': 8184844,
   'kingdom': 'Fungi',
   'kingdomKey': 5,
   'matchType': 'EXACT',
   'order': 'Boletales',
   'orderKey': 1063,
   'phylm': 'Basidiomycota',
   'phylmKey': 34,
   'rank': 'SPECIES',
   'scientificName': 'Xerocomells dryophils (Thiers) N. Siegel, C.F. Schwarz & J.L. Frank, 2014',
   'species': 'Xerocomells dryophils',
   'speciesKey': 7574003,
   'stats': 'ACCEPTED',
   'synonym': False,
   'sageKey': 7574003},
 'image_id': 11,
 'image_r1': 'http://mshroomobserver.org/images/320/11',
 'label': 'Xerocomells dryophils',
 'location': 38,
 'observation': 10,
 'thumbnail': 1,
 'user': 1}
```

Plusieurs paramètres ne semblent pas forcément intéressants pour la reconnaissance visuelle d'un champignon. Par exemple, la personne ayant prise la photo ("user") ou la date de l'ajout n'aident pas à reconnaître le type de champignon.

Résolution des images

On constate qu'il y a énormément de résolutions différentes (734 au total).

Ci-dessous l'affichage des 5 résolutions les plus présentes:

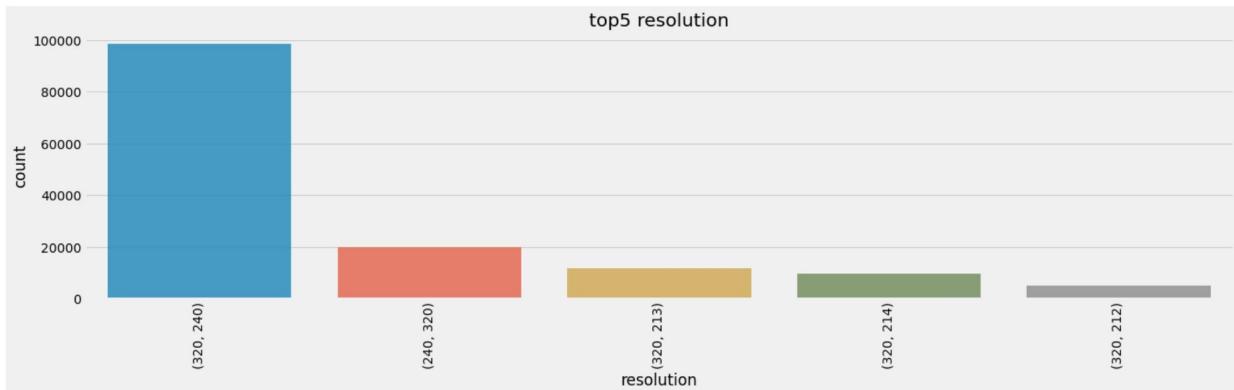


fig2. Top5 resolution

Par la suite, nous ne garderons que la principale résolution (320, 240) = (width, height).

Cela nous évite de changer la taille des images, évitant ainsi d'ajouter un biais en "déformant" les images. De plus, nous évitons la tâche fastidieuse de devoir sauvegarder les images transformées.

A ce stade nous obtenons 98,344 images.

Visualisation distribution de différents paramètres

La visualisation de rank permet de voir la hiérarchie de la dénomination:

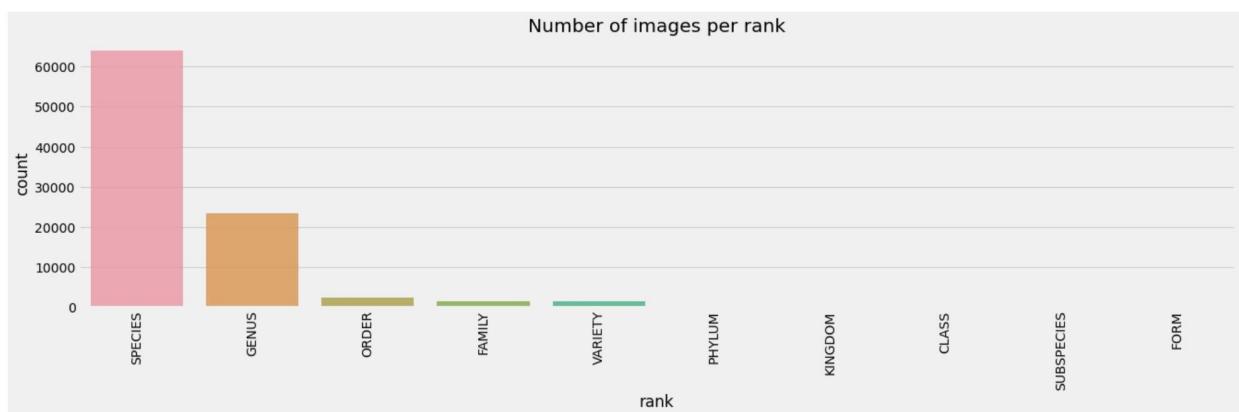


fig3. images per rank

SPECIES	64012
GENUS	23424
ORDER	2331
FAMILY	1560
VARIETY	1455
PHYLUM	229
KINGDOM	226
CLASS	182
SUBSPECIES	144
FORM	110

Mais, toutes les modalités de rank ne sont pas disponibles comme *FORM*, *SUBSPECIES* et *VARIETY*.

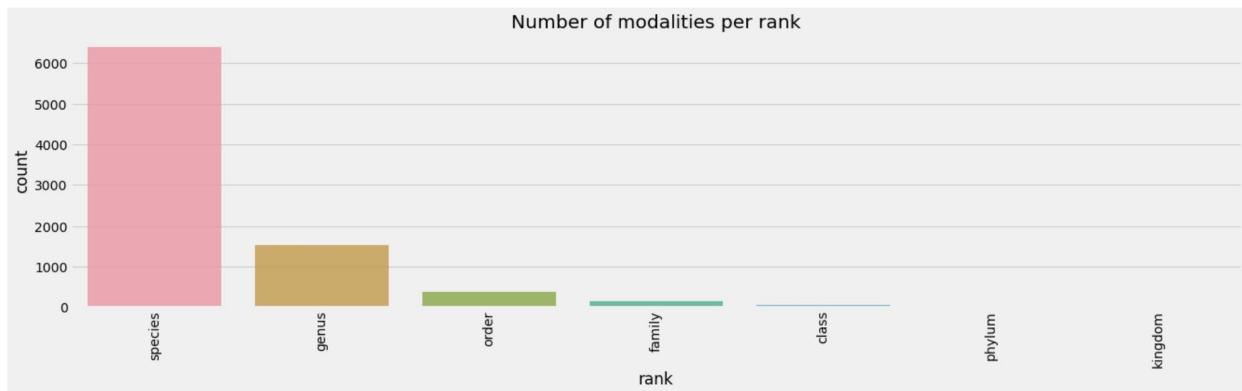


fig4. number of images per available rank

nombre de modalités pour species: 6405
 nombre de modalités pour genus: 1541
 nombre de modalités pour family: 390
 nombre de modalités pour order: 148
 nombre de modalités pour class: 48
 nombre de modalités pour phylum: 19
 nombre de modalités pour kingdom: 7

Les paramètres "*species*" et "*genus*" sont des branches trop détaillées ne permettant pas d'avoir suffisamment de modalités avec assez d'images par modalité.

- "*species*": moins de 600 images pour chaque modalité

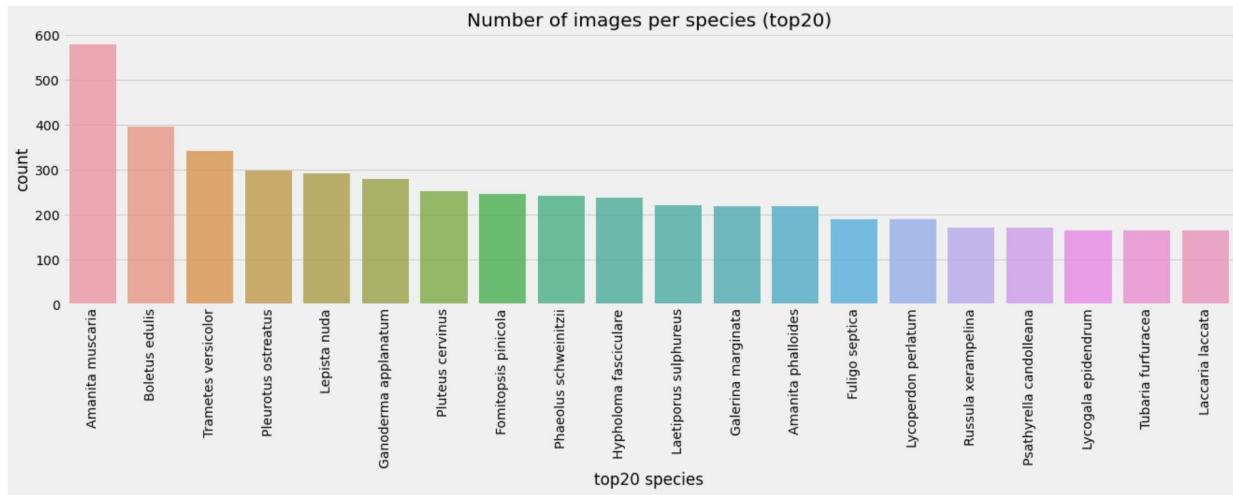


fig5. Top20 species

- “genus”: seulement 1 modalité avec plus 4000 images

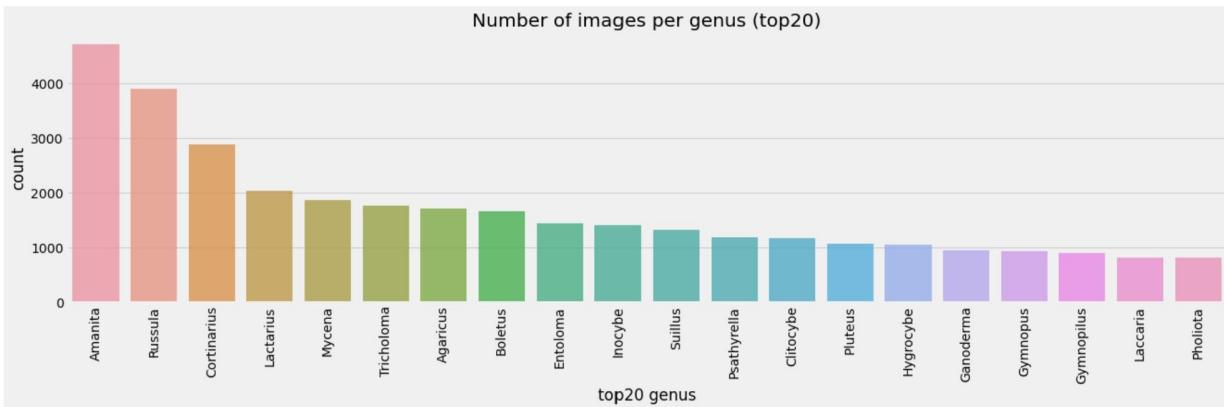


fig6. Top20 genus

- “family”: est le paramètre cible retenu avec suffisamment de modalités (7) ayant chacune plus de 4000 images.

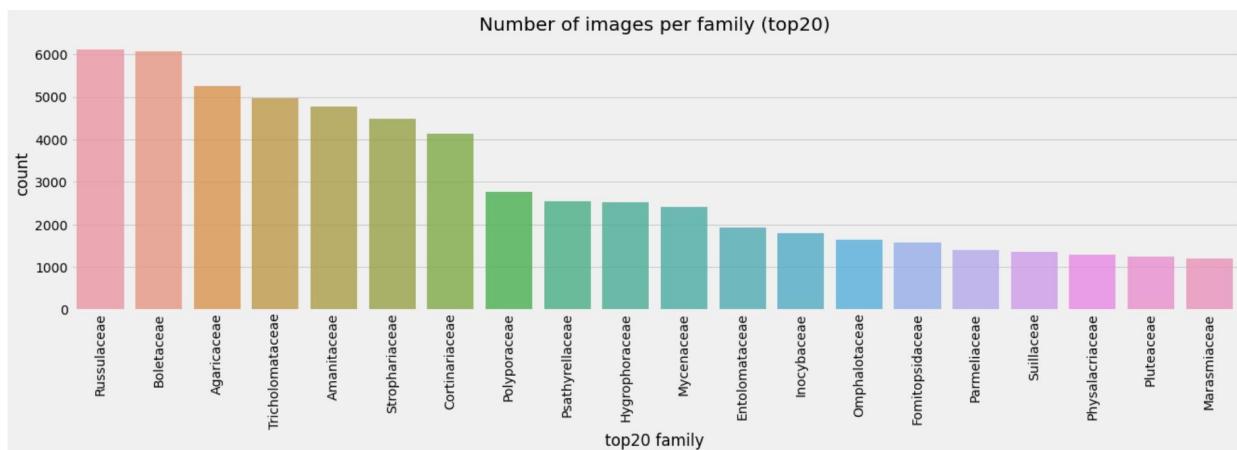


fig7. Top20 family

- “order”: jeu déséquilibré avec une modalité prépondérante. Il aurait pu servir de pré-filtre comme “class”.

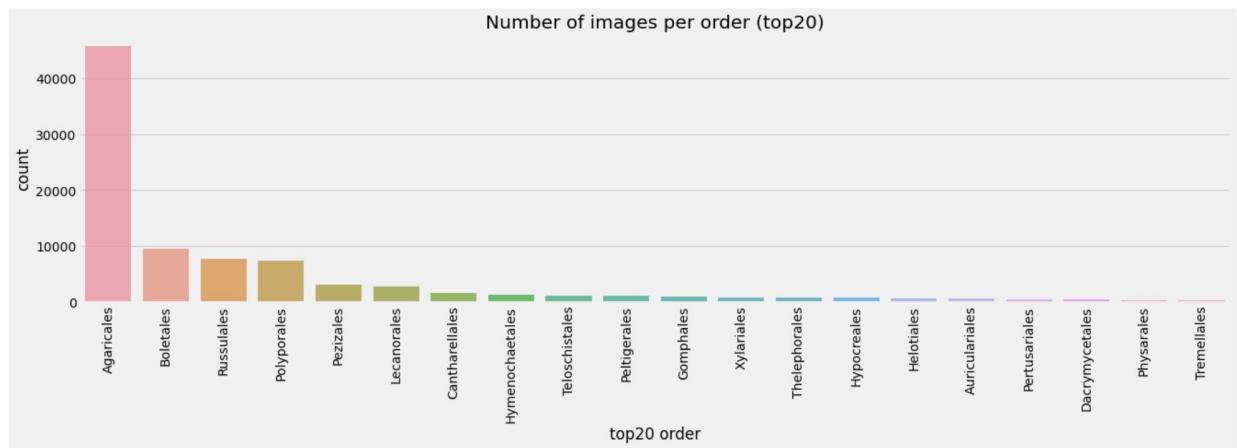


fig8. Top20 order

- “class”: Niveau haut. Mais, il sert de pré-filtre pour ne retenir que la modalité prépondérante qui est “Agaricomycetes”.



fig9. Top5 class

- “*phylum*”: idem “*class*”.
- “*kingdom*”: idem “*class*”.

Suppression des valeurs manquantes de “family”

On obtient 89,948 images.

Suppression des faibles “confidence”

Nous choisissons arbitrairement les images avec une identification claire (càd valeurs de “confidence” > 90%). Nous obtenons 89,751 images.

Choix du nombre de familles (“family”)

La distribution des familles (voir graphe ci-dessus) nous permet de voir combien de familles contiennent une quantité suffisante d’images pour pouvoir appliquer un deep learning satisfaisant.

On observe que les 7 familles les plus importantes se détachent des autres, avec plus de 4000 images par famille ce qui semble suffisant pour faire de la computer vision.

	label	gbif_info.family	size
0	0	Agaricaceae	5254
1	1	Strophariaceae	4494
2	2	Amanitaceae	4769
3	3	Boletaceae	6060
4	4	Tricholomataceae	4967
5	5	Cortinariaceae	4124
6	6	Russulaceae	6119

A ce stade, nous avons réduit notre jeu de données à 35,787 images.

Voir notebook ([lien](#)) pour l'exploration des données.

Finalement, nous avons retenu comme cible le paramètre "*family*" qui sera par la suite transformée en une autre variable numérique "*label*" pour le deep learning, et la variable "*file_path*" pour le chemin d'accès à chaque image.

Pour la suite du projet, le problème va donc consister à classifier les champignons dans un jeu de données plutôt équilibrées.

Visualisation de quelques images

Ci-dessous 5 images choisies au hasard pour chacune des 7 familles retenues:

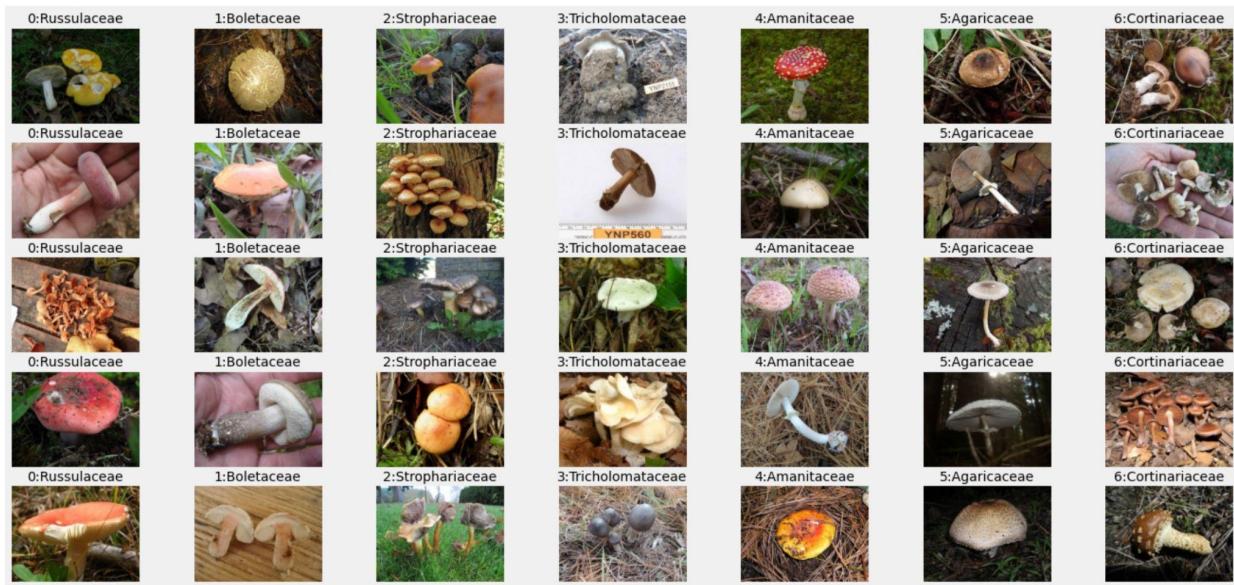


fig10. 7 familles de champignons

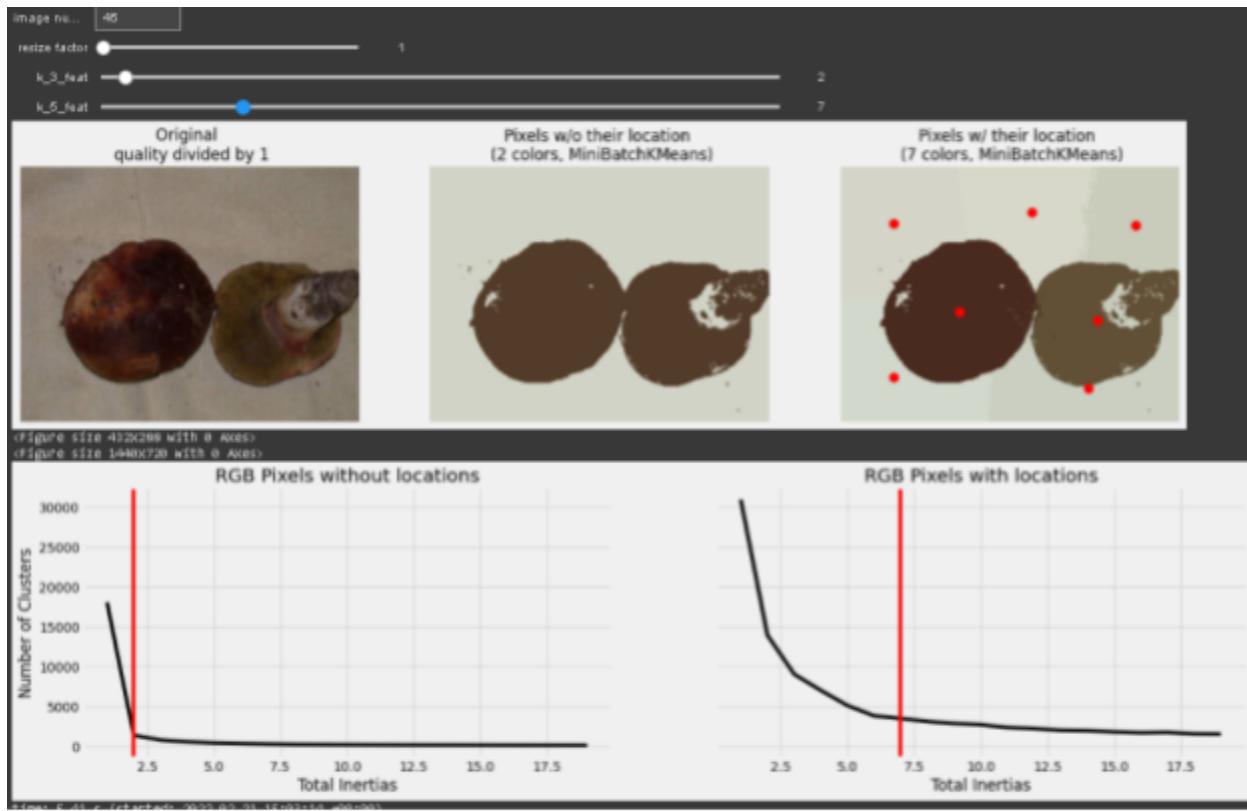
On constate qu'il est très difficile pour un néophyte de reconnaître des points communs évidents pour identifier une famille. Les couleurs et formes peuvent être différentes au sein de la même famille. Sans parler de la diversité des prises de vues des photos.

A première vue, prédire la famille d'un champignon à partir uniquement d'une image ne semble pas donc pas aisé.

Segmentation

Une partie de ce projet a concerné l'exploration de différentes solutions de segmentation d'image faisant intervenir des algorithmes de clustering (voir [lien](#) et [version colab avec widgets](#)).

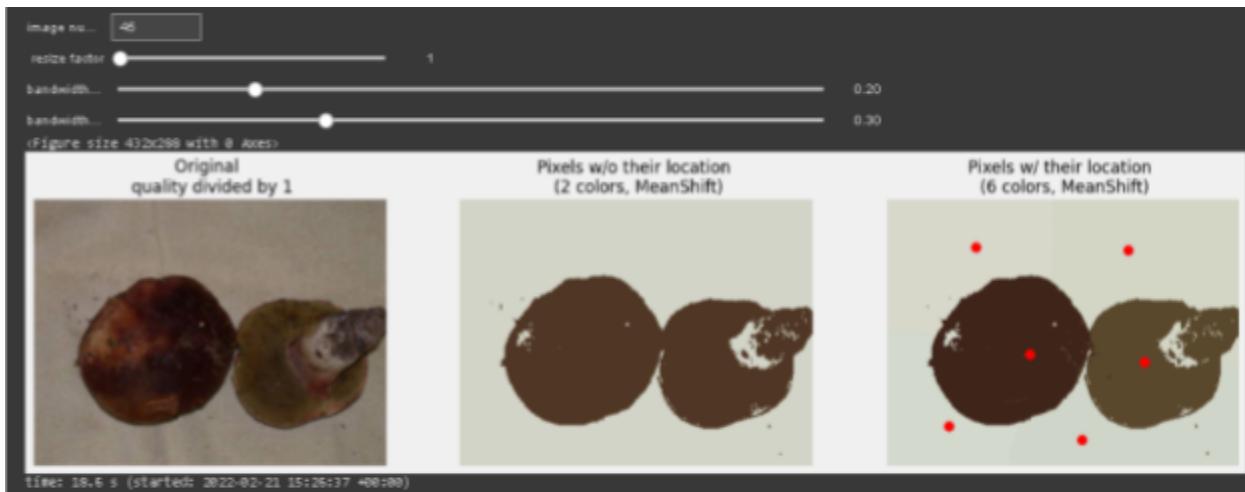
L'image ci-dessous montre qu'il est possible de segmenter l'image en différents clusters correspondant aux différents champignons et au *background* grâce à l'algorithme *Kmeans*. Par ailleurs, les propriétés de la segmentation dépendent du nombre de features attribuées à chaque pixel : valeurs des 3 canaux RGB ou valeurs des 3 canaux RGB + valeurs des coordonnées x/y.



Exemple de segmentation faisant intervenir l'algorithme Kmeans

fig11. Segmentation Kmeans

L'algorithme *MeanShift* permet également d'obtenir des résultats intéressants comme le montre la figure ci-dessous. Mais, il est pénalisé par son temps de calcul qui nécessite une compression des images, notamment pour estimer les hyper paramètres optimaux qui déterminent le nombre de clusters (*bandwidth*).



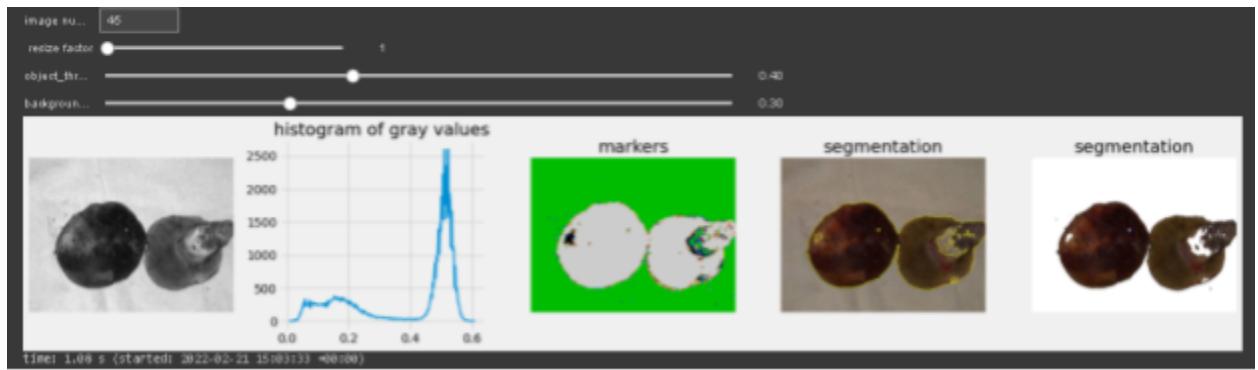
Exemple de segmentation faisant intervenir l'algorithme MeanShift

fig12. Segmentation MeanShift

Finalement, des méthodes plus élaborées basées sur la détection des bords et l'histogramme d'intensité des pixels en niveau de gris permettent aussi d'obtenir des champignons détournés comme le démontre l'exemple ci-dessous.

Toutefois, toutes les méthodes étudiées souffrent de leur temps de calcul et surtout du manque d'une méthode automatisée permettant de trouver les hyper paramètres optimaux pour chacune des images vu la grande diversité de ces dernières. Or, ce sont ces hyper paramètres qui déterminent le nombre de clusters et la qualité de la segmentation.

Des méthodes basées sur du deep learning avancé existent (cf [lien](#)). Mais ces dernières étaient hors de portée pour ce projet de 5 semaines.



Exemple de segmentation obtenue avec des méthodes plus élaborées

fig13. Autres exemples de segmentation

MODÉLISATION

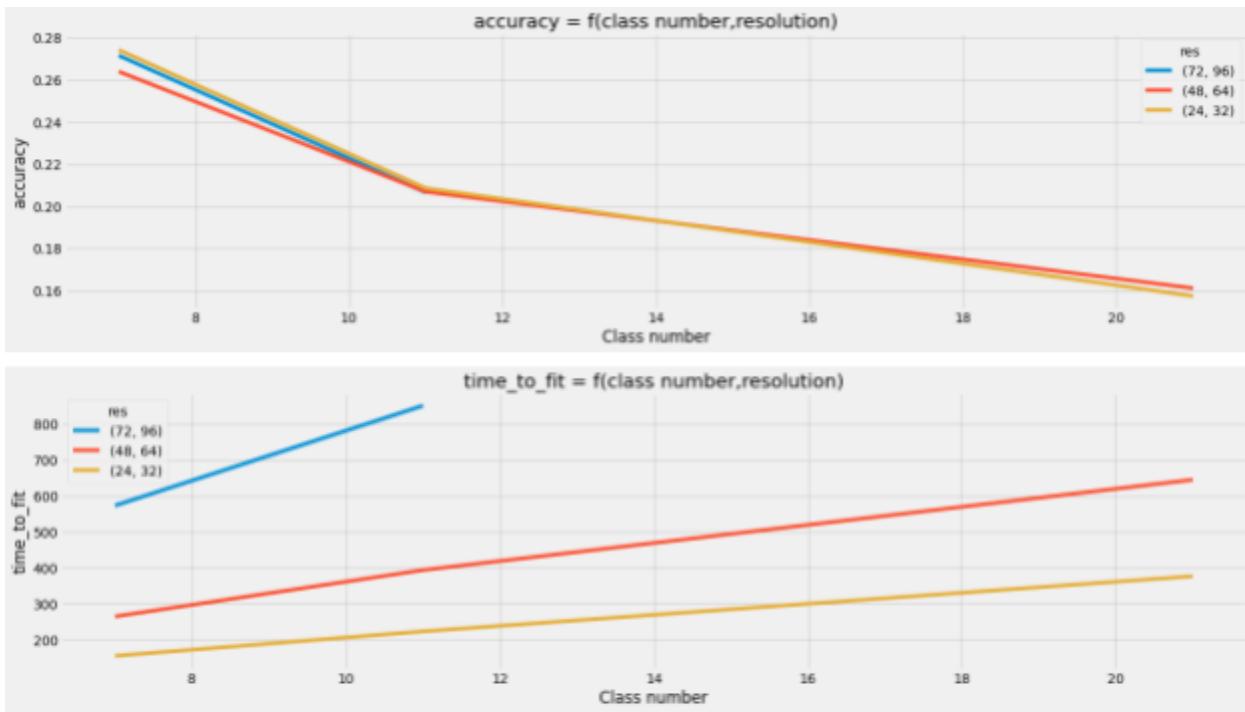
Approche *Machine Learning*

Influence des paramètres de base

Dans un premier temps, une approche utilisant uniquement des algorithmes de *machine learning* a été utilisée suivant cette méthode:

- chacune des images est compressée en réduisant sa résolution
- les images sont ensuite transformées en une matrice de dimensions $N = \text{colonnes de pixels} \times \text{lignes de pixels} \times 3 \text{ canaux RGB}$
- toutes les images sont concaténées dans une matrice de dimensions $N \times \text{nombre d'images}$ (un pixel \Leftrightarrow un feature , une ligne \Leftrightarrow une image)
- un split train/test de 80/20% est appliqué avec option stratify
- un algorithme de machine learning est entraîné sur le train test
- l'algorithme est finalement évalué sur le test set

La figure ci-dessous (cf [lien](#)) montre qu'avec un algorithme de type *random forest* (avec les hyper paramètres de base), une *accuracy* de l'ordre 0,27 est obtenue avec 5 familles contre 0,16 avec 21 familles. Par contre, la résolution n'a qu'un faible effet sur la précision du modèle alors qu'elle a un effet très important sur le temps d'entraînement et le besoin de RAM. C'est pour cette raison (crash du *kernel*) qu'il n'y a pas de point pour la résolution (72, 96) avec plus de 11 familles (courbes bleues). Des résultats similaires ont été obtenus avec un classifieur de type SVC.



Influence du nombre de classes et de la résolution sur les performances d'un modèle de type random forest

fig14. Performances RandomForest

XGBOOST & Optuna

Malgré les limitations évidentes d'une approche machine learning pour ce type de problème, des essais ont été menés avec le classifier *XGBOOST* et la librairie d'optimisation bayésienne des hyper paramètres *Optuna* (cf [lien](#)). Ce package est un logiciel d'optimisation automatique des hyperparamètres, particulièrement conçu pour l'apprentissage automatique. Il est basé sur l'optimisation bayésienne qui est une stratégie de conception séquentielle pour l'optimisation globale de fonctions boîte noire qui ne suppose aucune forme fonctionnelle. Elle est généralement employée pour optimiser des fonctions coûteuses à évaluer (cf [lien](#)). Le tableau ci-dessous montre que la combinaison de XGBOOST et de cette méthode a permis d'obtenir **une accuracy de 0,32 sur 7 familles**.

	precision	recall	f1-score	support
0	0.32	0.34	0.33	1051
1	0.35	0.27	0.31	954
2	0.31	0.52	0.39	1212
3	0.27	0.16	0.20	825
4	0.34	0.39	0.36	1224
5	0.32	0.24	0.28	899
6	0.33	0.21	0.26	993
accuracy			0.32	7158
macro avg	0.32	0.31	0.30	7158
weighted avg	0.32	0.32	0.31	7158

Report de classification obtenu avec un modèle XGBOOST
optimisé

Pour comprendre comment *XGBOOST* fait ses prédictions, l'importance de chacune des features (ici les pixels) a été extraite puis transformée en une image en niveaux de gris (couleur d'un pixel proportionnelle à la moyenne des 3 niveaux RGB). La figure ci-dessous montre ainsi que les pixels au centre sont les plus utilisés lors de la classification. Cela semble pertinent, car les photos sont en général réalisées avec le champignon en son centre.

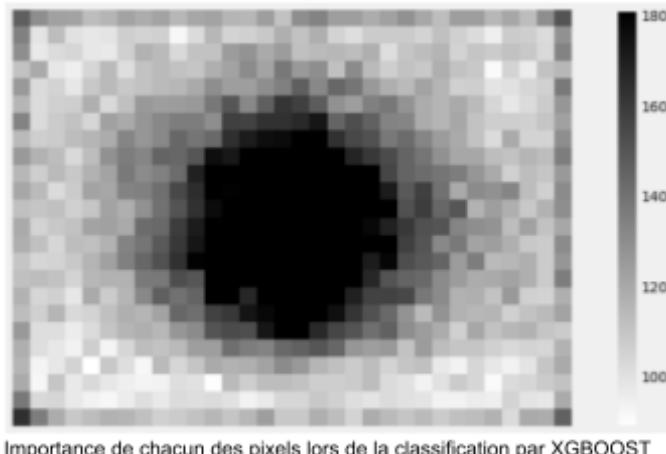


fig15. Importance features with XGBOOST

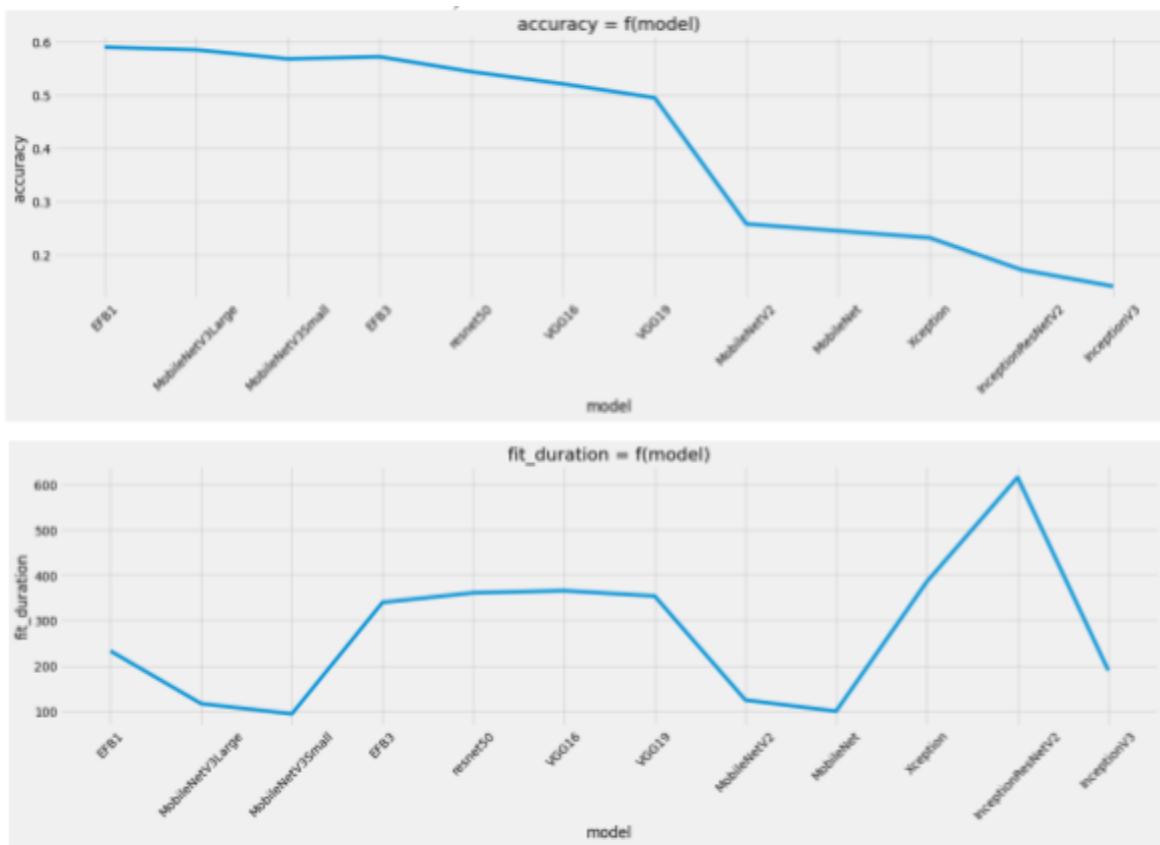
Approche Deep Learning

Comparaison rapide des modèles de base

Dans le but d'identifier rapidement les architectures des réseaux qui fonctionnent bien sur notre jeu de données, différents modèles *CNN* ont été testés sur une partie de notre jeu de

données limitée à 5000 images pour limiter le temps de calcul ([Lien](#)). Les modèles de base tels que *EfficientNetB1* ou *Inception* ont été chargés avec les poids 'imagenet' dans une architecture qui leur ajoute une couche *fully connected* (2 couples de couches *dense/dropout* + couche de prédiction). La figure suivante montre que le modèle *EfficientNetB1* (EFB1) permet d'obtenir les meilleures performances en un temps raisonnable.

On peut toutefois remarquer que le modèle *MobileNetV3Small* est le plus performant en temps de calcul avec une accuracy légèrement plus faible que celle d'*EfficientNetB1*.



Accuracy et durée d'entraînement (s) en fonction du modèle de base

fig16. Comparaison performances de plusieurs modèles

Par la suite, le jeu de données retenues (35787 images) a été divisé en trois jeux (train, validation et test; 64%, 16% et 20 du jeu).

Deux méthodes de générateur d'images ont été étudiées:

- *flow_from_dataframe*

-
- *flow_from_directory*

La première méthode est la moins contraignante puisqu'elle nous permet d'accéder aux images grâce à la colonne "*file_path*" d'un dataframe. C'est celle-ci qui a été privilégiée par la suite.

La deuxième méthode, nous oblige à enregistrer les images suivant la répartition des jeux train, validation et test. Cette méthode classe par défaut les données cibles dans l'ordre alphabétique (ce qui n'est pas le cas de la première méthode). Un exemple d'utilisation est disponible sous le lien suivant: [lien](#).

EFB1

Suite aux résultats précédents, le modèle EFB1 a été retenu pour tenter d'améliorer l'*accuracy*. La figure ci-dessous montre qu'en fonction du problème, différentes approches peuvent être retenues. Le jeu de données qui fait l'objet des prochaines expériences est constitué d'au moins 4000 images pour la classe minoritaire (ramené à 600 quand la taille du jeu de données est restreinte à 5000 images pour accélérer le temps de calcul) ce qui limite le choix à la partie supérieure.

Par ailleurs, des tests préliminaires ont montré qu'il était bien plus efficace de partir d'un réseau pré-entraîné (*transfer learning*) sur imangenet qu'à partir d'un réseau initialisé de façon aléatoire. C'est cette approche qui a été retenue. Cependant, comme il est difficile d'évaluer la ressemblance entre le jeu de données et la base de données imangenet, les deux méthodes (quadrants 1 & 2) de la figure suivante ont été comparées.

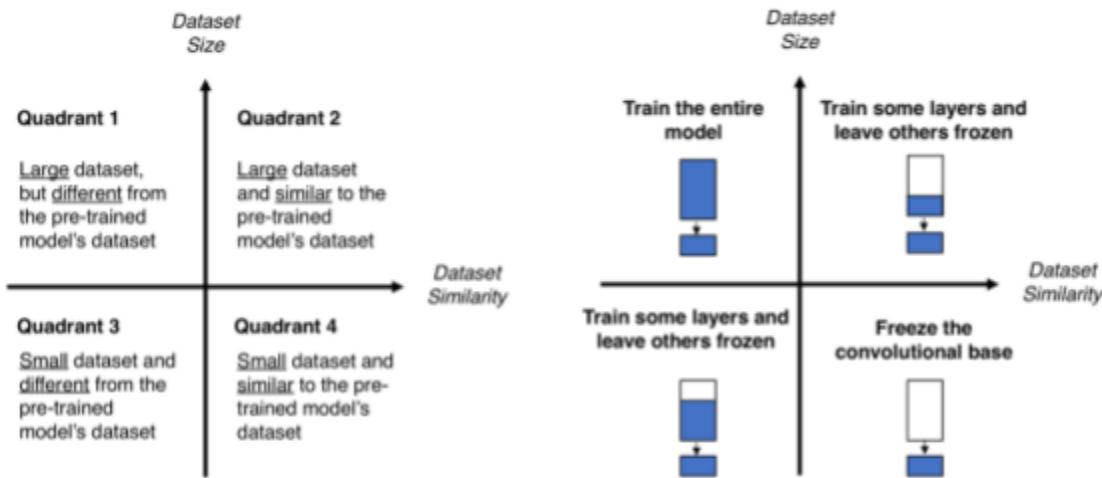
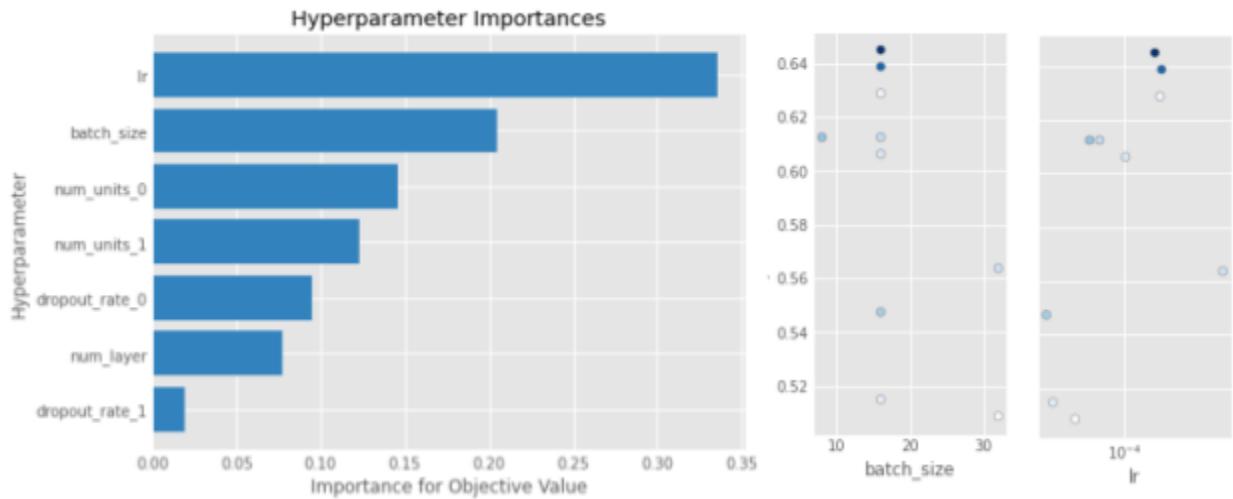


Diagramme d'aide au choix de l'approche à utiliser pour entraîner un CNN

fig17. Diagramme choix CNN

Tout au long des expériences suivantes, le package *Optuna* a été extensivement utilisé sur le jeu de données restreint à 5000 images pour optimiser les hyper paramètres (*learning rate*, *batch size*, architecture de la couche *fully connected*, utilisation de la *data augmentation*, ...). Par exemple, sur la figure suivante extraite du notebook [DL Model optimization with optuna + unfreeze base model](#), le *learning rate* a plus d'importance, que le *batch_size* et ainsi de suite. Il est ensuite possible de choisir l'essai ayant donné les meilleurs résultats ou de relancer une recherche dans un espace restreint (nombre d'hyper paramètres ou intervalle de recherche).



Exemple de l'importance des hyperparamètres et évolution de l'accuracy en fonction des deux principaux.

fig18. Importance hyper paramètres

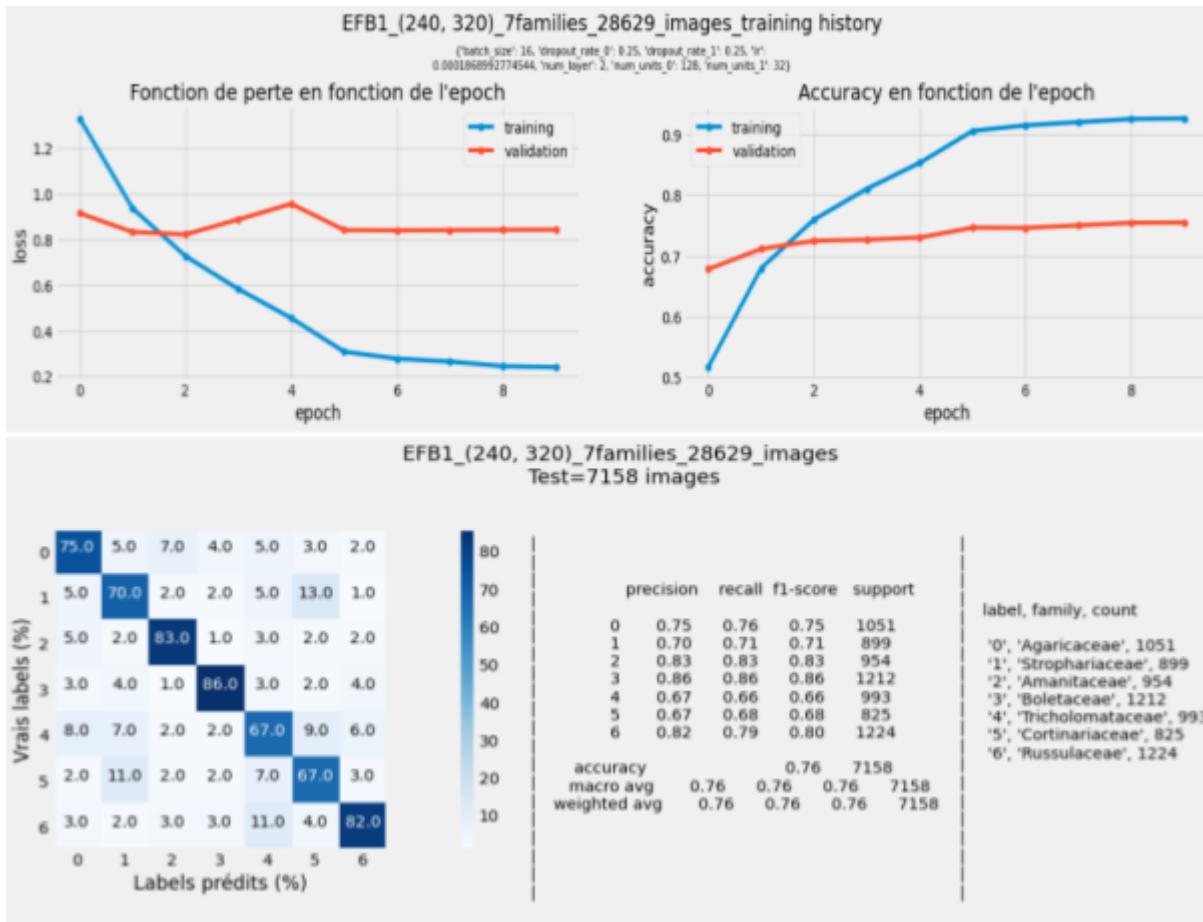
Les résultats présentés par la suite se concentrent donc sur les meilleurs modèles issus des expériences menées avec *Optuna* sur 5000 images, puis appliqués au jeu de données complet (un essai a montré qu'utiliser *Optuna* sur le jeu complet n'amène pas d'amélioration). Il faut également noter que tous ces tests ont été menés en synergie avec l'utilisation de deux *callbacks*:

- *ReduceLROnPlateau*: pour réduire le learning rate lorsque l'accuracy du jeu de données de validation atteint un plateau ou augmente durant N epochs (en général d'un facteur 10 au bout de 2 epochs dans ces travaux).
- *EarlyStopping*: pour stopper l'entraînement et limiter le surapprentissage lorsque l'accuracy du jeu de données de validation atteint un plateau ou augmente durant N epochs (en général au bout de 3 epochs dans ces travaux).

Approche “unfreeze at start”

Pour réaliser cette méthode, après une initialisation avec les poids imangenet, toutes les couches du réseau de neurones sont dégelées dès le début de l'entraînement. La figure

ci-dessous, issue du *notebook* [Best EFB1 model on all images unfreeze at start](#), montre qu'**une accuracy de 0,76 a pu être mesurée sur le test set**. Elle a été obtenue en adjoignant une couche *fully connected* composée de deux couples de couches *dense/dropout* (128 neurones/25% - 32 neurones/25%) et d'une couche de prédiction à la partie convulsive d'EFB1. Malgré l'utilisation des *callbacks*, les courbes d'entraînement montrent un certain *overfit* (gap entre les courbes obtenues sur les deux jeux de données).



Synthèse des résultats obtenus avec la méthode *unfreeze at start*

[fig19. Resultats EFB1 - unfreeze at start](#)

Certains des essais menés avec *Optuna* ont montré que l'*overfit* pouvait être réduit, mais c'est au détriment des performances sur le test set. Des essais ont également été menés en utilisant de la *data augmentation*. Mis à part une augmentation considérable du temps d'entraînement, ces derniers n'ont ni montré une amélioration sur le *test set*, ni une

réduction de l'*overfit*. Cette observation montre ainsi qu'avec plus de 4000 images par classe, le jeu de données est assez diversifié et ne bénéficie pas d'une *data augmentation* supplémentaire. Celà est confirmé en visualisant les images sur la figure ci-dessous qui montre la grande diversité de celles-ci :

- vue du dessus
- vue du dessous
- vue sur le côté
- champignon coupé
- champignon tout seul sur une table
- champignon tenu par une main
- plusieurs champignons sur la même image
- etc.



fig20. diversité images

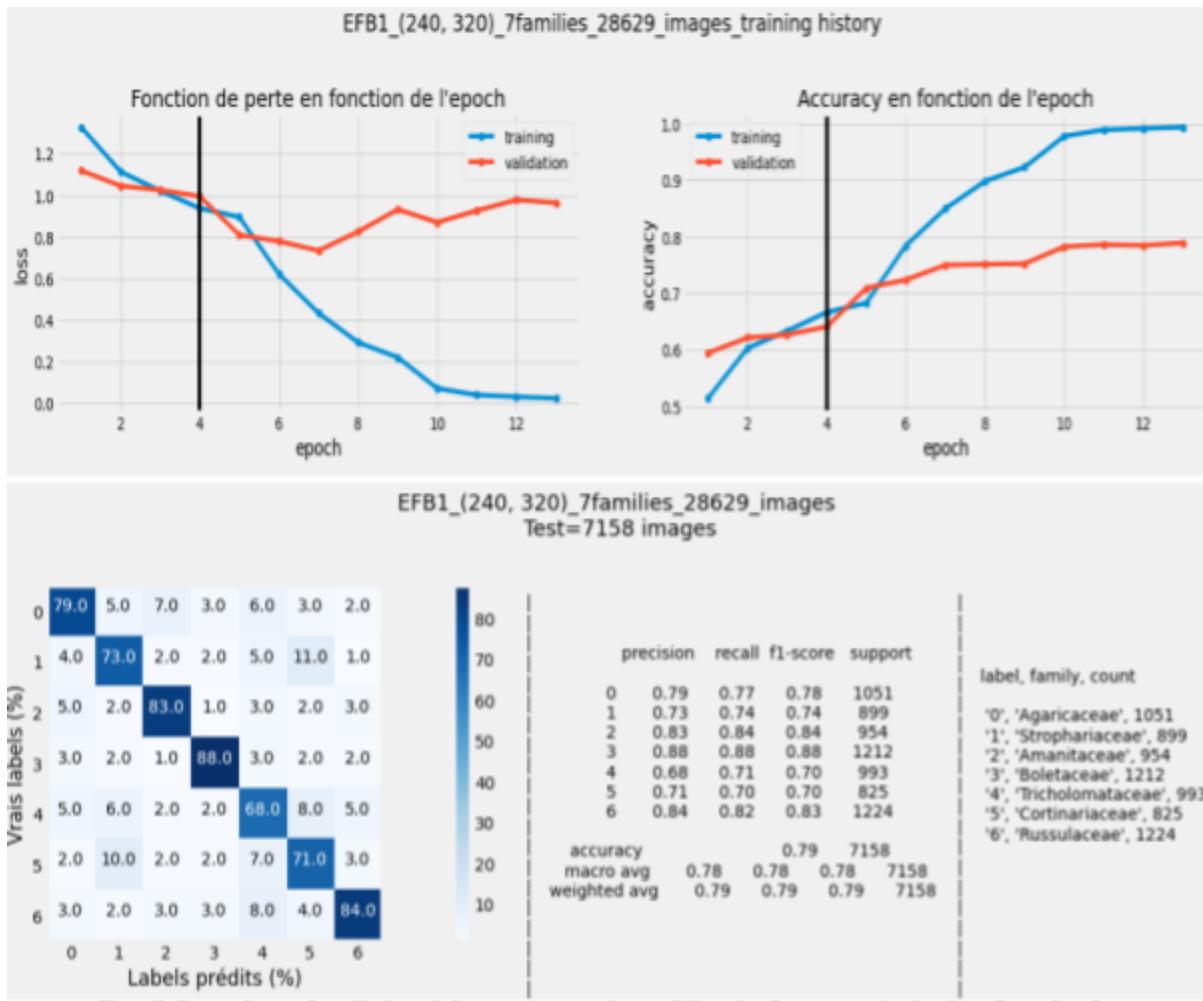
Finalement, des expériences ont été conduites en essayant d'appliquer soit une couche *GlobalAverage* à la place de la couche *fully connected*, soit un profil de *learning rate* (inspiré de ce *notebook* [lien](#) et optimisé via *Optuna* [lien](#)). Bien qu'elles aient permis de réduire considérablement l'*overfit*, les performances finales sur le jeu de test étaient moins bonnes (0,57 vs 0,65 sur le jeu de données réduit).

Approche ‘Freeze at start + fine tuning’

Durant cette approche, les essais ont été réalisés en deux temps:

- Dans un premier temps, la couche *fully connected* a été optimisée en fixant son architecture à deux couches *dense/dropout* et en laissant libre leur nombre de neurones et leur *dropout rate*, mais aussi le *learning rate* et le *batch size*. Durant cette étape, le réseau EFB1 a été initialisé avec les poids imagenet et ses couches ont été gelées. Le modèle a pu s'entraîner jusqu'à la détection de la convergence via un *callback early stopping*.
- Dans un second temps, toutes les couches du modèle sont dégelées avant de relancer un entraînement en commençant avec le *learning rate* de l'étape précédente et en réduisant grâce au *callback ReduceLROnPlateau*.

La figure ci-dessous montre qu'avec cette méthode, une *accuracy* de 0,65 est obtenue à la fin de la première phase, puis celle-ci augmente au-dessus de 0,70 dès que les couches du réseau convolutif sont dégelées (ligne verticale noire). Malgré l'*overfit* qui est supérieur à celui obtenu avec la méthode précédente, **un score de 0,79 a été mesuré sur le test set** (cf [\[Colab\] FINAL EFB1 model on all images freeze at start + fine tuning](#)).



Synthèse des résultats obtenus avec la méthode freeze at start + fine tuning

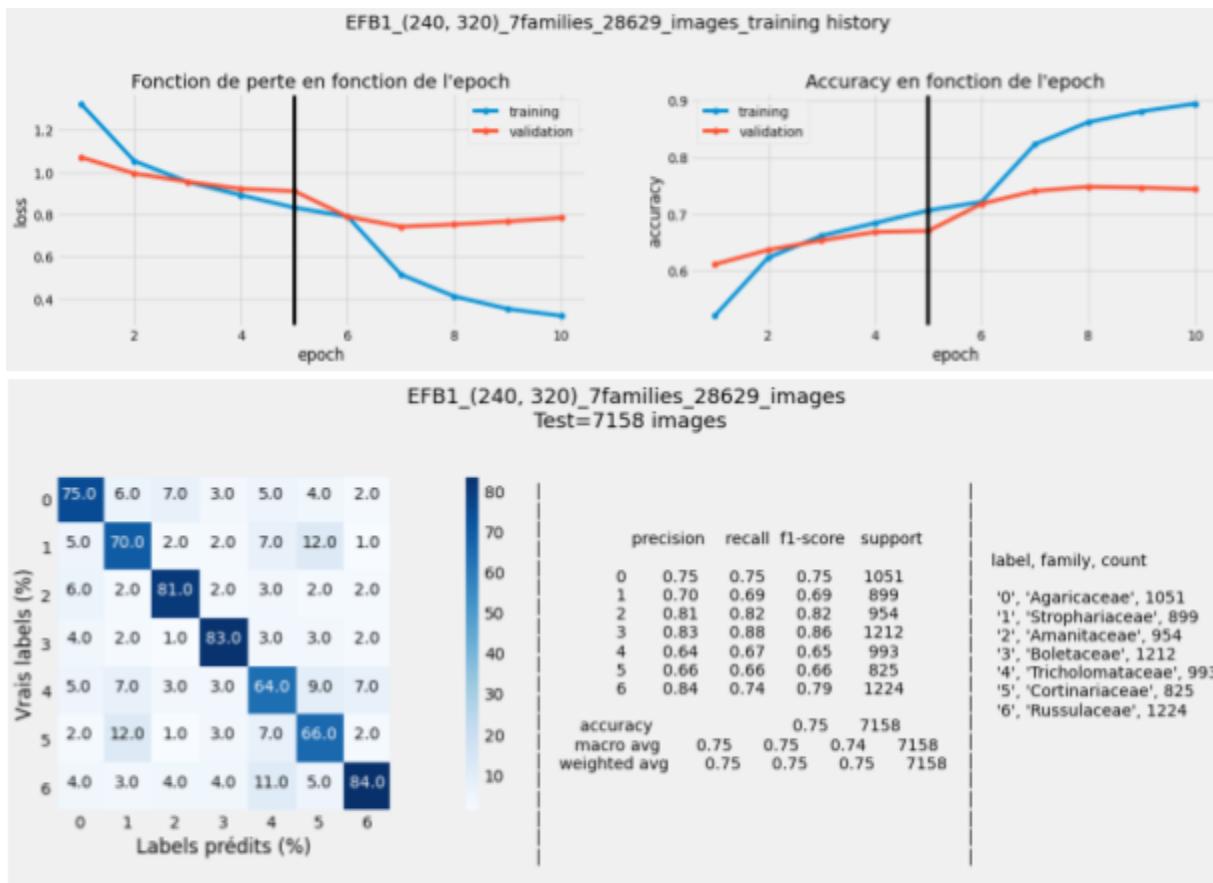
fig21. Résultats EFB1 - freeze at start + fine tuning

Pour tenter de réduire l'*overfit*, différents essais ont été menés toujours avec l'aide d'*Optuna*:

- Ajout de la *data augmentation* via un *data generator*
- Optimisation de l'étape de *fine tuning* en laissant *Optuna* choisir les meilleurs paramètres parmi ceux d'un *callback LearningRateScheduler* à décroissance exponentielle et le nombre de couches à dégeler (cf [lien](#))

Aucune de ces approches n'a permis d'égalier les performances présentées dans le paragraphe précédent. Par exemple, la figure ci-dessous (cf [lien](#)) montre qu'en dégelant

uniquement 250 sur les 339 couches et en appliquant une décroissance exponentielle du *learning rate* après décongélation des couches, l'*overfit* est légèrement réduit (*gap* entre les courbes de *training* et de *validation*), mais l'*accuracy* sur le *test set* subit une perte de 4% par rapport aux conditions précédentes.



Exemple de résultats obtenus avec un overfit limité.

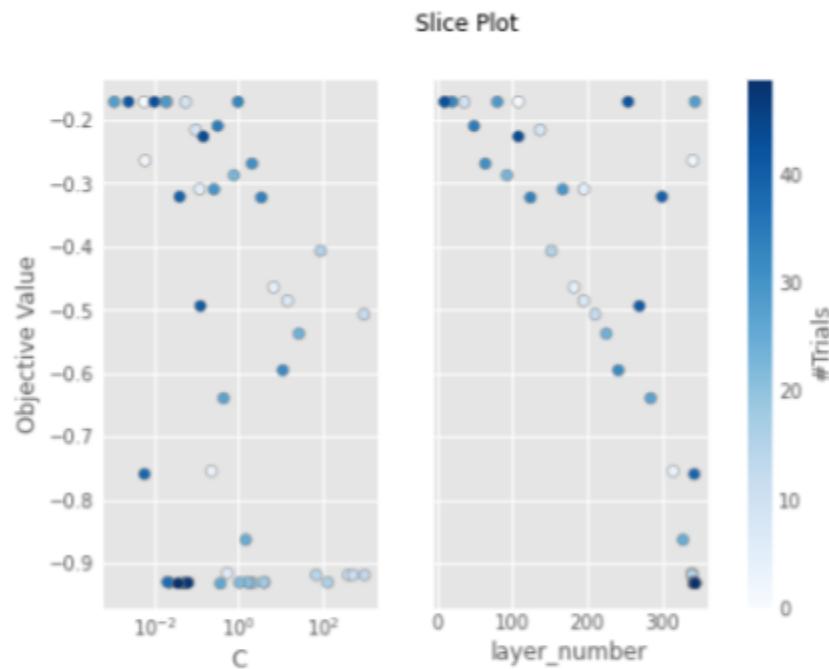
fig22. Résultats EFB1 - overfit limité

Approche Hybride: EFB1 + SVC

La méthode consistant à utiliser un classifieur de type *machine learning* après extraction des *features* par le réseau de neurone a également été étudiée ([lien](#)). Le modèle de la section précédente a d'abord été entraîné, puis un SVC a été utilisé pour classer les *features* extraits à différents endroits du réseau. Sur la figure ci-dessous, il est intéressant de noter que l'*accuracy* augmente avec la position de la couche utilisée pour extraire les *features*.

Cela démontre que plus les couches sont loin dans le réseau, plus elles sont spécialisées au problème étudié (N.B. l'accuracy est ici surévalué car le SVC a été appliqué sur une *cross-validation* de la combinaison du *train set* et du *set d'évaluation* utilisés pour entraîner le réseau de neurones).

En appliquant le meilleur *set* d'hyper paramètres (couche *global average* en sortie de la partie convulsive et $C = 1$) sur tout le jeu de données, cette méthode a donné des résultats comparables à celle de la section précédente avec **une accuracy de 0,79 sur le test set (lien)**.



Essais montrant l'évolution de -accuracy en fonction de l'hyperparamètre C et de la couche extraite

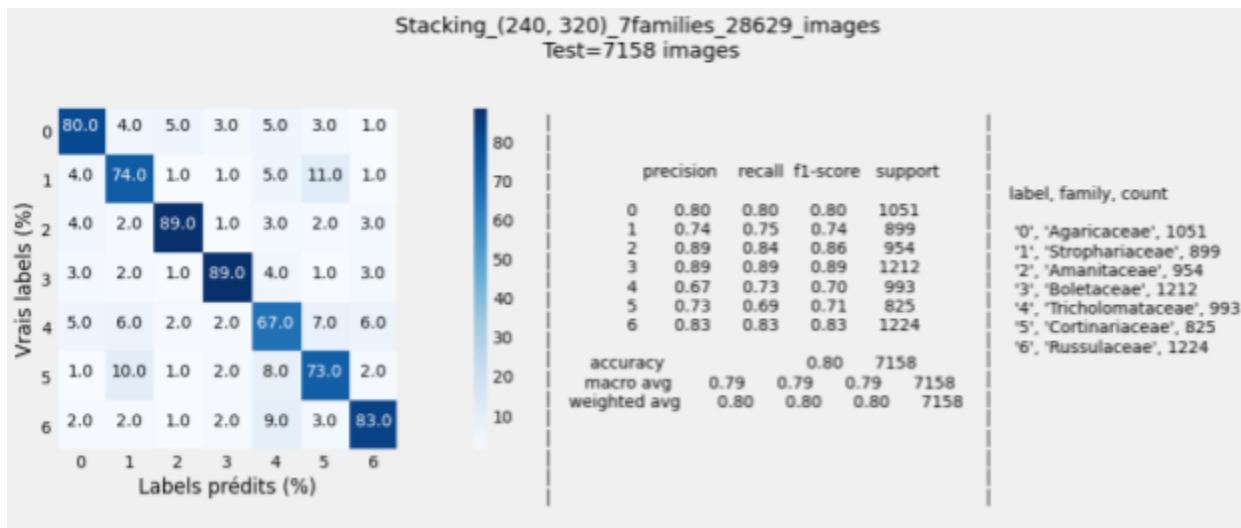
fig23. évolution accuracy versus C et couche

Stacking

Pour tenter d'améliorer encore l'accuracy, la méthode précédente (optimisation de l'approche 'freeze at start + fine tuning' sur 5000 images avec Optuna) a été reproduite avec les différents modèles ayant montré des bonnes performances dans la section

Comparaison rapide des modèles de base. Par la suite, chacun des modèles optimisés (*VGG19*, *EFB1*, *MobileNetV3Large*, *resnet50*, *VGG16*) a été entraîné sur le jeu d'entraînement, puis ils ont été utilisés pour générer des prédictions (N images x 7 probabilités d'appartenance aux différentes familles). Individuellement, ils montrent une *accuracy* sur le *test set* s'étalant de 0,68 pour *VGG19* à 0,79 pour *EFB1*.

Finalement, un modèle *XGBOOST* a permis de tirer parti de ces prédictions pour réaliser un *stacking* de ces modèles. La figure ci-dessous (cf [lien](#)) montre qu'avec cette approche un gain substantiel de 1% d'*accuracy* par rapport au meilleur modèle peut être obtenu. Peut-être que l'utilisation de modèles plus diversifiés (e.g. meilleurs pour prédire les classes 1, 4 et 5) ou une pondération de l'importance des prédictions en fonction des réseaux pourrait améliorer encore l'*accuracy*. Cependant, dans les quelques *notebooks kaggle* étudiés, le gain entre le meilleur modèle et le *stacking* de plusieurs modèles *CNN* restent souvent limités à 1 ou 2%.



Performances obtenues avec le stacking des 5 meilleurs modèles

fig24. Résultats Stacking - 5 meilleurs modèles

Autre bibliothèque utilisant la *Bayesian optimization*

Une autre bibliothèque plus simple qu'*Optuna* est la bibliothèque *bayesian-optimization*.

La description est disponible sur GitHub: <https://github.com/fmfn/BayesianOptimization>.

Cette piste a été étudiée en parallèle sur un modèle *MobileNetV3Small*. Nous avons choisi ce modèle pour le faible temps d'exécution. L'objectif ici n'était pas d'atteindre la meilleure optimisation possible (qui semblait être atteinte avec l'*EfficientNetB1* + optimisation). Mais, seulement d'explorer cette autre méthode.

Le *notebook* est disponible sous GitHub ([lien](#)).

Son application est relativement aisée.

Mais, il faut faire attention à notamment deux points:

- Par défaut, cette méthode va chercher la valeur maximale de performance. Il faut donc choisir la métrique "*accuracy*".
- Par défaut, à chaque itération les valeurs des hyper paramètres sont choisies dans l'étendue définie. Mais elles ne prennent pas de valeurs discrètes. Si on veut ajouter des paramètres discrets (comme *batch_size* ou *epochs*), il faut passer par la définition d'une fonction spécifique (voir: [lien](#); cf. paragraphe 2).

Comparaison avec AutoML et précédents travaux

Pour pouvoir comparer les performances des modèles développés lors de cette étude, la librairie d'AutoML *Autogluon* a été utilisée dans les mêmes conditions que les expériences précédentes (cf [lien](#)). La figure ci-dessous montre qu'**une *accuracy* de 0,70, a été mesurée, soit 10% en dessous des meilleurs résultats** présentés dans ce rapport.



Synthèse des résultats obtenus avec le package AutoGluon

fig25. Résultats AutoGluon

Par ailleurs, un groupe d'anciens élèves avait développé un modèle permettant d'obtenir un score de 0,77 sur 5 familles (cf [lien](#)) avec le modèle EFB1. En limitant à 5 familles, les modèles basés sur EFB1 et développés dans les sections précédentes permettent d'obtenir une *accuracy* supérieure à 0,83 (cf [lien](#)), le gain de 6% étant probablement dû à l'utilisation d'*Optuna* pour optimiser les hyper paramètres des modèles utilisés.

INTERPRÉTABILITÉ

GradCam

L'implémentation de cette méthode n'a pas été évidente. L'exemple donné par Keras (["https://keras.io/examples/vision/grad_cam/"](https://keras.io/examples/vision/grad_cam/)) semble facilement implémentable.

Or l'exemple se base sur un modèle trop simple. Dans notre cas, nous avons appliqué un transfer learning à partir du modèle "*EfficientNetB1*" disponible sur *Keras*. Puis, nous avons fait deux optimisations qui ont modifié les poids de plusieurs couches.

Pour pouvoir atteindre la dernière couche de convolution, il faut utiliser l'API fonctionnelle et par conséquent recréer un nouveau modèle pour identifier clairement cette dernière couche de convolution. Mais, il faut aussi s'assurer d'appliquer les poids appris durant l'optimisation du modèle.

Le notebook est disponible sous le lien GitHub suivant:

https://github.com/DataScientest-Studio/Pyck_a_Champy/tree/main/GradCam_final_model

Ci-dessous une description des différentes étapes:

1. importation des modules
2. création du *base model* = *EfficientNetB1* puis application des poids appris lors de notre optimisation

Il faut au préalable avoir enregistré les poids du modèle final optimisé. Puis, appliquer ses poids au *base_model* (attention à bien identifier les différentes couches).

3. identification de la dernière couche de convolution

"*top_conv*" est le nom de la dernière couche de convolution.

4. création d'un nouveau modèle avec l'API fonctionnelle

En plus d'avoir identifier clairement la dernière couche de convolution, nous avons aussi appliqué aux couches hautes "*Dense*" les poids du modèle optimisé.

Pour s'assurer d'avoir correctement appliqué les poids, nous pouvons faire un test d'équivalence.

Et, il faut aussi s'assurer que les performances du *new_model* soient les mêmes que celles du modèle final (enregistré ici sous la variable *efficientnet_final*)

Un énième vérification a aussi été faite sur les probabilités de chaque classe sur différentes images avec *new_model* (*API fonctionnelle*) et *efficientnet_final* (*Sequential*).

C'est une étape indispensable avant de commencer l'étude d'un *grad-cam*.

5. définition de l

Visualisation d'une image choisie au hasard parmi le jeu de test:



fig26. exemple champignon - class3

6. définition de l

L'*heatmap* est une représentation visuelle de l'activation des pixels de la dernière couche de convolution. Plus la valeur du pixel de l'*heatmap* est élevée, plus la probabilité d'activer ce pixel est élevée. Le calcul de l'*heatmap* se fait par une descente de gradient.

Ci-dessous une visualisation de l'*heatmap* suivant deux *cmap* différents. Le calcul de l'*heatmap* est fait à partir de la même image de départ.

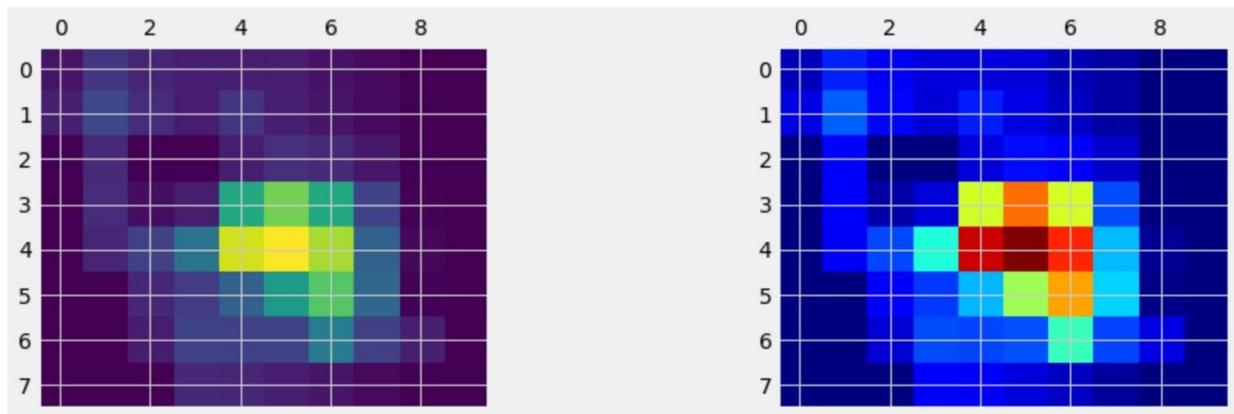


fig27. exemple heatmap

A ce stade la résolution de l'*heatmap* correspond à celle de la dernière couche de convolution, càd (8,10).

Pour l'*heatmap* à gauche, les pixels activés sont ceux proches du jaune ou du vert.

Pour l'*heatmap* à droite, les pixels activés sont ceux proches du marron ou rouge. Pour les prochaines étapes c'est cette *cmap* = "jet" qui sera utilisée.

7. redimensionnement de l'*heatmap* aux dimensions de l'image d'origine et superposition de l'*heatmap* avec l'image



fig28. exemple grad-cam

Dans cet exemple, tous les pixels représentant le champignon ne sont pas activés. Mais, cela reste malgré tout un résultat satisfaisant.

8. applications sur différentes images

Rappel résultats de la matrice de confusion du modèle optimisé:

	precision	recall	f1-score	support
0.0	0.79	0.76	0.77	1051
1.0	0.70	0.75	0.72	899
2.0	0.85	0.82	0.84	954
3.0	0.86	0.89	0.87	1212
4.0	0.70	0.66	0.68	993
5.0	0.66	0.72	0.69	825
6.0	0.83	0.82	0.82	1224
accuracy			0.78	7158
macro avg	0.77	0.77	0.77	7158
weighted avg	0.78	0.78	0.78	7158

- 6 images au hasard parmi la *class3* (ayant le meilleur *f1-score*)

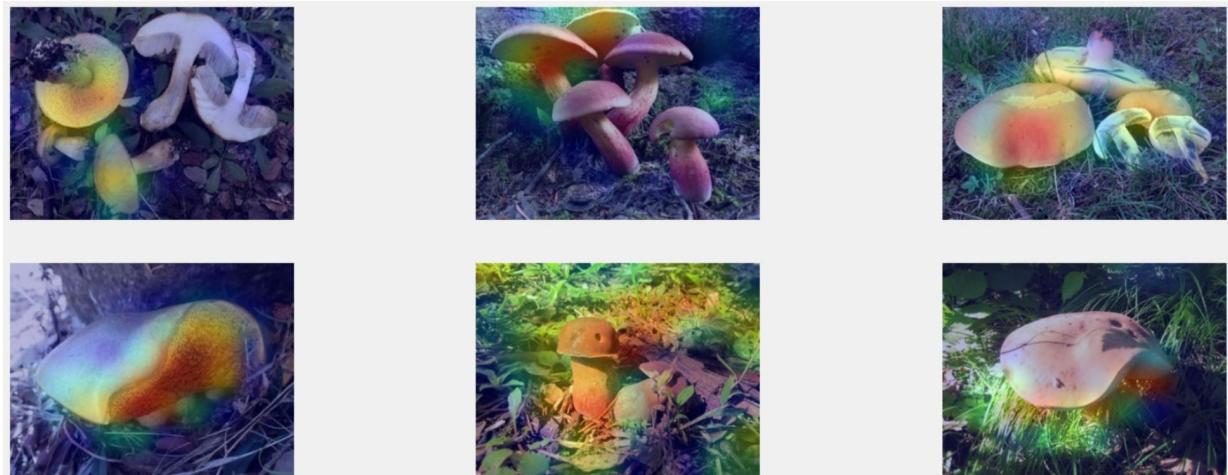


fig29. Exemples grad-cam - class3

- 6 images au hasard parmi la *class2* (deuxième meilleur *f1-score*)

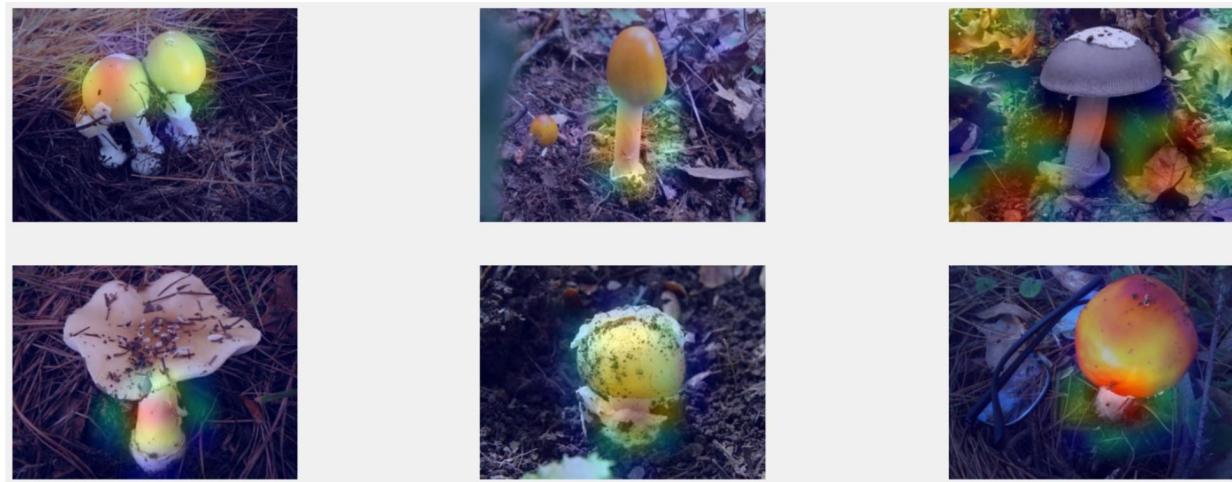


fig30. Exemples grad-cam - class2

- 6 images au hasard parmi la *class4* (moins bon *f1-score*)

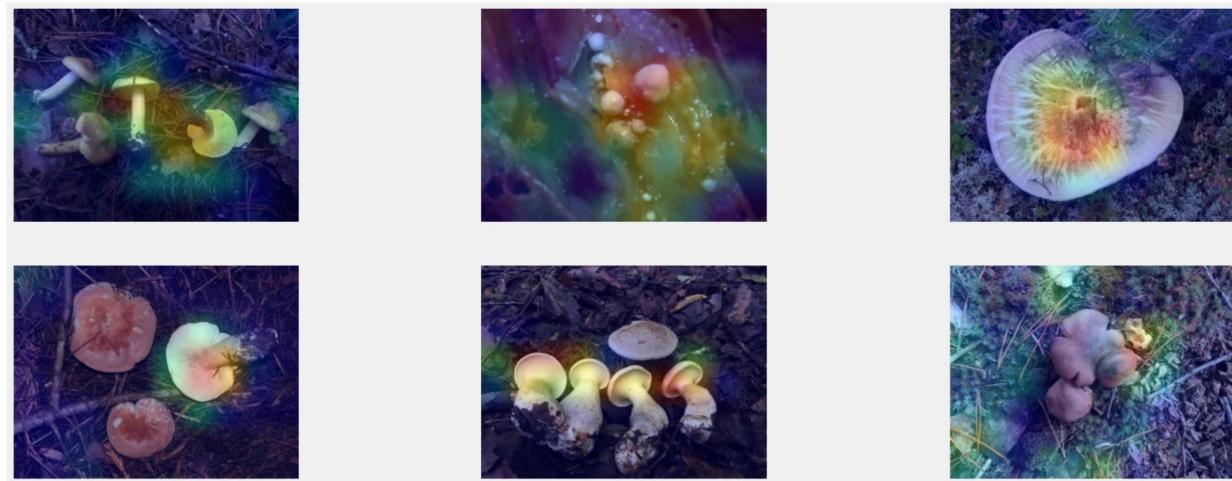


fig31. Exemples grad-cam - class4

Les résultats sont satisfaisants, malgré quelques imperfections:

- On a des images avec une bonne identification des pixels. Lorsque le champignon est vu sur le côté (càd avec la tige et le chapeau) on remarque que c'est souvent la tige qui est identifiée plutôt que le chapeau.

- Lorsqu'il y a plusieurs champignons sur la même image, tous les champignons ne sont pas forcément identifiés. Mais, cela ne devrait pas avoir d'impact sur la prédiction si au moins un champignon est bien identifié.
- Il y a aussi des images où ce sont tous les pixels excepté ceux où se trouve le champignon qui sont activés. Il faudrait faire l'analyse sur beaucoup plus d'images pour connaître la proportion de ce défaut pour confirmer si c'est un vrai problème ou non. Ce point ne sera pas traité dans ce projet par manque de temps.

9. Visualisation du *grad-cam* sur la même image mais pour les différentes classes prédictes



fig32. Exemples grad-cam - par label - avec la même image en entrée

L'image est classée à l'origine dans le *Label 3* (=Boletaceae).

Le modèle optimisé (*EfficientNetB1_final_model_seb*) ainsi que le modèle servant pour le *Grad-Cam* (*new_model*) donnent la même prédiction (Label 3).

Et visuellement, on voit que le *Grad-Cam* est le plus proche de la réalité pour le *Label 3*.

Shap

DIFFICULTÉS RENCONTRÉES LORS DU PROJET

Globalement, il n'a pas été évident de mener en parallèle le projet ainsi que les cours et les examens. Un investissement particulier notamment durant les week-ends est quasiment nécessaire.

De plus, certaines compétences techniques comme le deep learning n'étaient pas forcément acquises par rapport au cursus normal. Il a donc fallu redoubler d'efforts pour les acquérir en avance par rapport au planning normal pour être en phase avec le planning du projet.

D'autres compétences techniques n'étaient tout simplement pas dans le programme de formation telles que les optimisations Optuna et Bayesian ainsi que certaines méthodes d'interprétabilité comme *Grad-cam*.

Nous nous sommes aussi rendu compte de l'importance du matériel informatique pour le temps d'exécution. Chacun des membres ayant un ordinateur avec des caractéristiques différentes, nous avons pu comparer le temps d'exécution sur les mêmes modèles. Par exemple, pour entraîner un *VGG19* sur 5000 images, il a fallu 6 min avec *Google colab* (avec l'aide d'un *GPU* performant). Alors qu'il a fallu 5h30 avec un dell (Processeur i7-8550U CPU @ 1.80Ghz 1.99GHz; 16Go RAM; GPU 4Go). La gestion du temps en prenant en compte les performances du matériel informatique disponible est donc un point très important.

Remarque sur *Google colab*: l'utilisation d'un *GPU* étant limitée, il peut être nécessaire de prendre un abonnement payant pour enlever cette limitation. Ou alors investir dans un ordinateur performant.

Finalement, ce projet nous a permis de consolider nos compétences acquises lors de notre formation. Mais, il nous a aussi permis d'acquérir des compétences supplémentaires particulièrement sur le traitement des images et leur interprétations.

POUR ALLER PLUS LOIN

Il reste un grand nombre d'approches qui n'ont pas été testées et qui pourraient permettre d'améliorer un peu les performances sur ce jeu de données. L'utilisation d'un modèle EFB0 pour limiter l'*overfit*, l'application de différents profils de learning rate (cf [lien](#)), l'initialisation avec des poids autres que 'imagenet' ou encore le stacking de modèles plus diversifiés en font partie.

Toutefois, les résultats de la section interprétabilité semblent montrer que la principale limitation est dû au fait que les modèles ont parfois du mal à identifier la partie de l'image correspondant au champignon. Par conséquent, une approche combinant de la segmentation et du deep learning pourrait être envisagée. Comme la diversité des images semble empêcher l'utilisation d'algorithmes de segmentation simples, il faudrait se tourner vers les solutions plus élaborées telle que celle proposée dans ce [lien](#) pour supprimer le *background* des images avant leur classification.

BIBLIOGRAPHIE

Taxonomie et récupération des données:

1. <https://mushroomobserver.org/>
2. https://github.com/MushroomObserver/mushroom-observer/blob/master/README_API.md#csv-files
3. <https://github.com/bechtle/mushroomobser-dataset>
4. https://www.dropbox.com/sh/m1o91dwd1nto6w0/AABuDQVJWTq04lLyAF_G2MFa?dl=0

Google colab:

5. <https://ledatascientist.com/google-colab-le-guide-ultime/>

Bibliothèques et tutoriels:

- Gestion des images
6. <https://he-arc.github.io/livre-python/pillow/index.html>
 7. <https://pypi.org/project/python-resize-image/>
 8. <https://stackoverflow.com/questions/48001890/how-to-read-images-from-a-directory-with-python-and-opencv>
 9. <https://stackoverflow.com/questions/44078327/fastest-approach-to-read-thousands-of-images-into-one-big-numpy-array>
-
- Gestion des dossiers
10. <https://docs.python.org/3/library/glob.html?highlight=glob#module-glob>
 11. <https://docs.python.org/fr/3/library/json.html>
-
- Mesurer le temps d'exécution
12. <https://pypi.org/project/tqdm/>

-
- 13. <https://saladtomatonion.com/blog/2014/12/16/mesurer-le-temps-dexecution-de-code-en-python/>

- Classification images

- 14. <https://www.kaggle.com/zayon5/image-classification-dog-and-cat-images>

- 15. <https://github.com/fmfn/BayesianOptimization>

- Interprétabilité

- 16. https://keras.io/examples/vision/grad_cam/

- 17. <https://pypi.org/project/grad-cam/>

- 18. <https://shap.readthedocs.io/en/latest/>

- 19. <https://christophm.github.io/interpretable-ml-book/>

- Notebooks

- 20. https://github.com/DataScientest-Studio/Pyck_a_Champy

- 21. https://colab.research.google.com/drive/1ZO6_Zz3W8BX_RBihBwXktoyOwbchFf0T?usp=sharing

Liste des figures:

- *fig1. 7 familles de champignons*
- *fig2. Top5 resolution*
- *fig3. images per rank*
- *fig4. number of images per available rank*
- *fig5. Top20 species*
- *fig6. Top20 genus*
- *fig7. Top20 family*
- *fig8. Top20 order*
- *fig9. Top5 class*
- *fig10. 7 familles de champignons*
- *fig11. Segmentation Kmeans*
- *fig12. Segmentation MeanShift*
- *fig13. Autres exemples de segmentation*

-
- *fig14. Performances RandomForest*
 - *fig15. Importance features with XGBOOST*
 - *fig16. Comparaison performances de plusieurs modèles*
 - *fig17. Diagramme choix CNN*
 - *fig18. Importance hyper paramètres*
 - *fig19. Resultats EFB1 - unfreeze at start*
 - *fig20. diversité images*
 - *fig21. Résultats EFB1 - freeze at start + fine tuning*
 - *fig22. Résultats EFB1 - overfit limité*
 - *fig23. évolution accuracy versus C et couche*
 - *fig24. Résultats Stacking - 5 meilleurs modèles*
 - *fig25. Résultats AutoGluon*
 - *fig26. exemple champignon - class3*
 - *fig27. exemple heatmap*
 - *fig28. exemple grad-cam*
 - *fig29. Exemples grad-cam - class3*
 - *fig30. Exemples grad-cam - class2*
 - *fig31. Exemples grad-cam - class4*
 - *fig32. Exemples grad-cam - par label - avec la même image en entrée*