

# Rapport final ChampySeed

<b>Contexte</b>	<b>2</b>
<b>Outils et méthodes utilisées durant le projet :</b>	<b>2</b>
Découverte du projet	2
Renoncement rapide à utiliser Mushroom Observer	2
GBIF.org	4
Taxonomie	5
<b>Construction du Dataset</b>	<b>6</b>
Construction du dataframe de téléchargement	6
Récupération des données	8
Anticipation de la récupération des données	8
Structure du script de téléchargement	9
Optimisation du téléchargement	9
Optimisation des images pertinentes	10
Blacklister certains sites (sur le nom de domaine)	10
Optimisation tardive pour éliminer des images non pertinentes avec du Transfer Learning :	11
Partage des images	13
Construction du dataframe pour le modèle	13
<b>La modélisation</b>	<b>13</b>
Problématique	13
Le Transfer Learning	13
ImageDataGenerator	13
Code optimisé tf	16
Optimisation de l'entraînement	17
Second apprentissage en dégelant des couches du modèle	17
DirectML & Multi-GPU :	18
Optimisation tardive :	19
Modèles utilisés	19
EfficientNetB0	19
InceptionV3	19
VGG16 (ou VGG19) et ResNet50	20
Optimisations de l'entraînement	22
Mise en place des Callbacks	22
Learning rate	22
ReduceLROnPlateau :	23
Prédictions	24
Interprétabilité	25
<b>Utilisation d'un maximum de données</b>	<b>26</b>
<b>Conclusions</b>	<b>28</b>
<b>Pour aller plus loin...</b>	<b>29</b>
Entraînement distribué	29
BoundingBox	29
Optimisation de la data augmentation	29
Couches de Classification	30
Neurone de classification	30
<b>Annexe 1 : Machines à disposition</b>	<b>30</b>
<b>Annexe 2 : Liste des fichiers du code</b>	<b>31</b>
<b>Annexe 3 : Entraînement du modèle sur une instance déportée AWS</b>	<b>32</b>
Étapes réalisées :	32
Performances	32
<b>Annexe 4 : Diagramme de Gantt</b>	<b>33</b>

# Contexte

Notre projet ChampySeed consiste à entraîner un modèle de classification d'images de champignons à partir des données du site Mushroom Observer.

Le but est ensuite d'utiliser ce modèle pour identifier le champignon présent sur une photo choisie par un utilisateur.

Idéalement, la comestibilité et la localisation du champignon pourra être affichée.

## Outils et méthodes utilisées durant le projet :

Nous avons convenu dès le début du projet de mettre en place des process simples à l'efficacité reconnue pour garantir une progression fluide du projet.

Communication:

- Daily scrum meeting (15h tous les jours)
  - Point sur les avancées
  - Partage d'infos
  - Travaux pour le lendemain
  - Entraide sur les problèmes rencontrés
- Mise en place d'un Discord
  - Discussions audio/vidéo
  - Partage d'écrans
- Slack Datascientest pour les interactions rapides et call
- Mise en place d'un Google Drive "ChampySeed" partagé
  - Rédaction collective des livrables
  - Historique des calculs intermédiaires
  - Echange de fichiers

Une fois que le nombre d'images de champignons est devenu conséquent, les fichiers ont été centralisés sur un serveur FTP (Filezilla server) hébergé chez le membre de l'équipe qui avait la meilleure connexion en upload (Bastien)

## Découverte du projet

### Renoncement rapide à utiliser Mushroom Observer

La fiche du projet proposait 2 sources d'informations:

- Le site Mushroom Observer (MO) qui est un site participatif où les contributeurs postent une photo, indiquent le lieu d'observation et proposent un nom de champignon validé par la communauté après consensus. Les contributions proviennent presque uniquement des Etats-Unis.
- Un Github Mushroom Observer Dataset dont les photos (sur un Dropbox) proviennent du site précédent. Des métadonnées sont disponibles. Après examen du code il s'avère que ces informations proviennent d'un autre site GBIF.org

Le site MO propose une API mais celle-ci s'avère très contraignante en raison des limites sur le nombre de requêtes mais aussi la nécessité de gérer la pagination. (limite de 100 résultats par requête) (voir exemple de traitement en annexe 2, creationdataset mushroomobserver-herborist.zip)

Le site souffre aussi de performances dégradées en ce moment.

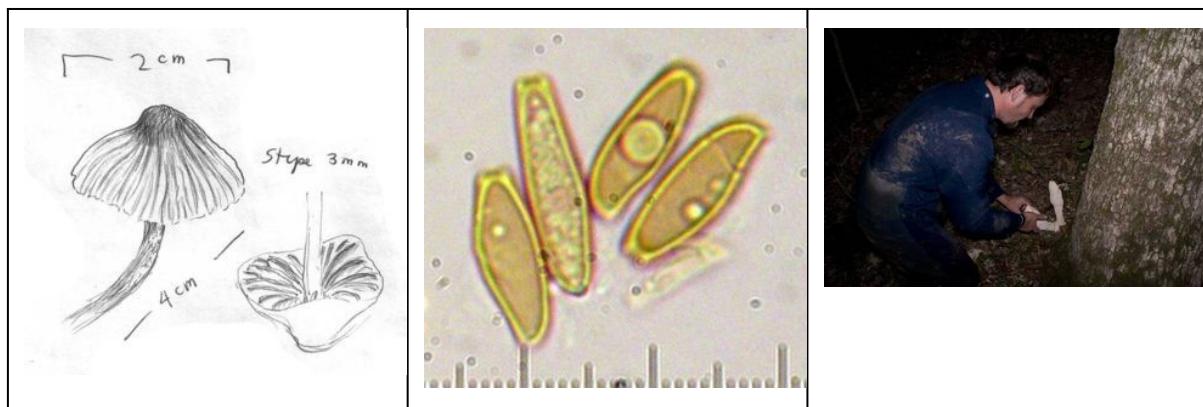
Finalement le fichier le plus pertinent sur le site MO est un fichier contenant 3 types d'infos: URL d'une photo, Nom du champignon, Date de l'observation et copyrights

	A	B	C	D	E
1	image	name	created	license	rightsHolder
2	<a href="#">images/640/2.jpg</a>	Xylaria magnolia	2004-07-17	<a href="http://creativecommons.org/licenses/by-sa/2.0/">http://creativecommons.org/licenses/by-sa/2.0/</a>	Nathan Wilson
3	<a href="#">images/640/16.jpg</a>	Volvopluteus glo	2005-01-07	<a href="http://creativecommons.org/licenses/by-sa/2.0/">http://creativecommons.org/licenses/by-sa/2.0/</a>	Nathan Wilson
4	<a href="#">images/640/26.jpg</a>	Panellus stipticus	2004-11-26	<a href="http://creativecommons.org/licenses/by-sa/2.0/">http://creativecommons.org/licenses/by-sa/2.0/</a>	Nathan Wilson
5	<a href="#">images/640/36.jpg</a>	Sutorius eximus	2004-07-14	<a href="http://creativecommons.org/licenses/by-sa/2.0/">http://creativecommons.org/licenses/by-sa/2.0/</a>	Nathan Wilson

Il est alors possible d'utiliser la colonne "Name" pour chercher les informations sur GBIF.org. Deux méthodes ont été explorées:

- En utilisant l'API de gbif.org pour ajouter les métadonnées ligne par ligne dans le fichier MO
- La récupération d'un Dataset [Herbarium GB, University of Gothenburg \(gbif.org\)](#) puis un merge avec le fichier MO

Nous avons ensuite visualisé quelques photos : Un examen partiel des images contenues sur le site a mis en lumière la qualité très médiocre des photos. Voici quelques exemples d'images plus ou moins faciles à détecter et éliminer :





De plus il y a très peu de photos par champignon, seulement 1 photo pour la grande majorité, 6 pour quelques autres et la seule exception est 70 photos pour l'Amanita.

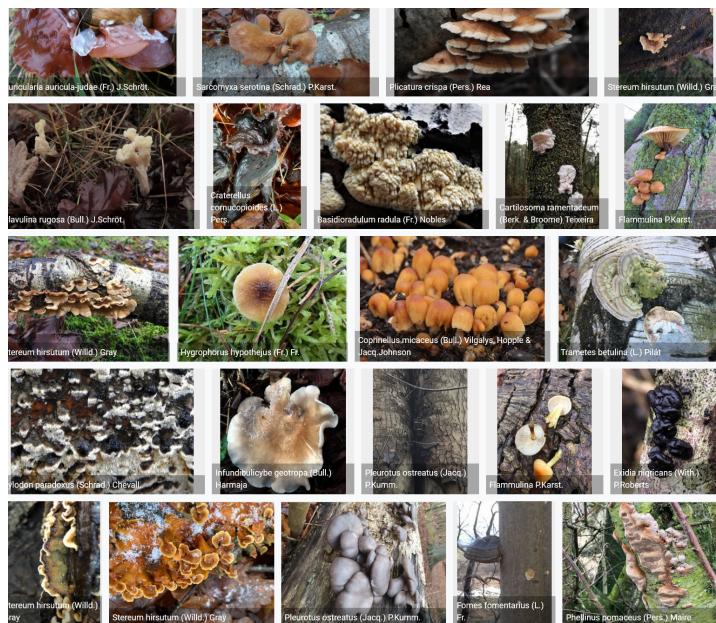
En conclusion, le contenu du site MO s'est révélé très décevant mais nous a permis de découvrir un site plus qualitatif et professionnel, GBIG.org

Les photos de MO feront toutefois une bonne source d'images pour nos démos dans la mesure où nous pourrons cette fois sélectionner des photos pertinentes.

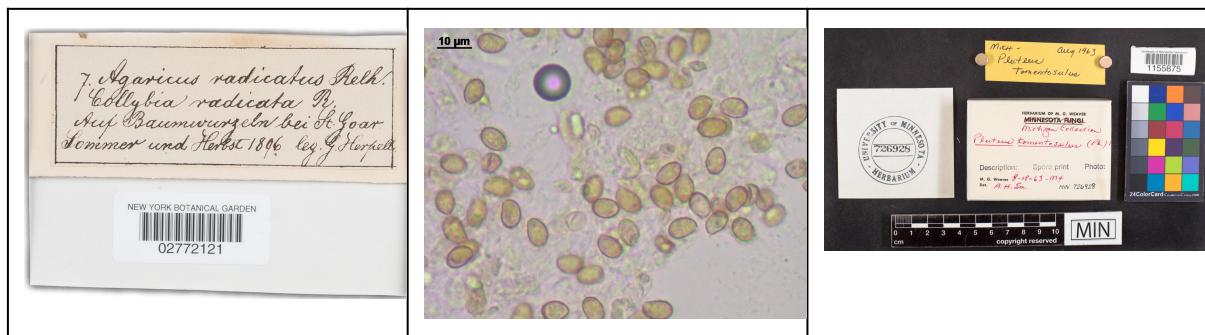
## GBIF.org

Le GBIF - Global Biodiversity Information Facility(Système mondial d'information sur la biodiversité) - est un réseau international et une infrastructure de données financés par les gouvernements mondiaux ayant pour but de fournir à tous et partout un accès libre aux données sur toutes les formes de vie sur Terre. Il est coordonné via son Secrétariat de Copenhague.

Il est possible de se faire une première idée des photos disponibles sur GBIF.org directement depuis ce lien : [https://www.gbif.org/occurrence/gallery?taxon\\_key=186](https://www.gbif.org/occurrence/gallery?taxon_key=186)



Nous avons également téléchargé aléatoirement des photos en utilisant les url du dataset. Les photos sont clairement plus qualitatives que MO, il reste néanmoins des photos à écarter comme sur ces exemples :



## Taxonomie

Après nous être familiarisés avec ces premières bases, nous avons réalisé la complexité de la classification des champignons dont voici tous les niveaux : kingdom, phylum, class, order, family, genus, species.

Exemple avec l'Amanita muscaria, connue sous le nom d'Amanite tue-mouches en français et Fly agaric ou Fly amanita en anglais.



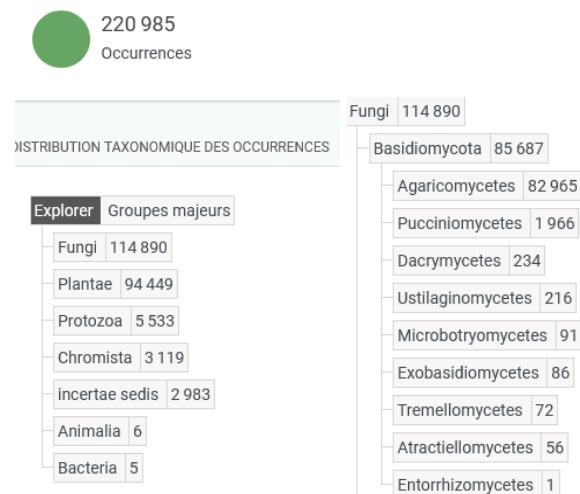
kingdom	Fungi
phylum	Basidiomycota
class	Agaricomycetes
order	Agaricales
family	Amanitaceae
genus	Amanita
species	Amanita muscaria

Étant donné l'étendue de la taxonomie, nous avons décidé de nous limiter à la classe des **Agaricomycetes** pour notre projet. Comme l'idée était de reconnaître les champignons pouvant être comestibles et trouvables facilement par un cueilleur de champignons, la recherche fut non exhaustive. Lister les espèces les plus connues (amanite, bolet, girole...) et comparer leur sous classe. Il se trouve que ces champignons font tous partie de l'ordre des Agaricomycètes.

## Construction du Dataset

### Construction du dataframe de téléchargement

Dans un premier temps, nous sommes partis d'un dataset existant : [Herbarium GB, University of Gothenburg \(gbif.org\)](#) pour n'utiliser que le sous-ensemble "fungi" afin de merger les métadonnées GBIF avec les images de MO. Il s'est avéré que cette base n'était pas complète sur les Agaricomycetes. En effet, cette base ne contenait pas uniquement des données sur les champignons. Le nombre d'occurrences sur les Agaricomycète tombait lui à 82 965, sachant qu'il y avait plusieurs occurrences pour un même champignon.



Dans un second temps, nous avons découvert qu'il était possible de créer une base à partir de toutes les occurrences référencées sur le site. Cela a été fait en appliquant les deux filtres suivants :

- Scientific Name = "Agaricomycetes"
- Media Type : Image (1 355 857 media)

Afin de ne récupérer que notre 'classe' de champignons dont les occurrences comportaient des photos.

Ceci nous a permis de télécharger un zip (format Darwin Core Archive) contenant notamment les fichiers :

- Occurrences.txt (Inclut les métadonnées)
- Multimedia.txt (Inclut les url des photos)

Le chargement en utilisant pd.read\_csv("occurrences.txt", sep = "\t") s'est avéré compliqué à cause de certains caractères parasites mais nous avons pu trouver un module **Python-dwca-reader** dédié à l'exploitation des fichiers Darwin Core Archive. Ce fichier au vu de son poids a donné beaucoup de mal lors de son traitement par nos PC, malgré leurs 16Go de RAM. Cela pouvait fréquemment générer des erreurs de mémoire. (voir annexe 2 traitement darwin core.ipynb)

Le tableau occurrence est organisé ainsi :

- Chaque ligne correspond à un champignon
- Un même champignon apparaît autant de fois qu'il est référencé : Pays où il a été trouvé, les coordonnées GPS, l'altitude ou la profondeur, les remarques, les difficultés rencontrées...

Le tableau Multimédia ne comportait quasiment que les liens **url** vers les images, qui étaient hébergées sur des sites divers et variés. Une fois ces deux datasets épurés, ils ont été fusionnés suivant la variable commune dans les 2 fichiers (ID et CoreID). (annexe 2 nettoyage du tableau.ipynb)

Voici le tableau final, les décisions prises pour garder les colonnes "finales"

Variable/ Colonne	description	Pourquoi la garder
ID	Identifiant GBIF unique	Identifiant de chaque champignon
countryCode	Pays	En cas d'affichage par pays
decimalLatitude	Coordonnées	Créer une carte
decimalLongitude		
hasCoordinate	bool	Tester la présence de coordonnées avec cette colonne
<b>Colonnes pour l'IA</b>		
family	Familles	Variable candidate
genus	Genre	Variable candidate
species	L'espèce du champignon, niveau le plus bas dans la Taxonomie	Variable candidate, le nom latin du champignon
verbatimScientificName	Nom scientifique.	Il s'agit du même nom que "species" parfois augmenté du nom de celui qui l'a découvert. Nous avons conservé cette variable au cas où nous trouvions des informations complémentaires utilisant ce nom.

identifier_y	Lien images	L'url des images
--------------	-------------	------------------

Les 2 variables cibles retenues ont été d'abord les "genus/genre" pour avoir un grand nombre d'images disponibles, puis nous avons pu entraîner les modèles directement au niveau des "species/espèces" en fin de projet.

## Récupération des données

### Anticipation de la récupération des données

Nous avons commencé à penser au téléchargement des images. A la vue de la taille de ce dataframe (2.3 millions de lignes), il était clair qu'il serait impossible de télécharger toutes les images dans leur résolution originale. Un rapide calcul, en considérant une taille moyenne de 1Mo par image, nous donnait plus de 2To d'images à stocker. Ce n'était pas raisonnable si l'on voulait pouvoir garder les images sur un SSD en vue de leurs lectures rapides. Il a donc été choisi d'intégrer le resizing à la boucle qui téléchargerait les images. La taille de 224\*224 a été choisie. Cette taille nous a semblé un bon compromis. Pas trop lourdes mais assez pour pouvoir extraire des informations. Cette résolution est communément utilisée dans la classification d'images dans le milieu du deep learning ce qui pouvait éviter une étape de redimensionnement systématique à chaque lecture des fichiers.

Ensuite, il nous a semblé intéressant de sauvegarder les images dans une architecture de dossier qui reprendrait l'architecture de la taxonomie des champignons. C'est-à-dire qu'il existe un dossier Fungi/Phylum qui contient toutes les classes à garder. Chaque dossier classe contient tous les dossiers family associés. Et ceci jusqu'à la branche species. Il n'a pas été utile de créer le sous-dossier "verbatimScientificName" puisque au finale, cette colonne de notre dataframe est très similaire à species. Le nom du scientifique n'étant pas toujours spécifié, et d'une relative utilité pour nous, il a été choisi de ne pas l'inscrire dans l'architecture de notre dossier de sauvegarde des images.

Il a aussi été choisi de nommer chaque image par un identifiant unique conservé dans le dataframe dans la colonne "notreid". Chaque image aurait alors un nom de fichier "im4545.jpg" où le nombre 4545 est l'identifiant unique associé à la ligne de notre dataframe. De cette manière, lors de difficultés avec des images, nous serions en capacité de retrouver rapidement la ligne du dataframe en correspondance.

### Structure du script de téléchargement

La première étape était de charger notre csv. Puis dans une même boucle, ces différentes étapes étaient effectuées:

- Téléchargement par requête URL à partir d' "identifier\_y"
- Ouverture de l'image téléchargée pour vérifier la bonne réception (PIL)
- Redimensionnement de l'image en 224\*224
- Sauvegarde du fichier dans le dossier correspondant à sa taxonomie. (im587876.jpg)

## Optimisation du téléchargement

La première optimisation fut de ne pas lancer le téléchargement sur tout le dataset d'un coup. Cela avait pour conséquence de ne remplir tous les dossiers que très lentement. Autrement dit, nous n'aurions pu utiliser nos images que lorsque toutes les images auraient été téléchargées. Et même avec un téléchargement de 100 000 images, la très grande taxonomie des champignons faisait que nous n'avions que quelques images pour chaque type de champignon. Il a donc été choisi de se restreindre à certaines "familles" de champignons. Notre choix s'est porté sur le "genus". Le "genus" nous offrait un grand nombre d'images par "genus" tout en permettant de couvrir un grand domaine de la taxonomie. Nous avons alors organisé le téléchargement par "genus" en ne conservant que les "genus" ayant plus de 2000 images à notre disposition.

La gestion des exceptions nous ont permis de prendre en compte, petit à petit, de plus en plus d'erreurs venant interrompre notre boucle. A chaque arrêt il a fallu comprendre d'où venait l'erreur et choisir ce que devrait faire la boucle si jamais elle rencontrait à nouveau cette erreur.

Afin de ne pas parcourir à nouveau tout le "genus" choisi, nous avons aussi créé une petite boucle permettant de retirer du dataframe toutes les lignes déjà téléchargées.

Malgré toutes ces étapes, le script de téléchargement était extrêmement lent et très hétérogène dans la vitesse de téléchargement. La vitesse de la connexion n'étant pas le frein (fibre optique), nous avons cherché à comprendre d'où venait ces disparités. En regardant précisément l'activité de la carte réseau nous avons pu comprendre que cela venait principalement de la réponse des différents serveurs auxquels nous demandions les images. Ce problème ne pouvant être "géré" par notre boucle, nous avons pensé à multiplier les scripts de téléchargement. Cela nous a permis de multiplier par 10 notre vitesse de téléchargement.

Si nous avons pu optimiser la vitesse de téléchargement des images nous avons néanmoins remarqué que la base contenaient également des images qui n'apportaient rien à notre projet et pouvaient même perturber l'entraînement : les images non pertinentes !

## Optimisation des images pertinentes

### Blacklister certains sites (sur le nom de domaine)

Nous avons amélioré notre process pour les retirer du dataset. Après avoir identifié visuellement une image non pertinente, nous recherchons le site web d'où elle provient (domaine). Ensuite nous évaluons la proportion d'images pertinentes / non pertinentes ainsi que le nombre de fichiers provenant de ce site. Si la proportion d'images non pertinentes est très élevée, nous "blacklistons" les contenus de ce domaine. C'est-à-dire que nous excluons ces fichiers de la boucle de téléchargement. Cette technique a permis de réduire d'un quart les répertoires (des "genus" ne comportant que 2 photos par exemple).

Un notebook dédié permet d'évaluer facilement la qualité d'un domaine suspect en ouvrant aléatoirement des images téléchargées avec leur URL :



Voici la liste actualisée des sources blacklistées:

```
domain_name = "data.huh.harvard.edu"
domain_name = "fm-digital-assets.fieldmuseum.org"
domain_name = "mycoportal.org"
domain_name = "s3.msi.umn.edu/"
domain_name = "mediaphoto.mnhn.fr/media/"
domain_name = "unimus.no/felles/"
domain_name = "oxalis.br.fgov.be/images"
```

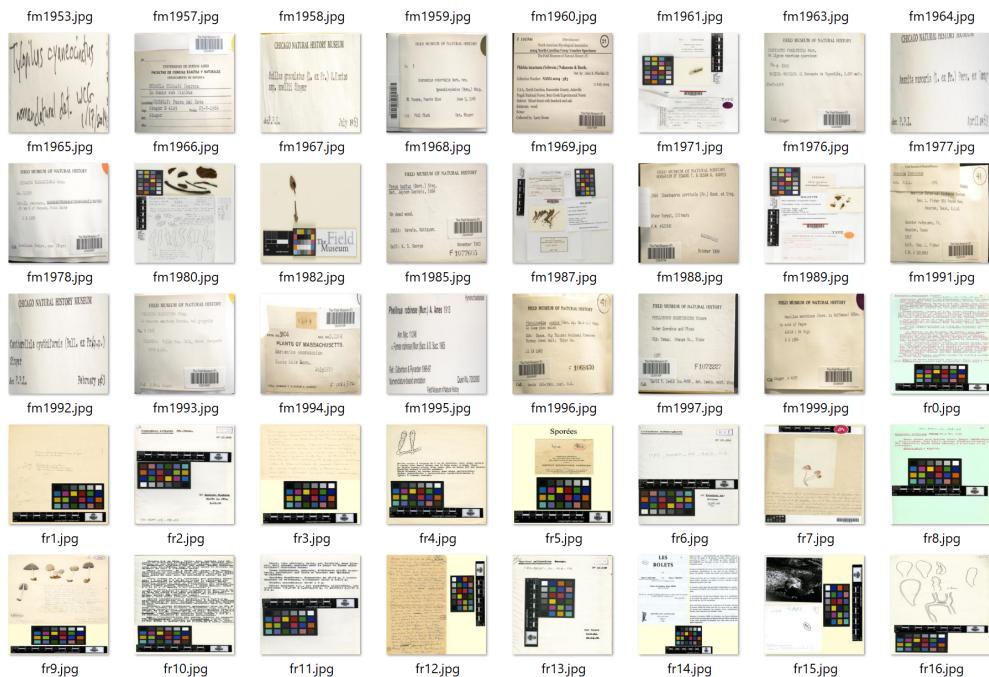
Cette étape nous a permis de ne pas perdre de temps à télécharger des images inutiles en provenance de certains sites, mais il restait encore une petite portion d'images non pertinentes.

Optimisation tardive pour éliminer des images non pertinentes avec du Transfer Learning :

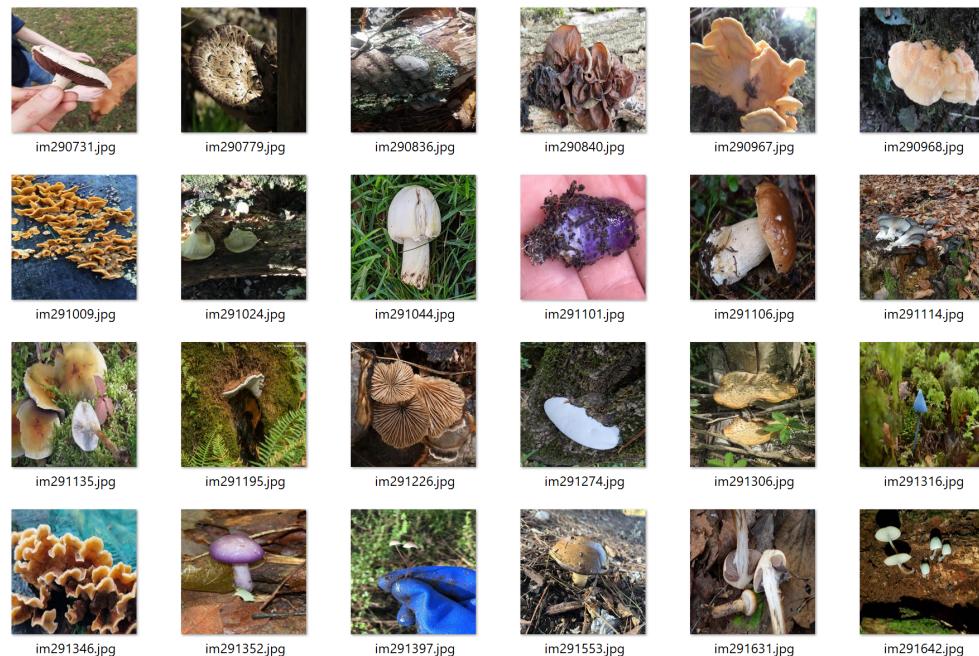
La première piste envisagée tournait autour de l'analyse d'images (Histogramme ou autre) mais après être devenus familiers avec le Transfer Learning, vers la fin du projet, nous avons appréhendé ce nettoyage de la façon suivante : avoir un modèle pour détecter qu'il n'y a **pas** de champignon dans une image.

Comme nous avions identifié les sites avec une majorité d'images non pertinentes, il a été très facile d'en télécharger 12000. Puis nous avons pris au hasard 12000 images de champignons déjà téléchargées.

Nous avons refait une vérification visuelle rapide pour s'assurer du bon contenu dans nos 2 répertoires, en voici des copies d'écrans :  
 Documents papier et images microscopiques :



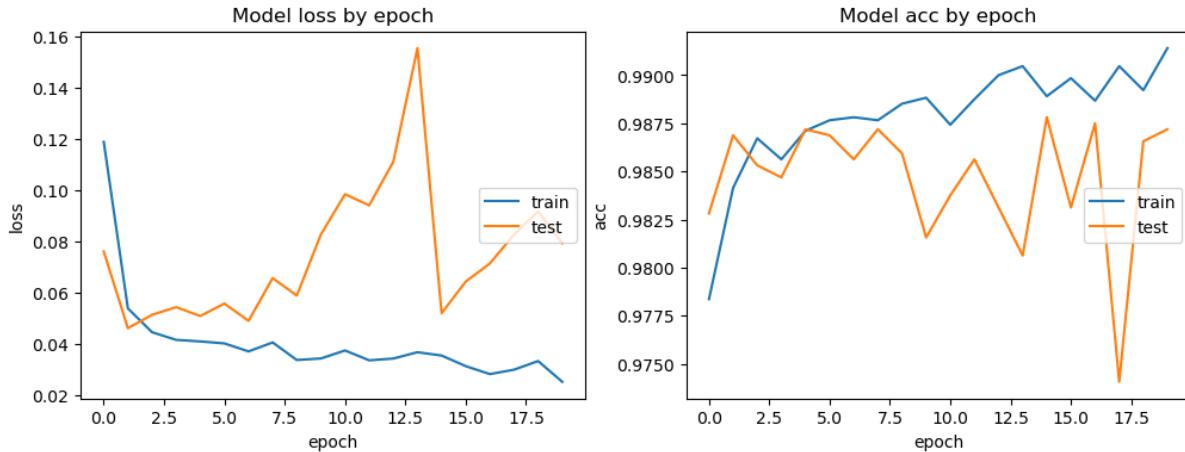
Images de vrais champignons :



Nous avons alors réutilisé le premier modèle VGG16 (les modèles seront décrits dans le chapitre Modélisation, seul le résultat est présenté ici) que nous avions testé quelques semaines auparavant en l'alimentant avec les images de ces 2 répertoires. Nous n'avons plus que 2 classes en sortie Fungi/NotFungi.

La précision de ce modèle était, comme attendu, très élevée : (plus de 98% de val\_acc)

### VGG16, 2 classes Fungi/notFungi 12000 images par classe



Nous avons une nouvelle fois ajouté un test visuel en affichant aléatoirement des images identifiées comme “Not Fungi”. (annexe 2 tirage image notfungi.py)

Pour l'étape suivante, il a fallu faire un test de prédiction sur toutes les images téléchargées, celui-ci a permis d'écartier **2%** des images disponibles.

Nous avons ainsi un set d'images qualitatif nous permettant de minimiser les risques de dégradation de nos résultats à cause de mauvaises données sources.

### Partage des images

Afin que tout le monde ait accès aux images, un serveur Filezilla a été créé et les images ont pu être téléchargées par tout le groupe. Le téléchargement brut a été effectué par la meilleure connexion internet. Le transfert FTP étant beaucoup plus rapide une fois les images redimensionnées.

Après cette étape, nous avions plus de 1.3 millions d'images sauvegardées.

### Construction du dataframe pour le modèle

Une fois tous les téléchargements effectués, et pour ne pas se retrouver avec des lignes ne menant pas à des images, nous avons reconstruit un fichier csv à partir de tout ce qui était réellement sur le disque dur. Cela nous a permis d'éviter de faire des vérifications en amont du modèle. Ce fichier a été nommé “CsvFinalDlded.csv”

## La modélisation

### Problématique

Nous souhaitons créer un modèle qui classe les champignons à partir d'une photo. Après avoir fait plusieurs essais avec de plus en plus de classe et de photos, des raisons temporelles et matérielles ont arrêté notre choix à deux types de modèles. Un qui classe les champignons selon leur “genus” (64 , 8000 photos par “genus) et un qui classe les photos

par “species” (64, 4000 photos par espèce). L’objectif sera, lors de la démo streamlit, qu’un utilisateur puisse importer une photo et découvrir ce que prédit le modèle.

L’orientation s’est naturellement faite vers le deep learning.

## Le Transfer Learning

La construction “from scratch” d’un modèle n’a jamais été envisagée. La complexité de mise au point de l’architecture d’un modèle de deep learning ainsi que l’entraînement de ses couches sont deux obstacles incompatibles avec nos ressources sur ce projet. Notre choix s’est naturellement porté vers des modèles pré-entraînés par des experts qui y ont consacré des mois/années avant de les présenter à la communauté.

Pour qu’un modèle puisse classifier les images, il faut respecter plusieurs étapes :

- Respecter le format d’entrée (dimensions, standardisation des pixels...)
- Créer un itérateur d’images

Dans notre cas, nous avons également eu recours à un amplificateur d’images pour renforcer l’apprentissage.

## ImageDataGenerator

Cette fonction a été utilisée dès le début de notre projet. Cela est surtout dû à sa simplicité de mise en œuvre ainsi qu’à ses nombreuses options disponibles.

Le générateur peut prendre en charge plusieurs opérations d’augmentation des images comme les rotations, les translations, les symétries horizontales ou verticales, les zooms ou encore la luminosité.

Le générateur d’image étant gourmand en temps de calculs, nous avons limité le nombre d’opérations :

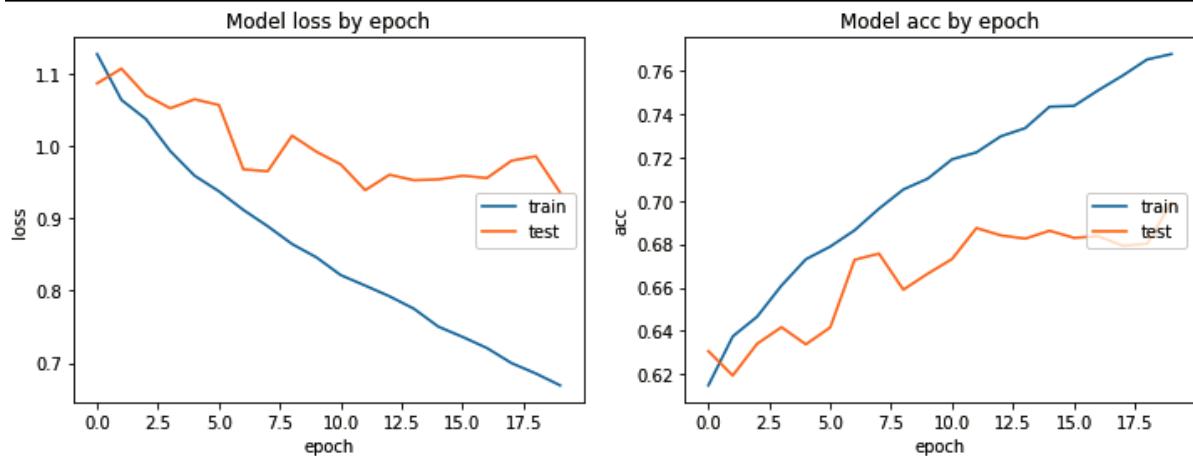
- preprocess\_input() nécessaire pour effectuer les opérations
- Et pour les augmentations :
  - une rotation des images avec un paramètre de 180°
  - un zoom maximum de 20%
  - une variation de la luminosité de 10%

La rotation permet d’éviter la prédominance de photos avec des champignons représentés pied en bas et chapeau en haut.

Pour illustrer l’importance des opérations d’augmentation sur les images, voici les résultats obtenus **sans** et **avec** augmentations dans ImageDataGenerator :

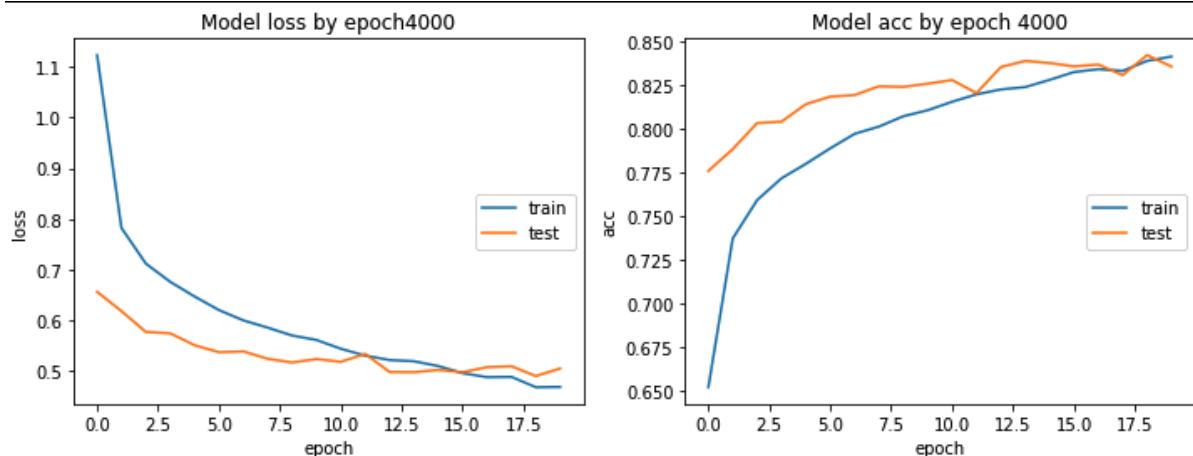
Exemple avec 40 000 images, ImageDataGenerator **avec uniquement** le preprocess\_input:

### VGG16 avec 40 000 images sans augmentation



Exemple avec 40 000 images, ImageDataGenerator avec **preprocess\_input ET rotation 180**

### VGG16 avec 40 000 images avec augmentation



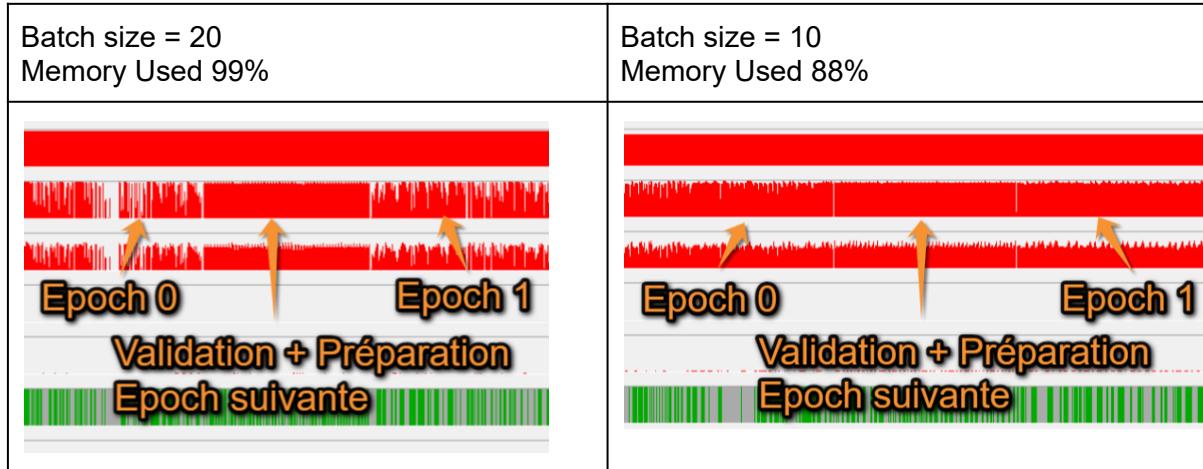
On note que le modèle apprend par cœur des images du train set et peine à généraliser sur le test set.

Problème : Nous avons constaté un temps d'entraînement pouvant être multiplié par deux ou trois avec l'amplification des données.

Nous avons pu constater que plusieurs paramètres peuvent influencer l'utilisation optimale du GPU. Les performances vont différer en fonction des points suivants :

- HDD et SSD : Il se trouvait qu'un HDD ralentissait l'entraînement, n'étant pas assez rapide.
- Il a été aussi vu que l'analyse en temps réel des anti-virus scannait chaque image ouverte par ImageDataGenerator, cause également d'une perte de temps.
- Batch Size : Ce paramètre semble jouer un rôle dans le temps GPU

Exemple visuel ci-dessous avec un batch de 20 puis un batch de 10.



Avec un batch de taille 20, on remarque que le GPU oscille très rapidement entre 0 et 100%, il est sous-utilisé. Avec un batch de taille 10 le graphe est plus dense mais on remarque qu'il utilise moins de mémoire du GPU.

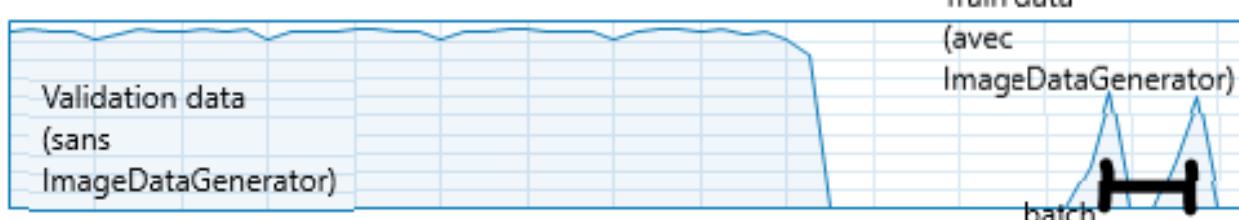
La taille des batchs devra donc être choisie pour s'adapter à chaque configuration. Nous conseillons de faire 1 ou 2 Epochs avec plusieurs valeurs pour définir une taille de batch adaptée.

On remarque également que le GPU est à son maximum lors d'une validation, cela ne nécessitant pas de d'augmentation des images.

Nous nous sommes demandé si `ImageDataGenerator` n'utilisait pas mal notre CPU (ou alors uniquement le GPU) pour générer ses batchs d'images. Nous avons donc décidé d'élaborer un autre générateur d'image.

## Code optimisé tf

### ▼ Cuda



Ce petit graphique résume à lui seul notre volonté de nous passer d'`ImageDataGenerator`. Le batch a été choisi volontairement énorme (512) afin de mieux identifier les différentes conséquences de `ImageDataGenerator` sur les performances de rapidité de l'entraînement. A chaque batch, le travail du GPU est stoppé. Il semble que c'est ici que `ImageDataGenerator` intervient pour créer le batch suivant avec la data augmentation.

Nous avons donc essayé de nous passer de lui.

La principale difficulté de cet abandon est de devoir nous frotter à une autre structure de donnée : les Tensors de Tensorflow. `ImageDataGenerator` faisait, avant, ce travail pour nous, sans que nous le sachions.

Les tensors sont comme des numpy arrays immuables. A chaque nouveau batch, il faut donc créer un nouveau tensors qui contiendra nos données. Une autre difficulté avec ces objets, c'est que leur contenu est difficilement lisible. Après beaucoup d'échecs, nous avons réussi à concevoir un modèle qui fonctionne. Cependant le choix limité des fonctions d'augmentation du type "tf.rot90", ne nous a pas permis de faire exactement les mêmes transformations qu'effectuaient notre modèle avec ImageDataGenerator. Les rotations sont limitées à des rotations de 90°.

Le modèle est exactement le même cependant avec les datasets de Tensorflow, nous devons définir en amont du modèles toutes les opérations que le dataset doit subir avant de rentrer dans le modèle :

- Aller chercher les données dans notre dataframe
- Mélanger les données (shuffle)
- Extraction de l'image à partir du chemin sur le disque dur
- Effectuer la data augmentation
- Spécifier la taille de chaque dataset
- Expliciter quelles tâches peuvent être effectuées en parallèle.

Nous avons donc une sorte de pipeline qui était parcouru à chaque batch. Il y a deux pipeline selon si les données sont des données d'entraînement (avec data augmentation) ou de validation (sans).

Cette méthode nous a permis de gagner énormément de temps (2-3 fois plus vite) sur l'apprentissage. Cependant notre script était moins performant qu'ImageDataGenerator en termes de précision. (10% moins précis )

De nouvelles méthodes ont été aperçues afin de mieux transcrire toutes les opérations de data augmentation, mais nous manquons de temps pour les implémenter.

## Optimisation de l'entraînement

### Second apprentissage en dégelant des couches du modèle

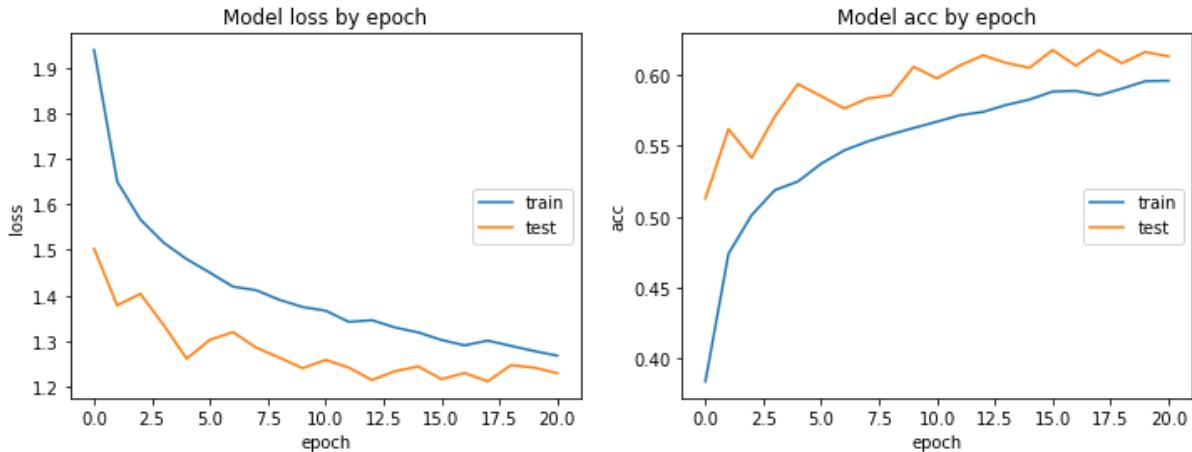
Il est possible d'améliorer un modèle une fois que les paramètres ont été réglés. Pour cela il faut dégeler une partie des couches (unfreeze) puis relancer un apprentissage. Les poids du modèle pré-entraîné seront affectés par ce second passage.

A noter que les couches à rendre "trainable" diffèrent selon la structure des modèles. Pour un VGG16 il suffit de rendre entraînable le dernier bloc de convolution (les 4 dernières couches) alors que cela peut être plus compliqué pour un autre modèle pas structuré en "funnel / entonnoir".

L'exemple suivant a été réalisé sur le modèle ResNet50 en dégelant toutes les couches:  
Test AWS ResNet50, 20 genus à 2000 photos par genus.

1er apprentissage : (la précision sur le jeu de test dépasse légèrement 60%)

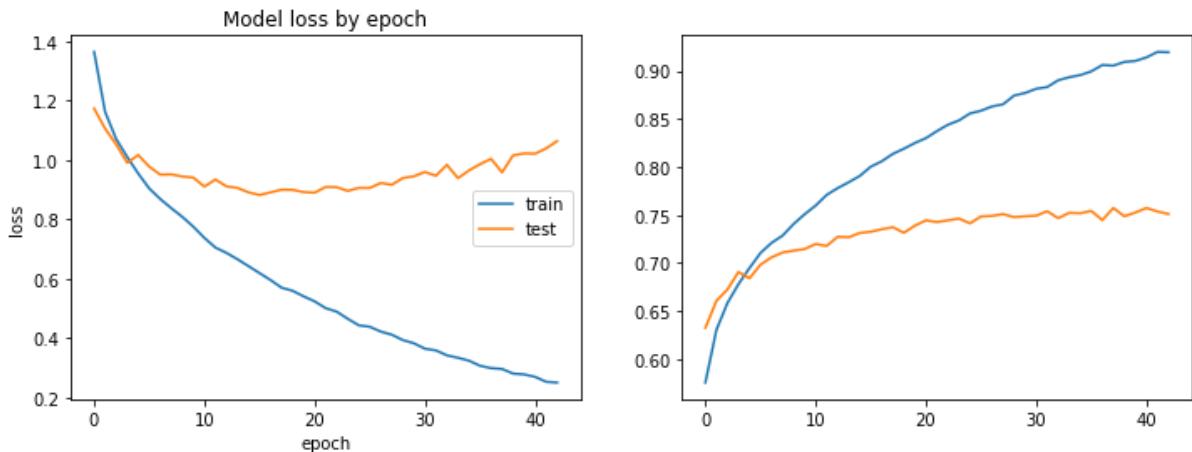
### ResNet50, 20 genres avec 400 000 images



Après unfreeze :

Resnet50, entraînement après unfreeze des couches, 20 classes genres, 2000 photos par genre (la précision du jeu de test atteint les 75%)

### ResNet50, 20 genres avec 40000 images et Unfreeze des couches



L'amélioration de la précision est significative dans ce cas précis, on notera que le second apprentissage est beaucoup plus lent que le premier.

DirectML & Multi-GPU :

Nous avons utilisé Tensorflow-DirectML pour profiter des GPU AMD mais aussi pour expérimenter du “Distributed Training”. En effet l’un des portables possède 2 cartes vidéo (Nvidia et Intel) compatibles avec DirectML. L’idée était d’essayer de répartir la charge de calcul sur les 2 processeurs. Hélas, si nous avions bien (2 GPUs are detected : `['/device:DML:0', '/device:DML:1']`) la fonction “tf.distribute.MirroredStrategy()” ne trouvait aucun GPU dans l’environnement DirectML. Nous avons abandonné cette piste qui aurait pourtant pu être intéressante dans la mesure où la carte Nvidia n’était pas très puissante. Cette piste ne présente pas d’intérêt quand on a une carte avec un GPU très puissant.

A noter que la version de DirectML for TensorFlow est antérieure à TensorFlow 2.6.0 ce qui explique une partie des incompatibilités rencontrées.

## Optimisation tardive :

Utilisation d'instances déportées sur AWS, voir ce qui a été fait dans l'annexe 3

## Modèles utilisés

Nous avons utilisé différents modèles pré-entraînés. De plus, les couches de classification sont identiques entre les modèles afin de faciliter la comparaison des résultats.

Afin de mesurer la performance de nos modèles, la métrique choisie est naturellement la précision (Accuracy)

Nous affichons des variantes de celle-ci :

- 'acc' et 'val\_acc' sont les précisions du train et du test.
- top\_k (3 et 5) est la probabilité que la target soit présente dans le top k des probabilités prédites.

Nous avons enregistré nos modèles de manière à maximiser "val\_acc". C'est en effet cette métrique qui synthétise le plus la performance des modèles. Choisir "acc" n'aurait pas de sens à cause de la data\_augmentation et en cas de sur-apprentissage.

Les métriques top\_k sont d'autres outils de visualisation mais rien n'a été paramétré sur leurs résultats.

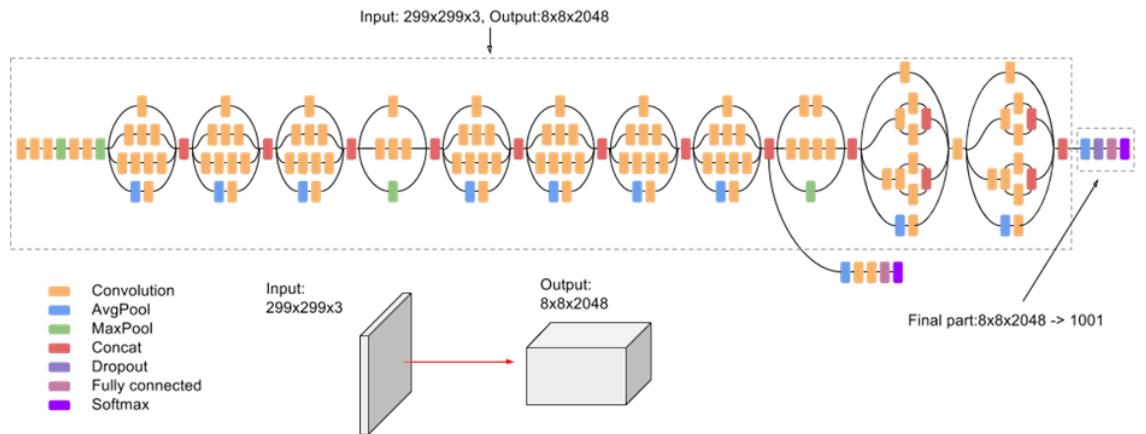
## EfficientNetB0

Nous avons aussi cherché à upgrader les modèles EfficientNet, (nous avions le B0). Cependant, même si la syntaxe utilisée peut le faire penser, les modèles B1,B2 etc sont en fait des versions différentes pour prendre en compte des images de résolutions plus élevées. Nous nous sommes donc désintéressés de ce champ d'amélioration. En effet le EfficientNetB0 est parfaitement adapté à la résolution de nos photos (224\*224)

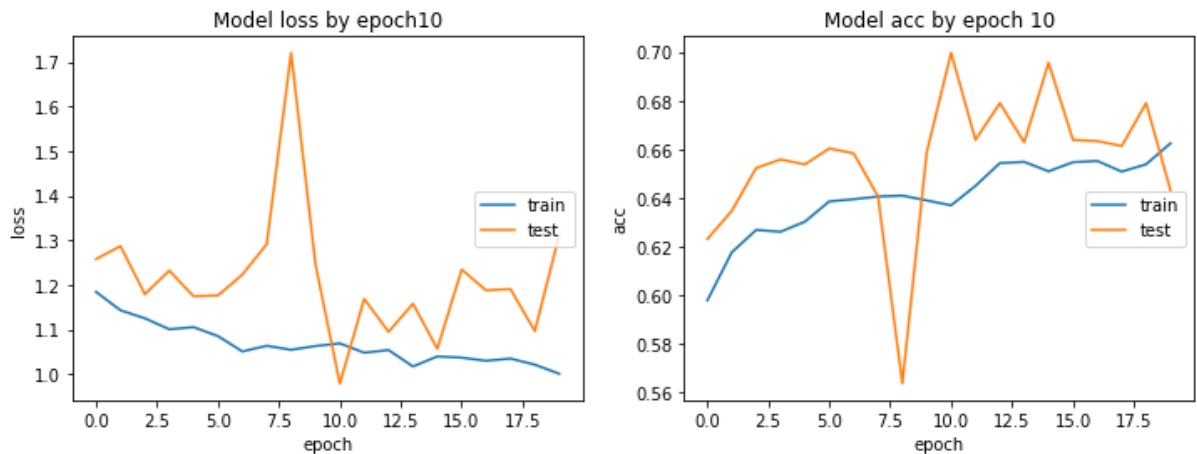
## InceptionV3

Après plusieurs tests, ce modèle ne réagissait pas comme les autres. On ne s'est pas attardé dessus compte tenu des meilleurs résultats sur les autres modèles testés. Il est

aussi le seul modèle de notre liste n'acceptant pas la même dimension d'entrée des images.



### Inception V3, 10 species, 10 x 1000 photos

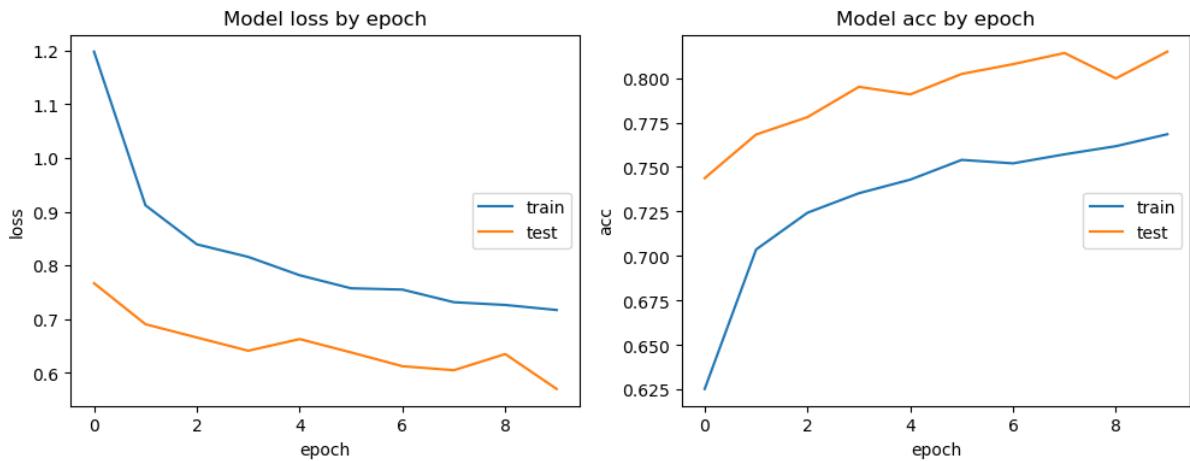


### VGG16 (ou VGG19) et ResNet50

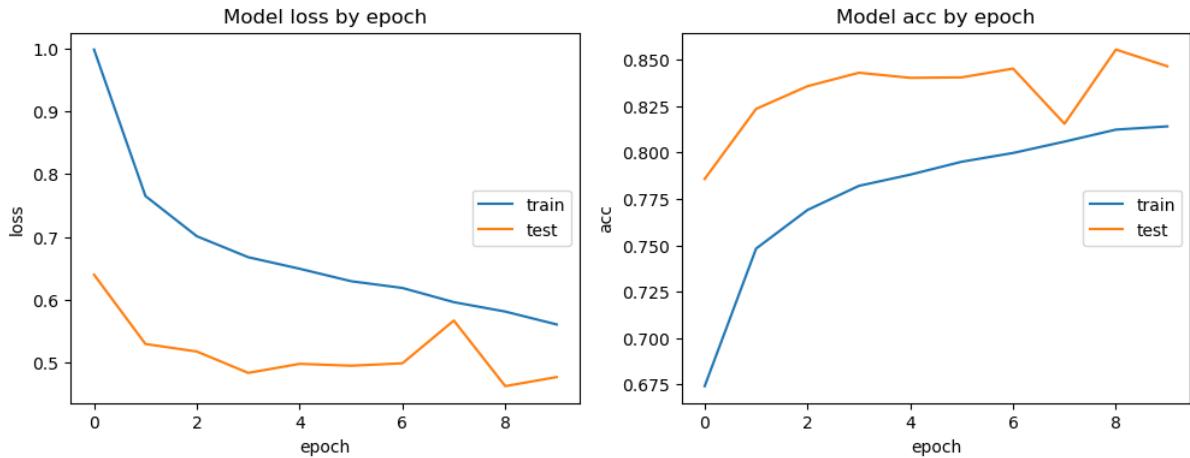
Ces modèles sont placés dans une même partie en raison de leurs résultats assez proches. En effet, tous nos entraînements sur ces modèles se sont révélés assez concluants. Nous sommes passés certes d'une précision de plus de 80% avec 10 classes à une précision au-delà des 60% avec 64 classes mais cela reste tout à fait logique et concluant en termes de probabilités. Nous avons donc utilisé ces modèles sans réelle préférence. VGG19 a été testé également, mais ses performances très similaires à VGG16 dans le cadre de notre entraînement n'a pas permis de conclure à une amélioration significative de nos résultats.

Petit test de comparaison entre nos deux meilleurs modèles

### VGG16 sur 10 family x 2000 images



### ResNet50 sur 10 family x 2000 images



### Couches de classification

Pour créer ces nouvelles couches permettant de classifier nos champignons, nous nous sommes inspirés du cours sur le transfer learning.

- Creation d'un modèle séquentiel
- Première couche destinée à réduire les dimensions d'entrée avec un average pooling
- Création d'une couche de 1024 neurones denses pour débuter la classification
- Dropout pour éliminer 20% des connections
- Une autre couche de 512 neurones denses pour se rapprocher de notre nombre de classes
- Dernière couche dropout pour éliminer à nouveau 20% des connections
- La couche finale de neurone dense où chaque neurone retourne la probabilité de la classe qui lui a été attribuée.

Quelques changements ont été effectués, aucun résultat concluant.

# Optimisations de l'entraînement

## Mise en place des Callbacks

Il n'est pas forcément facile de fixer à priori le nombre d'epoch idéal. Un nombre trop petit peut arrêter trop tôt l'entraînement et un nombre trop élevé peut ne rien apporter et même dégrader les performances. Pour éviter ces situations il est possible de donner à priori un nombre d'epochs élevé mais d'ajouter des fonctions qui vont évaluer le modèle après chaque epoch afin de déterminer s'il faut continuer ou non. C'est `model.fit()` qui appelle lui-même les fonctions déclarée en callback:

- La fonction `ModelCheckpoint()` nous permet de sauvegarder le modèle au meilleur moment de son entraînement.
- La fonction `EarlyStopping()` permet de définir plus d'epochs tout en ayant la garantie que l'entraînement s'arrêtera avant la fin si les performances ne progressent plus.

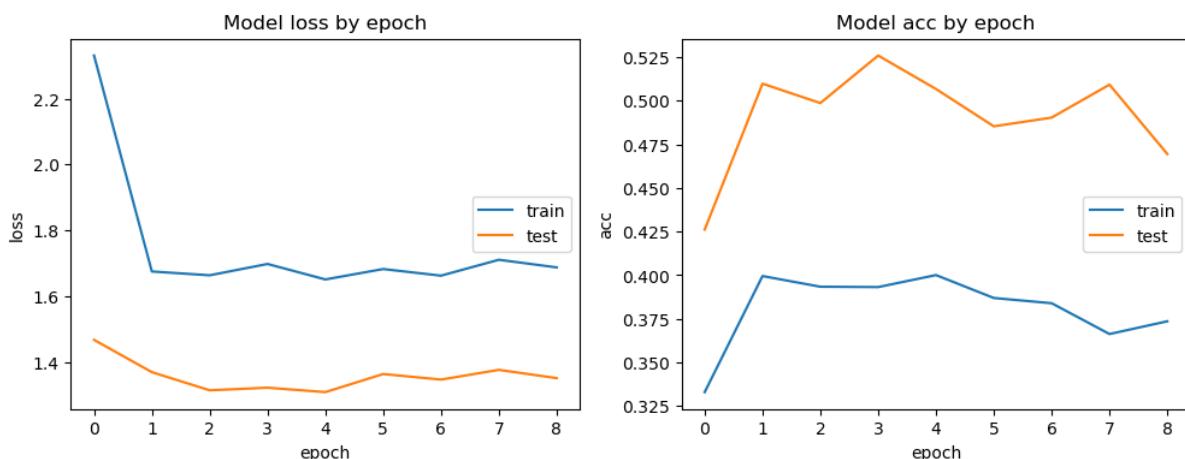
La variable monitorée est, dans les deux cas, la "Validation Accuracy" et nous avons choisi une patience entre 3 et 5 epochs. La patience étant le nombre d'epochs consécutives où le paramètre monitoré n'évolue pas.

## Learning rate

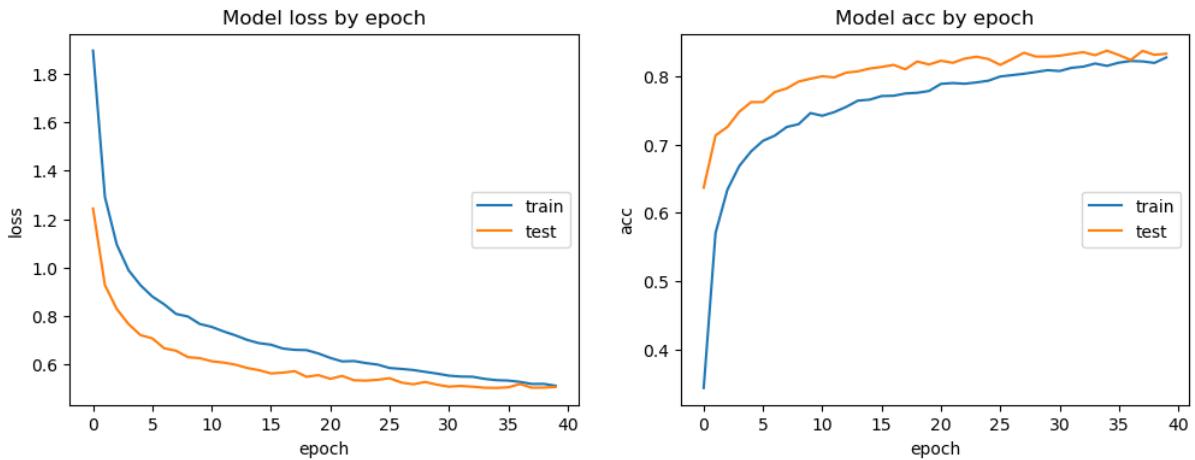
Nous avons testé plusieurs pistes, faire varier le paramètre de LR de l'optimizer Adam et l'utilisation de ReduceLROnPlateau.

Adam optimizer : Le LR par défaut est de 0,001, nous avons pu vérifier qu'un LR plus grand (0,01) dégrade les résultats alors qu'un LR de 0,00001 améliore la qualité de l'entraînement. Par ailleurs l'entraînement s'est arrêté beaucoup plus rapidement avec le LR=0,01

**Resnet50 : Lr = 0.01, 10 family x 2000 photos**



### Resnet50 : Lr = 0.00001, 10 family x 2000 photos



En parallèle nous avons testé avec un LR dynamique pour comparer l'impact sur la vitesse de convergence.

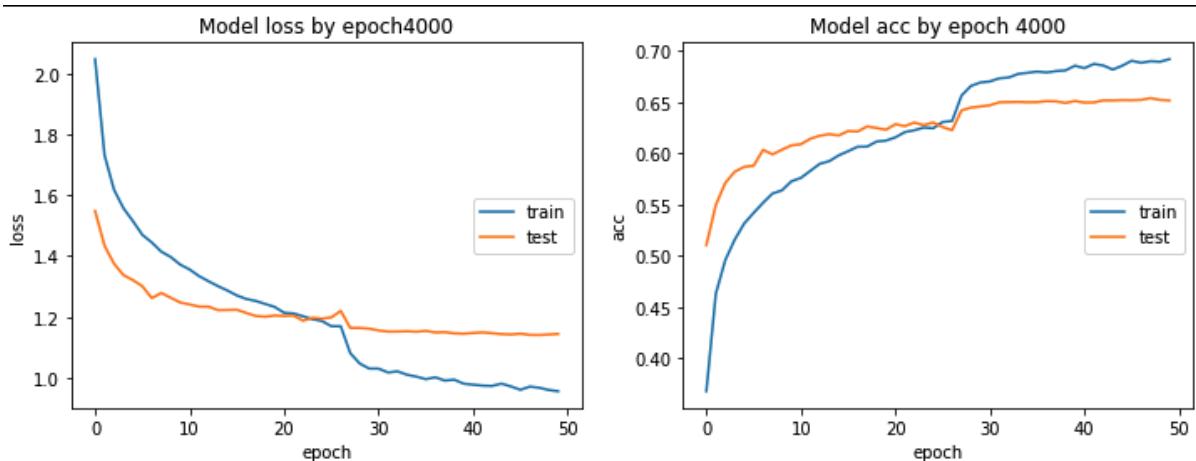
ReduceLROnPlateau :

Il a fallu beaucoup d'essais pour tenter de cerner le comportement de l'algorithme et l'influence de ses paramètres. Un mauvais paramétrage pouvait assez vite mettre le modèle sur un "faux plateau". La perte ne décroissait plus, obtenant une précision "moyenne" comparé aux autres entraînements sur le même modèle.

Le test présenté dans le notebook a été fait ainsi : Faire écho au précédent test effectué avec un learning rate constant de 0,00001 et sur 50 epoch. Cependant, ce test a été effectué avec quelques paramètres différents. Cela est dû au temps d'entraînement et à nos découvertes entre chaque test.

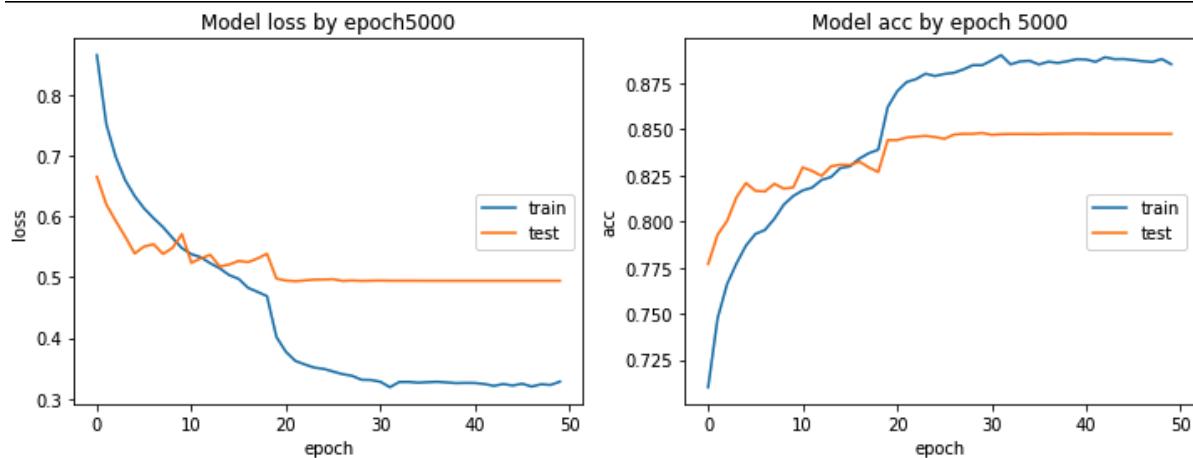
Ce graphique ci-dessous montre donc un entraînement sur 50 epoch, le val\_loss est surveillé par l'algorithme. On remarque que la courbe de test stagne à partir de 15-20 epochs. La courbe n'aurait sûrement pas évolué sans le changement du learning rate. Hors, lorsqu'il est modifié (aux alentours du 27ème epoch), la courbe fait un bond en avant. Cela est d'autant plus marquant pour la courbe de train.

### VGG16, changement dynamique du LR au 27ème epoch



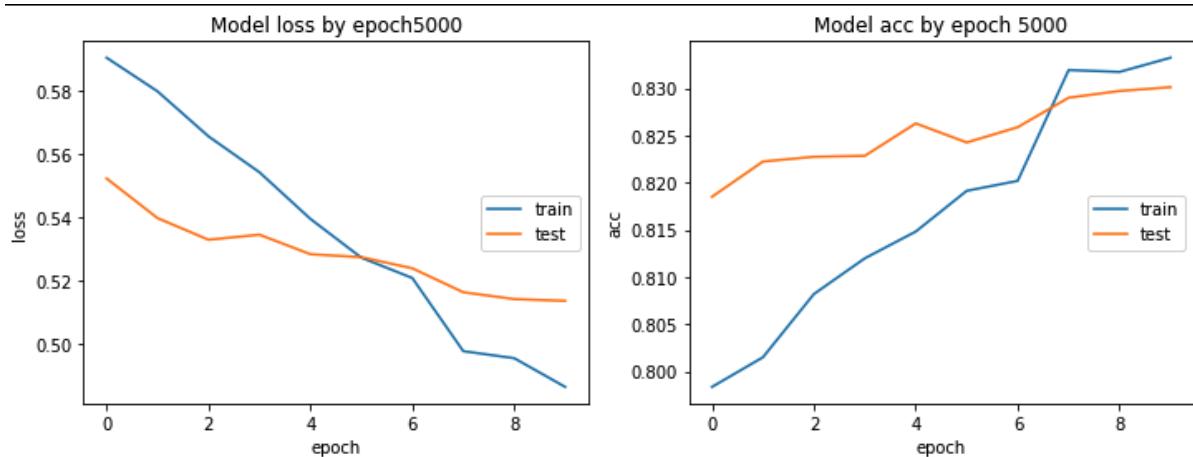
Ce “pic” a été observé lors d’entraînement précédents, ici deux autres exemples.

### Entrainement avec VGG16 et LrOnPlateau



Le graphique ci-dessous était un réentrainement du modèle. Il débutait alors avec une accuracy de 82% et a terminé avec 83%. Cela se révèle donc intéressant pour parfaire un modèle déjà entraîné.

### Réentrainement avec LrOnPlateau:



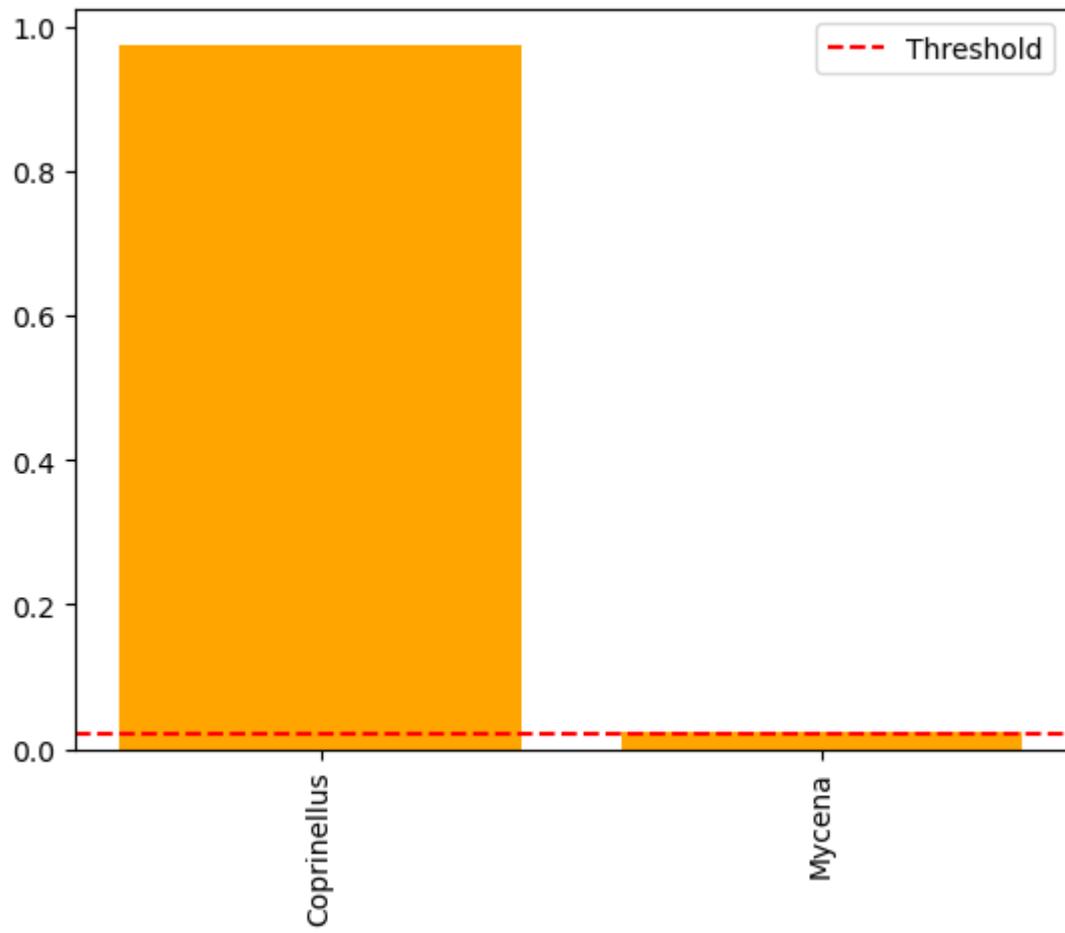
En conclusion, que ce soit pour entraîner ou pour parfaire un entraînement, cette fonction, bien configurée peut apporter quelques points supplémentaires en précision.

## Prédictions

Une fois le modèle entraîné, nous avons un fichier “\*.h5” qui peut être utilisé dans le cadre d’une application qui pourra prédire la classe d’un champignon.

La fonction `model.fit()` nous retourne la liste des probabilités de correspondance pour l’ensemble des classes. Nous avons écrit une fonction `my_decode_prediction()` pour nous retourner une liste ne contenant que les n meilleures probabilités et en supprimant les résultats en dessous d’un certain seuil.

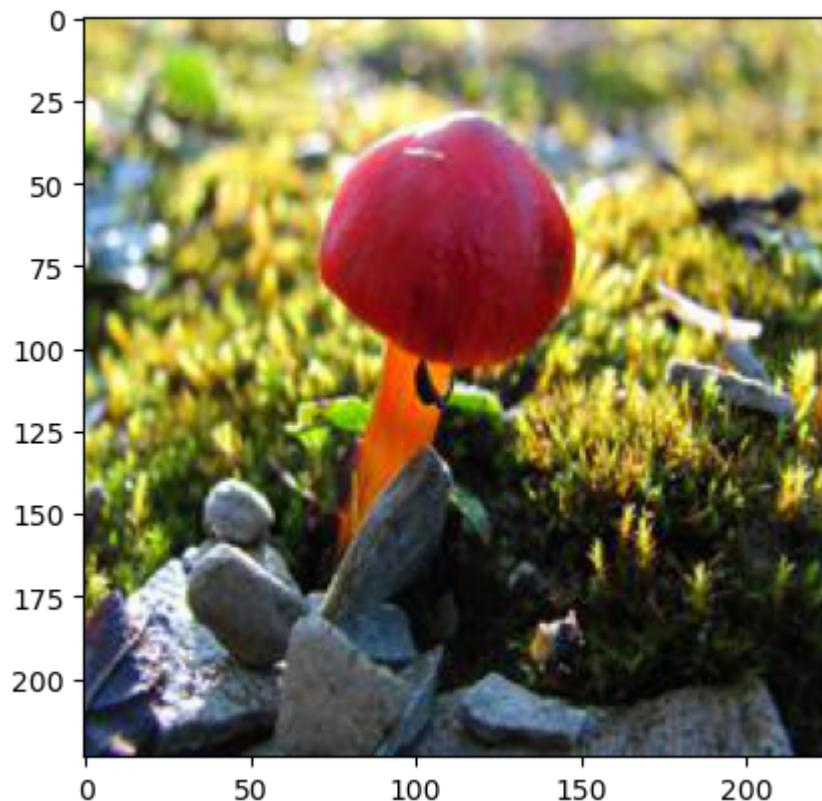
Nous pouvons alors afficher un graphique avec la liste réduite, voici un exemple :



## Interprétabilité

Les résultats que nous avons obtenus, que ce soit sur la valeur de la précision ou sur nos tests aléatoires, sont positifs. On rappelle qu'une prédiction aussi bonne que le hasard serait de 5% pour 20 classes. Nous avons souhaité vérifier les parties des images qui servaient à identifier les classes, notamment pour vérifier si les zones les plus actives correspondaient bien aux parties des images montrant un champignon ou bien un arbre ou des feuilles.

Pour ce faire, nous avons intégré la méthode de Grad Cam pour calculer une heatmap qui permet de localiser les parties d'une photo qui ont servi à la classification. La palette utilisée est un dégradé allant du bleu pour les zones les moins pertinentes pour notre modèle au rouge pour les zones réellement déterminantes pour la classification.



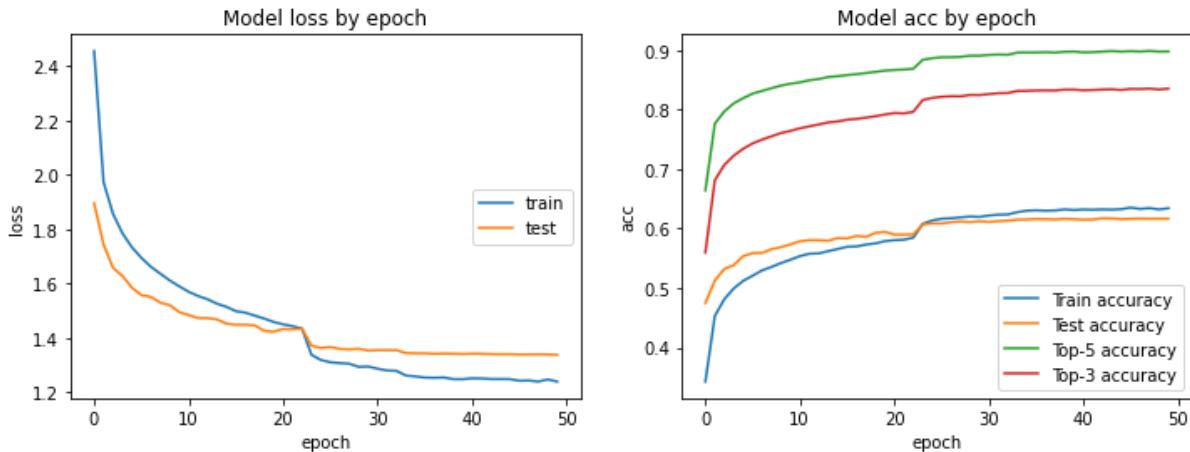
Cette fonction nous permet ainsi de vérifier la robustesse de notre modèle. Le résultat est visible dans le notebook de prédiction.

## Utilisation d'un maximum de données

Pour terminer ce projet, nous avons lancé des calculs en augmentant le nombre de classes à classifier ainsi que le nombre d'images par classe dans les limites des images que nous avions, tout en conservant un nombre d'images équilibré par classe :

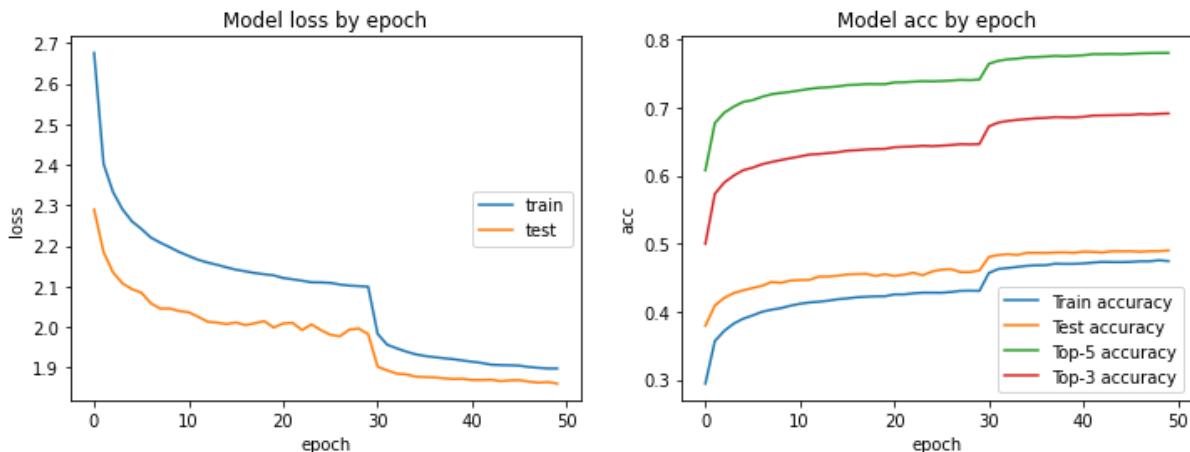
- VGG16, 64 species, 4000 photos, 300s par Epoch sur instance p3 d'AWS, avec l'alternative au ImageDataGenerator

### VGG16 64 species x 4000 photos sans IDG



- VGG16, 64 genus, 8000 photos, 650s par Epoch sur instance p3 d'AWS, avec l'alternative au ImageDataGenerator

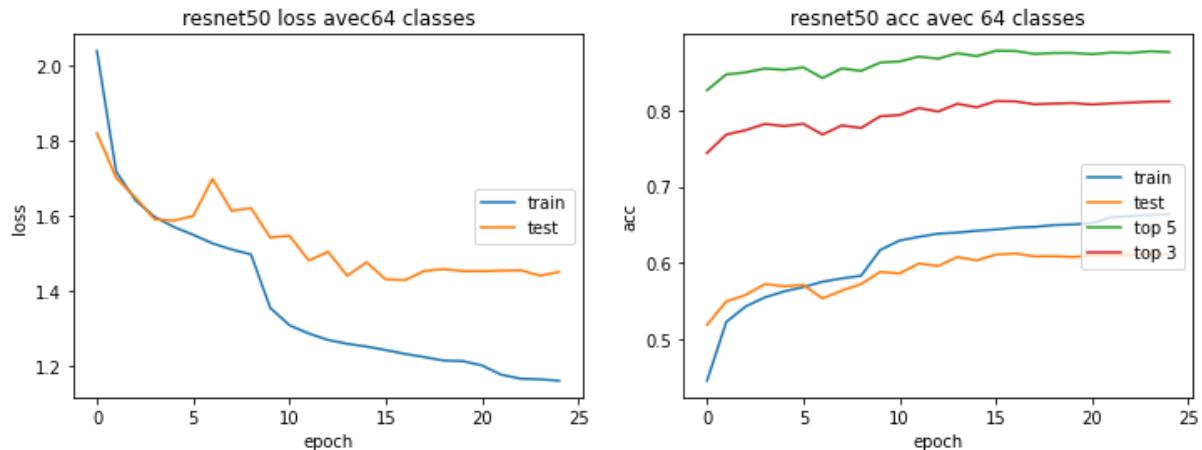
### VGG16 64 genus x 8000 photos sans IDG



On note la meilleure précision obtenue quand on descend dans la hiérarchie de la taxonomie.

- Resnet50 64 species, 4000 photos avec imagedatagenerator  
 $\text{val\_acc: } 0.6109$  -  $\text{val\_top\_5: } 0.8766$  -  $\text{val\_top\_3: } 0.8118$

### Resnet50 64 species x 4000 avec IDG



## Conclusions

Ce projet nous a permis d'explorer de nombreuses pistes autour des modèles pré-entraînés. Voici les points les plus remarquables :

- Nous pensons avoir atteint un très bon niveau de nettoyage des images, il reste peu d'images susceptibles de dégrader l'apprentissage
- Augmenter le nombre d'opérations d'augmentation dans le générateur d'images permet d'améliorer l'apprentissage jusqu'à une valeur plus haute mais plus lentement.
- Plus on utilise une variable cible qui descend dans la taxonomie, plus les résultats s'améliorent (Précision sur les genus < Précision sur les species). Néanmoins dans l'objectif d'une application, l'utilisation d'un modèle entraîné (même moins précis) sur les "genus" nous permet une couverture plus grande de la taxonomie.
- Plus on ajoute de classes à identifier, plus la précision absolue baisse tout en restant très satisfaisante. On rappelle que sur 20 classes la probabilité de prédiction aléatoire est de 5% et qu'avec 64 classes, elle est de 1,5%
- Les modèles VGG16, ResNet50 et EfficientNetB0 sont relativement proches et offrent des résultats satisfaisants avec une précision autour de 50% sur les genus et 60% sur les species
- Relancer l'apprentissage une seconde fois en dégelant certaines couches du modèle peut apporter une légère amélioration de l'ordre de quelques pourcents. Lors de notre test avec ResNet50 (20 classes), nous avons gagné 12 points sur la "val\_acc" mais en dégelant toutes les couches.
- Nous déclinons toute responsabilité en cas de décès suite à l'ingestion d'un champignons que nous aurions malencontreusement classé à tort comme comestible ;-)

# Pour aller plus loin...

## Entraînement distribué

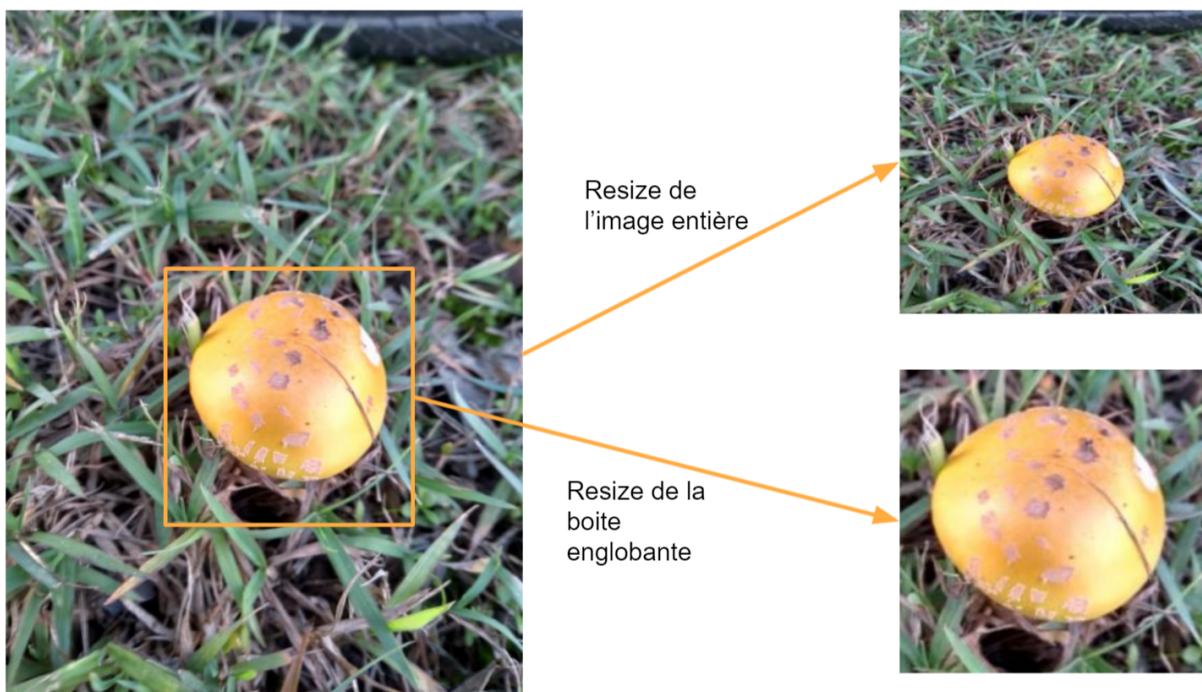
L'utilisation de GPU-Z a fait apparaître que l'un des ordinateurs avait 2 GPUs compatibles DirectML. Cela a donné l'idée de répartir la charge sur ces 2 GPUs en utilisant "tf.distribute.MirroredStrategy()". Malheureusement le code n'était pas compatible avec DirectML et nous avons dû abandonner cette piste.

Après avoir commencé à lancer des calculs sur des instances AWS déportées, nous pensons que ce serait une bonne piste pour significativement accélérer l'entraînement d'un modèle en répartissant les calculs sur des instances multi-GPU mais aussi pour mieux utiliser les vCPU de ces machines.

## BoundingBox

Nous avons décrit précédemment que nous faisions un redimensionnement des images en 224x224 en même temps que le téléchargement. Il s'agit d'un redimensionnement complet de l'image et nous aurions aimé pouvoir tester une étape intermédiaire en favorisant le redimensionnement au niveau de la boîte englobante du champignon. Le champignon exploiterait mieux les 224x224 pixels.

Toutefois, nous ne savons pas si la détermination de cette boîte englobante aurait bien fonctionné et si elle aurait eu un impact significatif.



## Optimisation de la data augmentation

Nous avons aperçu que les limites des fonctions telles que tf.rot90 pouvaient être contournées en ajoutant des layers en entrée de notre modèle. Cette technique semble permettre beaucoup plus de choix dans les paramètres de la data augmentation. Même si nous n'avons pas eu le temps de l'implémenter, nous avons compris qu'il faudrait ajouter une condition de prise en compte ou non des layers selon que le dataset vient de train ou de test. C'est une piste d'amélioration qui aurait pu potentiellement amener les résultats sans

ImageDataGenerator au même niveau de précision qu'avec. En espérant que cela n'aurait pas ralenti la vitesse d'entraînement.

## Couches de Classification

Pour aller plus loin dans notre modèle, nous pourrions modifier nos couches de classification. La meilleure approche serait sans doute d'aller se renseigner sur la classification utilisée lors des entraînements qui ont déjà montré leurs preuves sur les modèles pré entraînés que nous avons utilisés. (Sur Kaggle ou autre)

## Neurone de classification

Après la dernière Master Class, nous avons vu comment ajouter un dernier neurone de classification afin d'estimer la probabilité que le champignon fasse partie ou non des "classes" choisies. Cela aurait pu être une belle optimisation pour notre démo streamlit qui, elle, ne saura pas faire la distinction et estimera de la même manière un champignon faisant partie des 64 "classes" qu'un champignon n'y appartenant pas. Cependant, afin d'avoir des données équilibrées il aurait fallu doubler le nombre de photos, avec une partie appartenant aux 64 "classes" choisies et une autre ne faisant pas partie de celles-ci. Un obstacle à l'apprentissage difficilement conciliable avec nos ressources matérielles. (Un entraînement sur 64\*8000 photos nous prenait environ 4 jours sur une GTX 1070 avec ImageDataGenerator, 2 jours sans)

## Annexe 1 : Machines à disposition

Un PC fixe avec une carte NVidia GTX 1070, fonctionne très bien pour Tensorflow 2

Un PC portable avec une NVidia MX250 mais pour usage bureautique (moindres performances)

Un PC fixe avec une AMD 5700 xt :

- Essais avec PlaidML (rencontre de conflit avec d'autres packages)
- ROCM qui sont les pilotes propriétaires d'AMD compatibles avec Tensorflow, uniquement sous linux, mais non disponible pour cette série de carte graphique.
- Tensorflow-DirectML : conçu pour optimiser les performances de Tensorflow sous un maximum de machines. Moins efficace que les CUDA de NVidia, il permet tout de même de faire mieux que Tensorflow sur CPU. Problème : version bien antérieure à tensorflow 2.6.

Différentes Instances AWS, voir dans l'annexe 3

## Annexe 2 : Liste des fichiers du code

Le tableau suivant liste les fichiers livrés, ce qu'ils font et leur localisation

Fichier	Description
readme.txt GitHub: Documents Finaux\	Tuto des étapes pour faire fonctionner les notebooks et télécharger les données
combined.zip <a href="https://champyseed.s3.amazonaws.com/combined.zip">https://champyseed.s3.amazonaws.com/combined.zip</a>	Contient l'ensemble des images téléchargées avec l'arborescence qui reprend la taxonomie
Champyseed.csv <a href="https://champyseed.s3.amazonaws.com/Champyseed.csv">https://champyseed.s3.amazonaws.com/Champyseed.csv</a>	Dataframe Final
ConversionRGB.ipynb imagesnotRGB.txt GitHub: \Documents Finaux\Code\ConversionRGB	<b>Patch à appliquer sur le zip</b> Fichier à utiliser pour convertir les 600 images (niveau de gris) au format RGB
liste_comestible.xlsx liste_edible.py GitHub: Documents Finaux\Code\liste edible	création de la liste de comestibilité
tirage_image_notfungi.py GitHub: Documents Finaux\Code\tirage fungi or not	afficher des images au hasard sur le modèle des enveloppes
GitHub: \Documents Finaux\Code\Entrainement des modèles	Entraînement (Avec ImageDataGenerator et Sans IDG) VGG16, Resnet50, EfficientNetB0
Prediction_interpretability_multi_models.ipynb GitHub: \Documents Finaux\Code\Prediction Multi-Models	Prédiction et Interprétabilité ResNet50, EfficientNetB0, VGG16, avec ou sans ImageDataGenerator
<b>Fichiers H5 sauvegardés (Obligatoires pour Prediction_interpretability_multi_models.ipynb)</b>	
vgg16_64genus_noIDG.h5 <a href="https://champyseed.s3.amazonaws.com/vgg16_64genus_noIDG.h5">https://champyseed.s3.amazonaws.com/vgg16_64genus_noIDG.h5</a>	Fichier h5 sauvegardé après l'entraînement d'un modèle VGG16 sur 64 genus avec l'alternative à ImageDataGenerator
vgg16-64species-IDG.h5 <a href="https://champyseed.s3.amazonaws.com/vgg16-64species-IDG.h5">https://champyseed.s3.amazonaws.com/vgg16-64species-IDG.h5</a>	h5 sauvegardé après l'entraînement d'un modèle VGG16 sur 64 species avec ImageDataGenerator
vgg16_64species_noIDG.h5 <a href="https://champyseed.s3.amazonaws.com/VGG16_64species_noIDG.h5">https://champyseed.s3.amazonaws.com/VGG16_64species_noIDG.h5</a>	Fichier h5 sauvegardé après l'entraînement d'un modèle VGG16 sur 64 species avec l'alternative à ImageDataGenerator

resnet5064speciesIDG.h5 <a href="https://champyseed.s3.amazonaws.com/resnet5064speciesIDG.h5">https://champyseed.s3.amazonaws.com/resnet5064speciesIDG.h5</a>	h5 sauvegardé après l'entraînement d'un modèle ResNet50 sur 64 species avec ImageDataGenerator
res50_64genusIDG.h5 <a href="https://champyseed.s3.amazonaws.com/res50_64genusIDG.h5">https://champyseed.s3.amazonaws.com/res50_64genusIDG.h5</a>	h5 sauvegardé après l'entraînement d'un modèle ResNet50 sur 64 genus avec ImageDataGenerator
efficientnet_64genus_IDG.h5 <a href="https://champyseed.s3.amazonaws.com/efficientnet_64genus_IDG.h5">https://champyseed.s3.amazonaws.com/efficientnet_64genus_IDG.h5</a>	h5 sauvegardé après l'entraînement d'un modèle EfficientNetB0 sur 64 genus avec ImageDataGenerator
<b>Premières étapes de récupération du tableau sur GBIF et création du dataset</b>	
export GBIF.zip <a href="https://champyseed.s3.amazonaws.com/export+GBIF.zip">https://champyseed.s3.amazonaws.com/export+GBIF.zip</a>	Fichier exporté du site GBIF.org
traitement darwin core.ipynb GitHub: Documents Finaux\Code\tableau de darwin a csv et nettoyage	dépendance : <b>basegbif.zip</b> Traitement du fichier en une base csv
nettoyage du tableau.ipynb GitHub: Documents Finaux\Code\tableau de darwin a csv et nettoyage	dépendance : Dataframe issu du traitement darwin core sélection colonnes, conversions, trie création de → tableau_propre.csv
tableau_propre.csv <a href="https://champyseed.s3.amazonaws.com/tableau_propre.csv">https://champyseed.s3.amazonaws.com/tableau_propre.csv</a>	dépendance : Dataframe issu du traitement darwin core
shape et resize image tableau complet.py GitHub: Documents Finaux\Code\tableau de darwin a csv et nettoyage	script permettant le nettoyage et l'adaptation des images au modèle
champyseed path.ipynb GitHub: Documents Finaux\Code\tableau de darwin a csv et nettoyage	création du chemin relatif de chaque image
graphiques famille.ipynb Github: Documents Finaux\Code\graphique nombre de photos par famille	dépendance : <b>Comestible20210903.csv</b> affichage de nombre de photo par famille
<b>Téléchargement des images</b>	
Téléchargement des images Github: ChampySeed/Documents Finaux/Code/Téléchargement des images	.py pour télécharger les images depuis allpathidclear
allpathidclear.csv <a href="https://champyseed.s3.amazonaws.com/allpathidclear.csv">https://champyseed.s3.amazonaws.com/allpathidclear.csv</a>	Utilisé pour le téléchargement des images
GitHub: ChampySeed/Documents Finaux/Code/fabrication de	Afin de fabriquer les dossiers contenant les photos

l'arborescence/ fabrication de l'arborescence	
--	--

## Annexe 3 : Entraînement du modèle sur une instance déportée AWS

Nous avons rapidement pu constater que les temps de calculs pour l'entraînement de nos modèles allaient devenir conséquents en ajoutant plus de classes et de photos de semaine en semaine.

En complément des différentes optimisations entreprises pour réduire les calculs, nous avons également souhaité étudier la possibilité d'exécuter nos notebooks sur des ordinateurs plus puissants, notamment au niveau du GPU.

Nous avons temporisé par crainte que ce soit trop long à mettre en place, notamment sur la gestion des droits IAM, mais à postériori nous aurions dû nous lancer plus tôt car cela a été plus simple qu'anticipé.

Nous sommes maintenant en mesure de lancer un entraînement sur une nouvelle machine en moins d'1 heure !

Service utilisés :	Usage :
 <b>Amazon EC2</b>	<p>Service de computing, utilisation des instances suivantes:</p> <ul style="list-style-type: none"> <li>- c5ad.xlarge (sans GPU)</li> <li>- g4dn.xlarge (avec GPU)</li> <li>- p3.2xlarge (avec GPU)</li> </ul> <p>Avec AMI :</p> <ul style="list-style-type: none"> <li>- AWS Deep Learning v50</li> </ul>
 <b>Amazon S3</b>	<p>Service de stockage utilisé pour sauvegarder le zip contenant les images des champignons afin de pouvoir les recopier rapidement sur une nouvelle machine.</p>

### Étapes réalisées :

- Création d'un compte AWS
- Vérification des images disponibles permettant de lancer rapidement une machine avec Tensorflow pré-installées (AWS Deep Learning AMI)
- Choisir une instance:

- 1er test sur une instance c5ad.2xlarge (sans GPU mais ne nécessitant aucune demande au support AWS)
- g4dn.xlarge (avec carte NVidia Tesla T4, une fois que le support AWS ait accepté l'utilisation d'instances G avec 4 vCPU)
- A venir après approbation de la demande : p3.2xlarge (avec NVidia Tesla V100)
- Monter (mount) un disque car l'espace initial était trop petit
- Importer les images de champignons sur l'instance
  - Les 17 Go ont été découpés en 6 zips puis téléchargés par des commandes CURL depuis le serveur ftp que nous avions créé
  - Reconstruction du zip complet
  - Sauvegarde sur un bucket S3 pour ne pas avoir à refaire cette étape
  - Pour les nouvelles instances on télécharge directement le zip de 17Go depuis en bucket S3 en 10 min avec une simple commande CURL.
- Utilisation de WinSCP pour copier les Notebooks sur l'instance en SSH
- Nous pouvons alors lancer Jupyter notebook dans notre navigateur pour lancer un entraînement.

## Performances

Les GPU disponibles comme les Tesla M60 ou K80 ne sont pas très puissants, nous avions une puissance équivalente avec nos propres machines. Il faut choisir des machines avec par exemple une Tesla T4 et surtout Tesla V100 pour noter des gains de temps d'entraînement supplémentaires.

En chiffres :

- 40 min par Epoch avec un modèle ResNet50 sur 64 species et 4000 photos par classe en utilisant ImageDataGenerator
- 5 min par Epoch avec un modèle VGG16 sur 64 species et 4000 photos par classe **SANS** utiliser ImageDataGenerator
- 10 min par Epoch avec un modèle VGG16 sur 64 genus et 8000 photos par classe **SANS** utiliser ImageDataGenerator

A titre de comparaison, sur une GTX 1070, 1h15 par Epoch avec un modèle ResNet50 sur 64 genus et 8000 photos par classe en utilisant ImageDataGenerator

Pour aller plus loin, il serait intéressant de faire des tests de "Distributed Training" sur plusieurs GPU.

Ce que Datascientest pourrait proposer avec son partenariat AWS :

- Avoir un contact direct pour autoriser plus rapidement l'utilisation de machines avec GPU (instances G et P)
- Avoir une attribution de crédits, par exemple via le programme "AWS Activate"

## Annexe 4 : Diagramme de Gantt

Fichier pdf disponible sur le GitHub