Rapport Etape 4 Itération 3

Pour cette étape d'optimisation, nous avons travaillé sur 2 niveaux :

- Optimisation continue de la qualité du dataset
- Optimisation de l'entraînement

Qualité du dataset

Nous continuons de trouver des images qui ne présentent pas d'intérêt pour notre projet et peuvent même impacter l'entraînement du modèle. 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 est très élevée, nous "blacklistons" les contenus de ce domaine. Cette technique a permis de réduire d'un quart les répertoires (des genus ne comportant que 2 photos par exemple).

Le code écrit nous aide à évaluer facilement la qualité en ouvrant aléatoirement des images téléchargées depuis le domaine suspecté :

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"

Optimisation du téléchargement

Plusieurs threads sont exécutés en parallèle car un seul notebook n'effectue que des tâches en série, et la moindre contrainte dans le code bloquait tout le reste.
Le téléchargement prenait du temps, même pour une connexion fibrée. Avec un seul notebook lancé on pouvait clairement détecter que c'était la réponse des serveurs distants qui était le facteur limitant du script de téléchargement. Cela est devenu clair en regardant précisément l'activité de la connexion internet dans le gestionnaire des tâches. On voyait clairement partir la requête et l'attente de la réponse. Nous avons alors eu l'idée de relancer plusieurs fois le même code pour télécharger différentes familles. Nous avons ainsi pu multiplier le téléchargement par 10. Le téléchargement n'est du coup aujourd'hui plus un souci puisque nous avons la capacité de télécharger plus de 100 000 photos en une journée.

Optimisation de l'entraînement

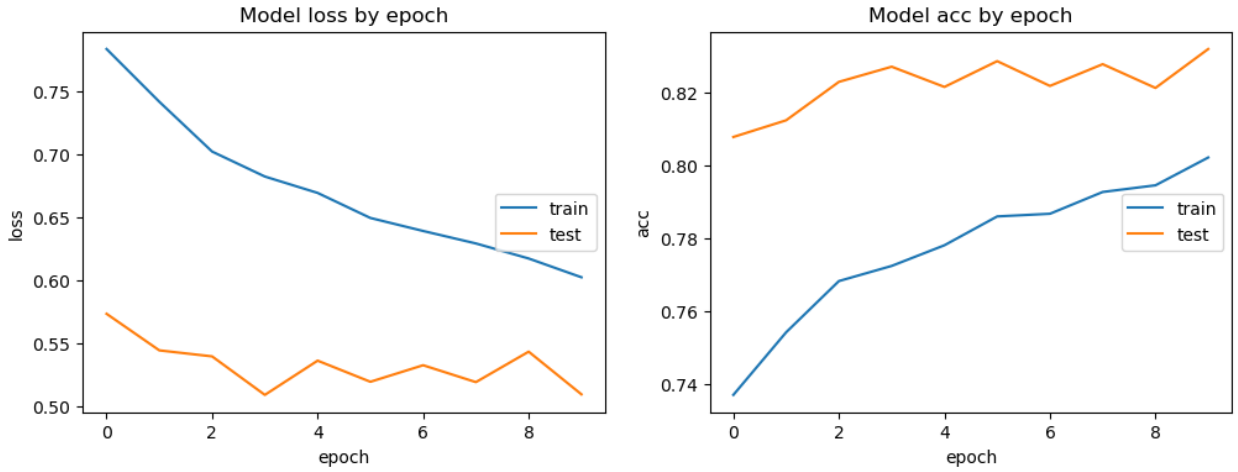
Là encore il est possible d'agir à plusieurs niveaux. Nous avons identifié 3 goulets d'étranglement:

- Quantité d'images
- ImageDataGenerator
- Le choix des hyperparamètres du modèle
- L'architecture des couches en sortie du modèle pré-entraîné

Quantité d'images nécessaires à l'entraînement

Les premières questions qui semblent intéressantes sont combien d'images et combien de classes pouvons nous entraîner?
Pour la première question, nous avons testé deux modèles parfaitement équivalents avec en seul changement de paramètre le nombre d'images utilisable par le modèle pour une classe (2000 puis 4000). Nos résultats semblent indiquer que cela n'implique pas de résultats très différents. Nous pouvons noter qu'il semble assez clair que le modèle avec 4000 images possède un potentiel plus conséquent que celui avec 2000. En effet nous semblons apercevoir un plateau avec celui à 2000 photos en termes de résultats, alors qu'il est probable que celui avec 4000 possède encore de la marge d'apprentissage.
Pour la seconde question, nous sommes encore en cours d'apprentissage.

Nous pouvons en conclure que pour nos essais il n'est pas d'une grande utilité d'utiliser un grand nombre de photos mais que pour l'entraînement final cela pourra nous être très utile afin d'obtenir de bonnes prédictions.
Exemple : Resnet50 4000 images sur 10 classes 10 epochs



Exemple : Resnet50 2000 images sur 10 classes 10 epochs

ImageDataGenerator

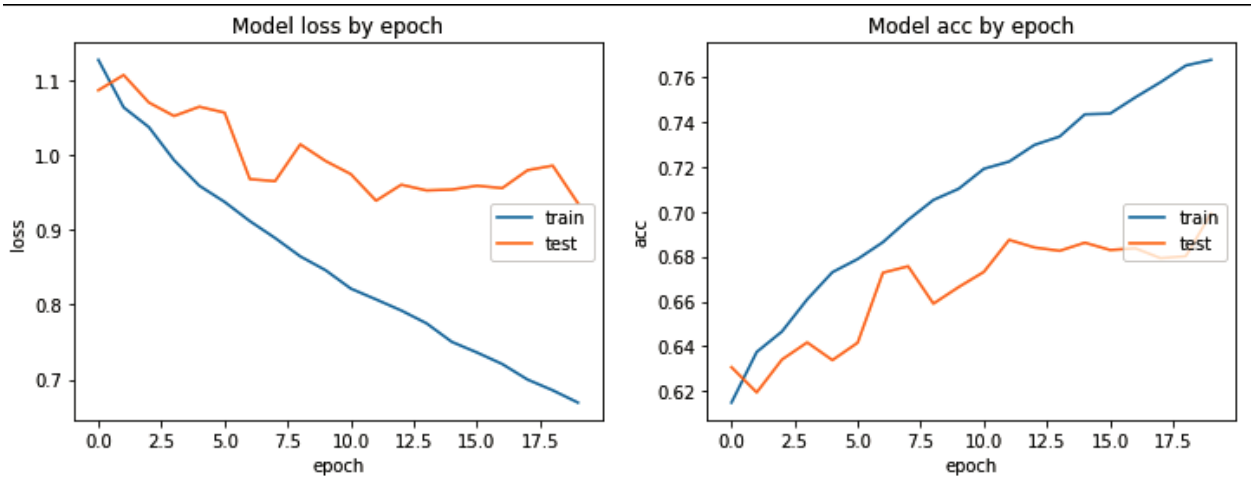
Batch_size

La variable batch_size va impacter les performances à deux niveaux. D'une part, nous cherchons à utiliser au mieux la VRAM, ce qui suppose d'augmenter la valeur de ce paramètre. Mais l'effet négatif est qu'en augmentant la taille du batch, on va aussi augmenter le temps nécessaire à ImageDataGenerator pour préparer nos images.

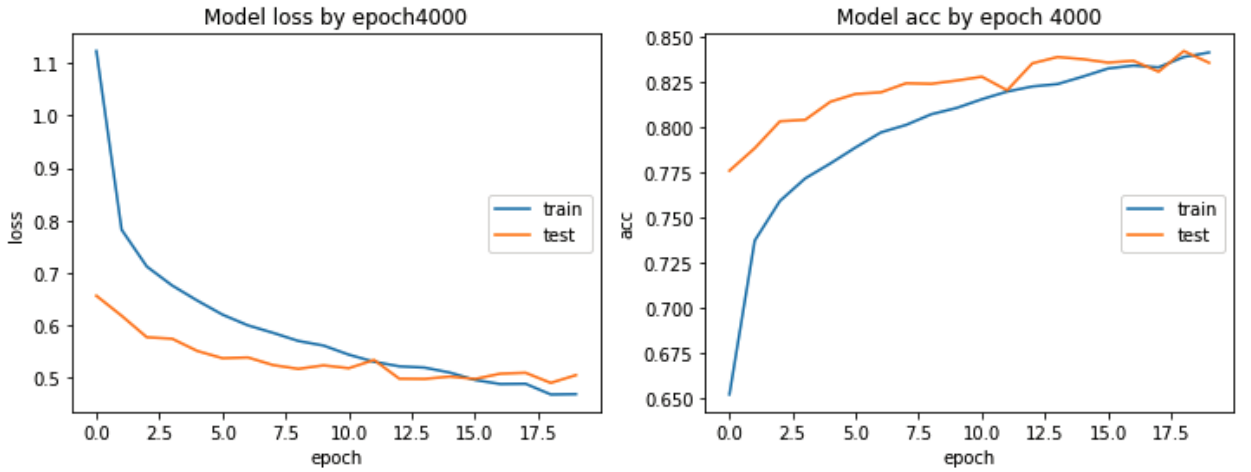
Le graphique montre la charge du processeur au cours du temps. On remarque qu'elle n'est pas constamment à 100% avec ImageDataGenerator

Les résultats obtenus en faisant varier le contenu d'ImageDataGenerator :

Exemple avec 40 000 images, ImageDataGenerator **avec uniquement** le preprocess_input



Exemple avec 40 000 images, ImageDataGenerator avec **preprocess_input ET rotation 180**



Sans la rotation, l'accuracy du test stagne assez rapidement. Elle ne dépasse pas les 0.70 .
Avec la rotation, le modèle est plus robuste et val_acc dépasse les 0.80, soit un gain non négligeable.

Transformations effectuées par ImageDataGenerator

Nous sommes partis sur de multiples transformations (luminosité, zoom, etc). Pour des raisons d'optimisation, nous avons tenté de limiter ces transformations.
L'idée de devoir appliquer une rotation aide beaucoup l'algorithme à progresser sur peu d'images.
Cependant, cette fonction multiplie par deux le temps d'entraînement. C'est pour cela que d'autres solutions sont envisagées.

Alternatives à ImageDataGenerator

En regardant la répartition de la charge entre les accès disques et le GPU, il s'est avéré pertinent de vérifier si nous pouvions accélérer cette partie. Après avoir dans un premier temps supprimé des transformations (cf. paragraphe ci-dessus) pour vérifier l'impact sur les

performances, nous avons exploré l'optimisation du code, soit grâce aux fonctions optimisées de TensorFlow, soit en changeant de générateur d'image.

Code optimisé tf

Nous cherchons à optimiser notre code pour avoir un meilleur rendement (temps de Calcul)/accuracy. Néanmoins pour le moment nous n'avons pas encore réussi à nous passer de ImageDataGenerator. En effet nous comprenons encore assez mal les différentes entrées et sorties des différents outils que l'on trouve. Nous n'avons pas abandonné et c'est toujours un axe que nous investissons.

GapML CV

Il s'agit d'un package réputé 2 fois plus rapide qu'ImageDataGenerator, il offre également l'avantage de pouvoir prendre en entrée des fichiers sur le HDD ou bien des url. Le GitHub n'a néanmoins pas bougé depuis 2018, ce qui est très long dans notre domaine.
Ce sujet n'a pas encore abouti.

Optimisation de l'entraînement

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.

Google Collab

Google Collab a également été envisagé. Cependant, la manière la plus pertinente était de le connecter à google drive. La lenteur d'upload des images (restriction volontaire), la difficulté de gestion des fichiers ainsi que la lenteur de chargement des images sous collab ont clairement été un frein pour ce projet. D'ailleurs, le règlement de collab favorise les petites utilisations. Cela se voyait au chargement d'une image (pour tester le code), qui pouvait aller d'une seconde à une minute selon l'allocation des ressources.

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'epoch 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.

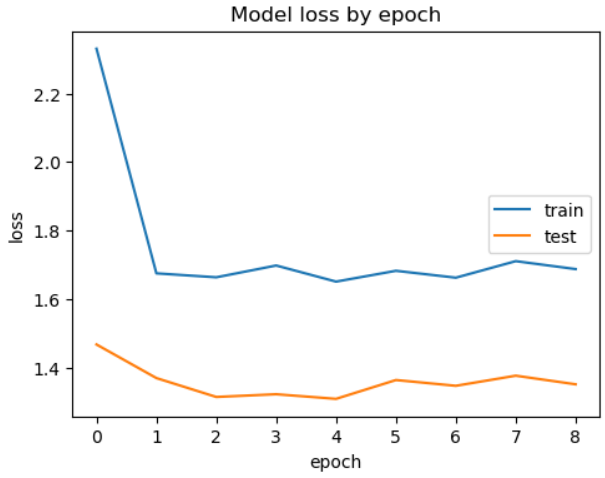
La variable monitorée est dans les deux cas la "Validation Accuracy" et nous avons choisi une patience de 5 epochs.

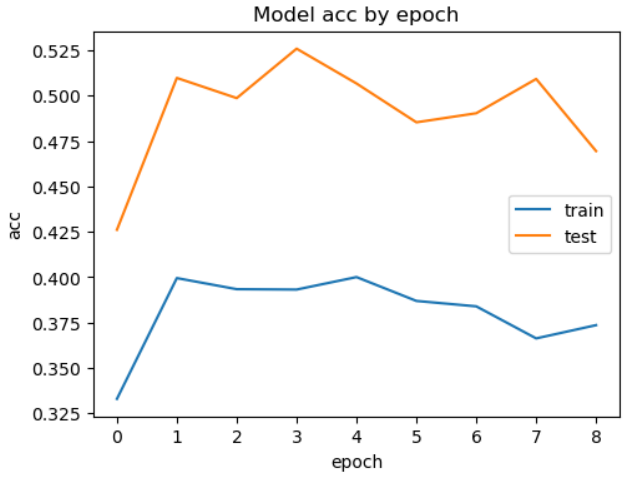
Learning rate

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

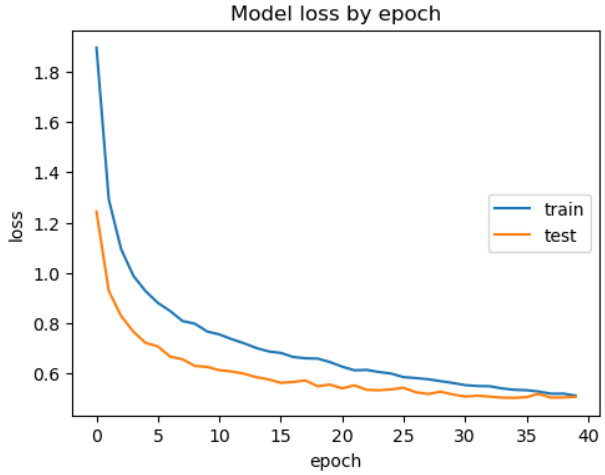
Adam optimizer : Le LR par défaut est de 0,001, nous avons pu vérifier qu'un LR plus grand (0,01) dégrade les résultats alors qu'un LR de 0,0001 améliore la qualité de l'entraînement. Par ailleurs l'entraînement s'est arrêté beaucoup plus rapidement avec le LR=0,01

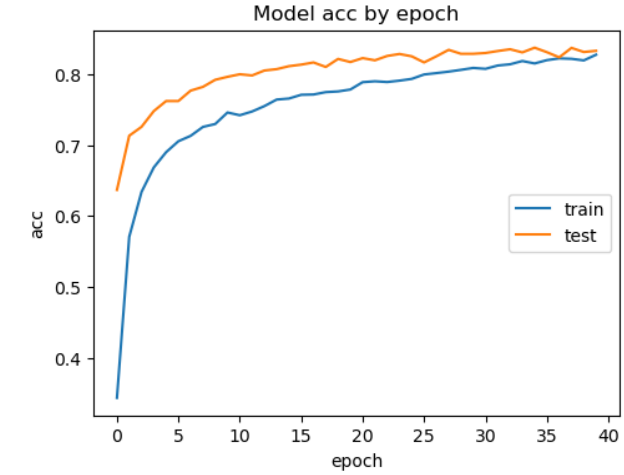
Exemple avec LR = 0,01 sur un ResNet50:





Et avec LR = 0,00001:





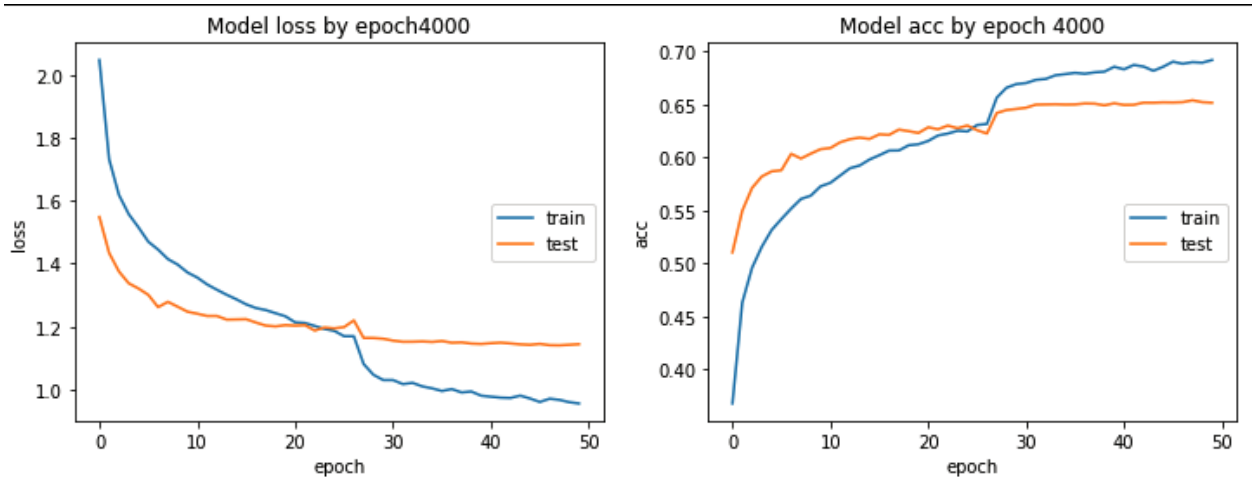
En parallèle nous avons testé avec un LR dynamique pour comparer l'impact sur la vitesse de convergence.

ReduceLRonPlateau :

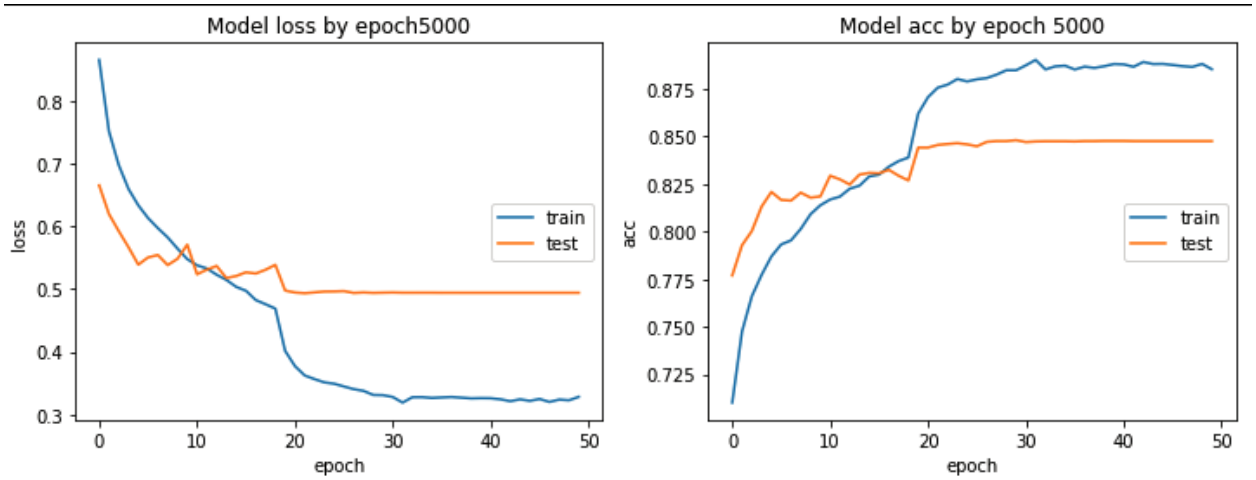
Il a fallu beaucoup d'essais pour tenter de cerner le comportement de l'algorithme et l'influence de ses paramètres. Un mauvais paramétrage pouvait assez vite mettre le modèle sur un "faux plateau".

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.

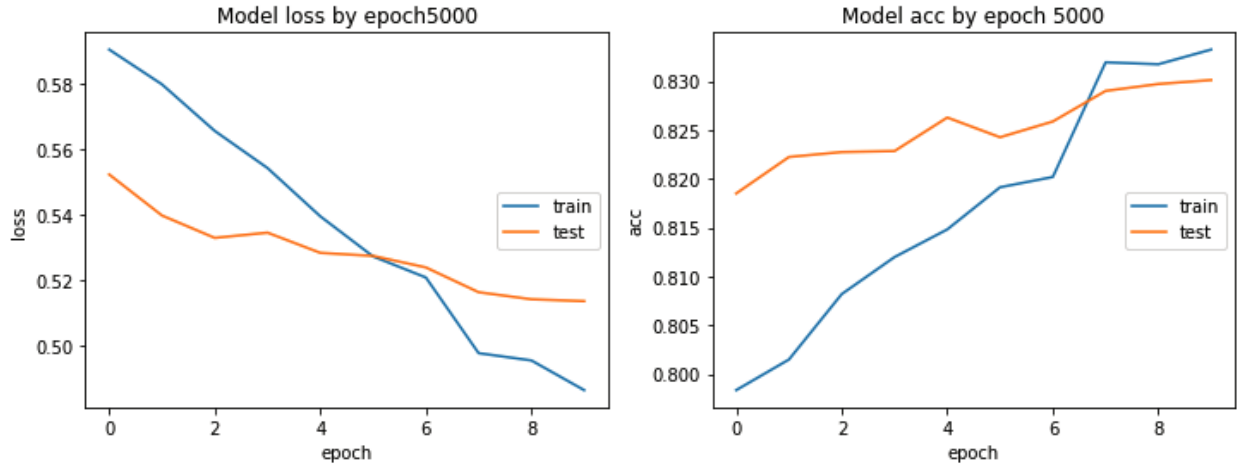
Ce graphique ci-dessous montre donc un entraînement sur 50 epoch, le val_loss est surveillé par l'algorithme. On remarque que la courbe de test stagne à partir de 15-20 epoch. La courbe n'aurait sûrement pas évolué sans le changement du learning rate. Hors, lorsqu'il est modifié (aux alentours du 27ème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



Ce "pic" a été observé lors d'entraînement précédents, ici deux autres exemples.



Ce dernier graphique 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.



En conclusion, que ce soit pour entraîner ou pour parfaire un entraînement, cet algorithme bien configuré peut apporter quelques points supplémentaires en précision.

Bilan sur les différents modèles

EfficentNet

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

Vu les faibles résultats lors de la seconde itération, il ne convenait sans doute pas à notre reconnaissance d'image.

ResNet50

Ce modèle a encore été utilisé sur cette itération pour tester les différentes optimisations présentées dans ce rapport. On note une dégradation de l'accuracy qui est descendue de 82% à 62% en passant de 10 à 20 classes à identifier. Nous avons différentes hypothèses que nous essayons de vérifier.

VGG16 et 19

Nous avons beaucoup utilisé ces deux modèles lors de nos tests. A quelques reprises, le VGG19 à donné de meilleurs résultats, mais cela reste d'un ordre de grandeur assez faible pour faire la différence. Comme RestNet50, on note une plus faible précision avec 20 classes.

Téléchargements, suite et … suite

Nous poursuivons les téléchargements en tâche de fond et nous avons pu augmenter le nombre de classes à tester. Nous continuerons d'augmenter ce nombre en vue de l'entraînement sur notre modèle final optimisé.
La semaine dernière nous étions sur 10 classes de référence, nous sommes passés à 20 classes à entraîner.
Nous avons également testé avec un nombre d'images différent par classes: (2000, 4000)

Nous devons aussi présenter nos errements. Au tout début du projet nous avons constitué un dataframe de 2 300 000 lignes. Afin d'obtenir un élément distinctif pour chaque images (lignes du dataframe) nous avions utilisé l'index de ce dataframe. Cet artifice nous convenait surtout pour son côté bijectif d'une ligne avec son image.

Il nous est arrivé une erreur de débutant, nous avons manipulé ce dataframe!!!!!! : olala oui

- -suppression de lignes
- -suppression de colonnes
- -masques
- - etc

Bilan: Quand au début du téléchargement certaines images étaient téléchargées avec cet index en nom d'identifiant, il ne correspondait plus à rien avec le dataframe trois semaines plus tard. Nous n'avons pas essayé longtemps de résoudre cette erreur. Au vues des progrès réalisés dans le téléchargement, nous avons choisi de reconstruire notre base de données avec un identifiant qui, lui, était bien une colonne fixe et stable.

Conclusions

Reste à tester

- Unfreeze des dernières couches du modèle
- L'optimisation d'ImageDataGenerator à arriver à finaliser

Exécution sur des serveurs AWS :

Nous sommes preneurs s'il y a un cours disponible pour utiliser un environnement chez AWS. L'idée serait d'uploader nos images sur un bucket S3 puis d'utiliser une instance EC2 de type P4 par exemple. Un tuto step by step serait l'idéal...
Nous pouvons chercher nous-même mais ça sera trop lent.

Qualité des photos :

Bien que nous ayons déjà bien épuré la base des images d'enveloppes ou autres, nous aimerions complètement automatiser cette détection. Les pistes envisagées tournent autour de l'analyse d'images (Histogramme ou autre). Une autre piste serait d'entraîner un modèle pour faire la différence entre une enveloppe ou une photo de champignon. Nous explorerons cette piste dans le week-end.

Annexe : Machines à disposition

Un pc fixe avec une carte nvidia 1070 gtx, fonctionne très bien pour tensorflow
Un pc portable avec une nvidia 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.