Deep Learning : Cassava Leaf Disease Classification
ESGI

Eddy CAI, Hakim MBAE, Rakib SHEIKH, Tom TARANTO

Jeudi 11 Février 2021

Résumé

Dans ce rapport, on utilise un système intelligent pour détecter les maladie des feuilles de manioc qui est le deep learning. Ce rapport présente avant tout la démarche employé pour parvenir à des résultats des entraînement de modèles tel que le modèle linéaire, perceptron multicouches, réseaux neurones de convolution. On utilise le python interactif par l'intermédiaire de jupyter-notebook. Cinq classes de maladies ont été utilisée dans ce jeux de données, à savoir la brûlure bactérienne du manioc (CBB pour Cassava Bacterial Blight), la maladie du bifteck brun du manioc (CBSD pour Cassava Brown Steak Disease), l'acarien vert du manioc (CGM pour Cassava Green Mite), la maladie de la mosaïque du manioc (CMD pour Cassava Mosaic Disease), et la manioc saine. Les résultats du rapport présenté montre que la précision globale sur les données de test obtenue est de $81,9\%$ avec le réseau de convolution de G. Sambasivam

1 Introduction et motivation

Deuxième fournisseur d'hydrates de carbone en Afrique, le manioc est une culture clé pour la sécurité alimentaire des petits exploitants agricoles, car il peut résister à des conditions difficiles. Au moins 80% des exploitations agricoles familiales l'Afrique subsaharienne cultivent cette racine amylacée, mais les maladies virales sont une source majeure de mauvais rendements. Avec l'aide de la Data Science, il pourrait être possible d'identifier les maladies courantes afin de les traiter.

Les méthodes existantes de détection des maladies exigent que les agriculteurs sollicitent l'aide d'experts agricoles financés par le gouvernement pour inspecter visuellement et diagnostiquer les plantes. Ces méthodes sont coûteuses, peu exigeantes en main-d'œuvre et en approvisionnement. En outre, les solutions efficaces pour les agriculteurs doivent être performantes dans des conditions très difficiles, car les agriculteurs africains n'ont parfois accès qu'à des caméras de qualité mobile à faible bande passante.

Kaggle, un site communautaire dont le but est de faire entraîner les Data Scientistes à résoudre des problématiques de la Data Science, machine learning et des analyses de problèmes prédicative, a lancé une compétition sur la classification des maladies du manioc. Les données sont composée d'un ensemble de 21 367 images labélisés, recueillis lors d'une enquête régulière en Ouganda. Ces données ont été fournis par les agriculteurs de leur jardin, annotées par des experts de la National Crops Resources Research Institute (NaCRRI), et en ayant en collaboration avec l'AI Lab de l'université Makerere, Kampala. Ces images sont présentées dans un format qui représente de la manière la plus réaliste possible ce que les agriculteurs devraient diagnostiquer dans la vie réelle.

Concrètement, notre tâche consiste à classer chaque image de manioc en quatre catégories de maladies ou une cinquième catégorie indiquant une feuille saine. [1, 2]

2 Méthodologie

Nous avons repris les meilleurs modèles que l'on ai eu sur le CIFAR-10. Les types de modèles sont les suivants :

— Perceptron Multi Couche.
— Réseaux de couche de convolution.

Nous avons trouvé un article qui prétend avoir atteint 93% en utilisant un bloc de couches de convolution et un perceptron totalement connecté. Nous avons décidé de le ré-implémenter le modèle afin de savoir si nous pouvons aussi atteindre ce résultat, voir même l'améliorer avec du fine tuning. [3]

L'intégralité des modèles sont implémenté en python, avec l'architecture Tensorflow 2.4.0. Une partie de nos codes ont été exécuté sur nos machines personnels d'une part, et sur Kaggle d'autre part.

Afin de lutter contre les initialisations aléatoires, nous avons décidé de lancer chaque modèle plusieurs fois avec une graine fixée et différente pour chaque expérience.

2.1 Perceptron Multi Couches

2.1.1 Préparation du jeux de données

Nous avons utilisé comme pré-traitement, les fonctions de ImageDataGenerator de tensorflow mais le temps de chaque epoch était extrêmement long, nous chargeons toutes les images à chaque epoch et cela nous faisait perdre beaucoup de temps. Pour palier à se problème, nous avons eu un magnifique cours de Deep Learning avec Mr. VIDAL qui nous a montré les caches. Les caches étant très intéressants, on charge une fois les images puis nos entraînements deviennent très rapide car le CPU pouvant travailler en parallèle du GPU, les temps de latence liés au chargement et à la transformation des images est donc globalement plus rapide. Dans nos ImageDataGenerator, on utilise le principe de normalisation pour avoir nos valeurs entre 0 et 1 afin d'éviter d'énormes écarts entre les poids du modèle. Puis, on utilise les principes de rotation, de translation, de zoom et d'inversement sur notre jeux de données d'entraînement seulement. Le pré-traitement que Mr. VIDAL nous a montré lors de son cours nous a beaucoup aidé à gagner du temps lors des entraînements, c'est pour cela que l'on utilise uniquement ce pré-traitement pour nos modèles de Perceptron Multi Couches.

Nous avons remarqué que le jeu de donnée n'était pas équilibré. On a donc décidé de prévoir un ajustement des poids des classes lors de l'entraînement de nos modèles. Les classes les plus représentées possédant un facteur de poids plus faible.

2.1.2 Architecture du modèle

L'architecture d'un Perceptron Multi Couches est simple, c'est une succession de couches Dense. Nous avons choisis arbitrairement 1 million de variables et nous voulons savoir si à nombre de variables équivalent, il était plus intéressant d'avoir plus de couches avec une taille plus petite ou s'il était plus intéressant d'avoir moins de couches avec une taille plus grande. Nous utilisons comme optimizer par défaut 'Adam' [4]. Et nous utilisons comme activation "Relu".

2.1.3 Fine-tuning

Nous avons donc entraîné des modèles de perceptron multi couches à un million de variable, notre premier modèle étant deux couches Dense de taille 1000, notre second modèle étant trois couches de de taille 707 et notre troisième modèle étant quatre couches de taille 577. Nous avons donc conclu que le fait d'avoir plusieurs couches avec de plus petite taille était plus intéressant pour notre modèle, sur la figure : 2.1.3

Nous remarquons que le troisième modèle avec quatre couches de tailles 577 commençait légèrement à apprendre dans le jeu de test. Nous aurions pu continuer sur ce chemin et ainsi augmenter le nombre de couches et diminuer la taille des couches pour atteindre du sur apprentissage et par la suite essayer de combattre cet over-fitting avec les différentes méthodes apprises en cours (Dropout, kernel regularizer, data augmentation, batch regularisation, pooling). Mais nous avons préféré nous concentrer sur le modèle de convolution de l'article suivant [3]. Nous avons tout de même utilisé de la batch normalisation pour voir son impact sur nos modèles et il est visible que nos modèles se généralisent très mal (Figure : 1). Nous n'avons pas donné suite à ces tentatives pour les raisons évoquées.

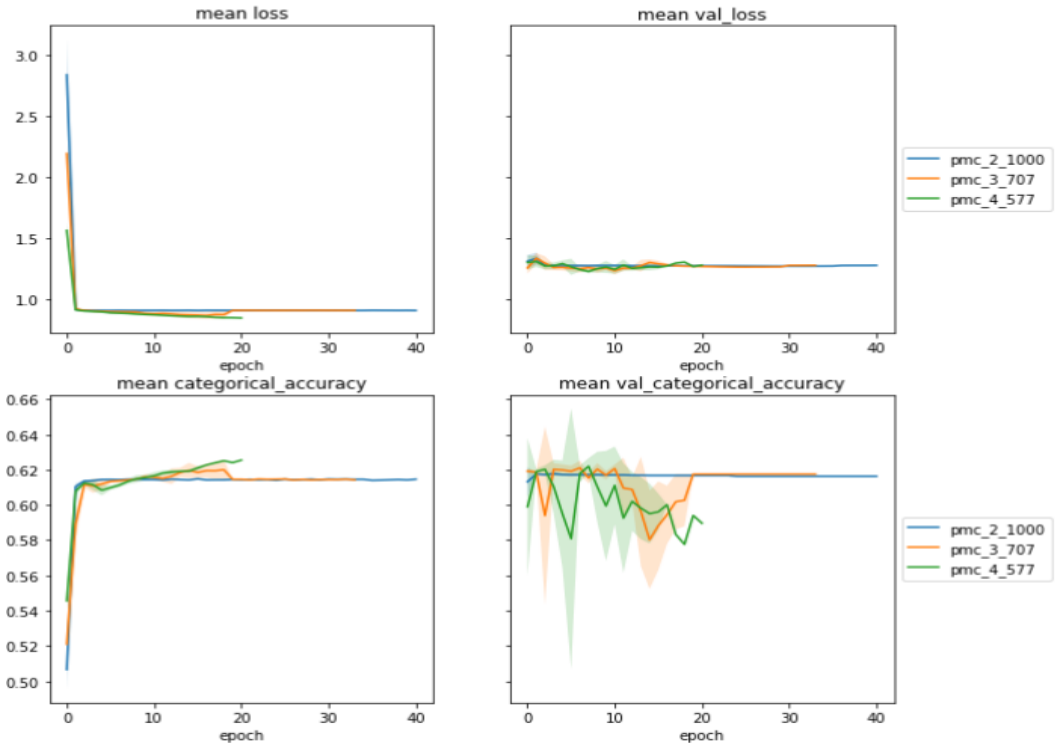


FIGURE 1 – Résultats obtenus avec perceptron multi couches

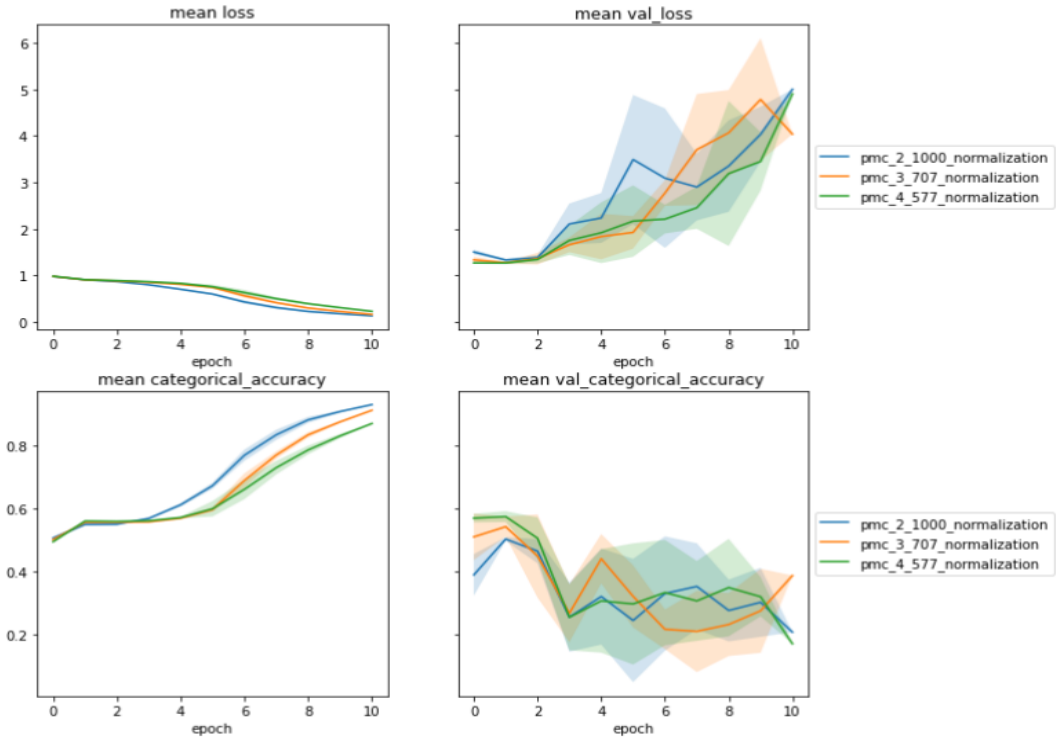


FIGURE 2 – Résultats obtenus avec perceptron multi -couches et régularisation

2.2 U Net

2.2.1 Préparation du jeu de données

Nous avons choisi de redimensionner les images à la taille 488×488. Nous avons optimisé les fonctions utilisées car le modèle a été créé grâce aux TPU Kaggle. Nous avons utilisé l'augmentation des données, les paramètres sur lesquels nous avons joué sont : des rotations aléatoires de gauche à droite, de la saturation,

de la luminosité, du contraste et la qualité de l'image.

Nous avons utilisé la mise en cache pour avoir un chargement des données plus rapide, en effet, le CPU pouvant travailler en parallèle du TPU, les temps de latence liés au chargement et à la transformation des images est donc globalement plus rapide.

2.2.2 Architecture du modèle

Pour le modèle U-net, nous avons utilisé une profondeur de 3.

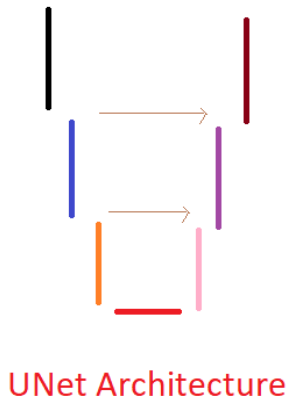


FIGURE 3 – figure
Modèle Unet par bloc

Le premier bloc d'entrée est constitué d'une couche de convolution 2D composée de 32 filtres avec des noyaux de taille 3. Nous avons utilisé un pas de 2. Nous y avons ajouté un layer de batchNormalisation, puis une couche d'activation avec la fonction relu. Pour la descente, les filtres valent successivement 64 et 128. Chacun de ces bloc est constitué d'une couche de convolution avec un noyau de taille 3, suivi d'une couche de batch normalisation, puis d'une couche d'activation relu. Ensuite, nous avons une nouvelle couche de convolution 2D de la même taille que la précédente, avec les mêmes paramètres, suivie d'une couche de batch normalisation. Nous avons terminé avec une couche de maxpooling de taille 3 avec un pas de 2. Pour chaque bloc, il ne faut pas oublier de créer les résidus que l'on appellera lors de la remontée.

Pour la remontée, nous avons utilisé des blocs dont les différents filtres sont de tailles successives 128, 64, 32. Chaque bloc étant composé d'une couche d'activation relu, suivi d'une convolution de taille 3 et d'une couche de batch normalisation. Ensuite, à nouveau une couche d'activation relu, suivi d'une couche de convolution avec les paramètres identiques, suivi d'une couche de batch normalisation. Et enfin une couche d'upsampling avec 2 en paramètre afin de doubler la taille de "l'image" obtenue. A chaque fin de bloc, nous rajoutons le résidu correspondant .

Pour terminer, nous ajoutons une dernière couche de convolution avec des filtres de taille 5 et des noyaux de taille 3. Puis nous aplatissons le résultat et ajoutons une couche Dense de taille 5 afin de pouvoir obtenir notre prédiction.

2.2.3 Fine-tuning

Au début, nous voulions utiliser une profondeur de 4, mais une erreur de mémoire apparaissait sur les machines Kaggle, l'erreur étant très obscure, nous avons donc décidé de supprimer le bloc le plus profond afin de réduire la quantité de mémoire utilisée. Nous avons essayé de modifier les fonctions d'activation, mais cela donnait de très mauvais résultats.

Ensuite nous avons essayé d'augmenter la taille du noyau de la dernière couche mais les résultats étaient encore pires.

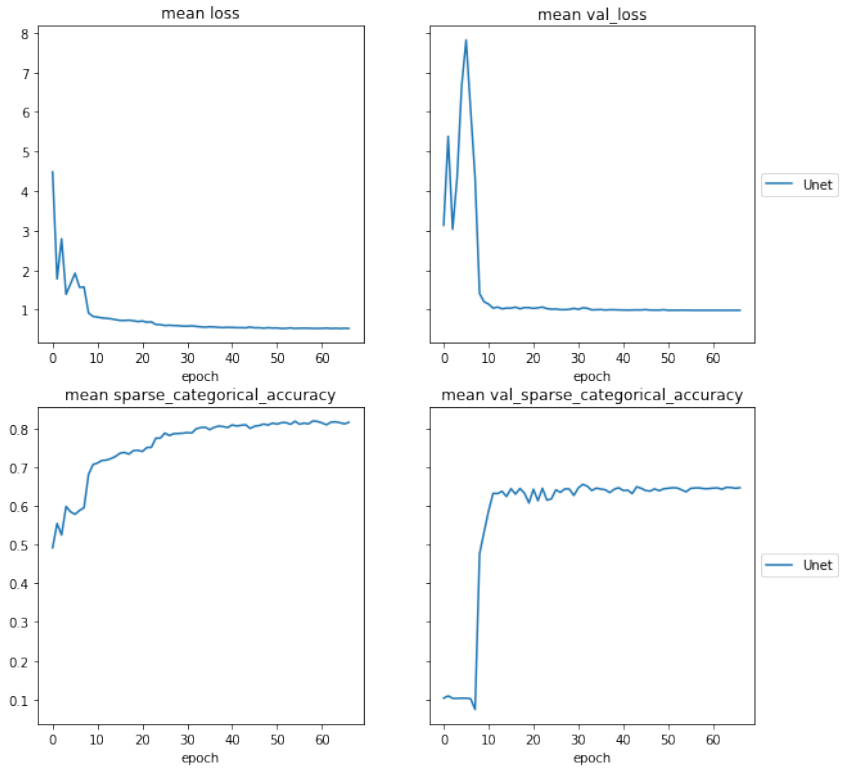


FIGURE 4 – Résultats obtenus avec Unet

2.3 ResNet50

2.3.1 Préparation du jeux de données

Nous avons essayé d'implémenter ce réseau sur nos machines ainsi que sur les machines Kaggle, nous avons très rapidement obtenu des erreurs de mémoire. En utilisant des machines possédant des cartes graphiques de classe industrielle, nous avons pu utiliser ce modèle. Par contre, n'ayant plus accès à cette machine, il nous a été impossible de réitérer nos résultats ainsi que d'utiliser les méthodes vues dans les derniers cours. Par exemple, nous n'avons pas utilisé le cache pour pré-charger les images modifiées. La durée d'exécution était très longue, nous n'avons pu faire que deux expériences en utilisant le modèle non entraîné Resnet50 disponible depuis la librairie Keras.

2.3.2 Architecture du modèle

Le Resnet 50 est composé de 50 couches de convolutions successives avec des liens entre des couches séparées. Ces liens connectent une couche sur trois et permettent la copie des poids de manière identique au début du prochain bloc. Chaque bloc étant constitué de 3 couches de convolutions avec des filtres de taille allant de 32 à 512.

2.3.3 Fine-tuning

Nous n'avons pas pu ajuster les paramètres de ce modèle.

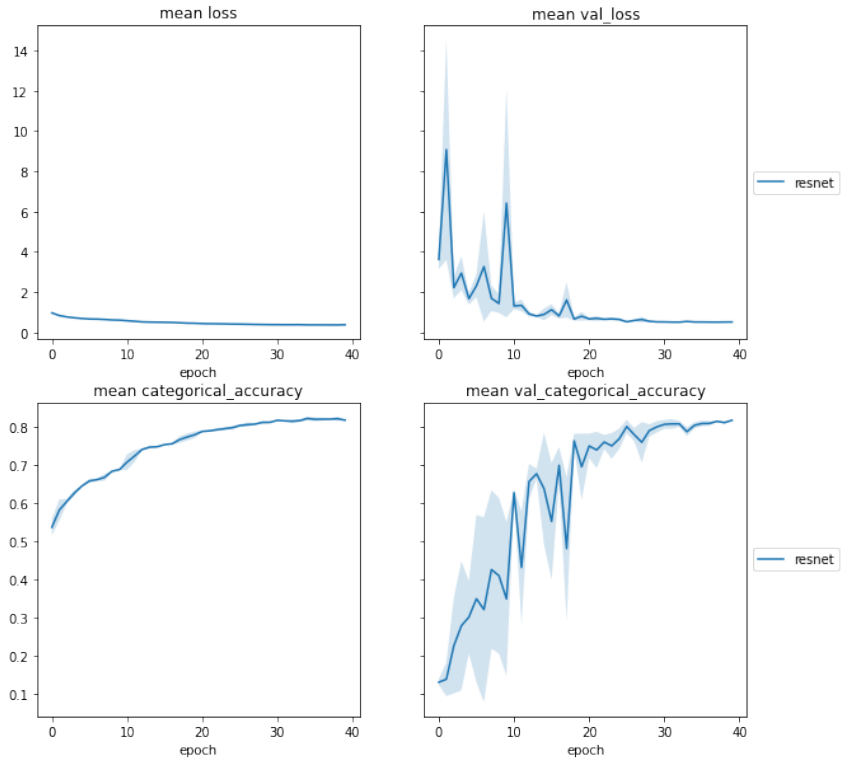


FIGURE 5 – Résultats obtenus avec resnet

2.4 Convolution Sambasivam

2.4.1 Préparation du jeu de données

On utilise les TFrecords [5] qui sont déjà générés dans le but de pouvoir accélérer le chargement du jeu de données. Le jeu de données est divisé en deux sous jeux de données nommés entraînement et validation qui sont inégaux (90% pour l'entraînement, 10% pour la validation, le pourcentage représente le nombre de fichiers TFrecords et non le nombre d'image). Il y a donc 14 TFrecords choisis aléatoirement dédiés à l'entraînement et 1 dédié à la validation. On applique ensuite d'un pré-traitement à ce jeu de données. La taille de l'image est réduite à 256 × 256 afin de lutter contre les dépassements de mémoires liés à la RAM et la VRAM. Vient ensuite l'augmentation des données. On a utilisé les augmentations suivantes :

- random_flip_left_right
- ramdom_flip_up_down
- random_saturation
- random_constrast
- random_jpeg quality
- image.rot90 (avec une fonction aléatoire uniforme)

2.4.2 Architecture du modèle

Comme décrit dans l'article [3], le modèle est composé d'un bloc de 3 couches de convolution suivi d'un bloc de 4 couches denses reliées entre elle par des couches de dropout. La première Conv2D se compose d'une couche 32 unit, avec un kernel de 5 × 5. Elle est suivie d'un BatchNormalization et d'un MaxPooling à 3 × 3. La deuxième couche et la troisième sont composées de deux Conv2D à la suite de 64 unit, 3 × 3 et 128 units, 3 × 3, suivies également d'un batch normalization et MaxPooling à 3 × 3. Pour les couches denses, chacune d'entre elles contiennent l'activation "ReLu", et un regularizer L1 L2 fixés respectivement à 1e-5 et 1e-4. Elles sont toutes suivies d'un dropout à valeur identique.

2.4.3 Fine-tuning

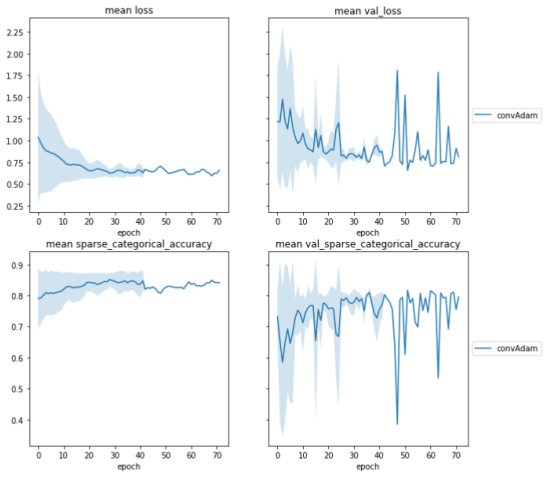
Au départ, nous avons eu une hausse assez importante de la perte (loss) lorsque le modèle ne contenait aucune régularisation. Nos validations initiales étaient aux alentours de 65 %. Nous avons ensuite rajouté

le dropout dans le but d'encourager le modèle à apprendre. On a commencé avec une valeur à 0.3 mais le modèle a subi une forte hausse de la perte avec une précision convergente vers 0. Nous avons ensuite rajouté une régularisation de noyaux L1 L2 à chaque couche dense selon les valeurs recommandées dans cet article [6]. On a donc choisi pour $L1 = 0.05$ et pour $L2 = 0.01$. Nous n'avons pas remarqué de gains significatifs jusqu'à ce que l'on mette les valeurs pour $L1$ à $1e-5$ et pour $L2$ à $1e-4$. En effet, avec ces valeurs, le modèle a enfin pu décoller au niveau de l'apprentissage avec un score de 75 %. Nous avons continué le fine-tuning avec les valeurs du dropout à 0.03, 0.2 et 0.3 toujours dans le même but de forcer le modèle à apprendre.

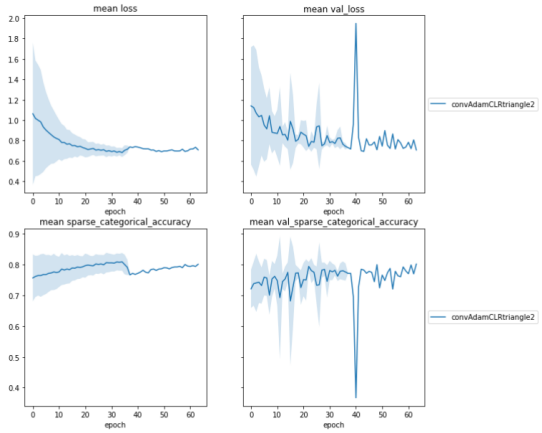
En étudiant l'article [3] plus en profondeur, nous avons remarqué que l'auteur utilise un plannificateur de taux d'apprentissage de type cyclique. Afin de gagner du temps, nous avons repris les valeurs de référence pour le taux d'apprentissage minimal et maximal qui est de $1e-7$ et $1e-3$. L'utilisation de ce nouveau taux d'apprentissage nous pousse à utiliser l'optimisation Adam [4] avec un paramètre de taux d'apprentissage cyclique Nous avons testé les deux méthodes de calcul du taux d'apprentissage cyclique c'est-à-dire la méthode de la "triangulation 2" ($\frac{1}{2.0\times(x-1)}$) et la méthode exponentielle $\gamma \times x$.

Nous sommes allés plus loin sur le choix de la fonction d'optimisation avec AdamW [7]. Par manque du temps, nous n'avons pas pu explorer le potentiel du fine-tuning pour la partie décroissance des poids en plus du fait que cette notion reste floue pour nous. Nous avons donc choisi arbitrairement une décroissance des poids égale à $1e-6$. Nos résultats se sont largement améliorés car nous avons réussi à avoir une précision de 83 % lors de la validation du modèle. Pour la partie test soumission Kaggle, la meilleure précision que l'on ait obtenu est de $81,9$ % Nous sommes cependant satisfaits de notre fine-tunning car nous avons dépassé la barre des 80% même si l'objectif initial pour ce modèle était d'atteindre au moins 85 %

Les hyper-paramètres utilisés pour avoir un score de $81,9$ sont les suivants :
- Taille d'image : 256×256
- Régularisation des noyaux L1 $=1e-5$ et L2 $1e-4$
- Dropout : 0.3
- Activation : ReLu (REctified Linear Unit) pour les perceptrons, et Softmax pour le dernier perceptron de classification.
- Optimisation avec AdamW avec une décroissance des poids à $1e-6$ associé au taux d'apprentissage cyclique de type exponentiel $\gamma * x$ avec $\gamma = 1$ (min $= 1e-7$, max $= 1e-3$).



(a) Sambasivam Adam 79.5%



(b) Sambasivam Adam et CLR Traingle 2 80.4%

FIGURE 6 – Graphes d'entraînement de la convolution Samasivam

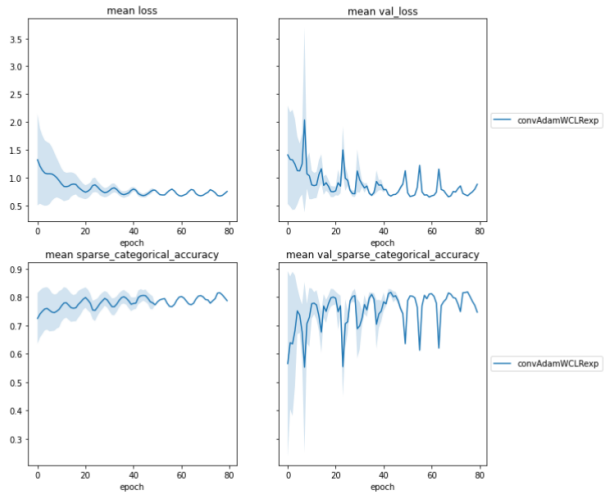
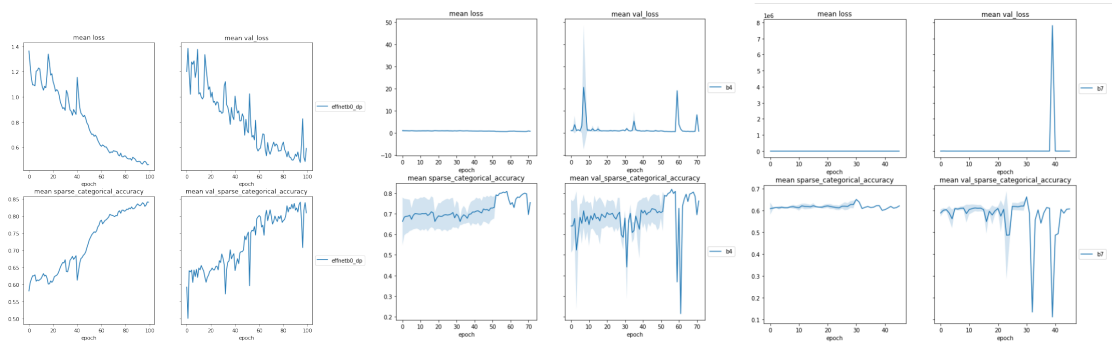


FIGURE 7 – Sambasivam AdamW et CLR Exponentiel 81.9%

2.5 EfficientNet-B0 B4 et B7

Nous avons remarqué que les modèles EfficientNet [8] sont actuellement la tendance lors de cette compétition lors du survol du forum de discussions. On pourrait même penser que les personnes ont obtenu des résultats à 90% rien qu'en utilisant ce modèle. De ce fait, nous avons suivi la tendance en implémentant l'EfficientNet-B4. Ce modèle est déjà disponible sur le framework Keras. [9] Sur recommandation de Monsieur VIDAL Nicolas, nous n'avons pas utilisé le modèle pré-entraîné.

La préparation du jeu de données reste identique à celle qui est présentée pour les convolutions Sambasivam.



(a) Graphe de Efficientnet B0 (b) Graphe de Efficientnet B4 (c) Graphe de Efficientnet B7

FIGURE 8 – Ensemble des graphes de Efficient B0, B4 et B7

3 Discussion des résultats et bilan

Pour le perceptron multi couches, nous avons essayé d'apporter une analyse critique par rapport à un nombre de variables total fixé. Nous en avons conclu que les PMC classiques sont plus efficaces avec plus de couches plus petites. Nous avons aussi vu qu'il fallait s'éloigner de l'architecture pure, et rajouter de la normalisation afin d'obtenir des résultats légèrement meilleurs. Globalement, ces modèles ne nous ont pas permis d'obtenir un résultat compétitif en validation.

Nous pensions que le u-net se comporterait beaucoup mieux en validation. En terme d'apprentissage, les valeurs sont correctes. Pour réellement pouvoir donner une conclusion, nous devrions rajouter a minima une profondeur supplémentaire. Notons quand même que ce modèle performe mieux que le perceptron multi couches.

Le ResNet50 est un réseau ultra performant. Ces bons retours en validation sont largement dus à la taille du modèle. Pour tirer des conclusions quant à la durée d'apprentissage, il faudrait réutiliser les connaissances que nous avons acquises depuis. Il aurait pu être intéressant d'essayer d'améliorer ce modèle.

Lors de nos runs sur les TPU, nous avons remarqué que la taille de l'image est un hyper-paramètre très important. En effet, une taille d'image plus grande permet de mieux apprendre et d'ainsi obtenir un meilleur score en validation. Ce point a également été mentionnée par l'article [3] recommandant de fixer la taille de l'image à 448×448. Nous n'avons pas modifié cet hyper-paramètre en raison des contraintes techniques imposée par la compétition Kaggle sur les GPU.

Nous remarquons que sur les figures 6b et 7 que les courbes ont une tendance de montagne russe. En effet, nous n'avons pas réussi à trouver les bons hyper-paramètres pour que le Cyclical Learning Rate puisse s'ajuster de manière optimale afin d'avoir une courbe régulière qui converge. De plus, cela ralentit le temps d'entraînement à intervalle régulier car la perte est repartie à la hausse suite à l'ajustement trop brusque des poids. En comparant ces résultats par rapport à notre EfficientNet-B4, ce modèle a réussi à avoir un meilleur test sur le serveur Kaggle avec un temps d'entraînement réduit (compter 1h à 2h pour 100 epochs).

Nous remarquons aussi par superposition des graphes de l'EfficientNet-B4 et de la convolution Sambasivam + AdamW + CLR Exponentiel (figure 9) que l'on pouvait atteindre une meilleure précision lors de l'entraînement avec un modèle plus léger qu'avec un modèle très lourd.

Avec le U-net, on converge déjà à partir de la 15ème epoch (figure 4). La durée de l'expérience est environ de 6 heures. On peut en déduire que le U-Net n'est pas efficace sur le temps de l'entraînement par rapport à la précision. Sans modification des paramètres et avec cette profondeur, les résultats globaux sont décevants.

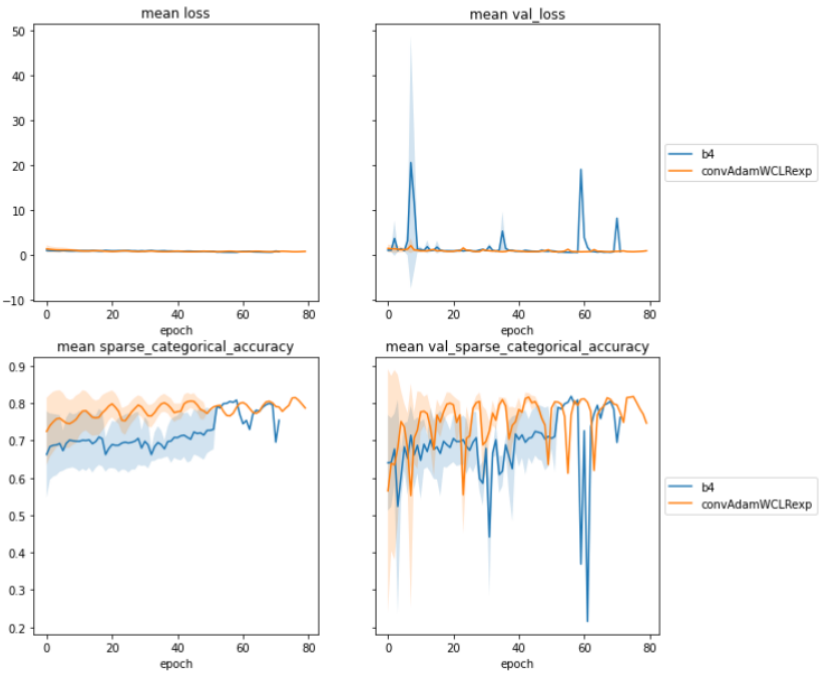


FIGURE 9 – Comparaison des deux modèles, B4 et Convolution Sambasivam CLR Exponentiel

Références

[1] Kaggle. *Cassava Leaf Disease Classification*. URL : https://www.kaggle.com/c/cassava-leaf-disease-classification.

[2] Kaggle. *What is Kaggle ?* URL : https://www.kaggle.com/getting-started/44916.

[3] G. Sambasivam et Geoffrey Duncan Opiyo. « A predictive machine learning application in agriculture : Cassava disease detection and classification with imbalanced dataset using convolutional neural networks ». In : *Egyptian Informatics Journal* (2020). ISSN : 1110-8665. DOI : https://doi.org/10.1016/j.eij.2020.02.007. URL : https://www.sciencedirect.com/science/article/pii/S1110866520301110.

[4] Diederik P. Kingma et Jimmy Ba. *Adam : A Method for Stochastic Optimization*. 2017. arXiv : 1412.6980 [cs.LG].

[5] Tensorflow. *TFRecords*. URL : https://www.tensorflow.org/tutorials/load_data/tfrecord.

[6] Ozgur Demir-Kavuk et al. « Prediction using step-wise L1, L2 regularization and feature selection for small data sets with large number of features ». In : *BMC bioinformatics* 12 (oct. 2011), p. 412. DOI : 10.1186/1471-2105-12-412.

[7] Ilya Loshchilov et Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv : 1711.05101 [cs.LG].

[8] Mingxing Tan et Quoc V. Le. *EfficientNet : Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv : 1905.11946 [cs.LG].

[9] Keras. *Keras EfficientNet API*. URL : https://keras.io/api/applications/efficientnet/.