OPENCLASSROOMS

Projet n°8
Rapport

Classification de fleurs en utilisant des TPU

Étudiant :
Fayz El Razaz

Tuteur :
Amine Hadj-Youcef

27 octobre 2022

Table des matières

1 Introduction

Dans le cadre du dernier projet du parcours Ingénieur machine learning, nous sommes amenés à participer à une compétition kaggle.

Nous avons fait le choix de participer à une compétition où l'on doit classifier des images de fleurs en utilisant des TPUs sur le cloud.

Nous avons pu constater au cours de la formation, le gain de temps lors du passage de l'utilisation du CPU au GPU. Nous avons alors souhaité travailler sur une problématique de deep learning utilisant cette technologie afin de constater le nouveau gain dans l'utilisation des TPUs pour effectuer de l'apprentissage automatique.

Nous présenterons donc au sein de ce document, la technologie TPU développée par Google, spécialement pour la librairie tensorflow, et pour l'apprentissage de réseaux de neurones profonds. Nous poursuivrons avec la présentation de notre jeu de données et de la préparation permettant l'apprentissage et terminerons avec les différents modèles implémentés ainsi que leur performance avant de conclure.

Notre travail se trouve au lien suivant : `https://www.kaggle.com/code/fayzerr/flower-classification-on-tpu-using-cnn-models`

2 Tensor Process Unit

2.1 Technologie TPU

Les TPU (Tensor Processing Units) sont des circuits intégrés spécifiques aux applications (Application-Specific Integrated Circuit ou ASIC), développés par Google et permettant d'accélérer les charges de travail de machine learning. Leur conception repose sur la vaste expérience de Google et sur son leadership en matière de machine learning. [1]

Les ressources Cloud TPU sont capables de calculer très rapidement des vecteurs et matrices denses. Le transfert de données entre une ressource Cloud TPU et la mémoire hôte apparaît comme lent comparé à la vitesse de calcul. En effet, la vitesse du bus PCIe est bien plus lente que l'interconnexion des ressources Cloud TPU et la mémoire à haut débit sur puce. La compilation partielle d'un modèle, dont l'exécution ne cesse de basculer entre l'hôte et l'appareil, entraîne l'inactivité du TPU la plupart du temps, en attendant l'arrivée des données sur le bus PCIe. Pour résoudre ce problème, le modèle de programmation Cloud TPU est conçu pour exécuter la majeure partie de l'entraînement sur le TPU, et idéalement la totalité de la boucle d'entraînement.

Voici quelques caractéristiques essentielles du modèle de programmation TPU :
Tous les paramètres du modèle sont conservés dans une mémoire à haut débit sur puce. Les coûts de calcul sur Cloud TPU sont amortis par l'exécution de nombreuses étapes d'entraînement au sein d'une boucle. Les données d'entraînement d'entrée sont diffusées dans une file d'attente "infeed" sur Cloud TPU. Un programme exécuté sur Cloud TPU récupère des lots de ces files d'attente à chaque étape de l'entraînement. Le serveur TensorFlow qui s'exécute sur la machine hôte (le processeur associé à l'appareil Cloud TPU) récupère les données et

les prétraite avant de les transmettre au matériel Cloud TPU. Parallélisme des données : les cœurs d'une ressource Cloud TPU exécutent un programme identique de manière synchrone au sein de leur propre mémoire à haut débit. Une opération de réduction est effectuée sur tous les cœurs à la fin de chaque étape du réseau de neurones.[2]

2.2 Utilisation sur le Cloud et limites temporelles du TPU

2.2.1 Implémentation sur Kaggle

Nous avons pour la première fois, travaillé sur la plateforme Kaggle, qui ouvre la possibilité d'implémenter le code sur la plateforme dans un contexte cloud. Nous avons directement codé sur la plateforme sans avoir besoin de récupérer les données ou créer un environnement virtuel en y installant les librairies nécessaires à la création de notre notebook.

Nous avons alors pu nous initier aux différentes commandes permettant de travailler directement avec les données mises à disposition dans la compétition et présentes directement sur le cloud. Nous nous sommes retrouvé ici confronté à certaines limites, notamment en matière de version des packages. Nous avons en particulier, souhaité mettre à jour la version de tensorflow (ici la version 2.4.1 était installé) mise à disposition. Sur Kaggle, les TPUs installé, ainsi que les GPUs, sont compatibles seulement avec certaines versions de Tensorflow, ce qui a limité notre utilisation de la bibliothèque où certains modèles n'étaient encore présent (notamment dans la bibliothèque Keras).

2.2.2 Limites temporelles du TPU

En plus des contraintes liés aux versions des bibliothèques, les TPU étant des ressources très demandées, celles-ci sont mises à disposition en durée limitée sur la journée et sur la semaine. Nous avons pu mettre en place nos modèles, et effectué nos tests, mais avons été limité, en particulier sur la fin du projet où le quota d'heures d'utilisation des TPU avait été dépassé.

Nous avons alors profité de cette limite pour revenir aux différents GPU proposé par la plateforme pour réimplémenter nos modèles et constater la différence de durée d'entrainement selon la technologie utilisée. On a constaté, un gain moyen de 900% (test effectué sur l'apprentissage d'un même modèle sur les photos de taille 192x192).

3 Compétition et données

3.1 Compétition

Nous avons choisi de participer à la compétition : Petals to the Metal - Flower Classification on TPU.
Cette compétition est idéale pour appréhender l'utillisation des TPU dans le traitement de l'image.
Nous avons pour mener à bien ce travail, récupérer le noyau de Phil Cullington, qui était mis à disposition pour appréhender le sujet. Nous avons repris ce notebook en prenant le temps de comprendre correctement l'ensemble des élements le composant en vue de l'améliorer. Initialement, l'accuracy du notebook était aux alentours de 35%. Nous avons réussi à élever ce taux à 95,89% après une quinzaine de tentative.

3.2 Données

Le jeu de données avec lequel nous avons travaillé est composé de 20 135 images de fleurs, réparties en 104 familles. L'ensemble du dataset était préchargé dans le dossier de travail sur kaggle, et le split était déjà prêt avec un jeu d'entraînement, de validation, et de test.

Nous avions à disposition l'ensemble du jeu de données avec des photos de différentes tailles. Nous avons choisi de garder les photos de plus grandes tailles (512 x 512) car c'est avec celle-ci, que les performances ont été les meilleures.

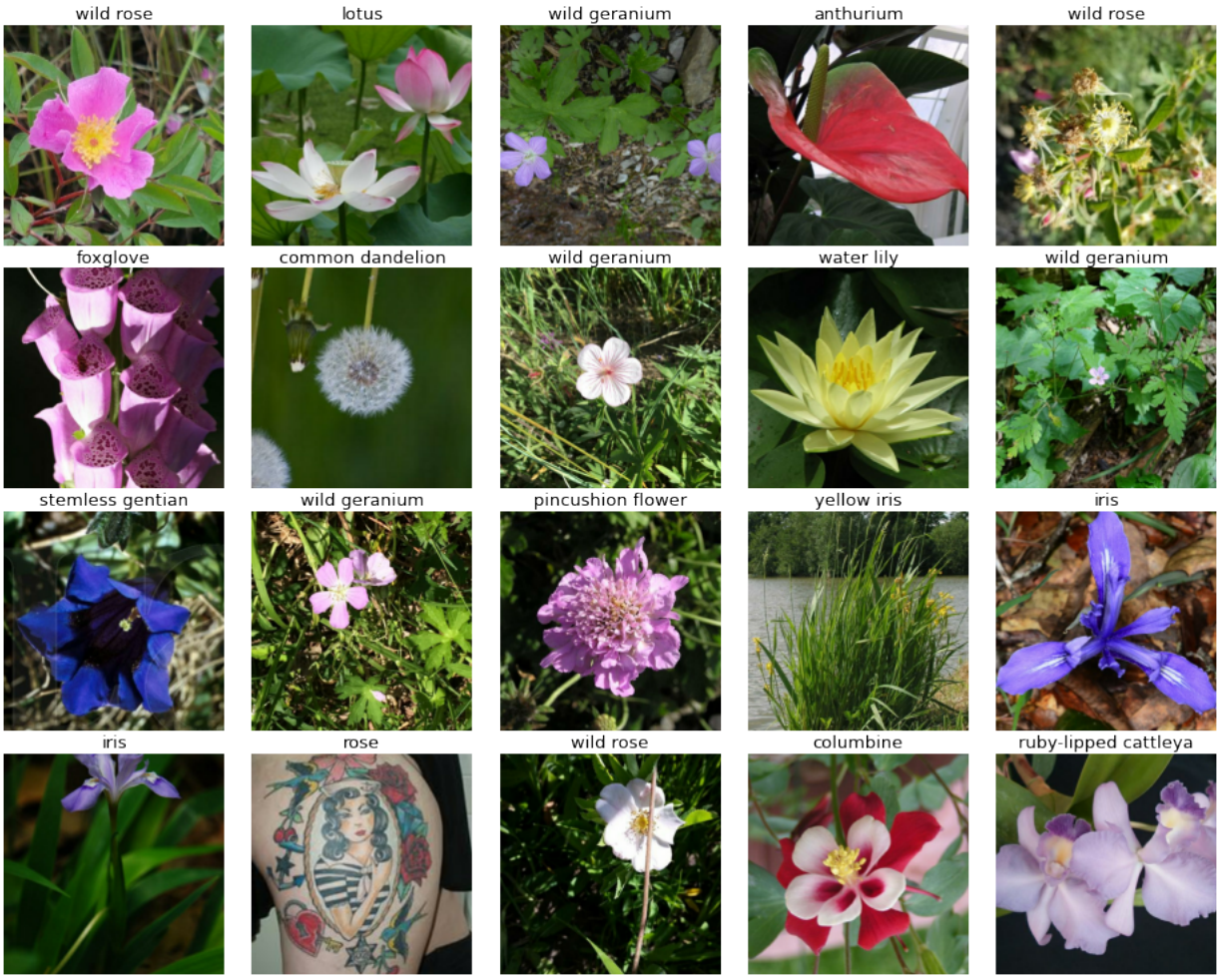

FIGURE 1 – Échantillon de données

3.2.1 Data Augmentation

Dans l'optique d'améliorer les performances des différents modèles, nous avons mis en place une fonction d'augmentation de données et l'illustrons ici :

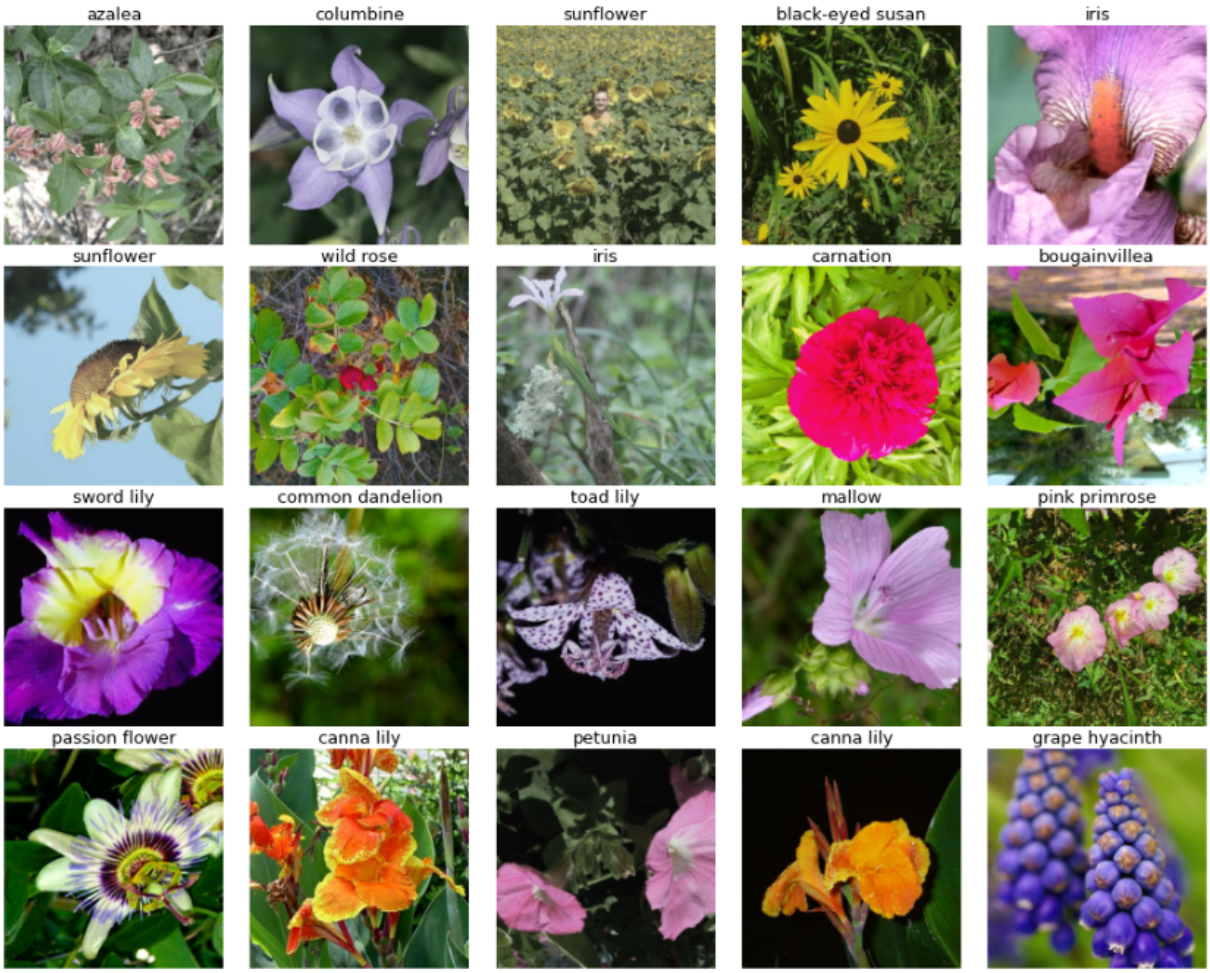

FIGURE 2 – Échantillon de données

On a effectué sur ces données les transformations suivantes :

- Symétrie horizontale aléatoire
- Symétrie verticale aléatoire
- Recadrage aléatoire
- Saturation aléatoire
- Contraste aléatoire

Il a été intéressant de constater qu'on a obtenu de meilleurs résultats en ne faisant aucune augmentation d'image. On a en effet constaté une perte allant de 2 à 4 points sur notre accuracy.

4 Modèles implémentés & performances

Nous avons testé différents modèles dans la réalisation de ce projet. Nous allons dans cette partie spécifié l'ensemble de ces modèles ainsi que les résultats obtenus.

4.1 Modèle DenseNet201 & performances

4.1.1 Performances

4.2 Modèle DenseNet201 & performances

Le premier modèle que l'on a testé, a été le DenseNet201. Nous avons récupéré le modèle pré-entraîné sur ImageNet qui compte plus d'un million d'images et mille classes d'objets. Le modèle compte 20,2 millions de paramètres, et nous avons ici fait le choix (après comparatif) de pouvoir modifié les valeurs paramètres (avec la commande pretrained_model.trainable = True). Ce modèle a la particularité d'être profond, avec 402 couches de profondeur.

4.2.1 Performances

On a obtenu les performances suivantes :

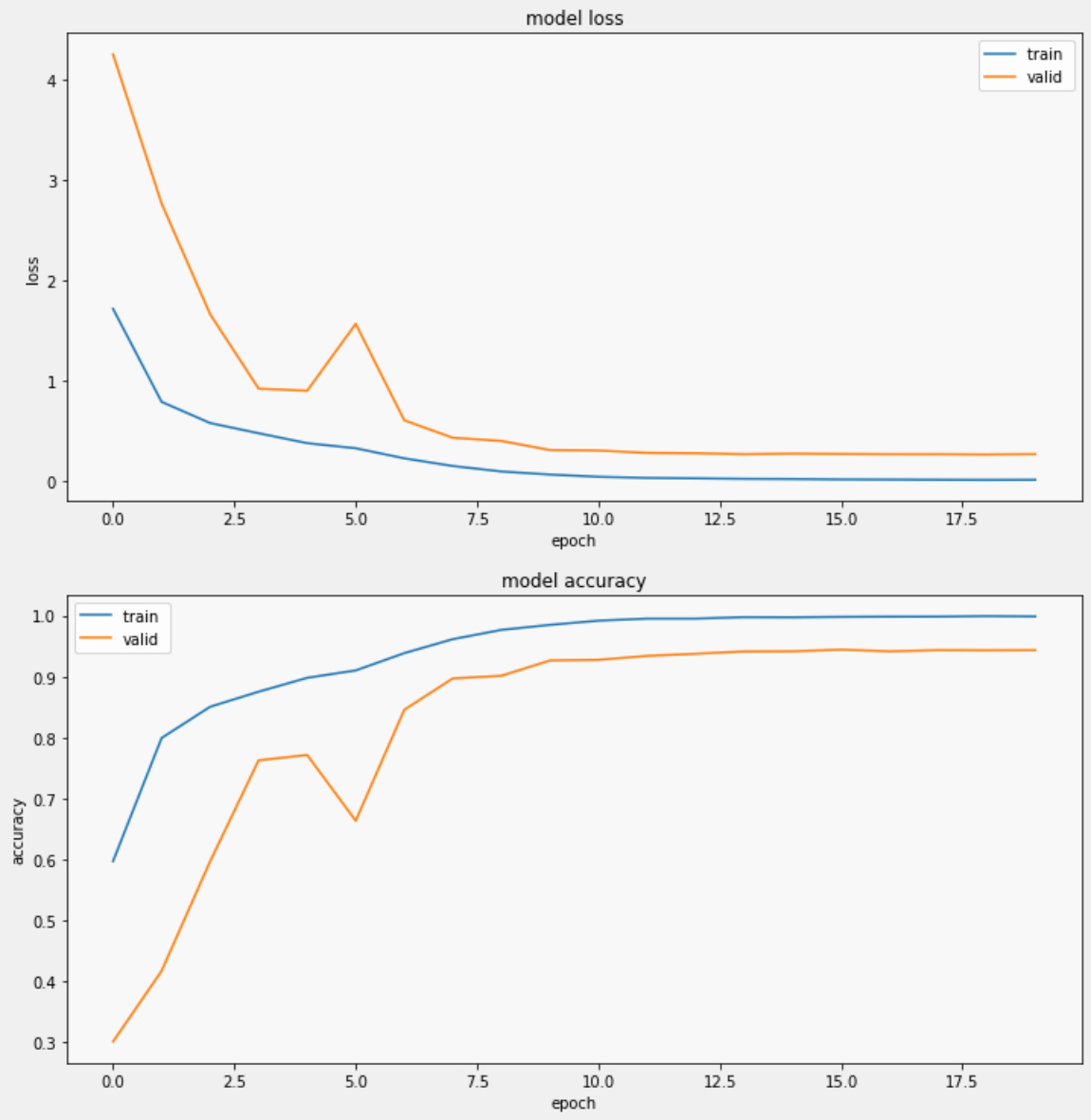

FIGURE 3 – Performances du modèle DenseNet201

Le modèle DenseNet201 nous a permis de dépasser la barre des 90% d'accuracy en convergeant aux alentours de 94% (sur le jeu de validation) pour une durée d'entraînement de 22 minutes et 37 secondes sur 20 epochs.

4.3 Modèle Xception & performances

Le modèle Xception a aussi été pré-entraîné sur ImageNet et compte 22,9 millions de paramètres avec 81 couches de profondeurs.

4.3.1 Performances

On a avec ce modèle obtenu les performances suivantes :

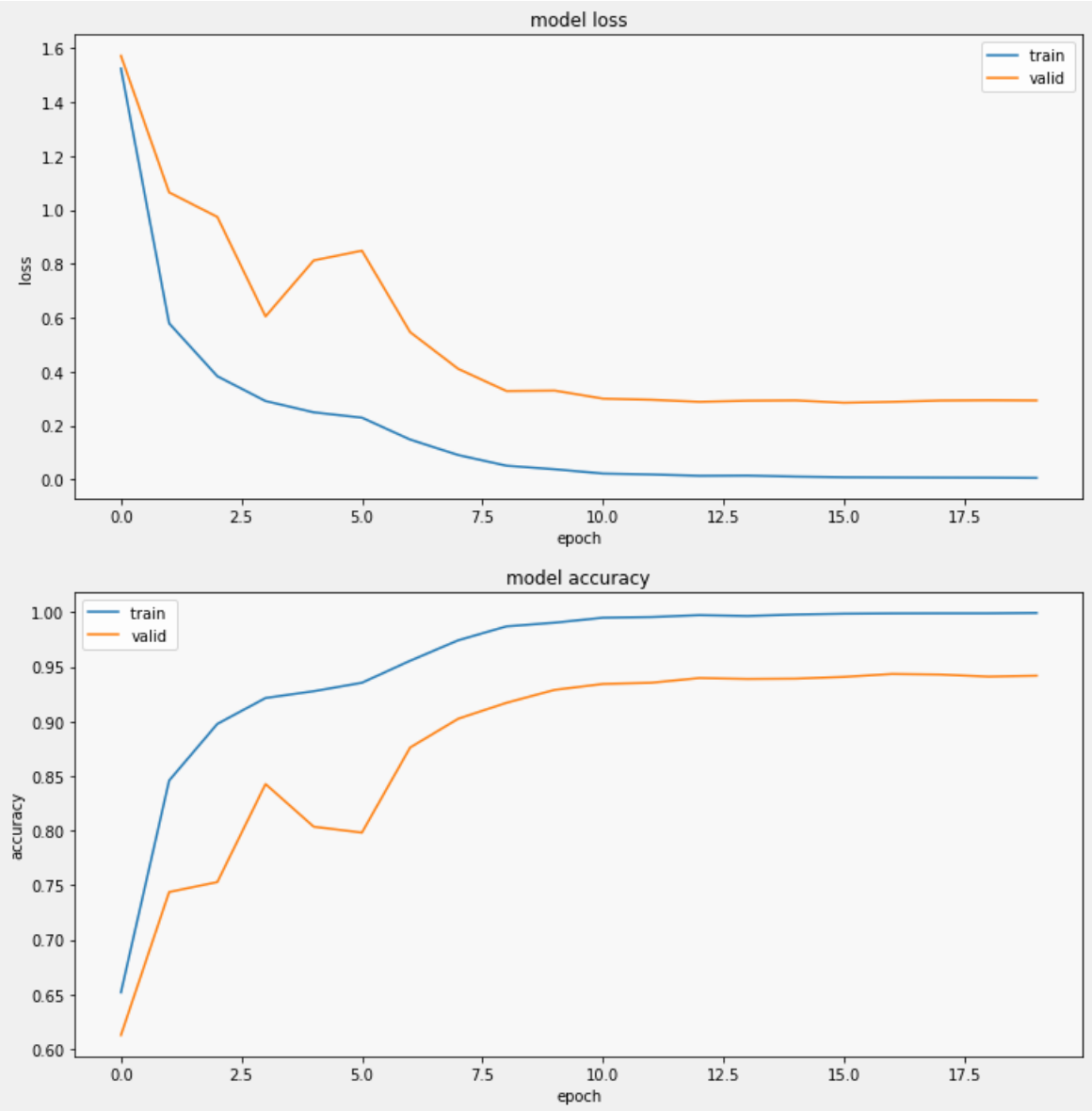

FIGURE 4 – Performances du modèle Xception

Le modèle a convergé à un taux d'accuracy aux alentours de 94% (sur le jeu de validation) pour une durée d'apprentissage de 18 minutes et 47 secondes sur 20 epochs.

4.4 Modèle ResNet152V2 & performances

Le modèle ResNet152V2 compte 60,4 millions de paramètre ce qui est très supérieurs aux deux autres modèles testés jusqu'à présent. Il a un profondeur intermédiaire avec 307 couches de profondeurs. Ce modèle a aussi été pré-entrainé sur ImageNet.

4.4.1 Performances

Nous avons avec ce modèle obtenu les performances suivantes :

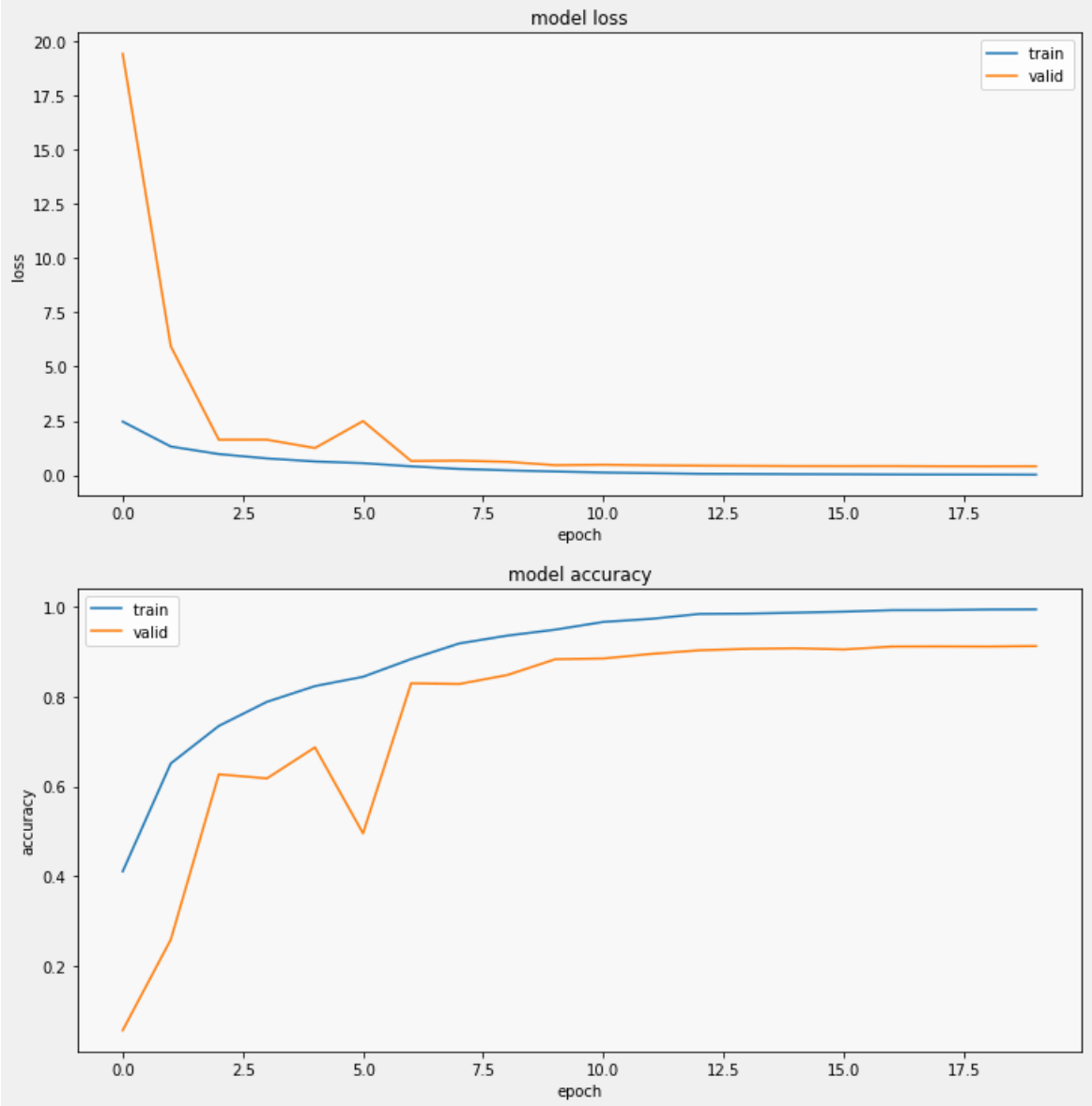

FIGURE 5 – Performances du modèle ResNet152V2

Le modèle a convergé à un taux d'accuracy aux alentours de 91% (sur le jeu de validation) pour une durée d'apprentissage de 20 minutes et 53 secondes sur 20 epochs.

4.5 Modèle EfficientNetB7 & performances

Le modèle EfficientNetB7 compte 66,7 millions de paramètre ce qui est très supérieurs aux deux autres modèles. Il a un profondeur importante avec 438 couches de profondeurs. Ce modèle nous a permis de dépasser le plafond des 95% d'accuracy sur le jeu de test en nous plaçant dans les 30 premières places. Ce modèle a été pré-entrainé sur le jeu de données "noisy-student". Pour utiliser ce modèle avec ce jeu de données, nous avons eu besoin d'installer le package efficientnet, le jeu de données noisy-student n'étant pas présent lors de l'appel de modèle via Keras.

4.5.1 Performances

On a obtenu les performances suivantes :

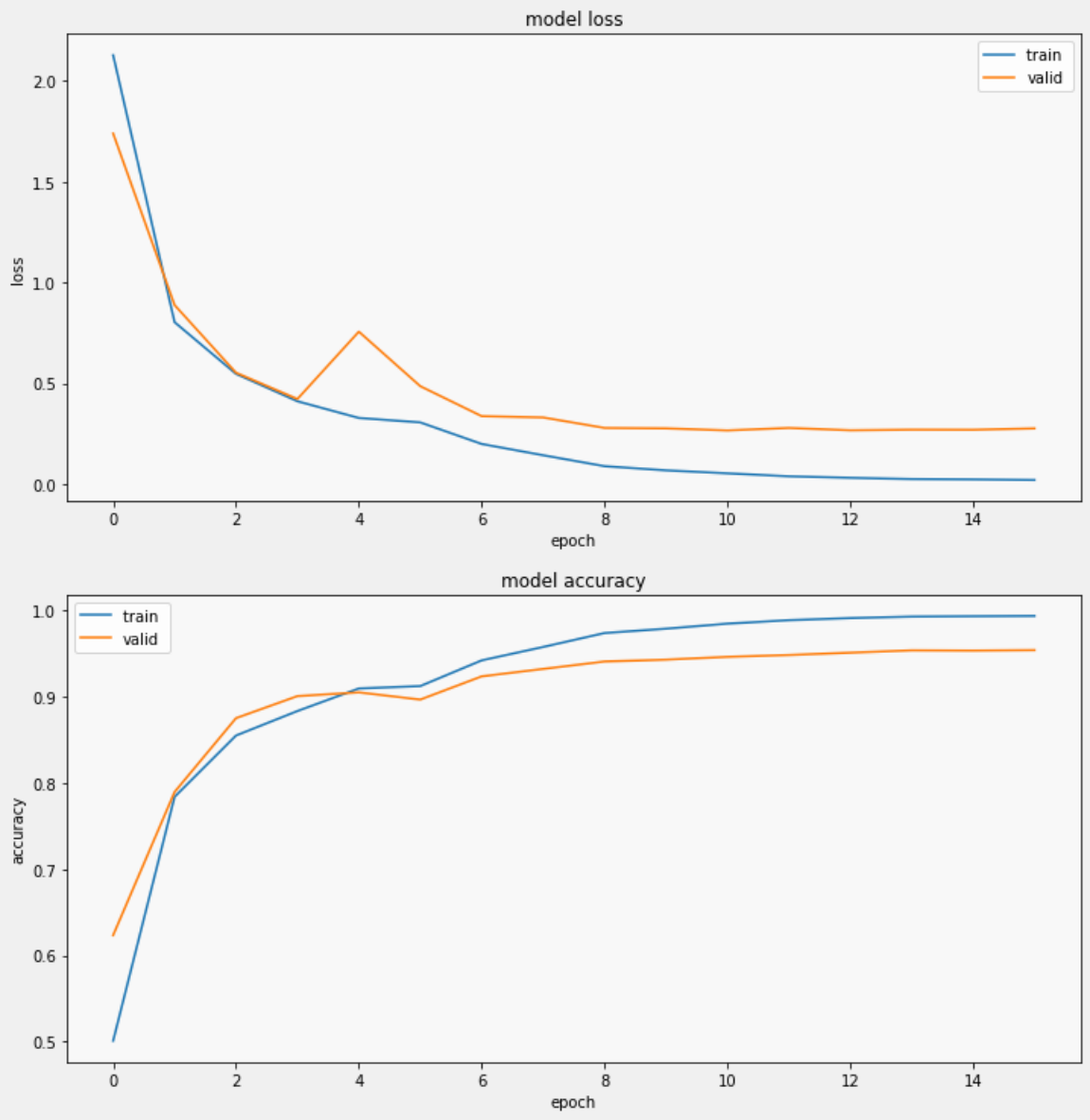

FIGURE 6 – Performances du modèle EfficientNetB7

Le modèle a convergé à un taux d'accuracy aux alentours de 95% (sur le jeu de validation)

pour une durée d'apprentissage de 28 minutes et 26 secondes sur 16 epochs. On obtient ici des performances supérieures pour un temps d'apprentissage allongé.

4.6 Modèle EfficientNetV2XL & performances

Nous avons enfin souhaité testé un modèle plus récent et qui pouvait à priori être plus performant. L'inconvénient de ce modèle à été sa non présence sur Keras ou dans la bibliothèque efficientnet. Nous avons alors été amené à le télécharger directement sur le github suivant : https ://github.com/leondgarse/keras_efficientnet_v2. Nous avons ensuite utiliser la fonction load_model pour charger nos poids et utiliser le modèle. Celui-ci a été entrainé sur le ImageNet21k-ft1k. Il compte 206,8 millions de paramètres, ce qui est très supérieur à tous les autres modèles utilisés jusqu'alors.

4.6.1 Performances

Nous avons obtenu les performances suivantes :

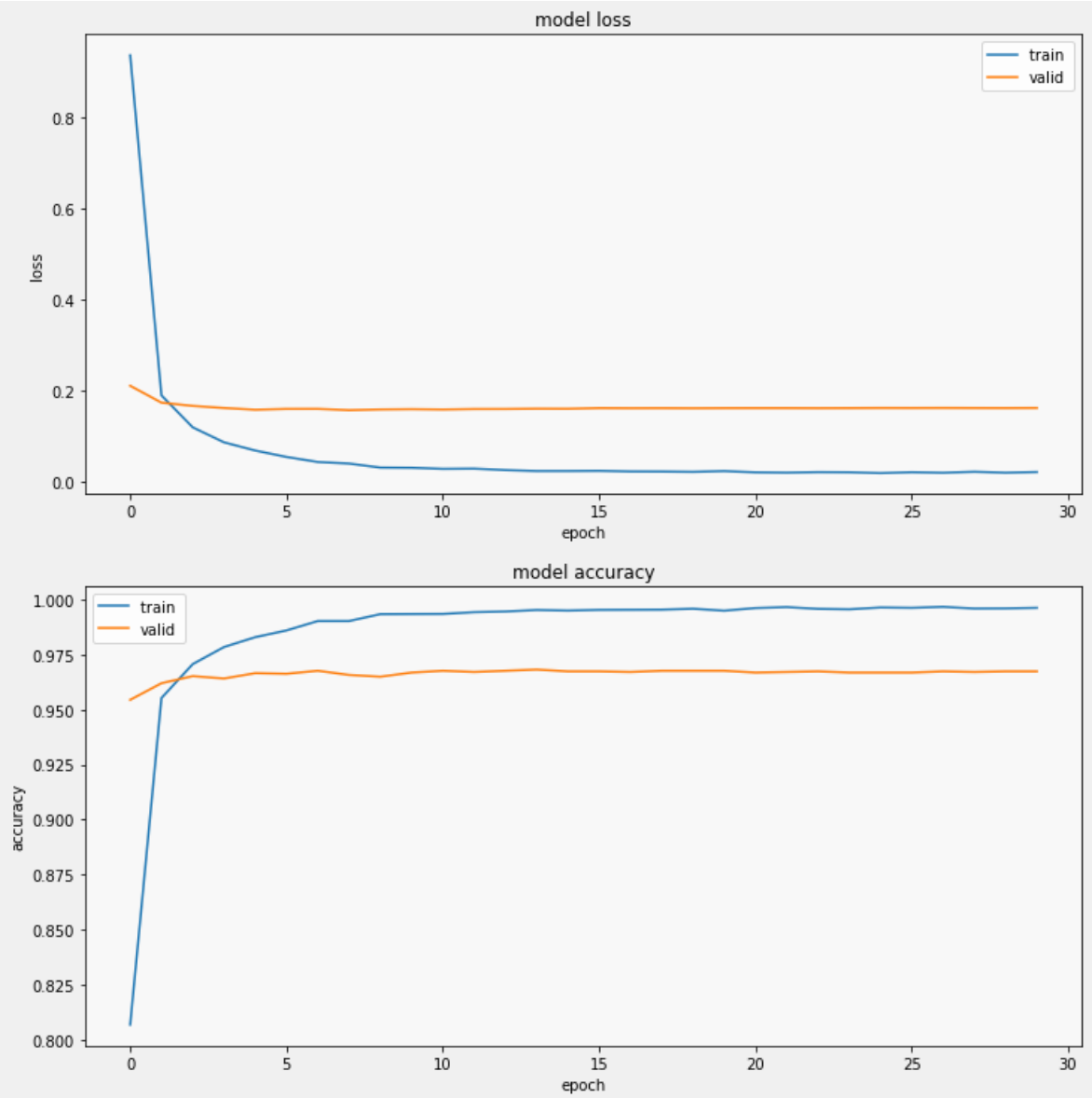

FIGURE 7 – Performances du modèle EfficientNetV2XL

Ce modèle nous a permis de dépasser la barre des 95% d'accuracy sur le jeu de données de test et 96% sur le jeu de données de validation. C'est le modèle retenu pour notre projet, et qui nous a permis d'obtenir le meilleur classement. Le temps d'apprentissage de ce modèle à été de 20 minutes et 52 secondes. Le temps d'apprentissage a été plus court, car dans ce cas, nous avons fixé les poids du modèle, ce qui a considérablement réduit le temps d'apprentissage, sans altérer les performances (nous avons testé d'ouvrir la modification sur les autres paramètres, mais nous avons ici constaté une perte importante de performance et une tendance au sur apprentissage, même avec ajout de couches de dropout).

4.7 Récapitulatif des performances

Nous récapitulons nos résultats dans le tableau suivant :

Modèle	Loss (validation)	Accuracy (validation)	Temps moyen/epochs	Ré-entraînable
DenseNet201	0.2671	0.9434	1 min 8 sec	Oui
Xception	0.2928	0.9418	56,3 sec	Oui
ResNet152V2	0.4193	0.9133	1 min 2 sec	Oui
EfficientNetB7	0.2769	0.9542	1 min 47 sec	Oui
EfficientNetV2XL	0.1642	0.9650	1 min 3 sec	Non

TABLE 1 – Tableau récapitulatif des performances

4.8 Différences de durées entre GPU et TPU

Nous regrettons ne pas avoir eu suffisamment de temps pour testé sur l'ensemble de nos modèles la différences de performances entre les GPU et TPU mis à disposition sur la plateforme. Néanmoins, nous avons pu mener quelques expériences nous permettant d'apprécier la différence sur la durée d'apprentissage.
Nous avons utilisé les GPU sur les images de tailles 224x224 (les GPU ne possédant pas suffisamment de mémoire pour travailler avec les images 512x512). La durée d'apprentissage moyen sur une epoch était ici de 220s soit près de 4 fois plus de temps pour des images deux fois plus petites. La performance d'un TPU est donc près de 8 fois supérieure à un GPU.

5 Conclusion

Ce projet nous a introduit aux compétitions kaggle. Pour mener à bien ce projet, nous avons repris un travail que nous avons complété et amélioré. Nous avons pu nous replonger sur les problématiques de classification d'images et avons implémenté des modèles de convolution. Nous avons effectué 15 dépôts sur la plateforme. Nous avons obtenu un score final de 95,89% et la 12ème position. Nous gardons comme piste d'amélioration l'éventualité de moyenner les sorties des différents modèles, ce procédé a été bénéfique dans certains cas et pourrait ouvrir une voie vers plus de performance.

Références

[1] In : (). URL : https://cloud.google.com/tpu/docs/tpus.

[2] In : (). URL : https://fr.wikipedia.org/wiki/Tensor_Processing_Unit.