Deep Learning

Multiclassification images

Projet

Deep Learning Challenge - Kaggle

Rapport

Auteurs :

Fabien Roussel

16 novembre 2019

Table des matières

1 Introduction

Lors de nos TP de Deep Learning, nous avons appris à créer des réseaux de neurones d'abord sans puis avec Keras. Nous avons également travaillé sur des données numériques, des images ou encore des phrases.

Nous avons donc appris assez pour commencer à nous challenger en réalisant le challenge **ECE Deep Learning**. Nous devrons classifier des images d'animaux (15 types ou classes d'animaux différents). Nous travaillerons sur un total de 7200 images pour le training et 1800 pour le test.

1.1 Objectifs

L'objectif du projet est simple. Je souhaite utiliser plusieurs modèles différents pour répondre au problème et tenter d'atteindre la meilleure accuracy possible du groupe de l'OA DataScience et Deep Learning.

2 Construction d'un modèle

2.1 Etape préliminaire

En amont de la construction d'un modèle, il est nécessaire de commencer par charger les données. En effet, si les données ne sont pas correctement chargées dans la dataframe alors nous n'aurons aucune chance que notre modèle fonctionne. Les modules de chargements de données ont donc été déclarés au début du code. Les *true* prédictions sont déjà correctement chargées grâce au fichier csv. Ensuite, on s'aperçoit que les images ne se chargent pas dans l'ordre et qu'il faut donc les trier avec un ordre alphanumérique. On peut ensuite vérifier que l'élément `i` de notre `X_train` correspond bien à l'élément `i` de `Y_train`. Cette étape étant réalisée on peut commencer à nous interesser à la création du modèle.

Mais avant même de nous interesser à la construction de nos layers, il faut réfléchir aux paramètres globale de notre modèle. Quel est l'objectif ? Que classifie-t-on ? Quel modèle est le plus approprié ? Les réponses à ces questions sont triviales. Nous avons des images que nous souhaitons classifier selon plusieurs classes (une image doit être associée à une unique classe (celle qui a la probabilité la plus haute) à la fin car les images ne contiennent qu'une classe d'animal). Nous devons donc préparer nos données avec `OneHotEncoder` qui va permettre d'avoir un array d'array de true label. Cependant, j'ai pense qu'il serait plus simple de réaliser avec un simple `LabelEncoder` qui, au lieu de créer des array de X entier pour chaque label,

va simplement associé au label un nombre entre 0 et X inclus. Ces deux techniques donneront le même résultat ultimement. Enfin le `LabelEncoder` nous servira pour formater nos prédictions sur le test dans un .csv avec une transformation inverse.

Nous pouvons désormais passer au coeur de la conception de notre modèle.

Je dois préciser que pour la réalisation de ce travail, j'ai réalisé **quatre** kernels différents.

- Le premier reprend mon premier modèle et représente une première vision de la réalisation du projet. Il est là titre facultatif.
- Le second kernel est une version aboutit du projet, Il présente toutes les améliorations qui ont été réalisées et donne la meilleure accuracy pour le test. **C'est celui qu'il faut garder s'il ne fallait en retenir qu'un.**
- Le troisième Kernel correspond à l'usage du *Transfert learning* et en particulier *VGG-19*.
- Et enfin un quatrième Kernel reprenant l'*assembling*.

L'objectif ce tous ces kernels est de montrer le travail qui été fourni et les tests qui ont été fait.

2.2 Comment obtenir une bonne accuracy ?

Nous avons choisi de répartir en 80%-20% les données entre le train et la validation. Dans le cadre du projet, j'ai fait le choix de partir sur un modèle simple que je tâcherai d'améliorer au long du projet.

Le tout premier modèle m'a permis de continuer à paramètrer le projet. En effet dans ce premier modèle, j'ai choisi d'utiliser *ReLu* en activation pour *ConvNet* car cette fonction d'activation est plus appropriée pour notre modèle ne contenant pas énormenent de données et qu'elle permet d'apprendre bien grâce à une probabilité faible que le gradient soit "éffacé" ce qui la rend très interessante pour une multiclassification comme la notre. Un petit point négatif cependant c'est qu'un neurone à qui *ReLu* a attribué un 0, il est peu probable qu'on lui réattribue une valeur positive (mais pas impossible !). Cela est due à la fonction de *ReLu* (= 0 quand devient négative). Nous essayerons de voir ce qu'on peut faire plus tard pour changer cela. On choisit abritrairement un *kernel* (3,3) pour commencer. De façon évidente on choisit (32,32,3) en taille de l'image d'entrée car nous n'avons que des images de taille (32,32) et RGB. J'ai également choisi après ça de réaliser un *MaxPooling* pour extraire de nouvelles features et essayer d'apprendre en fittant plus sur la data. Ensuite j'ai essayé d'extraire de la même façon les nouvelles features (kernel de taille 3, ReLu). J'ai également ici rajouté un *Dropout* de 0.25 pour essayer de ne pas overfitter sur certaines données non pertinentes.

Nous avons besoin de flatten la sortie ainsi que d'appliquer une couche *Dense* de 15 (car 15 classes différentes) et nous choisissons une activation avec une *softmax* ici. Le choix de softmax est simple, nous allons choisir la classe avec celle qui a la probabilité la plus haute de correspondre à l'image et *softmax* permet d'avoir la somme des probabilité égale à 1 (si ce n'était pas le cas, nous risquerions d'avoir plusieurs classe avec la même probabilités ce qui rendrait nos données de sorties inutilisables).

On finalise notre modèle par la sélection de l'*optimizer Adam* d'une *metrics* en *accuracy* et d'une loss qui est calculée avec *sparse_ categorical_ crossentropy* car nous avons une multiclassification (*categorical_ crossentropy*) et nous souhaitons obtenir un nombre entre 0 et 14 inclus (d'où *sparse*). On choisit un batch de 32 pour 20 epochs dans un premier temps. Voici les résutats :

Le tout premier modèle que j'avais fait n'était composé que d'une seule *Conv2D*

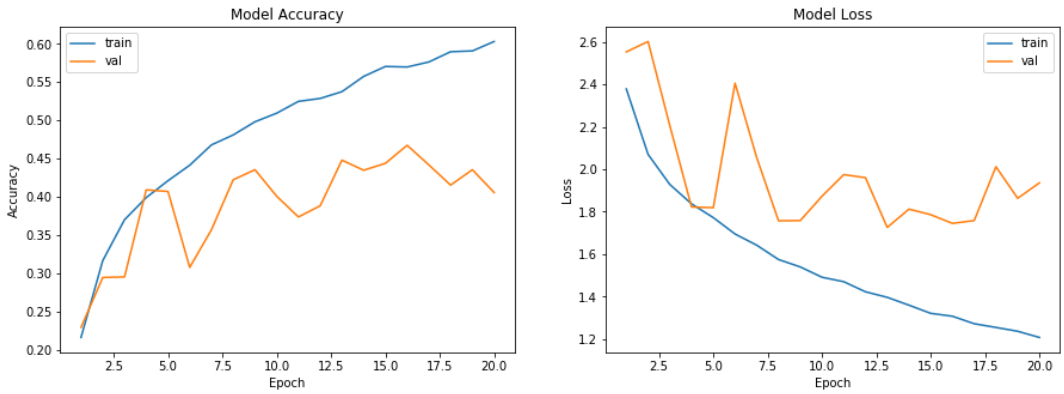

FIGURE 1 – Evolution de l'Accuracy et de la Loss en fonction des Epochs

(celle décrite plus haut) mais n'avait atteint que 35% sur la validation et 23% sur le test (ce modèle n'est pas décrit ici car il est très proche en tout point de celui ci-dessus). L'utilisation de deux couches *Conv2D* m'a donc permis de passer à 43%-44% sur la validation et un peu plus de 40% sur le test, ce qui est satisfaisant pour un premier essai mais pas assez pour s'arrêter ici.

On est encore loin d'un idéal 95% d'accuracy. J'ai donc multiplié par trois la taille du modele actuel avec un dropout croissant et en changeant de fonction d'activation. En parcourant internet, j'ai trouvé une activation *eLu* qui est très similaire à *ReLu* à la différence que dans ses valeurs négatives elle a une légère pente croissante (logaritmique) la rendant interessante pour éviter d'avoir des neurones

qui se désactive (et risque de le rester). Cela nous permet d'atteindre un peu plus de 47% sur la validation au bout de 50 epochs. Toujours pas satisfaisant.

Dans le TP3 nous avions utiliser *ImageDataGenerator* qui permet justement pour des petits dataset de génèrer de nouvelles images pour augmenter l'accuracy. Donc j'ai essayé plusieurs paramêtres de *ImageDataGenerator* et après plusieurs essais j'arrive à atteindre 58% sur le train, ce qui est une augmentation intéressante puisque nous dépassons la baseline.csv.

Avant d'essayer d'utiliser du transfert learning, je voudrais essayer de pousser mon modèle au max en visant 70% si c'est possible. Donc en regardant ce qui se fait sur internet pour des modèles similaire je vois que certains recentre leur modèle. Donc il est peut etre interessant de le faire également avec pourquoi pas une Gaussienne (pour permettre de garder le contrôle du gradient). Donc j'applique la formule suivante avec `erreur` une valeur proche de 0 mais non nulle :

$$X = \frac{X - moyenne}{ecart - type + erreur}$$

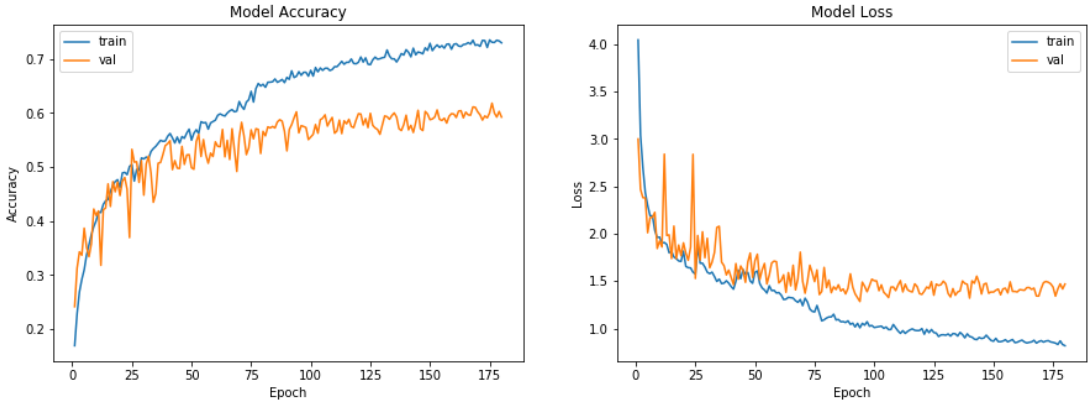

FIGURE 2 – Evolution de l'Accuracy et de la Loss en fonction des Epochs(2)

A la clef une amélioration à peine perceptible mais une courbe qui à l'air de se stabiliser moins vite et moins d'overfit. Cependant on observe toujours rapidement un overfit sur le train (train à 75% alors que la validation est à 55% à 80 epochs). J'ai donc ajouté une régularisation L2 pour essayer de limiter cet overfit. L'effet est immédiat on atteint 70% sur le train et presque 60% sur la validation à 120 epochs.

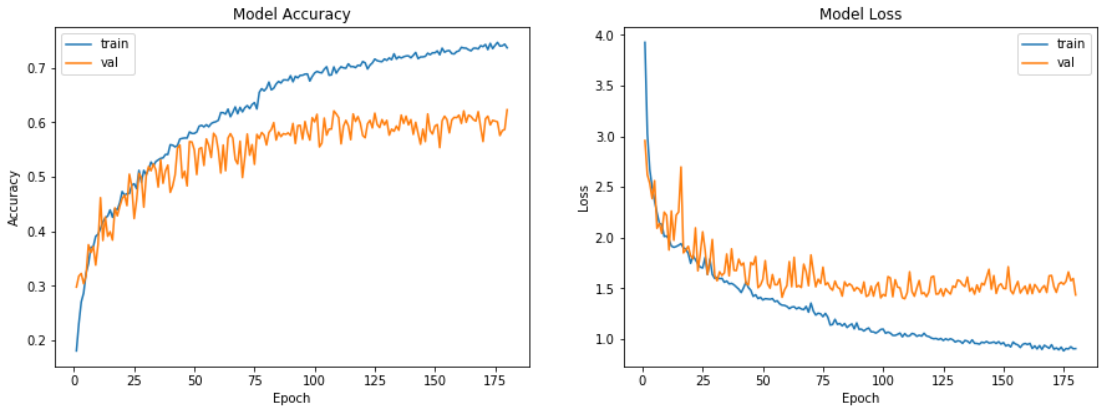

FIGURE 3 – Evolution de l'Accuracy et de la Loss en fonction des Epochs(2)

J'ai créé plusieurs LearningRateScheduler pour améliorer les prédictions en diminuant le learning rate avec l'augmentation des epochs. J'ai *scaler* mes *inputs* pour essayer de gagner quelques pourcentages. Cela m'a permit d'atteindre 63% sur la validation et 61% sur les tests. On constate que les améliorations rajouté ne font maintenant qu'un faible changement sur l'accuracy qui peine à monter et la loss à descendre.

On remarque bien ici qu'on arrive proche de nos limite même si je pense qu'il est toujours possible, sans utiliser de transfert learning, d'atteindre 70%. On peut donc essayer de changer la taille des images en input pour essayer d'obtenir plus de features et donc peut-être plus de données intéressantes. On resize donc les images en (64,64,3) et on rajoute une nouvelle couche (*Conv2D* et une *batchnormalization*) à notre réseau de neuronnes. Et là on note que notre modèle fait mieux. En effet on voit qu'il monte jusqu' 66% sur la validation et 64.7% sur le set de test.

J'ai également testé la crossvalidation sur mon modèle décrit ci-dessus afin de voir qu'est ce que cela pourrait donner. J'ai donc réalisé une CrossValidation à 4 folder. Les résultats sont très probants, en effet j'obtiens d'accuracy 80% sur la validation. On peut observer une accuracy de 68% sur le test ce qui me place 5ème du classement. Ce kernel est disponible sous le nom **Roussel_Projet_Deep_Learn_Ensembling**.

J'ai réalisée de l'*Assembling* c'est à dire combiner plusieurs modèles que j'aurais entrainé au préalable. J'ai choisi cinq modèle au total dont un que j'avoue avoir trouvé sur internet. Je l'ai légèrement modifié mais ce individuellement son score le plaçait en dernier. Il y a également un modèle sur les cinq que j'ai dé-

cidé de ne pas garder pour l'assembling dans un soucis de temps. Enfin les trois autres modèles sont relativement différents (avec une structure commune). Malheureusement, l'entrainement a pris un temps que j'avais sous-estimée et je n'ai pas pu rendre à temps le submissions.csv. Ce kernel est disponible sous le nom **Rousel_Fabien_Projet_Deep_Learning**.

Enfin, je me suis également lancé dans le *Transfer Learning*, j'ai regardé ce qui ce faisait et j'ai essayé de trouver le plus adapater. Donc j'ai choisi VGG-19, mais j'ai décidé de tester également à coté, deux autre modèles : VGG-16 et Xception. Les résultats les meilleurs étaient pour VGG-19 avec 75% d'accuracy sur la validation mais seulement 50% sur le test... Donc je n'ai gardé aucun des trois. Ils sont cependant disponible dans mon kernel **Roussel_Projet_Deep_Learn_Transfert_Learning**. Si j'avais eu plus de temps, j'aurais tenté de faire de l'*Assembling* sur ces modèles.

2.3 Analyse, tentatives d'améliorations et difficultés rencontrées

Tout d'abord, je me suis aperçu que certains changements n'avaient plus d'effet lorsqu'ils étaient couplés avec d'autres améliorations. Par exemple la régularisation *L2* n'affectait pas l'accuracy finale au bout de 150 epochs dans notre modèle finale. Certe elle diminue l'accuracy (l'overfitting) sur le train de 2 ou 3 % mais la validation ne s'en voit pas affectée. Cela peut paraitre logique puisque finalement on finit par tendre par une valeur d'accuracy pour un certain modèle, cependant j'ai trouvé interessant de tester si certaines améliorations ne pouvaient pas être incompatibles avec d'autres.

Par ailleurs, au fur et à mesure des versions réalisées, j'ai trouvé que certaines classes en particulier étaient mal catégorifiées (surtout le léopard) donc j'ai décidé d'afficher une *confusion matrix* (voir annexe 4.1). Celle-ci a bien confirmée mon intuition et m'a permis de réfléchir à comment résoudre le problème. Pourquoi pas faire plus d'*image data generation* sur les classes léopard, écureil, requin et dophin. Cependant, j'ai préféré privilégier la réalisation de nouveaux modèles (décrit plus haut) afin de complèter mes connaissances en Deep Learning.

3 Conclusion

Mon objectif personnel d'atteindre 70% d'accuracy n'est malheureusement pas atteint. Cependant j'ai réussi à implémenter 4 modèles distincts pour la réalisation de ce projet : un modèle unique, un modèle unique *cross-validé*, du Transfert

Learning et de l'Assembling. Ainsi, je pense que d'un point de vue compétences, **étant un débutant en machine learning et en deep learning**, mon objectif personnel est largement atteint.

3.1 Axes d'amélioration

En amélioration, j'aurais aimé pouvoir combiné l'ensemble des techniques vues c'est-à-dire, réalisée de l'*assembling* avec un algorithme de *Transfer Learning*, mon modèle actuel cross validé et deux autres modèles, peut-être un de *Transfer Learning* et un modèle également implémenté par moi-même. Ensuite on pourrait essayer de regarder des indicateurs comme la matrice de confusion afin de voir qu'est ce qui est perfectible dans notre modèle.

On peut voir que mon score finale est de 68% cependant, j'ai oublié de mettre le score en FinalScore donc il est possible que vous ne le voyez pas.

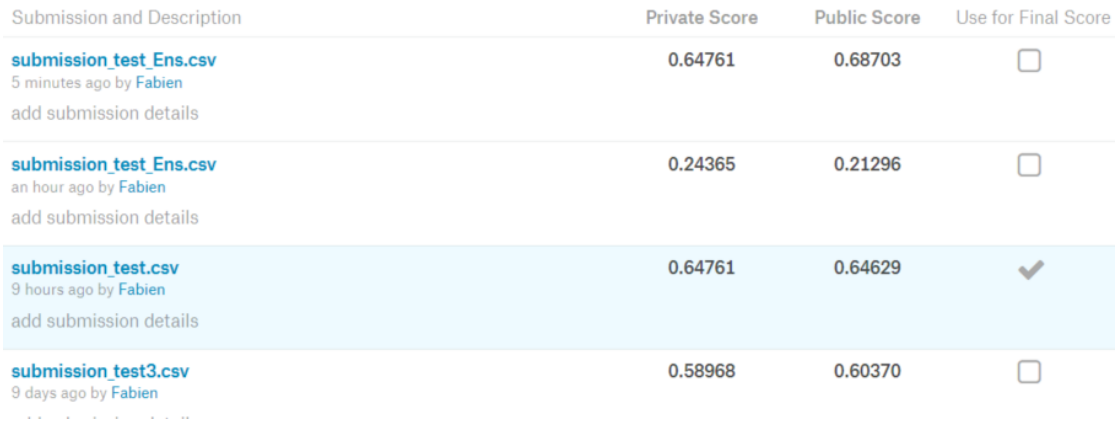

FIGURE 4 – Score finale

4 Annexes

4.1 Matrice de Confusion

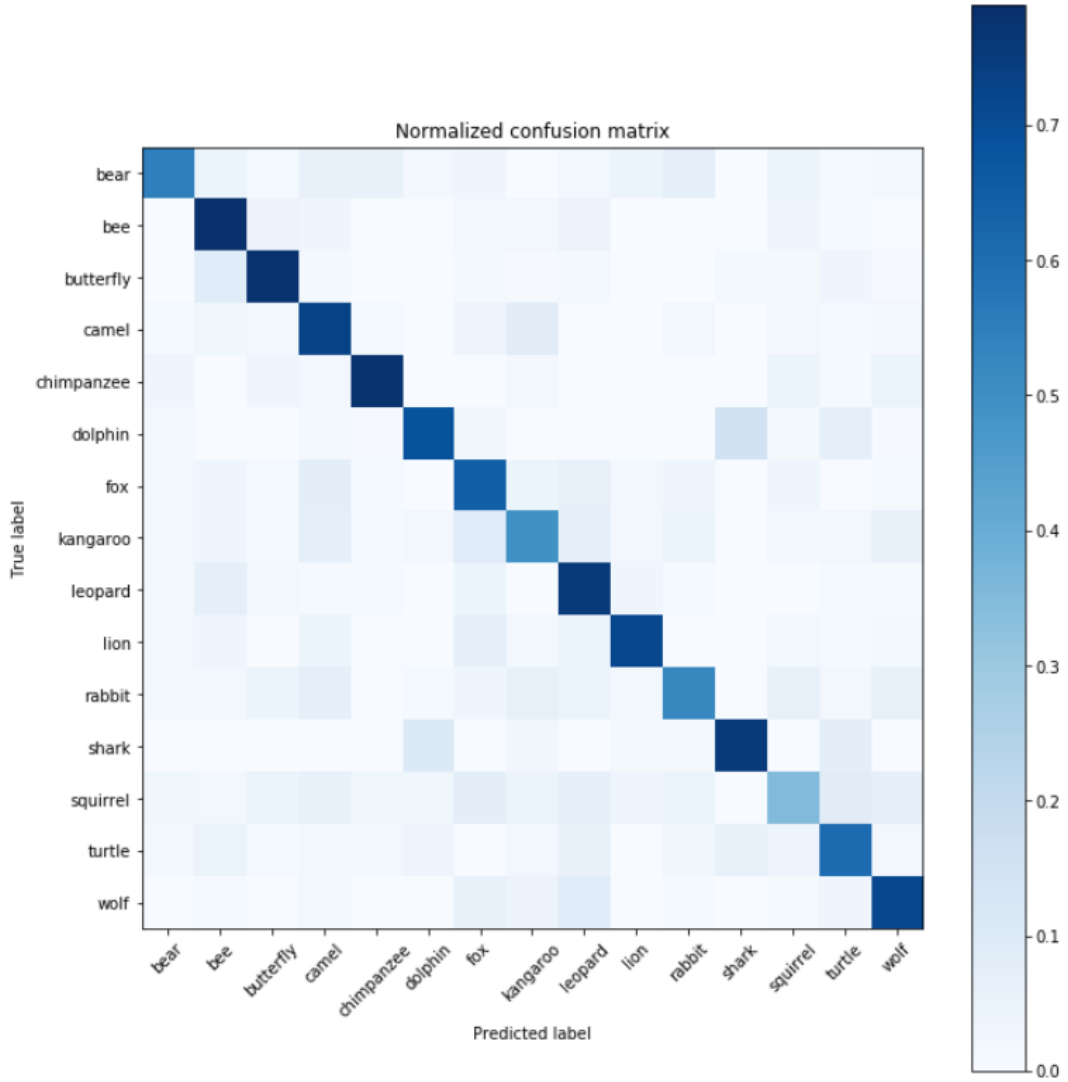

FIGURE 5 – Matrice de confusion pour le modèle le plus pertinent