RCP209 : Stanford Dogs

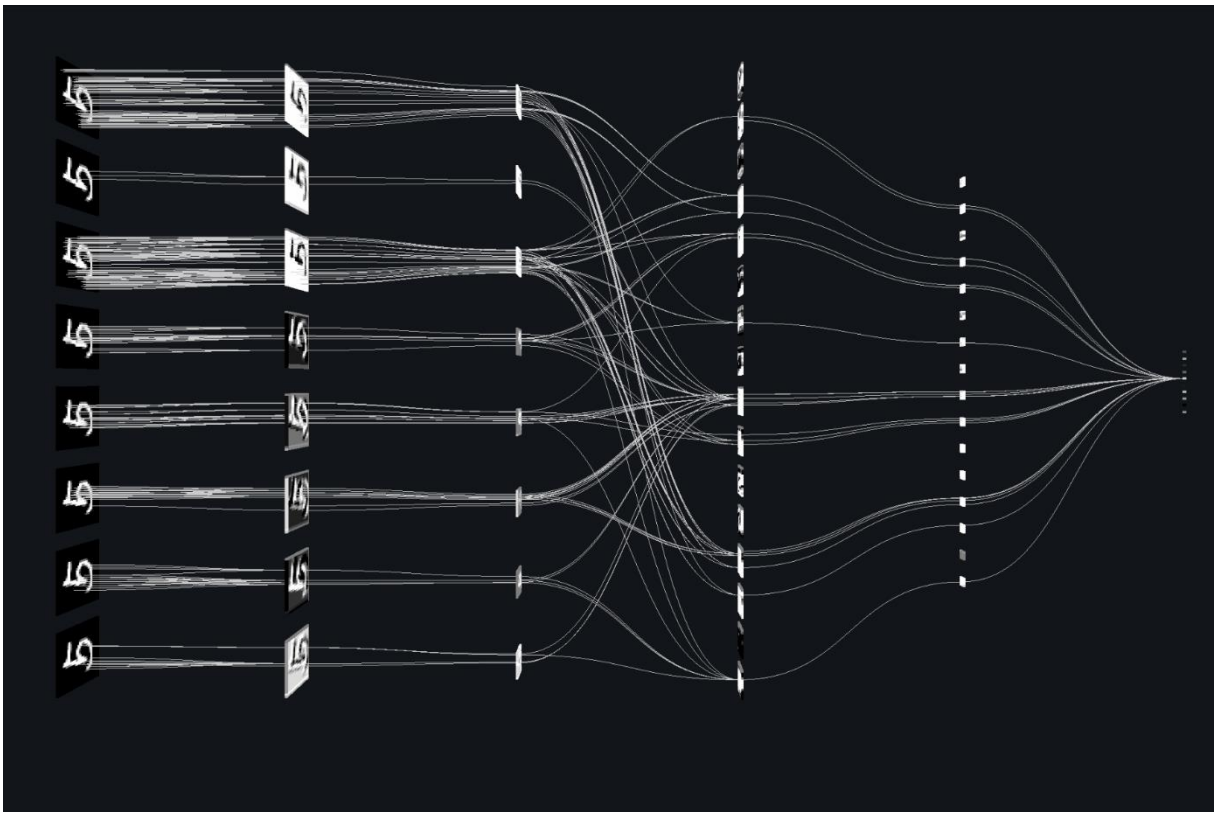


Table des matières

I- Présentation du projet

L'objectif de ce projet est la classification de 120 races de chiens. Pour mener à bout ce projet, j'ai à ma disposition un dataset : « Stanford dogs dataset ». Ce dataset contient 20580 images, soit environ 150 par race de chien, donc le nombre de données par classes suit une loi de distribution uniforme. Il contient aussi un fichier d'annotations permettant de connaître des informations supplémentaires sur chaque image.

II- Les objectifs

L'objectifs principal à atteindre est d'avoir un modèle capable de classer correctement les 120 races de chien présentent dans le dataset.

Afin d'atteindre cet objectif, je souhaite dans un premier temps, je veux avoir un taux de prédiction convenable quant à la classification des différentes images de chien en fonction de leur race sur les données de test, taux à atteindre : est de 80%. J'ai par conséquent dû effectuer plusieurs tâches afin d'optimiser la prédiction. Ce qui implique de réaliser un prétraitement sur les images, choisir le modèle à utiliser et ses paramètres et enfin interpréter les résultats. Une fois cet objectif atteint, le suivant est de comparer plusieurs modèles Keras (Application Keras si on suit la documentation fournie) afin de déterminer laquelle est la plus performante.

III- Éléments constituant ce dossier

Tout d'abord, vous trouverez mon compte-rendu du projet expliquant la réflexion et les éléments qui m'ont permis d'obtenir mes résultats. Ensuite, vous trouverez une archive contenant mon code. Cette archive est aussi disponible sur mon Github WGorag. A l'intérieur, des notebooks qui détaillent l'entraînement et l'évaluation des modèles choisis. Vous y trouverez aussi un Readme.md qui vous explique comment installer le projet sur votre machine et lancer le code.

IV- Prétraitement

Après une vérification visuelle des données, j'ai pu constater que la plupart des images qui nous sont fournies sont d'une qualité suffisante à mon sens pour en tirer de l'information. La résolution est bonne, les images ne sont pas floues, elles sont labellisées, on y distingue correctement les chiens. J'ai pu voir que nous y trouvons des chiens comme des chiots dans certaines classes, mais ceci me semble logique, car notre modèle doit pouvoir classifier un chien de tout âge. En consultant le dataset, j'ai aussi constaté que sur certaine image, on pouvait voir plusieurs chiens de la même espèce sur la même image. Comme dit précédemment, j'avais à ma disposition un dataset d'annotation concernant les images, j'ai vu que celui-ci décrit les coordonnées du ou des chiens sur les images. J'ai donc pu me construire un deuxième dataset « nettoyé » contenant uniquement des images de chien sans les informations superflues. Mais aussi par la même occasion augmenter la taille de mon dataset en séparant les images avec plusieurs chiens en d'autres images. Donc j'ai maintenant deux datasets, ce qui me permettra de les comparer au moment où j'évaluerai mon modèle.

Dans un second temps, j'ai vu que les données ne sont pas séparées afin d'avoir des datasets distincts en entraînement, test et validation. J'ai donc fait le nécessaire pour que cela soit possible ainsi 70 % des données sont consacrées à l'entraînement, 25 % aux tests et 5 % à la validation. J'ai fait ce choix car avoir une grande variété de données pour l'apprentissage du modèle est important pour éviter le sur apprentissage, et vu le nombre faible d'image à notre disposition, j'ai dû consacrer une grande partie de ses données à l'entraînement.

Pour continuer, comme écrit au-dessus, je ne dispose que d'un peu plus de 20 000 images, ce qui fait assez peu de variété de données pour détecter les 120 races de manière automatique. J'ai donc fait le choix d'augmenter de manière artificielle mes données. Pour ce faire, j'ai réalisé trois prétraitements, le premier et le deuxième sont assez similaire, il consiste à faire des rotations aléatoires sur les images dans le seul but d'en générer plus. Le premier fait des rotations sur l'axe horizontal de l'image alors que le deuxième fais rotation aléatoire selon un facteur. Le troisième quant à lui est plus important, car il permet d'augmenter ou de diminuer les niveaux de luminosité sur une image de manière aléatoire, ce qui permettra au modèle de s'entraîner sur une variété de situations plus vaste concernant l'éclairage. Ces étapes de prétraitement sont incluses directement dans le modèle mais ne sont exécutées que lors de la phase d'entraînement.

J'ai dû aussi redimensionner toutes mes images dans une taille fixe qui est 224x224, j'ai choisi cette taille car les application Keras que j'utilise exige ce format. Cela est important notamment pour l'utilisation des deep features qui nécessitent une certaine taille d'image pour fonctionner mais aussi que toute les images d'entrée d'un réseau neuronal convolutif doivent faire la même taille. Cette étape de redimensionnement est effectuée à l'extérieur du modèle, ce qui oblige le formatage des images manuellement avant de les envoyer au modèle pour la prédiction. Dans le même objectif d'utilisation des deep features, j'ai utilisé une fonction de préprocessing fournie par Keras en fonction de l'application choisie pour préparer les données à être utilisées par le modèle, pour notre cas, elle va faire passer les valeurs des pixels d'un intervalle de [0 ;255] à un autre de [-1 ; 1].

Une autre étape de prétraitement est réalisée sur le nom des classes, l'objectif est de représenter chaque classe par un vecteur one-hot. Cette étape est importante, car c'est ce vecteur qui va être utilisé pour calculer la précision par catégorie mais aussi l'accuracy globale du modèle.

Enfin la dernière étape de prétraitement des données, est une étape de normalisation qui consiste à flouter légèrement au même niveau toutes nos images lorsqu'elles passent dans le réseau. Ce qui permettra au modèle d'apprendre à gérer des images dans le cas où on distingue moins nettement le chien. Cette étape de normalisation est aussi bien réalisée lors de l'entraînement du modèle que lors de son utilisation.

V- Augmenter l'efficacité de l'entraînement

Pour commencer, j'ai mis en place un batch de taille 32, ce qui permet un entraînement parallélisé tout en gardant une bonne efficacité dans l'estimation des gradients qui sera mis à jour assez régulièrement. La valeur de 32 a été choisie, car c'est celle recommandée dans la documentation et qu'après en avoir testé plusieurs, c'est cette valeur qui m'a convenu le plus en me basant sur un ratio temps entraînement par l'accuracy de test.

Ensuite, j'ai mis en place la prélecture des données ce qui permet de superposer l'entraînement du modèle et en même temps de récupérer le prochain batch qui sera utilisé pour l'entraînement. En mettant la valeur « tf.data.AUTOTUNE » comme paramètre cela

permet à la prélecture d'ajuster dynamiquement le nombre d'éléments à pré extraire à chaque étape de l'apprentissage. La prélecture améliorer la latence et le débit de l'apprentissage causant une hausse d'utilisation de la mémoire.

VI- Les métriques

A- Métrique personnalisée

Lors de l'entraînement du modèle, j'ai pu m'apercevoir que la fonction « fit » utilisée pour entraîner le modèle ne m'affichait que l'accuracy globale du modèle. Je me suis dit que dans l'hypothèse où uniquement certaines races de chiens feraient baisser la précision globale, je n'ai aucun élément permettant de constater cela. J'ai donc mis en place une métrique personnalisée que j'appelle durant l'entraînement de mon modèle qui me permettra de m'afficher la précision issue des données de validation pour chacune des races prédites sur mes données de validation. Je discuterais de ses données lors de l'évaluation des résultats en fin de rapport.

Dans le même objectif d'avoir plus de précision sur les éléments problématiques, je me suis fait une nouvelle métrique qui me permet de savoir exactement quelle race de chien prédit mon modèle à la place de celle qu'il aurait dû prédire. Je me sers alors de la matrice de confusion et en extrait chaque ligne qui représente le résultat de toutes les prédictions pour une race. Ensuite, en retravaillant les données, j'obtiens les races en erreur. J'ai pu tirer des conclusions de cette métrique que je vous présenterai plus tard.

Les résultats obtenus par ces métriques seront analysés dans la partie IX-A.

B- Métrique d'évaluation

La métrique que j'ai choisi d'utiliser pour calculer l'accuracy de mon modèle lors de son entraînement est la « CategoricalAccuracy ». Choisir cette métrique me semble évident quand on prend en compte que notre problème de classification concerne plusieurs races de chien donc plusieurs catégories. La fonction de perte que j'ai utilisée pour l'entraînement de mon modèle est la « CategoricalCrossentropy ». On sait que lors d'un problème de classification, le mieux, est d'utiliser la « crossentropy » et étant dans un problème de classification

multiclasse, il va de soi d'utiliser une fonction qui permet de calculer la « crossentropy » sur plusieurs classes.

VII- Traitement

A- Choix de la technique

J'ai à ma disposition trois techniques me permettant d'entraîner un modèle permettant la reconnaissance de la race d'un chien à partir d'une image :

- Créer mon propre réseau neuronal convolutif (CNN)
- Mettre en place un réseau avec du transfert learning
- Mettre en place un réseau avec du finetunning

Vu le peu de donnée d'apprentissage, mettre en place un réseau à partir de zéro ne me semble pas une solution envisageable, néanmoins, par curiosité, j'ai mis en place cette solution pour la tester. Donc, la solution sera d'utiliser une application en transfert learning qui permet de générer un modèle performant tout en ayant un petit nombre de données. Je pourrais ainsi ensuite l'améliorer en utilisant le finetunning.

B- Un modèle en partant de zéro

J'ai mis en place un CNN en partant de zéro dans un premier temps pour voir quels sont les résultats que je pouvais obtenir avec le jeu de données que j'ai à ma disposition. Comme les différents exemples que nous avons vu en cours, entrainer un CNN avec peu de donnée ne nous permet pas d'avoir de résultat satisfaisant, j'ai été curieux. Mon cas n'a pas fait exception en plus d'avoir eu une accuracy assez base, j'ai aussi eu un modèle qui a eu tendance à surapprendre du fait qu'il voyait souvent les mêmes données. J'ai donc eu la confirmation que cette solution n'est pas la bonne pour ce dataset.

C- Transfert learning

J'ai alors commencé à mettre en place une solution permettant de générer un modèle se basant sur la méthode du transfert learning. Dans un premier temps, j'ai dû choisir quelle application de Keras utiliser pour mon problème de classification. En me documentant, sur la documentation de Keras, j'ai choisi d'utiliser ResNet50V2 car parmi toutes les applications

proposées, je trouve que celle-ci permet d'avoir un bon équilibre entre la taille du modèle, l'accuracy et la vitesse de prédiction. De plus, nous avons traité des réseaux résiduels lors des cours et je sais par conséquent qu'ils jouissent d'une bonne réputation. Mais encore, avoir une application ayant de bons résultats en accuracy, mais aussi entraînable rapidement me permet de faire de nombreux tests et ainsi améliorer mon modèle afin d'affiner l'accuracy des résultats.

Aussi, afin d'éviter le surapprentissage j'ai mis en place du earlystopping lors de l'entraînement de mon modèle qui coupe l'entraînement dans le cas où la valeur de la fonction de perte sur les données de validation ne diminue pas significativement durant 5 époques. Par défaut, toute diminution de la loss est définie comme significative, c'est ce paramètre que je choisis de garder dans un premier temps, on reviendra dessus plus tard. Pour finir, la fonction restaure les poids du modèle avec lesquels nous avons obtenu les meilleurs résultats.

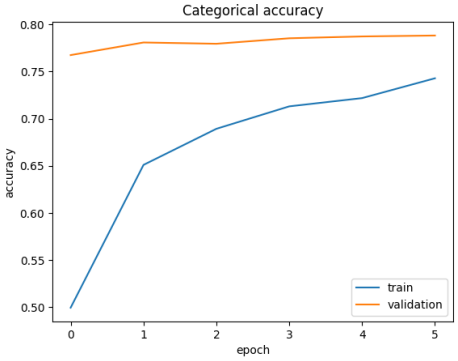
Concernant le choix des poids du réseau que je dois pré charger, j'ai choisi les poids d'imagenet, car en effet, ces poids sont reconnus dans la classification d'images ce qui me semble alors le choix le plus pertinent. Pour l'utilisation de l'application de Keras, je n'ai pas inclus le haut du réseau afin d'y intégrer des couches personnalisées qui seront entraînées afin de prédire les 120 races de chien. J'ai bien sûr bloqué l'entraînement des couches de ResNet50V2 afin de profiter des poids préalablement chargés. Pour entraîner les couches que je vais ajouter, j'ai choisi d'utiliser comme optimiseur pour la descente des gradients « Adam » car j'ai pu trouver dans mes recherches qu'il constitue l'état de l'art dans ce domaine. Au niveau du paramétrage de cet optimiseur j'ai choisi de garder ses paramètres par défaut après une série de tests, soit un learning rate de 0.0001. Je vais vous expliquer la méthode que j'ai mis en place pour choisir mes paramètres. Le premier facteur est que le changement de valeur des paramètres ne permettait pas un gain significatif des performances en comparaison des valeurs par défaut. Mais aussi, comme on a pu le voir en cours ce qui influe le plus sur la qualité de prédiction du modèle est la qualité des données d'entraînement, pour cette raison, j'ai fait le choix de prendre plus de temps à améliorer comme je pouvais la qualité de mes données que passer beaucoup de temps à chercher exactement les paramètres me permettant de gagner quelques dixièmes de pourcent d'accuracy.

Par exemple, j'avais choisi de ne pas prétraiter mes images avec les annotations qui permettaient d'extraire plus précisément les chiens dans l'image. La raison de ce choix était

de garder un maximum du contexte. Avec ce choix, j'obtenais une accuracy de 71% sur mes données de test ce qui commence à être raisonnable mais pas suffisant. Afin d'augmenter ce score, je me suis dit qu'il serait peut-être pertinent de donner des images plus précises à mon modèle, c'est ce que j'ai réalisé en faisant mon prétraitement et en isolant les chiens grâce aux annotations. Cette initiative a permis d'augmenter l'accuracy à près de 78% ce qui est un gap qui n'a jamais était atteint en ajustant les hyperparamètres du modèle. C'est cette constatation qui m'a orienté sur l'importance du prétraitement au détriment de l'hyperparamétrisation.

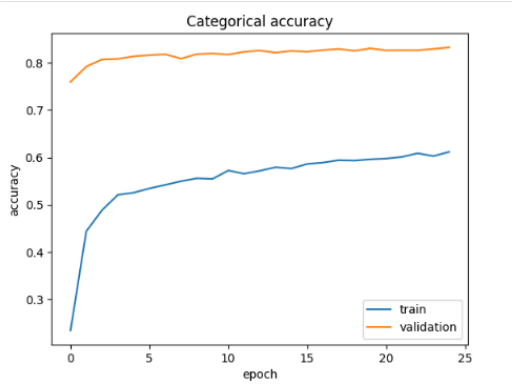
Je vais maintenant vous présenter les couches que j'ai rajouté au modèle afin de pouvoir y appliquer une prédiction. Tout d'abord, nous commençons par une couche de « GlobalAveragePooling2D » pour obtenir un vecteur contenant les informations du tensor précédent, j'ai choisi cette couche au détriment du « GlobalMaxPooling2D » car j'obtiens de meilleurs résultats avec celle-ci. Ensuite, j'ajoute une couche de normalisation qui permet de normaliser la moyenne et l'écart-type des données qu'elle a vu lors de l'entraînement. Ensuite, je continue avec une couche de dropout qui nous permet de limiter encore plus le surapprentissage en évitant que les neurones s'adaptent en fonction de leurs voisins et ainsi assurer une meilleure généralisation du modèle. On termine par une couche « dense » qui est une couche complétement connectée, c'est cette couche qui prédit la race du chien. J'utilise comme fonction d'activation pour cette couche le softmax, car elle permet de mettre en avant la race de chien la plus probable pour le réseau de neurones.

Lors de mes premiers essais avec ces couches, j'ai obtenu une accuracy de test allant jusqu'à 78% ce qui fait une différence significative en comparaison des résultats que j'ai pu obtenir avec un modèle fromscratch. L'utilisation de l'earlystopping a été justifié, car assez tôt, lors de l'entraînement, j'ai pu voir que ma fonction de perte sur les données de validation stagnait alors que celle sur les données d'entraînement diminuait constamment. J'ai donc pu éviter le surapprentissage comme on peut le voir sur le graphique.



J'ai par la suite, cherché quand même à améliorer ce résultat en trouvant une solution qui me permettrait d'améliorer mon accuracy sur les données de test tout en limitant toujours

au moins autant le surapprentissage. Après réflexion, j'ai opté pour l'utilisation d'une couche de dropout qui répond parfaitement à cette problématique. J'ai donc cherché quelle était la meilleure valeur à mettre dans son paramètre « rate ». J'avais mis comme valeur 0.2 en me basant uniquement sur les exemples que j'avais pu trouver lors de mes recherches. Mais en me renseignant un peu plus et en faisant quelques tests de mon côté, j'ai choisi de mettre comme nouvelle valeur 0.8. Ce qui permet d'augmenter la accuracy à 82% sur les données de tests ce qui est un bond significatif. En revanche, le temps d'entraînement est devenu beaucoup plus long, car on a toujours tendance à avoir une fonction de perte sur les données de validation qui diminue légèrement tout au long de l'entraînement ce qui réinitialise à chaque époque le earlystopping. Donc la solution que j'ai mise en place pour limiter le temps d'entraînement tout en gardant des résultats convenables est de dire que tout changement inférieur à la valeur absolue de 0.01 dans la fonction de perte n'est pas considéré comme un changement significatif par le earlystopping. Cette solution m'a paru une bonne idée, car comme on peut le voir sur ce graphe l'accuracy sur les données de validation n'augmente quasiment plus au bout d'un certain temps, donc arrêter l'entraînement plus tôt ne nous fera perdre que très peu d'accuracy. Les résultats obtenus à la suite de la modification de la fonction de earlystoppping sont même meilleurs 83.75% d'accuracy sur les données de tests. Cette différence peut s'expliquer par le fait que d'arrêter l'entraînement plus tôt permets au modèle de mieux généraliser les données, car il s'habitue moins aux données d'apprentissage. Donc cette solution semble être bonne, car l'accuracy du modèle ne diminue pas, et même a tendance à augmenter. Autre point positif, je suis passé de 25 époques d'entraînement à 14 ce qui diminue énormément le temps d'apprentissage du réseau. Au total, j'obtiens un temps d'entraînement d'environ 1h50.



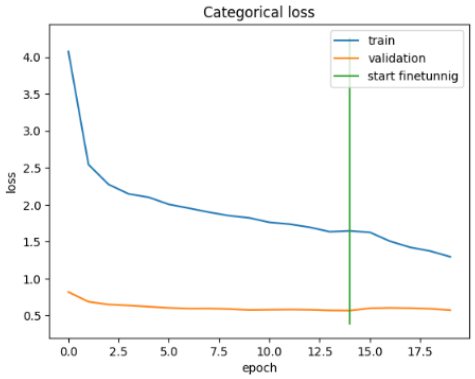
Avec ces résultats, je valide le premier objectif que je m'étais fixé qui était d'avoir plus de 80% d'accuracy sur mes données de tests. Mais je pense qu'il est encore possible d'augmenter l'accuracy en utilisant le finetunning.

D- Finetunning

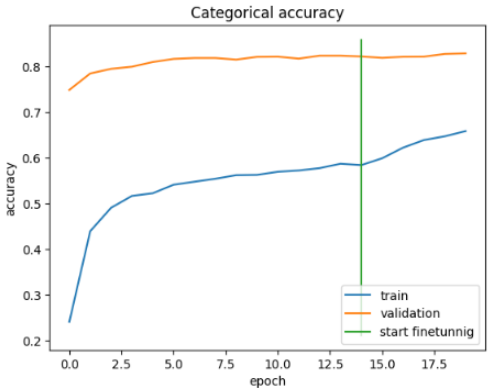
Pour réaliser le finetunning, mon modèle de base est le modèle que je viens de mettre en place en utilisant la méthode de transfert learning simple. L'utilisation du finetunning devrait augmenter les performances du modèle, néanmoins, il y a un fort risque de surapprentissage si on gère mal certains paramètres. Pour éviter le surapprentissage, j'ai modifié 3 paramètres d'entraînement de mon réseau :

- J'ai diminué le nombre d'époques d'entraînement. Je n'ai plus que 5 époques d'entraînement. Ce nombre a été choisi de manière empirique selon les tests que j'ai pu faire. Ce qui évite que mon modèle s'adapte trop aux données sachant qu'en plus, il a déjà parcouru 14 époques sur ces données.
- Il est aussi important de diminuer la valeur du learning rate de la fonction d'optimisation pour éviter d'apprendre trop vite et que le modèle s'adapte trop vite aux données d'entraînement. Pour ce faire j'ai passé la valeur de 0.0001 à 0.0000075 pour l'optimiseur. De la même façon, cette valeur a été choisi de manière empirique.
- Il faut aussi choisir le nombre de couches qui va être affinées. Sachant que les premières couches d'un réseau généralisent bien les concepts et que ce sont les dernières qui sont spécialisées. Par ce principe, et après plusieurs tests, je décide de ne réentraîner uniquement que les 15 dernières couches du modèle à l'exception des couches de « BatchNormalization » qui ne doivent pas être réentraîner. Ce qui représente pour notre modèle 176 variables qui seront affinées.

Lors de mes tests, j'ai pu constater que du fait du risque important de surapprentissage, la fonction de perte sur les données de validation et d'entraînement ne faisait que décroître ce qui rend l'utilisation de l'earlystopping inutile pour limiter le surapprentissage et donc m'a obligé à chercher des valeurs de paramètre plus finement que pour le transfert learning classique.



À la suite de l'entraînement, l'accuracy obtenue sur les données de test est très satisfaisante, car le modèle a gagné 1.5% d'accuracy en passant à 85.25%. En regardant les graphes obtenus lors de l'entraînement sur la fonction de perte et sur l'accuracy, on peut constater visuellement le risque important de surapprentissage. On constate clairement que ces valeurs sur les données d'entraînement changent en s'améliorant rapidement et on comprend bien qu'en faisant tourner l'entraînement du modèle un peu plus longtemps celui-ci se serai sur-adapté aux données.



Pour réaliser le finetunning il faut rajouter 31 minutes d'entraînement ce qui porte sur un temps total d'entraînement du réseau à 2h21min.

Suite à ces différentes étapes d'entraînement, je peux valider mon premier objectif qui était d'obtenir un score de 80% d'accuracy sur les données de tests. Je vais donc par la suite comparer l'entraînement du modèle avec d'autres applications disponible avec Keras.

VIII- Comparaison avec d'autre modèle

Étant donné que mon objectif est de faire une comparaison entre les applications de Keras, j'ai décidé de fixer les paramètres que je vais utiliser. Je vais donc utiliser exactement les mêmes paramètres pour chacune des 3 applications. Comme dit précédemment, la première application utilisée est ResNet50V2, Keras fournit d'autres applications ResNet que j'ai donc choisi de confronter. Les applications confrontées sont ResNet101V2 et ResNet152V2, la différence avec ces deux applications est la profondeur du réseau néanmoins, comme ResNet50V2, ces applications utilisent aussi les couches résiduelles.

Voici un tableau comparatif fourni par Keras qui nous résume les 3 applications :

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6

Lors du test de ces deux nouvelles applications, j'ai eu un souci concernant le finetunning de l'application ResNet152V2 qui me provoquait une erreur : Out Of Memory (OOM). J'ai donc dû pour cette application l'entraîner dans un second temps avec un batch plus petit afin de limiter la place en mémoire.

Avant de comparer les résultats, je vais vous afficher un tableau permettant de voir les résultats de chaque application.

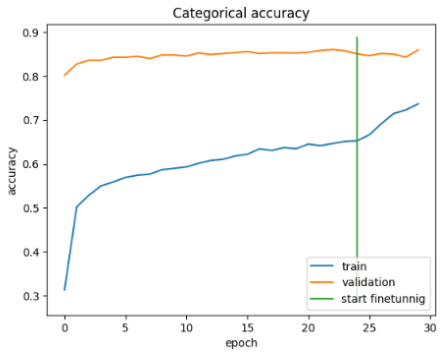
	ResNet50V2	ResNet101V2	ResNet152V2(32)***	ResNet152V2(16)
TF* temps apprentissage	1h50	2h09	2h34	3h06
TF accuracy test	83.75%	86.5%	86.6%	85.5%
FT** temps apprentissage	2h21min	2h48	/	4h05
FT accuracy test	85.25%	87.3%	/	84,5%

*Transfert Learnig **Finetunning *** La valeur entre parenthèse est la taille d'un batch

Le temps d'apprentissage pour le finetunning est le temps total, je prends donc en compte le temps d'apprentissage du transfert learning classique.

Je vais maintenant interpréter ces résultats. Je commence par ResNet152V2(16), au vu de la taille réduite de son batch et son nombre de couches ce modèle est par toute attente le modèle le plus long à entraîner du fait que ses poids sont mis à jour plus fréquemment. Mais nous pouvons aussi constater que c'est celui avec les moins bons résultats au niveau de l'accuracy sur le jeu de test et que celle-ci a même tendance à diminuer lors du finetunning. J'aurai sûrement pu améliorer ses résultats en augmentant un peu le nombre d'époques lors

de l'entraînement mais je m'éloignerais encore des contraintes que je me suis fixées. Puis le risque de surapprentissage augmente grandement à chaque époque quand on fait du finetunning et ce modèle ne fait pas exception quand on voit la trajectoire de la courbe de l'accuracy sur les données d'entraînement qui augmente rapidement, d'autant plus que les poids sont mis à jour plus fréquemment au vu de la taille des batch. De plus, il obtient des résultats que très légèrement supérieurs que ce obtenu par ResNet50V2, cette solution n'est donc pas viable, en tout cas avec les contraintes que je me suis imposé.



Ensuite, je vais traiter du cas de l'application Keras ResNet152V2 avec des batch de 32. Le premier point que je peux constater, c'est que lors du transfert learning classique, c'est avec lui qui j'ai obtenu la meilleure accuracy sur les données de test. Néanmoins, le matériel que j'ai à ma disposition ne me permet pas de faire du finetunning sur cette solution comme expliqué au-dessus. Du fait de cette contrainte, je préfère ne pas utiliser ce modèle, car pour moi l'utilisation du finetunning me semble essentiel pour avoir des résultats maximisés.

Enfin, je vais examiner les résultats de l'application ResNet101V2. On peut constater qu'avec un temps d'apprentissage beaucoup plus rapide celui-ci obtient quasiment la même accuracy que le ResNet152V2(32) ce qui est très convaincant. De plus, en utilisant du finetunning, les résultats s'améliorent de près de 1% ce qui en fait le meilleur modèle que j'ai pu obtenir en terme d'accuracy. Mais aussi, la taille de ce modèle reste raisonnable ce qui me permet de l'entraîner sans problème et de le sauvegarder sans qu'il prenne trop de place sur le disque. Autre point avantageux, il dispose d'un temps d'inférence raisonnable même si un peu plus élevé que ResNet50V2, ce qui est important, car c'est ce qui représente la rapidité des prédictions. Au vu des résultats obtenus, je pense que cette solution est la plus viable selon mes tests. Ainsi, je peux répondre à ma deuxième problématique qui concerne le choix de l'application à choisir.

IX- Les résultats

A- Métrique personnalisée

Comme je l'ai écrit plus tôt j'ai mis en place plusieurs métriques personnalisées. La première me permet d'afficher exactement la précision pour chaque race de chien. Mon hypothèse de base est que certaines races de chiens font baisser la précision générale du modèle. Après analyse, je peux confirmer cette hypothèse. En effet, ce qu'on peut constater dans un premier temps c'est que toutes les races de chien ne sont pas égales quant à la précision obtenue par le modèle et que cette précision varie de manière significative. Certaines races comme les « Scotch terrier » a une précision très bonne, en revanche, d'autres comme les « miniature poodle » ont une précision très mauvaise. J'ai donc cherché à comprendre pourquoi une si mauvaise précision est présente. C'est ainsi, que ma deuxième métrique a été créée (celle-ci a été exécutée sur un modèle moins bien entraîné, car ma machine n'est pas assez puissante pour subir un entraînement complet (entraînement sans finetuning et 5 époques) avec cette métrique, mais ce qui est suffisant pour remarquer une tendance), afin de savoir quelles races de chiens sont prédites à la place. Avec une analyse de cette métrique et une comparaison avec mes images dans le dataset. Je me suis aperçu que les races de chiens dont la précision est mauvaise sont les races qui peuvent facilement être confondues avec d'autres races même par l'homme. De ce fait, cette problématique aurait pu être soulevée sans une analyse apriori, mais je n'y ai pas pensé.

La solution que j'ai à apporter pour résoudre ce problème est de fusionner les races pouvant être confondues ensemble. Par exemple, si je prends l'exemple du « minature poodle », il est en général confondu avec le « toy poodle » ou le « standard poodle », la solution serait alors de les regrouper dans une classe « poodle ». Ainsi, la précision sur cette catégorie augmentera et celle du modèle aussi parallèlement. J'ai choisi de ne pas réaliser ce prétraitement après coup, car le sujet de projet porte sur la classification des 120 races de chiens et donc j'ai gardé comme objectif d'avoir les 120 races.

B- La méthode à choisir

Comme j'ai pu l'annoncer au début du projet mettre en place un réseau neuronal convolutif avec si peu de données c'est avéré être une très mauvaise idée. Nous savons qu'au vu de la littérature actuelle, en général la meilleure façon d'obtenir de bons résultats avec si peu de données nécessite l'utilisation du transfert learning avec du finetunning. C'est aussi ce

qui a pu être vérifié par mon projet car il y a une différente flagrante dans les performances de mon modèle dès que j'ai mis en place cette solution.

Donc la méthode à choisir pour la classification des 120 races de chiens en fonction du jeu de données Stanford Dog est le transfert learning avec finetunning.

En revanche, étonnamment, ce n'est pas avec le réseau le plus profond que j'ai obtenu les meilleurs résultats. Ce qui est contre intuitif, car ce sont eux en général qui généralise le mieux. Mais je pense que cela peut s'expliquer par les conditions d'entraînement qui n'étaient pas optimum : matériel pas assez puissant, règles que je me suis imposées sur le paramétrage, taille du dataset. Je pense que des tests supplémentaires peuvent être faits pour améliorer cette solution.

X- Conclusion

Pour terminer voici ce que je retiens de mon projet, dès le début, j'ai opté pour le transfert learning, au vu de la documentation que j'ai pu consulter et des connaissances que j'ai pu acquérir en cours. Mon choix s'est porté sur ResNet50V2 avec lequel j'ai pu obtenir des résultats convenables à force de recherches, prétraitements et ajustements, ce qui m'a permis de comprendre encore mieux les différents points abordés. J'ai pu aussi constater l'importance qu'a le prétraitement de nos données pour avoir un modèle qui s'entraîne efficacement. En procédant de cette manière, j'ai pu valider mon premier objectif qui était d'obtenir plus de 80% d'accuracy sur mes données de test.

Par la suite, j'ai comparé différentes applications Keras. J'ai choisi de rester dans la même famille d'application, mais aussi de fixer mes paramètres afin d'avoir un point de comparaison commun. Après avoir testés les 3 applications, j'ai pu en déduire laquelle est la plus performante selon mes tests et ainsi répondre à ma deuxième problématique en spécifiant que le meilleur modèle obtenu est ResNet101V2.

Mais même après avoir réalisé ce projet, je suis conscient du travail qui me reste à réaliser. Je compte bien alors continuer à progresser dans un premier temps avec le CNAM, mais par la suite en approfondissant mes connaissances dans un milieu professionnel, tout en pratiquant une veille régulière.

XI- Amélioration et ouverture

Je ne prétends pas avoir trouvé la meilleure solution possible concernant ce problème de classification, néanmoins, j'ai des idées qui pourraient être mises en place qui permettraient peut-être d'améliorer les performances de ce modèle. Une hypothèse qui serait à vérifier est de savoir si la couleur du pelage des chiens influe réellement sur la classification. Pour ce faire, il faudrait ajouter un prétraitement permettant de transformer toutes les images en niveau de gris. La bibliothèque que j'utilise ne me permet pas d'implémenter ça facilement, mais lors de mes recherches, j'ai pu voir que la bibliothèque « Keras_cv » le permet. Donc voici un premier point d'amélioration. Un second point, comme je l'ai évoqué précédemment est de regrouper les classes qui se confondent entre elles pour augmenter l'accuracy globale et la précision des classes. On pourrait aussi essayer d'implémenter un réseau complétement connecté localement qui permettrait peut-être de saisir plus facilement les différences mineures qu'il y a entre les classes qui se ressemblent.

Lors de mes entraînements, j'ai aussi créé un biais lors du transfert learning en n'utilisant que des applications Keras de la même « famille » qui sont les ResNet. Il pourrait aussi être intéressant de tester d'autres modèles différents afin d'avoir une comparaison plus élargie.

On pourrait aussi en amont créer un encodeur/décodeur qui isole automatiquement les chiens sur l'image et ainsi, on donnera à notre modèle uniquement des images de chien isolé ce qui permettrait d'avoir moins d'information parasite.

Enfin, dans le but de mettre en production ce modèle je pense qu'il faudra un modèle de classification binaire qui détecterait si sur l'image il y a un chien ou non. Ce qui éviterait le genre de situation où le modèle donne une race de chien à un humain. Par exemple, quand j'ai envoyé une image de moi au modèle par curiosité, il m'a classifié comme un chihuahua. Voici les pistes d'amélioration et de travail que je vois pour améliorer généralement ce modèle.