TD5 : Intel images - CNN

1. Mise en forme des images

- Décompressez les fichiers *seg_train.zip* et *seg_test.zip*. Installez la bibliothèque *pillow* pour le traitement des images. Cette bibliothèque sera à importer sous le nom *PIL*.
- Explorez l'arborescence des dossiers *seg_train* et de *seg.test*. Quels sont les sous-répertoires présents, a quoi correspondent-ils.
- Combien d'images au format *.jpg* contient chaque sous-répertoire (*train* et *test*). Affichez le nombre d'images de chaque sous répertoires de *seg_train* et de *seg.test*.

  Informations : Utilisez les commandes *os.listdir()* et *glob(pathname="rep/*.jpg")* pour lister les sous-dossiers et pour compter les fichiers *.jpg* présent dans un dossier *rep*.
- La répartition des images entre les différentes classes est-elle équilibrée.
- Créez un dictionnaire associant à chaque type d'image (classe) un entier. Proposez une fonction permettant, à partir d'un entier, de retrouver le nom de la classe correspondante.
- Déterminez la taille de chaque image du jeu de données. Stockez toutes les tailles dans une liste.

  Informations : *plt.imread(file)* permet de lire l'image dans le fichier *file*. L'attribut *size* des images permet d'accéder à sa taille.
- Transformez cette liste en *DataFrame* ou en *Series.* Quelles sont les différentes tailles présentes dans le jeu de données. Combien d'images correspondent à chaque taille.
- Quelle est la taille minimale et la taille maximale des images. Peut-on les utiliser directement dans un modèle de deep learning sans prétraitement.
- Nous souhaitons travailler avec des images couleur de taille normalisée (150, 150). Récupérer dans les listes *X_train* et *y_train*, *X_test* et *y_test*, les images et leur classe associées. Vous transformez les images en

tenseurs via la commande *np.array(img)* avant de les ajouter dans les listes.

Informations : *img=Image.open(file)* de la bibliothèque *PIL* permet de lire une image contenue dans le fichier *file*. *img.resize((w,h))* permet de redimensionner une image à la taille spécifiée.

- Sélectionnez aléatoirement 20 images du jeu d'entraînement et affichez-les.

  Informations : Vous pourrez utiliser pour cela la fonction *randint()* de *numpy.random*.
- Convertissez les listes *X_train* et *X_test* en tableaux *NumPy.* Pourquoi selon vous cette étape est-elle nécessaire pour utiliser un modèle *séquentiel* avec *TensorFlow*.

2. Premier modèle

- Nous souhaitons construire un modèle composé de deux blocs de convolutions suivis d'un réseau de classification. Quelle est la forme de la couche d'entrée, combien de neurones doit contenir la couche de sortie, quelle activation doit être utilisée en sortie.
- Pourquoi est-il nécessaire d'ajouter une couche de normalisation des pixels en entrée. La couche *Rescaling(scale)* permet de normaliser les valeurs. Quelle valeur choisir pour *scale* justifiez votre réponse.
- Proposez un premier bloc composé de deux couches de convolution, avec 32 et 64 filtres. Vous supprimerez le padding et choisirez des matrices de convolution de taille 3. Quelle activation peut-on utiliser.
- Ajoutez à la suite de chacune de ces deux couches une couche *MaxPooling2D()* de taille 2 et une couche *Dropout()* avec un taux de désactivation faible. Quel est le rôle du *MaxPooling*, pourquoi introduire un *Dropout*.
- Ajoutez un bloc composé de 3 couches de convolution avec 64, 128 et 256 filtres, utilisez les mêmes paramètres que ceux du premier bloc.
- Ajoutez de nouveau les couches de *MaxPooling2D()* et *Dropout()* identiques à celle de du bloc précédent. Pourquoi augmente-t-on progressivement le nombre de filtres.

- Bloc de classification : Ajoutez un bloc *dense* composé d'une couche avec 128 neurones et d'une couche de 64 neurones.
- Après chacune de ces couches *Dense* ajoutez une couche *Dropout().* Pourquoi choisir ici un taux de Dropout plus élevé que précédemment.
- Créez le modèle complet, affichez son résumé. Que représentent les colonnes *Output Shape* et *Param #* et comment sont-ils calculés.
- Quelles couches contiennent le plus de paramètres ? Pourquoi ?
- Quels sont les principaux paramètres de la fonction *compile()*. Quel *optimizer* choisir. Vous fixerez son *learning_rate* à une valeur assez élevée *(1e-3)*. Pourquoi pensez-vous que cette valeur doit être assez élévée, pensez-vous qu'elle reste constante durant tout l'apprentissage.
- Quelle fonction de perte est adaptée pour ce problème. Quelle métrique utiliser.
- Créez un objet *EarlyStopping* avec une surveillance sur val_loss, une *patience* de 5 et la possibilité de conserver le meilleur modèle.
- Entrainez le modèle sur 20 *epoch*, des *batch_size* de taille 64, un découpage pour la validation de 20% et un *callbacks* sur *EarlyStopping*. Vous ajouterez l'option *shuffle* à *True*, qui permet de mélanger les images, lors des découpages en *batch*. Pourquoi mélanger les données (*shuffle=True*) peut-il être important pour les images de *X_train*.
- Tracez les courbes *(loss, val_loss)* et *(accuracy, val_accuracy)*. Observe-t-on du surapprentissage. Le modèle converge-t-il correctement.
- Effectuer la prédiction. Sous quelle forme sont retournés les résultats. Que représentent les valeurs obtenues.
- Pour comparer les prédictions avec *y_test* Quelle transformation faut-il appliquer prédictions.
- Affichez la matrice de confusion. Quelles classes sont le plus souvent prédites.
- Choisissez les images les plus mal prédites et affichez-en 20. Essayez d'expliquer pourquoi le modèle commet ces erreurs.

3. Modification du modèle

- Pourquoi peut-il être pertinent d'ajouter une couche de *Normalisation* des données avant la première couche de convolution. Quel est l'intérêt

d'obtenir une distribution centrée autour de zéro. Quel impact cela peut-il avoir sur la vitesse d'apprentissage.

- Dans les architectures modernes, les couches *Conv2D* sont souvent remplacées par le triplet : *Conv2D (sans activation)* + *BatchNormalization* + *Activation*. Quel est selon vous l'intérêt de la couche de *BatchNormalization,* pourquoi est-il préférable de séparer la convolution et l'activation.
- Modifiez le réseau pour intégrer ce nouveau schéma.
- Entraînez ce nouveau modèle dans les mêmes conditions que précédemment. Comparez les courbes d'apprentissage, le temps de convergence, et les performances finales.
- Nous souhaitons maintenant utiliser des images de taille plus importante : (256,256). Pourquoi l'utilisation simple de *resize()* qui recadre les images à la taille demandée, peut-elle poser problème lorsque les images ne sont pas carrées. Que se passe-t-il si hauteur et largeur sont différentes.
- On se propose d'effectuer un redimensionnement proportionnel identique dans les deux directions. Quel est l'avantage de ce redimensionnement sur le *resize()* de *Pillow*. Pourquoi est-il important de conserver les proportions des objets dans une tâche de classification.
- Pourquoi faut-il dans ce cas ajouter un *padding* sur la dimension la plus petite.
- Appliquez cette méthode à l'ensemble du jeu de données.

```
#Redimensionnement des images et padding
def resize_and_padding(img, w, h) :
    expand = min(w/img.size[0], h/img.size[1])   #coefficient d'expansion
    #Calcul de la nouvelle taille de l'image
    new_w, new_h = int(img.size[0]*expand), int(img.size[1]*expand)
    #Redimensionnement de l'image sans déformation en utilisant une fonction
    #  d'interpolation
    img_expand = img.resize((new_w, new_h), Image.LANCZOS)
    #Création d'un fond noir
    new_img = Image.new("RGB", (w,h), (0,0,0))
    #Ajout de l'image redimensionnée au centre de new_img
    new_img.paste(img_expand, ((w-new_w)//2,(h-new_h)//2))
    return new_img
```

- Recréez un modèle identique au précédent mais adapté à la nouvelle dimension d'entrée (256,256,3). Affichez le *summary()* du modèle. Que constatez-vous concernant le nombre de paramètres de la couche *Flatten()*. Pourquoi ce nombre augmente-t-il fortement.

- Remplacez la couche *Flatten()* par une couche *GlobalAveragePooling2D()*. Comment évolue le nombre total de paramètres. Pourquoi y a-t-il une réduction significative.
- Entraînez ce nouveau modèle dans les mêmes conditions que les précédents (epochs, batch_size, validation_split, callbacks). Comment évoluent les performances par rapport au modèle précédent.
- Affichez le résultats (*loss, accuracy*) sur le test. Comparez le nombre total d'images correctement classifiées entre les deux modèles.
- Analyse globale. Quel modèle vous semble le plus pertinent, vous prendrez en compte tous les paramètres (complexité du modèle, nombre de paramètres, vitesse de convergence, performances).

4. Stratégie pré-entrainé (transfer learning + fine-tuning)

Le jeu de donnée *Intel* est un *DataSet* de petite taille, et ne peut pas être entrainé par un *CNN* profond depuis zéro. L'idée est d'utiliser les poids d'un modèle pré-entrainé *keras.applications.EfficientNetB0* sur des données *imagenet*. Le principe consister à classer les images en se basant les caractéristiques identifiées par le modèle pré-entrainé en deux étapes.

- **1er étape - Transfert learning** : Elle consiste à geler les poids du modèle *EfficientNetB0* (pré-entraîné) et à entraîner uniquement les couches du *MLP* ajoutées en sortie pour l'adapter aux images. Cette phase permet d'ajuster le modèle aux classes du jeu de données *Intel* sans modifier les représentations générales déjà apprises.
- **2nd étape - Fine-tuning** : Elle consiste à déverrouiller et entraîner uniquement les dernières couches de *EfficientNetB0*, ainsi que celles du *MLP*. Les premières couches du réseau détectent des caractéristiques visuelles générales (bords, textures, formes simples), communes à toutes les images ne doivent pas être modifiée. En revanche, les derniers blocs *EfficientNetB0*, plus spécialisés dans la classification des images d'*ImageNet*, doivent être réajustés afin de mieux s'adapter aux spécificités des images du dataset *Intel*.
- Créez un modèle *EfficientNetB0* En utilisant les poids *imagenet*. Vous devez supprimer a couche de classification finale (*include_top=False*) afin de pouvoir la remplacer par une classification adaptée pour les images d'*intel*.

```
#Modèle EfficientNetB0
from keras.applications import EfficientNetB0
model_effNet = EfficientNetB0(include_top=False, weights="imagenet",
               input_shape=(h, w, 3)))
```

- Affichez le *summary()* du modèle, combien y-a-t-il de couches. Quelle est la structure générale du modèle. Comment évoluent les dimensions spatiales au fil des blocs.
- Quelle est la première couche du modèle et la dernière couche du modèle lorsque *include_top=False*.
- Redimensionnez les images en taille (180,180) avec *resize()*.
- Quelle est la première et la dernière couche du modèle.
- Pour la première étape les poids du modèle *EfficientNetB0* ne sont pas à mettre à jour. Vous utiliserez pour cela l'attribut *trainable* du modèle. Pourquoi ces poids ne sont pas a modifier.
- Pour être conforme aux images attendues par *EfficientNetB0,* une couche *keras.applications.efficientnet.preprocess_input()* qui normalise les données (mais ne les centre pas) doit être ajoutée avant l'ajout du modèle *EfficientNetB0*, dans le réseau séquentiel.
- Ajoutez ensuite le modèle *EfficientNetB0* en mettant à false le paramètre *training*. Cette variable bloque le comportement des couches en particulier (*BatchNormalisation* et *Dropout*), qui pourraient impacter le comportement du modèle même si les poids ne sont pas mis à jour.
- Pour transformer la sortie convolutionnelle en vecteur exploitable par un *MLP* : quelle couche de doit-on ajouter *Flatten()* ou *GlobalAveragePooling2D()*. Pour cela vous analyserez le réseau du modèle *EfficientNetB0* et en particulier la couche de transformation des données convolutionnelles.
- Ajoutez ensuite une couche de *BatchNormalisation* et deux couches *Dense* de 256 et 128 neurones avec des *Dropout* de 0.4 et 0.3.
- Analysez le *summary()* du modèle comment interprétez-vous les résultats obtenus sur les *Output Shape* et les *Param*. Combien de paramètres sont entraînables.
- Effectuer l'apprentissage sur 10 *epoch* et une *validation_split* de 20%. Vous choisirez un *learning_rate* assez élevé (1e-3) pour l'*optimizer*.
- On doit ensuite effectuer le *Fine-tuning*, pour cela on doit rendre les poids des dernières couches du modèle *EfficientNetB0 trainable*. On doit

par contre conserver tous les poids calculés par les premiers blocs de convolution (1 à 6) de ce modèle mais rendre le dernier bloc et les dernières couches *trainable* pour s'adapter aux images *intel*.

```
#Rendre les dernières couches non trainable
model_effNet = True
for layers in model_effNet[:-20] : layers.trainable=False
```

- Analysez via summery() le modèle, que constatez-vous.
- Effectuer le même apprentissage que précédemment et utilisant un *learnin_rate* plus petit (1e-5), maintenant que le modèle à été pré-entrainé par la première étape, et que les poids censés être plus proches de leurs valeurs finales.
- Affichez les courbes *(loss, val_loss)* et *(accuracy, val_accuracy)*. Commentez-les résultats obtenus.
- Affichez le résultats (*loss, accuracy*) sur le test, ainsi que la matrice de confusion. Combien y-a-t-il d'images mal classées.
- Affichez quelques images correspondant à celles le plus mal classées.
- Analyse finale : comparez le modèle *CNN* entraîné depuis zéro, avec et sans *BatchNormalization*, le modèle avec *EfficientNetB0*.