8INF804 - Traitement numérique des images

UQAC

Université du Québec
à Chicoutimi

Travail Pratique n°3

Entrainement de CNN

Groupe :

LACLAVERIE Pierre (LACP03119904)

SIMON Thibaud (SIMT15039901)

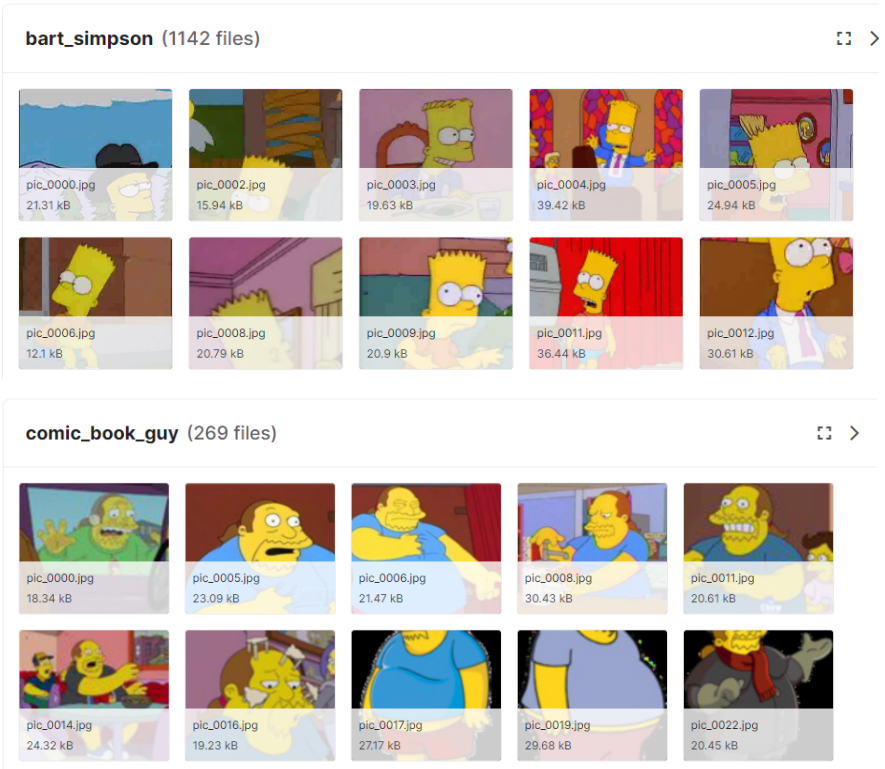
OUHAOUADA Nihal (OUHN11629909)

REYNAUD Yann (REYY07110005)

Choix d’un dataset

Pourquoi avons-nous choisi ce dataset?
Après recherche sur la banque de données Kaggle, nous sommes tombés sur le dataset “Simpson Images” qui nous a satisfait pour les raisons suivantes:

- Ce sont des simpsons : ça parle à tout le monde
- Le dataset est complet et sa taille est raisonnable (moins de 500mo)
- Le dataset propose un set de test préparé et équitable de 50 images par classe
- Le dataset propose un set de training assez complet, avec des images en grande quantité, surtout pour les personnages principaux.
- La résolution des images n’est pas trop grande permettant d’éviter les longs temps de chargement.



Analyse du Dataset

Le dataset est composé de 19 classes représentant 19 personnages de la série télévisée The Simpson. Ce dataset comporte des images de tailles différentes. Les personnages sont globalement centrés sur l’image. Il y a trois grandes catégories de données : proche de la tête du personnage, celles présentant une grande partie du corps du personnage, et enfin des images ou le personnage est plus difficilement reconnaissable. Après avoir parcouru manuellement la base de données, les deux premières catégories sont prépondérantes sur la troisième. Cependant cette troisième catégorie ne doit pas être négligée. Elle sera étudiée plus tard dans le rapport. Ci-dessous un exemple de chaque catégorie: (images issues de la partie du dataset pour l'entraînement).



Sur la première image, une vue de proche de la tête de Homer Simpson. Sur la seconde image un plan américain ou une grande partie du corps est visible, les éléments de décors occupent une proportion plus grande en nombre de pixels que sur la première image. Sur la troisième image, Homer est clairement visible, mais deux gardes sont aussi présents sur l'image et occupent proportionnellement une place plus importante que Homer sur l'image. Les images appartenant à la troisième catégorie sont celles qui ont besoin *à priori* d'un regard particulier.

Séparation de l'ensemble de données

Le dataset propose déjà un dossier test qui contient, pour chaque sous-dossier nommé comme une classe, 50 images du personnage à reconnaître.
Nous avons ainsi décidé de procéder à la séparation suivante:

- Training set: 64% des images du dossier training
- Validation set: les autres 16% des images du dossier training
- Testing set: 15% des images du dossier training

Le dossier test sera réservé à des fins de test du modèle chargé avec un affichage du simpson reconnu, en aval.

Transfer learning

Choix d'un modèle

En observant le tableau des architectures proposées, nous cherchions à obtenir un modèle qui soit un bon compromis entre entraînement et efficacité.

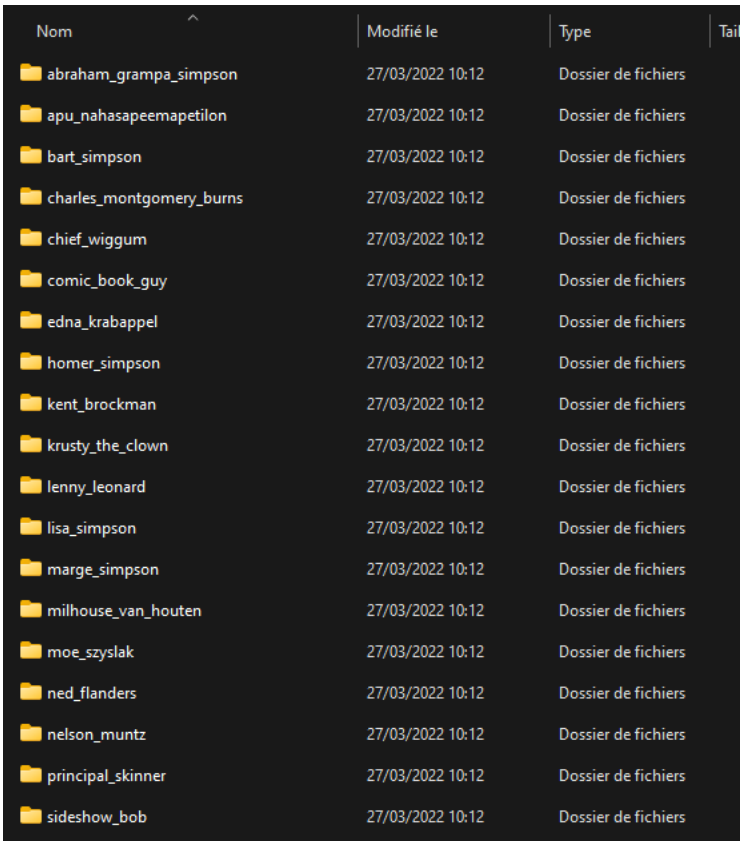
Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4

Nous avons décidé de choisir le modèle ResNet101V2 car son accuracy Top-5 est très bonne (93.8%), mais surtout car même s'il prend plus de place (size en MB) que Xception, il est près de 1,5 fois plus rapide à calculer sur le GPU, ce qui est un critère très intéressant pour notre équipe, car cela aura un impact direct sur le temps d'entraînement.

Pour résumer, ResNet101V2 nous semble être un très bon compromis entre rapidité d'exécution et qualité des résultats.

Méthode d'entraînement, pas à pas

1. Chargement des données
   Notre set de données est découpé tel quel:



Il sera ainsi possible d'itérer sur chaque sous dossier du dossier training afin d'obtenir les images correspondant à chaque simpson. Pour nous faciliter la tâche, nous utiliserons les

noms de ces sous-dossiers comme noms de classes lors de nos affichages et entraînements.

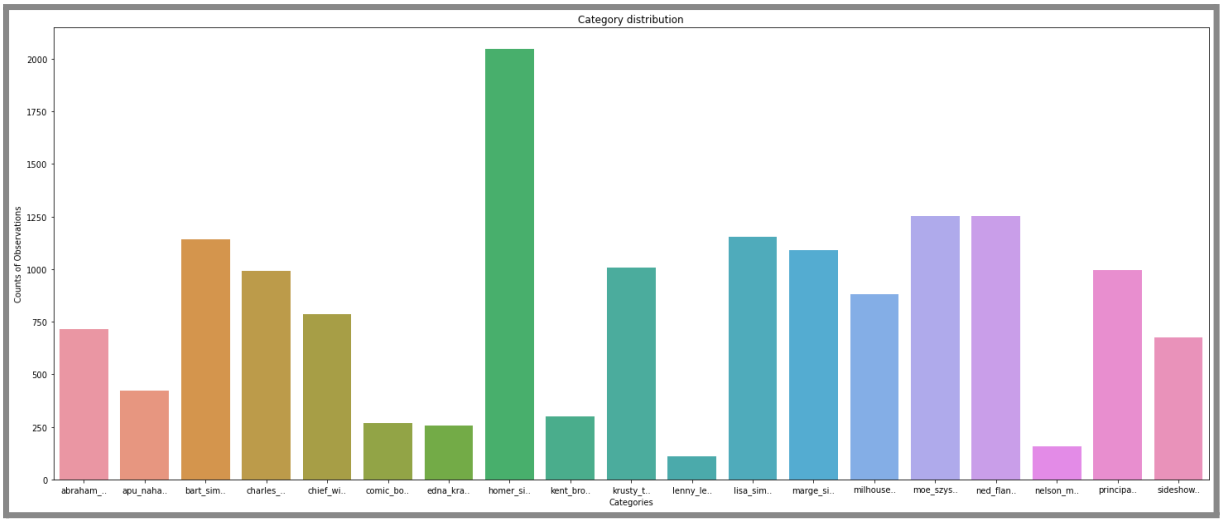
```
filenames= os.listdir(train_path) # Récupération des sous dossiers

names = []
for filename in filenames: # Attribuer chaque sous dossier à un nom
    names.append(filename)

train_list = []
for name in names:
  train_list .extend(glob.glob(train_path+"/"+name+"/*.jpg"))
```

(Code complet dans le notebook fourni en annexe)

Un dataframe sera ensuite créée pour associer les fichiers identifiés dans la liste d'entraînement avec nos noms de classes:

	label	image
0	charles_..	/content/drive/MyDrive/tni/archive/train/charles_montgomery_burns/pic_0233.jpg
1	charles_..	/content/drive/MyDrive/tni/archive/train/charles_montgomery_burns/pic_0113.jpg
2	charles_..	/content/drive/MyDrive/tni/archive/train/charles_montgomery_burns/pic_0227.jpg
3	charles_..	/content/drive/MyDrive/tni/archive/train/charles_montgomery_burns/pic_0224.jpg
4	charles_..	/content/drive/MyDrive/tni/archive/train/charles_montgomery_burns/pic_0127.jpg
...	...	...
15513	nelson_m..	/content/drive/MyDrive/tni/archive/train/nelson_muntz/pic_0067.jpg
15514	nelson_m..	/content/drive/MyDrive/tni/archive/train/nelson_muntz/pic_0104.jpg
15515	nelson_m..	/content/drive/MyDrive/tni/archive/train/nelson_muntz/pic_0142.jpg
15516	nelson_m..	/content/drive/MyDrive/tni/archive/train/nelson_muntz/pic_0282.jpg
15517	nelson_m..	/content/drive/MyDrive/tni/archive/train/nelson_muntz/pic_0290.jpg

Ce dataframe, généré à partir des données et des classes nous permet, avec le package seaborn, de visualiser la distribution des données au sein du set de training initial (pas encore sous-découpé).



On constate nos 19 classes (dont le nom a été raccourci après 8 caractères), et la distribution des données. On note dès lors que le personnage Homer Simpson se voit attribuer beaucoup plus d'images que certains des autres, cela pourra avoir un impact car plus de données induira possiblement une meilleure performance sur ce personnage.

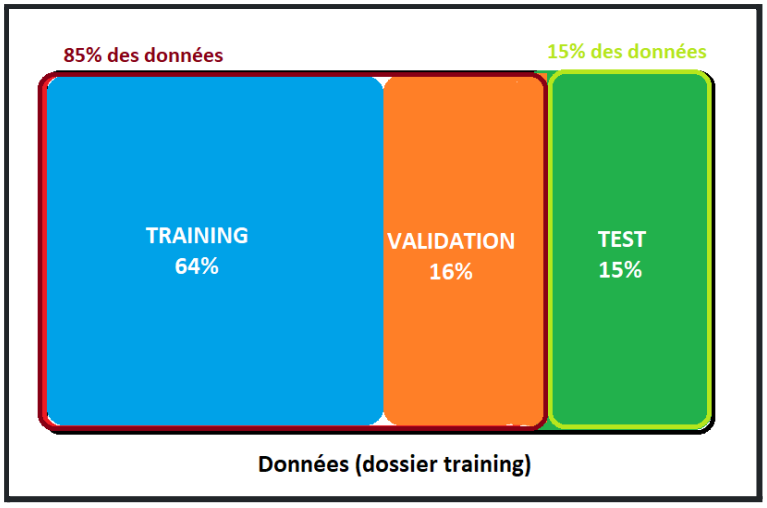
Nous remédierons à certains manques à l'aide de transformations d'images par la suite.

2. Déclaration du modèle

Nous créons alors un tenseur pour nos labels permettant de reshape les labels stockés auparavant dans notre dataframe (df_train) :

```
y = OneHotEncoder(dtype='int8',
sparse=False).fit_transform(df_train['label'].values.reshape(-1,1))
```

On sépare ensuite notre dataset en trois différents set, que sont les sets d'entraînements, de validation et de test.
On considère 15% des images de notre dossier train qui seront utilisés pour le test, et les 85% restants pour le training. De même, on prendra 20% des données d'entraînement pour la validation de notre modèle.
On peut retrouver une représentation de ce découpage (figure a)



(figure a)

Concernant le code, nous avons utilisé la fonction *train_test_split* du package sklearn.model_selection

```
X_data, X_test, y_data, y_test = train_test_split(X, y, test_size=0.15, random_state=SEED)
X_train, X_val, y_train, y_val = train_test_split(X_data, y_data, test_size=0.2,
random_state=SEED)
```

Appliquons maintenant des transformations aléatoires aux sets d'entraînement de de validation pour permettre au modèle un meilleur apprentissage. Parmi ces transformations, on effectue par exemple des rotations, effet miroir, zoom/dézoom, et décalage en hauteur et/ou largeur.

```
train_gen = ImageDataGenerator(horizontal_flip=True,
                        rotation_range = 45,
                        zoom_range=0.2,
                        height_shift_range = 0.5,
                        width_shift_range = 0.5)
validation_gen = ImageDataGenerator(horizontal_flip=True,
                            rotation_range = 45,
                            zoom_range=0.2,
                            height_shift_range = 0.5,
                            width_shift_range = 0.5)
train_gen.fit(X_train)
validation_gen.fit(X_val)
```

Nous avons également créé un petit programme permettant d'afficher des images tirées aléatoirement de notre dataset afin de vérifier leur bon import.

```
indices = np.random.randint(0, X.shape[0],8)
i = 1
plt.figure(figsize=(14,7))
for each in indices:
```

```
    plt.subplot(2,4,i)
    plt.imshow(X[each])
    plt.title(df_train['label'].loc[each])
    plt.xticks([])
    plt.yticks([])
    i += 1
```

3. Entrainement

Commençons par définir les principaux paramètres que nous allons utiliser lors de l'apprentissage :

```
batch_size = 64
epochs = 35
nb_categories = y.shape[1]
```

Batch_size : hyperparamètre représentant le nombre d'échantillons utilisés pour calculer le gradient d'erreur durant l'entraînement. La batch_size doit être une puissance de 2. Plus il est grand, plus notre modèle sera précis, mais cela augmente également le nombre de calculs et risque donc de saturer la RAM.
nb_categories : nombre de classes. Ici, nous en avons 19.
Epochs : nombre de périodes d'entraînement.

On définit alors notre modèle, comme expliqué auparavant, nous utiliserons le modèle Resnet101V2 de *Tensorflow.keras*.

```
base = ResNet101V2(include_top=False,
            weights='imagenet',
            input_shape=(128,128,3))
x = base.output
x = GlobalAveragePooling2D()(x)
```

Le modèle exclut la couche d'entrée ainsi que la couche de sortie
Ici, nous définissons x comme couche de sortie du modèle, puis nous appliquons à cette couche un *GlobalAveragePooling2D*, pour calculer la moyenne de la feature map.
On définit également la couche d'entrée comme une couche dense avec une fonction d'activation Softmax :

```
head = Dense(nb_categories, activation='softmax')(x)
```

Puis on ajoute ces deux couches (d'entrée et sortie) à notre modèle :

```
model = Model(inputs=base.input, outputs=head)
```

On compile alors le modèle :

```
model.compile(optimizer=Adam(lr=0.0001),
              loss = 'categorical_crossentropy',
              metrics=['accuracy'], run_eagerly=True)
```

Après test de différents optimizers (par exemple SGD) sur notre modèle, nous avons fini par conclure, au vu des résultats, que Adam était le plus adapté. Le loss est défini comme la perte de l'entropie croisée entre les images labellisées (vérité terrain) et les prédictions, d'où l'argument *categorical_crossentropy*: Enfin nous utilisons le 'categorical' car nous avons plus de deux classes (en opposition avec binary_crossentropy).

Nous pouvons alors afficher toutes les couches de notre modèle à l'aide de *model.summary()*, mais nous avons trouvé plus judicieux de les afficher sous forme graphique, permettant une meilleure représentation visuelle de sa constitution (voir notebook).

Enfin, nous définissons nos callbacks, pour permettre un "EarlyStopping" , ou arrêt hâtif de l'entraînement lorsque notre modèle ne s'améliore plus :

```
callbacks = [
  tf.keras.callbacks.EarlyStopping(monitor="val_loss",min_delta=1e-2,
                                   patience=2,
                                   verbose=1)]
```

Nous avons ici choisi de stopper notre entraînement lorsque *val_loss* cesse de s'améliorer (voire même augmente), et pour un minimum de 2 époques.
Nous entraînons alors notre modèle, en enregistrant ses données dans history:

```
history = model.fit_generator(
              train_gen.flow(X_train, y_train,
              batch_size=batch_size),
              epochs = epochs,
              validation_data = validation_gen.flow(X_val, y_val),
              callbacks = callbacks
)
```

Nous constatons ainsi les résultats du modèle pour chaque époque (ainsi que le temps) :

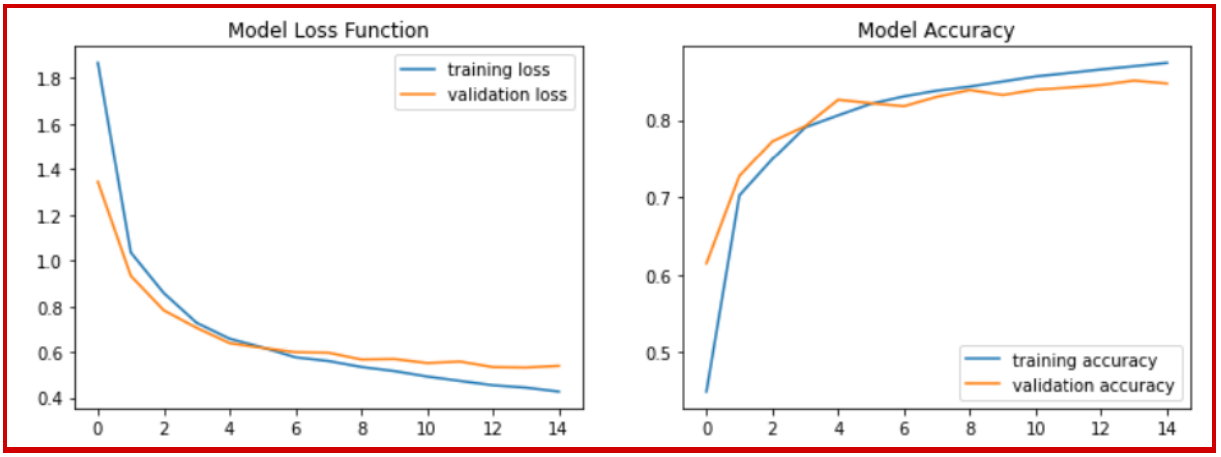
```
Epoch 1/35
330/330 [==============================] - 114s 327ms/step - loss: 1.8673 -
accuracy: 0.4492 - val_loss: 1.3443 - val_accuracy: 0.6146
Epoch 2/35
330/330 [==============================] - 107s 323ms/step - loss: 1.0353 -
accuracy: 0.7021 - val_loss: 0.9328 - val_accuracy: 0.7272
Epoch 3/35
330/330 [==============================] - 107s 323ms/step - loss: 0.8577 -
accuracy: 0.7491 - val_loss: 0.7822 - val_accuracy: 0.7724
Epoch 4/35
330/330 [==============================] - 109s 331ms/step - loss: 0.7273 -
accuracy: 0.7906 - val_loss: 0.7054 - val_accuracy: 0.7921
Epoch 5/35
330/330 [==============================] - 110s 334ms/step - loss: 0.6578 -
accuracy: 0.8061 - val_loss: 0.6386 - val_accuracy: 0.8263
Epoch 6/35
330/330 [==============================] - 108s 328ms/step - loss: 0.6197 -
accuracy: 0.8211 - val_loss: 0.6175 - val_accuracy: 0.8217
Epoch 7/35
330/330 [==============================] - 109s 331ms/step - loss: 0.5764 -
accuracy: 0.8306 - val_loss: 0.5988 - val_accuracy: 0.8179
Epoch 8/35
330/330 [==============================] - 109s 329ms/step - loss: 0.5603 -
accuracy: 0.8380 - val_loss: 0.5967 - val_accuracy: 0.8300
Epoch 9/35
330/330 [==============================] - 109s 331ms/step - loss: 0.5342 -
accuracy: 0.8432 - val_loss: 0.5668 - val_accuracy: 0.8388
Epoch 10/35
330/330 [==============================] - 109s 330ms/step - loss: 0.5163 -
accuracy: 0.8497 - val_loss: 0.5690 - val_accuracy: 0.8323
Epoch 11/35
330/330 [==============================] - 109s 331ms/step - loss: 0.4923 -
accuracy: 0.8563 - val_loss: 0.5514 - val_accuracy: 0.8392
Epoch 12/35
330/330 [==============================] - 109s 331ms/step - loss: 0.4730 -
accuracy: 0.8606 - val_loss: 0.5578 - val_accuracy: 0.8418
Epoch 13/35
330/330 [==============================] - 109s 331ms/step - loss: 0.4542 -
accuracy: 0.8653 - val_loss: 0.5336 - val_accuracy: 0.8452
Epoch 14/35
330/330 [==============================] - 110s 334ms/step - loss: 0.4438 -
accuracy: 0.8694 - val_loss: 0.5318 - val_accuracy: 0.8509
Epoch 15/35
330/330 [==============================] - 110s 333ms/step - loss: 0.4265 -
accuracy: 0.8737 - val_loss: 0.5390 - val_accuracy: 0.8471

Epoch 00015: early stopping
```

Nous avions demandé au modèle de s'entraîner sur 35 époques afin d'en balayer un grand nombre et d'optimiser nos résultats. On remarque ainsi que notre EarlyStopping a fonctionné à la quinzième époque car notre val_loss finit par stagner, voire augmenter (passage de 53.18% à 53.9% alors que les pertes lors de la validation diminuaient jusqu'à la quatorzième époque). Notre modèle retiendra donc les poids et biais de la quatorzième époque, pour un accuracy de 86.94%.

En lance ensuite une succession d'entraînements (cinq fois), on peut alors obtenir notre accuracy moyen :

	Itération 1	Itération 2	Itération 3	Itération 4	Itération 5	Moyenne
Accuracy (%)	87,31	89,9	87,08	88,23	87,41	**87,99**

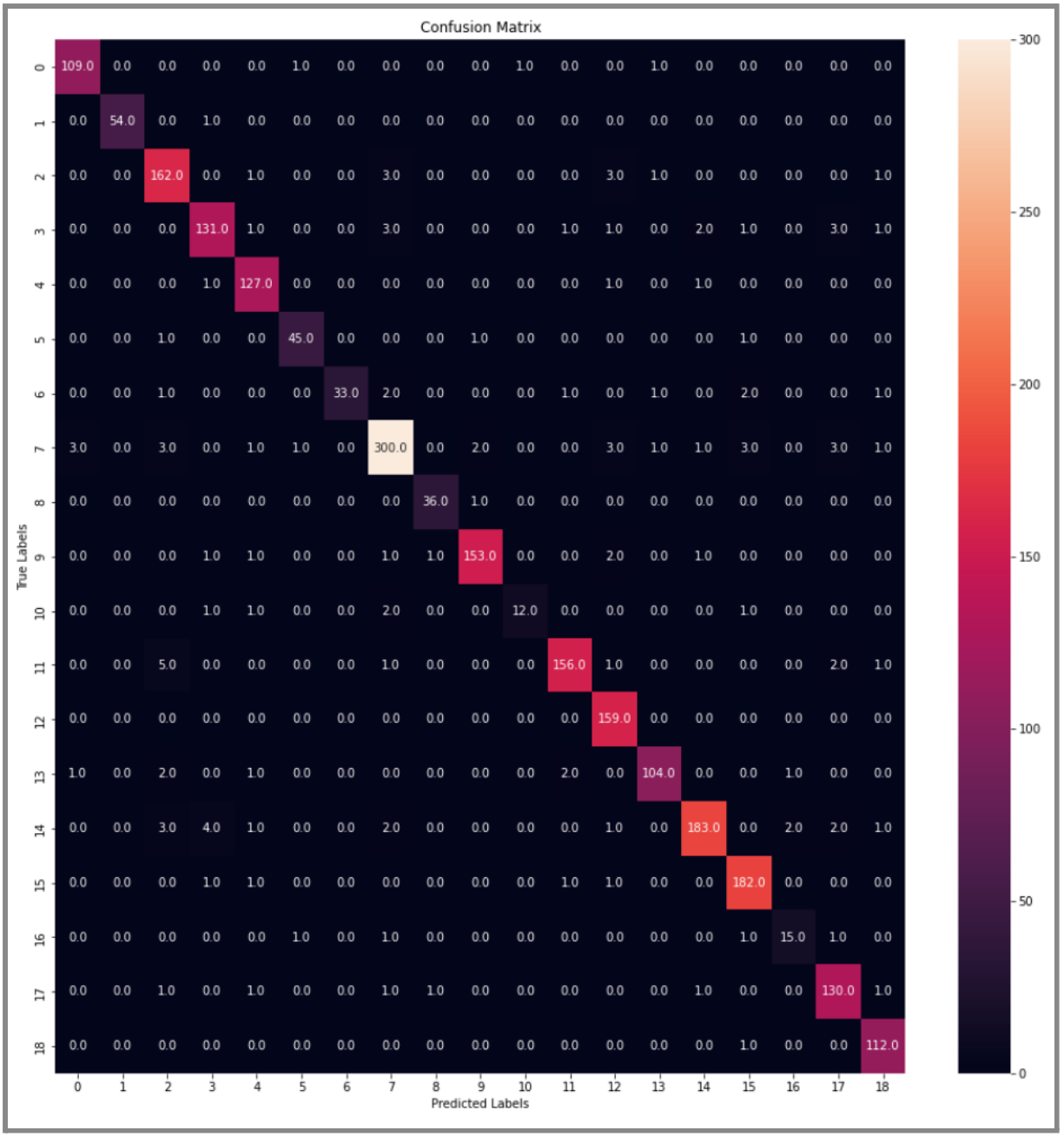
Grâce aux sauvegardes faites dans history, on peut alors afficher deux graphiques représentant d'une part les pertes durant l'entraînement et la validation, et d'autre part la précision de notre modèle au cours de ces 2 phases.



On peut faire la même observation que précédemment : sur le graphe de gauche, on voit bien que nos pertes lors de la validation cessent de s'améliorer à partir de la douzième époque (alors que durant le training elles continuaient de baisser).

Nous affichons et analysons par la suite la matrice de confusion :

```
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis = 1)
y_test_classes = np.argmax(y_test, axis = 1)
confmx = confusion_matrix(y_test_classes, y_pred_classes)
f, ax = plt.subplots(figsize = (16,16))
sns.heatmap(confmx, annot=True, fmt='.1f', ax = ax)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show();
```



Cette matrice de confusion illustre le nombre d'éléments qui ont été associés à leur classe estimée respective.
On dénote la qualité de notre reconnaissance avec cela.
Plus la matrice est diagonale, plus notre modèle est bon. Dans l'ensemble, les coefficients hors diagonale sont assez faibles, ce qui nous rassure sur la qualité de notre modèle.
Il est important de remarquer que nous n'avons pas le même nombre d'images pour chaque classe (ou personnage). On peut assez facilement deviner que la classe numéro 7 (sur-représentée par rapport aux autres classes en nombre d'images) correspond au personnage principal : *Homer Simpson.*
Avec un dataset en entrée plus équilibré, nous aurions eu une matrice de confusion de meilleure qualité.

Nous pouvons aussi afficher le rapport de classification :

```
print(classification_report(y_test_classes, y_pred_classes))
```

	precision	recall	f1-score	support
0	0.96	0.97	0.97	112
1	1.00	0.98	0.99	55
2	0.91	0.95	0.93	171
3	0.94	0.91	0.92	144
4	0.93	0.98	0.95	130
5	0.94	0.94	0.94	48
6	1.00	0.80	0.89	41
7	0.95	0.93	0.94	322
8	0.95	0.97	0.96	37
9	0.97	0.96	0.97	160
10	0.92	0.71	0.80	17
11	0.97	0.94	0.95	166
12	0.92	1.00	0.96	159
13	0.96	0.94	0.95	111
14	0.97	0.92	0.94	199
15	0.95	0.98	0.96	186
16	0.83	0.79	0.81	19
17	0.92	0.96	0.94	136
18	0.94	0.99	0.97	113
accuracy			0.95	2326
macro avg	0.94	0.93	0.93	2326
weighted avg	0.95	0.95	0.95	2326

Chaque colonne de ce rapport permet d'avoir des informations sur la qualité de la classification.
Pour la fin de cette partie,

- TP représente les True Positive : Le nombre d'images que le programme a considéré appartenant à la classe X et qui appartiennent réellement à la classe X

- TN représente les True Negative : Le nombre d'images que le programme a considéré n'appartenant pas à la classe X et qui n'appartiennent pas à la classe X

- FP représente les False Positive : Le nombre d'images que le programme a considéré appartenant à la classe X et qui n'appartiennent pas à la classe X

- FN représente les False Negative : Le nombre d'images que le programme a considéré n'appartenant pas à la classe X et qui appartiennent réellement à la classe X

En premier lieu, chaque colonne de la première partie de ce rapport représente le numéro de la classe, la précision, le recall, le f1 score et le support.

Nous testons ensuite le modèle entraîné, sur nos données de test :

La précision est calculée à l'aide de la formule suivante :

$$precision_x = \frac{TP}{TP+FP}$$

La précision pour une classe x représente la justesse des prédictions: plus la précision est proche de 1, meilleure elle est. On aura tendance à faire confiance au programme si la précision est proche de 1 et qu'il retourne une réponse positive.
Pour vouloir calculer la précision, il faut se poser la question :
Quelle est la proportion des prédictions qui ont été correctes?

Le recall est calculé à l'aide de la formule suivante :

$$recall_x = \frac{TP}{TP+FN}$$

Le recall pour une classe x représente la proportion de résultats positifs qui a été réellement trouvée par le programme. Plus le recall est proche de 1 pour une classe, plus le programme n'aura pas tendance à laisser de côté des images qui auraient dû être considérées comme résultat positif.
Pour vouloir calculer le recall il faut se poser la question :
Quelle est la proportion des cas positifs que le programme à trouvé?

Le F1 score est calculé à l'aide de la formule suivante :

$$F_{1,x} = 2\frac{recall_x * precision_x}{recall_x + precision_x}$$

Le F1 score représente la proportion de prédictions positives qui a réellement été correcte.
Le F1 score est compris entre 0 et 1. Plus le F1 score est proche de 1 plus les prédictions qui sont considérées comme positives par le programme sont justes.
Pour vouloir calculer le F1 score, il faut se poser la question :
Quelle est la proportion des cas positifs qui ont réellement été corrects?

Le support est le nombre d'occurrence de la classe dans le dataset.

```
test_loss = model.evaluate(X_test, y_test)
```

```
73/73 [==============================] - 5s 73ms/step - loss: 0.1885 - accuracy:
0.9592
```

Sur nos données de test, nous obtenons finalement un accuracy de 95.92%, ce qui est convenable.

```
test_loss = model.evaluate(X_val, y_val)
```

```
83/83 [==============================] - 6s 75ms/step - loss: 0.1850 - accuracy:
0.9507
```

En testant le modèle sur nos données de validation, nous trouvons un accuracy de 95.07%, résultat à peu près similaire à l'évaluation du modèle sur les données de test.

Afin de tester notre modèle sur d'autres images de test, nous avons utilisé le dossier test (non utilisé auparavant). Ce choix nous a paru judicieux car ce dataset est équilibré (50 images/personnages).

Comme précédemment, nous créons notre dataframe des données de test à partir du dossier test (et non pas train cette fois-ci).

```
absolute_path = os.path.abspath(os.curdir)
main_path = "drive/MyDrive/tni/archive"
main_path = os.path.join(absolute_path, main_path)
print(main_path)
test_path = os.path.join(main_path,"test")

import os

filenames= os.listdir(test_path) # get all files' and folders' names in the current directory

names = []
for filename in filenames: # loop through all the files and folders
   names.append(filename)

test_list = []
for name in names:
 test_list.extend(glob.glob(test_path+"/"+name+"/*.jpg"))
#Construction d'un dataframe montrant les longueurs de chaques labels

name_img_counts = {}
for name in names:
 name_img_counts[name] = len(os.listdir(test_path+"/"+name))
print(name_img_counts)

np_array = []
for name in names:
 np_array = np.append(
     np_array, [(name[:8] + '..') if len(name) > 8 else name]*name_img_counts[name]
 )

df_test = pd.DataFrame(np_array, columns = ['label'])
df_test['image'] = [x for x in test_list]
```

On crée ensuite un tableau d'images qui ont été resize en 128 x 128, ainsi que le tenseur des labels.

```
path = main_path+'/test'
#List of image:
img_list = list(df_test['image'])
data_img = []
for each in img_list:
  #Each image path:
  each_path = os.path.join(path, each)
  #Read each image:
  each_img = plt.imread(each_path)
  #Resize the images:
  each_img_resized = cv2.resize(each_img, (128,128))
  #Save arrays to a list:
  data_img.append(each_img_resized)
# Converting list to numpy array
X = np.array(data_img)
print('Shape of X: ', X.shape)
y = OneHotEncoder(dtype='int8',
sparse=False).fit_transform(df_test['label'].values.reshape(-1,1))
print('Shape of y: ', y.shape)
```

Sur le notebook, nous affichons un graphique avec la distribution du nombre d'images par classe, celui-ci nous confirme bien que notre dataset est équilibré (toutes les barres sont à la même hauteur).

Nous chargeons ensuite le modèle entraîné, qui était sauvegardé dans un fichier d'extension h5. Pour ce faire nous utilisons la fonction *models.load_model* de tensorflow keras.

```
new_model = tf.keras.models.load_model('drive/MyDrive/tni/saved_model/my_model.h5')
```

Puis nous évaluons les pertes et l'accuracy de notre modèle sur ce set de test.

```
loss, acc = new_model.evaluate(X, y, verbose=2)
```

Remarque : Nous avons testé plusieurs modèles tels que InceptionV3, ou InceptionResNetV2 cependant ils ont donnés des résultats moins probant que celui actuellement utilisé : ResNet101V2

Le coefficient de Kappa permet d'estimer l'accord entre plusieurs jugements appliqués à des mêmes objets (ici nos images) provenant de 2 observateurs techniques. Celui-ci prend en compte le hasard. Il nous permet donc de corriger ou interpréter des désaccords.

```
cohen_kappa_score(y_test_classes, y_pred_classes)
```

```
[50]: 0.9430295432016704
```

Comme nous pouvons le voir, nous obtenons un coefficient de kappa de 94%

Conformément au tableau ci-contre, nous en concluons que nous avons un accord presque parfait entre nos observateurs.

Valeurs observées de Kappa	Interprétation
<0	Accord quasi-inexistant
0-0,20	Faible accord
0,21-0,40	Accord passable
0,41-0,60	Accord modéré
0,61-0,80	Accord important
0,81-1,00	Accord presque parfait

(figure b)

Application du modèle sur des images de simpsons :

Nous commençons par créer une fonction prenant en argument une image (array) et qui retourne la prédiction du label associé ainsi que son score.

```
def imgTest(im_arr):
   #Prediction
   prediction = new_model.predict(im_arr) score=
tf.nn.softmax(prediction[0]) return names[np.argmax(score)], score
   plt.figure(figsize=(19,19))
```

Enfin, nous appliquons cette fonction à plusieurs images tirées aléatoirement grâce au code suivant :

```
import random
plt.figure(figsize=(19,19))
for i in range(19):
    plt.subplot(5,4,i+1)
    im = X[random.randint((i*50),((i+1)*50))]
    plt.imshow(im)
    im_arr = tf.keras.preprocessing.image.img_to_array(im)
    im_arr = tf.expand_dims(im_arr, 0)
    name, score = imgTest(im_arr)
    plt.title(name + " {:.2f}%".format(100*np.max(score)))
    plt.xticks([])
    plt.yticks([])
```

Dans la plupart des cas, notre modèle reconnaît bien les labels associés aux images, mais il arrive qu'il se trompe, par exemple il a assigné le label Chief Wiggum au mauvais personnage. Il faut noter qu'il avait une confiance plus faible (inférieure à 11%) tandis que pour le reste des images où il n'a pas échoué, il obtient un score supérieur à 11-12%. Même si les fois où on a une mauvaise assignation sont peu nombreuses, on constate que le score de prédiction reste assez faible.

Construction de l'architecture CNN personnalisée :

Démarche :

Après avoir chargé correctement notre dataset, nous avons procédé comme suit pour bâtir notre propre modèle CNN :

1) Prétraitement de données :

Pour prétraiter les données d'entraînement et de test, nous avons utilisé la fonction **ImageDataGenerator** de Keras. Nous allons voir si nos données (training et test) sont bien récupérées.

```
Found 15502 images belonging to 19 classes.
```

Nous constatons que pour le training set, nous avons 15502 images appartenant à 19 classes.

Nous cherchons à ce que notre modèle soit généralisé pour des données qu'il n'a jamais rencontré, cela a pour conséquence d'éviter le problème dit d'overfitting, donc, nous allons

nous servir d'un ensemble de données de validation qui nous aidera à évaluer la performance de notre architecture CNN . Pour ce faire, les données d'entraînement seront réparties en 2 sous-parties de la manière suivante :

- Training_set : 80% de l'ensemble des données d'entraînement
- Validation_set : 20% de l'ensemble des données d'entraînement

La nouvelle configuration se présente comme suit :

```
Found 12411 images belonging to 19 classes.
Found 3091 images belonging to 19 classes.
```

Nous avons bien 12411 données pour le training et 3091 données pour la validation.

```
Found 950 images belonging to 19 classes.
```

Pour le test set, nous avons 950 images appartenant à 19 classes.

NB : le mode de classification choisi est 'categorical' car notre problème est multi classification.

Le but maintenant c'est de créer notre CNN, l'entraîner , le valider , puis finalement le tester sur les données de test.

2) Création du CNN :

Pour créer notre CNN, nous avons suivi les étapes suivantes :

Etape 1 : Initialisation

Nous initialisons le CNN comme une séquence de couches, à l'aide de la fonction **models.Sequential()**de Keras. Par la suite, nous allons ajouter des couches à notre CNN de telle sorte d'améliorer l'accuracy finale de notre modèle.

Etape 2 : Convolution / pooling

C'est l'idée centrale des CNN. En effet, pour extraire les caractéristiques des images, on applique plusieurs filtres de convolution sur nos images qui vont produire des cartes de convolution. Ces cartes seront par la suite aplaties et concaténées pour former le vecteur d'entrée de notre CNN (input layer). Une correction de ces cartes peut éventuellement être utilisée (à l'aide de la fonction d'activation ReLu) afin de ne garder que les nombres positifs et en remplaçant les nombres négatifs par des zéros.

Tel qu'expliqué au cours, la couche pooling , qui a pour but de réduire d'une manière progressive la taille de nos images (en ne gardant que les informations les plus importantes), peut produire l'effet inverse et détruire ces informations. Donc , nous n'allons pas l'utiliser pour éviter ce genre de problèmes.

Nous allons configurer notre CNN pour traiter des entrées de format (64,64,3) qui est la dimension de nos images en mode RGB.

Jusqu'à maintenant, l'architecture de notre CNN est la suivante :

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 126, 126, 32)      896

 conv2d_7 (Conv2D)           (None, 124, 124, 64)      18496

 conv2d_8 (Conv2D)           (None, 122, 122, 64)      36928

=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

Nous avons appliqué 3 couches de convolutions. Plus on progresse dans le réseau, plus la dimension (largeur, hauteur) est diminuée .

Etape 3 : Couches CNN

Le réseau de neurones final doit être composées de 3 parties :

- Couche d'entrée (Input layer) :

Pour obtenir cette première couche, il suffit d'aplatir les cartes de convolution trouvées dans la partie précédente .

- Couches cachées (Hidden layers) :

Ces couches définissent la densité de notre réseau, et c'est à travers ces couches que l'information (sous forme d'un signal) se propage en combinant les caractéristiques pour une meilleure prédiction. Nous allons considérer un schéma « fully-connected » c'est-à-dire chaque neurone est connecté avec tous les neurones de la couche précédente.

Encore une fois, des problèmes de gradient peuvent se manifester à la suite d'une mauvaise initialisation des poids ou alors un mauvais choix de la fonction d'activation. Pour pallier ce problème, nous allons utiliser l'initialisation He et la variante de ReLU : leaky ReLU (tel que proposé dans le cours).

Toujours pour éviter les problèmes de gradient au cours de l'entraînement, une autre approche peut être utilisée. Il s'agit de la normalisation des lots. On peut faire cela, avec tensorflow, en ajoutant une couche **BatchNormalization** avant ou après chaque fonction d'activation des couches cachées.

Dans notre cas, nous avons utilisé une seule couche cachée.

- Couche de sortie (output layer) :

  C'est la dernière couche de notre réseau, elle permet de prédire la valeur de la sortie suivant l'information reçue des couches précédentes. Elle doit contenir 19 unités (19 neurones) qui correspondent au nombre de nos classes.

  Nous présentons le modèle final de notre CNN :

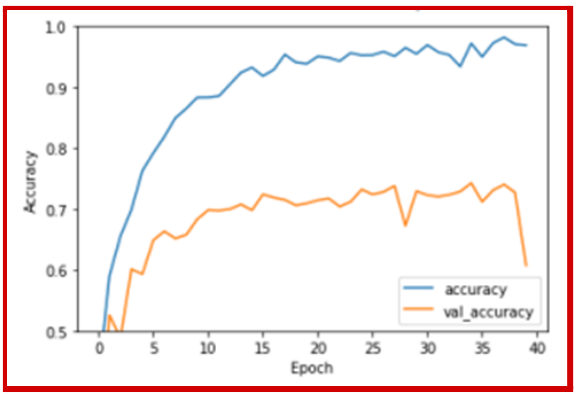
```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_15 (Conv2D)          (None, 62, 62, 32)        896

 conv2d_16 (Conv2D)          (None, 60, 60, 64)        18496

 conv2d_17 (Conv2D)          (None, 58, 58, 64)        36928

 flatten_5 (Flatten)         (None, 215296)            0

 batch_normalization_15 (Bat  (None, 215296)           861184
 chNormalization)

 dense_10 (Dense)            (None, 64)                13779008

 leaky_re_lu_5 (LeakyReLU)   (None, 64)                0

 batch_normalization_16 (Bat  (None, 64)               256
 chNormalization)

 dense_11 (Dense)            (None, 19)                1235

=================================================================
Total params: 14,698,003
Trainable params: 14,267,283
Non-trainable params: 430,720
_________________________________________________________________
```

Nous remarquons que nous avons passé du tenseur 3D (58,58,64) à un vecteur 1D de taille 215296.

3) Entrainement du modèle :

Maintenant que nous avons défini les différentes couches de notre modèle, nous allons le compiler, et mettre à jour (minimiser) l'erreur de la sortie (définie par la différence entre la valeur exacte et la valeur prédite) après chaque itération . Pour ce faire, nous avons utilisé l'optimiseur Adam. Le choix de cet optimiseur a été fait en se basant sur le tableau présenté dans le cours, Ce dernier offre un bon compromis rapidité/qualité de convergence.

Après avoir lancé l'exécution, nous obtenons un premier tracé qui présente les 2 précisions (d'entraînement et de validation) en fonction du nombre d'epochs.



Nous pouvons voir un surajustement dans notre modèle. En effet, la précision de l'ensemble des données d'entraînement est très élevée par rapport à celles de validation.

Notre modèle CNN est très performant sur les données du training , nous avons pu obtenir une précision de 98.8% au bout de 40 epochs. Cependant, ce qui nous intéresse vraiment, c'est la création d'un modèle capable de se généraliser à des données qu'il n'a jamais vues. Et comme nous le constatons, nous avons obtenu une précision de 73% sur les données de validation. Cette valeur, bien qu'elle soit correcte, n'est pas très satisfaisante. Pour l'améliorer, nous allons rajouter des couches de dropout à notre réseau. Ces couches ont pour but de retirer les neurones qui n'ont pas un grand impact sur la prédiction. En paramètre de ces couches, nous passons les valeurs 0.2 et 0.7 qui enlèvent 20% (respectivement 70%) du nombre total de neurones constituant notre couche cachée.

La qualité et la quantité des données à disposition jouent aussi un rôle très important dans la performance du modèle. Or, dans notre cas, il s'agit d'un problème avec de nombreuses classes et pour chaque classe, le nombre d'images peut ne pas être suffisant. Chose qui peut justifier les résultats obtenus. donc nous avons augmenté ces données à l'aide des lignes du code suivantes:

```
data_augmentation = tf.keras.Sequential(
  [
    tf.keras.layers.RandomFlip("horizontal", input_shape=(128,128,3)),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
  ]
)
```

Pour mieux détecter les caractéristiques de notre dataset, nous avons augmenté le kernel size en passant de la valeur 3 à 5.

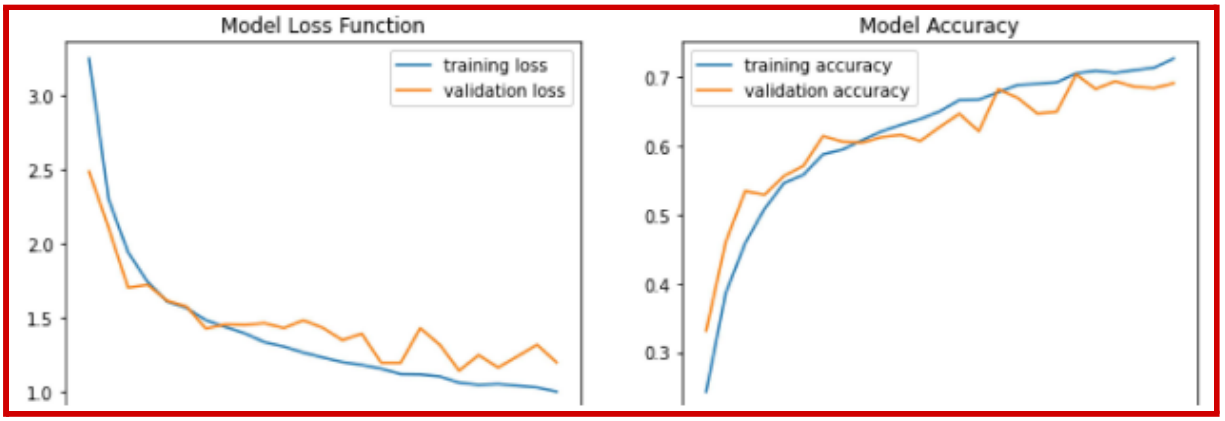
De plus, nous avons utilisé une régularisation du type L2 (plus stable que L1) afin de rendre notre modèle le moins complexe possible.

Un callback earlystopping est également utilisé pour indiquer le nombre d'epoch au niveau duquel une quantité spécifié (le val_accuracy dans notre cas) doit s'arrêter , et cela pour ne pas perdre du temps dans des calculs inutiles.

Nous présentons donc le modèle finale de notre CNN qui tient en compte tous ces critères :

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_1 (Sequential)   (None, 128, 128, 3)       0

 conv2d (Conv2D)             (None, 126, 126, 32)      896

 conv2d_1 (Conv2D)           (None, 124, 124, 64)      18496

 conv2d_2 (Conv2D)           (None, 122, 122, 64)      36928

 flatten (Flatten)           (None, 952576)            0

 batch_normalization (BatchN  (None, 952576)           3810304
 ormalization)

 dropout (Dropout)           (None, 952576)            0

 dense (Dense)               (None, 128)               121929856

 leaky_re_lu (LeakyReLU)     (None, 128)               0

 batch_normalization_1 (Batc  (None, 128)              512
 hNormalization)

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 19)                2451

=================================================================
Total params: 125,799,443
Trainable params: 123,894,035
Non-trainable params: 1,905,408
_________________________________________________________________
```

Nous finissons avec ce résultat :




```
194/194 [==============================] - 168s 864ms/step - loss: 0.9998 - accuracy: 0.7260 - val_loss: 1.1968 - val_accuracy: 0.6904
```

On remarque qu'après avoir appliquer les modifications précédentes, il y a moins d'overfitting et les 2 précisions (entraînement et validation) sont plus proches. Peut être qu'en augmentant le nombre d'epochs (initialement fixé à 25) les courbes vont continuer à s'améliorer.
Sur nos données de test, nous avons une accuracy qui vaut 71.58% :

```
15/15 - 248s - loss: 1.0643 - accuracy: 0.7158 - 248s/epoch - 17s/step
```

Cette valeur nous semble bien correcte pour un petit nombre d'itérations.

Comme nous l'avons fait pour la partie du transfer learning, nous allons afficher le rapport de classification et calculer le coefficient kappa qui nous donneront des informations sur la qualité de notre modèle:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	108
1	0.67	0.03	0.06	67
2	1.00	0.01	0.01	144
3	0.19	0.62	0.29	159
4	0.44	0.11	0.17	113
5	0.25	0.02	0.04	46
6	0.50	0.14	0.22	42
7	0.41	0.49	0.45	301
8	0.15	0.48	0.23	46
9	0.32	0.81	0.46	149
10	0.00	0.00	0.00	15
11	0.24	0.69	0.35	170
12	0.70	0.15	0.24	176
13	0.34	0.74	0.46	117
14	0.25	0.11	0.15	194
15	0.00	0.00	0.00	175
16	0.00	0.00	0.00	20
17	0.33	0.01	0.01	177
18	1.00	0.03	0.05	107
accuracy			0.29	2326
macro avg	0.36	0.23	0.17	2326
weighted avg	0.39	0.29	0.21	2326

```
[20] cohen_kappa_score(y_test_classes, y_pred_classes)
```

```
0.22966050107042402
```

Comme nous pouvons le voir, nous obtenons un coefficient de kappa de 22%. Conformément au tableau ci-dessous, nous en concluons que nous avons un accord passable entre nos observateurs.

Comparaison:

Le tableau suivant récapitule les résultats d'un modèle construit à main avec celui bâti grâce au transfert learning.

métrique	CNN à main	CNN avec du transfert learning
training accuracy	71%	87%
validation accuracy	69%	84%
test accuracy	71%	95%
kappa	22%	94%

Conclusion

Ce travail pratique a permis de découvrir deux manières de concevoir un réseau de neurones. D'un côté, commencer à partir de rien et de soi-même mettre en place les couches cachées, les inputs et output layers. Cela permet d'avoir le contrôle sur tous les paramètres du réseau de neurones. L'inconvénient est qu'il faut mettre en place une architecture, faire beaucoup de tests et complexifier l'architecture afin de pouvoir avoir un résultat satisfaisant. D'un autre côté, réutiliser le travail d'autres personnes et l'adapter à notre problème. Cependant il y a un travail de recherche préliminaire pour savoir quelle architecture utiliser et comment l'adapter à notre problème : Le transfert learning.

Dans le cadre de notre sujet, il est préférable de faire le transfert learning plutôt que de coder à partir de rien notre réseau de neurones. En effet, cette méthode permet d'économiser en temps de recherche car il est plus facile d'adapter une architecture de référence que d'en construire une à partir de rien. Les résultats sont aussi de nature différentes : avec le transfert learning on obtient des résultats très satisfaisant tandis qu'ils sont moins intéressants quand on regarde ce qui a été fait avec le réseau de neurone fait à partir de rien.

Il est donc préférable de regarder si des architectures existantes ne permettent pas de résoudre le problème au lieu de créer un modèle.

Piste d'améliorations

Au cours de la rédaction de ce rapport nous avons constaté des pistes d'améliorations :

- Répéter plusieurs fois l'entraînement
- Utiliser des classes plus équilibrées

En effet :

- L'entraînement et l'utilisation d'un réseau de neurones n'est pas forcément déterministe. Même si, lors de nos nombreux tests, les résultats étaient globalement

ressemblants, ils ne l'étaient pas strictement. Pour avoir une meilleure idée des résultats, il aurait fallu procéder à une série d'entraînement et les comparer (moyenne, écart type, variance, moment d'ordre 2,...). Dans le cas du réseau de neurones simple, nous n'avons pas eu le temps de le faire car nous avons passé beaucoup de temps à chercher la meilleure combinaison de paramètres possibles.

- Certaines classes sont surreprésentées par rapport à d'autres : par exemple, Homer Simpson contient 2046 images dans le dossier train tandis que Lenny Leonard n'en contient que 110. Une telle disparité ne permet pas d'avoir une robustesse.
- Un réseau de neurones trop ''simple'' possède ses limites, il ne permet pas de pouvoir traiter des informations qui permettraient d'avoir des caractéristiques plus fines des personnages afin de pouvoir mieux les différencier.