Dogs vs. Cats Classification

Projet de :

MANSOURI Anas - TOUZANI Mouad - RAVICHANTHIRAN Nishani

I- Definitions:

Introduction :

Les Chats et les chiens sont des animaux courants au seins des foyers humains. Il nous est donc facile de faire une distinction entre les deux.

Au lieu de faire cela manuellement, nous allons écrire un algorithme pour classifier si les images contiennent un chien ou un chat.

Nous avons donc affaire à un problème de classification, avec une quantité importante de données que l'on souhaite utiliser pour entrainer notre modèle pour ensuite réaliser des prédictions plus ou moins précises.

Indicateurs:

Les indicateurs ou paramètres les plus importants pour mesurer la performance de la classification sont la précision (accuracy) et la Log Loss (loss function) .

Accuracy

La précision dans les problèmes de classification est le nombre de prédictions correctes faites par le modèle sur tous les types de prédictions effectuées.

Termes associés à la matrice de confusion :

1. True Positives (TP): Les vrais positifs sont les cas où la classe réelle du point de données était (Vraie) et où la prédiction est également (Vraie).
2. True Negatives (TN): Les vrais négatifs sont les cas où la classe réelle du point de données était 0(Fausse) et où la prédiction est également 0(Fausse).
3. False Positives (FP): Les faux positifs sont les cas où la classe réelle du point de données était 0 (fausse) et la prédiction est 1 (vraie). Les faux sont dus à une prédiction incorrecte du modèle et les positifs à une prédiction positive de la classe. (1)
4. False Negatives (FN): Les faux négatifs sont les cas où la classe réelle du point de données était 1(Vraie) et la prédiction est 0(Fausse). Faux parce que le modèle a prédit de manière incorrecte et négatif parce que la classe prédite était négative. (0)

Log Loss

La perte logarithmique mesure la performance d'un modèle de classification où l'entrée de la prédiction est une valeur de probabilité entre 0 et 1. L'objectif de notre modèle d'apprentissage automatique est de minimiser cette valeur. Un modèle parfait aurait une perte logarithmique de 0. La perte logarithmique augmente à mesure que la probabilité prédite diverge de l'étiquette réelle.

II- Analyse

Exploration de données:

Notre DATASET présent sur KAGGLE contient 2 dossiers (training et test1). Le dossier de training contient 25000 images de chats et de chiens, 12500 pour les chats et 12500 pour les chiens, chaque image est étiquetée par le nom de l'image dans le format suivant : cat.n.jpg, où n est un nombre, de sorte que nous pouvons savoir si l'image est un chat ou un chien à partir du nom de l'image.
Dans le dossier test, l'image n'a pas d'étiquette, elle n'est nommée que par des numéros.

III- Methodologie:

Traitement préalable des données:

1. Charger toutes les données du dossier de training :

```
#On importe le module os qui fournit une façon portable d'utiliser les fonctionnalités dépendantes du système d'exploitation.
import os

#On spécifie notre réperoite de travil ou se trouve les données.
base_dir = '../input/dogs-vs-cats'

#On spécifie le chemin d'accès des fichiers zip : train et test1
train_dir = os.path.join(base_dir, 'train.zip')
test_dir = os.path.join(base_dir, 'test1.zip')
```

```
# On Importe le module zipfile qui permet de manipuler des fichiers ZIP.
import zipfile

#On extrait nos deux fichiers zip:

#Fichier train.zip:
with zipfile.ZipFile(train_dir,'r') as z:
    z.extractall()

#Fichier test1.zip:
with zipfile.ZipFile(test_dir,'r') as z:
    z.extractall()

#Après l'extraction des données, on obtient deux dossiers stockés dans l'output.
```

2. Diviser les données en (chat , chien) par le biais de Labels :

```
#On crée une liste "pics" qui contient le nom de tous les fichiers dans le dossier "train"
images = os.listdir('./train')

#On importe la bibliothèque pandas permettant la manipulation et l'analyse des données.
import pandas as pd

#On utilise la fonction DataFrame() qui organise les données en lignes et en colonnes:
data = pd.DataFrame(images)

#On change le nom de la colonne de "0" à "image":
data = data.rename(columns = {0:'image'})

#On ajoute "./train/" avant le nom de chaque element dans la colonne "image":
data['image'] = data['image'].apply(lambda x: './train/' + x)

#Notons que la colonne "image" contient le chemin d'accès de chaque image !

#On crée une nouvelle variable "label" qui prend les valeurs "dog" ou "cat" selon x
data['label'] = data['image'].apply(lambda x: 'cat' if 'cat' in x else 'dog')

#On observe les 5 premières lignes de notre data:
data.head()
```

3. Scinder les données en données de training et en données de validation.

Ici 80% de nos données sont utilisées pour le training et 20 % pour la validation.

```
#On scinde notre data en deux sous-ensembles de données:

#Données d'entraînement : 0.8 de lignes et 2 colonnes
train_data = data.iloc[0:20000, 0:2]
print(train_data.shape)

#Données de validation: 0.2 de lignes et 2 colonnes
validation_data = data.iloc[20000:25000, 0:2]
print(validation_data.shape)
```

```
(20000, 2)
(5000, 2)
```

4. Remettre à l'échelle toutes les couleurs dans les images de 0~255 à 0~1 :

```
# On importe le module ImageDataGenerator qui génère des lots de données
# d'images tensorielles avec une augmentation des données en temps réel.

from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

+ Code + Markdown

```
#Comme 255 est la valeur maximale d'un pixel, on change l'echelle de ces valeurs en divisant par 255
# afin qu'elles prennent de nouvelles valeurs allant de 0 à 1.

train_datagen =  ImageDataGenerator(rescale=1/255)
validation_datagen =  ImageDataGenerator(rescale=1/255)
test_datagen = ImageDataGenerator(rescale=1/255)
```

5. Redimensionner toutes les images à une taille fixe (150x150)

```
#Transformer nos dataframes en batches d'images uniformisées:

train_genarator = train_datagen.flow_from_dataframe(train_data, target_size=(150,150),
                                                    x_col='image',y_col='label',class_mode='binary',batch_size = 32)

validation_genarator = train_datagen.flow_from_dataframe(validation_data,target_size=(150,150),
                                                         x_col='image',y_col='label',class_mode='binary',batch_size= 32)
```

IV- Implementation

```
#Importer tous les bibliothèques et les modules nécéssaire à la construction de notre modèle:

import tensorflow as tf

from tensorflow import keras

from keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

from tensorflow.keras import regularizers

#from keras.preprocessing import image

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from keras import optimizers

#from sklearn.model_selection import train_test_split

#from tensorflow.keras.utils import to_categorical

from tqdm import tqdm

from keras.models import load_model
```

Trois étapes principales lors de l'implémentation :

1) Construire 3 couches de CNN, chaque couche contient la fonction d'activation Relu et du MaxPooling.

```
#On spécifie un modèle sequentiel : empilement simple de couches où chaque couche a exactement un tenseur d'entrée et un de sortie
model = Sequential()

#On définit trois couches de convolution, chacune suivie d'une couche de maxpooling de dimension 2x2

#La première couche contient 32 filtres, de dimension 3x3 et avec la fonction d'activation "relu".
#Notons qu'elle prenne des élements d'entrée de dimension 150x150x3.
model.add(Conv2D(32, kernel_size=(3,3), input_shape=(150, 150, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=2))

#En passant à la deuxième couche, on a augmenté le nombre des filtres à 64.
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=2))

#Et finalement, la dernière couche contient 128 filtres.
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=2))
```

- Utiliser flatten pour convertir notre feature 3D en un vecteur 1D.

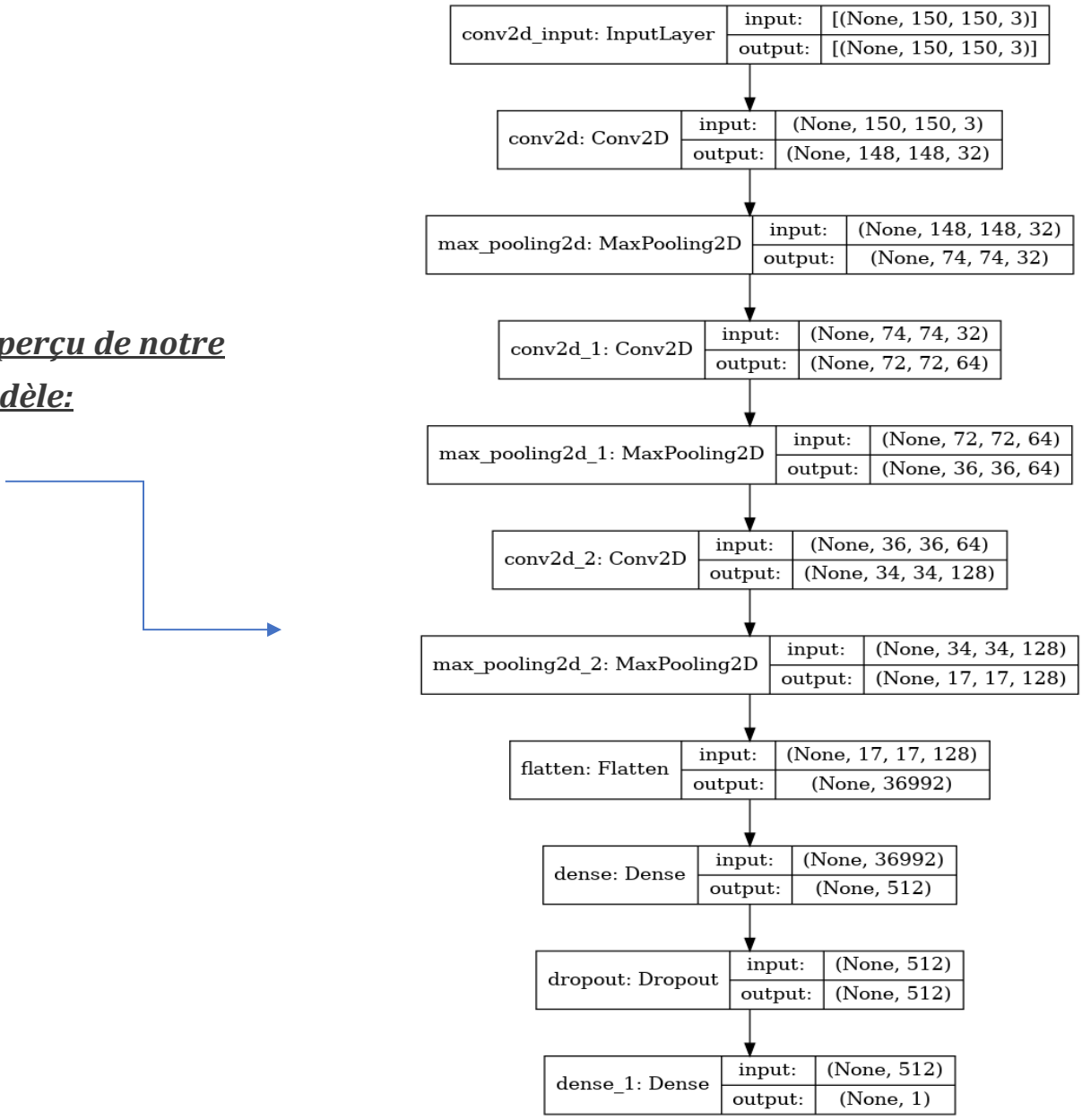
```
#On utilise la fonction Flatten pour applatir les tenseurs d'entrée multidimensionnels en une seule dimension.
model.add(Flatten())
```

- Créer les 2 dernières couches en étant Dense, la dernière étant de taille 1 car notre sortie est soit un chat soit un chien.
- Créer une couche Dropout pour diminuer l'overfitting.
- Utiliser le regularisateur L2.

```
#La couche Dense transforme la matrice d'entrée en vecteur ligne, elle performe aussi des opérations de simplification.
model.add(Dense(512, activation='relu', kernel_regularizer=keras.regularizers.l2(l=0.01)))

#La couche Dropout donne de manière aléatoire la valeur 0 aux elements d'entrée avec une
#fréquence de 0.25 à chaque étape pendant la durée de l'entrainement.

model.add(Dropout(0.25))

#Cette couche dense génère un seul nombre et utilise la fonction d'activation "sigmoid".
model.add(Dense(1, activation='sigmoid'))
```

```
#Un aperçu syntéthique de notre modèle et ses couches:
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='cnn_model.png', show_shapes=True, show_layer_names=True)
```

L'aperçu de notre modèle:



2) Après avoir construit le modèle, nous ajustons notre Dataset sur le modèle keras et obtenons le graphique de Loss et de la précision, mais avant cela, nous devons spécifier certains paramètres.

- Modification du learning rate pour une meilleure précision.

```
#On utilise l'optimisateur ADAMS avec un Learning rate de 0.0001
opt = keras.optimizers.Adam(learning_rate=0.0001)
#On configure le modèle pour l'entrainement : la fonction Loss , l'optimisateur et la fonction metrics permettent d'evaluer le modèle
model.compile(loss='binary_crossentropy',optimizer=opt, metrics=['accuracy'])
```

```
#Un résumé utile du modèle, qui comprend : Le nom et le type de toutes les couches du modèle. Forme de sortie pour chaque couche.
model.summary()
#On remarque une réduction de l'image (dimension de la matrice) et augmentation de la profondeur de la matrice.
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
flatten_2 (Flatten)          (None, 36992)             0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               18940416
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 513
=================================================================
Total params: 19,034,177
Trainable params: 19,034,177
Non-trainable params: 0
_________________________________________________________________
```

```
#On spécifie la mesure de performance à monitorer, le déclencheur, et une fois déclenché, il arrêtera le processus d'entrainement.

#Aussi, On réduit le pas d'apprentissage lorsqu'une métrique a cessé de s'améliorer.

cb = [tf.keras.callbacks.EarlyStopping(monitor = 'val_loss', patience=5,
                                       restore_best_weights=True),
      tf.keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss' , patience=3)]
```

3) Nous pouvons enfin ajuster notre DataSet sur le modèle :

```
#Une fois le modèle spécifié, on commence l'entrainement avec 100 époque:
history = model.fit(train_genarator, epochs = 100, validation_data= validation_genarator,
                    callbacks = cb)
```

```
Epoch 1/100
625/625 [==============================] - 71s 113ms/step - loss: 0.7862 - accuracy: 0.5703 - val_loss: 0.6744 - val_accuracy: 0.5968
Epoch 2/100
625/625 [==============================] - 70s 113ms/step - loss: 0.6352 - accuracy: 0.6562 - val_loss: 0.6071 - val_accuracy: 0.7188
Epoch 3/100
625/625 [==============================] - 70s 112ms/step - loss: 0.5949 - accuracy: 0.7096 - val_loss: 0.5560 - val_accuracy: 0.7406
Epoch 4/100
625/625 [==============================] - 71s 114ms/step - loss: 0.5609 - accuracy: 0.7409 - val_loss: 0.5198 - val_accuracy: 0.7728
Epoch 5/100
625/625 [==============================] - 70s 112ms/step - loss: 0.5376 - accuracy: 0.7556 - val_loss: 0.5741 - val_accuracy: 0.7400
Epoch 6/100
625/625 [==============================] - 71s 114ms/step - loss: 0.5100 - accuracy: 0.7783 - val_loss: 0.4860 - val_accuracy: 0.7976
Epoch 7/100
625/625 [==============================] - 70s 111ms/step - loss: 0.4972 - accuracy: 0.7863 - val_loss: 0.4971 - val_accuracy: 0.7914
Epoch 8/100
625/625 [==============================] - 69s 111ms/step - loss: 0.4869 - accuracy: 0.8019 - val_loss: 0.4687 - val_accuracy: 0.8184
Epoch 9/100
625/625 [==============================] - 70s 111ms/step - loss: 0.4716 - accuracy: 0.8159 - val_loss: 0.5038 - val_accuracy: 0.7966
Epoch 10/100
625/625 [==============================] - 69s 110ms/step - loss: 0.4595 - accuracy: 0.8235 - val_loss: 0.4800 - val_accuracy: 0.8104
Epoch 11/100
625/625 [==============================] - 71s 113ms/step - loss: 0.4356 - accuracy: 0.8339 - val_loss: 0.4170 - val_accuracy: 0.8450
Epoch 12/100
625/625 [==============================] - 69s 111ms/step - loss: 0.4274 - accuracy: 0.8418 - val_loss: 0.4252 - val_accuracy: 0.8478
Epoch 13/100
625/625 [==============================] - 69s 111ms/step - loss: 0.4165 - accuracy: 0.8480 - val_loss: 0.4193 - val_accuracy: 0.8520
Epoch 14/100
625/625 [==============================] - 69s 111ms/step - loss: 0.4069 - accuracy: 0.8527 - val_loss: 0.3997 - val_accuracy: 0.8582
Epoch 15/100
625/625 [==============================] - 69s 111ms/step - loss: 0.3995 - accuracy: 0.8561 - val_loss: 0.4039 - val_accuracy: 0.8564
Epoch 16/100
```

.
.
.
.
.
.
.

```
625/625 [==============================] - 70s 113ms/step - loss: 0.2423 - accuracy: 0.9186 - val_loss: 0.3387 - val_accuracy: 0.8756
Epoch 23/100
625/625 [==============================] - 69s 111ms/step - loss: 0.2368 - accuracy: 0.9230 - val_loss: 0.3411 - val_accuracy: 0.8770
Epoch 24/100
625/625 [==============================] - 71s 113ms/step - loss: 0.2310 - accuracy: 0.9245 - val_loss: 0.3377 - val_accuracy: 0.8748
Epoch 25/100
625/625 [==============================] - 71s 114ms/step - loss: 0.2270 - accuracy: 0.9251 - val_loss: 0.3379 - val_accuracy: 0.8772
Epoch 26/100
625/625 [==============================] - 70s 112ms/step - loss: 0.2221 - accuracy: 0.9261 - val_loss: 0.3713 - val_accuracy: 0.8682
Epoch 27/100
625/625 [==============================] - 69s 111ms/step - loss: 0.2182 - accuracy: 0.9282 - val_loss: 0.3397 - val_accuracy: 0.8808
Epoch 28/100
625/625 [==============================] - 71s 114ms/step - loss: 0.2005 - accuracy: 0.9372 - val_loss: 0.3404 - val_accuracy: 0.8784
Epoch 29/100
625/625 [==============================] - 70s 113ms/step - loss: 0.1984 - accuracy: 0.9391 - val_loss: 0.3410 - val_accuracy: 0.8772
```

- On sauvegarde notre modele:

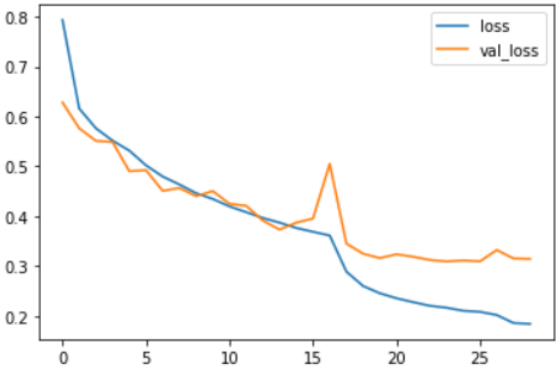
```
os.makedirs("./Saved_Models")
```

```
from keras.models import load_model

model.save('./Saved_Models/model_cat_dog.h5')
```

- Sortie des graphiques de Loss et de notre accuracy :

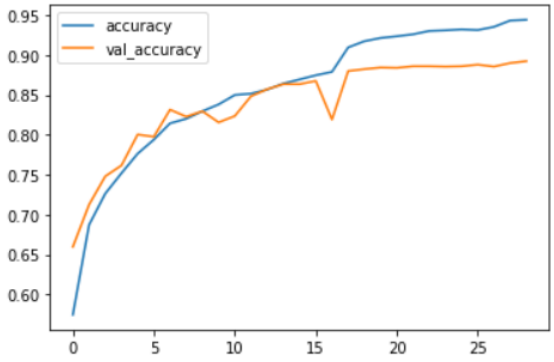
```
#Le graphique de la fonction de perte:
history_df.loc[:, ['loss', 'val_loss']].plot();
print("Minimum validation loss: {}".format(history_df['val_loss'].min()))
```

```
Minimum validation loss: 0.3091488182544708
```




```
#Le graphique de la fonction de précision:
history_df.loc[:, ['accuracy', 'val_accuracy']].plot();
print("Maximum validation accuracy : {}".format(history_df['val_accuracy'].max()))
```

```
Maximum validation accuracy : 0.8924000263214111
```



+ Code + Markdown

V- TEST :

Pour l'étape de test nous avons été contraints de rescinder nos données en allouant 10 % au test, cela est dû au fait que les données présentes sur le dossier test1 ne soient pas labialisées donc peu compatibles avec notre modèle.

```
#On scinde notre data en trois sous-ensembles de données:

#Données d'entraînement : 80% de lignes (20000) et 2 colonnes
train_data = data.iloc[0:20000, 0:2]
print(train_data.shape)

#Données de validation: 10% de lignes (2500) et 2 colonnes
validation_data = data.iloc[20000:22500, 0:2]
print(validation_data.shape)

#Données de test: 10% de lignes (2500) et 2 colonnes:
test_data = data.iloc[22500:25000, 0:2]
print(test_data.shape)
```

```
(20000, 2)
(2500, 2)
(2500, 2)
```

On commence nos prédictions :

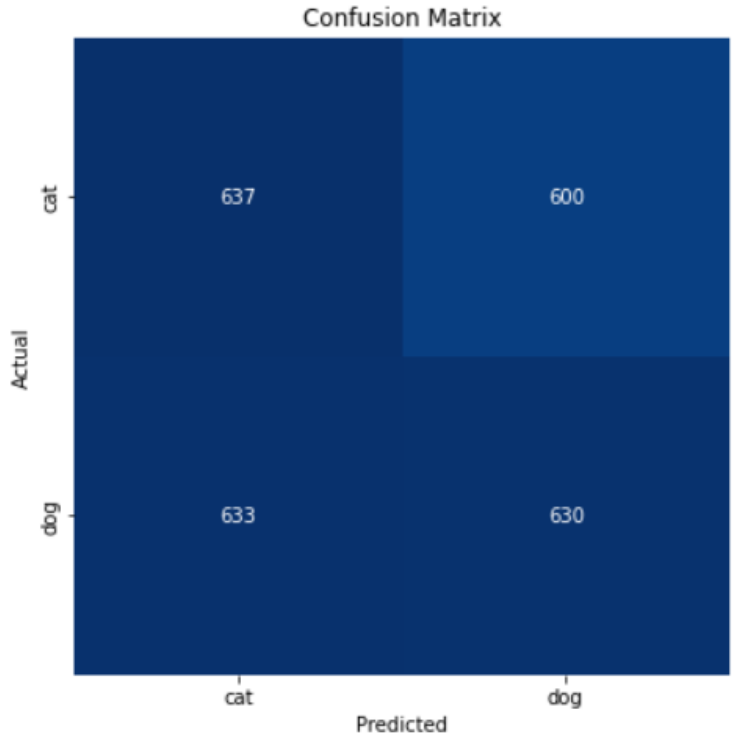
```
#On génère les predictions du modèle par rapport aux données de test:
predict = model.predict_generator(test_generator)

predict = pd.DataFrame(predict)

predict = predict.rename(columns = {0:'Cat probability'})

predict['Dog probability'] = 1 - predict['Cat probability']

display(predict)
```

	Cat probability	Dog probability
0	0.071783	0.928217
1	0.999124	0.000876
2	0.845757	0.154243
3	0.959149	0.040851
4	0.000831	0.999169
...	...	...
2495	0.002988	0.997012
2496	0.858206	0.141794
2497	0.010935	0.989065
2498	0.474753	0.525247
2499	0.012742	0.987258

2500 rows × 2 columns

Matrice de confusion:

```
#Importation des modules:
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

#Si la probabilité dépasse 0.5, la prédiction renvoie "cat":
predictions = (model.predict(test_generator) >= 0.5).astype(np.int)

#On génère la matrice de confusion fournissant une comparaison de la classes prédite et la classe actuelle:
cm = confusion_matrix(test_generator.labels, predictions, labels=[0, 1])

plt.figure(figsize=(6, 6))
sns.heatmap(cm, annot=True, fmt='g', vmin=0, cmap='Blues', cbar=False)
plt.xticks(ticks=[0.5, 1.5], labels=["cat", "dog"])
plt.yticks(ticks=[0.5, 1.5], labels=["cat", "dog"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```




```
#Rapport de classification:
clr = classification_report(test_generator.labels, predictions, labels=[0, 1], target_names=["cat", "dog"])
print(clr)
```

```
              precision    recall  f1-score   support

         cat       0.50      0.51      0.51      1237
         dog       0.51      0.50      0.51      1263

    accuracy                           0.51      2500
   macro avg       0.51      0.51      0.51      2500
weighted avg       0.51      0.51      0.51      2500
```

```
#On crée un répertoire de travail où on va enregistrer notre modèle
os.makedirs("./Saved_Models")

#Ensuite, on enregistre le modèle
from keras.models import load_model
model.save('./Saved_Models/model_cat_dog.h5')
```

```python
#On observe l'évolution des valeurs de perte et de précision selon les époques :
history_df = pd.DataFrame(history.history)
print(history_df)
```

```
        loss  accuracy  val_loss  val_accuracy       lr
0   0.793295   0.57445  0.627880        0.6596  0.00100
1   0.615834   0.68705  0.576182        0.7124  0.00100
2   0.576168   0.72630  0.550973        0.7480  0.00100
3   0.551321   0.75185  0.548529        0.7616  0.00100
4   0.531370   0.77640  0.490168        0.8004  0.00100
5   0.501832   0.79395  0.492092        0.7976  0.00100
6   0.479349   0.81440  0.450633        0.8316  0.00100
7   0.463360   0.82015  0.456379        0.8228  0.00100
8   0.446021   0.82965  0.440107        0.8296  0.00100
9   0.434409   0.83815  0.450289        0.8156  0.00100
10  0.419477   0.85000  0.424692        0.8236  0.00100
11  0.407935   0.85165  0.420823        0.8484  0.00100
12  0.396291   0.85675  0.390991        0.8572  0.00100
13  0.386899   0.86430  0.372972        0.8636  0.00100
14  0.376276   0.86950  0.386970        0.8636  0.00100
15  0.368583   0.87470  0.395213        0.8676  0.00100
16  0.361218   0.87900  0.504918        0.8192  0.00100
17  0.288669   0.90965  0.344917        0.8800  0.00010
18  0.259834   0.91740  0.324620        0.8824  0.00010
19  0.245352   0.92145  0.315897        0.8844  0.00010
20  0.235302   0.92360  0.323524        0.8840  0.00010
21  0.227464   0.92605  0.318563        0.8860  0.00010
22  0.220188   0.93025  0.312141        0.8860  0.00010
23  0.216357   0.93105  0.309149        0.8856  0.00010
24  0.210202   0.93220  0.311090        0.8860  0.00010
25  0.208129   0.93140  0.309338        0.8880  0.00010
26  0.201735   0.93540  0.332132        0.8856  0.00010
27  0.185724   0.94330  0.315240        0.8900  0.00001
28  0.184018   0.94425  0.314182        0.8924  0.00001
```

VI- Résultats

Evaluation de notre modèle:

Comme mentionné ci-dessus, nous atteignons une précision de 89 %, ce qui est relativement très bon, et la perte de log est faible, ce qui est bon.

Nous avons 29 epochs sur lesquelles notre modèle s'est entrainé , et nous pouvons remarquer que l'accuracy (et la Validation Accuracy) augmente au fil des epochs avec une diminution de la Loss(et la Validation Loss) .

Cependant, notre matrice de confusion est peu appréciée, cela peut être explique par la faible quantité de données de validation/test que nous avons utilisé.