# Dogs vs. Cats Classification

## 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. Etant donne que les données de test1 ne sont pas labélisées , nous avons utilisé un nouveau dataset de 100 images étiquetées pour le test .

# 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')
```

```
# 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()
```

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

3. Scinder les données en données de training et en données de validation. *Ici 80% de nos données*

```
#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()
```

*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)
```

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.models import load_model

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

from keras import optimizers

from tensorflow.keras import regularizers

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from tqdm import tqdm
```

---

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))
```

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

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

- Créer les 2 dernières couches en étant Dense, la dernière étant de taille 1 car notre sortie est soit un chat soit un chien.

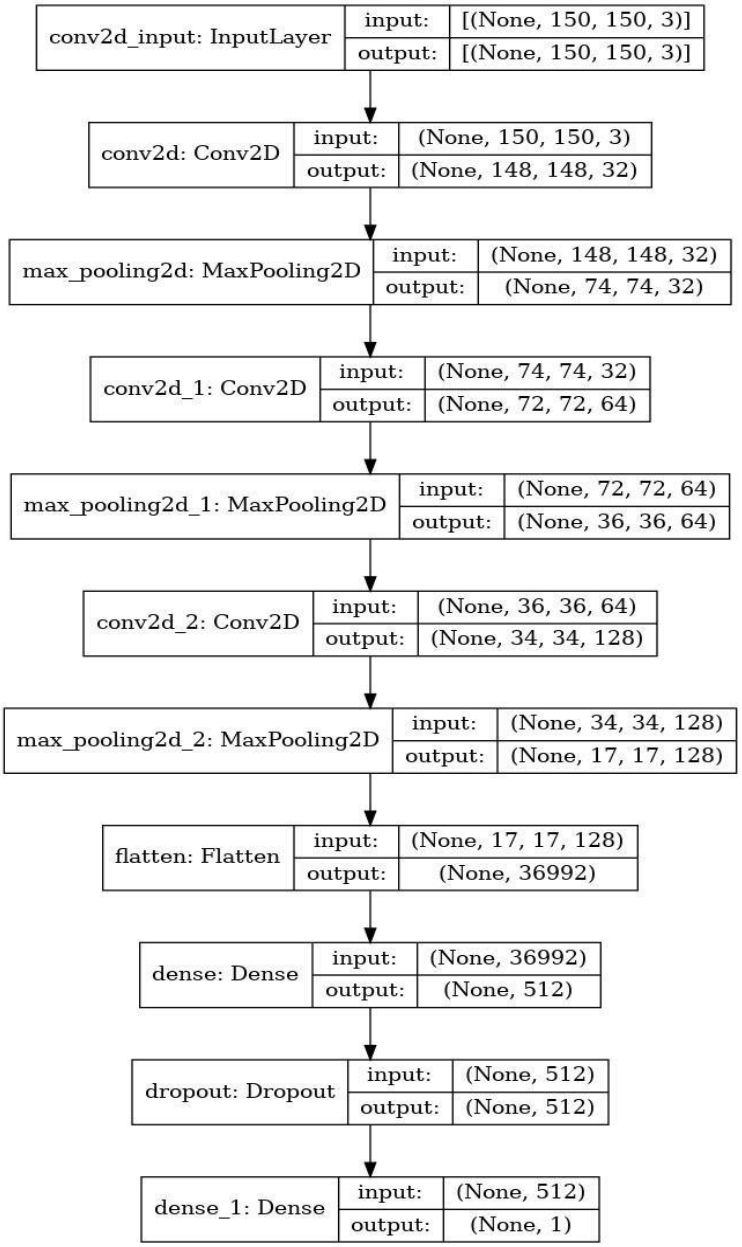
```
#La couche Dense transforme la matrice d'entrée en vecteur ligne, elle performe aussi des opérations de simplification.
model.add(Dense(512, activation='relu', kernel_regularizer=keras.regularizers.l2(l=0.01)))

#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)
```

- Créer une couche Dropout pour diminuer l'overfitting.
- Utiliser le r”gularisateur L2.

## ***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:

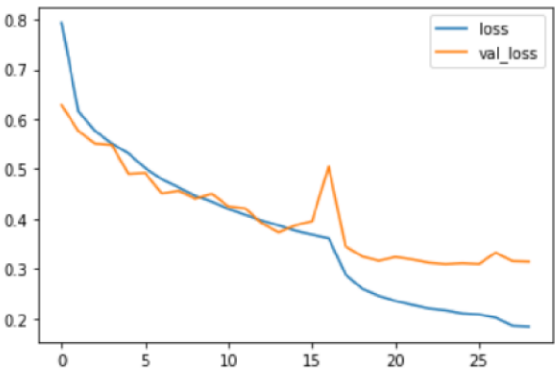
```
os.makedirs("./Saved_Models")
```

```
from keras.models import load_model

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

- Sortie des graphiques de Loss et de notre accuracy :

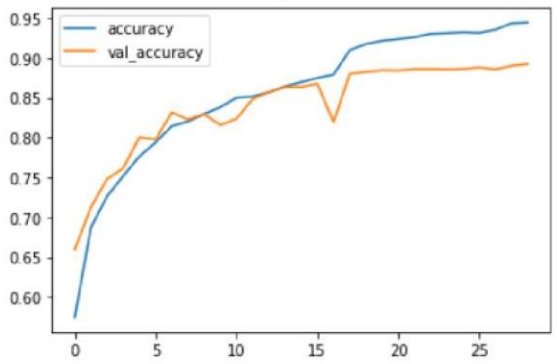
```
#Le graphique de la fonction de perte:
history_df.loc[:, ['loss', 'val_loss']].plot();
print("Minimum validation loss: {}".format(history_df['val_loss'].min()))
```

```
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- Choix des paramètres :

Dans cette partie, on souhaite changer les valeurs de nos paramètres clés dans l'optique d'améliorer la précision globale de notre modèle sans tomber dans de l'overfitting/underfitting.

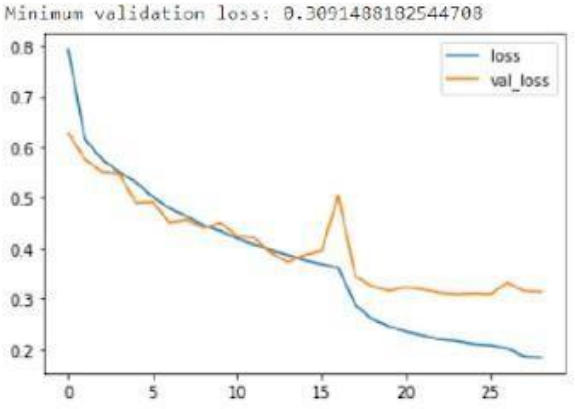
# ***Paramètres à modifier :***

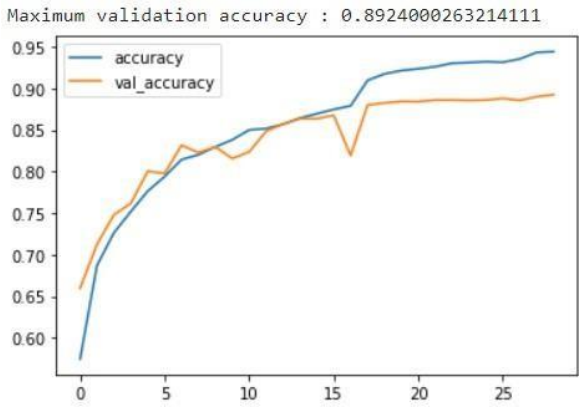
## 1. Déterminer le nombre de couches dans le modèle :

L'impact d'une couche en moins ou supplémentaire sur le modèle :

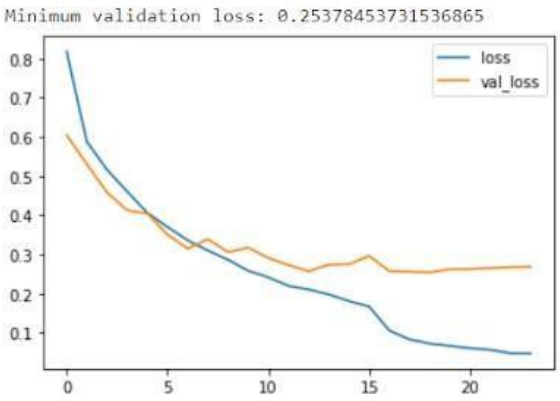
- Deux couches : modèle pas assez complexe et entraînement insuffisant.

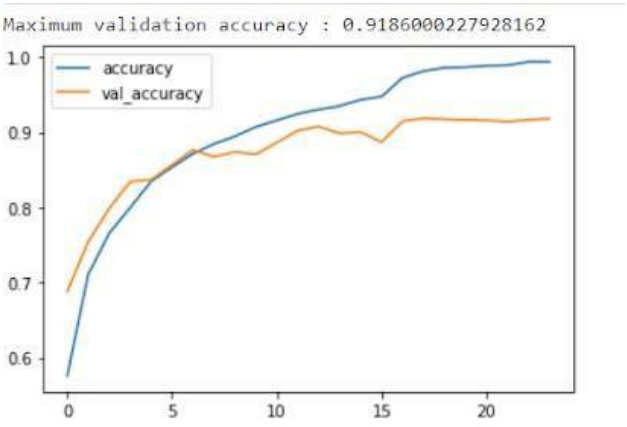
- Trois couches : modèle assez acceptable.




- Quatre couches : la couche supplémentaire a fait augmenter la précision du modèle et a fait diminuer la fonction Loss.

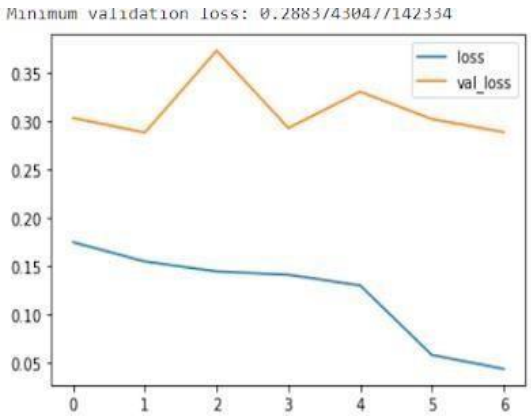


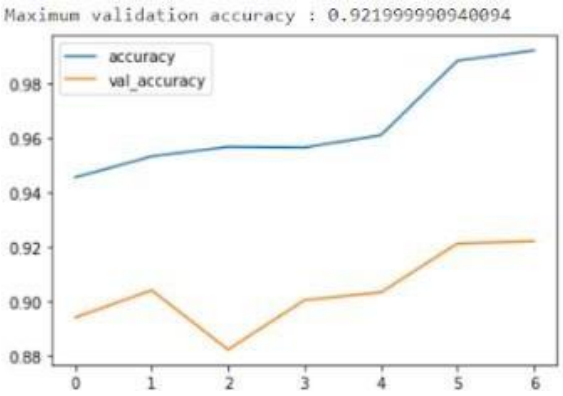




## 2. Choisir l'optimiseur:

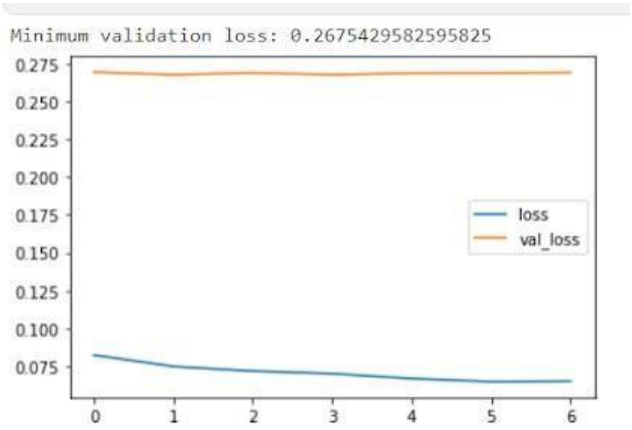
Pour commencer, on a pris "Adam" dont les résultats figurent sur les deux graphes en dessus.

On essaie de changer vers **RMSprop** pour voir ce que cela donne de différent. Le résultat s'est avéré très insatisfaisant au regard du modèle précédent.



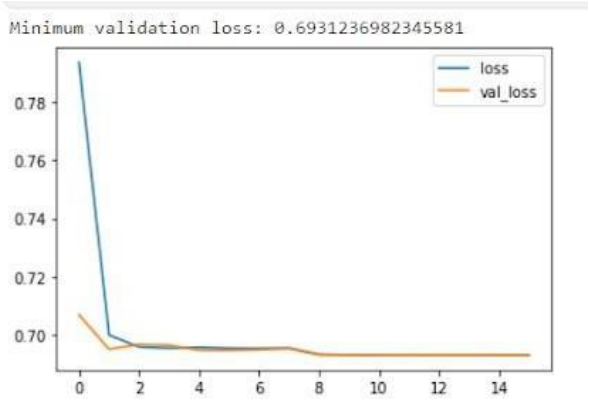



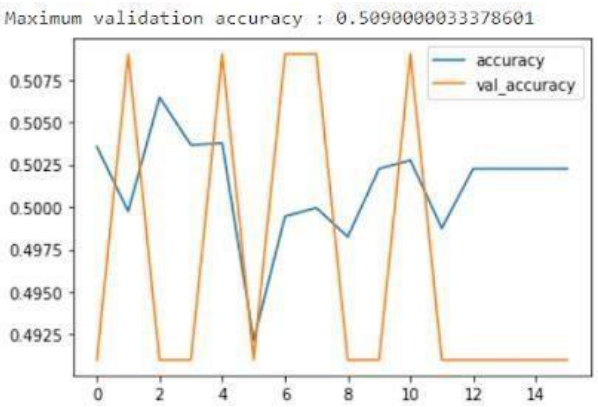

Ensuite, on teste un troisième optimiseur **SGD**.
Malgré la bonne précision et la valeur minimale de la fonction Loss, la qualité de l'ajustement des données de l'entraînement aux données de validation est très mauvaise.




### 3. Déterminer le Learning Rate :

On revient sur le modèle ayant **Adam** comme optimiseur.
Dans le modèle initial construit lors de la partie implémentation, nous avons opté pour un Learning rate de 0.0001.
Essayons d'augmenter ce dernier pour qu'il prenne une valeur de 0.01



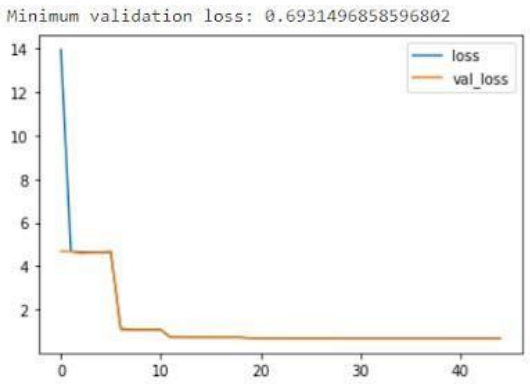



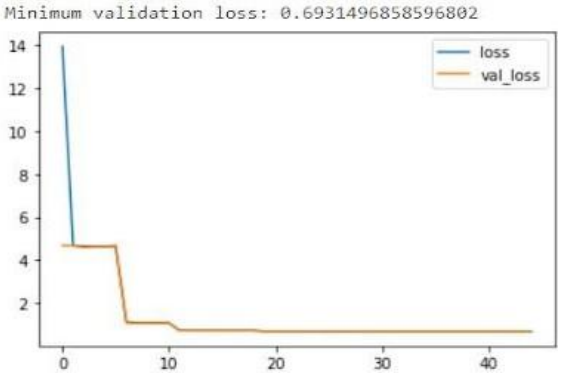

On constate alors que le Learning Rate le plus optimal reste de 0.001 (valeur par défaut).

## 4. Choisir une technique de régularisation :

Il existe deux techniques de régularisation: L1 qui permet d'effectuer une feature selection plus précise (trouver des sous-ensembles de données pertinents) et L2 qui permet d'effectuer un apprentissage plus rapide.

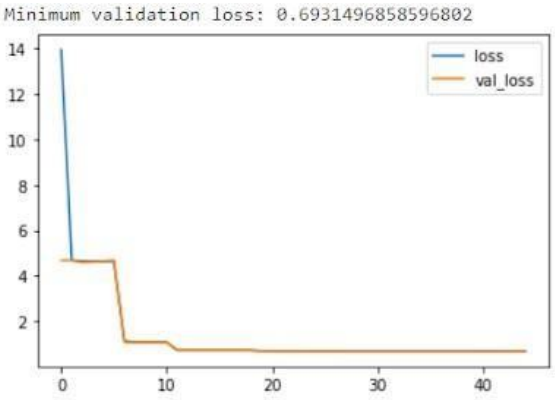
Dans un premier lieu, on opte pour le régularisateur L1 et on fait tourner le code.

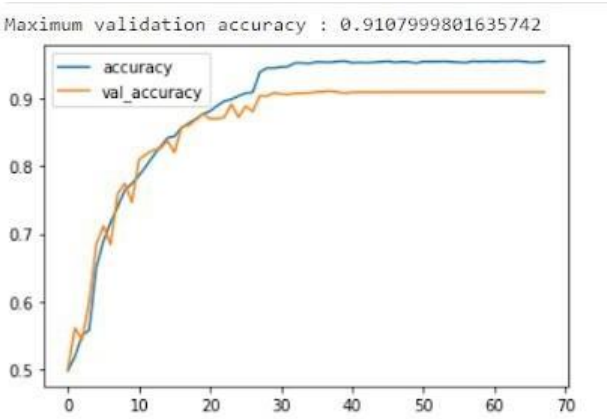



Ensuite, on essaye de combiner les deux régularisateurs (L1L2):

L'output affiche les résultats suivants :



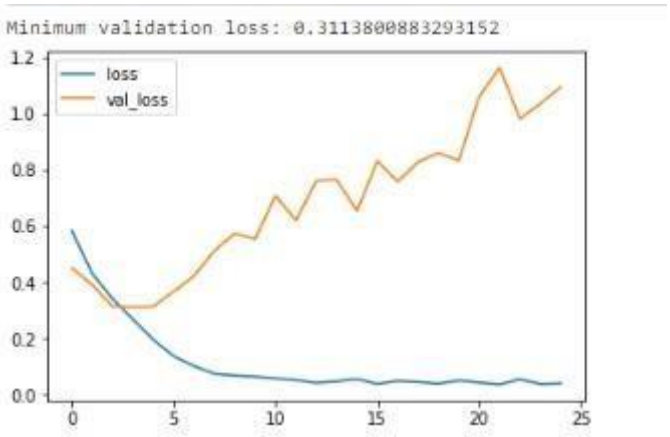

Par rapport au modèle de base qui utilise le régularisateur L2, les résultats ne sont pas assez bons et le modèle est moins performant.

## 5. Choisir les fonctions d'activation:

Nous avons essayé de remplacer la fonction d'activation 'relu' par 'leakyReLU' et fixer les autres paramètres de la manière suivante:

| | |
|---|---|
| nombre d'image pour le *training* | 20000 |
| nombre d'image pour *validation* | 5000 |
| modèle | 4 conv-pool layers (32,64,128,128)<br>2 dense layers (512,1) |
| fonction d'activation | LeakyReLU<br>sigmoid |
| nombre d'epochs | 25 |
| optimiseur | Adam, lr par défaut 0.001 |

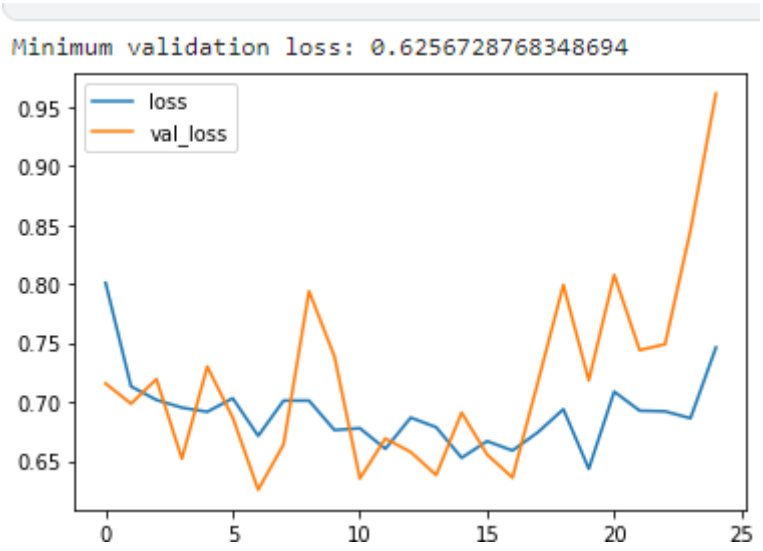
Le résultat n'était pas terrible du tout.



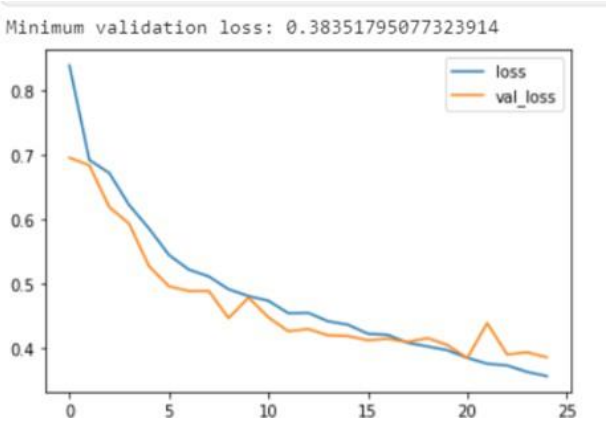

Ayant un problème d'overfitting nous avons été contraint d'utiliser du dropout et de la régularisation. Dans le modèle suivant nous avons utilisé la fonction d'activation 'swish'.

| | |
|---|---|
| modèle | 3 conv-pool layers (64,128,256)<br>2 dense layers (512,1) |
| fonction d'activation | swish<br>sigmoid |
| Dropout | 0.3 |
| régularisateur | L2=0,01 |
| nombre d'epochs | 25 |

Le résultat obtenu ne correspondait pas du tout à nos attentes. Nous avons donc rejeté ce modèle aussi.




Cependant lorsque l'on utilise la fonction d'activation Relu, on s'aperçoit que la tendance des courbes est très nettement meilleure.




Le résultat ci-dessus a été obtenu avec le modèle suivant utilisant du 'relu'.

| | |
|---|---|
| modèle | **3** conv-pool layers (32,64,128)<br>2 dense layers (512,1) |
| fonction d'activation | relu<br>sigmoid |
| **Dropout** | **0.5** |
| **régularisation** | **L2=0,01** |
| nombre d'epochs | **25** |
| optimiseur | adam lr par défaut 0.001 |

Nous avons donc décidé de garder la fonction 'Relu'.Puisque nous sommes en face d'un probleme de classification binaire , nous avons aussi opté pour la fonction d'activation 'sigmoid' au lieu de 'softmax'.

### 6. Déterminer le nombre d'epochs:

On utilise l'astuce du Early Stopping des epochs d'entraînement. Cette méthode consiste à spécifier la mesure de performance à monitorer, le déclencheur, et une fois déclenché, il arrêtera le processus d'entraînement. Une fois le modèle spécifié, on commence l'entraînement avec 100 époques.

## **Modèle final :**

Apres avoir retenu l'ensemble des combinaisons optimales de nos paramètres, nous avons obtenu le modele suivant :

Activation functions: ‘Relu’ and ‘Sigmoid’

Optimiser : ADAM

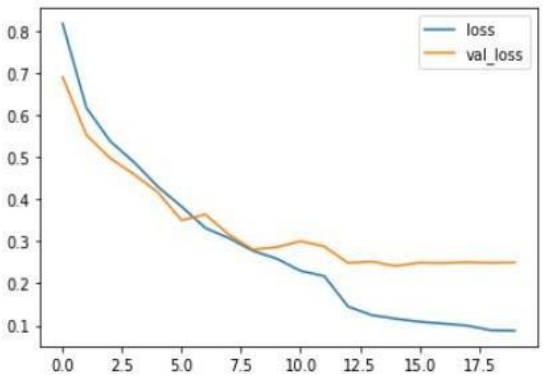
Learning Rate : 0.001

Regularisateur : L2 (alpha=0.1)

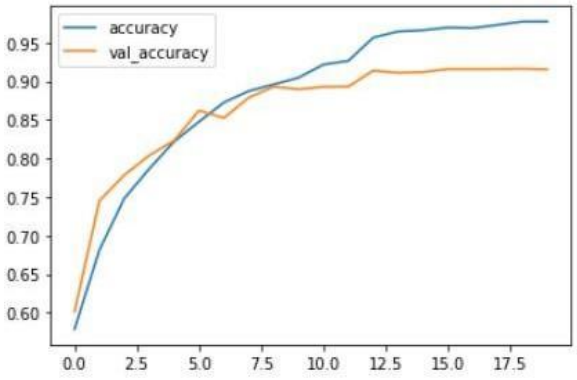
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3211776
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

Avec les outputs suivants :

```
Minimum validation loss: 0.24135653674602509
```




```
Maximum validation accuracy : 0.9164000153541565
```




```
:
```

```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 | 0.816701 | 0.57855 | 0.689571 | 0.6012 | 0.00100 |
| 1 | 0.616766 | 0.68135 | 0.552113 | 0.7452 | 0.00100 |
| 2 | 0.537518 | 0.74815 | 0.496956 | 0.7784 | 0.00100 |
| 3 | 0.487730 | 0.78640 | 0.458912 | 0.8038 | 0.00100 |
| 4 | 0.429783 | 0.82185 | 0.416475 | 0.8232 | 0.00100 |
| 5 | 0.382459 | 0.84755 | 0.349840 | 0.8620 | 0.00100 |
| 6 | 0.331423 | 0.87255 | 0.363800 | 0.8524 | 0.00100 |
| 7 | 0.307186 | 0.88735 | 0.315437 | 0.8788 | 0.00100 |
| 8 | 0.277432 | 0.89590 | 0.280174 | 0.8930 | 0.00100 |
| 9 | 0.258623 | 0.90470 | 0.286107 | 0.8898 | 0.00100 |
| 10 | 0.229829 | 0.92180 | 0.300432 | 0.8928 | 0.00100 |
| 11 | 0.217212 | 0.92645 | 0.287747 | 0.8930 | 0.00100 |
| 12 | 0.145031 | 0.95665 | 0.248331 | 0.9140 | 0.00010 |
| 13 | 0.124570 | 0.96465 | 0.251609 | 0.9110 | 0.00010 |
| 14 | 0.116196 | 0.96630 | 0.241357 | 0.9120 | 0.00010 |
| 15 | 0.109032 | 0.96975 | 0.248815 | 0.9160 | 0.00010 |
| 16 | 0.104586 | 0.96910 | 0.248106 | 0.9160 | 0.00010 |
| 17 | 0.099769 | 0.97315 | 0.250121 | 0.9160 | 0.00010 |
| 18 | 0.088855 | 0.97745 | 0.248351 | 0.9164 | 0.00001 |
| 19 | 0.087758 | 0.97745 | 0.249760 | 0.9156 | 0.00001 |

# VI- TEST :

## Choix des données d'entraînement, de validation et de test:

Après l'extraction du dossier zip et la préparation des données, on scinde notre data en deux sousensembles. Le premier est dédié à l'entraînement et contient 20000 lignes (80% des observations) et deux colonnes (deux classes). Le deuxième est dédié à la validation et contient 5000 lignes (20% des observations) et deux colonnes (deux classes).

Concernant les données de test, nous n'avons pas pu utiliser les images du dossier test1 présent sur Kaggle étant donné qu'elles ne sont pas étiquetées. Donc on a créé un nouveau dossier test100 qui contient 100 images de test étiquetées individuellement manuellement.

Avant de commencer nos prédictions, nous avons modifier notre code de base pour la preparation de notre nouveau Dataset de test :

```
test_dir = "../input/test100/test100"
```

```
#On crée une liste "test_images" qui contient le nom de tous les fichiers dans le dossier "tes
t100"
test_images = os.listdir(test_dir)
```

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

#On change le nom de la colonne pour data et test data de "0" à, respectivement, "image" et "t
est image":
data = data.rename(columns = {0:'image'})
test_data = test_data.rename(columns = {0: 'test image'})

#On ajoute le chemin d'accès avant le nom de chaque element dans chaque colonne de nos deux do
nnées:
data['image'] = data['image'].apply(lambda x: './train/' + x)
test_data['test image']= test_data['test image'].apply(lambda x: test_dir + "/"+ x)

#Notons que la colonne "image"/"test 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')
test_data['label'] = test_data['test image'].apply(lambda x: 'cat' if 'cat' in x else 'dog')

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

| | image | label |
|---|---|---|
| 0 | ./train/dog.8154.jpg | dog |
| 1 | ./train/cat.10618.jpg | cat |
| 2 | ./train/dog.7678.jpg | dog |
| 3 | ./train/cat.11192.jpg | cat |
| 4 | ./train/dog.11191.jpg | dog |

| | test image | label |
|---|---|---|
| 0 | ../input/test100/test100/dog.83.jpg | dog |
| 1 | ../input/test100/test100/dog.1.jpg | dog |
| 2 | ../input/test100/test100/cat.14.jpg | cat |
| 3 | ../input/test100/test100/dog.74.jpg | dog |
| 4 | ../input/test100/test100/cat.22.jpg | cat |

```
# Data contient 25000 observations (lignes) et 2 variables (colonnes)
display(data.shape)

# Test data contient 100 observations (lignes) et 2 variables (colonnes)
display(test_data.shape)
```

```
(25000, 2)
```

```
(100, 2)
```

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

```
test_generator = test_datagen.flow_from_dataframe(test_data,target_size=(150,150),
                                                  x_col='test image',y_col='label',cla
ss_mode='binary',batch_size= 32)
```

```
Found 100 validated image filenames belonging to 2 classes.
```

- On commence nos prédictions :

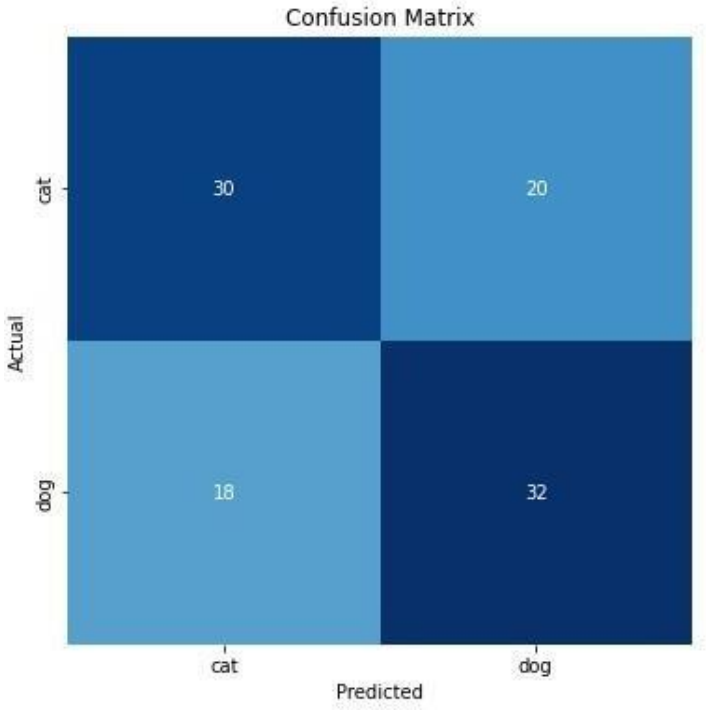
```
#On génère les predictions du modèle par rapport aux données de test:
predict = model.predict_generator(test_generator)

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 | 9.177248e-01 | 0.082275 |
| 1 | 1.704655e-02 | 0.982953 |
| 2 | 5.109867e-02 | 0.948901 |
| 3 | 4.458274e-04 | 0.999554 |
| 4 | 2.599682e-02 | 0.974003 |
| ... | ... | ... |
| 95 | 2.027345e-02 | 0.979727 |
| 96 | 9.995578e-01 | 0.000442 |
| 97 | 2.772089e-01 | 0.722791 |
| 98 | 9.057754e-07 | 0.999999 |
| 99 | 8.583114e-01 | 0.141689 |

100 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 cla
sse 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 :

```
#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       0.62      0.60      0.61        50
         dog       0.62      0.64      0.63        50

    accuracy                           0.62       100
   macro avg       0.62      0.62      0.62       100
weighted avg       0.62      0.62      0.62       100
```

# VII- Résultats

## Evaluation de notre modèle:

Grace aux modifications apportées aux different hyperparamètres du modèle imlementé au début , nous atteignons une précision de 91,64 % , ce qui est considéré comme très précis sachant couplé à minimum de validation Loss assez faible.

Nous avons à peu preès 20 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).

On constate que nos prédictions positives sont plus présentes que celles négatives avec un taux de 62% de précision. Nous remarquons également que notre modèle reconnait légèrement mieux les chiens que les chats 61% vs 63% de précision.