

# **M2 Logiciel sûrs - IA**

---

**TP1 : cats and dogs**

---

**Réalisé par :**

**Merzeg Ramzi  
Khitous Rania**

**2022/2023**

En partant d'un code personnel non extrait de Kaggle, nous avons modifié quelques paramètres du modèle tels que : taux d'apprentissage, régularisation, optimiseurs, nombre de couches dans CNN, fonction d'activation, époques, ensemble de données pour la formation afin de l'améliorer et d'obtenir les meilleures résultats possible.

Lien git vers le code : [https://github.com/ramzimerzeg/cats\\_and\\_dogs](https://github.com/ramzimerzeg/cats_and_dogs) ( Modèle 1)

### A. Parties du codes non-changeable :

#### 1. Clonage du dataset :

Nous avons choisi de travailler avec un dataset contenant 20000 images d'apprentissage et 5000 images de test. Ce dataset est publique et disponible sur git :

`https://github.com/laxmimerit/dog-cat-full-dataset.git`

```
[1] !git clone https://github.com/laxmimerit/dog-cat-full-dataset.git

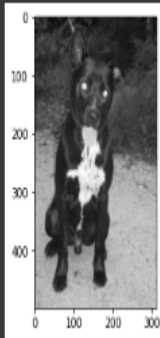
Cloning into 'dog-cat-full-dataset'...
remote: Enumerating objects: 25027, done.
remote: Total 25027 (delta 0), reused 0 (delta 0), pack-reused 25027
Receiving objects: 100% (25027/25027), 541.62 MiB | 17.09 MiB/s, done.
Resolving deltas: 100% (5/5), done.
Checking out files: 100% (25001/25001), done.
```

#### 2. Prétraitement

Puisque dans notre cas les couleurs des images n'affectent pas la décision du modèle d'apprentissage, on a éliminé les couleurs(extraction du RGB) lors de la visualisation. Nous avons effectué ce traitement sur les deux dataset **d'entraînement** et **test**.

```
DATA_DIR2 = "/content/dog-cat-full-dataset/data/test"
CATEGORIES = ["dogs", "cats"]

for category in CATEGORIES:
    path = os.path.join(DATA_DIR2, category)
    for img in os.listdir(path):
        img_array = cv2.imread(os.path.join(path, img), cv2.IMREAD_GRAYSCALE)
        plt.imshow(img_array, cmap='gray')
        plt.show()
        break
    break
```



```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2

DATA_DIR = "/content/dog-cat-full-dataset/data/train"
CATEGORIES = ["dogs", "cats"]

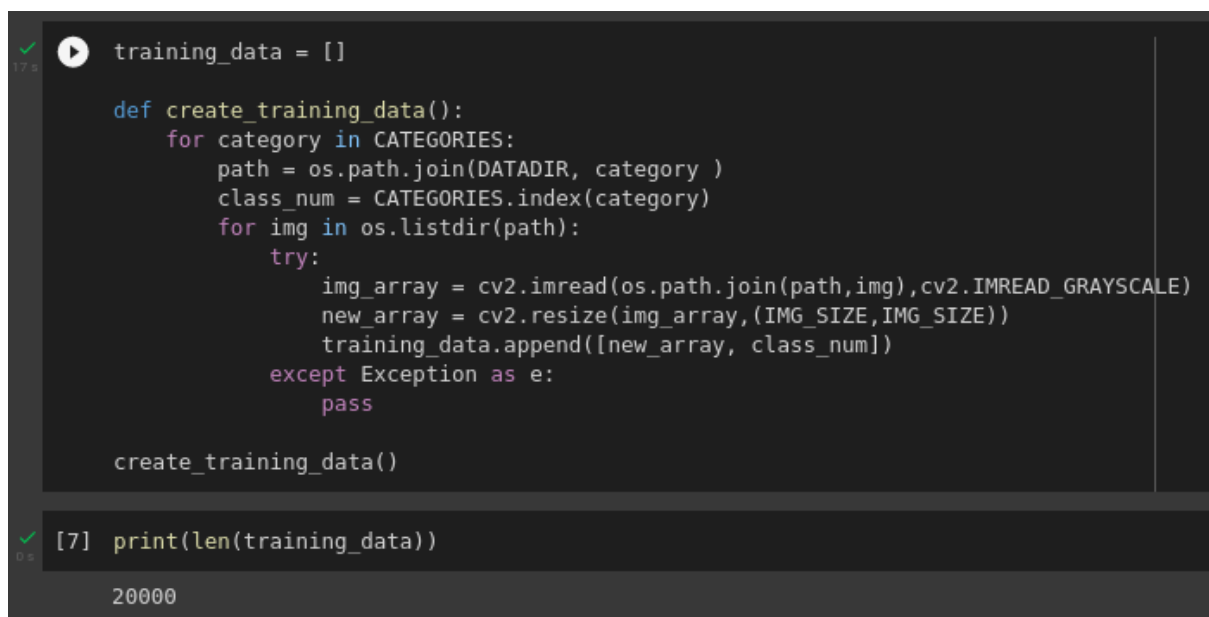
for category in CATEGORIES:
    path = os.path.join(DATA_DIR, category)
    for img in os.listdir(path):
        img_array = cv2.imread(os.path.join(path, img), cv2.IMREAD_GRAYSCALE)
        plt.imshow(img_array, cmap='gray')
        plt.show()
        break
    break
```



On a unifié les tailles de l'ensemble des images d'**entraînement** et de **tests** en sorte que chacune d'elles aura une dimension de **100 x 100**.



Après avoir effectué les deux traitements ci-dessous, nous avons commencé l'entraînement du modèle avec **20000 images**.



Vu que les images des chats des chiens dans le dataset utilisé étaient dans **deux différents dossiers**, on a mélangé toutes les images afin de faire en sorte que le modèle ne reçoit pas des images du même type (portant le même label) en séquentiel ce qui nous entraînera à avoir un modèle à faible précision.

```
[8] import random
     random.shuffle(training_data)
```

Création des données d'entraînement avec caractéristique et label.

```
[39] for features, label in training_data:
      X.append(features)
      Y.append(label)
      #np.array((Y,label))
X = np.array(X).reshape(-1, IMG_SIZE, IMG_SIZE,1)
Y = np.array(Y)
```

## B. Parties de codes changeable :

Notre modèle de départ était configuré comme suit :

- Fonction de loss : **binary\_crossentropy**
- optimizer : **adam**
- metrics : **accuracy**
- batch size = **32**
- epoch = **13**
- validation split = **0.1**
- 7 couches ( 1 couche d'entrée + 6 couches intermédiaires + 8 couche de sortie) :
  - Couche 1 : flatten.
  - Couche 2 : avec **256** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 3 : avec **256** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 4 : avec **128** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 5 : avec **128** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 6 : avec **64** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 7 : avec **32** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 8 : **1** neurone fonction **Softmax**.

```

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D
import pickle

X = pickle.load(open("X.pickle","rb"))
Y = pickle.load(open("Y.pickle","rb"))

X = X/255.0

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten()) #Flatten the images! Could be done with numpy reshape
model.add(tf.keras.layers.Dense(256, activation=tf.nn.relu, input_shape= X.shape[1:]))
model.add(tf.keras.layers.Dense(256, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(32, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(1, activation=tf.nn.softmax))

model.compile(loss="binary_crossentropy",
              optimizer="opt",
              metrics=['accuracy'])

[ ] history = model.fit(X,Y,batch_size=10,epochs=20, validation_split=0.1)

```

Les résultats obtenus avec cette configuration n'étaient **pas pertinents** pour nous. On a eu **59%** d'accuracy et **60%** de loss.

```

Epoch 1/20
1800/1800 [=====] - 10s 4ms/step - loss: 0.7007 - accuracy: 0.5042 - val_loss: 0.6933 - val_accuracy: 0.5010
Epoch 2/20
1800/1800 [=====] - 7s 4ms/step - loss: 0.6933 - accuracy: 0.4990 - val_loss: 0.6911 - val_accuracy: 0.5265
Epoch 3/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6888 - accuracy: 0.5175 - val_loss: 0.6864 - val_accuracy: 0.5240
Epoch 4/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6853 - accuracy: 0.5274 - val_loss: 0.6869 - val_accuracy: 0.5155
Epoch 5/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6859 - accuracy: 0.5292 - val_loss: 0.6865 - val_accuracy: 0.5370
Epoch 6/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6848 - accuracy: 0.5382 - val_loss: 0.6784 - val_accuracy: 0.5520
Epoch 7/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6826 - accuracy: 0.5399 - val_loss: 0.6820 - val_accuracy: 0.5400
Epoch 8/20
1800/1800 [=====] - 6s 4ms/step - loss: 0.6843 - accuracy: 0.5313 - val_loss: 0.6816 - val_accuracy: 0.5365
Epoch 9/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6842 - accuracy: 0.5216 - val_loss: 0.6816 - val_accuracy: 0.5500
Epoch 10/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6825 - accuracy: 0.5383 - val_loss: 0.6786 - val_accuracy: 0.5445
Epoch 11/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6836 - accuracy: 0.5263 - val_loss: 0.6825 - val_accuracy: 0.5330
Epoch 12/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6817 - accuracy: 0.5349 - val_loss: 0.6798 - val_accuracy: 0.5550
Epoch 13/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6829 - accuracy: 0.5318 - val_loss: 0.6809 - val_accuracy: 0.5410
Epoch 14/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6827 - accuracy: 0.5351 - val_loss: 0.6801 - val_accuracy: 0.5380
Epoch 15/20
1800/1800 [=====] - 7s 4ms/step - loss: 0.6813 - accuracy: 0.5354 - val_loss: 0.6773 - val_accuracy: 0.5500
Epoch 16/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6761 - accuracy: 0.5592 - val_loss: 0.6754 - val_accuracy: 0.5710
Epoch 17/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6669 - accuracy: 0.5899 - val_loss: 0.6629 - val_accuracy: 0.5880
Epoch 18/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6618 - accuracy: 0.5976 - val_loss: 0.6598 - val_accuracy: 0.6050
Epoch 19/20
1800/1800 [=====] - 6s 4ms/step - loss: 0.6583 - accuracy: 0.6086 - val_loss: 0.6624 - val_accuracy: 0.5995
Epoch 20/20
1800/1800 [=====] - 6s 3ms/step - loss: 0.6579 - accuracy: 0.6043 - val_loss: 0.6623 - val_accuracy: 0.5945

```

Après avoir effectué plusieurs essais, nous nous sommes mis d'accord de vous parler sur les 2 plus tentatives ayant des résultats pertinents.

## Modèle 1 (le bon):

- Fonction de loss : **binary\_crossentropy**
- optimizer : **adam**
- metrics : **accuracy**
- batch size = **25**
- epoch = **10**
- validation split = **0.1**
- 7 couches ( 1 couche d'entrée + 5 couches intermédiaires + 1 couche de sortie) :
  - Couche 1 : avec **256** neurones et une fonction d'activation conv2D **relu** avec un max pooling (2,2)
  - Couche 2 : avec **256** neurones et une fonction d'activation conv2D **relu** avec un max pooling (2,2)
  - Couche 3 : avec **128** neurones et une fonction d'activation conv2D **relu** avec un max pooling (2,2)
  - Couche 4 : avec **64** neurones et une fonction d'activation conv2D **relu** avec un max pooling (2,2)
  - Couche 5 : avec un **flatten** de densité **64** qui transforme le tout en vecteur.
  - Couche 7 : **1** neurone fonction **Sigmoid**.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D
import pickle

X = pickle.load(open("X.pickle","rb"))
Y = pickle.load(open("Y.pickle","rb"))

X = X/255.0

model = Sequential()
model.add(Conv2D(256,(3,3),activation='relu',input_shape = X.shape[1:]))
model.add(MaxPooling2D(2,2))

model.add(Conv2D(256,(3,3),activation='relu'))
model.add(MaxPooling2D(2,2))

model.add(Conv2D(128,(3,3),activation='relu'))
model.add(MaxPooling2D(2,2))

model.add(Conv2D(128,(3,3),activation='relu'))
model.add(MaxPooling2D(2,2))

model.add(Conv2D(64,(3,3),activation='relu'))
model.add(MaxPooling2D(2,2))

model.add(Flatten())
model.add(Dense(64))

model.add(Dense(1,activation = 'sigmoid'))

model.compile(loss="binary_crossentropy",
              optimizer="adam",
              metrics=['accuracy'])

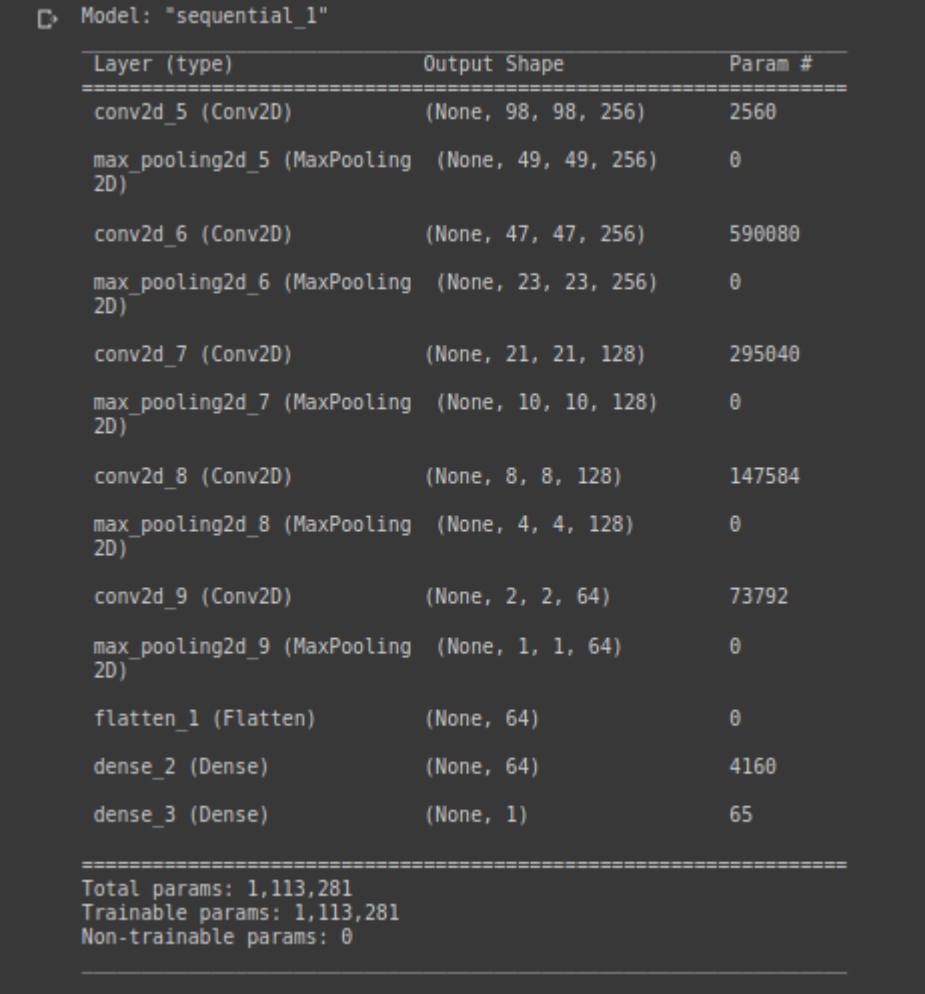
model.fit(X,Y,batch_size=32,epochs=10, validation_split=0.1)
```

Afin de s'assurer de la réception des meilleures caractéristiques des images en entrées de chacune des couches, nous sommes parties sur un modèle à plusieurs couches convolutives ayant un nombre de neurones et une taille du noyau différents.

Ce modèle comporte deux couches convolutives successives de 256 neurones et noyau 3 x 3 chacune, 2 couches convolutives plus petites à 128 neurones et noyau de 3 x 3 une dernière couche convolutive de 64 neurones avec un noyau de 3 x 3.

Nous avons équipé ces couches d'une fonction linéaire rectifiée 'relu' pour éviter d'avoir des matrices caractéristiques de grande taille et donc optimiser notre efficacité de calcul.

Finalement, pour l'utilisation des caractéristiques extraites, nous avons dû passer d'une forme tensorielle convolutive à une forme vectorielle. Nous avons ajouté une couche d'aplatissement de 64 neurones qui précède un neurone à une fonction d'activation sigmoïde qui permet d'obtenir la probabilité de classification.



Model: "sequential\_1"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 98, 98, 256)	2560
max_pooling2d_5 (MaxPooling 2D)	(None, 49, 49, 256)	0
conv2d_6 (Conv2D)	(None, 47, 47, 256)	590080
max_pooling2d_6 (MaxPooling 2D)	(None, 23, 23, 256)	0
conv2d_7 (Conv2D)	(None, 21, 21, 128)	295040
max_pooling2d_7 (MaxPooling 2D)	(None, 10, 10, 128)	0
conv2d_8 (Conv2D)	(None, 8, 8, 128)	147584
max_pooling2d_8 (MaxPooling 2D)	(None, 4, 4, 128)	0
conv2d_9 (Conv2D)	(None, 2, 2, 64)	73792
max_pooling2d_9 (MaxPooling 2D)	(None, 1, 1, 64)	0
flatten_1 (Flatten)	(None, 64)	0
dense_2 (Dense)	(None, 64)	4160
dense_3 (Dense)	(None, 1)	65

=====  
Total params: 1,113,281  
Trainable params: 1,113,281  
Non-trainable params: 0  
=====

Avec cette configuration nous avons réussi à avoir un très bon score **85%** d'accuracy et **36%** de loss en entraînement.

```

Epoch 1/10
563/563 [=====] - 36s 63ms/step - loss: 0.6931 - accuracy: 0.5022 - val_loss: 0.6967 - val_accuracy: 0.4945
Epoch 2/10
563/563 [=====] - 35s 62ms/step - loss: 0.6856 - accuracy: 0.5469 - val_loss: 0.6676 - val_accuracy: 0.6205
Epoch 3/10
563/563 [=====] - 34s 61ms/step - loss: 0.6285 - accuracy: 0.6517 - val_loss: 0.5843 - val_accuracy: 0.6965
Epoch 4/10
563/563 [=====] - 35s 62ms/step - loss: 0.5291 - accuracy: 0.7384 - val_loss: 0.4716 - val_accuracy: 0.7695
Epoch 5/10
563/563 [=====] - 35s 62ms/step - loss: 0.4361 - accuracy: 0.7969 - val_loss: 0.4001 - val_accuracy: 0.8260
Epoch 6/10
563/563 [=====] - 35s 62ms/step - loss: 0.3654 - accuracy: 0.8372 - val_loss: 0.3658 - val_accuracy: 0.8370
Epoch 7/10
563/563 [=====] - 35s 62ms/step - loss: 0.3168 - accuracy: 0.8635 - val_loss: 0.3638 - val_accuracy: 0.8365
Epoch 8/10
563/563 [=====] - 36s 64ms/step - loss: 0.2639 - accuracy: 0.8874 - val_loss: 0.3402 - val_accuracy: 0.8605
Epoch 9/10
563/563 [=====] - 35s 62ms/step - loss: 0.2232 - accuracy: 0.9070 - val_loss: 0.3417 - val_accuracy: 0.8575
Epoch 10/10
563/563 [=====] - 35s 62ms/step - loss: 0.1907 - accuracy: 0.9190 - val_loss: 0.3613 - val_accuracy: 0.8500
<keras.callbacks.History at 0x7f5f696b04d0>

```

```

import matplotlib.pyplot as plt

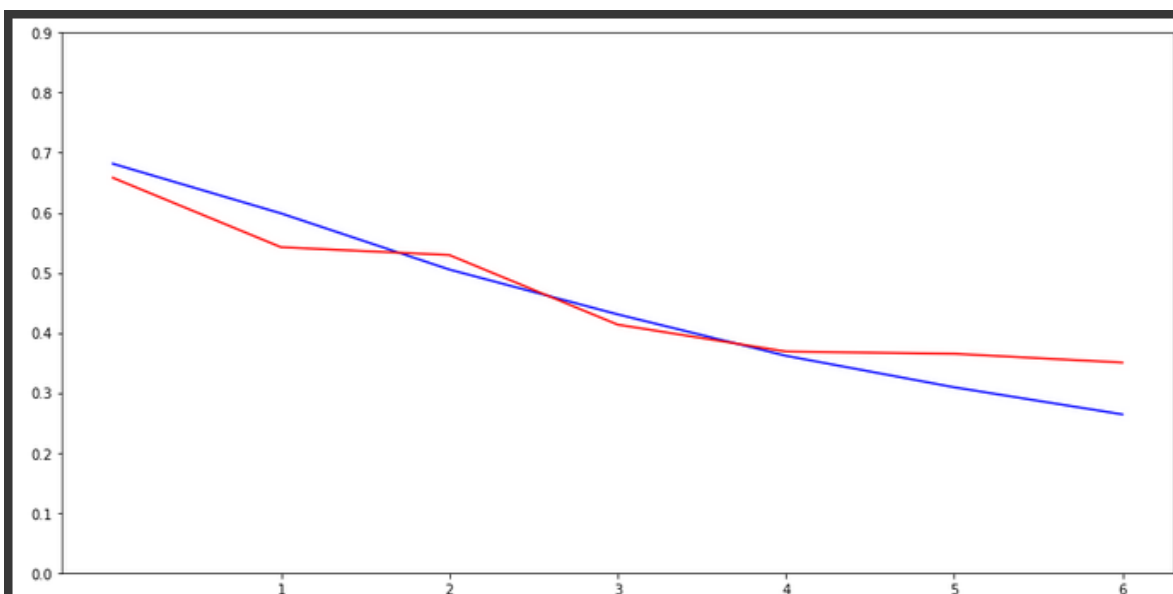
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 12))
ax1.plot(history.history['loss'], color='b', label="Training loss")
ax1.plot(history.history['val_loss'], color='r', label="validation loss")
ax1.set_xticks(np.arange(1, 7, 1))
ax1.set_yticks(np.arange(0, 1, 0.1))

ax2.plot(history.history['accuracy'], color='b', label="Training accuracy")
ax2.plot(history.history['val_accuracy'], color='r', label="Validation accuracy")
ax2.set_xticks(np.arange(1, 7, 1))

legend = plt.legend(loc='best', shadow=True)
plt.tight_layout()
plt.show()

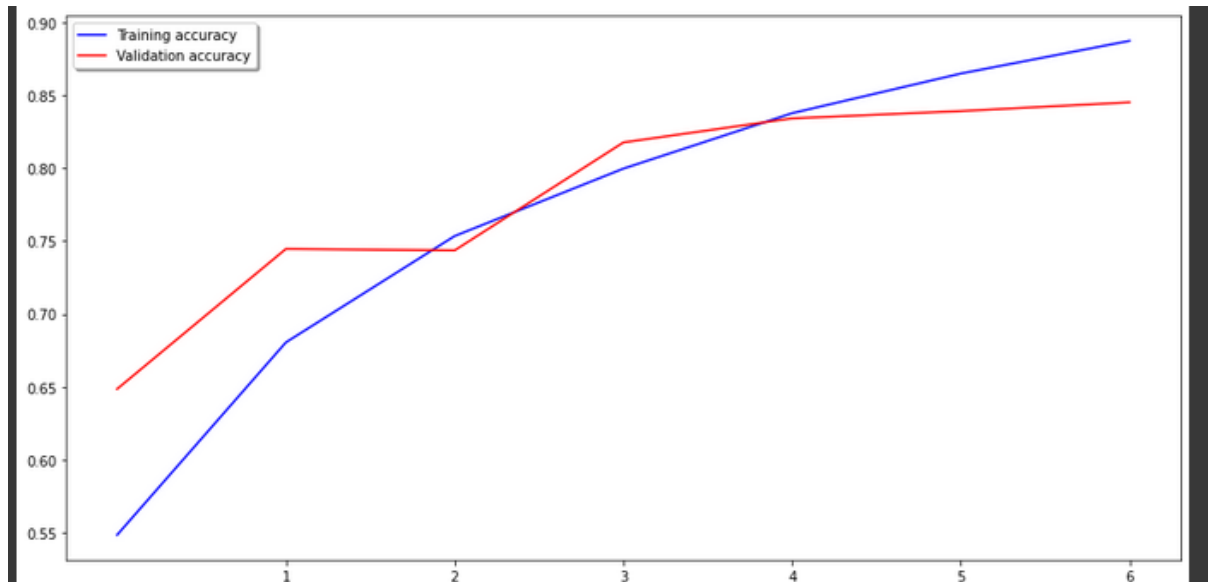
```

voici ci joint le graph du loss en bleu comparé au validation loss en rouge, on peut remarquer que le loss commence à 69% et ca descends en 7 epoch pour arriver à 19% alors que le validation loss commence à 69% et évolue en 7 epoch a 36%





et voici ci joint le graph de l'accuracy en bleu comparé au validation accuracy en rouge, on peut remarquer que l'accuracy commence à 55% et ça évolue en 7 epoch pour arriver à 91% alors que la validation accuracy commence à 65% et évolue en 7 epoch à 85%



en testant le modèle avec le donnée de test qui sont à 5000 images on peut voir qu'on a 86.97% d'accuracy et 30% de loss

```
[ ] result = model.evaluate(X_test,Y_test)
print(result)

157/157 [=====] - 4s 23ms/step - loss: 0.3046 - accuracy: 0.8698
[0.30455535650253296, 0.8697999715805054]

▶ print('> %.3f' % (result[1] * 100.0))

📄 > 86.980
```

**Matrice de confusion :**

```
[ ] from sklearn.metrics import confusion_matrix

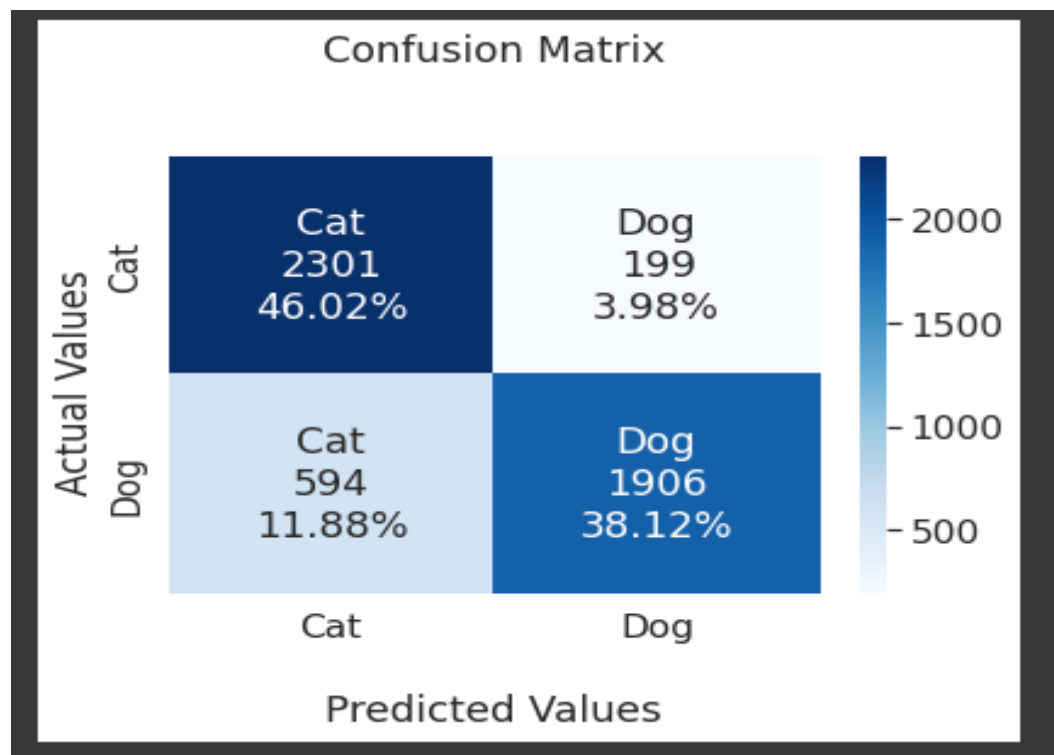
pred = model.predict(X_test)
pred = np.where(pred > 0.5, 1, 0) #<--to get the binary category
y_pred = np.argmax(pred, axis=1)

conf_mat = confusion_matrix(Y_test, pred)

157/157 [=====] - 3s 21ms/step

▶ print(conf_mat)

📄 [[2256 244]
   [ 407 2093]]
```



## Modèle 2 :

- Fonction de loss : **binary\_crossentropy**
- optimizer : **adam**
- metrics : **accuracy**
- batch size = **100**
- epoch = **8**
- validation split = **0.1**
- 7 couches ( 1 couche d'entrée + 5 couches intermédiaires + 1 couche de sortie) :
  - Couche 1 : avec **400** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 2 : avec **300** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 3 : avec **200** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 4 : avec **100** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 4 : avec **50** neurones et une fonction d'activation conv2D **relu** avec un max pooling (3,3)
  - Couche 5 : avec un **flatten** de densité **10** qui transforme le tout en vecteur.
  - Couche 7 : **1** neurone fonction **Sigmoid**.

```
X = pickle.load(open("X.pickle","rb"))
Y = pickle.load(open("Y.pickle","rb"))

X = X/255.0

model = Sequential()
model.add(Conv2D(400,(3,3),input_shape = X.shape[1:]))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(300,(3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(200,(3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(100,(3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(50,(3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(10))

model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss="binary_crossentropy",
              optimizer="adam",
              metrics=['accuracy'])

history = model.fit(X,Y,batch_size=100,epochs=8, validation_split=0.1)
```

Architecture de ce modèle :

```
model.summary()
```

Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 98, 98, 400)	4000
activation (Activation)	(None, 98, 98, 400)	0
max_pooling2d (MaxPooling2D)	(None, 49, 49, 400)	0
conv2d_1 (Conv2D)	(None, 47, 47, 300)	1080300
activation_1 (Activation)	(None, 47, 47, 300)	0
max_pooling2d_1 (MaxPooling2D)	(None, 23, 23, 300)	0
conv2d_2 (Conv2D)	(None, 21, 21, 200)	540200
activation_2 (Activation)	(None, 21, 21, 200)	0
max_pooling2d_2 (MaxPooling2D)	(None, 10, 10, 200)	0
conv2d_3 (Conv2D)	(None, 8, 8, 100)	180100
activation_3 (Activation)	(None, 8, 8, 100)	0
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 100)	0
conv2d_4 (Conv2D)	(None, 2, 2, 50)	45050
activation_4 (Activation)	(None, 2, 2, 50)	0
max_pooling2d_4 (MaxPooling2D)	(None, 1, 1, 50)	0
flatten (Flatten)	(None, 50)	0
dense (Dense)	(None, 10)	510
dense_1 (Dense)	(None, 1)	11
activation_5 (Activation)	(None, 1)	0
Total params: 1,850,171		
Trainable params: 1,850,171		
Non-trainable params: 0		

Avec cette configuration nous avons réussi à avoir un bon score de **86%** d'accuracy et **34%** de loss en **entraînement**.

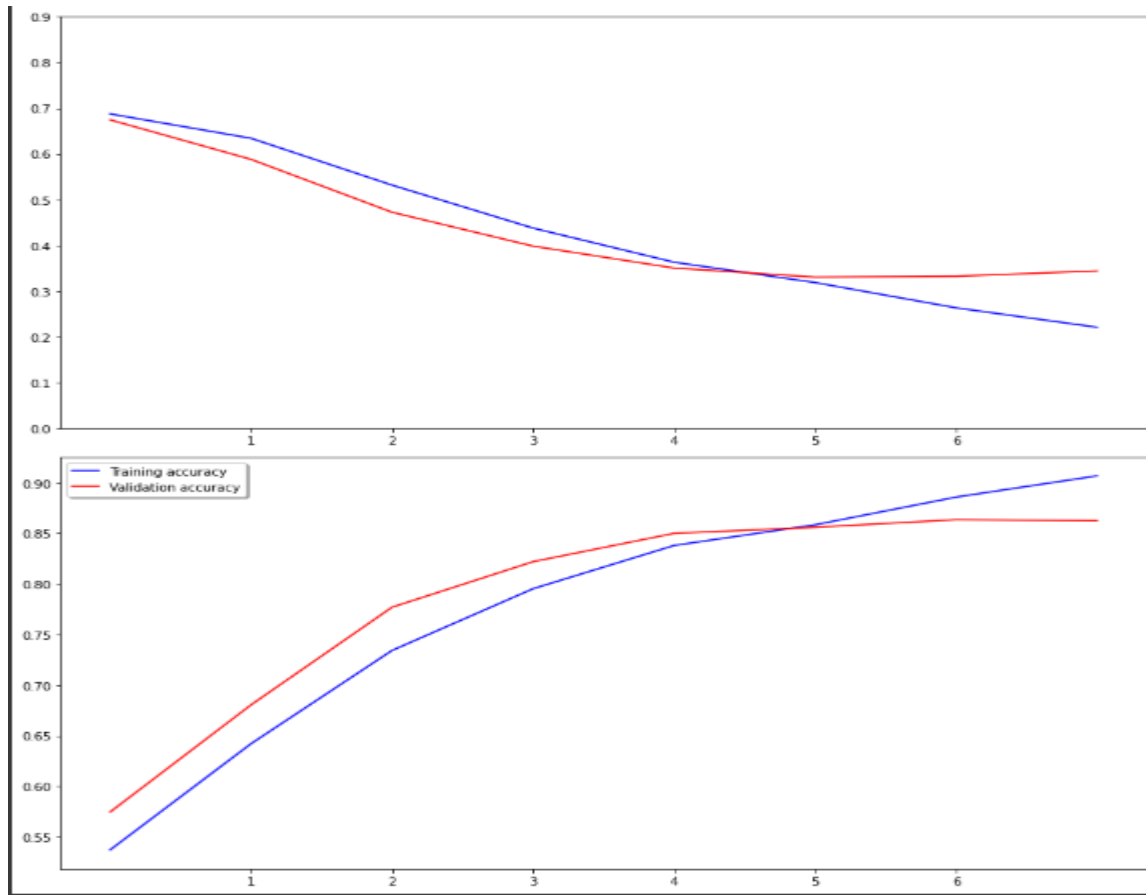
```
Epoch 1/8
180/180 [=====] - 68s 307ms/step - loss: 0.6877 - accuracy: 0.5371 - val_loss: 0.6747 - val_accuracy: 0.5745
Epoch 2/8
180/180 [=====] - 57s 319ms/step - loss: 0.6343 - accuracy: 0.6421 - val_loss: 0.5882 - val_accuracy: 0.6805
Epoch 3/8
180/180 [=====] - 59s 326ms/step - loss: 0.5319 - accuracy: 0.7343 - val_loss: 0.4725 - val_accuracy: 0.7770
Epoch 4/8
180/180 [=====] - 59s 327ms/step - loss: 0.4379 - accuracy: 0.7953 - val_loss: 0.3984 - val_accuracy: 0.8220
Epoch 5/8
180/180 [=====] - 59s 328ms/step - loss: 0.3633 - accuracy: 0.8380 - val_loss: 0.3505 - val_accuracy: 0.8500
Epoch 6/8
180/180 [=====] - 59s 326ms/step - loss: 0.3188 - accuracy: 0.8584 - val_loss: 0.3307 - val_accuracy: 0.8560
Epoch 7/8
180/180 [=====] - 59s 328ms/step - loss: 0.2635 - accuracy: 0.8858 - val_loss: 0.3326 - val_accuracy: 0.8635
Epoch 8/8
180/180 [=====] - 59s 326ms/step - loss: 0.2210 - accuracy: 0.9068 - val_loss: 0.3442 - val_accuracy: 0.8625
```

Les résultats obtenus après le test n'étaient pas très pertinents, **81%** d'accuracy et **51%** de loss ce qui reste très élevé pour un pourcentage de taux de mauvaise prédiction.

```
[28] result = model.evaluate(X_test,Y_test)
print(result)

157/157 [=====] - 7s 37ms/step - loss: 51.2974 - accuracy: 0.8190
[51.297428131103516, 0.8190000057220459]
```

## Graphe de variation de l'accuracy training et validation accuracy :



## Matrice de confusion :

```
[33] from sklearn.metrics import confusion_matrix

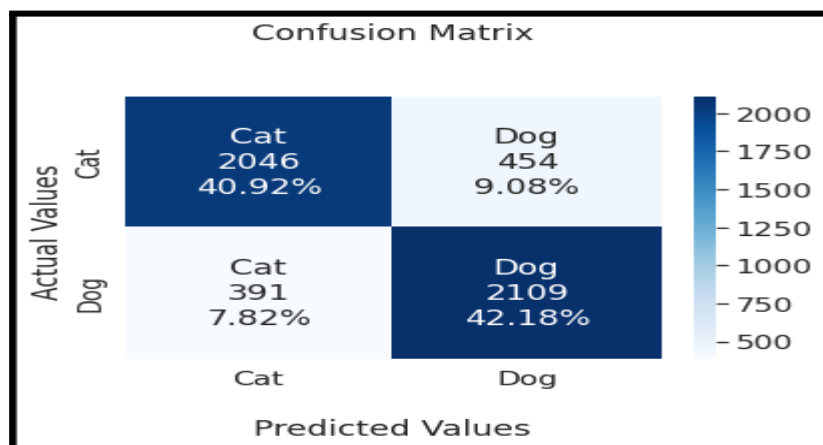
pred = model.predict(X_test)
pred = np.where(pred > 0.5, 1, 0) #<--to get the binary category
y_pred = np.argmax(pred, axis=1)

conf_mat = confusion_matrix(Y_test, pred)

157/157 [=====] - 5s 34ms/step

[34] print(conf_mat)

[[2046  454]
 [ 391 2109]]
```



## **Conclusion :**

Pour réussir à avoir de bons résultats, nous avons essayé plusieurs paramétrages ou on a changé et essayé plusieurs paramètres : learning rate, regularization, optimizers, number of layers in CNN, activation fonction, epochs, dataset for training, validation and test. Mais au final, on a réussi à faire un meilleur score (représenté dans la modèle 1) **86% acc et 30%.**