Projet 8 :
Classer des Images

Florent Margery

Formation ingénieur Machine Learning
21 Juin 2022

Objectifs du Projet

- Réaliser un algorithme de détection de chiens sur une photo.
- Effectuer une recherche sur l’ état de l’art dans ce domaine.
- Comprendre et utiliser le preprocessing d’images.
- Comprendre et utiliser la Data Augmentation.
- Essayer plusieurs modèles différents (modèle complet, transfer learning).

Environnement de travail : Tensorflow et Keras

Environnement de travail

Plusieurs librairies existent pour le traitement d'image (Computer vision) dans le monde du deep learning : Open CV, PyTorch, Keras, etc….

Choix : **Keras**

Le module Keras est conçu pour le Deep Learning et les réseaux neuronaux .

- Parfait pour de la computer vision
- Intuitif
- Complet
- Documenté

Environnement de travail

Machine de travail :

- Macbook pro M1 Pro

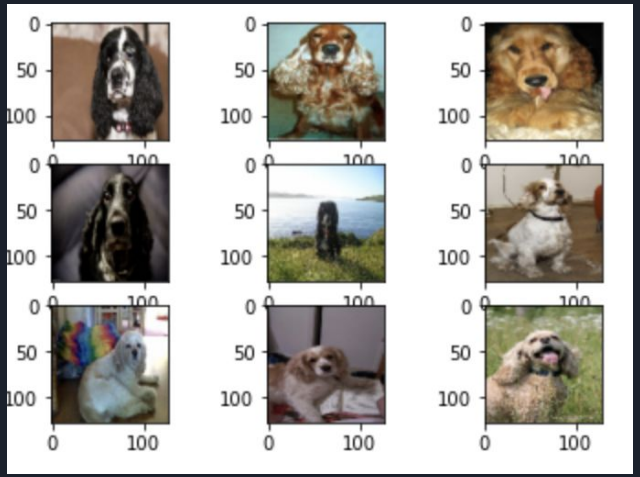
Afin de pouvoir utiliser les pleines ressources de ma machine de travail :

- Installation de METAL, permettant la reconnaissance du GPU par tensorflow (réduit les temps de calcul).

⇒ Création d'un environnement de travail dédié, avec les modules nécessaires.

Le Dataset

Le Dataset



Stanford Dogs Dataset :

- 120 races de chiens
- 20,500 images de chiens réparties sur les 120 classes
- Autant d'annotations que d'images (pour le training des modèles).

Points d'attention :

- Dataset volumineux.
- 120 classes : Problème de classification relativement complexe.

Preprocessing

Preprocessing

Afin de rendre notre projet faisable, et de rendre les temps de calcul raisonnables :

⇒ Nombre de races de chiens = 20.

```
#On définit le nombre de classes
n_classes = 20
```

Ensuite, on charge les images et leur annotations dans des listes :

Target Size : 128,128,3

Normalisation :

```
# On initialise l'array images qui prendra ensuite toutes nos images.
images = []

# On parcours le dossier père :
for imgDir in os.listdir("/Users/florentmargery/Documents/Open Classroom/Projet 6/Images/10classes/"):
    directory = "/Users/florentmargery/Documents/Open Classroom/Projet 6/Images/10classes/" + imgDir
    # On se place dans le dossier fils
    os.chdir(directory)
    # POur chaque image du dossier fils, on la charge au format souhaité, on la convertie en array et on la normalise
    # avant de l'ajoutere à notre array d'images.
    for image in os.listdir(directory):
        loadedImg = load_img(image,target_size=(128, 128,3))
        img = img_to_array(loadedImg)
        images.append(img/255.) #/255.
```

Preprocessing

X = images

⇒ Comprends les images sous forme de matrices (128,128,3)

y = annotations

⇒ Pour chaque image correspond une étiquette (ex : “Malinois”)

<u>Encodage des étiquettes :</u>

1- Label_encoder() ⇒ Pour chacune des 20 classes, attribue un chiffre entre 0 et 19.

2- OneHotEncoder() ⇒ Créer un vecteur de taille n, avec toute valeur nulle sauf la classe concernée = 1.

sauvegarde des correspondances du label_encoder :

On enregistre notre encoder

```
: np.save('classes.npy', label_encoder.classes_)
```

Preprocessing

Séparation des données en 2 :

1- Training set

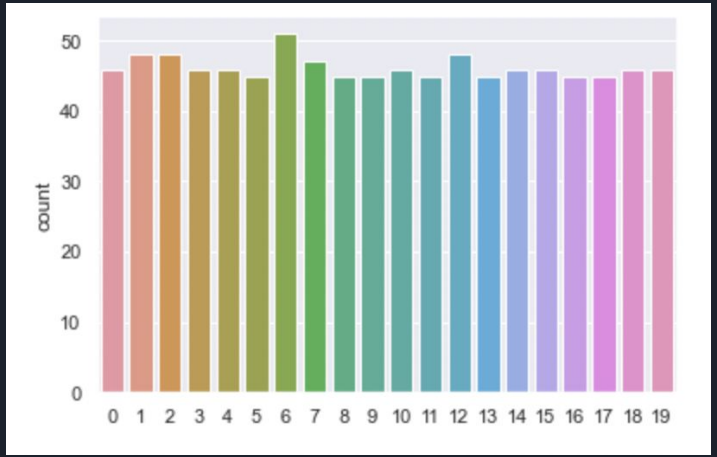
2- Validation set

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.3,
                                                 shuffle=True, stratify=y)

y_train = np.array(y_train)
X_train = np.array(X_train)
X_test = np.array(X_test)
y_test = np.array(y_test)
```

shuffle = True : Permet de mélanger les données.

stratify = y : Permet d'avoir une répartition égale des 20 classes dans les jeux d'entraînements et de validations.



Preprocessing

Afin d'optimiser les performances de nos algorithmes, il est intéressant de manipuler nos images avec certaines techniques :

Whitening :

Redimensionnement de la taille des images :

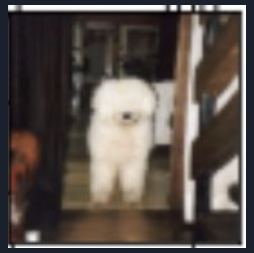
Initialement les images sont sous forme 128, 128,3 avec pour chaque pixel une valeur comprise entre **0 et 255**.

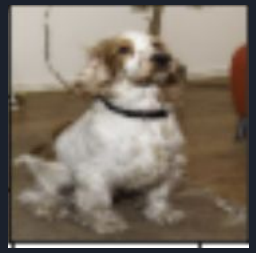
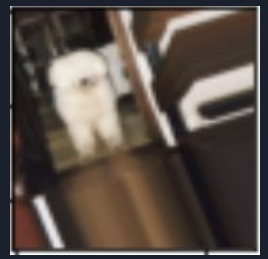
Le redimensionnement peut permettre de transformer les données pour qu'elles soient comprises entre **0 et 1**.

Preprocessing : Data Augmentation

Il est également possible d'augmenter notre data set, en appliquant de légères transformations à nos images existantes : Data augmentation.

Exemples de transformations : rotations, zooms, flips, shifts….



Preprocessing : Data Augmentation

Concrètement, on utilise la librairie Keras et l'objet ImageDataGenerator:

```
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        zca_whitening= False,  # apply ZCA whitening
        rotation_range=20,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.5, # Randomly zoom image
        width_shift_range=0.3,  # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.3,  # randomly shift images vertically (fraction of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False
        )  # randomly flip images


datagen.fit(X_train)
```

CNN :
Convolutional
Neural network

CNN : Convolution neural Networks

1- Identification des Features

⇒ Couche de Convolution : Identification des features de taille (n,n).

2- Réduction de la taille des images, tout en conservant les features identifiées.

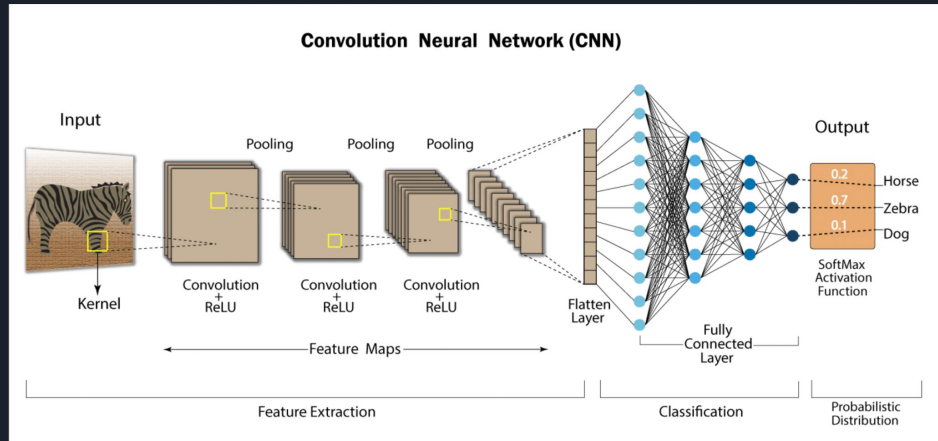
⇒ Couche de Pooling (souvent placée entre deux couches de convolution)

3- Vectorisation de nos features map en un vecteur “Plat”

⇒ Flatten Layer

4- Classification des images

⇒ Fully Connected Layer



CNN : Convolutional neural Networks

Création d'un modèle 'Baseline' :

```
from tensorflow.keras import datasets, layers, models
model_bl2 = models.Sequential()
model_bl2.add(layers.Conv2D(128, (3, 3), activation='relu',
                            input_shape=(128, 128, 3)))
model_bl2.add(layers.MaxPooling2D((2, 2)))
model_bl2.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_bl2.add(layers.MaxPooling2D((2, 2)))
model_bl2.add(layers.Conv2D(32, (3, 3), activation='relu'))

model_bl2.add(layers.Flatten())
model_bl2.add(layers.Dense(32, activation='relu'))
model_bl2.add(layers.Dense(n_classes,activation='softmax'))
```

Compilation du modèle :

```
optimizer = SGD(lr=0.01, momentum=0.9)
```

```
model_bl2.compile(loss="categorical_crossentropy",
                  optimizer=optimizer, metrics=["accuracy"])
```

CNN : Convolutional neural Networks

Création d'un callback : permet de changer les paramètres d'exécution du modèle en cours de route si certaines conditions sont remplies :

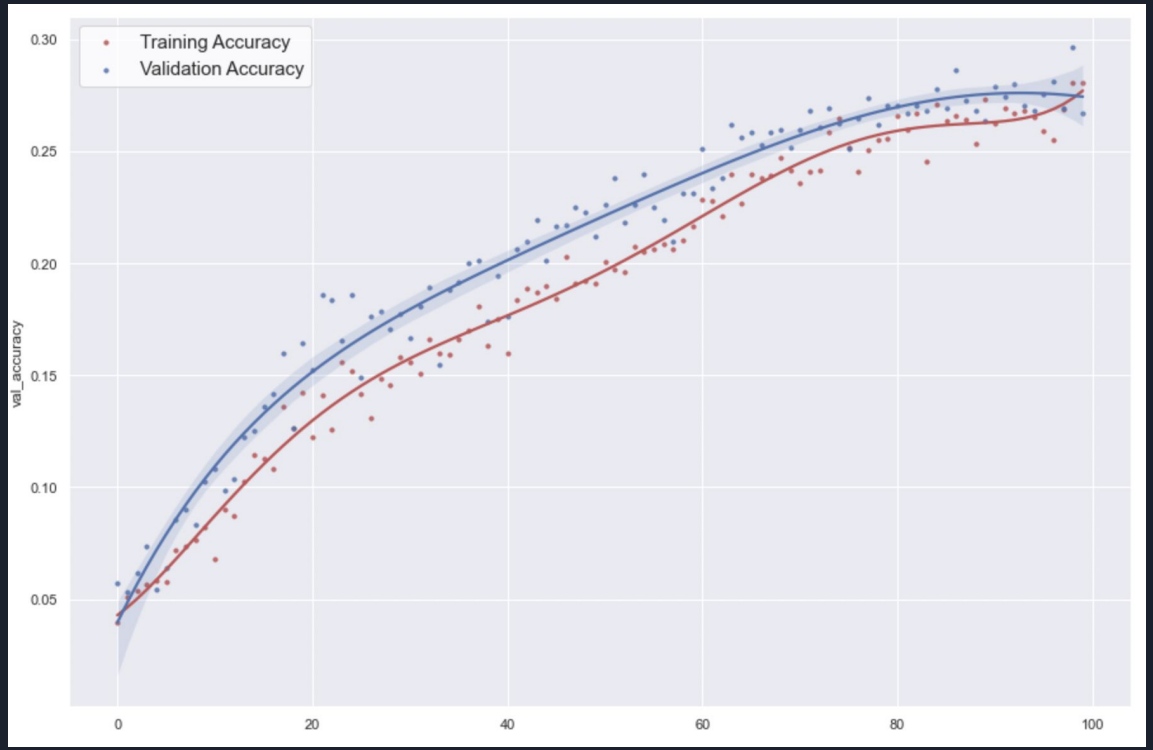
```
learning_rate_reduction = ReduceLROnPlateau(monitor='val_loss', patience=5, verbose=1,
                                            factor=0.5, min_lr=0.00001)

batch_size = 64
epochs = 100
history_bl2 = model_bl2.fit(datagen.flow(X_train,y_train, batch_size=batch_size),
                            epochs = epochs,validation_data=(X_test,y_test),
                            verbose = 1, steps_per_epoch=X_train.shape[0] // batch_size
                           ,callbacks=[learning_rate_reduction])
```

.fit() du modèle sur nos données d'entraînement (à travers le Data Image Generator).

CNN : Convolutional neural Networks

Modèle Baseline :



Hyper-parameter tuning

Hyper-parameter tuning

Un réseau de neurones convolutifs comprends beaucoup de paramètres :

1- **Layers** :
- Nombre de layers
- Types de layers (conv2d, pooling, dense, etc…)
- Paramétrage de ces layers :
  - Conv2D :
    - Nombre de filtres
    - Taille du kernel
    - Fonction d'activation
  - Pooling : taille du kernel

2 - **Compilation :**
- Loss (*categorical_crossentropy*)
- Optimizer (*SGD,RMSprop, etc….)*
  - Learning rate, momentum, etc….

3 - **Exécution du modèle :**
- Batch_size
- Nombre d'époques

Hyper-parameter tuning

Keras Tuner

Module Keras permettant de faciliter l'optimisation d'hyper-paramètres d'un modèle Keras.

Effectue plusieurs fit avec différentes configurations pour identifier les paramètres optimaux.

```
# Tune the number of units in the  Dense layer
# Choose an optimal value between 32-512
hp_units = hp.Int('units', min_value = 32, max_value = 512, step = 32,)

# Tune the activation function for Dense layer
# Choose an optimal value from relu, tanh, sigmoid
hp_activation_dense = hp.Choice("dense_activation", values=["relu", "tanh", "sigmoid"], default="relu")
model.add(keras.layers.Dense(units = hp_units, activation = hp_activation_dense))
model.add(keras.layers.Dense(n_classes, activation='softmax'))
```

Transfer Learning

Transfer Learning

Entraîner un réseau convolutionnel est extrêmement gourmand :

- En ressources de calcul
- En temps d'exécution.

Certaines sociétés passent des mois à entraîner des modèles sur d'énormes datasets d'images. (VGG16, resnet50, Inception RESNET 50 , etc…)

Il serait intéressant de pouvoir charger ces modèles, tout en les adaptant à notre problème de classification.

⇒ Transfer Learning

Transfer Learning

1 - Création du modèle et de ses layers

```
from keras.applications.vgg16 import VGG16
from keras.applications.resnet_v2 import ResNet50V2,ResNet101V2
from keras.applications.inception_resnet_v2 import InceptionResNetV2
from keras.applications.inception_v3 import InceptionV3

# Charger InceptionResnet50 pré-entraîné sur ImageNet et sans les couches fully-connected
model = InceptionResNetV2(
    include_top=False,
    weights="imagenet",
    input_shape=(128,128, 3)
)

#weights="resnet50"
#weights="resnetv2" (v4)

# Récupérer la sortie de ce réseau
x = model.output

# Ajouter la nouvelle couche fully-connected pour la classification à n classes
flat1 = Flatten()(model.layers[-1].output)
class1 = Dense(512, activation='relu')(flat1)
output = Dense(n_classes, activation='softmax')(class1)

# Définir le nouveau modèle
model_tl = Model(inputs=model.input, outputs=output)
```

Import des librairies

Récupération du modèle pré-entraîné.

Définition des couches fully-connected : Classification

Transfer Learning

2 - Paramétrage et compilation

```
# On n'entraine que les layers du dessus déstinés à la classification de nos images.
for layer in model_tl.layers[:(len(model_tl.layers)-3)]:
    layer.trainable = False

# Compiler le modèle
#optimizer = SGD(lr=0.001, momentum=0.9)
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)

model_tl.compile(loss="categorical_crossentropy",
                 optimizer=optimizer, metrics=["accuracy"])
```

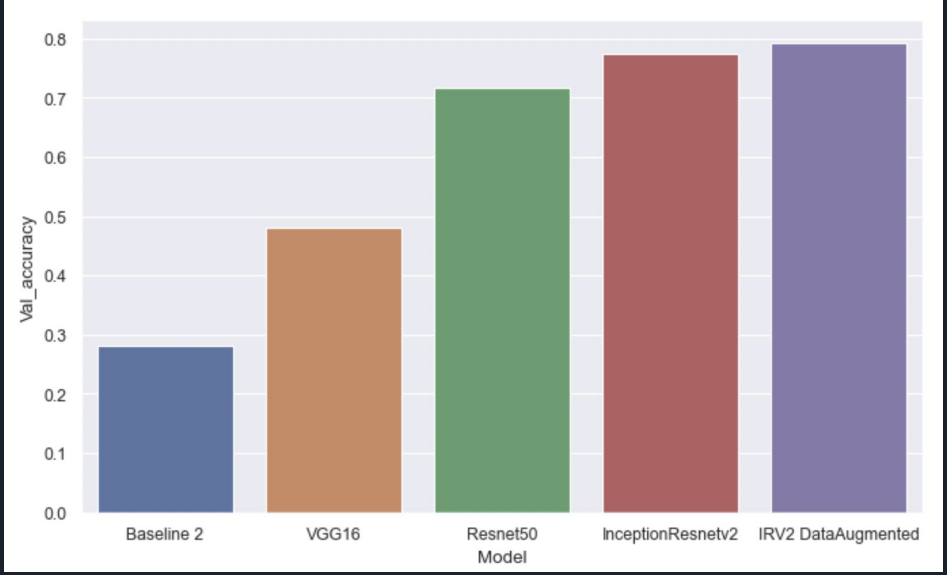
2 - Entraînement du modèle (uniquement les dernières couches) sur nos données d'entraînement.

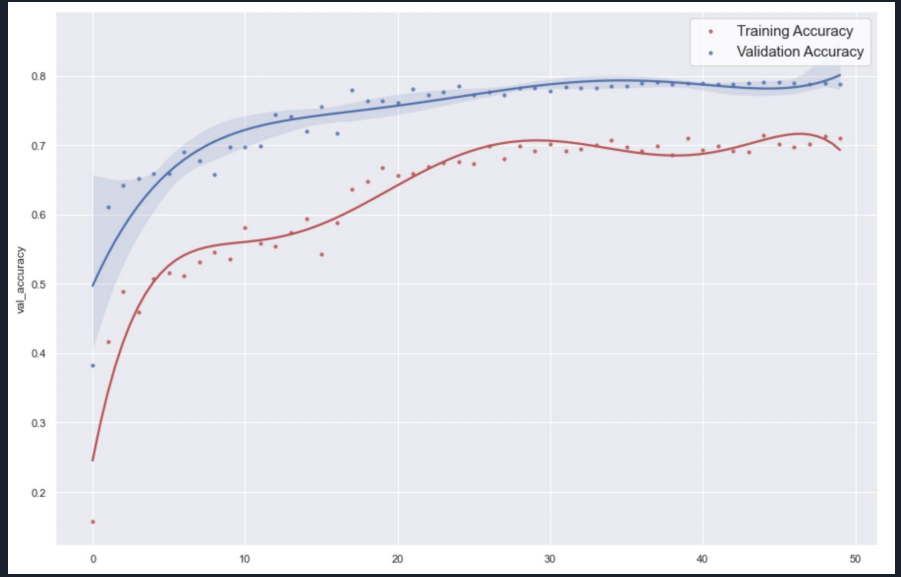
```
learning_rate_reduction = ReduceLROnPlateau(monitor='val_loss', patience=3, verbose=1,
                                            factor=0.5, min_lr=0.00001)

batch_size = 128
epochs = 50
history_tl_InceptionResNetV2 = model_tl.fit_generator(datagen.flow(X_train,y_train, batch_size=batch_size),
                              epochs = epochs,validation_data=(X_test,y_test),
                              verbose = 2, steps_per_epoch=X_train.shape[0] // batch_size
                              , callbacks=[learning_rate_reduction])
```

Résultats

Résultats


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Val_accuracy
Baseline 2
VGG16
Resnet50
InceptionResnetv2
IRV2 DataAugmented
Model


Training Accuracy
Validation Accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
val_accuracy
0
10
20
30
40
50

Conclusion

Conclusion

- Modèle entier (baseline) créé et entraîné sur nos données :
  - Résultats peu satisfaisants :
    - Pas assez de données
    - Temps de calculs très longs
- Modèle de transfert learnings :
  - Résultats satisfaisants :
    - Il peut être intéressant d'observer le comportement de ces modèles avec plus de classes.

- Importance et impact de la data augmentation démontré par l'amélioration des résultats pour un même modèle.

⇒ Démo application.