

# Livrable 2 : Traitement d'images

Erwan  
Martin

Thibaut Liger-  
Hellard

Arnaud  
Maturel

Guillaume Le  
Cocguen

Victorien  
Goudeau

---

## Contexte:

L'entreprise TouNum, spécialisée dans la numérisation de documents, collabore avec des spécialistes en Data Science de CESI pour développer une solution de Machine Learning capable de générer automatiquement des légendes pour les images numérisées. Ce projet vise à enrichir leur offre de services en répondant aux besoins de clients ayant d'importantes quantités de données à classer. Le défi inclut le nettoyage des images de qualité variable et la distinction entre photos et autres types d'images avant l'analyse. L'approche utilisera des technologies avancées telles que les réseaux de neurones convolutifs (CNN) et récurrents (RNN), en s'appuyant sur Python et des bibliothèques spécialisées. Un prototype est attendu dans cinq semaines, suivi d'une présentation détaillée et d'une discussion sur l'intégration et la maintenance de la solution.

## Objectif:

L'objectif de ce projet est de développer un autoencodeur qui servira à améliorer la qualité des images qui entreront dans notre algorithme si jamais elles sont bruitées. Pour l'entraîner, un dataset de photos nous est fourni. Nous allons brouter chacune d'elles et entraîner un réseau de neurones convolutionnels pour extraire les différentes features, et utiliser leurs transposées pour reconstruire l'image au mieux, l'objectif étant de comparer le résultat à l'image de base (non bruitée) afin de voir à quel point notre débruitage est fonctionnel.

## Données:

Notre jeu de données contient plusieurs milliers d'images de différentes tailles et qualités. Elles seront toutes aléatoirement plus ou moins bruitées avant d'entraîner notre modèle. Pas besoin d'étiquetage dans notre cas, on regardera les différences entre l'image non bruitée et l'image de sortie débruitée.

## Défis techniques:

- Qualité des images: Les images sont de qualité variable, ce qui peut affecter les performances du modèle.

- Autoencodeur : Le modèle doit être capable d'améliorer la qualité de différents types de photographies tout en gardant les couleurs d'origine et les éléments importants distinguables.
- Évaluation du modèle: Le modèle sera évalué sur un ensemble de données de test distinct pour mesurer sa perte qui sera ici notre indicateur de performance. On sélectionnera le modèle avec la perte la plus basse sur notre jeu de validation.

```
In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import PIL
import os
import sklearn
import sklearn.model_selection
import tensorflow as tf
import tensorboard
import cv2
from datetime import datetime
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
```

```
In [ ]: gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        tf.config.experimental.set_virtual_device_configuration(gpus[0],
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=9144)
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        print(e)
```

1 Physical GPUs, 1 Logical GPUs

## Récupération du dataset, séparation en jeux d'entraînement, de validation et de test

Toutes les photographies du dataset sont dans un même dossier. Nous les récupérons et les séparons en :

- un jeu d'entraînement (70% du dataset) sur lequel le modèle apprend.
- un jeu de validation (10% du dataset) sur lequel l'apprentissage est testé à chaque itération.
- un jeu de test (20% du dataset) sur lequel on conduit nos essais, une fois le modèle terminé.

```
In [ ]: import os
from sklearn.model_selection import train_test_split

def split_data(image_folder, train_size=0.7, test_size=0.2):

    image_files = [os.path.join(image_folder, f) for f in os.listdir(image_
    train_files, val_test_files = train_test_split(image_files, train_size=
```

```

val_size = test_size / (1 - train_size) # Calcul pour garder la proportion
val_files, test_files = train_test_split(val_test_files, test_size=val_size)

return train_files, val_files, test_files

filePath = open('../data/path.txt', "r")
datapath = filePath.read()
print(datapath)

image_folder = datapath+'/Photo'
train_files, val_files, test_files = split_data(image_folder)

```

D:\CESI\A5\datascience\Projet\data

## Bruitage des images

La fonction `image_noise_generator` va recevoir les photographies en entrée, et leur appliquer un bruitage aléatoire puis les renvoyer en sortie.

Ici l'objectif est de produire des données d'entrées dégradées à notre algorithme afin qu'il puisse au mieux en améliorer la qualité. Dans un contexte plus global cela permet de limiter l'impact des images de mauvaise qualité sur des opérations telles que de la classification.

```
In [ ]: def image_noise_generator(image_files, batch_size, noise_factor=0.5, img_size=(224, 224)):
    while True:
        image_files = shuffle(image_files)
        for i in range(0, len(image_files), batch_size):
            batch_files = image_files[i:i + batch_size]
            batch_images = []
            batch_noisy_images = []
            for file in batch_files:
                img = cv2.imread(file)
                if img is not None:
                    img = cv2.resize(img, img_size)
                    img = img.astype('float32') / 255.0
                    noisy_img = img + np.random.random()*noise_factor * np.random.uniform(-1, 1, img.shape)
                    noisy_img = np.clip(noisy_img, 0, 1)
                    batch_images.append(img)
                    batch_noisy_images.append(noisy_img)
            yield np.array(batch_noisy_images), np.array(batch_images)
```

## Débruitage des images

Après avoir bruité les images à des fins de test sur des données d'entrées dégradées, nous voulons apporter une solution à ce problème potentiel. Pour ce faire nous nous mettons en place un fonction qui débruittera les images afin de les faire se rapprocher au maximum de ce qu'elles représentaient à l'origine.

L'utilisation de convolutions et de max poolings permettra de faire ressortir les caractéristiques principales de l'image et les couches upsampling ainsi que les transposées de nos convolutions permettront de reconstruire une image aussi proche que possible de l'image d'entrée sans bruit.

```
In [ ]: def denoises_model():
    inputs = tf.keras.Input(shape=(400, 400, 3))
    x = tf.keras.layers.Conv2D(32, 3, activation="relu", strides=1, padding="same")(inputs)
    xp1 = tf.keras.layers.MaxPool2D(2,2)(x)
    x1 = tf.keras.layers.Conv2D(64, 3, activation="relu", strides=1, padding="same")(xp1)
    xp2 = tf.keras.layers.MaxPool2D(2,2)(x1)
    x2 = tf.keras.layers.Conv2D(128, 3, activation="relu", strides=1, padding="same")(xp2)
    xp3 = tf.keras.layers.MaxPool2D(2,2)(x2)
    encodeur = tf.keras.Model(inputs, xp3, name="encoded")

    decoder_input = tf.keras.Input(shape=(50, 50, 128))
    z1 = tf.keras.layers.Conv2DTranspose(64, 5, activation="relu", strides=2, padding="same")(decoder_input)
    zp2 = tf.keras.layers.UpSampling2D((2,2))(z1)
    z2 = tf.keras.layers.Conv2DTranspose(32, 5, activation="relu", strides=2, padding="same")(zp2)
    zp3 = tf.keras.layers.UpSampling2D((2,2))(z2)
    z3 = tf.keras.layers.Conv2DTranspose(3, 3, activation="sigmoid", padding="same")(zp3)
    outputs = tf.keras.layers.UpSampling2D((2,2))(z3)
    decodeur = tf.keras.Model(decoder_input, outputs, name="decoded")

    auto_input = tf.keras.Input(shape=(400, 400, 3))
    encoded = encodeur(auto_input)
    decoded = decodeur(encoded)
    auto_encodeur = tf.keras.Model(auto_input, decoded, name="auto_encodeur")
    auto_encodeur.compile(optimizer='adam', loss=['mean_squared_error'])

    return auto_encodeur

auto_encodeur_model = denoises_model()
auto_encodeur_model.summary()
```

Model: "auto\_encodeur"

Layer (type)	Output Shape	Param #
<hr/>		
input_6 (InputLayer)	[(None, 400, 400, 3)]	0
encoded (Functional)	(None, 50, 50, 128)	93248
decoded (Functional)	(None, 400, 400, 3)	256963
<hr/>		
Total params: 350,211		
Trainable params: 350,211		
Non-trainable params: 0		

---

```
In [ ]: batch_size = 32

train_generator = image_noise_generator(train_files, batch_size)
val_generator = image_noise_generator(val_files, batch_size)
test_generator = image_noise_generator(test_files, batch_size)

steps_per_epoch = len(train_files) // batch_size
validation_steps = len(val_files) // batch_size
test_steps = len(test_files) // batch_size
```

## Optimisation des performances

Pour éviter les écueils lors de l'entraînement de notre algorithme tels que l'overfitting, nous utilisons une fonction qui surveillera notre valeur de perte (loss) ainsi à chaque epoch nous

mettrons à jour la valeur minimale de la loss (par rapport au jeu de validation). Si cette dernière cesse de diminuer après plusieurs epochs nous considérons que les poids utilisés étaient optimaux et donc que nous pouvons arrêter notre entraînement en conservant les poids de la meilleure epoch.

```
In [ ]: class EarlyStoppingAtMinLossAndSave(tf.keras.callbacks.Callback):
    """Stop training when the loss is at its min, i.e. the loss stops decreasing.

    Arguments:
        patience: Number of epochs to wait after min has been hit. After this
                  number of no improvement, training stops.
    """

    def __init__(self, patience=0):
        super().__init__()
        self.patience = patience
        # best_weights to store the weights at which the minimum loss occurs
        self.best_weights = None

    def on_train_begin(self, logs=None):
        # The number of epoch it has waited when loss is no longer minimum.
        self.wait = 0
        # The epoch the training stops at.
        self.stopped_epoch = 0
        # Initialize the best as infinity.
        self.best = np.Inf

    def on_epoch_end(self, epoch, logs=None):
        current = logs.get("val_loss")
        print("The average loss for epoch {} is {:.2f} ".format(epoch, logs))

        if np.less(current, self.best):
            self.best = current
            self.wait = 0
            # Record the best weights if current results is better (less).
            self.best_weights = self.model.get_weights()
            self.model.save_weights('./models/last_training_best_weights')
        else:
            self.wait += 1
            if self.wait >= self.patience:
                self.stopped_epoch = epoch
                self.model.stop_training = True
                print("Restoring model weights from the end of the best epoch")
                self.model.set_weights(self.best_weights)

    def on_train_end(self, logs=None):
        if self.stopped_epoch > 0:
            print("Epoch %5d: early stopping" % (self.stopped_epoch + 1))

#TENSORBOARD
time = datetime.now()
foldername = f"./tensorboard/{time.day}_{time.month}_{time.year}_{time.hour}"

tensorflowCallback = tf.keras.callbacks.TensorBoard(
    log_dir=foldername,
    histogram_freq=0,
    write_graph=True,
    write_images=False,
    write_steps_per_second=False,
    update_freq='epoch',
    profile_batch=0,
```

```
embeddings_freq=0,  
embeddings_metadata=None  
)
```

```
In [ ]: history = auto_encodeur_model.fit(  
        train_generator,  
        steps_per_epoch=steps_per_epoch,  
        epochs=10,  
        validation_data=val_generator,  
        validation_steps=validation_steps,  
        callbacks=[EarlyStoppingAtMinLossAndSave(patience=2), tensorflowCallback])
```

```
Epoch 1/10  
218/218 [=====] - ETA: 0s - loss: 0.0191The average loss for epoch 0 is 0.02  
218/218 [=====] - 174s 764ms/step - loss: 0.0191 - val_loss: 0.0089  
Epoch 2/10  
218/218 [=====] - ETA: 0s - loss: 0.0082The average loss for epoch 1 is 0.01  
218/218 [=====] - 155s 705ms/step - loss: 0.0082 - val_loss: 0.0077  
Epoch 3/10  
218/218 [=====] - ETA: 0s - loss: 0.0073The average loss for epoch 2 is 0.01  
218/218 [=====] - 154s 708ms/step - loss: 0.0073 - val_loss: 0.0068  
Epoch 4/10  
218/218 [=====] - ETA: 0s - loss: 0.0069The average loss for epoch 3 is 0.01  
218/218 [=====] - 150s 691ms/step - loss: 0.0069 - val_loss: 0.0065  
Epoch 5/10  
218/218 [=====] - ETA: 0s - loss: 0.0066The average loss for epoch 4 is 0.01  
218/218 [=====] - 150s 688ms/step - loss: 0.0066 - val_loss: 0.0064  
Epoch 6/10  
218/218 [=====] - ETA: 0s - loss: 0.0064The average loss for epoch 5 is 0.01  
218/218 [=====] - 150s 691ms/step - loss: 0.0064 - val_loss: 0.0062  
Epoch 7/10  
218/218 [=====] - ETA: 0s - loss: 0.0062The average loss for epoch 6 is 0.01  
218/218 [=====] - 149s 688ms/step - loss: 0.0062 - val_loss: 0.0060  
Epoch 8/10  
218/218 [=====] - ETA: 0s - loss: 0.0061The average loss for epoch 7 is 0.01  
218/218 [=====] - 149s 686ms/step - loss: 0.0061 - val_loss: 0.0059  
Epoch 9/10  
218/218 [=====] - ETA: 0s - loss: 0.0059The average loss for epoch 8 is 0.01  
218/218 [=====] - 150s 691ms/step - loss: 0.0059 - val_loss: 0.0060  
Epoch 10/10  
218/218 [=====] - ETA: 0s - loss: 0.0058The average loss for epoch 9 is 0.01  
218/218 [=====] - 149s 687ms/step - loss: 0.0058 - val_loss: 0.0055
```

mesure sur notre jeu de test :

```
In [ ]: test_loss = auto_encodeur_model.evaluate(test_generator, steps=test_steps)
print('Test Loss:', test_loss)

62/62 [=====] - 41s 662ms/step - loss: 0.0055
Test Loss: 0.005544453393667936
```

## Visualisation après entraînement

Ici, nous allons visualiser des images avant, pendant et après débruitage afin de voir à l'oeil comment notre modèle performe.

```
In [ ]: import random

def prepare_example(number, folder, model, noise_factor=0.5, img_size = (400, 400)):
    image_files = [os.path.join(folder, f) for f in os.listdir(folder) if os.path.isfile(os.path.join(folder, f))]
    random.shuffle(image_files)
    image_files = image_files[:number]

    original_images = []
    noised_images = []
    denoised_images = []

    for file in image_files:
        img = cv2.imread(file)
        if img is not None:
            img_correct_color = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img_resized = cv2.resize(img_correct_color, img_size)
            img_normalized = img_resized.astype('float32') / 255.0
            img_batch = np.expand_dims(img_normalized, axis=0)
            noisy_img = img_batch + np.random.random()*noise_factor * np.random.normal(0, 1)
            noisy_img = np.clip(noisy_img, 0, 1)

            predicted = model.predict(noisy_img)
            predicted_image = predicted.squeeze()

            original_images.append(img_resized)
            noised_images.append(noisy_img[0, :, :, :])
            denoised_images.append(predicted_image)

    return original_images, noised_images, denoised_images

def display_images(original_images, noised_images, denoised_images):
    plt.figure(figsize=(30, 10))

    num_images = len(original_images)
    for i in range(num_images):
        ax = plt.subplot(3, num_images, i + 1)
        plt.imshow(original_images[i])
        plt.title("Original")
        plt.axis("off")

        ax = plt.subplot(3, num_images, i + 1 + num_images)
        plt.imshow(noised_images[i])
        plt.title("noised")
        plt.axis("off")

        ax = plt.subplot(3, num_images, i + 1 + num_images * 2)
        plt.imshow(denoised_images[i])
        plt.title("Denoised")
        plt.axis("off")
```

```

        ax = plt.subplot(3, num_images, i + 1 + 2*num_images)
        plt.imshow(denoised_images[i])
        plt.title("Denoised")
        plt.axis("off")

plt.show()

```

In [ ]:

```

test_image_folder = datapath+'\Dataset_L2'

original_imgs, noised_images, denoised_imgs = prepare_example(10, test_image)

display_images(original_imgs, noised_images, denoised_imgs)

```

```

1/1 [=====] - 0s 16ms/step
1/1 [=====] - 0s 18ms/step
1/1 [=====] - 0s 16ms/step
1/1 [=====] - 0s 17ms/step
1/1 [=====] - 0s 16ms/step
1/1 [=====] - 0s 17ms/step
1/1 [=====] - 0s 15ms/step
1/1 [=====] - 0s 15ms/step
1/1 [=====] - 0s 18ms/step
1/1 [=====] - 0s 16ms/step

```



## Sauvegarde du modèle

On sauvegarde le modèle à chaque fois, et on choisira lequel est le meilleur par la suite après plusieurs entraînements.

In [ ]:

```
auto_encodeur_model.save(f'./models/{time.day}_{time.month}_{time.year}_{time.hour}_{time.minute}_{time.second}')
```

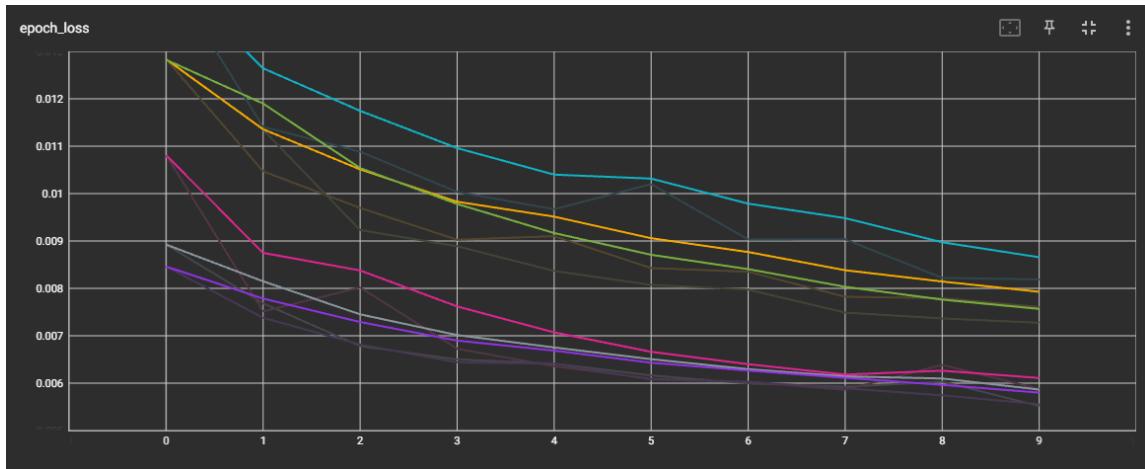
## Choix du meilleur modèle

Nous avons testé différents entraînements avec différents paramètres. Nous avons :

- Fait varier le nombre de filtres sur nos convolutions pour améliorer la précision de la reconstruction de l'image.
- Bruit et fait varier le bruit de manière aléatoire (le bruit lui-même mais aussi son intensité) pour ne pas avoir un modèle entraîné que sur des images bruitées.
- Fait varier le nombre de couches de convolution

Tous ces paramètres nous ont permis de tester la différence d'efficacité d'entraînement. Nous avons notre callback qui nous enregistre la loss sur tensorboard. Le graphique ci-

dessous montre l'évolution de la loss par epoch sur les différents modèles entraînés.



Voici les valeurs associées à la courbe:

Run	Smoothed Value	Step Time	Relative
17_4_2024_10h39\validation	0.007928	0.007606 9	4/17/24, 11:04 AM 22.23 min
17_4_2024_11h34\validation	0.007567	0.007275 9	4/17/24, 12:00 PM 22.83 min
17_4_2024_15h56\validation	0.005865	0.005518 9	4/17/24, 4:22 PM 22.62 min
17_4_2024_16h43\validation	0.006109	0.005879 9	4/17/24, 5:08 PM 22.02 min
17_4_2024_17h12\validation	0.0058	0.005556 9	4/17/24, 5:36 PM 21.87 min
17_4_2024_9h27\validation	0.008657	0.008188 9	4/17/24, 9:55 AM 25.07 min

Ayant la loss la plus basse sur le modèle entraîné le 17/4/2024 à 17h12, c'est celui-ci qu'on choisira puisqu'il généralise le mieux et qu'on a a vue d'oeil le meilleur rendu d'images avec celui-ci.

Le modèle est enregistré ici : ./best\_model/17\_4\_2024\_17h12.keras

## Chargement du modèle pour une visualisation de sa performance

```
In [ ]: #test_image_folder = "C:/Users/erwan/Desktop/test_data" #vous pouvez tester
          test_image_folder = image_folder

          #Charger le modèle évitera d'avoir à en réentraîner un.
          model_path = "./best_model/17_4_2024_17h12.keras"
          model = tf.keras.models.load_model(model_path)

          # (nb_images, pa
          original_imgs, noised_images, denoised_imgs = prepare_example(10, test_im
          display_images(original_imgs, noised_images, denoised_imgs)
```

```
1/1 [=====] - 0s 84ms/step
1/1 [=====] - 0s 18ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 15ms/step
1/1 [=====] - 0s 14ms/step
1/1 [=====] - 0s 16ms/step
1/1 [=====] - 0s 17ms/step
1/1 [=====] - 0s 14ms/step
1/1 [=====] - 0s 16ms/step
1/1 [=====] - 0s 15ms/step
```



## Conclusion

Nous avons donc notre modèle capable de débruiter des images en conservant leurs caractéristiques principales.

Débruiter les images en extrayant leurs features nous permettra par la suite d'alimenter avec des images de qualité un Recurrent Neural Network (RNN) pour le captioning.