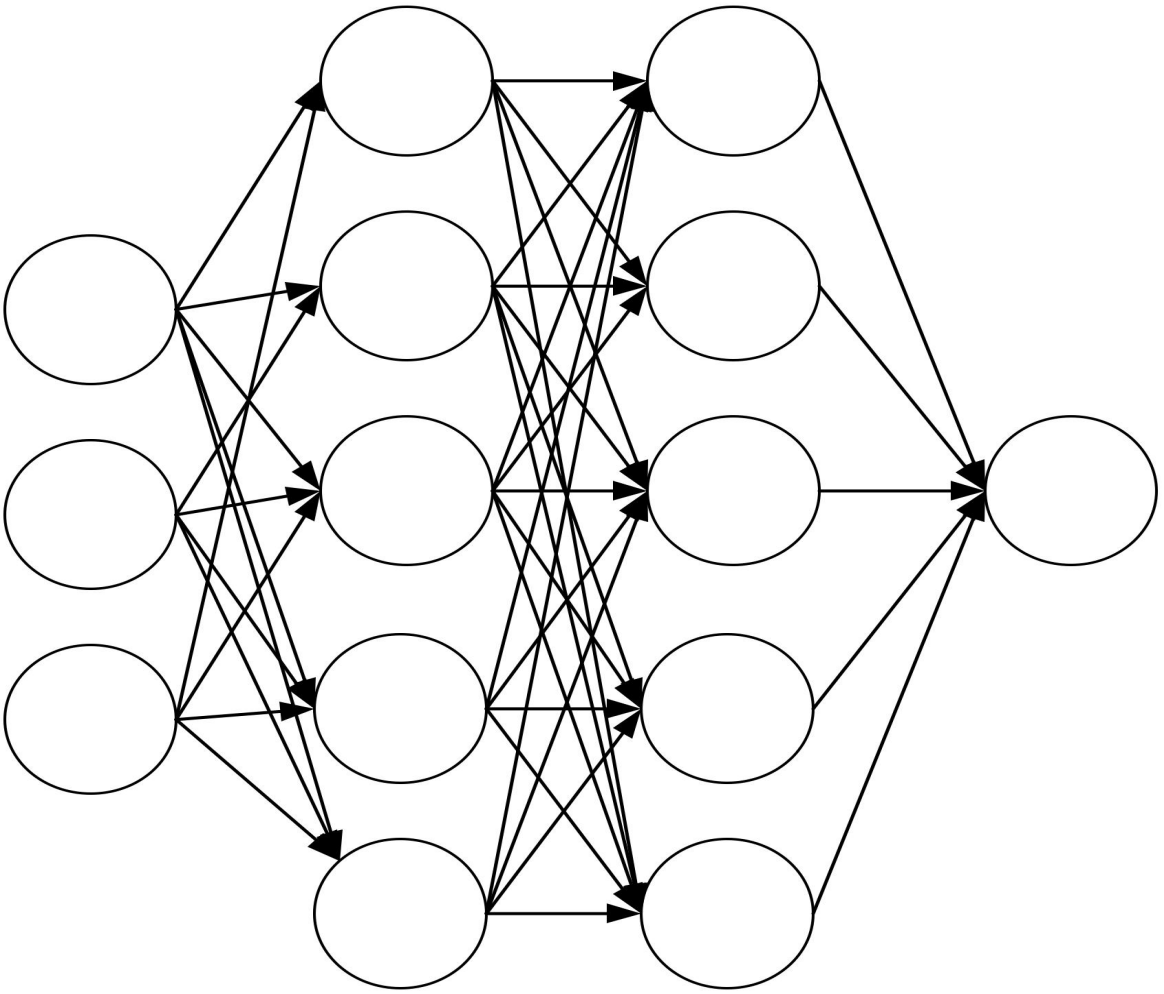
Zoidberg

Compte rendu



Loann MR
Tarek D
Jules B
Melvin C
Xiangwei J

{EPITECH.}

Introduction

We have access to three datasets comprising X-ray images, which will be used for model training, validation, and testing. We will follow a standardized train-validation-test procedure to ensure reliable and generalized models.

Initially, we will preprocess the datasets, including tasks like data cleaning, normalization, and augmentation. This step ensures the data is in a suitable format and exhibits a fair distribution across the predicted classes: no pneumonia, viral pneumonia, and bacterial pneumonia.

Next, we will select suitable machine learning algorithms for the classification task. We will explore traditional classifiers and advanced techniques like neural networks, comparing their performance to determine the most effective approach.

To evaluate our models, we will employ both train-test split and cross-validation procedures. Train-test split estimates model performance on unseen data, while cross-validation provides more robust estimates by averaging results over multiple splits. We will utilize evaluation metrics such as accuracy, precision, recall, F1 score, and ROC-AUC score to quantify and compare model performance.

Moreover, we will use one dataset for algorithm tuning. Tuning involves finding optimal hyperparameters that significantly impact model performance. Through systematic search or optimization processes, we will fine-tune parameters to achieve the best results.

Finally, we will present our findings and results. Our technical document will include code, text, and graphics.

1. Scikit-Learn, the library of algorithms

Scikit-learn is a popular open-source machine learning library in Python. It provides a wide range of efficient tools for various tasks in machine learning, including classification, regression, clustering, dimensionality reduction, and model selection. Scikit-learn offers a user-friendly interface and supports a rich set of algorithms and evaluation metrics, making it suitable for both beginners and experienced practitioners. It also integrates well with other Python libraries, such as NumPy, Pandas, and Matplotlib, for data manipulation, analysis, and visualization.

1.1 Setup and Load Data

Using Miniconda we installed Jupyter Notebook

```
conda install -c anaconda jupyter
conda install -c conda-forge opencv
conda install -c anaconda numpy
conda install -c conda-forge matplotlib
```

Then as setup :

```
import os
import cv2 #open cv
import numpy as np
import matplotlib.pyplot as plt
```

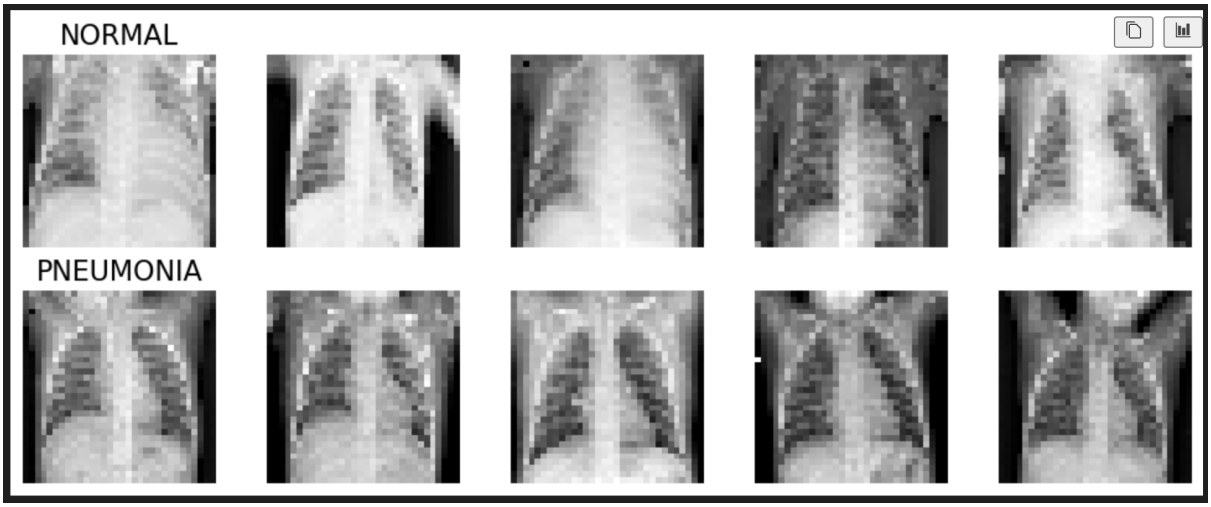
Load:
To load data we defined a data loading function named load_dataset which takes as a parameter the path of the data. This function also performs data standardization and filtering. We standardized the images to the same size 32x32, for easier comparison and by lowering the resolution faster training.
We skipped incorrect file such as files with the extension .DS_store

```
def load_dataset(dataset_path):
    data = []
    labels = []
    label_id = 0
    for folder in os.listdir(dataset_path):
        folder_path = os.path.join(dataset_path, folder)

        if os.path.isdir(folder_path):
            for img_path in os.listdir(folder_path):
                if not (img_path.endswith(".jpeg") or img_path.endswith(".jpg")) :
                    print("Skipping file: ", img_path)
                    continue
                img = cv2.imread(os.path.join(folder_path, img_path), cv2.IMREAD_GRAYSCALE)
                #print(img, os.path.join(folder_path, img_path))
                if img is None:
                    print("Error: Could not read the image")
                else:
                    img_resized = cv2.resize(img, (32, 32))
                    img_flattened = img_resized.flatten()

                    data.append(img_flattened)
                    labels.append(label_id)
            label_id =1

    return np.array(data), np.array(labels)
```

Visualize:
For data visualization, we have defined a function visualize_data which takes as a parameters X_data, y_data, label_names (Normal/Pneumonia), num_images_per_class

```
def visualize_data(data, labels, label_names, num_images_per_class=10):
    num_classes = len(set(labels))
    fig, axes = plt.subplots(num_classes, num_images_per_class, figsize=(num_images_per_class * 2, num_classes * 2))

    for label_id in range(num_classes):
        label_indices = np.where(labels == label_id)[0]
        sample_indices = np.random.choice(label_indices, size=num_images_per_class, replace=False)

        for i, img_index in enumerate(sample_indices):
            img = data[img_index].reshape(32, 32)
            axes[label_id, i].imshow(img, cmap='gray')
            axes[label_id, i].axis('off')

            if i == 0:
                axes[label_id, i].set_title(label_names[label_id], fontsize=16)

    plt.tight_layout()
    plt.show()
```

```
label_names = ["NORMAL", "PNEUMONIA"]
visualize_data(X_train, y_train, label_names, num_images_per_class=5)
```



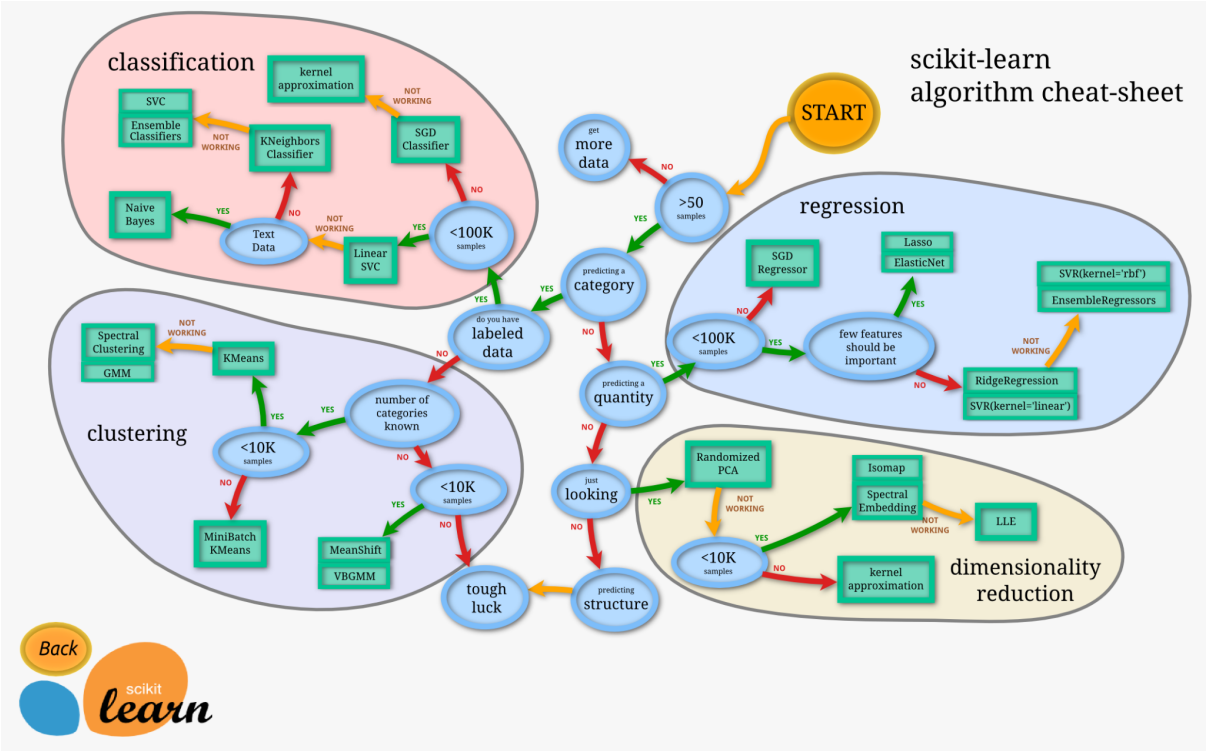
Standardization:
We standardize our data with StandardScaler which transforms the data in such a way that each feature has a mean of zero and a standard deviation of one. The StandardScaler is a preprocessing technique that standardizes the features of a dataset by scaling them to have zero mean and unit variance. It is beneficial for feature scaling, handling outliers, achieving a Gaussian distribution, and improving the interpretability of machine learning models.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Our data is ready to use to train our model.

1.2 Which algorithm should I use ?

Scikit-learn contains an enormous amount of algorithms and as beginners, we don't know which one to use. Thanks to this cheat-sheet.



We have:

- ~6000 samples
- Predicting à category
- We have labeled data (NORMAL or PNEUMONIA)

If we follow the route from the start, we can start with Linear SVC.

1.3 Linear SVC

Linear SVC (Support Vector Classifier) is a classification algorithm provided by the scikit-learn library. It is a variant of the Support Vector Machine (SVM) algorithm that is specifically designed for linearly separable datasets. Linear SVC works by finding an optimal hyperplane that best separates the data into different classes. It aims to maximize the margin between the support vectors (data points closest to the decision boundary) and the classes, while minimizing misclassifications. Linear SVC is efficient for large-scale datasets and works well in scenarios where the classes are linearly separable.

1.3.1 Building and training the model

You can import the model from sklearn.svm

```
from sklearn.svm import LinearSVC
# build the model
svc = LinearSVC(max_iter=1000, random_state=0)
```

In the `LinearSVC` model in scikit-learn, the `max_iter` parameter determines the maximum number of iterations allowed for the solver to converge.

The solver in `LinearSVC` is responsible for finding the optimal solution for the linear support vector classifier. It uses an iterative optimization algorithm called Coordinate Descent by default. The `max_iter` parameter specifies the maximum number of iterations that the solver will perform before considering the optimization process as converged.

You can train your model With the method fit(),

```
# train the model
svc.fit(X_train,y_train)
```

In scikit-learn, the `fit` method is a fundamental method available in most machine learning estimators. The `fit` method is used to train the model on the provided training data. It takes the input features (X) and the corresponding target values (y) as arguments.

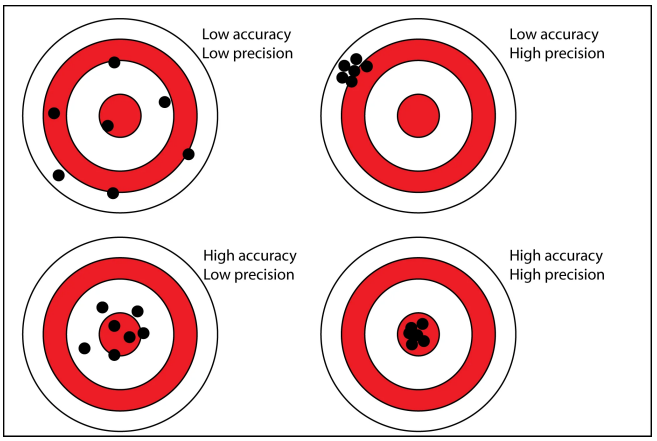
When the `fit` method is called, the estimator analyzes the input data and learns the underlying patterns or relationships between the features and the target. The specific learning process varies depending on the algorithm being used. For example, in linear regression, the `fit` method calculates the coefficients that minimize the sum of squared residuals, while in decision trees, it determines the optimal split points.

1.3.2 Evaluate the model

```
evaluateModel(svc,X_test,y_test)
```

LinearSVC (max_iter = 1000) en 23s

```
**********************
   Evaluation du modèle
**********************
Accuracy on test data: 0.74
Precision on test data: 0.75
Recall on test data: 0.74
F1-score on test data: 0.71
Confusion matrix on test data:
[[ 98 144]
 [ 24 374]]
Classification report on test data:
              precision    recall  f1-score   support

           0       0.80      0.40      0.54       242
           1       0.72      0.94      0.82       398

    accuracy                           0.74       640
   macro avg       0.76      0.67      0.68       640
weighted avg       0.75      0.74      0.71       640
```



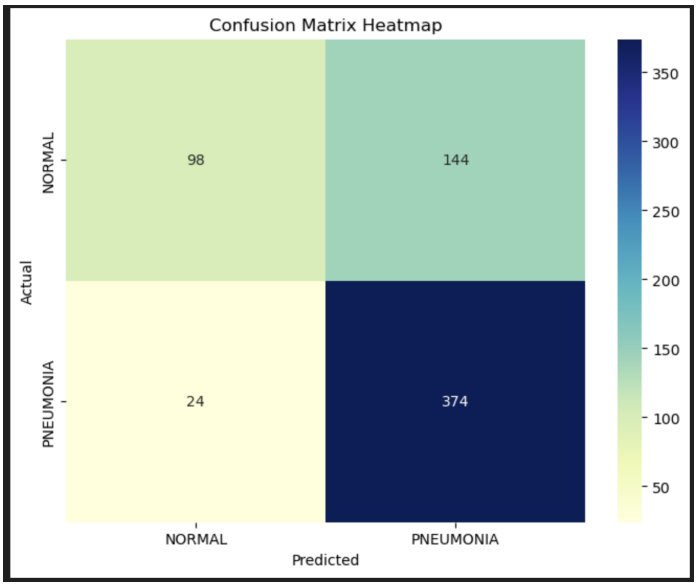
F1 Score is the average of Precision and Recall.

Accuracy is the ratio of correctly predicted observation to the total observations.

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations.

A classification report shows the performance of a model by comparing precision, recall, F1 Score and support.

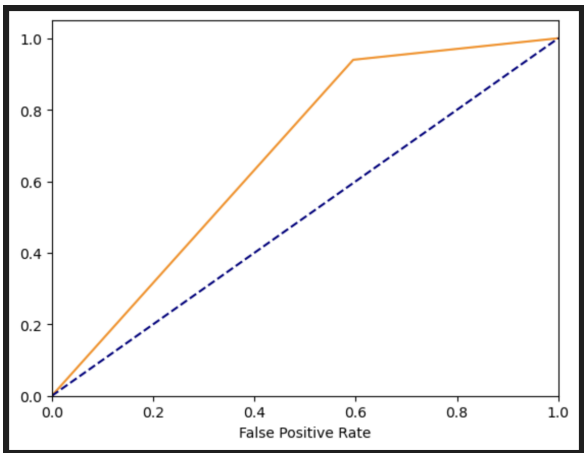
A confusion matrix shows the performance of a model. It compares True Positives and Negatives with False Positives and Negatives.



The ROC (Receiver Operating Characteristic) curve and AUC (Area Under the Curve) are evaluation metrics commonly used in binary classification problems.

The ROC curve is a graphical representation of the performance of a binary classifier as the discrimination threshold is varied. It plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) for different threshold values. The curve illustrates the trade-off between the classifier's ability to correctly identify positive instances (true positives) and its tendency to incorrectly classify negative instances (false positives). A steeper ROC curve, closer to the top-left corner, indicates better classification performance.

The AUC is the area under the ROC curve, which provides a single scalar value to summarize the classifier's overall performance. The AUC ranges from 0 to 1, where an AUC of 0.5 represents a random classifier, and an AUC of 1 represents a perfect classifier. A higher AUC indicates better discrimination ability of the classifier across different threshold values.



1.4 KNeighbors Classifier

The K-Nearest Neighbors (KNN) Classifier is a simple and popular supervised learning algorithm used for classification tasks. It is a non-parametric algorithm that makes predictions based on the similarity of a new data point to its neighboring data points in the feature space.

```
from sklearn.neighbors import KNeighborsClassifier

neigh = KNeighborsClassifier(n_neighbors=4, weights="uniform")
neigh.fit(X_train, y_train)
evaluateModel(neigh,X_test,y_test)
```

Evaluation du modèle

****

Accuracy on test data: 0.77

Precision on test data: 0.82

Recall on test data: 0.77

F1-score on test data: 0.73

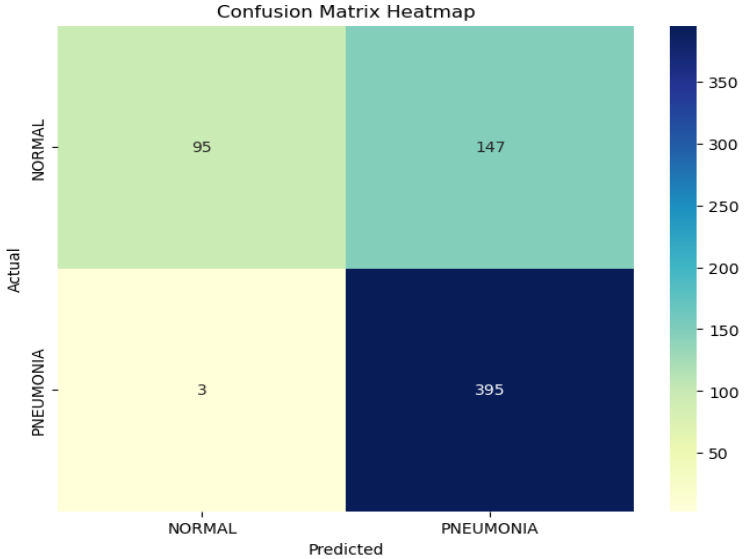
Confusion matrix on test data:

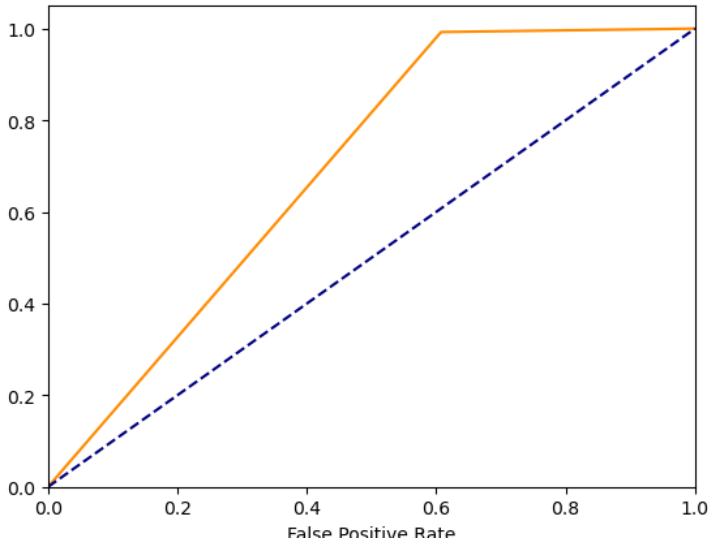
[[95 147]

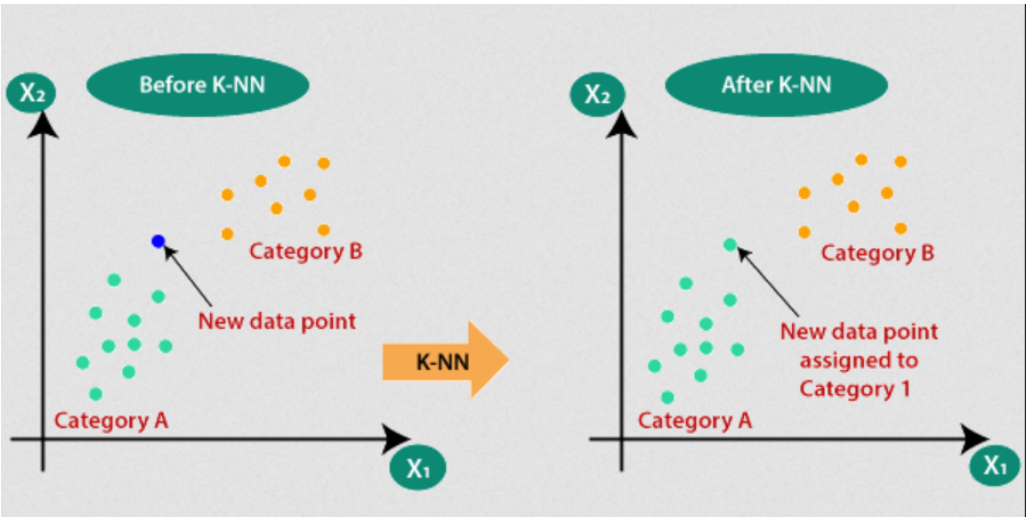
[3 395]]

Classification report on test data:

	precision	recall	f1-score	support
0	0.97	0.39	0.56	242
1	0.73	0.99	0.84	398
accuracy			0.77	640
macro avg	0.85	0.69	0.70	640
weighted avg	0.82	0.77	0.73	640



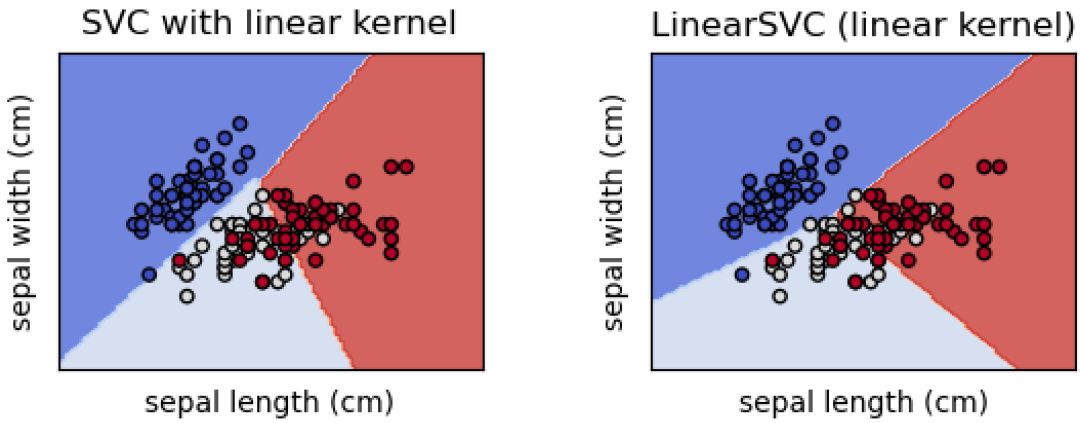




The KNeighbors take as a parameter the number of neighbor n we will work with, then calculate the euclidean distance between those neighbors and the point then select the numbers of neighbors n closest to the point then predict the classification of the point.

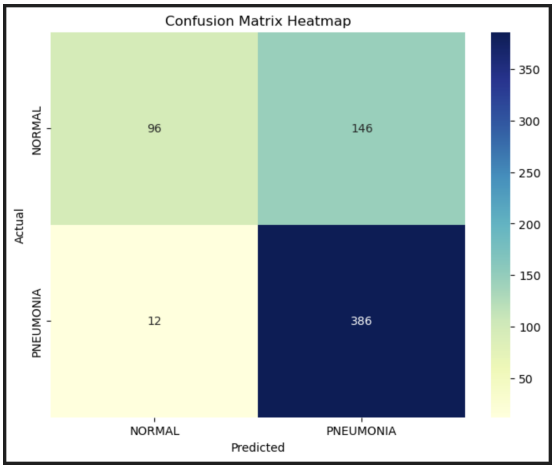
Kneighbors is simple and scalable with few parameters to manipulate but require more power to calculate.

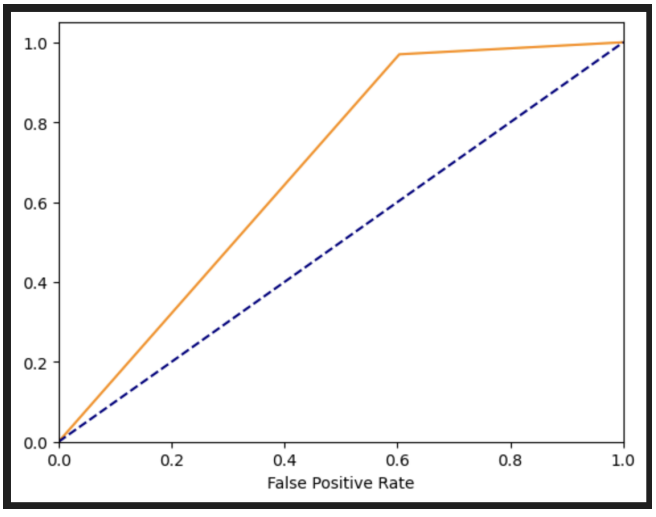
1.5 SVC



SVC kernel=”linear” 4m36s

```
**********************
   Evaluation du modèle
**********************
Accuracy on test data: 0.75
Precision on test data: 0.79
Recall on test data: 0.75
F1-score on test data: 0.72
Confusion matrix on test data:
[[ 96 146]
 [ 12 386]]
Classification report on test data:
              precision    recall  f1-score   support

           0       0.89      0.40      0.55       242
           1       0.73      0.97      0.83       398

    accuracy                           0.75       640
   macro avg       0.81      0.68      0.69       640
weighted avg       0.79      0.75      0.72       640
```





SVC linear is slower than LinearSVC, SVC linear solves a system of equations to find the best parameters for the model. LinearSVC on the other hand uses an optimization method that updates the parameters of a model using individual samples at a time, this allows large datasets to be processed efficiently and results to be obtained quickly.

1.6 Boosting

Boosting is a machine learning technique that combines multiple weak models to create a strong predictive model. The idea behind boosting is to sequentially train models in a way that each subsequent model focuses on correcting the mistakes made by the previous models. One popular algorithm that uses boosting is Gradient Boosting.

Gradient Boosting is a boosting algorithm that builds an ensemble of decision trees. The GradientBoostingClassifier, specifically, is a variant of Gradient Boosting used for classification tasks

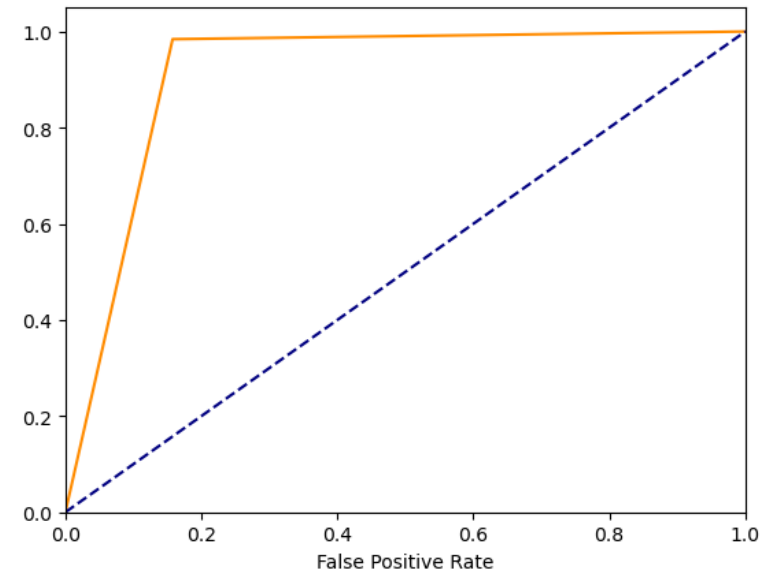
Gradient Boosting, including the GradientBoostingClassifier, has several advantages. It can handle complex datasets and capture non-linear relationships between features and the target variable. It also provides robustness against overfitting and performs well even with high-dimensional data. However, it can be computationally expensive and requires careful tuning of hyperparameters to optimize performance.

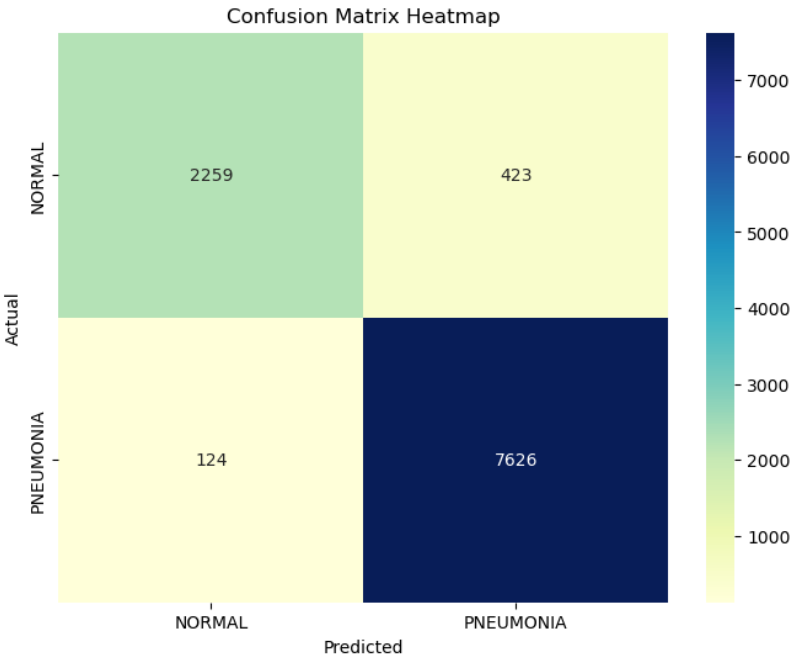
1.6.1 XGboost

XGBoost is a supervised machine learning algorithm based on boosting that combines multiple weak prediction models into a stronger model. It uses a set of features to create a decision tree at each iteration, placing more emphasis on previously mispredicted examples. The trees are sequentially added to improve predictions, using an optimized cost function. XGBoost is known for its ability to handle large datasets, robustness to missing values, and efficiency in terms of execution time.

```python
from xgboost import XGBClassifier

# Définir les hyperparamètres de votre choix
hyperparams = {
    'max_depth': 3,
    'learning_rate': 0.1,
    'n_estimators': 100,
    'subsample': 0.75,
    'colsample_bytree': 0.5
}
```

```
Performance du modèle sur les données de test:
**********
   Evaluation du modèle
**********
Accuracy on test data: 0.77
Precision on test data: 0.82
Recall on test data: 0.77
F1-score on test data: 0.74
Confusion matrix on test data:
[[ 99 143]
 [  5 393]]
Classification report on test data:
              precision    recall  f1-score   support

           0       0.95      0.41      0.57       242
           1       0.73      0.99      0.84       398

    accuracy                           0.77       640
   macro avg       0.84      0.70      0.71       640
weighted avg       0.82      0.77      0.74       640
```



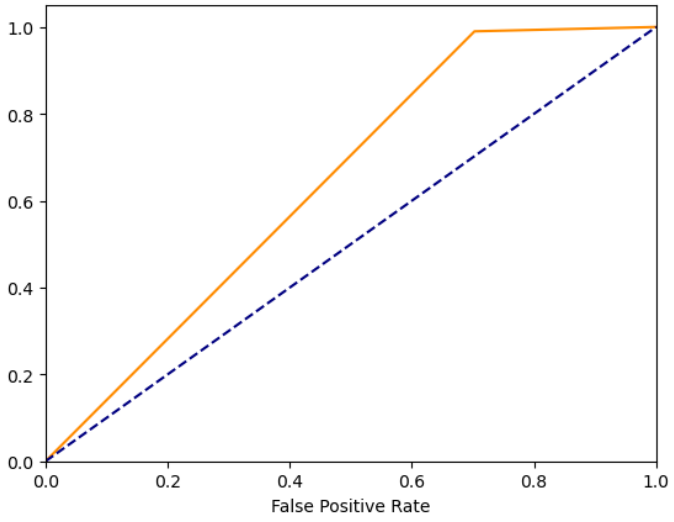


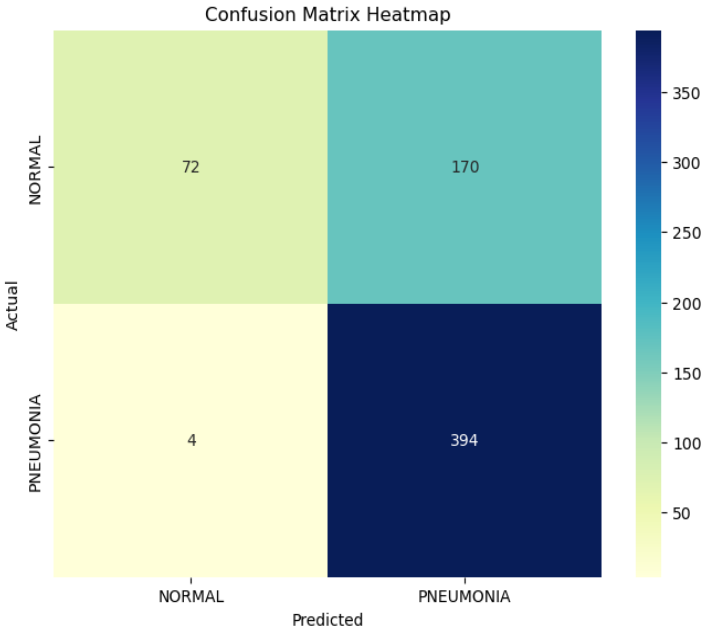
1.6.2 Catboost

The newest of the popular gradient boosting libraries, CatBoost (Categorical Boosting) was developed by the Russian tech company Yandex in mid-2017, following closely on the heels of LightGBM.

Unfortunately, I have yet to see CatBoost consistently outperform its competitors (though with many categorical features it does tend to come out on top), nor match the speed of LightGBM, but that could definitely change with future updates.

However, CatBoost was meant for cases such as categorical and text data, so please take the results of this article with a grain of salt when deciding which methods to try for your use-case.






```
Accuracy on test data: 0.73
Precision on test data: 0.79
Recall on test data: 0.73
F1-score on test data: 0.68
Confusion matrix on test data:
[[ 72 170]
 [  4 394]]
Classification report on test data:
              precision    recall  f1-score   support

           0       0.95      0.30      0.45       242
           1       0.70      0.99      0.82       398

    accuracy                           0.73       640
   macro avg       0.82      0.64      0.64       640
weighted avg       0.79      0.73      0.68       640
```

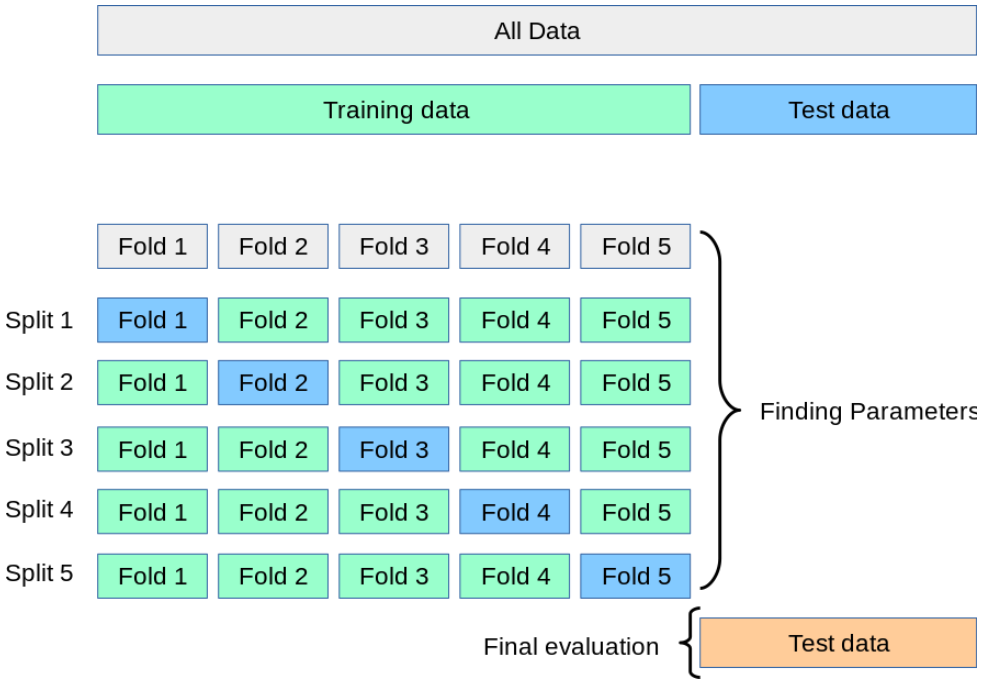
1.7 Optimisation of the model

1.7.1 Cross-Validation

Cross-Validation is a statistical method of evaluating and comparing learning algorithms by dividing data into two segments: one used to learn or train a model and the other used to validate the model.

The main advantage of cross-validation is that it allows for a more reliable estimate of a model's performance by reducing the dependence on a single train-test split. It helps in detecting overfitting (when a model performs well on the training data but poorly on new data) and provides a better understanding of how a model may perform on unseen data.

Common types of cross-validation include k-fold cross-validation, stratified k-fold cross-validation (used for imbalanced datasets), leave-one-out cross-validation (k equals the number of samples), and hold-out validation (where a separate validation set is kept aside from the training set).



1.7.2 Tune Hyperparameters Grid Search

Hyperparameter optimization is the process of finding the right combination of hyperparameter values to achieve maximum performance on the data in a reasonable amount of time.
Hyperparameters are parameters that are not learned during the training process, but rather set by the user before training the model.
GridSearch is a hyperparameter tuning technique used in machine learning to find the best combination of hyperparameters for a given model.It works by exhaustively searching through a specified set of hyperparameter values to find the combination that yields the best performance .It operates by defining a grid of possible values for each hyperparameter and then evaluating the model's performance using cross-validation for each combination of values in the grid.

We can try to find our best hypermater

```
# Définir les hyperparamètres possibles
param_grid = {
    'max_depth': [3, 5],
    'learning_rate': [0.1, 0.01],
    'n_estimators': [50, 100],
    'subsample': [0.5, 0.75],
    'colsample_bytree': [0.5, 0.75]
}

# Créer un objet GridSearchCV avec votre modèle et la grille de paramètres
grid_search = GridSearchCV(XGBClassifier(random_state=0, eval_metric='mlogloss'), param_grid, cv=5, n_jobs=-1)
```

```
xgb_classifier = XGBClassifier(**best_params, random_state=0, eval_metric='mlogloss')
xgb_classifier.fit(X_train, y_train)
```

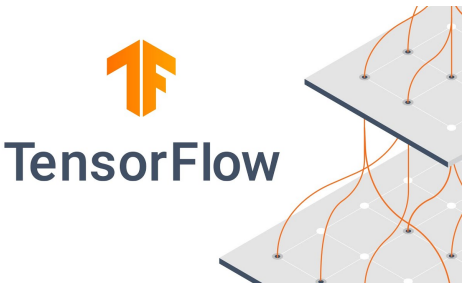
XGBoost with GridSearch parameters = 14 minutes

```
Performance du modèle sur les données de test:
**********
   Evaluation du modèle
**********
Accuracy on test data: 0.77
Precision on test data: 0.81
Recall on test data: 0.77
F1-score on test data: 0.74
Confusion matrix on test data:
[[101 141]
 [  6 392]]
Classification report on test data:
              precision    recall  f1-score   support

           0       0.94      0.42      0.58       242
           1       0.74      0.98      0.84       398

    accuracy                           0.77       640
   macro avg       0.84      0.70      0.71       640
weighted avg       0.81      0.77      0.74       640
```

```
Performance du modèle sur les données de test:
**********
   Evaluation du modèle
**********
Accuracy on test data: 0.77
Precision on test data: 0.82
Recall on test data: 0.77
F1-score on test data: 0.74
Confusion matrix on test data:
[[ 99 143]
 [  5 393]]
Classification report on test data:
              precision    recall  f1-score   support

           0       0.95      0.41      0.57       242
           1       0.73      0.99      0.84       398

    accuracy                           0.77       640
   macro avg       0.84      0.70      0.71       640
weighted avg       0.82      0.77      0.74       640
```

WITH GRIDSEARCH / WITHOUT GRIDSEARCH

2. Deep Learning with Tensorflow



TensorFlow is an open-source machine learning framework developed by Google. It is designed to provide a flexible and efficient way to implement and deploy machine learning models. TensorFlow supports various types of numerical computations and is particularly popular for building deep learning models.

TensorFlow offers a high-level API called Keras, which provides a user-friendly interface for building and training neural networks. Keras simplifies the process of defining network architectures, specifying layers, and configuring training parameters. TensorFlow also provides a lower-level API that gives users more control and flexibility over the model's implementation.

2.1 Setup and Load Data

Tensorflow can also work on GPU, choose your proper install for tensorflow:

https://www.tensorflow.org/install

```
# CPU
%pip install tensorflow-cpu matplotlib sklearn seaborn chardet scipy pandas numpy
```

```
# GPU
%pip install "tensorflow==2.10.*" matplotlib sklearn seaborn chardet scipy pandas numpy
```

You can import tensorflow and its libraries

```
# acess to our files
import os
import json

# manipulate our data
import numpy as np
import pandas as pd

# build our model
import tensorflow as tf
from tensorflow import keras
from keras import layers

# visualize
import matplotlib.pyplot as plt
import seaborn as sns
```

With keras.utils.image_dataset_from_directory(), We can load data, 2 parameters are expected :

image size: necessary to ensure consistency in the shape of the input data. Images can have different dimensions, and by specifying a fixed height and width, you define a standard size that the input images will be resized or cropped to.

batch_size: refers to the number of training examples processed together in a single iteration during training. It affects memory usage, computational efficiency, and the optimization process. Smaller batch sizes allow more frequent weight updates but can be slower, while larger batch sizes can be more computationally efficient but might generalize less. The choice of batch size is a trade-off and can be tuned to optimize training performance based on the specific task and available resources.

In our dataset, we have 10 validation data for 5000 training data. However Tensorflow required a large amount of validation data compared to scikit learn. Because of that, we decided to move the validation data into training and take 20% of the test data as validation data.

```
# Define the paths to the training and test directories
train_data_dir = '../data/chest_Xray/train'
test_data_dir = '../data/chest_Xray/test'

# Define the path to the directory containing the augmented images
aug_data_dir = '../data/chest_Xray/train_augmented'


# Image size and batch size
img_height, img_width = 180, 180
batch_size = 32

# load and preprocess the dataset and store them in a batch (32 images per 32)
print('TRAINING_DS:')
train_ds, val_ds = tf.keras.utils.image_dataset_from_directory(
    train_data_dir,
    image_size=(img_height, img_width),
    batch_size=batch_size,
    validation_split=0.2,
    subset="both",
    color_mode="grayscale",
    seed=0,
)

# load class_names
class_names = train_ds.class_names
num_classes = len(class_names)
# used for the last layer of models (the output): it is 2 ["NORMAL", "PNEUMONIA"]
```

```
TRAINING_DS:
Found 5216 files belonging to 2 classes.
Using 4173 files for training.
Using 1043 files for validation.
```

```
TEST_DS:
Found 640 files belonging to 2 classes.
```

In total, we have:
For training: 4173 and 1043 validation
For testing: 640 data

2.2 Visualize Data

A dataset is constituted with tensorflow Tensor:

```
for batch in train_ds:
    inputs, labels = batch
    print(tf.shape(inputs))
```

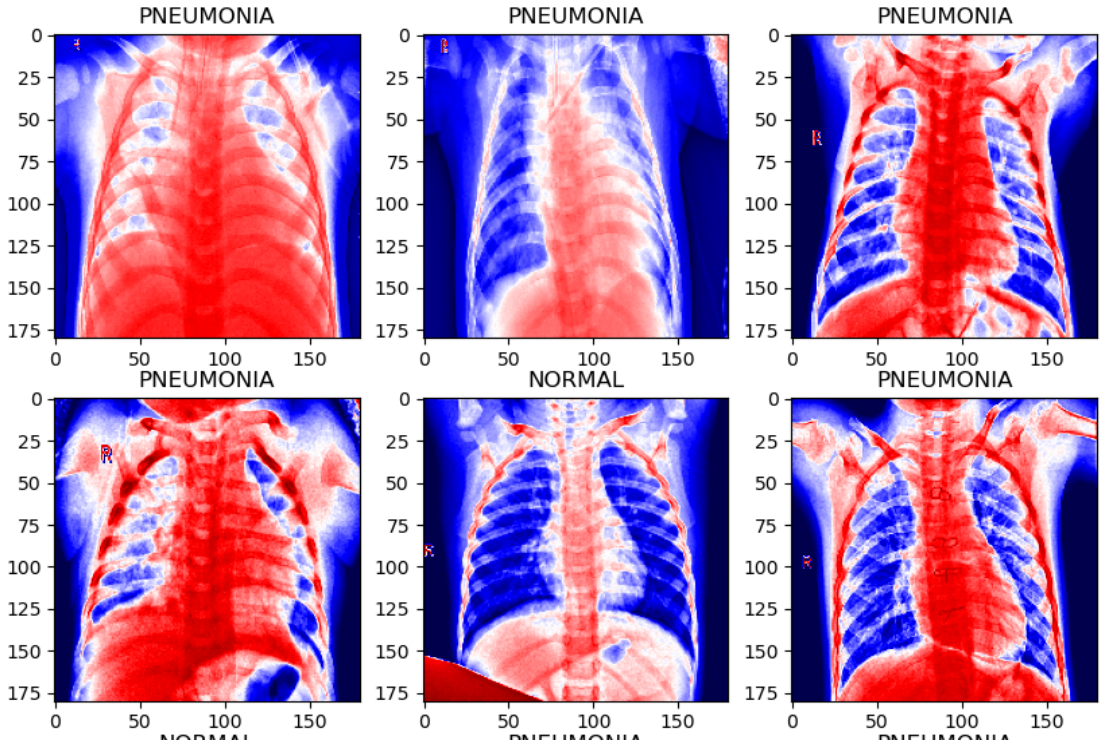
With the library matplotlib, we can visualize our datasets

```
tf.Tensor([ 32 180 180   1], shape=(4,), dtype=int32)
tf.Tensor([ 32 180 180   1], shape=(4,), dtype=int32)
...
tf.Tensor([ 32 180 180   1], shape=(4,), dtype=int32)
tf.Tensor([ 32 180 180   1], shape=(4,), dtype=int32)
tf.Tensor([ 32 180 180   1], shape=(4,), dtype=int32)
tf.Tensor([ 13 180 180   1], shape=(4,), dtype=int32)
```

There are tensor of 32 batch size, 180x180, and grayscale. (There are some 13 because 5216 %32 != 0.

You can visualize your images and classes with the library matlotlib

```
plt.figure(figsize=(10,10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i], cmap="seismic") # cmap works only with grayscale
        plt.title(class_names[labels[i]])
```



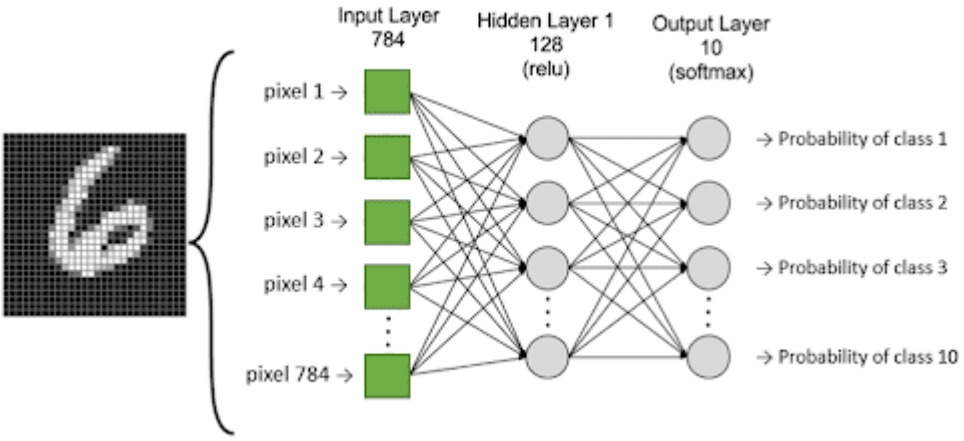
2.3 What is a Neural network ?

A neural network is a computer model inspired by the human brain. It consists of interconnected artificial neurons organized into layers. These neurons take inputs, perform computations, and produce outputs. The connections between neurons have weights that determine their influence on the output. Neural networks learn by adjusting these weights based on the difference between their predictions and desired outputs.

Neural networks are used to solve complex problems by learning patterns from data. They have layers, including an input layer for receiving data, hidden layers for processing information, and an output layer for producing results. The network adjusts its weights during training to make better predictions.

There are different types of neural networks. Feedforward networks, like multi-layer perceptrons, process data in one direction. Recurrent networks handle sequential or time-dependent data by incorporating feedback connections. Convolutional networks specialize in analyzing grid-like data, such as images.

You can experiment and get more informations on tensorflow:
https://playground.tensorflow.org/



In this article, we will see MLP (Multi-layer perceptrons) and with CNN (Convolutional networks).

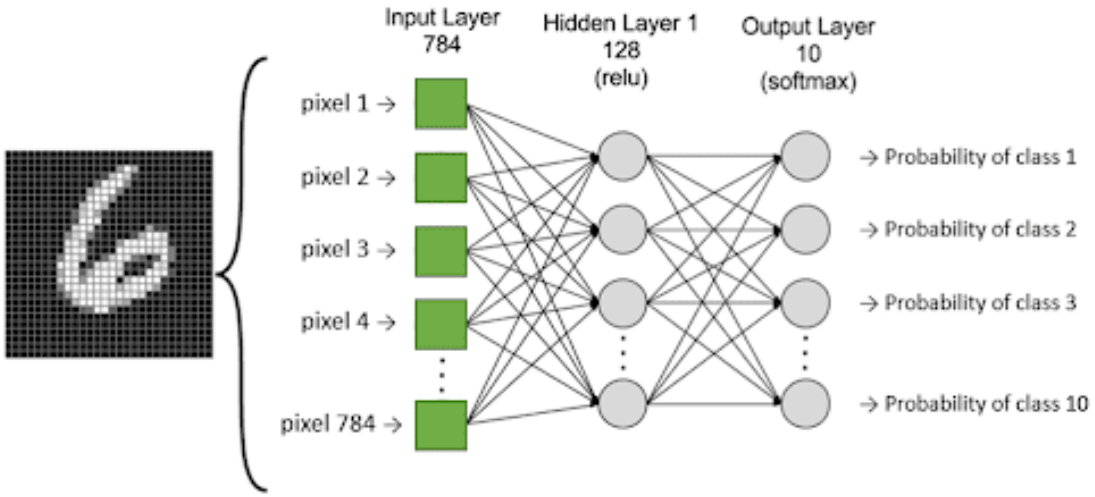
2.4 The Multi-Layer Perceptron (MLP)

2.4.1 Définition

MLP stands for Multi-Layer Perceptron, which is a type of artificial neural network. An MLP consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function.

MLP uses a supervised learning technique called backpropagation for training. It's a class of feedforward artificial neural network, meaning that data travels in one direction from input to output.

While it can be used for image classification tasks, it is generally less effective than CNNs because it does not exploit the spatial structure and local patterns in images. Moreover, MLPs typically require more parameters and are more prone to overfitting compared to CNNs, especially when dealing with larger images.



2.4.2 Building

We can build the model with keras.Sequential()

keras.Sequential() is like a container that holds different layers of a neural network in a specific order. It helps you build a neural network by stacking layers one after another, forming a sequential flow of information.

```
# Build our model
model = tf.keras.Sequential([
    # input
    layers.Rescaling(1./255, input_shape=(img_height, img_width, 1)),
    # hidden layers
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    # output
    layers.Dense(num_classes, activation='softmax'),
])
```

1. `layers.Rescaling()`: used for rescaling the input image data. It divides the pixel values by 255, which brings them into the range of 0 to 1. This rescaling step is important for standardizing the input data and improving the model's performance. The `input_shape` parameter specifies the dimensions of the input images, in this case, `img_height` and `img_width`, with a single color channel (grayscale).
2. `tf.keras.layers.Flatten()`: This layer flattens the input data into a 1D vector. In your case, it takes the output from the previous layer (which is a 2D image representation) and reshapes it into a single long vector. This flattening step is necessary before connecting to a fully connected layer.
3. `tf.keras.layers.Dense()`: This layer is a fully connected layer with 128 neurons. Each neuron is connected to every neuron in the previous layer. The `activation='relu'` parameter specifies the activation function used by the neurons, which is the Rectified Linear Unit (ReLU) function. ReLU introduces non-linearity to the network, allowing it to learn complex patterns and relationships in the data.
4. `layers.Dense(num_classes, activation='softmax')`: This is the output layer of the model. It consists of `num_classes` neurons, which corresponds to the number of classes or categories in the dataset you are working with. The `activation='softmax'` parameter applies the softmax function to the output, which converts the raw predictions into probabilities. The probabilities represent the likelihood of the input image belonging to each class.

   Compiling the model in Keras is an essential step before training it. When you compile a model, you're setting up the learning process, which is done by defining key aspects of the training mechanics.

   Here's what happens when you compile a model:

- Optimization Method is Selected: The optimizer is the algorithm that adjusts the weights and biases during training. Examples include Adam, RMSprop, and Stochastic Gradient Descent (SGD). These optimizers use the backpropagation algorithm to minimize the model's error function.

- Loss Function is Selected: The loss function, also known as the objective function, is the function that the model will aim to minimize during training. The choice of loss function depends on the problem at hand (e.g., binary cross-entropy for binary classification, categorical cross-entropy for multi-class classification, mean squared error for regression).

- Metrics for Evaluation are Selected: These are the criteria that we use to evaluate the performance of the model. Examples of metrics include accuracy, precision, recall, and F1 score. These metrics are not used for training the model, but provide additional context for how well the model is performing.

```
model.compile(
    optimizer="adam",
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

You can visualize your model with the function summary()

```
# view all the layers of the network
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling (Rescaling)       (None, 180, 180, 1)       0

 flatten (Flatten)           (None, 32400)             0

 dense (Dense)               (None, 128)               4147328

 dense_1 (Dense)             (None, 2)                 258

=================================================================
Total params: 4,147,586
Trainable params: 4,147,586
Non-trainable params: 0
_________________________________________________________________
```

2.4.3 Training

Once our model is built, we can train with our training dataset.

The fit method in Keras is used to train the model for a fixed number of epochs (iterations on a dataset). Here's what each of the parameters does:

epochs: This is the number of times the learning algorithm will work through the entire training dataset. For example, if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete one epoch.
callbacks: These are functions that can be applied at certain stages of the training process, such as at the end of each epoch.
Early stopping: is a form of regularization used to avoid overfitting when training a machine learning model with an iterative method, such as gradient descent.

During training, the model's ability to generalize often improves up until a certain point. After that, as the model continues to minimize the loss on the training data, it starts to learn patterns that are specific to the training data, which may not generalize to unseen data. This is known as overfitting.

Early stopping works by monitoring a specified metric (like validation loss or validation accuracy) during the training of the model. If the performance on the validation set doesn't improve for a specified number of epochs (referred to as the "patience"), the training process is stopped.

The main benefits of using early stopping are:

- Preventing Overfitting: By stopping training when the validation error starts to increase, early stopping prevents the model from learning spurious patterns in the training data that don't generalize.
- Reducing Computational Burden: If early stopping halts training early, you can save computational resources as you don't have to perform as many epochs.

Automating the choice of the number of epochs: Without early stopping, you'd have to specify the number of epochs to train for. If you set this number too low, the model may underfit. If you set it too high, the model may overfit. Early stopping automates this decision: you can set a high maximum number of epochs, and let early stopping decide when to halt training.

```python
early_stopping = tf.keras.callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
    verbose=1
)

history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=100,
    callbacks=[early_stopping]
)
```

```
Epoch 1/100
131/131 [==============================] - 16s 117ms/step - loss: 0.8290 - accuracy: 0.8390 - val_loss: 0.3741 - val_accuracy: 0.8629
Epoch 2/100
131/131 [==============================] - 7s 47ms/step - loss: 0.2152 - accuracy: 0.9214 - val_loss: 0.1460 - val_accuracy: 0.9396
Epoch 3/100
131/131 [==============================] - 7s 50ms/step - loss: 0.2395 - accuracy: 0.9104 - val_loss: 0.4814 - val_accuracy: 0.8092
Epoch 4/100
131/131 [==============================] - 7s 51ms/step - loss: 0.1869 - accuracy: 0.9312 - val_loss: 0.1209 - val_accuracy: 0.9530
Epoch 5/100
131/131 [==============================] - 7s 48ms/step - loss: 0.1351 - accuracy: 0.9497 - val_loss: 0.1868 - val_accuracy: 0.9243
Epoch 6/100
131/131 [==============================] - 8s 55ms/step - loss: 0.1526 - accuracy: 0.9442 - val_loss: 0.1012 - val_accuracy: 0.9655
Epoch 7/100
131/131 [==============================] - 7s 51ms/step - loss: 0.1332 - accuracy: 0.9528 - val_loss: 0.1089 - val_accuracy: 0.9578
Epoch 8/100
131/131 [==============================] - 7s 52ms/step - loss: 0.1330 - accuracy: 0.9494 - val_loss: 0.0961 - val_accuracy: 0.9684
Epoch 9/100
131/131 [==============================] - 7s 55ms/step - loss: 0.1382 - accuracy: 0.9466 - val_loss: 0.2118 - val_accuracy: 0.9195
Epoch 10/100
131/131 [==============================] - 7s 52ms/step - loss: 0.1555 - accuracy: 0.9425 - val_loss: 0.2134 - val_accuracy: 0.9175
Epoch 11/100
131/131 [==============================] - 7s 52ms/step - loss: 0.1102 - accuracy: 0.9607 - val_loss: 0.0930 - val_accuracy: 0.9703
Epoch 12/100
131/131 [==============================] - 8s 54ms/step - loss: 0.1266 - accuracy: 0.9533 - val_loss: 0.1631 - val_accuracy: 0.9281
Epoch 13/100
...
Epoch 62/100
130/131 [============================>.] - ETA: 0s - loss: 0.0605 - accuracy: 0.9762Restoring model weights from the end of the best epoch: 42.
131/131 [==============================] - 9s 58ms/step - loss: 0.0603 - accuracy: 0.9763 - val_loss: 0.1413 - val_accuracy: 0.9607
Epoch 62: early stopping
```

2.4.4 Plotting our model performance

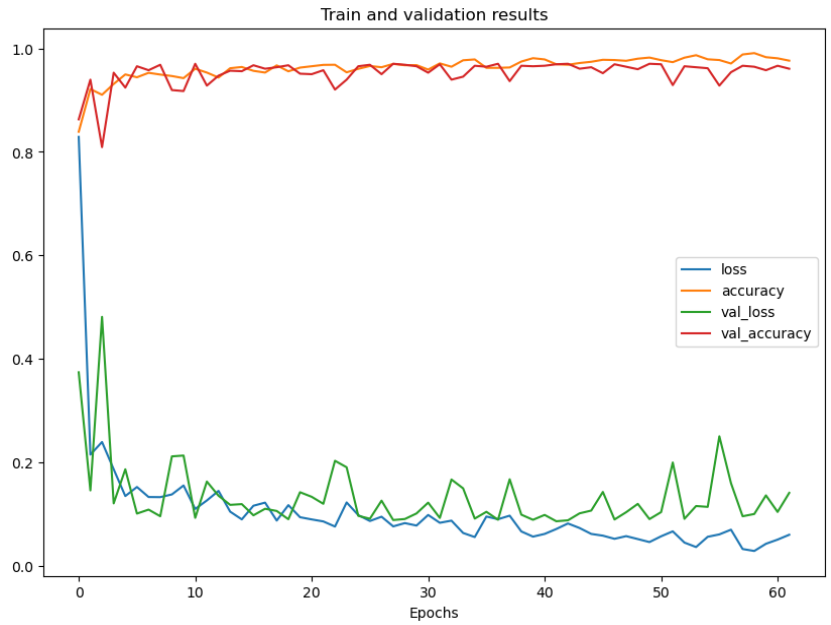
The fit method returns a History object, which is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values.

```
history.history
```

```
{'loss': [0.8289539813995361,
  0.21520675718784332,
  0.2395300269126892,
  0.18690195679664612,
  0.1350652426481247,
  0.1526181548833847,
  0.13322484493255615,
  0.1330084651708603,
  0.1382337510585785,
  0.15553416311740875,
  0.11015284061431885,
  0.1265794783830642
```

With the library pandas and matplotlib, we can visualize the training:

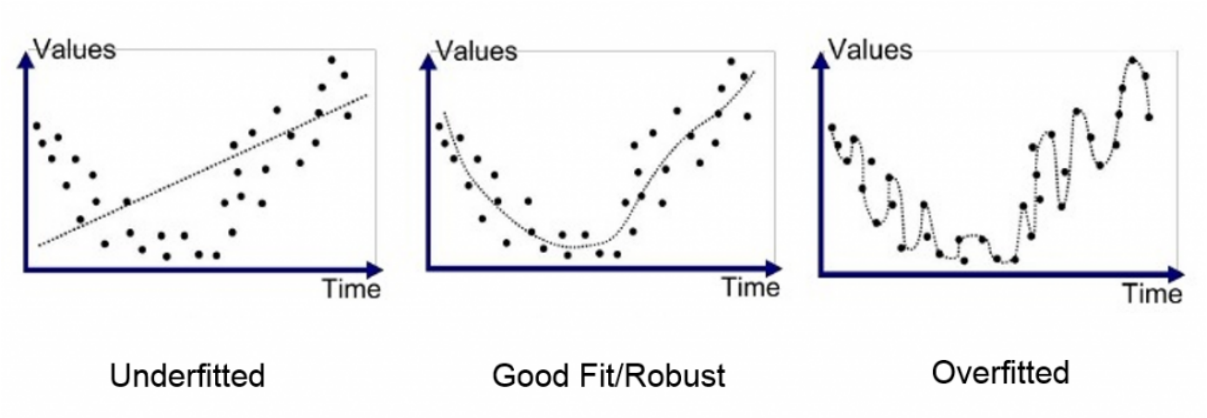
```
def visualize_model_training(history):
    pd.DataFrame(history).plot(title="Train and validation results",figsize=(10,7));
    plt.xlabel('Epochs')
    plt.legend()
    plt.show()
```



The early stopping stopped the process to go back to epoch 42.
We can see that it is the best result: starting from ’epoch 45 the validation loss increases: while the accuracy is at 95%. no more increase in validation loss, the model start to become overfitted (explained after)
The loss is the penalty of a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater.

Now that our model has obtained a good training, we can test on new data: the “test dataset”.

What are Overfitting and Underfitting ?



To begin with, both are bad, Overfitting means the model got trained to target exactly the data it was trained with, making it useless as it won't be able to recognise more randomized data.
Overfitting can happen when the model is too trained.
There are ways to avoid overfitting :

- Using larger dataset
- Cross Validation
- Early stopping
- Simpler model

Underfitting means the model fails to recognise the relationship betweens the data, it happens when there is not enough data or too much and not enough training.

The trick is to find the perfect balance between the two.

2.4.5 Evaluate performance

We can test our model with the function evaluate()

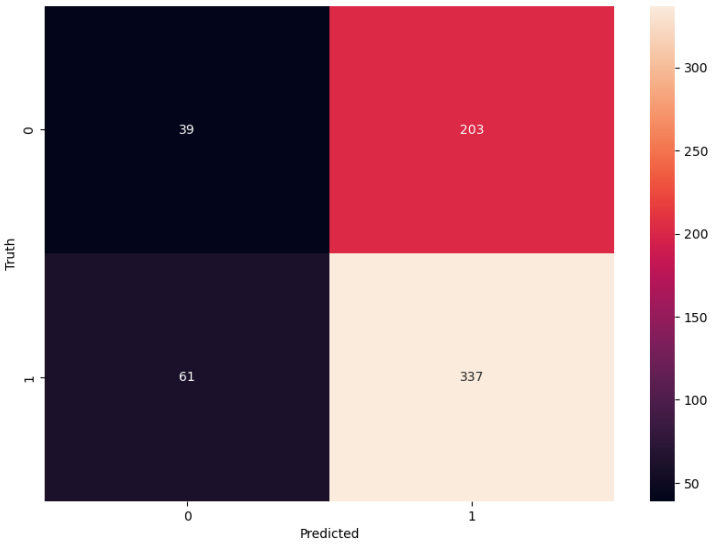
```
# Test model with test dataset
loss, accuracy = model.evaluate(test_ds)
print(f"Test accuracy: {accuracy}")
```

```
20/20 [==============================] - 1s 23ms/step - loss: 1.0387 - accuracy: 0.7656
Test accuracy: 0.765625
```

Our model accuracy is not good: 76%. He is equivalent to our Scikit learn models.
We can try to identify the problem with a confusion matrix:

```
from sklearn.metrics import confusion_matrix
# Get the model's predictions
def visualize_confusion_matrix(model):
    y_pred = model.predict(test_ds)

    # If your model outputs probabilities and you want to convert these probabilities into class labels
    y_pred_classes = np.argmax(y_pred, axis=1)

    # Now, extract the true labels from test_ds
    y_true = np.concatenate([y for x, y in test_ds], axis=0)

    # Compute the confusion matrix
    cm = confusion_matrix(y_true, y_pred_classes)
    # Print the confusion matrix
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='d')
    plt.xlabel('Predicted')
    plt.ylabel('Truth')
    plt.show()
```

Our model correctly predicted "NORMAL" 61 times and incorrectly predicted "NORMAL" 61 times. Similarly, it correctly predicted "PNEUMONIA" 337 times and incorrectly predicted "PNEUMONIA" 203 times.
On this new data, our model cannot identify pneumonia. For him the majority of images are pneumonia, the cause is from our dataset, we have only 1300 “normal” data for 4000 “pneumonia”. He is overtraining on pneumonia.
MLP = not good for image classification

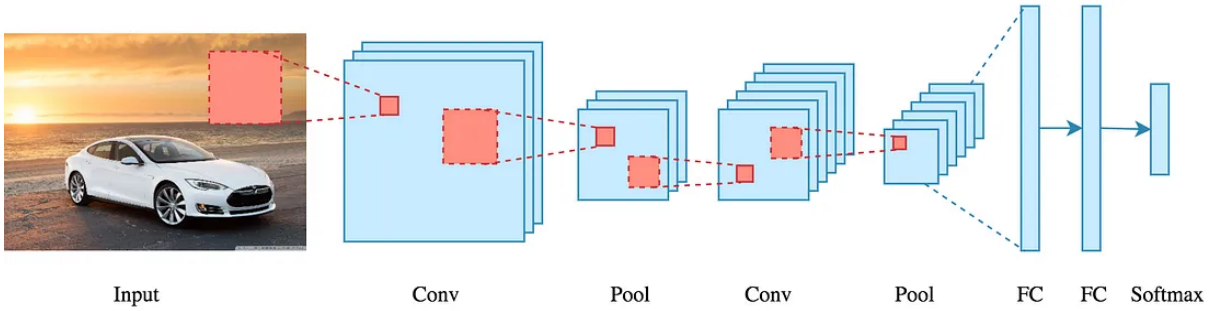


CNN are known to be stronger for this type of classification because it can detect patterns in images. Let’s see it.

2.5 Convolutional Neural network (CNN)

2.5.1 Définition

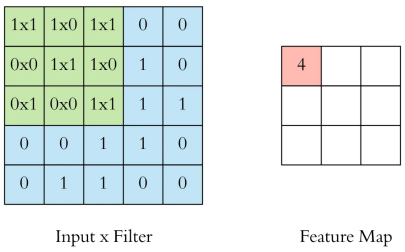
A Convolutional Neural Network (CNN) is a type of artificial neural network used primarily for processing images. Its main advantage compared to other neural networks is its ability to detect “features” (patterns) inside the input. A CNN uses 3 types of layers, which all have unique responsibilities : *convolutions*, *pooling layers*, and *fully connected layers.*



1. Convolution layers

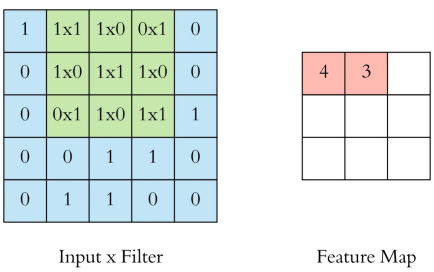
The term "convolution" refers to the mathematical operation that is applied to the input data using a "filter" or a "kernel". This filter is a small matrix of weights which is passed over the input image to transform it into a feature map. This matrix can be of any size, but common sizes include 3*x3, 5*5 and 7*7. The size chosen depends on the level of detail we want to capture. Other parameters are the stride, which controls how much the filter moves each time it shifts on the input, and the padding allows to add pixels around the input image, often in order to have an output of the same shape as the input. Here’s how a filter works :

The filter is applied on the first 3*3 window …



Input x Filter Feature Map

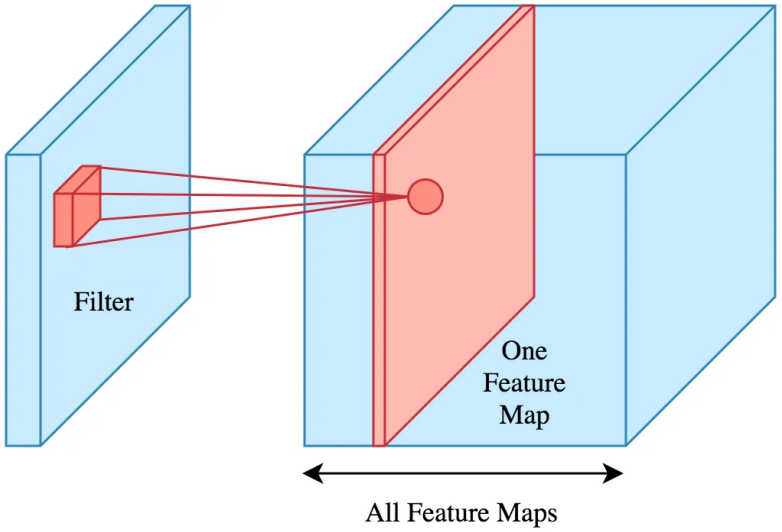
… then it moves to the next 3*3 window …



Input x Filter Feature Map

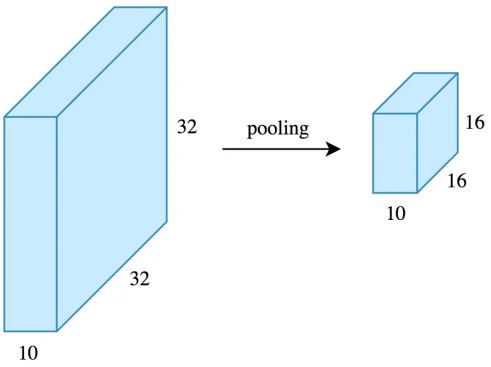
… and so on, until the feature map is filled.

A single convolution layer usually has multiple filters, resulting in multiple feature map. This means that convolution layers increase the dimensionality of the input. For instance, a convolution applied to an image of dimensions 180*180*1 (a grayscale image of size 180*180) might output an image of dimensions 180*180*10. That would mean that 10 filters were applied on the input, resulting in 10 feature maps.

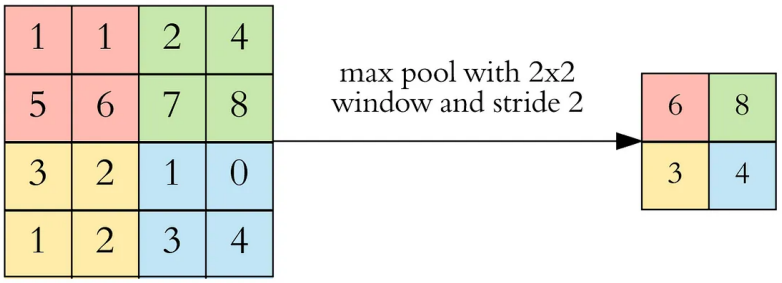


2. Pooling

Pooling is used after convolution layers to decrease the dimensionality of the input. It works in the same way filters do : a window shifts on the input image and outputs scalar values which represent the entire window in the output. A stride greater than 1 is used so the dimensions of the output is lesser than that of the input.

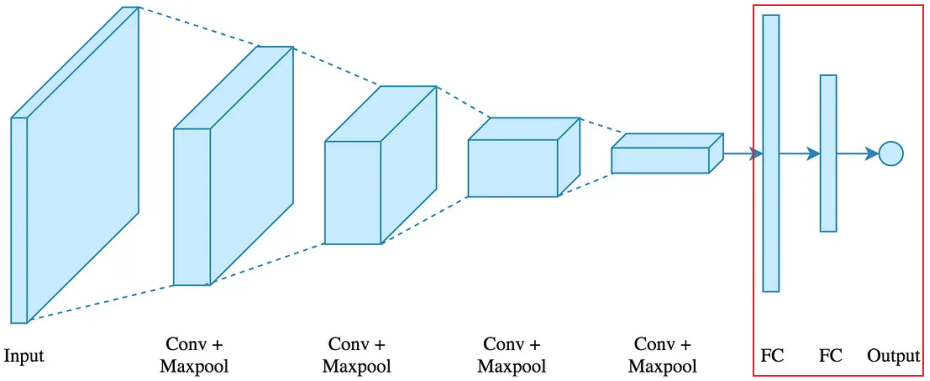


The most common type of pooling is *max pooling*, which consists of taking the maximum value of a selection of pixels as output.



3. Fully connected layers

After the convolution and pooling layers, the results of all feature maps are flattened into a vector which is linked to a basic Dense Neural Network, which is responsible for returning the model's output.



2.5.2 Building

We can build a CNN model with Keras's sequential API. Here's an example :

```
model = tf.keras.Sequential([
    layers.Rescaling(1./255, input_shape=(img_height, img_width, 1)),

    layers.Conv2D(16, 3, activation="relu"),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, activation="relu"),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation="relu"),
    layers.MaxPooling2D(),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation="relu"),
    layers.Dense(num_classes, activation="softmax"),
])

model.compile(
    optimizer="adam",
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

1. Layers.Conv2D(16, 3, "relu")

Defines convolution layers. Here are the arguments we used :

- *filters* : number of filters to apply to the input. Consequently, this also defines the 3rd dimension of the output (the number of feature maps produced by the layer)
- *filter size*: size of the filter that will shift over the input. 3*3 is the standard choice
- *strides:* Here the argument isn't specified and defaults to (1,1), which means the filter moves one pixel at a time both horizontally and vertically.
- *padding:* Here the argument isn't specified and defaults to "valid", which means no padding is applied. This will result in width and height dimensions being reduced by 2 in the output, since the filter size is 3.

2. Layers.MaxPooling2D()

Define max pooling layers. The default values result in the dimensions of the input being divided by 2 in the output.

- *pool size :* Size of the window from which the maximum will be extracted. Default value is (2, 2).
- strides : Specifies how far the pooling window moves for each pooling step. Default value is "None", which means it will use the pool size as strides.

3. Layers.Flatten() :

Flattens the input into a vector. For example, a 16*16*64 input will result in a 1*16,384 vector.

4. Layers.Dense(128, "relu")

Defines a basic DNN layer, where each of the 128 neurons is connected to every neuron of the previous layer.

5. Layers.Dense(2, "softmax")

Defines a basic DNN layer, where each of the 2 neurons represents the likelihood of the input belonging to a class. This is the output of our model. The softmax activation function turns the vector of values into a vector of probabilities (the sum of the values is equal to 1).

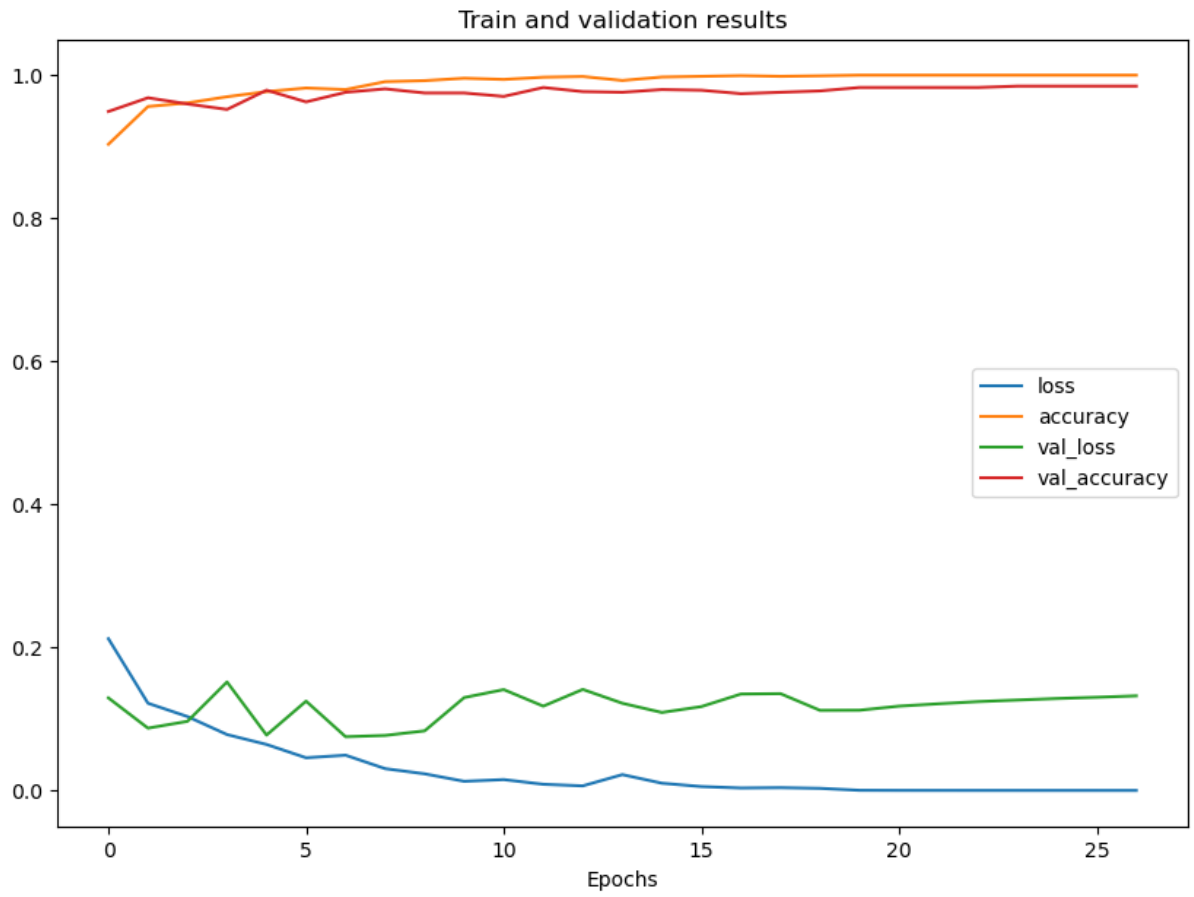
2.5.3 Training

```
early_stopping = tf.keras.callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
    verbose=1
)

history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=100,
    callbacks=[early_stopping]
)
```

```
Epoch 7/100
131/131 [==============================] - 8s 56ms/step - loss: 0.0493 - accuracy: 0.9799 - val_loss: 0.0751 - val_accuracy: 0.9760
```

```
...
Epoch 27/100
130/131 [============================>.] - ETA: 0s - loss: 2.4620e-05 - accuracy: 1.0000Restoring model weights from the end of the best epoch: 7.
131/131 [==============================] - 7s 48ms/step - loss: 2.4635e-05 - accuracy: 1.0000 - val_loss: 0.1323 - val_accuracy: 0.9847
Epoch 27: early stopping
```

2.5.4 Performance

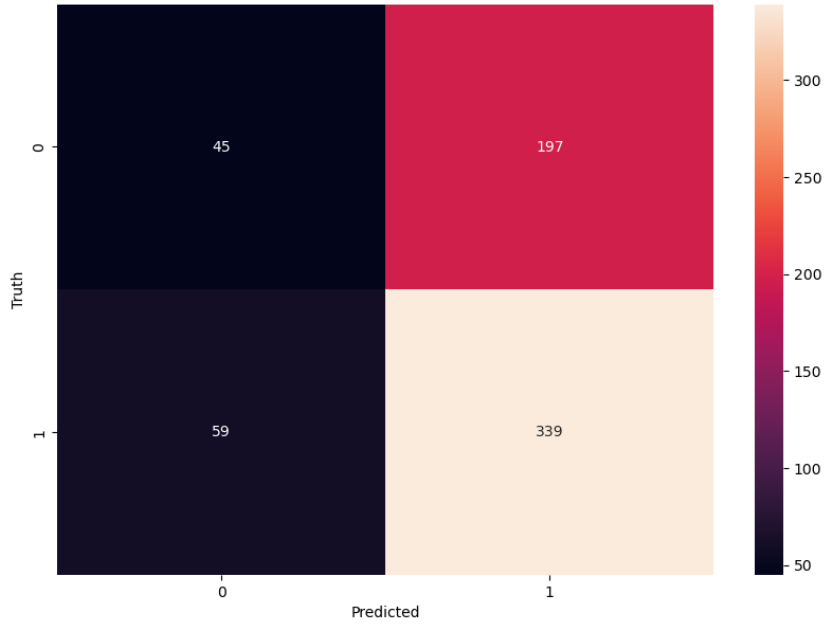


According to the results from our training, our model is already quite effective from the beginning, with only 7 epochs needed to reach a precision of 97% on the validation dataset. We can say it could effectively find the patterns to identify.

2.5.4 Test

```
# Test model with test dataset
loss, accuracy = model.evaluate(test_ds)
print(f"Test accuracy: {accuracy}")
```

```
20/20 [==============================] - 1s 22ms/step - loss: 0.9935 - accuracy: 0.7688
Test accuracy: 0.768750011920929
```



However, on the new dataset, it couldn’t identify the already trained patterns.
The model is promising, we can try to optimize it, our goal is to find a reliable and effective model.

2.5.4 Data augmentation

Data augmentation is a strategy used to increase the diversity of the data available to train the models without actually collecting new data. This technique helps to prevent overfitting and improve the performance of deep learning models.

Data augmentation techniques are applied to the training data, thereby augmenting it with new transformed versions of the original data. This means the model is exposed to more varied, but still relevant, data points during training, helping it to learn more robust, generalizable features.

In the context of images, these transformations can be:

- Flipping the image either horizontally or vertically.
- Rotating the image.
- Zooming in or out on the image.
- Cropping the image.
- Varying the color on the image.
- Translating the image either horizontally or vertically.

For this project, we have ~5000 data for training, which is very poor. Data augmentation can be helpful.
Keras contains its own layers to apply data augmentation:

```
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1)
])
```

But on tensorflow 2.10, it is bugged and slows the training. Instead of downgrading, we generated our data augmented images with ImageDataGenerator from tensorflow too.
Please check the "data_augmentation.ipynb" notebook to generate the augmented images.
To have a good variation, we multiplied the number of data by 2
data size:
- NORMAL: 1341 * 2 = 2682
- PNEUMONIA: 3875 *2 = 7750

In total we have 10468 images in our test_ds

```
# Create an instance of ImageDataGenerator
datagen = ImageDataGenerator(
    rotation_range=20,      # Rotate images by 20 degrees
    zoom_range=0.2,         # Zoom in/out images by 20%
    width_shift_range=0.1,  # Shift the width of the image by 10%
    height_shift_range=0.1, # Shift the height of the image by 10%
    horizontal_flip=True,   # Flip images horizontally
    vertical_flip=True      # Flip images vertically
)

aug_iter = datagen.flow_from_directory(
    train_dir,
    classes=['NORMAL', 'PNEUMONIA'],
    target_size=(224, 224),  # Resize images to 224x224 pixels
    batch_size=32,           # Generate images in batches of 32
    # save_to_dir='augmented_images/', # Save the augmented images to this directory
    save_prefix='aug_',      # Add a prefix to the filenames of the augmented images
    class_mode='categorical',
    save_format='jpg'        # Save the augmented images in JPEG format
)
```

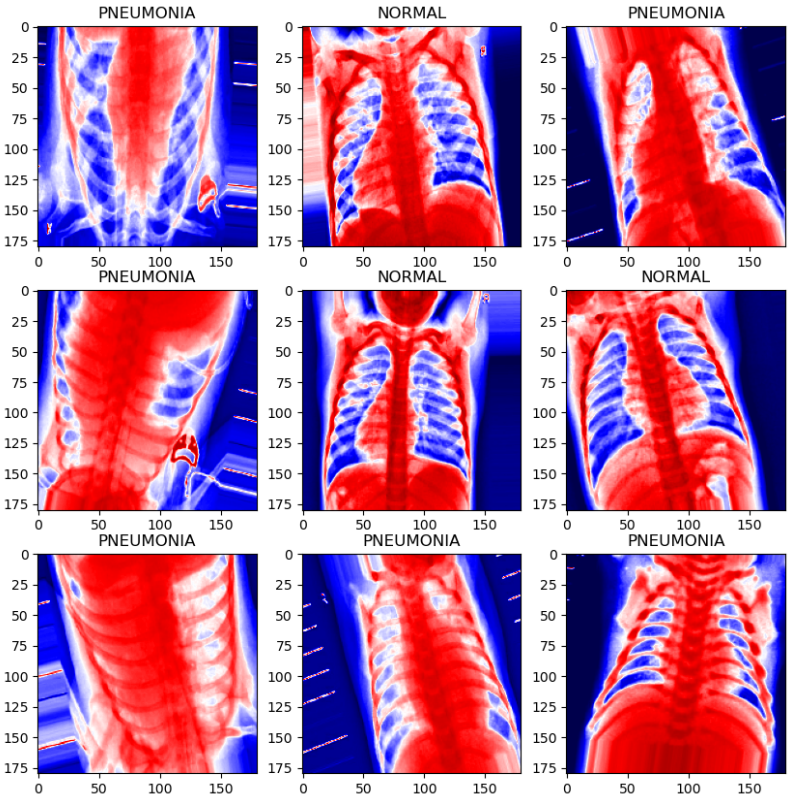
(you can check the full code in the data_augmentation.ipynb)

After generated our data, we can fetch them:

```
train_aug_ds, val_aug_ds = tf.keras.utils.image_dataset_from_directory(
    aug_data_dir,
    image_size=(img_height, img_width),
    batch_size=batch_size,
    color_mode="grayscale",
    validation_split=0.2,
    subset="both",
    seed=0,
)
```

Visualize them:

```
plt.figure(figsize=(10,10))
for images, labels in train_aug_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i], cmap="seismic") # cmap works only with grayscale
        plt.title(class_names[labels[i]])
```



We can merge our augmented images to the original dataset with the method concatenate()

```
# merge augmented data with real_data
train_ds = train_ds.concatenate(train_aug_ds)
val_ds = val_ds.concatenate(val_aug_ds)
```

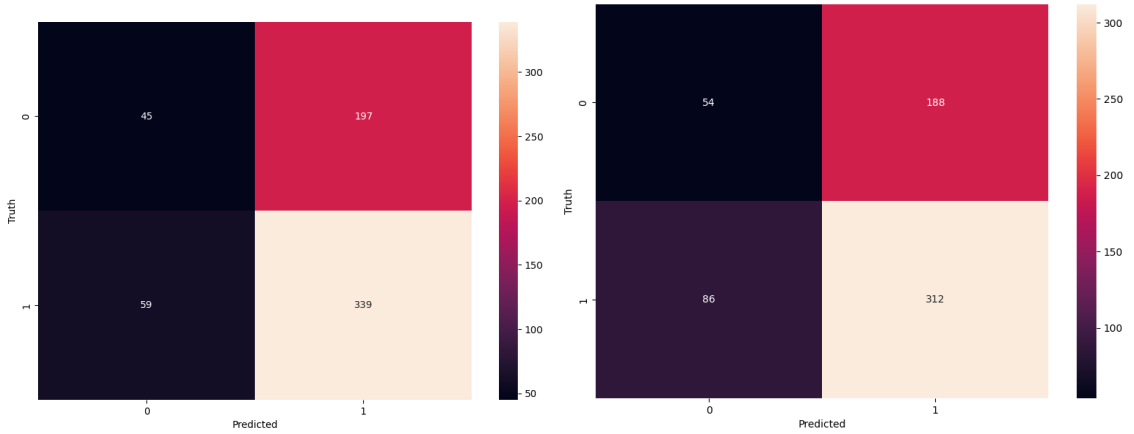
Once we have our augmented images on the training and v validation dataset, we can retrain our last model with this new dataset.

```python
# Retrain our last model with the used augmented ds
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=100,
    callbacks=[early_stopping]
)
```

```
Epoch 1/100
262/262 [==============================] - 10s 39ms/step - loss: 0.2339 - accuracy: 0.8983 - val_loss: 0.2418 - val_accuracy: 0.9185
Epoch 2/100
262/262 [==============================] - 9s 33ms/step - loss: 0.1783 - accuracy: 0.9274 - val_loss: 0.1610 - val_accuracy: 0.9391
Epoch 3/100
262/262 [==============================] - 8s 29ms/step - loss: 0.1391 - accuracy: 0.9426 - val_loss: 0.1525 - val_accuracy: 0.9386
...
Epoch 23/100
262/262 [==============================] - ETA: 0s - loss: 3.2429e-05 - accuracy: 1.0000Restoring model weights from the end of the best epoch: 3.
262/262 [==============================] - 9s 35ms/step - loss: 3.2429e-05 - accuracy: 1.0000 - val_loss: 0.4048 - val_accuracy: 0.9463
Epoch 23: early stopping
```

```python
# Test model with test dataset
loss, accuracy = model.evaluate(test_ds)
print(f"Test accuracy: {accuracy}")
```

```
20/20 [==============================] - 1s 19ms/step - loss: 0.6019 - accuracy: 0.8125
```



WITHOUT AUGMENTATION WITH AUGMENTATION

It’s quite effective as we increased the accuracy by 5% . More data is always the better

Now to optimize our data, we need to go to our model’s layers to try enhancing the accuracy even more.

2.5.5 Tuning

Our model already has an accuracy of 81% with no additional optimization. Each layer has parameters that can be modified.

Goal: find the best parameters to get a better model

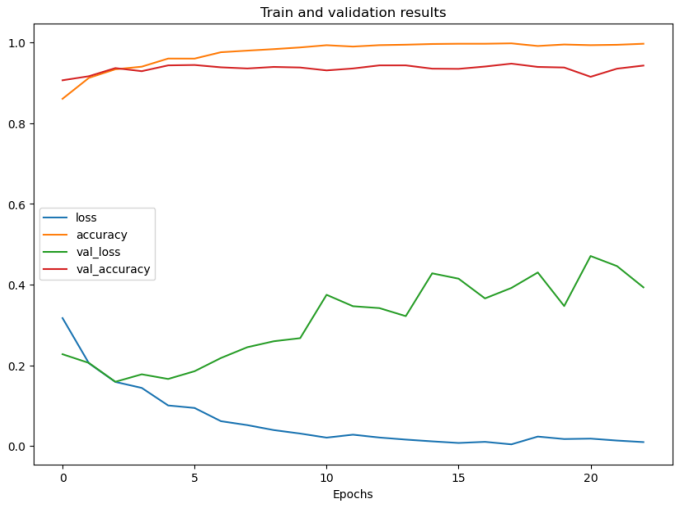
A model with more layers is not the best.

2.5.5.1 Manual searching

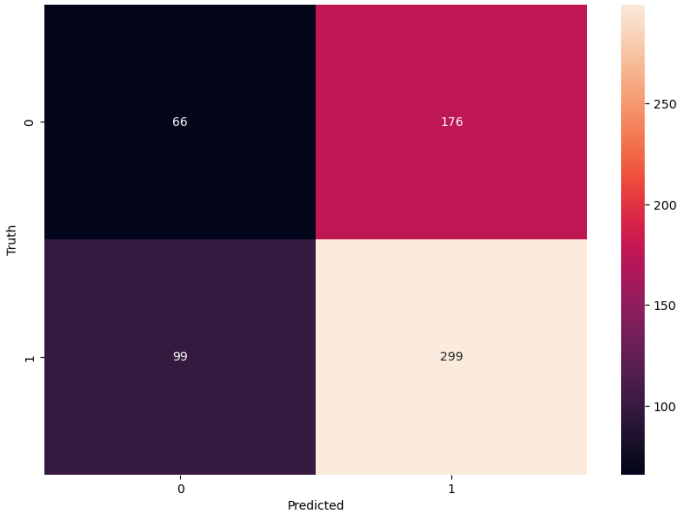
Here, we created a bigger screen and we added a Dropout Layer.

The Dropout layer in a neural network is a form of regularization that helps to prevent overfitting. During training, the dropout layer randomly sets a fraction of input units to 0 at each update, which helps to prevent overfitting. This fraction is the rate, and it's a hyperparameter that can be tuned. For example, if the dropout rate is set to 0.5, approximately half of the input units are dropped out during training.

```
model = tf.keras.Sequential([
    Input(shape=(img_height, img_width, 1)),
    layers.Rescaling(1./255),

    layers.Conv2D(64, (3, 3), activation="relu"),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation="relu"),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(256, (3, 3), activation="relu"),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation="relu"),
    layers.Dense(num_classes, activation="softmax"),
])
```




```
20/20 [==============================] - 3s 133ms/step - loss: 0.4871 - accuracy: 0.8359
```



More accuracy can be obtained by altering the parameters

This method works but is slow.
What if we wanted to find the optimal parameters for each layer ?
Keras has its own alternative of “grid search”, Keras-tuner

2.5.5.2 Automatize the search: keras_tuner

Random Search, Hyperband, and Bayesian Optimization are three different hyperparameter optimization methods used in machine learning to find the best model architecture and parameters.

- Random Search: Random Search is a simple and widely used method for hyperparameter optimization. In this method, hyperparameter combinations are sampled randomly from a predefined search space. The model is then trained and evaluated with each combination, and the best-performing combination is chosen. Random Search is computationally inexpensive but may require a large number of trials to find a good set of hyperparameters, especially in high-dimensional search spaces.

- Hyperband: Hyperband is an optimization method based on random search but with adaptive resource allocation and early stopping. It focuses on finding the best model with a limited budget of resources (like training time). The key idea behind Hyperband is to allocate more resources to the more promising models and stop training the less promising ones early. This is achieved by training models for varying numbers of epochs and adaptively allocating resources based on their performance. Hyperband is more efficient than Random Search as it can explore the search space more effectively by leveraging early stopping.

- Bayesian Optimization: Bayesian Optimization is a more advanced and efficient method for hyperparameter optimization. It uses a probabilistic model (usually a Gaussian Process) to model the objective function, which maps hyperparameter combinations to model performance. Based on the performance of previously evaluated models, it selects the next set of hyperparameters to try by optimizing an acquisition function that balances exploration (trying new hyperparameter combinations) and exploitation (focusing on known good combinations). Bayesian Optimization is more efficient than Random Search, as it uses prior knowledge about the performance of previous models to make informed decisions about which hyperparameter combinations to try next.

In summary, Random Search is the simplest method but may require more trials to find good hyperparameters. Hyperband improves upon Random Search by adaptively allocating resources and using early stopping, making it more efficient in finding good models within a limited budget. Bayesian Optimization is the most advanced method, using a probabilistic model to guide the search for optimal hyperparameters, often resulting in better model performance in fewer trials.

Here, we focused on the Convolutional layer. In general, tuning the MaxPooling2D layers is not as crucial as tuning other hyperparameters like the number of filters, kernel size, or learning rate. MaxPooling2D layers are used to reduce the spatial dimensions of the feature maps and provide a form of translation invariance to the model. The most common pooling size used is (2, 2), which reduces the spatial dimensions by a factor of 2.

In this project we use Bayesian Optimisation.

To use the keras-tuner, we have to create methods for the model and callbacks, just like the gridsearch it will create and train the model with selected parameters.

```
from keras_tuner import RandomSearch, Hyperband, BayesianOptimization
from keras_tuner.engine.hyperparameters import HyperParameters
import copy

def build_model(hp):
    model = tf.keras.Sequential([
        layers.Input(shape=(img_height, img_width, 1)),
        layers.Rescaling(1./255),

        layers.Conv2D(
            filters=hp.Int("conv_1_filter", min_value=32, max_value=128, step=16),
            kernel_size=hp.Choice("conv_1_kernel", values=[3, 5]),
            activation="relu"
        ),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(
            filters=hp.Int("conv_2_filter", min_value=32, max_value=256, step=16),
            kernel_size=hp.Choice("conv_2_kernel", values=[3, 5]),
            activation="relu"
        ),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(
            filters=hp.Int("conv_3_filter", min_value=32, max_value=256, step=16),
            kernel_size=hp.Choice("conv_3_kernel", values=[3, 5]),
            activation="relu"
        ),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.2),

        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(
            units=hp.Int("dense_units", min_value=32, max_value=128, step=16),
            activation="relu"
        ),
        layers.Dense(num_classes, activation="softmax"),
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(hp.Float("learning_rate", 1e-4, 1e-2, sampling="log")),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )

    return model
```

```
def create_early_stopping_callback():
    return EarlyStopping(
        monitor="val_loss",
        patience=10,
        min_delta=0.001,
        restore_best_weights=True
    )
```

```
tuner = BayesianOptimization(
    build_model,
    objective="val_accuracy",
    max_trials=20,
    executions_per_trial=3,
    directory="output",
    project_name="KerasTunerCNN"
)


# find the best parameters
tuner.search(train_ds,
             epochs=100, # You can increase the maximum number of epochs since early stopping will halt training when there is no improvement
             callbacks=[create_early_stopping_callback()],
             validation_data=val_ds,
             steps_per_epoch=len(train_ds),
             validation_steps=len(val_ds))
```

```
Trial 20 Complete [00h 13m 58s]
val_accuracy: 0.9576541980107626

Best val_accuracy So Far: 0.9576541980107626
Total elapsed time: 04h 16m 43s
INFO:tensorflow:Oracle triggered exit
```

near 4 hour needed to test lots of different parameters

```
best_model = tuner.get_best_models(num_models=1)[0]
best_hyperparameters = tuner.get_best_hyperparameters(num_trials=1)[0]

print("Best hyperparameters:")
print(best_hyperparameters.values)
```

We found the Best hyperparameters are:
{'conv_1_filter': 32, 'conv_1_kernel': 3, 'conv_2_filter': 176, 'conv_2_kernel': 5, 'conv_3_filter': 176, 'conv_3_kernel': 5, 'dense_units': 96, 'learning_rate': 0.00011451155720572537}

```
# Test model with test dataset
loss, accuracy = best_model.evaluate(test_ds)
print(f"Test accuracy: {accuracy}")

20/20 [==============================] - 1s 31ms/step - loss: 0.9683 - accuracy: 0.8188
```

Bayesian optimization is really efficient but it takes a lots of time.
For us, this model is less efficient than our manual search. But if we used a better searching range in the logic of the CNN parameters it could possibly find a better model.

3. Save and Load Model

3.1 Scikit-Learn

To save and load our model we used an external library named pickle

```
import pickle
```

saveModel:

```
    # enregistrer le modèle
    with open(model_path, 'wb') as file:
        pickle.dump(model, file)
```

loadModel:

```
    # lire le modèle
    with open(model_path, 'rb') as file:
        model = pickle.load(file)
    return model
```

3.2 Tensorflow

Keras contains his own methods to save and load our models using the Hierarchical Data Format (HDF5) file format with the `.h5` extension.

The `.h5` file format is a commonly used and supported format in Keras and can be easily used across different platforms and environments. It provides a straightforward way to store and share trained models while preserving all the necessary information required to reproduce or utilize the model.

```
from keras.models import load_model
```

```
folder = "./tf_models/MLP/"
# Save model
model.save(os.path.join(folder, "MLP.h5"))

# Save the history as a JSON file
with open(os.path.join(folder, "history.json"), 'w') as f:
    json.dump(history.history, f)
```

```
folder = "./tf_models/MLP/"

# Load the model
new_model = load_model(os.path.join(folder, "MLP.h5"))

# Load the history from the JSON file
with open(os.path.join(folder,"history.json"), 'r') as f:
    new_history = json.load(f)
```

4. Deploy a tensorflow model

For this project, we developed a basic website to show our tensorflow models. The goal was to allow the user to test our models either on a sample of images extracted from our validation dataset, or on an image they uploaded. To achieve that, we used react for the web app, and firebase for the back end. The prediction happens in the back end, so the user never has access to our models.

We decided to only deploy our tensorflow models. We learned that there are three major steps to using a model trained with tensorflow inside a javascript runtime :

1. Saving the model in the correct format

By default, tensorflow models are saved under the .h5 format in python. However, this format can not be loaded by javascript. There are 2 options to handle this :

a. Save the model using *tensorflowjs* in python

In python, add these lines to directly save the model in the correct format:

```
import tensorflowjs as tfjs

tfjs.converters.save_keras_model(model, f"../models")
```

b. Convert the .h5 save file using node's *tensorflowjs-converter*.

Install python's *tensorflowjs-converter* package, then use the following command to convert the .h5 save file to a format loadable in javascript :

```
tensorflowjs_converter --input_format keras --output_format tfjs_graph_model
<path/to/model.h5> <path/to/js/model>
```

Whether you use one method or the other, you will get a "model.json" file that defines the structure of the model, and multiple "part_x_of_X.shard" files that store the weights of the model. The "model.json" file contains the name of the .shard files, which has an importance for the next step.

2. Loading the model in a javascript environnement

To load a model from a javascript runtime, you need to install tensorflowjs and use the following code :

```
const model = await tensorflowjs.loadGraphModel(modelUrl);
```

Where "modelUrl" is a public link to download the model. One thing to note is that *tensorflowjs.loadGraphModel()* will download the model.json file, read it to get the names of the .shard files, then request the same domain for each .shard file, expecting to find them at

the same route (ex: if the model was downloaded at *www.domain.com/models/model.json,* tensorflowjs will try to get the weight at *www.domain.com/models/shardFileX* by default). This can be a problematic behavior, but it can be overwritten.

3. Performing normalization on the user data

For the model to work on the data we feed it, the data must be formatted in the same way the training data was. For instance, we trained our models on 180*180 grayscale images, so we had to resize the images sent by the user and convert them to grayscale before using the model on them. We did this using node's sharp library :

```
const imageData = await sharp(imageBuffer)
                                    .resize(180, 180)
                                    .greyscale()
                                    .toBuffer();
```

4. Performing the prediction

To use the model and make a prediction, simply use :

```
// run the model on the desired image
// prediction array looks like [ [0.691351321321, 0.209132131] ]
const prediction = await model.predict(tensor);

// get indexes of max values in prediction tensor
// (-1 is for selecting the last axis)
const predictedClass = prediction.argMax(-1).dataSync()[0];

// use index of most probable class to get its label
if (predictedClass === 0) {
    const result = "sane";
} else {
    const result = "sick";
}
```

Something we noticed is that using the same model on the same image after normalizing it to the same format resulted in slightly different results whether we loaded the model with python or with javascript. For example, when predicting a specific image in python with one of our CNN, we obtained the class "PNEUMONIA" with a probability of 56,7% in python, while using the same model on the same image formatted in the same way in nodejs resulted in a probability of 57,3%. There are several reasons explaining this :

- Image processing libraries we use in python and javascript are not the same, which means they can handle the same image normalization operations (conversion to grayscale, …) in slightly different ways, resulting in slightly different data.
- Python and JavaScript handle floating point arithmetic slightly differently. This can introduce small discrepancies, especially when a lot of operations are involved like in deep learning predictions.

    - For models trained in Python and then converted to tensorflowjs format, some small discrepancies may occur during the conversion process.

Solutions to this problem include :

- Training the model directly in javascript :

  Most things you can do in python’s tensorflow, you can also do in tensorflowjs.

- Handling data preprocessing inside the model :

  Keras has native data preprocessing layers which can be included directly inside the model. The downside to this approach is that it decreases training performances, because preprocessing must happen as the model trains, rather than concurrently in a pipeline.

5. Conclusion

Scikit-learn is a very powerful library that provides a wide range of machine learning algorithms for both supervised and unsupervised learning, including tools for model selection and evaluation, data transformation, and data loading. It's an excellent choice for many machine learning tasks with structured data.

However, when it comes to image classification, deep learning models, specifically Convolutional Neural Networks (CNNs), have shown superior performance in most cases. This is because CNNs are designed to automatically and adaptively learn spatial hierarchies of features, which is ideal for image data. Libraries like TensorFlow and PyTorch, along with their high-level APIs like Keras, provide the necessary tools to design and train these kinds of models.

That said, scikit-learn can still be used for image classification tasks, particularly for smaller datasets or as a baseline model. Techniques like PCA for dimensionality reduction, SVMs for classification, and grid search for hyperparameter tuning can all be used effectively with image data in scikit-learn. However, for large-scale image classification tasks, deep learning models are generally a better choice.

In conclusion, while scikit-learn is an excellent library for many machine learning tasks, for image classification, you might achieve better results with a deep learning library like TensorFlow or PyTorch.

Sources :
- Wikipedia
- Scikit Learn
- tensor flow
- packt