Zoirdberg 2.0

Classification results

Introduction

Machine learning is the process of building algorithms that are able to learn from previous datasets, and leverage that experience to predict new unseen datasets.

How to test our notebooks

To test our notebooks, you can connect to our VPS with the following command:
ssh -L 8889:localhost:8889 epitech@54.38.185.154
With the password: "onabeaucoupbosserdoncgradeAsvp

Then in your browser, you can go to localhost:8889
to access a Jupyter-lab where you can find our notebooks.
The password of the jupyter-lab is the following
04ced18884c3c729dcfc3034fd0898b7a99cf9d3b27d7157

You can also connect with classic ssh and go to the /home/share/jupyter/ folder to find all our documents/IA in various formats/notebooks/etc

Material and Methods

Materials

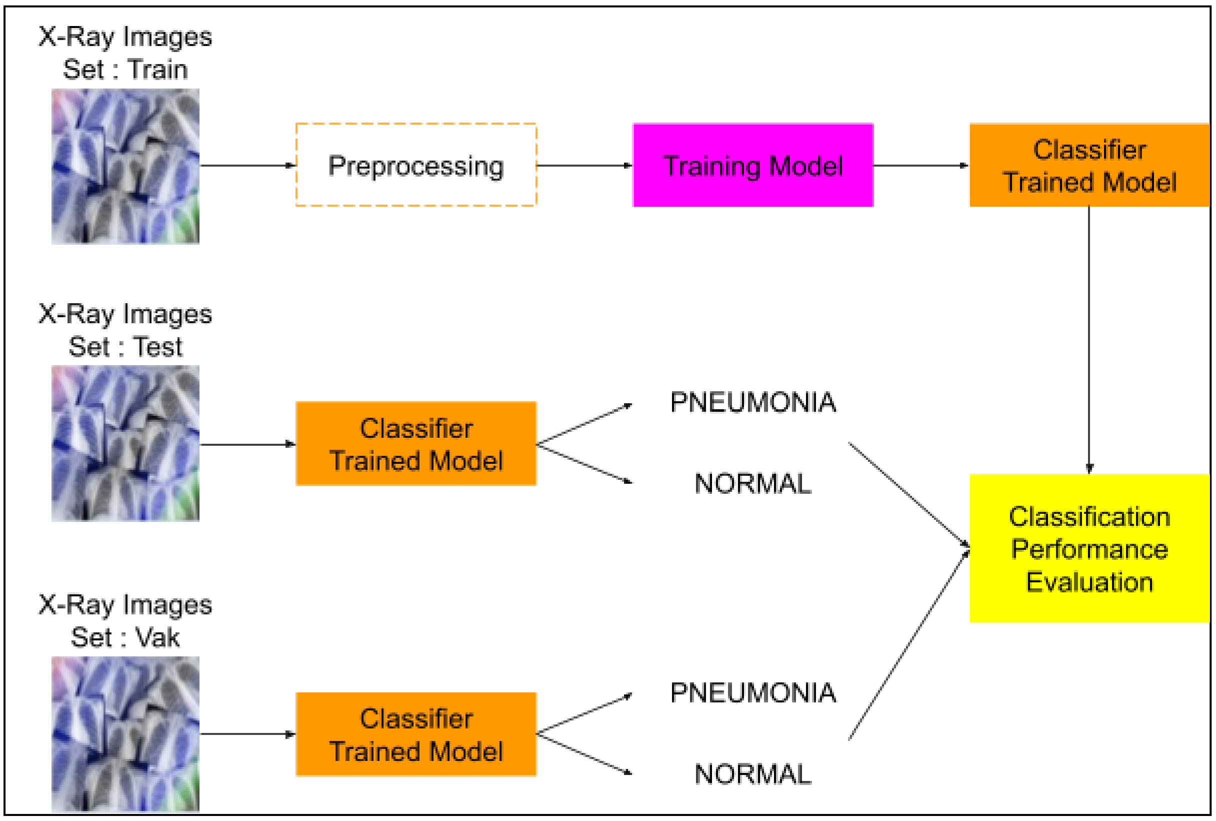
A total of 5856 chest X-ray images randomly selected and organized in three datasets :

- A train dataset containing 3875 X-ray images of confirmed PNEUMONIA patients and 1341 normal X-ray images (89,07%)
- A test dataset containing 390 X-ray images of confirmed PNEUMONIA patients and 234 normal X-ray images (10.66%)
- A val dataset containing 8 X-ray images of confirmed PNEUMONIA patients and 8 normal X-ray images (0.27%)

We tried different classification models : Machine learning with supervised and unsupervised learning and deep learning.
The models were trained by the train dataset.
The models were validated using the val dataset.
The performance of the model was evaluated using an accuracy calculation.
Python programming language will be used for the implementation of the X-ray image classification system.
The experiment will be conducted using the train-validation-test procedure. One set of X-ray images namely : X-ray images of two categories : NORMAL and PNEUMONIA.
The train images will be sometimes preprocessed in function of the classification model used.

Methods

First, we preprocessed the train images if needed. After that, we trained our models to detect and classify X-ray images in the two categories for the multi-class classification task. Then, the model is now trained to detect and classify the two categories for the binary classification task.



CNN

Introduction to CNN

CNN or Convolutional Neural Network, is a class of artificial networks, part of the deep learning family.
This neural network architecture is called like that because it is an architecture based on the mathematical operation “Convolution”.
It is an operation working with the pixels of an image and is very often used for the recognition of an object or a pattern in an image.

Understand what used a CNN

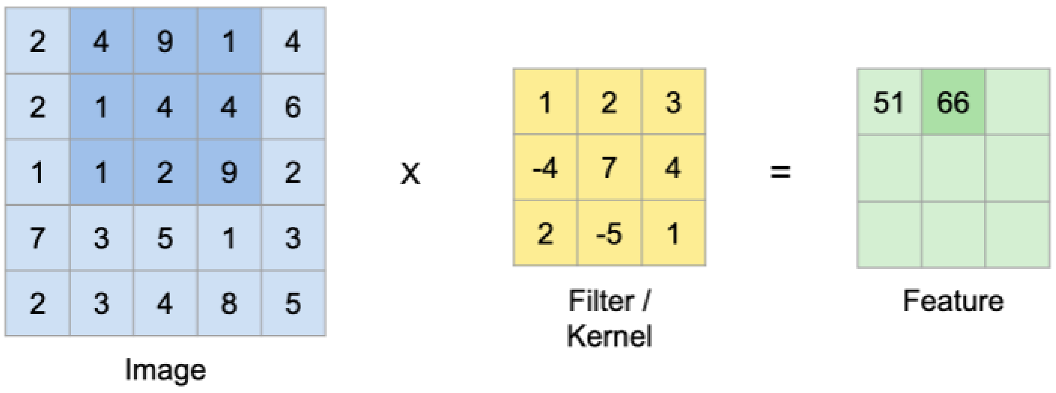
A CNN takes as “input” an image.
An image is a tensor composed of the following shape: width, height, channels

A convolutional neural network has an architecture generally very similar to the following.
A sequence of operations starting with a convolution, an activation function called Relu (most often) and the maxpool operation (we'll look at what this is later.)
Once this sequence has been repeated a few times you only need to perform some operations called Dense in order to perform matrix operations to change the dimension of the vectors using each neuron.

Obviously, other architectures exist, but this is the best known and most widely used by scientists/developers around the world.

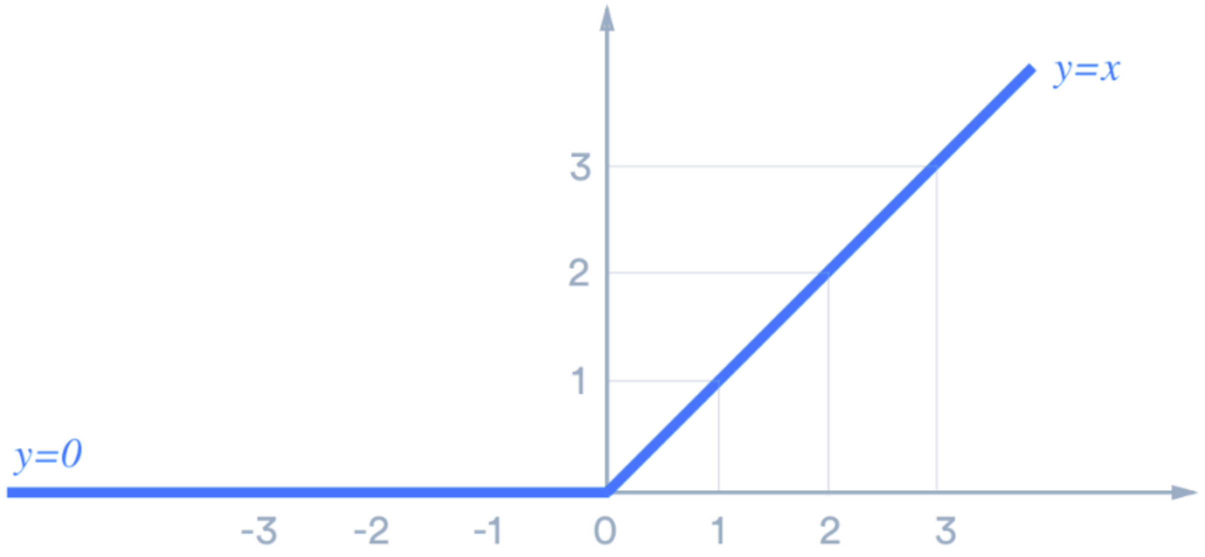
Convolution layer

This layer tries to learn the feature representation of the inputs, whether they are pictures of cats versus dogs or numbers. To compute the different feature maps, it consists of several kernels/matrices. This, a filter/matrix kernel (n*n) depends on the type of problem we are solving, and then it is applied to the input data (or image) to obtain the convolution feature. This convolution feature is then passed on to the next layer after adding a bias and applying any appropriate activation function.



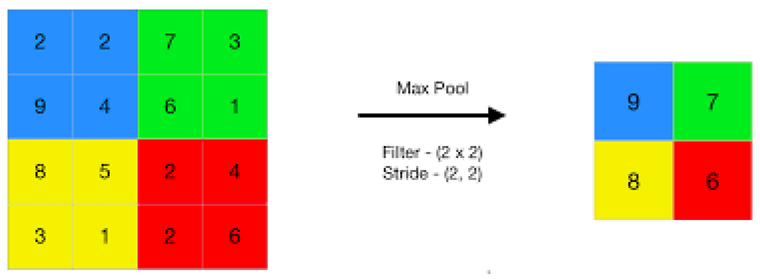
Activation function Relu

Any selected activation function has a considerable impact on the performance of a convolutional neural network for a given problem. For CNN, Re Lu is the preferred activation function due to its simple differentiability and speed compared to other activation functions such as tanh and sigmoid. Re Lu is usually followed after the convolution operation. Other names in a list of activation functions include Sigmoid, softmax, Leaky Re Lu, ELU, etc.



Maxpool layer

The maxpool layer is placed between the convolutional layers. It is used to achieve shift invariance which is achieved by decreasing the resolution of the feature maps. Widely used pooling operations are average pooling and maximum pooling. Basically, reducing the number of connections between convolutional layers reduces the computational load on the processing units.



Applications of a CNN

Convolution is widely used in three areas in particular.

- Image classification
- Object detection in an image or video
- In object tracking

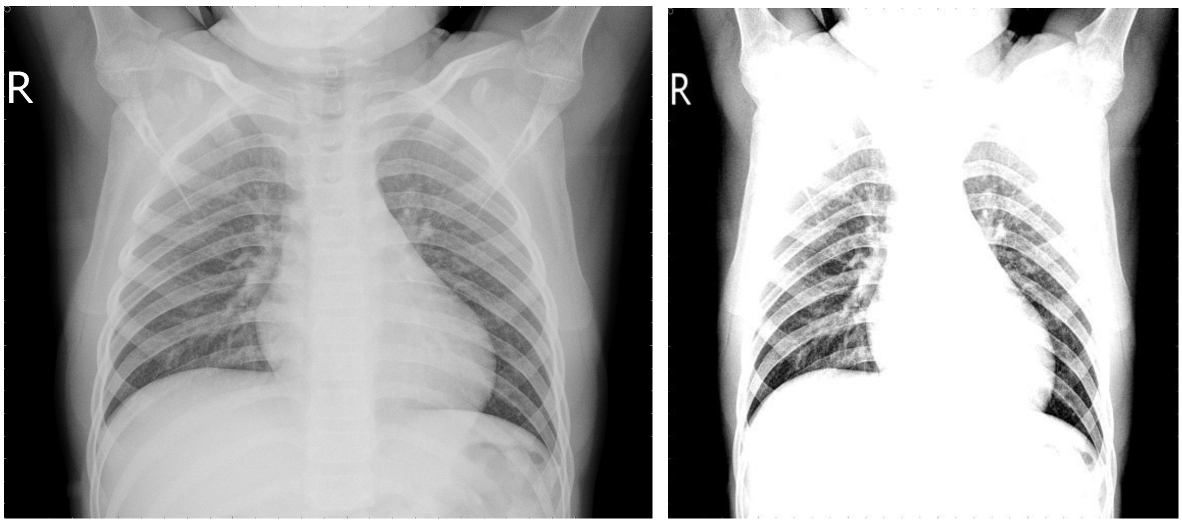
Method

Preprocessing

What is a preprocess ?

Preprocessing means to modify the data in order to improve the efficiency of the AI. It can be used to change pictures by modifying its sharpness, contrast, luminosity or colorimetry. Also, in our .onnx models, pictures needs to be on a precise format : black and white colors and 500*500 pixels.

For example, this is an original picture vs the same pictures, with :
- Luminosity 1.3 times superior
- Contrast twice superior
- Sharpness 5 times superior
- Resized in 500*500 pixels
- Black and White



Why preprocess data ?

The first step in data preprocessing is to understand the data. Actually, just looking at the dataset can give an intuition of what things need to be focused on.
The principal idea is to orient the classifier to look at the differences between classes instead of looking at the input globally.
As a human learnt how to tell the difference between a dog and a cat by looking at these animals multiple times, a classifier AI needs a full dataset. But it is possible to orient this learning by highlighting the difference between these animals (for example, highlighting the ears or the nose).

Moreover, the classifier model can be waiting for a particular data format (as told before, format such as colorimetry and/or picture’s shape), and the preprocessing must match the model’s needs

How to preprocess data ?

In our case, as the decision was to visualize using Jupyter Notebook (web interface executing Python code cell by cell), it has been decided to use a simple python script to preprocess the data.
This script needs the preprocessing data (sharpness, orientation, luminosity, contrast and final size), the path to the directory containing the dataset, and the path to the directory to place the preprocessed data.
It's a "plug and play" script that can be used both as a main program (calling like "python3 preprocess.py [...]) or as a function, passing the argument that the function needs.

To know which augmentation we needed to make on our dataset, we had three differents parts :

- Making research on the Internet
- Asking a data-scientist we are working with
- By trial and error, modifying a little bit each parameter and run a benchmark on the model trained on the new dataset

Conclusion about preprocessing for of AI classifier

After running the benchmarks and finding that the model having the best result was trained on a dataset without any preprocessing (except resizing the pictures and removing color to have gray-scale pictures), the first conclusion was that the preprocessing was not necessary on this dataset.
To warrant our idea, we talked with a data-scientist in the R&D section at Numalis company who had already worked on this kind of AI with the same dataset for a use-case. He confirmed that the best preprocessing he did was not doing any preprocessing on sharpness, contrast nor luminosity.

Tools

Tensorflow

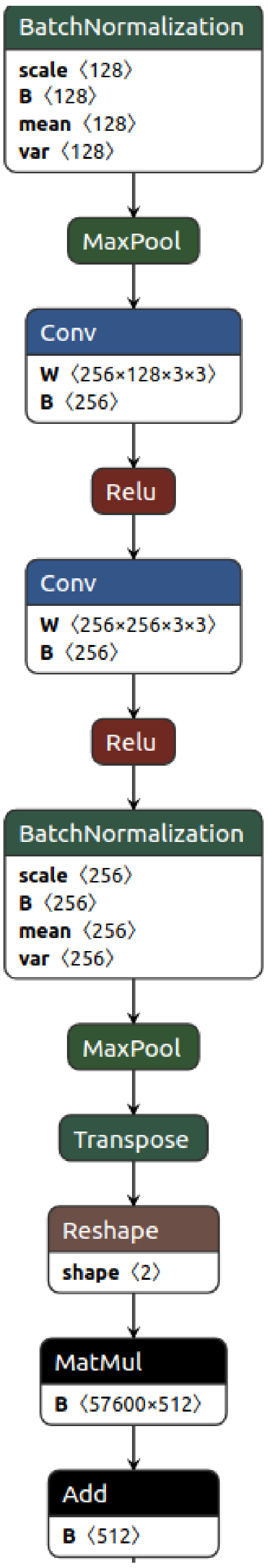
For this we used different tools.
First of all, we used a Jupyter-notebook.
This is a web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and organize workflows in data science, scientific computing, computer journalism and machine learning.

We used the Tensorflow library to create our neural network, to train it etc.
It is a library dedicated to machine learning. It offers a complete and flexible ecosystem of tools, libraries and community resources that allow researchers to advance in the field of machine learning, and developers to easily create and deploy applications that exploit this technology.

Onnx

Onnx is a file format, created by Facebook, Microsoft and many other companies.
This file format consists of a layer graph, making it easy to use a registered ia.

Here is an image of one of our AIs opened with Netron software.



Jupyter-notebook

JupyterLab is a web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and arrange workflows in data science, scientific computing, computational journalism, and machine learning. A modular design invites extensions to expand and enrich functionality.
This tool allows us to create our ia, our different graphs etc.

Training

Intro

Now that we have determined the architecture of our neural network and we have our dataset with and without pre-processing, it is time to train our neural network.

To train our ia, we used Tensorflow.

With tensorflow we could create a Sequential to which we gave a list of operations to execute. It is these operations that will be called by the training of our neural networks. In reality, we are not training an AI. We are performing a series of operations in a loop to find out what weight to give to each neuron. The computation of neural networks looks like this: Vector of weights * Vector of input (here the images) + bias.

Architectures

We use Three différents list of opérations :

1) cnn = Sequential()

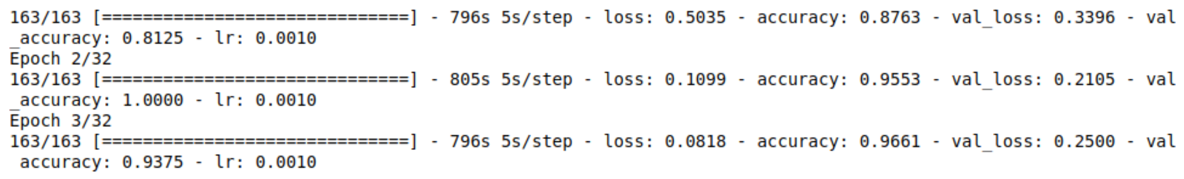
```
cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(img_width,
img_height, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(img_width,
img_height, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(img_width,
img_height, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(img_width,
img_height, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(img_width,
img_height, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Flatten())
cnn.add(Dense(activation = 'relu', units = 128))
cnn.add(Dense(activation = 'relu', units = 64))
cnn.add(Dense(activation = 'sigmoid', units = 1))
cnn.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
```

```
        cnn.summary()
2)  cnn = Sequential([

        Conv2D(16, kernel_size=(3, 3), activation='relu', padding='same', input_shape =
    (img_width, img_height, 1)),
        Conv2D(16, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.25),

        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.25),

        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.25),

        Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same' ),
        Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.25),

        Conv2D(256, kernel_size=(3, 3), activation='relu', padding='same' ),
        Conv2D(256, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.25),


        Flatten(),

        Dense(512, activation='relu'),
        Dropout(0.5),

        Dense(256, activation='relu'),
        Dropout(0.5),

        Dense(64, activation='relu'),
        Dropout(0.5),
        Dense(1, activation = "softmax")

     ])
    cnn.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =
    ['accuracy'])
    cnn.summary()
3)  cnn = Sequential([

        Conv2D(16, kernel_size=(3, 3), activation='relu', padding='same', input_shape =
    (img_width, img_height, 1)),
        Conv2D(16, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.2),

        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.2),

        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.2),

        Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
        Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.2),

        Conv2D(256, kernel_size=(3, 3), activation='relu', padding='same'),
        Conv2D(256, kernel_size=(3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2)),
        Dropout(0.2),

        Flatten(),

        Dense(1024, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),

        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(0.4),

        Dense(256, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),

        Dense(64, activation='relu'),
        BatchNormalization(),
        Dropout(0.2),

        Dense(1, activation = "softmax")
    ])
cnn.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =
['accuracy'])
cnn.summary()
```

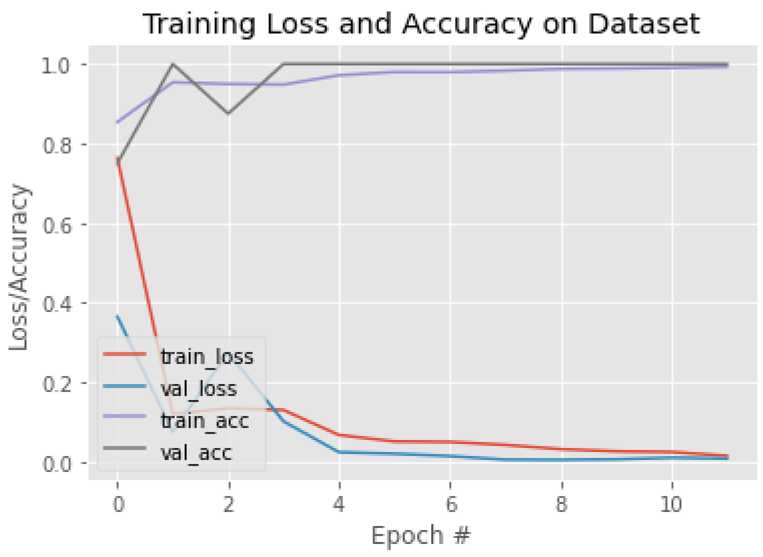
These three lists represent our neural network architectures. These are architectures that are classic in the CNN world.

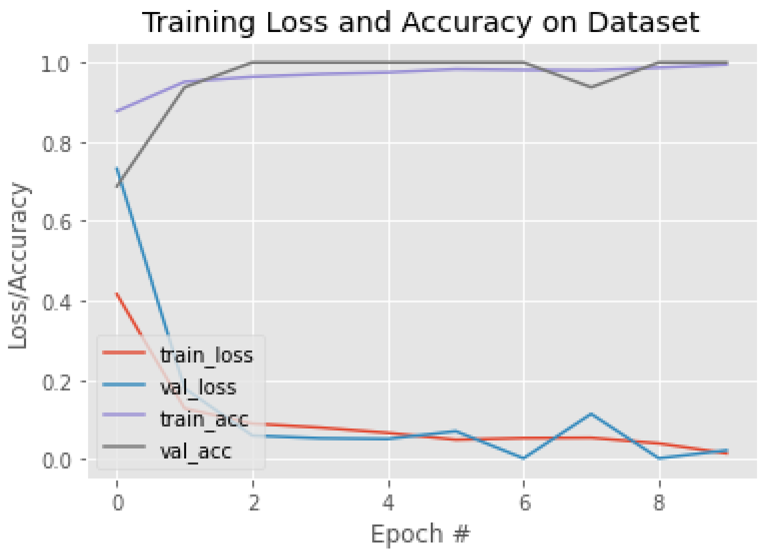
Training

To train our ia, we used the Tensorflow fit function. This is what the results look like during the many hours of waiting during training



```
163/163 [==============================] - 796s 5s/step - loss: 0.5035 - accuracy: 0.8763 - val_loss: 0.3396 - val
_accuracy: 0.8125 - lr: 0.0010
Epoch 2/32
163/163 [==============================] - 805s 5s/step - loss: 0.1099 - accuracy: 0.9553 - val_loss: 0.2105 - val
_accuracy: 1.0000 - lr: 0.0010
Epoch 3/32
163/163 [==============================] - 796s 5s/step - loss: 0.0818 - accuracy: 0.9661 - val_loss: 0.2500 - val
_accuracy: 0.9375 - lr: 0.0010
```





These two images show the training history of two models we generated, using the same datasets but not the same Tensorflow settings.

Dataset

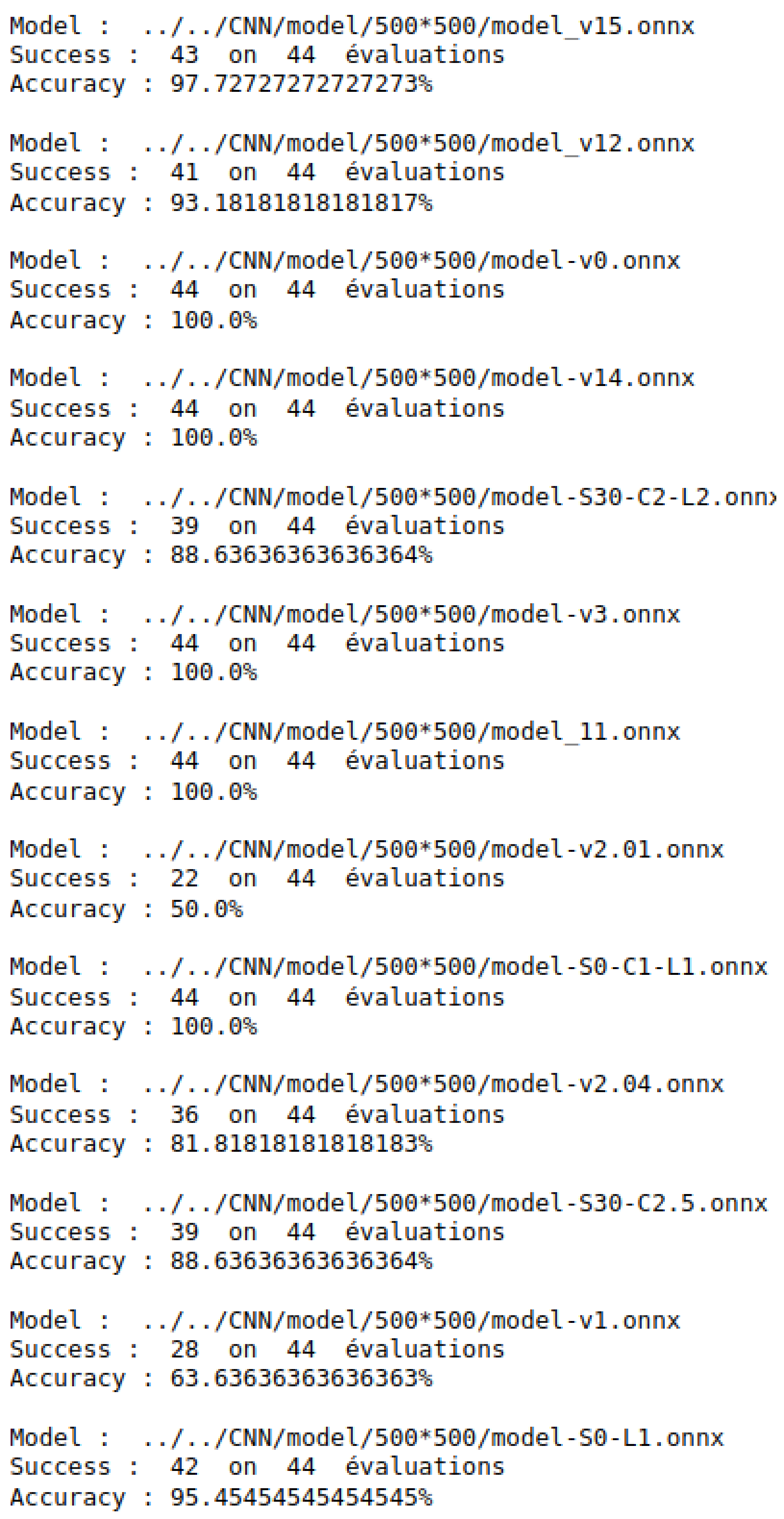
Following the pre-processing work we have done, we have generated different Datasets from the Dataset given by Epitech.
Some by increasing/decreasing the brightness, the contrast or the sharpness.
However, we also resized the images in 500*500 allowing us to have shapes common to all images.
We also made the choice to modify the test folder of the dataset. Indeed, at the time of our various tests we had very high results but that test only 44 images (22 Normal and 22 with Pneumonia).
We decided to move 580 images from the train folder to the test folder. In order to enlarge the test folder to 624 images and to be able to evaluate our AI more precisely.

Results

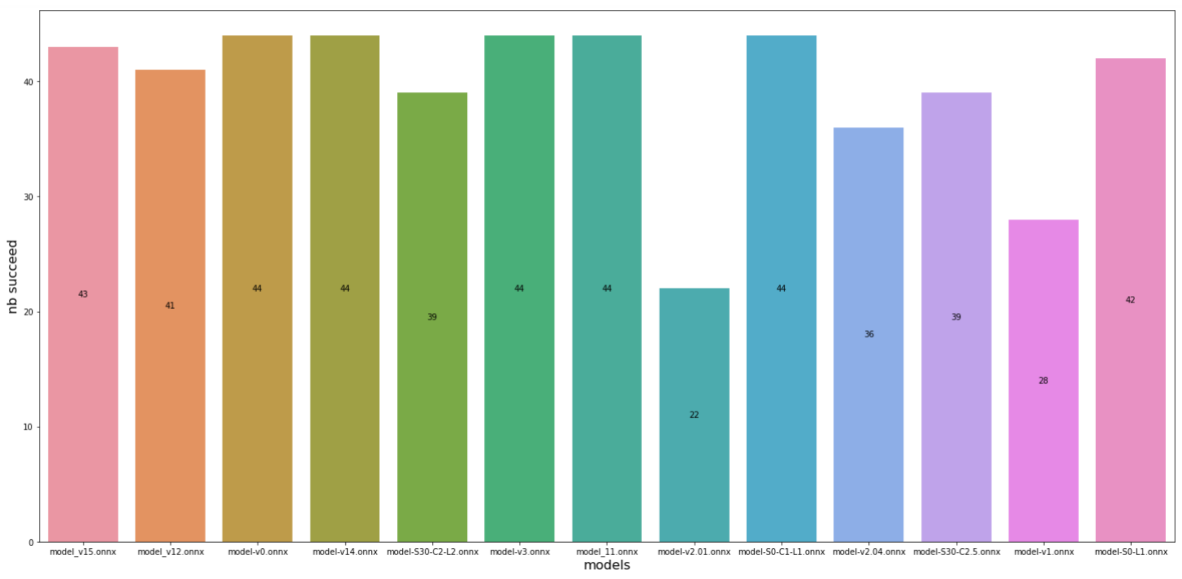
Now let's talk about our results.
We generated 16 different AIs, 13 of which were input at 500*500 and 3 at 600*600.
These AIs were all trained with at least one of these factors different from the others.
List of factors :
Computer used for training: OVH VPS server, laptop and desktop PC with a lot of power
Batch
Epoch
With or without EarlyStopping
Different learning_rate_reduction options
Different datasets for the training
Different neural network architectures

Understanding the names of the models :
Model-v0 to model-v5: were trained on a laptop with the dataset without pre-processing
Model-v6 to model-v11: were trained on a vps
Model post v11: were trained on a fixed pc
Model-SXX-LXX-CXX: were trained with a pre-processed dataset
Model-V2.XX: are models with a different architecture.

Now let's take a look at our results.

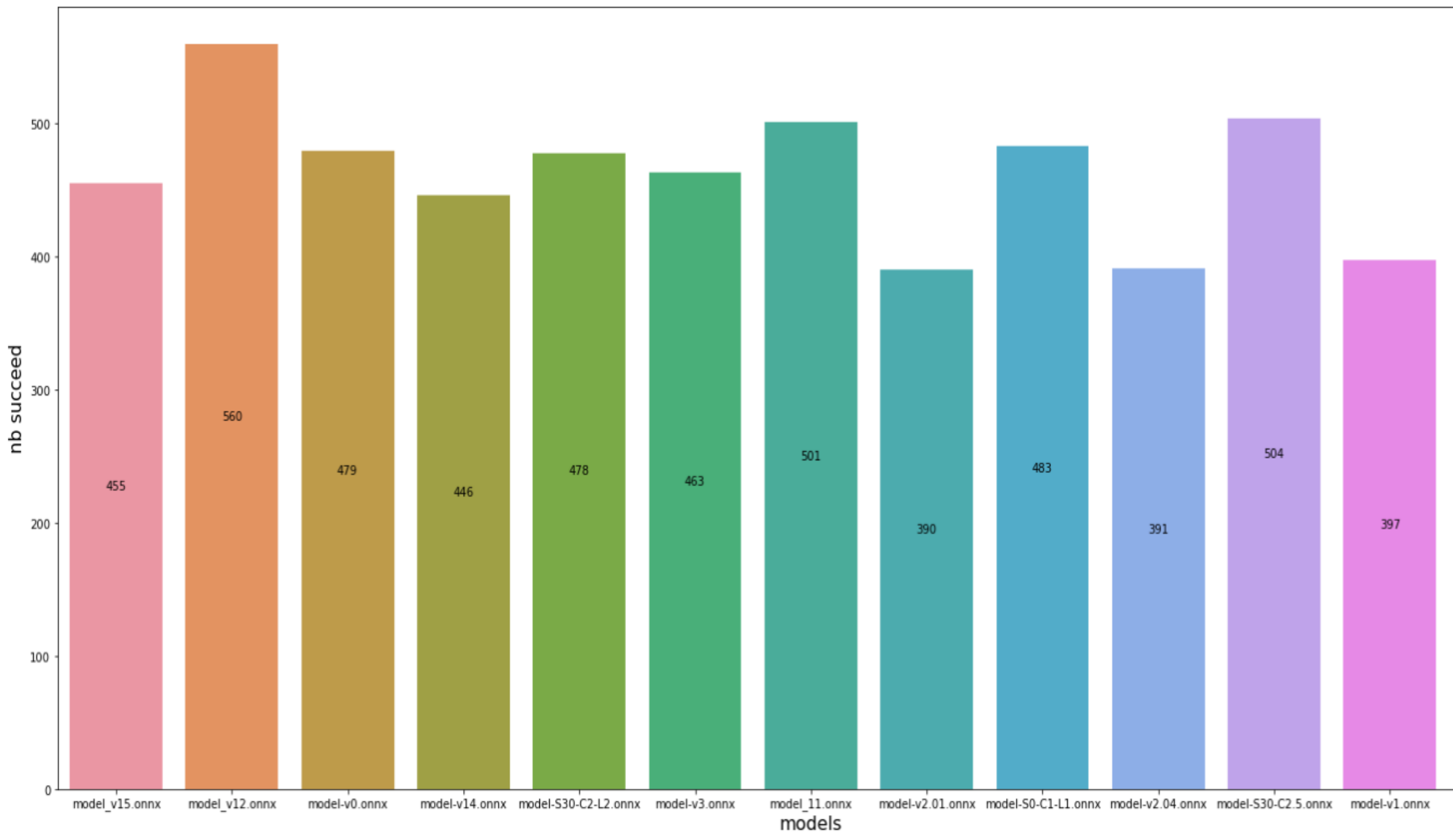


```
Model :  ../../CNN/model/500*500/model_v15.onnx
Success :  43  on  44  évaluations
Accuracy : 97.72727272727273%

Model :  ../../CNN/model/500*500/model_v12.onnx
Success :  41  on  44  évaluations
Accuracy : 93.18181818181817%

Model :  ../../CNN/model/500*500/model-v0.onnx
Success :  44  on  44  évaluations
Accuracy : 100.0%

Model :  ../../CNN/model/500*500/model-v14.onnx
Success :  44  on  44  évaluations
Accuracy : 100.0%

Model :  ../../CNN/model/500*500/model-S30-C2-L2.onn
Success :  39  on  44  évaluations
Accuracy : 88.63636363636364%

Model :  ../../CNN/model/500*500/model-v3.onnx
Success :  44  on  44  évaluations
Accuracy : 100.0%

Model :  ../../CNN/model/500*500/model_11.onnx
Success :  44  on  44  évaluations
Accuracy : 100.0%

Model :  ../../CNN/model/500*500/model-v2.01.onnx
Success :  22  on  44  évaluations
Accuracy : 50.0%

Model :  ../../CNN/model/500*500/model-S0-C1-L1.onnx
Success :  44  on  44  évaluations
Accuracy : 100.0%

Model :  ../../CNN/model/500*500/model-v2.04.onnx
Success :  36  on  44  évaluations
Accuracy : 81.81818181818183%

Model :  ../../CNN/model/500*500/model-S30-C2.5.onnx
Success :  39  on  44  évaluations
Accuracy : 88.63636363636364%

Model :  ../../CNN/model/500*500/model-v1.onnx
Success :  28  on  44  évaluations
Accuracy : 63.63636363636363%

Model :  ../../CNN/model/500*500/model-S0-L1.onnx
Success :  42  on  44  évaluations
Accuracy : 95.45454545454545%
```



In the first two images, we use the test file BEFORE we enlarge the number of images in it. As you can see, the accuracy is very high. This could be good but when we use the same Benchmark notebook with a higher dataset.

```
Model :  ../../CNN/model/500*500/model_v15.onnx
Success :  455  on  624  évaluations
Accuracy : 72.91666666666666%

Model :  ../../CNN/model/500*500/model_v12.onnx
Success :  560  on  624  évaluations
Accuracy : 89.74358974358975%

Model :  ../../CNN/model/500*500/model-v0.onnx
Success :  479  on  624  évaluations
Accuracy : 76.76282051282051%

Model :  ../../CNN/model/500*500/model-v14.onnx
Success :  446  on  624  évaluations
Accuracy : 71.47435897435898%

Model :  ../../CNN/model/500*500/model-S30-C2-L2.onnx
Success :  478  on  624  évaluations
Accuracy : 76.6025641025641%

Model :  ../../CNN/model/500*500/model-v3.onnx
Success :  463  on  624  évaluations
Accuracy : 74.19871794871796%

Model :  ../../CNN/model/500*500/model_11.onnx
Success :  501  on  624  évaluations
Accuracy : 80.28846153846155%

Model :  ../../CNN/model/500*500/model-v2.01.onnx
Success :  390  on  624  évaluations
Accuracy : 62.5%

Model :  ../../CNN/model/500*500/model-S0-C1-L1.onnx
Success :  483  on  624  évaluations
Accuracy : 77.40384615384616%

Model :  ../../CNN/model/500*500/model-v2.04.onnx
Success :  391  on  624  évaluations
Accuracy : 62.66025641025641%

Model :  ../../CNN/model/500*500/model-S30-C2.5.onnx
Success :  504  on  624  évaluations
Accuracy : 80.76923076923077%

Model :  ../../CNN/model/500*500/model-v1.onnx
Success :  397  on  624  évaluations
Accuracy : 63.62179487179487%

Model :  ../../CNN/model/500*500/model-S0-L1.onnx
Success :  508  on  624  évaluations
Accuracy : 81.41025641025641%
```



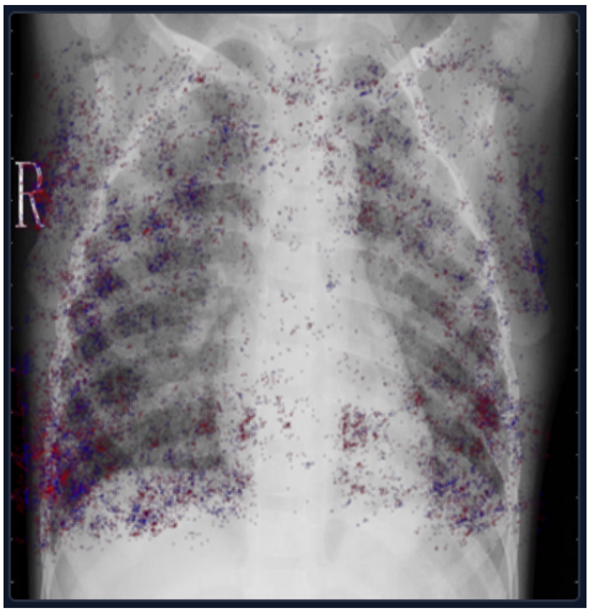
We can notice that the accuracy is falling.

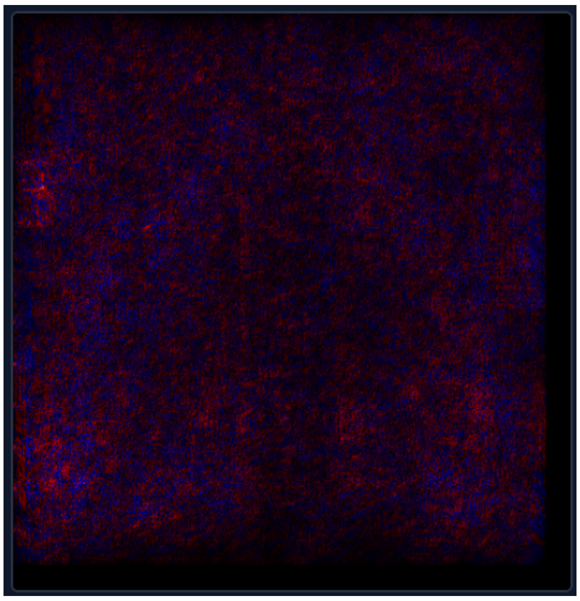
Conclusion in our pneumonia detection AI.

After dozens of hours of training our models, we learned a lot. We tested dozens of different configurations to find the most robust ia.
Many solutions were considered and then rejected because of the biases that this entailed in the neural network.

In view of the previous results, we can deduce that the best performing ia we have is model_v12 which was generated with the fixed pc/ 100 epoch/ non-pre-processor dataset/ 64 batchs

We used a tool that is currently developed by the company Numalis to analyse where the model will look in the image by its execution. You can see the results below





🟥 Pixels that was helpful to help AI saying that this image is in the class it choose

🟦 Pixels that was helpful to help AI saying that this image is not in the other class

⬛ Pixels that was not helping AI to make a decision

(The more the red or the blue is intense, the more the pixel had been useful)

Supervised Models

SVM

The support vector machine (SVM) classifier algorithm is one of the most widely used supervised machine learning algorithms. This study will use a machine learning model trained by an SVM classifier optimized by Sequential Minimal Optimization (SMO) algorithm.

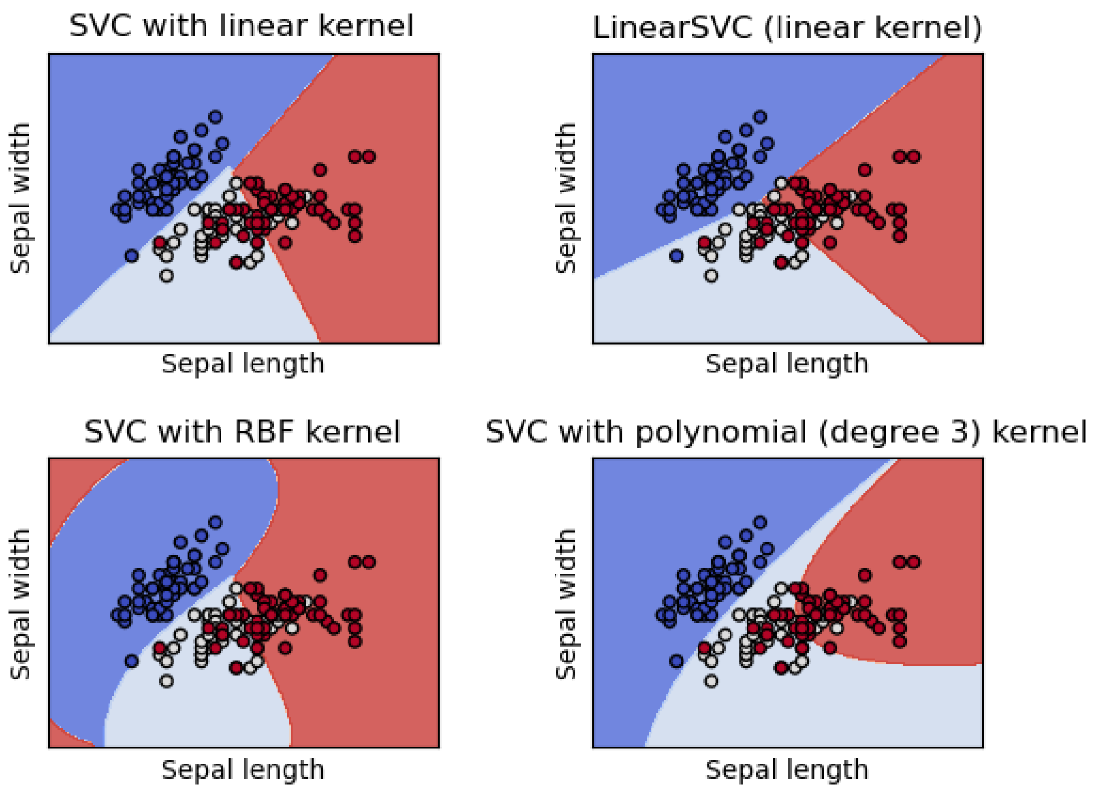


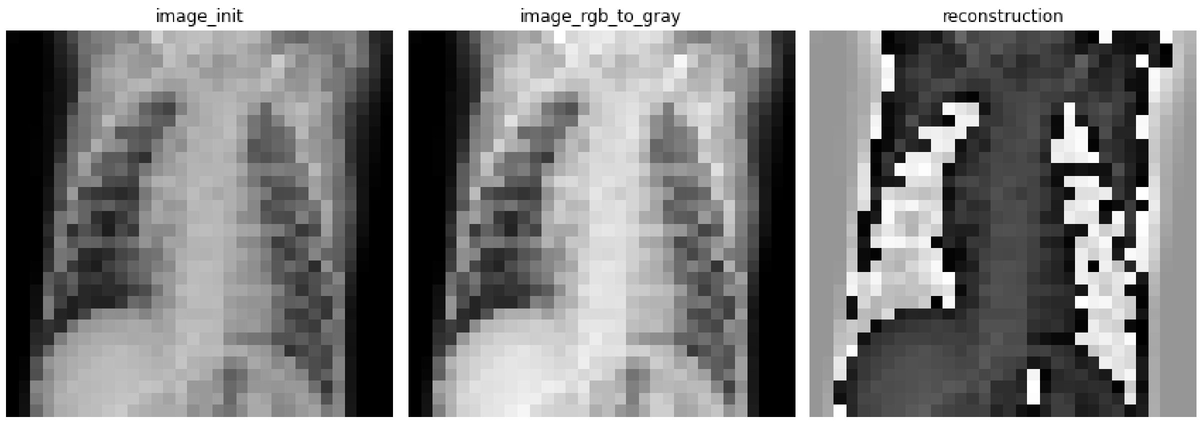
Image preprocessing

Only the train images will be preprocessed.

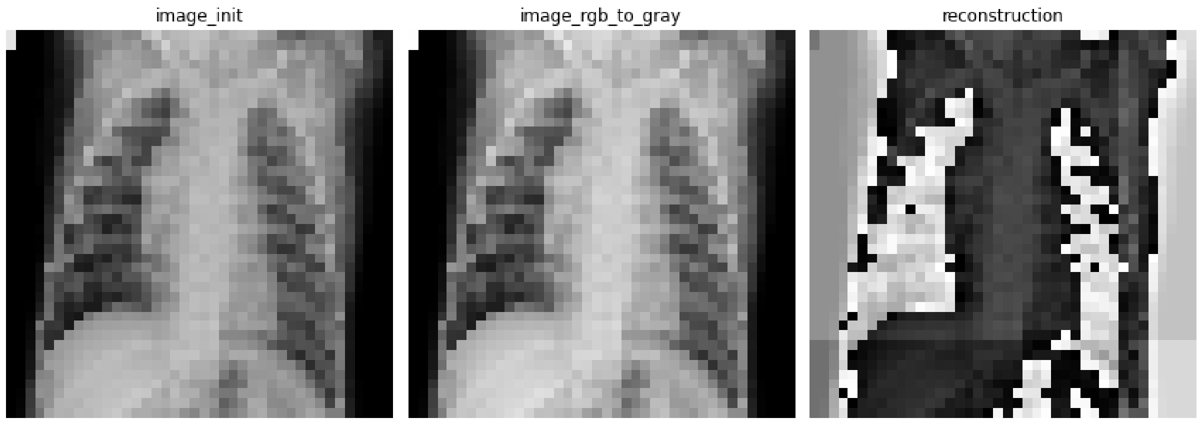
Wavelet Haar

The wavelet transform is a mathematical technique which can decompose a signal into multiple lower resolution levels by controlling the scaling and shifting factors of a single wavelet function

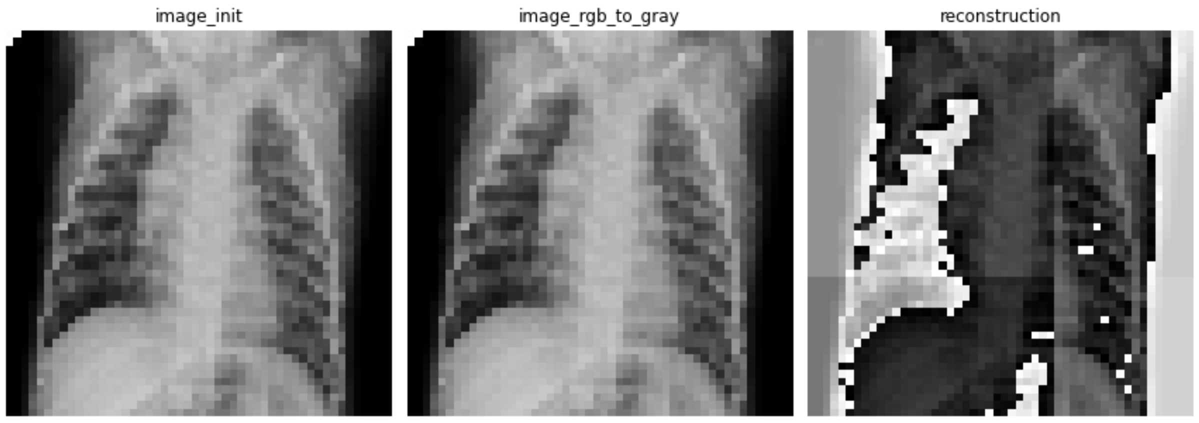
32x32



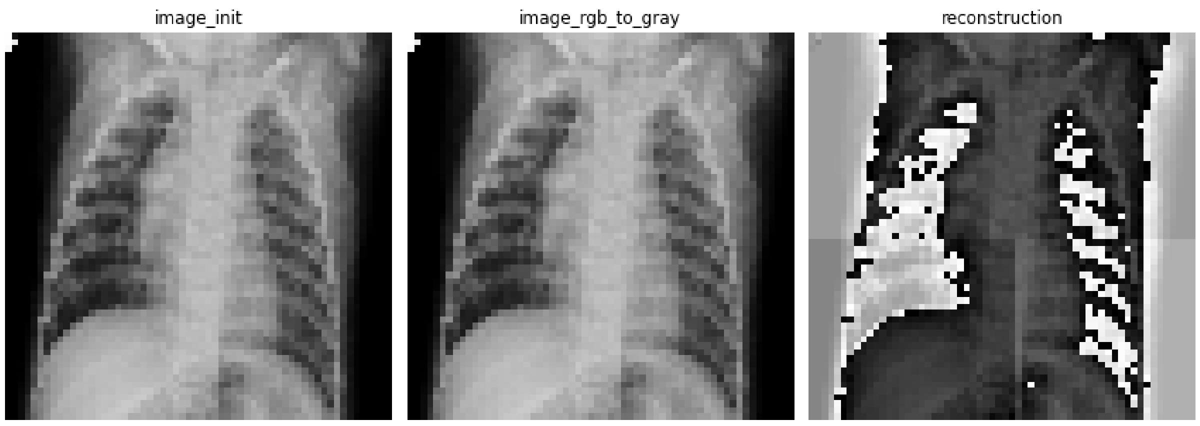
40x40



50x50



60x60



Results

#	preprocessing	size	C	kernal	test accuracy	val accuracy
1.1	wavelet haar only the train images	32 x 32	5	linear	0.8012	0.9375
1.2	wavelet haar only the train images	32 x 32	5	rbf	0.7275	0.625
1.3	wavelet haar only the train images	32 x 32	10	linear	0.8012	0.9375
1.4	wavelet haar only the train images	32 x 32	10	poly		
2	wavelet haar only the train images	40 x 40	10	linear	0.8317	0.8125
3	wavelet haar only the train images	50 x 50	10	linear	0.8221	0.8125
4	wavelet haar only the train images	60 x 60	10	linear	0.8429	0.875
1	wavelet haar all of the images	32 x 32	10	rbf	0.7756	0.75
2	wavelet haar all of the images	40 x 40	10	rbf	0.7756	0.75
3	wavelet haar all of the images	50 x 50	10	rbf	0.7660	0.8125
4	wavelet haar all of the images	60 x 60	10	rbf	0.7708	0.8125

Linear kernel shows better results than rbf.

Unsupervised Models

In this section, we'll be reviewing how unsupervised models and techniques can fit into this xray medical field.

When compared to supervised models, unsupervised ways to process data are quite interesting in this point: no label needed in order for your model to work. Labeling datasets and finding good datasets are real problems in supervised methods: it requires professionals to spend hours creating thousands of labeled images.

Unsupervised methods can fix this problem, but for medical diagnosis support, you might want to consider those things:

- As the data is not labeled, you will not be able to get a complete diagnosis from your unsupervised method as it can't just say "Sick or Normal".
- This means that you will not be able to create an AI capable of acting on its own, but more like a good seeker that will be able to point out tiny changes between two images.

It might be interesting as you only need a few normal images (5-10 of them are clearly enough): all you have to do is to take the unknown image and give it to the AI. It will highlight anomalies and assist the doctor in finding problems.

Samuel notes: My personal lack of knowledge and time (first time in an AI project) pushed me to a point where I was not able to implement any sort of working solution using an unsupervised method. So what I decided is to analyze and report a project that could have been a real finalized functional solution of anomaly detection AI. You will find the repo right here →

GitHub - Valentyn1997/xray: Unsupervised Anomaly Detection for X-Ray Images

Anomaly detection

They conducted their search like this:

- Preprocessing
- Testing different anomaly detection techniques

As data was quite noisy, they had to take it through a preprocess step. The problem with such a thing while trying to build anomaly detection is that you have to make sure your preprocess deletes noise and keeps anomalies that can sometimes be very close to noise.

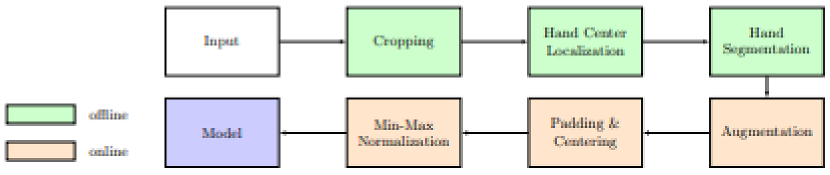


Fig. 2. The full image preprocessing pipeline. Steps highlighted in green are performed once and the result is stored to disk. Steps highlighted in orange are done on-the-fly.

Note: These were not working with CXR but Hand Xray

They used autoencoders to learn over normal data how to reconstruct normal hands. If it hasn't seen any abnormality, it will hence fail to reproduce them. They can use heatmap to set the point of interest on a render: there is something abnormal here.

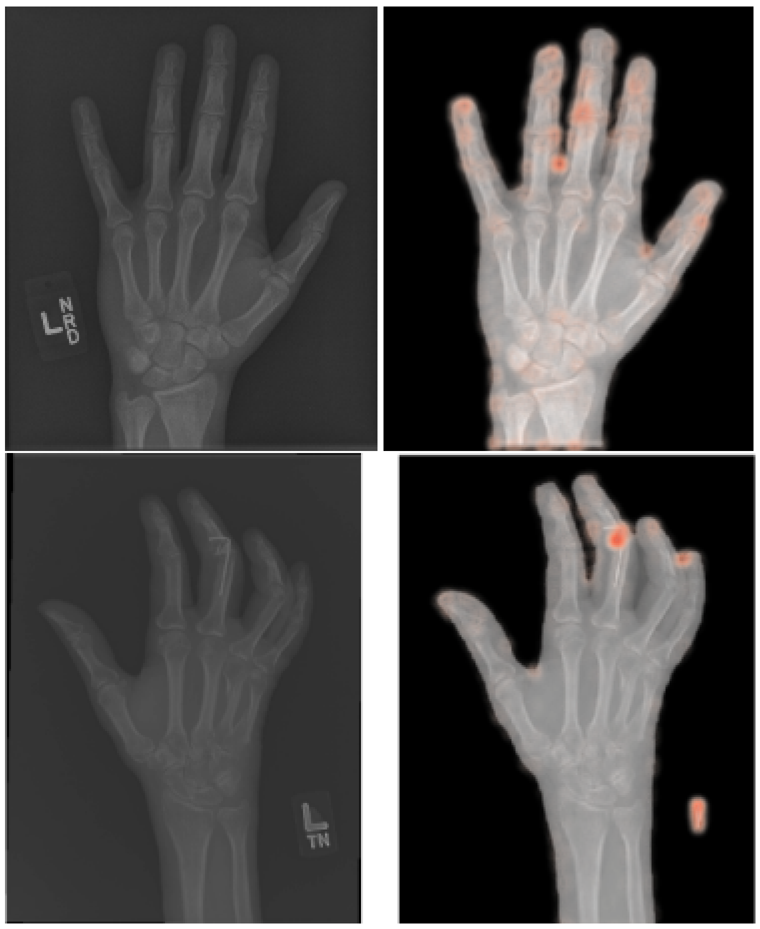


Fig. 5. Example heatmaps of reconstruction error of CAE. The left image-pair shows a hand from a study labeled as normal hand. Here we can see that the reconstruction error is relatively wide spread. The right image pair shows an abnormal hand, where the abnormality is clearly highlighted.

To handle the anomaly identification, they also tested different Generative Adversarial Networks and concluded that GAN and AE were the tools to use for anomaly detection.

K-Nearest neighbors

Google colab notebook available here.

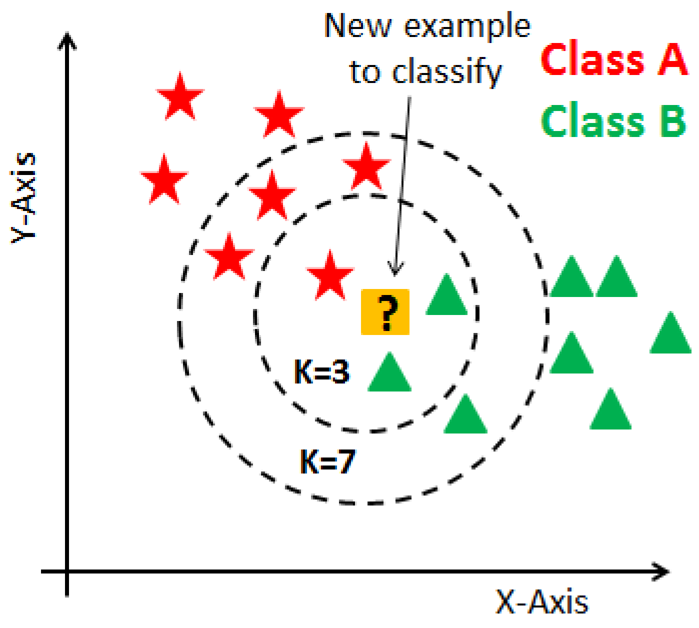
Algorithm

K-nearest neighbors is a supervised algorithm that computes the distance between data points in order to recognize similarities (works with regression as well).

The metrics we used for the computation of the distance is “Minkowski” and consist of the following formula:

```
sum(w*|x - y|^p)^(1/p)
```

The parameter “k” which gives his name to the algorithm represents the number of neighbor(s) that are taken into account and it’s at the core of the algorithm.



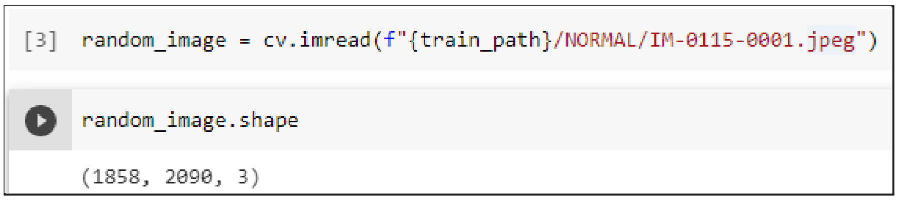
We can find an implementation of this algorithm in the library **sklearn** where the “k” parameter corresponds to the “nb_neighbors” keyword:

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, *, weights='uniform',
algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None,
n_jobs=None)
```

Formatting

In order to use our knn architecture, we first need to format our data.

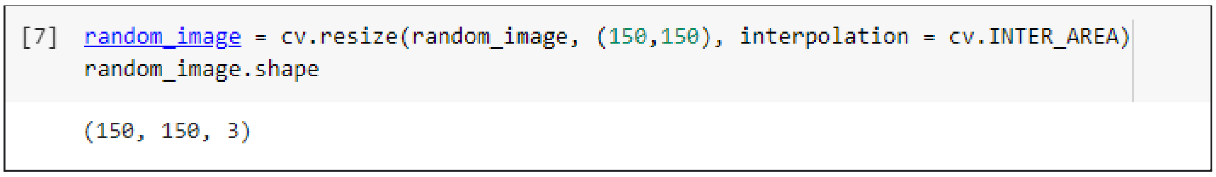
- We first open our data with the opencv library that give use a numpy array with that kind of shape:



```
[3]  random_image = cv.imread(f"{train_path}/NORMAL/IM-0115-0001.jpeg")

     random_image.shape

     (1858, 2090, 3)
```

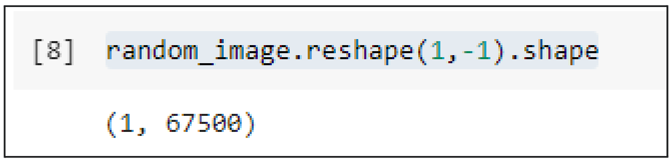
(width, height, rgb channels)

- We normalize our data by choosing resize dimensions as instance for dimension 150/150 we have that shape:



```
[7]  random_image = cv.resize(random_image, (150,150), interpolation = cv.INTER_AREA)
     random_image.shape

     (150, 150, 3)
```

- We reshape our data with reshape function:

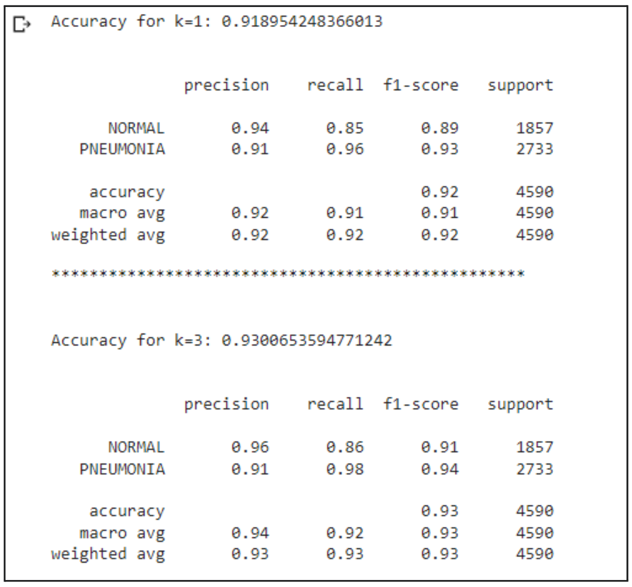


```
[8]  random_image.reshape(1,-1).shape

     (1, 67500)
```

- After formating all our images we concatenate all of the numpy arrays in a matrix **X** and we set the corresponding labels in a vector **y**.

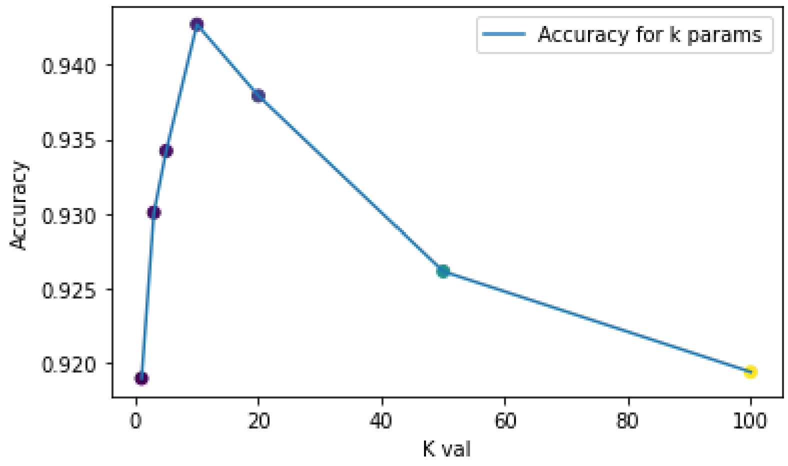
K Benchmark

In order to build an efficient model, we have to analyze which number of “k” neighbors best suits our data.
For that purpose we use the metrics module from sklearn which contain “accuracy” metrics and also the classification_report function which help us monitor the differents results:



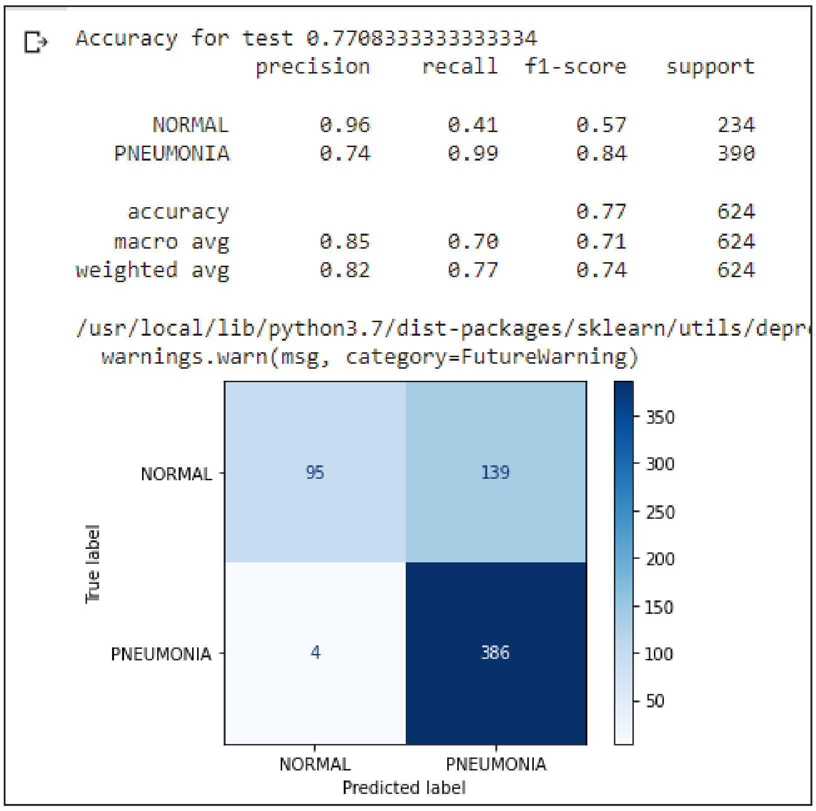
```
Accuracy for k=1: 0.918954248366013

              precision    recall  f1-score   support

      NORMAL       0.94      0.85      0.89      1857
   PNEUMONIA       0.91      0.96      0.93      2733

    accuracy                           0.92      4590
   macro avg       0.92      0.91      0.91      4590
weighted avg       0.92      0.92      0.92      4590

**************************************************


Accuracy for k=3: 0.9300653594771242

              precision    recall  f1-score   support

      NORMAL       0.96      0.86      0.91      1857
   PNEUMONIA       0.91      0.98      0.94      2733

    accuracy                           0.93      4590
   macro avg       0.94      0.92      0.93      4590
weighted avg       0.93      0.93      0.93      4590
```

We summarize this step by plotting our k value related to accuracy: we notice that it seems to be **k = 10** that is the best in our situation:



Results

Finally we build our model and analyze our model building a confusion matrix:



```
Accuracy for test 0.7708333333333334
              precision    recall  f1-score   support

      NORMAL       0.96      0.41      0.57       234
   PNEUMONIA       0.74      0.99      0.84       390

    accuracy                           0.77       624
   macro avg       0.85      0.70      0.71       624
weighted avg       0.82      0.77      0.74       624

/usr/local/lib/python3.7/dist-packages/sklearn/utils/depr
  warnings.warn(msg, category=FutureWarning)
```

This kind of plot is used to analyze the performance of our model: in our case we notice that:

- our model have a 0.77 accuracy
- Our model predict properly “pneumonia” data
- Our model have a huge number of false positive: 139 “normal’ images have been classified as “pneumonia”

Now in order to improve we could try different configurations on the tools we used to build our model, or we can also try to work on the preprocessing of our data.

Finally we save our model under the onnx format in order to benchmark it with models from other architectures.