DETECTION OF LUNG ABNORMALITIES FROM X-RAY IMAGES USING DEEP LEARNING TECHNIQUES

MASTER DATA SCIENCE

2020/21

Alberto Galego Redondo

INDEX

1. INTRODUCTION 3
a. Justification and objectives 3
2. ENVIRONMENT AND USED TECHNOLOGY 3
3. DATA 5
a. Image processing and display 6
4. LIFE-CYCLE FOR NEURAL NETWORK MODEL IN KERAS 8
a. Define Network 8
b. Compile Network 11
c. Fit Network 11
d. Evaluate Network 12
e. Make Predictions 12
5. RESULTS AND METRICS 12
a. Values of Loss and Sparse Categorical Accuracy 12
b. Confusion Matrix 13
c. Precision and Recall 14
d. AUC - ROC Curve 15
e. Precision-Recall (PR) Curve 17
6. CONCLUSIONS 18

1. INTRODUCTION

Lung disease is common throughout the world. These include chronic obstructive pulmonary disease, pneumonia, asthma, tuberculosis, fibrosis, etc.

In this work we are going to detect sick lungs through X-ray images applying Deep Learning techniques

I will use a database with images reduced to 64x64, which will be obtained from the website *https://data.mendeley.com/*

A deep convolutional network will be designed and trained, and an optimization study of results and execution times will be carried out.

a. Justification and objectives

The motivation for this thesis arises from a situation that is occurring in a large number of countries: the overload of work in many professions related to the field of health.

Today, techniques related to image analysis are being used to detect a large number of lung diseases, including X-rays. These images are carefully analyzed by radiologists and doctors, experts in interpreting the results obtained and make an appropriate diagnosis.

Due to the long training processes, exposure to intensive tasks for long periods of time and the high levels of stress arising from the responsibility that their interpretations imply, radiologists are a profession very prone to suffering sick leave

The final objective of this work is to develop an application to support medical diagnosis for lung disease.

2. ENVIRONMENT AND USED TECHNOLOGY

Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

All modern versions of Python are copyrighted under a GPL-compatible license certified by the Open Source Initiative.

Keras

Keras is one of the leading high-level neural networks APIs. It is written in Python and supports multiple back-end neural network computation engines.

Keras was created to be user friendly, modular, easy to extend, and to work with Python.

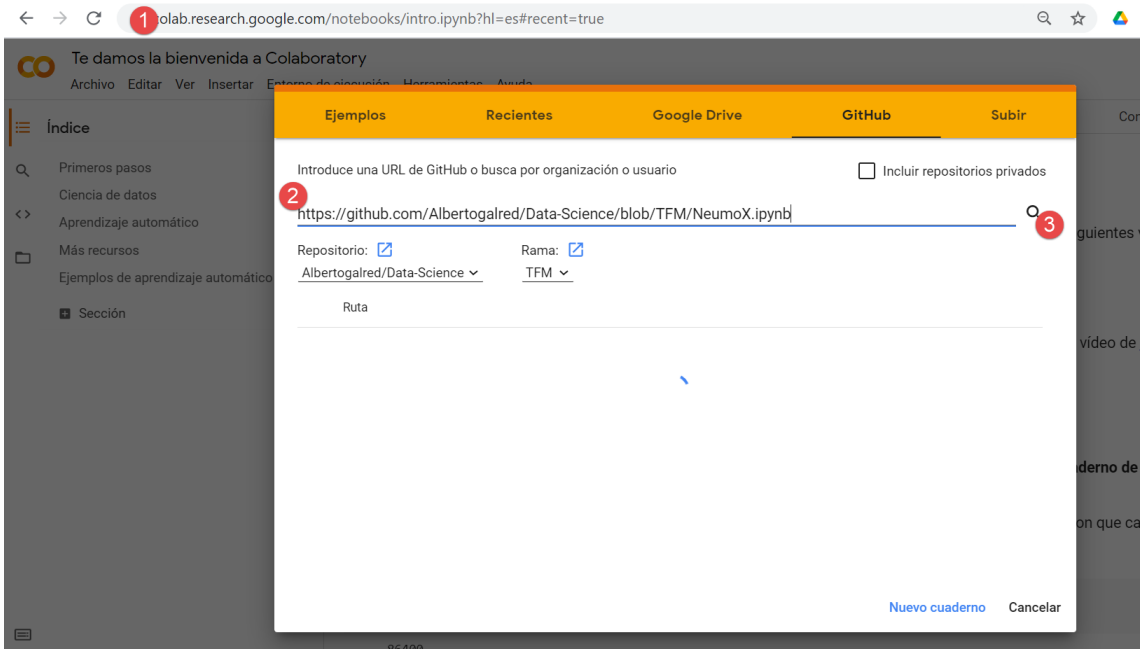
Neural layers, cost functions, optimizers, initialization schemes, activation functions, and regularization schemes are all standalone modules that you can combine to create new models. New modules are simple to add, as new classes and functions. Models are defined in Python code, not separate model configuration files.

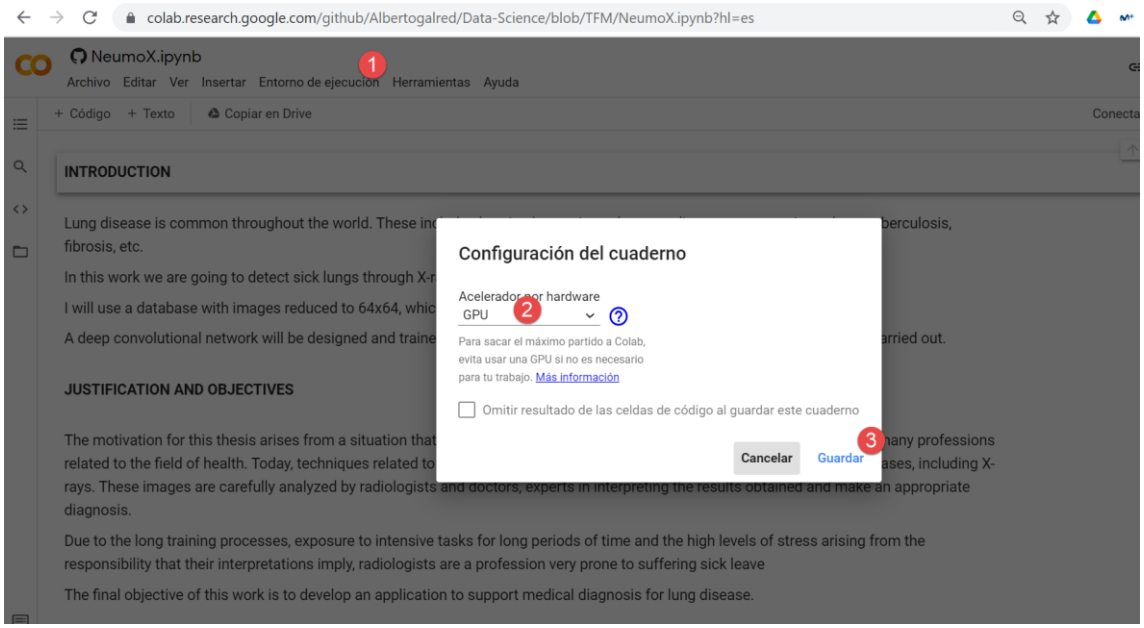
The biggest reasons to use Keras stem from its guiding principles, primarily the one about being user friendly. Beyond ease of learning and ease of model building, Keras offers the advantages of broad adoption, support for a wide range of production deployment options, integration with at least five back-end engines (TensorFlow, CNTK, Theano, MXNet, and PlaidML), and strong support for multiple GPUs and distributed training. Plus, Keras is backed by Google, Microsoft, Amazon, Apple, Nvidia, Uber, and others.

Google Colab

Google Colaboratory is a free online cloud-based Jupyter notebook environment that allows us to train our machine learning and deep learning models on GPUs. We can:

- Improve Python programming language coding skills.
- Develop deep learning applications using popular libraries such as Keras, TensorFlow, PyTorch, and OpenCV.





3. DATA

Images have been obtained from:

https://data.mendeley.com/datasets/rscbjbr9sj/2/files/f12eaf6d-6023-432f-acc9-80c9d7393433

PyDrive is a wrapper library of google-api-python-client that simplifies many common Google Drive API tasks. This API is used to get the identifier of the files in the MyDrive folder

```
from pydrive.drive import GoogleDrive
drive = GoogleDrive(gauth) # Create GoogleDrive instance with authenticated GoogleAuth instance
file_list = drive.ListFile({'q': "'root' in parents and trashed=false"}).GetList()
for file1 in file_list:
  print('title: %s, id: %s' % (file1['title'], file1['id']))
```

```
title: val.zip, id: 1c9VfEbc_WezjvwstMy6D1Zpl5xcmcYH4
title: .ipynb_checkpoints, id: 1-2mIbiWIas_CexviyUBV0SDUNQNRSRT8
title: train.zip, id: 1udaEEc4Zd9Rc-JJE7dyl5KWzpWl_F8Sp
title: NeumoX, id: 1bSyXkNyzwrxPajbY-WNTiCCs4ycwcYnW
title: test.zip, id: 1auHH5qiRaHh4T0bNR_MVe1kOkczY4nmx
title: Colab Notebooks, id: 1Iz8nrdf6kHl5Jimc2xL9zcws8JEA1MUZ
```

```
[ ]  file_id = '1udaEEc4Zd9Rc-JJE7dyl5KWzpWl_F8Sp'
     downloaded = drive.CreateFile({'id': file_id})
     downloaded.GetContentFile("train.zip")

     file_id = '1auHH5qiRaHh4T0bNR_MVe1kOkczY4nmx'
     downloaded = drive.CreateFile({'id': file_id})
     downloaded.GetContentFile("test.zip")
```

We unzip the information and create the folders for the training, test and validation data

```
!mkdir NeumoX
!cd NeumoX && unzip -q ../train.zip
!cd NeumoX && unzip -q ../test.zip
```

```
mkdir: cannot create directory ‘NeumoX’: File exists
replace train/NORMAL/IM-0115-0001.jpeg? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
A
replace test/NORMAL/IM-0001-0001.jpeg? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
```

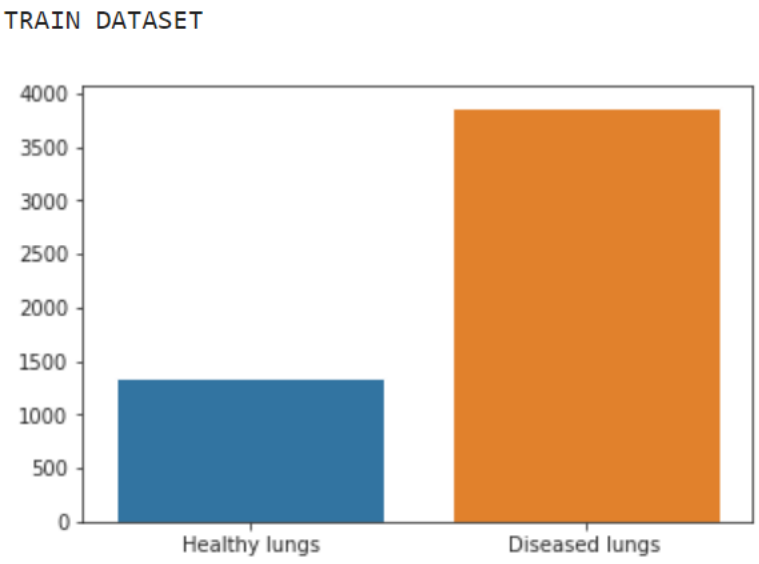
a. Image processing and display

List containing the names of the entries in the directory given by path

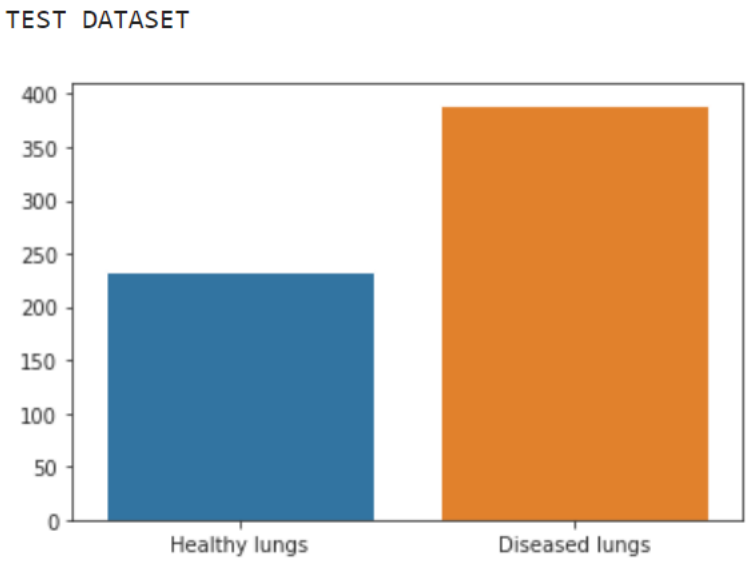
```
Train_normal = os.listdir('../content/NeumoX/train/NORMAL/')
Train_neumonia = os.listdir('../content/NeumoX/train/PNEUMONIA/')
```

```
Test_normal = os.listdir('../content/NeumoX/test/NORMAL/')
Test_neumonia = os.listdir('../content/NeumoX/test/PNEUMONIA/')
```

Our TRAIN dataset is an imbalanced dataset



Our TEST dataset is an imbalanced dataset too



We import Backend utilities in order to returns the default image data format convention ('channels_first' or 'channels_last').

- Channels first means that in a specific tensor (consider a photo), you would have (Number_Of_Channels, Height , Width).
- Channels last means channels are on the last position in a tensor (n-dimensional array).

We have only one channel (greyscale images)

```
from tensorflow.keras import backend as K

img_width, img_height = 64, 64

if K.image_data_format() == 'channels_first':
    input_shape = (1, img_width, img_height)
else:
    input_shape = (img_width, img_height, 1)

input_shape
```

```
(64, 64, 1)
```

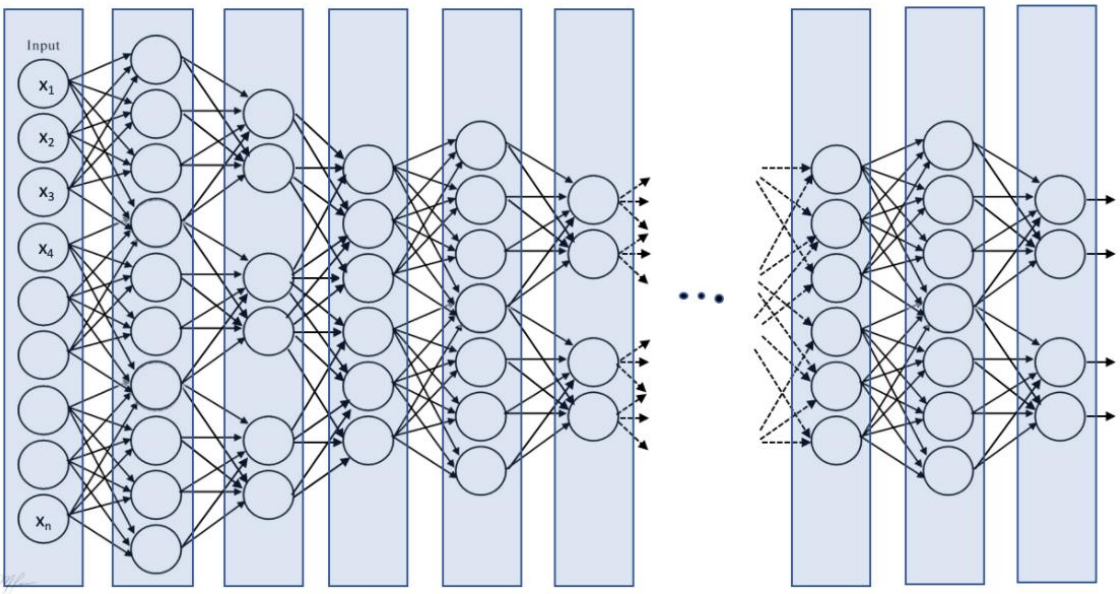
4. LIFE-CYCLE FOR NEURAL NETWORK MODEL IN KERAS

We must follow a strict model life-cycle to create and evaluate Deep learning neural networks in Python with Keras

1. Define Network
2. Compile Network
3. Fit Network
4. Evaluate Network
5. Make Predictions

    a. Define Network

Sequential model type will be used. Sequential is the easiest way to build a model in Keras. It allows you to build a model layer by layer.



We use the 'add()' function to add layers to our model.

Our layers are Conv2D layers. These are convolution layers that will deal with our input images, which are seen as 2-dimensional matrices.

32 in the first layer and 64 in the second layer are the number of nodes in each layer. This number can be adjusted to be higher or lower, depending on the size of the dataset. In our case, 32 and 64 work well.

Kernel size is the size of the filter matrix for our convolution. So a kernel size of 3 means we will have a 3x3 filter matrix.

Activation is the activation function for the layer. The activation function we will be using for the most of our layers is the ELU, Exponential Linear Unit, is a function that tend to converge cost to zero faster and produce more accurate results. This activation function has been proven to work well in neural networks.

In between the Conv2D layers and the dense layer, there is a 'Flatten' layer. Flatten serves as a connection between the convolution and dense layers.

‘Dense’ is the layer type we will use in for our output layer. Dense is a standard layer type that is used in many cases for neural networks.

```
## Construction of the Keras Sequential model begins.
model=tf.keras.models.Sequential()

#Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.
model.add(tf.keras.layers.BatchNormalization(input_shape=input_shape))
#First convolutional layer with elu-activation
#2D convolution layer (e.g. spatial convolution over images)
model.add(tf.keras.layers.Conv2D(32, (3, 3), padding='same', activation='elu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2,2)))
model.add(tf.keras.layers.Dropout(0.25))

# For the Conv2D function the specifications and recommendations of the web page have been followed:
#https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/
model.add(tf.keras.layers.Conv2D(64, (3, 3), padding='same', activation='elu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2,2)))
model.add(tf.keras.layers.Dropout(0.25))

model.add(tf.keras.layers.BatchNormalization(axis=1))
model.add(tf.keras.layers.Conv2D(128, (3, 3), padding='same', activation='elu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.BatchNormalization(input_shape=input_shape))
model.add(tf.keras.layers.Conv2D(256, (3, 3), padding='same', activation='elu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2,2)))
model.add(tf.keras.layers.Dropout(0.25))

#Fully connected layers are defined using the Dense class. We can specify the number of neurons or nodes in the layer as the first argument
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128))
model.add(tf.keras.layers.Activation('elu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(256))
model.add(tf.keras.layers.Activation('elu'))
model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.Dense(2))
#As the training is not multiclass (a lung is healthy or not), we will use Softmax for the final layer.
model.add(tf.keras.layers.Activation('softmax'))

# Different neural network models have been tested, being the one indicated in this code the one that has given the best results
model.summary()
```

Our Sequential model:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization (BatchNo (None, 64, 64, 1)         4
_________________________________________________________________
conv2d (Conv2D)              (None, 64, 64, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 32, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 16, 16, 64)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 64)        64
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 8, 8, 128)         0
_________________________________________________________________
batch_normalization_2 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 256)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 4, 4, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 128)               524416
_________________________________________________________________
activation (Activation)      (None, 128)               0
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               33024
_________________________________________________________________
activation_1 (Activation)    (None, 256)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 514
_________________________________________________________________
activation_2 (Activation)    (None, 2)                 0
=================================================================
Total params: 946,374
Trainable params: 946,084
Non-trainable params: 290
_________________________________________________________________
```

Image data preprocessing

```
TRAIN_DIR = "../content/NeumoX/train/"
TEST_DIR =  "../content/NeumoX/test/"
```

```
#Generate batches of tensor image data with real-time data augmentation
train_datagen=image.ImageDataGenerator(rescale=1./255.0, shear_range=0.2,zoom_range=0.2, horizontal_flip=True)
test_datagen=image.ImageDataGenerator(rescale=1./255.0)
```

```
%matplotlib inline
gen = image.ImageDataGenerator()
#https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator
train_batches = train_datagen.flow_from_directory("../content/NeumoX/train/",model.input_shape[1:3],color_mode="grayscale",shuffle=True,seed=1,
                                          batch_size=16, class_mode='binary')
test_batches = test_datagen.flow_from_directory("../content/NeumoX/test/", model.input_shape[1:3], shuffle=False,
                                        color_mode="grayscale", batch_size=16, class_mode='binary')
```

```
Found 5216 images belonging to 2 classes.
Found 624 images belonging to 2 classes.
```

b. Compile Network

Next, we need to compile our model. Compiling the model takes three parameters: optimizer, loss and metrics.

The optimizer controls the learning rate. We will be using ‘adam’ as our optmizer. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout training.

The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.

As part of the optimization algorithm, the error for the current state of the model must be estimated repeatedly. This requires the choice of an error function, conventionally called a loss function that can be used to estimate the loss of the model so that the weights can be updated to reduce the loss on the next evaluation. We use sparse categorical crossentropy because our classes are mutually exclusive

To make things even easier to interpret, we will use the ‘sparse categorical accuracy’ metric to see the accuracy score on the validation set when we train the model.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['sparse_categorical_accuracy'])
```

c. Fit Network

Now we will train our model. To train, we will use the ‘fit()’ function on our model with the following parameters: training data, validation data and the number of epoch

The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point. After that

point, the model will stop improving during each epoch. For our model, we will set the number of epochs to 100

```
#We have defined our model and compiled it ready for efficient computation
#Trains the model for a given number of epochs
history=model.fit(train_batches,validation_data=test_batches,epochs=100, steps_per_epoch=80, validation_steps=20)
```

d. Evaluate Network

Evaluation is a process during development of the model to check whether the model is best fit for the given problem and corresponding data.

The model evaluates the loss across all of the test patterns, as well as any other metrics specified when the model was compiled, like classification accuracy.

After 100 epochs, we have gotten to 91.66% sparse categorical accuracy on our validation set

```
score = model.evaluate(test_batches, verbose=0)
print('Test sparse_categorical_accuracy:', score[1])
print('Test loss:', score[0])
```

```
Test sparse_categorical_accuracy: 0.9166666865348816
Test loss: 0.29593148827552795
```

e. Make Predictions

Finally, once we are satisfied with the performance of our fit model, we can use it to make predictions on new data.

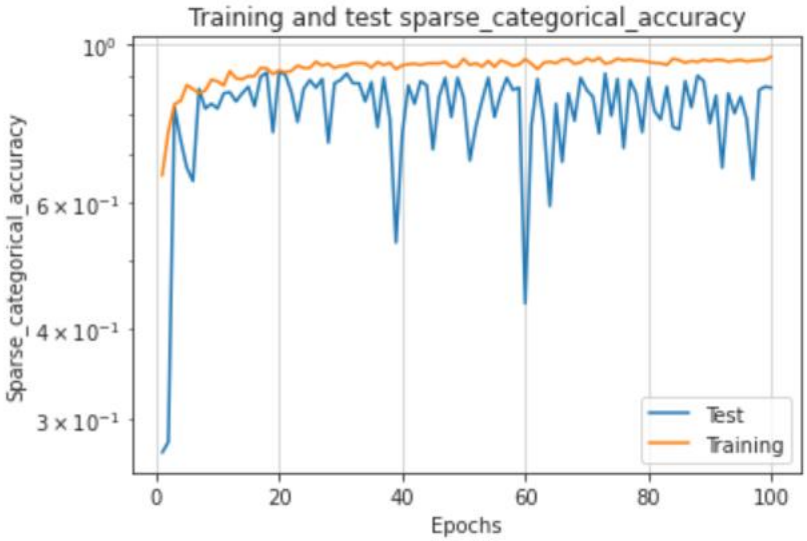
This is as easy as calling the predict() function on the model with an array of new input patterns.

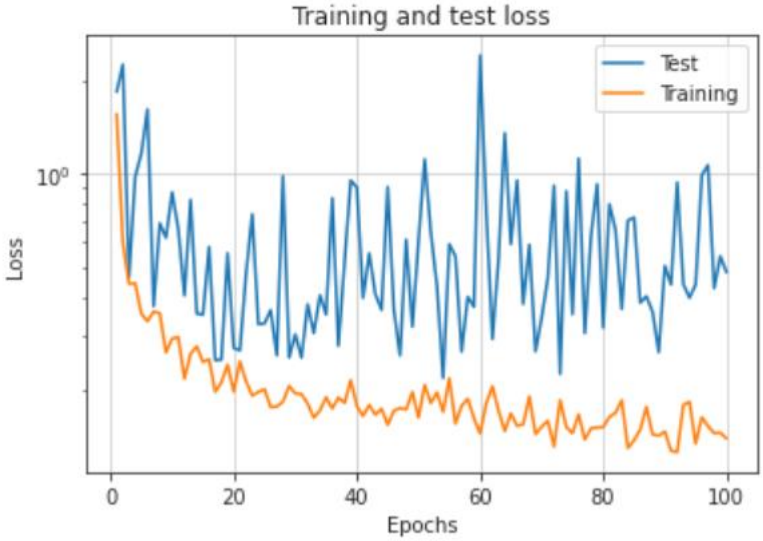
```
tam = len(test_batches)
prediccion_1 = model.predict(test_batches, verbose=True)
prediccion_2 = pd.DataFrame(prediccion_1)
prediccion_2['item']=prediccion_2.reset_index().index
prediccion_2["filename"] = test_batches.filenames
prediccion_2["label"] = (prediccion_2["filename"].str.contains("PNEUMONIA")).apply(int)
prediccion_2['prediccion_2'] = (prediccion_2[1]>0.5).apply(int) #In the test dataset, I indicate that more than a 0.5 probability of being 1 is a 1
print(prediccion_2)
print(prediccion_2.columns)
#batch_size=16 / Found 624 images belonging to 2 classes --> 39
```

5. RESULTS AND METRICS

a. Values of Loss and Sparse Categorical Accuracy

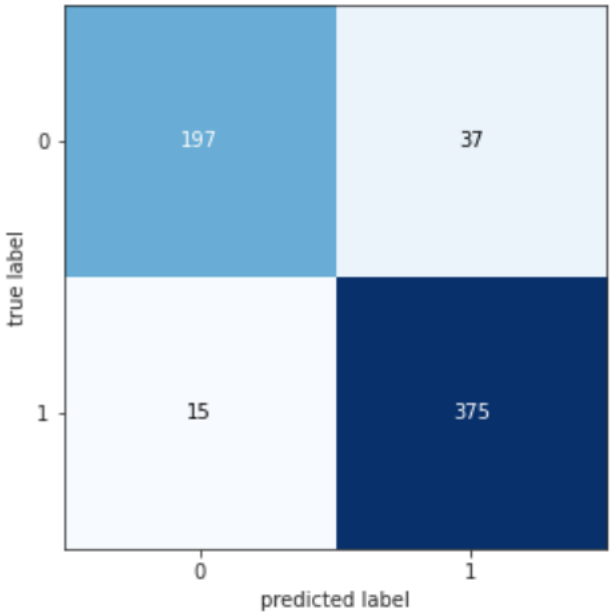
The value of sparse_categorical_accuracy indicates the part of the correct conjectures: we can see that as we run epochs it increases its value until it reaches the maximum. On the other hand, the value of loss, which is calculated with the test data that has not been used in the training, is lower reaching a threshold that cannot be exceeded. This is exactly what tells us that the **Overfitting** effect is taking place from a number of epochs.





b. Confusion Matrix

It is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one. Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class



c. Precision and Recall

The precise definition of **recall** is the number of true positives divided by the number of true positives plus the number of false negatives

Precision is defined as the number of true positives divided by the number of true positives plus the number of false positives

The **F1 score** is the harmonic mean of precision and recall taking both metrics into account. F1 Score might be a better measure to use if we need to seek a balance between Precision and Recall AND there is an uneven class distribution (large number of Actual Negatives).

A false positive (a non-ill person who is diagnosed with a disease) would not cause physical harm in most cases, because the disease would be easily ruled out by additional tests and the medical follow-up that would be carried out.

If a false negative occurs (a sick person who is not diagnosed with the disease), the patients' lives could be at risk. Because of this, **the recall value is very important**

	Precision	Recall	F1 score	Observaciones
Normal lung	0,93	0,84	0,88	234
Diseased lung	0,91	**0,96**	0,94	390
Macro avg	0,92	0,90	0,91	624
Weighted avg	0,92	0,92	0,92	624

d. AUC - ROC Curve

It is one of the most important evaluation metrics for checking any classification model's performance

AUC - ROC curve is a performance measurement for the classification problems at various threshold settings. ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. Higher the AUC, the better the model is at predicting 0s as 0s and 1s as 1s. By analogy, the Higher the AUC, the better the model is at distinguishing between patients with the disease and no disease.

The ROC curve is plotted with TPR (True Positive Rate or Recall) against the FPR (False Positive Rate).

An excellent model has AUC near to the 1 which means it has a good measure of separability. A poor model has AUC near to the 0 which means it has the worst measure of separability. In fact, it means it is reciprocating the result. It is predicting 0s as 1s and 1s as 0s. And when AUC is 0.5, it means the model has no class separation capacity whatsoever.

When AUC is approximately 0.5, this is the worst situation, the model has no discrimination capacity to distinguish between positive class and negative class.

When AUC is approximately 0, the model is actually reciprocating the classes. It means the model is predicting a negative class as a positive class and vice versa.

```
# calculate AUC
auc = roc_auc_score(prediccion_2["label"], prediccion_2["prediccion_2"])
```

```
print('AUC: %.3f' % auc)
```

```
AUC: 0.902
```

```
print('AUC is {:.4f}' .format(auc) ,end=""),
print(', it means there is a {:.2f}' .format(auc*100) ,end=""),
print('% chance that the model will be able to distinguish between positive class and negative class.')
```

```
AUC is 0.9017, it means there is a 90.17% chance that the model will be able to distinguish between positive class and negative class.
```

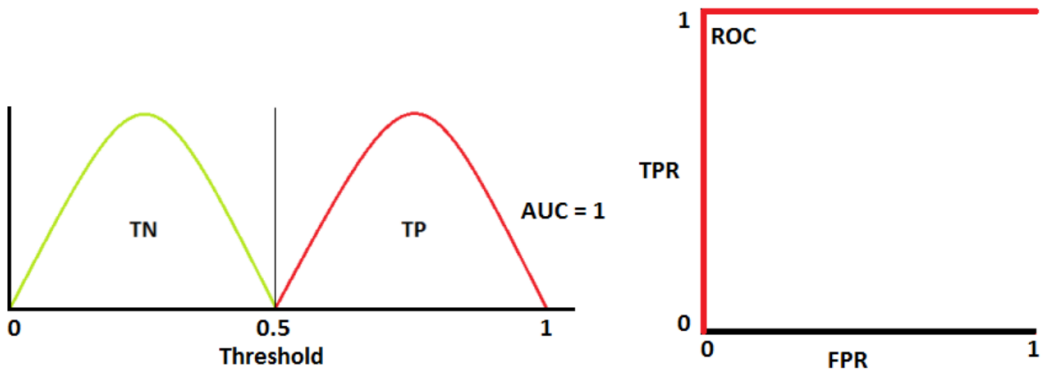
Graphic examples

TN: True negatives; TP: True positives; FN: False negatives; FP: False positives

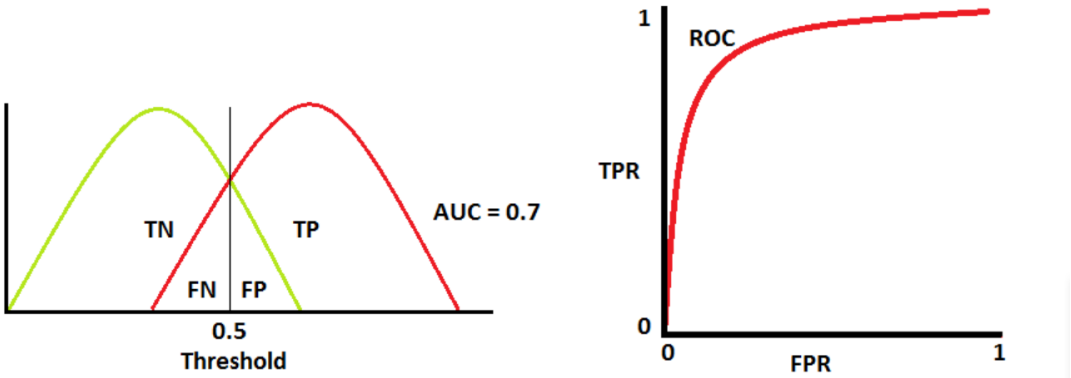
TPR = TP/(TP+FN)

FPR = FP/(FP+TN)

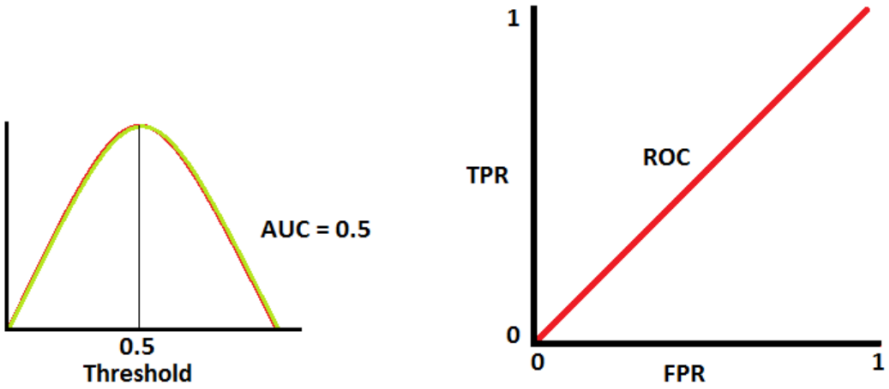
AUC = 1



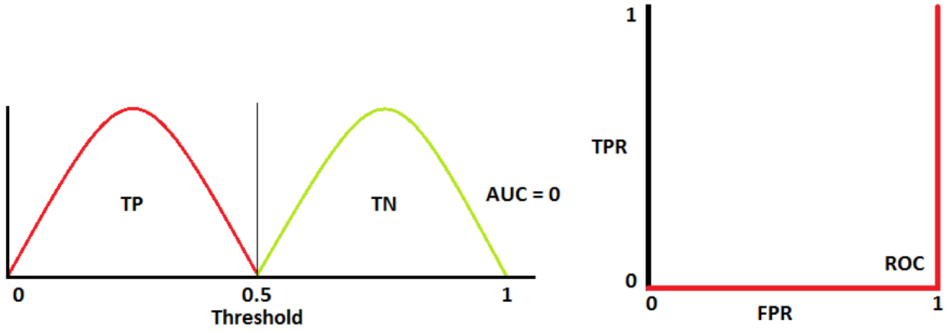
AUC = 0.7



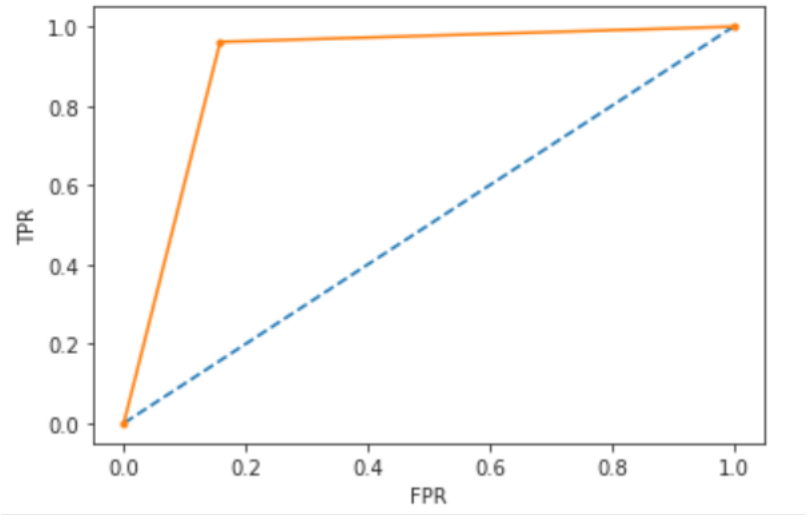
AUC = 0.5



AUC = 0



In our model the graph looks like this



e. Precision-Recall (PR) Curve

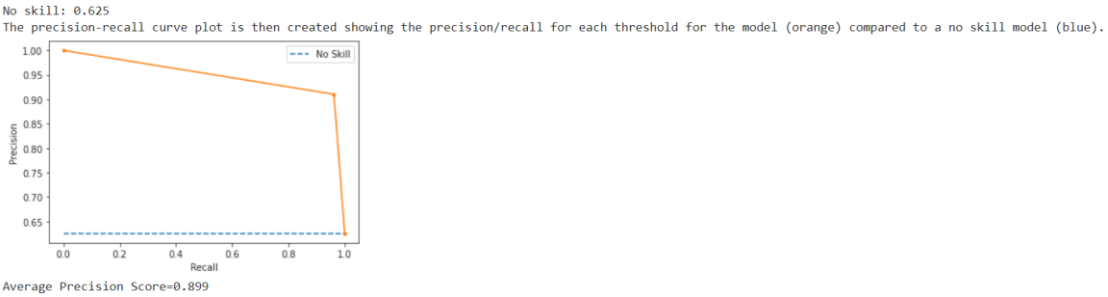
The precision and recall metrics are related so that if you train your classifier to increase **precision**, the **recall** will decrease and vice versa.

Suppose we want to make our classifier very sensitive to images of diseased lungs, that is, we want to increase the recall, probably to a value close to 1 (100%). We could do this easily if we make our model a simple function that always returns 1. The downside to this is that our precision would be very poor, practically random, probably close to 0.5 if our dataset is balanced.

Now suppose we want a very precise classifier, with a precision close to 1. Obviously, if our classifier is very strict when it comes to considering a diseased lung image, we will miss many by the way and this will reduce the recall. Our classifier will be very accurate, but less sensitive.

And how do we tell the classifier if we want it to be more precise, or more sensitive? Generally, the model will make the decision of whether the classification is positive or negative if the value returned by the model exceeds a decision threshold. If we increase

this value, we will be increasing the precision, if we decrease it, we will increase the sensitivity (recall)



Generally, the use of ROC curves and precision-recall curves are as follows:

1. ROC curves should be used when there are roughly equal numbers of observations for each class.
2. Precision-Recall curves should be used when there is a moderate to large class imbalance.

The reason for this recommendation is that ROC curves present an optimistic picture of the model on datasets with a class imbalance.

6. CONCLUSIONS

The results described in this work prove that deep learning models can be applied to anticipate a high-level diagnosis (sick / not sick) of a patient's lung, that is, these models can be used to make a first filter.

Treatment of the affected patient can be improved and anticipated. For this, a volume of data (images) will be needed to train the model and a much higher processing capacity.

As we indicated at the beginning of this work, with these deep learning models we will avoid the stress derived from the high workload of health workers and we will make the health system more efficient, reducing resources and increasing productivity and patient well-being.

In the future, applying transfer learning, this work can be expanded to include the detection and diagnosis of a certain pulmonary pathology such as cancer or fibrosis.

In no case is the substitution of the healthcare professional who will always be responsible for supervision and final diagnosis concluded from this work