UNIVERSITÀ DI PISA

University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Computational Intelligence and Deep Learning

Chest X-ray Image Classification

Academic year 2021-2022

Drive:
https://drive.google.com/drive/folders/1NEL5Kxnp32_fc306ejykcNkxQnFVRM6S?usp=sharing

Valerio Giannini , Alessio Serra, Marco Simoni

Sommario

1 Introduction 4
1.1 Related Works 4
1.2 Dataset 4
1.3 METRICS USED 6
2 Methods & Techniques 7
2.1 Data Augmentation 7
2.2 ReduceLROnPlateau 7
2.3 EarlyStopping and ModelCheckpoint 7
2.4 Gathering More Data 7
2.5 Data Preprocessing 8
2.6 Learnable Resizer 12
3 CNN from Scratch 15
3.1 FIRST MODEL 15
3.2 SECOND MODEL 17
3.3 HYPERPARAMETER TUNING WITH HYPERBAND KERAS TUNER 19
3.4 FUSED 20
4 Pre-Trained Models 25
4.1 VGG16 25
4.1.1 Feature Extraction 26
4.1.2 Fine Tuning 29
4.1.3 Gathering More Data 30
4.2 ResNet152 32
4.2.1 Feature Extraction 32
4.2.2 Fine Tuning 34
4.2.3 Gathering More Data 35
4.2.4 ResNet50 37
4.3 CheXNet 38
4.3.1 Feature Extraction 39
4.3.2 Fine Tuning 42
4.3.3 Gathering More Data 43
5 Error Analysis 46
6 Explainability 51
7 Ensemble 60
7.1 Average Voting 60
7.2 Weighted Average Voting 61
7.2.1 Genetic *Algorithm* 61
7.2.2 Results 61

8 Conclusions 63
9 References 64

1 Introduction

Our goal consists in the classification of chest X-Ray images in order to detect the pathologies Effusion, Infiltration and Atelectasis and distinguish them from a patient devoid by any kind of pathology.

1.1 Related Works

Recent advancements in deep learning and large datasets have enabled algorithms to surpass the performance of medical professionals in a wide variety of medical imaging tasks.

Automated diagnosis from chest radiographs has received increasing attention with algorithms for pulmonary tuberculosis classification [13] and lung nodule detection [14]. [15] studied the performance of various convolutional architectures on different abnormalities using the publicly available OpenAI dataset [16]. Then ChestX-ray-14 [9] was released, an order of magnitude larger than previous datasets of its kind, and also benchmarked different convolutional neural network architectures pre-trained on ImageNet. Recently [17] exploited statistical dependencies between labels in order make more accurate predictions, outperforming [9] on 13 of 14 classes.

1.2 Dataset

The dataset has been retrieved from a portion of the ChestX-ray14 dataset [9]. It is a medical imaging dataset which comprises 112,120 frontal-view X-ray images of 30,805 unique patients. It annotates each image with up to 14 different thoracic pathology labels with the text-mined fourteen common disease labels using automatic extraction methods on radiology reports through NLP techniques. In our case, we have considered just four classes of which just one indicates a normal condition of the patients' lungs; the names of the classes are the following:

- **No Finding:** Normal Lung.
- **Effusion**: excess fluids outside the lungs.
- **Infiltration**: is a substance denser than air, such as pus, blood, or protein, which lingers within the parenchyma of the lungs.
- **Atelectasis**: It occurs when the tiny air sacs (alveoli) within the lung become deflated or possibly filled with alveolar fluid.

The distribution of the classes in each set is shown in the following figures:

- Training Set:

  Number of Atelectasis: 2384, proportion: 0.21
  Number of Infiltration: 3240, proportion: 0.29
  Number of No Finding: 3240, proportion: 0.29
  Number of Effusion: 2279, proportion: 0.21

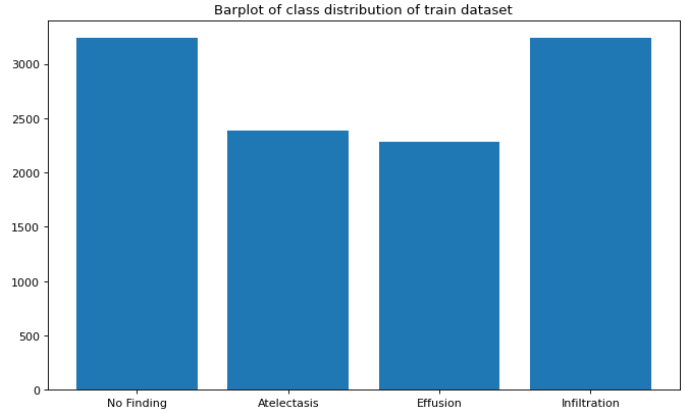


FIGURE 1: TRAINING SET DISTRIBUTION.

- Validation Set:

Number of Effusion: 283, proportion: 0.22
Number of Infiltration: 360, proportion: 0.28
Number of No Finding: 360, proportion: 0.28
Number of Atelectasis: 289, proportion: 0.22

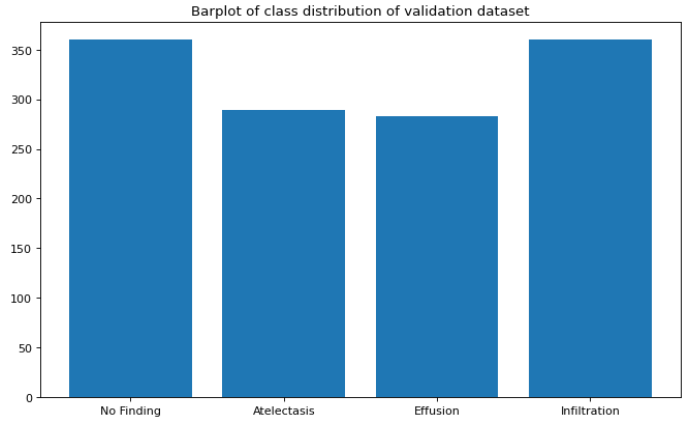


FIGURA 2: VALIDATION SET DISTRIBUTION.

- Test Set:

Number of Effusion: 314, proportion: 0.22
Number of Infiltration: 400, proportion: 0.28
Number of No Finding: 400, proportion: 0.28
Number of Atelectasis: 324, proportion: 0.22

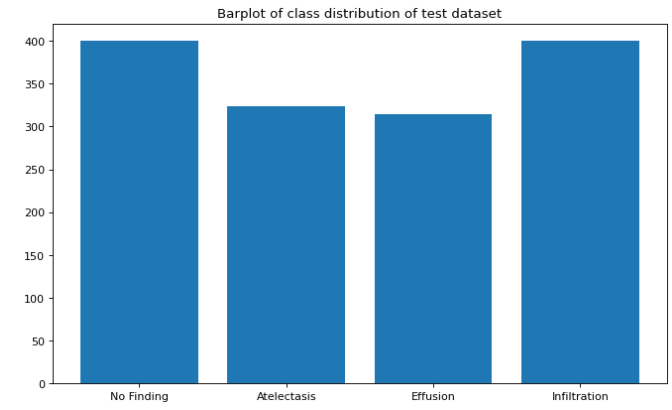


FIGURA 3: TEST SET DISTRIBUTION.

1.3 METRICS USED

To evaluate the network performance has not been used only the accuracy, in fact, even if we supposed to have the same misclassification cost for all the classes, we also want to verify the performance on minority ones, that was quite consistent in the first version of the dataset.
For this reason, we have considered also the F1 on single classes, that corresponds to the harmonic mean of precision and recall, and the F1-macro to have a more compact index of performance.

We have also used the **ROC AUC** (Receiver Operating Characteristic Area Under the Curve) score, that considers the ROC curve. The latter is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied.
It is created by plotting the fraction of true positives out of the positives (sensitivity) vs. the fraction of false positives out of the negatives (one minus specificity), at various threshold settings.

To adapt this metric to our single label multi-class classification problem we have used the **one-vs-rest** algorithm that computes the average of the ROC AUC scores for each class against all the others.

2 Methods & Techniques

2.1 Data Augmentation

Data Augmentation is used to generate more training data from existing training samples through a number of random transformations that produce "believable images", meaning that do not corrupt the class of the image. This is helpful both to fight overfitting and to help the model generalize better.

As suggested by literature [8], *rotation* and *flipping* are best suited when dealing with the kind of images we are dealing with.

```
data_augmentation = tf.keras.Sequential([
  layers.RandomFlip('horizontal'),
  layers.RandomRotation(0.2)
])
```

2.2 ReduceLROnPlateau

Dynamic approach which adjusts the learning rate when stagnation in model performance is detected.

```
keras.callbacks.ReduceLROnPlateau(
    monitor = monitor,
    factor = 0.8,
    patience = (patience*6/10),
    min_lr = min_lr,
    min_delta=0.001,
    verbose = 1),
```

2.3 EarlyStopping and ModelCheckpoint

Early Stopping is a form of regularization used to avoid overfitting. It provides guidance as to how many iterations can be run before the learner begins to overfit.

Used in conjunction with *Model Checkpoint*, it allows to save the best model obtained before the learner starts to overfit the training data.

```
keras.callbacks.EarlyStopping(
    monitor = monitor,
    patience = patience),
keras.callbacks.ModelCheckpoint(
    filepath = save_path,
    monitor = monitor,
    verbose=1,
    save_best_only=True)
```

2.4 Gathering More Data

Increase the size of the training data to try improving generalization capabilities of the model, reducing overfitting. In particular, we went from a training dataset containing 7.996 images to one containing 12.435 images.

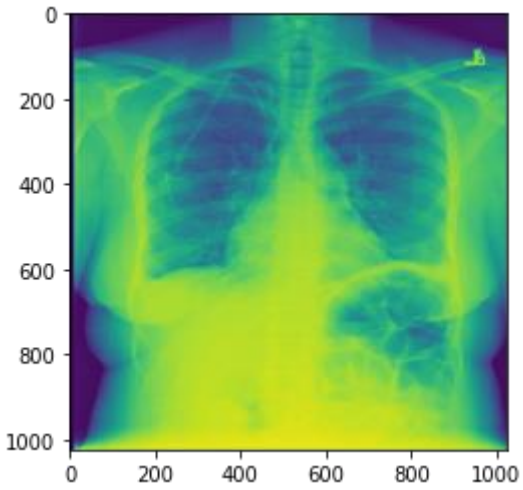
2.5 Data Preprocessing

In order to improve performances of the classifiers, data preprocessing operations have been applied to the images.

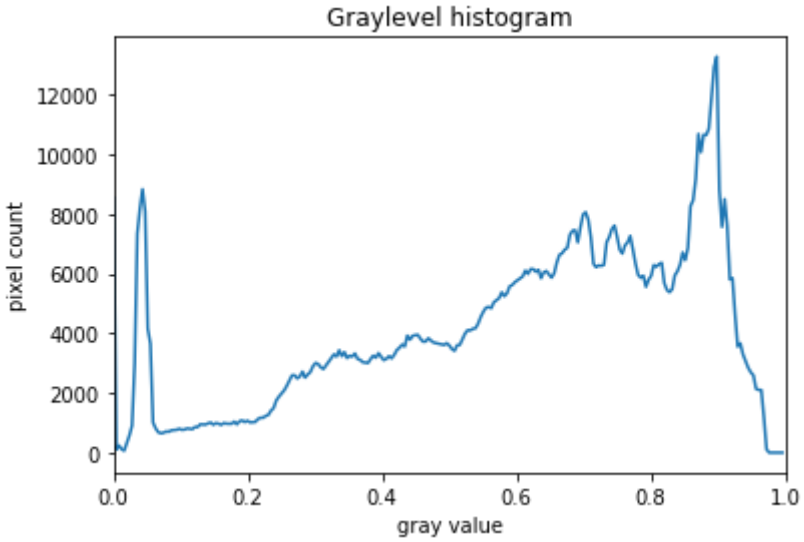
More precisely, the technique that has been adopted consists into a variation of the classical 'Thresholding' technique. In particular, the following strategies have been implemented:

- select only the shapes from the image, that is leave the pixels belonging to the shapes "on," while turning the rest of the pixels "off," by setting their color channel values to zeros.
- apply the threshold t such that pixels with grayscale values on one side of t will be turned "on", while pixels with grayscale values on the other side will be turned "off".
- turn "on" all pixels which have values smaller than the threshold, so use the less operator < to compare the blurred image to the threshold t. The operator returns a mask, that captures in the variable binary mask. It has only one channel, and each of its values is either 0 or 1.
- apply the binary mask to the original-colored image. What we are left with is only the colored shapes from the original.

The described steps could be better analysed in the following illustrative; the figure shows a picture of a patience affected by Atelectasis.



From this point on, firstly the image has been transformed into a 'grey scale' image and then a gaussian filter has been applied. One way to determine a "good" value for threshold t is to look at the grayscale histogram of the image and try to identify what grayscale ranges correspond to the shapes in the image or the background.

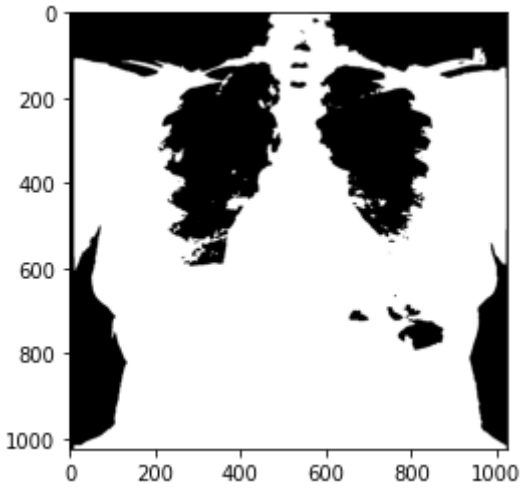


The downside of the simple thresholding technique is that we must make a guess about the threshold `t` by inspecting the histogram. There are also *automatic thresholding* methods that can determine the threshold automatically for us. One such method is 'Otsu Method'. It is particularly useful for situations where the grayscale histogram of an image has two peaks that correspond to background and objects of interest.

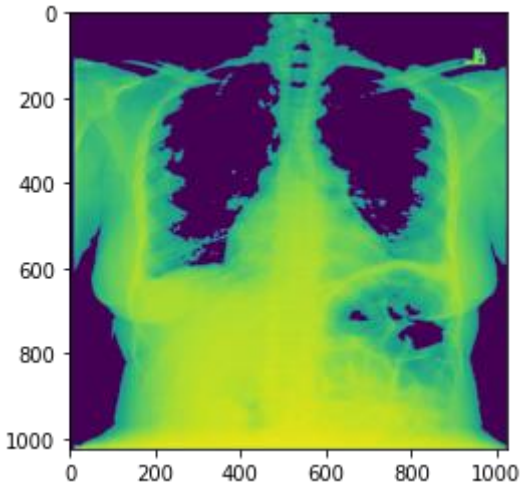
For our cases, Otsu's method has been applied.

```
t = skimage.filters.threshold_otsu(blurred_image)
```

The binary mask created by the thresholding operation can be seen in the following picture.



The final picture that we could obtain performing the described steps is the following:



Although all the applied pre-process operations, the outcomes that we could get performing pre-process are worse than the ones we get without any kind of applied pre-process operation. In all the tried configurations we have obtained worse results applying pre-processed dataset. Here, there are two examples.

Model	Accuracy	Loss	Roc Auc Score	F1-macro	Inference time
Prep_scratch.h5	0.578173	1.015191	0.813689	0.562666	7.863272
Prep_fused.h5	0.517802	1.148516	0.755353	0.505125	**2.009151**
No_prep_fused.h5	0.526316	1.146247	0.765587	0.485992	2.026920
No_prep_scratch.h5	**0.581269**	**0.997064**	**0.822521**	**0.564399**	7.856832

TABLE 11: GLOBAL PERFORMANCE METRICS OF MODELS TRAINED WITH AND WITHOUT THE PRE-PROCESSED DATASET.

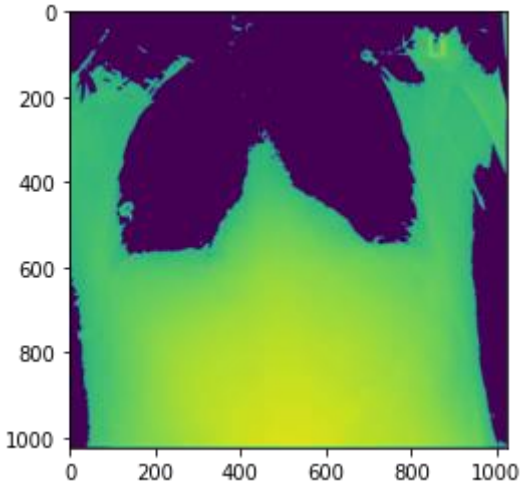
Model	Atelectasis	Infiltration	No Finding	Effusion
Prep_scratch.h5	0.400802	0.584392	0.592000	0.673469
Prep_fused.h5	**0.413428**	0.456693	0.524272	0.626109
No_prep_fused.h5	0.273839	0.454902	0.579365	0.635864
No_prep_scratch.h5	0.398357	**0.584440**	**0.592686**	**0.682111**

TABLE 22: F1 ON SINGLE CLASSES OF MODELS TRAINED WITH AND WITHOUT THE PRE-PROCESSED DATASET.

One possible explanation related to these poor performances could be given looking at the following picture:

Original Atelectasis

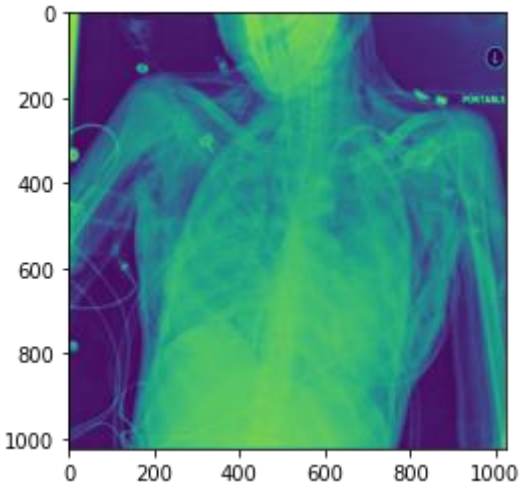
Pre-processed Atelectasis



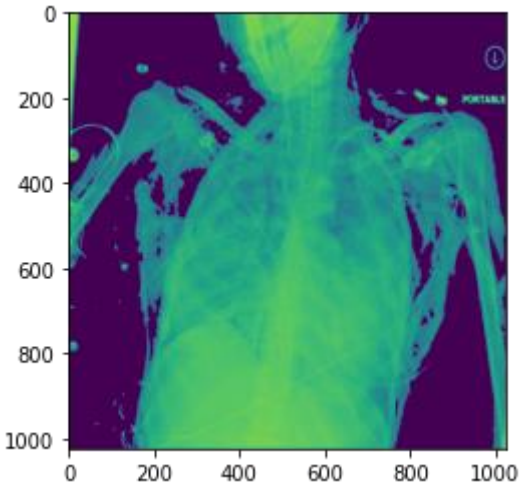
From the two figures we can spot the heavy operation of preprocess, and in general we can say that these kinds of operations are probably going to cut off some meaningful detail that the networks leverage to understand the pathology.

One exception is probably represented by the class of ‘infiltration’; in fact, if we look at the following images. In these two cases, the pre-process operations don’t change heavily the morphology of the pneumonia’s x-ray but instead the network (in general) benefits from these kinds of implementations.

Original Infiltration

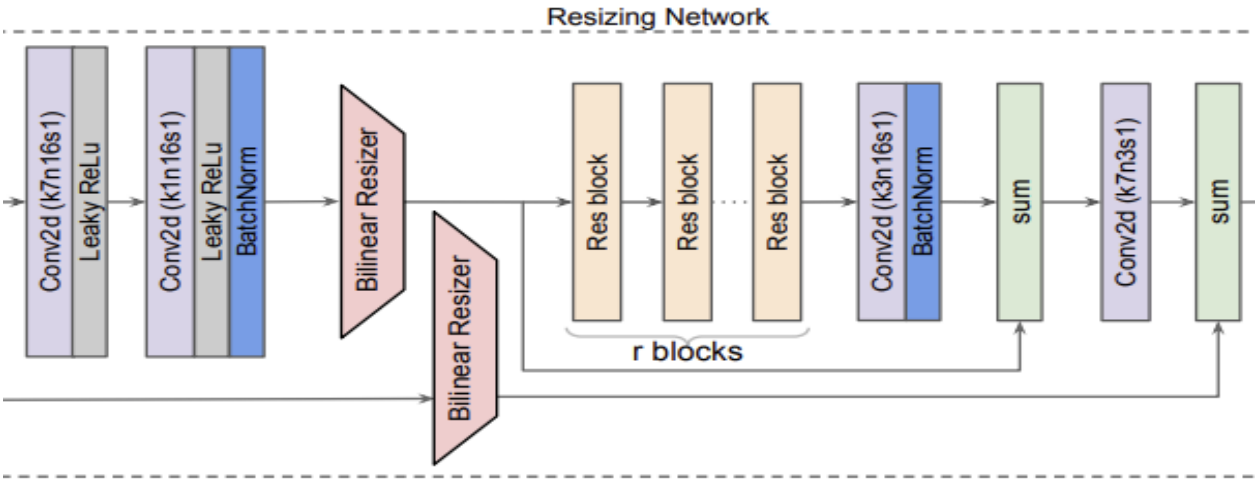


Pre-processed Infiltration



2.6 Learnable Resizer

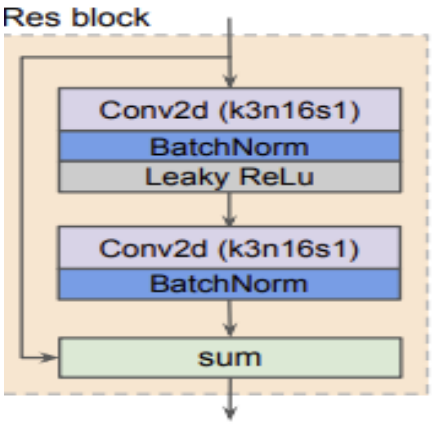
In order to allow minibatch learning and keep up computation limitations, the implementation of a learnable resizer is needed. The structure we have taken cue from is the one presented by Keras and it is showed in the following figure.



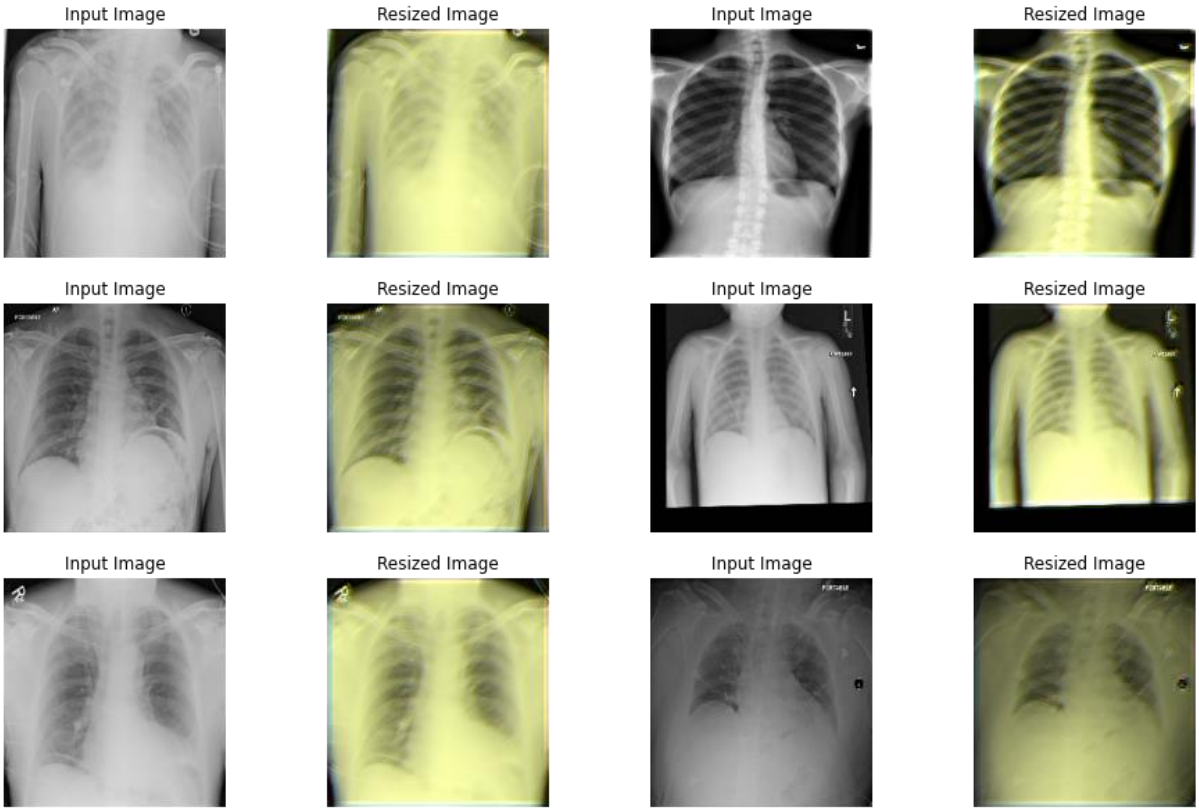
The main characteristics of this model are its bilinear feature resizing and skip connection. The former allows for the computed features of the CNN to be incorporated into the model, while the latter enables the bilinearly resized image to be passed into the baseline task.

The proposed architecture allows for the simultaneous resizing of an image to a wide range of aspect and size. In the model, the main characteristics could be summarized by the following parts:

- There are r identical residual blocks in our model and in our experiments, we set r = 1 or 2.
- All intermediate convolutional layers have n = 16 kernels of size 3 × 3.
- The first and the last layers consist of 7 × 7 kernels. The larger kernel size in the first layer allows for a 7 × 7 receptive field on the original image resolution.
- batch normalization layers and LeakyReLu activations with a 0.2 negative slope coefficient



In the following pictures we could easily spot the effects of the learnable resizer; for visual purposes, we have just collected the first 6 images of the train set.



In the tables below, we are going to show the real benefits of the learnable resizer; as we can see we have applied two different model: Resizer_1 has not Data Augmentation layer while Resizer_2 has Data Augmentation layer. For all the metrics here exposed both models are better than no-learnable-resizer counterparts.

Model	Accuracy	Loss	Roc Auc Score	F1-macro	Inference time
Resizer_2.h5	**0.594427**	**0.996613**	**0.832783**	**0.557707**	6.925866
No_resizer_2.h5	0.526316	1.146247	0.765587	0.485992	2.010015
Resizer_1.h5	0.565015	1.033257	0.810733	0.551248	6.784299
No_resizer_1.h5	0.531734	1.111113	0.773273	0.508024	**1.796607**

TABLE 33: GLOBAL PERFORMANCE METRICS OF MODELS TRAINED WITH AND WITHOUT THE LEARNABLE RESIZER.

Model	Atelectasis	Infiltration	No Finding	Effusion
Resizer_2.h5	0.307305	**0.574956**	**0.642369**	**0.706199**
No_resizer_2.h5	0.273839	0.454902	0.579365	0.635864
Resizer_1.h5	**0.412639**	0.542694	0.580468	0.669192
No_resizer_1.h5	0.354212	0.463602	0.576832	0.637450

TABLE 44: F1 ON SINGLE CLASSES OF MODELS TRAINED WITH AND WITHOUT THE LEARNABLE RESIZER.

3 CNN from Scratch

In this section, we are going to show the main (from scratch) model configurations we have implemented, starting from the one that has achieved the lowest performances to the one that has achieved the best performances.

3.1 FIRST MODEL

The base structure of the first simplest model is the following:

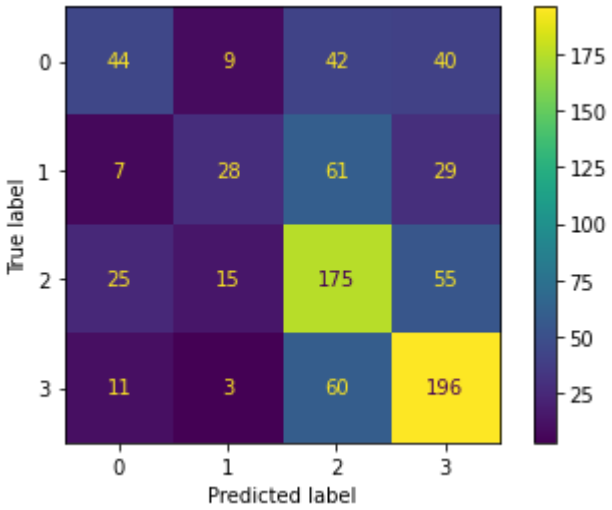
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 227, 227, 3)]	0
conv2d_15 (Conv2D)	(None, 227, 227, 32)	11648
conv2d_16 (Conv2D)	(None, 227, 227, 32)	123936
conv2d_17 (Conv2D)	(None, 227, 227, 32)	25632
max_pooling2d_8 (MaxPooling 2D)	(None, 113, 113, 32)	0
conv2d_18 (Conv2D)	(None, 57, 57, 64)	51264
conv2d_19 (Conv2D)	(None, 57, 57, 64)	102464
max_pooling2d_9 (MaxPooling 2D)	(None, 56, 56, 64)	0
conv2d_20 (Conv2D)	(None, 28, 28, 64)	102464
conv2d_21 (Conv2D)	(None, 28, 28, 64)	102464
max_pooling2d_10 (MaxPoolin g2D)	(None, 27, 27, 64)	0
conv2d_22 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_23 (Conv2D)	(None, 14, 14, 128)	147584
max_pooling2d_11 (MaxPoolin g2D)	(None, 13, 13, 128)	0
conv2d_24 (Conv2D)	(None, 7, 7, 128)	147584
conv2d_25 (Conv2D)	(None, 7, 7, 128)	147584
max_pooling2d_12 (MaxPoolin g2D)	(None, 6, 6, 128)	0
conv2d_26 (Conv2D)	(None, 6, 6, 256)	295168
conv2d_27 (Conv2D)	(None, 6, 6, 256)	590080
max_pooling2d_13 (MaxPoolin g2D)	(None, 5, 5, 256)	0

```
max_pooling2d_13 (MaxPoolin  (None, 5, 5, 256)        0
g2D)

conv2d_28 (Conv2D)           (None, 5, 5, 256)        590080

max_pooling2d_14 (MaxPoolin  (None, 4, 4, 256)        0
g2D)

conv2d_29 (Conv2D)           (None, 4, 4, 512)        1180160

max_pooling2d_15 (MaxPoolin  (None, 4, 4, 512)        0
g2D)

flatten_1 (Flatten)          (None, 8192)             0

dense_1 (Dense)              (None, 128)              1048704

classifier (Dense)           (None, 4)                516

=================================================================
Total params: 4,741,188
Trainable params: 4,741,188
Non-trainable params: 0
_________________________________________________________________
```

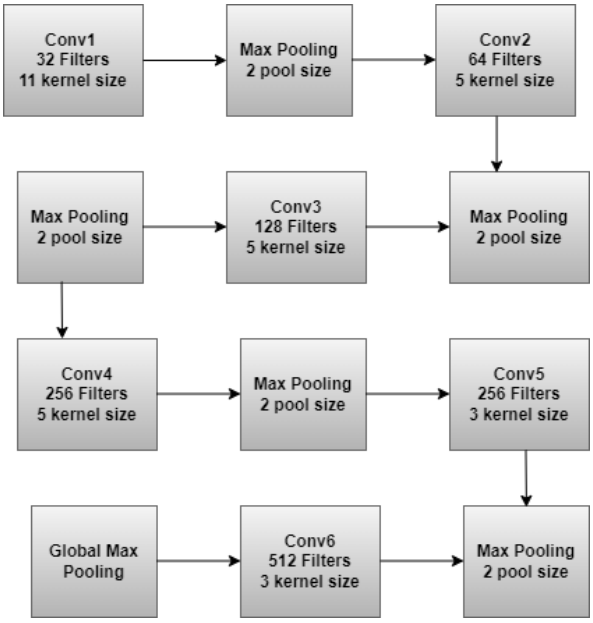
The performances that can be obtained training this model are showed in the snippet below:

Classification Report

class	precision	recall	f1-score	support
0	0.5057	0.3259	0.3964	135
1	0.5091	0.2240	0.3111	125
2	0.5178	0.6481	0.5757	270
3	0.6125	0.7259	0.6644	270
accuracy			0.5537	800
macro avg	0.5363	0.4810	0.4869	800
weighted avg	0.5464	0.5537	0.5340	800



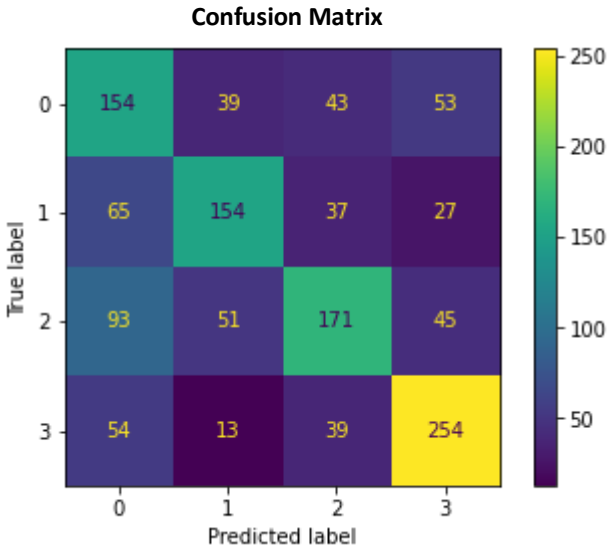
3.2 SECOND MODEL



The structure of the second model is quite different from the previous one. In particular, the approach that has been followed consists in doubling the number of filters at each layer (starting from 32 to 512) the simultaneous decreasing of the kernel sizes. The peculiar aspects could be seen in the summary below.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 227, 227, 3)]     0

conv2d_6 (Conv2D)            (None, 227, 227, 32)      11648

max_pooling2d_5 (MaxPooling  (None, 113, 113, 32)      0
2D)

conv2d_7 (Conv2D)            (None, 113, 113, 64)      51264

max_pooling2d_6 (MaxPooling  (None, 56, 56, 64)        0
2D)

conv2d_8 (Conv2D)            (None, 56, 56, 128)       204928

max_pooling2d_7 (MaxPooling  (None, 28, 28, 128)       0
2D)

conv2d_9 (Conv2D)            (None, 28, 28, 256)       819456

max_pooling2d_8 (MaxPooling  (None, 14, 14, 256)       0
2D)

conv2d_10 (Conv2D)           (None, 14, 14, 256)       590080

max_pooling2d_9 (MaxPooling  (None, 7, 7, 256)         0
2D)

conv2d_11 (Conv2D)           (None, 7, 7, 512)         1180160

global_max_pooling2d_1 (Glo  (None, 512)               0
balMaxPooling2D)

dense_1 (Dense)              (None, 4)                 2052

=================================================================
Total params: 2,859,588
Trainable params: 2,859,588
Non-trainable params: 0
_________________________________________________________________
```

The performances that we have obtained for this model are improved with respect to the ones that we have retrieved applying the first configuration.



As we can notice from the snippet below, values associated with accuracy are better than the one we get from the same model trained on top of the preprocessed dataset (see paragraph 3.1)

Model	Accuracy	Loss	Roc Auc Score	F1-macro	Inference time
CNN_from_scratch.h5	0.581269	0.997064	0.822521	0.564399	7.856832

TABLE 55: GLOBAL PERFORMANCE METRICS OF BEST CNN FROM SCRATCH MODEL.

Model	Atelectasis	Infiltration	No Finding	Effusion
CNN_from_scratch.h5	0.398357	0.58444	0.592686	0.682111

TABLE 66: F1 ON SINGLE CLASSES OF BEST CNN FROM SCRATCH MODEL.

3.3 HYPERPARAMETER TUNING WITH HYPERBAND KERAS TUNER

To find the best hyperparameters for the convolutional neural network trained from scratch, we have used the HyperBand Keras Tuner, introduced by (Li et al., 2018) in order to carry out random search optimizing the budget at our disposal, i.e. the training time.

First it is defined a search space, which depends on the hyperparameter to be tested and their possible values, in particular we have a 4D search space to tune:

- **Number of MLP layers**, searching in the discrete range [1, 3].
- **Dropout rate**, searching in the continue range [0, 0.5].
- **Number of neurons for Dense MLP layers**, searching in the discrete range [1024, 4096] with a step of 512.
- **Learning rate,** choosing among [1e-3, 1e-4, 1e-5].

The best hyperparameters found are reported in *Table 7* below.

Layers	Dropout rate	Neurons	Learning rate
1	0.23847	2560	1e-4

TABLE 7: BEST HYPERPARAMETERS FOUND WITH THE KERAS TUNER.

How Keras tuner works. After the space definition, the Hyperband runs the *Successive Halving* algorithm considering a budget B of 70 and a reduction factor **η** equal to 3.

With this algorithm at each iteration are tried a number of configuration, sampled from the search space, that depends on the budget chosen. At each iteration the number of candidate network is reduced by a factor of **η,** just keeping the best configurations and further training them for a longer number of epochs.

This process is repeated until just one configuration survives, thus obtaining an optimal configuration, exploring many possibilities through a **good compromise between *exploration and exploitation.***

Result obtained. Below are shown the differences in performances obtained with the usage of hyperparameter configuration found with the tuner and the model with the best hyperparameters found using a **trial and error approach.**

Model	Accuracy	Loss	Roc Auc Score	F1-macro	Inference time
Pre_tuner.h5	0.581269	1.043573	0.804218	0.567482	8.279156
Tuner.h5	**0.597523**	**0.996669**	**0.823669**	**0.579081**	8.297672

TABLE 8: GLOBAL PERFORMANCE METRICS OF MODEL TUNED WITH TUNER AND WITH TRIAL AND ERROR.

Model	Atelectasis	Infiltration	No Finding	Effusion
Pre_tuner.h5	**0.453686**	0.541935	0.601432	0.672872
Tuner.h5	0.401786	**0.624390**	**0.610039**	**0.680108**

TABLE 97: F1 ON SINGLE CLASSES OF MODEL TUNED WITH TUNER AND WITH TRIAL AND ERROR.

3.4 FUSED

By examining the literature regarding the training of models for the classification of images in the biomedical domain, we found interesting the work of (Pang et al., 2018) in which is introduced the **fused architecture** for neural networks.

The latter, unlike traditional convNet, does not give as input to the classifier only the feature maps produced by the last convolutional layer, but **concatenates the convolutional layer outputs at different levels of the network**, thus resulting in a more robust and adaptive deep model for biomedical image classification.

With this technique the model **exploits both shallow features and deep features** to distinguish biomedical images, obtaining more detailed local features with the shallow layers and a higher level of abstraction from deeper layers.

The architecture obtained is the one shown in *Figure 4*, where can be seen that are combined three different feature maps, respectively two shallow representations after the first and the second convolutional layer and a deeper one after the fifth layer.

Network architecture. The filter size, the stride and the padding were accurately selected in order to obtain features maps of the same size, since that they are concatenated before given as input to the Multi-Layer Perceptron classifier. The latter is composed just by 3 dense layers with a ReLU activation function, the first two with 2048 neurons and the last with a number of neurons equal to the number of classes.

Notice that:

- After each convolutional layer has been placed a max-pooling and a Normalization layer.
- A large stride has been used to achieve a faster computation and to significantly reduce the size of the inputs, as suggested by the paper.

COMPARISON BETWEEN BASE AND FUSED MODEL

In *Table 10* and in *Table 11* can be seen the results obtained with the fused-architecture model compared with the ones obtained with a model that only gives the feature maps of the last convolutional layer as input to the MLP.

They were both trained with a batch size equal to 16 and for at most 30 epochs, unless the validation loss did not improve for five consecutive epochs, in fact in that case early stopping interrupts the training to avoid overfitting. Moreover, as optimizer was used Adam with 10^{-4} as learning rate and as loss the sparse categorical cross entropy.

It can be noticed that thanks to this technique we were able to reach a significative gain in all the performance metrics used, both global and F1 related to single classes. The only exception is for the F1 on Atelectasis class where they perform almost the same.

Model	Accuracy	Loss	Roc Auc Score	F1-macro	Inference time
Fused.h5	**0.617647**	**0.971038**	**0.836229**	**0.605598**	**7.426303**
Base.h5	0.574303	1.032782	0.828950	0.566883	10.767316

TABLE 108: GLOBAL PERFORMANCE METRICS OF FUSED AND BASE MODEL.

Model	Atelectasis	Infiltration	No Finding	Effusion
Fused.h5	0.472727	**0.593625**	**0.647815**	**0.708223**
Base.h5	**0.477745**	0.545833	0.569106	0.674847

TABLE 119: F1 ON SINGLE CLASSES OF FUSED AND BASE MODEL.

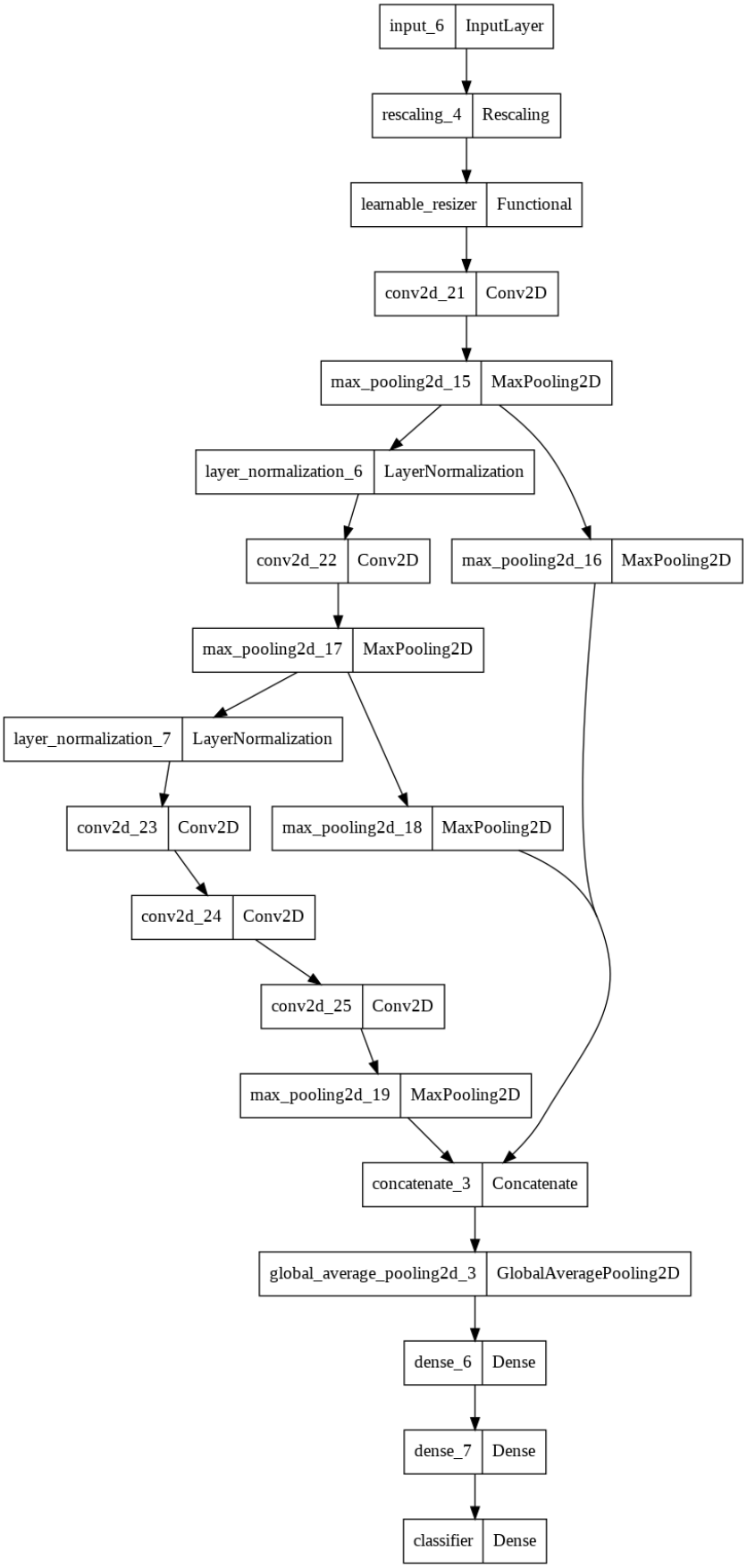


FIGURE *410:* PLOT OF FUSED ARCHITECTURE.

Layer (type)	Output Shape	Param #	Connected to
input_6 (InputLayer)	[(None, 250, 250, 3)]	0	[]
rescaling_4 (Rescaling)	(None, 250, 250, 3)	0	['input_6[0][0]']
learnable_resizer (Functional)	(None, 227, 227, 3)	12163	['rescaling_4[0][0]']
conv2d_21 (Conv2D)	(None, 55, 55, 96)	34944	['learnable_resizer[0][0]']
max_pooling2d_15 (MaxPooling2D)	(None, 27, 27, 96)	0	['conv2d_21[0][0]']
layer_normalization_6 (LayerNormalization)	(None, 27, 27, 96)	192	['max_pooling2d_15[0][0]']
conv2d_22 (Conv2D)	(None, 27, 27, 256)	614656	['layer_normalization_6[0][0]']
max_pooling2d_17 (MaxPooling2D)	(None, 13, 13, 256)	0	['conv2d_22[0][0]']
layer_normalization_7 (LayerNormalization)	(None, 13, 13, 256)	512	['max_pooling2d_17[0][0]']
conv2d_23 (Conv2D)	(None, 13, 13, 384)	885120	['layer_normalization_7[0][0]']
conv2d_24 (Conv2D)	(None, 13, 13, 384)	1327488	['conv2d_23[0][0]']
conv2d_25 (Conv2D)	(None, 13, 13, 256)	884992	['conv2d_24[0][0]']
max_pooling2d_16 (MaxPooling2D)	(None, 6, 6, 96)	0	['max_pooling2d_15[0][0]']
max_pooling2d_18 (MaxPooling2D)	(None, 6, 6, 256)	0	['max_pooling2d_17[0][0]']
max_pooling2d_19 (MaxPooling2D)	(None, 6, 6, 256)	0	['conv2d_25[0][0]']
concatenate_3 (Concatenate)	(None, 6, 6, 608)	0	['max_pooling2d_16[0][0]', 'max_pooling2d_18[0][0]', 'max_pooling2d_19[0][0]']
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 608)	0	['concatenate_3[0][0]']
dense_6 (Dense)	(None, 2048)	1247232	['global_average_pooling2d_3[0][0]']
dense_7 (Dense)	(None, 2048)	4196352	['dense_6[0][0]']
classifier (Dense)	(None, 4)	8196	['dense_7[0][0]']

Total params: 9,211,847
Trainable params: 9,211,719
Non-trainable params: 128

FIGURE 511: SUMMARY OF FUSED MODEL.

TRIAL AND ERROR TO GET THE FUSED MODEL

A trial and error approach was used to search for the optimal hyperparameters for training the model. Several possibilities have been considered, in particular the main ones include:

1) The **pre-processing of the images**, described in paragraph 2.5 (indicated as *prep* in the filename).
2) The presence of the **learnable resizer**, described in paragraph 2.6 (indicated as *no/yes_rs* in the filename).
3) **Data augmentation** in order to reduce the effects of overfitting by presenting every epoch new images, obtained by slightly distorting the original ones, in particular by applying RandomFlip, RandomRotation and RandomContrast (indicated as *no/yes_da* in the filename).
4) The **number of neurons** in the final dense layers, considering the possibility of having 2048 and 4096, in order to reduce the capacity of the network as with too many learnable parameters it can simply store training data and act as a lookup table, so just indexing instead of understanding the underlying function.

The last two options, along with early stopping, have been tried to reduce overfitting, which turns out to be quite consistent in the first versions tested.

Trying to fix only one hyperparameter at a time looking for the best would be insufficient, as hyperparameters **can be correlated with each other** and therefore, with a good probability, there is a combination of them that leads to a better result.

For this reason several combination of the previous possibilities were tried, but only a few, among the most significative ones, are shown in *Table 12* and in *Table 13*, where can be noticed that the final configuration chosen is the overall best model, as it performs better than all the others considering all the metrics, both global and related to single classes.

Model	Accuracy	Loss	Roc Auc Score	F1-macro	Inference time
no_da_no_rs_prep.h5	0.517802	1.148516	0.755353	0.505125	1.038314
no_da_yes_rs_2048.h5	**0.617647**	**0.971031**	**0.836235**	**0.605598**	3.091209
yes_da_yes_rs_4096.h5	0.565015	1.033255	0.810735	0.551248	3.047725
no_da_yes_rs_4096.h5	0.594427	0.996613	0.832780	0.557707	3.084051
yes_da_no_rs_4096.h5	0.531734	1.111113	0.773273	0.508024	**0.952579**
no_da_no_rs_4096.h5	0.526316	1.146247	0.765587	0.485992	1.147568

TABLE 12: GLOBAL PERFORMANCE METRICS OF DIFFERENT TRIAL OF THE FUSED MODEL.

Model	Atelectasis	Infiltration	No Finding	Effusion
no_da_no_rs_prep.h5	0.413428	0.456693	0.524272	0.626109
no_da_yes_rs_2048.h5	**0.472727**	**0.593626**	**0.647815**	**0.708223**
yes_da_yes_rs_4096.h5	0.412639	0.542694	0.580468	0.669192
no_da_yes_rs_4096.h5	0.307305	0.574956	0.642369	0.706199
yes_da_no_rs_4096.h5	0.354212	0.463602	0.576832	0.637450
no_da_no_rs_4096.h5	0.273839	0.454902	0.579365	0.635864

TABLE 1312: F1 ON SINGLE CLASSES OF DIFFERENT TRIAL OF THE FUSED MODEL.

4 Pre-Trained Models

This section describes the results obtained using different pre-trained architectures and strategies.

The pre-trained networks we tested are:

- **VGG16** [1]
- **ResNet152** [2]
- **CheXNet** [3]

For each architecture we tried fitting the model to our task by exploiting both *feature extraction* and *fine tuning*, using a ***trial and error*** approach for *Hyperparameters Optimization*.

During the *feature extraction* phase, we conduct experiments to compare several transfer learning configurations. The configuration we use is the variation in number of employed *Dense layers*, number of employed *Neurons* in each dense layer, learning rate, dropout rate, and L2 regularization rate. Each configuration run for 20 *epochs* on *mini-batches* of size 32 and the best validation accuracy is recorded for comparison. All configurations are optimized using *Adam* or *RMSprop* optimization, with *ReLU* activation for intermediate Dense layers and *Softmax* activation for the output layer, and with *categorical cross-entropy* as loss function. We also used *Global Average Pooling layers*, *Batch Normalization layers* (trying to accelerate and stabilize the training [7]) and other techniques described in *chapter 3* such as *ReduceLROnPlateau*.

All the *"Feature Extraction"* experiments use *learning rate of 1e-3*, as recommended in [11].

In the following paragraphs, we are going to only present the flow of models that guided us towards the final model with the best performance. All other tested models can be found in pre-trained related notebooks.

To speed up the training process, we exploited *feature reuse* when possible, running the convolutional base over our dataset just once and using its output as input for many MLPs. This was only possible for models which did not exploit data augmentation.

4.1VGG16

The original VGG16 architecture is shown in the *Figure* below. The network has *14.714.688 parameters* and expects an input of fixed dimension of *224x224x3* and generates an output of *1000* probabilities (since it's been previously trained on the *ImageNet [4]* database).

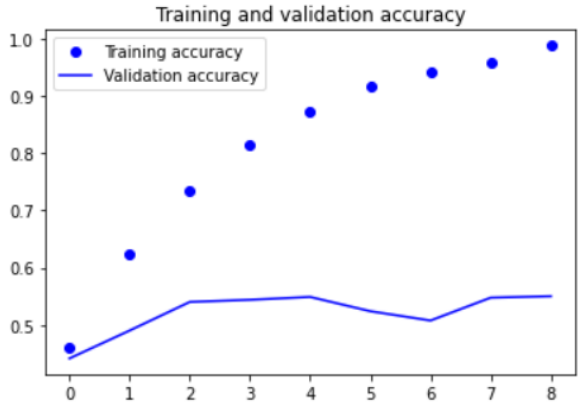
4.1.1 Feature Extraction

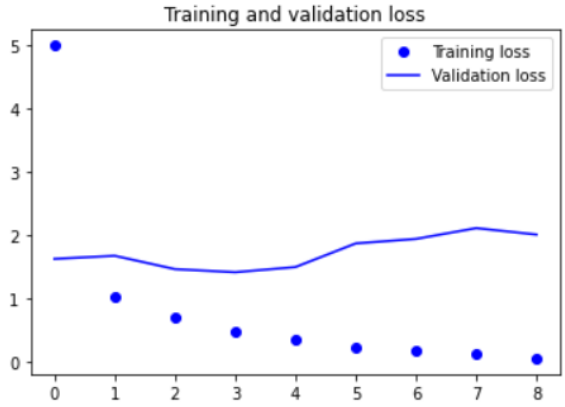
Since our dataset differs greatly wrt ImageNet dataset, we didn't expect to achieve good results during this phase, and so it was. What we did was try to find a model cable to better generalize on the validation set without overfitting and obtaining satisfactory results.

In the end, the configuration with a single Dense layer of 256 neurons turned out to be the best performing. We will now present the results that led us to obtain the best final configuration.

4.1.1.1 Dense 256 Layer

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
3	0.4854	81.43	1.41268	54.37



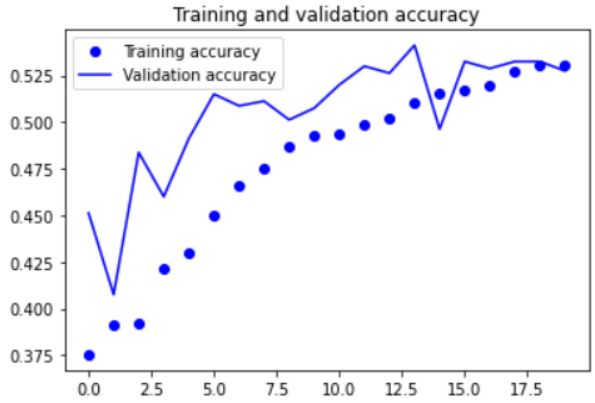


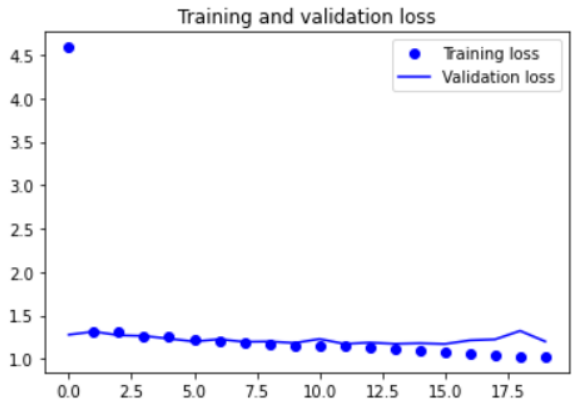

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 7, 7, 512)]       0

flatten (Flatten)            (None, 25088)             0

dense (Dense)                (None, 256)               6422784

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 6,423,812
Trainable params: 6,423,812
Non-trainable params: 0
```

As can be seen, the network starts to overfit very quickly and the obtained accuracy is not satisfactory, hence some *regularization methods* are needed.

4.1.1.2 Dense 256 + Dropout 0.5

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
15	1.0792	51.72	1.1699	53.25





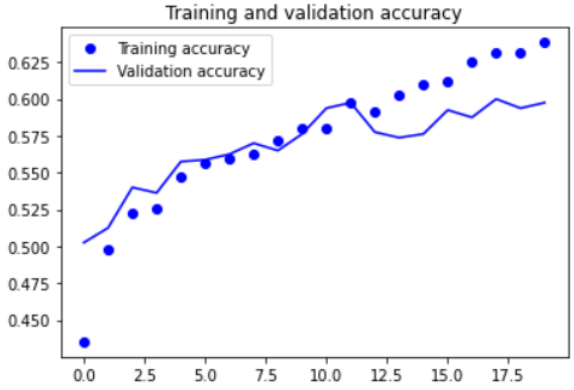
Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 7, 7, 512)]	0
flatten_3 (Flatten)	(None, 25088)	0
dense_6 (Dense)	(None, 256)	6422784
dropout_3 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 4)	1028

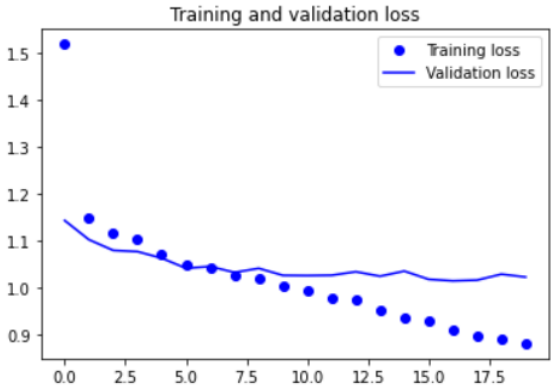
Total params: 6,423,812
Trainable params: 6,423,812
Non-trainable params: 0

As expected, dropout mitigated the magnitude of overfitting, however the network performed slightly worst on accuracy, underfitting the training data. This may be due to the fact that, adding dropout, we reduced the capacity of the network. Still, it shown good improvements in the validation loss value.

4.1.1.3 Global Average Pooling 2D + Dense 256 + Dropout 0.5

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
16	0.91	62.51	1.0143	59.25





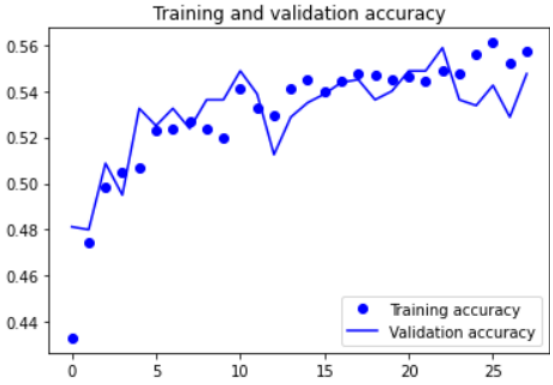

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 7, 7, 512)]       0

global_average_pooling2d (G  (None, 512)               0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               131328

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 132,356
Trainable params: 132,356
Non-trainable params: 0
_________________________________________________________________
```

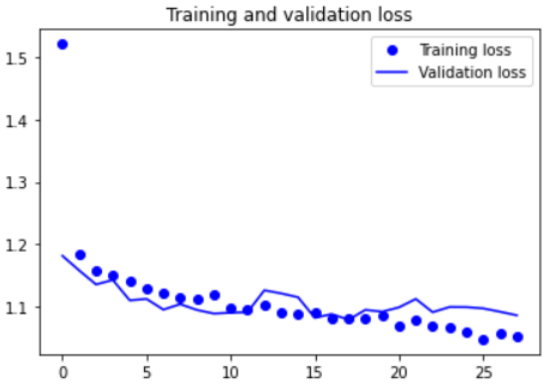
Adding a *Global Average Pooling layer* to **reduce the spatial extent of the feature map of the last convolutional layer**, we were able to both decrease the validation loss and increase the accuracy, but the validation loss curve starts diverging from the training fairly quickly, overfitting.It can be caused by an inability of the network to generalize the training data.

We tried to increment the capacity of the network without any significant increase, so we decided to apply *Data Augmentation*.

4.1.1.4 Data Augmentation + Global Avg. Pooling 2D + Dense 256 + Dropout 0.5

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
18	1.0809	54.78	1.0796	54.50






```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

sequential (Sequential)      (None, 224, 224, 3)       0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

vgg16 (Functional)           (None, 7, 7, 512)         14714688

global_average_pooling2d (G  (None, 512)               0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               131328

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 14,847,044
Trainable params: 132,356
Non-trainable params: 14,714,688
_________________________________________________________________
```

We then tried to add *Data Augmentation* to the model, reducing overfitting but obtaining worst results in terms of validation loss and accuracy wrt the previous model, underfitting the training. The network still seems to be underfitting the data.

At this point, we decided to try fine tuning the network.

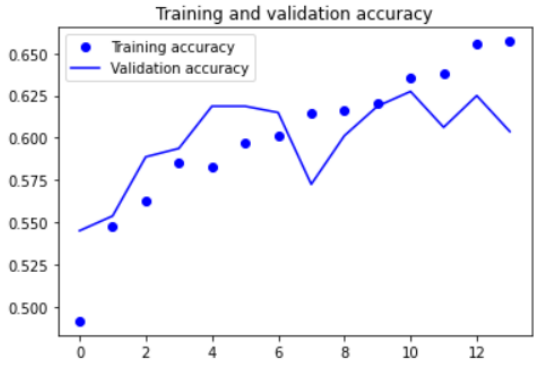
4.1.2 Fine Tuning

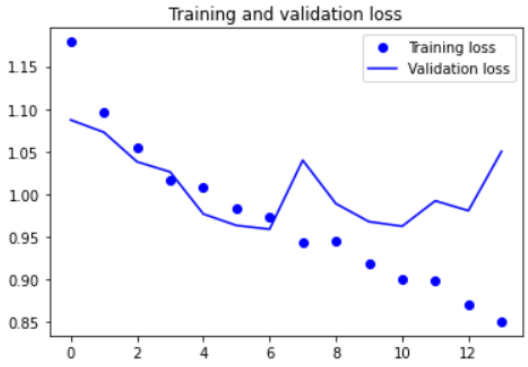
We fine tuned different models, unfreezing at different layers. Since the highest layers of the network have most likely learned to recognize features that are more specific to the original ImageNet dataset, we thought that performing fine tuning could have be beneficial to the performance of the model.

In the end, we were able to slightly increase performance, but not satisfactorily. Even unfreezing few layers (out of concern of increasing too much the number of layers and consequently the risk of overfitting) gave little improvements.

4.1.2.1 Unfreeze last Block (3 Convolutional layers, block5_conv1) from 5.1.1.4

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
6	0.973	60.16	0.9591	61.50






```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

sequential (Sequential)      (None, 224, 224, 3)       0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

vgg16 (Functional)           (None, 7, 7, 512)         14714688

global_average_pooling2d (G  (None, 512)               0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               131328

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 14,847,044
Trainable params: 7,211,780
Non-trainable params: 7,635,264
_________________________________________________________________
```

As can be seen, the model quickly went into overfitting even if we unfreeze just 3 layers.

We tried adding more regularization, but every model ended up underfitting the training.

4.1.3 Gathering More Data

We tried to increase our training data in order to fight overfitting and then tested the best models obtained previously over the new dataset.

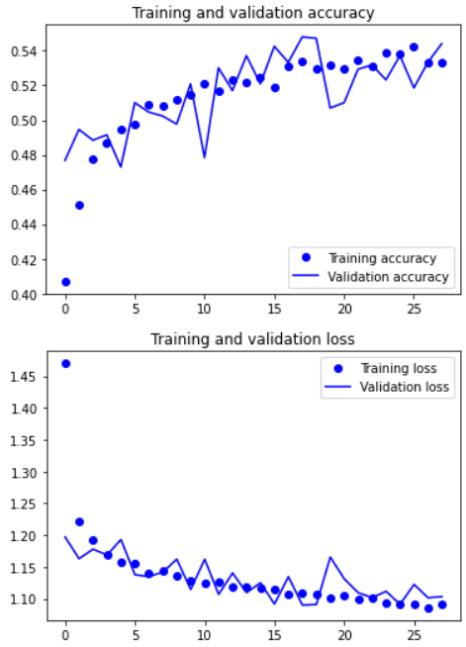
4.1.3.1 Data Augmentation + Global Avg. Pooling 2D + Dense 256 + Dropout 0.5

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
17	1.1091	53.37	1.0901	54.80

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

sequential (Sequential)      (None, 224, 224, 3)       0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

vgg16 (Functional)           (None, 7, 7, 512)         14714688

global_average_pooling2d (G  (None, 512)               0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               131328

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 14,847,044
Trainable params: 132,356
Non-trainable params: 14,714,688
_________________________________________________________________
```

4.1.3.2 Unfreeze last Block (3 Convolutional layers, block5_conv1) from 5.1.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
11	0.8779	64.81	0.9364	61.76



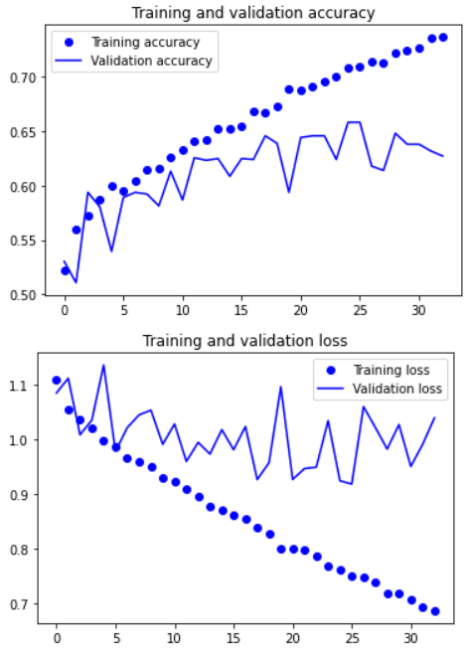

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

sequential (Sequential)      (None, 224, 224, 3)       0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

vgg16 (Functional)           (None, 7, 7, 512)         14714688

global_average_pooling2d (G  (None, 512)               0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               131328

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 14,847,044
Trainable params: 7,211,780
Non-trainable params: 7,635,264
_________________________________________________________________
```

Even using the new dataset, the performance of the model on the validation didn't get much better. The validation loss follows the training loss pretty good, but they reach a plateau from which they are unable to advance even using *ReduceLROPlateau*.

Since our dataset is very different wrt ImageNet, we decided to try to unfreeze more layers.

4.1.3.3 Unfreeze last 3 Blocks (9 Convolutional layers, block3_conv1) from 5.1.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
25	0.7514	70.87	0.9189	65.79



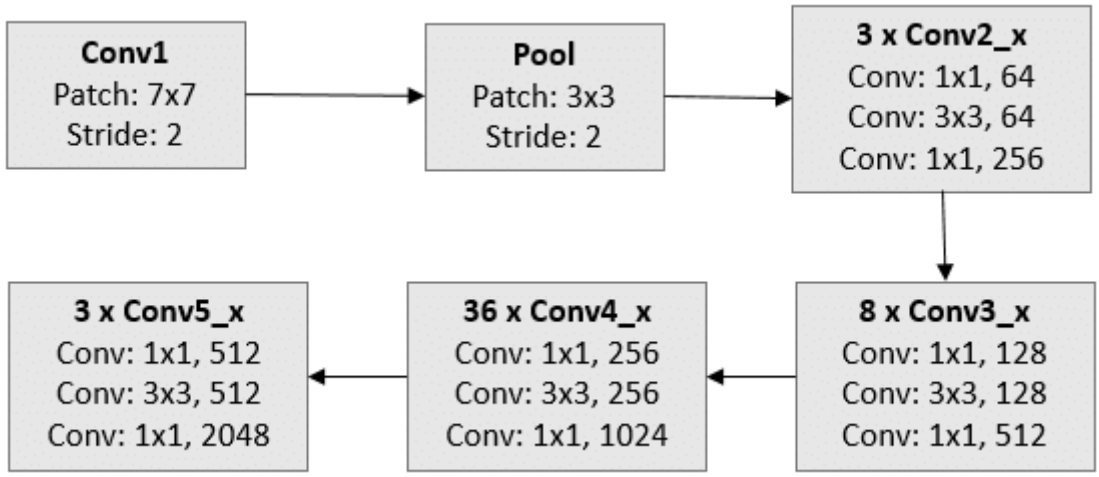

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

sequential (Sequential)      (None, 224, 224, 3)       0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

vgg16 (Functional)           (None, 7, 7, 512)         14714688

global_average_pooling2d (G  (None, 512)               0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               131328

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 14,847,044
Trainable params: 14,586,884
Non-trainable params: 260,160
_________________________________________________________________
```

Leveraging the fact that we had more images, we were able to fine tune more layers without immediately overfitting, obtaining a more significant increase in performance.

This is the best model we were able to obtain using VGG16.

4.2 ResNet152

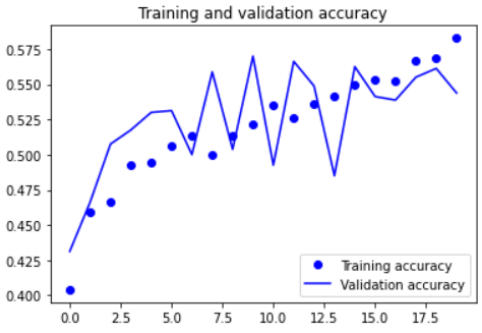
The original ResNet152 architecture is shown in the *Figure* below. It is a much deeper network wrt VGG16 (*58.370.944 parameters*) and expects an input of fixed dimension of *224x224x3* and generates an output of *1000* probabilities (since it's been previously trained on the *ImageNet [4]* database).

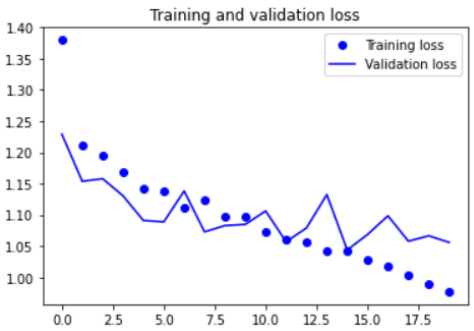


4.2.1 Feature Extraction

4.2.1.1 Global Avg. Pooling 2D + Dense 256 + Dropout 0.5

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
14	1.0414	54.96	1.0438	56.25



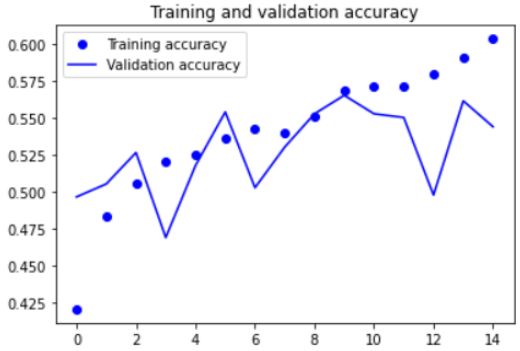


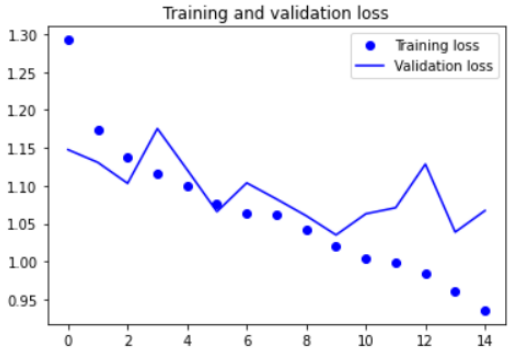

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 256)               524544

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 4)                 1028

=================================================================
Total params: 58,896,516
Trainable params: 525,572
Non-trainable params: 58,370,944
_________________________________________________________________
```

Even with dropout and a pooling layer, the network tends to overfit. We thought the model might be too complex and tried to reduce the network capacity accordingly.

4.2.1.2 Global Avg. Pooling 2D + Dense 64 + Dropout 0.25

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
9	1.0199	56.85	1.0350	56.50



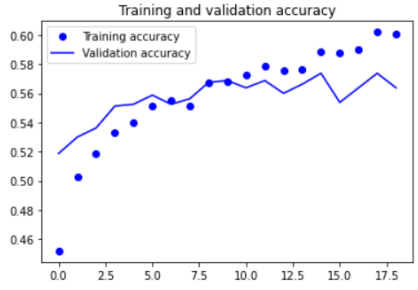


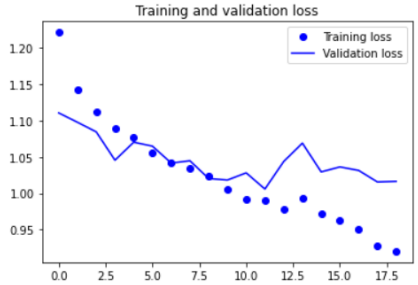

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 131,396
Non-trainable params: 58,370,944
_________________________________________________________________
```

By reducing the network capacity, we got a slight performance improvement but nothing substantial despite still having overfitting problems. We then tried to add *Data Augmentation*.

4.2.1.3 Data Augmentation + Global Avg. Pooling 2D + Dense 64 + Dropout 0.33

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
11	0.9895	57.89	1.0058	56.88






```
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem_1   (None, 224, 224, 3)       0
(SlicingOpLambda)

tf.nn.bias_add_1 (TFOpLambd  (None, 224, 224, 3)       0
a)

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d_1   (None, 2048)              0
(GlobalAveragePooling2D)

dense_2 (Dense)              (None, 64)                131136

dropout_1 (Dropout)          (None, 64)                0

dense_3 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 131,396
Non-trainable params: 58,370,944
_________________________________________________________________
```

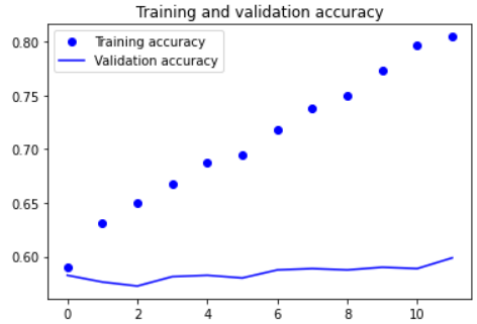
Up to a certain point, the loss curve on validation correctly follows the training curve, then there starts to be overfitting. We had a slight improvement wrt the previous model but still nothing satisfactory.

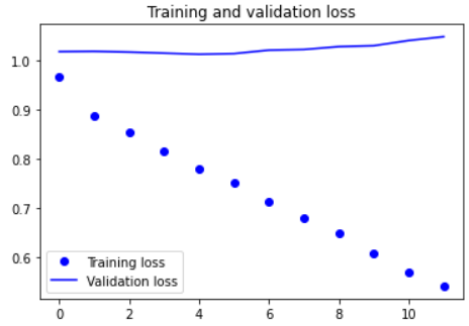
Several experiments have been performed to fight this behaviour, but then we decided to switch to fine tuning.

4.2.2 Fine Tuning

4.2.2.1 Unfreeze last Block (6 layers, conv5_block3_1_conv) from 5.2.1.2

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
4	0.7804	68.76	1.0121	58.25






```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)      0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 4,597,060
Non-trainable params: 53,905,280
_________________________________________________________________
```

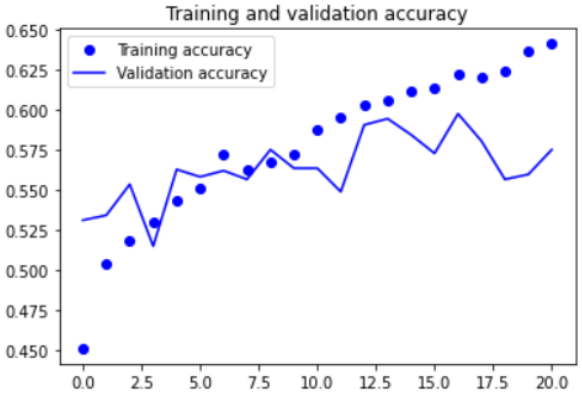
Performance improved wrt the pre-fine-tuned models but the network suffers a severe overfitting.

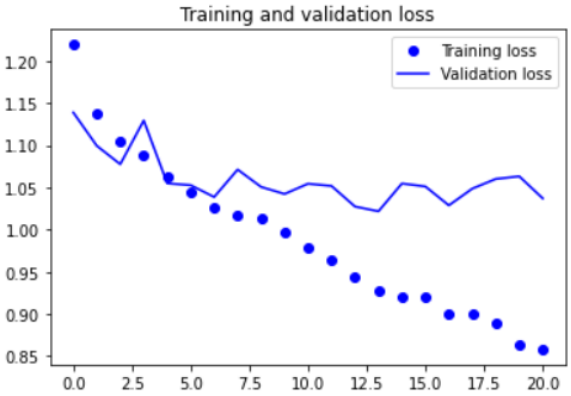
We tried to fight this behaviour both by training different models and unfreezing at different layers (both shallower and deeper), without any improvement. We then decided to test the best models over the new, bigger dataset.

4.2.3 Gathering More Data

4.2.3.1 Global AVG Pooling 2D + Dense 64 + Dropout 0.25

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
12	0.9284	60.62	1.0218	59.44

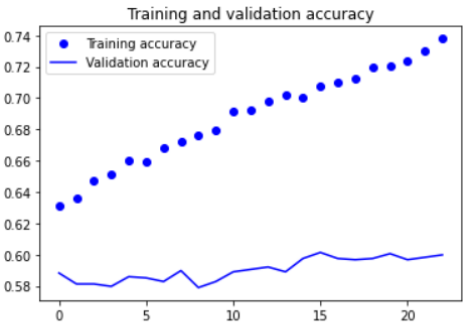


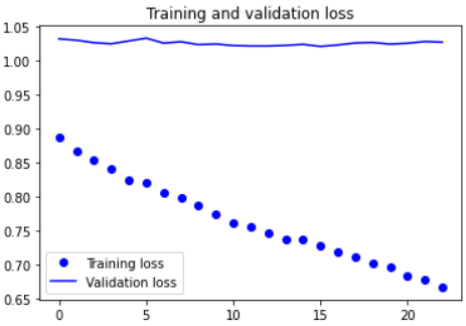



```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 131,396
Non-trainable params: 58,370,944
_________________________________________________________________
```

4.2.3.2 Unfreeze last 2 layers (conv5_block3_3_conv) from 5.2.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
15	0.7269	70.73	1.0201	60.14





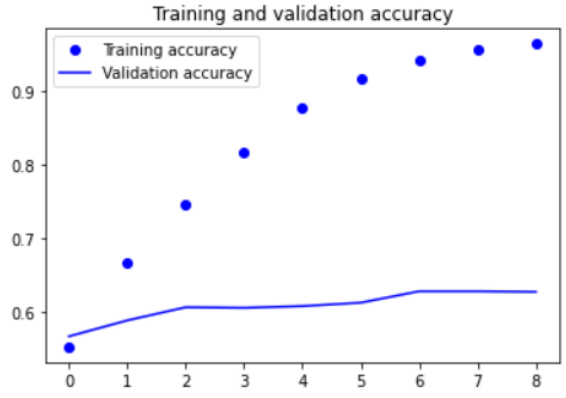

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 1,186,116
Non-trainable params: 57,316,224
_________________________________________________________________
```

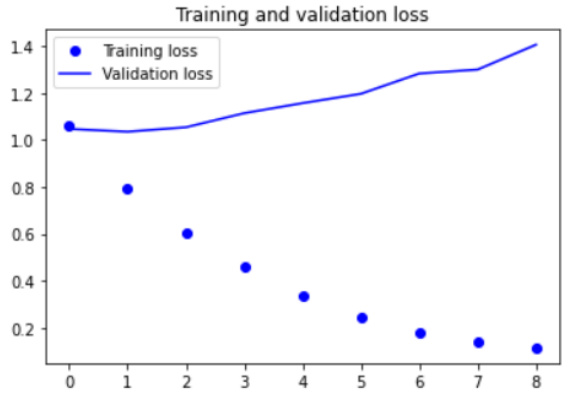
Even using the new dataset, we still have the same behaviour where the validation overfits immediately. It is not able to generalize the training set and so the validation immedately reaches a plateau.

We then tried to exploit the greater amount of data available to be able to fine tune deeper layers, obtaining subsequent unsatisfactory results.

4.2.3.3 Unfreeze last 50 layers (conv4_block32_1_conv) from 5.2.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
2	0.7917	66.70	1.0353	58.82

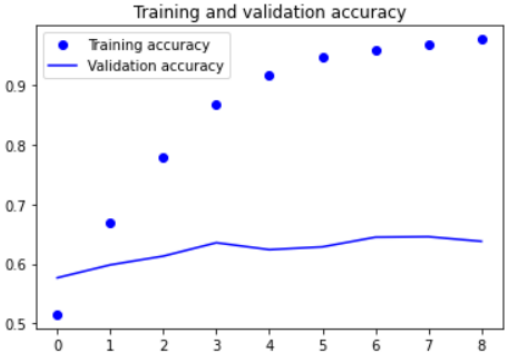


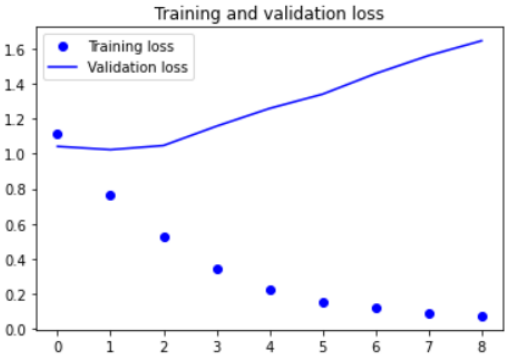



```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 20,700,996
Non-trainable params: 37,801,344
_________________________________________________________________
```

4.2.3.4 Unfreeze last 150 layers from 5.2.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
2	0.7638	66.82	1.02131	59.83






```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem (S  (None, 224, 224, 3)       0
licingOpLambda)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

resnet152 (Functional)       (None, 7, 7, 2048)        58370944

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 58,502,340
Trainable params: 39,456,324
Non-trainable params: 19,046,016
_________________________________________________________________
```

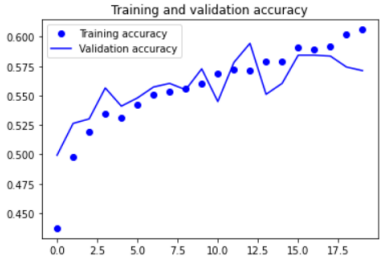
4.2.4 ResNet50

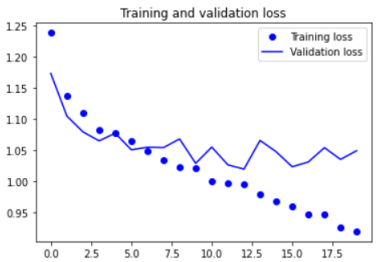
[10] states that "much deeper neural networks should not really be used to classify biomedical images. Much deeper neural networks tend to capture more abstract features and hence, tend to overlook the minor variances between similar images that may be the key to diagnosing different diseases". Following this reasoning, we tested a shallower version of ResNet.

ResNet50 *23.587.712 parameters*, while ResNet152 has *58.370.944 parameters*.

4.2.4.1 Global AVG Pooling 2D + Dense 64 + Dropout 0.25

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
12	0.9954	57.17	1.0192	59.44

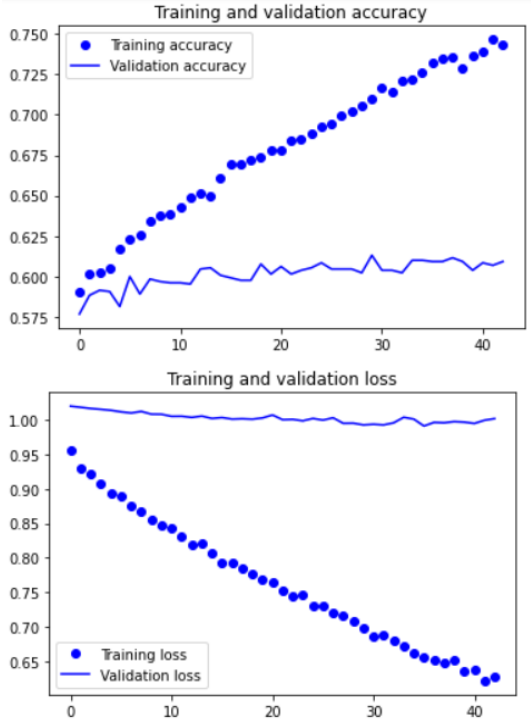





```
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem_1   (None, 224, 224, 3)       0
(SlicingOpLambda)

tf.nn.bias_add_1 (TFOpLambd  (None, 224, 224, 3)       0
a)

resnet50 (Functional)        (None, 7, 7, 2048)        23587712

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 23,719,108
Trainable params: 131,396
Non-trainable params: 23,587,712
_________________________________________________________________
```

4.2.4.2 Unfreeze last 2 layers (conv5_block3_3_conv) from 5.2.4.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
35	0.6565	73.24	0.9911	60.91




```
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 224, 224, 3)]     0

tf.__operators__.getitem_1   (None, 224, 224, 3)       0
(SlicingOpLambda)

tf.nn.bias_add_1 (TFOpLambd  (None, 224, 224, 3)       0
a)

resnet50 (Functional)        (None, 7, 7, 2048)        23587712

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                131136

dropout (Dropout)            (None, 64)                0

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 23,719,108
Trainable params: 1,186,116
Non-trainable params: 22,532,992
_________________________________________________________________
```

Using ResNet50 we were able to obtain better performance both in validation loss and accuracy just by taking the best model found for ResNet152 and training it over ResNet50, but we still have overfitting. The network still isn't capable to generalize.

We decided to stop here with the tests on ResNet, as the results obtained were not satisfactory and we were unable to reach those obtained using VGG16 (which had just *14.714.688 parameters*).

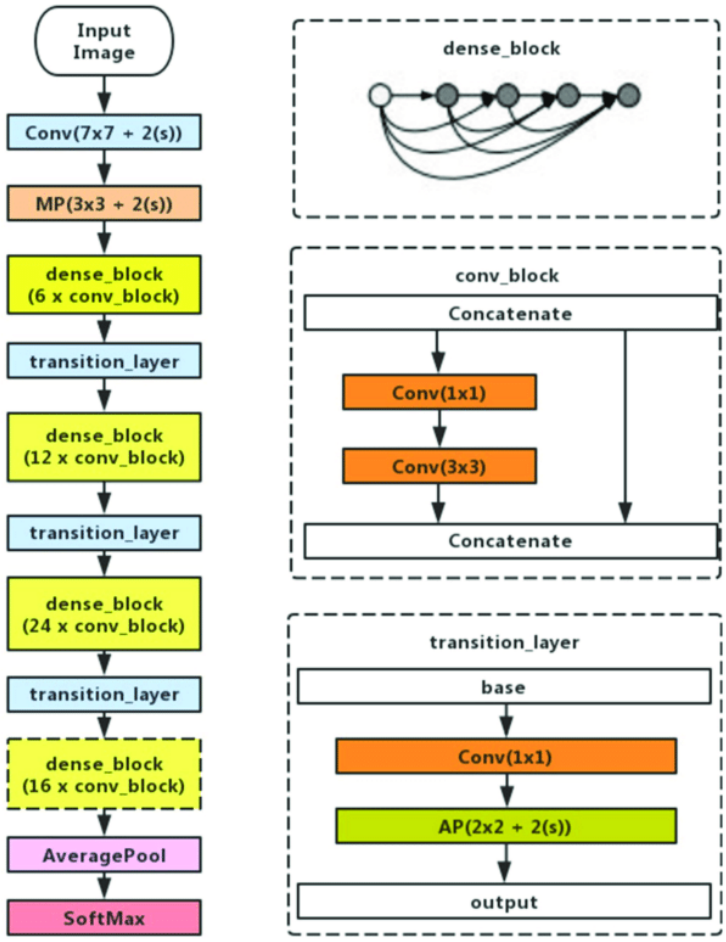
4.3 CheXNet

Since VGG16 and ResNet pretrained over ImageNet weren't able to reach satisfactory performance, we decided to look for a different model pretrained on a dataset which was more similar to ours.

CheXNet is a 121 layers DenseNet model with *7.037.504 parameters*, which layers are grouped into 4 dense blocks. This model has been pre-trained on Chest-X-Ray14 dataset and was acquired from CheXNet reimplementation project of Machine Intelligence Lab, Institute of Computer Science & Technology, Peking University [12].

CheXNet was initially developed to detect pneumonia from chest X-rays and has been extended to detect all 14 diseases present in Chest-X-Ray14 dataset, obtaining state of the art results.

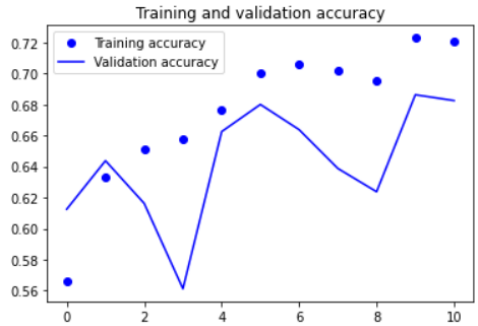
In the original CheXNet paper, they "*downscale images to 224x224 and augument the training data with random horizontal flipping*".

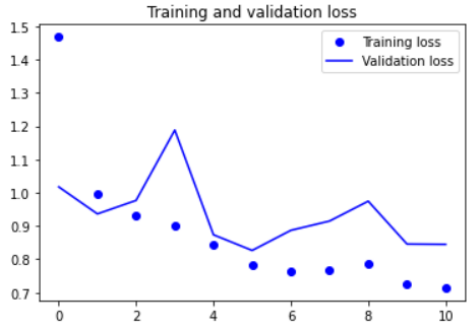


4.3.1 Feature Extraction

4.3.1.1 Dense 64

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
5	0.7817	70.00	0.8264	68.00





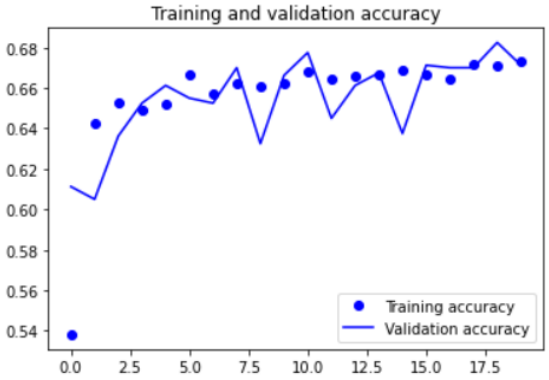

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)       0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)       0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

flatten (Flatten)            (None, 50176)             0

dense (Dense)                (None, 64)                3211328

dense_1 (Dense)              (None, 4)                 260

=================================================================
Total params: 10,249,092
Trainable params: 3,211,588
Non-trainable params: 7,037,504
_________________________________________________________________
```

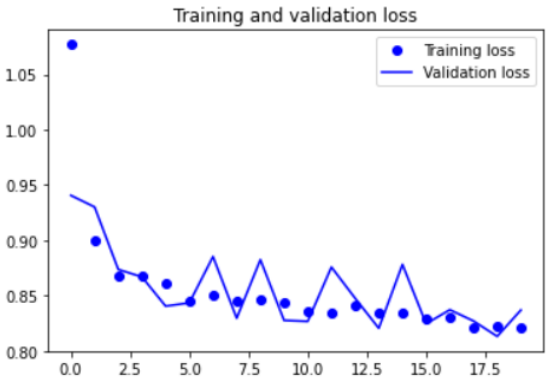
Performance already overcome those obtained using VGG16 and ResNet, but the network ended up overfitting.

As can be seen from the model summary on the right, we have 7.037.504 parameters from the CheXNet and 3.211.588 from the Dense layer on top. This is due to the output from CheXNet, which has shape (7, 7, 1024). We then decided to reduce the complexity of the network by pooling the output from CheXNet.

4.3.1.2 Global AVG Pooling 2D + Dense 64

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
18	0.8255	67.09	0.8131	68.25





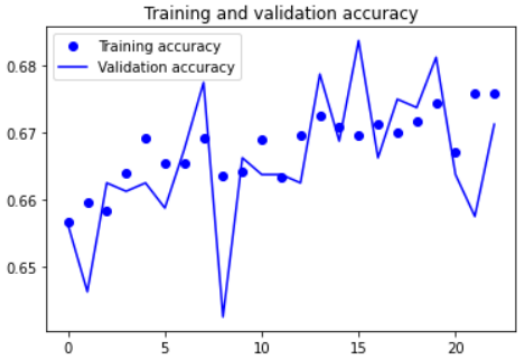

```
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv_2 (TFOpLamb  (None, 224, 224, 3)       0
da)

tf.nn.bias_add_1 (TFOpLambd  (None, 224, 224, 3)       0
a)

tf.math.truediv_3 (TFOpLamb  (None, 224, 224, 3)       0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d_1   (None, 1024)              0
(GlobalAveragePooling2D)

dense_2 (Dense)              (None, 64)                65600

dense_3 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,103,364
Trainable params: 65,860
Non-trainable params: 7,037,504
_________________________________________________________________
```

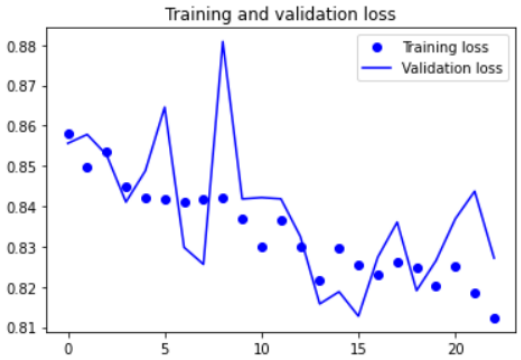
We were able to reduce overfitting and improve the performance of the network while using significantly less parameters. This new simpler model seems now to be underfitting the training, reaching a plateau.

We then tried different configurations, increasing the number of layers and of neurons for layers.

4.3.1.3 Global AVG Pooling 2D + two Dense 64

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
15	0.8255	66.95	0.8127	68.37






```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)       0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)       0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d (G  (None, 1024)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                65600

dense_1 (Dense)              (None, 64)                4160

dense_2 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,107,524
Trainable params: 70,020
Non-trainable params: 7,037,504
```

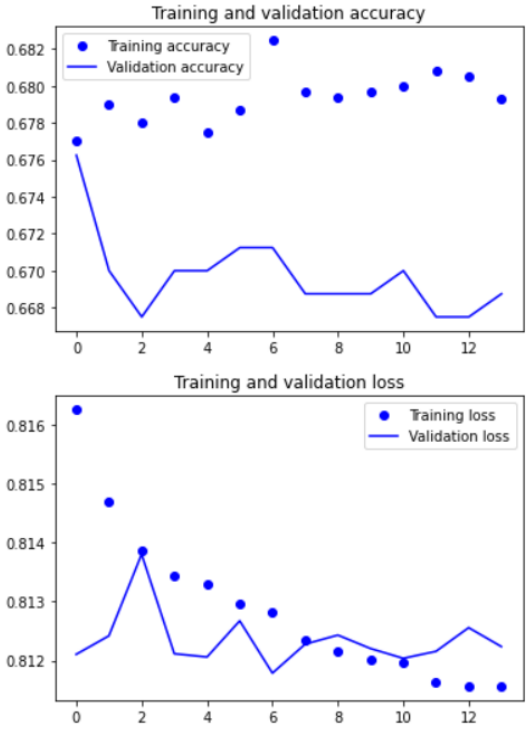
Even though the validation curves are noisy, we can see how the model doesn't seem to reach a plateau as it did in the previous network.

We tried a lot of other settings (they can be found in the *Pre-trained_CheXNet.ipynb* notebook), but in the end we weren't able to obtain any substantial increase in performance, so we decided to try fine tuning.

4.3.2 Fine Tuning

4.3.2.1 Unfreeze last 5 layers (conv5_block15_2_conv) from 5.3.1.3

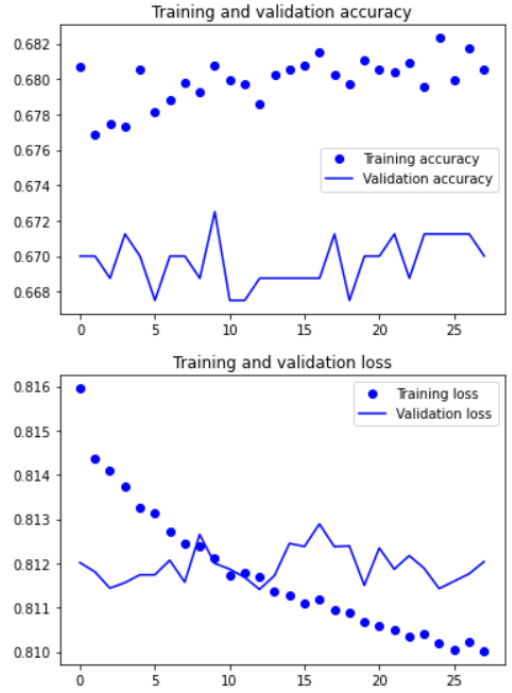
Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
6	0.8128	68.25	0.8118	67.12




```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)      0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)      0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d (G  (None, 1024)             0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                65600

dense_1 (Dense)              (None, 64)                4160

dense_2 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,107,524
Trainable params: 238,148
Non-trainable params: 6,869,376
_________________________________________________________________
```

4.3.2.2 Unfreeze last 9 layers (conv5_block15_0_bn) from 5.3.1.3

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
12	0.8117	67.86	0.8114	66.87




```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)      0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)      0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d (G  (None, 1024)             0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                65600

dense_1 (Dense)              (None, 64)                4160

dense_2 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,107,524
Trainable params: 400,068
Non-trainable params: 6,707,456
_________________________________________________________________
```

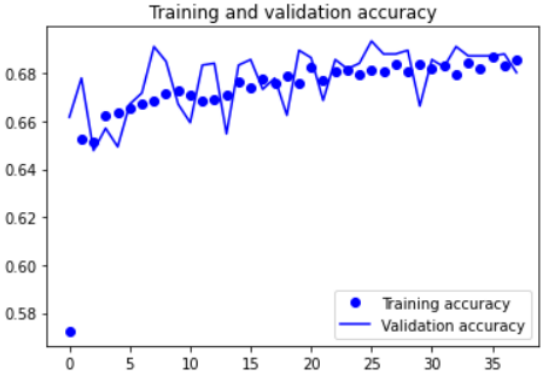
In both the shown fine tuned models can be seen that the network is overfitting; while the training loss curve continues to decrease, the validation loss reaches a plateau.

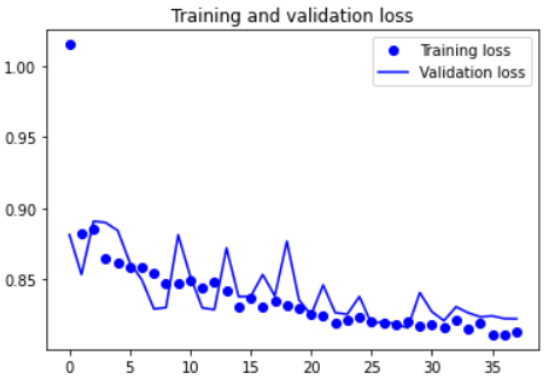
To overcome this overfitting, we tried to use the increased dataset.

4.3.3 Gathering More Data

4.3.3.1 Global AVG Pooling 2D + 2 Dense 64

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
28	0.8196	68.10	0.8153	68.96

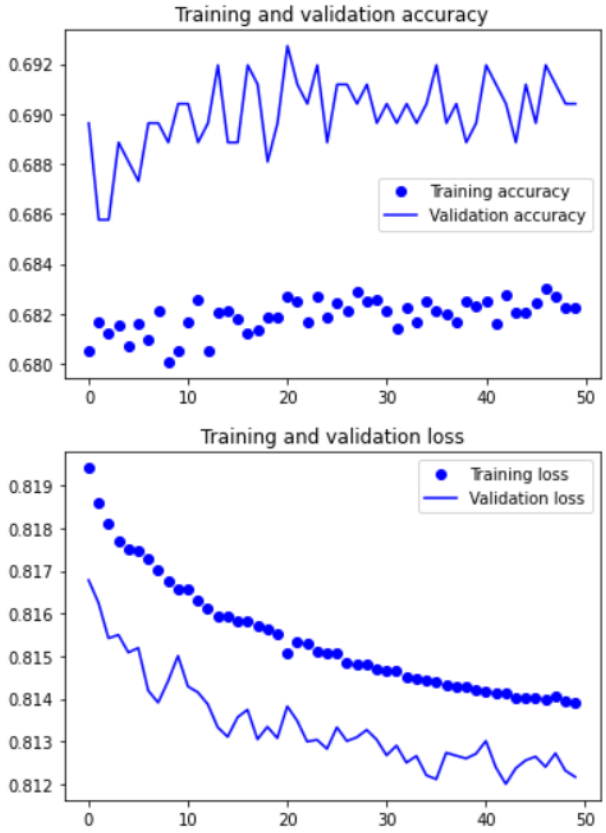





```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)      0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)      0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d (G  (None, 1024)             0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                65600

dense_1 (Dense)              (None, 64)                4160

dense_2 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,107,524
Trainable params: 70,020
Non-trainable params: 7,037,504
_________________________________________________________________
```

4.3.3.2 Unfreeze last 5 layers (conv5_block15_2_conv) from 5.3.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
28	0.8196	68.10	0.8153	68.96

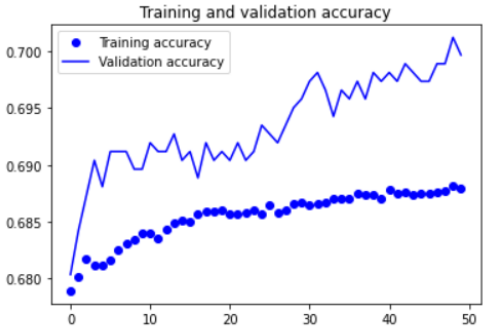


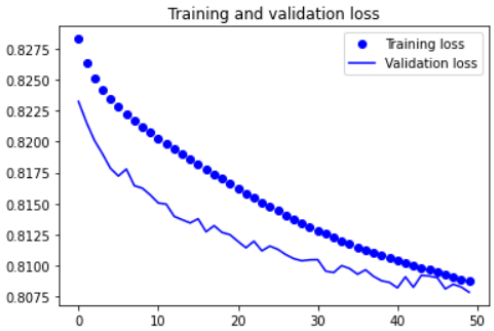

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)       0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)       0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)       0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d (G  (None, 1024)              0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                65600

dense_1 (Dense)              (None, 64)                4160

dense_2 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,107,524
Trainable params: 238,148
Non-trainable params: 6,869,376
_________________________________________________________________
```

Here we can see strange behavior; the validation loss is always less than the training loss. One possible explanation is that the training set may contain more "hard cases" than the validation set. In particular, since the labels have been mined by an NLP system with an F1-score of 94.4%, and since the difference in number of images in the training and validation sets is considerable, increasing the size of the training set also increases the chance of having uncorrectly classified images. Since the difference is in the order of hundredths of accuracy and loss, we think it can be neglected.

4.3.3.3 Unfreeze last 2 Dense_Blocks (conv4_block1_0_bn) from 5.3.3.1

Best Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
50	0.8088	68.79	0.8078	69.97






```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)      0
)

tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

tf.math.truediv_1 (TFOpLamb  (None, 224, 224, 3)      0
da)

model_1 (Functional)         (None, 7, 7, 1024)        7037504

global_average_pooling2d (G  (None, 1024)             0
lobalAveragePooling2D)

dense (Dense)                (None, 64)                65600

dense_1 (Dense)              (None, 64)                4160

dense_2 (Dense)              (None, 4)                 260

=================================================================
Total params: 7,107,524
Trainable params: 5,594,244
Non-trainable params: 1,513,280
_________________________________________________________________
```

Lastly, we tried to unfreeze a big portion of the network (2 of the 4 Dense Blocks, which are composed of 163 unfreezable layers). The results obtained are the best so fare and the behaviour of the training and validation curves looks promising albeit very slow.

5 Error Analysis

In this paragraph, we are going to exploit the major analysis and technique we have implemented to study the correct and incorrect behavior of the best selected models.

The first operation consists of carrying out a general study on the total amount of misclassified images by all the models with the simultaneous analysis of the whole set of correct classified images by all the models; it has allowed to understand the general behaviour of each network. This kind of study is called 'Error Analysis'.

The second analysis is strictly related to the previous one, but this time we are going to take a deeper look to the behaviour of each different model. This kind of study is called '*Explainability'*.

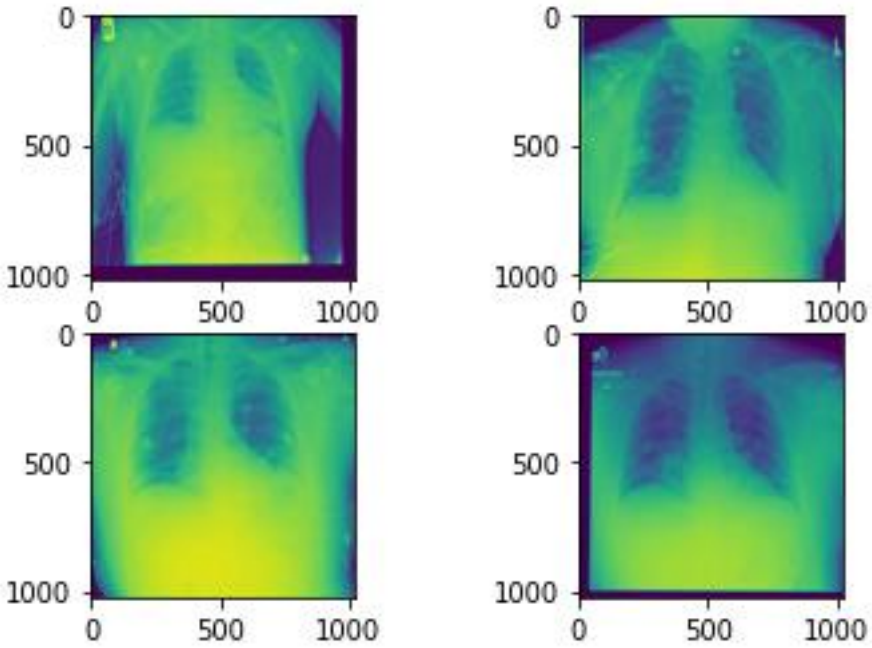
In order to provide a focused analysis and capture what our networks have struggled to recognize, we have carried out an Error Analysis in which we have plotted the images that have been misclassified by all the 5 best classifiers that we have built. The classifiers that have been tested are the following:

- VGG16.
- RESNET152.
- FUSED MODEL.
- CNN FROM SCRATCH.
- CHEXNET.

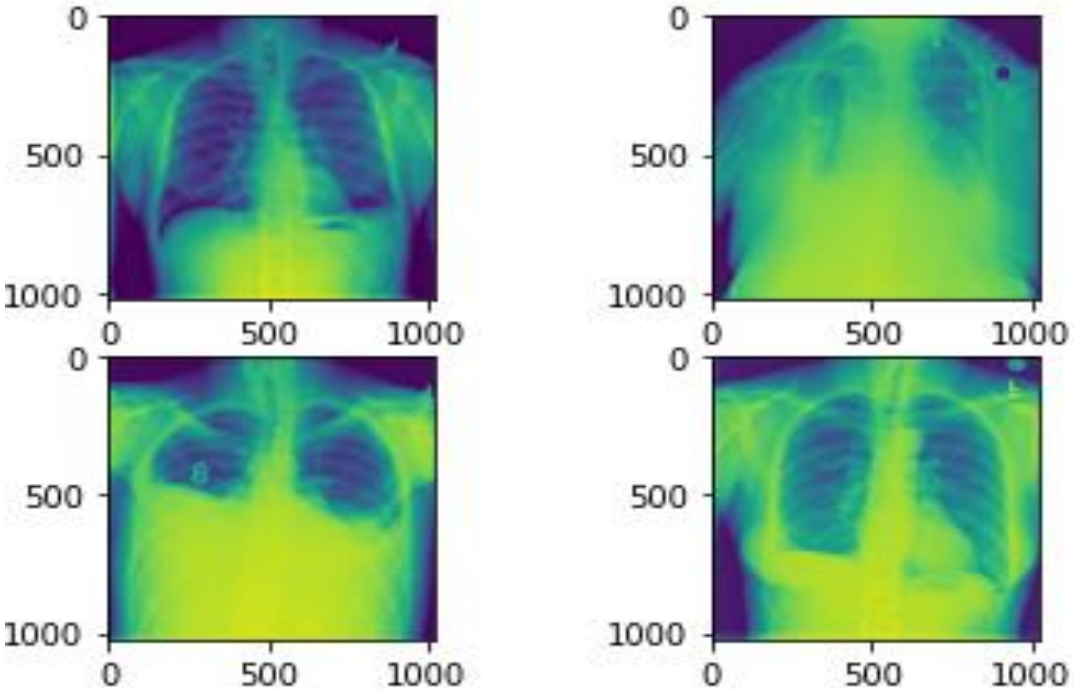
The total number of misclassified images, that is the amount of x-rays that are misclassified by each model, is equal to 176. Here, 4 images for each class are reported.

MISCLASSFIED EXAMPLES:

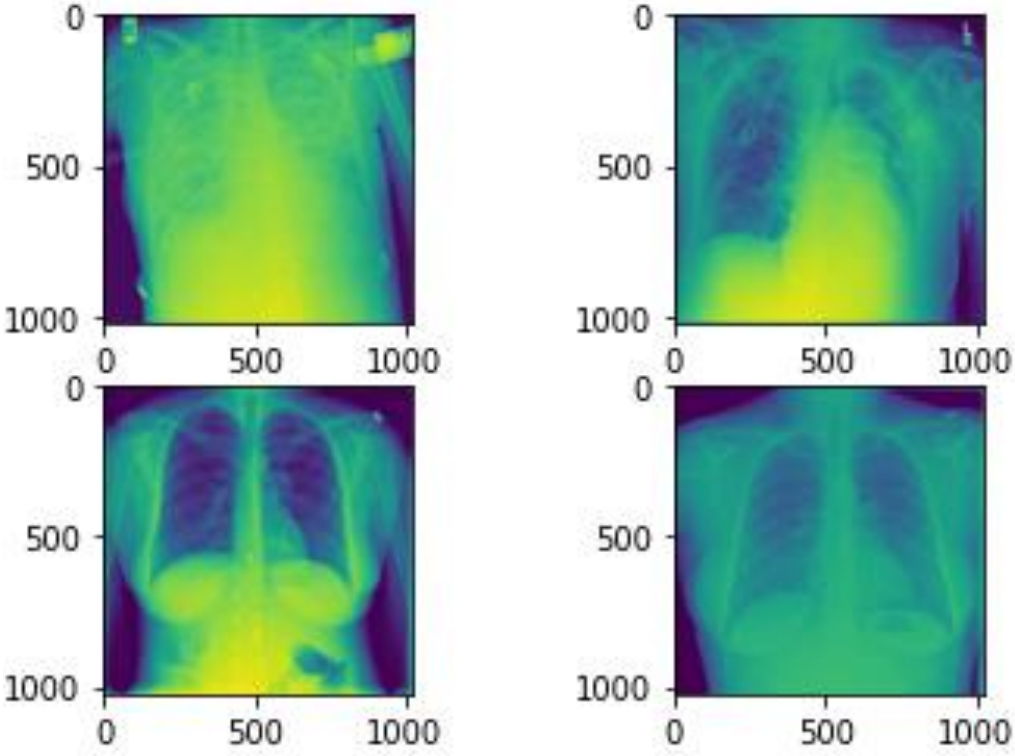
Atelectasis



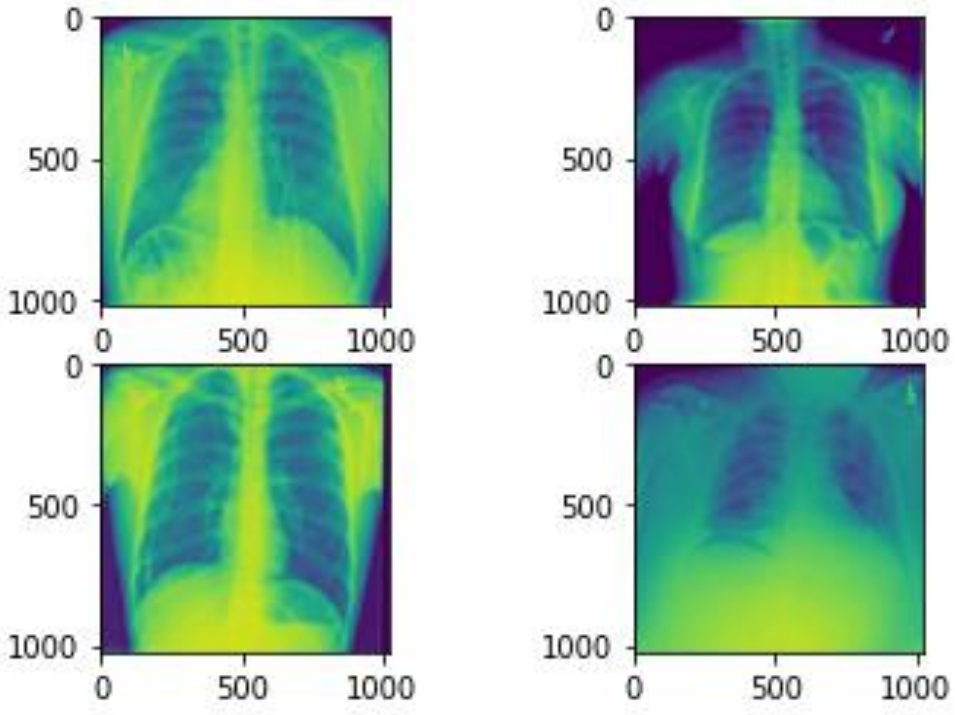
Effusion



Infiltration



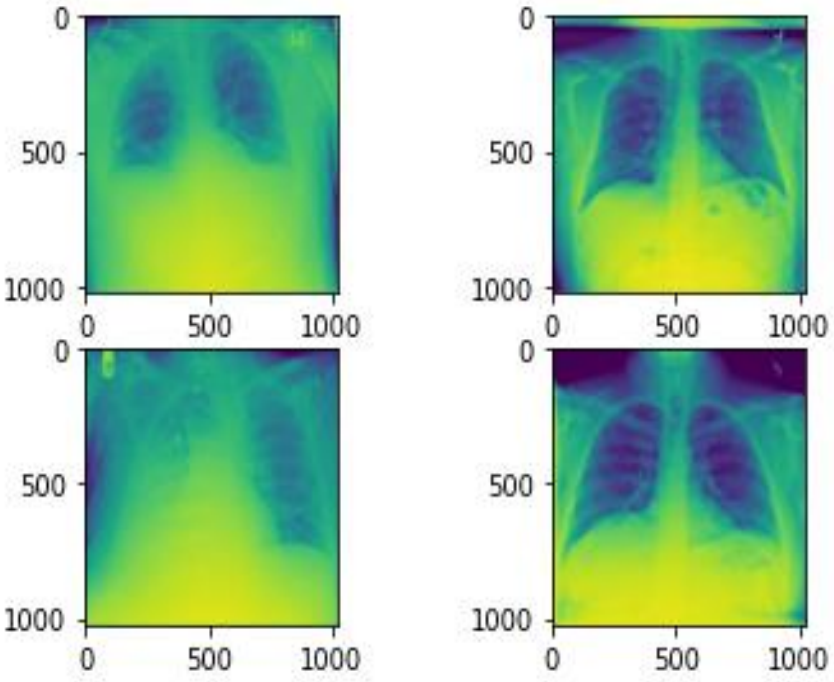
No Finding



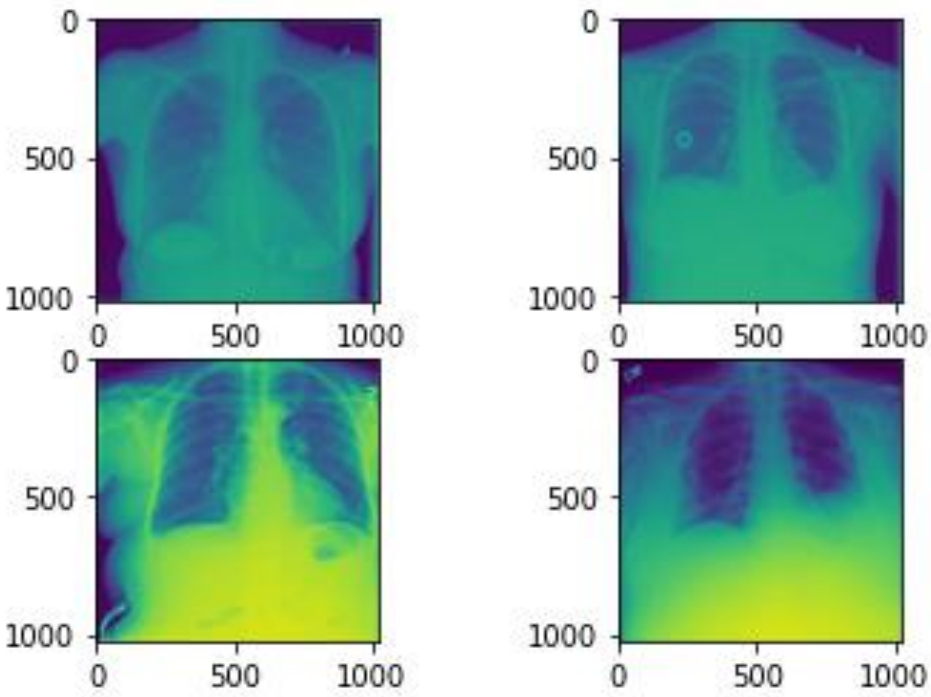
Moreover, we can also look at some examples of images that are correctly classified by all the available models. Even in this case, we have chosen to show just 4 images for each class.

CORRECTLY CLASSIFIED:

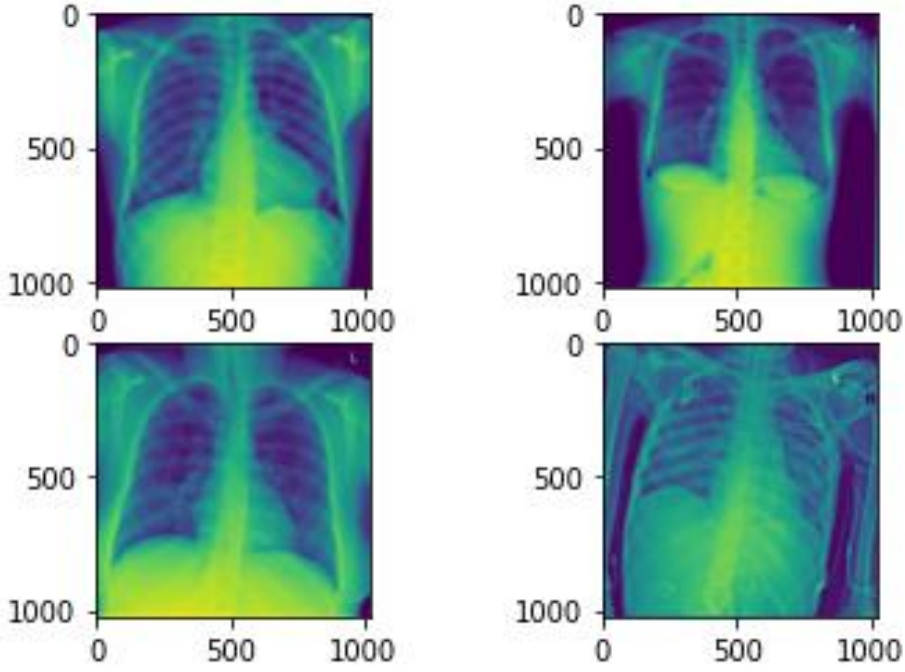
Atelectasis



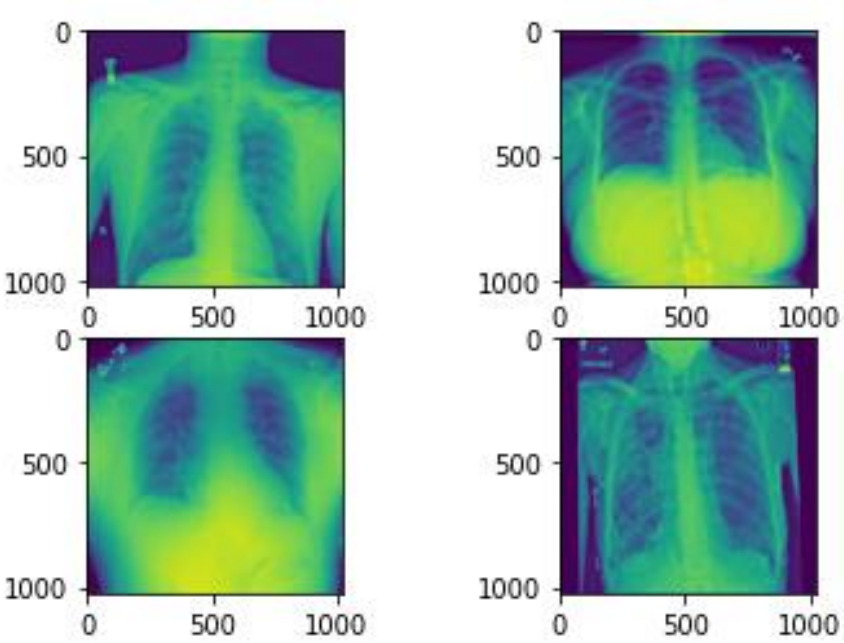
Effusion



Infiltration



No Finding



Summary

Looking at the following report we can easily understand differences in terms of performance of each one of the selected classifiers.

#Images misclassified by all the models	***176***
#Images correctly classified by all the models	***501***

A. ***Misclassified images:*** *images misclassified only by the model in the corresponding row*

B. ***Correctly Classified images:*** *images correctly classified only by the model in the corresponding row*

Model	***#Misclassified images***	***#Correctly classified images***
Vgg16	***43***	***60***
Resnet152	***88***	***48***
ChexNet	***40***	***84***
Fused	***49***	***53***
Cnn From Scratch	***53***	***27***

6 Explainability

In this analysis we will try to understand how the model is able to predict the class of X-ray images received in input through an analysis of the *intermediate activations* and the *class activation heatmaps in an image*.

INTERMEDIATE ACTIVATIONS

We have built a multi-output model which, given an image as input, allows you to print on the screen all the activations maps obtained from each convolutional and Max-pooling layer of the network.
Following are shown some activations of the network created from scratch, that has 6 convnet-max pooling blocks, each one producing a number of activation maps that depends on the number of kernels used and varies from 32 to 256.

First layer. Looking at *Figure 7*, we can see how the level of abstraction with which the images are considered is ***very low,*** in fact initially the network deals with identifying some generic characteristics, such as corner or edge, and for this reason almost all the information present in the image is preserved.
In the notebooks it is also possible to view the activation maps extracted from the other models and it can be seen that for the first layers the information extracted is almost the same. This highlights why in transfer learning through fine-tuning usually the first layers are left blocked, in fact it is possible to reuse already pre-trained features without reducing performances.

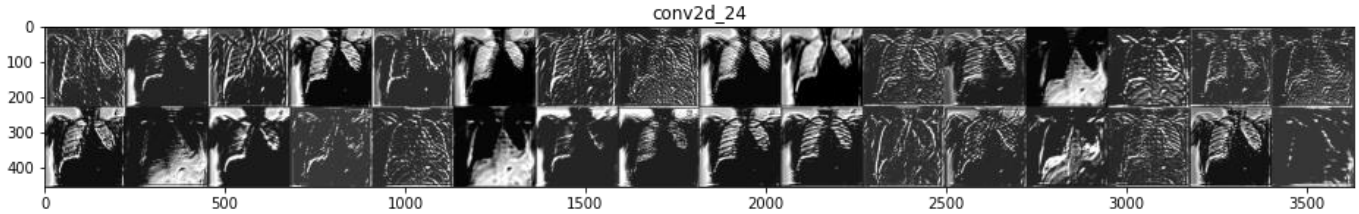


FIGURE 13: THE 32 ACTIVATION MAPS PRODUCED BY THE FIRST LAYER OF THE FROM SCRATCH MODEL.

Third layer. Considering the output of the third convolutional layer, we can see how the interpretability of activations is low, as they contain less and less visual information about the images and more and more specific characteristics that allow the model to identify the class associated with the image.

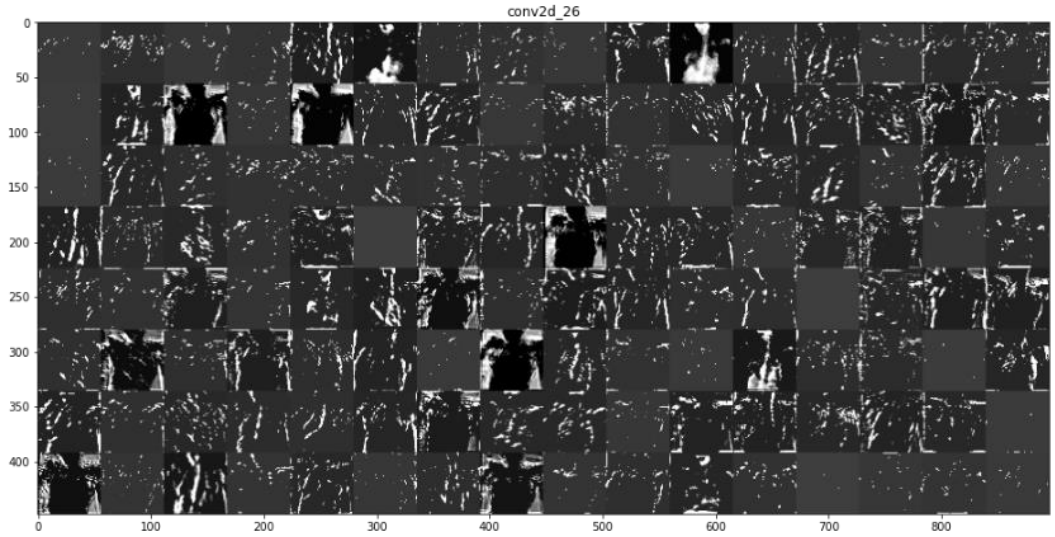


FIGURE 14: THE 128 ACTIVATION MAPS PRODUCED BY THE THIRD LAYER OF THE FROM SCRATCH MODEL.

Last layer. Finally, considering the activations of the last level, we can see that the level of abstraction is **so high** that the resulting activation cannot be traced back to the original image.

Moreover, it can be seen that many feature maps are black as each of them **looks for a specific feature of the input**, which is not always present. Therefore, since the output is negative, the ReLU, which is used as activation function, clears the final output of the feature map.

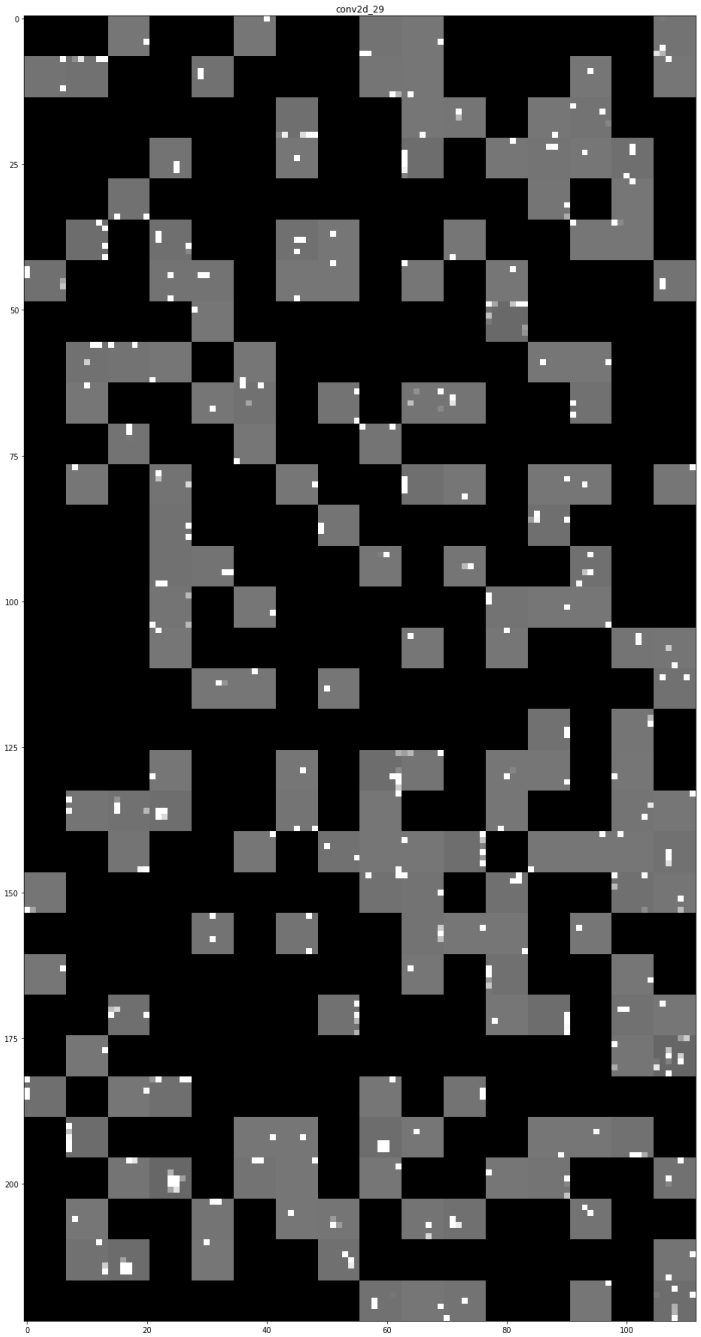


FIGURE 15: THE 512 ACTIVATION MAPS PRODUCED BY THE LAST LAYER OF THE FROM SCRATCH MODEL.

CLASS ACTIVATION HEATMAPS IN AN IMAGE

This section is particularly important to understand **how our models take decision** and if they take that **focusing on the same part of the image.**

To obtain it we have built a model that can produce an heatmap of **class activation over input image**, that is a grid that visually show which location is more considered to detect the right class.

This heatmap is given by the **channel-wise mean** of the feature maps produced by the last convolutional layer, that weighs them depending on **the importance** of each channel with regard to the top predicted class.
Following are shown heatmaps analysis obtained with our five best models and using three images, respectively of the *Atelectasis*, the *Infiltration* and the *Effusion* class, that were correctly classified by all the models.

ATELECTASIS

The medical description of the Atelectasis indicates this one such as condition where the **alveoli are deflated down to little or no volume.**

In the following image, taken from the test set, is rounded with a red box the most interesting region to detect that the patient is affected by Atelectasis.

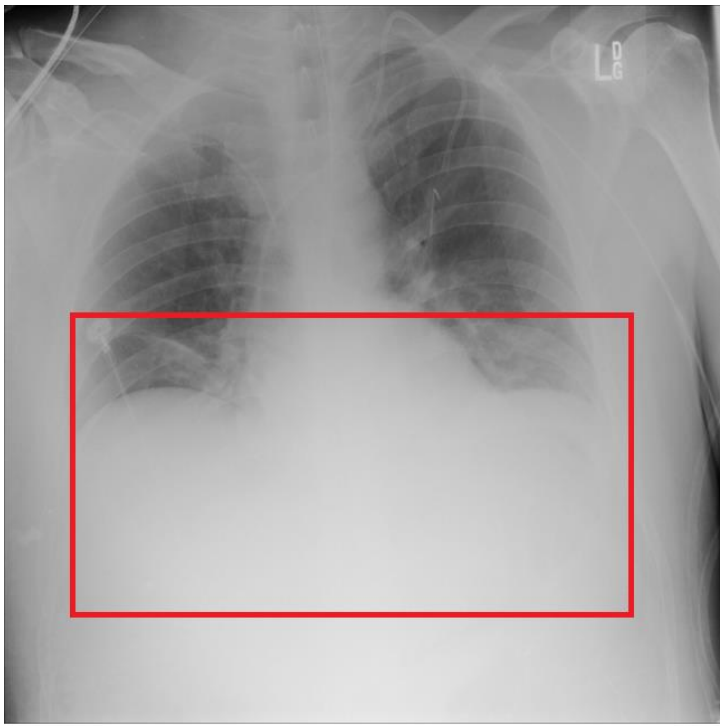
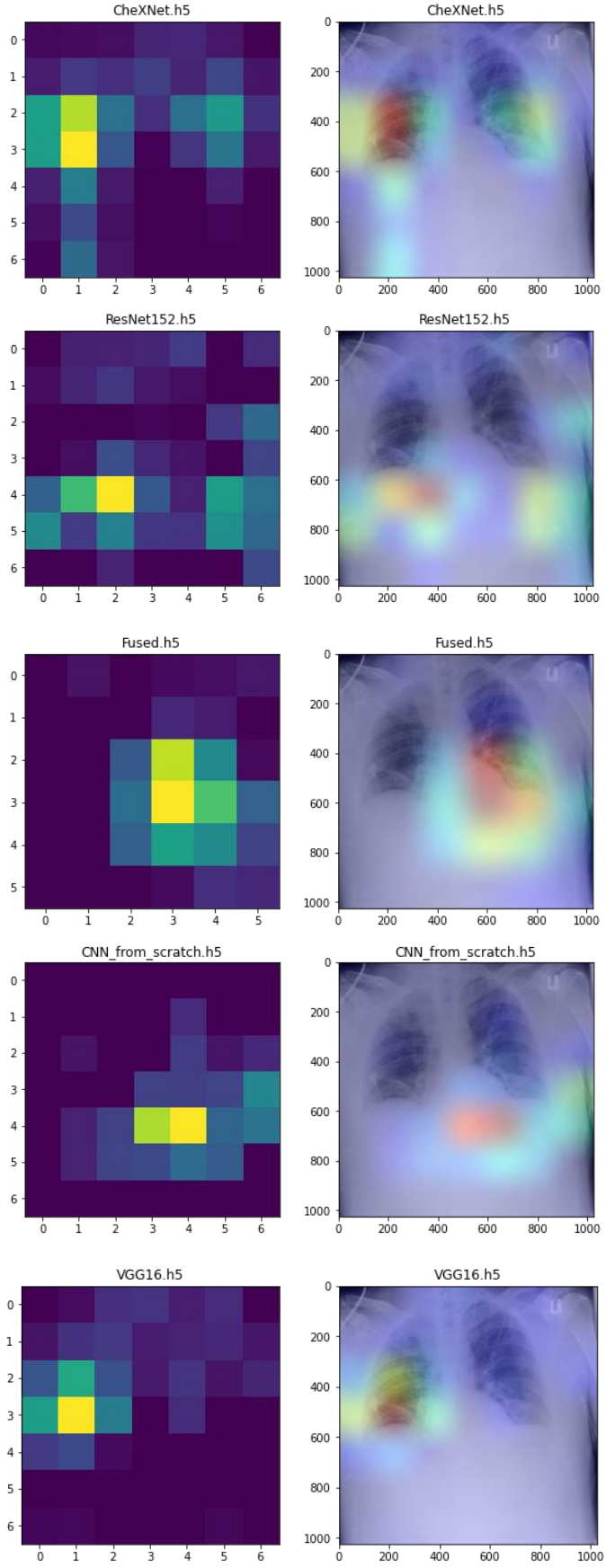


FIGURE 16: CHEST X-RAY IMAGE OF THE ATELECTASIS CLASS.

As we can notice, each one of the configurations is able to capture at least a part of the region of interest. In particular we can spot a little difference between the portion identified by the pre-trained models and the model trained from scratch; in fact, the former focus **on the left part** of the bounded region while the latter focus on the **right part** of the bounded region.


CheXNet.h5
CheXNet.h5
ResNet152.h5
ResNet152.h5
Fused.h5
Fused.h5
CNN_from_scratch.h5
CNN_from_scratch.h5
VGG16.h5
VGG16.h5

EFFUSION

Effusion consists in an excess of fluids and in the image below we can clearly spot the region of interest, that is the region of the thorax in which the models should focus on.

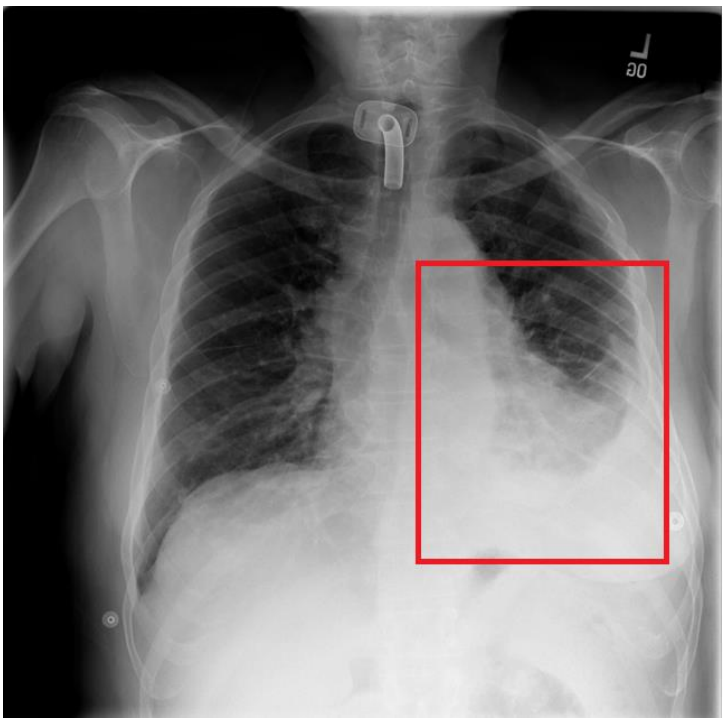
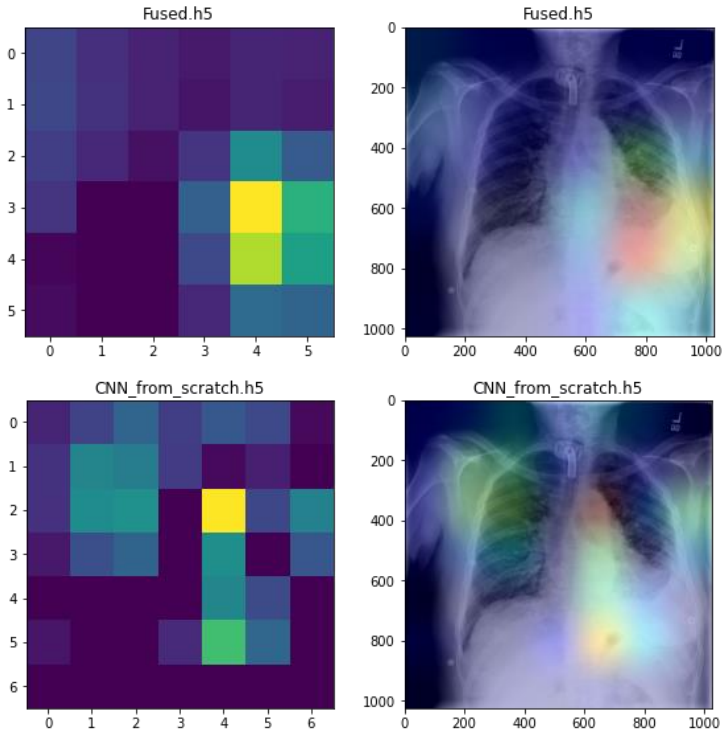
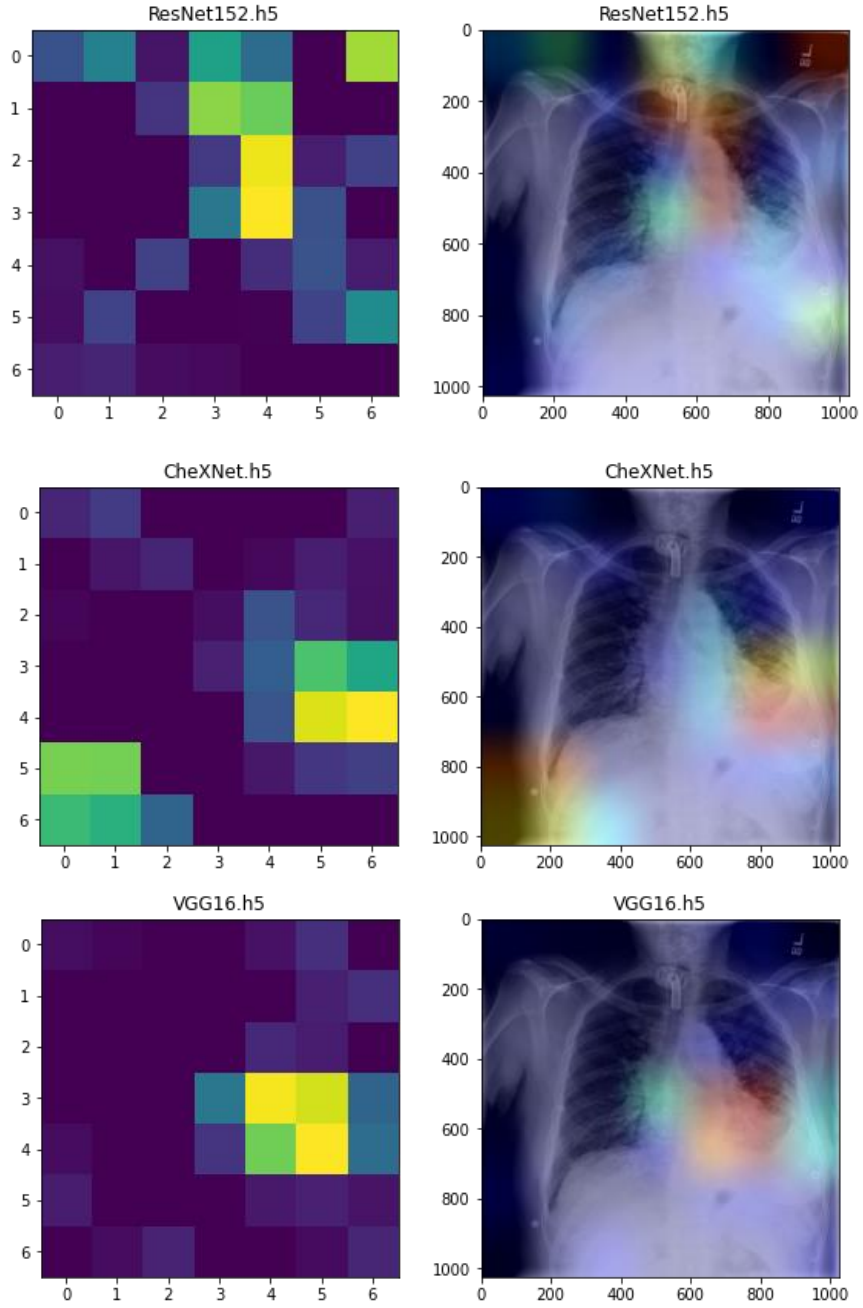


FIGURE 17: CHEST X-RAY IMAGE OF THE EFFUSION CLASS.

With respect to Atelectasis, all the configurations behave homogeneously and they can correctly identify the region that better than others suggest a patient affected by Effusion.





INFILTRATION

Infiltration corresponds to a substance denser than air, such as pus, blood, or protein, which lingers within the parenchyma of the lungs.

The recognition of this kind of pathology **represents the most difficult task for our classifiers**; the reason behind this struggle is probably tributed to the nature of the same pathology that is quite difficult to spot just with the usage of x-ray.

Like in the other cases, we have reported an image that has been correctly classified by all the best models and the region of interest is rounded by a red rectangle.

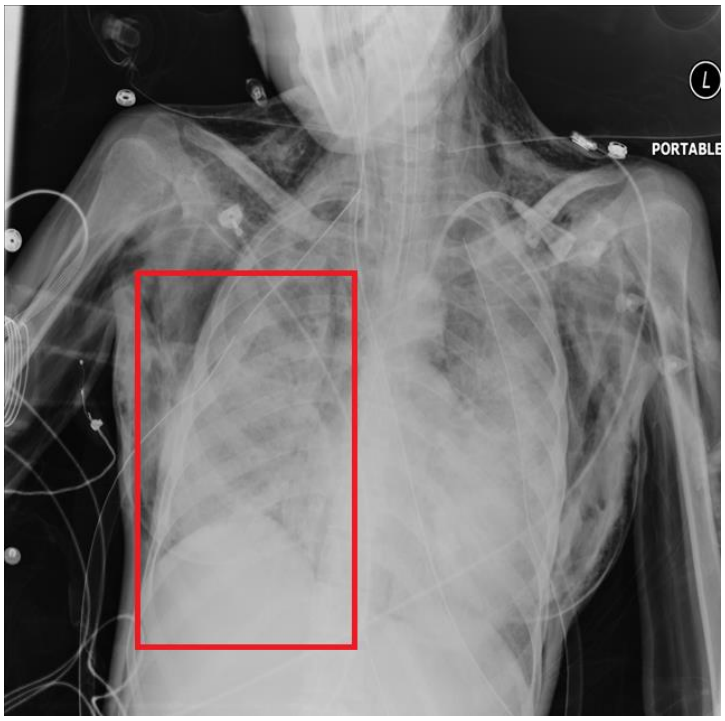
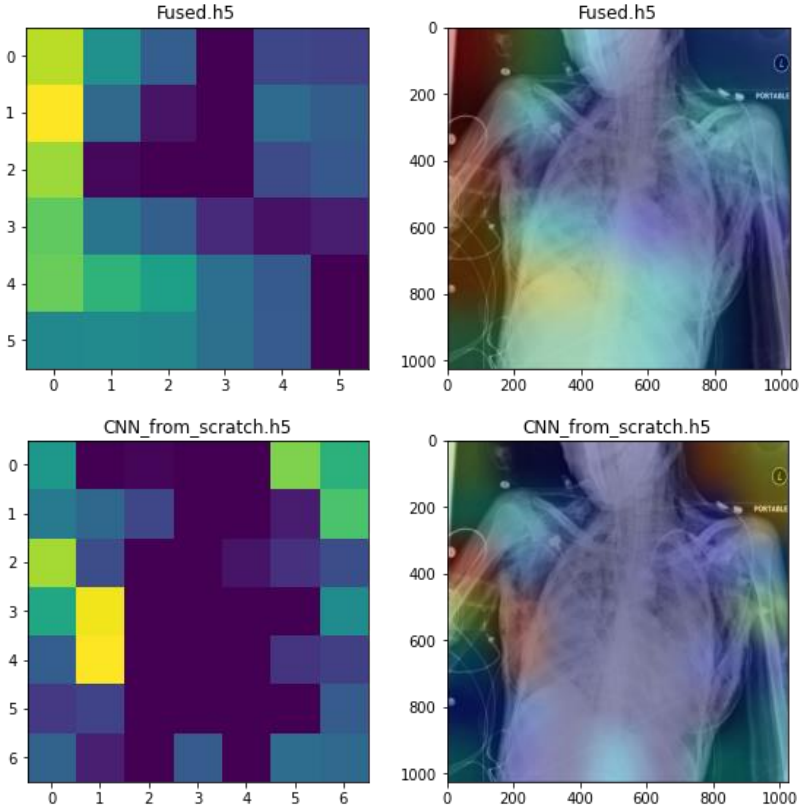
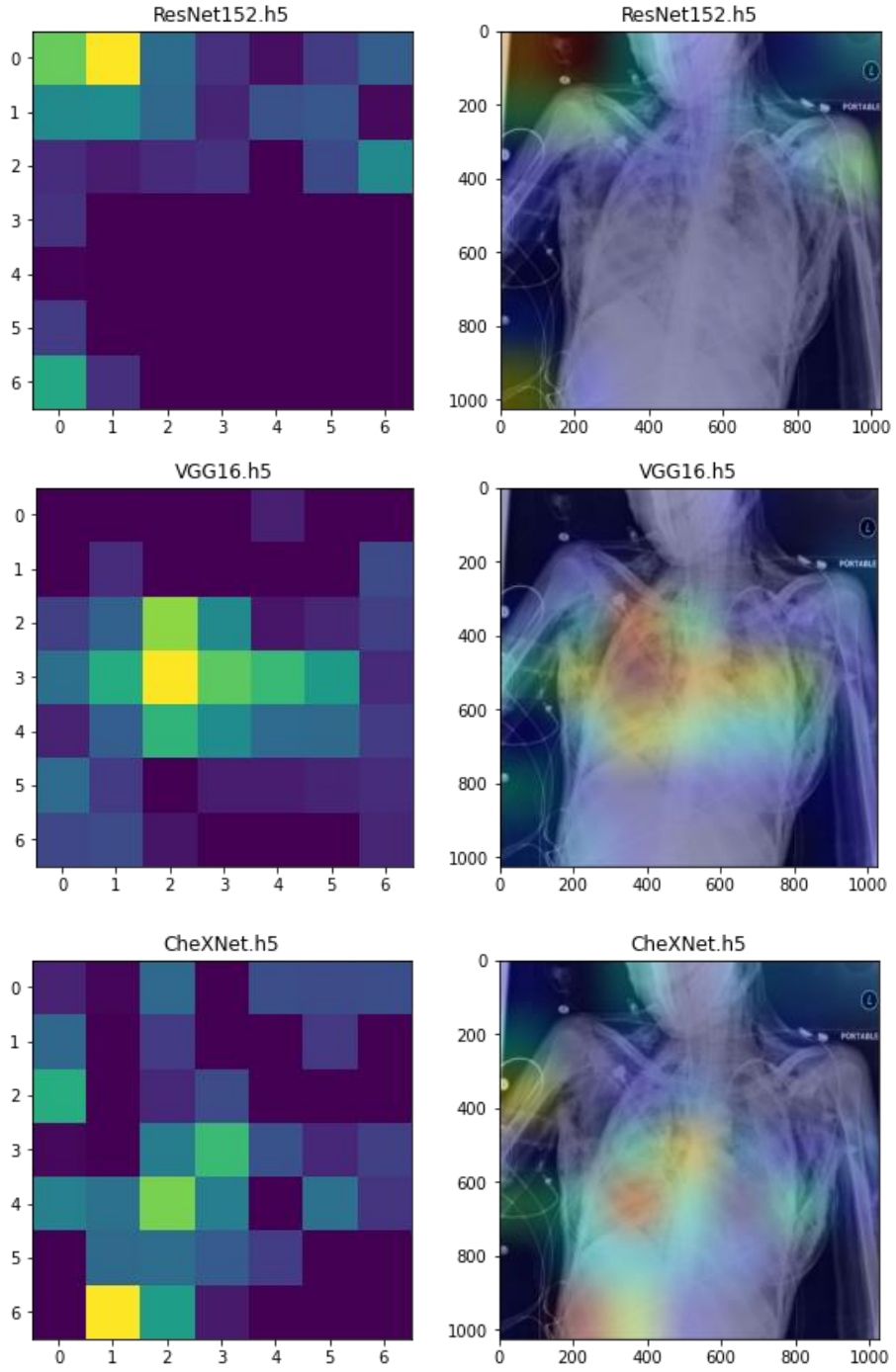


FIGURE 18: CHEST X-RAY IMAGE OF THE INFILTRATION CLASS.

Even if it has been correctly classified, it is visible the difficulty that models meet in recognizing the region in which they should focus; in particular, the Resnet model it the worse one because we can verify an anomaly behaviour but besides this it correctly classifies the image.




ResNet152.h5
ResNet152.h5
VGG16.h5
VGG16.h5
CheXNet.h5
CheXNet.h5

7 Ensemble

Model ensembling is a very powerful technique that typically achieves (or tries to achieve) the best possible results. Many of the solutions employed in the state of the art and in challenges usually involve the use of ensembles. For this reason, we decided to ensemble our best models.

We ensembled 6 different models with the subsequent individual performance on the test set:

Model	Accuracy	Loss	ROC AUC	FNFR	Inference Time	F1-Macro	F1-Atelectasis	F1-Infiltration	F1-No Finding	F1-Effusion
ResNet152	0.5987	0.9818	0.8317	0.1416	31.027	0.5906	0.4949	0.5773	0.6045	0.6847
VGG16	0.6405	0.9369	0.8606	0.0925	21.046	0.6334	0.5237	0.6771	0.6559	0.6793
CheXNet1	**0.7197**	0.7568	**0.9072**	**0.0857**	21.046	0.7179	0.6656	0.7464	**0.7344**	**0.7254**
CheXNet2	**0.7197**	**0.7567**	**0.9072**	0.0886	21.356	**0.7181**	**0.6667**	**0.75**	0.7316	0.7243
Fused	0.6363	0.9456	0.8447	0.1166	**7.864**	0.6296	0.5417	0.6302	0.6473	0.6992
CNN From Scratch	0.5987	0.9494	0.8394	0.1310	9.404	0.5855	0.4237	0.6534	0.6048	0.6602

FNFR: False No-Finding Rate. Images which have been mislabeled as No-Finding over the total number of images which aren't No-Finding

7.1 Average Voting

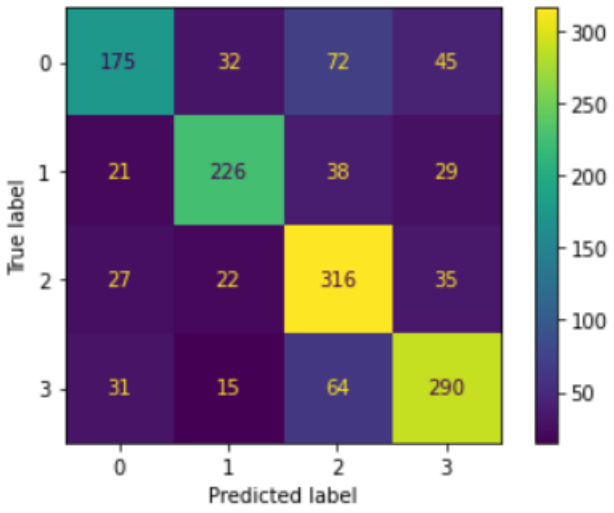
Since we already have the trained models, we can immediately ensemble them by averaging their scores (softmax output) for each input test image and then select the class with the highest score:

```
preds_a=model_a.predict(x_val)
preds_b=model_b.predict(x_val)
preds_c=model_c.predict(x_val)
preds_d=model_d.predict(x_val)

final_preds = 0.25 * (preds_a + preds_b + preds_c + preds_d)
```

FIG. 1 EXAMPLE OF AVERAGE VOTING WITH 4 MODELS

With this simple approach, we obtain a model with the subsequent performance:

Model	Accuracy	ROC AUC	FNFR	F1-Macro	F1-Atelectasis	F1-Infiltration	F1-No Finding	F1-Effusion
Average voting	0.7003	0.8945	**0.0809**	0.6959	0.6055	0.7422	0.7101	**0.7259**



This simple model has been able to outperform all the other models in *FNFR* and *F1-Effusion*.

7.2 Weighted Average Voting

In order to make the most of the ensemble technique, we decided to implement a weighted average of the predictions. In order to compute such weights, we decided to implement a *Genetic Algorithm* which searches for an optimal solution for the weights using the *validation set*.

7.2.1 Genetic *Algorithm*

We defined the problem as follows:

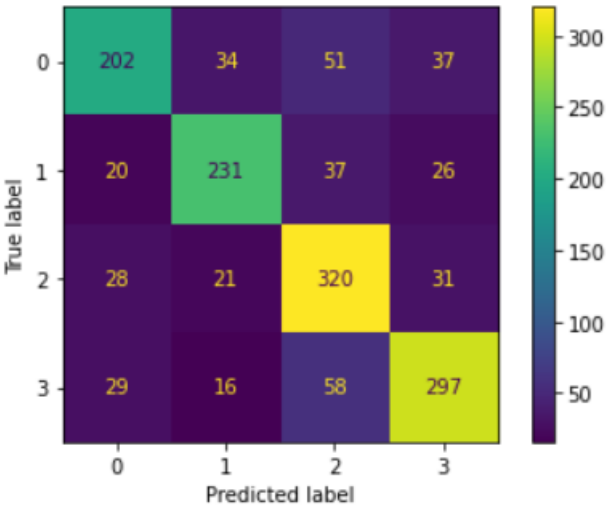
- Each chromosome is composed of a gene for each model in the ensemble. Each gene represents a weight (*real encoding*).
- The initial population of candidate solutions is randomly generated and then each individual is normalized in order to ensure that the sum of each gene in a chromosome is always 1 (weights must add up to 1, representing percentages).
- As fitness function, since the validation set is unbalanced, we decided to use the *F1-Macro*.
- As selection technique for recombination, in the end we implemented a *k-tournament selection*.
- As crossover operator, *average crossover* has been used. Each new chromosome is normalized to ensure the sum of its genes adds up to 1.
- For mutation, a single gene is chosen at random and is replaced with a randomly generated real number in the range [0, 1], then applying normalization.
- We implemented an elitism mechanism to ensure that the best solutions of each generation are maintained in the subsequent generation.

The hyperparameters of such genetic algorithm (*population size, number of generations, probability of crossover, probability of mutation, percentage of genes to use during crossover* and the *elitism percentage*) have been chosen experimentally.

The genetic algorithm has been run multiple times with different initial random populations. At the end, the best solution found was reported.

7.2.2 Results

Model	Accuracy	ROC AUC	FNFR	F1-Macro	F1-Atelectasis	F1-Infiltration	F1-No Finding	F1-Effusion
Weighted average voting	**0.7302**	**0.9073**	**0.0771**	**0.7275**	**0.67**	**0.75**	**0.739**	**0.751**



The ensembled model was able to outperform all the previous models in every statistic. In particular, we were able to increase the F1-Macro of 0.94% wrt the best model and the Accuracy of 1.05%.

The final weights are the following:

ResNet152	0.13074769
VGG16	0.13225159
CheXNet1	0.27880784
CheXNet2	0.25517746
Fused	0.16125192
CNN From Scratch	0.0417635

8 Conclusions

This project reports a series of possible solutions for the analysis of chest X-ray images.

In the end we weren't able to obtain great results, reaching a maximum of 73% of Accuracy and 72.75% of F1-Macro. In general, each model we tested ended up reaching a plateau during the training phase or overfitting the training set. Even gathering more data didn't help our model better generalize. Even the CheXNet, which was pretrained exactly on the *ChestXray14* dataset, was not able to obtain satisfying results in terms of Accuracy and F1-Macro.

There may be multiple reasons for it, starting from the *ChestXray14* dataset. The label for each image in this dataset have been mined by an NLP system with an F1-score of 94.4%, already introducing a degree of imprecision to our problem. Searching on the web, we also found the work of a medical AI researcher and radiologist [18] which states that *ChestXray14* alone as it is, is not enough for training medical AI systems to do diagnostic works; he analyzes the goodness and accuracy of labels, the actual medical meaning behind each label and how, in reality, more data is necessary in order to make a diagnosis (e.g. frontal images as we have in our dataset are not enough, we may also need lateral images and other information about the specific patient's clinical case).

That said, back to our problem; since we used a random subset retrieved from the ChestXray14 and analyzed a sub-problem composed of 4 random classes from the original 14, any comparison with performance obtained in the literature is useless. In the future, it could be interesting to use the full extension of *ChestXray14* and complement it with other datasets from other sources.

[10] states that "much deeper neural networks should not really be used to classify biomedical images. Much deeper neural networks tend to capture more abstract features and hence, tend to overlook the minor variances between similar images that may be the key to diagnosing different diseases". This can be seen even in our project; ResNet, the model with more parameters, was also the one performing the worst out of all, while our *Fused* model created from scratch was able to perform as good as the model fine tuned from a pretrained VGG16.

9 References

[1] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, 2014.

[2] K. He, X. Zhang, S.Ren and J. Sun, “Deep Residual Learning for Image Recognition”, 2015.

[3] P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya, M. P. Lungren, A. Y. Ng, “CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning”, 2017.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, L. Fei-Fei, “ImageNet: A large-scale hierarchical image database”, 2009.

[5] Hossein Talebi, and Peyman Milanfar Google Research “Learning to Resize Images for Computer Vision Tasks” , 2020.

[6] Caseneuve and Vilanova ”Chest XRay for Image Preprocessing”, 2021.

[7] S. Ioffe and C. Szegedy ,“Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, 2015.

[8] M. Elgendi, M. U. Nasir, Q. Tang, D. Smith, J.-P. Grenier, C. Batte, B. Spieler, W. D. Leslie, C. Menon, R. R. Fletcher, N. Howard, R. Ward, W. Parker and S. Nicolaou , “The Effectiveness of Image Augmentation in Deep Learning Networks for Detecting COVID-19: A Geometric Transformation Perspective”, 2021.

[9] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri and R. M. Summers, “ChestX-ray8: Hospital-scale Chest X-ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases”, 2017.

[10] S. Pang, A. Du, M. A. Orgun and Z. Yu , “A novel fused convolutional neural network for biomedical image classification”, 2018.

[11] Kingma, D.P., Ba and J. Adam: “A Method for Stochastic Optimization. The International Conference on Learning Representations”, 2015 2014; :1–15URL http://arxiv.org/abs/1412.6980.

[12] Weng, X., Zhuang, N., Tian, J, Liu, Y. “Chexnet for classification and localization of thoracic diseases”, https://github.com/arnoweng/CheXNet, 2017.

[13] Lakhani, Paras and Sundaram, Baskaran, “Deep learning at chest radiography: Automated classification of pulmonary tuberculosis by using convolutional neural networks”, 2017.

[14] Huang, Peng, Park, Seyoun, Yan, Rongkai, Lee, Junghoon, Chu, Linda C, Lin, Cheng T, Hussien, Amira, Rathmell, Joshua, Thomas, Brett, Chen, Chen, “Added value of computer-aided ct image features for early lung cancer diagnosis with small pulmonary nodules: A matched case-control study”, 2017.

[15] Islam, Mohammad Tariqul, Aowal, Md Abdul, Minhaz, Ahmed Tahseen, and Ashraf, Khalid, “Abnormality detection and localization in chest x-rays using deep convolutional neural networks”, 2017.

[16] Demner-Fushman, Dina, Kohli, Marc D, Rosenman, Marc B, Shooshan, Sonya E, Rodriguez, Laritza, Antani, Sameer, Thoma, George R, and McDonald, Clement J., “Preparing a collection of radiology examinations for distribution and retrieval. Journal of the American Medical Informatics Association”, 2015.

[17] Yao, Li, Poblenz, Eric, Dagunts, Dmitry, Covington, Ben, Bernard, Devon, and Lyman, Kevin. “Learning to diagnose from scratch by exploiting dependencies among labels”, 2017.

[18] https://laurenoakdenrayner.com/2017/12/18/the-chestxray14-dataset-problems/