Using Deep Neural Networks to Detect Pneumonia

Euthimios Koutsimanis
Department of Science and Technology
International Hellenic University
Thessaloniki, Greece
makiskoutsiman@gmail.com

Christos Galanis
Department of Science and Technology
International Hellenic University
Thessaloniki, Greece
xristosgala@gmail.com

***Abstract*—This paper is prepared as a coursework in the context of the course "Advanced Machine Learning". The authors participated in a competition on Kaggle.com which given Chest X Rays images the model needs to predict whether the patient has pneumonia or not. The evaluation metric was accuracy. This paper includes preprocessing steps of the data meaning handling imbalanced classes, standard-normalization, and data augmentation. After that, a series of different models were used. That ranges from author's build models to prebuild models and ensemble models. Some of the latter include Xception, DenseNet121, average ensemble techniques and more. At the end, three models resulted in high performance regarding the submission accuracy. These were the Xception models with weight warm starting and the ensemble model which combines the previous referred models.**

Keywords—ensemble models, data augmentation, standardization, normalization)

I. Problem & Data Description

A. Problem Introduction

In the context of the course "Advanced Machine Learning" we have been assigned to participate in a Kaggle competition named "Detect Pneumonia (Spring 2023)". As the name of the competition suggests, the goal is to detect pneumonia from a variety of chest X-rays images.

B. Data Introduction

The images are gray-scale with various sizes and they consist of the following 3 classes:

- Class 0: no disease
- Class 1: bacterial pneumonia
- Class 2: viral pneumonia

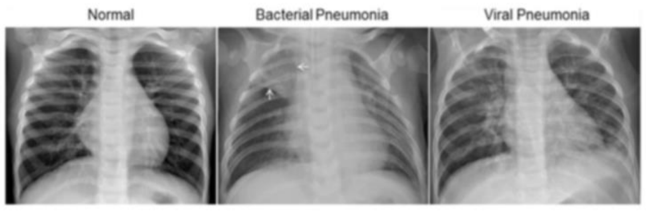


Figure 1: Examples of Chest X-Rays in Patients with Pneumonia.

The above figure shows the 3 target classes. More specifically, the normal chest X-ray (left panel) illustrates clear lungs without any areas of abnormal opacification in the image. Continuously, the Bacterial pneumonia (middle) typically exhibits a focal lobar consolidation, in this case in the right upper lobe (white arrows), whereas viral pneumonia (right) manifests with a more spread ''interstitial'' pattern in both lungs.

C. Background of data

As part of patient's routine clinical care, all the Chest X-ray images (anterior-posterior) were obtained from retrospective cohorts of pediatric patients of one to five years old from Guangzhou Women and Children's Medical Center, Guangzhou.

In order these images to be suitable for analysis, all chest radiographs were initially screened for quality control by removing all low quality or unreadable scans. After that, two expert physicians graded the images for diagnoses before being cleared for training the ML system. A third expert was also utilized for checking the evaluation set aiming to mitigate any grading errors.

D. Dataset description

The data comprise of two folders:

Folder: contains 4672 train images, out of which:

- 1227 images belong to class 0
- 2238 images belong to class 1
- 1207 images belong to class 2

Folder: contains 1168 test images.

Also, the dataset includes a csv file which contains the target labels of the train images in the form of pairs i.e. file_name, class_id.

E. Evaluation metric

The evaluation criteria for this competition is the test accuracy as depicted below:

$$Accuracy = \frac{Number\ of\ correctly\ predicted\ test\ images}{Total\ number\ of\ test\ images}$$

II. Description of models used

This section will provide a brief background on the deep neural networks that have been used in the classification purpose of this coursework. However, it is vital a reference to be made to the preprocessing steps before proceeding to the model's description.

A. Loading the images

The images are grayscale meaning that they have only 1 channel depicting the scale between 0 and 255 for each pixel. However, some of them have 3 channels, all channels with the same value resulting again in a grayscale image. Some of the models that will be used are built by the researchers, some others are prebuilt and pretrained. The latter obtained from the Keras applications require the data to have 3 channels. Therefore, all the images will be loaded having 3 channels i.e., all the 3 channels will have the same value in order to depict the grayscale image. Finally, the images are resized in order to have size 224, 224, 3.

B. Handling the imballance class issue

As it was mentioned previously, the dataset contains a target variable meaning 3 classes. However, all classes are not in the same number. This results in having a dataset highly imbalanced and decrease in the performance of the model. The imbalanced issue is depicted below:

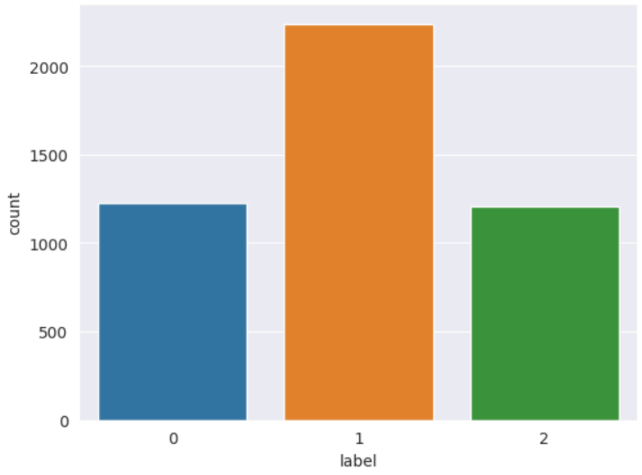


Figure 2: Imbalance Classes.

To overcome this problem and increase the performance and accuracy of the model the authors applied oversampling in both of the minority classes i.e. 0 and 2. The final distribution of the target variable is shown below in the figure and the table:

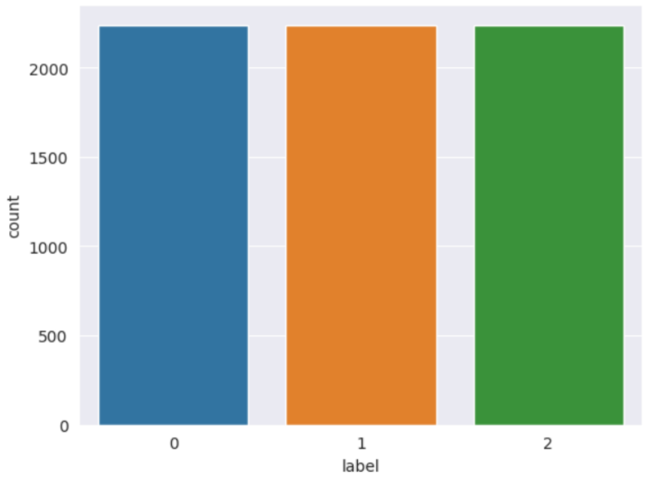


Figure 3: Final class distribution after performing oversampling.

Table 1: Class distribution after oversampling.

Classes	Count
0	2238
1	2238
2	2238
Total	**6714**

C. *Standard-Normalization*

A necessary part of every machine learning – data mining task is to perform data transformation aka feature scaling. Each pixel takes a value between 0 and 255 with 0 means black and 255 means white combining all the pixels of an image results in a grayscale visualization of the image. However, when fitting the pixels of every image to the model then it often compares high and small values together resulting in poor performance. Thus, one should rescale every image in order the pixels of every image to be easily comparable. In this coursework, the data were divided by 255.0. After that, the mean of the train data was calculated and subtraction by the train data and the test data. The above techniques significantly improved the results of the model. The mean of the train data is illustrated below:

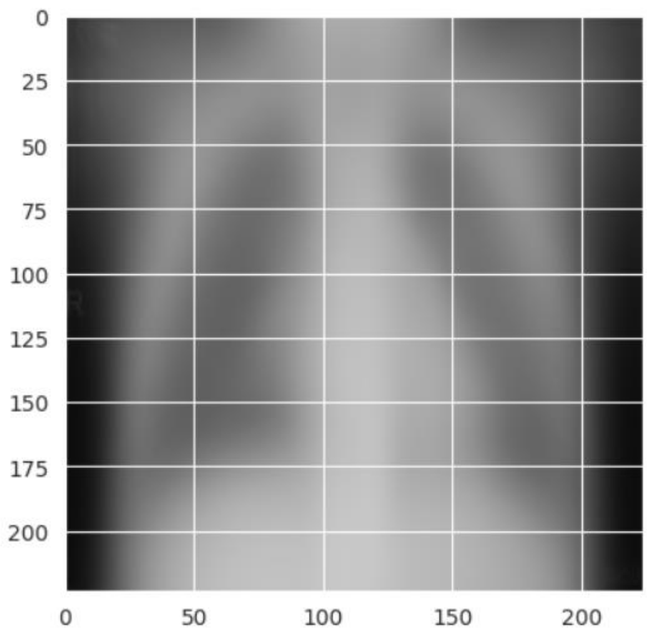
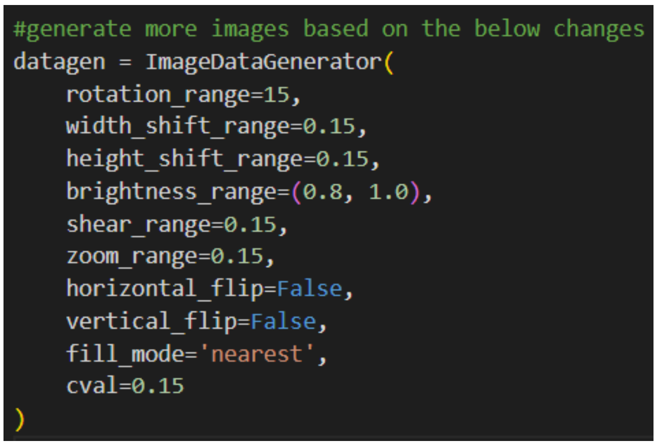


Figure 4: Mean of the train data.

D. *Data Augmentation*

Another important step to be referred is that of the data generation. When using deep learning models with millions of trainable parameters a lot of data are needed in order the model to be able to see and understand many different cases. Models trained in a low number of training data appear the phenomenon of overfitting. That is, model is able to understand very well the training data, however, it is not performing well on the test data meaning it does not generalize well. To overcome this problem a keras method was utilized. In other words, this data generation method from keras generates batches of tensor image data with real-time data augmentation. The following parameters of the augmented images were changed:



```
#generate more images based on the below changes
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    brightness_range=(0.8, 1.0),
    shear_range=0.15,
    zoom_range=0.15,
    horizontal_flip=False,
    vertical_flip=False,
    fill_mode='nearest',
    cval=0.15
)
```

Figure 5: Data Augmentation parameters

E. *Different Models*

1) *Naive Model*

The first model that was used is a model built by the authors in a naïve way meaning without many parameters and limited ability for generalization.

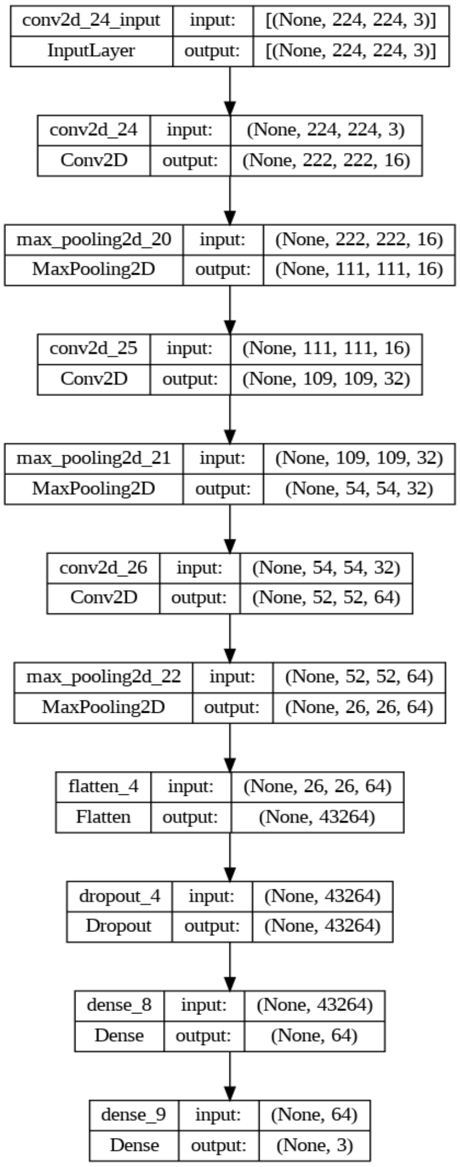

conv2d_24_input | InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
conv2d_24 | Conv2D | input: (None, 224, 224, 3) | output: (None, 222, 222, 16)
max_pooling2d_20 | MaxPooling2D | input: (None, 222, 222, 16) | output: (None, 111, 111, 16)
conv2d_25 | Conv2D | input: (None, 111, 111, 16) | output: (None, 109, 109, 32)
max_pooling2d_21 | MaxPooling2D | input: (None, 109, 109, 32) | output: (None, 54, 54, 32)
conv2d_26 | Conv2D | input: (None, 54, 54, 32) | output: (None, 52, 52, 64)
max_pooling2d_22 | MaxPooling2D | input: (None, 52, 52, 64) | output: (None, 26, 26, 64)
flatten_4 | Flatten | input: (None, 26, 26, 64) | output: (None, 43264)
dropout_4 | Dropout | input: (None, 43264) | output: (None, 43264)
dense_8 | Dense | input: (None, 43264) | output: (None, 64)
dense_9 | Dense | input: (None, 64) | output: (None, 3)

Figure 6: Naive Model Architecture

The parameters are the following:

- Total params: 2,792,739
- Trainable params: 2,792,739
- Non-trainable params: 0

2) Improving the naïve model

In order to improve the previous model with the naïve architecture, the researchers used both the batch normalization and dropout techniques. These are two commonly used techniques in deep learning to improve the performance and generalization of neural networks. Batch Normalization is a method used to normalize the activations of each layer in a neural network. It helps in reducing the internal covariate shift and allows the network to converge faster. While Dropout is a regularization technique that helps prevent overfitting in neural networks by randomly dropping out (setting to zero) a fraction of input units during training. It introduces noise and forces the network to learn more robust and redundant representations. The new architecture is as follows:

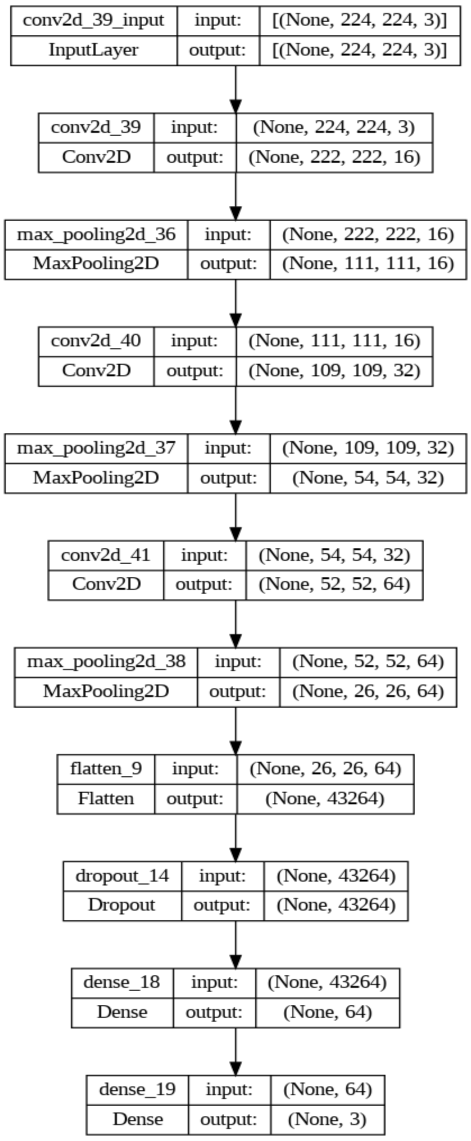

conv2d_39_input | InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
conv2d_39 | Conv2D | input: (None, 224, 224, 3) | output: (None, 222, 222, 16)
max_pooling2d_36 | MaxPooling2D | input: (None, 222, 222, 16) | output: (None, 111, 111, 16)
conv2d_40 | Conv2D | input: (None, 111, 111, 16) | output: (None, 109, 109, 32)
max_pooling2d_37 | MaxPooling2D | input: (None, 109, 109, 32) | output: (None, 54, 54, 32)
conv2d_41 | Conv2D | input: (None, 54, 54, 32) | output: (None, 52, 52, 64)
max_pooling2d_38 | MaxPooling2D | input: (None, 52, 52, 64) | output: (None, 26, 26, 64)
flatten_9 | Flatten | input: (None, 26, 26, 64) | output: (None, 43264)
dropout_14 | Dropout | input: (None, 43264) | output: (None, 43264)
dense_18 | Dense | input: (None, 43264) | output: (None, 64)
dense_19 | Dense | input: (None, 64) | output: (None, 3)

Figure 7: Improved Naive Architecture Model

The parameters are illustrated below:

- Total params: 877,451
- Trainable params: 852,055
- Non-trainable params: 25,396

F. Prebuild Models

In this section, it is important to note the benefits of using prebuild models:

- **Time and Resource Efficiency:** Prebuilt neural networks have already been trained on large datasets, often with powerful hardware and extensive computational resources. This means that one can initialize the model tested with pretrained weights learned from large datasets. Therefore, one can save time and computational resources that would otherwise be required to train a network from scratch.
- **Transfer Learning:** Pretrained models can be used as a starting point for transfer learning.

Transfer learning is the process of taking a pretrained model and fine-tuning it on a different but related task or dataset. By utilizing the learned knowledge and features, you can achieve better performance with less training data and training time.

- **Feature Extraction:** Prebuilt networks provide a convenient way to extract high-level features from input data. Using a network up to a certain layer one can extract the output from that layer as feature representations. This results in exploiting these representations as input for other machine learning algorithms or for building custom models.
- **State-of-the-Art Architectures:** Prebuilt networks are often based on state-of-the-art architectures that have been proven to perform well on various tasks. Furthermore, well-known models like DenseNet121, Xception, EfficientNetB0 and more from the keras applications have been designed and optimized by experts in the field, incorporating techniques and architectural choices that have shown to be effective for a wide range of problems. In simple words, these models win competitions.

1) Xception

The model that was used and perform better than the others was the Xception. A very well-known model which is exploited by many scientists and tested in many applications. After seeing that this prebuild model performs well, the authors fine-tuned it resulting in two models with two slightly different architectures regarding the layer before the final dense layer. Specifically, the authors used 3942 neurons for one model and for the other model 942 neurons after careful fine tuning.

Furthermore, a technique called weight warm-starting (weight initialization) was applied to each of the two above models since it was essential to increase their performance more. The idea behind weight initialization is to leverage the knowledge gained from the previous training session by starting the subsequent training with initial weights that are already close to the optimal solution. This can potentially accelerate the convergence and improve the overall performance of the neural network.

2) A variety of models

Furthermore, models including DenseNet121, EfficientNetB0, VGG16 and ResNet50 were also used and trained for predicting pneumonia.

Each of the above prebuild models belong to specific family of models for example one can use DenseNet169 or DenseNet201 instead of using the simple version of the DenseNet Family. That is, in some families of models there are architectures more complicated and each model more suitable for a variety of problems.

G. Ensemble Model

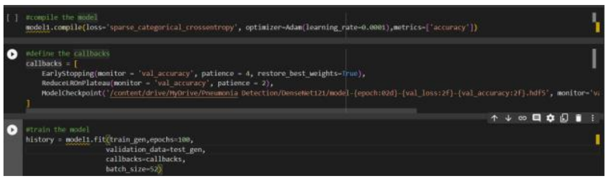
The Xception models with the weight warm starting technique perform better than the other algorithms and were combined and the average of their predictions were kept.

H. Discussion and comments

After submissions for each model, the most common "settings" were kept. Clarifications regarding the models and parameters selections are discussed thoroughly below:

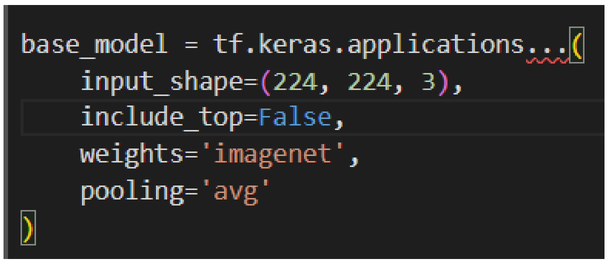
- Due to computational power and resources limitations only the "simple" models from each of the referred families were selected. That is, complex networks with complex architectures that require high computational needs were not selected.
- All the models were compiled with the Adam optimizer and an initial learning rate 1e-4 except the ensemble model which was initialized by a learning rate of 1e-5 after trial and error.
- Callbacks such as early stopping, reduce the learning rate and model checkpoint are very popular and good techniques for improving the performance of the model and restoring the best weights. Specifically, the model stopped when the validation accuracy did not increase after 4 epochs. The learning rate was reduced by a factor of 0.1 after the validation accuracy did not increase for 2 epochs. Lastly, the best weights regarding the highest validation accuracy were restored using the model checkpoint technique.
- For the prebuild models, the same architecture was used before the final dense layer. Also, the "imagenet" weights were given initially.
- A batch size of 52 for all models was given.
- The models with the best performance were fine-tuned by the researchers.

Some of the above modifications are shown below:



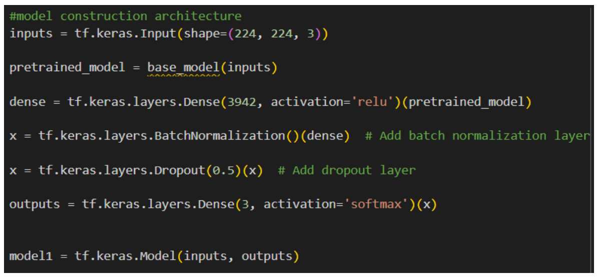
```
#compile the model
model1.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(learning_rate=0.0001),metrics=['accuracy'])

#define the callbacks
callbacks = [
    EarlyStopping(monitor = 'val_accuracy', patience = 4, restore_best_weights=True),
    ReduceLROnPlateau(monitor = 'val_accuracy', patience = 2),
    ModelCheckpoint('/content/drive/MyDrive/Pneumonia Detection/DenseNet121/model-{epoch:02d}-{val_loss:2f}-{val_accuracy:2f}.hdf5', monitor='v
]

#train the model
history = model1.fit(train_gen,epochs=100,
                     validation_data=test_gen,
                     callbacks=callbacks,
                     batch_size=52)
```

Figure 8: Different modifications of the models.



```
base_model = tf.keras.applications...(
    input_shape=(224, 224, 3),
    include_top=False,
    weights='imagenet',
    pooling='avg'
)
```

Figure 9: Prebuild models' code.



```
#model construction architecture
inputs = tf.keras.Input(shape=(224, 224, 3))

pretrained_model = base_model(inputs)

dense = tf.keras.layers.Dense(3942, activation='relu')(pretrained_model)

x = tf.keras.layers.BatchNormalization()(dense)  # Add batch normalization layer

x = tf.keras.layers.Dropout(0.5)(x)  # Add dropout layer

outputs = tf.keras.layers.Dense(3, activation='softmax')(x)

model1 = tf.keras.Model(inputs, outputs)
```

Figure 10: Architecture of prebuild models before the final dense layer.

III. Comparative Experiments and Results

In this section, the results of each model are presented. As said earlier, all the models trained with the same callbacks and were compiled the same. However, one model performed better than the others and subsequently it is mentioned thoroughly.

Let's begin with the first model, which was simple in terms of complex architecture and generalization. This model was unable to reach above approximate 73% validation accuracy. It can be seen from the below figure that the training accuracy and the validation accuracy do not increase resulting the process to stop:

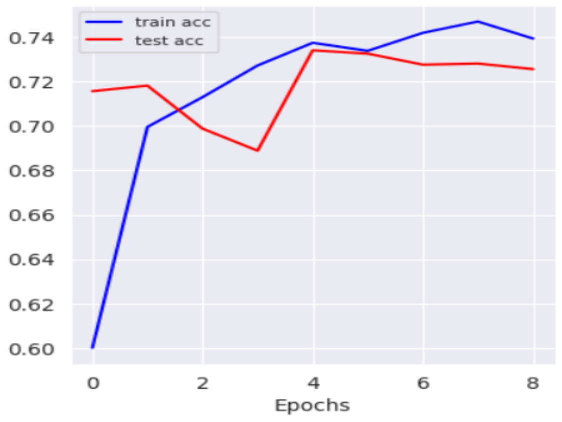


Figure 11: Naive model training and testing accuracy

In the case of the improved one, although the authors did used batch normalization and dropout techniques in order to increase the performance of the model the final result was poorest in comparison to the previous model i.e. naïve model. The results can be shown below:

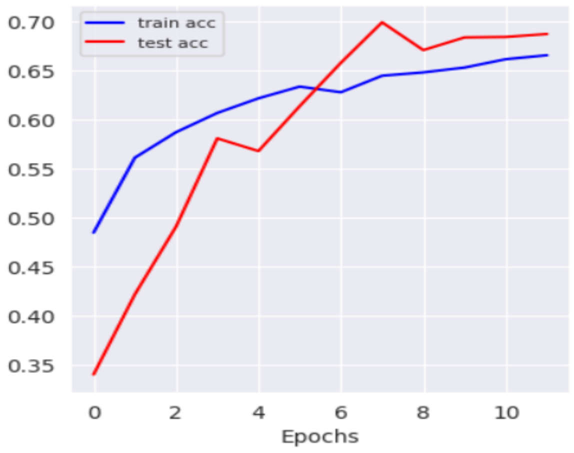


Figure 12: "Improved" model training and testing accuracy

This can be for many reasons. Some of them are the architecture is not complex, wrong thresholds in the dropout method or wrong padding values.

As far as now, both models built by the authors did not perform well. Therefore, prebuild models are taken place.

For visualization and space efficiency, the following table concludes the best performance of each model meaning the highest validation accuracy on the validation data and the corresponding submission score:

Models	Val. Loss	Val. Acc.	Submission Score
Naïve	0.67	0.73	<=0.80136
Improve Naïve	0.67	0.70	<=0.80136
DenseNet121	0.51	0.86	<=0.82876
EfficientNetB0	0.64	0.80	<=0.82534
VGG16	0.38	0.85	<=0.82876
ResNet50	0.42	0.87	<=0.82876
Xception (3942 neurons)	0.42	0.91	<=0.8476
Xception (942 neurons)	0.46	0.88	<=0.84589
Xception (3942 neurons) –weight warming start	0.27	0.91	<=0.85787
Xception (942 neurons) - WWS	0.25	0.92	<=0.85102
Ensemble	0.28	0.92	<=0.86130

So, from the above table we can clearly see that the Xception models produced good results in the submission score. Also, the Xception models with the weight warming start resulted in ever greater performance than the previous referred models. In addition, the ensemble model which combines both the two Xception algorithms with the weight warming start touched the 86% submission score.

IV. Conclusions

All in all, this coursework aims to predict pneumonia based on images. Preprocessing steps took place in order for the data to be ready for the model. These methods include handling the imbalanced dataset, standard-normalization, and data augmentation. After that, a variety of models trained and tested on the data from models built by the authors to prebuild models. In the end, the results showed that 3 models performed better than the others regarding the submission accuracy