**CS 464 Term Project Final Report**

**Spring 2019-2020**

**Group 6**

# Deep Learning - Based Classification of Pneumonia and Normal Lung Using Chest X-Ray Images

**İsmail Tuna Demirayak - 21802504 - CS**

**Melisa Taşpınar - 21803668 - CS**

**Mert Tunç - 21602944 - EE**

**Ömer Faruk Akgül - 21703163 - CS**

**Yiğit Gürses - 21702746 - CS**

**10/05/2020**

## TABLE OF CONTENTS

## 1. Introduction

The aim of this project is to determine whether a patient has pneumonia in his/her lungs based on the given X-ray image using deep-learning methods. We used two different approaches to solve the problem. One is transfer learning and the other is using InceptionV3 and an added custom CNN framework. 91%(?) test accuracy was achieved in the model created with transfer learning. The model using our own Custom CNN framework correctly classified 87%(?) of the test data.

## 2. Problem Description

X-ray is a method that is widely used by physicians in terms of practicality of use and cost. However, it is much more difficult to diagnose on X-Ray compared to other imaging methods such as Computed Tomography and MRI. For this reason, deep learning models can be used to increase X-ray diagnostic accuracy. In this project, we designed models to help physicians increase the success rate in pneumonia diagnosis. In addition, by comparing our two different models that give pneumonia and non-pneumonia outputs, that is, binary classification, it was investigated which model performed better for diagnosis.

## 3. Methods

Our first method uses transfer learning where a model developed for ImageNet is reused as the starting point of our model. In the first stage, pre-processing was performed on the images in both train and validation sets in the data set. For this purpose, the ImageDataGenerator function in the tensorflow library was used. Then, in order to extract features from X-ray images, we loaded an Inception v3 model that is pre-trained on ImageNet [1] dataset. We have found that not freezing any layers gave the best performance, so without freezing any layers, we added our own custom framework at the end of the Inception v3 network. After expanding the CNN with two more layers using ReLU as activation function, the final layer using sigmoid activation is added. After completing all the described steps, the model is fit to the training data for 20 epochs using RMSprop optimizer. In addition, L1 and L2 regularization parameters are tuned to prevent overfitting to the training data.

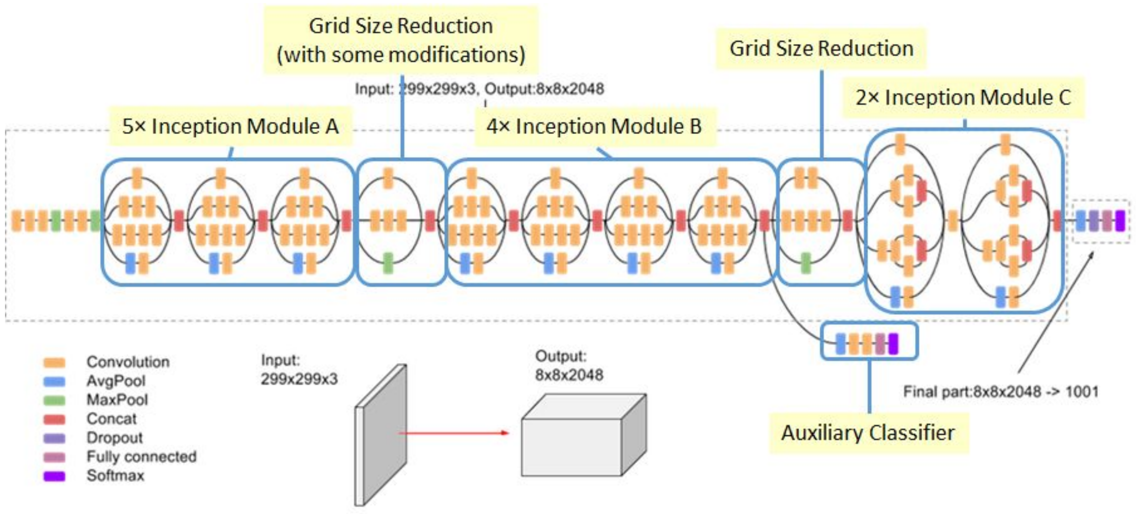

**Figure 1**

Our second approach was to build our own custom CNN, which consists of 3 Convolutional layers and 2 fully connected layers. A Pooling layer using max pooling is added to the end of each convolution. This significantly reduces the training time and preserves significant information[3]. Then, two fully connected layers take the features and make the prediction. The last layer has only one unit because prediction will be binary, that is, pneumonia or not-pneumonia. While compiling the neural network Adam optimizer is used which optimizes how fast our model learns the correct classification of the image. Similarly, before implementing our own CNN, images are pre-processed using ImageGenerator.

The data described below is used.

### 3.1 Data

The dataset consists of chest X-Ray images of pediatric patients of 1 to 5 years old from Guangzhou Women and Children's Medical Center, Guangzhou. The dataset is organized into 3 folders (train, test, val) and contains subfolders for each image category (Pneumonia/Normal). There are 5,863 X-Ray images (JPEG) and 2 categories (Pneumonia/Normal) [2]. The training set is imbalanced with 1341 x-rays of healthy lungs and 3875 x-rays of lungs with pneumonia.

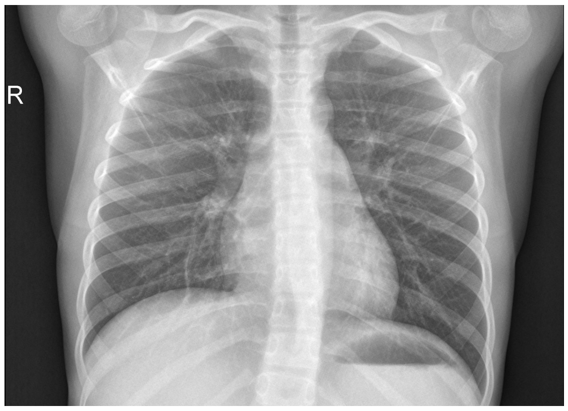

**Figure 2**
Sample from normal class

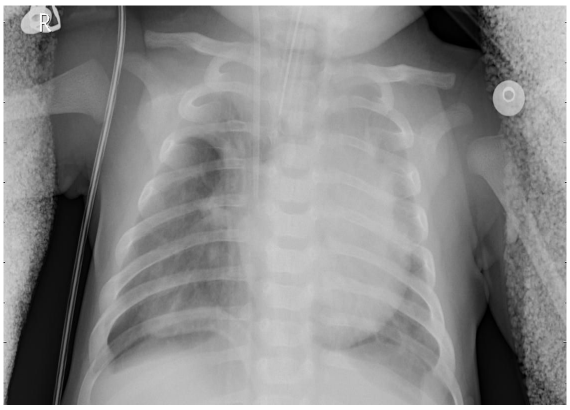

**Figure 3**
Sample from pneumonia class

## 4. Results

The models for both Inception-v3 and our own custom CNN were trained on Nvidia RTX-2060 GPU and Intel I7-9750 CPU.

### 4.1 Inception v3

Here, we are using the Inception v3 neural network with transfer learning, using weights obtained from training on ImageNet dataset.

To be able to conduct an experiment, we first needed to create a model with a given set of parameters. This will be Model 1. Then, to create a new model, we needed to change a chosen parameter every time. This way, that chosen parameter can act as an independent variable, with the other parameters being constant variables (equal to their values in Model 1). Hence, we could observe the changes in the new model's performance, that is, the changes that were caused by the modified parameter. This makes the performance of the model, e.g. test accuracy, our dependent variable.

To implement this, first, we created a base model. Our purpose was to ensure that the initial weights would be the same for all models. Then, we used this single base model to create our 6 actual models. Model 1 is where we used our optimized parameters, as before we began our experiment, we had optimized the parameters to have a good validation accuracy. This was done by trial and error, and deduction. We used these values in Model 1. Then, to create each subsequent model, from model 2 to model 6, we took the set of parameters we used in model 1 and  changed exactly 1 parameter. This way, we can compare a given model with model 1 and observe how the change we made in a parameter affects the performance of the new model.

In all our models, we decided not to freeze any layers of our models, and we used batch size = 16. For the first 3 models,  number of epochs = 20, and for the last 3, number of epochs = 10. The training time per epoch was a little more than 2 minutes.

The summary of the models will not be put on this report as they are very long. However, to give an idea of what we did, these were the modifications we made on the layers:

```
x = Flatten()(last_output)
x = Dense( 128, activation = "relu", kernel_regularizer = l2(hp["l2"]), bias_regularizer = l2(hp["l2"]) )(x)
x = Dropout(hp["dropout_rate"])(x)
x = Dense( 128, activation = "relu", kernel_regularizer = l2(hp["l2"]), bias_regularizer = l2(hp["l2"]) )(x)
x = Dropout(hp["dropout_rate"])(x)
x = BatchNormalization()(x, training = True )
x = Dense( 2, activation = "sigmoid")(x)
```

**Note:** At the end, we selected the epoch with the maximum validation accuracy. We are expecting and hoping that this will correlate with high values of test accuracy. Therefore, we believe to have chosen one of the best model weights.

#### 4.1.1 Model 1 / 6

This model will act as a main model for us, as the set of parameters used in this model are optimized to get a good validation accuracy. Therefore, we will be able to evaluate the performances of our future models by comparing it to Model 1.

The parameters used in model 1 are:

| | |
|---|---|
| learning_rate: 0.0005 | image_width = 256 |
| L2: 10 | image_height = 256 |
| dropout_rate: 0.2 | |

The results we have gotten are, using the epoch with the minimum validation loss:

Train & Validation: 138s - loss: 6.2954 - acc: 0.9389 - f1_m: 0.9350 - auc: 0.9769 - val_loss: 6.0596 - val_acc: 0.9590 - val_f1_m: 0.9600 - val_auc: 0.9819

| | | |
|---|---|---|
| Test: | loss: 6.1237 | test accuracy: 0.9103 |
| | f1_measure: 0.9103 | auc: 0.9617 |

Final Testing Accuracy: 0.9103

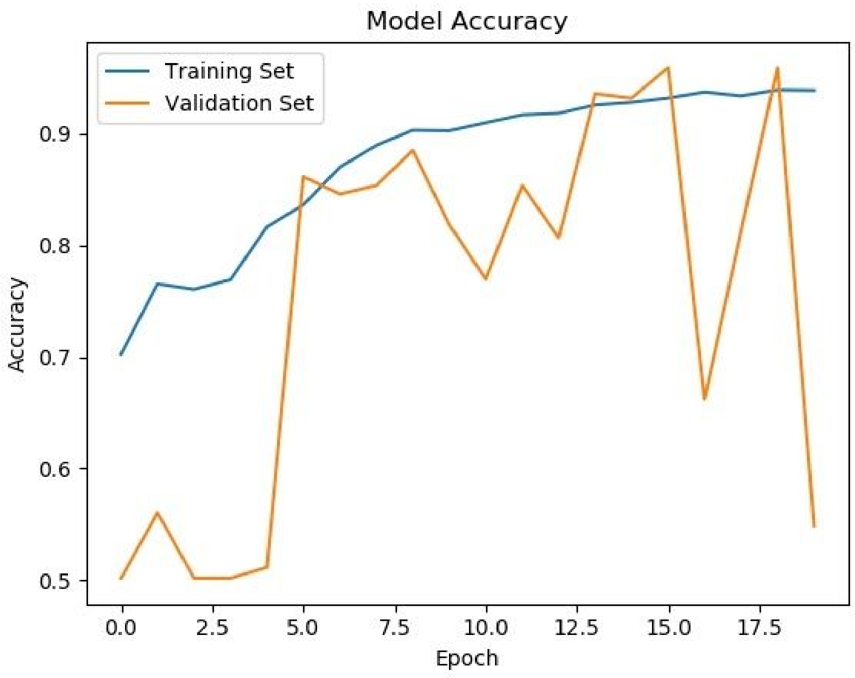

**Figure 4**

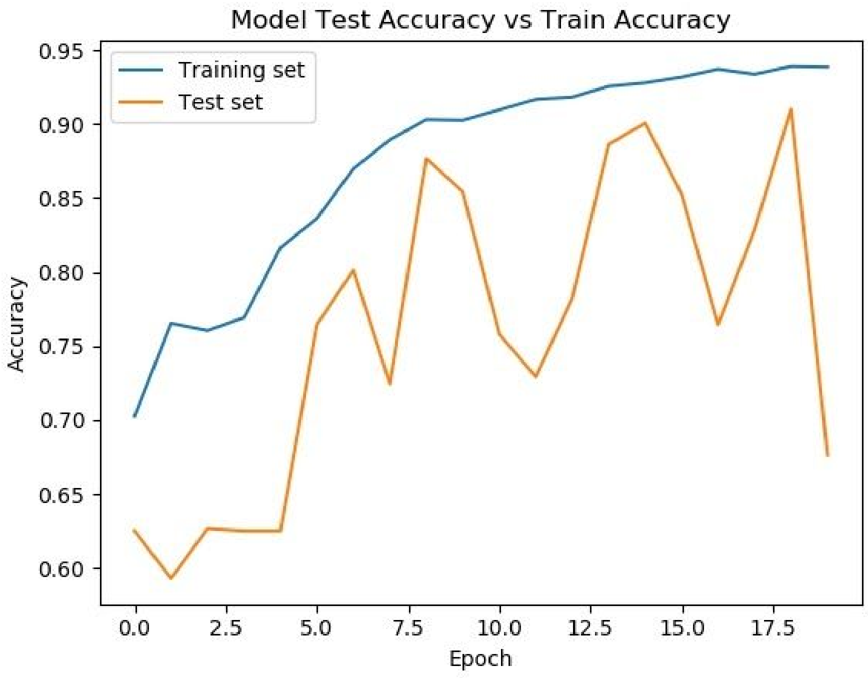

**Figure 5**

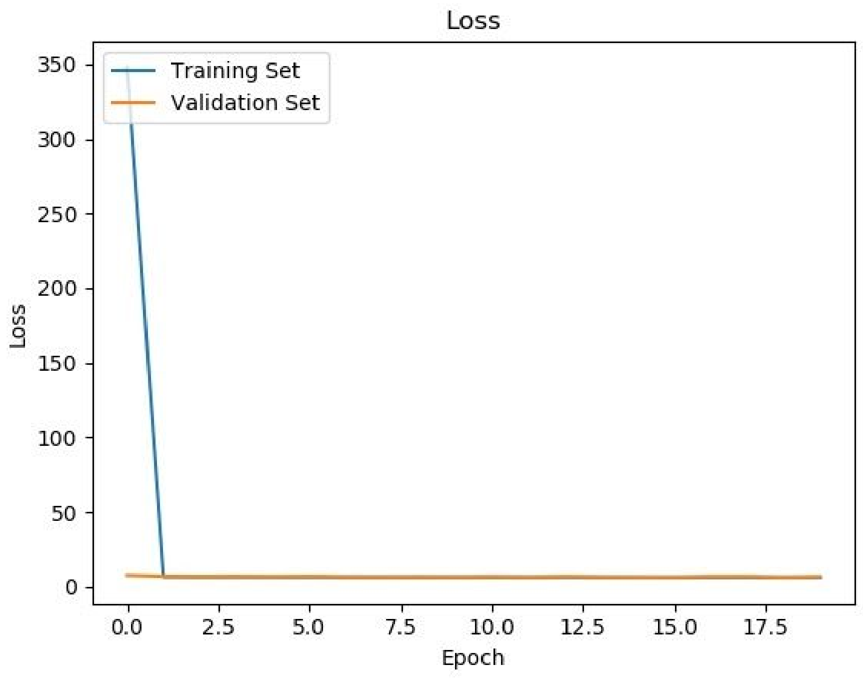

**Figure 6**

### 4.1.2 Model 2 / 6

In this model, we used the same parameters we used in Model 1, the only difference being that we dropped the hyperparameter L2 to 0.1 (it was equal to 10 in model 1), while keeping all the other parameters constant.

The results we have gotten are, using the epoch with the minimum validation loss:

<u>Train & Validation:</u> 133s - loss: 0.2149 - acc: 0.9674 - f1_m: 0.9652 - auc: 0.9801 - val_loss: 0.4207 - val_acc: 0.8984 - val_f1_m: 0.8925 - val_auc: 0.9606

<u>Test:</u> loss: 0.4699

test acc: 0.8894

f1_m: 0.8877

auc: 0.9423

<u>Final Testing Accuracy:</u> 0.8894

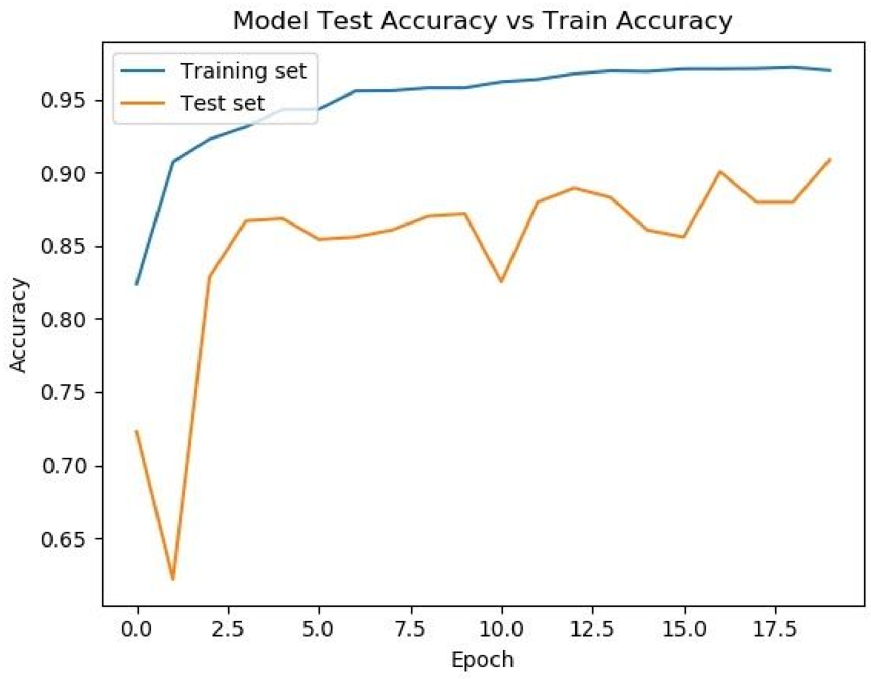

**Figure 7**

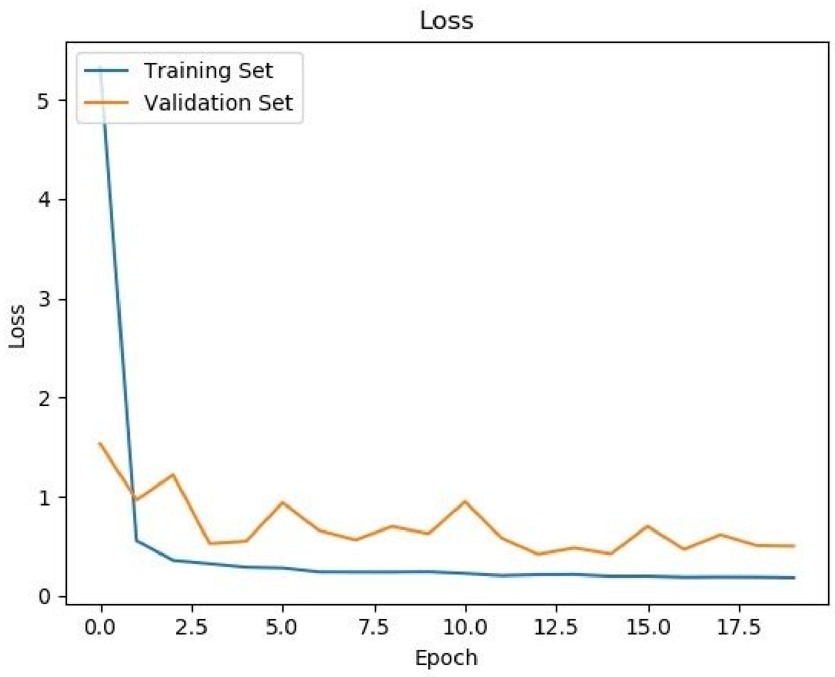

**Figure 8**

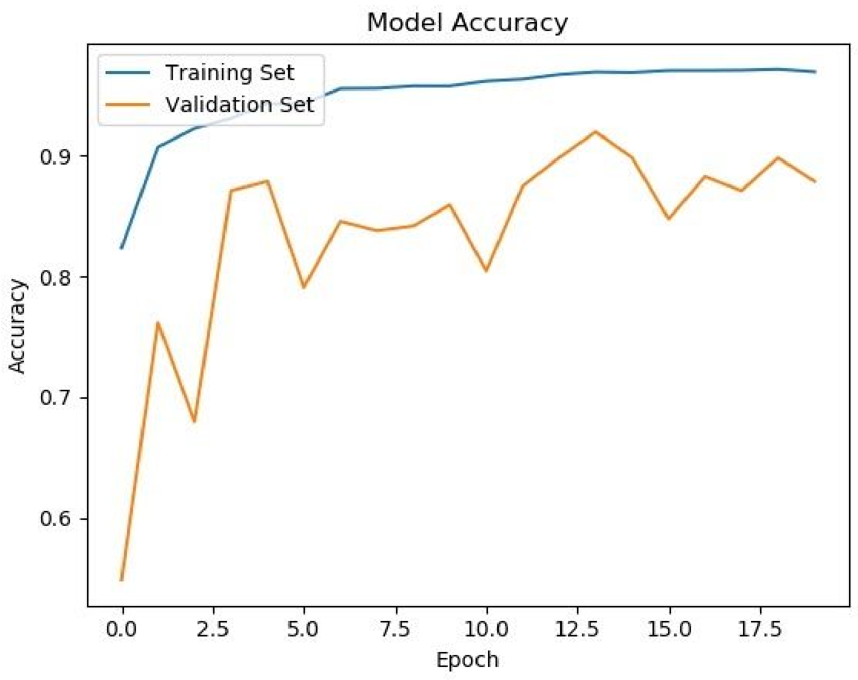

**Figure 9**

### 4.1.3 Model 3 / 6

In this model, we used the same parameters we used in Model 1, the only difference being that we increased the value of hyperparameter L2 to 1000 (it was equal to 10 in model 1), while keeping all the other parameters constant.

The results we have gotten are, using the epoch with the minimum validation loss:

Train & Validation: 142s - loss: 591.1704 - acc: 0.9366 - f1_m: 0.1463 - auc: 0.9778 - val_loss: 590.9334 - val_acc: 0.9082 - val_f1_m: 0.1336 - val_auc: 0.9640

Test: loss: 590.8828

test acc: 0.9327

f1_m: 0.2040

auc: 0.9744

Final Testing Accuracy: 0.9327

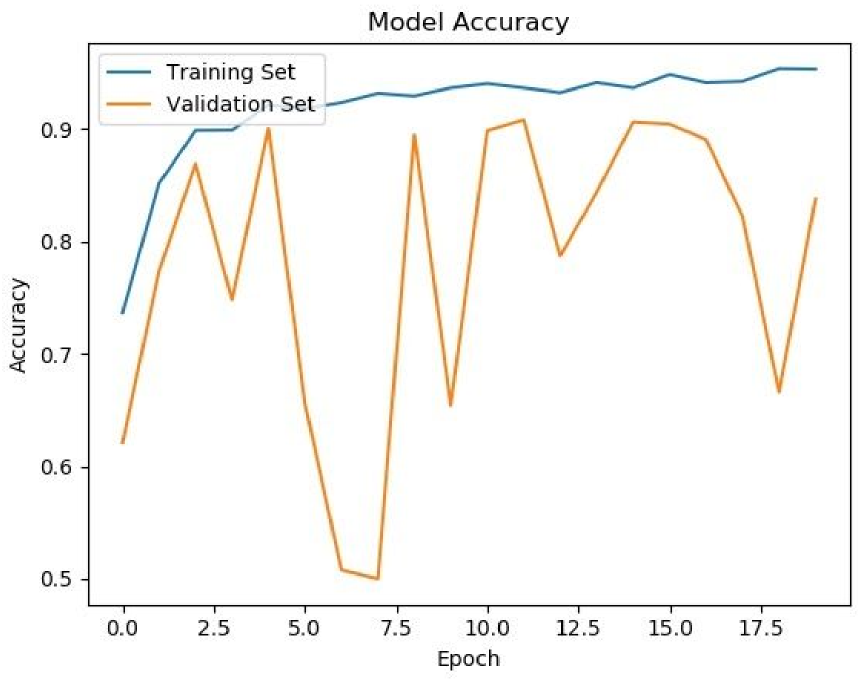

**Figure 10**

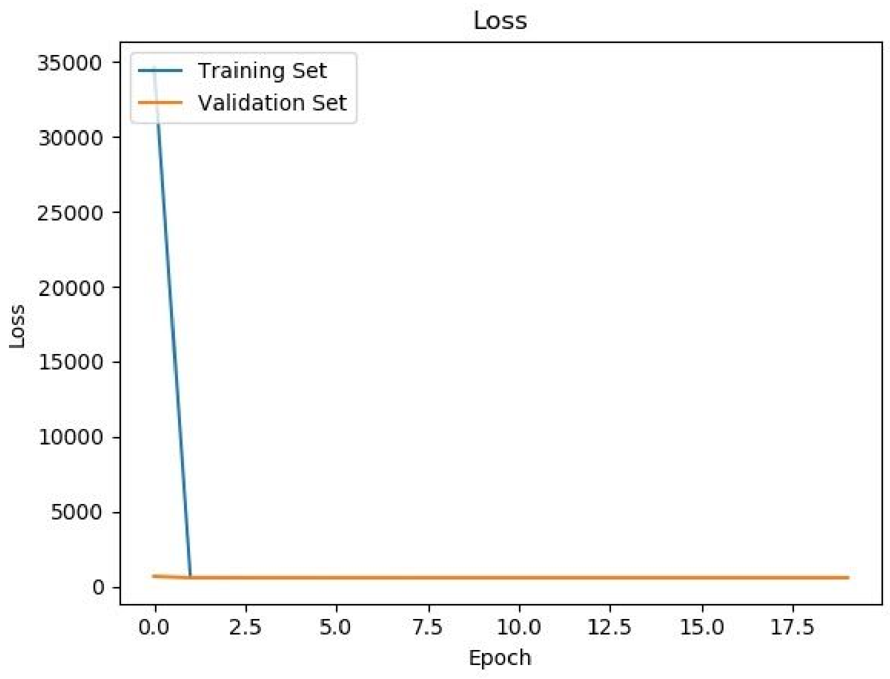

**Figure 11**

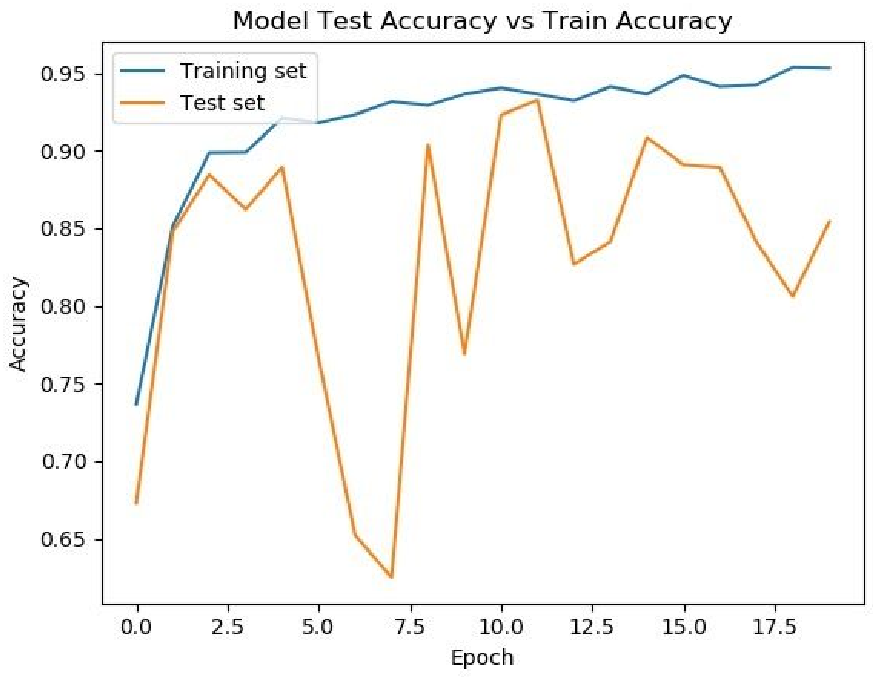

**Figure 12**

### 4.1.4 Model 4 / 6

In this model, we used the same parameters we used in Model 1, the only difference being that we decreased the number of neurons we have in the 2 dense layers that we added (they were 128 in Model 1, now they are 64), while keeping all the other parameters constant.

The results we have gotten are, using the epoch with the minimum validation loss:

Train & Validation: 136s - loss: 295.5866 - acc: 0.8749 - f1_m: 0.5195 - auc: 0.9441 - val_loss: 295.4045 - val_acc: 0.8594 - val_f1_m: 0.5583 - val_auc: 0.9369

Test: loss: 295.4605

test acc: 0.8462

f1_m: 0.5643

auc: 0.9160

Final Testing Accuracy: 0.8462

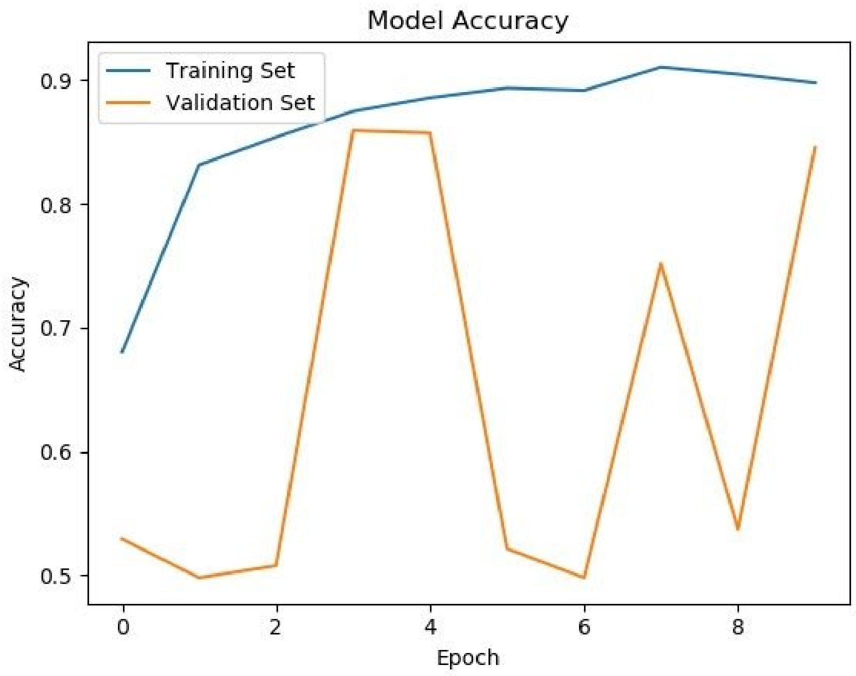

**Figure 13**

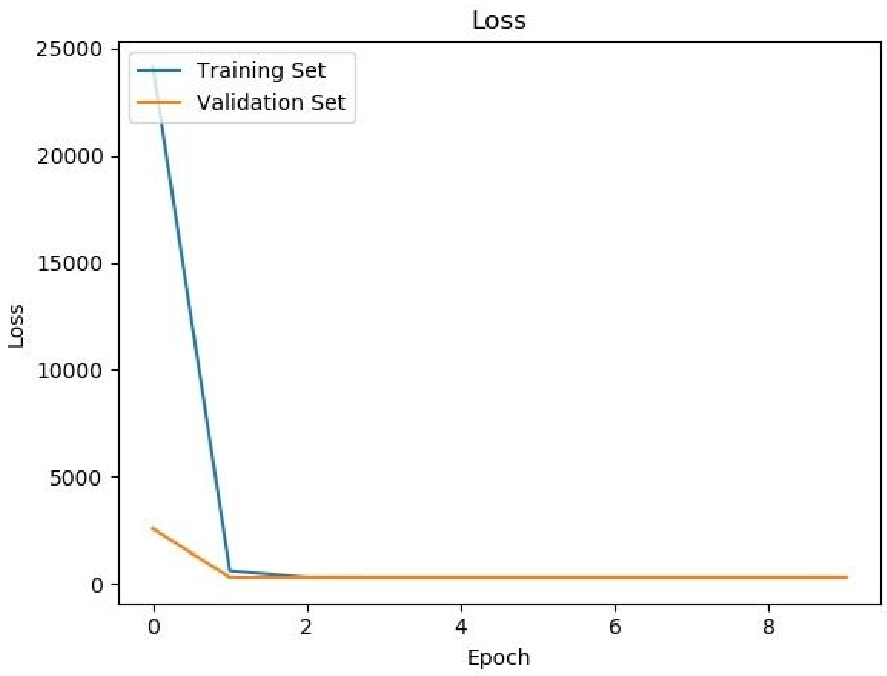

**Figure 14**

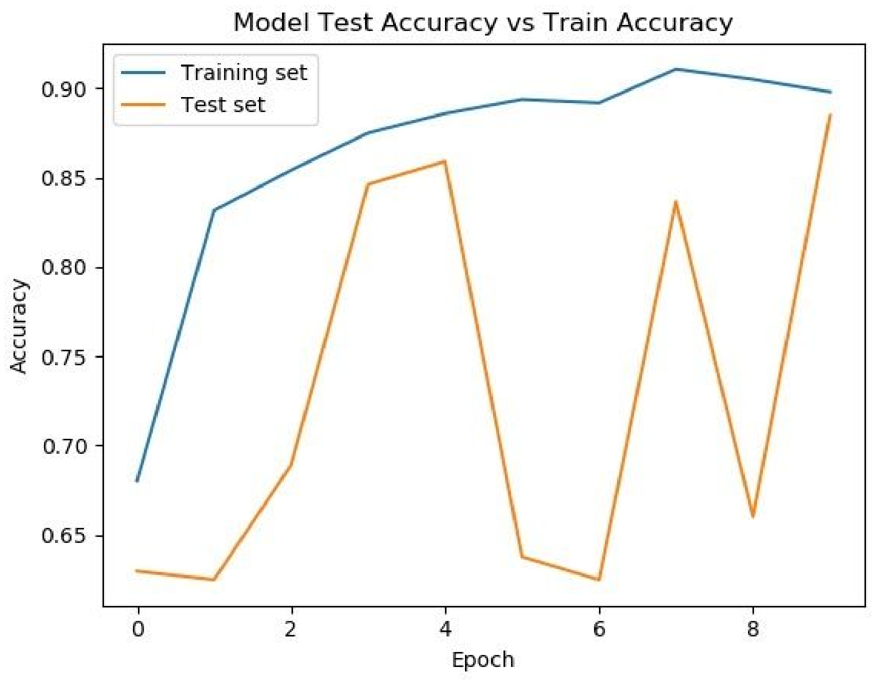

**Figure 15**

### 4.1.5 Model 5 / 6

In this model, we used the same parameters we used in Model 1, the only difference being that we decreased the image size to 128x128 (it was equal to 256x256 in model 1), while keeping all the other parameters constant.

The results we have gotten are, using the epoch with the minimum validation loss:

Train & Validation: 80s - loss: 67.1141 - acc: 0.7687 - f1_m: 0.7687 - auc: 0.7795 - val_loss: 67.2780 - val_acc: 0.6270 - val_f1_m: 0.6254 - val_auc: 0.7193

Test: loss: 67.1469

test acc: 0.7163

f1_m: 0.7153

auc: 0.7918

Final Testing Accuracy: 0.7163

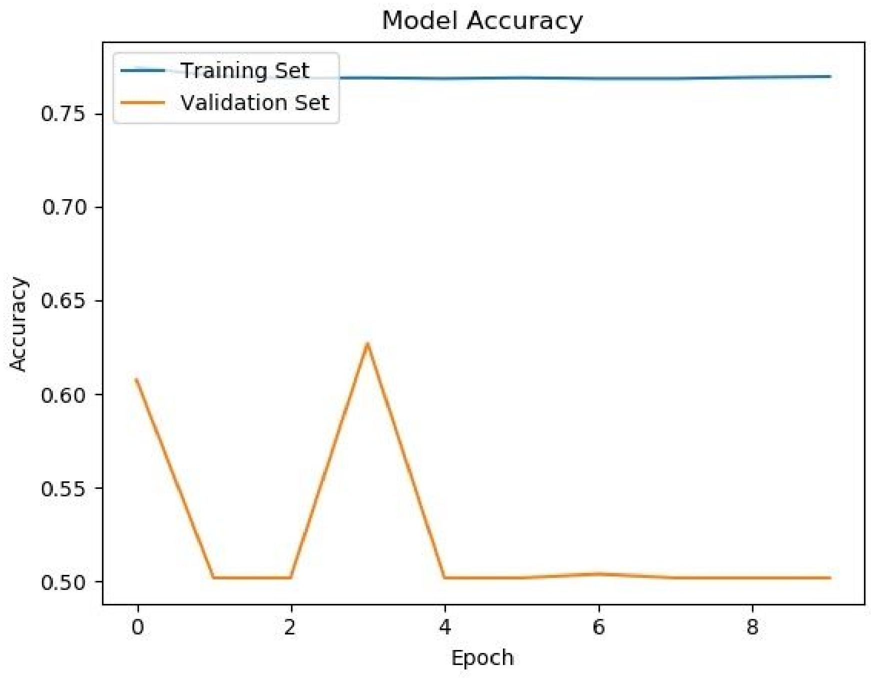

**Figure 16**

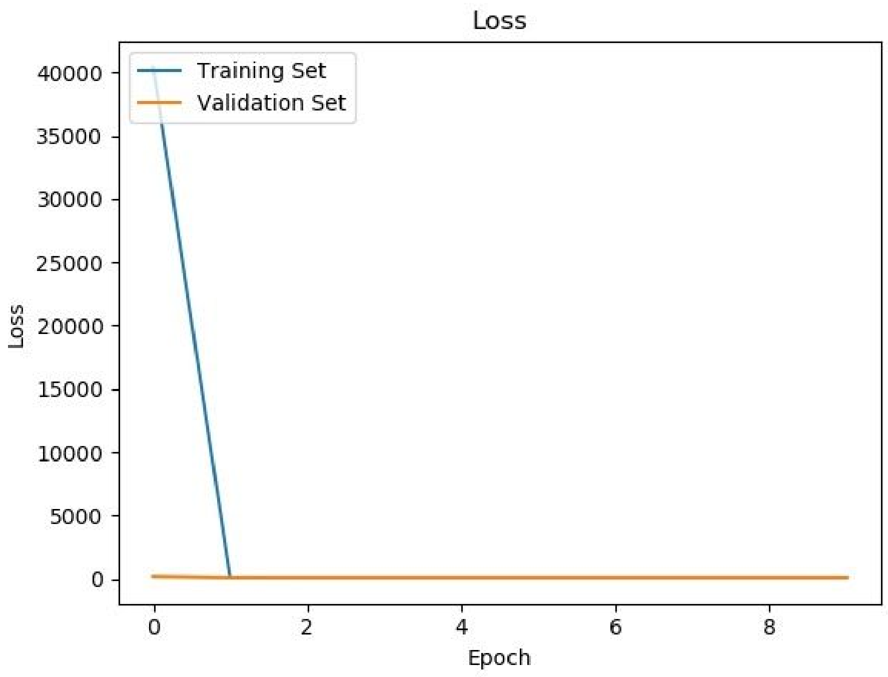

**Figure 17**

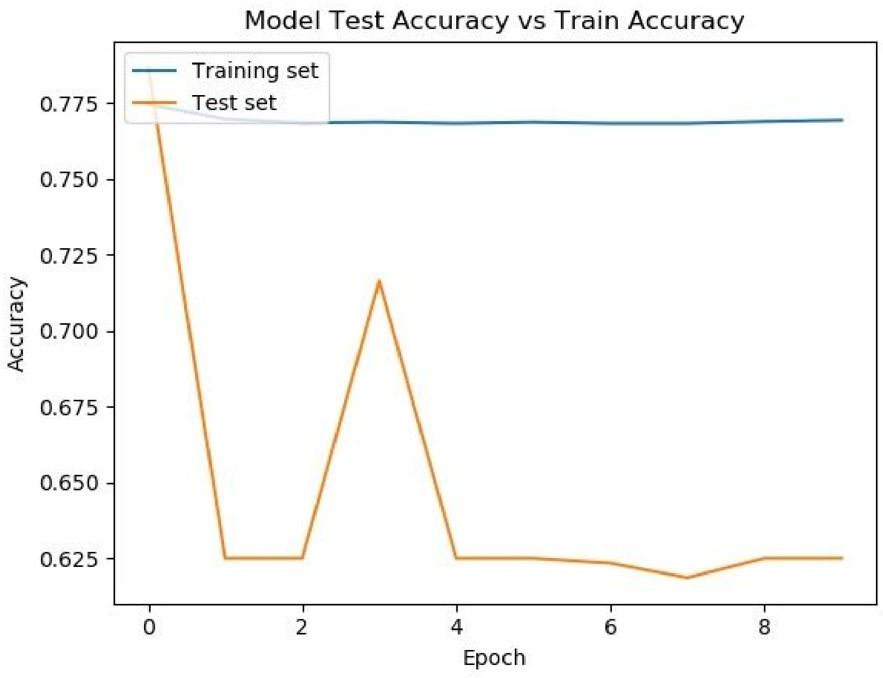

**Figure 18**

### 4.1.6 Model 6 / 6

In this model, we used the same parameters we used in Model 1, the only difference being that we increased the learning rate to 0.1 (it was equal to 0.0005 in model 1), while keeping all the other parameters constant.

The results we have gotten are, using the epoch with the minimum validation loss:

Train & Validation: 129s - loss: 23634553.7891 - acc: 0.2430 - f1_m: 0.6680 - auc: 0.5037 - val_loss: 23634554.3750 - val_acc: 0.4824 - val_f1_m: 0.6314 - val_auc: 0.4895

Test: loss: 23634554.2564

test acc: 0.3766

f1_m: 0.6325

auc: 0.4959

Final Testing Accuracy: 0.3766

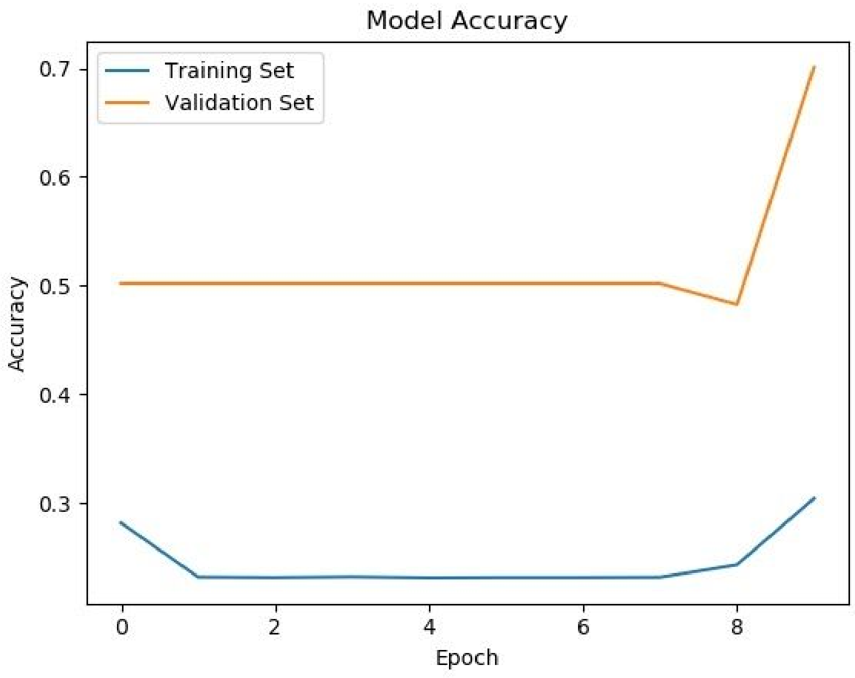

**Figure 19**

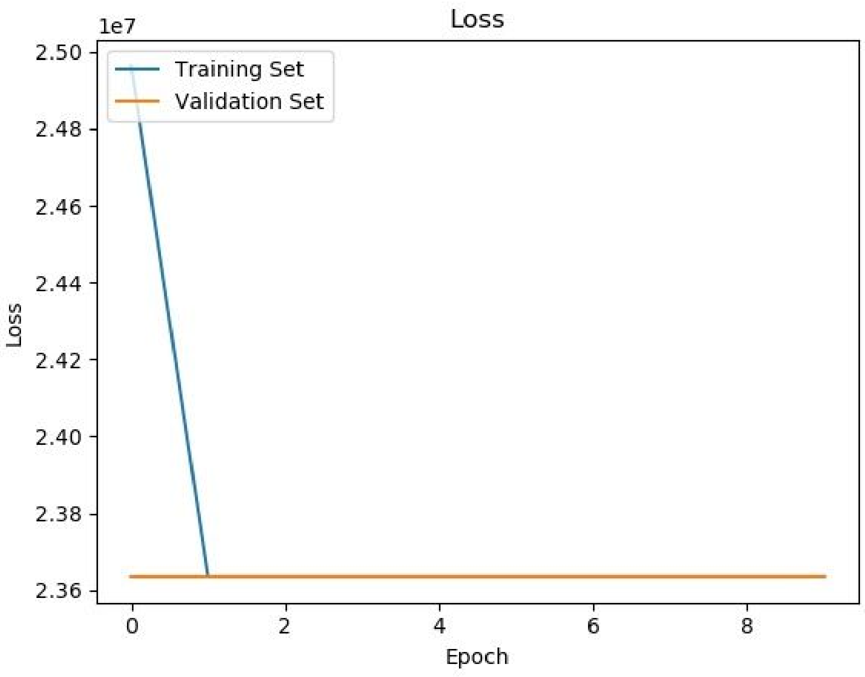

**Figure 20**

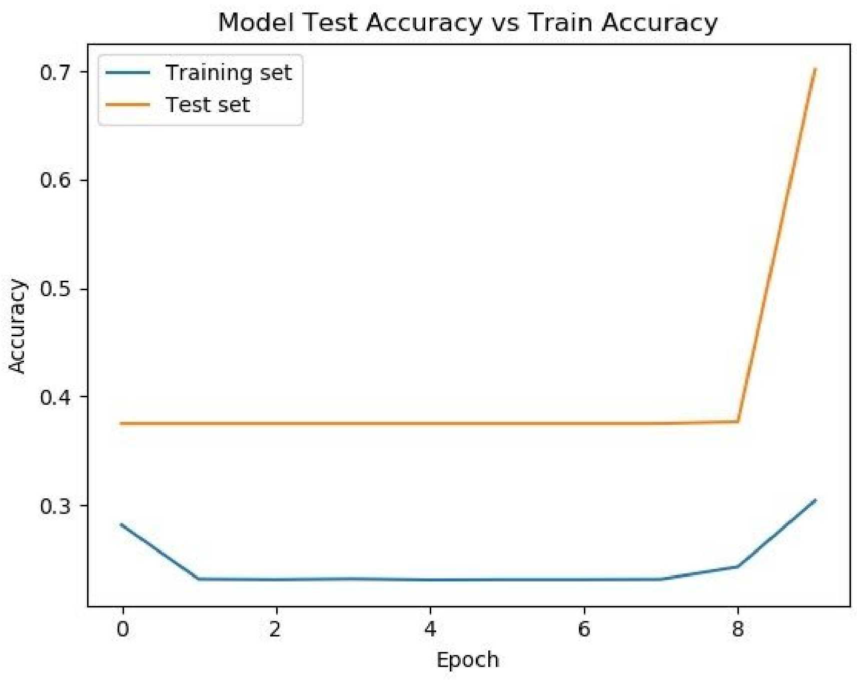

**Figure 21**

### 4.2 Custom CNN:

In this part, we created our own custom CNN to compare the results with Inception v3 network. When playing with hyperparameters, to get consistent results, we first created and saved a base model and loaded its initial weights in each iteration we changed a hyperparameter.

Our custom CNN was created using the Keras library. It has a total of 12 layers excluding the input layer. In the first six layers we have our convolutional and pooling layers. We do 3 convolutions and a pooling after each to hopefully extract features in the region. Then we have 2 dense layers each having a dropout layer after them. The dropout layers are there to provide regularization. In the end, we have a dense activation layer with 1 node. This is our output layer. Since our classification is binary, we can classify based on the activation or non-activation of one output node.

To prevent overfitting, other than the dropout layers, we also used L1 and L2 regularization in the dense layers. We tried trying different values for L2 to tune it and get better results.

Another parameter we tried to tune was the input image size. Having smaller input sizes reduces the number of trainable parameters in the layer and hence reduces the training time.

However, small image sizes also mean we are losing information so taking the sizes too small can reduce the performance of our model.

Using call backs, we managed to find the epoch with the lowest validation loss and stored its weights. We are hoping this will correlate with test accuracy. As a result, our selected weights should perform well on the test dataset. We reported this final accuracy for each model in this report.

Here is the summary of our model:
Model: "sequential_19"

```
_________________________________________________________________
Layer (type)              Output Shape              Param #
=================================================================
conv2d_55 (Conv2D)           (None, 254, 254, 32)      896
_________________________________________________________________
max_pooling2d_55 (MaxPooling (None, 127, 127, 32)      0
_________________________________________________________________
conv2d_56 (Conv2D)           (None, 125, 125, 32)      9248
_________________________________________________________________
max_pooling2d_56 (MaxPooling (None, 62, 62, 32)        0
_________________________________________________________________
conv2d_57 (Conv2D)           (None, 60, 60, 32)        9248
_________________________________________________________________
max_pooling2d_57 (MaxPooling (None, 30, 30, 32)        0
_________________________________________________________________
flatten_19 (Flatten)         (None, 28800)             0
_________________________________________________________________
dense_55 (Dense)             (None, 128)               3686528
_________________________________________________________________
dropout_37 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_56 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_38 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_57 (Dense)             (None, 1)                 65
=================================================================
Total params: 3,714,241
Trainable params: 3,714,241
Non-trainable params: 0
_________________________________________________________________
```

### 4.2.1 Model 1/5

```
"epochs" : 20,
"l2": 0.0001,
"l1": 0.00001,
"dropout_rate": 0.2
WIDTH = 256
HEIGHT = 256
```

Final testing accuracy is : 85.8552634716034 %

The following are the graphs we obtained over 20 epochs of training.

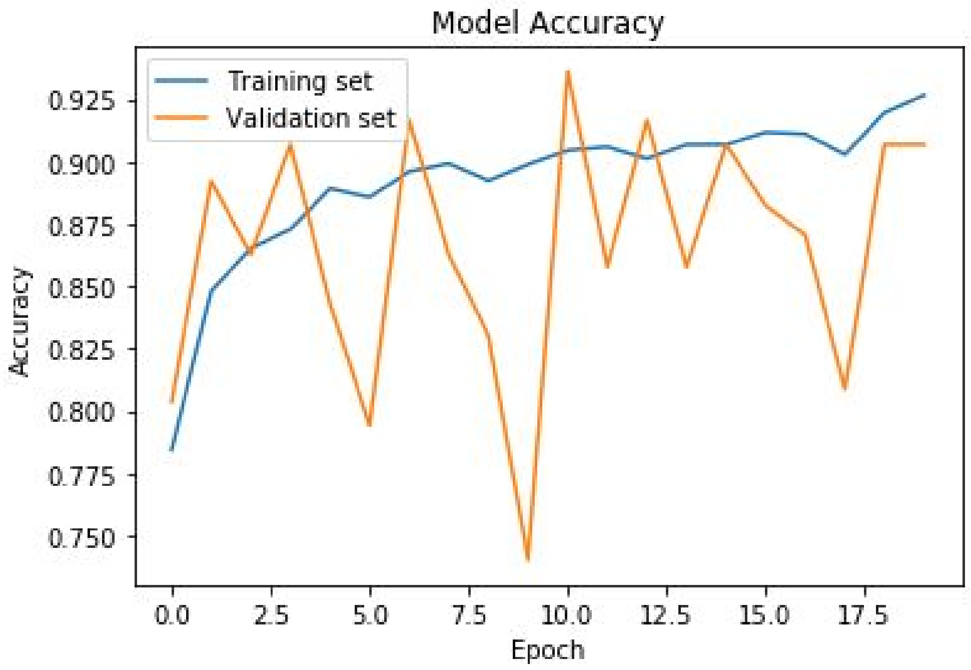

**Figure 22**

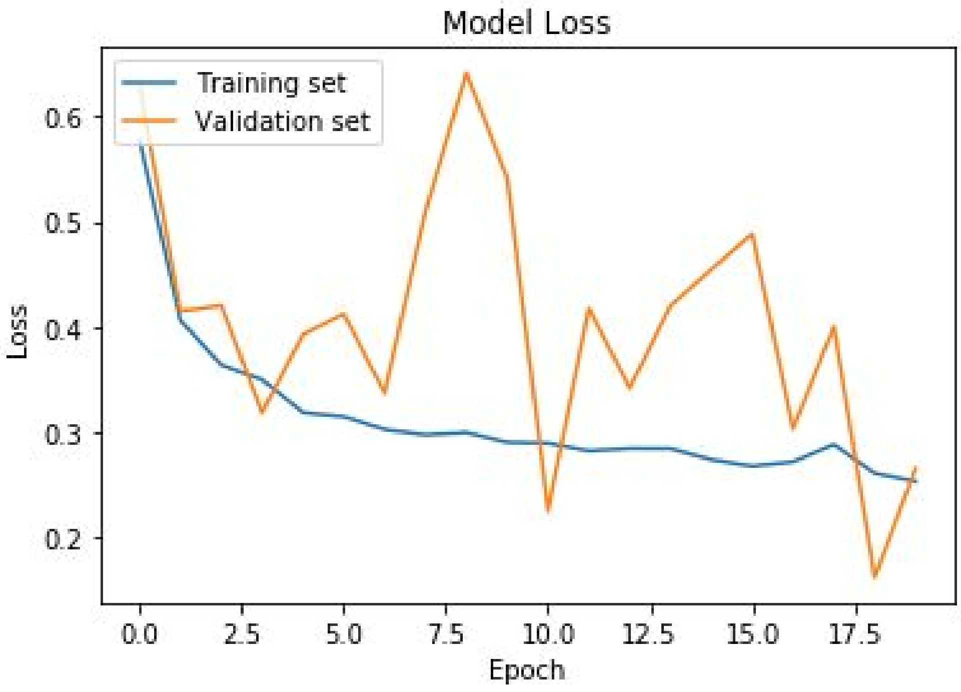

**Figure 23**

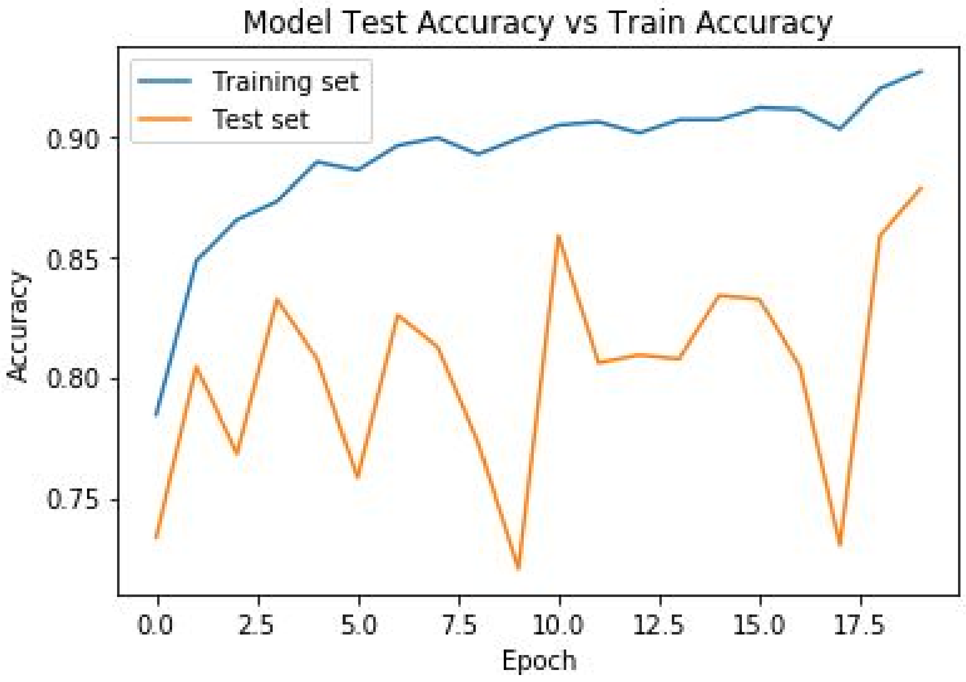

**Figure 24**

### 4.2.2 Model 2/5

"epochs" : 20,
"l2": 0.01,
"l1": 0.00001,
"dropout_rate": 0.2
WIDTH = 256
HEIGHT = 256

Final testing accuracy is : 81.41447305679321 %

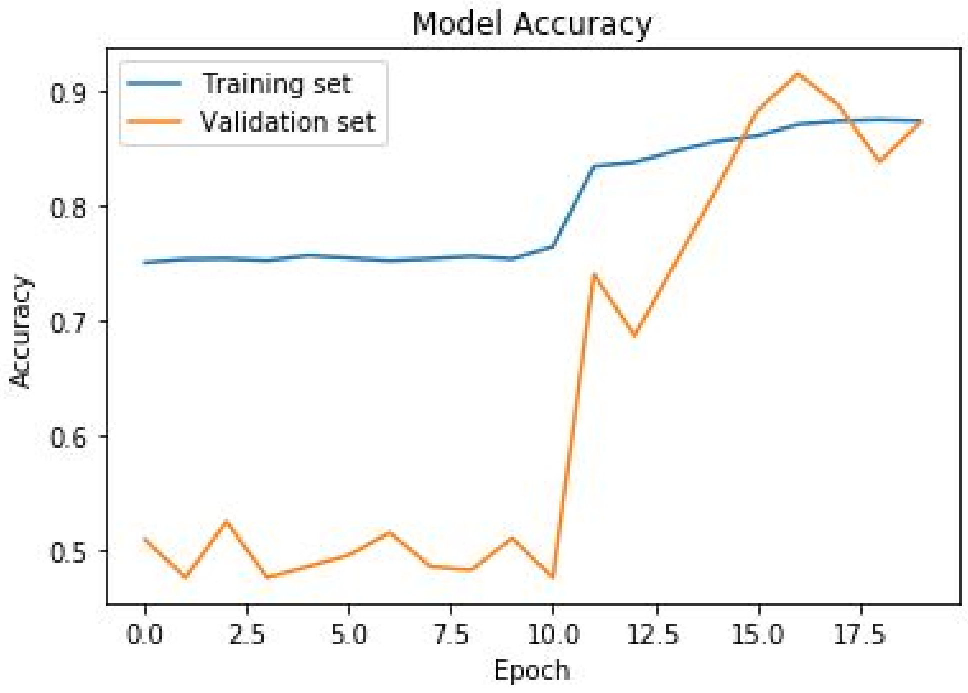

**Figure 25**

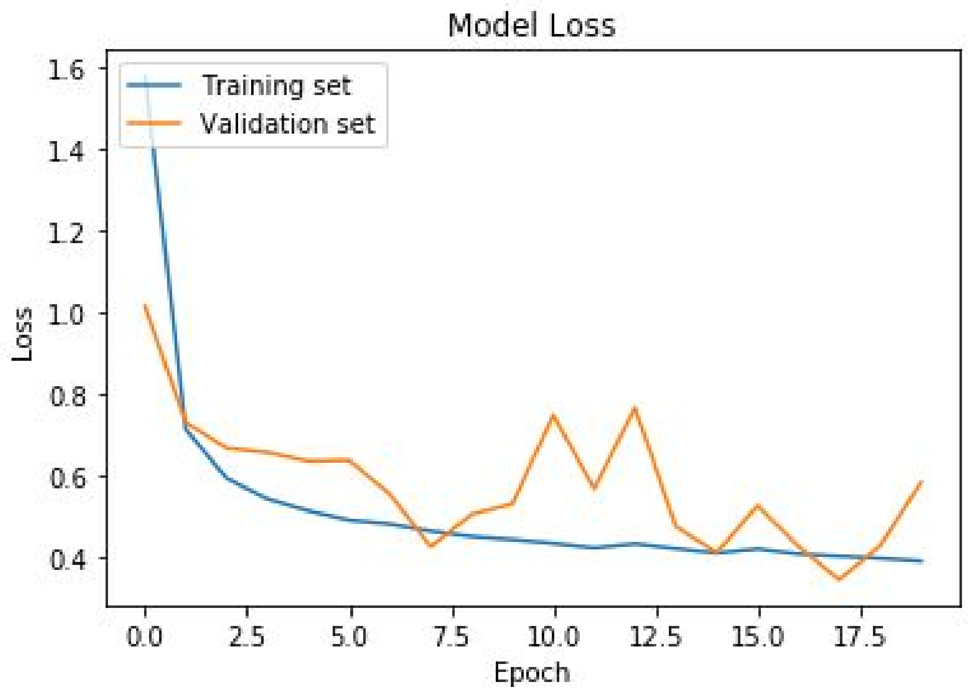

**Figure 26**

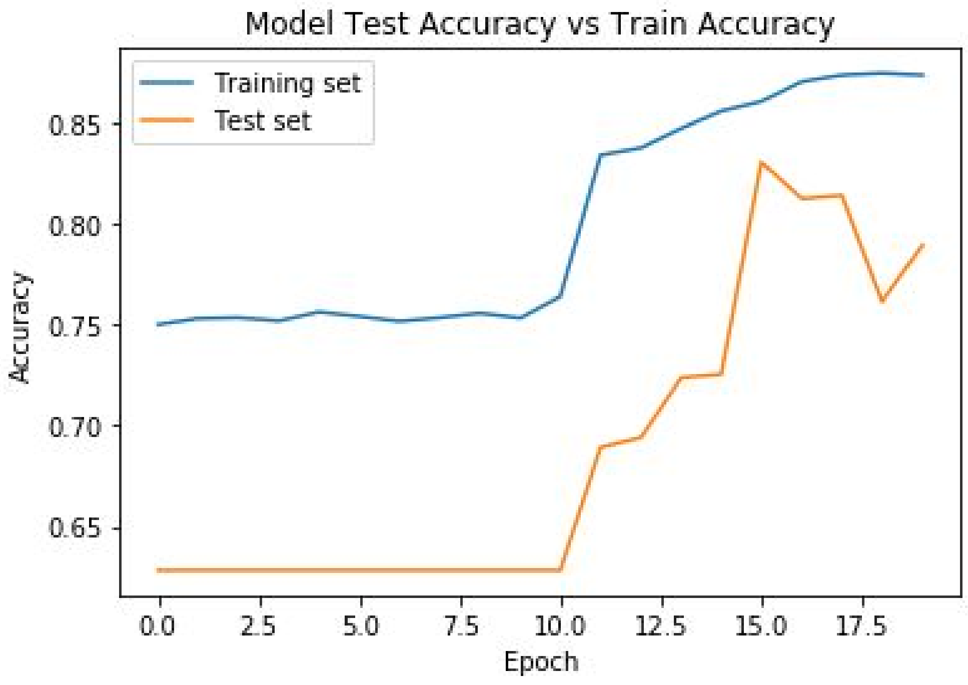

**Figure 27**

### 4.2.3 Model 3/5

"epochs" : 20,
"l2": 0.000001,
"l1": 0.00001,
"dropout_rate": 0.2
WIDTH = 256
HEIGHT = 256

Final testing accuracy is : 88.15 %

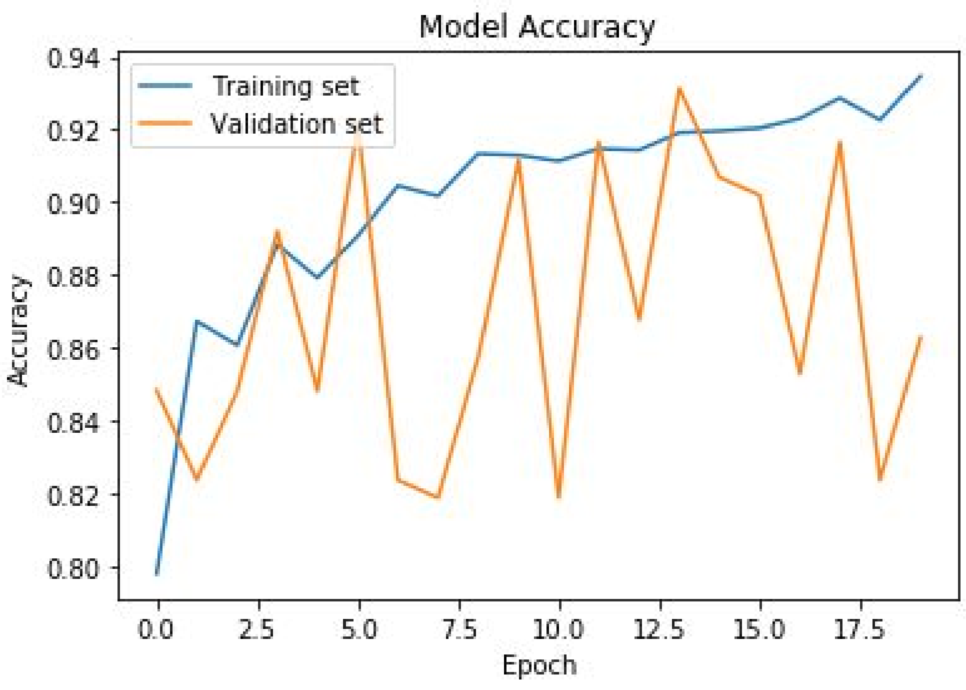

**Figure 28**

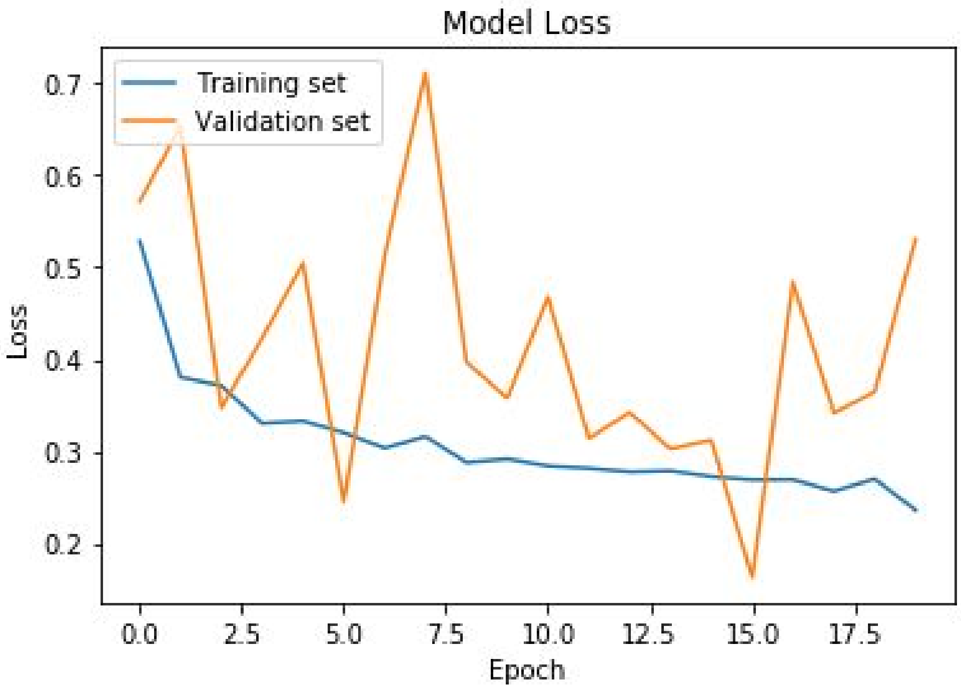

**Figure 29**

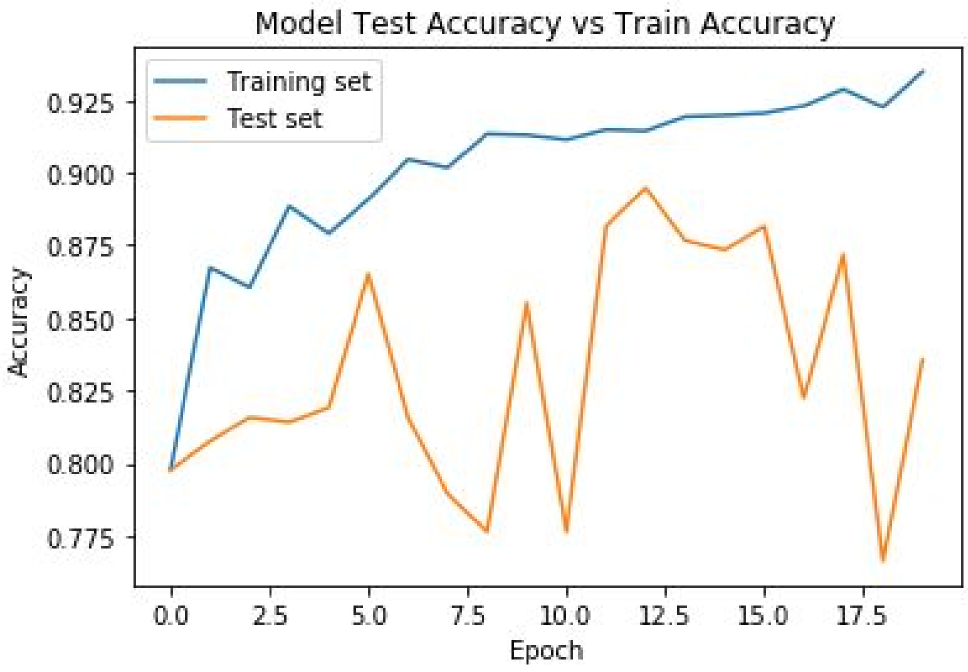

**Figure 30**

### 4.2.4 Model 4/5

This model has the same hyperparameters as model 1, but the input image size is 128x128.

```
"epochs" : 20,
"l2": 0.0001,
"l1": 0.00001,
"dropout_rate": 0.2
WIDTH = 128
HEIGHT = 128
```

Final testing accuracy is : 84.70 %

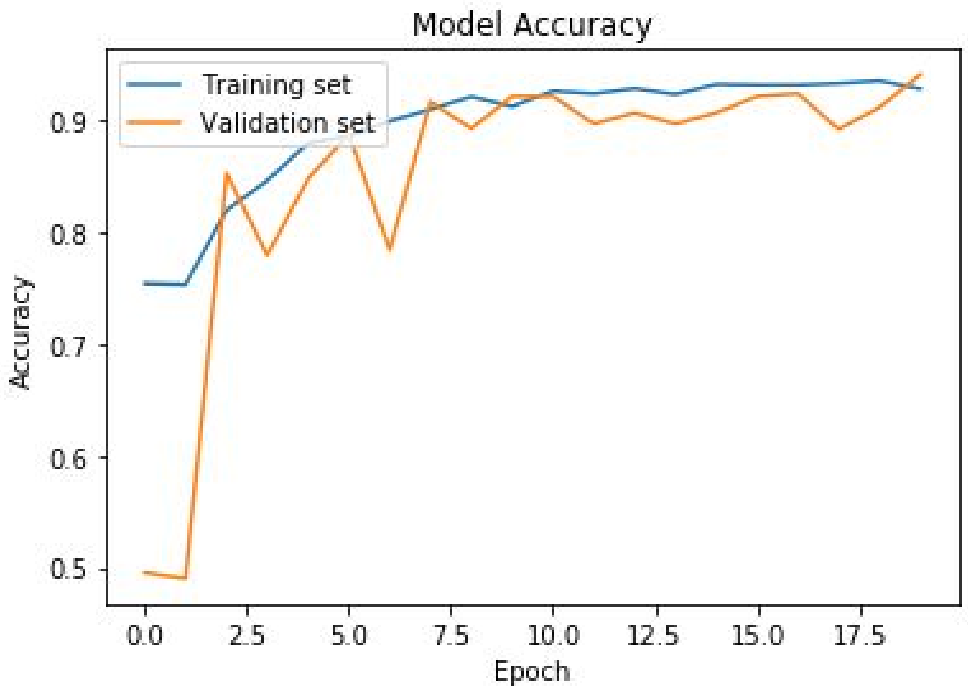

**Figure 31**

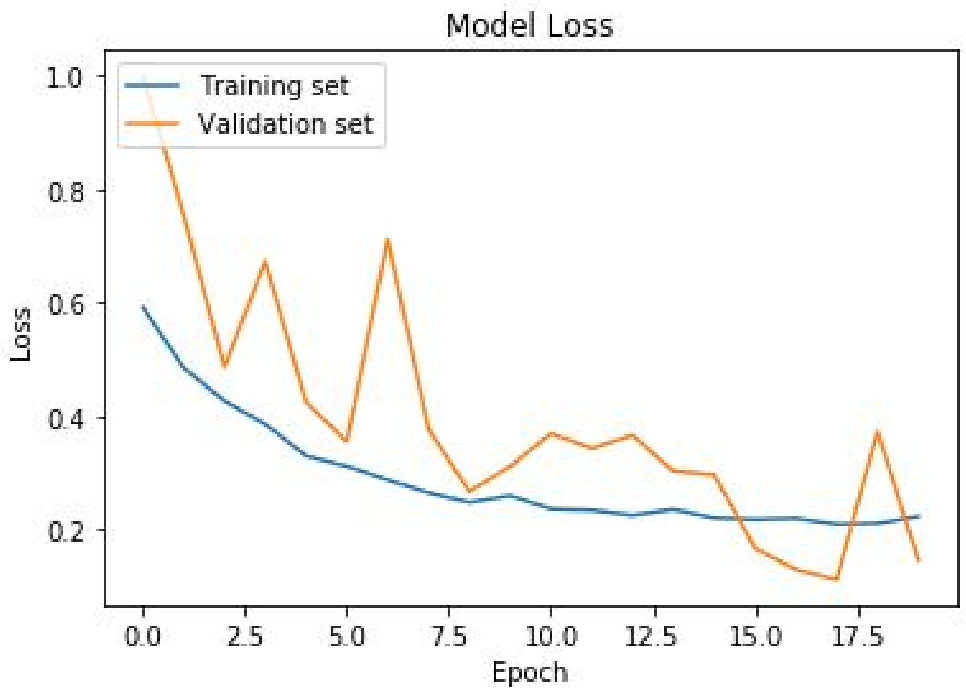

**Figure 32**

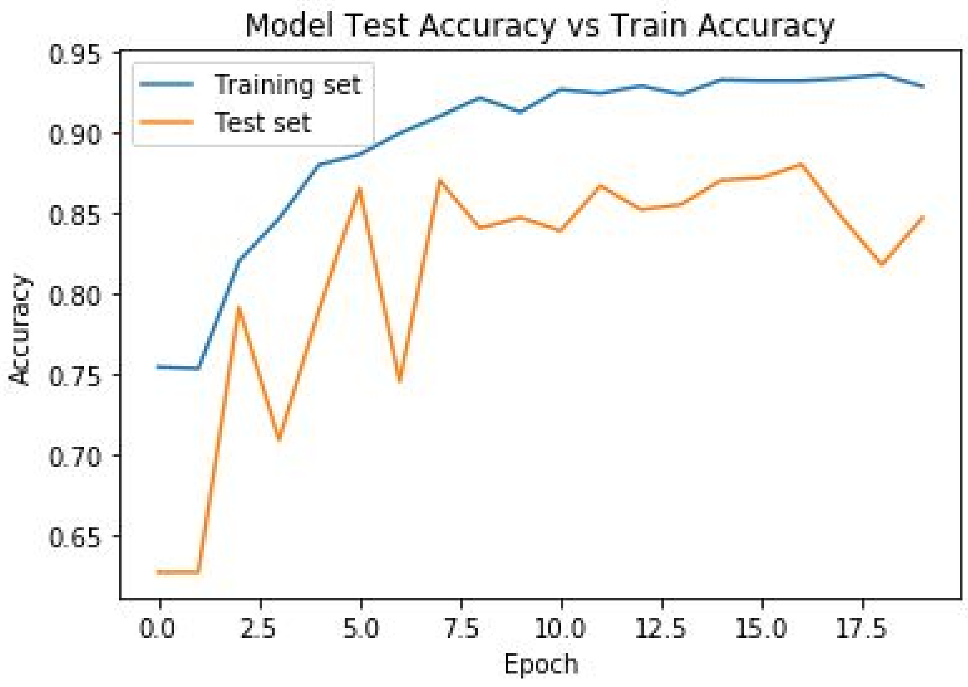

**Figure 33**

### 4.2.5 Model 5/5

"epochs" : 20,

"l2": 0.0001,

"l1": 0.00001,

"dropout_rate": 0.2

WIDTH = 64

HEIGHT = 64

Final testing accuracy is : 87.17%

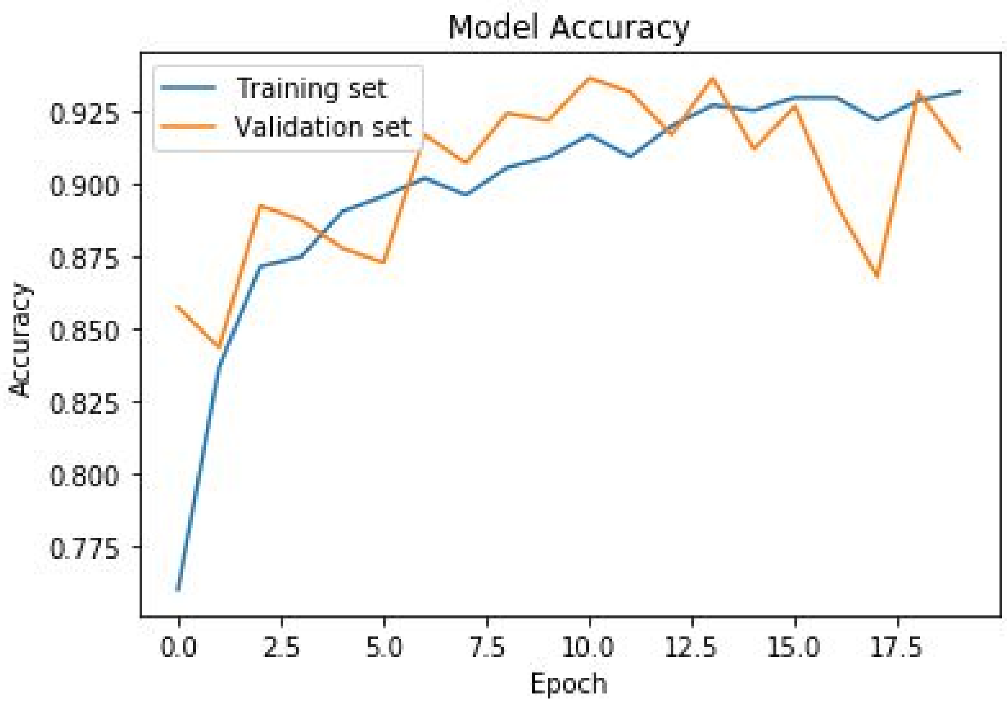

**Figure 34**

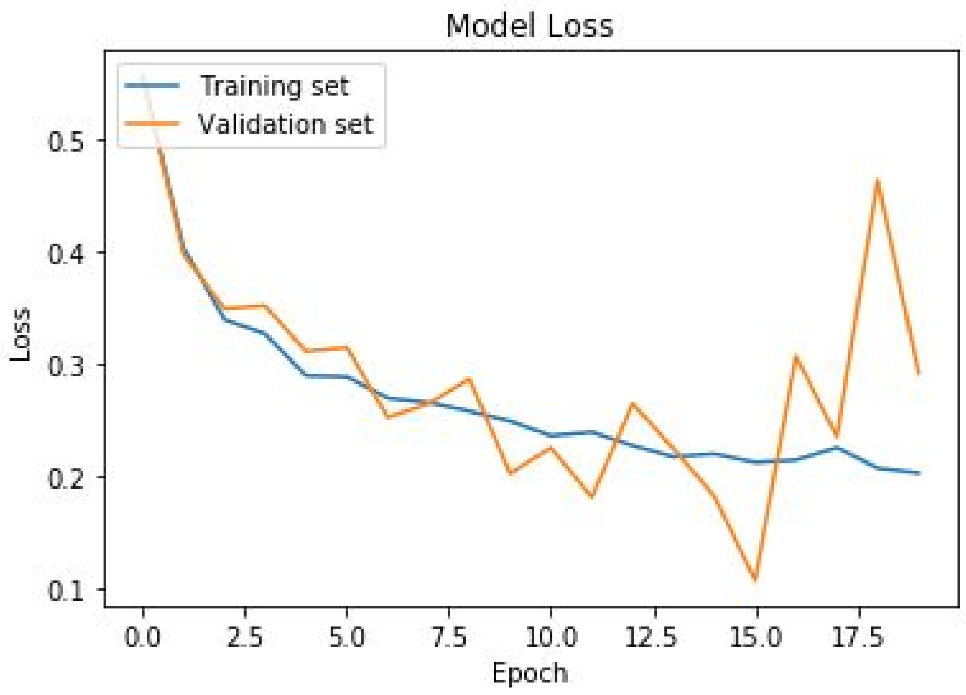

**Figure 35**

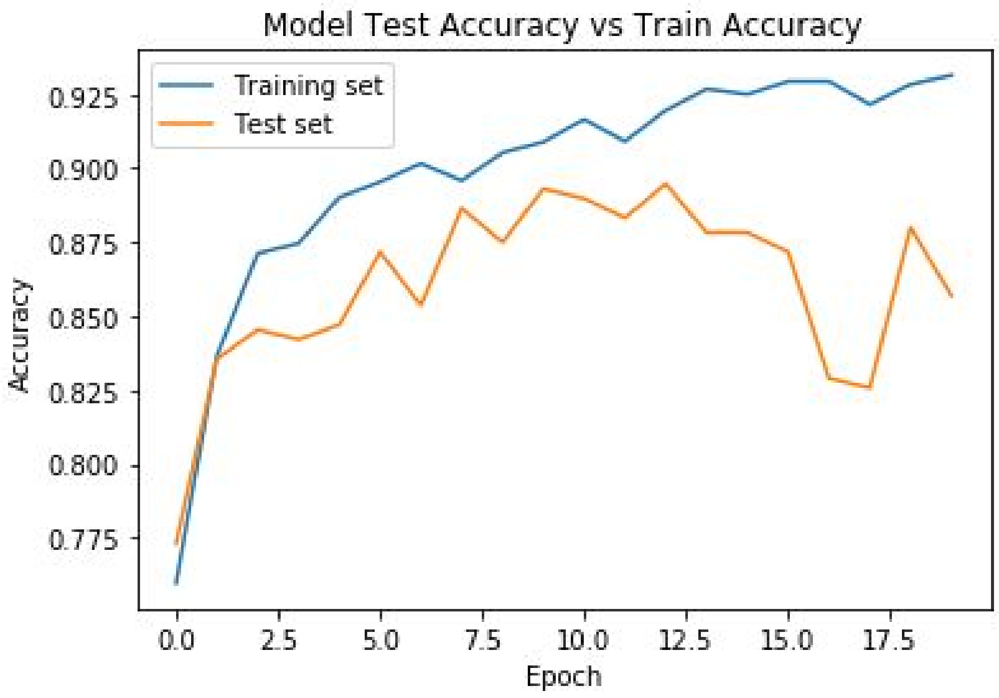

**Figure 36**

# 5. Discussion

## 5.1 Inception-v3

**Model 1**

This was the model we created with the optimum parameters. We found those values by trial and error. This is how we were able to get such a high value for test accuracy, ~91%. We could prevent over or underfitting with these values, Moreover, we took some precautions like batch normalizations, regularizations and dropouts. The graphs show that these methods have indeed worked.

Still, it should be noted that this is a model designed to be used in healthcare, and thus 91% is not actually high enough.

**Model 2**

Here, as we dropped L2 compared to our original model (divided it by 100), we expected to have some overfitting. That would mean that training accuracy increases while validation accuracy drops (and fluctuates). But, as can be seen from our graphs, we did not really experience overfitting. This might be due to the fact that we prevent such situations using batch normalization and dropout layers.

Still, as we got further away from the optimal value for L2, our test accuracy was not as good as the one we got in Model 1.

**Model 3**

As we increased L2 (multiplied it by 100), we expected to have some underfitting. In our case both validation and training accuracy dropped which means the model is now underfitting and the previous value of L2 was a better choice. This conclusion is made using the training and validation scores, however if we check the test score we see the highest test accuracy with 93%. This does not mean that the third model is the best out of the 6 presented Inception v3 models as explained in the note given in the introduction of  Inception v3 models.

**Model 4**

After decreasing the size of two dense layers from 128 to 64, training, validation and test accuracies decreased. This implies the case of underfitting and it means the size of 128 was a better choice compared to 64 for the size of dense layers.

**Model 5**

In this model the size of the images were decreased from 256x256 to 128x128. This reduced the training duration by 43% but introduced underfitting where the accuracy for each set decreased.

**Model 6**

This model uses increased learning rate compared to Model 1. The accuracy results show that the model did not converge to a stable solution and the weights in the network changed drastically with each batch of training examples. This is to be expected when using too large of a learning rate.

## 5.2 Custom CNN

**Model 1**

This was a relatively good model compared to model 3, but it lacks behind model 2. The resulting validation accuracies were fluctuating a lot which made taking the model in the last epoch unreliable. Thanks to our custom callback class, we managed to select a relatively successful model from one of the epochs. This resulted in 86% accuracy. The following were the hyperparameters we used. We also found a good L2 value which prevented overfitting but also allowed the network to achieve low train loss. The input image size is 256x256 which results in a lot of trainable parameters. For this reason, the average training time for each epoch was about 104 seconds. This is the same for models 2 and 3 since they have the same input size. However models 4 and 5 have lowered input sizes resulting in lower training times.

**Model 2**

This model had the lowest performance result among all the custom CNN models we trained with an accuracy of 81%. Since we increased the L2 regularization value, the model was not suffering from overfitting, and the validation accuracies were not fluctuating as much. However, as you can see, in the first 10 epochs, the model was not able to properly increase neither its training accuracy nor validation accuracy. It took the model 10 epochs to come over the regularizations limitations, and the accuracies rapidly increased, then finalized around 80%. This meant we could reliably take the model from last epoch, however we still used our custom callback to get the model from epoch with lowest validation loss. This resulted in the best performing model we have so far, which shows us that there is a fine line between underfitting and a model that requires a lot of epochs to get to high accuracy.

**Model 3**

In this model, we lowered the L2 regularization value even more to observe its effects. We saw that the fluctuations got even worse, however we were still getting good validation accuracies in some epochs. This shows that our simple model, even with such low L2 value was not able to overfit too much on the training data, unlike the inceptionV3 model. This was partly caused by the dropout layers and L1 regularization still preventing some of the overfitting.
As a result, we got a model that was unreliable, but using our custom callback class, we managed to get a model that had an accuracy of 88%. This was the best performing model we had so far.

**Model 4**

With this model, we were experimenting on the input image size. Lowering the input sizes lowers the number of trainable parameters and hence lowers the training time of the model. However, since we lose information while compressing the image, it could cause worse performance overall. However, in this experiment we did not observe that much difference in the performance, while the training times were reduced by about 25% resulting in 75 seconds for each epoch.

**Model 5**

With this model, we lowered the input image size even more. Now the input size is 64x64 and the average epoch training time is 54 seconds. This is a big improvement, it almost takes half the time of the original model 1. However this also meant we were losing 15/16 of our information compared to model 1. We observed that this did not affect the resulting performance negatively, it may have even had a positive effect. The resulting accuracy was 87% which was higher than what we got with model 1. This shows us that compressing the inputs can sometimes be a good way to reduce training times while possibly not affecting the performance negatively.

## 5.3 Comparison of Transfer Learning and Custom CNN Methods

After optimizing the hyperparameters for each model, we ended up with the highest accuracy score on our Transfer Learning model 3 with 93%. Compared to this, our highest accuracy score with Custom CNN was 88%. This shows that transfer learning with inceptionV3 was more successful than our custom CNN which was expected.

## 5.4 Final Remarks

Some of our models' validation accuracies were fluctuating too much to make them reliable if we were to take the model from the last epoch each time. However, these models were getting very high validation accuracies in some epochs. For this reason we created a custom callback class which allowed us to reliably select a good model from any of the epochs using their validation loss values. This made it possible to reliably get models with very high performances from cases that would normally not be possible.

We observed with the experiments from model 4 and 5 of our custom CNN that lowering the input sizes can sometimes reduce the training times significantly without affecting the overall performance of the model. This can be useful especially when there is a lot of data to process and we set the epoch numbers to high values like 100 or even 1000.

Note: While determining which model is “better” from the given epochs, we sorted validation loss values and chose the one which yields the least loss. In this manner, our aim was to maximize the test accuracy. However, several of the other Inception models yielded higher test accuracies despite their higher loss values. This might be caused by our validation data size being smaller compared to test data size. Also, these exceptions did not contradict the trend of correlating high validation accuracies and high test accuracies. Since we cannot predict which test accuracy will be obtained in the last epoch due to fluctuating results, we had to resort to using validation loss which yielded good results overall.

## 6. Conclusions

In this project, accuracy can be considered as a reliable performance metric since there is no huge imbalanced distribution of the class samples. We have chosen the best model as the inceptionV3 transfer learning model based on its performance metrics, mainly accuracy. However, we were surprised that a simple custom CNN like the one we implemented could give good results on such a complex problem. During the process of creating these models, we learned how to use transfer learning and create custom CNNs. We learned about different kinds of layers, their properties and effects on the performance of models.

We have found that it is possible to differentiate between x-rays of normal patients and patients with pneumonia using neural networks. Our model using transfer learning was able to successfully predict the class of the given xray 93% of the time after we optimized the hyperparameters and trained it on our train dataset. This is not a good enough result to make using this model as the only predictor of pneumonia feasible. Especially on a subject like healthcare, we expect more reliable models. However, this model can still be used as a helper by doctors to point out patients with pneumonia risk which they might have missed. This should help them get better results in the long run.

For future, as we have established we could get good results using transfer learning, we could try different pretrained models (such as VGG or ResNet) to try and get better results. Also, there is still room to improve the custom framework we add at the end of the pretrained models and to optimize the hyperparameters.

## 7. Appendix

To sum up, our project consisted of the following steps:

- Finding a project to do
- Finding a dataset to use
- Writing the proposal
- Researching models that could be used
- Creating the models using Python and Tensorflow, 6 models with Inception-v3 and 6 models with our own custom CNN
- Training the models on the dataset and recording their performances
- Plotting the performance metrics of the models based on our recordings
- Writing the report

Every member has collaborated for each and every part of the project. As a group, we had discussions and brainstorming at each step along the way.

In our proposal, we had already planned what we were going to do and when we were going to do them, like the fact that we would prepare a custom CNN and an Inception-v3 model after the demo. We worked according to that schedule we put on our proposal and we really did do what we said we would.

The implementation of the project was overall done by each member, using our separate computers. We then discussed the results over our group chat. We created and trained our inception-v3 models using the text editor Notepad++ and ran the code on Anaconda Prompt. The custom CNN models on the other hand were created using Spyder, as at that point we wanted to start using an IDE. As we shared our findings with each other, we were able to optimize our code and parameters by trial and error.

The report was written on Google Drive, by all the members simultaneously. We already had the trained models, their performance results and the corresponding graphs. We wrote the discussion and conclusion parts while sharing our ideas and having discussions among each other via our group chat.

## 8. References

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” International Journal of Computer Vision (IJCV), vol. 115, no. 3, pp. 211–252, 2015.

[2] Mooney, Paul. “Chest X-Ray Images (Pneumonia).” Kaggle, 24 Mar. 2018, www.kaggle.com/paultimothymooney/chest-xray-pneumonia

[3] sanwal092, “Intro to CNN using Keras to predict pneumonia,” *Kaggle*, 09-May-2020. [Online]. Available: https://www.kaggle.com/sanwal092/intro-to-cnn-using-keras-to-predict-pneumonia. [Accessed: 09-May-2020].