Chapter Five (Implementation)

Deep learning model

This section describes in detail the model implementation. The model has been written in python and developed in Kaggle, google collaborator (colab) in some experiments and features testing and finally MS Azure trying to get more computation power.

We use the convolution neural network to build our model and train this model on the official datasets of the SIIM-ISIC Melanoma Classification over multiple Challenges and use TensorFlow as a framework to implement the model.

Some techniques are used like making class weights balanced instead of unbalanced, transfer learning, fine tuning, learning rate techniques (constant LR, schedule LR or CLRs), standardize or normalize images or some regularization techniques. Execute a lot of experiments by changing all the hyperparameter like the mentioned techniques and other hyperparameters.

Why CNN

A convolutional neural network is very good at classifying images. This is one of the widely known and well-publicized facts, but why is it so, and why is it so in dealing with skin cancer images?

CNN Features

The number of parameters in a neural network grows rapidly with the increase in the number of layers. This can make training for a model computationally heavy (and sometimes not feasible). Tuning so many of parameters can be a very huge task. The time taken for tuning these parameters is diminished by CNNs.

CNNs are fully connected feed forward neural networks. CNNs are very effective in reducing the number of parameters without losing on the quality of models. Images have high dimensionality (as each pixel is considered as a feature) which suits the above-described abilities of CNNs. CNNs are trained to identify the edges of objects in any image.

All the layers of a CNN have multiple convolutional filters working and scanning the complete feature matrix and carry out the dimensionality reduction. This enables CNN to be a very apt and fit network for image classifications and processing.

Skin Cancer Images require:

When work in the problem of skin cancer you must:

- Make the huge number of parameters under control this is very useful when dealing with skin cancer images as you must have control on every feature inside the image.
- Treat each pixel as a feature so, when reducing the number of parameters, it must be ensured that this process occurs without losing the quality of the model to reveal that the image is benign or malignant.
- Take caution when dealing with edges as edges are very important factor in classifying skin cancer images (edges shape, texture, density, ...etc.). so, identifying and detecting images is very important.

From these points and the characteristics of CNN identified above, we have concluded that CNN is the best solution in our case.

Why TensorFlow

Whether you're an expert or a beginner, TensorFlow is an end-to-end platform that makes it easy for you to build and deploy ML models. This discussed in the next three points:

- **Easy model building**

  TensorFlow offers multiple levels of abstraction so you can choose the right one for your needs. Build and train models by using the high-level Keras API, which makes getting started with TensorFlow and machine learning easy.

If you need more flexibility, eager execution allows for immediate iteration and intuitive debugging. For large ML training tasks, use the Distribution Strategy API for distributed training on different hardware configurations without changing the model definition.

- Robust ML production anywhere

  TensorFlow has always provided a direct path to production. Whether it's on servers, edge devices, or the web, TensorFlow lets you train and deploy your model easily, no matter what language or platform you use.

  Use TensorFlow Extended (TFX) if you need a full production ML pipeline. For running inference on mobile and edge devices, use TensorFlow Lite. Train and deploy models in JavaScript environments using TensorFlow.js.

- Powerful experimentation for research

  Build and train state-of-the-art models without sacrificing speed or performance. TensorFlow gives you the flexibility and control with features like the Keras Functional API and Model Subclassing API for creation of complex topologies. For easy prototyping and fast debugging, use eager execution.

  TensorFlow also supports an ecosystem of powerful add-on libraries and models to experiment with, including Ragged Tensors, TensorFlow Probability, Tensor2Tensor and BERT.

About used dataset

The dataset is combination of the following:

- ISIC 2020 challenge dataset

  The dataset contains 33,126 dermoscopic training images of unique benign and malignant skin lesions from over 2,000 patients. Each image is associated with one of these individuals using a unique patient identifier. All malignant diagnoses have been confirmed via histopathology, and benign diagnoses have been confirmed using either expert agreement, longitudinal follow-up, or histopathology. A thorough publication describing all features of this dataset is available in the form of a pre-print that has not yet undergone peer review.

The dataset was generated by the International Skin Imaging Collaboration (ISIC) and images are from the following sources: Hospital Clínic de Barcelona, Medical University of Vienna, Memorial Sloan Kettering Cancer Center, Melanoma Institute Australia, University of Queensland, and the University of Athens Medical School.

- ISIC 2019 challenge dataset

  25,331 images are available for training across 8 different categories (Melanoma, Melanocytic nevus, Basal cell carcinoma, Actinic keratosis, Benign keratosis (solar lentigo / seborrheic keratosis / lichen planus-like keratosis), Dermatofibroma, Vascular lesion, Squamous cell carcinoma, None of the others). Additionally, the test dataset contains an additional outlier class not represented in the training data, which developed systems must be able to identify.

Balancing class weights

As we mentioned above our dataset consist of all categories (types) of skin cancer and all of them divided into malignant (melanoma) which represent 4922 image (8.60% of total) and benign (other types) which represent 52,302 image that causes imbalance class weights and we will discuss it in three parts what, why and how.

- **What is imbalance Class Weights?**
  Class imbalance is a problem that occurs in machine learning classification problems. It merely tells that the target class's frequency is highly imbalanced, i.e., the occurrence of one of the classes is very high compared to the other classes present. In other words, there is a bias or skewness towards the majority class present in the target. Suppose we consider a binary

classification where the majority target class has 10000 rows, and the minority target class has only 100 rows. In that case, the ratio is 100:1, i.e., for every 100-majority class, there is only one minority class present. This problem is what we refer to as class imbalance. Some of the general areas where we can find such data are fraud detection, churn prediction, medical diagnosis, e-mail classification, etc.

We will be working on our dataset to understand class imbalance properly. In the below image, you can see the number of images in each class in our target variable.

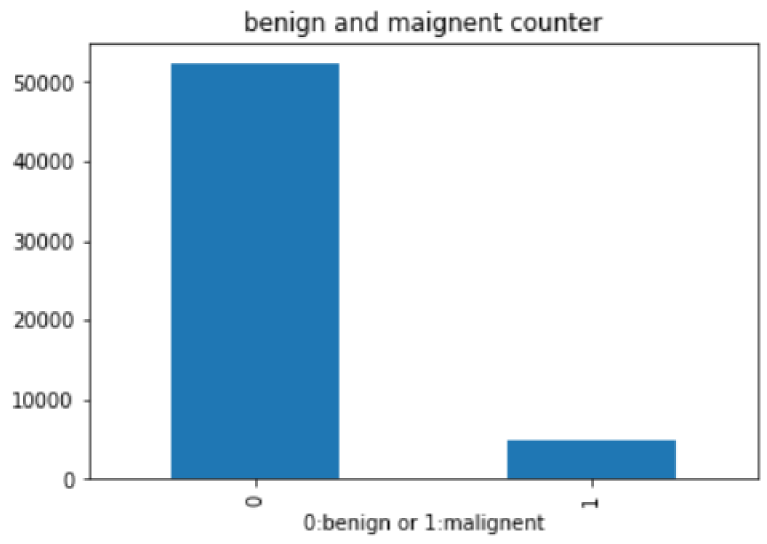


Figure 5.1: benign and malignant counter

Here,

0: signifies that the patient didn’t have a melanoma.

1: signifies that the patient had a melanoma.

From the counter, we can see that there are only 4922 image (8.6 %) of patients who had a melanoma. So, this is a classic class imbalance problem.

- **Why is it essential to deal with class imbalance?**

So far, we got the intuition about class imbalance. But why is it necessary to overcome this, and what problems does it create while modeling with such data?

Most machine learning algorithms assume that the data is evenly distributed within classes. In the case of class imbalance problems, the extensive issue is that the algorithm will be more biased towards predicting the majority class (Malignant our case). The algorithm will not have enough data to learn the patterns present in the minority class (benign).

Let’s take a real-life example to understand this better.

Consider you have shifted from your hometown to a new city, and you been living here for the past month. When it comes to your hometown, you will be very familiar with all the locations like your home, routes, essential shops, tourist spots, etc. because you had spent your whole childhood there. But when it comes to the new city, you would not have many ideas about where each location exactly is, and the chances of taking the wrong routes and getting lost will be very high. Here, your hometown is our majority class, and the new city is the minority class.

Similarly, this happens in class imbalance. The model has adequate information about the majority class but insufficient information about our minority class. That is why there will be high misclassification errors for the minority class.

When we train our model without solving this problem, we got 8% wrong prediction images which is so good but when we use AUC matrix we got 42% error rate this discussing the impact of imbalance class weights.

- **How to use class weights to solve this imbalance classes**

Most machine learning algorithms are not very useful with biased class data. But we can modify the current training algorithm to consider the skewed distribution of the classes. This can be achieved by giving different weights to both the majority and minority classes. The difference in weights will influence the classification of the classes during the training phase. The whole purpose is to penalize the misclassification made by the minority class by setting a higher-class weight and at the same time reducing weight for the majority class.

To make this a bit clear, we will be reviving the city example we considered earlier.

Please think of it this way that the last month you have spent in the new city, instead of going out when it is needed, you spent the whole month exploring the city. You spent more time understanding the city routes and places the entire month. Giving more time to research will help you to understand the new city better, and the chances of getting lost will reduce. And this is precisely how class weights work. During the training, we give more weightage to the minority class in the cost function of the algorithm so that it could provide a higher penalty to the minority class and the algorithm could focus on reducing the errors for the minority class.

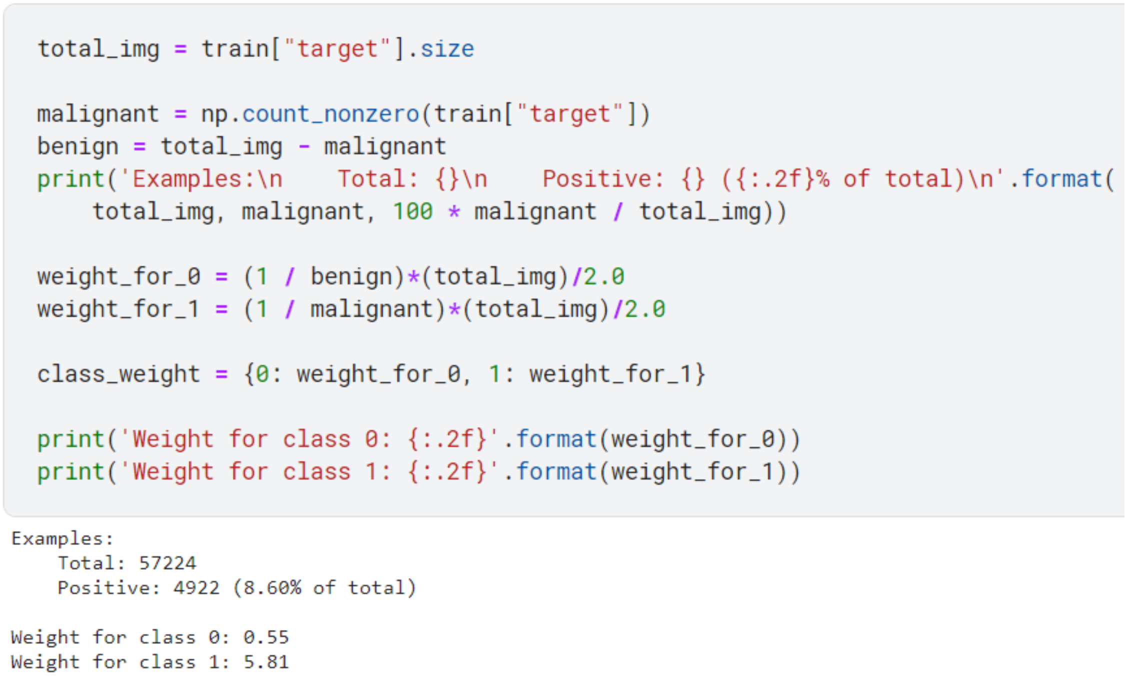
There is a threshold to which you should increase and decrease the class weights for the minority and majority class respectively. If you give very high-class weights to the minority class, chances are the algorithm will get biased towards the minority class, and it will increase the errors in the majority class.

In our case we can increase the class weights of minority class (malignant) as if we tend to zero false negative this is very useful in our case even if false positive matrix increases a little. This comes from the logic that if a patient has a malignant tumor, we must diagnose it correctly, this is a higher priority than the diagnosis of a benign condition.

When using CNN in TensorFlow the fit method has an in-built parameter “class_weight” which helps us optimize the scoring for the minority class just the way we have learned so far.

By default, the value of class_weight=None, i.e., both the classes have been given equal weights. Other than that, we can either give it as ‘balanced’ or we can pass a dictionary that contains manual weights for both the classes.

To be more precise, the formula to calculate a balance class weight in our case is:



```
total_img = train["target"].size

malignant = np.count_nonzero(train["target"])
benign = total_img - malignant
print('Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total_img, malignant, 100 * malignant / total_img))

weight_for_0 = (1 / benign)*(total_img)/2.0
weight_for_1 = (1 / malignant)*(total_img)/2.0

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

```
Examples:
    Total: 57224
    Positive: 4922 (8.60% of total)

Weight for class 0: 0.55
Weight for class 1: 5.81
```

Figure 5.2: computing class weights

Image standardization

Standardization is one kind of scaling. We need to scale when the features are of different scales, units, ranges etc. But in image all the feature columns are nothing but the intensities. Hence, all of them are already scaled in the same range [0-255]. So, what's the necessity of re-scaling them? But we can still standardize the data and which is helpful in our case.

Consider how a neural network learns its weights. C(NN)s learn by continually adding gradient error vectors (multiplied by a learning rate) computed from backpropagation to various weight matrices throughout the network as training examples are passed through.

The thing to notice here is the "multiplied by a learning rate".

If we didn't scale our input training vectors, the ranges of our distributions of feature values would likely be different for each feature, and thus the learning rate would cause corrections in each dimension that would differ (proportionally speaking) from one another. We might be overcompensating a correction in one weight dimension while undercompensating in another.

This is non-ideal as we might find ourselves in a oscillating (unable to center onto a better maxima in cost(weights) space) state or in a slow moving (traveling too slow to get to a better maxima) state.

It is of course possible to have a per-weight learning rate, but it's yet more hyperparameters to introduce into an already complicated network that we'd also have to optimize to find. Generally learning rates are scalars.

Thus, we try to normalize images before using them as input into NN (or any gradient based) algorithm.

Benefits of using pretrained model

Pre-Trained models most of them have a good architecture and this architecture are trained on a very powerful machines talking a lot of time with huge datasets, so this improves its weights to a very good stage.

So, in our case we have not large dataset with 57224 image and not having the ability to run our model in a powerful machine we use the online free machine, but this does not give you all what you need.

There are two techniques to use pretrained models with them this is discussed below:

- **Transfer learning**

Transfer learning means taking the relevant parts of a pre-trained machine learning model and applying it to a new but similar problem. This will usually be the core information for the model to function, with new aspects added to the model to solve a specific task. we have to identify which areas of the model are relevant to the new task, and which parts will need to be retrained. For example, a new model will keep the processes that allow the machine to identify objects but retrain the model to identify a different specific object.

A machine learning model which identifies (recognize) a certain subject within a set of images is a prime candidate for transfer learning. The bulk of the model which deals with how to select different subjects can be kept. The part of the algorithm which highlights a specific subject to categorize is the element that will be retrained. In this case, there's no need to rebuild and retrain a machine learning algorithm from scratch.

In supervised machine learning, models are trained to complete specific tasks from labelled data during the development process. Input and desired output are clearly mapped and fed to the algorithm. The model can then apply the learned trends and pattern recognition to new data. Models developed in this way will be highly accurate when solving tasks in the same environment as its training data. It will become much less accurate if the conditions or environment changes in real-world application beyond the training data. The need for a new model based on new training data may be required, even if the tasks are similar.

Transfer learning is a technique to help solve this problem. As a concept, it works by transferring as much knowledge as possible from an existing model to a new model designed for a similar task. For example, transferring the more general aspects of a model which make up the main processes for completing a task. This could be the process behind how objects or images are being identified or categorized. Extra layers of more specific knowledge can then be added to the new model, allowing it to perform its task in new environments.

Benefits of transfer learning in our case:

The main benefits of transfer learning include the saving of resources and improved efficiency when training new models.

- ✓ **Saving training data**

  A huge range of data is usually required to accurately train a machine learning algorithm. Labelled training data takes time, effort and expertise to create.

Transfer learning cuts down on the training data required for new machine learning models, as most of the model is already previously trained.

- ✓ **Efficiently train multiple models**

  Machine learning models designed to complete complex tasks can take a long time to properly train. Transfer learning means that you don't have to start from scratch each time a similar model is required. The resources and time put into training a machine learning algorithm can be shared across different models. The whole training process is made more efficient by reusing elements of an algorithm and transferring the knowledge already held by a model.

- Fine-Tune

Specifically, fine-tuning is a process that takes a model that has already been trained for one given task and then tunes the model to make it perform a second similar task.

Building and validating our model can be a huge task in its own right, depending on what data we're training it on.

This is what makes the fine-tuning approach so attractive. If we can find a trained model that already does one task well, and that task is similar to ours in at least some remote way, then we can take advantage of everything the model has already learned and apply it to our specific task.

If we have a model that has already been trained to recognize (categorize) images and we want to fine-tune this model to our project, we can first import our original model.

Generally, in fine tune we may remove last layer that classify the output may in thousand class like image_net dataset so we add our own classifying layer

In some experiments, we may want to remove more than just the last single layer, and we may want to add more than just one layer. This will depend on how similar the task is for each of the models.

Layers at the end of our model may have learned features that are very specific to the original task, whereas layers at the start of the model usually learn more general features like edges, shapes, and textures.

After we've modified the structure of the existing model, we then want to freeze the layers in our new model that came from the original model.

By freezing, we mean that we don't want the weights for these layers to update whenever we train the model on our new data for our new task. We want to keep all of these weights the same as they were after being trained on the original task. We only want the weights in our new or modified layers to be updating.

After we do this, all that's left is just to train the model on our new data. Again, during this training process, the weights from all the layers we kept from our original model will stay the same, and only the weights in our new layers will be updating.

Loss and accuracy matrix and reasons behind them

❖ Loss function

For loss we use binary cross entropy this according to a lot of literature work and papers we read after this we find the following benefits of it:

- cross entropy is equivalent to fitting the model using maximum likelihood estimation. This on the other hand can be interpreted as minimizing the dissimilarity between the empirical distribution of training data and the distribution induced by the model.
- why maximizing the likelihood. Maximum Likelihood estimators have nice asymptotical properties (they are the best estimators in terms of convergence rates), they are consistent and statistically efficient.
- Given the prevalence of gradient-based learning algorithms (especially in deep learning), the logarithm in the cross-entropy will undo any exponential behavior that is often given through popular activation/output units like the sigmoid/softmax. Thus, the logarithm will avoid that the gradient saturates for extreme values. Large gradients are important for gradient-descent algorithms to make sufficient progress in each iteration.

This make us use it in the first time, but we tried a lot of loss functions like Dice loss, Focal Loss…etc. and we found that binary is actually best for us.

❖ Accuracy Matrix

We use AUC according to evaluation in the Kaggle competition to test our results on it. We will discuss why we use it regardless the Kaggle competition.

We will start by explaining what is ROC-curve?

An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

- True Positive Rate
- False Positive Rate

True Positive Rate (TPR) is a synonym for recall and is therefore defined as follows: $TPR = \frac{TP}{TP+FN}$

False Positive Rate (FPR) is defined as follows: $FPR = \frac{FP}{FP+TN}$

An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. The following figure shows a typical ROC curve.

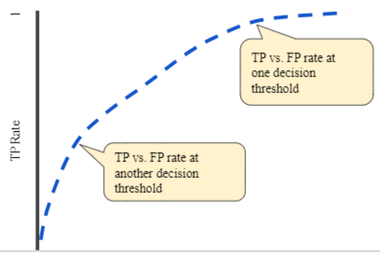
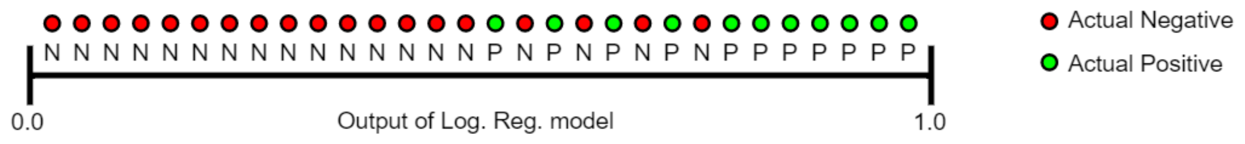


Figure 5.3: TP vs. FP rate at different classification

To compute the points in an ROC curve there's an efficient, sorting-based algorithm that can provide this information for us, called AUC.

AUC: Area Under the ROC Curve, That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).

AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example. For example, given the following examples, which are arranged from left to right in ascending order of logistic regression predictions:



N N N N N N N N N N N N N N N P N P N P N P N P P P P P P P
0.0
Output of Log. Reg. model
1.0
Actual Negative
Actual Positive

Figure 5.4: Predictions ranked in ascending order of logistic regression

Experiments

In the previous sections we discussed techniques that wouldn't used at the same time because of we use some combination of them and more of changing hyperparameters in the experiments as we will discuss next.

Some explanations before start experiments

How the experiment will be evaluated

We will evaluate our experiments using validation accuracy matrices and by submitting prediction to "SIIM-ISIC Melanoma Classification" competition on Kaggle.

Ways of deciding the next step and what we will do

When we start building our model, we can either go about it randomly or intentionally. A random approach to model building is tempting to take. It requires less effort and still brings about results. A random approach would be when after each iteration of a model, you decide to try something new to make it better without much thought as to why. There is no systematic way to improve your model. It is simply this: an idea popped into your head that might improve the AUC-score so why not try it? It’s easy and it works.

The intentional approach to building a model is using error analysis. Error analysis requires you to dig into the results of your model after each iteration. You look at the data and predictions on an observational level and form hypotheses as to why your model failed on certain predictions. Then you test your hypothesis by changing the model in a way that might fix that error and begin the next iteration. Each iteration of modeling becomes more time consuming with error analysis, but the final result is better and will likely arrive faster.

In error analysis the important factors are human level and train, dev and test accuracy. There is no human level in our problem so we will consider it to reach to 94.9% accuracy as the higher accuracy in the SIIM ISIC competition.

So, we choose to do error analysis after each experiment and depending on the results of error analysis we will try what to do next, but you may see that we sometimes do some experiments to run at the same time its changes was taken from error analysis trying to make the process faster.

Explanation about the distribution of train/div/test set

In all experiments you will see that train may be from a distribution that different from div and teat set that they are always from the same distribution instead of any preprocessing like normalizing images this must be the same for the three sets.

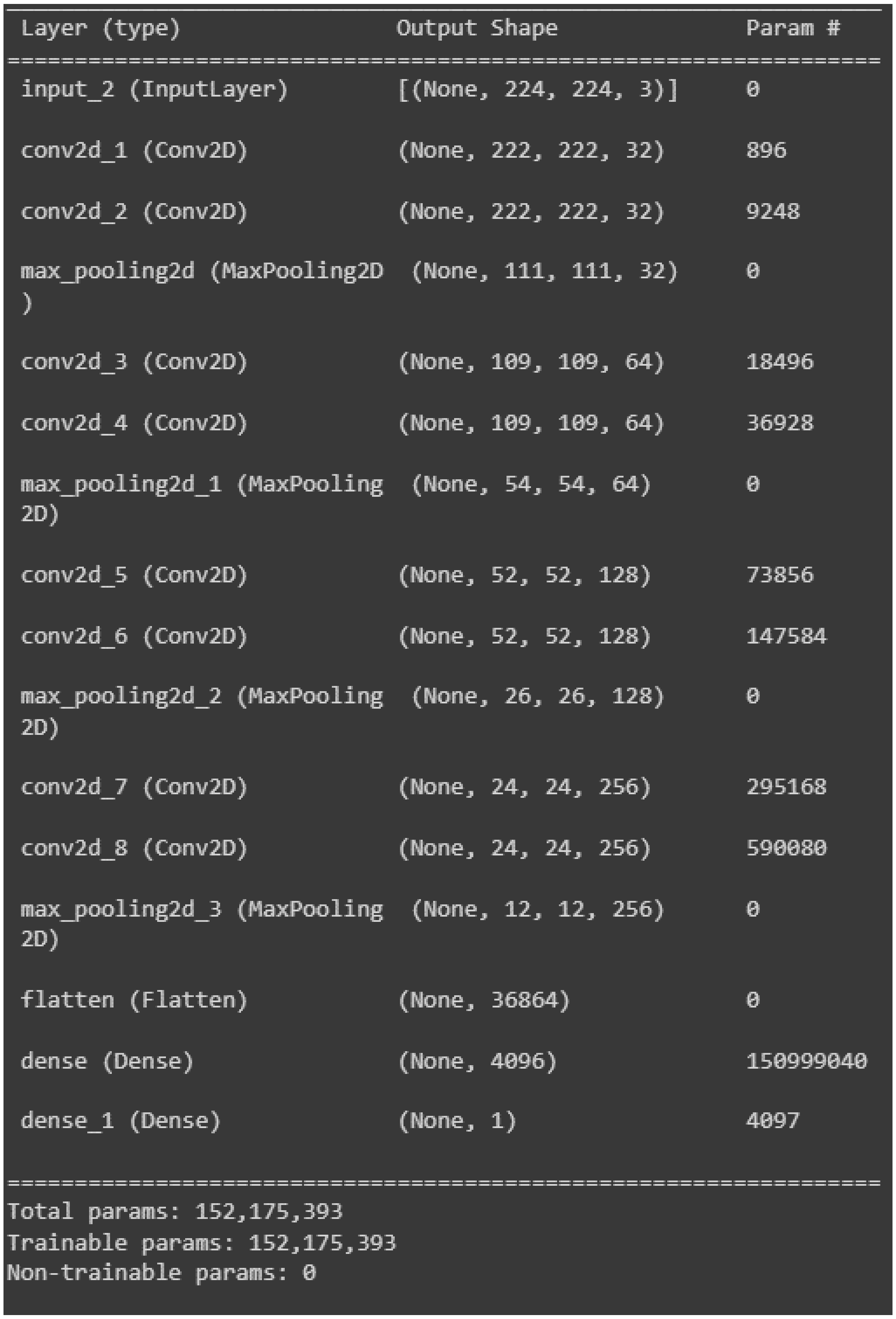
Base Model

In this section, the model will be explained in detailed to avoid explaining the model add each experiment.

Let’s describe the base model in the next steps:

- Import all libraries that will be used in the experiments from Tensorflow, Keras, SKLearn and Matplotlib, and import numpy, pandas and os.

- Use “marking.csv” from the dataset to create pandas’ data frame that contains training images names and their labels.
- Train_test_split is used to split train data frame into train_df and validate_df to use validation data to evaluate the model at each step.
  Use test_size equal 0.2 making the number of validation images equal 11445 that close to number of test set images and use a certain random state to see changes in experiments accuracy without outside effect.
- Print a figure using plot in matplotlib that explain the difference between the number of images in each of the two classes (benign, malignant).
- We use class weights to try to avoid the effect of this huge difference.
  There are a lot of techniques to avoid this difference rather than class weights but using it backing two seeing a lot of previous work and experiments and the difficult to collect malignant images sometimes it takes years to collect the number of images that we want to balance our dataset.
- In this step we used the images data generator to collect data set images into a tensor data structure that is optimized to work with GPU in parallel by run baches of the data set at the same time, we assign image size to (224,224) and batch size to 32 batch.
  In image data generator we rescale our images to make pixel values in range from zero to one and add augmentation to present it in the dataset at this step we use only zoom augmentation with ratio equal 0.3.
- In validation dataset we use image data generator with the same image size and batch size but only rescale the images without augmentation as it must be from the same distribution of the test set.
- The model explained in the model summary in figure 5.5 and the model compiled using binary_crossentropy loss function, Adam optimizer with LR equal 10^{-8} as small number in the first and as accuracy matrices we used Accuracy Under the Curve (AUC) matrix and False Negative matrix as the False classified malignant as benign as more hurtful for use.



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

 conv2d_1 (Conv2D)           (None, 222, 222, 32)      896

 conv2d_2 (Conv2D)           (None, 222, 222, 32)      9248

 max_pooling2d (MaxPooling2D  (None, 111, 111, 32)     0
 )

 conv2d_3 (Conv2D)           (None, 109, 109, 64)      18496

 conv2d_4 (Conv2D)           (None, 109, 109, 64)      36928

 max_pooling2d_1 (MaxPooling  (None, 54, 54, 64)       0
 2D)

 conv2d_5 (Conv2D)           (None, 52, 52, 128)       73856

 conv2d_6 (Conv2D)           (None, 52, 52, 128)       147584

 max_pooling2d_2 (MaxPooling  (None, 26, 26, 128)      0
 2D)

 conv2d_7 (Conv2D)           (None, 24, 24, 256)       295168

 conv2d_8 (Conv2D)           (None, 24, 24, 256)       590080

 max_pooling2d_3 (MaxPooling  (None, 12, 12, 256)      0
 2D)

 flatten (Flatten)           (None, 36864)             0

 dense (Dense)               (None, 4096)              150999040

 dense_1 (Dense)             (None, 1)                 4097

=================================================================
Total params: 152,175,393
Trainable params: 152,175,393
Non-trainable params: 0
```

Figure 5.5: base model summary

- Define callback to save model and model weights of the best validation accuracy after training the model.

- Fit the model with the train and validation datasets to start training the number of epochs depend on that the maximum number that can be assign must not exceeds the nine hours of training on GPU as constraints in Kaggle notebooks the pass the number of training steps (total_train//batch_size), validation steps (total_validate//batch_size), callback variable and classes weights.
- Display loss, AUC accuracy, and False negative accuracy in the following figures.

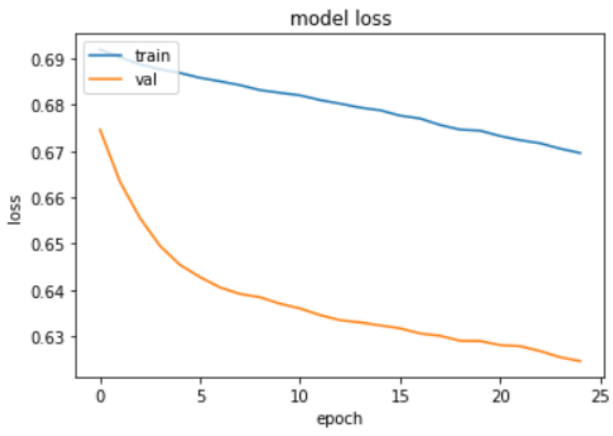


Figure 5.6: Model Loss

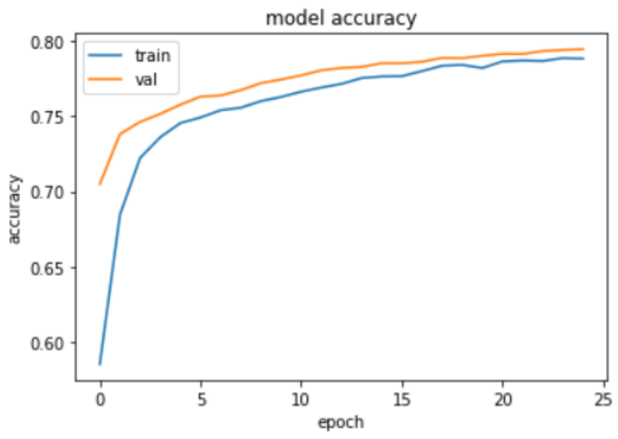


Figure 5.7: Model accuracy

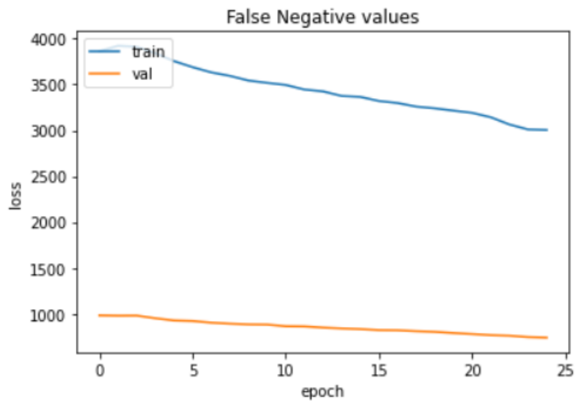


Figure 5.8: False Negative Values

- List images names in test image directory in test_filenames list and put this list in test data frame.
- For each image in test_filenames import the image using Image.open from PIL and resize it to (224,224,3), convert it to numpy array and rescale it by dividing each pixel on 255.0 then using this array predict the probability of

the image that identify it's benign or malignant append this prediction to "target" list.

- Add each prediction in the target list to test data frame in a column called target then save the data frame as a submission.csv to submit it into the competition to get test accuracy, Test accuracy of the base model is 0.7415 as private score and 0.7104 as public score.

This is the base model that used in the next experiments.

Experiment *one*

After analyzing the previous figure, we see that train is close to validation accuracy, so we face high bias low variance problem, depending on this we have to increase training one of the techniques to do so is reach to the optimal learning rate.

So, we try to reach to a learning rate close to the optimal by try to change learning rate.

First, we change learning rate to 10^{-7}, the result of this change is in the following figures:

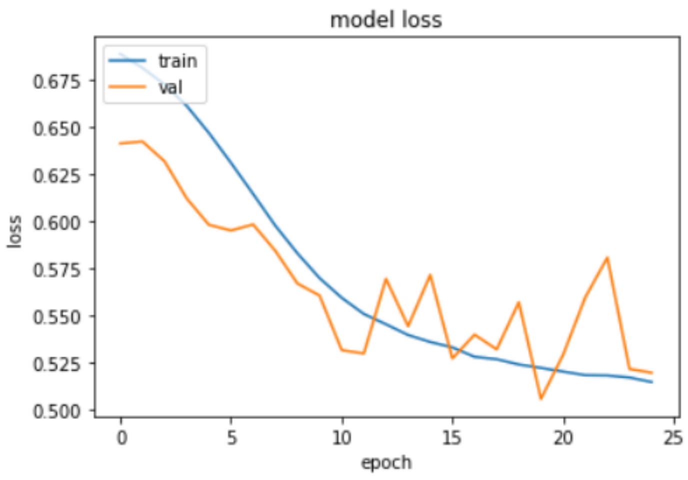


Figure 5.9: Model Loss

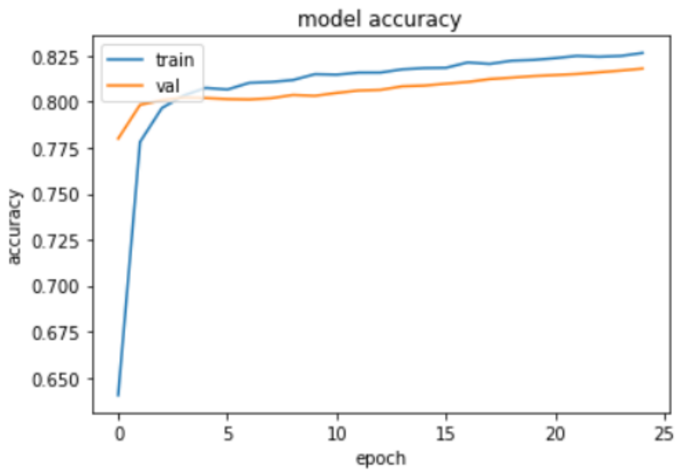


Figure 5.10: Model accuracy

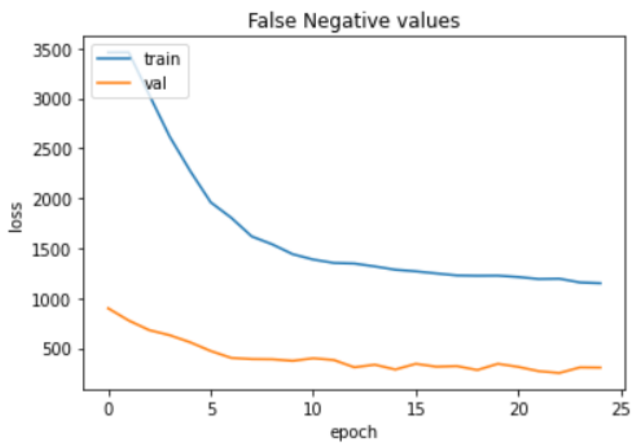


Figure 5.11: Model False negative values

After changing learning rate to 10^{-7} the accuracy increased by 3% in both train and validation, loss function decreases in training but become not stable in validation comparing to the previous model but it's a bigger decrease that what happen in the last change and false negative values decreased by 2000 values in the training and 500 in the validation.

From the previous the model has a better improve than the previous model but still in the same state of high bias low variance, so we will increase the learning rate again to 10^{-6} and the following figures present what happen.

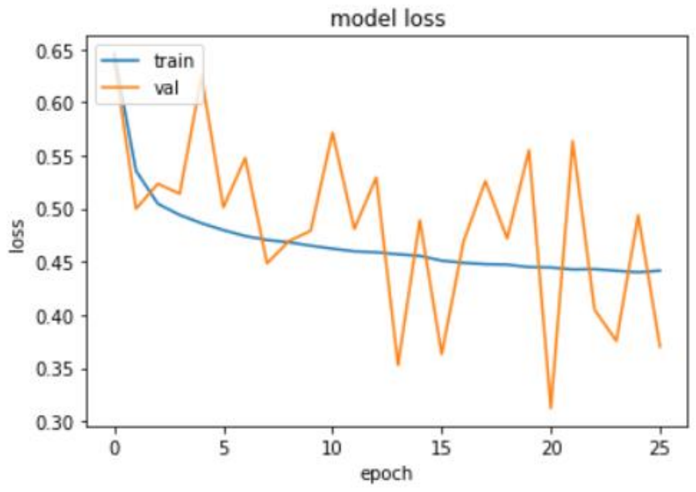


Figure 5.12: Model Loss

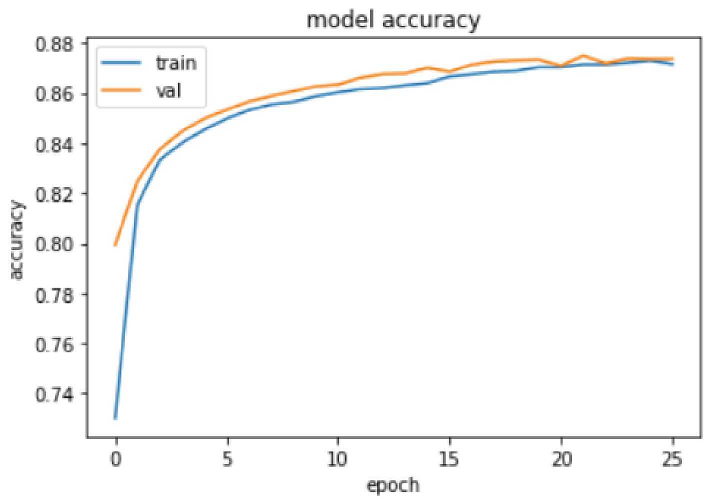


Figure 5.13: Model accuracy

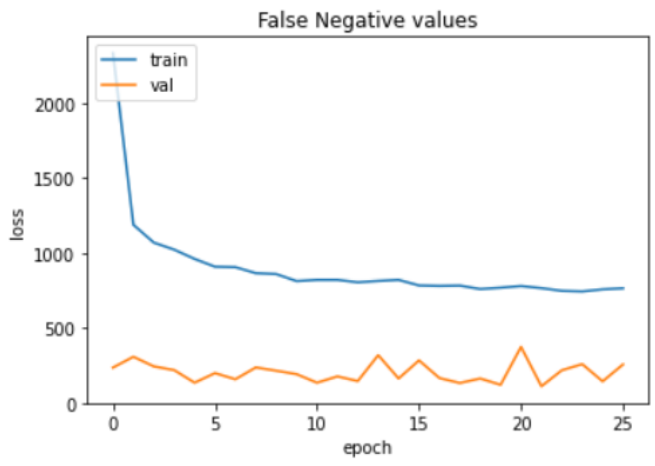


Figure 5.14: Model False negative values

After changing learning rate to 10^{-6} the accuracy increased by 5% in both train and validation, loss function decreases in training and validation but become not stable in validation comparing to the previous model and false negative values decreased by a little in training values and almost decreased by 110 in the validation values.

From the previous the model has a better improve than the previous change but still in the same state of high bias low variance, so we will increase the learning rate again to 10^{-5} and the following figures present what happen.

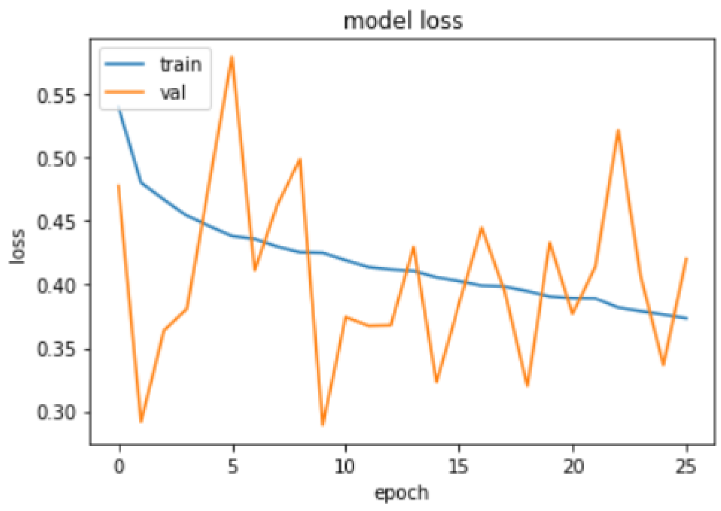


Figure 5.15: Model Loss

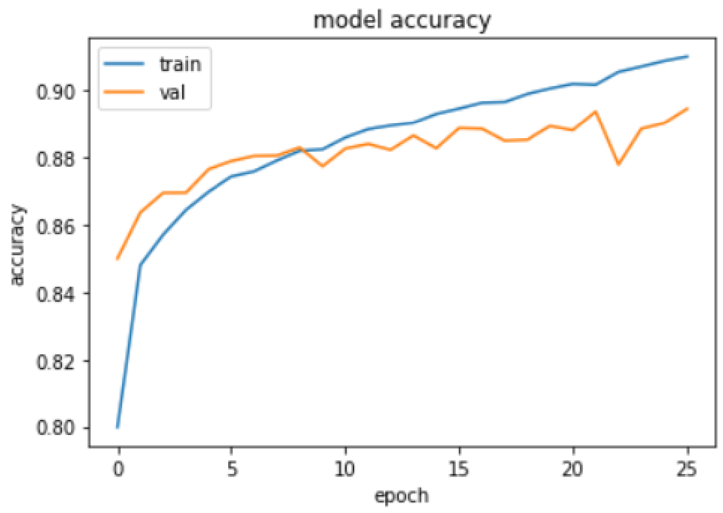


Figure 5.16: Model Accuracy

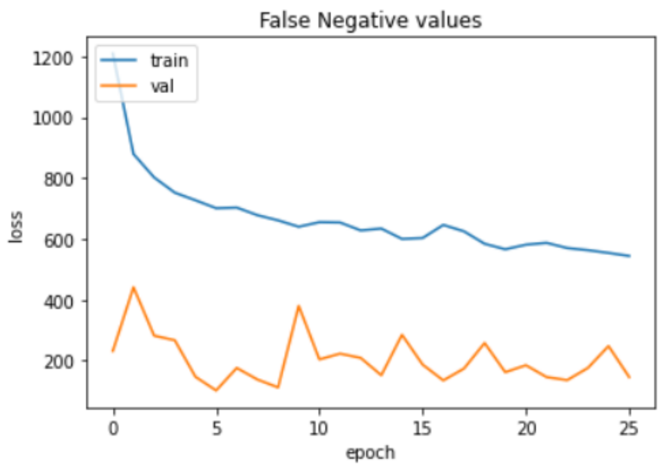


Figure 5.17: Model False negative values

After changing learning rate to 10^{-5} the accuracy increased by 4% in training and 2.5% in validation, loss function decreases in training and validation almost with the same difference as the previous change and false negative values almost doesn't change from the previous model.

From the previous the model has a better improve than the previous change but the difference between train and validation accuracy increases a little but still in the same state high bias and low variance, so we will increase the learning rate again to 10^{-4} and see whether the model will improve or not let's see in the next figures.

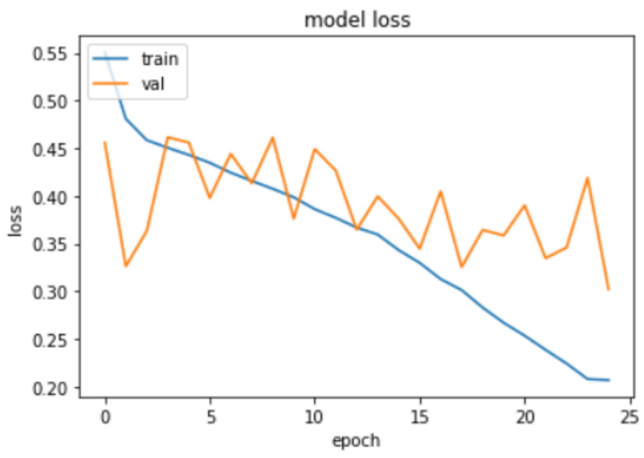


Figure 5.18: Model Loss

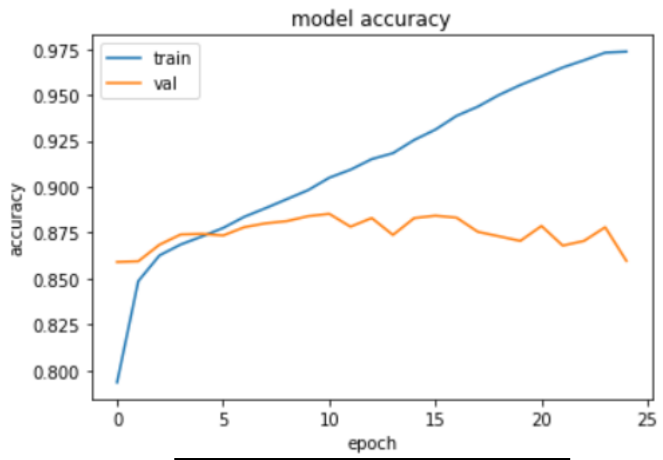


Figure 5.19: Model Accuracy

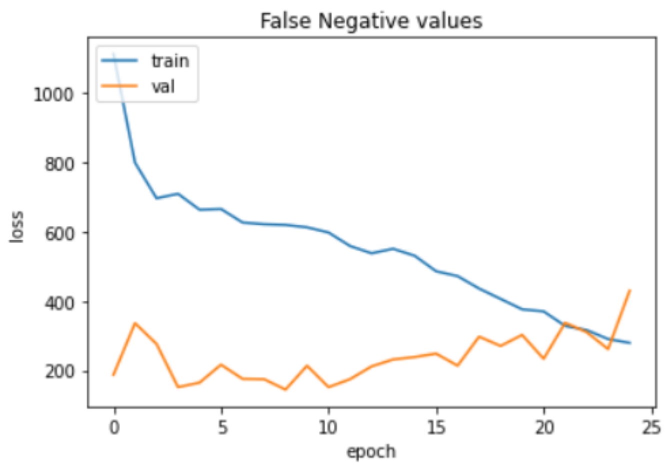


Figure 5.20: Model False negative values

After changing learning rate to 10^{-4} the accuracy increased by 6% in training and decreased by 1% in validation, loss function decreases a lot in training and almost is the same in validation and false negative decreases a lot in training and increases with small values in validation.

From the previous the model has a better improve in training but getting bad in validation, so we are facing overfitting at this step (low bias high variance).

The decrease of learning rate in this change makes the model overfit the training data so we will stop at this step and go to the next experiment.

After submitting the test prediction of each of the previous changes the test accuracy after changing learning rate to 10^{-7} is 0.7489 as private score and 0.7633 as public score, and with 10^{-6} learning rate is 0.8239 as private score and 0.8288 as public score, and with 10^{-5} learning rate is 0.8104 as private score and 0.8139 as public score, and with 10^{-4} learning rate is 0.7605 as private score and 0.7556 as public score.

Experiment two

In this experiment we will use additional augmentation trying to reduce overfitting that happen with 10^{-4} learning rate and this augmentation will increase the amount of data this increase in data may improve the accuracy of the model.

So, next we will discuss what happen when adding horizontal and vertical flip to the augmentation with learning rates $10^{-6}, 10^{-5}, 10^{-4}$.

We will start with learning rate 10^{-6}, the following figures present what happen.

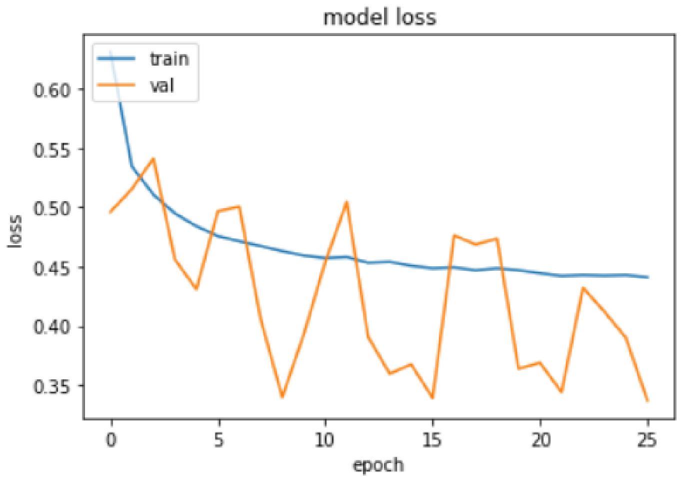


Figure 5.21: Model Loss

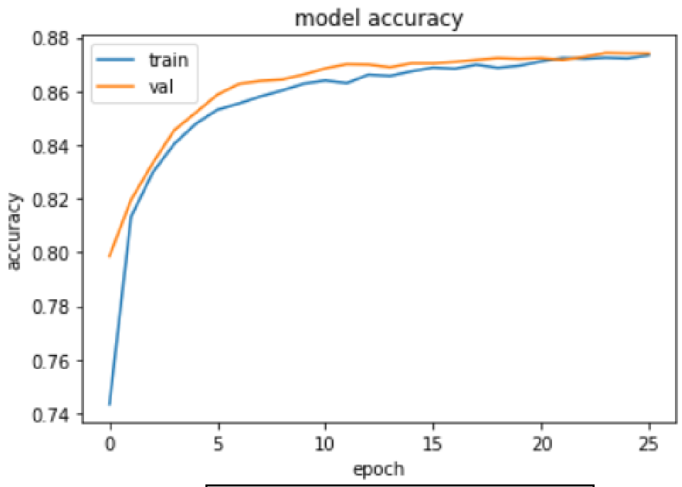


Figure 5.22: Model Accuracy

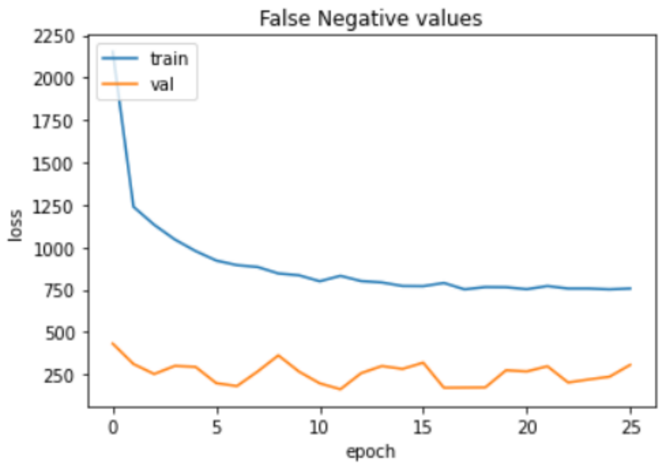


Figure 5.23: Model False negative values

After adding augmentation with learning rate 10^{-6} the training and validation accuracy almost the same without augmentation and also the loss function but the False Negative values increased in validation and almost the same in training.

From the previous after adding augmentation with learning rate 10^{-6} there is no improve in the model and it still in the same state high bias low variance.

Next, we will see what happen when adding augmentation with learning rate 10^{-5}

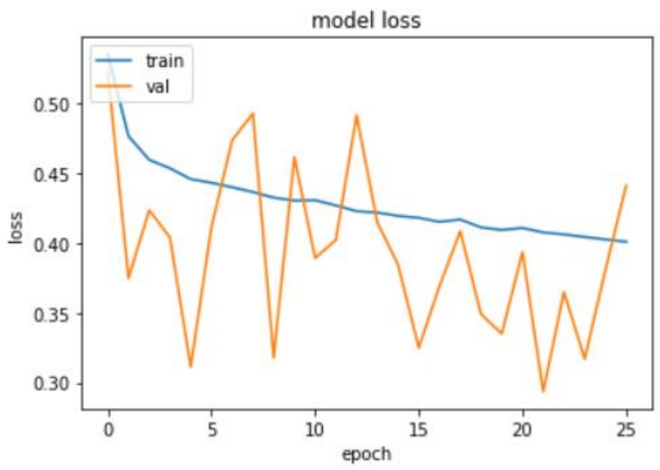


Figure 5.24: Model Loss

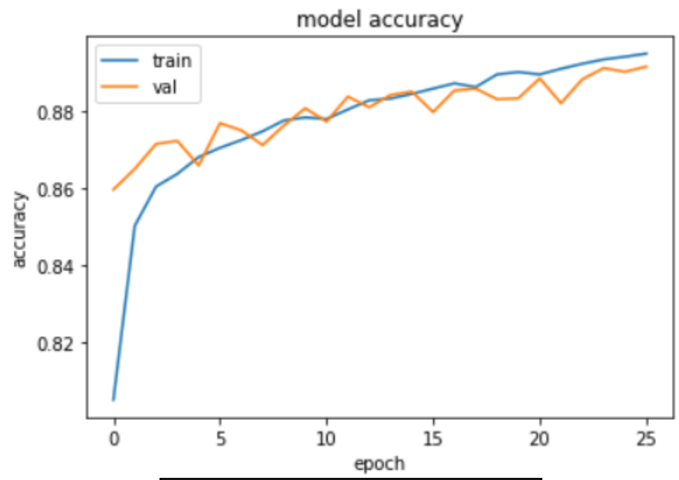


Figure 5.25: Model Acciracy

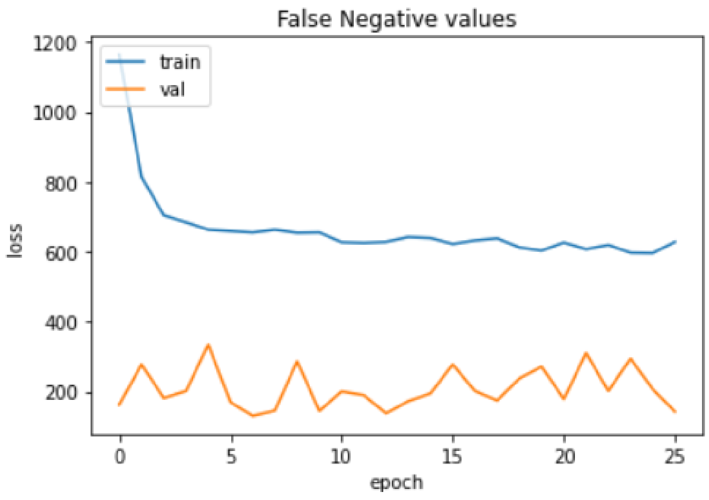


Figure 5.26: Model False negative values

After adding augmentation with learning rate 10^{-5} the training accuracy decreases by 1% and validation accuracy almost the same as without augmentation, the loss function increased a little bit we can say that it is not changed, and the False Negative values increased in training and validation.

From the previous after adding augmentation with learning rate 10^{-5} there is improve in the model by decreasing overfitting and it is in the state high bias low variance.

Next, we will see what happen when adding augmentation with learning rate 10^{-4}

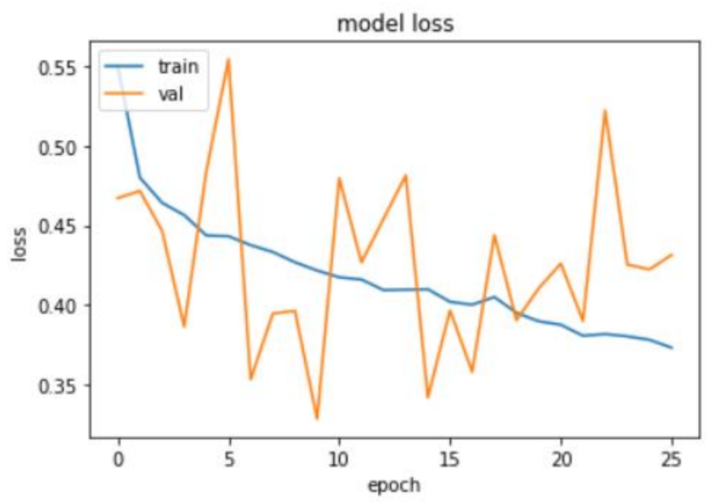


Figure 5.27: Model Loss

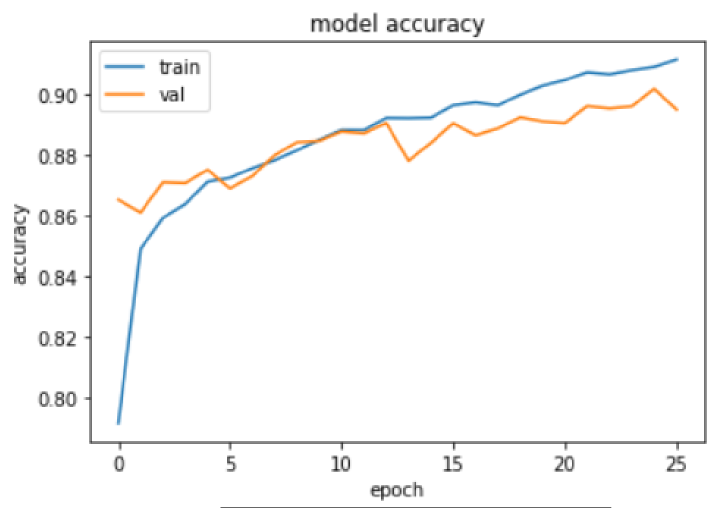


Figure 5.26: Model Accuracy

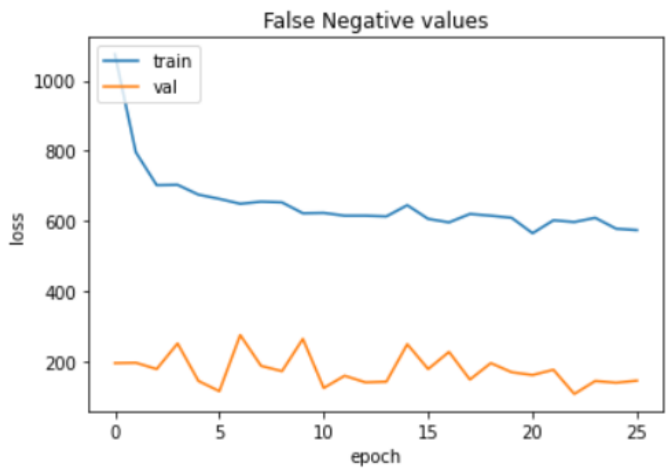


Figure 5.29: Model False negative values

After adding augmentation with learning rate 10^{-4} the training accuracy decreases by 6% and validation accuracy increases by 4%, the loss function increased in both training and validation, and the False Negative values increased in training and decreased in validation.

From the previous after adding augmentation with learning rate 10^{-4} there is improve in the model by decreasing overfitting and it is become in the state high bias low variance.

Adding augmentation improves models by decreasing overfitting in the next experiment we will try to increase the mode accuracy.

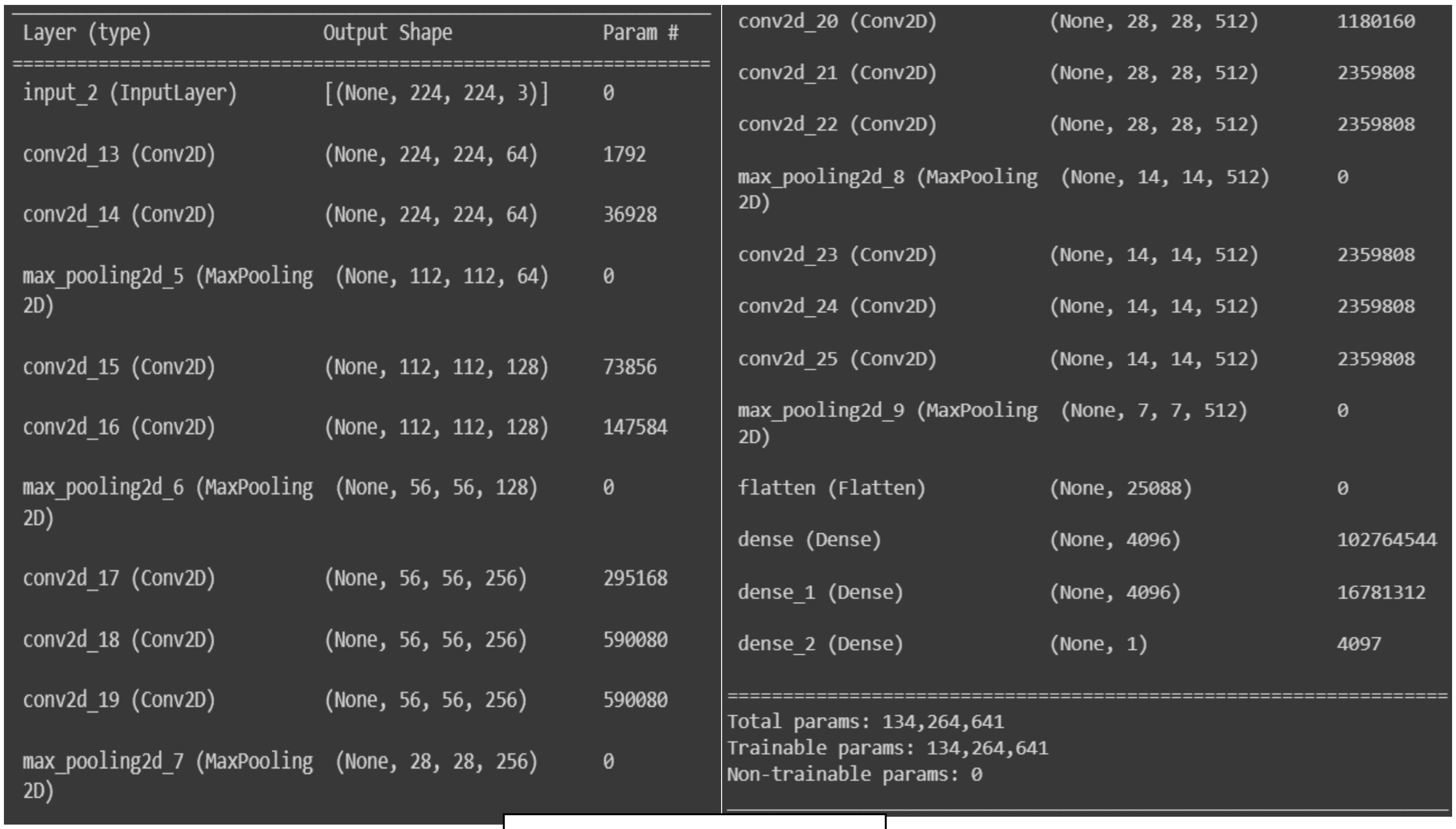
After submitting the test prediction of each of the previous changes the test accuracy after adding augmentation with learning rate 10^{-6} 0.8191 as private score and 0.8325 as public score, and with 10^{-5} learning rate is 0.8287 as private score and 0.8507 as public score, and with 10^{-4} learning rate is 0.8086 as private score and 0.8356 as public score.

Experiment three

In this experiment we will try two techniques to increase accuracy, we will make the neural network bigger and use the weights of the model after each change as initialization to the next to train the model more hours to see the effect of this.

Explanation: we save the model with the higher validation accuracy to get the model weights in a point with least overfitting and run the next model using these weights as initialization with some changes if the model starts to overfit the training data.

We start with rescaling, no augmentation and use a bigger model, the model summary presented in figure 5.30.



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

 conv2d_13 (Conv2D)          (None, 224, 224, 64)      1792

 conv2d_14 (Conv2D)          (None, 224, 224, 64)      36928

 max_pooling2d_5 (MaxPooling  (None, 112, 112, 64)     0
 2D)

 conv2d_15 (Conv2D)          (None, 112, 112, 128)     73856

 conv2d_16 (Conv2D)          (None, 112, 112, 128)     147584

 max_pooling2d_6 (MaxPooling  (None, 56, 56, 128)      0
 2D)

 conv2d_17 (Conv2D)          (None, 56, 56, 256)       295168

 conv2d_18 (Conv2D)          (None, 56, 56, 256)       590080

 conv2d_19 (Conv2D)          (None, 56, 56, 256)       590080

 max_pooling2d_7 (MaxPooling  (None, 28, 28, 256)      0
 2D)

 conv2d_20 (Conv2D)          (None, 28, 28, 512)       1180160

 conv2d_21 (Conv2D)          (None, 28, 28, 512)       2359808

 conv2d_22 (Conv2D)          (None, 28, 28, 512)       2359808

 max_pooling2d_8 (MaxPooling  (None, 14, 14, 512)      0
 2D)

 conv2d_23 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_24 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_25 (Conv2D)          (None, 14, 14, 512)       2359808

 max_pooling2d_9 (MaxPooling  (None, 7, 7, 512)        0
 2D)

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 4096)              102764544

 dense_1 (Dense)             (None, 4096)              16781312

 dense_2 (Dense)             (None, 1)                 4097

=================================================================
Total params: 134,264,641
Trainable params: 134,264,641
Non-trainable params: 0
```

Figure 5.30: Model summary

The effect of the above changes after 55 epoch, shown in the next figures.

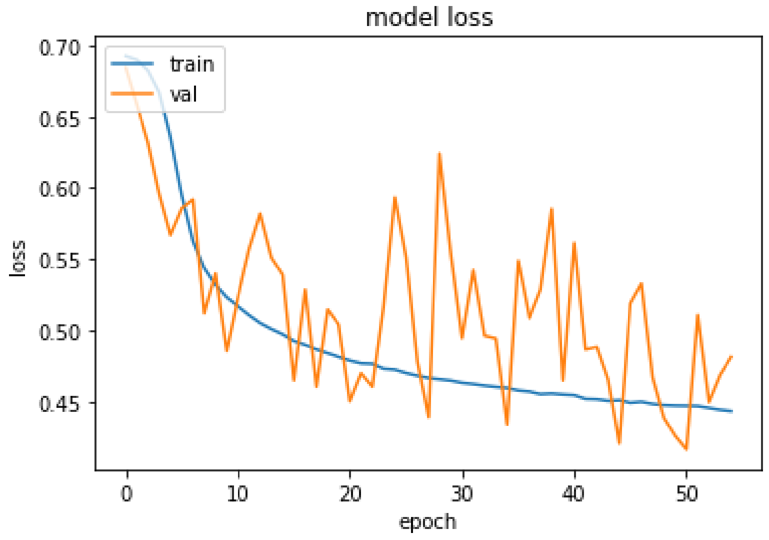


Figure 5.31: Model Loss

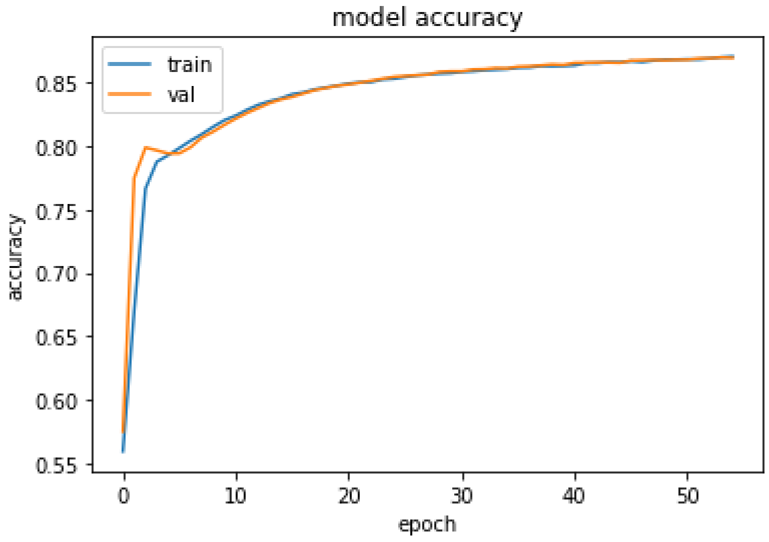


Figure 5.32: Model Accuracy

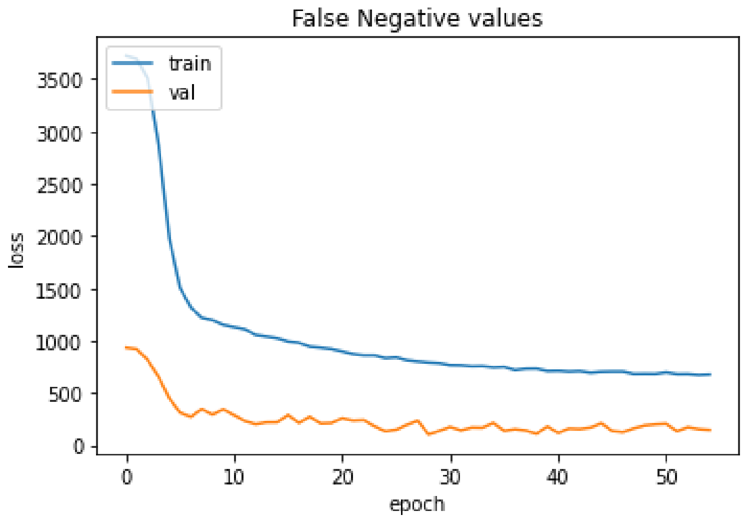


Figure 5.33: Model False negative values

From the figures the model performance very good as start because there is no overfitting it do very will in both train and validation, the model loss slightly bigger and the False Negative values is almost in the same range.

Every two changes of the next changes trained in parallel to see the effect of them on the accuracy matrices and the loss.

- Firstly, use the previous model weights and run additional 42 epoch, the effect shown in the next figures.

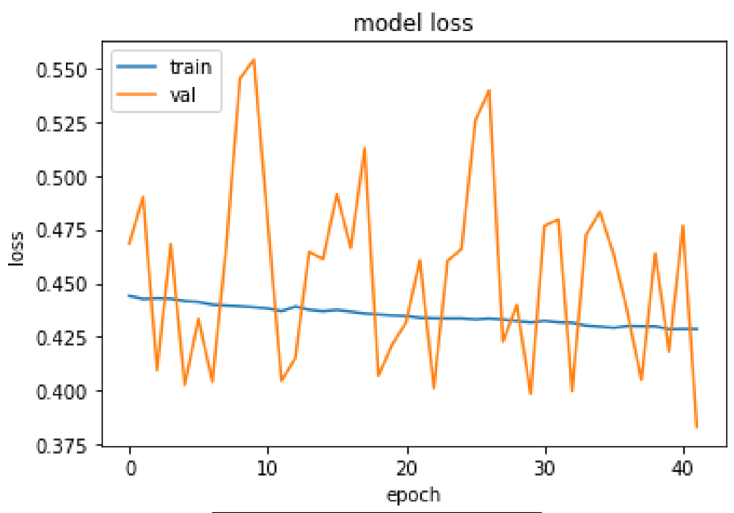


Figure 5.34: Model Loss

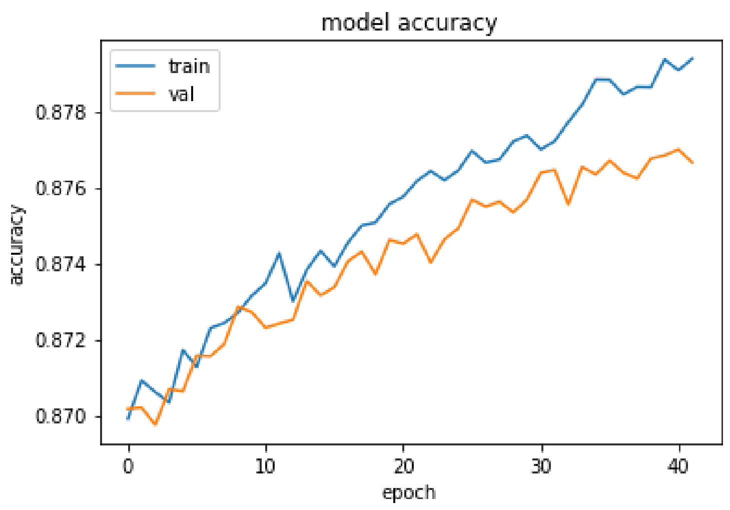


Figure 5.35: Model Accuracy

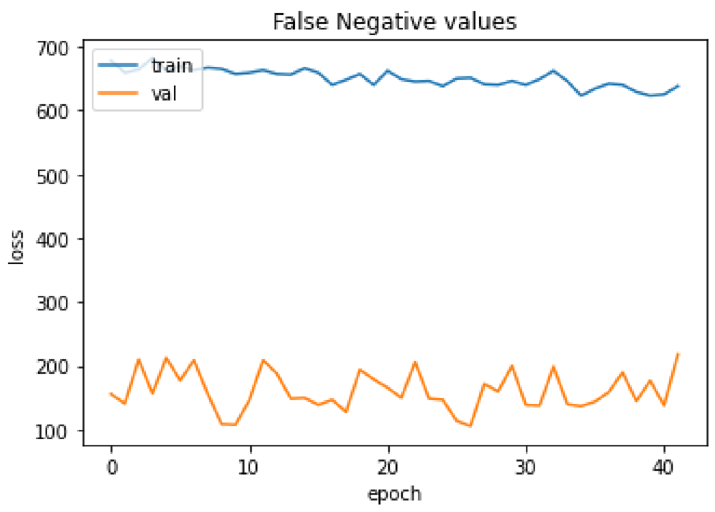


Figure 5.36: Model False negative values

The performance of the model slightly improves, the validation loss not stable but finally it is decreased, and False Negative values decreases with good difference comparing to the previous.

From the above the model doesn't improve comparing to the number of epochs and GPU hours.

- Secondly, add three dense layers with dropout layers to the model and run with its weights as initialization.

The effect of this change shown in the next figures.

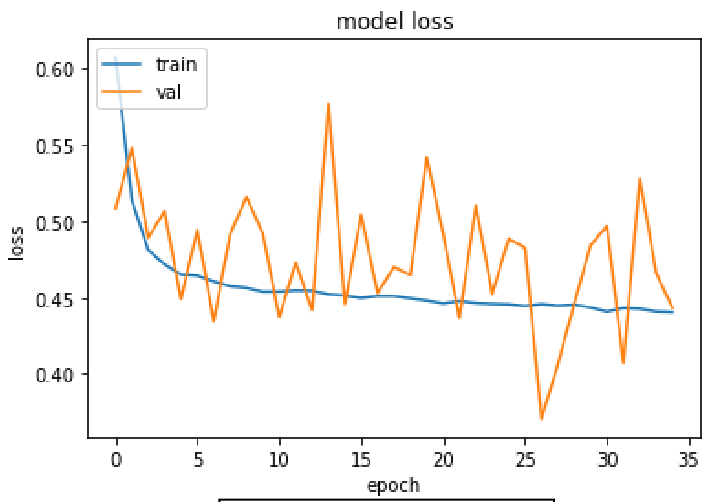


Figure 5.36: Model Loss

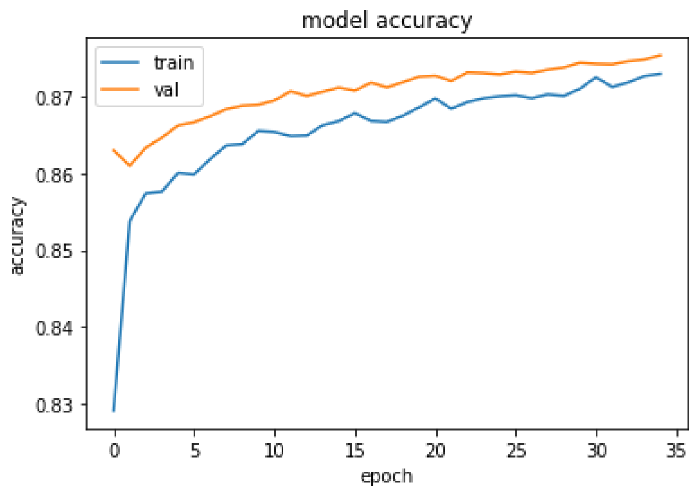


Figure 5.38: Model Accuracy

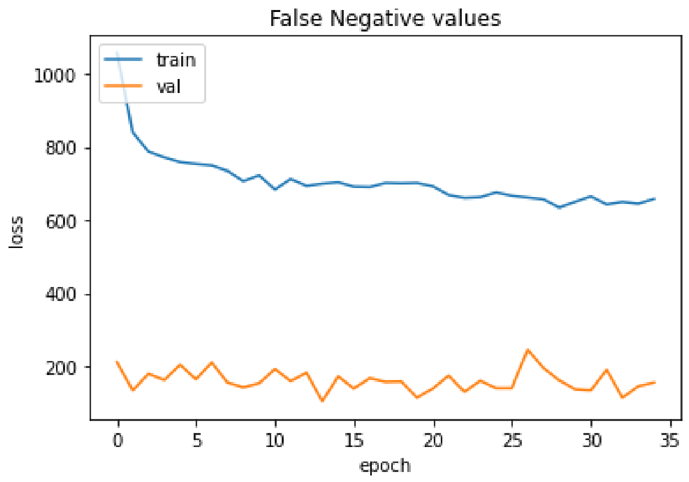


Figure 5.39: Model False negative values

The model does will in the validation accuracy than the training accuracy but comparing to its parallel change the validation accuracy almost the same, The Loss and False Negative values decreased.

From the above the model improves in overfitting but totally it doesn't improve comparing to the number of epochs and GPU hours.

- Firstly, add more augmentation to the fist model of the previous two changes and use 10^{-5} learning rate and like all in this experiment weights of the previous model will be used as initialization to the model.

The next figures present what happen after train 37 epoch with this model.

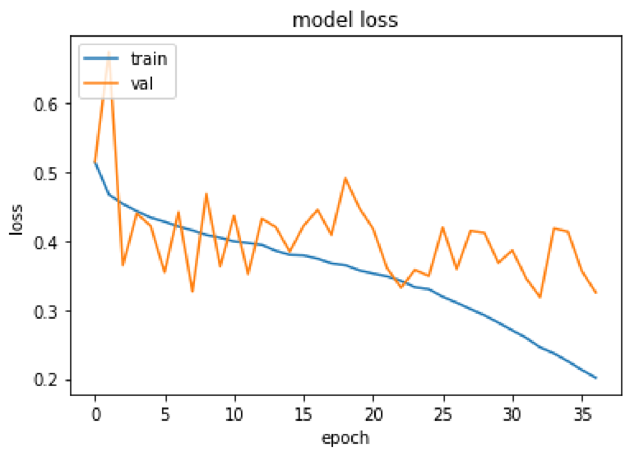


Figure 5.40: Model Loss

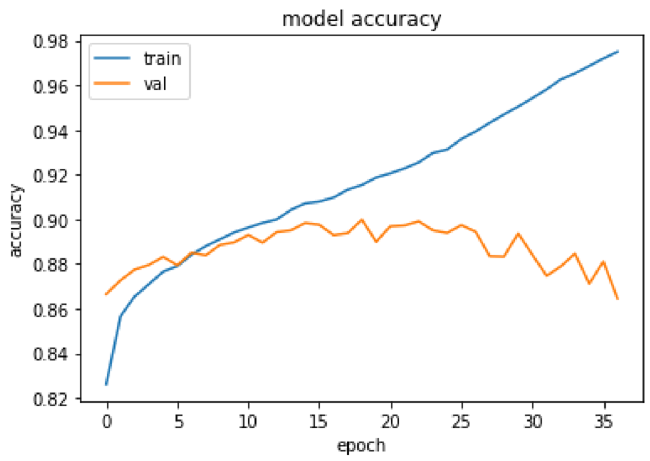


Figure 5.41: Model Accuracy

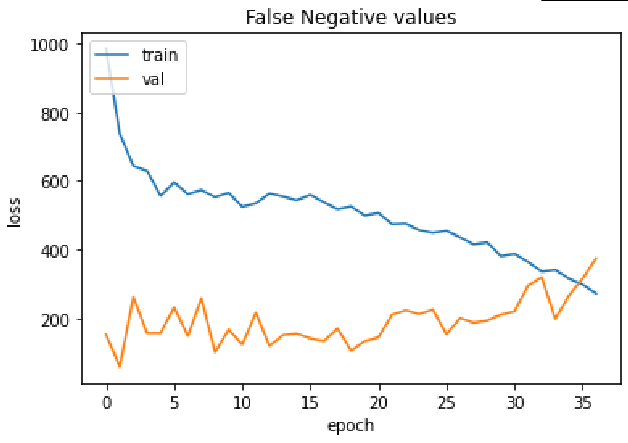


Figure 5.42: Model False negative values

From the figures model Accuracy start increasing in both train and validation but after ten epochs increases in training but still in the same range in validation, loss firstly decreases in both but after some epochs training accuracy decreases and validation still in the same values, and the False Negative values start with decreasing in validation but after almost 20 epochs start increasing but training continued decreasing.

From the above overfit the data, but since we save the model with the best validation accuracy, it will not affect the next model much.

- Secondly, use the first change but with 10^{-4} learning rate.

The next figures present what happen after train 47 epoch with this model.

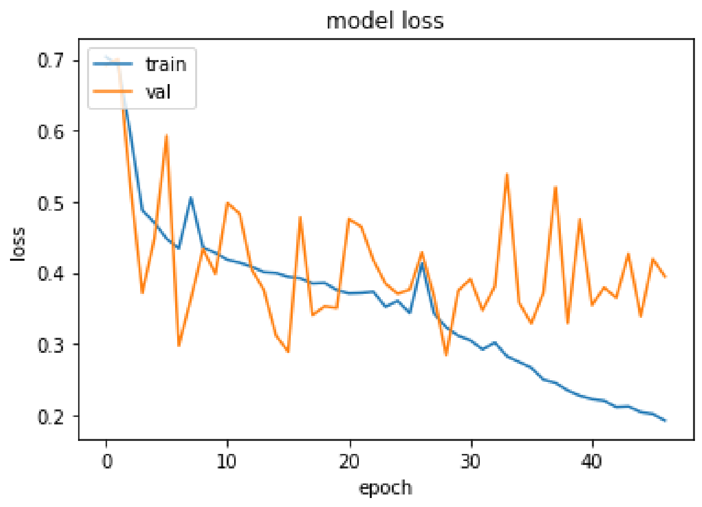


Figure 5.43: Model Loss

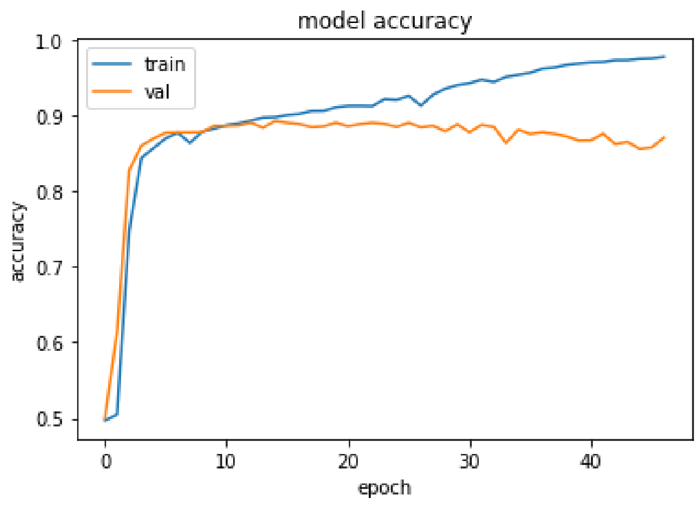


Figure 5.44: Model Accuracy

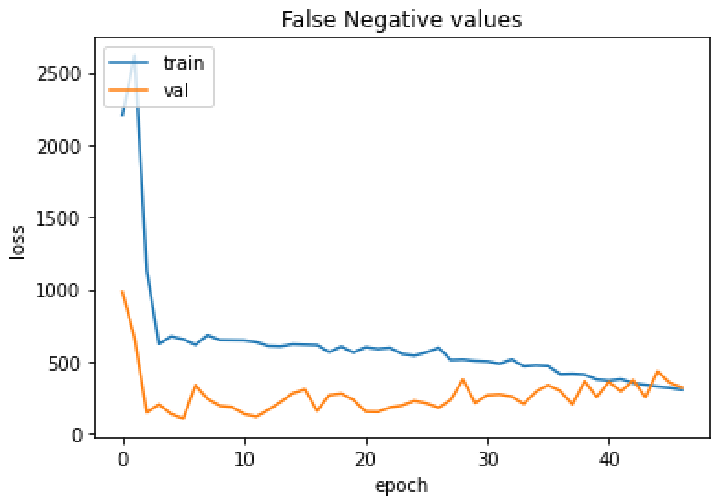


Figure 5.45: Model False negative values

From the figures model Accuracy start increasing in both train and validation but after almost fifteen epoch increases in training but slightly decreased in validation, loss firstly decreases in both but after some epochs training accuracy decreases and validation still in the same values but the loss here is better than the previous change, and the False Negative values start with decreasing in validation but after almost 20 epochs start slightly increasing but training continued decreasing.

From the above overfit the data, comparing to the previous model this is better in loss but the previous has better accuracy so we will continue with both models and see.

In the next two changes add two dropout layers in the last two blocks in the model and add zoom augmentation with range [0.5,1.0].

- Firstly, with learning rate 10^{-5}.

The next figures present what happen after train 30 epoch with this model.

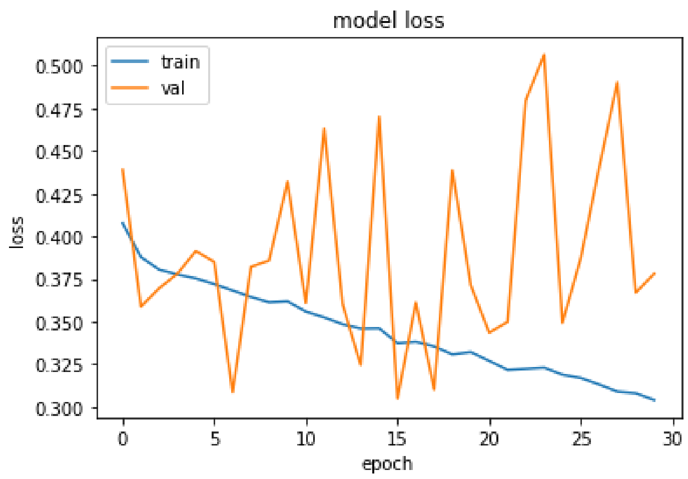


Figure 5.46: Loss

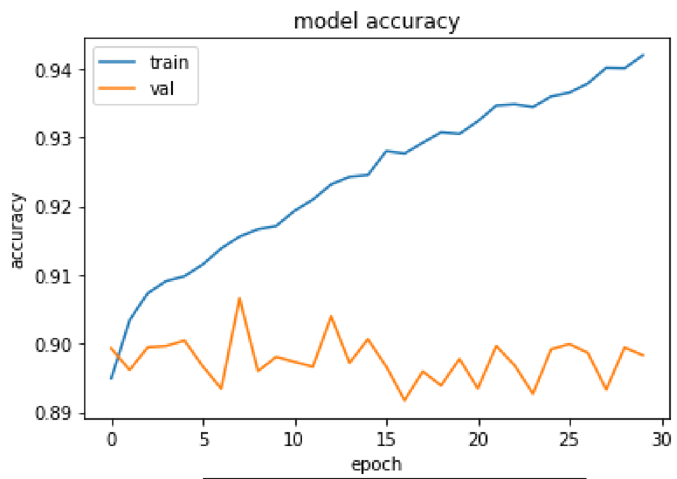


Figure 5.47: Model Accuracy

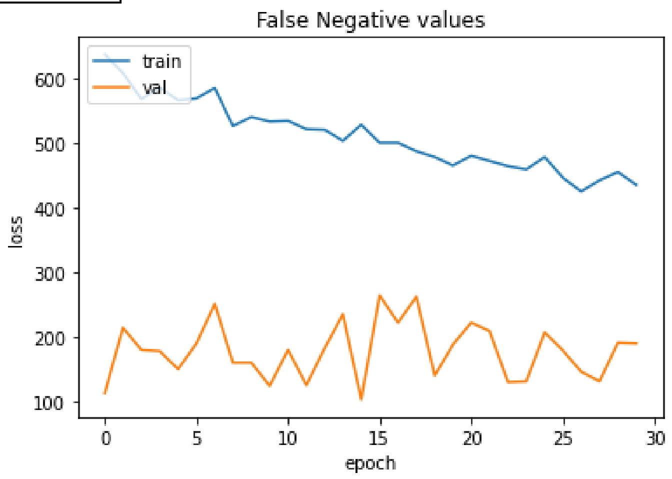


Figure 5.48: Model False negative values

From the figures training accuracy increased in almost straight line but validation in most time it decreased, it increased slightly in some experiments, the Loss function in training decreases in almost a straight line but in validation it frequently changes sometimes takes small values and sometime a very high values, and False Negative values it decreases in both but in validation it isn't stable.

- Secondly, with 10^{-4} learning rate

The next figures present what happen after train 32 epoch with this model.

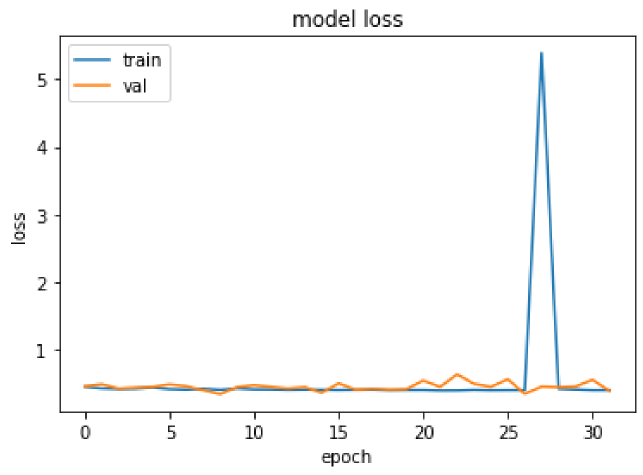


Figure 5.49: Model Loss

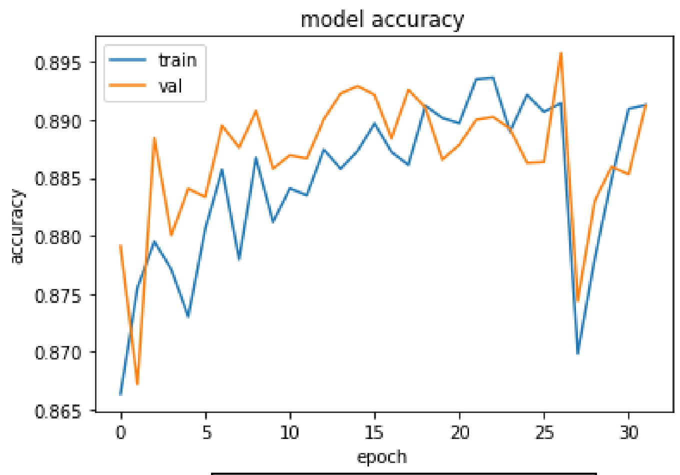


Figure 5.50: Model Accuracy

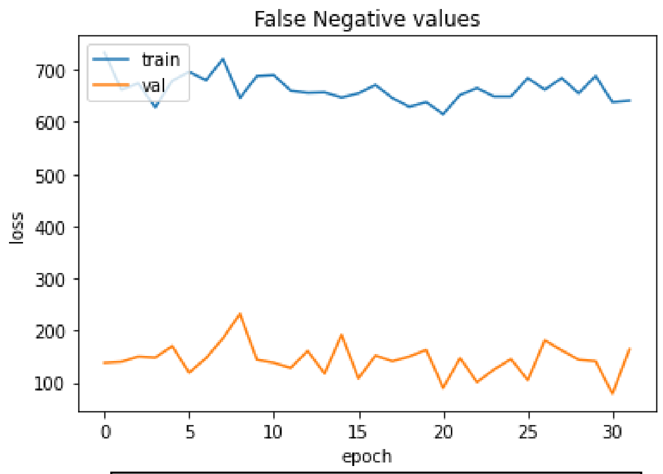


Figure 5.51: Model False negative values

From the figures train and validation accuracy increases in the same range but both are not stable but at most of the epochs the validation accuracy is higher, the loss is very good in both, training increases suddenly but quickly back to the small values, and False Negative values in validation is very good but in training if compared to the last model, it's not very good.

Next, we used the weights of the fist change (the highest validation accuracy) and train it after standardizing the dataset images using training images STD and mean.

The next figures present what happen after train 28 epoch with this model.

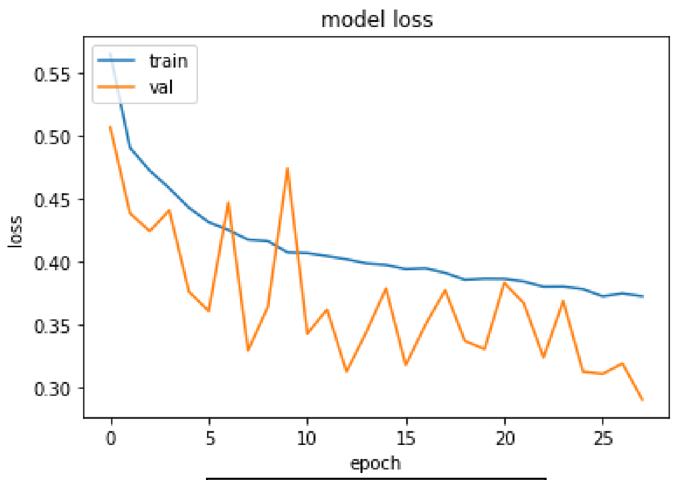


Figure 5.52: Model Loss

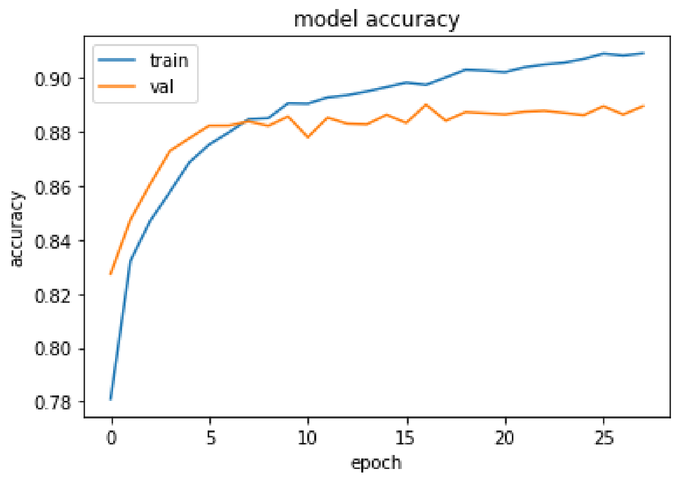


Figure 5.53: Model Accuracy

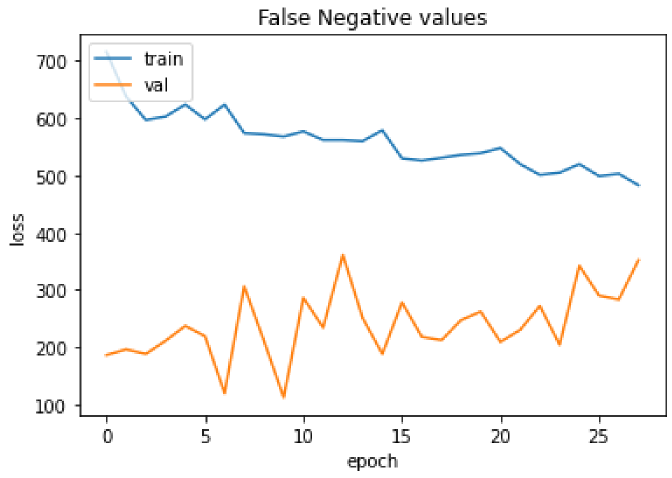


Figure 5.54: Model False negative values

From the figures the model accuracy in creases but after epoch nine the validation almost fixed and training continue increasing, Loss is not very well in both train and validation, but it isn’t bad, and False Negative values did like Accuracy.

After this change we did a lot of experiments but the performance in them is like what shown in the next figures.

As shown in the figures training and validation are very different.

So, to solve this problem we must connect the output of network layers not to only the next layer but to of the layers in the same block of layers.

There are some networks that have this advantage like ResNet and DenseNet.

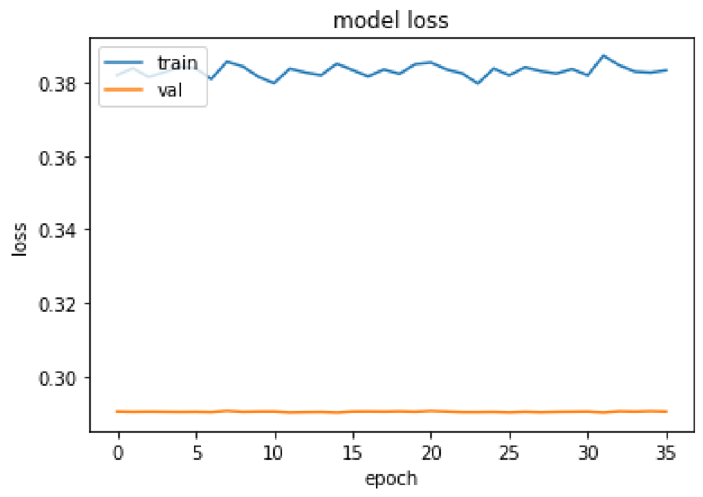


Figure 5.55: Model Loss

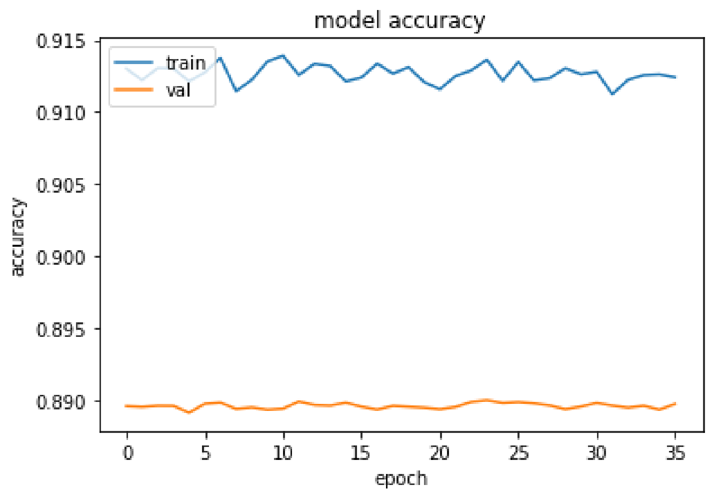


Figure 5.56: Model Accuracy

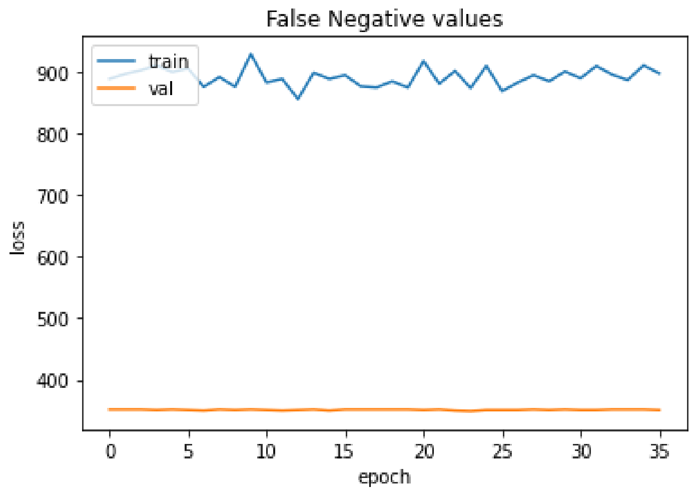


Figure 5.57: Model False negative values

Experiment four

In this experiment we will use ResNet50 architecture with constant and cycling learning rate and other hyperparameter tuning.

1. Use the model with augmentation used in the last experiment, 10^{-5} constant learning rate, and standardization.

The next figures present what happen after train 24 epoch with this model.

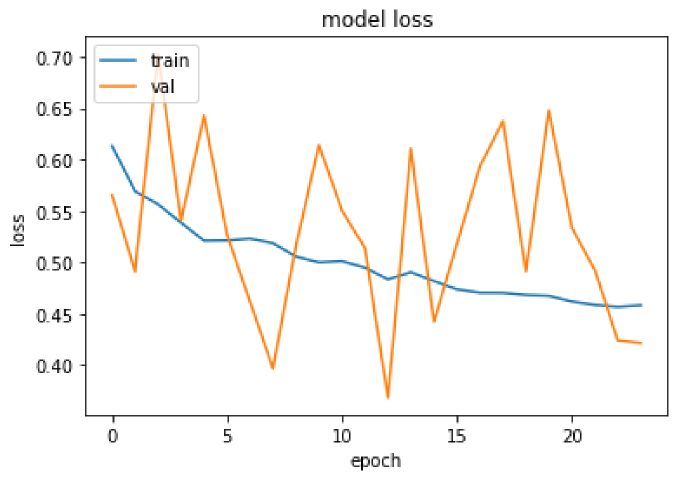


Figure 5.58: Model Loss

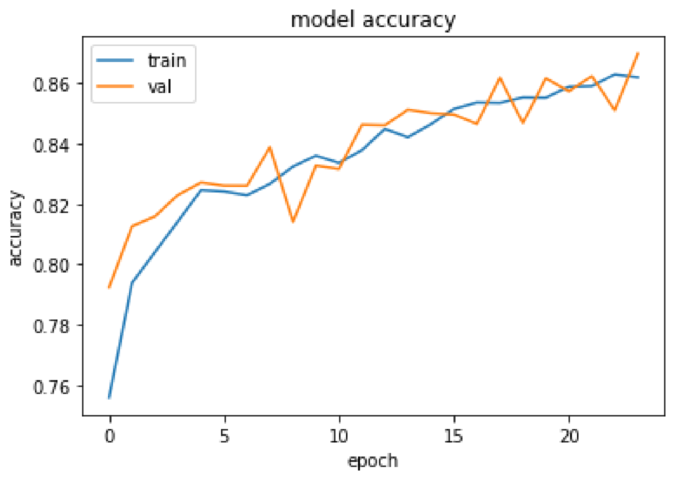


Figure 5.59: Model Accuracy

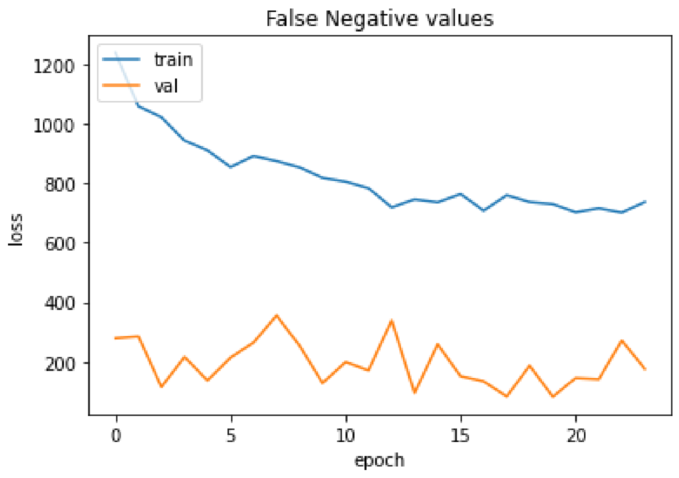


Figure 5.60: Model False negative values

The model performance is good and not overfit training data so we will only change zoom augmentation value to range and run again.

As shown in the figures (61,62,63) after 35 epoch the model did very well in Accuracy until epoch 15 and then it overfit the training data, in loss the results as high in both training and validation, and in False Negative values it did will in validation but not very good in training, but it is acceptable.

But until now it is not satisfied, we change a lot of hyper parameter and run the model many times on high number of GPU hours and still in the same range of validation accuracy next we will try using cycling learning rate and see the difference.

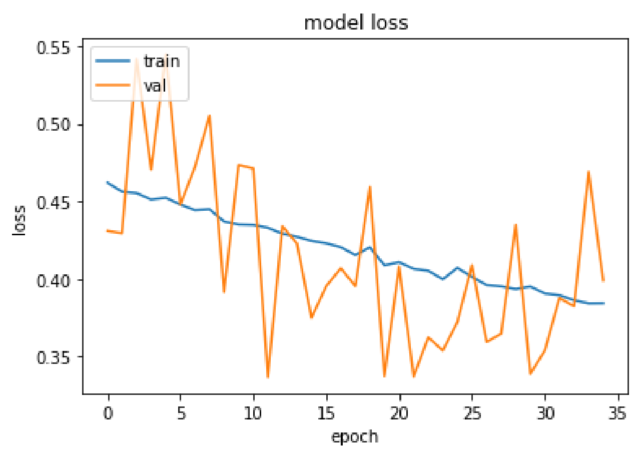


Figure 5.60: Model Loss

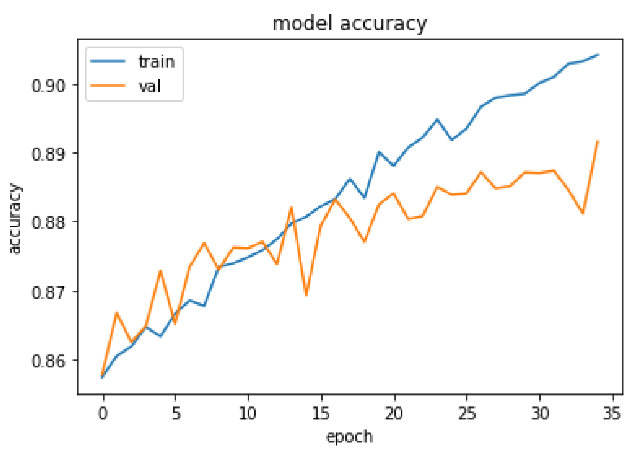


Figure 5.62: Model Accuracy

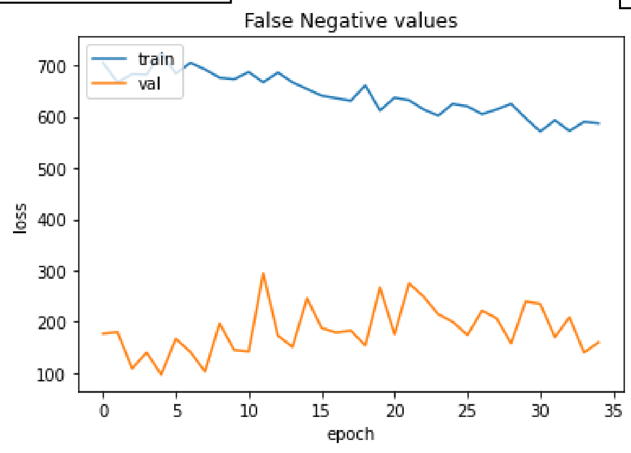


Figure 5.63: Model False negative values

2. Using cycling learning rate with 10^{-2} as max and 10^{-5} as min this numbers defined after a lot of experiments to define the best values.

The next figures present what happen after train 28 epoch with this model.

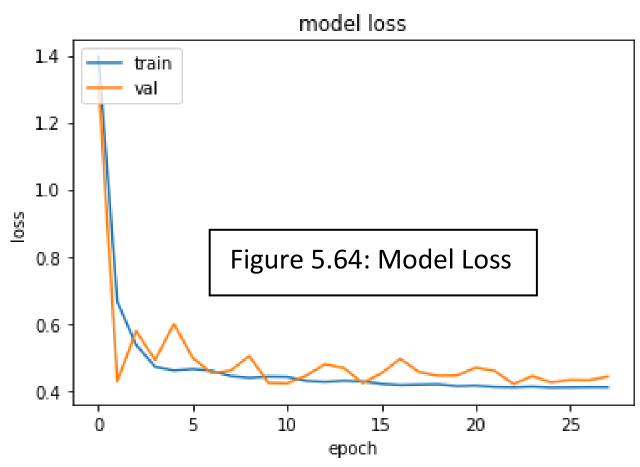


Figure 5.64: Model Loss

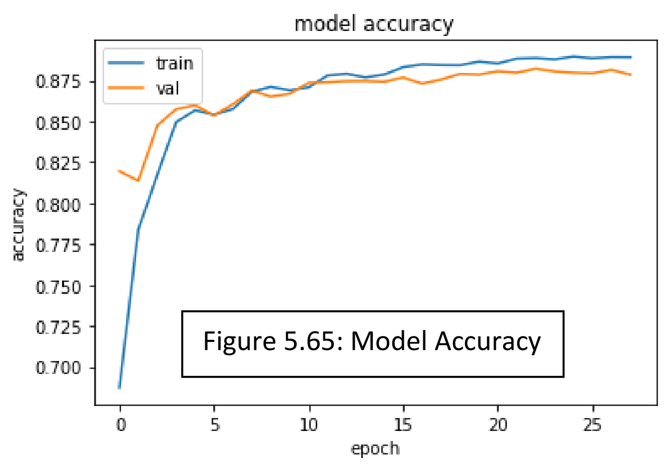


Figure 5.65: Model Accuracy

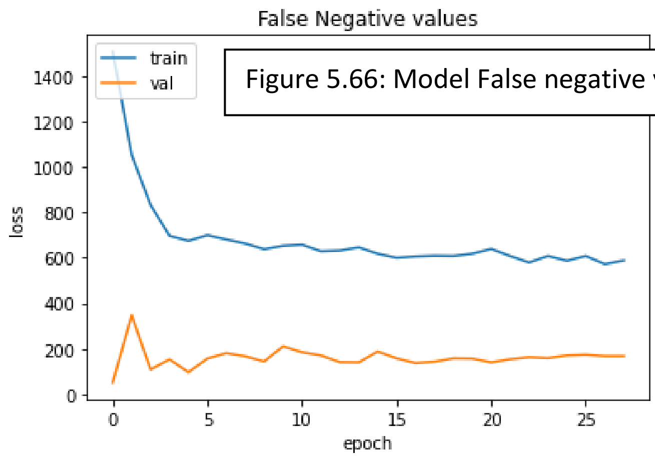


Figure 5.66: Model False negative values

The model performance is very good in all accuracy, loss and false negative values comparing to the model with constant learning rate.

Next, we will add some linear layers with dropout to run the model a lot 71 epoch on three times to see the effect of running the model on a lot of GPU hours.

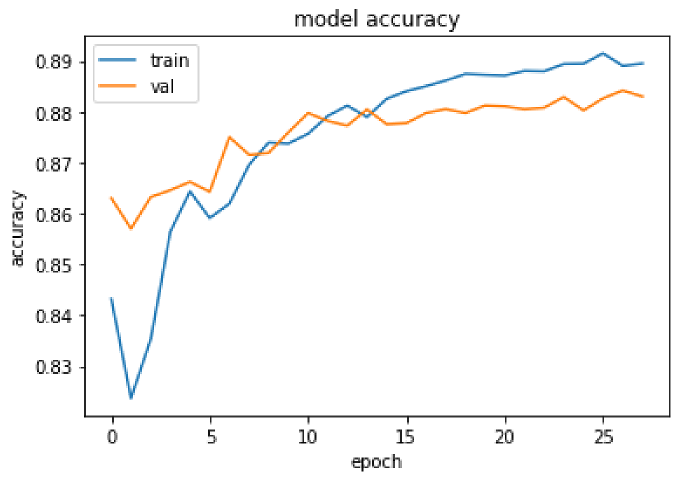


Figure 5.67: Model Accuracy of first 28 epoch

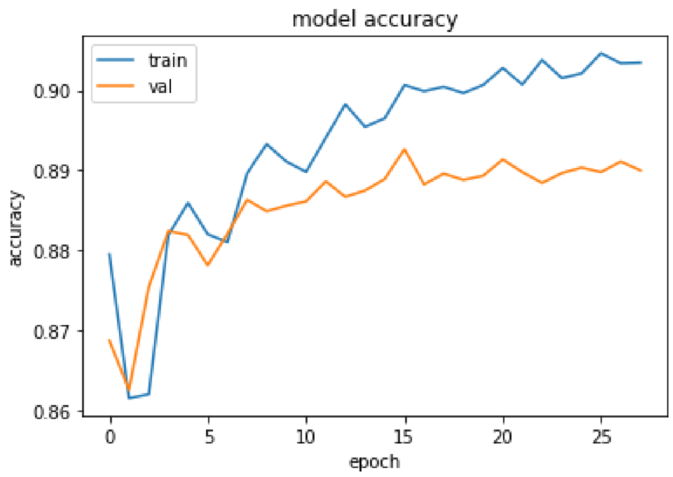


Figure 5.68: Model Accuracy of second 28 epoch

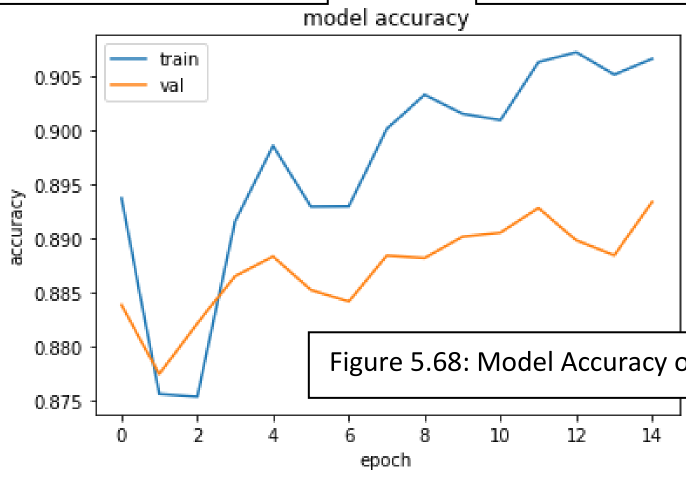


Figure 5.68: Model Accuracy of the last 15 epoch

The model improves slightly so we will try to go with transfer learning and see the effect.

Experiment five

A lot of tunning happened but the model still needs to be improved so next we will use weights of "imagenet" dataset as initialization to the model (VGG16), we back to VGG16 to see first what will be the improve with this "simple" model.

The changes presented next:

- First the images standardized as in the previous experiment, use zoom, vertical flip and horizontal flip as augmentation, back to constant learning rate instead of cycling learning rate and VGG16 model with three dense layers with dropout.

The next figures present what happen after train 26 epoch with this model.

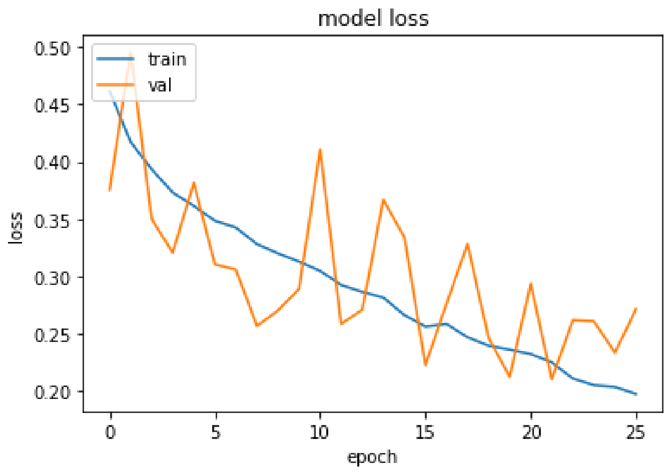


Figure 5.69: Model Loss

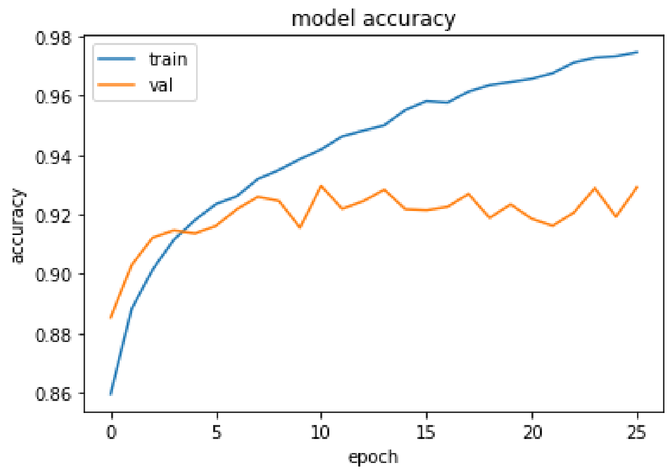


Figure 5.70: Model Accuracy

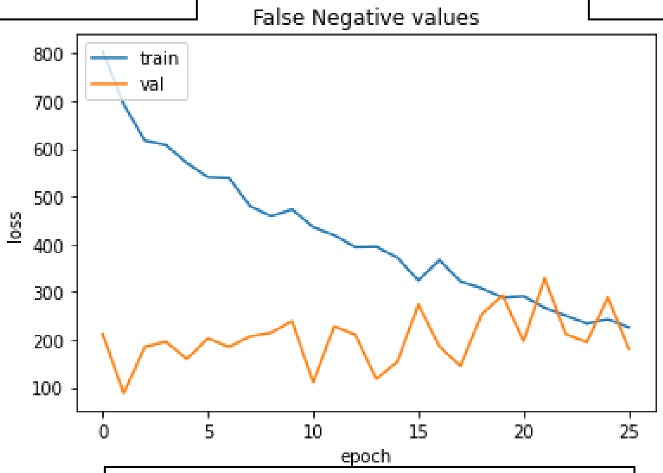


Figure 5.71: Model False negative values

The training and validation accuracy reach to 97.5% and 92.4% respectively, the Loss in training is so good but not in validation.

The test accuracy in the competition is 0.8837 as private score and 0.9149 as public score.

So, the model reaches to a good point, but we have to treat this overfitting.

- Add rotation, shear range, width shift range and height shift range augmentation.

The next figures present what happen after train 26 epoch with this model.

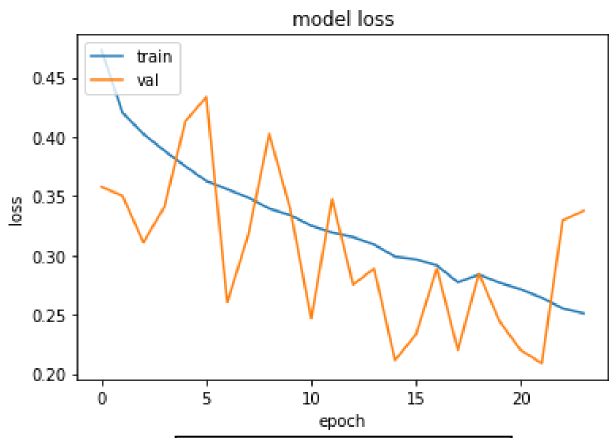


Figure 5.72: Model Loss

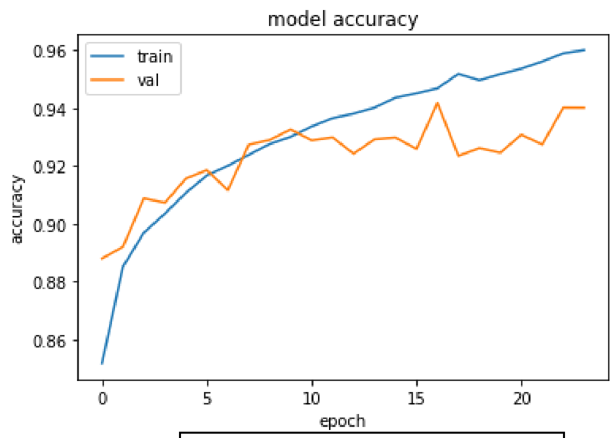


Figure 5.73: Model Accuracy

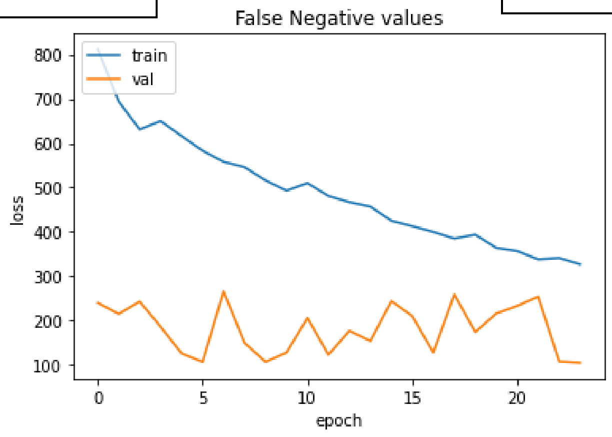


Figure 5.74: Model False negative values

The model improved as shown in the figures the training accuracy decreased and validation accuracy increased, the Loss of validation is better than the previous and in False negative values its so good comparing to the previous.

The test accuracy in the competition is 0.8813 as private score and 0.8890 as public score.

The test accuracy of the previous model is slightly better than this model so next we will try to merge the two models.

- Take the weights of the first model and use it as initialization with the second model augmentation.

The next figures present what happen after train 27 epoch with this model.

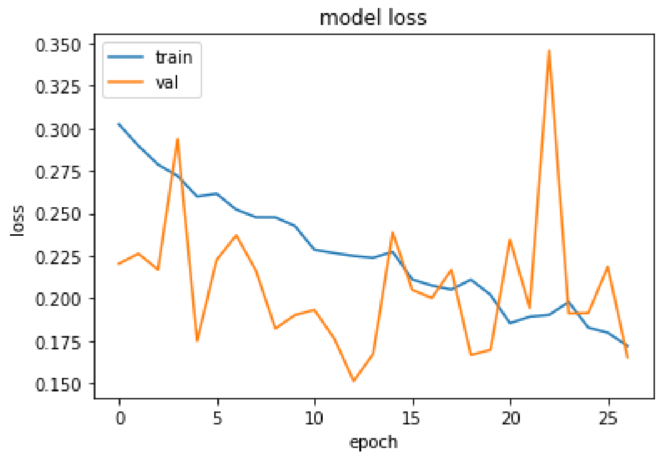


Figure 5.75: Model Loss

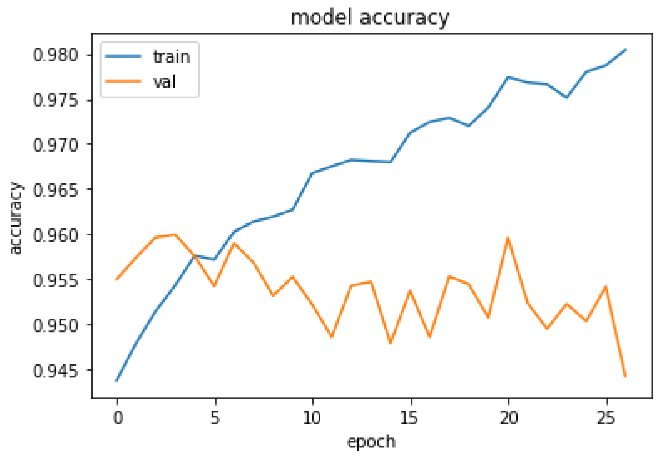


Figure 5.76: Model Accuracy

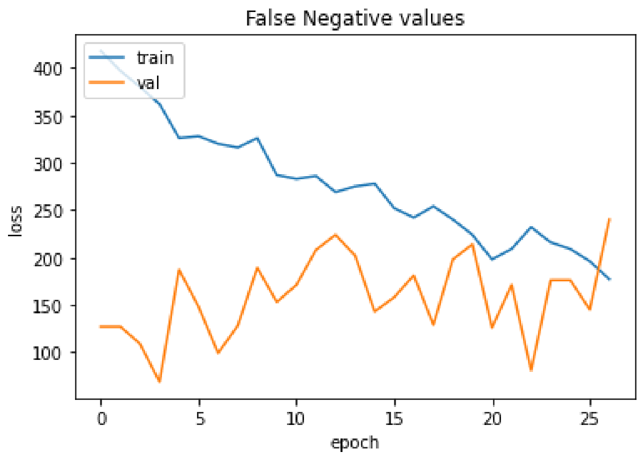


Figure 5.77: Model False negative values

From these figures and the previous experiments when retrain the model with another model weights it always overfit the data may this because the weights still not as good to use as initialization.

Experiments Six

The main idea in this experiment is using the “imagenet” dataset weights as initialization to the model with a lot of tunning to reach to the best model.

Firstly, we will use VGG16 model and do a lot of tunning then go with a more complex model.

Next figures present the results of using VGG16 with imagenet weights with the last model.

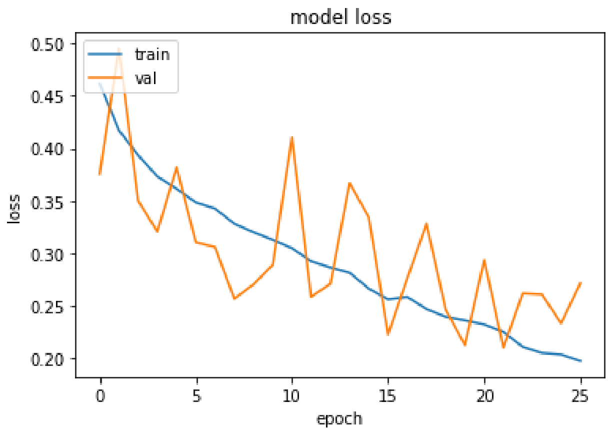


Figure 5.78: Model Loss

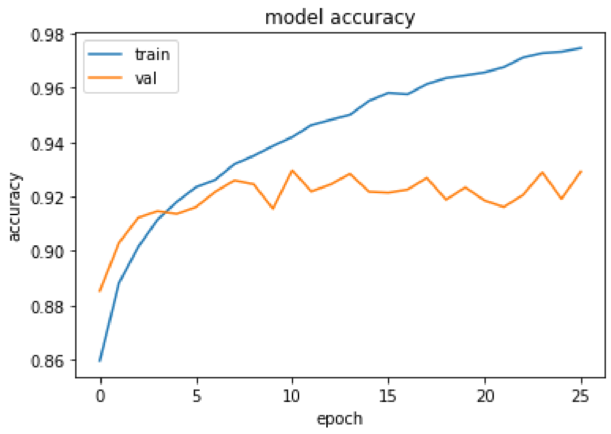


Figure 5.79: Model Accuracy

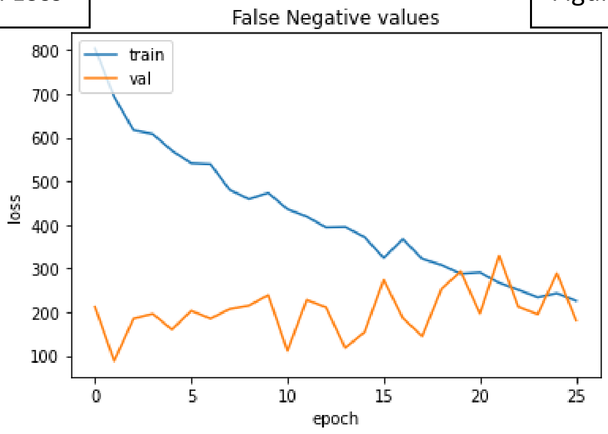


Figure 50: Model False negative values

From the above model is a very satisfied for us in the training accuracy needs to some improve in validation we will try to that next.

The test results of this model n the competition is 0.8837 as private score and 0.9149as public score.

Like the other experiments we see that augmentation reduces the overfitting so we will add some augmentation like rotation range, width shift range, shear range.

Next figures represent the model results after adding the above augmentation and run with 24 epochs.

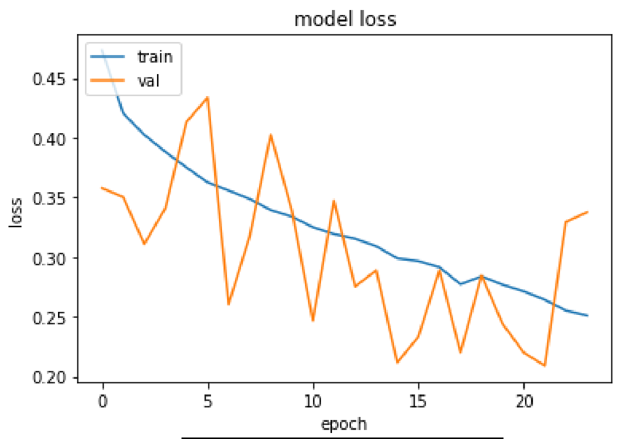


Figure 51: Model Loss

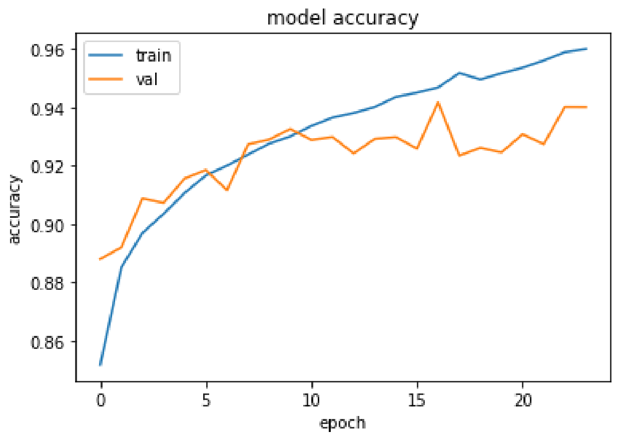


Figure 52: Model Accuracy

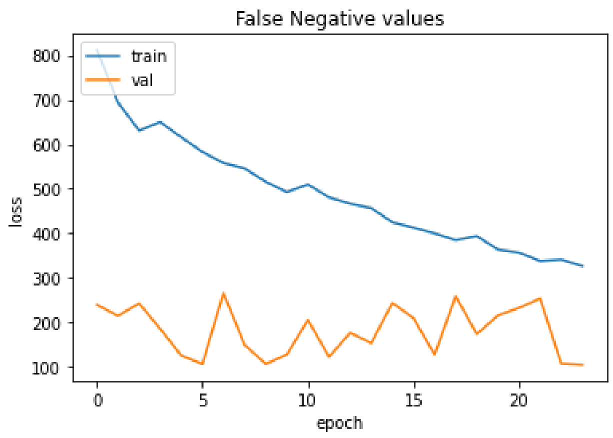


Figure 53: Model False negative values

The model train accuracy decreased, and the validation increased causing a great performance in reducing overfitting.

But we are computing the model performance according to the competition, so this model test results in the competition is 0.8813 as private score and 0.8890 as public score, again it reduces the overfitting.

Next, we will try a bigger network architecture (VGG19), trying to increase the accuracy.

The next figures present this model results after using a bigger network.

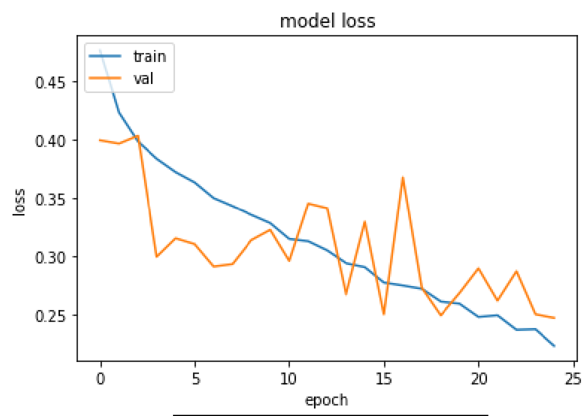


Figure 54: Model Loss

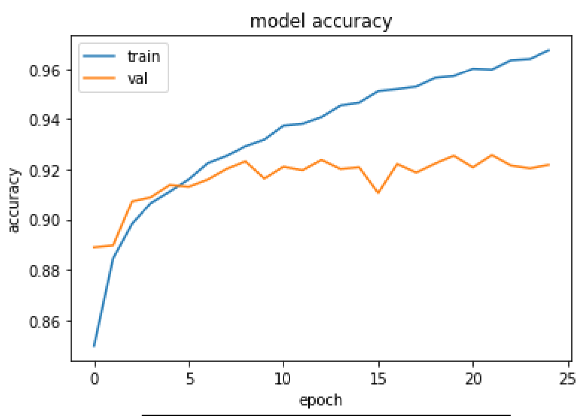


Figure 55: Model Accuracy

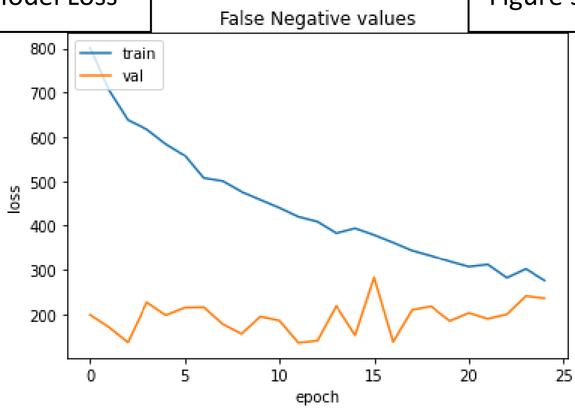


Figure 56: Model False negative values

The model performance goes down in both train and validation and return to the previous overfitting.

We will back the previous model and tune learning rate to reach to one close to the optimal learning rate, we try 12 values up and down the 10^{-5} learning rate and find that $10^{-5.5}$ is the closest of what we try this is according to the competition results.

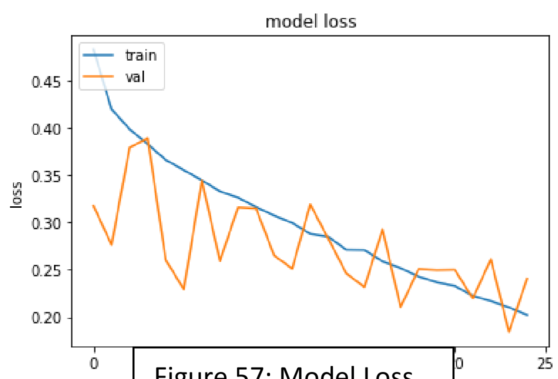


Figure 57: Model Loss

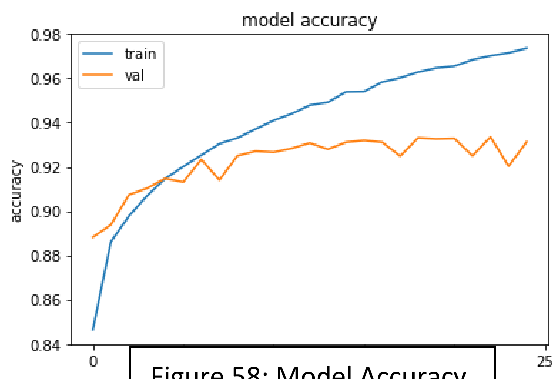


Figure 58: Model Accuracy

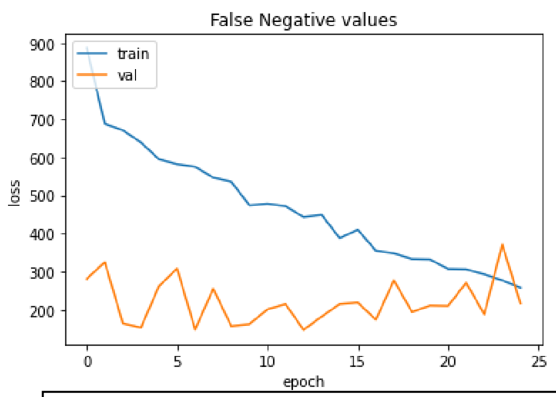


Figure 59: Model False negative values

The competition test results of this model are 0.8912 as private score and 0.8912 as public score.

Let’s try removing dropouts and organize the values of dense layers units, we use this experiment with both learning rates 10^{-5} and $10^{-5.5}$ and let’s see the results.

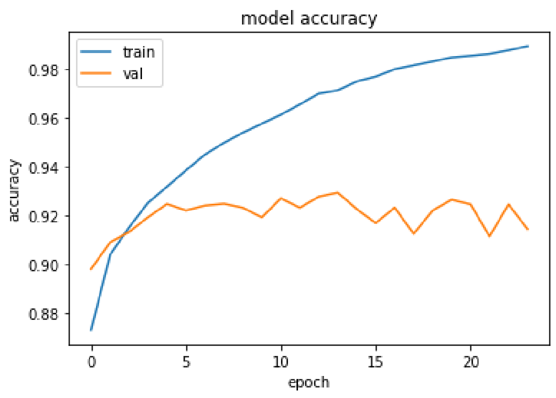


Figure 61: Model Accuracy with LR=$10^{-5.5}$

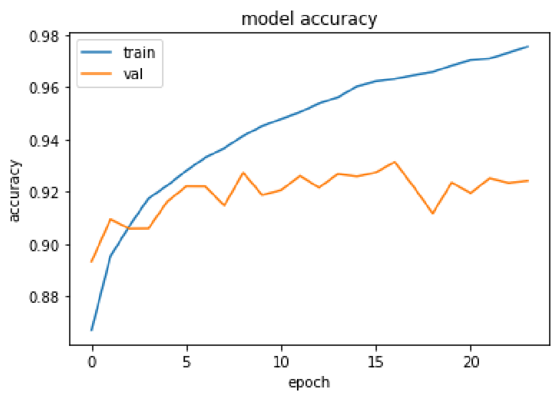


Figure 60: Model Accuracy with LR=10^{-5}

The test results of the first is 0.8873 as private score and 0.9092 as public score, for the second model private score is 0.8826 and public score is 0.8977.

From that we found that adding the dense layers and removing dropout changes the optimal learning rate for this model.

Next, we will try bigger networks like ResNet50, ResNet101 and DenseNet169, these network architectures share the complexity of connecting the previous layers with the next.

We will tray the three architectures in parallel with different augmentation and with 10^{-5} learning rate.

The different augmentation back to the size of each network layers and the learning rate 10^{-5} because it was tried with architectures like this before and did will, next we will try the learning rate that tuned above.

- Use ResNet50 with imagenet weights and zoom, horizontal flip and vertical flip augmentation.

Next, figures represent the mode performance in 27 epochs.

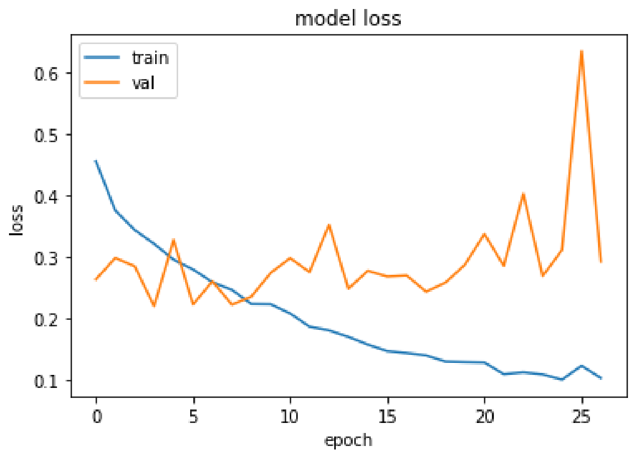


Figure 62: Model Loss

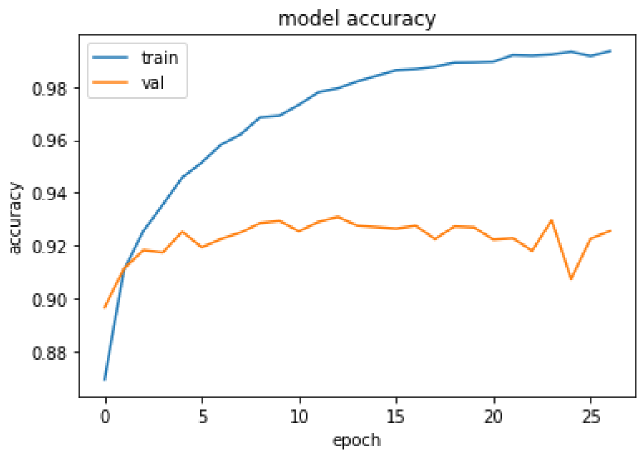


Figure 63: Model Accuracy

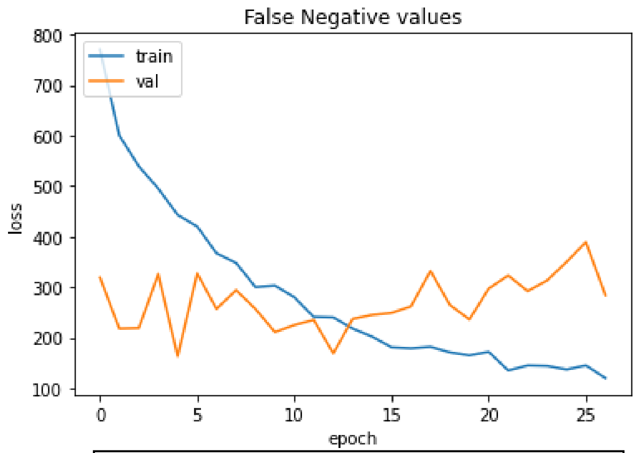


Figure 64: Model False negative values

The model performance is not good training goes in a very good way, but validation does not improve in the two accuracy matrices and loss function.

This model test results in the competition is 0.8695 as private score and 0.8755 as public score, the low performance defines in the competition results.

- Use ReseNet101 with imagenet weights, zoom, horizontal flip, vertical flip augmentation, width shift and height shift augmentation and add more dense layers.

Next, figures represent the mode performance in 27 epochs.

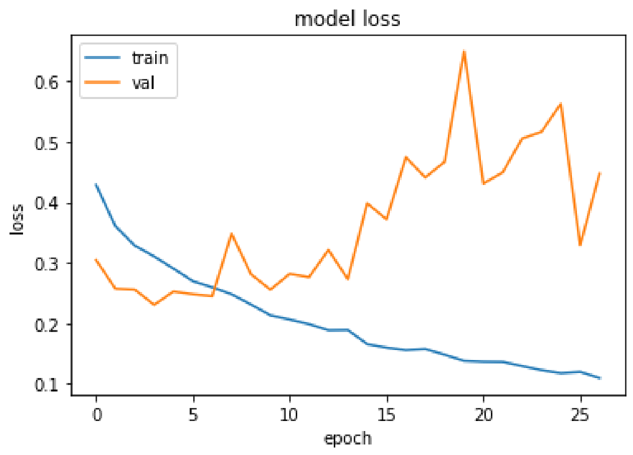


Figure 65: Model Loss

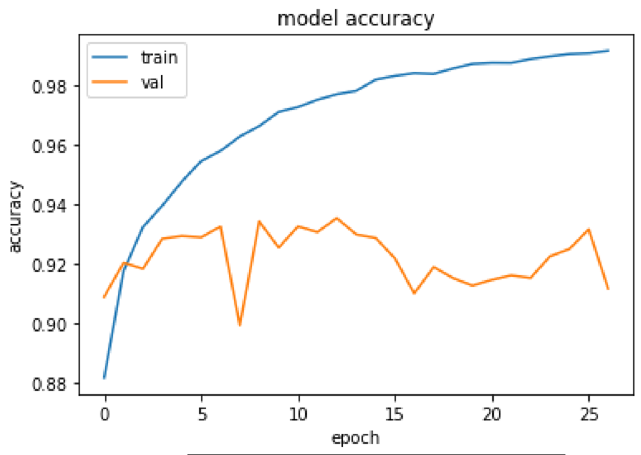


Figure 66: Model Accuracy

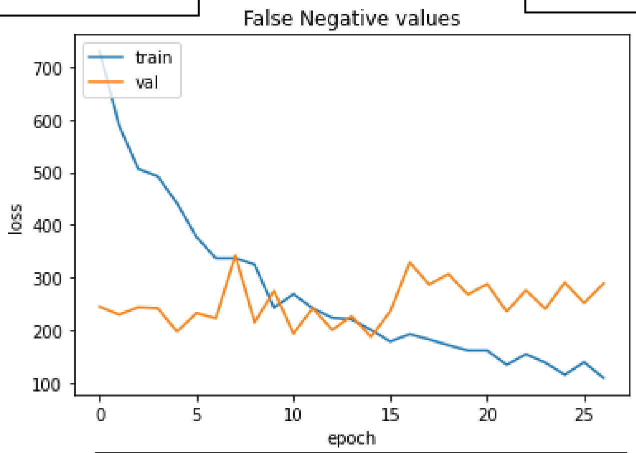


Figure 67: Model False negative values

The model doing very good in training and start good in validation at a certain epoch does not increase may decrease but we save the model with the highest accuracy may this affect the testing performance.

This model test results in the competition is 0.8901 as private score and 0.8645 as public score, the effect of saving the model with the highest accuracy presented in the competition results.

- Use DenseNet169 with imagenet weights, zoom, horizontal flip, vertical flip augmentation, width shift, height shift, shear augmentation and add more dense layers.

Next, figures represent the mode performance in 28 epochs.

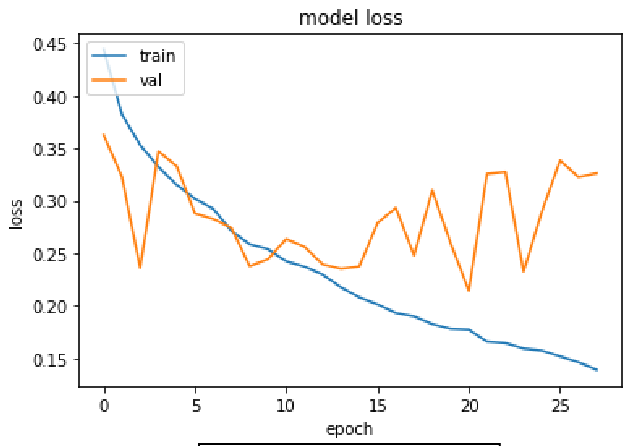


Figure 68: Model Loss

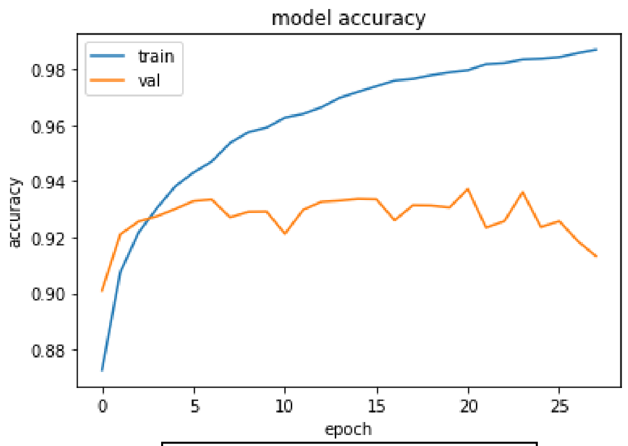


Figure 69: Model Accuracy

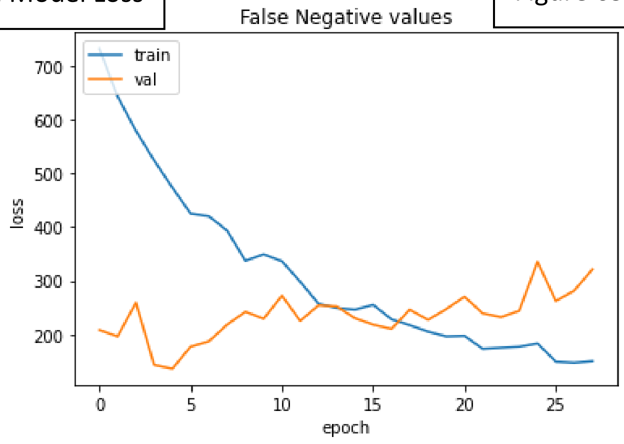


Figure 70: Model False negative values

Like the previous model this model doing very good in training and start good in validation at a certain epoch does not increase may decrease but we save the model with the highest accuracy may this affect the testing performance.

This model test results in the competition is 0.8819 as private score and 0.8984 as public score, the previous model doing good in the test results than this model.

The best model of the three in the test results is ResNet101, so we will us $10^{-5.5}$ learning rate to see the effect on this model.

Next, figures represent the mode performance in 26 epochs.

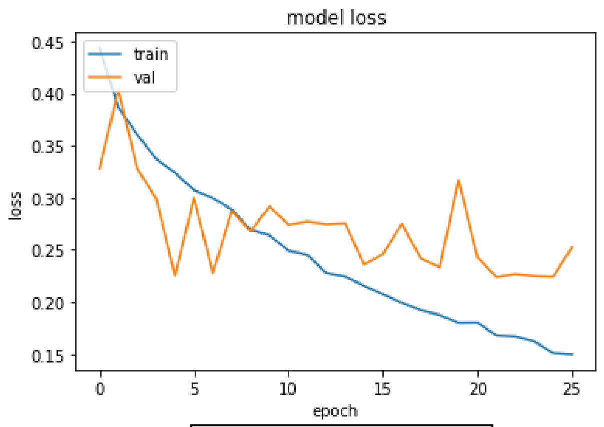


Figure 71: Model Loss

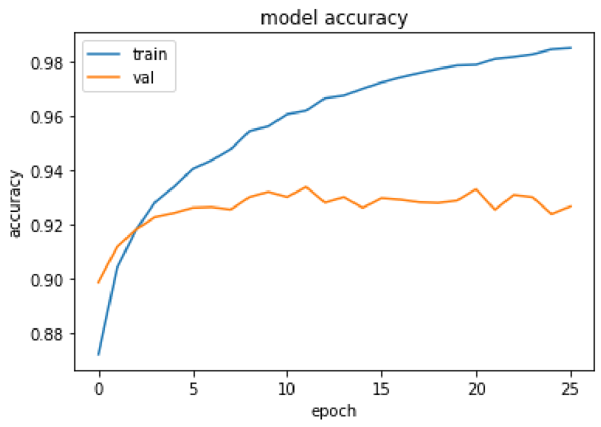


Figure 72: Model Accuracy

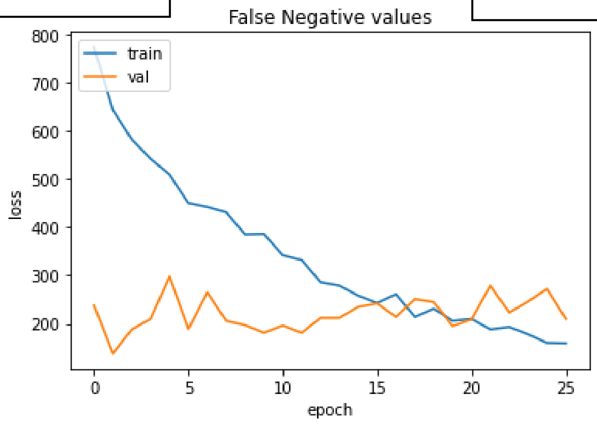


Figure 73: Model False negative values

The improves in the matrices and Loss Function comparing to it before changing the learning rate.

This model test results in the competition is 0.9002 as private score and 0.8784 as public score.

Now we are with the best model of all that we try in the experiments but we facing a problem that the size of the model as very big this is not good so we try a lot of techniques to reduce the make model with this accuracy and small size, some of the tried techniques are L1 regularization, L2 regularization, Drop out regularization…etc, all of these techniques reduces the size but the accuracy become so bad until we reach to using 1x1 convolution instead of dense layers and we did a lot of experiments on it until reaching to the next model.

Next, figures represent the mode performance in 28 epochs.

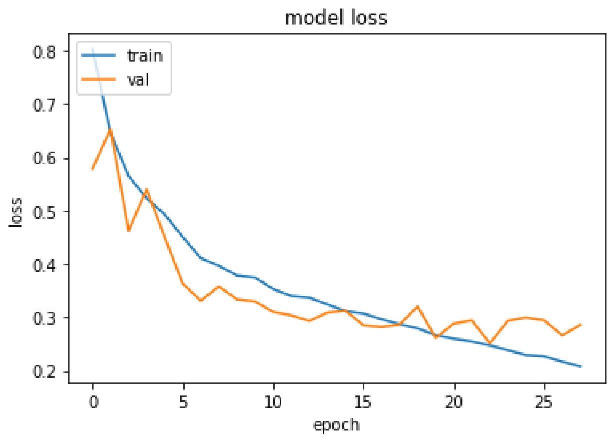


Figure 74: Model Loss

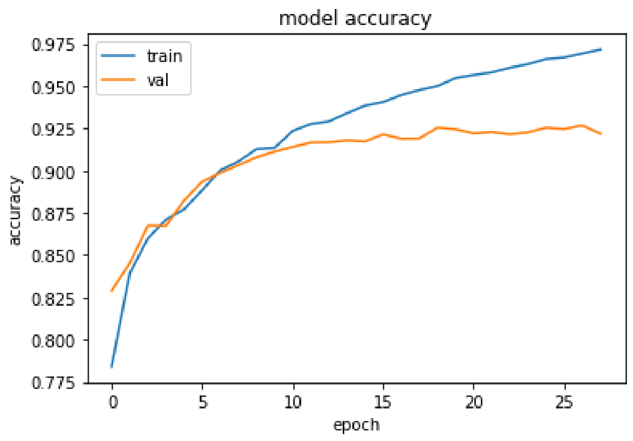


Figure 75: Model Accuracy

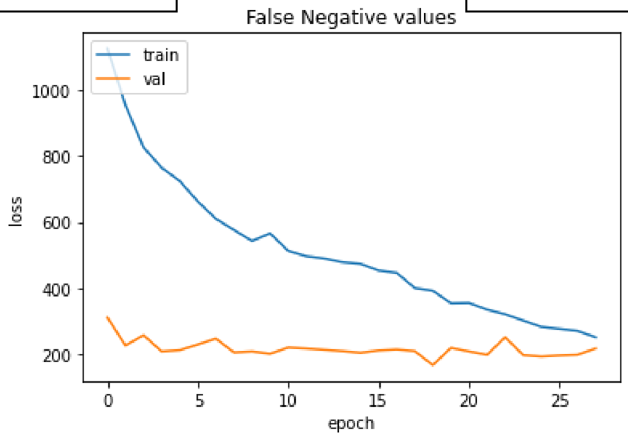


Figure 76: Model False negative values

As shown in the figure this model has a very good performance comparing to the last models.

This model test results in the competition is 0.9017 as private score and 0.8863 as public score, so we tend to go with this model and deploy it to connect with the embedded device.

We can say that the process of improving the model stopped now as its will not stop ever we enter the market to save lives in we will always work on this goal.

Deployment

This section will concern on deploy Flask web server to Azure cloud services. Firstly, we will create a Azure Virtual Machine (VM) which is one of several types of on-demand, scalable computing resources that Azure offers.

We will use here Ubuntu VM with Standard B2s size to deploy our Python application. Once we created the VM, we connect to the machine using SSH protocol. After that we cloned the API from GitHub.

```
$ git clone https://github.com/GProjet/Melanosizer.git
```

We navigate to API directory and run the following commands

```
$ sudo apt-get update
```

```
$ python3 -m venv env
```

```
$ source env/bin/activate
```

```
$ pip3 install -r requirements.txt
```

```
$ export FLASK_APP=run.py
```

```
$ python run.py
```

Now, our application is running at port 5000.

Next, we will use Nginx reverse proxy server that takes a client request, passes it on to servers, and subsequently delivers the server's response back to the client. We install the nginx using the following command:

```
$ sudo apt install nginx -y
```

One last thing, we will use Gunicorn which is built so many different web servers can interact with it. It also does not really care what you used to build our web application.

To install Gunicorn use the following command:

```
$ pip3 install gunicorn
```

And we will navigate to API directory with running the following command

```
$ gunicorn –bind 0.0.0.0:5000 app:app
```

Now, our web server is running on port 80 based on Nginx reverse proxy within the VM.

Embedded

Design Constraints:

In this chapter we discuss the basic design constraints, the hardware and software environments.

Hardware Environment

A melanosizer is a device drawn or powered by external electricity or connected to the pc or laptop.

it is controlled from the Raspberry Pi TOUCH SCREEN LCD HDMI that connects to the Raspberry Pi Camera Module to your Raspberry Pi and take pictures, record video, and apply image effects.

Raspberry Pi TOUCH SCREEN LCD HDMI

1. **Raspberry pi touch display**

The Raspberry Pi Touch Display is an LCD display which connects to the Raspberry Pi through the DSI connector. In some situations, it allows for the use of both the HDMI and LCD displays at the same time (this requires software support)

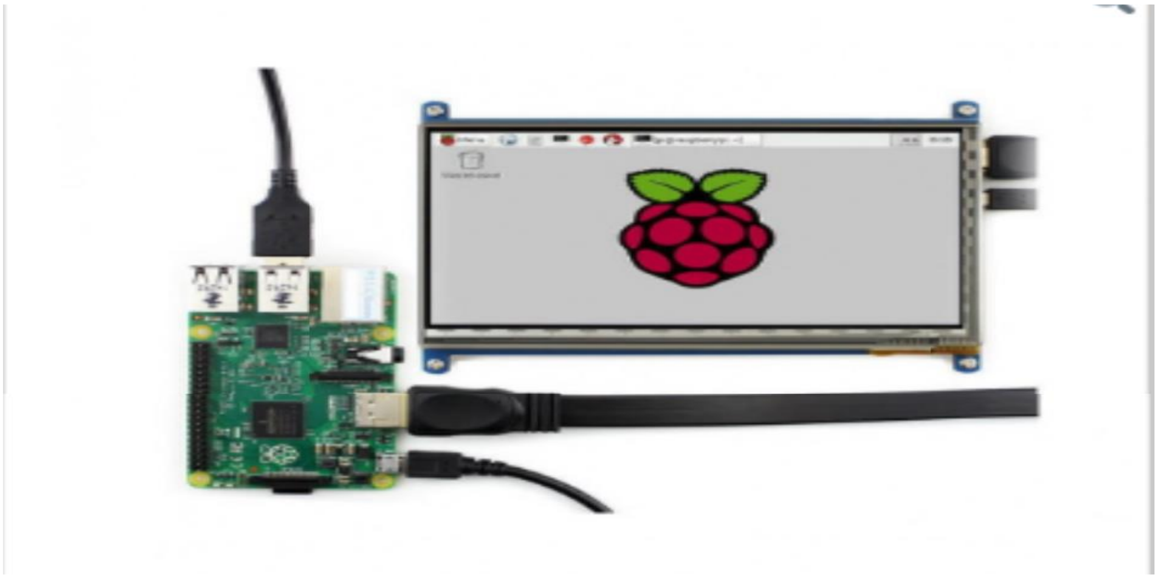


Figure 5.77: Raspberry Pi TOUCH SCREEN LCD HDMI

2. **Board support**

The DSI display is designed to work with all models of Raspberry Pi, however early models that do not have mounting holes (the Raspberry Pi 1 Model A and B) will require additional mounting hardware to fit the HAT-dimensioned stand-offs on the display PCB

3. **Physical Installation**

The following image shows how to attach the Raspberry Pi to the back of the Touch Display (**if required**), and how to connect both the data (**ribbon cable**) and power (**red/black wires**) from the Raspberry Pi to the display. If you are not attaching the Raspberry Pi to the back of the display, take extra care when attaching the ribbon cable to ensure it is the correct way round. The black and red power wires should be attached to the GND and 5v pins respectively.

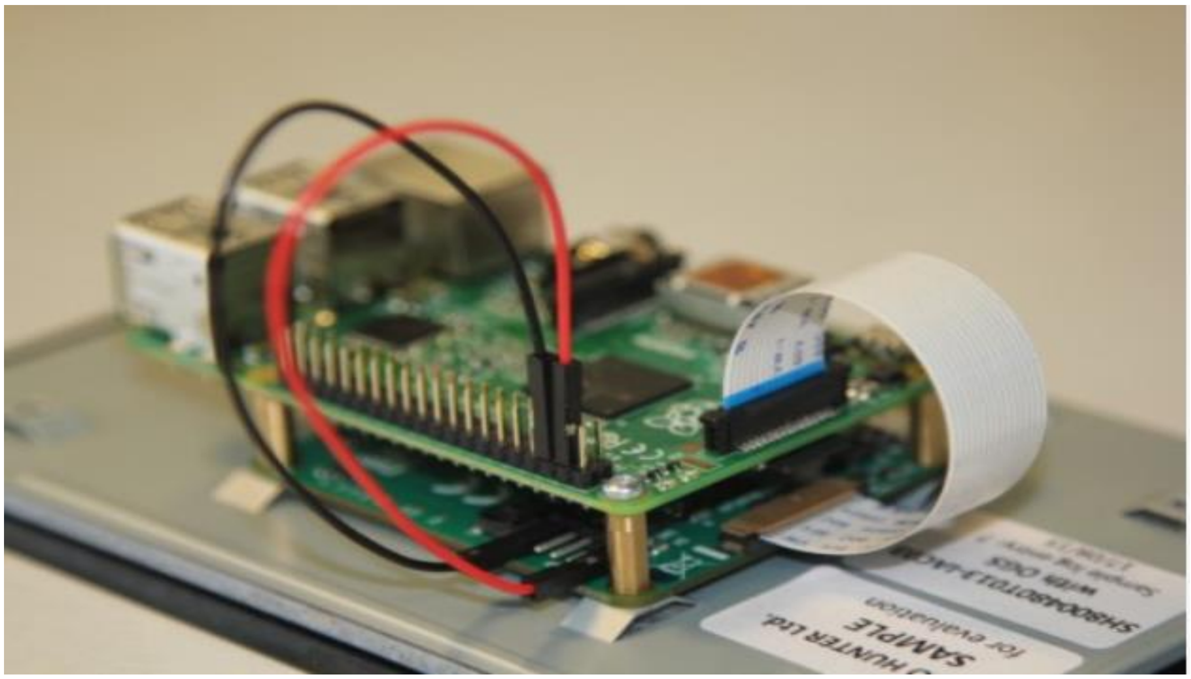


Figure 5.78 Raspberry Pi to the back of the Touch Display

4. **specifications**

- 800×480 RGB LCD display
- 24-bit colour
- Industrial quality: 140-degree viewing angle horizontal, 130-degree viewing angle vertical
- 10-point multi-touch touchscreen
- PWM backlight control and power control over I2C interface
- Metal-framed back with mounting points for Raspberry Pi display conversion board and Raspberry Pi
- Backlight lifetime: 20000 hours
- Operating temperature: -20 to +70 degrees centigrade
- Storage temperature: -30 to +80 degrees centigrade

- Contrast ratio: 500
- Average brightness: 250 cd/m2
- Power requirements: 200mA at 5V typical, at maximum brightness.

Raspberry Pi computer with a Camera Module port

All current models of Raspberry Pi have a port for connecting the Camera Module.

There are two versions of the Camera Module:

- The standard version, which is designed to take pictures in normal light
- The NoIR version, which doesn't have an infrared filter, so you can use it together with an infrared light source to take pictures in the dark

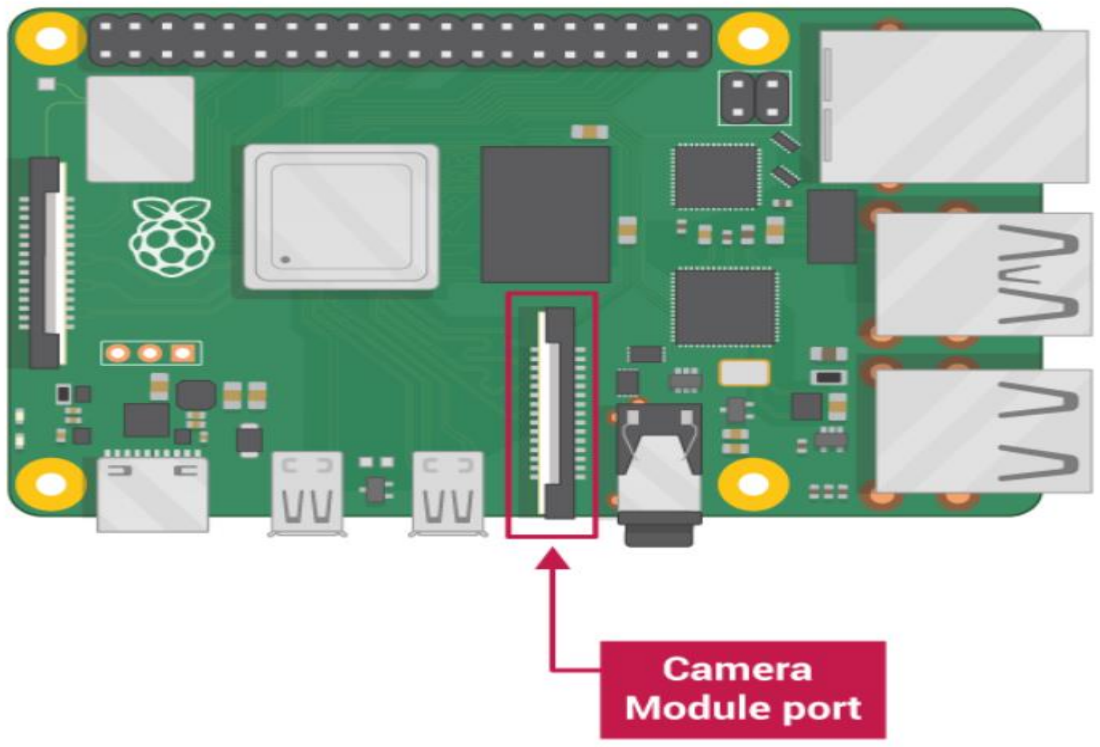


Figure 5.79: camera module port

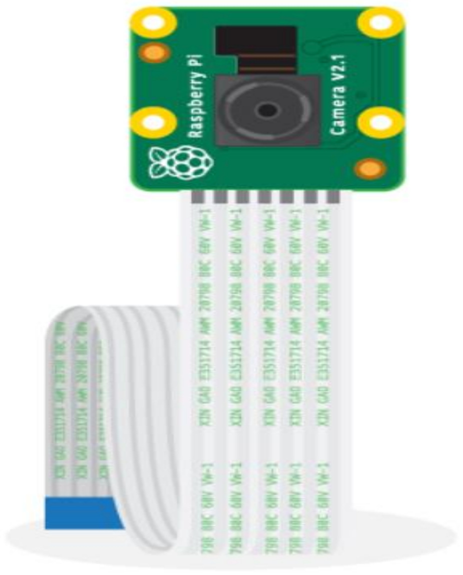


Figure 5.80: Raspberry Pi Camera Module

1. **Connect the Camera Module**

Ensure your Raspberry Pi is turned off.

1. Locate the Camera Module port
2. Gently pull up on the edges of the port’s plastic clip
3. Insert the Camera Module ribbon cable; make sure the connectors at the bottom of the ribbon cable are facing the contacts in the port.
4. Push the plastic clip back into place

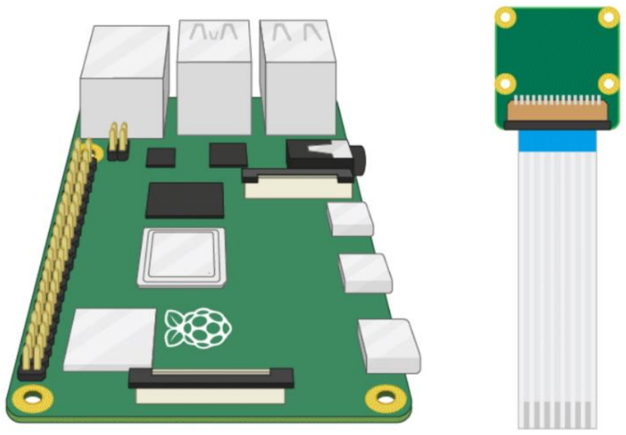


Figure 5.81: shows step 1.2

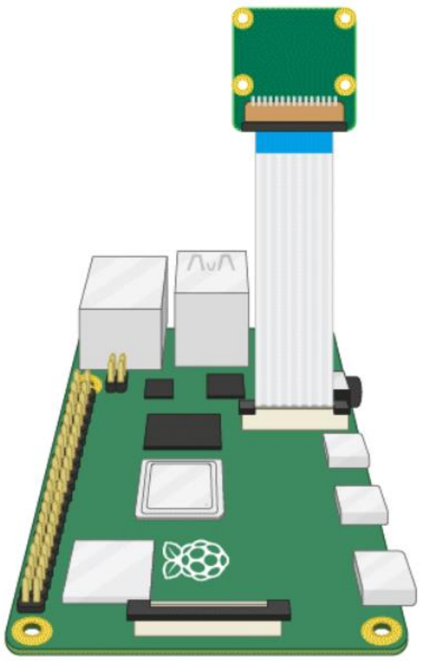
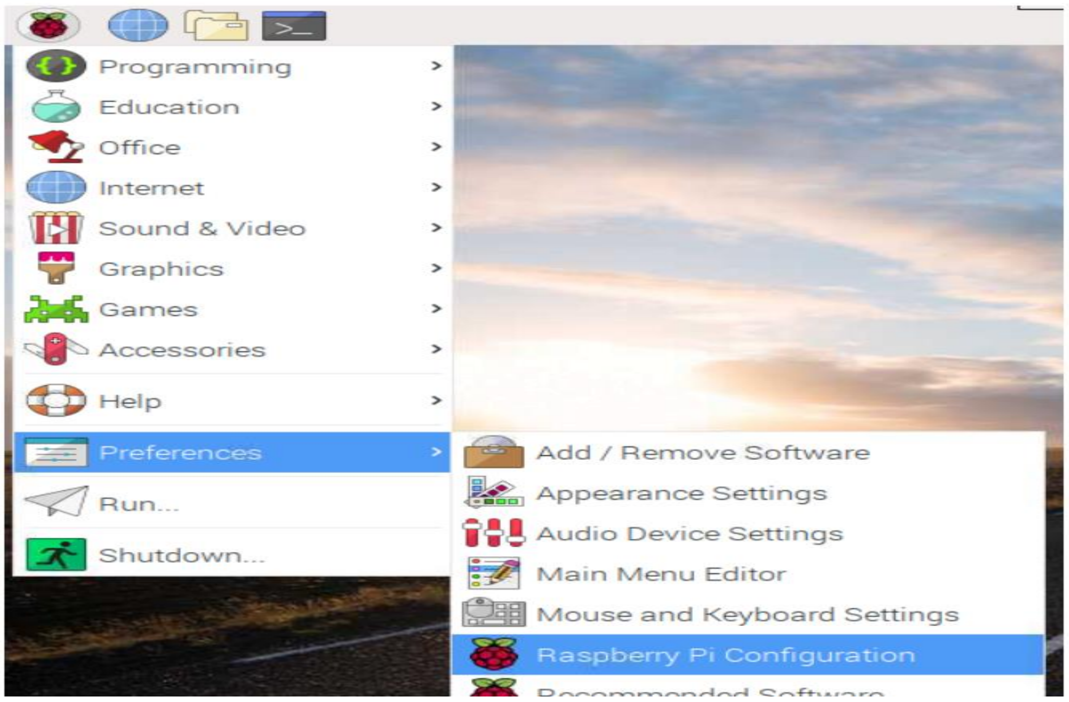
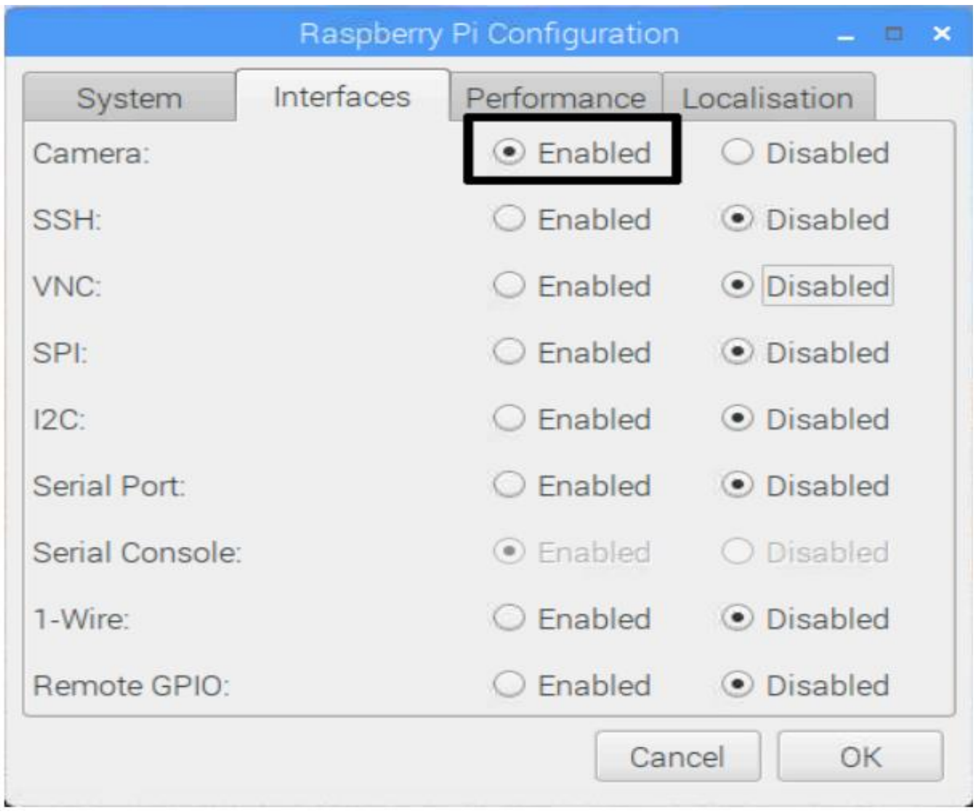


Figure 5.82:
show steps 3,4

- Start up your Raspberry Pi.
- Go to the main menu and open the Raspberry Pi Configuration tool.



- Select the Interfaces tab and ensure that the camera is enabled:



Camera:	[✓ Enabled]	Disabled
SSH:	Enabled	[✓ Disabled]
VNC:	Enabled	[✓ Disabled]
SPI:	Enabled	[✓ Disabled]
I2C:	Enabled	[✓ Disabled]
Serial Port:	Enabled	[✓ Disabled]
Serial Console:	[✓ Enabled]	Disabled
1-Wire:	Enabled	[✓ Disabled]
Remote GPIO:	Enabled	[✓ Disabled]

- Reboot your Raspberry Pi.

Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B (Pi4B) is the first of a new generation of Raspberry Pi computers supporting more RAM and with siginficantly enhanced CPU, GPU and I/O performance; all within a similar form factor, power envelope and cost as the previous generation Raspberry Pi 3B+. The Pi4B is avaiable with either 1, 2 and 4 Gigabytes of LPDDR4 SDRAM.

1. **GPIO**

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all current Raspberry Pi boards (unpopulated on Raspberry Pi Zero, Raspberry Pi Zero W and Raspberry Pi Zero 2 W). Prior to the Raspberry Pi 1 Model B+ (2014), boards comprised a shorter 26-pin header.

Voltages

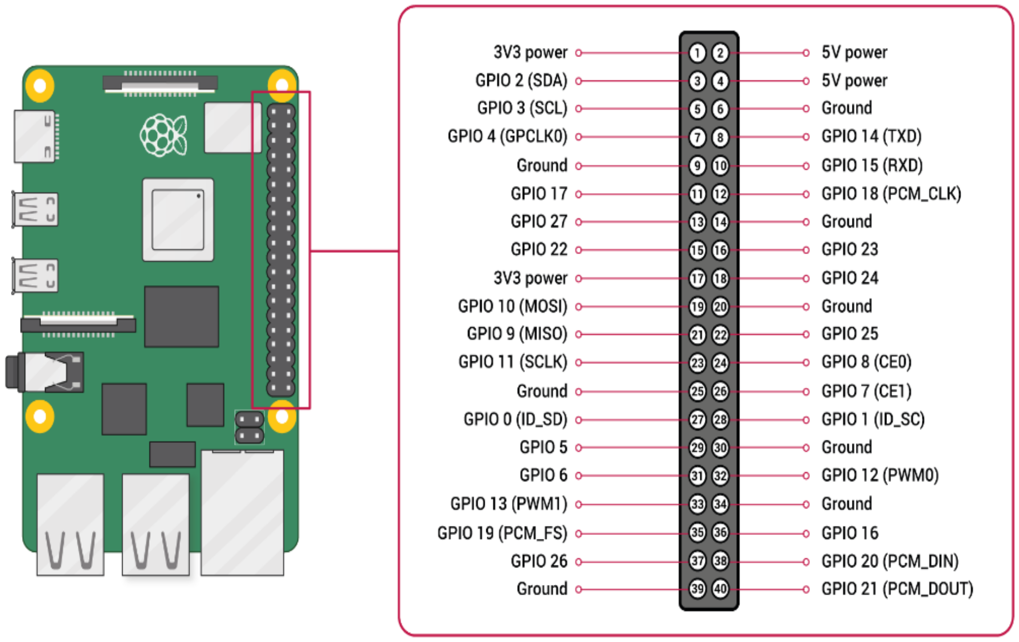
Two 5V pins and two 3.3V pins are present on the board, as well as a number of ground pins (0V), which are unconfigurable. The remaining pins are all general purpose 3.3V pins, meaning outputs are set to 3.3V and inputs are 3.3V-tolerant.

Outputs

A GPIO pin designated as an output pin can be set to high (3.3V) or low (0V).

Inputs

A GPIO pin designated as an input pin can be read as high (3.3V) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.



Software Environment

Operating system

Raspberry Pi OS is a free operating system based on Debian, optimised for the Raspberry Pi hardware, and is the recommended operating system for normal use on a Raspberry Pi. The OS comes with over 35,000 packages: precompiled software bundled in a nice format for easy installation on your Raspberry Pi. Raspberry Pi OS is under active development, with an emphasis on improving the stability and performance of as many Debian packages as possible on Raspberry Pi.

Updating Raspberry Pi OS

It’s important to keep your Raspberry Pi up to date. The first and probably the most important reason is security A device running Raspberry Pi OS contains

millions of lines of code that you rely on. Over time, these millions of lines of code will expose well-known The only way to mitigate these exploits as a user of Raspberry Pi OS is to keep your software up to date
The second reason, related to the first, is that the software you are running on your device most certainly contains bugs. Some bugs are CVEs, but bugs could also be affecting the desired functionality without being related to security. By keeping your software up to date, you are lowering the chances of hitting these bugs.

Using APT

The easiest way to manage installing, upgrading, and removing software is using APT (Advanced Packaging Tool) from Debian. To update software in Raspberry Pi OS, you can use the apt tool from a Terminal window.

Using rpi-update

rpi-update is a command line application that will update your Raspberry Pi OS kernel and VideoCore firmware to the latest pre-release versions.