A DEEP LEARNING PROJECT

COVID X-RAY IMAGE CLASSIFICATION USING DEEP LEARNING

Submitted By:

Shikhar Saxena

Akash Rawat

Abhishek Chauhan

CONTENT

INTRODUCTION

Most people in the world right now are genuinely concerned about COVID-19. Everyone is finding themselves constantly analysing their personal health and wondering if/when they will contract it. As humans, there is nothing more terrifying than the unknown. But thanks to the ever working health professionals, researchers and other related agencies, situation has definitely improved, if not totally resolved. Multiple vaccines have now been in usage, there are different tests to confirm if someone has gotten any variant of the virus.

The present report on the topic, **"Detecting COVID-19 in X-Ray images using Deep Learning, Keras and Tensorflow",** is an attempt to design a way for identifying COVID positive individuals from the rest of the people. It uses basis Convolution Neural Network Model, with different techniques applied to optimize and give better accuracy and results.

Though the model presented here gives fairly good results, but in no case can be considered as a substitute or as a valid health certified test among many tests that are available today. This automatic COVID-19 detection is for educational purposes only. It is not meant to be a reliable, highly accurate COVID-19 diagnosis system, nor has it been professionally or academically vetted.

MOTIVATION AND OBJECTIVE

Though the current scenario of the COVID pandemic has definitely gotten better, but now everyone realises how small details and carelessness can lead to terrible consequences. There was a time when there was situation of panic, unknown and unsaid tension that prevailed during pandemic. Thanks to those hardworking health experts, researchers, workers, and other such involved individuals, we are in a much better space now and still preparing to better deal with it.

As a scholar or researcher you don't need a degree in medicine to make an impact in the medical field — deep learning practitioners working closely with doctors and medical professionals can solve complex problems, save lives, and make the world a better place. Our goal is simply to inspire people and open their eyes to how studying computer vision/deep learning and then applying that knowledge to the medical field can make a big impact on the world.

We decided to do what we do best — focus on the overall CV/DL community by writing code, running experiments, and educating others on how to use computer vision and deep learning in practical, real-world applications. That said, we'll be honest, this is not the most scientific report ever written. Far from it, in fact. The methods and datasets used would not be worthy of publication. But they serve as a starting point for those who need to feel like they're doing something to help. It is about caring about the people and community and helping in a way we know.

BRIEF SURVEY/RELATED WORK

When there's panic, there are nefarious people looking to take advantage of others, namely by selling fake COVID-19 test kits after finding victims on social media platforms and chat applications. Given that there are limited COVID-19 testing kits, we need to rely on other diagnosis measures.

For the purposes of this mini-project, we thought to explore X-ray images as doctors frequently use X-rays and CT scans to diagnose pneumonia, lung inflammation, abscesses, and/or enlarged lymph nodes.

Since COVID-19 attacks the epithelial cells that line our respiratory tract, we can use X-rays to analyse the health of a patient's lungs.

And given that nearly all hospitals have X-ray imaging machines, it could be possible to use X-rays to test for COVID-19 without the dedicated test kits.

A drawback is that X-ray analysis requires a radiology expert and takes significant time — which is precious when people are sick around the world. Therefore developing an automated analysis system is required to save medical professionals valuable time.

Note: There are newer publications that suggest CT scans are better for diagnosing COVID-19, but all we have to work with for this report is an X-ray image dataset. Secondly, we are not medical experts and presume there are other, more reliable, methods that doctors and medical professionals will use to detect COVID-19 outside of the dedicated test kits.

EMPLOYED DATASET

The COVID-19 X-ray image dataset we'll be using for this tutorial was curated by Dr. Joseph Cohen, a postdoctoral fellow at the University of Montreal.

Dr. Cohen started collecting X-ray images of COVID-19 cases and publishing them in the following GitHub repository: ieee8023/covid-chestxray-dataset: We are building an open database of COVID-19 cases with chest X-ray or CT images. (github.com)

Inside the repo we found example of COVID-19 cases, as well as MERS, SARS, and ARDS.

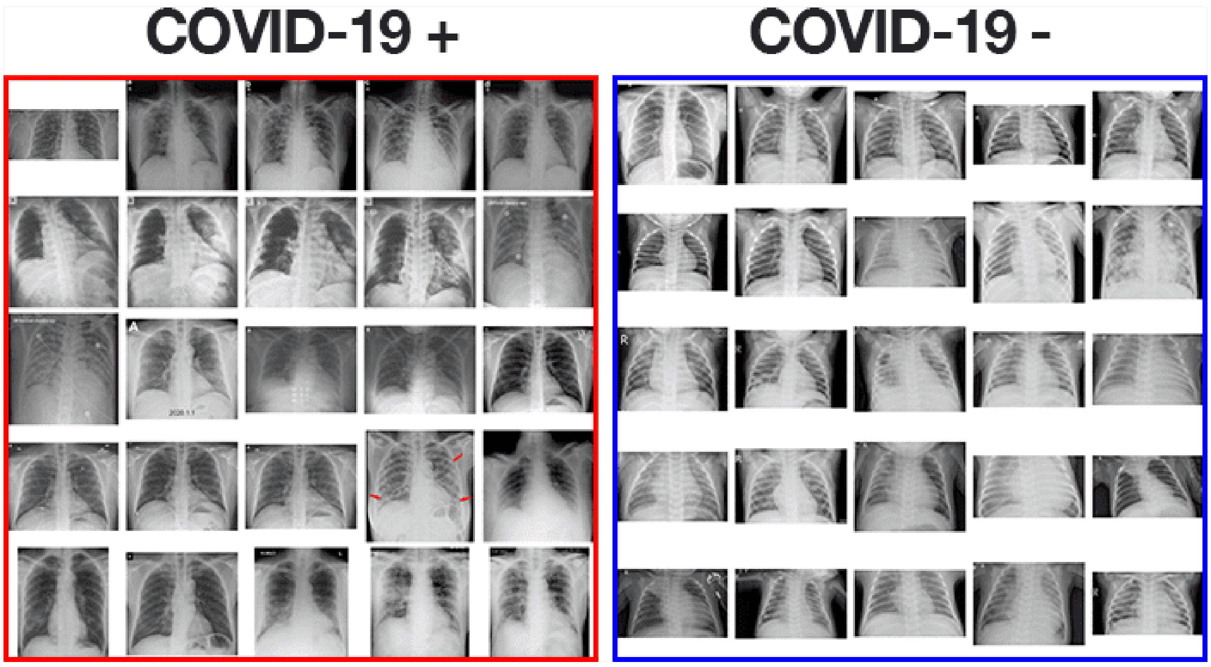


Fig1 3: Corona Virus (COVID-19) chest X-ray image data. On the left we have positive (i.e., infected) X-ray images, whereas on the right we have negative samples.

Collecting CoVID-19 X-Ray images:

Thanks to API provided in the GitHub repo, we were able to load the entire dataset in a python notebook. But the challenge was to sample the CoVID X-Rays from the other bacterial/pneumonia, MERS, SARS, and ARDS cases. For this we first separated the CoVID positive case indices using the dictionary data structure, and after that we separated Anterior-Posterior view X-ray indices from those positive cases. Since the normal X-Ray dataset was available in Anterior-Posterior view only.

After the indices were obtained it was easy to plot and save images into the google drive folder using python's matplotlib.pyplot library. The code for this data extraction part is also attached with the remaining work.

Collecting normal X-ray images:

To do so, I used Kaggle's Chest X-Ray Images (Pneumonia) dataset and sampled 25 X-ray images from healthy patients**.** There are a number of problems with Kaggle's Chest X-Ray dataset, namely noisy/incorrect labels, but it served as a good enough starting point for this proof of concept COVID-19 detector.

Organising Dataset:

The total number of images obtained from the CoVID positive X-Ray was 146. Another 146 normal X-Ray were taken from the Kaggle Dataset. We separated 16 images from each set into a test folder for a final test at the end, leaving with a total of 260 images which we uploaded into the drive and used that.

METHOD DETAILS

Since the dataset included images, we decided to use **Convolutional Neural Network model**, since it works very well with image data, with initial layers detecting essential features like edges, corners, background and the later layers dealing with more general information to classify the images.

Since the dataset was small, **data augmentation** was used to like rotating, zooming and scaling to increase the dataset as well as to deal with different invariances in the real-world dataset.

After that a basic CNN model with filters and max-pooling layers was implemented with relu activation function in the intermediate layers, followed by flattening into dense layers at the end and a sigmoid activation unit to finally classify the images into the two classes. Metrics for evaluation included training and validation losses, validation set accuracy, confusion matrix, precision, recall and F1-score.

To get better results we then tried several techniques on this basic CNN which included applying **dropouts** to avoid overfitting or high variance, **batch normalization** to handle internal covariance shift, **learning rate decay** to ensure smooth reaching of gradient descent. We were able to obtain much better results with these basic implementations.

Finally we tried using **transfer learning** which is basically applying a pre-trained model's parameters which are already well trained and optimized on a huge dataset to our model. By freezing the parameters, we will ensure that only the variables of the last classification layer get trained, while the variables from the other layers of the pre-trained model are kept the same.

EXPERIMENTS AND RESULTS

<u>**EXPERIMENT 1:**</u>

Train-Validation split: 3:1 or 194 images in train set and 66 images in validation set.

Pre-processing:

<u>Image Augmentation:</u> Artificially boosting the number of images in our training set by applying random image transformations to the existing images in the training set.

<u>Data normalization:</u> is an important step which ensures that each input parameter (pixel, in this case) has a similar data distribution. This makes convergence faster while training the network. For image inputs we need the pixel numbers to be positive, so we might choose to scale the normalized data in the range [0, 1] or [0, 255].

Rescaling and data augmentation was applied to train set as follows: Rotation range of 45, width shift and height shift of 0.15, zoom range of 0.5 and a horizontal flip. This was followed by shuffling the images in train set.

Only rescaling and no shuffling was done to images in validation set.

Model:

<u>Convolutions</u>: When working with RGB/grayscale images we convolve each colour channel (only one in grayscale) with its own convolutional filter. Convolutions on each colour channel are performed in the same way as with grayscale images, *i.e.* by performing element-wise multiplication of the convolutional filter (kernel) and a section of the input array. The result of each convolution is added up together with a bias value to get the convoluted output.

<u>Max Pooling:</u> When working with RGB images we perform max pooling on each colour channel using the same window size and stride. Max pooling on each colour channel is performed in the same way as with grayscale images, *i.e.* by selecting the max value in each window. Max pooling helps reduce image size after each CNN layer without increasing number of learnable parameters.

<u>ReLU Activation</u>: The sigmoid activation function is actually quite problematic in deep networks. It squashes all values between 0 and 1 and when you do so repeatedly, neuron outputs and their gradients can vanish entirely. It was mentioned for historical reasons, but modern networks use the RELU (Rectified Linear Unit) which looks like this:

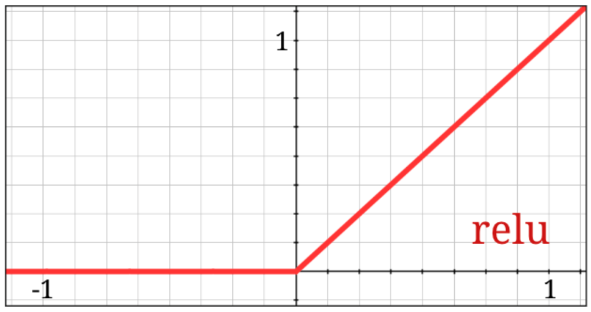


Fig 6.1.1: ReLU function

Why is the sigmoid problematic? Look how it behaves on the sides: it gets flat.

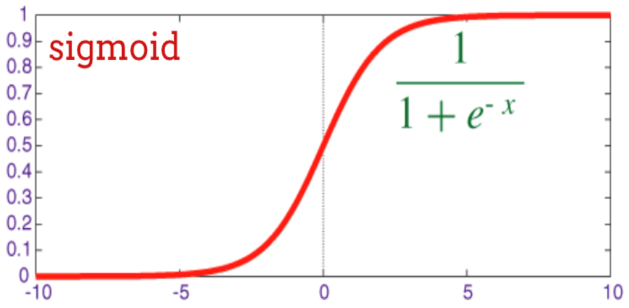


Fig 6.1.2: Sigmoid function

And that means its derivative there is close to zero. Remember how the training progresses, by following the gradient, which is a vector of derivatives. With sigmoid activation, especially if there are many layers, the gradient can become very small and training get slower and slower.

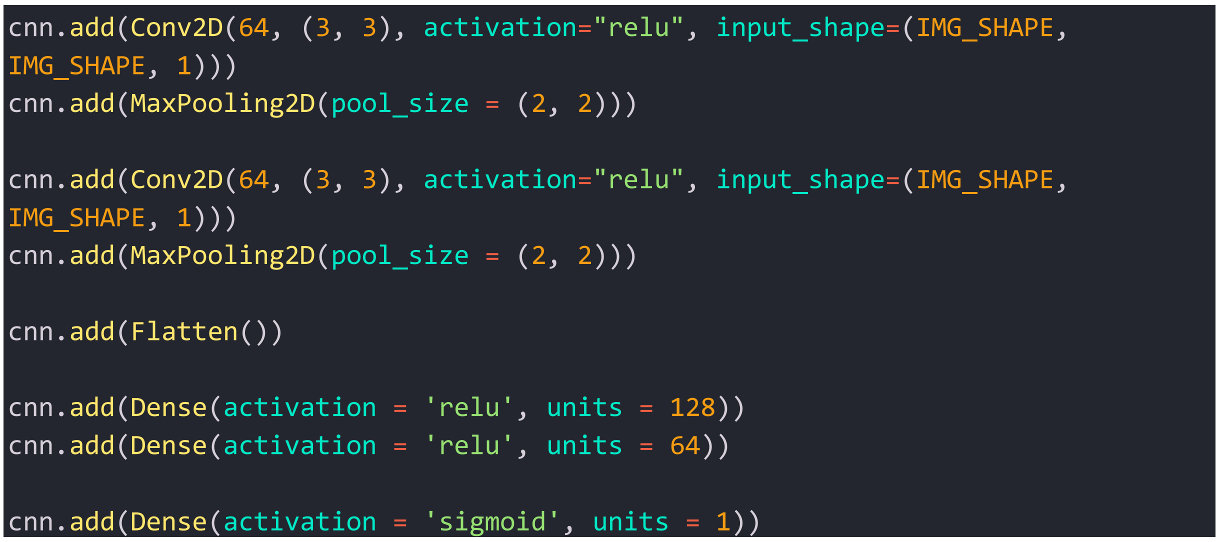
The relu on the other hand has a derivative of 1, at least on its right side. With RELU activation, even if the gradients coming from some neurons can be zero, there will always be others giving a clear non-zero gradient and training can continue at a good pace.

```
cnn = Sequential()

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
```



```
cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Flatten())

cnn.add(Dense(activation = 'relu', units = 128))
cnn.add(Dense(activation = 'relu', units = 64))

cnn.add(Dense(activation = 'sigmoid', units = 1))
```

Fig 6.1.3: Basic CNN Model with MAX POOLING and ReLU activation

Hyper parameters and functions involved:

Batch size: 20
Image Size: 180 X 180 X 1
Number of Epochs: 30
Optimizer function: Adam
Loss function: Binary Cross entropy

Results:

Validation accuracy of upto 98% was achieved after training 30 epochs. However test accuracy of only 81.25% was obtained on unseen test data.

Following are the different result metrics:

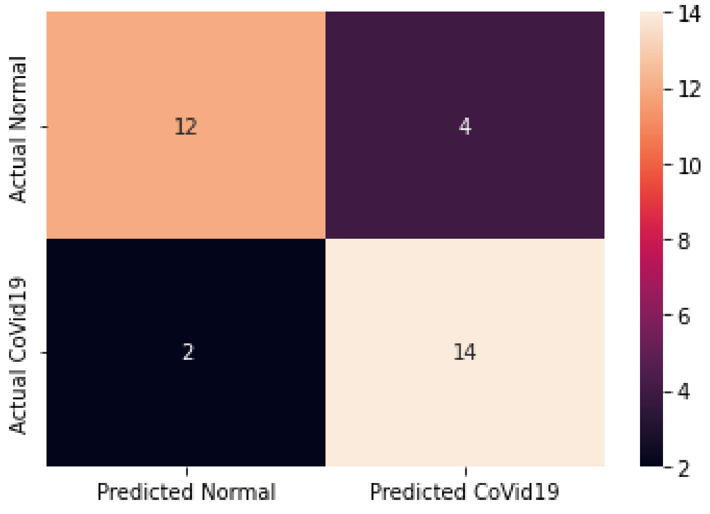


Fig 6.1.4: Confusion Matrix

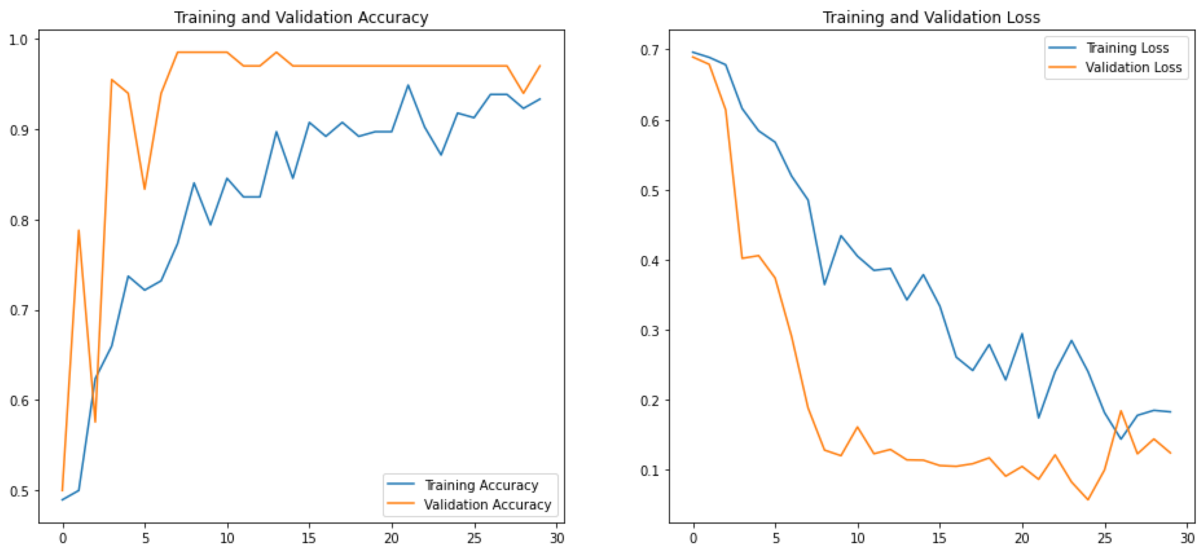


Fig 6.1.5: Plots for accuracy vs epochs and loss vs epochs

EXPERIMENT 2:

Train-Validation split: 3:1 or 194 images in train set and 66 images in validation set.

Pre-processing: Rescaling and data augmentation was applied to train set as follows: Rotation range of 45, zoom range of 0.5 and a horizontal flip. This was followed by shuffling the images in train set.

Only rescaling and no shuffling was done to images in validation set.

Model:

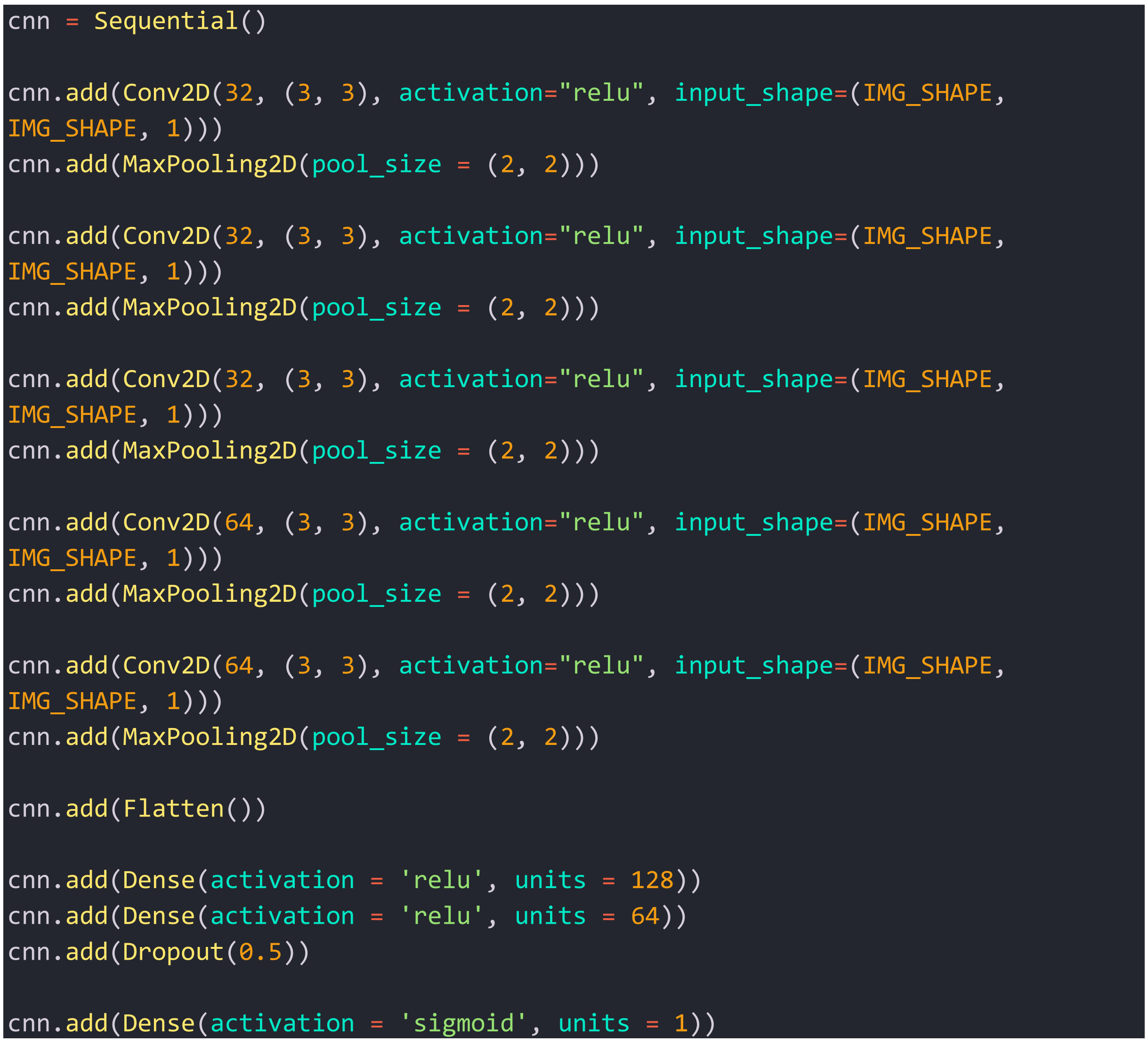
Overfitting and Dropouts: The learning algorithm works on training data only and optimises the training loss accordingly. It never sees validation data so it is not surprising that after a while its work no longer has an effect on the validation loss which stops dropping and sometimes even bounces back up.

This does not immediately affect the real-world recognition capabilities of your model, but it will prevent you from running many iterations and is generally a sign that the training is no longer having a positive effect. This disconnect is usually called "overfitting" and when you see it, you can try to apply a regularisation technique called "dropout". The dropout technique shoots random neurons at each training iteration.

Dropout is one of the oldest regularization techniques in deep learning. At each training iteration, it drops random neurons from the network with a probability p (typically 25% to 50%). In practice, neuron outputs are set to 0. The net result is that these neurons will not participate in the loss computation this time around and they will not get weight updates. Different neurons will be dropped at each training iteration.

When testing the performance of your network of course you put all the neurons back (dropout rate=0). Keras does this automatically, so all you have to do is add a **tf.keras.layers.Dropout** layer. It will have the correct behaviour at training and eval time automatically.

Why does it work? The theory is that neural networks have so much freedom between their numerous layers that it is entirely possible for a layer to evolve a bad behaviour and for the next layer to compensate for it. This is not an ideal use of neurons. With dropout, there is a high probability that the neuron "fixing" the problem is not there in a given training round. The bad behaviour of the offending layer become obvious and weights evolve towards a better behaviour.



```
cnn = Sequential()

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Flatten())

cnn.add(Dense(activation = 'relu', units = 128))
cnn.add(Dense(activation = 'relu', units = 64))
cnn.add(Dropout(0.5))

cnn.add(Dense(activation = 'sigmoid', units = 1))
```

Fig 6.2.1: Same CNN model as previous with added dropout rate of 0.5

Hyper parameters and functions involved:

Batch size: 15
Image Size: 224 X 224 X 1
Number of Epochs: 30

Optimizer function: Adam
Loss function: Binary Cross entropy

Results:

Validation accuracy of upto 98% was achieved after training 30 epochs. However test accuracy of about 96% was obtained on unseen test data.

Following are the different result metrics:

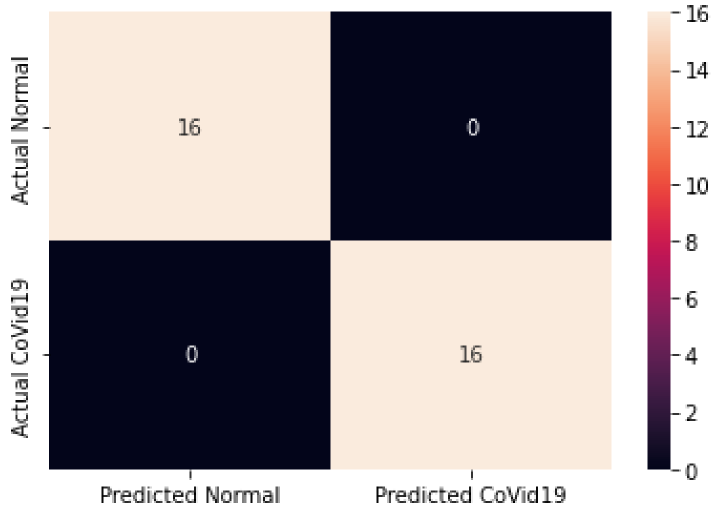


Fig 6.2.2: Confusion Matrix

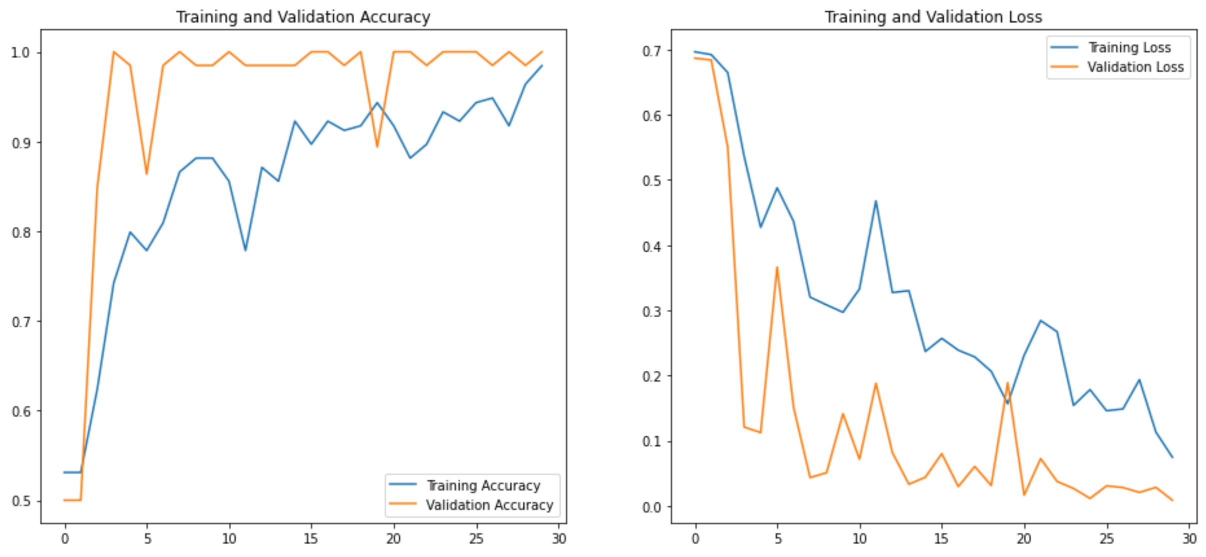


Fig 6.2.3: Plots for accuracy vs epochs and loss vs epochs

EXPERIMENT 3:

Train-Validation split: 3:1 or 194 images in train set and 66 images in validation set.

Pre-processing: Rescaling and data augmentation was applied to train set as follows: Rotation range of 45, zoom range of 0.5 and a horizontal flip. This was followed by shuffling the images in train set.

Only rescaling and no shuffling was done to images in validation set.

Model:

Learning Rate Decay: The training curves are really noisy and look at both validation curves: they are jumping up and down. This means that we are going too fast. We could go back to our previous speed, but there is a better way.

The good solution is to start fast and decay the learning rate exponentially. In Keras, we can do this with the tf.keras.callbacks.LearningRateScheduler callback.

Batch Normalization: In a nutshell, batch norm tries to address the problem of how neuron outputs are distributed relatively to the neuron's activation function. Across a mini-batch of training data, a neuron output, before activation, could have the following distributions:

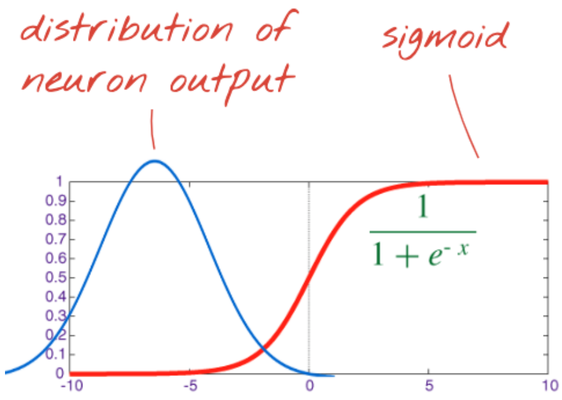


Fig 6.3.1: Too far to the left, after sigmoid activation, this neuron almost always outputs 0.

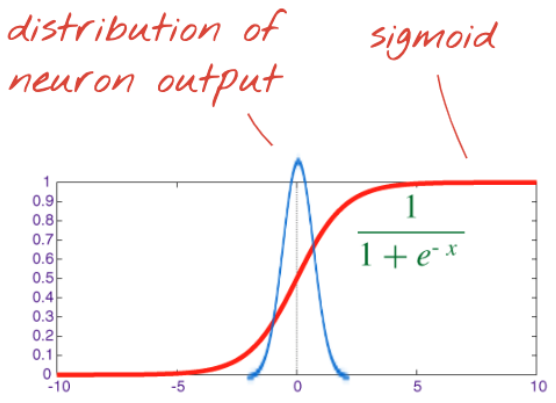


Fig 6.3.2: Too narrow, this neuron never outputs a clear 0 or 1.

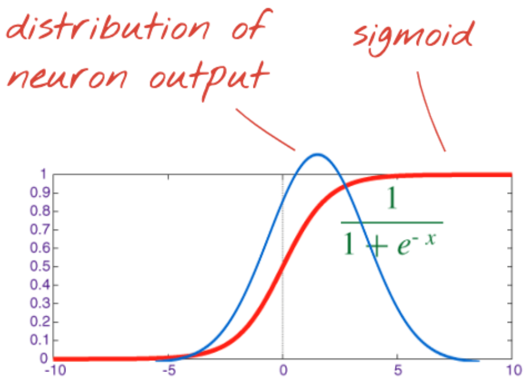


Fig 6.3.3: This is not too bad. The neuron will have a fair range of behaviours across this mini-batch.

To fix this, batch norm normalizes neuron outputs across a training batch of data, i.e it subtracts the average and divides by the standard deviation. However, doing just that would be too brutal. With a perfectly centred and normally wide distribution everywhere, all neurons would have the same behaviour. The trick is to introduce two additional learnable parameters per neuron, α and β, and to compute:

α . normalized_out + β

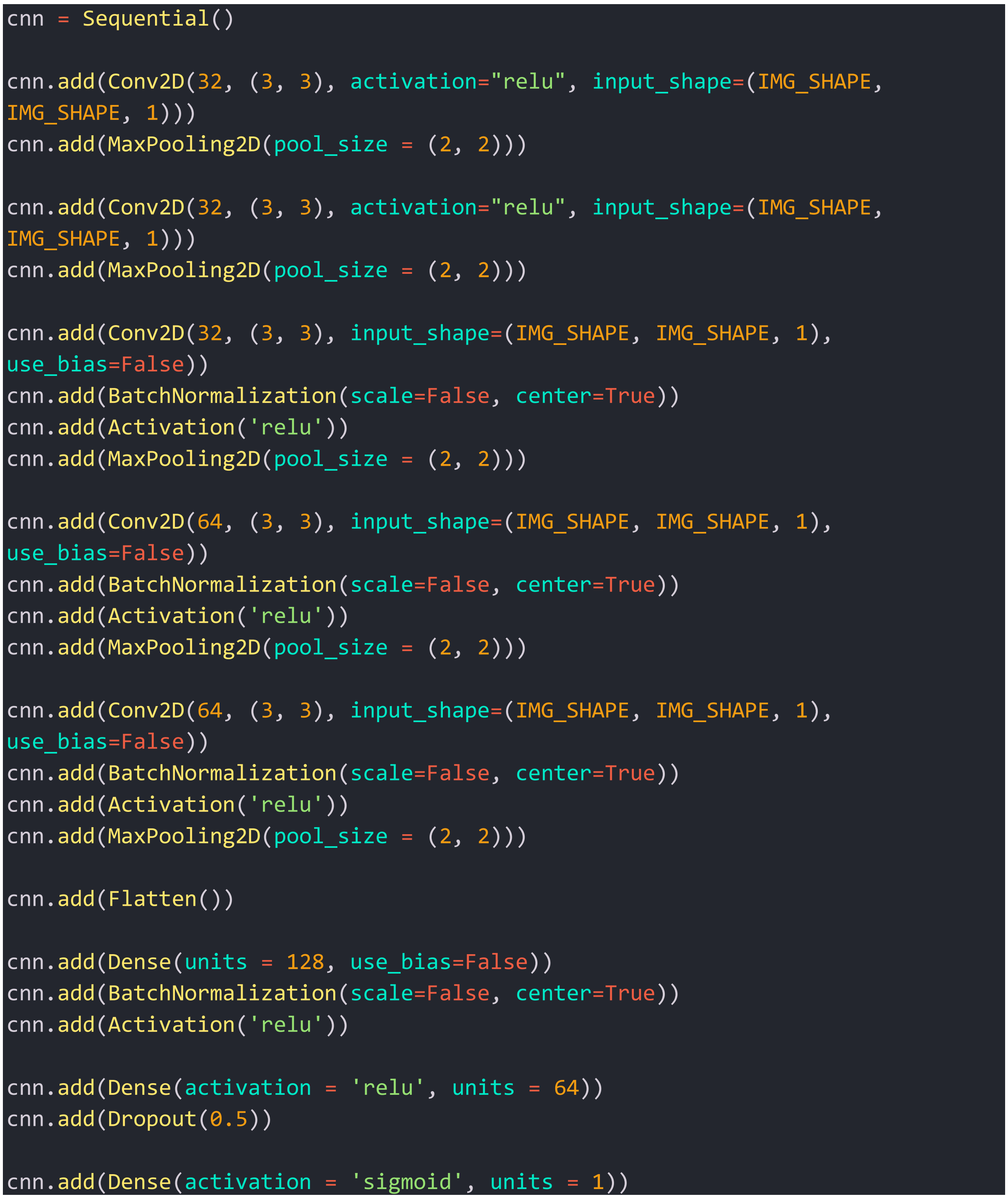
And then apply the activation function (sigmoid, relu, ...). This way, the network decides, through machine learning, how much centering and re-scaling to apply at each neuron.

The problem with batch norm is that at prediction time, you do not have training batches over which you could compute the statistics of your neuron outputs. But you still need those values. This part is a bit hacky. During training, a "typical" neuron output average and standard deviation is computed across a "sufficient" number of batches, in practice using a running exponential average. These stats are then used at inference time.

The good news is that in Keras you can use a `tf.keras.layers.BatchNormalization` layer and all this accounting will happen automatically.

Batch norm "by the book":

1. Batch normalization goes between the output of a layer and its activation function.
2. If you use center=True in batch norm, you do not need biases in your layer. The batch norm offset plays the role of a bias.
3. If you use an activation function that is scale-invariant (i.e. does not change shape if you zoom on it) then you can set scale=False. Relu is scale-invariant. Sigmoid is not.



```
cnn = Sequential()

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_SHAPE,
IMG_SHAPE, 1)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(32, (3, 3), input_shape=(IMG_SHAPE, IMG_SHAPE, 1),
use_bias=False))
cnn.add(BatchNormalization(scale=False, center=True))
cnn.add(Activation('relu'))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(64, (3, 3), input_shape=(IMG_SHAPE, IMG_SHAPE, 1),
use_bias=False))
cnn.add(BatchNormalization(scale=False, center=True))
cnn.add(Activation('relu'))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Conv2D(64, (3, 3), input_shape=(IMG_SHAPE, IMG_SHAPE, 1),
use_bias=False))
cnn.add(BatchNormalization(scale=False, center=True))
cnn.add(Activation('relu'))
cnn.add(MaxPooling2D(pool_size = (2, 2)))

cnn.add(Flatten())

cnn.add(Dense(units = 128, use_bias=False))
cnn.add(BatchNormalization(scale=False, center=True))
cnn.add(Activation('relu'))

cnn.add(Dense(activation = 'relu', units = 64))
cnn.add(Dropout(0.5))

cnn.add(Dense(activation = 'sigmoid', units = 1))
```

Fig 6.3.4: CNN model with Batch Normalization and Dropouts

Hyper parameters and functions involved:

Batch size: 15
Image Size: 224 X 224 X 1
Number of Epochs: 25
Optimizer function: Adam with initial learning rate decay of 0.666 and decaying as: 0.01 X (0.6^epoch number)
Loss function: Binary Cross entropy

Results:

Validation accuracy of upto 94% was achieved after training 25 epochs. However test accuracy of about 96% was obtained on unseen test data.

Following are the different result metrics:

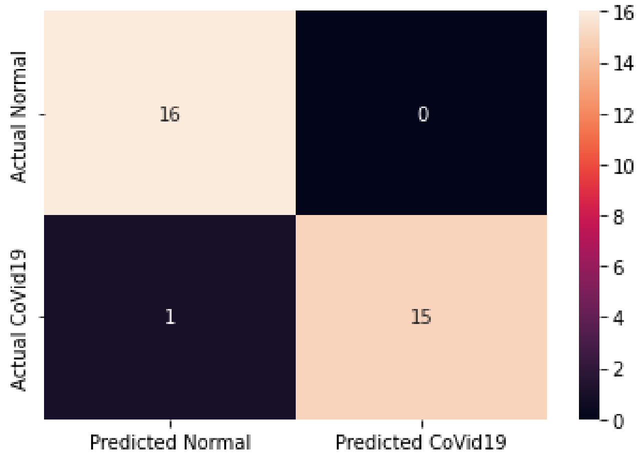


Fig 6.3.5: Confusion Matrix

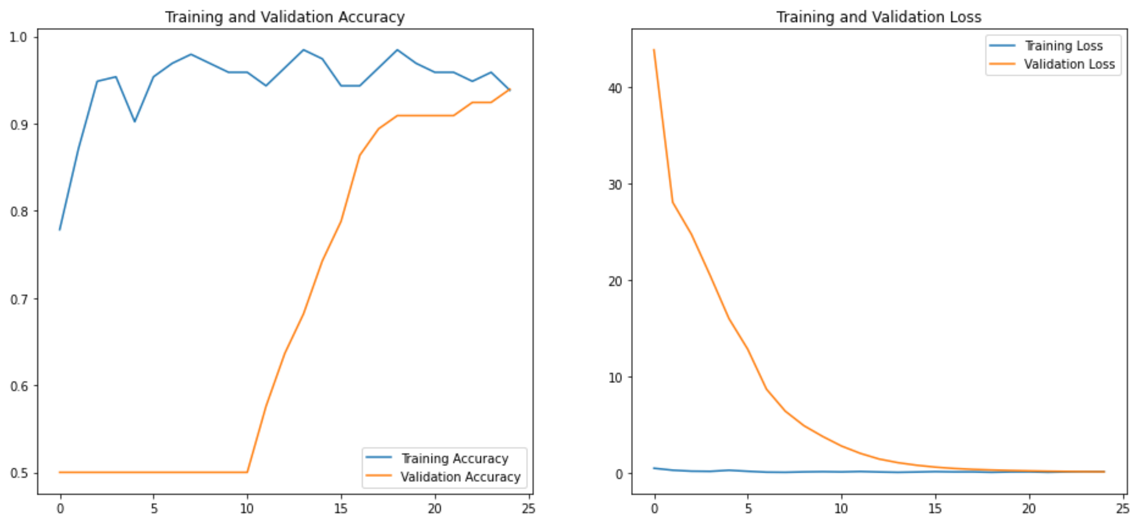


Fig 6.3.6: Plots for accuracy vs epochs and loss vs epochs

EXPERIMENT 4:

Train-Validation split: 3:1 or 194 images in train set and 66 images in validation set.

Pre-processing: Rescaling and data augmentation was applied to train set as follows: Rotation range of 45, zoom range of 0.5 and a horizontal flip. This was followed by shuffling the images in train set.

Only rescaling and no shuffling was done to images in validation set.

Model:

Transfer Learning: A technique that reuses a model that was created by machine learning experts and that has already been trained on a large dataset. When performing transfer learning we must always change the last layer of the pre-trained model so that it has the same number of classes that we have in the dataset we are working with.

VGG 16: Given image it finds the object name in the image. It can detect any one of 1000 images. It takes input image of size 224 * 224 * 3 (RGB image)
Built using:
 Convolutions layers (used only 3*3 size)
 Max pooling layers (used only 2*2 size)
 Fully connected layers at end
 Total 16 layers

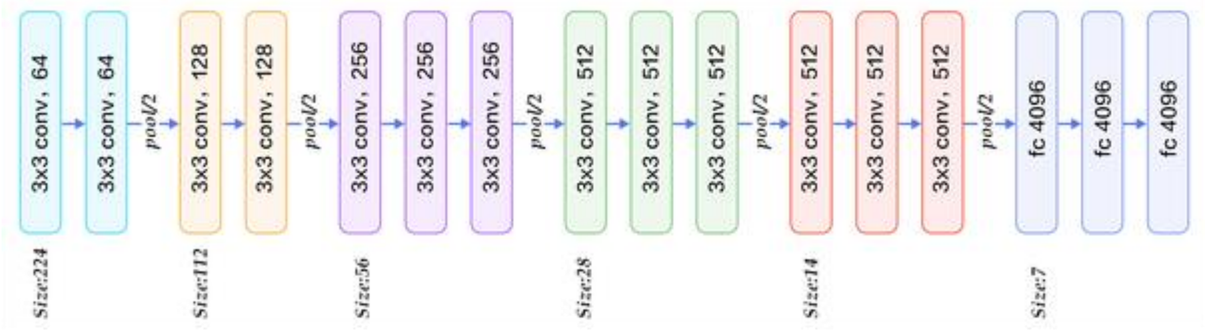


Fig 6.4.1: VGG 16 Network

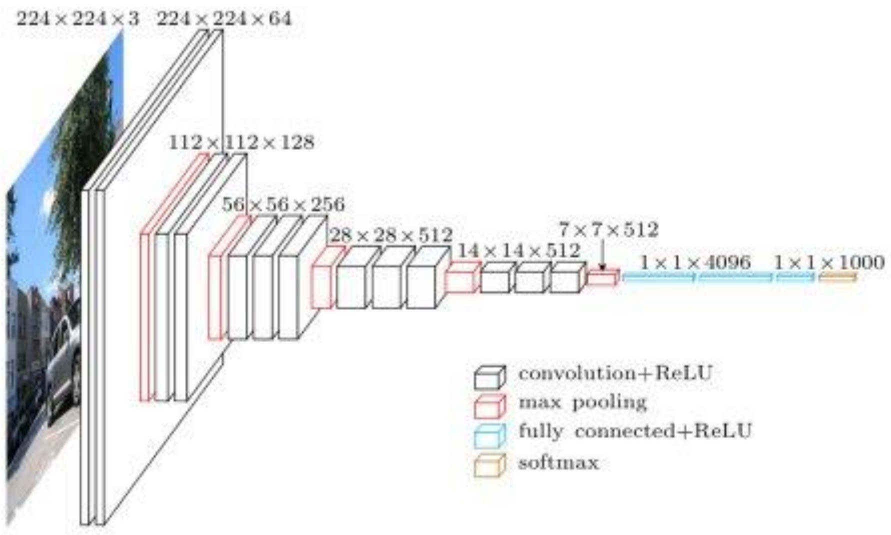
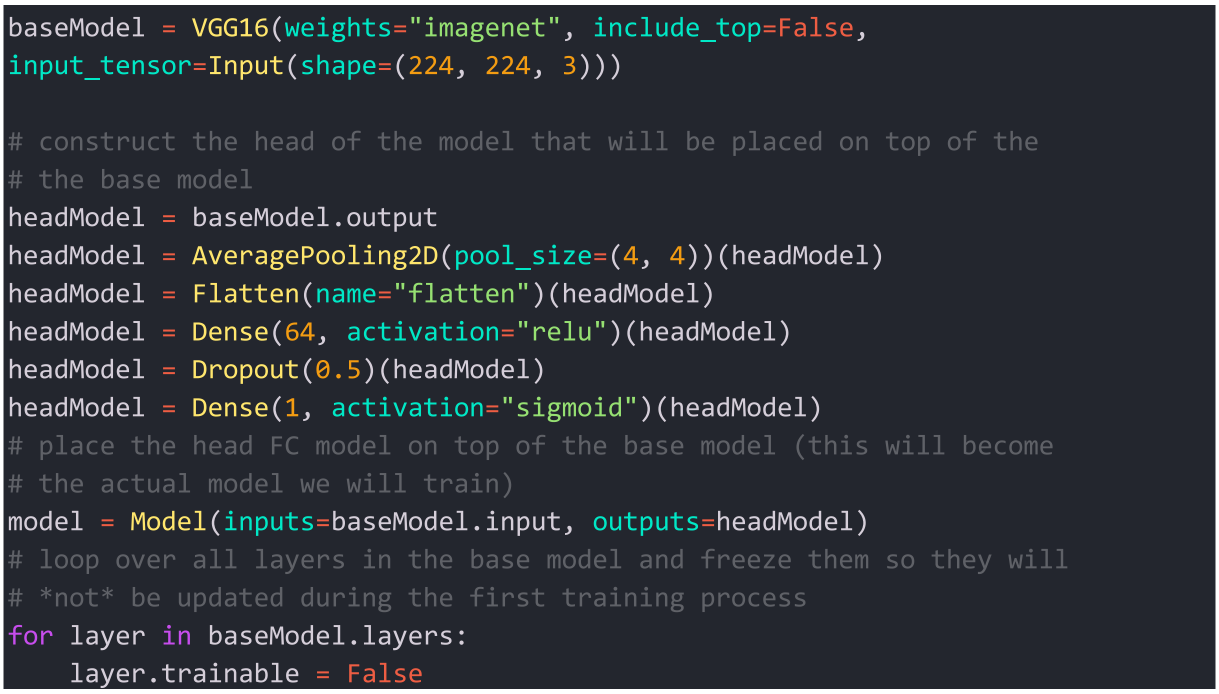


Fig 6.4.2: Full view at image level



```
baseModel = VGG16(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(64, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(1, activation="sigmoid")(headModel)
# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)
# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
    layer.trainable = False
```

Fig 6.4.3: CNN model with transfer learning implementation

Hyper parameters and functions involved:

Batch size: 15
Image Size: 224 X 224 X 1
Number of Epochs: 25
Optimizer function: Adam
Loss function: Binary Cross entropy

Results:

Validation accuracy of upto 97% was achieved after training 25 epochs. However test accuracy of about 91% was obtained on unseen test data.

Following are the different result metrics:

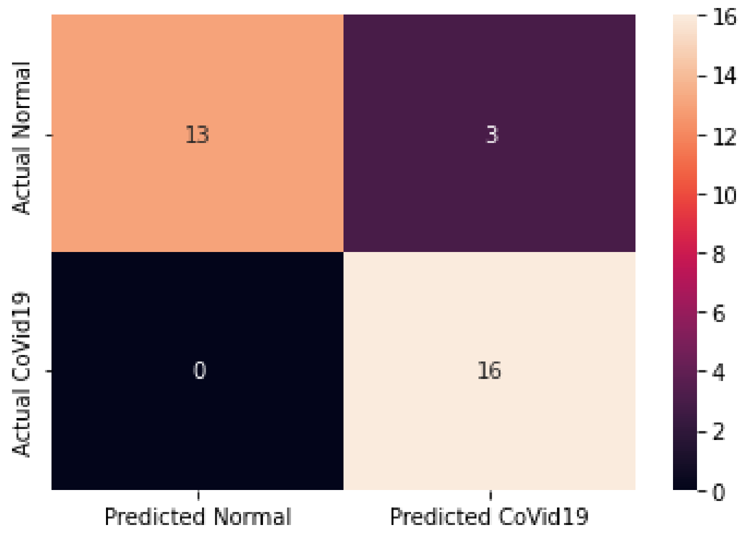


Fig 6.4.4: Confusion Matrix

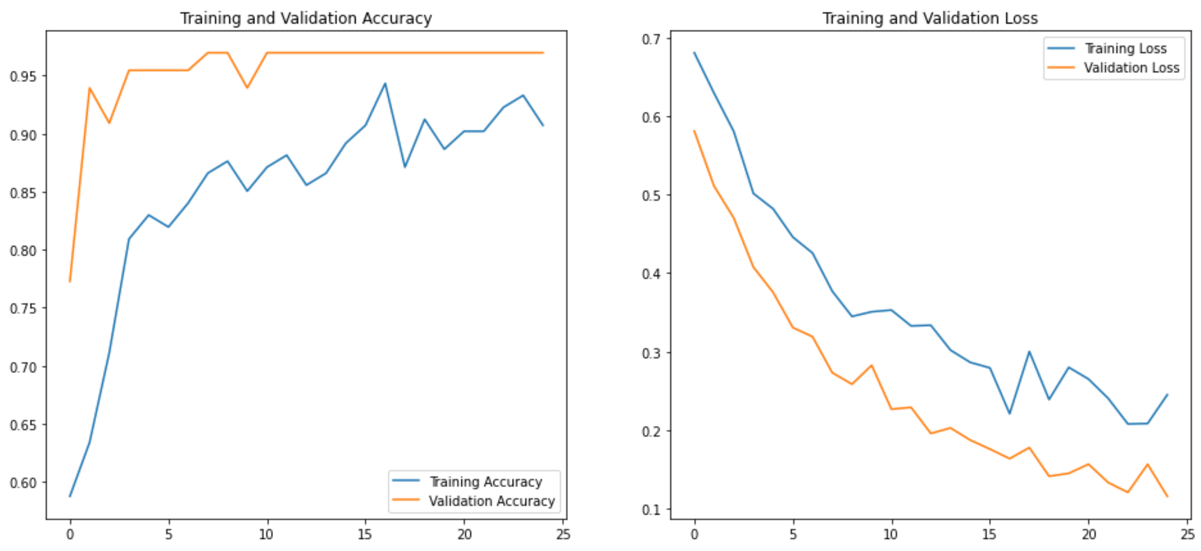


Fig 6.4.5: Plots for accuracy vs epochs and loss vs epochs

CONCLUSION

As seen from the above experiments we can conclude that our model was able to achieve 90-95% accuracy most of the time.

Apart from the first basic model we were able to identify those who had COVID most of the time as seen from the confusion matrix. Though there were times when those who did not had COVID were falsely identified as positive, but if all people (including actually positive or not) take precautionary quarantine, then also it is not bad.

Another positive result was, we did not encounter the problem of overfitting, despite limited dataset, thanks to data augmentation and other techniques like dropout which handled overfitting if any.

Being able to accurately detect COVID-19 with 100% accuracy is great; however**,** we don't want to classify someone as "COVID-19 negative" when they are "COVID-19 positive".

In fact, the last thing we want to do is tell a patient they are COVID-19 negative, and then have them go home and infect their family and friends; thereby transmitting the disease further.

Balancing sensitivity and specificity is incredibly challenging when it comes to medical applications, especially infectious diseases that can be rapidly transmitted, such as COVID-19.

When it comes to medical computer vision and deep learning, we must always be mindful of the fact that our predictive models can have very real consequences — a missed diagnosis can cost lives**.**

Again, these results are gathered for educational purposes only.

LIMITATIONS:

One of the major problem which we faced was the limited dataset. Thanks to such educational and research repositories that we were able to gather some images.

We imagine in the next 12-18 months, we'll have more high-quality COVID-19 image datasets, but for the time being, we can only make do with what we have.

For the COVID-19 detector to be deployed in the field, it would have to go through rigorous testing by trained medical professionals, working hand-in-hand with expert deep learning practitioners. The method covered here today is certainly not such a method, and is meant for educational purposes only.

Furthermore, we need to be concerned with what the model is actually “learning”.

It is possible that our model is learning patterns that are not relevant to COVID-19, and instead are just variations between the two data splits (i.e., positive versus negative COVID-19 diagnosis).

It would take a trained medical professional and rigorous testing to validate the results coming out of our COVID-19 detector.

Right now we are using only image data (i.e., X-rays) — better automatic COVID-19 detectors should leverage multiple data sources not limited to just images, including patient vitals, population density, geographical location, etc. Image data by itself is typically not sufficient for these types of applications.

REFERENCES

- GoogleCloudPlatform/tensorflow-without-a-phd: A crash course in six episodes for software developers who want to become machine learning practitioners. (github.com)
- Detecting COVID-19 in X-ray images with Keras, TensorFlow, and Deep Learning - PyImageSearch
- ieee8023/covid-chestxray-dataset: We are building an open database of COVID-19 cases with chest X-ray or CT images. (github.com)
- Intro to TensorFlow for Deep Learning - Udacity
- Chest X-Ray Images (Pneumonia) | Kaggle
- A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science
- Memorizing is not learning! — 6 tricks to prevent overfitting in machine learning. | Hacker Noon
- Understanding your Convolution network with Visualizations | by Ankit Paliwal | Towards Data Science