Detecting COVID-19 from Medical X-Ray Images

Rayan Wali, Cornell University

Contents

Detecting COVID-19 from Medical X-Ray Images

Rayan Wali (rsw244@cornell.edu)

December 2021

1 Introduction

In medical imaging, radiologists have to take scans of patients and analyze them to produce an output. With the advent of AI and deep learning, this process can be automated, saving manual workload, improving efficiency and results in precise decisions, and enabling speedy delivery of results to patients.

Medical imaging is most commonly studied in detecting pneumonia from patient X-rays. COVID-19, like Pneumonia, is studied to have a strong correlation with a patients' chest X-ray images. Because of this, we are able to develop a model that we know that will achieve a reasonably high accuracy when given a new chest X-ray image. This system will make radiologists' job easier in classifying patients' X-ray images, and will likely make the process faster.

My project is a machine learning classification task with the help of a **deep learning model** that takes in as input an X-ray image of a patient's chest and predicts whether the patient is positive or negative for COVID-19. Since there are two output classes of prediction, this is a binary classification problem.

For this project, I used Python as my primary programming language, and I used the GPU built in to Google Colab in order to speed up the training and testing processes. The following are the key libraries I used for my project:

- **Deep Learning:** Keras, TensorFlow
- **Scientific Processing:** Scikit-Learn

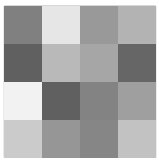
To construct the (convolutional) deep learning model architecture, I applied *transfer learning*, that is, importing an existing, pre-defined network architecture. I chose the most appropriate model, taking into consideration the number of layers of the network, and toggled hyperparameter values to strive towards optimality.

To display the classification results, I designed a web user interface with Flask (a built-in Python web development library), JavaScript, and HTML, that allows users (radiologists) to input an image of any dimension. The image gets sent over to the trained model, and the final classification result is be returned. The final output to the user (radiologist) will be the output of the trained model.

2 Training and Testing Images

The images that I used to train and test the model were grayscale X-ray images, i.e., there is a single channel, and each pixel contains a property representing its intensity between 0 (black) and 255 (white). The following is a sample chest X-ray image that will be part of my training dataset:

(a) Sample X-Ray Image



(b) Sample Grayscale Image

20	120	50	80
10	85	76	15
225	10	22	78
100	48	24	90

(c) Pixel Matrix for **(b)**

The image is represented as a matrix of pixel intensities, with each element of the matrix ranging from 0 to 255.

3 Preprocessing Raw Data

The first step I performed was preprocessing the raw images. The images that are inputted by radiologists could be of any dimension. However, since my model strictly accepts 256×256 dimensional images, the image needs to be reshaped to a 256×256 dimensional image that is able to be taken in by my model.

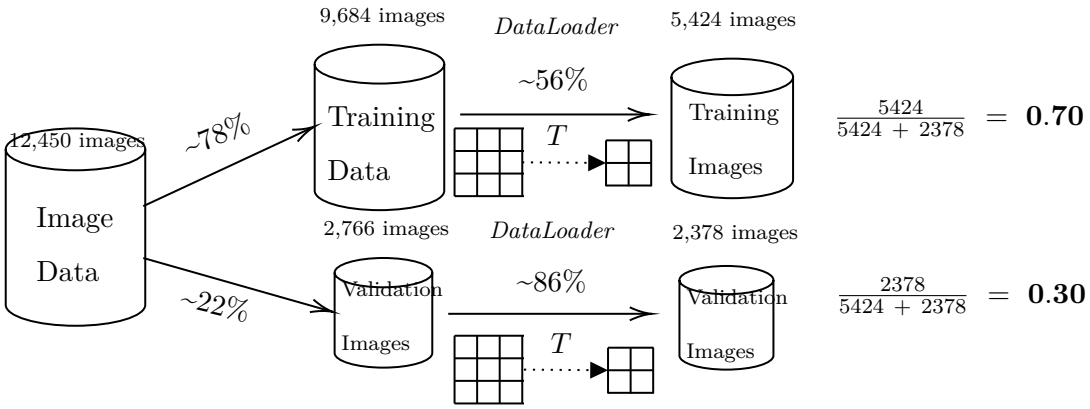
I also performed data augmentation where I performed reshaping, re-scaling, zooming, shifting, shearing, rotation, and flipping to each image of the training and validation datasets. I used Keras' `DataLoader` functionality for this, creating two `DataLoaders`, one exclusively for the training data and one exclusively for the validation data.

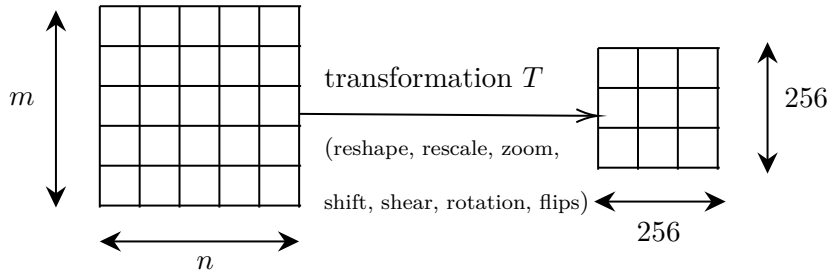
```
datagen_train = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    validation_split=0.44,
    fill_mode='nearest'
)
```

```
datagen_val = ImageDataGenerator(
    rescale=1./255,
```

```
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    validation_split=0.86,
    fill_mode='nearest'
)
```

The purpose of this step was to reduce overfitting and make the model more generalizable and recognizable to distortions in images.

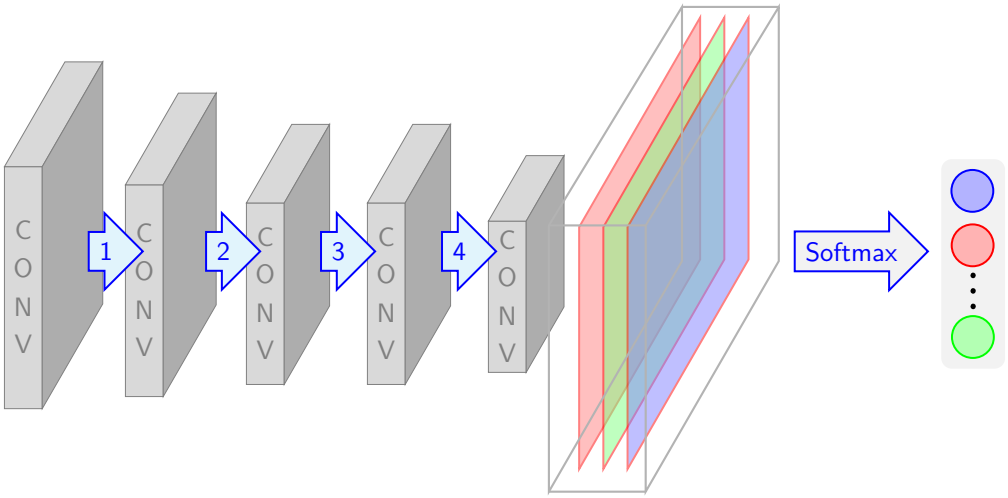




4 Deep Learning Architectures

Deep learning architectures are based upon neural networks. In our case of image classification, the features that are independently inputted into the network are the individual pixel intensities.

The deep learning architecture I used was a convolutional neural network, which tends to have a combination of convolutional and pooling layers.



4.1 Convolutions

Convolutional layers rely on individual convolution operations. The VGG-16 architecture uses same convolution and a 3×3 filter with a stride of 1. A *convolutional operator* '*' applied to a 2-dimensional image $f(x, y)$ and a filter ω performs the following operation:

$$\omega * f(x, y) = \sum_{i=-\alpha}^{\alpha} \sum_{j=-\beta}^{\beta} \omega(i, j) \cdot f(x + i, y + j) \tag{1}$$

The following is an example of the specific convolution the VGG-16 model uses, but with a lower resolution image. If a pixel in the output image is computed to have a negative intensity, we clamp it, i.e., we set it to 0.

$f(x, y)$

10	80	105	218
102	40	120	225
68	210	210	8
202	42	210	20

$*$

ω

1	-1	1
0	0	0
-1	1	-1

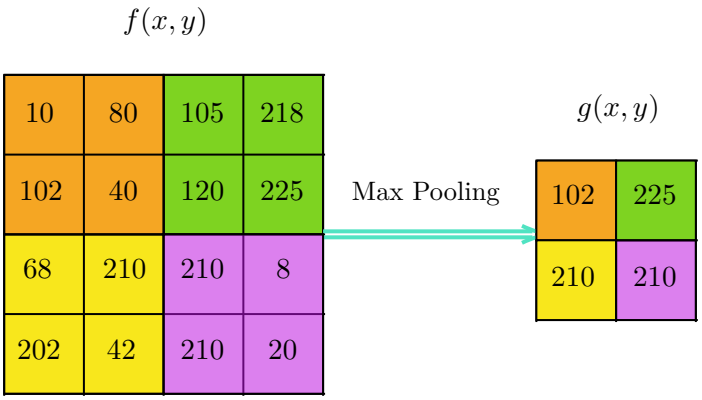
$=$

$g(x, y)$

62	0	0	105
0	0	185	0
0	0	293	0
142	0	0	0

4.2 Max Pooling

The max pooling layers of a VGG-16 architecture compute the maximum intensity for each patch and replace that patch with the computed maximum intensity. This reduces the dimensionality of the original image, creating a down-sampled feature map that smooths the features. The following figure illustrates the max-pooling process on a 4×4 image:



5 Constructing My Deep Learning Architecture

After preprocessing the raw input images, I applied *transfer learning*, i.e., imported an existing deep learning model architecture. Specifically, I imported the VGG-16 architecture, which is a convolutional neural network (CNN) architecture developed at the University of Oxford. The VGG-16 model consists of 16 layers — a mix of 2-dimensional convolutional layers and 2-dimensional max pooling layers — and approximately 138 million parameters.

The purpose of applying convolutional layers to an image is to extract information about the features of the image. For example, the max pooling layer selects the maximum pixel intensity from each patch of an image. This serves to down-sample the input image by reducing the dimensionality of the image. Similarly, convolutional layers serve the same purpose — they apply filters, also known as kernels, to the input image which blur the image.

The VGG-16 architecture is proven to achieve a high accuracy on classification tasks on the ImageNet database. The VGG-19 architecture, which is similar to VGG-16 model, except with 3 more layers, is studied to perform a higher accuracy than VGG-16 on medical imaging. However, we did not choose the VGG-19 due to memory constraints. The following table supports this by comparing the high performing architectures applied to medical imaging tasks:

Model	**Normal vs pneumonia**			**Bacteria vs virus**		
	Accuracy	**Specificity**	**Recall**	**Accuracy**	**Specificity**	**Recall**
Baseline	0.776	0.809	0.776	0.643	0.64	0.585
VGG16 [34][a]	0.923	0.926	0.923	0.923	0.909	0.85
VGG16 [38][b]	0.938	0.944	0.938	0.915	0.917	0.879
Inception V3	0.869	0.854	0.869	0.851	0.86	0.779
CapsNet	0.824	0.846	0.824	0.862	0.875	0.785
Stateof-art [3][c]	0.928	0.901	0.932	0.907	0.909	0.886

Table 1: Comparison of Deep Learning Architectures for Medical Imaging Use

The following code in Python demonstrates the Python code I wrote to import the VGG-16 model:

```
model_vgg16 = VGG16(include_top=False)

model_vgg16.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, None, None, 3)]   0

 block1_conv1 (Conv2D)       (None, None, None, 64)    1792

 block1_conv2 (Conv2D)       (None, None, None, 64)    36928

 block1_pool (MaxPooling2D)  (None, None, None, 64)    0

 block2_conv1 (Conv2D)       (None, None, None, 128)   73856

 block2_conv2 (Conv2D)       (None, None, None, 128)   147584

 block2_pool (MaxPooling2D)  (None, None, None, 128)   0

 block3_conv1 (Conv2D)       (None, None, None, 256)   295168

 block3_conv2 (Conv2D)       (None, None, None, 256)   590080

 block3_conv3 (Conv2D)       (None, None, None, 256)   590080

 block3_pool (MaxPooling2D)  (None, None, None, 256)   0

 block4_conv1 (Conv2D)       (None, None, None, 512)   1180160

 block4_conv2 (Conv2D)       (None, None, None, 512)   2359808

 block4_conv3 (Conv2D)       (None, None, None, 512)   2359808

 block4_pool (MaxPooling2D)  (None, None, None, 512)   0

 block5_conv1 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv2 (Conv2D)       (None, None, None, 512)   2359808

 block5_conv3 (Conv2D)       (None, None, None, 512)   2359808

 block5_pool (MaxPooling2D)  (None, None, None, 512)   0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Since the output probability score of the VGG-16 model does not reflect the input dimension of our images and the binary classification task we have, we will add an input layer before the VGG-16 layers and four more layers after them, namely, one `Flatten` layer and three `Dense` layers.

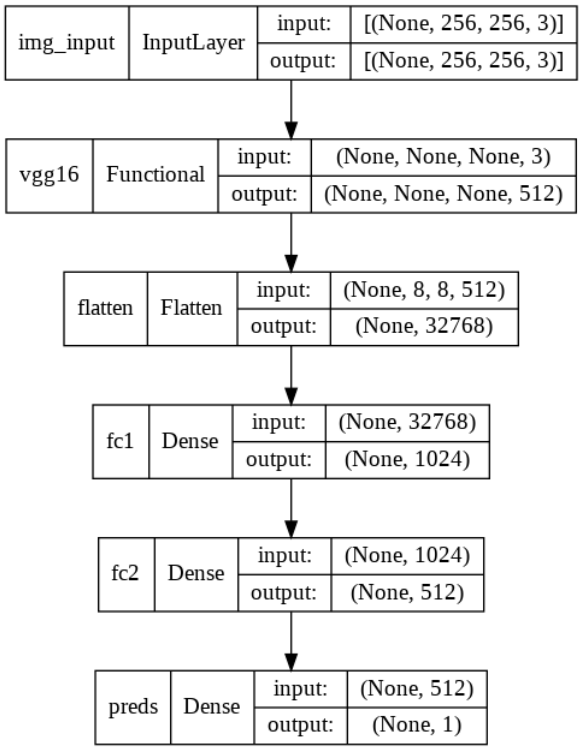
```
input = Input(shape=(256,256,3), name='img_input')

output_vgg16 = model_vgg16(input)

x = Flatten(name='flatten')(output_vgg16)

x = Dense(1024, activation='relu', name='fc1')(x)

x = Dense(512, activation='relu', name='fc2')(x)

x = Dense(1, activation='sigmoid', name='preds')(x)
```

After the addition of these layers to the imported VGG-16 architecture, we print out the model summary:

```
my_model = Model(inputs=input, outputs=x)

my_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 img_input (InputLayer)      [(None, 256, 256, 3)]     0

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

 flatten (Flatten)           (None, 32768)             0

 fc1 (Dense)                 (None, 1024)              33555456

 fc2 (Dense)                 (None, 512)               524800

 preds (Dense)               (None, 1)                 513

=================================================================
Total params: 48,795,457
Trainable params: 48,795,457
Non-trainable params: 0
_________________________________________________________________
```

The following diagram is a simplified flow diagram of the architecture above:



6 Training the Model

Once the model architecture was constructed, I trained the complete model with the 5,424 preprocessed training images (70% of the total training + validation images before preprocessing), which were a composition of COVID-positive and COVID-negative images.

In order to train the model, I used the Keras library. I used the **binary cross-entropy loss function** and performed hyperparameter search and trained with a learning rate $\alpha = \mathbf{0.01}$ for **20 epochs** using the `SGD optimizer`.

After training my model on a range of epochs, I was able to come to the conclusion that 20 epochs achieved the highest accuracy as I saw signs of a monotonically increasing accuracy score and a monotonically decreasing binary cross-entropy loss score. To tune the learning rate parameter, I conducted hyperparameter search, and found that $\alpha = 0.01$ was the right balance between training time and unstable training with low accuracy. From my experiments, $\alpha = 0.001$ took quite long to train, which was not feasible to use since there were multiple experiments I had to run with that fixed learning rate. And, a too high learning rate $\alpha = 0.1$ led to an unexpected decrease in accuracy.

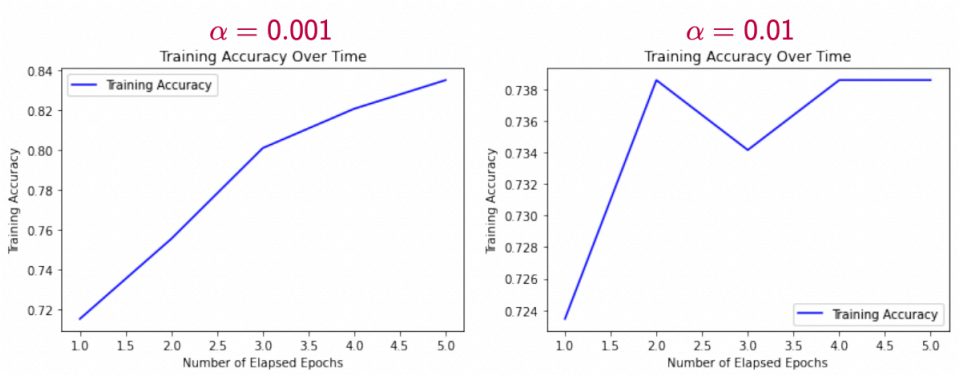


Figure 2: $\alpha = 0.001$ (left), $\alpha = 0.01$ (right)

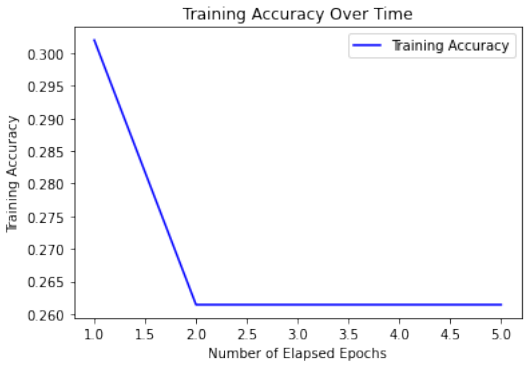
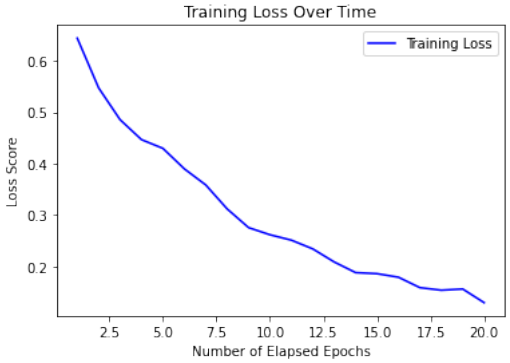


Figure 3: $\alpha = 0.1$ (when the learning rate is set too high, the training makes huge jumps; in this case, unexpected behavior occurs.)

My decision for the SGD optimizer came from an experiment I ran comparing the SGD optimizer and the Adam optimizer using the same model architecture and 5 epochs.

The following line plot shows the decrease in training loss over time as the number of epochs that the model is trained with increases with the hyperparameters of my final model:



Below, we will decompose the phases of the overall training process described above into the feedforward stage and the backpropogation phases.

6.1 The Feedforward Stage

The *feedforward* stage of training a neural network is the propagation of an input through all the layers of the network. The feedforward stage of a test image will output softmax probability scores for each class, which by property of the softmax function, all sum up to 1.

$$Sigmoid(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{2}$$

The process of obtaining these class probability scores for a classification task is known as *soft classification*. However, our goal is to output a single output for an test input image. I achieved this by placing a threshold on the soft classification probability scores, and selecting the class that equals or exceeds the threshold.

6.2 The Backpropagation Phase

The purpose of the backpropagation phase is to update the parameters (weights) of the model. The most common backpropogation method is *gradient descent*, which can be categorized into two types: stochastic and batch. A stochastic gradient descent optimizer updates the parameters of the model after the feedforward stage of each training instance, and the batch gradient descent optimizer updates the parameters of the model after every epoch has been completed.

7 A Deeper Dive into Deep Learning

As discussed above, deep learning is the training of deep neural networks. Training deep learning models consists of two stages: (1) the feed-forward stage and (2) the back-propagation stage. The feed-forward stage simulates a forward pass through the network by taking as input the features of a particular training example and outputting a value or a set of values representing the prediction or predictions for that example.

7.1 Activation Functions

An activation function, denoted by $\sigma(\cdot)$, is a function applied to a value that returns another value; it is essentially a transformation of the inputted value. Some common activation functions are ReLU and sigmoid, which are defined as $\text{ReLU}(x) = max(0, x)$ and $\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$ respectively.

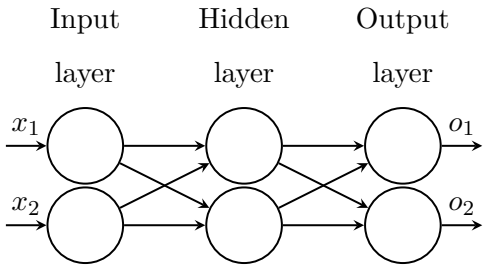


Figure 4: Fully-Connected Neural Network with 1 hidden layer

7.2 Weights of the Network

Each edge connecting two nodes in the network has a weight, defined by $w_{a,b}$, where a is the incoming edge node and b is the outgoing edge node. For example, the weight connecting the top node of the input layer to the bottom node of the hidden layer is represented by $w_{1,2}$.

7.3 The Feedforward Phase

In Figure 4, we call the output of the node of the hidden layer on the top h_1, and the output of the node of the hidden layer on the bottom h_2. Then, the following equations must satisfy for activation function σ:

$$h_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2) \tag{3}$$

$$h_2 = \sigma(w_{1,2}x_1 + w_{2,2}x_2) \tag{4}$$

7.4 The Backpropagation Phase

The backward pass through the neural network updates the weights and biases of the network based on the cost function derived from the forward pass through the network. This can be performed by a method known as *gradient descent*, which moves the weights and biases in a direction to optimize (minimize) the cost function. Gradient descent is performed by using the following equation:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha * \nabla J(\theta) \tag{5}$$

$$b^{(t+1)} \leftarrow b^{(t)} - \alpha * \nabla J(\theta) \tag{6}$$

In the equations above, $J(\theta)$ represents the cost function and α is a hyperparameter that represents the learning rate. For m examples, $J(\theta)$ is defined as follows:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(h_\theta(x_i), y_i\right) \tag{7}$$

In each timestep, the equation above is used to update the parameters of the network, until the weights and biases minimize the cost function. In the end, gradient descent satisfies the following optimization problem:

$$w^*,\, b^* \;=\; \arg\min_{w,b} \; J(\theta) \tag{8}$$

7.5 Forms of Gradient Descent

There are two main types of gradient descent: (1) batch and (2) stochastic. The difference between the two is that the batch form updates the parameters of the network after one epoch (the forward pass of all the examples), whereas the stochastic form updates the parameters of the network after the forward pass of each training example.

8 Evaluating the Model

I evaluated my model by taking my trained model architecture and applying it to the validation image dataset consisting of 2,378 labeled and preprocessed images (30% of the total training + validation images before preprocessing). The evaluation metrics I used for this process were the accuracy score, the binary cross-entropy loss score, and the F1 score, which is defined as the combination of the precision and recall statistics. The accuracy metric simply computes the proportion of correct instances in the entire dataset.

8.1 The Accuracy Metric

The accuracy metric is computed as follows for a model f, a set of true labels $\mathbf{y}$, and m instances in the training dataset:

$$\text{Accuracy(f, } \mathbf{y}) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}[f(i) = y_i]. \tag{9}$$

The accuracy score on the training dataset was 0.953 and the accuracy score on the held-out validation dataset was 0.956. Hence, there does not appear to be underfitting or overfitting, because of the low margin between accuracy scores on the dation datasets.

No. Epochs	Avg. Training Accuracy	Avg. Validation Accuracy
5	82%	81%
10	88%	86%
20	94%	95%

Table 2: Training and Validation Accuracies for Different Epochs

8.2 The F1 Score Metric

The F1 Score metric can be easily computed by performing an aggregate operation on the confusion metric. Because of this, I created the confusion matrix for the model applied to the validation dataset.

```
hard_preds_val = [1 if lbl > 0.50 else 0 for lbl in preds_val]
confusion_matrix(val_generator.classes, hard_preds_val)
```

Output Confusion Matrix:

```
array([[542, 79], [25, 1731]])
```

The confusion matrix above can be visualized easier in the following table:

	Predicted Normal	**Predicted COVID**
True Normal	542	79
True COVID	25	1731

Table 3: Confusion Matrix

The confusion matrix tells us that $542 + 1731 = 2273$ images from the validation set were correctly classified and $79 + 25 = 104$ were incorrectly classified.

$$\text{Validation Accuracy} = \frac{2273}{2273 + 104} = 0.956.$$

The validation accuracy computed above matches with the validation accuracy Python returned. What the confusion matrix does tell us, that the plain validation accuracy output does not indicate, is the exact number of true positives, true negatives, false positives, and false negatives. We compute the precision and recall statistics, as well as the F1 score below:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{542}{542 + 79} = 0.8728.$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{542}{542 + 25} = 0.9559.$$

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \cdot \frac{0.8728 \times 0.9559}{0.8728 + 0.9559} = \boxed{0.9125}.$$

Since the F1 score ranges from 0 to 1, with higher scores indicating better performance, the F1 score of 0.9125 above is reasonably high.

No. Epochs	**Training F1 Score**	**Validation F1 Score**
5	0.8929	0.8734
10	0.9052	0.8805
20	0.9187	0.9125

Table 4: Training and Validation F1 Scores for Different Epochs

8.3 The Binary Cross-Entropy Loss Score

The accuracy metric does not reflect the confidence of the model on the data that it is asked to predict, since it relies on the simple 0-1 loss Δ. For example, if the model predicts an X-ray image as COVID-negative when its true label is COVID-positive, its 0-1 loss will be 0 and the accuracy for this instance only will be 1.

Consider two models that both achieve an accuracy of 1 for this particular instance with true label COVID-positive: Model A predicts COVID-negative for the instance with a high probability, and Model B predicts COVID-negative for the instance too but with a low probability. In this situation, both Models A and B will have an accuracy of 1 on the instance, but will differ in their binary cross-entropy loss scores on the instance. However, we might want to give more preference to the model that makes a wrong classification with a lower confidence, and in this case, the binary cross-entropy loss score is the more appropriate evaluation metric.

As stated above, I used the binary cross-entropy loss function to optimize my model on. The binary cross-entropy loss is a soft-classification (probability) score that ranges from 0 to 1. The following line plot is the binary cross-entropy loss over time for the Stochastic Gradient Descent (SGD) optimizer:

Figure 5: Binary Cross-Entropy Loss vs. Time with SGD Optimizer

The following line plot is the binary cross-entropy loss over time for the Adam optimizer:

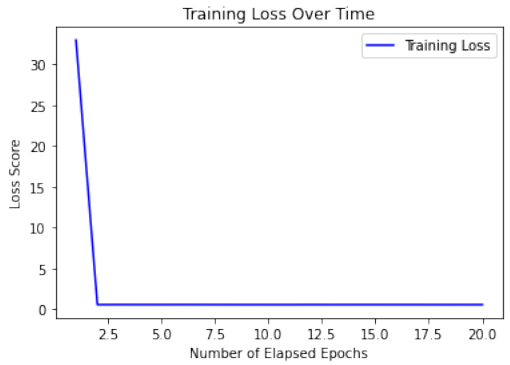


Figure 6: Binary Cross-Entropy Loss vs. Time with Adam Optimizer

As my final optimizer, I used SGD, since it resulted in a lower binary cross-entropy loss score after training the model for 20 epochs.

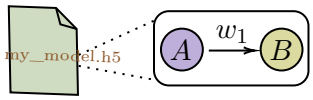
The binary cross-entropy loss score with the SGD optimizer monotonically decreased as the number of elapsed epochs increased, and near Epoch #20, it began to approach 0, indicating that the model performed well on the Chest X-ray image dataset to detect COVID-19. with respect to this evaluation method.

8.4 Final Evaluation Metric

As a final evaluation metric, I ended up giving more weight to the F1 score metric, since the distribution of the training dataset classes was uneven, i.e., there was a significant inbalance in the number of instances in the positive and the negative classes. Therefore, the F1 score is the more representative measure in this situation. Since the F1 score is high along with the high accuracy and the low binary cross-entropy loss score, I can confidently conclude that the specific VGG-16 architecture configuration I used fitted the data well and was able to generalize to untrained chest X-ray images well.

9 Designing the Web Application & Deploying the Model

Once the model had been trained and after evaluating the performance of the model on a held-out validation set, I exported the model architecture along with its trained weights with the Keras library as an `.h5` file.



Now, having the complete model, I was able to create a button with HTML and used Flask to encode the operation that needed to be performed on the button click. On the button click, the image that is inputted by the user (radiologist) is resized to a 256×256 image, and is then passed into the complete model. The complete model returns back a probability score measuring the likelihood of the model predicting the 'Normal' class, i.e., COVID-negative. Figure 7 on the following page supplements the process described above.

10 Possible Improvements & Optimizations

The following lists possible improvements and optimizations that can be made to my current architecture:

- Perform hyperparameter optimization with random search or Bayesian optimization to obtain the optimal number of epochs and the optimizer type resulting in a lower overall loss.
- Add more data augmentation to the training data in the preprocessing phase (to increase model generalizability).
- Re-perform the training process on a different model architecture, e.g., on Microsoft's ResNet architecture.

- Increase the size of the training data , and if this results in overfitting, add dropout to regularize the CNN and reduce the overfitting.
- Make the system interactive, i.e., the input of radiologists should be able to influence the output prediction.

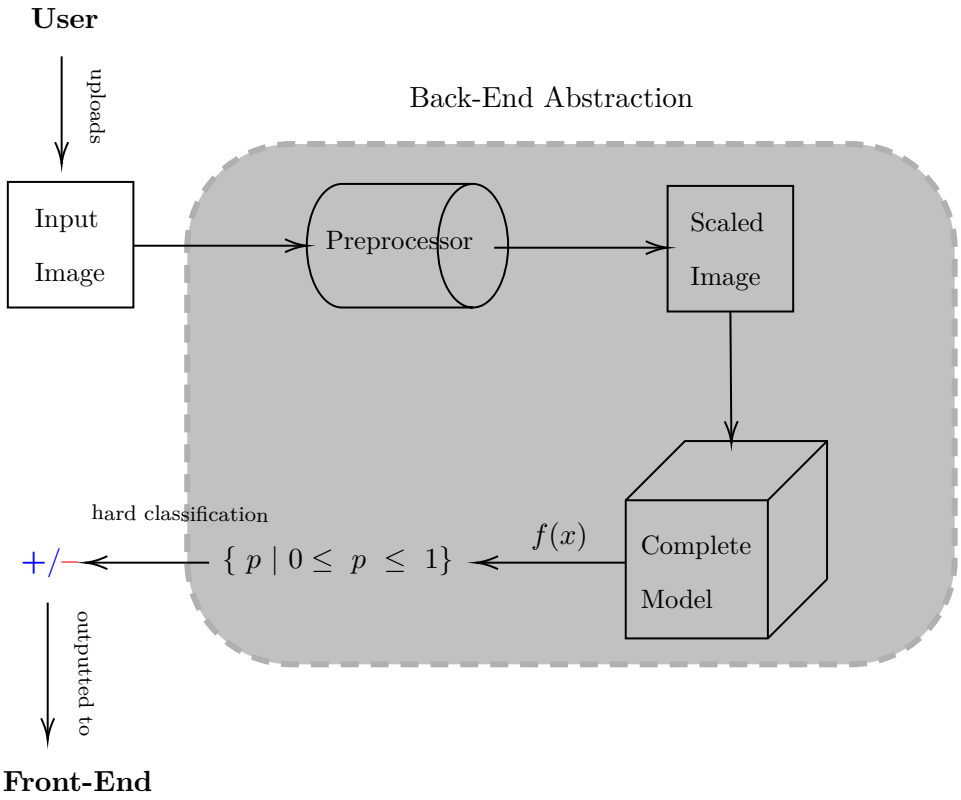


Figure 7: End-to-End Process

11 References

1. Alakus, Talha Burak, and Ibrahim Turkoglu. "Comparison of deep learning approaches to predict COVID-19 infection." Chaos, solitons, and fractals vol. 140 (2020): 110120. doi:10.1016/j.chaos.2020.110120.
2. Bhatia S., Sinha Y., Goel L. (2019) Lung Cancer Detection: A Deep Learning Approach. In: Bansal J., Das K., Nagar A., Deep K., Ojha A. (eds) Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing, vol 817. Springer, Singapore. https://doi.org/10.1007/978-981-13-1595-4_55.
3. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012): 1097-1105.
4. LeCun, Y., Bengio, Y. Hinton, G. Deep learning. Nature 521, 436444 (2015). https://doi.org/10.1038/nature14539.

5. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

6. Yadav, S.S., Jadhav, S.M. Deep convolutional neural network based medical image classification for disease diagnosis. J Big Data 6, 113 (2019). https://doi.org/10.1186/s40537-019-0276-2.

12 Link to Synopsis Video & Presentation Slides

The following are the links to the synopsis video and the presentation slides:

Link to Synopsis Video:

`https://cornell.box.com/s/9ylotec51rje93lzh1vbjobsxge29f0e`.

Link to Presentation Slides:

`https://github.com/rw544/Detecting-COVID-19-from-Medical-X-Ray-Images/blob/main/CS_4701_Project_Slides.pdf`