Face Mask Detection with Deep learning

Problem Statement

In recent trend worldwide due to COVID19 outbreak, a Face Mask has become mandatory for everyone. People around the world are wearing masks as a precautionary measure to prevent the spread of infection. In this project, a face mask detection model that can accurately detect whether a person is wearing a mask or not is proposed.

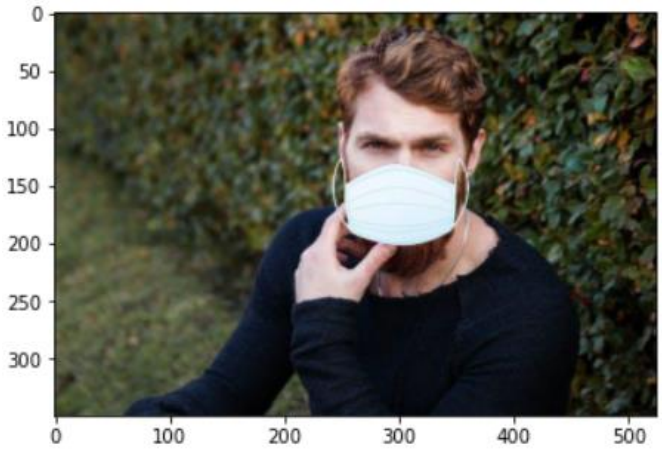
Data Preparation

The original dataset used in this project was from Kaggle Dataset. Data set consists of 7553 images with 3(RGB) channels in 2 folders as with_mask and without_mask. Images are named as label with_mask and without_mask. There are 3725 images of faces with_mask and 3828 images of faces without mask. The image data input parameters are the number of images, image height, image width, number of channels, and the number of levels per pixel. Typically, we have 3 channels of data corresponding to the colors Red, Green, Blue (RGB) Pixel levels are usually [0,255].

Exploratory Data Analysis

The image dimensions were 350 * 525, and the number of channels were 3(RGB). The below picture was randomly selected image from the data set. The image needed to be scaled down before building a model. The distribution of the pixel is shown in the Figure-1 below.

```
The Dimensions of the image are 525 width and 350 height. The mean of the pixel 0.2461 and std is 0.2157
```



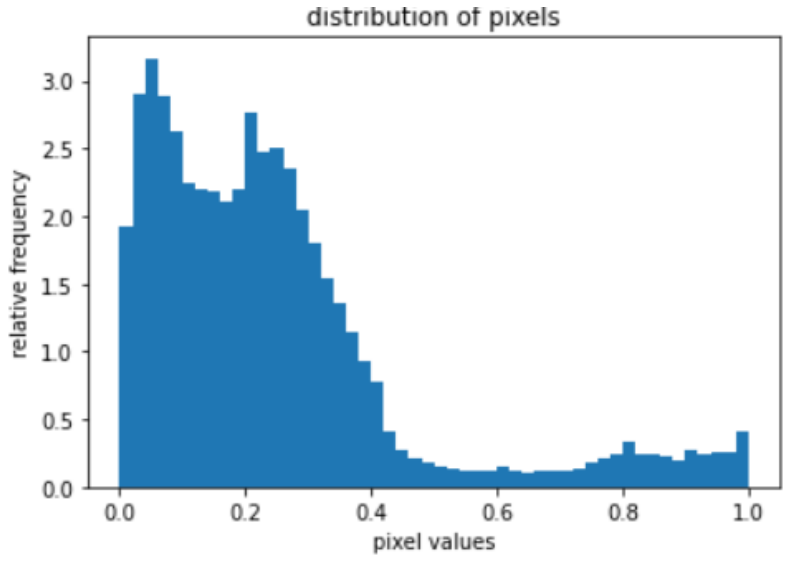


Figure 1

Data Preprocessing

The image dataset needed to be scaled to a standard size of 224 * 224 before modeling. The mean and standard deviation of pixel values were normalized to 0 and 1. The normalization helps the deep leaning models perform better as it helps to get data within a range and reduces the skewness since it's centered around 0. This helps learn faster and better. The number of color channels were not reduced, and the shape of tensor was (3,224,224). The channel dimension was left as 3 channels for the input layer. The final tensor will be of the form (C * H * W). Along with this, a scaling operation is also performed from the range of 0–255 to 0–1. The dataset was split into training set with 80% of dataset and testing set with 20% of dataset.

While training a model, samples in "minibatches" were passed for training and reshuffle the data at every epoch to reduce model overfitting. The dataset was loaded into the DataLoader and can iterate through the dataset as needed. The 'batch_size' is the number of training samples in one iteration or one forward/backward pass. The batch_size argument was set as 4 in Dataloader, as 4 images were passed at every iteration of training the network. The first 4 images (1 to 4) are passed into the network, then the next 4 (5–8), and so on until all the samples were processed. Splitting the data into batches was crucial because the network was constantly learning and updating its weights. The batch size could be increased to achieve higher accuracy level, but it could be taking up more memory space. To speed up the training process, we could make use of

the num_workers optional attribute of the DataLoader. The num_workers attribute tells the data loader instance how many sub-processes to use for data loading. By default, the num_workers value is set to zero, and a value of zero tells the loader to load the data inside the main process. This means that the training process will work sequentially inside the main process. The num_workers was set as 2. This means there are 2 workers simultaneously putting data into the computer's RAM. The figure-2 below shows the image batch of 4 from the trainloader.

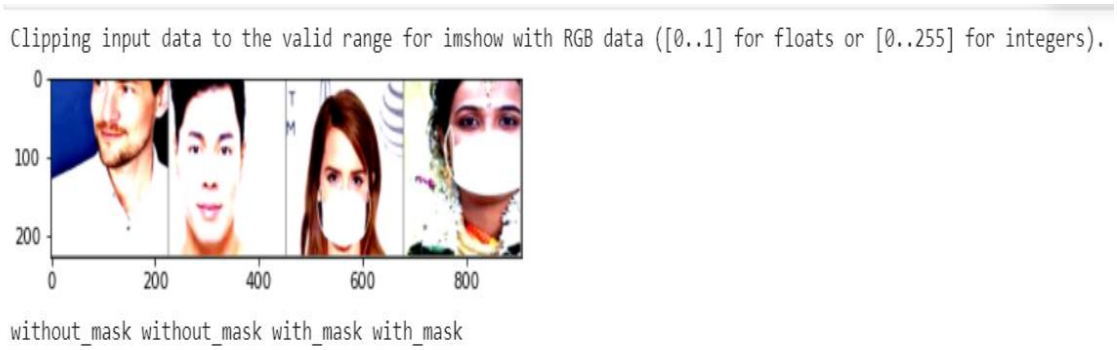


Figure 2

Modeling

A Convolutional Neural Network (CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various objects in the image and be able to differentiate one from the other. The figure-3 below represents the basic architecture of CNN.

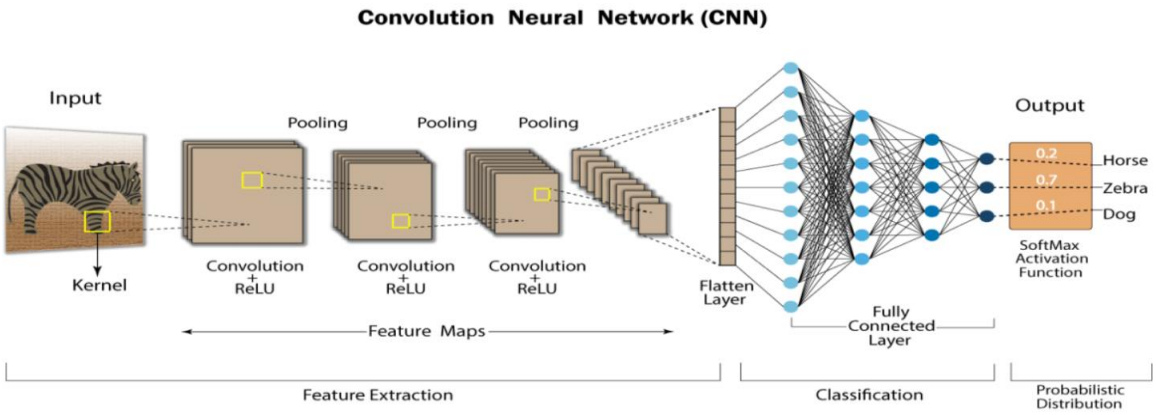


Figure 3

Pytorch has different ways to create custom CNN model. They are OOP based and Functional based for creating CNN network. In this project, CNN model was developed using object-

oriented approach. This allowed to tweak every aspect of the network and could easily visualize the network along with how the forward algorithm works. The 'Net' class was created by inheriting the pytorch nn.Module class. The Net class for the CNN model is visualized in the below Figure-4.

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc): Linear(in_features=100352, out_features=2, bias=True)
)
```

Figure 4

The Net class was initialized with Convolutional Layers (conv), pooling, and Fully Connected (fc) layers. The three important elements of CNN architecture are below.

1. Convolutional Layer

The CNN model was developed with 3 convolution layers. The convolution operation was performed by sliding the filter over the input. The kernel_size was set as 3 which indicates 3*3 Filter. At every location element wise matrix multiplication was done and the result were summed. The below figure-5 shows an example of convolution operation on the image.

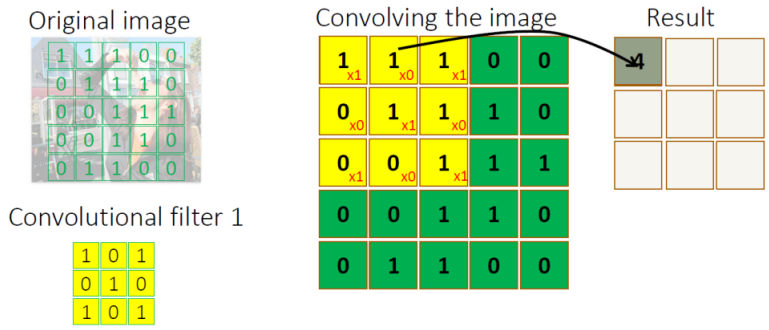


Figure 5

The 'input_channel' was set to 3 in the first Cov2d because of the image had 3 channels(RGB). The 'output_channel' was set to 32. The resulting feature from the first layer was 32 and was set as the 'input_channel' for the second Cov2d. Padding was used to preserve the size of the feature

map. Otherwise, the feature map would shrink at each layer which is not desirable. The 'padding' was set to 1 for adding one layer of padding. In convolution operations, stride specifies how much we move the convolution filter at each step. The 'stride' value was set as 1.

2. Pooling

After a convolution operation, pooling was performed to reduce the dimensionality. This enables to reduce the number of parameters. That helps both shorten the training time and combat overfitting. Pooling layers down-sample each feature map independently and reduces the height and width, keeping the depth intact. The most common type of pooling is max pooling. It just takes the max value in the pooling window. In Net model (2,2) was into MaxPool2d and image was tuned into 2x2 dimensions while retaining "important" features

The figure-6 shows an example of Max pooling

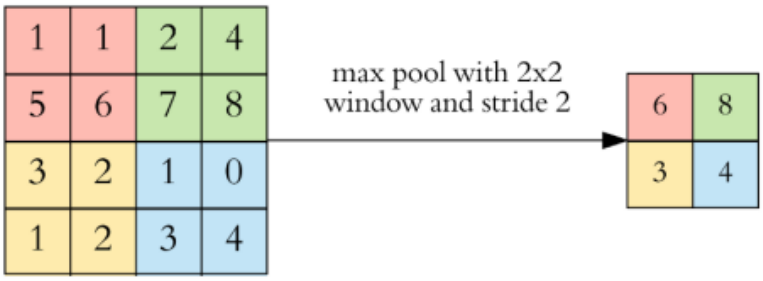


Figure 6

3. Fully connected Layer

Fully connected layers means that every neuron from the previous layers connects to all neurons in the next layer. Fully connected layers are the last few layers in the network. The input to the fully connected layer was the output from the final Pooling or Convolutional Layer, which was flattened and then fed into the fully connected layer. The Linear transformation in 'fc' layers was the same mathematic operation that happens in Artificial Neural Network. The operation follows the form of g (Wx + b) where x is our input vectors, W is the weight matrix, b is the bias vector and g is an activation function, which was ReLU in this model. ReLU (Rectified Linear Unit) is one of the most used activation functions: y = max (0, x) x >= 0 y = x ; x < 0 y = 0. The figure-7 below shows the ReLu activation function.

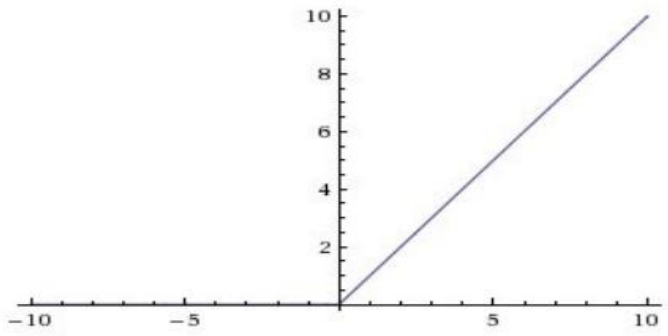


Figure 7

After declaring all the parameters of CNN, it was implemented by the net's forward() method. In the forward() method, Conv2d was passed into Relu non-linearity function followed by the pooling operation and this was done for all the three Conv2d layers. Then, the view() method was called to transform the feature map to be passed into the full connected layer. The reason for using view () was that we need to flatten the output from our conv layer and pass it to the fully connected layer. Another option is to use flatten () which could be easier to understand.

Loss function and Optimizer

After developing the CNN model, we must define the loss function and Optimizer. CrossEntropyLoss from PyTorch is used for training classification problems. It combines SoftMax and Negative Log-Likelihood. Pytorch has 'optim' which is an optimization algorithm. The optimizer was constructed using this and it updates the parameters based on the computed gradients. This cycle happens until the training ends. It is basically a fundamental tool for the network to "learn" and update its weights from backpropagation. Stochastic Gradient Descent (SGD) was used in implementing gradient descent. The parameter to the optimizer was the net.parameters() from the model based on the Net class. Once, we have defined all the functions, the trainloader was passed over the Net model. An 'epoch' is one pass over the entire training set one time. The training set was passed over ten time for learning the features. The Figure-8 below shows the training and testing losses in each iteration.

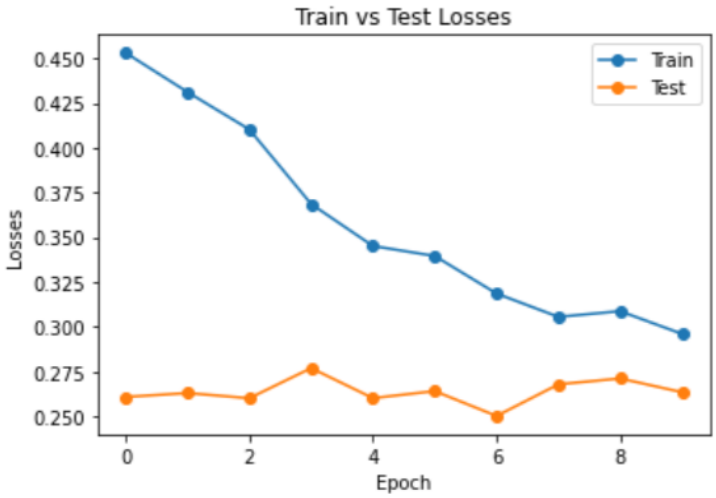


Figure 8

The training and the testing accuracy of the model was 89% . The model was tested with random images from the testing set. The Figure-9 below shows the actual labels and labels predicted by the model.

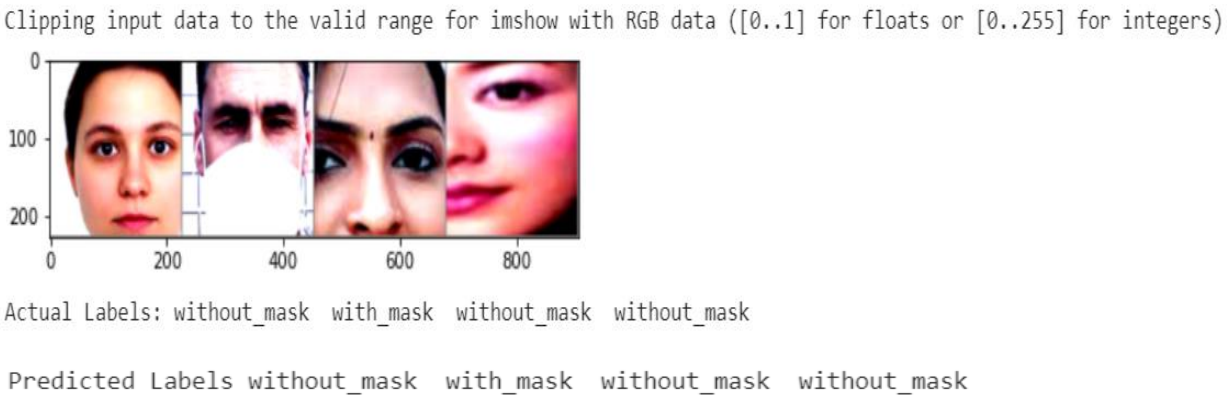


Figure 9

Transfer Learning

Transfer learning is a technique by which we can use the model weights trained on standard datasets such as ImageNet to improve the efficiency of our given task. The learning process during transfer learning is fast and accurate. Generally, a Transfer learning model performs 20% better than a custom-made model. It needs less training data, being trained on a large dataset, the model could already detect specific features and need less training data to further improve the model. To perform transfer learning import a pre-trained model using PyTorch, then freeze the

weights for all the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained. The figure-10 shows an example of transfer learning.

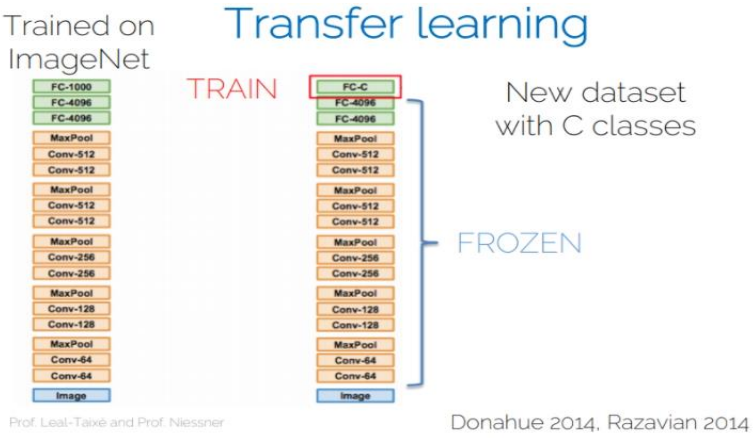


Figure 10

Transfer Learning using ResNet18

ResNet-18 is a convolutional neural network that is 18 layers deep. This is a pretrained version of the network which is trained on more than a million images from the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network can take an image input size of 224-by-224. The below figure-11 shows the original architecture

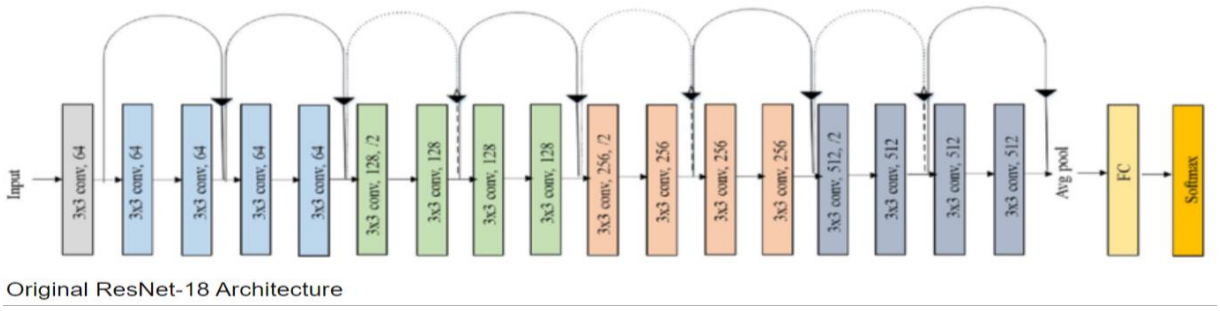


Figure 11

The torchvision package has a pretrained ResNet18 network. We must Freeze all the layers bar the final one. Then change the last fully connected layer to correspond to the number of classes dataset in this case is 2. The parameters were frozen by (not pass the gradient) using requires_grad

= False. The CrossEntropyLoss function and the Stochastic gradient descent optimizer was used model tuning. The dataset is further divided into training and validation set to avoid overfitting. The training and testing set were passed over 15 time for learning the features. This model predicted with the validation accuracy of around 98%. The figure-11 below shows the training and validation losses and the accuracy of the model at each epoch set.

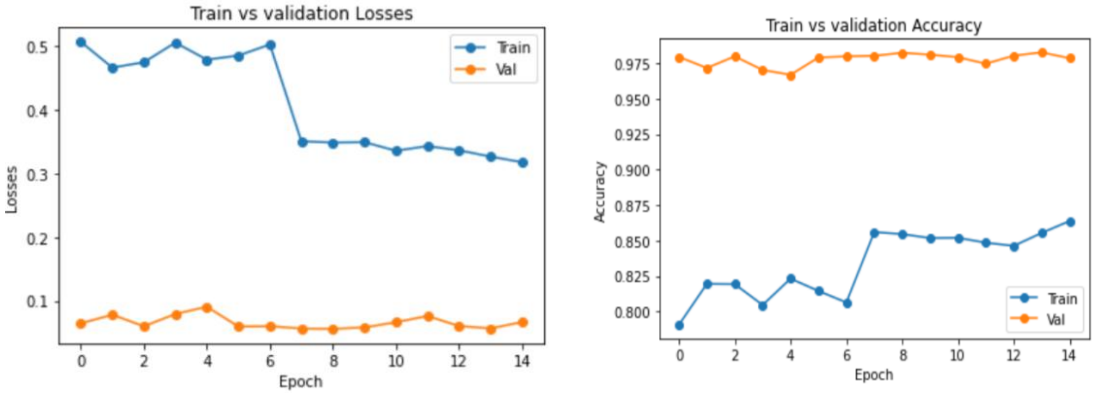


Figure 12

The model was tested with random images from the validation set. The Figure-12 below shows the actual labels and the labels predicted by the model

```
Actual Labels: with_mask  without_mask  without_mask  without_mask

Predicted Labels with_mask  without_mask  without_mask  without_mask
```

Figure 13

Transfer Learning using VGG16

There are two models available in VGG, VGG-16, and VGG-19. VGG-16 mainly has three parts: convolution, Pooling, and fully connected layers. In the Convolution layer, filters are applied to extract features from images. The most important parameters are the size of the kernel and stride. The function of Pooling layer function is to reduce the spatial size to reduce the number of parameters and computation in a network. The fully connected layer connects to the previous layers as in a simple neural network. The figure-14 shows the architecture of VGG-16.

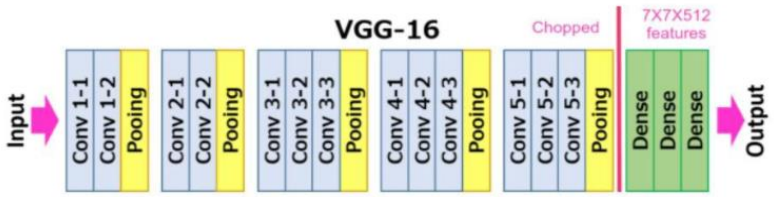
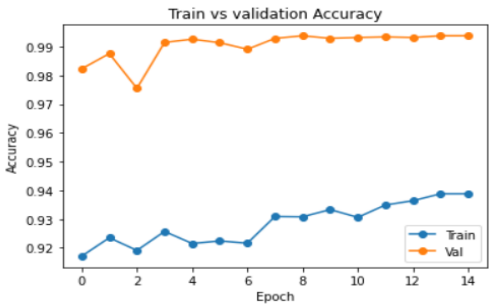
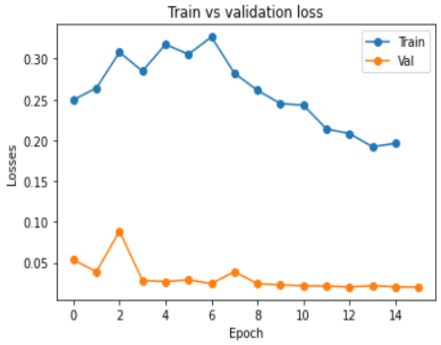


Figure 14

To perform transfer learning a pre-trained model was imported using PyTorch, removed the last fully connected layer (as this model gives 1000 outputs and we can customize it to give a required number of outputs) and run the model. The CrossEntropyLoss function and the Stochastic gradient descent optimizer was used model tuning. The model predicted with a validation accuracy of 99.3%. The figure below shows training and validation losses and accuracy for the model at each epoch.





The model was tested with random images from the validation set. The Figure-12 below shows the actual labels and the labels predicted by the model.

Actual Labels: without_mask with_mask with_mask with_mask

Predicted Labels without_mask with_mask with_mask with_mask

Conclusion

In this project, the focus was to classify images with mask and without mask. A custom deep learning model was developed and compared with pretrained models using transfer learning. The VGG16 model gave the best accuracy in classifying the images. This model could be used in classifying the image in real time.