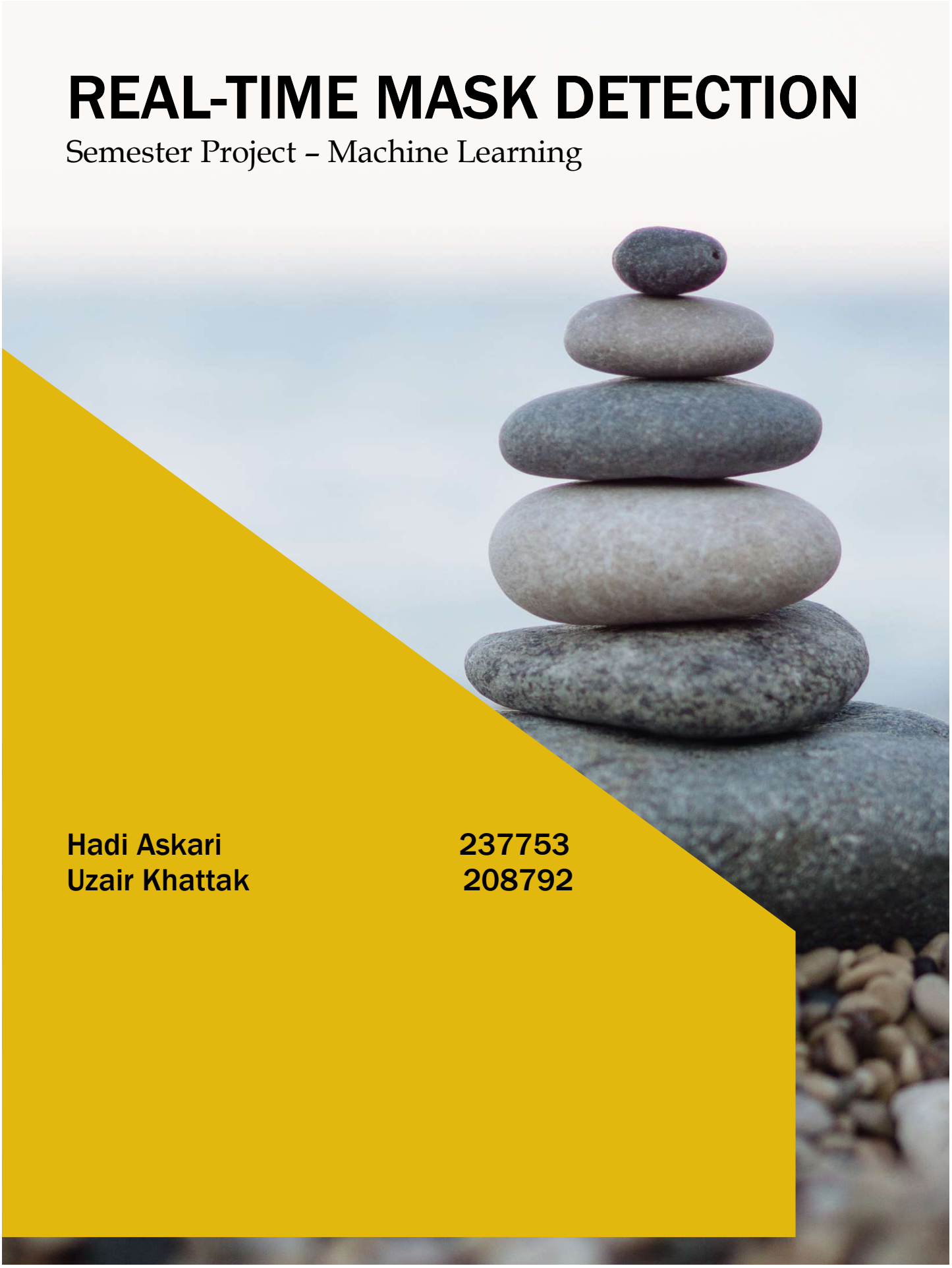
REAL-TIME MASK DETECTION

Semester Project – Machine Learning



Hadi Askari 237753

Uzair Khattak 208792

Contents

INTRODUCTION:

IMPORTS:

The following snippet shows the libraries imported and used in this pytorch project.

```
import torch
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
```

Pytorch Imports:
Their functionalities are as follows:

- torch: the top-level PyTorch package and tensor library
- torchvision: a package that provides access to popular datasets, model architectures, and image transformations for computer vision
- torchvision.transforms: an interface that contains common transforms for image processing
- torch.nn: a sub-package that contains modules and extensible classes for building neural networks.
- torch.nn.functional: a functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations

Other Imports:
These are standard packages used for data science in Python:

- NumPy
- matplotlib.pyplot

DATA LOADING AND PREPROCESSING:

PyTorch has some built-in packages and classes that make this process pretty easy. The following snippet shows the loading and preprocessing of the dataset.

```
transform=transforms.Compose([transforms.Resize((160,160)),transforms.ToTensor()])
#step 1 Loading the data ad preprocessing
local_address=r'/home/uzair/Desktop/dataset/dataset/'
train_data=torchvision.datasets.ImageFolder(root=((local_address) + 'train/') ,
                                            transform=transforms.Compose([transforms.Resize((160,160)),transforms.ToTensor()]))
test_data=torchvision.datasets.ImageFolder(root=((local_address)+'valid/'),
                                           transform=transforms.Compose([transforms.Resize((160,160)),transforms.ToTensor()]))
train_loader=torch.utils.data.DataLoader(train_data,batch_size=8,
                                         shuffle=True,
                                         num_workers=4)
test_loader=torch.utils.data.DataLoader(test_data,batch_size=2,
                                        shuffle=True,
                                        num_workers=4)
complete_data_dictionary={'train': train_loader , 'valid': test_loader}
print(complete_data_dictionary)
```

```
{'train': <torch.utils.data.dataloader.DataLoader object at 0x7fa261539cf8>, 'valid': <torch.utils.data.dataloader.DataLoader object at 0
x7fa261539f28>}
```

Here, by using the "torchvision" module that is provided by PyTorch, we use Datasets and DataLoaders in the PyTorch "torch.utils.data" package to streamline the loading and preprocessing tasks. These classes apply separately on the training data and the test data.

Pytorch Dataset Class:

It extracts the images from the folders "train" and "valid" of the training data and the test data respectively. Its arguments are:

- root: the location on disk where the data is located
- transform: composition of transformations that should be performed on the dataset elements

The images in our dataset are not of the same size. Most neural networks expect the images of a fixed size. So, we create two transforms "Resize" and "ToTensor". Resize (160,160) scales all the images to a size of 160×160. ToTensor () converts the NumPy images to tensors. We compose these two transforms in a single callable command "Compose".
We have named the instances "train_data" and "test_data" for training and testing the dataset respectively.

Pytorch DataLoader Class:

Here, we pass the instances "train_data" and "test_data" as arguments separately to their respective Dataloader classes. Then, we leverage the loader for tasks that would otherwise be pretty complicated to implement by hand:

- batch_size (8 for "train_loader" and 2 for "test_loader")
- shuffle (True in our case)
- num_workers (4 for both dataloader classes)

Here, "batch_size" tells how many samples or images to load in a single batch. We set "shuffle" to True, so that we get a new order of exploration at each pass. Setting the argument "num_workers" as a positive integer will turn on multi-process data loading with a specified number of processes generating batches in parallel.

Exploring the Data:

Now, we take a look at the following code snippet from the project:

```
print(len(train_loader))
batch=next(iter(test_data))
print(len(train_data))
print(len(test_data))
```

```
276
2206
551
```

The first command prints the number of batches of images in our training dataset i.e. 276. We have set the batch size of the training dataset as 8. So, multiplying the number of batches in training dataset (i.e. 276) with the batch size (i.e. 8) gives us the total number of images in our training dataset that is 2206. The third command in the above snippet prints the number of images in our training dataset (i.e. 2206).

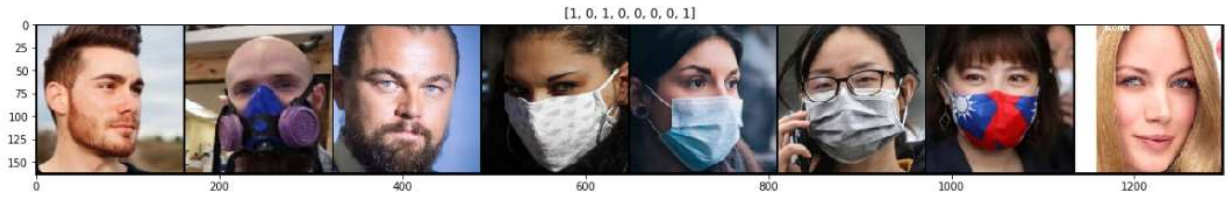
Visualizing the Data:

Now, let's visualize our data:

```
#step 2 Visualizing the data
def view_data(loaded_data):
    batch=next(iter(loaded_data))
    images,labels=batch
    print(images.shape)
    print(images.shape[0])
    my_grid=torchvision.utils.make_grid(images,nrow=images.shape[0])
    plt.figure(figsize=(20,20))
    plt.imshow(np.transpose(my_grid,(1,2,0)))
    plt.title([x.item() for x in labels])
view_data(train_loader)

device=torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device=torch.device("cpu")
```

```
torch.Size([8, 3, 160, 160])
8
```



To visualize our data, we create a function "view_data" that takes "loaded_data" as an argument.

To access an individual element from the training set, we first pass the "loaded_data" object to Python's "iter()" built-in function, which returns an object representing a stream of data. With the stream of data, we use Python's built-in "next()" function to get the next data element in the stream of data. From this we get a single sample. We name it "batch".

We can see that the "batch" contains two items, and this is because the dataset contains image-label pairs. Each sample we retrieve from the training set contains the image data as a tensor and the corresponding label as a tensor.
Now, we check the shape to see that the image is a 8×3×160×160 tensor while the label is a scalar valued tensor. The shape shows the batch size is 8, color channel is 3 (i.e. RGB), and the height and width of the image are 160×160.
To plot a batch of images, we use the "torchvision.utils.make_grid()" function to create a grid which takes the instance "images" as argument and the nrow specifies the number of images to be displayed in each row of the grid. Here we set nrows=8.
We set the figure size of the plot as 20×20.
Next, to show the plot on the grid we take transpose
We set the title of the plot that shows the scalar value of 0 meaning wearing a mask and 1 meaning not wearing a mask.

MODEL FRAMEWORK:

Network Architecture:

We make a class named Network which inherits its characteristics from nn.Module class.

Attributes:
It has two convolutional layers and three linear layers as its attributes. We have three input channels (color channels), 6 filters and 6 output channels. We define our kernel size to be 5. The input channels/features of next layer are same as output channels/features of previous layer. The height and width of our images also decreases when passed onto next layer.

```
class Network(nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1=nn.Conv2d(in_channels=3,out_channels=6,kernel_size=5)
    self.conv2=nn.Conv2d(in_channels=6,out_channels=12,kernel_size=5)
    self.fc1=nn.Linear(in_features=12*37*37,out_features=120)
    self.fc2=nn.Linear(in_features=120,out_features=60)
    self.out=nn.Linear(in_features=60,out_features=2)
```

Methods:
Then we define a forward function and pass our training set ‘t’ through it. The input layer just reflects our training set. Then the convolution layers contain activation and max pool functions in which we define our kernel size and stride value. When we come to linear layers, we flatten the output of convolutional layer. And our data is passed onwards till the last output layer.

```
def forward(self,t):
  #input layer
  t=t
  #conv1 layer
  t=self.conv1(t)
  t=F.relu(t)
  t=F.max_pool2d(t,kernel_size=(2,2),stride=2)
  #2nd conv2d layer
  t=self.conv2(t)
  t=F.relu(t)
  t=F.max_pool2d(t,kernel_size=(2,2),stride=2)
  #now linear layer implementation
  #first flatten the conv layer output
  t=t.reshape(-1,12*37*37)
  t=self.fc1(t)
  t=F.relu(t)
  #next linear
  t=self.fc2(t)
  t=F.relu(t)
  #final layer
  t=self.out(t)
    t=torch.sigmoid(t)

  return t
```

Training and Valdidation:

After preparing our data, we work on building our model and then training it. In our training model, we pass model of our network, prepared data, optimizer (used to optimize weights), the given loss and scheduler (used to decrease learning rates after each epoch). For training our data, we will do both forward and backward passes. And for testing data, we will do only forward pass.

```
def my_train_model(model,data,optimizer,given_loss,scheduler,total_epochs=40):
    #making variables for plotting of graphs at later visualization stage
    train_loss , train_acc, val_loss, val_accuracy = [],[],[],[]
    #first epochs , then another loop in each epoch to cover training data
    #in each epoch,first train the model , and then run the validation data to test accuracy
    #ptimizer=optimize(model.parameters(),lr=learning_rate)
    my_sizes={ 'train': len(train_data), 'valid' : len(test_data)}
    #first loop for the epochs
    for i in range (total_epochs):
        print('EPOCH NO:' + str(i) )
```

We use two for loops. Each epoch will run two times, one for training set and one for test set. Schedular function is used for training set and eval() function is used for test set.

```
for current_run in ['train' , 'valid']:
    current_epoch_loss=0.0
    total_correct=0.0

    if current_run=='train':
        model.train()       #run the model in the training mode
        scheduler.step()
    else:
        model.eval()   #run model in validation mode
```

The second for loop is used to obtain both data and labels from the batch. They are loaded into GPU processing if it's available. The .to(device) command is used for this purpose. It loads data into either CPU/GPU (whichever is available).

```
    for batch in data[current_run]:


        #now performing the forward steps
        input_data,labels=batch
        #put data into GPU processing if available
        input_data=input_data.to(device)
        labels=labels.to(device)
```

After this, gradients are calculated fot our training set by performing both forward and backward passes. And then they are updated with the help of optimizer function.

```
            with torch.set_grad_enabled(current_run=='train'):
                #forward pass
                my_prediction=model(input_data)
                #find loss
                loss=given_loss(my_prediction,labels)
                #clear previous gradients
                #Run backward for only training time
                if current_run=='train':
                    optimizer.zero_grad()
                    #update gradients
                    loss.backward()
                    optimizer.step()



                current_epoch_loss +=loss.item()*input_data.size(0)
                total_correct+=my_prediction.argmax(dim=1).eq(labels).sum().item()
```

Then we get our accuracy printed which is Total correct results/ Batch size

```
        print(str(current_run) + '  loss = '  + str(current_epoch_loss))
        print(str(current_run) + ' Accuracy= ' +  str(total_correct/my_sizes[current_run]))
        if current_run=='train':
                    train_acc.append(total_correct/my_sizes[current_run])
                    train_loss.append(current_epoch_loss/my_sizes[current_run])
        else:
            val_accuracy.append(total_correct/my_sizes[current_run])
            val_loss.append(current_epoch_loss/my_sizes[current_run])
return model
```

Defining Hyper Parameters:

```
my_model=Network()
learning_rate1=0.001
my_loss=nn.CrossEntropyLoss()
my_optimizer=torch.optim.Adam(my_model.parameters(),lr=learning_rate1)
dynamic_learning_rate=torch.optim.lr_scheduler.StepLR(my_optimizer,step_size=7,gamma=0.1)
trained_model=my_train_model(model=my_model.cuda(),data=complete_data_dictionary,
               optimizer=my_optimizer,
                    given_loss=my_loss,
               scheduler=dynamic_learning_rate)
```

We have 2 classes: Mask/ No mask.
We create network instance and pass arguments for learning rate, loss function and optimizer. After every epoch, we decrease our learning rate by 7 times using the scheduler function. Then our training process starts.

TRAINED MODEL TESTING:

Saving and Loading the model:
After training the model we save the updated weights on our local computer and then load these weights for future use using *torch.save* and *torch.load* commands.

```
torch.save(trained_model ,r"C:\Users\Hadi Askari\Desktop\Dataset\mask_detector1")
```

```
trained_model=torch.load(r"C:\Users\Hadi Askari\Desktop\Dataset\mask_detector1")
```

Testing our model:
Next, we want to test our model via an external image. We make use of the *Python Image Library (PIL)* to load a personal picture and check to see if our model was producing the appropriate results. The following picture was uploaded:

We applied the same transformations on this picture as to the pictures in our dataset and received the following prediction:

```
test=transform(image)
test=test.unsqueeze(0)
```

```
trained_model.eval()
my_prediction=F.softmax(trained_model(test.cuda()))
my_prediction.argmax(dim=1).item()
#remember 1 means NO MASK
#zero means MASK
```

```
Out[10]: 1
```

As we can see the model recognized that the person in the image is not wearing a mask and produced the result as expected.

IMPLEMENTING REAL TIME MASK DETECTION:

In order to create our real time mask detector, we used to libraries primarily. *OpenCV* for computer vision and *MTCNN* for real-time face detection and extraction. There were several steps through which we traversed.

Capturing and Displaying Video:

We used *cv2.VideoCapture(0)* to create a video capture object for the camera. After that we create an infinite while loop and called the *read ()* method on the object to read the frames. To display the video, we implemented *cv2.imshow()* inside the loop to show the frames in the video. The loop will break after a key is pressed and the object will be released using the *release ()* method. *cv2.destroyAllWindows()* is used to destroy the window. We also used *cv2.putText()* to display the access_text and deny_text based on the result of our model.

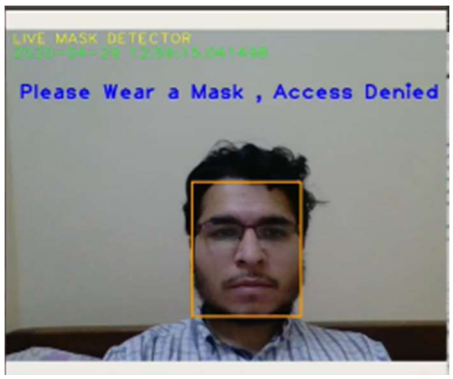
Creating the bounding box and extracting the face:

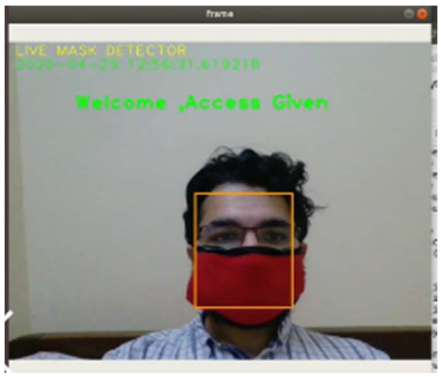
We created a detector object using the *MTCNN()* method and sent the frames we extracted via the read() call to the *detect_faces* object. We save this in the result variable. Now for each individual face that the *MTCNN()* object detects we create a bounding box. We save the co-ordinates of the bounding box in the box variable using *box=person['box']*. Using the co-ordinates of the box and *cv2.rectangle* we draw the bounding box on the live frame. To extract the face and send it to the model we cut out the edges of the detected box on each frame and send it to our model.

```
extracted_face=frame[box[1]:box[1]+box[3],box[0]:box[0]+box[2]]
```

Creating and Displaying Results:

At this point each extracted frame from the live feed takes the form of an individual picture and can be analyzed by the model that we have trained. We use the same process as we did in testing the individual picture before to get a 0 or 1 prediction. Based on the result we display the deny or access text.





CONCLUSION:

Pytorch is thus an extremely user-friendly machine learning library with all the necessary functions that are required to design a practically implementable models of computer vision, natural language processing etc. The presence of variety of algorithms and hyper parameters help us to tune our model in the best possible shape. Considering our model, after multiple trails, we were able to fine tune our model and resultantly achieved the following accuracies.

Training Accuracy = 99.2 %
Validation Accuracy = 96.9 %

```
train  loss = 52.442998823476955
train Accuracy= 0.9918404351767905
valid  loss = 53.81924865486714
valid Accuracy= 0.969147005444646
```

Although the model works brilliantly for testing random images owing to the train and test accuracies, yet it sometimes underperforms when it comes to real time sensing using webcam. The reason of this underperformance is due to the fact that that the webcam images have resolutions that are far inferior to our provided data set. In order to further optimize the model, webcam images should be added to our training set for enhancing the accuracy for real time sensing.

Considering the current situation with the COVID-19 pandemic, this very project is extremely relevant and implementable. This very model could be interlinked with the CCTV cameras of various workplaces, malls, restaurants, marts and offices to ensure maximum safety which is a top priority for the current health care system.