Evaluating on the Test Data

In this notebook we are going to **evaluate our model for the test dataset**.

In the **first cell** we are going **import the necessary packages**.

In [9]:
```
## Importing necessary packages ##

import torch
import torch.nn as nn
import torchvision
from torchvision.datasets import ImageFolder
from torchvision.utils import make_grid
from torch.utils.data import Dataset , DataLoader , Subset
from torch.utils.data.sampler import WeightedRandomSampler
from torchvision.transforms import transforms
import torch.nn.functional as F
from torch.nn.modules.utils import _pair, _quadruple
from torchvision.transforms.functional import resize
from sklearn.metrics import classification_report

from tqdm import tqdm
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Now we are going to bring in the test dataset using the `ImageFolder`.

In [10]:
```
## Implementing transformations ##

trans = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((64 , 64)),
    transforms.ToTensor()
])


## Importing test dataset ##

covid_test_dataset = ImageFolder(root = '../input/cowarriornet-test/test/test'
,
                                 transform = trans)

print('Test Dataset Loaded successfully!')
print('Length of test dataset :' , len(covid_test_dataset))

## Implementing the DataLoader ##

test_dataloader = DataLoader(covid_test_dataset , batch_size = 64 , shuffle =
True)
```

```
Test Dataset Loaded successfully!
Length of test dataset : 1288
```

In [11]:
```
## Sending the dataloader to GPU ##

## Defining utility function to check for gpu and set the torch device as cuda
##

def get_device():

    if torch.cuda.is_available():

        return torch.device('cuda')

    return torch.device('cpu')

## Setting the current device ##

device = get_device()

## Defining transfer data to given device utility function ##

def transfer_data(data , device):

    if isinstance(data , (list , tuple)):

        return [transfer_data(each_data , device) for each_data in data]

    return data.to(device)

## Defining our GPU Dataloader class ##

class GPUDataloader:

    def __init__(self , dl , device):

        self.dl = dl

        self.device = device

    def __iter__(self):

        for batch in self.dl:

            yield transfer_data(batch , self.device)

    def __len__(self):

        return len(self.dl)


## Setting our dataloader as a GPU dataloader ##

test_dl = GPUDataloader(test_dataloader , device)

print('GPU Dataloader successfully created!')

print('Dataloader has' , len(test_dl) , 'mini-batches!')
```

```
GPU Dataloader successfully created!
Dataloader has 21 mini-batches!
```

Now we are going to load the model. But, for the pytorch model to work we need to re-create our class too.

In [12]:
```
## Creating the Alpha Trimmed Average Pooling Layer ##

class AT_AvgPool(nn.Module):
    """ Alpha Trimmed Average Pooling module.

    Args:
         kernel_size: size of pooling kernel, int or 2-tuple
         stride: pool stride, int or 2-tuple
         d: Percentage of values to not consider when calculating average, nee
d more than 1 value to remove, float
         padding: pool padding, int or 4-tuple (l, r, t, b) as in pytorch F.pa
d
         same: override padding and enforce same padding, boolean
    """
    def __init__(self, kernel_size=3, d = 0 ,stride=1, padding=0, same=False):

        super().__init__()

        self.k = _pair(kernel_size) # repeat the kernel. Suppose input is kern
el_size = 3, then this makes it 3*3

        self.stride = _pair(stride)

        self.padding = _quadruple(padding)  # repeated four times for the four
sides.

        self.same = same # if same padding is given

        self.num_remove = math.floor(d * self.k[0] * self.k[1]) // 2

        if self.num_remove == 0:
            self.num_list = list(range(self.k[0] * self.k[1]))

        else:
            self.begin = max(0 , self.num_remove)
            self.end = min(0 , -self.num_remove)
            self.num_list = list(range(self.k[0] * self.k[1]))[self.begin:self
.end]

    def _padding(self, x):

        if self.same:
            ih, iw = x.size()[2:] #Input height = x.size()[2] , Input width =
 x.size()[3]

            if ih % self.stride[0] == 0: # if stride matches with the height d
imension
                ph = max(self.k[0] - self.stride[0], 0) # Padding height

            else:
                ph = max(self.k[0] - (ih % self.stride[0]), 0)

            if iw % self.stride[1] == 0: # Similarly for width
                pw = max(self.k[1] - self.stride[1], 0)

            else:
```

```
                pw = max(self.k[1] - (iw % self.stride[1]), 0)

            pl = pw // 2 # padding in the left part

            pr = pw - pl # padding in the right part

            pt = ph // 2 # padding in the top part

            pb = ph - pt # padding in the bottom part

            padding = (pl, pr, pt, pb)

        else:
            padding = self.padding

        return padding

    def forward(self, x):
        # using existing pytorch functions and tensor ops so that we get autog
rad,
        # would likely be more efficient to implement from scratch at C/Cuda l
evel



        x = x.to(torch.float)

        x = F.pad(x, self._padding(x), mode='reflect') # Putting padding on al
l sides of x

        x = x.unfold(2, self.k[0], self.stride[0]).unfold(3, self.k[1], self.s
tride[1]) # Getting the kernel sized patch

        x = torch.sort(x.contiguous().view(x.size()[:4] + (-1,))).values[: , :
, : , : , [self.num_list]].mean(dim=-1).squeeze(-1)

        #print(x.size())

        return x

## Defining the Identity Conv Block ##

class IdentityConv(nn.Module):

    def __init__(self , in_dim , out_dim , kernel_size = 3 , padding = 1 , str
ide = 1):

        super().__init__()

        self.conv_block = nn.Sequential(nn.Conv2d(in_channels = in_dim , out_c
hannels = out_dim , kernel_size = kernel_size ,
                                          padding = padding , stride = stride , b
ias = False),
                                nn.BatchNorm2d(num_features = out_dim) ,
                                nn.ReLU(inplace = True),
                                nn.Conv2d(in_channels = out_dim , out_channels = o
ut_dim , kernel_size = kernel_size ,
```

```
                                        padding = padding , stride = stride , b
ias = False),
                              nn.BatchNorm2d(num_features = out_dim) ,
                              nn.ReLU(inplace = True))

    def forward(self , inp):

        return self.conv_block(inp)

## Defining the Derived Unet Architecture ##

class DerivedUnet(nn.Module):

    def __init__(self , in_dim = 1 , out_dim = 1 , filters = [64 , 128 , 256 ,
512]):
        super().__init__()
        self.filters = filters
        self.out_dim = out_dim
        self.pool = AT_AvgPool(kernel_size = 2 , d = 0.5 , stride = 2)

        self.down_block = nn.ModuleList([])
        self.up_block = nn.ModuleList([])

        for each_filter in self.filters:
            self.down_block.append(IdentityConv(in_dim = in_dim , out_dim = ea
ch_filter))
            in_dim = each_filter

        for each_filter in reversed(self.filters):
            self.up_block.append(nn.ConvTranspose2d(in_channels = each_filter
* 2 , out_channels = each_filter ,
                                                    kernel_size = 2 , stride=
2))
            self.up_block.append(IdentityConv(in_dim = each_filter * 2 , out_d
im = each_filter))


        self.bridge_block = IdentityConv(in_dim = self.filters[-1] ,
                                         out_dim = self.filters[-1] * 2)

        self.final_layer = nn.Conv2d(in_channels = self.filters[0] , out_chann
els = self.out_dim,
                                     kernel_size = 1)


    def forward(self , x):

        add_x = x

        down_conv_out = []

        for layer in self.down_block:
            x = layer(x)
            down_conv_out.append(x)
            x = self.pool(x)

        x = self.bridge_block(x)
```

```
        down_conv_out = down_conv_out[::-1]

        for layer_num in range(0 , len(self.up_block) , 2):

            x = self.up_block[layer_num](x)
            skip_conn = down_conv_out[layer_num // 2]

            if x.shape != skip_conn.shape:

                x = resize(x , size = skip_conn.shape[2:])


            x = torch.cat((x , skip_conn) , dim = 1)

            x = self.up_block[layer_num + 1](x)

        x = self.final_layer(x)

        #print(x.shape)

        add_x = resize(add_x , size = x.shape[2:])

        #print(add_x.shape)

        x = torch.cat((x , add_x) , dim = 1)

        return x

## Implementing Residence module ##

class ResiDense(nn.Module):

    def __init__(self , in_dim , out_dim , kernel_size = 3  , stride = 1 , pad
ding = 1):

        super().__init__()

        self.conv_1 = nn.Conv2d(in_channels = in_dim , out_channels = out_dim
, kernel_size = kernel_size ,
                                stride = stride , padding = padding)

        self.conv_2 = nn.Conv2d(in_channels = out_dim , out_channels = out_dim
, kernel_size = kernel_size ,
                                stride = stride , padding = padding)

        self.conv_3 = nn.Conv2d(in_channels = out_dim , out_channels = out_dim
, kernel_size = kernel_size ,
                                stride = stride , padding = padding)

        self.conv_4 = nn.Conv2d(in_channels = out_dim * 2 , out_channels = out
_dim , kernel_size = kernel_size ,
                                stride = stride , padding = padding)

        self.conv_5 = nn.Conv2d(in_channels = out_dim * 4, out_channels = out_
dim , kernel_size = kernel_size ,
                                stride = stride , padding = padding)
```

```
        self.bn1 = nn.BatchNorm2d(num_features = out_dim)
        self.bn2 = nn.BatchNorm2d(num_features = out_dim)
        self.bn3 = nn.BatchNorm2d(num_features = out_dim)
        self.bn4 = nn.BatchNorm2d(num_features = out_dim)
        self.bn5 = nn.BatchNorm2d(num_features = out_dim)

        self.relu = nn.ReLU()

    def forward(self , x):

        out1 = self.conv_1(x)

        out1 = self.bn1(self.relu(out1))

        #print(out1.shape)

        out2 = self.conv_2(out1) + out1

        out2 = self.bn2(self.relu(out2))

        #print(out2.shape)

        out3 = self.conv_3(out2)

        out3 = self.bn3(self.relu(out3))

        out3 = torch.cat((out3 , out2) , dim = 1)

        #print(out3.shape)

        out4 = self.conv_4(out3)

        out4 = self.bn4(self.relu(out4))

        out4 = torch.cat((out4 , out3 , out2) , dim = 1)

        #print(out4.shape)

        out5 = self.conv_5(out4)

        out5 = self.bn5(self.relu(out5))

        out5 = torch.cat((out5 , out4 , out3 , out2) , dim = 1)

        return out5

## Constructing our feature extractor module ##

class FeatureExtractor(nn.Module):

    def __init__(self):

        super().__init__()

        self.d_unet = DerivedUnet()
```

```
        self.residense_1 = ResiDense(in_dim = 2 , out_dim = 64)

        self.conv_1 = nn.Conv2d(in_channels = 512 , out_channels = 64 , kernel
_size = 1)

        self.relu = nn.ReLU()

        self.residense_2 = ResiDense(in_dim = 64 , out_dim = 64)

        self.flatten = nn.Flatten()

    def forward(self , x):

        out = self.d_unet(x)

        out = self.residense_1(out)

        out = self.conv_1(out)

        out = self.relu(out)

        out = self.residense_2(out)

        out = out.mean(dim = (-2 , -1)) ## Global Average Pooling ##

        out = self.flatten(out)

        return out

## Making the classifier ##

class Main_Classier(nn.Module):

    def __init__(self):

        super().__init__()

        self.classifier = nn.Linear(in_features = 512 , out_features = 3)

    def forward(self , x):

        out = self.classifier(x)

        return out

## Making the Confidence model ##

class ConfidNet(nn.Module):

    def __init__(self):

        super().__init__()

        self.confid = nn.Sequential(
            nn.Linear(in_features = 512 , out_features = 256),
            nn.ReLU(),
            nn.Linear(in_features = 256 , out_features = 128),
```

```
            nn.ReLU(),
            nn.Linear(in_features = 128 , out_features = 64),
            nn.ReLU(),
            nn.Linear(in_features = 64 , out_features = 32),
            nn.ReLU(),
            nn.Linear(in_features = 32 , out_features = 1)
        )

    def forward(self , x):

        out = self.confid(x)

        return out

## End to End Classifier ##

class CoWarriorNet(nn.Module):

    def __init__(self):

        super().__init__()

        self.feature_extractor = FeatureExtractor()

        self.classifier = Main_Classier()

    def forward(self , x):

        features = self.feature_extractor(x)

        out = self.classifier(features)

        return features , out
```

Now we can **load** the model.

In [13]:
```
## Loading the model ##

class_model = torch.load('../input/cowarriornet-test/class_model.pth')
conf_model = torch.load('../input/cowarriornet-test/conf_model.pth')
```

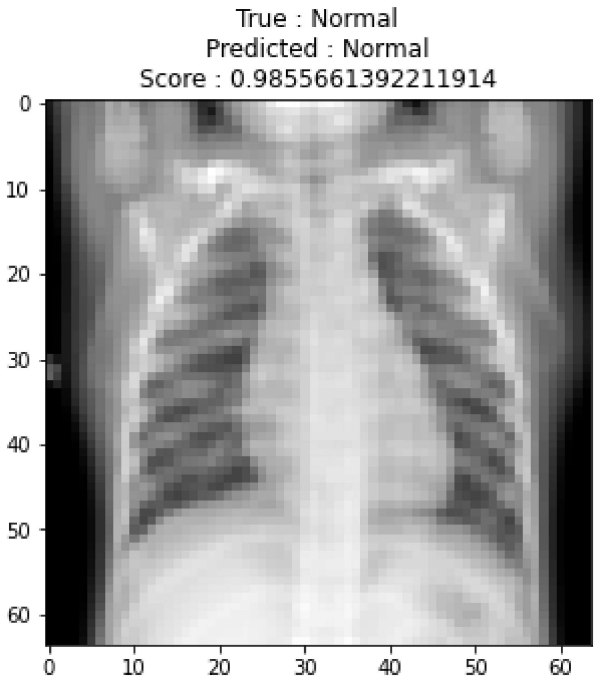
Now we are going to get the predictions of the test data.

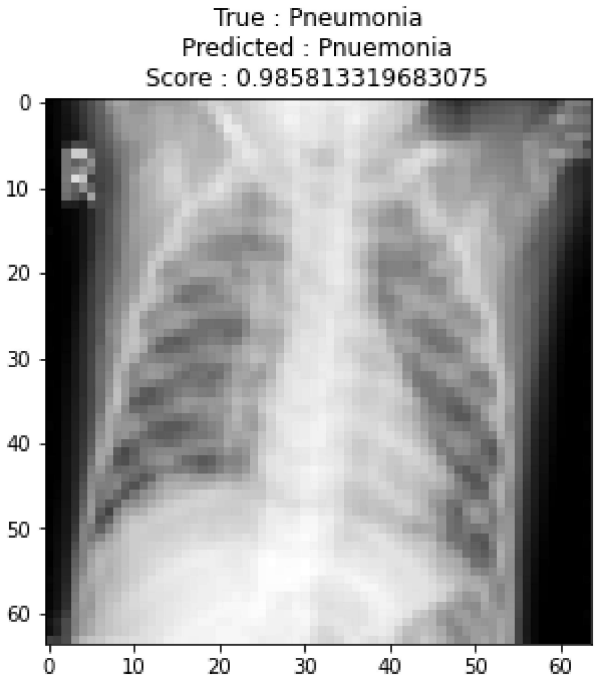
In [14]:
```
## Getting prediction of test data ##

class_model.eval()
conf_model.eval()

class_labels = ['CoVID' , 'Normal' , 'Pneumonia']

i = 1
class_0 = 1
class_1 = 1
class_2 = 1
j = 1
wrong_0 = 1
wrong_1 = 1
wrong_2 = 1

output = []
labels = []
conf_scores = []

loop = tqdm(test_dl)

for idx , (img , label) in enumerate(loop):

    feature , pred = class_model(img)

    out = torch.argmax(pred , dim = 1)

    output.extend(out.tolist())

    labels.extend(label.tolist())

    score = nn.Sigmoid()(conf_model(feature))

    for it in range(len(img)):
        if i<= 3:
            if out[it] == 0 and label[it] == 0 and class_0 <= 1:
                plt.figure(figsize = (5 , 5))
                plt.imshow(img[it].to('cpu').permute(1,2,0) , cmap = 'gray')
                plt.title('True : CoVID\nPredicted : CoVID\nScore : {}'.format
(score[it].to('cpu').item()))
                plt.xticks()
                plt.yticks()
                plt.savefig('./true_covid.jpg', dpi=200)
                plt.show()
                i = i + 1
                class_0 = class_0 + 1

            elif out[it] == 1 and label[it] == 1 and class_1 <= 1:
                plt.figure(figsize = (5 , 5))
                plt.imshow(img[it].to('cpu').permute(1,2,0) , cmap = 'gray')
                plt.title('True : Normal\nPredicted : Normal\nScore : {}'.form
at(score[it].to('cpu').item()))
                plt.xticks()
                plt.yticks()
                plt.savefig('./true_normal.jpg', dpi=200)
```

```
                plt.show()
                i = i + 1
                class_1 = class_1 + 1

            elif out[it] == 2 and label[it] == 2 and class_2 <= 1:
                plt.figure(figsize = (5 , 5))
                plt.imshow(img[it].to('cpu').permute(1,2,0) , cmap = 'gray')
                plt.title('True : Pneumonia\nPredicted : Pnuemonia\nScore : {}
'.format(score[it].to('cpu').item()))
                plt.xticks()
                plt.yticks()
                plt.savefig('./true_pneumonia.jpg', dpi=200)
                plt.show()
                i = i + 1
                class_2 = class_2 + 1

        if j<= 3:
            if out[it] != 0 and label[it] == 0 and wrong_0 <= 1 and score[it].
to('cpu').item() < 0.80:
                plt.figure(figsize = (5 , 5))
                plt.imshow(img[it].to('cpu').permute(1,2,0) , cmap = 'gray')
                plt.title('True : CoVID\nPredicted : {}\nScore : {}'.format(cl
ass_labels[int(out[it].to('cpu').item())] ,
                                                                           sc
ore[it].to('cpu').item()))
                plt.xticks()
                plt.yticks()
                plt.savefig('./wrong_covid.jpg', dpi=200)
                plt.show()
                j = j + 1
                wrong_0 = wrong_0 + 1

            elif out[it] != 1 and label[it] == 1 and wrong_1 <= 1 and score[it
].to('cpu').item() < 0.80:
                plt.figure(figsize = (5 , 5))
                plt.imshow(img[it].to('cpu').permute(1,2,0) , cmap = 'gray')
                plt.title('True : Normal\nPredicted : {}\nScore : {}'.format(c
lass_labels[int(out[it].to('cpu').item())] ,
                                                                            s
core[it].to('cpu').item()))
                plt.xticks()
                plt.yticks()
                plt.savefig('./wrong_normal.jpg', dpi=200)
                plt.show()
                j = j + 1
                wrong_1 = wrong_1 + 1

            elif out[it] != 2 and label[it] == 2 and wrong_2 <= 1 and score[it
].to('cpu').item() < 0.80:
                plt.figure(figsize = (5 , 5))
                plt.imshow(img[it].to('cpu').permute(1,2,0) , cmap = 'gray')
                plt.title('True : Pneumonia\nPredicted : {}\nScore : {}'.forma
t(class_labels[int(out[it].to('cpu').item())] ,

score[it].to('cpu').item()))
                plt.xticks()
                plt.yticks()
```

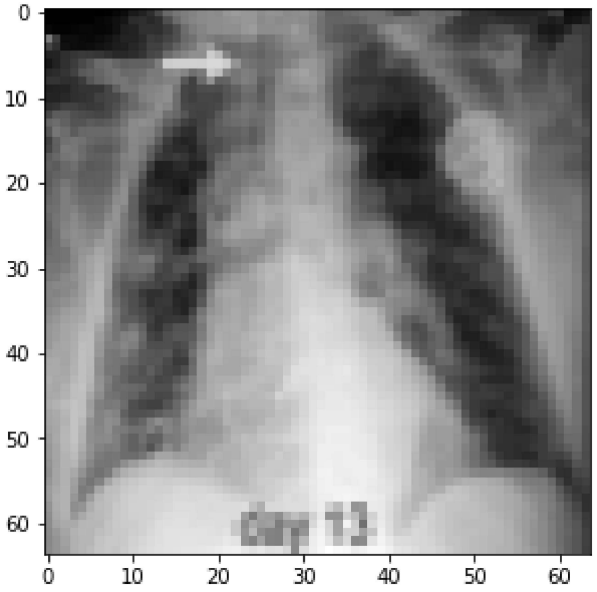
```
                plt.savefig('./wrong_pneumonia.jpg', dpi=200)
                plt.show()
                j = j + 1
                wrong_2 = wrong_2 + 1

    conf_scores.extend(score.tolist())
```

```
0%|          | 0/21 [00:00<?, ?it/s]
```

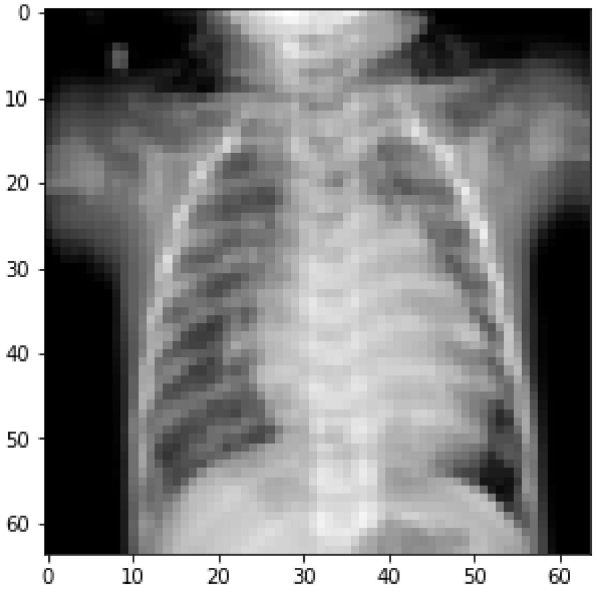




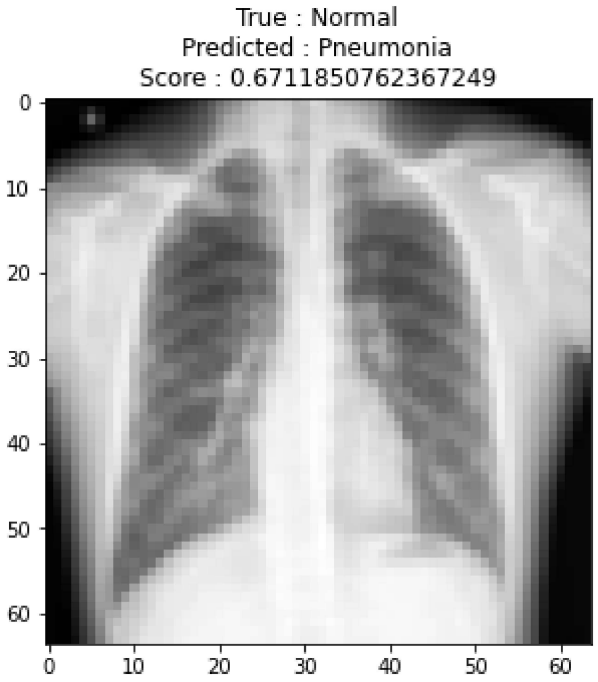
True : CoVID
Predicted : CoVID
Score : 0.9845447540283203



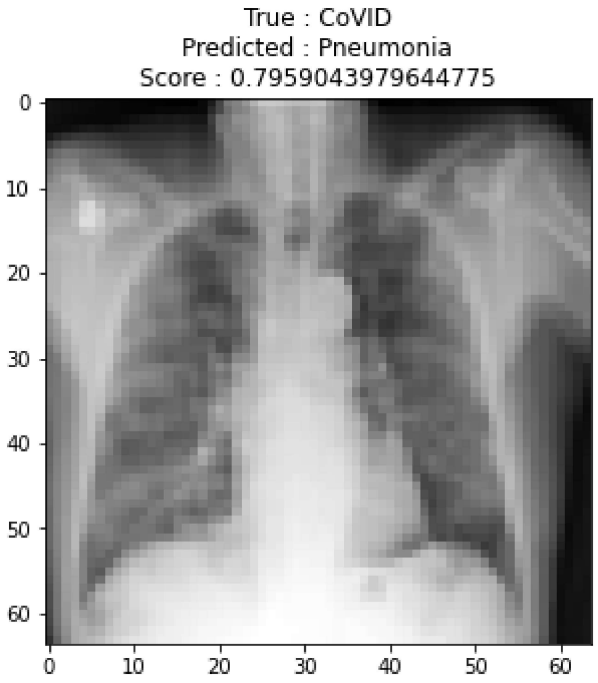
True : Pneumonia
Predicted : Normal
Score : 0.6930248737335205



10%|▉ | 2/21 [00:04<00:39, 2.07s/it]



19%|██ | 4/21 [00:08<00:33, 1.98s/it]



100%|██████████| 21/21 [00:38<00:00, 1.83s/it]

Now lets check the **accuracy, precision, recall and f1 score** of the test set.

But first lets define these metrics.

In [15]:
```
## Accuracy utility function ##

def accuracy(pred , target):
    num = (torch.sum(pred == target)).item()
    den = pred.numel()
    return num / den

## Getting the precision ##

def precision(pred , target):

    covid_tp = 0
    covid_fp = 0

    normal_tp = 0
    normal_fp = 0

    pneumonia_tp = 0
    pneumonia_fp = 0

    for i in range(len(pred)):

        if pred[i] == 0:

            if target[i] == 0:
                covid_tp = covid_tp + 1
            else:
                covid_fp = covid_fp + 1

        elif pred[i] == 1:

            if target[i] == 1:
                normal_tp = normal_tp + 1
            else:
                normal_fp = normal_fp + 1

        else:

            if target[i] == 2:
                pneumonia_tp = pneumonia_tp + 1
            else:
                pneumonia_fp = pneumonia_fp + 1

    covid_precision = (covid_tp) / (covid_tp + covid_fp + 1e-8)

    normal_precision = (normal_tp) / (normal_tp + normal_fp + 1e-8)

    pneumonia_precision = (pneumonia_tp) / (pneumonia_tp + pneumonia_fp + 1e-8
)

    return covid_precision , normal_precision , pneumonia_precision

## Getting the recall ##

def recall(pred , target):
```

```
    covid_tp = 0
    covid_fn = 0

    normal_tp = 0
    normal_fn = 0

    pneumonia_tp = 0
    pneumonia_fn = 0

    for i in range(len(target)):

        if target[i] == 0:

            if pred[i] == 0:
                covid_tp = covid_tp + 1
            else:
                covid_fn = covid_fn + 1

        elif target[i] == 1:

            if pred[i] == 1:
                normal_tp = normal_tp + 1
            else:
                normal_fn = normal_fn + 1

        else:

            if pred[i] == 2:
                pneumonia_tp = pneumonia_tp + 1
            else:
                pneumonia_fn = pneumonia_fn + 1

    covid_recall = (covid_tp) / (covid_tp + covid_fn + 1e-8)

    normal_recall = (normal_tp) / (normal_tp + normal_fn + 1e-8)

    pneumonia_recall = (pneumonia_tp) / (pneumonia_tp + pneumonia_fn + 1e-8)

    return covid_recall , normal_recall , pneumonia_recall


## Getting f1 score ##

def f1_score(covid_precision , normal_precision , pneumonia_precision , covid_
recall , normal_recall , pneumonia_recall):

    covid_f1 = (2 * covid_precision * covid_recall) / (covid_precision + covid
_recall + 1e-8)

    normal_f1 = (2 * normal_precision * normal_recall) / (normal_precision + n
ormal_recall + 1e-8)

    pneumonia_f1 = (2 * pneumonia_precision * pneumonia_recall) / (pneumonia_p
recision + pneumonia_recall + 1e-8)

    return covid_f1 , normal_f1 , pneumonia_f1
```

In [16]:
```
## Checking the scores ##

output_tensor = torch.FloatTensor(output)
label_tensor = torch.FloatTensor(labels)

acc = accuracy(output_tensor , label_tensor)
covid_precision , normal_precision , pneumonia_precision = precision(output_te
nsor , label_tensor)
covid_recall , normal_recall , pneumonia_recall = recall(output_tensor , label
_tensor)
covid_f1 , normal_f1 , pneumonia_f1 = f1_score(covid_precision , normal_precis
ion , pneumonia_precision ,
                                               covid_recall , normal_recall ,
pneumonia_recall)

print('The accuracy is :' , acc)

print('Covid Cases Precision :' , covid_precision)
print('Normal Cases Precision :' , normal_precision)
print('Pneumonia Cases Precision :' , pneumonia_precision)

print('Covid Cases Recall :' , covid_recall)
print('Normal Cases Recall :' , normal_recall)
print('Pneumonia Cases Recall :' , pneumonia_recall)


print('Covid Cases f1 score :' , covid_f1)
print('Normal Cases f1 score :' , normal_f1)
print('Pneumonia Cases f1 score :' , pneumonia_f1)
```

```
The accuracy is : 0.9417701863354038
Covid Cases Precision : 0.9732142856273917
Normal Cases Precision : 0.9430604981870797
Pneumonia Cases Precision : 0.9374301675872913
Covid Cases Recall : 0.9396551723327884
Normal Cases Recall : 0.835962145084039
Pneumonia Cases Recall : 0.9812865496961253
Covid Cases f1 score : 0.95614034579486
Normal Cases f1 score : 0.8862876204065391
Pneumonia Cases f1 score : 0.9588571378487968
```

In [ ]: