Common Test I. Multi-Class Classification

Task: Build a model for classifying the images into lenses using PyTorch or Keras. Pick the most appropriate approach and discuss your strategy.

Dataset: [dataset.zip - Google Drive]

Dataset Description: The Dataset consists of three classes, strong lensing images with no substructure, subhalo substructure, and vortex substructure. The images have been normalized using min-max normalization, but you are free to use any normalization or data augmentation methods to improve your results.

Evaluation Metrics: ROC curve (Receiver Operating Characteristic curve) and AUC score (Area Under the ROC Curve)

Importing required libraries

```
#Utilities
import os
import gc
import glob
import numpy as np
import pandas as pd
from tqdm.notebook import tqdm

#Loading image and plotting visualizations/images
from PIL import Image
import seaborn as sns
import matplotlib.pyplot as plt

#PyTorch framework
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import CosineAnnealingWarmRestarts
from torch.utils.data import DataLoader, Dataset
from torchvision import utils

#Evaluation metrics
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve, auc

#For pre-trained model
import sys
sys.path.append('../input/timm-pytorch-image-models/pytorch-image-models-master')
import timm

np.random.seed(7)
torch.manual_seed(7)

device='cuda' if torch.cuda.is_available() else 'cpu'
device
```

```
'cuda'
```

Creating a custom dataset class

```
class CustomDataset(Dataset):
    def __init__(self, root_dir, transform = None):
        root_list = glob.glob(root_dir)
        self.class_map = {}
        self.class_distribution = {}
        self.transform = transform

        for img_path in root_list:
            class_name = img_path.split(os.sep)[-2]
            if class_name not in self.class_distribution:
```

```
                self.class_distribution[class_name] = 1
            else:
                self.class_distribution[class_name] +=1

        for index, entity in enumerate(self.class_distribution):
            self.class_map[entity] = index
        print("Dataset Distribution:\n")
        print(self.class_distribution)
        print("\n\nClass indices:\n")
        print(self.class_map)

        self.data = []
        for img_path in tqdm(root_list):
            class_name = img_path.split(os.sep)[-2]
            self.data.append([img_path, class_name])

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_path, class_name = self.data[idx]
        img = np.load(img_path)
        img = torch.tensor(img, dtype=torch.float)

        class_id = self.class_map[class_name]
        class_id = torch.tensor(class_id)

        return img, class_id
```

```
#Using a batch size of 128
BS = 128
```

```
train_path = r'../input/ml4sci-deeplense-commontask/dataset/train/*/*'
train_dataset = CustomDataset(train_path)

val_path = r'../input/ml4sci-deeplense-commontask/dataset/val/*/*'
```

```
val_dataset = CustomDataset(val_path)

print(len(train_dataset), len(val_dataset))
```

```
Dataset Distribution:

{'no': 10000, 'vort': 10000, 'sphere': 10000}


Class indices:

{'no': 0, 'vort': 1, 'sphere': 2}
  0%|          | 0/30000 [00:00<?, ?it/s]
Dataset Distribution:

{'no': 2500, 'vort': 2500, 'sphere': 2500}


Class indices:

{'no': 0, 'vort': 1, 'sphere': 2}
  0%|          | 0/7500 [00:00<?, ?it/s]
30000 7500
```

Creating data loaders separately for train data and val/test data

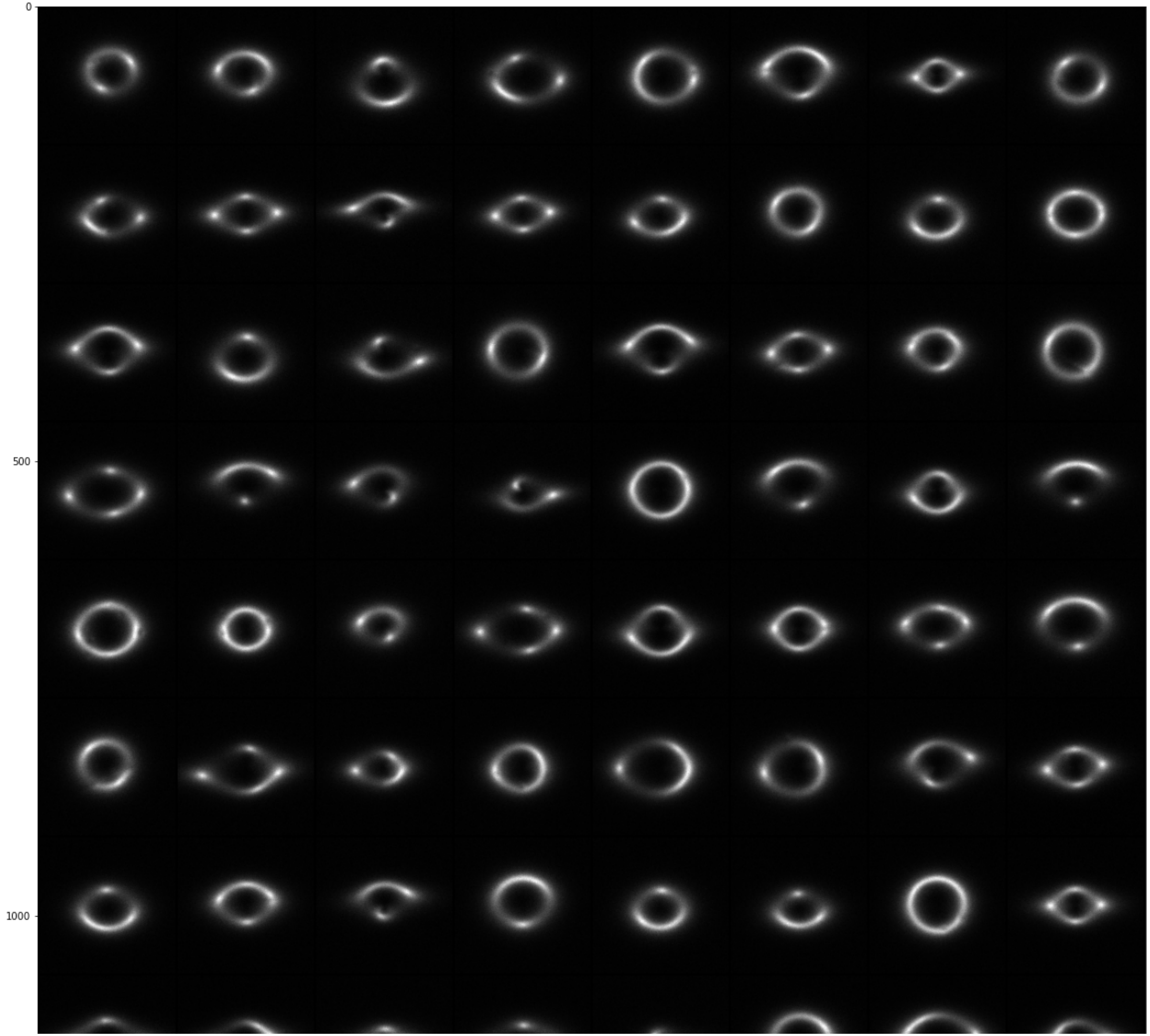
```
train_loader = DataLoader(train_dataset, batch_size = BS, shuffle = True)
val_loader = DataLoader(val_dataset, batch_size = BS, shuffle = False)
```

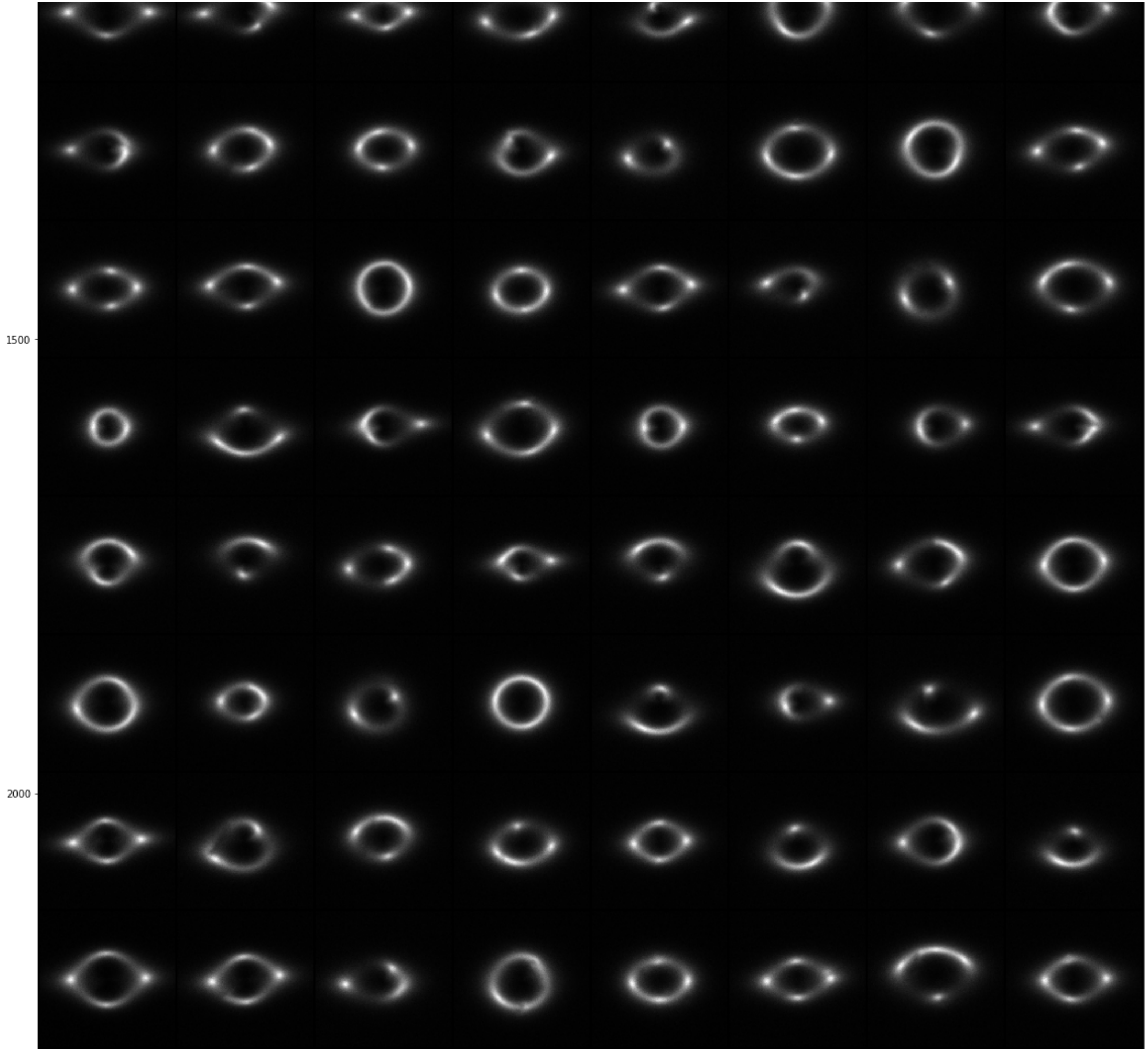
```
single_batch = next(iter(train_loader))
print(f"The dimensions of a single batch is {single_batch[0].shape}")
```

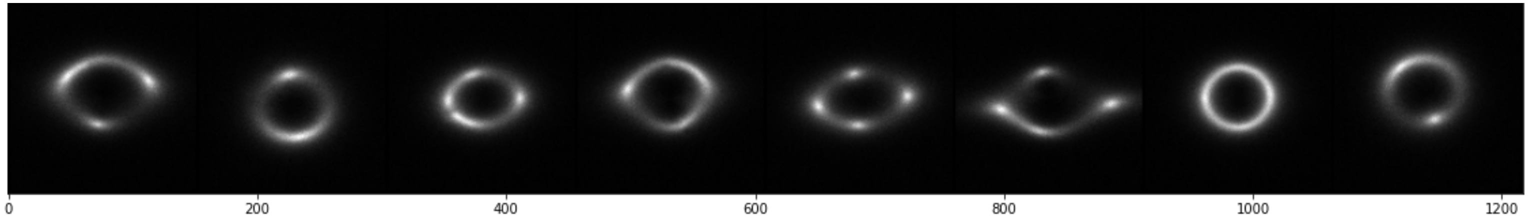
```
The dimensions of a single batch is torch.Size([128, 1, 150, 150])
```

Plotting a shuffled single batch from the train loader

```
single_batch_grid = utils.make_grid(single_batch[0], nrow=8)
plt.figure(figsize = (20,70))
plt.imshow(single_batch_grid.permute(1, 2, 0))
plt.savefig("Single_batch_of_train_loader.png")
```


0
500
1000


1500
2000



Creating model

Using the Inception Resnet V2 pre-trained model as the backbone, adding my own classifier head and fine-tuning to this dataset

```
class pre_trained_model(nn.Module):

    def __init__(self, pretrained = True):
        super().__init__()
        self.model = timm.create_model('inception_resnet_v2',pretrained = pretrained, in_chans = 1)
#         num_in_features = self.model.get_classifier().in_features

        for param in self.model.parameters():
            param.requires_grad = True

        self.fc = nn.Sequential(
                                nn.Linear(1536 * 3 * 3, 1024),
                                nn.PReLU(),
                                nn.BatchNorm1d(1024),
                                nn.Dropout(p = 0.5),
```

```

                                  nn.Linear(1024, 128),
                                  nn.PReLU(),
                                  nn.BatchNorm1d(128),
                                  nn.Dropout(p = 0.5),

                                  nn.Linear(128, 3)
                                  )

    def forward(self, x):
        x = self.model.forward_features(x)
        x = x.view(-1, 1536 * 3 * 3)
        x = self.fc(x)
        return x
```

```
model = pre_trained_model()

#Verifying output of model

x = torch.randn(128, 1, 150, 150)
model(x).shape
```

```
Downloading: "https://github.com/rwightman/pytorch-image-models/releases/download/v0.1-weights/inception_resnet_v2-940b1cd6.pth
torch.Size([128, 3])
```

```
def calculate_accuracy(y_pred, y_truth):
    y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
    _, y_pred_labels = torch.max(y_pred_softmax, dim = 1)

    correct_preds = (y_pred_labels == y_truth).float()
    acc = correct_preds.sum() / len(correct_preds)
    acc = torch.round(acc*100)

    return acc
```

```
def train_epoch(model, dataloader, criterion, optimizer):
    model.train()
    train_loss = []
    train_accuracy = []

    loop=tqdm(enumerate(dataloader),total = len(dataloader))

    for batch_idx, (img_batch,labels) in loop:

        X = img_batch.to(device)
        y_truth = labels.to(device)

        #forward prop
        y_pred = model(X)

        #loss and accuracy calculation
        loss = criterion(y_pred, y_truth)
        accuracy = calculate_accuracy(y_pred, y_truth)

        #backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        #batch loss and accuracy
        # print(f'Partial train loss: {loss.data}')
        train_loss.append(loss.detach().cpu().numpy())
        train_accuracy.append(accuracy.detach().cpu().numpy())

    return model, np.mean(train_loss), np.mean(train_accuracy)
```

```
def val_epoch(model, dataloader,criterion):
    model.eval()
    val_loss = []
    val_accuracy = []

```

```
    with torch.no_grad():

        loop=tqdm(enumerate(dataloader),total=len(dataloader))

        for batch_idx, (img_batch,labels) in loop:
            X = img_batch.to(device)
            y_truth = labels.to(device)

            #forward prop
            y_pred = model(X)

            #loss and accuracy calculation
            loss = criterion(y_pred, y_truth)
            accuracy = calculate_accuracy(y_pred, y_truth)


            #batch loss and accuracy
            # print(f'Partial train loss: {loss.data}')
            val_loss.append(loss.detach().cpu().numpy())
            val_accuracy.append(accuracy.detach().cpu().numpy())

    return np.mean(val_loss), np.mean(val_accuracy)
```

```
def fit_model(model,criterion,optimizer):
    loss_dict = {'train_loss':[],'val_loss':[]}
    acc_dict = {'train_accuracy':[],'val_accuracy':[]}

    for epoch in range(EPOCHS):
        print(f"Epoch {epoch+1}/{EPOCHS}:")
        model, train_loss, train_accuracy = train_epoch(model, train_loader, criterion, optimizer)
        val_loss, val_accuracy = val_epoch(model, val_loader, criterion)

        print(f'Train loss:{train_loss}, Val loss:{val_loss}')
        loss_dict['train_loss'].append(train_loss)
        loss_dict['val_loss'].append(val_loss)
        print(f'Train accuracy: {train_accuracy}, Val accuracy:{val_accuracy}')
        acc_dict['train_accuracy'].append(train_accuracy)
```

```
        acc_dict['val_accuracy'].append(val_accuracy)


    return model, loss_dict, acc_dict
```

Initializing the model and deciding the hyperparameter values

Multi-class classification problem -> Loss function : CrossEntropyLoss

Model trained for 10 epochs

Adam Optimizer used with learning rate 3e-4

```
model = pre_trained_model().to(device)

criterion = nn.CrossEntropyLoss()
EPOCHS = 10
LR = 3e-4

optimizer = optim.Adam(model.parameters(),lr=LR)
```

```
#Training model
model, loss_dict, acc_dict = fit_model(model,criterion,optimizer)
```

```
Epoch 1/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.9953879714012146, Val loss:0.9294667840003967
Train accuracy: 51.11489486694336, Val accuracy:58.779659271240234
Epoch 2/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.5411764979362488, Val loss:0.45875853300094604
Train accuracy: 77.87659454345703, Val accuracy:81.66101837158203
Epoch 3/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.35474199056625366, Val loss:0.3722762167453766
Train accuracy: 86.22553253173828, Val accuracy:85.30508422851562
Epoch 4/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.2617902457714081, Val loss:0.36338678002357483
Train accuracy: 90.2170181274414, Val accuracy:87.40677642822266
Epoch 5/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.19792300462722778, Val loss:0.4865666925907135
Train accuracy: 92.77021026611328, Val accuracy:80.16949462890625
Epoch 6/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.1542099267244339, Val loss:0.3650676906108856
Train accuracy: 94.31063842773438, Val accuracy:88.5762710571289
Epoch 7/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.11358343064785004, Val loss:0.38710132241249084
Train accuracy: 95.94893646240234, Val accuracy:86.81356048583984
Epoch 8/10:
  0%|          | 0/235 [00:00<?, ?it/s]
  0%|          | 0/59 [00:00<?, ?it/s]
Train loss:0.09248575568199158, Val loss:0.39652055501937866
```

```
#saving the trained model
```

```
PATH = "inception_resnetV2_finetuned.pth"
torch.save(model.state_dict(), PATH)
```

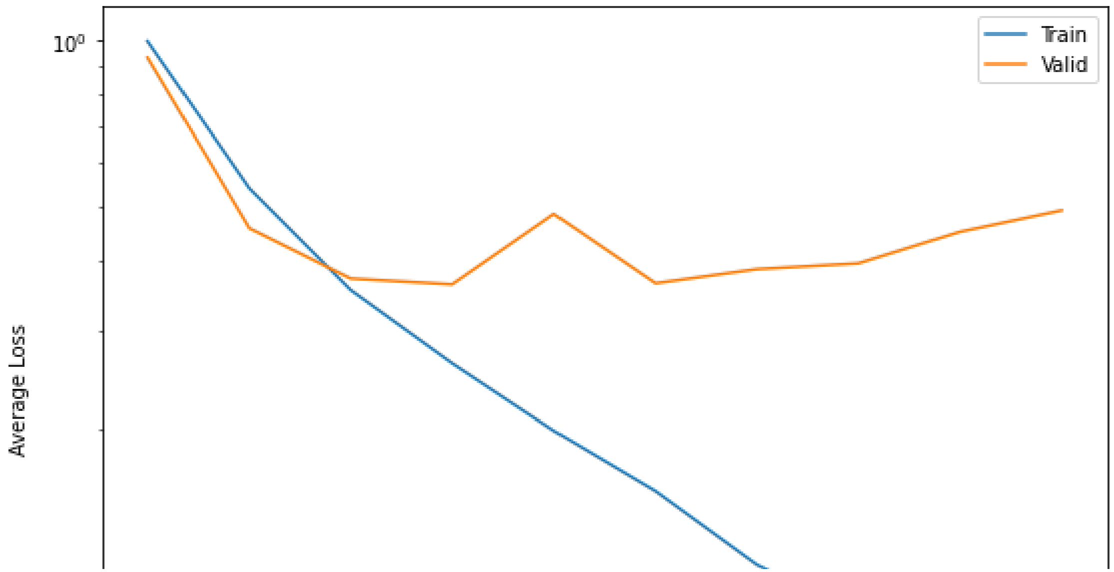
```
#deleting the trained model instance, creating a new one and loading the saved train model
del model
gc.collect()

model = pre_trained_model().to(device)
model.load_state_dict(torch.load(PATH))
```

```
<All keys matched successfully>
```

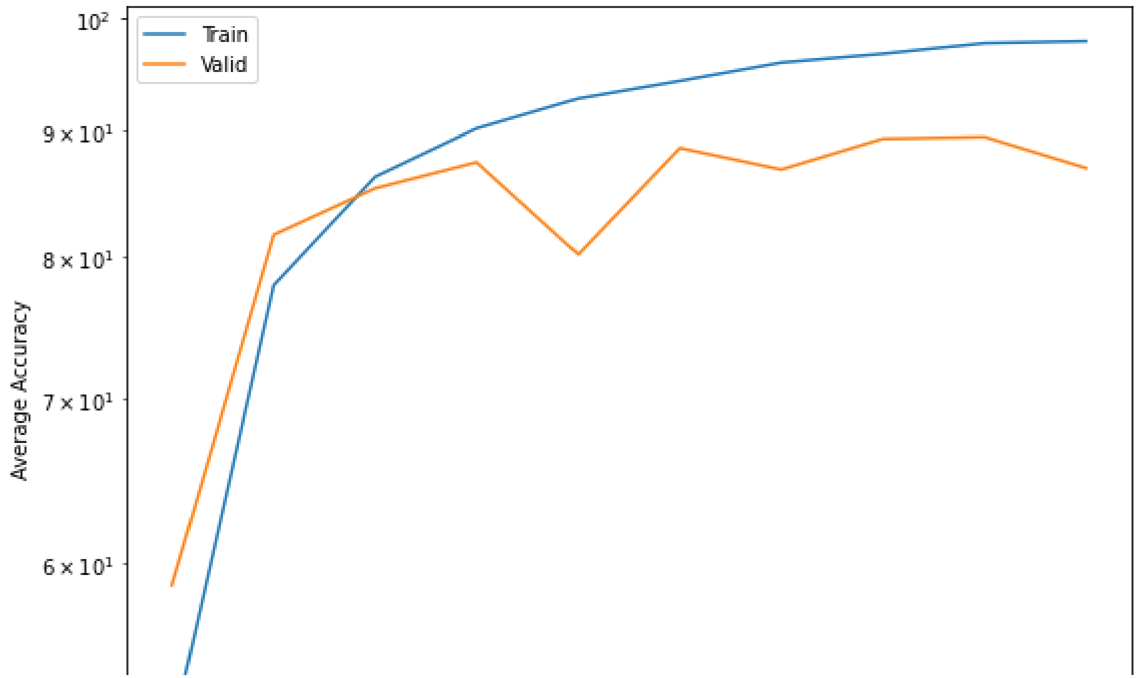
Loss through the epochs

```
# Plot losses
plt.figure(figsize=(9,7))
plt.semilogy(loss_dict['train_loss'], label='Train')
plt.semilogy(loss_dict['val_loss'], label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
#plt.grid()
plt.legend()
#plt.title('loss')
plt.show()
plt.savefig("Loss_history.png")
```



Val/test accuracy through the epochs

```
# Plot accuracy
plt.figure(figsize=(9,7))
plt.semilogy(acc_dict['train_accuracy'], label='Train')
plt.semilogy(acc_dict['val_accuracy'], label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Average Accuracy')
#plt.grid()
plt.legend()
#plt.title('loss')
plt.show()
plt.savefig("Accuracy_history.png")
```



Final prediction

(Usually this is done on another separate test set. In this case, I have considered the test and val sets to be same)

```
<Figure size 432x288 with 0 Axes>
```

```
def test_epoch(model, dataloader,criterion):

    model.eval()
    test_loss = []
    test_accuracy = []

    y_pred_list = []
    y_truth_list = []
    y_pred_prob_list= []

    with torch.no_grad():

        loop=tqdm(enumerate(dataloader),total=len(dataloader))
```

```
        for batch_idx, (img_batch,labels) in loop:
            X = img_batch.to(device)
            y_truth = labels.to(device)
            y_truth_list.append(y_truth.detach().cpu().numpy())

            #forward prop
            y_pred = model(X)
            y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
            y_pred_prob_list.append(y_pred_softmax.detach().cpu().numpy())
            _, y_pred_labels = torch.max(y_pred_softmax, dim = 1)
            y_pred_list.append(y_pred_labels.detach().cpu().numpy())

            #loss and accuracy calculation
            loss = criterion(y_pred, y_truth)
            accuracy = calculate_accuracy(y_pred, y_truth)


            #batch loss and accuracy
            # print(f'Partial train loss: {loss.data}')
            test_loss.append(loss.detach().cpu().numpy())
            test_accuracy.append(accuracy.detach().cpu().numpy())

    return y_pred_prob_list, y_pred_list, y_truth_list, np.mean(test_loss), np.mean(test_accuracy)
```

```
y_pred_prob_list, y_pred_list, y_truth_list, test_loss, test_accuracy = test_epoch(model, val_loader, criterion)

print(test_loss, test_accuracy)
```

```
  0%|          | 0/59 [00:00<?, ?it/s]
0.49393848 86.932205
```

An accuracy of 89.86% has been achieved on the test set

```
#To flatten the outputs since the predictions are generated in batches w.r.t the data loader
```

```

def flatten_list(x):
    flattened_list = []
    for i in x:
        for j in i:
            flattened_list.append(j)

    return flattened_list
```

```
y_pred_list_flattened = flatten_list(y_pred_list)
y_truth_list_flattened = flatten_list(y_truth_list)
y_pred_prob_list_flattened = flatten_list(y_pred_prob_list)
```

```
idx2class = {v: k for k, v in train_dataset.class_map.items()}
class_names = [i for i in train_dataset.class_map.keys()]
idx2class
```

```
{0: 'no', 1: 'vort', 2: 'sphere'}
```

Classification Report on the test set

```
print(classification_report(y_truth_list_flattened, y_pred_list_flattened,target_names = class_names))
```

```
              precision    recall  f1-score   support

          no       0.89      0.87      0.88      2500
        vort       0.92      0.85      0.88      2500
      sphere       0.81      0.89      0.85      2500

    accuracy                           0.87      7500
   macro avg       0.87      0.87      0.87      7500
weighted avg       0.87      0.87      0.87      7500
```

```
print(confusion_matrix(y_pred_list_flattened, y_truth_list_flattened))
```

```
[[2179   92  172]
 [  77 2117  102]
 [ 244  291 2226]]
```

Confusion Matrix from test set predictions

```
confusion_matrix_df = pd.DataFrame(confusion_matrix(y_truth_list_flattened, y_pred_list_flattened)).rename(columns=idx2class, inde
fig, ax = plt.subplots(figsize=(19,12))
sns.heatmap(confusion_matrix_df, fmt = ".0f", annot=True, ax=ax)
plt.savefig("Confusion_matrix.png")
```

Plotting ROC curve

```
#One-hot-encoding ground truths
temp_test_y = []

for i in range(len(y_truth_list_flattened)):
    a = [0, 0, 0]
    a[y_truth_list_flattened[i]] = 1
    temp_test_y.append(a)

temp_test_y = np.array(temp_test_y)
temp_test_y.shape
```

```
(7500, 3)
```

```
#Note: The test set was not shuffled
temp_test_y[0:5]
```

```
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
```

```
       [1, 0, 0],
       [1, 0, 0]])
```

```
y_pred_prob_list_flattened = np.array(y_pred_prob_list_flattened)
y_pred_prob_list_flattened.shape
```

```
(7500, 3)
```

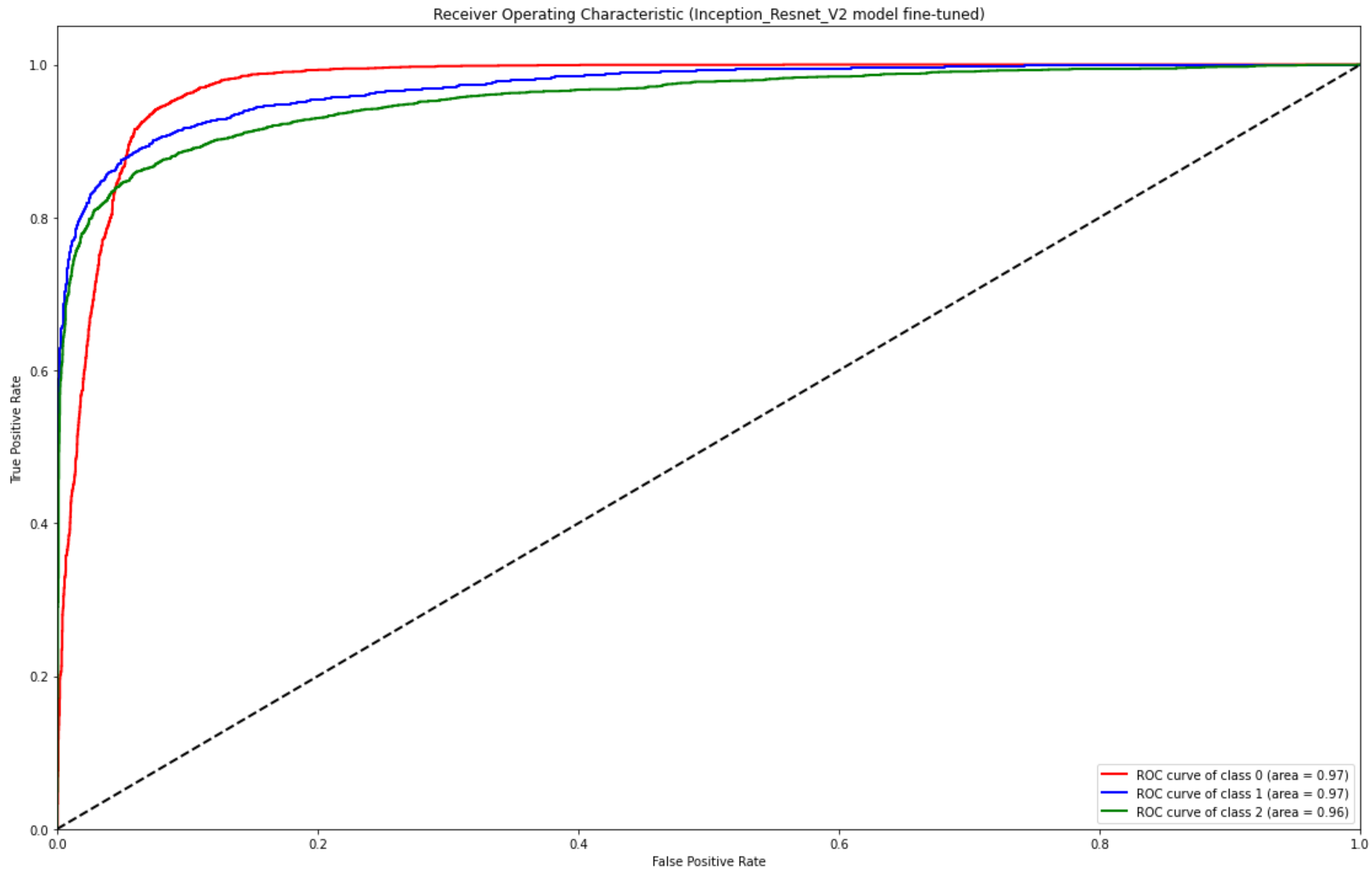
```
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(3):
    fpr[i], tpr[i], _ = roc_curve(temp_test_y[:, i], y_pred_prob_list_flattened[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
```

```
colors = ['red', 'blue', 'green']
plt.figure(figsize = (19, 12))

for i, color in zip(range(3), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2, label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (Inception_Resnet_V2 model fine-tuned)')
plt.legend(loc="lower right")
plt.show()
plt.savefig("ROC_curve.png")
```




```
<Figure size 432x288 with 0 Axes>
```

ROC-AUC score