```
from google.colab import drive
```

```
drive.mount("/content/MyDrive")
```

```
from torchvision.models import resnet50, ResNet50_Weights
import torch.nn as nn
```

```
model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)
```

```
import torch
from torchvision import datasets, transforms
import torchvision
```

```
import matplotlib.pyplot as plt
import numpy as np
import random
```

```
torch.manual_seed(0)
random.seed(0)
np.random.seed(0)
```

```
transform_train = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

```
dataset = datasets.ImageFolder('/content/MyDrive/MyDrive/Marine-Dataset/SOUVIK/train', transform = transform_train)
```

```
dataset.classes
```

```
def class_count(dataset):
    class_names = ['immature', 'mature']
    class_counts = {class_name: 0 for class_name in class_names}

    for _, label in dataset:
        class_counts[class_names[label]] += 1

    for class_name, count in class_counts.items():
        print(f"Class {class_name}: {count} images")
```

```
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, test_size])
```

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=16, shuffle=False)
```

```
classes = ['no-plastic', 'plastic']
```

```
dataiter = iter(train_loader)
images, labels = next(dataiter)
```

```
print(labels)
print(labels.shape)
print(labels.dtype)
```

```
tensor([1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1])
torch.Size([16])
torch.int64
```

```
def unnormalize_img(np_image):
    mean = np.array([0.4914, 0.4822, 0.4465])
    std = np.array([0.2023, 0.1994, 0.2010])
    npimg = np.transpose(np_image, (1, 2, 0))
```

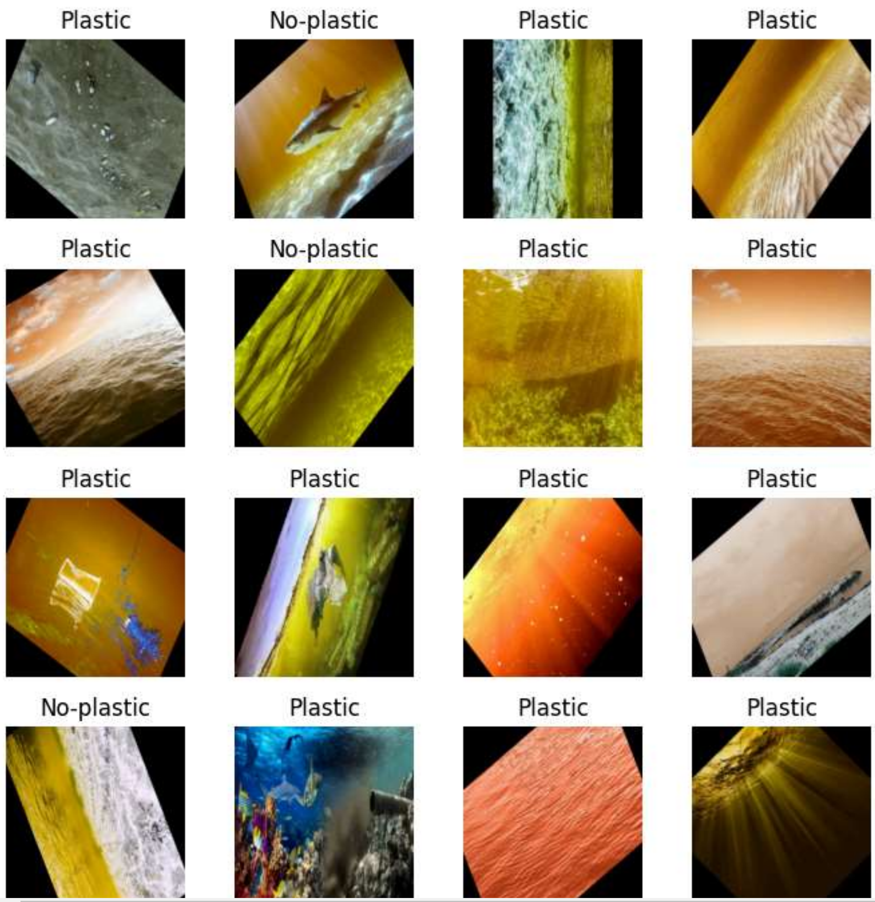
```
    npimg = (npimg * std) + mean

    return npimg
```

```
def imshow(img, title=None):
    npimg = img.numpy()

    npimg = unnormalize_img(npimg)

    plt.imshow(npimg)

    if title is not None:
        plt.title(title)
    plt.show()
```

```
def display_images_in_grid(image_list, labels,grid_size=(3, 3)):
    num_images = len(image_list)
    num_rows, num_cols = grid_size

    if num_images < num_rows * num_cols:
        print(f"Warning: Not enough images to fill the {num_rows}x{num_cols} grid.")
        return

    f, axarr = plt.subplots(num_rows, num_cols, figsize=(7, 7))

    for i in range(num_rows):
        for j in range(num_cols):
            img_idx = i * num_cols + j
            axarr[i, j].imshow(unnormalize_img(image_list[img_idx].numpy()))
            axarr[i, j].axis("off")

            axarr[i, j].set_title(f"{classes[labels[img_idx]].capitalize()}")

    plt.tight_layout()
    plt.show()
```

```
display_images_in_grid(images, labels, (4,4))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



ResNet 50

```
from torchvision.models import resnet50, ResNet50_Weights
import torch.nn as nn
```

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

```
for params in model.parameters():
    params.requires_grad_ = False
```

```
last_layer_in_features = model.fc.in_features
```

```
model.fc = nn.Sequential(
            nn.Linear(last_layer_in_features, 1),
            nn.Sigmoid()
)
```

```
model.to(device)
```

```
      (4): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (5): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Sequential(
      (0): Linear(in_features=2048, out_features=1, bias=True)
      (1): Sigmoid()
    )
  )
```

```
learning_rate = 0.01
```

```
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate, momentum=0.9,weight_decay=5e-4)
```

```
import copy

def train_model(model, train_data, optimizer, criterion, epochs=12, val_data=None, early_stopping=False, early_stopping_patience=5):
    """
        Reusable function to train a pytorch model.

        Input:
            model: PyTorch Model
            train_data: DataSet Loader with Train Data
            epochs: (default = 12) Number of epochs the model should be trained
            val_data: (Optional) DataSet Loader with Validation Data to perform the model on unseen data
            early_stopping: (default = False) Whether the training should stop early to avoid overfitting
            early_stopping_patience: (default = 5) Patience value for early stopping

        Returns:
            model: PyTorch model trained on the given data
```

```
        history: History of the values containing, Train loss, Train Accuracy, Val Loss and Val Accuracy (if validation data provided)

    """

    if val_data is None and early_stopping is True:
        raise ValueError("Early stopping is done based on the models performance on validation data, so inorder to perform early stopping, pa


    train_loader = train_data
    val_loader = val_data

    best_loss = float('inf')
    best_model_weights = None
    best_model_weights = copy.deepcopy(model.state_dict())
    history = {}
    history['train_loss'] = []
    history['val_loss'] = []
    history['train_accuracy'] = []
    history['val_accuracy'] = []

    for epoch in range(epochs):

        model.train() # model in train mode

        train_losses = []
        train_correct = 0
        train_total = 0

        for batch_num, input_data in enumerate(train_loader):
            optimizer.zero_grad()
            x, y = input_data
            x = x.to(device).float()

            y = y.unsqueeze(1).float()
            y = y.to(device)

            output = model(x)

            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            train_losses.append(loss.item())

            predicted = (output >= 0.5).float()

            train_total += y.size(0)
            train_correct += (predicted == y).sum().item()

            if batch_num % 500 == 0:
                print('\tEpoch %d | Batch %d | Loss %6.2f' % (epoch+1, batch_num, loss.item()))


        train_avg_loss = sum(train_losses) / len(train_losses)
        train_accuracy = 100 * train_correct / train_total
        print('Epoch %d | Training Loss %6.2f | Training Accuracy: %2.2f %%' % (epoch+1, train_avg_loss, train_accuracy))
        history['train_loss'].append(train_avg_loss)
        history['train_accuracy'].append(train_accuracy)

        if val_loader is not None:
            # Validation phase
            model.eval()

            val_losses = []
            val_correct = 0
            val_total = 0

            with torch.no_grad():
                for batch_num, val_data in enumerate(val_loader):
                    x, y = val_data
                    x = x.to(device).float()

                    y = y.unsqueeze(1).float()

                    y = y.to(device)

                    val_output = model(x)
                    val_loss = criterion(val_output, y)
```

```
                val_losses.append(val_loss.item())

                val_predicted = (val_output >= 0.5).float()

                val_total += y.size(0)
                val_correct += (val_predicted == y).sum().item()

        val_avg_loss = sum(val_losses) / len(val_losses)
        val_accuracy = 100 * val_correct / val_total
        print('Epoch %d | Validation Loss %6.2f | Validation Accuracy: %2.2f %%' % (epoch+1, val_avg_loss, val_accuracy))
        history['val_loss'].append(val_avg_loss)
        history['val_accuracy'].append(val_accuracy)

    if early_stopping is not False:
        # Early stopping
        if val_avg_loss < best_loss:
            best_loss = val_avg_loss
            best_model_weights = copy.deepcopy(model.state_dict())  # Deep copy here
            patience = early_stopping_patience  # Reset patience counter
            print(f"Saving the best model at {epoch+1}th epoch.")
        else:
            patience -= 1
```