```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tr
```

```
import os
import time
import torch
import torch.nn as nn
import torchvision
from torchvision import datasets
from PIL import Image
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader

import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt


class CustomImageDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_paths = [os.path.join(root_dir, fname) for fname in os.listdir(root_dir)]

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

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        image = Image.open(img_path)

        if self.transform:
            image = self.transform(image)

        return image


# Define your data transformation
train_transform = transforms.Compose([
    transforms.Resize((100, 100)),
    transforms.RandomHorizontalFlip(p=0.2),
    transforms.RandomVerticalFlip(p=0.2),
    # transforms.GaussianBlur(kernel_size=(5, 9), sigma=(0.1, 5)),
    transforms.RandomRotation(degrees=(5, 15)),
    transforms.ToTensor(),
    # transforms.Normalize(
    #     mean=[0.5, 0.5, 0.5],
    #     std=[0.5, 0.5, 0.5]
    # )
])

test_transform = transforms.Compose([
    transforms.Resize((100, 100)),
    transforms.ToTensor(),
])



batch_size = 16

# Load data
train_dataset = CustomImageDataset(root_dir='/content/drive/MyDrive/AE_xray/train', transform=train_transform)
test_dataset = CustomImageDataset(root_dir='/content/drive/MyDrive/AE_xray/test', transform=test_transform)

# Dataloader
train_dl = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dl = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)


class Encoder(nn.Module):
  def __init__(self , input_size = 10000 , hidden_size1 = 2500, hidden_size2 = 1000 , hidden_size3 = 500, hidden_size4 = 200, z_d
    super().__init__()
    self.fc1 = nn.Linear(input_size , hidden_size1)
```

```
    self.fc2 = nn.Linear(hidden_size1 , hidden_size2)
    self.fc3 = nn.Linear(hidden_size2 , hidden_size3)
    self.fc4 = nn.Linear(hidden_size3 , hidden_size4)
    self.fc5 = nn.Linear(hidden_size4 , z_dim)
    self.relu = nn.ReLU()
  def forward(self , x):
    x = self.relu(self.fc1(x))
    x = self.relu(self.fc2(x))
    x = self.relu(self.fc3(x))
    x = self.relu(self.fc4(x))
    x = self.fc5(x)
    return x


class Decoder(nn.Module):
  def __init__(self , output_size = 10000 , hidden_size1 = 2500, hidden_size2 = 1000 , hidden_size3 = 500, hidden_size4 = 200, z_
    super().__init__()
    self.fc1 = nn.Linear(z_dim , hidden_size4)
    self.fc2 = nn.Linear(hidden_size4 , hidden_size3)
    self.fc3 = nn.Linear(hidden_size3 , hidden_size2)
    self.fc4 = nn.Linear(hidden_size2 , hidden_size1)
    self.fc5 = nn.Linear(hidden_size1 , output_size)
    self.relu = nn.ReLU()
  def forward(self , x):
    x = self.relu(self.fc1(x))
    x = self.relu(self.fc2(x))
    x = self.relu(self.fc3(x))
    x = self.relu(self.fc4(x))
    x = torch.sigmoid(self.fc5(x))
    return x


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

```
device(type='cuda')
```

```
enc = Encoder().to(device)
dec = Decoder().to(device)


loss_fn = nn.MSELoss()
optimizer_enc = torch.optim.Adam(enc.parameters())
optimizer_dec = torch.optim.Adam(dec.parameters())


train_loss = []
num_epochs = 300
checkpoint_path = '/content/drive/MyDrive/model/Autoencoder/2.aug_xray_checkpoint_30z_5h_200e.pth'


# Check if a checkpoint exists to resume training
if os.path.exists(checkpoint_path):
  checkpoint = torch.load(checkpoint_path)
  enc.load_state_dict(checkpoint["enc_state_dict"])
  dec.load_state_dict(checkpoint["dec_state_dict"])
  optimizer_enc.load_state_dict(checkpoint["optimizer_enc_state_dict"])
  optimizer_dec.load_state_dict(checkpoint["optimizer_dec_state_dict"])
  train_loss = checkpoint["loss"]
  start_epoch = checkpoint["epoch"] + 1  # Start from the next epoch after the loaded checkpoint
  print("Resume training from epoch", start_epoch)
else:
  start_epoch = 1
```

```
Resume training from epoch 275
```

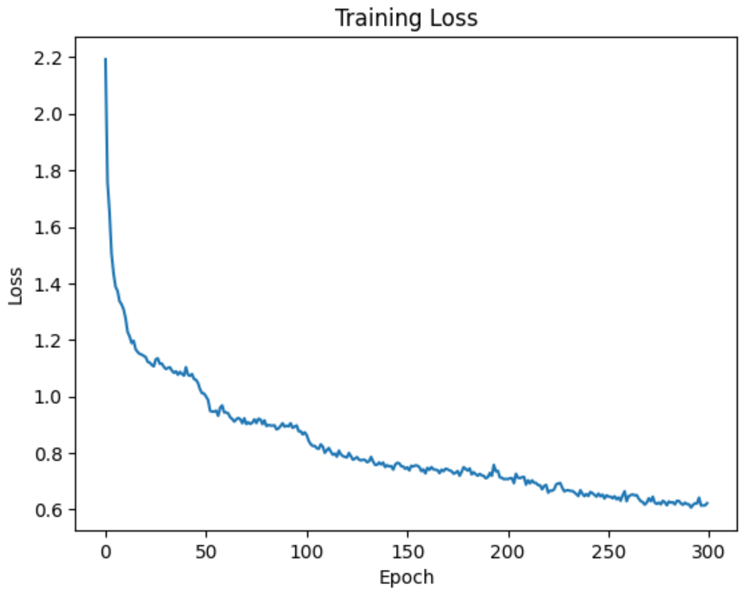
```
total_batches = len(train_dl)
for epoch in range(start_epoch,num_epochs+1):
    train_epoch_loss = 0
    start_time = time.time()
    # Create a tqdm progress bar for the epoch
    epoch_progress = tqdm(enumerate(train_dl, 1), total=total_batches, desc=f'Epoch {epoch}/{num_epochs}', leave=False)
    for step, imgs in epoch_progress:
        imgs = imgs.to(device)
        imgs = imgs.flatten(1)
        # print(imgs.shape)
        latents = enc(imgs)
        output = dec(latents)
```

```
            loss = loss_fn(output, imgs)
            train_epoch_loss += loss.item()
            optimizer_enc.zero_grad()
            optimizer_dec.zero_grad()
            loss.backward()
            optimizer_enc.step()
            optimizer_dec.step()
            # Update the progress bar description with current step and loss
            epoch_progress.set_description(f'Epoch {epoch}/{num_epochs}, Step {step}/{total_batches}, Loss: {loss.item():.4f}')
        train_loss.append(train_epoch_loss)
        # Close the tqdm progress bar for the epoch
        epoch_progress.close()

        # Print the epoch loss after each epoch
        print('\n')
        print(f'Epoch {epoch}/{num_epochs}, Loss: {train_epoch_loss:.4f}, Time taken: [{time.time() - start_time:.2f}s]')

        # Save the model checkpoint along with training-related information
        checkpoint = {
            'epoch': epoch,
            'enc_state_dict': enc.state_dict(),  # Save the encoder model's state dictionary
            'dec_state_dict':dec.state_dict(),
            'optimizer_enc_state_dict': optimizer_enc.state_dict(),  # Save the optimizer state
            'optimizer_dec_state_dict': optimizer_dec.state_dict(),
            'loss': train_loss,  # Save the loss
        }
        torch.save(checkpoint, checkpoint_path)
```

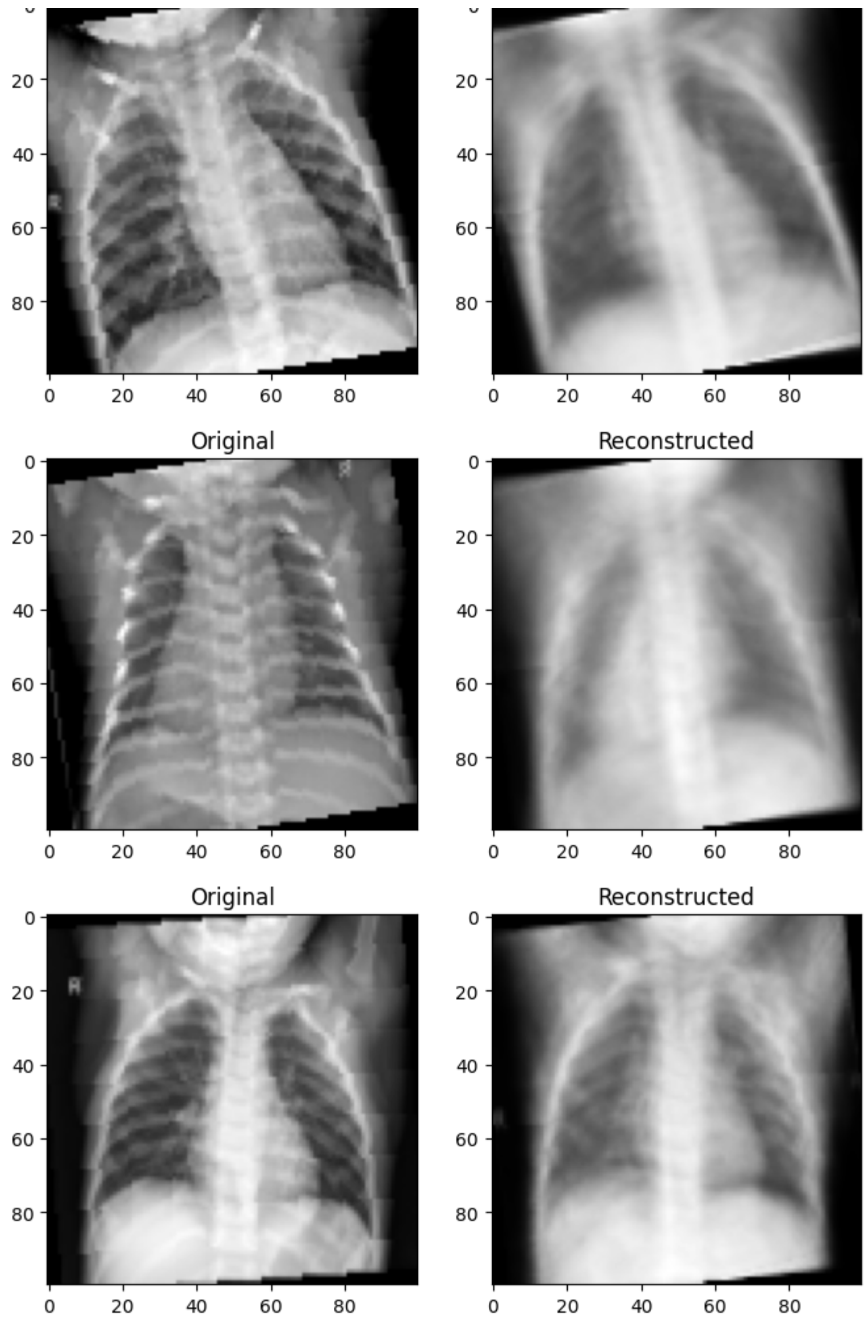
```
Epoch 281/300, Loss: 0.6284, Time taken: [30.42s]


Epoch 282/300, Loss: 0.6241, Time taken: [31.17s]


Epoch 283/300, Loss: 0.6258, Time taken: [33.04s]


Epoch 284/300, Loss: 0.6181, Time taken: [31.59s]


Epoch 285/300, Loss: 0.6305, Time taken: [32.63s]


Epoch 286/300, Loss: 0.6303, Time taken: [31.60s]


Epoch 287/300, Loss: 0.6213, Time taken: [31.69s]


Epoch 288/300, Loss: 0.6168, Time taken: [30.80s]


Epoch 289/300, Loss: 0.6240, Time taken: [32.00s]


Epoch 290/300, Loss: 0.6190, Time taken: [30.80s]


Epoch 291/300, Loss: 0.6184, Time taken: [33.03s]
```

```
Epoch 299/300, Loss: 0.6144, Time taken: [31.35s]


Epoch 300/300, Loss: 0.6222, Time taken: [30.69s]
```

```
checkpoint = torch.load(checkpoint_path)
saved_losses = checkpoint['loss']

# Plot the loss values
plt.plot(saved_losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.show()
```




```
# Plot some original and reconstructed images
n_samples = 3  # Number of samples to visualize
with torch.no_grad():
    for i, batch in enumerate(train_dl):
        if i >= n_samples:
            break
        batch = batch.to(device)
        batch = batch.flatten(1)
        reconstructed = dec(enc(batch))

        plt.figure(figsize=(8, 4))
        plt.subplot(1, 2, 1)
        plt.title('Original')
        plt.imshow(batch[0].view(100, -1).cpu().numpy(), cmap='gray')  # Reshape to original size

        plt.subplot(1, 2, 2)
        plt.title('Reconstructed')
        # plt.imshow(reconstructed.view(100, -1).cpu().numpy(), cmap='gray')  # Reshape to original size
        plt.imshow(reconstructed[0].view(100,100).cpu().numpy(), cmap='gray')  # Reshape to original size


        plt.show()
```


Original
Reconstructed
Original
Reconstructed