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

```
Mounted at /content/drive
```

```
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
from sklearn.model_selection import train_test_split


def aspect_ratio_preserving_resize(image, target_size):
    width, height = image.size
    target_width, target_height = target_size

    # Calculate the aspect ratio
    aspect_ratio = width / height

    if width > height:
        new_width = target_width
        new_height = int(new_width / aspect_ratio)
    else:
        new_height = target_height
        new_width = int(new_height * aspect_ratio)

    # Perform the resize
    image = transforms.functional.resize(image, (new_height, new_width))

    # Create a new image with the target size and paste the resized image in the center
    new_image = Image.new("L", target_size)
    new_image.paste(image, ((target_width - new_width) // 2, (target_height - new_height) // 2))

    return new_image


class MyDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform
        self.image_paths = [os.path.join(data_dir, file) for file in os.listdir(data_dir)]

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

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

        # Apply aspect ratio-preserving resize
        resized_image = aspect_ratio_preserving_resize(image, (100, 100))

        if self.transform:
            transformed_image = self.transform(resized_image)
        else:
            transformed_image = resized_image

        return transformed_image


# Define your data transformation
train_transform = transforms.Compose([
    transforms.Resize((100, 100)),
    transforms.RandomHorizontalFlip(p=0.2),
    transforms.RandomVerticalFlip(p=0.2),
    transforms.RandomRotation(degrees=(5, 15)),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),
```

```
    transforms.RandomResizedCrop((100, 100), scale=(0.8, 1.0)),
    transforms.ToTensor(),
])


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



batch_size = 16

# Load data
train_dataset = MyDataset(data_dir='/content/drive/MyDrive/AE_xray/train', transform=test_transform)
test_dataset = MyDataset(data_dir='/content/drive/MyDrive/AE_xray/test', transform=test_transform)
train_data, valid_data = train_test_split(train_dataset, test_size=0.1, random_state=42)

# Dataloader
train_dl = DataLoader(train_data, batch_size=batch_size, shuffle=True)
valid_dl = DataLoader(valid_data, 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 = 5000, hidden_size2 = 2000 , hidden_size3 = 1000, hidden_size4 = 500, z_
    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()
    self.dropout = nn.Dropout(0.2)  # Add dropout with a 50% probability
  def forward(self, x):

    x = self.relu(self.fc1(x))
    x = self.dropout(x)  # Apply dropout after the first layer
    x = self.relu(self.fc2(x))
    x = self.dropout(x)  # Apply dropout after the second layer
    x = self.relu(self.fc3(x))
    x = self.dropout(x)  # Apply dropout after the third layer
    x = self.relu(self.fc4(x))
    x = self.fc5(x)
    return x



class Decoder(nn.Module):
  def __init__(self , output_size = 10000 , hidden_size1 = 5000, hidden_size2 = 2000 , hidden_size3 = 1000, hidden_size4 = 500, 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()
    self.dropout = nn.Dropout(0.5)  # Add dropout with a 50% probability
  def forward(self, x):
    x = self.relu(self.fc1(x))
    x = self.dropout(x)  # Apply dropout after the first layer
    x = self.relu(self.fc2(x))
    x = self.dropout(x)  # Apply dropout after the second layer
    x = self.relu(self.fc3(x))
    x = self.dropout(x)  # Apply dropout after the third layer
    x = self.relu(self.fc4(x))
    x = torch.sigmoid(self.fc5(x))
    return x


# class Encoder(nn.Module):
#     def __init__(self, input_size=10000, hidden_size1=2500, hidden_size2=1000, hidden_size3=500, hidden_size4=200, z_dim=100):
#         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()
#         self.dropout = nn.Dropout(0.5)  # Add dropout with a 50% probability

    # def forward(self, x):
    #     x = self.relu(self.fc1(x))
    #     x = self.dropout(x)  # Apply dropout after the first layer
    #     x = self.relu(self.fc2(x))
    #     x = self.dropout(x)  # Apply dropout after the second layer
    #     x = self.relu(self.fc3(x))
    #     x = self.dropout(x)  # Apply dropout after the third layer
    #     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_dim=100):
#         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()
#         self.dropout = nn.Dropout(0.5)  # Add dropout with a 50% probability

    # def forward(self, x):
    #     x = self.relu(self.fc1(x))
    #     x = self.dropout(x)  # Apply dropout after the first layer
    #     x = self.relu(self.fc2(x))
    #     x = self.dropout(x)  # Apply dropout after the second layer
    #     x = self.relu(self.fc3(x))
    #     x = self.dropout(x)  # Apply dropout after the third layer
    #     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(), weight_decay=1e-5)
optimizer_dec = torch.optim.Adam(dec.parameters(), weight_decay=1e-5)
```

```
train_loss = []
val_loss = []
num_epochs = 200
checkpoint_path = "/content/drive/MyDrive/model/Autoencoder/decay_dropout_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["train_loss"]
  val_loss = checkpoint["val_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
```

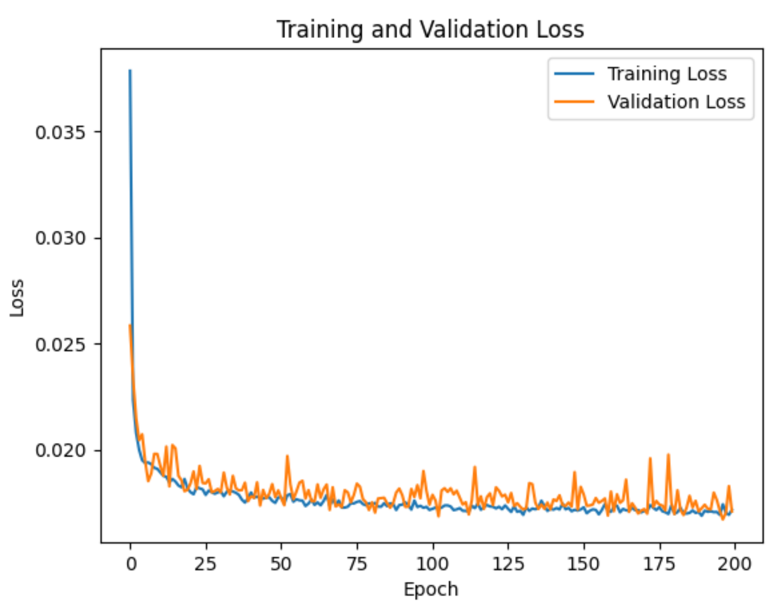
```
Resume training from epoch 13
```

```
total_batches_train = len(train_dl)
total_batches_valid = len(valid_dl)
```

```
    total_batches_valid = len(valid_dl)
    for epoch in range(start_epoch,num_epochs+1):
        train_epoch_loss = 0
        valid_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_train, desc=f'Epoch {epoch}/{num_epochs}', leave=False)
        for step, imgs in epoch_progress:
            imgs = imgs.to(device)
            imgs = imgs.flatten(1)
            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()

        with torch.no_grad():
          for val_imgs in valid_dl:
            val_imgs = val_imgs.to(device)
            # val_imgs = add_noise(val_imgs)
            val_imgs = val_imgs.flatten(1)
            val_reconstructed = dec(enc(val_imgs))
            step_loss = loss_fn(val_reconstructed, val_imgs)
            valid_epoch_loss += step_loss.item()

        # epoch_progress.set_description(f'Epoch {epoch}/{num_epochs}, Step {step}/{total_batches}, Train_step_loss: {loss.item():.4f
        # Calculate average loss
        train_epoch_loss /= total_batches_train
        valid_epoch_loss /= total_batches_valid

        train_loss.append(train_epoch_loss)
        val_loss.append(valid_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}, Train_loss: {train_epoch_loss:.4f}, Val_loss: {valid_epoch_loss:.4f}, Time taken: [{time.

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

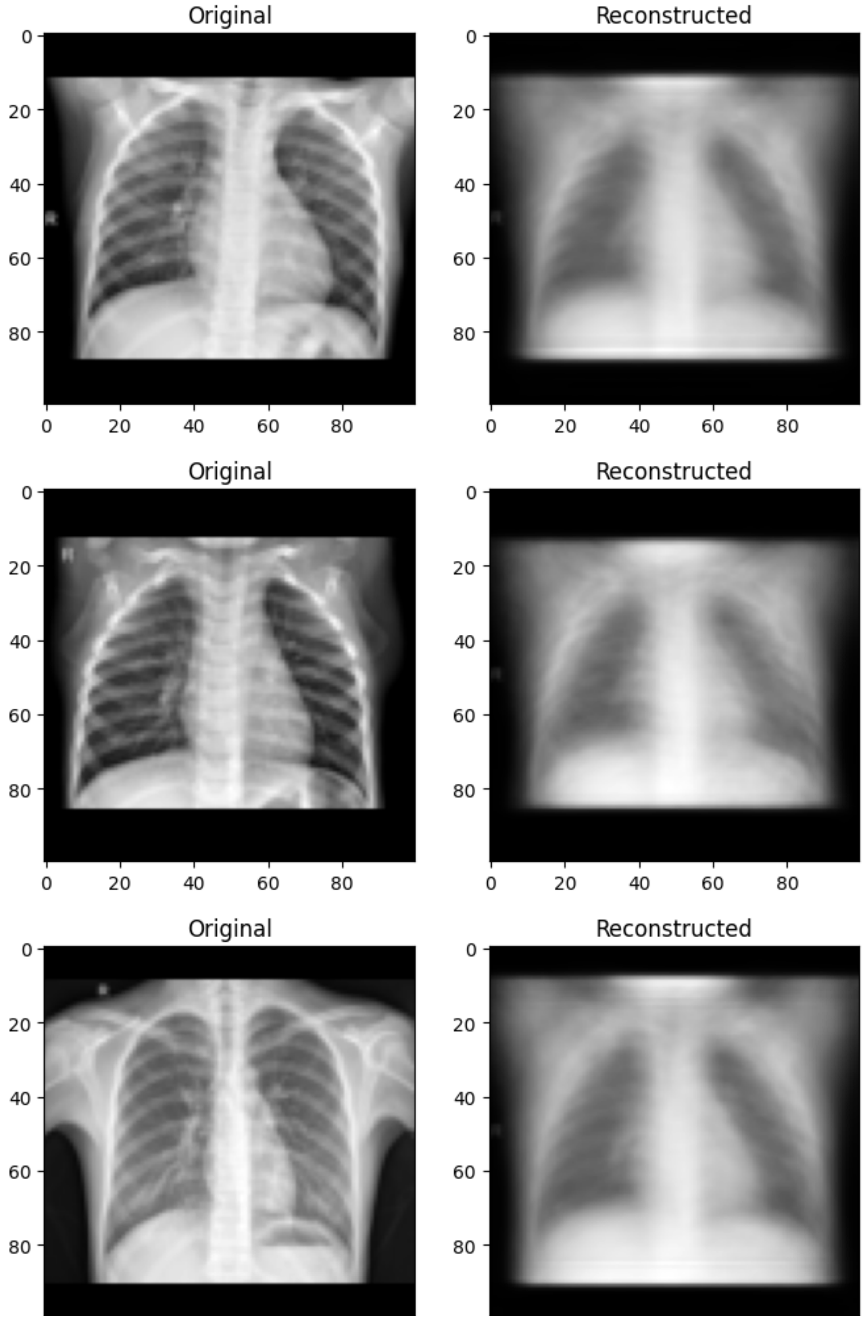
```
Epoch 189/200, Train_loss: 0.0171, Val_loss: 0.0171, Time taken: [4.38s]

Epoch 190/200, Train_loss: 0.0169, Val_loss: 0.0172, Time taken: [4.38s]

Epoch 191/200, Train_loss: 0.0171, Val_loss: 0.0174, Time taken: [4.38s]

Epoch 192/200, Train_loss: 0.0171, Val_loss: 0.0172, Time taken: [4.38s]

Epoch 193/200, Train_loss: 0.0171, Val_loss: 0.0172, Time taken: [4.38s]

Epoch 194/200, Train_loss: 0.0170, Val_loss: 0.0180, Time taken: [4.38s]

Epoch 195/200, Train_loss: 0.0171, Val_loss: 0.0176, Time taken: [4.37s]

Epoch 196/200, Train_loss: 0.0169, Val_loss: 0.0171, Time taken: [4.37s]

Epoch 197/200, Train_loss: 0.0174, Val_loss: 0.0167, Time taken: [4.37s]

Epoch 198/200, Train_loss: 0.0170, Val_loss: 0.0170, Time taken: [4.38s]

Epoch 199/200, Train_loss: 0.0169, Val_loss: 0.0183, Time taken: [4.38s]

Epoch 200/200, Train_loss: 0.0172, Val_loss: 0.0171, Time taken: [4.37s]
```

```
# checkpoint = torch.load(checkpoint_path)
train_loss = checkpoint['train_loss']
valid_loss = checkpoint['val_loss']

plt.plot(train_loss, label='Training Loss')
plt.plot(valid_loss, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
```




```
# Plot some original and reconstructed images
n_samples = 5  # Number of samples to visualize

with torch.no_grad():
    for i, batch in enumerate(test_dl):
        if i >= n_samples:
            break
```

```
    batch = batch.to(device)
    batch = batch.flatten(1)
    reconstructed = dec(enc(batch))

    original_image = batch[0].view(100,-1).cpu().numpy()
    reconstructed_image = reconstructed[0].view(100,-1).cpu().numpy()

    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.title('Original')
    plt.imshow(original_image, cmap='gray')  # Convert to grayscale for display

    plt.subplot(1, 2, 2)
    plt.title('Reconstructed')
    plt.imshow(reconstructed_image, cmap='gray')  # Convert to grayscale for display

    plt.show()
```


Original
Reconstructed
0
20
40
60
80
0 20 40 60 80
Original
Reconstructed
0
20
40
60
80
0 20 40 60 80
Original
Reconstructed
0
20
40
60
80