```
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 transforms

import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
transform = transforms.ToTensor()
```

```
train_dataset = torchvision.datasets.MNIST(root = "./data" , train = True , download = True ,  transform = transform)
test_dataset = torchvision.datasets.MNIST(root = "./data" , train = False , download = True ,  transform = transform)

train_data, valid_data = train_test_split(train_dataset, test_size=0.2, random_state=42)
```

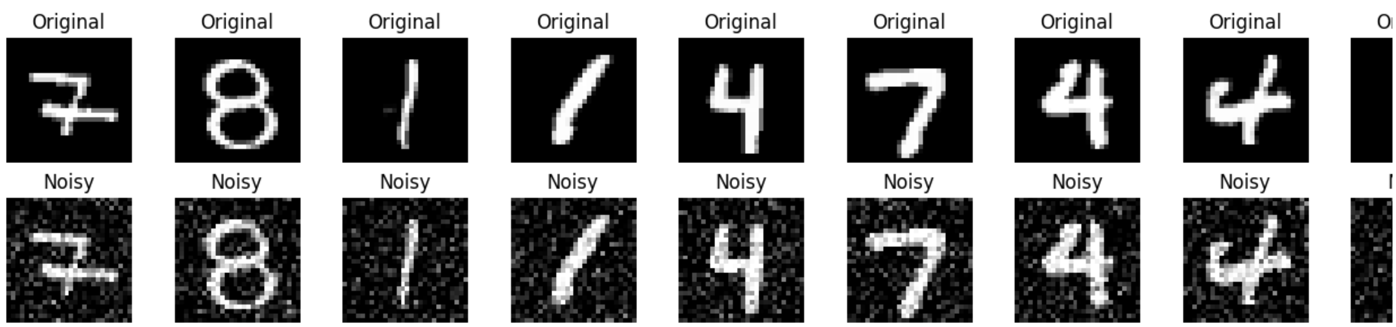
```
train_dl = torch.utils.data.DataLoader(train_data, batch_size=100, shuffle=True)
valid_dl = torch.utils.data.DataLoader(valid_data, batch_size=100)
test_dl = torch.utils.data.DataLoader(test_dataset, batch_size = 100)
```

```
class Encoder(nn.Module):
  def __init__(self , input_size = 28*28 , hidden_size1 = 500, hidden_size2 = 250 , hidden_size3 = 100, hidden_size4 = 50, z_dim
    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 = 28*28 , hidden_size1 = 500 , hidden_size2 = 250 , hidden_size3 = 100, hidden_size4 = 50, z_di
    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
```

```
# Add noise to the input images
def add_noise(images, noise_factor=0.2):
    noisy_images = images + noise_factor * torch.randn_like(images)
    return torch.clamp(noisy_images, 0.0, 1.0)  # Ensure pixel values are in [0, 1]
```

```
# Number of sample images to display
num_samples = 10

# Create a DataLoader iterator
data_iterator = iter(train_dl)

# Get the next batch of data
sample_batch, _ = next(data_iterator)

# Display original images
plt.figure(figsize=(15, 3))
for i in range(num_samples):
    plt.subplot(2, num_samples, i + 1)
    plt.imshow(sample_batch[i].squeeze().numpy(), cmap='gray')
    plt.title('Original')
    plt.axis('off')

# Add noise to the images
noisy_batch = add_noise(sample_batch)
noisy_batch = torch.clamp(noisy_batch, 0.0, 1.0)  # Ensure pixel values are in [0, 1]

# Display noisy images
for i in range(num_samples):
    plt.subplot(2, num_samples, i + num_samples + 1)
    plt.imshow(noisy_batch[i].squeeze().numpy(), cmap='gray')
    plt.title('Noisy')
    plt.axis('off')

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




```
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 = []
val_loss = []
num_epochs = 300
checkpoint_path = '/content/drive/MyDrive/model/Autoencoder/new_noisy_checkpoint_30z_5h_250e.pth'
```

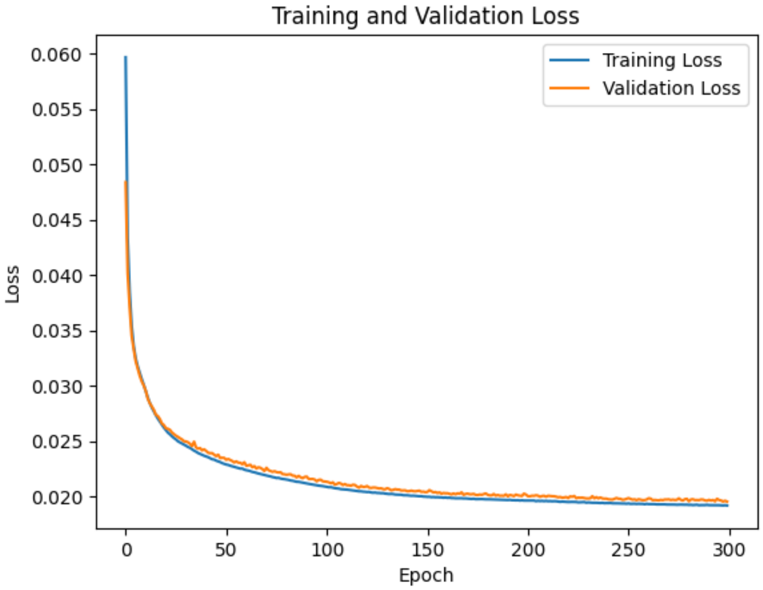
```
# Check if a checkpoint exists to resume training
if os.path.exists(checkpoint_path):
  checkpoint = torch.load(checkpoint_path)
  enc.load_state_dict(checkpoint["enc_state_dict"])
  dec.load_state_dict(checkpoint["dec_state_dict"])
  optimizer_enc.load_state_dict(checkpoint["optimizer_enc_state_dict"])
  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
```

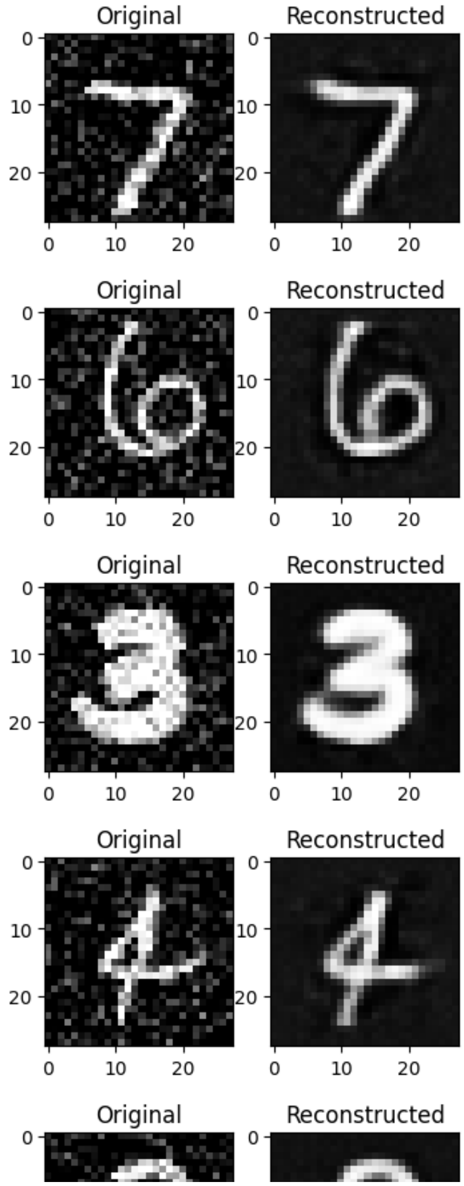
```
    Resume training from epoch 201
```

```
total_batches_train = len(train_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 = add_noise(imgs)
        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)
```

```
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()
```




```
n_samples = 5  # Number of samples to visualize
with torch.no_grad():
    for i, (batch, _) in enumerate(test_dl):
        if i >= n_samples:
            break
        # Transfer the batch to the device(GPU)
        batch = batch.to(device)
        batch = add_noise(batch)

        # Flatten the batch
        batch = batch.view(batch.size(0), -1)

        # Pass the batch through the encoder and decoder to obtain reconstructed images
        reconstructed = dec(enc(batch))

        # Plot the original and reconstructed images
        plt.figure(figsize=(4, 2))
        plt.subplot(1, 2, 1)
        plt.title('Original')
        plt.imshow(batch[0].view(28, 28).cpu().numpy(), cmap='gray')  # Reshape to original size

        plt.subplot(1, 2, 2)
        plt.title('Reconstructed')
        plt.imshow(reconstructed[0].view(28, 28).cpu().numpy(), cmap='gray')  # Reshape to original size

        plt.show()
```


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed
Original
Reconstructed
Original
Reconstructed