This can be run [run on Google Colab using this link (https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.2-Diffusion.ipynb)](https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.2-Diffusion.ipynb)

STABLE DIFFUSION ASSIGNMENT

Preliminary

In this homework assignment, you will delve deep into Stable Diffusion Models based on the DDPMs paper. The homework is fragmented into three main parts: Forward Diffusion, the Unet Architecture of Noise Predictor Model with training and the Sampling part of Stable Diffusion Models. By completing this assignment, you will gain a comprehensive understanding of the mathematics underlying stable diffusion and practical skills to implement and work with these models.

Setup and Data Preparation

Execute the provided cell to import essential libraries, ensure result reproducibility, set device configurations, download the MNIST dataset, and initialize DataLoaders for training, validation, and testing.

Note: Run the cell as is; no modifications are necessary.

In [1]:
```
##########################################################################
#########
#                                  TO DO
#
#                 Execute the block to load & Split the Dataset
#
##########################################################################
#########

import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
```

```
# Ensure reproducibility
torch.manual_seed(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Check if CUDA is available  # Using APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
print(torch.ones(1, device=device))
# Download and Load the MNIST dataset
transform = transforms.ToTensor()
full_trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)

# Splitting the trainset into training and validation datasets
train_size = int(0.8 * len(full_trainset))  # 80% for training
val_size = len(full_trainset) - train_size  # remaining 20% for validation
train_dataset, val_dataset = torch.utils.data.random_split(full_trainset, [train_size, val_size])

trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
valloader = torch.utils.data.DataLoader(val_dataset, batch_size=32, shuffle=False)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)
```

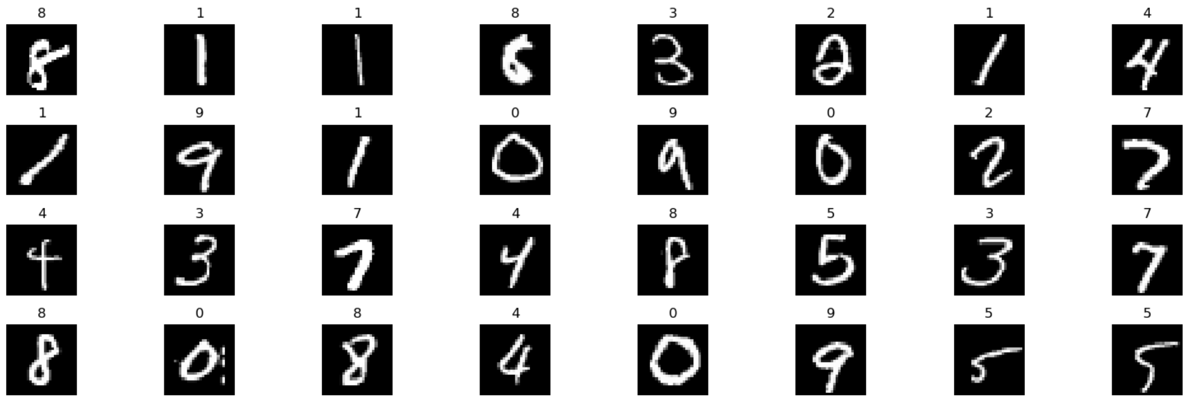
```
/Users/karan_mudaliar/miniconda3/lib/python3.11/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm

tensor([1.], device='mps:0')
```

Image Display Function

Below is a utility function, `display_images` , used for visualizing dataset and monitoring diffusion process for slight intuitive way of choosing parameter purposes and display results post training in this assignment.

Note: Run the cell to view the images from the dataset.

In [2]:
```
################################################################
#########
#                               TO DO
#
#                 Execute the block to display images of MNIST
#
################################################################
#########

import matplotlib.pyplot as plt

def display_images(images, n, images_per_row=5, labels = None):
    """
    Display n images in rows where each row contains a specified nu
mber of images.

    Parameters:
    - images: List/Tensor of images to display.
    - n: Number of images to display.
    - images_per_row: Number of images per row.
    """
    # Define the number of rows based on n and images_per_row
    num_rows = (n + images_per_row - 1) // images_per_row  # Roundi
ng up
    plt.figure(figsize=(2*images_per_row, 1.25 * num_rows))
    for i in range(n):
        plt.subplot(num_rows, images_per_row, i+1)
        plt.imshow(images[i].cpu().squeeze().numpy(), cmap='gray')
        if labels is not None:
          plt.title(labels[i])
        plt.axis('off')
    plt.tight_layout()
    plt.show()


for batch in trainloader:
  # In a batch from many batches in trainloader, get the the first
one and work with that
  batch_size = len(batch[0])
  display_images(images= batch[0],n = batch_size, images_per_row=8,
labels = batch[1].tolist())
  break
```



EXERCISE 1: FORWARD DIFFUSION

Noise Diffusion

The following block `Noise Diffusion` is to give you a high level intuition of what forward diffusion process is and how we achieve results without any dependency on prior results. There is a detailed derivation on how we landed on the formula mentioned in the paper and below, if you're interested in the math, we recommend reading [Denoising Diffusion Probabilistic Models (https://arxiv.org/abs/2006.11239)](https://arxiv.org/abs/2006.11239) for clear understanding of ***Forward Diffusion Process*** and mathematical details involved in it!

Noise Diffusion

The idea behind adding noise to an image is rooted in a simple linear interpolation between the original image and a noise term. Let's use the concept of a blending or mixing factor (which we'll refer to as α)

1. Linear Interpolation:

Given two values, A and B, the linear interpolation between them based on a blending factor α (where $0 \leq \alpha \leq 1$) is given by:

$$\text{Result} = \alpha A + (1 - \alpha) B$$

If $\alpha = 1$, the Result is entirely A. If $\alpha = 0$, the Result is entirely B. For values in between, you get a mixture.

2. Applying to Images and Noise:

In our context:

- A is the original image.
- B is the noise (often drawn from a standard normal distribution, but could be any other distribution or type of noise).

So, for each pixel (p) in our image, and at a given timestep (t):

$$\text{noisy\_image}_p(t) = \alpha(t) \times \text{original\_image}_p + (1 - \alpha(t)) \times \text{noise}_p$$

Where:

- $\alpha(t)$ is the blending factor at timestep t
- original_image_p is the intensity of pixel p in the original image.
- noise_p is the noise value for pixel p, typically drawn from a normal distribution.

3. Time-Dependent α:

For the Time-Dependent Alpha Noise Diffusion method, our α isn't a constant; it changes over time. That's where our linear scheduler or any other scheduler comes in: to provide a sequence of values over timesteps.

Now, considering cumulative products: The reason for introducing the cumulative product of αs was to have an accumulating influence of noise over time. With each timestep, we multiply the original image with the cumulative product of α values up to that timestep, making the original image's influence reduce multiplicatively. The noise's influence, conversely, grows because it's based on $1-$ the cumulative product of the αs.

That's why the formula becomes:

$$\text{noisy\_image}_t = \text{original\_image} \times \prod_{i=1}^{t} \alpha_i + \text{noise} \times (1 - \prod_{i=1}^{t} \alpha_i)$$

In essence, this formula is just a dynamic way to blend an original image and noise, with the blending ratios changing (and typically becoming more skewed toward noise) over time.

4. Linear Scheduling of Noise Blending:

One of the core components of this noise diffusion assignment is how the blending of noise into the original image is scheduled. To accomplish this, we utilize a linear scheduler that determines the progression of the β (noise level parameter) over a series of timesteps.

Imagine you wish to transition β from a `start_beta` of 0.1 to an `end_beta` of 0.2 over 11 timesteps. The goal is for the rate of noise blending into the image to increase progressively. In this case, the sequence of β values would look like this: [0.1, 0.11, 0.12,..., 0.2].

This sequence, `self.betas`, is precisely what the `linear_scheduler` generates.

```
self.betas = self.linear_scheduler().to(self.device)
```

In essence, the `linear_scheduler` method calculates the sequence of β values for the diffusion process, ensuring that the noise blending into the image increases linearly over the given timesteps.

Terminologies:

1. β: Represents the noise level parameter, defined between the start and end beta values.
2. α: Represents the blending factor, calculated as $(1 - \beta)$.
3. Cumulative Product of α: Understand its significance in dynamically blending the original image and noise over timesteps, without any dependency on prior timesteps.

NoiseDiffuser Class

TO DO

Implement `NoiseDiffuser` Class, ***Follow Instructions in the code cell***

In [30]:
```
import torch

class NoiseDiffuser:
  def __init__(self, start_beta, end_beta, total_steps, device='cpu
'):

    assert start_beta < end_beta < 1.0

    self.device = device
    self.start_beta = start_beta
    self.end_beta = end_beta
    self.total_steps = total_steps
    ################################################################
#############
    #                                  TO DO
#
    #                    Compute the following variables needed
#
    #                        for Forward Diffusion Process
#
    #                    schedule betas,  compute alphas & cumulative
#
    #                                product of alphas
#
    ################################################################
#############
    self.betas = self.linear_scheduler().to(self.device)
    self.alphas = 1- self.betas
```

```
    self.alpha_bar = torch.cumprod(self.alphas,dim = 0)


  def linear_scheduler(self):
    """Returns a linear schedule from start to end over the specifi
ed total number of steps."""
    ##############################################################
#############
    #                          TO DO
#
    #                Return a linear schedule of `betas`
#
    #                  from `start_beta` to `end_beta`
#
    #                       hint: torch.linspace()
#
    ##############################################################
#############

    #creates a set of beats, one for each time stamp, from start_be
ta to end_betam equally spaces.
    #eg: if srart_beta is 1 and end_beta is 10 and total_steps is 2
0 then we get: 1, 1.5, 2, 2.5.....10.
    return torch.linspace(self.start_beta, self.end_beta, self.tota
l_steps, device=self.device)




  def noise_diffusion(self, image, timesteps):

    image_num = 0
    noisy_image = image.new_zeros(image.size())

    for t in timesteps:

      image = image.to(self.device)
      noisy_image = noisy_image.to(self.device)

      # Compute alphas
      alphas = 1.0 - self.betas[:t]
      alpha_bar = torch.prod(alphas)

      # Apply noise diffusion to the image using alpha_bar
      # got stuck here, worked in collaboration with VARUN MOHAN
      noisy_image[image_num] = image[image_num]*torch.sqrt(alpha_ba
r) + torch.randn_like(image[image_num])*(torch.sqrt(1-alpha_bar))
      image_num += 1

    true_noise = image - noisy_image
    return noisy_image, true_noise
```

Testing NoiseDiffuser Class (SANITY CHECK)

In [31]:
```
# SANITY CHECK
in_channels_arg = 1
out_channels_arg = 1
batch_size = 32
height = 28
width = 28
total_timesteps = 50
start_beta, end_beta = 0.001, 0.2


# Check if CUDA is available  # Using APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() el
se "cpu")
print(torch.ones(1, device=device))

# Sanity check
x = torch.randn((batch_size, in_channels_arg, height, width)).to(de
vice)
diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps, dev
ice)


timesteps_to_display = torch.randint(0, total_timesteps, (batch_siz
e,), device=device).long().tolist()
y, _ = diffuser.noise_diffusion(x, timesteps_to_display)
assert len(x.shape) == len(y.shape)
assert y.shape == x.shape

print("Sanity Check for shape mismatches")
print("Shape of the input : ", x.shape)
print("Shape of the output : ", y.shape)
```

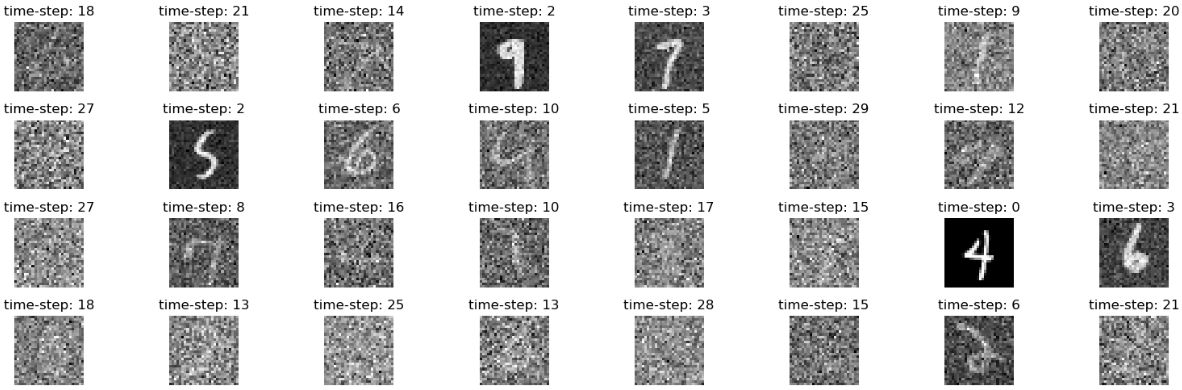
```
tensor([1.], device='mps:0')
Sanity Check for shape mismatches
Shape of the input :  torch.Size([32, 1, 28, 28])
Shape of the output :  torch.Size([32, 1, 28, 28])
```

Demonstrating Examples

Note: Observe the visual effect of noise diffusion for different images at random timesteps. How does the noise appear? As the time step increases, the noise in the image increases.

In [50]:
```
#################################################################
#########
#                              TO DO
#
#       Initialize some start_beta, end_beta & total_timesteps
#
#                        and execute the block
#
#################################################################
#########

# Check if CUDA is available  # Using APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() el
se "cpu")
print(torch.ones(1, device=device))
total_timesteps = 30
start_beta, end_beta = 0.001, 0.2
diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps, dev
ice)

for batch in trainloader:
    minibatch = batch[0]
    batch_size = len(minibatch)
    timesteps_to_display = torch.randint(0, total_timesteps, (batch
_size,), device=device).long().tolist()
    noisy_images,_ = diffuser.noise_diffusion(minibatch, timesteps_
to_display)
    display_images(images=noisy_images, n=batch_size, images_per_ro
w=8, labels=list(map(lambda x: "time-step: " + str(x), timesteps_to
_display)))
    break
```

```
tensor([1.], device='mps:0')
```



HyperParameters

Smartly setting the start and end values of beta can control the noise diffusion's character.

- **Lower Start and Higher End**: Starting with a lower beta and ending with a higher one means that original image's contribution remains dominant in the beginning and slowly diminishes. This can be useful when the goal is to have a gradual transition from clear image to noisier version.
- **Higher Start and Lower End**: The opposite approach, starting with a Higher beta and ending with a lower one, can be useful when goal is to introduce noise more aggressively initially and taper off towards the end.
- **THINK WHAT WOULD WE NEED** `Higher Start and Lower End` or `Lower Start and Higher End` - I think The first part is needed as the model will have to learn about the noise slowly

The precise values can be fine-tuned based on specific requirements, visual assessments (like in the cell below) or even metrics.

Exploration with Varied beta Values and Timesteps:

- In the below cell, you are encouraged to tweak values of `start_beta` and `end_beta` and even modify `total_timesteps` to observe the effect over a longer/shorter period

Note: Pay close attention to how the noise diffusion evolves over time. Can you see a clear transition from the start to the end timestep? How do different images react to the same noise diffusion process? Yes, I can see a clear transition from the start to the end timestep

In [51]:
```
#######################################################################
#########
#                                  TO DO
#
#        Initialize some start_beta, end_beta & total_timesteps
#
#          play around and see the effect of noise introduced
#
#          and think what parameters would you use for training
#
#######################################################################
#########

# Check if CUDA is available  # Using APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() el
se "cpu")
print(torch.ones(1, device=device))
total_timesteps = 100
start_beta, end_beta = 0.0001,0.01
minibatch_size = 1
diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps, dev
ice)

# PLay around in this cell with different value of alpha (start and
end) and different number of time steps to initially guess and deci
de on how many time steps would you like to train the model going f
orward.
for batch in trainloader:
    repetitions = torch.tensor([total_timesteps]).repeat(minibatch_
size)
    minibatch = batch[0][:minibatch_size,:,:].repeat_interleave(rep
etitions, dim=0)
    batch_size = len(minibatch)
    timesteps_to_display = torch.linspace(0, total_timesteps-1, tot
al_timesteps, dtype=int).tolist() * minibatch_size
    noisy_images,_ = diffuser.noise_diffusion(minibatch, timesteps_
to_display)
    display_images(images=noisy_images, n=batch_size, images_per_ro
w=10, labels=list(map(lambda x: "time-step: " + str(x), timesteps_t
o_display)))
    break
```

```
tensor([1.], device='mps:0')
```

EXERCISE 2: REVERSE DIFFUSION

Model Architecture

Implementing Skip Connections in U-Net Architecture

While the architecture of the U-Net is provided to you, a critical component—skip connections—needs to be integrated by you. The original paper, "[U-Net: Convolutional Networks for Biomedical Image Segmentation (https://arxiv.org/abs/1505.04597)](https://arxiv.org/abs/1505.04597)" showcases the importance of these skip connections, as they allow the network to utilize features from earlier layers, making the segmentation more precise.

Placeholder for Skip Connections:

In the given architecture, you will find lines like the one below, which are the components of upsampling process in the U-Net:

```
y2 = self.afterup2(torch.cat([y2, torch.zeros_like(y2)], axis = 1))
```

Here, `torch.zeros_like(y2)` acts as a placeholder, indicating where the skip connection should be added. Your task is to replace this placeholder with the appropriate feature map from an earlier corresponding layer in the network.

Important Points to Keep in Mind:

- The U-Net architecture has multiple layers, so you'll need to repeat this process for each layer where skip connections are required.
- The provided helper function, `self.xLikeY(source, target)` , will be crucial in ensuring the feature maps you concatenate have matching dimensions.
- While the focus of this assignment is on crucial idea of stable diffusion, the U-Net architecture is provided to you but it is importatnt you implement skip connections, as understanding their role and significance in the U-Net architecture will be beneficial.
- ***Note: Feel free to modify architecture, parameters including number & types of layers used, kernel Sizes, padding, etc, you won't be judged on the architecture you use if you have the desired results post training.***

UNet Class

TO DO

Fill in `UNet` Class, ***Follow Instructions above***

In [52]:
```
class UNet(nn.Module):

    def __init__(self, in_channels, out_channels):
        """
        in_channels: input channels of the incoming image
        out_channels: output channels of the incoming image
        """
        super(UNet, self).__init__()

        #------------------------ Encoder ------------------------#
        ##############################################################
```

```
###############
        #        Initial Convolutions (Using doubleConvolution() func
tion)        #
        #           Building Down Sampling Layers (Using Down() funct
ion)         #
        #################################################################
###############
        self.ini = self.doubleConvolution(inC = in_channels, oC = 16)
        self.down1 = self.Down(inputC = 16, outputC = 32)
        self.down2 = self.Down(inputC = 32, outputC = 64)

        #------------------------- Decoder -------------------------#
        #################################################################
###############
        #                          For each Upsampling block
#
        #         Building Time Embeddings (Using timeEmbeddings() fu
nction)        #
        #       Building Up Sampling Layer (Using ConvTranspose2d() f
unction)       #
        #       followed by Convolution (Using doubleConvolution() fu
nction)        #
        #################################################################
###############

        self.time_emb2 = self.timeEmbeddings(1, 64)
        self.up2 = nn.ConvTranspose2d(in_channels=64, out_channels=32
, kernel_size=3, stride=2)
        self.afterup2 = self.doubleConvolution(inC = 64 , oC = 32)

        self.time_emb1 = self.timeEmbeddings(1, 32)
        self.up1 = nn.ConvTranspose2d(in_channels=32, out_channels=16
, kernel_size=3, stride=2)
        self.afterup1 = self.doubleConvolution(inC = 32 , oC = 16, kS
1=5, kS2=4)

        #------------------------- OUTPUT -------------------------#
        #################################################################
###############
        #            Constructing final Output Layer (Use Conv2d() fu
nction)        #
        #################################################################
###############
        self.out = nn.Conv2d(in_channels=16, out_channels=out_channel
s, kernel_size=1, stride=1, padding=0)

    def forward(self, x, t=None):
        assert t is not None

        # Check if CUDA is available  # Using APPLE SILICON
        device = torch.device("mps" if torch.backends.mps.is_availabl
e() else "cpu")
```

```
        x = x.to(device)

        #------------------------- Encoder -------------------------#
        ##############################################################
###############
        #                          Processing Inputs by
#
        #                     performing Initial Convolutions
#
        #                      followed by Down Sampling Layers
#
        ##############################################################
###############
        x1 = self.ini(x)                  # Initial Double Convolutio
n
        x2 = self.down1(x1)               # Downsampling followed by
Double Convolution
        x3 = self.down2(x2)               # Downsampling followed by
Double Convolution

        # t is a list. We need to convert it back to a tensor
        t = torch.tensor(t).to(device)

        #------------------------- Decoder -------------------------#
        ##############################################################
###############
        #             For each Upsampling block, we add time Embedding
s to          #
        #                    Feature Maps, process this by
#
        #             Up Sampling followed by concatenation & Convolut
ion           #
        ##############################################################
###############
        t = t.view(t.size(0), 1)  # Reshapes it to (batch_size, 1)
        t = t.to(torch.float32)

        t2 = self.time_emb2(t)[:,:, None, None]
        y2 = self.up2(x3 + t2)                       # Upsampling
        y2 = self.afterup2(torch.cat([y2, self.xLikeY(x2, y2)], axis=
1))  # Crop corresponding Downsampled Feature Map, Double Convoluti
on
        t1 = self.time_emb1(t)[:,:, None, None]
        y1 = self.up1(y2 + t1)                  # Upsampling
        y1 = self.afterup1(torch.cat([y1, self.xLikeY(x1, y1)], axis=
1)) # Crop corresponding Downsampled Feature Map, Double Convolutio
n

        #------------------------- OUTPUT -------------------------#
        ##############################################################
###############
        #                          Processing final Output
#
```

```
        ################################################################
###############
        outY = self.out(y1)                    # Output Layer (ks-1, st-1,
pa-0)

        return outY



    #-----------------------------------------------------------------
------------------------------------- Helper Functions Within Model
Class

    def timeEmbeddings(self, inC, oSize):
        """
        inC: Input Size, (for example 1 for timestep)
        oSize: Output Size, (Number of channels you would like to mat
ch while upsampling)
        """
        return nn.Sequential(nn.Linear(inC, oSize),
                             nn.ReLU(),
                             nn.Linear(oSize, oSize))

    def doubleConvolution(self, inC, oC, kS1=3, kS2=3, sT=1, pA=1):
        """
        Building Double Convolution as in original paper of Unet
        inC : inputChannels
        oC : outputChannels
        kS1 : Kernel_size of first convolution
        kS2 : Kernel_size of second convolution
        sT: stride
        pA: padding
        """
        return nn.Sequential(
              nn.Conv2d(in_channels= inC, out_channels=oC, kernel_siz
e=kS1, stride=sT, padding=pA),
              nn.ReLU(inplace=True),
              nn.Conv2d(in_channels = oC,out_channels=oC, kernel_size
=kS2, stride=sT, padding=pA),
              nn.ReLU(inplace=True),
        )


    def Down(self, inputC, outputC, dsKernelSize = None):
        """
        Building Down Sampling Part of the Unet Architecture (Using M
axPool) followed by double convolution
        inputC : inputChannels
        outputC : outputChannels
        """

        return nn.Sequential(
            nn.MaxPool2d(2),
```

```
            self.doubleConvolution(inC = inputC, oC = outputC)
        )


    def xLikeY(self, source, target):
        """
        Helper function to resize the downsampled x's to concatenate
with upsampled y's as in Unet Paper
        source: tensor whose shape will be considered ---------UPSAMP
LED TENSOR (y)
        target: tensor whose shape will be modified to align with tar
get ---------DOWNSAMPLED TENSOR (x)
        """
        x1 = source
        x2 = target
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2, diffY // 2, d
iffY - diffY // 2])
        return x1
```

Testing UNet Class (SANITY CHECK)

In [53]:
```
# SANITY CHECK FOR UnetBottleNeck (Single Channeled B/W Images)
in_channels_arg = 1
out_channels_arg = 1
batch_size = 32
height = 28
width = 28
total_timesteps = 50

# Check if CUDA is available # USING APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() el
se "cpu")

# Positional Encoding Object
timesteps_to_display = torch.randint(0, total_timesteps, (batch_siz
e,), device=device).long().tolist()

# Sanity check
x = torch.randn((batch_size, in_channels_arg, height, width)).to(de
vice)
model = UNet(in_channels=in_channels_arg, out_channels=out_channels
_arg)
model = model.to(device)

y = model.forward(x = x, t = torch.tensor(timesteps_to_display).to(
torch.float32).to(device).view(-1,1))
assert len(x.shape) == len(y.shape)
assert y.shape == (batch_size, out_channels_arg, height, width)

print("Sanity Check for Single Channel B/W Images")
print("Shape of the input : ", x.shape)
print("Shape of the output : ", y.shape)
```

```
Sanity Check for Single Channel B/W Images
Shape of the input :  torch.Size([32, 1, 28, 28])
Shape of the output :  torch.Size([32, 1, 28, 28])

/var/folders/dn/jhd9djgd393gp0dk2lz40m2c0000gp/T/ipykernel_9563/30
08510448.py:60: UserWarning: To copy construct from a tensor, it i
s recommended to use sourceTensor.clone().detach() or sourceTenso
r.clone().detach().requires_grad_(True), rather than torch.tensor(
sourceTensor).
  t = torch.tensor(t).to(device)
```

In [54]:
```
# SANITY CHECK FOR UnetBottleNeck (Colored Images)
in_channels_arg = 3
out_channels_arg = 1
batch_size = 32
height = 28
width = 28

    # Check if CUDA is available # USING APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() el
se "cpu")

# Positional Encoding Object
timesteps_to_display = torch.randint(0, total_timesteps, (batch_siz
e,), device=device).long().tolist()

# Sanity check
x = torch.randn((batch_size, in_channels_arg, height, width)).to(de
vice)
model = UNet(in_channels=in_channels_arg, out_channels=out_channels
_arg)
model = model.to(device)

y = model.forward(x=x, t = torch.tensor(timesteps_to_display).to(to
rch.float32).to(device).view(-1,1))
assert len(x.shape) == len(y.shape)
assert y.shape == (batch_size, out_channels_arg, height, width)

print("Sanity Check for Multi-channel or colored Images")
print("Shape of the input : ", x.shape)
print("Shape of the output : ", y.shape)
```

```
Sanity Check for Multi-channel or colored Images
Shape of the input :  torch.Size([32, 3, 28, 28])
Shape of the output :  torch.Size([32, 1, 28, 28])

/var/folders/dn/jhd9djgd393gp0dk2lz40m2c0000gp/T/ipykernel_9563/30
08510448.py:60: UserWarning: To copy construct from a tensor, it i
s recommended to use sourceTensor.clone().detach() or sourceTenso
r.clone().detach().requires_grad_(True), rather than torch.tensor(
sourceTensor).
  t = torch.tensor(t).to(device)
```

In [55]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_
grad)

num_params = count_parameters(model)
print(f"The model has {num_params:,} trainable parameters.")
```

```
The model has 145,233 trainable parameters.
```

Train the Model

In the following block, the train function is defined. You have to calculate the noisy data, feed forward through the model and pass the predicted noise and true noise to the criterion to calculate the loss.

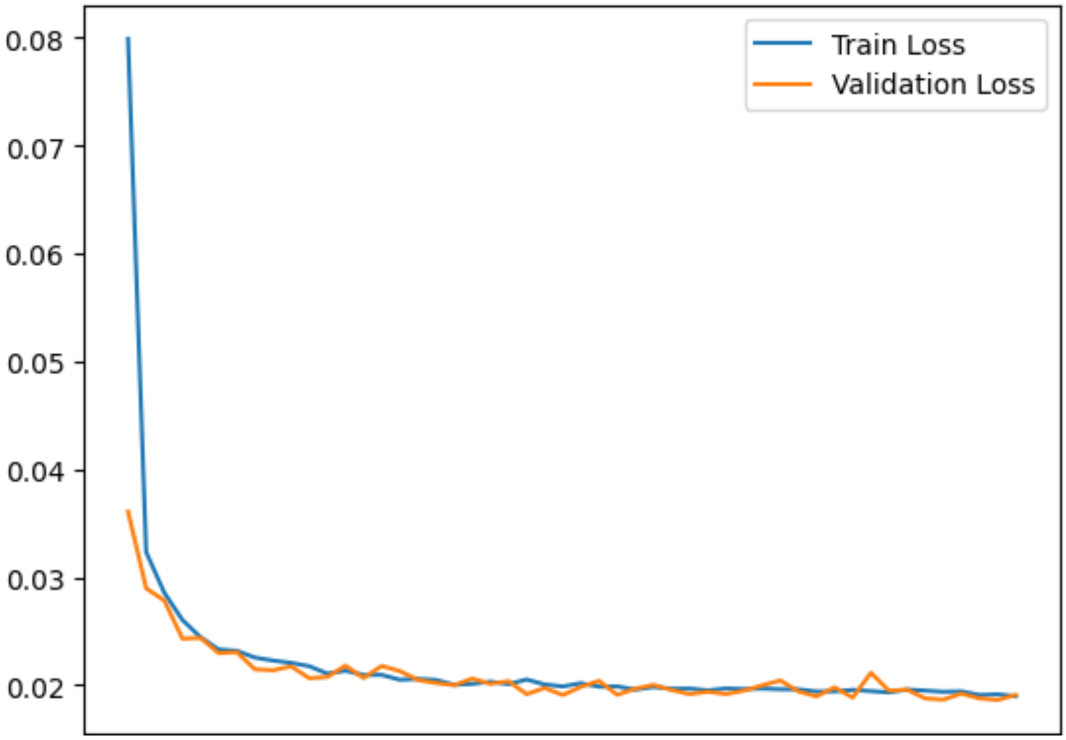
In [41]:
```
from tqdm import tqdm

def train(model, train_loader, val_loader, optimizer, criterion, de
vice, num_epochs, diffuser, totalTrainingTimesteps):
    """
    model: Object of Unet Model to train
    train_loader: Training batches of the total data
    val_loader: Validation batches of the total data
    optimizer: The backpropagation technique
    criterion: Loas Function
    device: CPU or GPU
    num_epochs: total number of training loops
    diffuser: NoiseDiffusion class object to perform Forward diffus
ion
    totalTrainingTimesteps: Total number of forward diffusion times
teps the model is to be trained on
    """
    # Check if CUDA is available # USING APPLE SILICON
    device = torch.device("mps" if torch.backends.mps.is_available(
) else "cpu")

    train_losses = []
    val_losses = []

    for epoch in range(num_epochs):
        model.train()
        total_train_loss = 0

        # Wrapping your loader with tqdm to display progress bar
        train_progress_bar = tqdm(enumerate(train_loader), total=le
n(train_loader), desc=f"Epoch {epoch+1}/{num_epochs} [Train]", leav
e=False)
        for batch_idx, (data, _) in train_progress_bar:
            data = data.to(device)

            optimizer.zero_grad()

            # Use a random time step for training
            batch_size = len(data)
            timesteps = torch.randint(0, totalTrainingTimesteps, (b
atch_size,), device=device).long().tolist()

            #######################################################
```

```
####################
            #                                  TO DO
#
            #                        Calculate Noisy data, True noise
#
            #                    and Predicted Noise, & then feed it t
o criterion         #
            ##################################################
####################

            noisy_data, true_noise = diffuser.noise_diffusion(data,
timesteps)
            predicted_noise = model(noisy_data, timesteps)

            loss = criterion(predicted_noise, true_noise.to(device)
)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()
            train_progress_bar.set_postfix({'Train Loss': f'{loss.i
tem():.4f}'})

        avg_train_loss = total_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        # Validation
        model.eval()
        total_val_loss = 0

        # Wrapping your validation loader with tqdm to display prog
ress bar
        val_progress_bar = tqdm(enumerate(val_loader), total=len(va
l_loader), desc=f"Epoch {epoch+1}/{num_epochs} [Val]", leave=False)
        with torch.no_grad():
            for batch_idx, (data, _) in val_progress_bar:
                data = data.to(device)

                # For simplicity, we can use the same random timest
ep for validation
                batch_size = len(data)
                timesteps = torch.randint(0, totalTrainingTimesteps
, (batch_size,), device=device).long().tolist()

                ###################################################
#########################
                #                                  TO DO
#
                #                    Calculate Noisy data, True no
ise                 #
                #                    and Predicted Noise, & then feed
it to criterion         #
                ###################################################
#########################
```

```
                    noisy_data, true_noise = diffuser.noise_diffusion(d
ata, timesteps)
                    predicted_noise = model(noisy_data,timesteps)


                    loss = criterion(predicted_noise, true_noise.to(dev
ice))
                    total_val_loss += loss.item()
                    val_progress_bar.set_postfix({'Val Loss': f'{loss.i
tem():.4f}'})

          avg_val_loss = total_val_loss / len(val_loader)
          val_losses.append(avg_val_loss)

          print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_trai
n_loss:.4f}, Validation Loss: {avg_val_loss:.4f}')



     return train_losses, val_losses
```

In the following code block, initialize the necessary variables and then Execute to train, save model and plot the loss

Just to give you an idea of how loss curve would look like approximately (not necssarily same for everybody), x-axis represents epochs and y-axis represents loss.



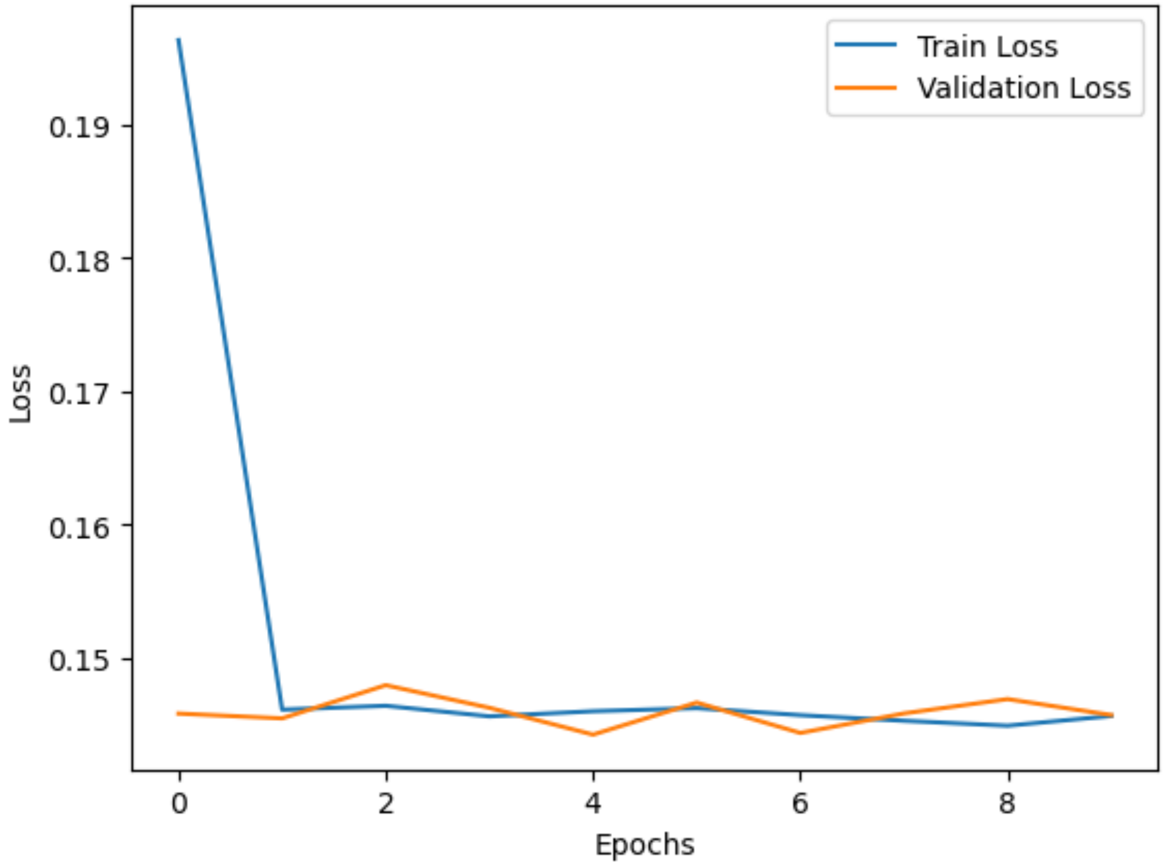
In [48]:
```
################################################################################
#########
```

```
#                                TO DO
#
#                      Initialize the Constants below
#
##################################################################
#########
"""
- `total_time_steps`: Total time steps of forward diffusion
- `start_beta`: Initial point of Noise Level Parameter
- `end_beta`: End point of Noise Level Parameter
- `inputChannels`: 1 for Grayscale Images (Since we're Using MNIST)
- `outputChannels`: How many channels of predicted noise are aiming
for? THINK!
- `num_epochs`: How many epochs are you training for? (*We'd love t
o see best results in minimum epcohs of training*)
"""
# Initialize the Constants
total_timesteps = 100

start_beta, end_beta = 0.0001, 0.01
inputChannels, outputChannels = 1, 1  # Grayscale image input and o
utput channels
num_epochs = 10  # Number of training epochs


# Check if CUDA is available # USING APPLE SILICON
device = torch.device("mps" if torch.backends.mps.is_available() el
se "cpu")

##################################################################
#########
#                                TO DO
#
#                          Initialize the Model
#
#                        Initialize the Optimizer
#
#                      Initialize the Loss Function
#
#                       Initialize the NoiseDiffuser
#
##################################################################
#########

# Initialize UNET model
stableDiffusionModel = UNet(in_channels=inputChannels, out_channels
=outputChannels).to(device)
optimizer = torch.optim.Adam(stableDiffusionModel.parameters(), lr=
0.01, weight_decay= 1e-3)
# Initialize the loss function
criterion = nn.MSELoss()
diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps)
```

```
##################################################################
#########
#                                TO DO
#
#               Execute this Block, Train & Save the Model
#
#                            And Plot the Progress
#
##################################################################
#########
stableDiffusionModel = stableDiffusionModel.to(device)
train_losses, val_losses = train(model= stableDiffusionModel,
                                 train_loader= trainloader,
                                 val_loader= valloader,
                                 optimizer= optimizer,
                                 criterion= criterion,
                                 device= device,
                                 num_epochs= num_epochs,
                                 diffuser= diffuser,
                                 totalTrainingTimesteps=total_times
teps)

# Save the model
torch.save(stableDiffusionModel.state_dict(), 'HW3SDModel.pth')

#Plot the losses
import matplotlib.pyplot as plt
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/10 [Train]:   0%|          | 0/1500 [00:00<?, ?it/s]

Epoch 1/10, Train Loss: 0.1963, Validation Loss: 0.1459

Epoch 2/10, Train Loss: 0.1462, Validation Loss: 0.1455

Epoch 3/10, Train Loss: 0.1465, Validation Loss: 0.1480

Epoch 4/10, Train Loss: 0.1457, Validation Loss: 0.1463

Epoch 5/10, Train Loss: 0.1460, Validation Loss: 0.1443
```

```
Epoch 6/10, Train Loss: 0.1463, Validation Loss: 0.1467

Epoch 7/10, Train Loss: 0.1458, Validation Loss: 0.1444

Epoch 8/10, Train Loss: 0.1453, Validation Loss: 0.1459

Epoch 9/10, Train Loss: 0.1450, Validation Loss: 0.1470

Epoch 10/10, Train Loss: 0.1457, Validation Loss: 0.1458
```



EXERCISE 3 : SAMLING GENERATION

Sampling formula

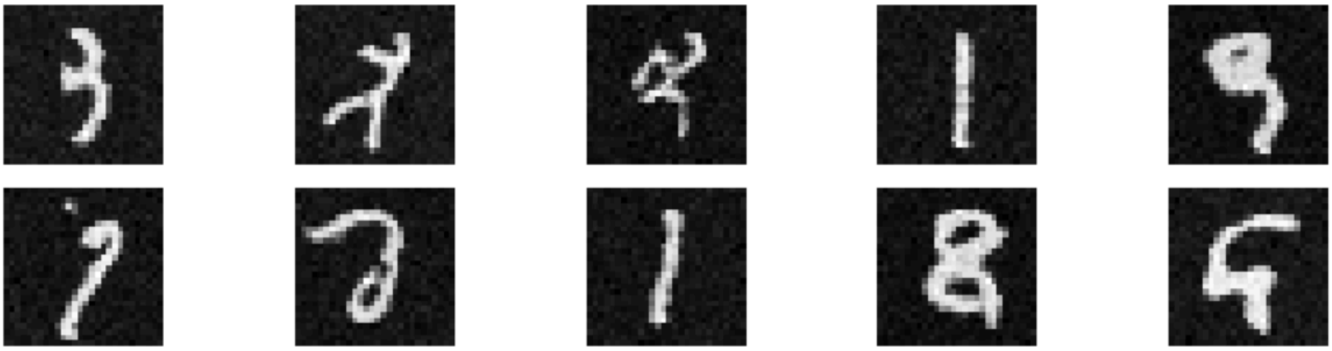
The Stable Diffusion Model sampling code involves generating images from a trained model by iteratively denoising an initial random noise tensor. This process is executed in the reverse manner as compared to the diffusion process, where the noise is incrementally added. The iteration happens for a defined number of timesteps. The goal is to move from a purely noisy state to a clear, denoised state that represents a valid sample from the data distribution learned by the model. Refer to the DDPMs Paper for detailed documentation. The formula for sampling part is as follows:

$$X_{t-1} = \frac{1}{\sqrt{\alpha}} * \left(X_t - \frac{1-\alpha}{\sqrt{(1-\bar{\alpha})}} * \epsilon_t \right) + \sqrt{\beta} * z$$

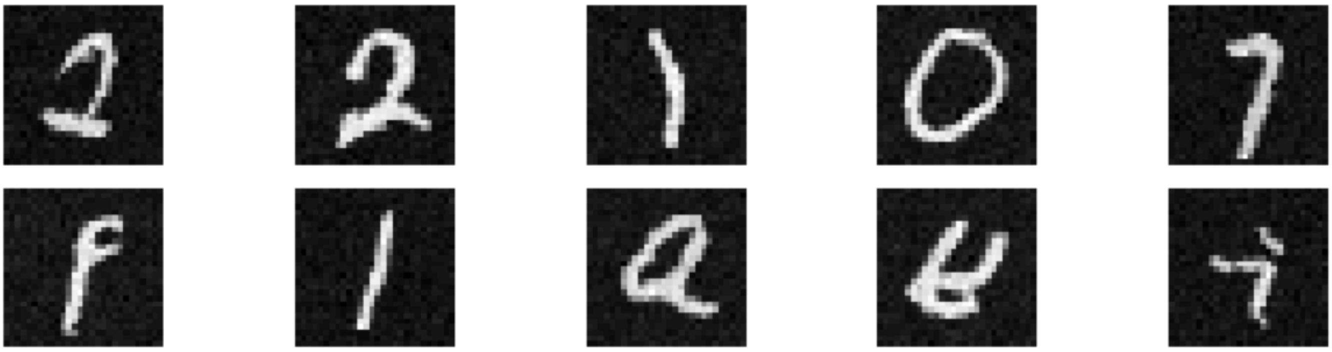
Sample Images

Some sample outputs for random seeds as specified in the code cell of sampling generation and mentioned in the image below are as follows:

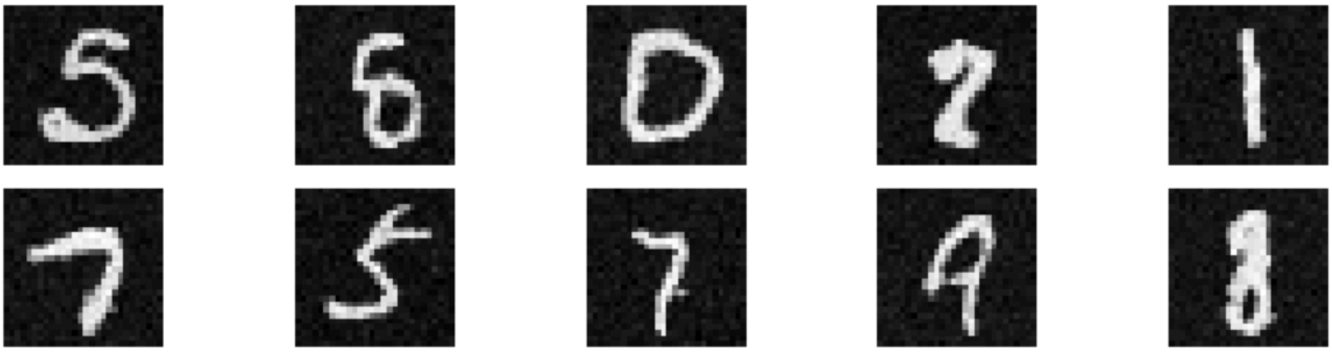
The Outputs for Random Seed {96}



The Outputs for Random Seed {786}



The Outputs for Random Seed {7150}

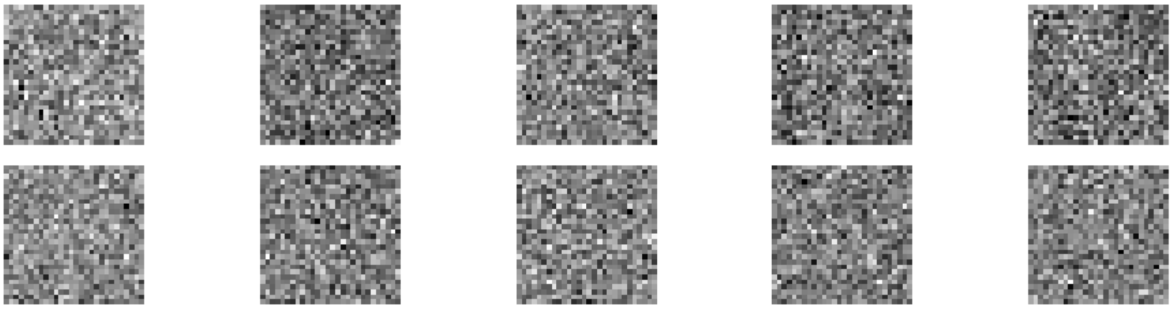


In [56]:
```
def generate_samples(x_t, model, num_samples, total_timesteps, diffuser, device):
    """
    Generate samples using the trained DDPM model.

    Parameters:
    - model: Trained UNetBottleneck model.
    - num_samples: Number of samples to generate.
    - total_timesteps: Total timesteps for the noise process.
    - diffuser: Instance of NoiseDiffuser.
    - device: Computing device (e.g., "cuda" or "cpu").

    Returns:
    - generated_samples: A tensor containing the generated samples.
    """

    #############################################################################
```

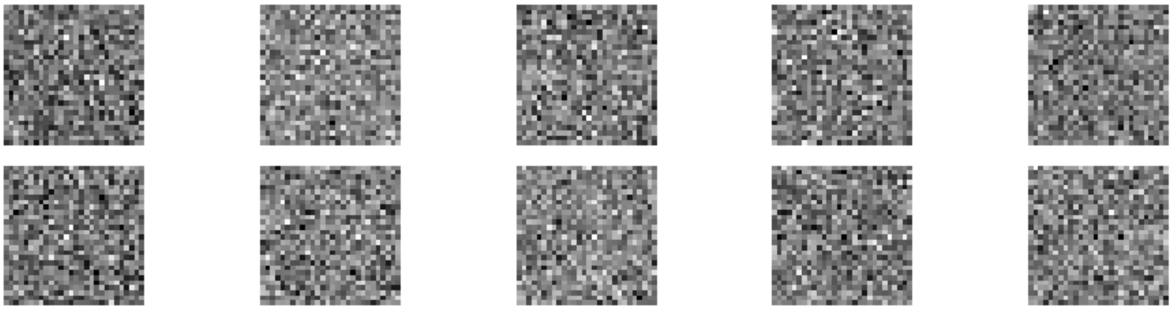
```
    #                                   TO DO
#
    #                    Implement the Sampling Algorithm, start with
#
    #                       pure noise, using the trained model
#
    #                     perform denoising to generate MNIST Images
#
    ################################################################
#############
    # Iterate in reverse order to "denoise" the samples
    for timestep in range(total_timesteps-1, -1, -1):
      one_by_sqrt_alpha = 1 / torch.sqrt(diffuser.alphas[timestep])
      beta_by_sqrt_one_minus_alpha_cumprod = 1-diffuser.alphas[time
step]/torch.sqrt(1-diffuser.alpha_bar[timestep])

      z = torch.randn_like(x_t)
      timestep_list = []
      timestep_list.append(timestep)

      epsilon_t = model.forward(x_t, timestep_list)
      x_t = one_by_sqrt_alpha * (x_t - beta_by_sqrt_one_minus_alpha
_cumprod*epsilon_t) + torch.sqrt(diffuser.betas[timestep])*z
    return x_t.detach()




####################################################################
#########
#                                   TO DO
#
#                  Post Implementation of Sampling Algorithm,
#
#                         Execute the following lines by
#
#             using the same constants (timesteps and beta values)
#
#                          as you used while training,
#
#                 initializing instance of NoiseDiffuser Object
#
#                       and Loading the pretrained model
#
####################################################################
#########
# Create instance of NoiseDiffuser
total_timesteps = 10
start_beta, end_beta = 0.0001, 0.001
diffuser = NoiseDiffuser(start_beta=start_beta, end_beta=end_beta,
total_steps=total_timesteps, device= device)
# Using the function:
model_path = 'HW3SDModel.pth'
```

```
model = UNet(in_channels=inputChannels, out_channels=outputChannels
).to(device)
model.load_state_dict(torch.load(model_path))
model.eval()



SEED = [96, 786, 7150]  # You can set any integer value for the see
d

for S in SEED:
  print("The Outputs for Random Seed {%d}"%S)
  # Set seed for both CPU and CUDA devices
  torch.manual_seed(S)
  if torch.cuda.is_available():
      torch.cuda.manual_seed(S)
      torch.cuda.manual_seed_all(S)
      torch.backends.cudnn.deterministic = True
      torch.backends.cudnn.benchmark = False

  num_samples_to_generate = 10
  # Initialize with random noise
  xt = torch.randn((num_samples_to_generate, 1, 28, 28), device=dev
ice)

  samples = generate_samples(xt, model, num_samples_to_generate, to
tal_timesteps, diffuser, device)

  # Display the generated samples
  display_images(samples, num_samples_to_generate, images_per_row=5
)
```

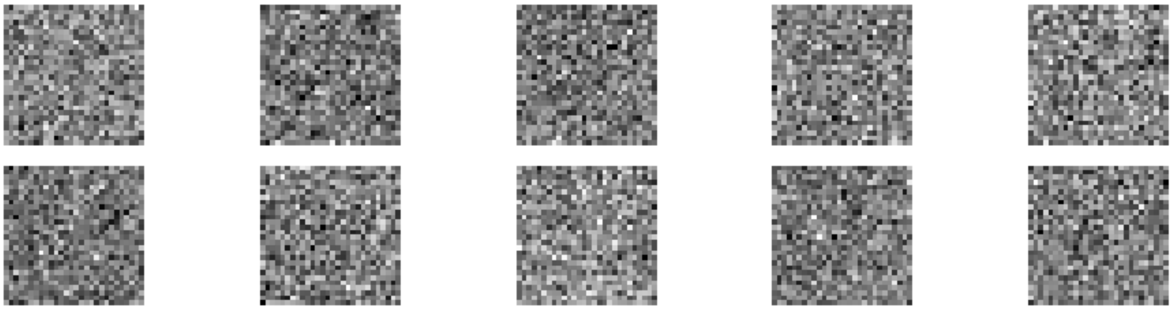
The Outputs for Random Seed {96}



The Outputs for Random Seed {786}



The Outputs for Random Seed {7150}



In [ ]: