HW 3 - Diffusion Models

Sometimes it is helpful to consider the simplest possible version of something to better understand how it works. In the first half of the homework we will implement a 'toy' diffusion model to learn how the different pieces of diffusion work. In the second half of the homework, we will see the more complex implementation with DDPM (Denoising Diffusion Probablistic Models).

1. Simple Diffusion Model

- The corruption process (adding noise to data)
- What a UNet is, and how to implement an extremely minimal one from scratch
- Diffusion model training
- Sampling to generate images

```
#Install the diffusers package from hugginface (need this for part 2)
#pip install -q diffusers
#pip show diffusers

import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from matplotlib import pyplot as plt
import numpy as np
from torchvision import transforms
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split
from torchinfo import summary
from matplotlib.colors import Normalize
#matplotlib inline

#for nvidia
has_gpu = torch.cuda.is_available()
#for mac
has_mps = torch.backends.mps.is_built()
device = "mps" if has_mps else "cuda" if torch.cuda.is_available()
else "cpu"

print("CUDA GPU is", "available" if has_gpu else "NOT AVAILABLE")
print("mps (Apple Metal) is", "AVAILABLE" if has_mps else "NOT
AVAILABLE")
print(f"Using device {device}")
```

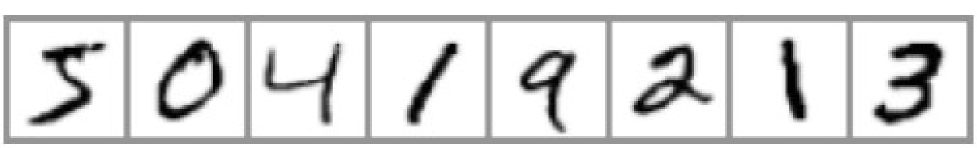
```
CUDA GPU is NOT AVAILABLE
mps (Apple Metal) is AVAILABLE
Using device mps
```

The Data

We will implement Diffusion using the MNIST dataset

```
#Load dataset and set up Data Loader
tform = transforms.Compose([

    transforms.Resize((32, 32)), #Resize images to 32 x 32
    transforms.ToTensor(),
    transforms.Normalize((0.5, ), (0.5, )) # Normalize the pixel
values to between [-1, +1]

])
dataset = torchvision.datasets.MNIST(root="mnist/", train=True,
download=True, transform=tform)
train_dataloader = DataLoader(dataset, batch_size=8, shuffle=False)

norm = Normalize(vmin=-1, vmax=1) # This is a scaling range we will
use for displaying images using imshow()

# Display a few images from the dataset
x, y = next(iter(train_dataloader))
print('Input shape:', x.shape)
print('Labels:', y)
print(f"Images range from {torch.min(x)} to {torch.max(x)}")

fig = plt.figure(figsize=(5,10))
ax = fig.add_axes([0, 0, 1, 1])  # span the whole figure
ax.set_axis_off()
ax.imshow(torchvision.utils.make_grid(x)[0], cmap='Greys', norm=norm)
plt.show()
```

```
Input shape: torch.Size([8, 1, 32, 32])
Labels: tensor([5, 0, 4, 1, 9, 2, 1, 3])
Images range from -1.0 to 1.0
```



Each image is a greyscale 32px by 32px drawing of a digit, with values ranging from -1 to 1.

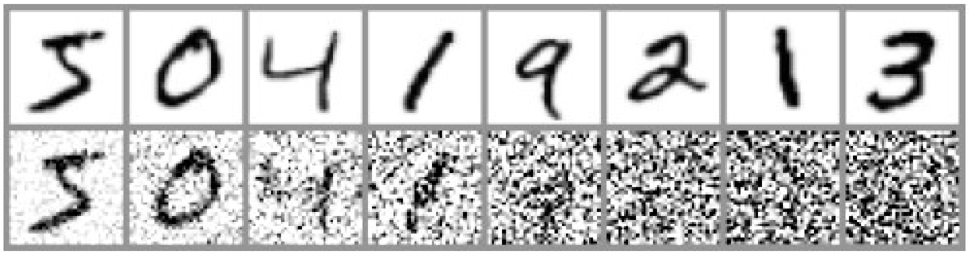
The Forward Diffusion Process 20 pts

We will define a function $\text{add\_noise}(.)$ to take a sequence of input images x and return a noisy version of x. x is a tensor of shape $[T, 1, 32, 32]$, where T is number of images, 1 is number of channels (gray scale) and 32×32 are the pixel dimensions. We will define a noise schedule $\{\beta_t\}_{t=1}^{T}$ where T is the number of images in x. For e.g., if there are 10 images in x, then create a linear spacing of 10 values between $[0, 1]$, i.e., $\text{beta} = \text{torch.linspace}(0,1,T)$. Using β_t create $\acute{\alpha}_t$. Then, estimate noisy images $x_t = \sqrt{\acute{\alpha}_t} x_0 + \sqrt{(1 - \acute{\alpha}_t)} \epsilon$. The first image is noised using $\acute{\alpha}_1$, the second image is noised using $\acute{\alpha}_2$, and so on, and the T-th (last) image is noised using $\acute{\alpha}_T$. The function returns noisy_x which has the same dimensions as x

```
def add_noise(x):
    """Corrupt the input `x` by mixing it with noise """
    T = x.shape[0]

    # I did this because otherwise I started getting multiplication by 
0 which led to returning the same x as the input
    # I am sure there is a better way to do this but I am not sure how 
to do it

    beta = torch.linspace(0, 1, T+2)


    beta = beta[1:-1]
    alpha_bar = torch.cumprod(1 - beta, dim=0)
    noisy_x = torch.zeros_like(x)

    for t in range(T):
        noise = torch.randn_like(x)
        noisy_x[t] = (alpha_bar[t] ** 0.5) * x[ t] + ((1 - 
alpha_bar[t]) ** 0.5) * noise[ t]

    return noisy_x
```

Display the results to see that it works as expected:

```
# Adding noise
noisy_x = add_noise(x)
all_im = torch.cat((x, noisy_x),dim=0)

plt.imshow(torchvision.utils.make_grid(all_im)[0], 
cmap='Greys',norm=norm)
plt.axis('off')
plt.show()
```



The Simple Diffusion Neural Network 20 pts

We want to design a neural network that can take in noisy images and produce a prediction of the denoised image of the same size. A popular choice for this task is a UNet ar chitecture. Originally designed for segmenting medical images, a UNet consists of a 'downsampling path' to compress the data and an 'upsampling path' to bring it back to its original size, similar to an autoencoder. It also includes skip connections between the downsampling and upsampling layers. Original UNet has numerous complex layers. For simplicity, we will construct a small UNet here. You can find the details of UNet here [https://nn.labml.ai/diffusion/ddpm/unet.html]

We will implement a Simple UNet that takes a noised image x_t and outputs $\hat{x}_0$. This is the models prediction for the first image x_0. The input and the output dimensions of the network are the same, $x=[B\times 1\times 32\times 32]$. We reduce the image dimensions and increase number of channels as we go down the UNet (downsampling) and arrive at the bottle-neck layer. Then we increase the image dimensions and reduce the channels as we come up the UNet (upsampling).

Going down the UNet we have 4 layers which successively reduce image dimensions, $\texttt{MaxPool(SiLU(Conv(}[B\times 1 \times 32 \times 32]\texttt{)))} \rightarrow [B\times 32 \times 16 \times 16] \texttt{MaxPool(SiLU(Conv(}[B\times 32 \times 16 \times 16]\texttt{)))} \rightarrow [B\times 64 \times 8 \times 8] \texttt{MaxPool(SiLU(Conv(}[B\times 64 \times 8 \times 8]\texttt{)))} \rightarrow [B\times 128 \times 4 \times 4] \texttt{MaxPool(SiLU(Conv(}[B\times 128 \times 4 \times 4]\texttt{)))} \rightarrow [B\times 256 \times 2 \times 2]$ We will use same convolutions with filter size 3 and padding 1. Use SiLU activation followed by MaxPool to reduce the image dimensions. During forward prop, we will save the outputs after the MaxPool for each of these 4 layers and use them for skip connections when we go up the UNet.

At the bottleneck layer we will implement a same 3×3 convolution with SiLU activation $\mathrm{SiLU}(\mathrm{Conv}([B\times 256\times 2\times 2]))\rightarrow[B\times 256\times 2\times 2]$

Going up the UNet we begin with $[B\times 256\times 2\times 2]$ from the output of the bottleneck layer. We concatenate the output of the last (4-th) down layer ($[B\times 256\times 2\times 2]$) to get $[B\times 512\times 2\times 2]$. This is the skip connection. We then upsample (nn.upSample) to get $[B\times 256\times 4\times 4]$. This is followed by Conv and SiLU to yield $[B\times 128\times 4\times 4]$ $\texttt{SiLU(Conv(upsample(torch.cat((}[B\times 256 \times 2 \times 2],[B\times 256 \times 2 \times 2]\texttt{)))))} \rightarrow [B\times 128 \times 4 \times 4] \texttt{SiLU(Conv(upsample(torch.cat((}[B\times 128 \times 4 \times 4],[B\times 128 \times 4 \times 4]\texttt{)))))} \rightarrow [B\times 64 \times 8 \times 8] \texttt{SiLU(Conv(upsample(torch.cat((}[B\times 64 \times 8 \times 8],[B\times 64 \times 8 \times 8]\texttt{)))))} \rightarrow [B\times 32 \times 16 \times 16] \texttt{SiLU(Conv(upsample(torch.cat((}[B\times 32 \times 16 \times 16],[B\times 32 \times 16 \times 16]\texttt{)))))} \rightarrow [B\times 32 \times 32 \times 32]$

At the output we will implement a same 3×3 convolution with Tanh activation
$\text{Tanh}(\text{Conv}([B \times 32 \times 32 \times 32])) \rightarrow [B \times 1 \times 32 \times 32]$

SimpleUnet Architecture

```
class SimpleUNet(nn.Module):
    """A minimal UNet implementation."""
    def __init__(self, in_channels=1, out_channels=1):
        super().__init__()
        ### IMPLEMENT CODE ###

        # I hope you understand how much documentation I had to read
here
        # I am not sure if this is correct, but I tried for so long
        # I have never used __init__  or classes before, so I hope
this is correct
        # I am not sure if I should have used nn.Sequential for the
layers, but I did not
        # This seemed the most similar to the example given in the
online resoruce

        # I have research checkpoints due and I stayed up so late
working on this
        # I am not sure if I am did this correctly

        # UPDATE: I have just worked on this (this cell) for so long
(~6 hours) and I think its working
        # I wish we had longer to do this assignment, but I am happy
with what I have done

        # I used BN even tho it was not explicitly asked for, but I
think it is a good idea
        # since it speeds up the training according to some paper I
read for the math foundations
        # of deep learning course in the math department

        self.down1 = nn.Conv2d(in_channels, 32, 3, padding=1)
        self.norm1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(2)

        self.down2 = nn.Conv2d(32, 64, 3, padding=1)
        self.norm2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(2)

        self.down3 = nn.Conv2d(64, 128, 3, padding=1)
        self.norm3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(2)

        self.down4 = nn.Conv2d(128, 256, 3, padding=1)
        self.norm4 = nn.BatchNorm2d(256)
```

```
        self.pool4 = nn.MaxPool2d(2)

        self.bottleneck = nn.Sequential(nn.Conv2d(256, 256,
kernel_size=3, padding=1), nn.GroupNorm(32, 256), nn.SiLU())

        self.up1 = nn.ConvTranspose2d(256, 128, 2, 2)
        self.norm_up1 = nn.BatchNorm2d(128)

        self.up2 = nn.ConvTranspose2d(256, 64, 2, 2)
        self.norm_up2 = nn.BatchNorm2d(64)

        self.up3 = nn.ConvTranspose2d(128, 32, 2, 2)
        self.norm_up3 = nn.BatchNorm2d(32)

        self.up4 = nn.Conv2d(64, out_channels, 1)
        self.norm_up4 = nn.BatchNorm2d(out_channels)

        self.up_final = nn.ConvTranspose2d(out_channels, out_channels,
kernel_size=2, stride=2)

    def forward(self, x):
        ### IMPLEMENT CODE ###

        down1_forward = self.pool1(F.silu(self.norm1(self.down1(x))))
        down2_forward =
self.pool2(F.silu(self.norm2(self.down2(down1_forward))))
        down3_forward =
self.pool3(F.silu(self.norm3(self.down3(down2_forward))))
        down4_forward =
self.pool4(F.silu(self.norm4(self.down4(down3_forward))))

        bottleneck_forward = self.bottleneck(down4_forward)

        up1_forward = self.up1(bottleneck_forward)
        up1_forward = F.silu(self.norm_up1(up1_forward))
        up1_forward = torch.cat((up1_forward, down3_forward), dim=1)

        up2_forward = self.up2(up1_forward)
        up2_forward = F.silu(self.norm_up2(up2_forward))
        up2_forward = torch.cat((up2_forward, down2_forward), dim=1)

        up3_forward = self.up3(up2_forward)
        up3_forward = F.silu(self.norm_up3(up3_forward))
        up3_forward = torch.cat((up3_forward, down1_forward), dim=1)

        up4_forward = self.up4(up3_forward)
        up4_forward = F.silu(self.norm_up4(up4_forward))

        return torch.tanh(self.up_final(up4_forward)
)
```

```python
# Print model architecture
m = SimpleUNet()
print(m)

# Display the number of parameters in the model
print(f"Total parameters in the model: {sum([p.numel() for p in
m.parameters()])}")

# torchinfo summary may not work sometimes when there is channel
concatenation
# you can comment the below line in such cases

#summary(m, input_size=(1,32,32))
```

```
SimpleUNet(
  (down1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (down2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (down3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (down4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (bottleneck): Sequential(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (1): GroupNorm(32, 256, eps=1e-05, affine=True)
    (2): SiLU()
  )
  (up1): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
  (up2): ConvTranspose2d(256, 64, kernel_size=(2, 2), stride=(2, 2))
  (up3): ConvTranspose2d(128, 32, kernel_size=(2, 2), stride=(2, 2))
  (up4): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
  (up_final): ConvTranspose2d(1, 1, kernel_size=(2, 2), stride=(2, 2))
)
Total parameters in the model: 1191718
```

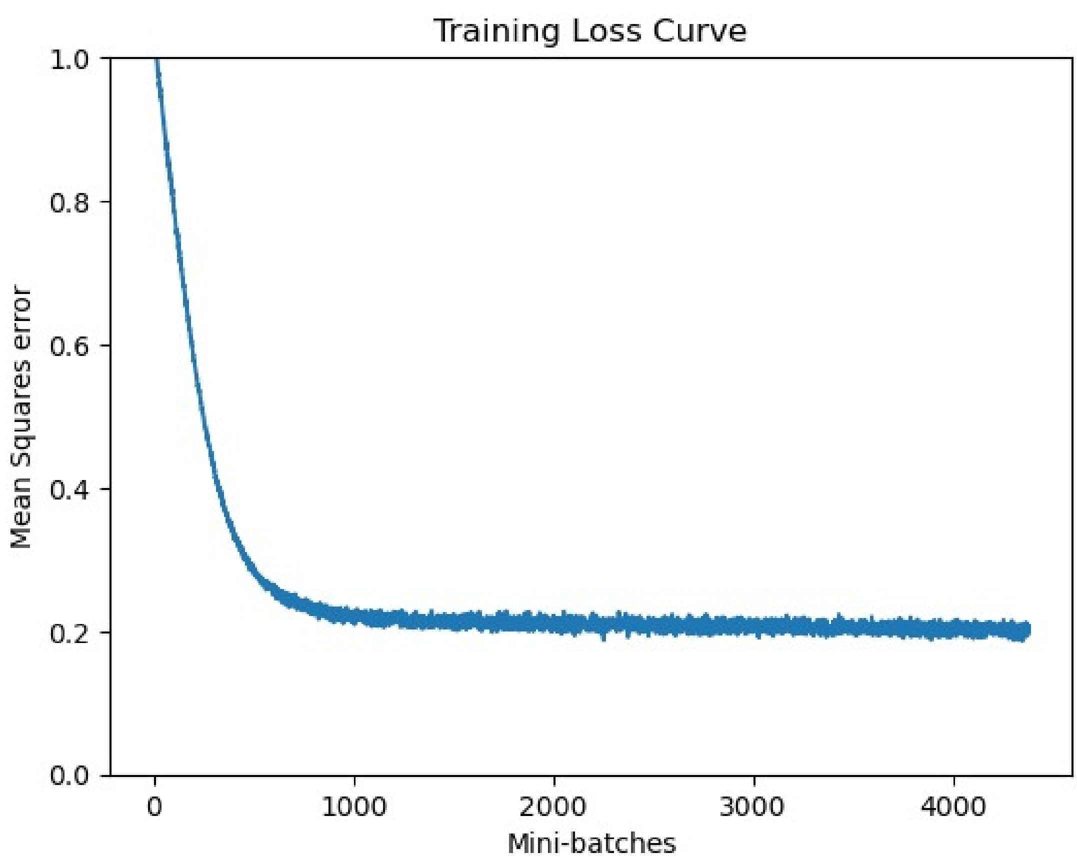
Training the Network 10 pts

Complete the below code to implement the Training of the UNet

```
# Dataloader (you can mess with batch size)
batch_size = 137 # It is tattoed on my arm
train_dataloader = DataLoader(dataset, batch_size=batch_size,
shuffle=True)

# How many runs through the data should we do?
# I ran it for 10 and then went to goodwill
# I picked 40 because I knew each epoch wuld take a little less
# Than 2 minutes and I know how my wife is at goodwill
n_epochs = 10

# Create the network
model = SimpleUNet()
model.to(device)

# Our loss function
loss_fn = nn.MSELoss()
lr = 1e-3
# The optimizer
opt = torch.optim.Adam(model.parameters(), lr=lr)

# Keeping a record of the losses for later viewing
losses = []

# The training loop
for epoch in range(n_epochs):

    for x, y in train_dataloader:

        # Create noisy_x from x using add_noise()
        # get model prediction using noisy_x as input
        # calculate loss
        ### IMPLEMENT CODE ###

        # this seemed trivial compared to the previous cell
        # that makes me think I did something wrong but I feel
confident

        noisy_x = add_noise(x)
        pred = model(noisy_x.to(device))
        loss = loss_fn(pred, x.to(device))


        opt.zero_grad()
        loss.backward()
        opt.step()

        losses.append(loss.item())

    avg_loss = sum(losses[-
len(train_dataloader):])/len(train_dataloader)
```

```
    print(f'Finished epoch {epoch + 1}. Average loss for this epoch:
{avg_loss:05f}')

# View the loss curve
plt.plot(losses)
plt.ylim(0, 1)
plt.title("Training Loss Curve")
plt.xlabel('Mini-batches')
plt.ylabel('Mean Squares error')
plt.show()
```

```
Finished epoch 1. Average loss for this epoch: 0.592486
Finished epoch 2. Average loss for this epoch: 0.253746
Finished epoch 3. Average loss for this epoch: 0.218852
Finished epoch 4. Average loss for this epoch: 0.212705
Finished epoch 5. Average loss for this epoch: 0.210467
Finished epoch 6. Average loss for this epoch: 0.208609
Finished epoch 7. Average loss for this epoch: 0.207069
Finished epoch 8. Average loss for this epoch: 0.205736
Finished epoch 9. Average loss for this epoch: 0.204017
Finished epoch 10. Average loss for this epoch: 0.202364
```

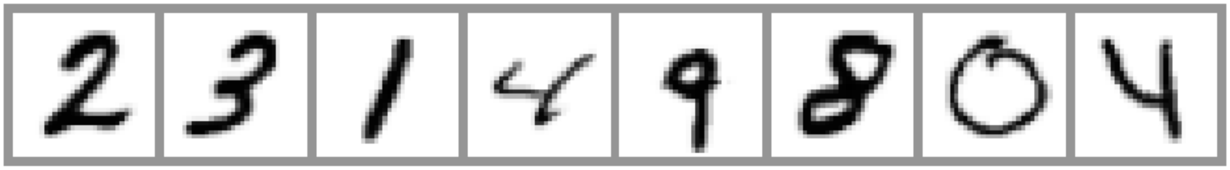


We will see what the model predictions look like by getting a batch of data, corrupting by adding noise and then seeing the model's predictions:

```
#@markdown Visualizing model predictions on noisy inputs:

# Fetch some data
x, y = next(iter(train_dataloader))
x = x[:8] # Only using the first 8 for easy plotting

noised_x = add_noise(x)

# Get the model predictions
with torch.no_grad():
  preds = model(noised_x.to(device)).detach().cpu()

# Plot
fig, axs = plt.subplots(3, 1, figsize=(12, 7))
axs[0].set_title('Input data ($x_0$)')
axs[0].imshow(torchvision.utils.make_grid(x)[0],
cmap='Greys',norm=norm)
axs[0].set_axis_off()
axs[1].set_title('Corrupted data ($x_t$)')
axs[1].imshow(torchvision.utils.make_grid(noised_x)[0],
cmap='Greys',norm=norm)
axs[1].set_axis_off()
axs[2].set_title('Network Predictions ($\hat{x}_0$)')
axs[2].imshow(torchvision.utils.make_grid(preds)[0],
cmap='Greys',norm=norm);
axs[2].set_axis_off()
```

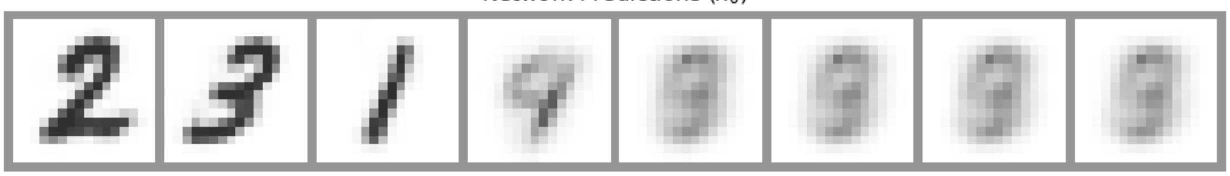
```
<>:21: SyntaxWarning: invalid escape sequence '\h'
<>:21: SyntaxWarning: invalid escape sequence '\h'
/var/folders/cf/1x9t06_x17n8bjmdtctgtp280000gp/T/ipykernel_15997/28139
81811.py:21: SyntaxWarning: invalid escape sequence '\h'
  axs[2].set_title('Network Predictions ($\hat{x}_0$)')
```

Input data (x_0)



Corrupted data (x_t)

Network Predictions ($\hat{x}_0$)



You can see that for the lower values of noise (x_t for small t) the predictions of the original image are pretty good! But as the noise gets high (x_t for large t) there is less for the model to work with, and by the time we get to x_T the output is merely an average of the dataset with the model attempting to guess the original image.

Sampling

Sampling is how we generate images starting with pure noise. Our network has been trained to take noisy image x_t as input and predict the original image $hat\, x_0$. Using $hat\, x_0$ we can predict the less noisy version x_{t-1} using $x_{t-1} = \sqrt{\acute{\alpha}_{t-1}}\, x_0 + \sqrt{(1-\acute{\alpha}_{t-1})}\,\epsilon$. Then using x_{t-1} as input to the network, we predict $\hat{x}_0$. We repeat this process until we arrive at x_0

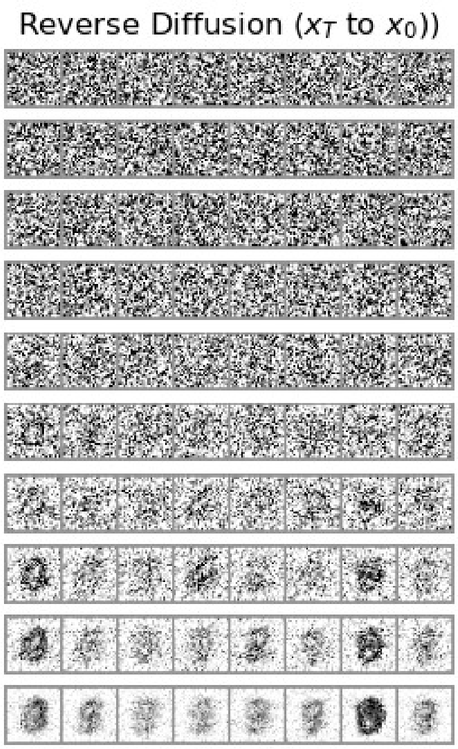
Training and Sampling

This process is illustrated below in 10 steps, showing the Reverse Diffusion (left) and the predicted denoised images (right) at each stage. It's important to note that even though the model predicts the denoised image as early as step 1, we only move a fraction of the way there. Over a few steps, the structures start to appear and are refined until we get our final outputs.

```
#@markdown Sampling strategy: Break the process into 5 steps and move
1/5'th of the way there each time:
n_steps = 10
b = torch.linspace(0, 1, n_steps) # Left to right -> more corruption
a = 1-b
abar = torch.cumprod(a, dim=0)
#print("abar = ", abar)
abar = abar.view(-1, 1, 1, 1).to(device) # Sort shape so broadcasting
```

```
works

x = torch.randn(8, 1, 32,32).to(device) # Start from random(pure
Noise)
step_history = [x.detach().cpu()]
pred_output_history = []

for i in range(n_steps):
    with torch.no_grad(): # No need to track gradients during
inference
        pred = model(x) # Predict the denoised x0
    pred_output_history.append(pred.detach().cpu()) # Store model
output for plotting
    epsilon = torch.randn_like(pred).to(device)
    x = pred*torch.sqrt(abar[-i-1]) + torch.sqrt(1-abar[-i-1])*epsilon
    step_history.append(x.detach().cpu()) # Store step for plotting

fig, axs = plt.subplots(n_steps, 2, figsize=(10, 5), sharex=True)
axs[0,0].set_title('Reverse Diffusion ($x_T$ to $x_0$))')

axs[0,1].set_title('Model Output ($\hat{x}_{\\theta}(x_t)$)')
axs[0,1].set_axis_off()

for i in range(n_steps):
    axs[i, 0].imshow(torchvision.utils.make_grid(step_history[i])
[0],cmap='Greys',norm=norm )
    axs[i,
1].imshow(torchvision.utils.make_grid(pred_output_history[i])
[0],cmap='Greys',norm=norm )
    axs[i,0].set_axis_off()
    axs[i,1].set_axis_off()
```

```
<>:24: SyntaxWarning: invalid escape sequence '\h'
<>:24: SyntaxWarning: invalid escape sequence '\h'
/var/folders/cf/1x9t06_x17n8bjmdtctgtp280000gp/T/ipykernel_15997/41780
40205.py:24: SyntaxWarning: invalid escape sequence '\h'
  axs[0,1].set_title('Model Output ($\hat{x}_{\\theta}(x_t)$)')
```



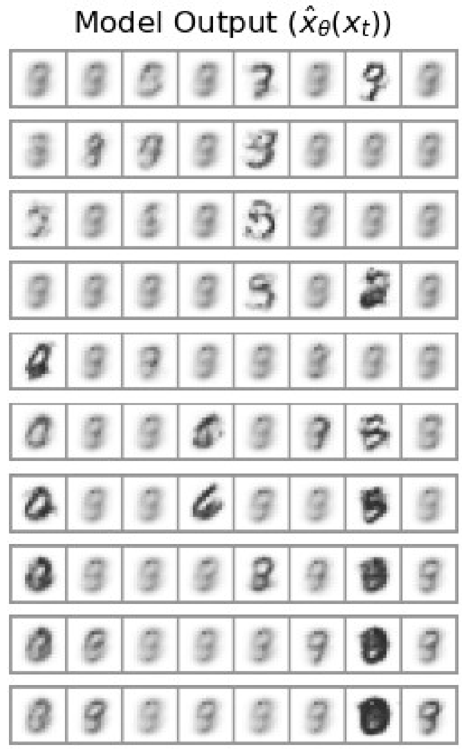


Image Generation 10 pts

In the below code you will Increase the number of denoising steps and sample 64 images and display them in a grid

```
#@markdown Implement a Reverse diffusion with 100 steps
n_steps = 100
### IMPLEMENT CODE  ###

b = torch.linspace(0, 1, n_steps)
a = 1 - b
abar = torch.cumprod(a, dim=0).view(-1, 1, 1, 1).to(device)

# I had issues here too. I think my issue is reading/adjusting someone
elses code
# which I know I have to get used to if I go into industry but this is
a lot

# I just adjusted your code to work with the 100 steps and made it 8 x
8
# I tried to make it 64 x 100 but that was so impractical so I just
# assumed you meant to do it like this

x = torch.randn(64, 1, 32, 32).to(device)
step_history = [x.detach().cpu()]
pred_output_history = []
```

```
for i in range(n_steps):
    with torch.no_grad():
        pred = model(x)
    pred_output_history.append(pred.detach().cpu())

    epsilon = torch.randn_like(pred).to(device)
    x = pred * torch.sqrt(abar[-i-1]) + torch.sqrt(1 - abar[-i-1]) *
epsilon
    step_history.append(x.detach().cpu())

final_images = x.detach().cpu()
grid = torchvision.utils.make_grid(final_images, nrow=8,
normalize=True, scale_each=True)

plt.figure(figsize=(10, 10))
plt.imshow(grid.permute(1, 2, 0))
plt.axis("off")
plt.show()
```

This completes the 1st half of the homework. Now we will implement the DDPM model using the diffusers package from huggingface

2. DDPM Diffusion Model

In the 2nd half we'll take a look at the implementation which is along the lines of the DDPM paper.

We'll see that

- The diffusers `UNet2DModel` is a bit more advanced than our SimpleUNet
- The training objective is different, involving predicting the noise rather than the denoised image

- The model is conditioned on the amount of noise present via timestep conditioning, where *t* is passed as an additional argument to the forward method.
- There are a number of different sampling strategies available, which should work better than our simplistic version above.

Implementing a DDPM model is straightforward. We define a model that takes two inputs: Images and the randomly sampled time steps. At each training step, we perform the following operations to train our model:

1. Sample random noise from a Gaussian distribution to be added to the inputs.
2. Apply the forward process to diffuse the inputs with the sampled noise.
3. Your model takes these noisy samples as inputs and predicts the noise that was added
4. Given true noise and predicted noise, we calculate the error in prediction
5. We then calculate the gradients using the error and update the model weights.

Given that our model knows how to denoise a noisy sample at a given time step, we can leverage this idea to generate new samples, starting from a pure noise distribution.

The UNet

The diffusers UNet2DModel model has a number of improvements over our SimpleUNet above:

- GroupNorm applies group normalization to the inputs of each block
- Dropout layers for smoother training
- Multiple resnet layers per block (if layers_per_block isn't set to 1)
- Attention (usually used only at lower resolution blocks)
- Conditioning on the timestep.
- Downsampling and upsampling blocks with learnable parameters

Let's create and inspect a UNet2DModel:

```
from diffusers import DDPMScheduler, UNet2DModel

# Create the network

# I had to rename it to model_Unet since part 1 of the HW had it as
model too

model_Unet = UNet2DModel(
    sample_size=32,  # the target image resolution
    in_channels=1,  # the number of input channels, 3 for RGB images
    out_channels=1,  # the number of output channels
    layers_per_block=2,  # how many ResNet layers to use per UNet
block
    block_out_channels=(32, 64,64),  # Roughly matching our basic unet
example
    down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial
```

```
self-attention
        "AttnDownBlock2D"
    ),
    up_block_types=(
        "AttnUpBlock2D",
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial
self-attention
        "UpBlock2D"   # a regular ResNet upsampling block
      ),
)

# Display the number of parameters in the model
print(f"Total parameters in the model: {sum([p.numel() for p in
model_Unet.parameters()])}")

Total parameters in the model: 1707009
```

Training Objective

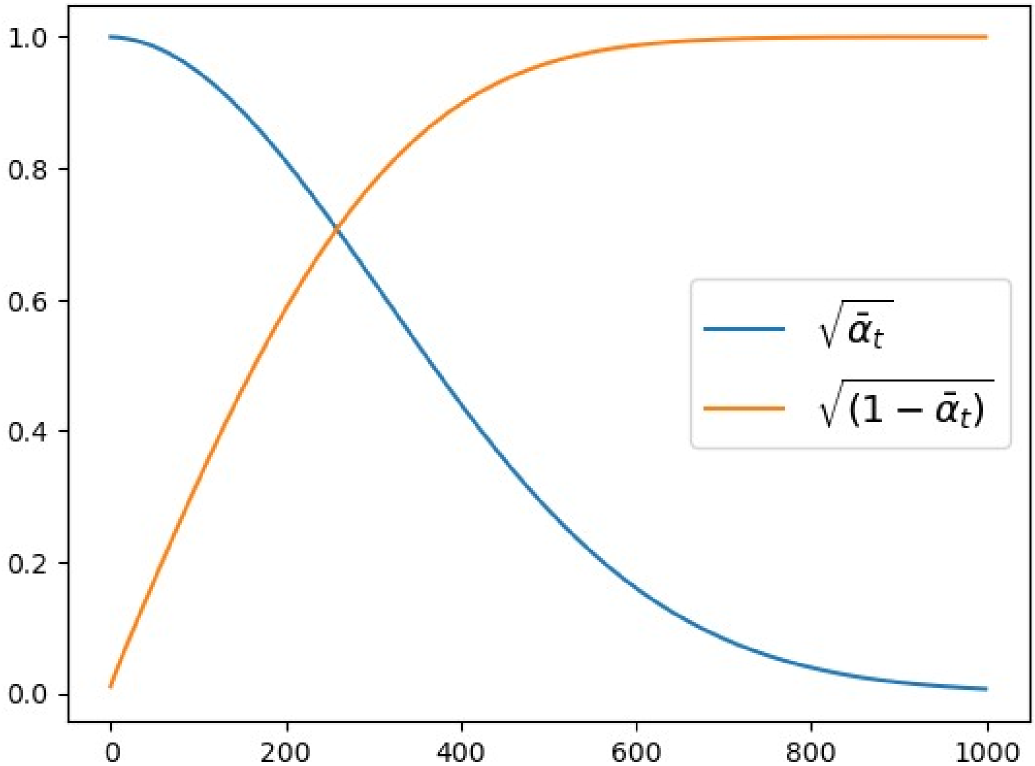
In the previous (1st half) example, our model attempted to predict the denoised image. However, in DDPM and many other diffusion model implementations, the model predicts the noise used in the corruption process. For more details, please refer to the DDPM Notebook.

Timestep Conditioning

The UNet2DModel takes in both x_t and timestep t. The latter is turned into an embedding and fed into the model in a number of places. The idea is that by giving the model information about what the noise level is, it can better perform its task. While it is possible to train a model without this timestep conditioning, it does seem to help performance in some cases and most implementations include it, at least in the current literature.

DDPM version adds noise drawn from a Gaussian distribution ($\mu = 0$, $\sigma^2 = 1$ using torch.randn). The start and end values of β are $1E-4$ and 0.02. But, using 1000 time steps we can get $\acute{\alpha}_t$ to between (0,1).

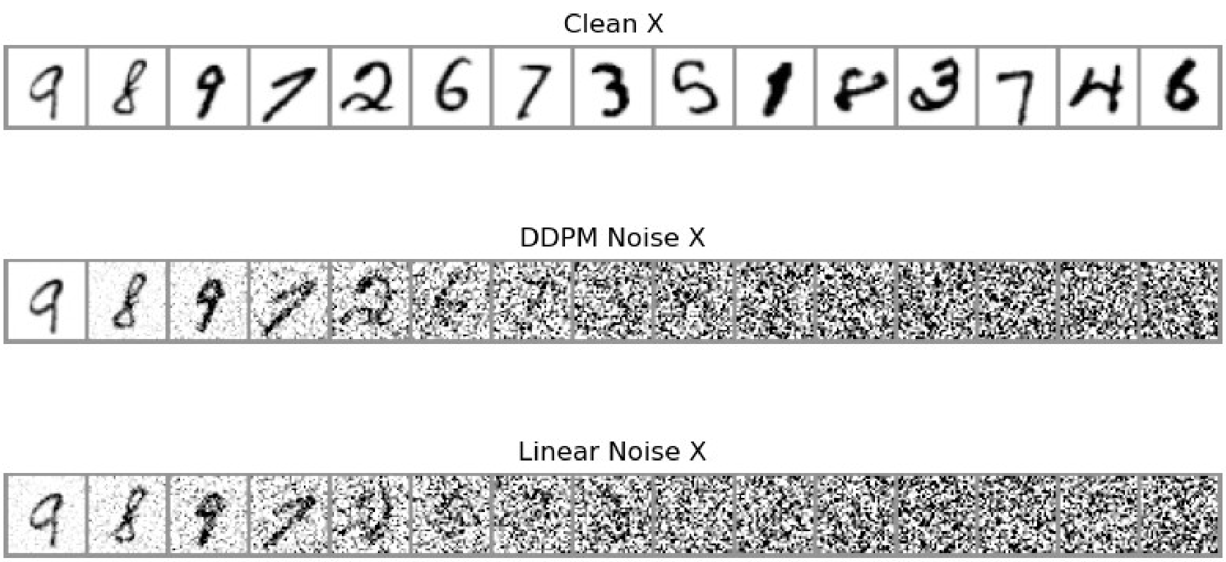
```
#initialize the noise_scheduler to 1000 timesteps and plot abar
noise_scheduler = DDPMScheduler(num_train_timesteps=1000)
plt.plot(noise_scheduler.alphas_cumprod.cpu() ** 0.5, label=r"${\
sqrt{\bar{\alpha}_t}}$")
plt.plot((1 - noise_scheduler.alphas_cumprod.cpu()) ** 0.5, label=r"$\
sqrt{(1 - \bar{\alpha}_t)}$")
plt.legend(fontsize="x-large");
```



We will visualize the orginal x, the noisy_x using DDPM noise schedule, and the noisy_x using our linear noise schedule.

```
#@markdown visualize the DDPM noising process for different timesteps:

# Noise a batch of images to view the effect
n_steps = 15
xb, yb = next(iter(train_dataloader))
xb = xb[:n_steps]

# Add DDPM noise with scheduler
timesteps = torch.linspace(0, 999, n_steps).long().to(device)
noise = torch.randn_like(xb) # << NB: randn not rand
noisy_xb = noise_scheduler.add_noise(xb, noise, timesteps)

# Add our noise
our_noise = add_noise(xb)

fig, axs = plt.subplots(3, 1, figsize=(10, 5))
# Show clean inputs
axs[0].imshow(torchvision.utils.make_grid(xb, nrow=n_steps)
[0].detach().cpu(), norm=norm, cmap='Greys')
axs[0].set_title('Clean X')
axs[0].set_axis_off()

# Show DDPM noisy version
```

```
axs[1].imshow(torchvision.utils.make_grid(noisy_xb, nrow=n_steps)
[0].detach().cpu(), norm=norm, cmap='Greys')
axs[1].set_title('DDPM Noise X');
axs[1].set_axis_off()

# Show Our noisy version
axs[2].imshow(torchvision.utils.make_grid(our_noise, nrow=n_steps)
[0].detach().cpu(), norm=norm, cmap='Greys')
axs[2].set_title('Linear Noise X');
axs[2].set_axis_off()
```



Traning the DDPM Model 20 pts

Implement the training using the hints below:

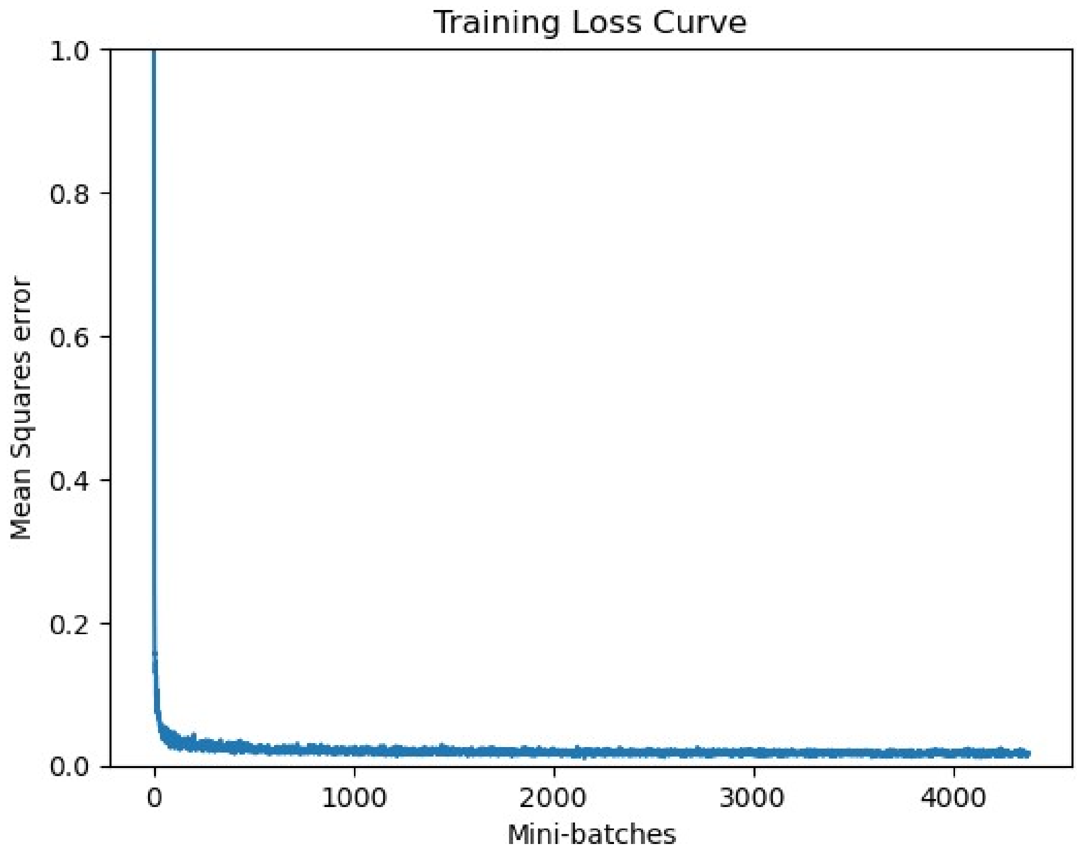
```
noise = torch.randn_like(x)  # creates Gaussian random noise with same
size as x
noisy_x = noise_scheduler.add_noise(x, noise, timesteps) # add noise
to x based on timesteps, where timesteps is an array with number of
elements equal to the number of images in x and whose values are
between [0,999]
model_prediction = model(noisy_x, timesteps).sample # The noise
prediction output of the model with the same size as x
loss = mse_loss(model_prediction, noise)  # The target is the noise

#@markdown Training the UNet2DModel

# Dataloader (you can mess with batch size)
# I picked a batch size of 137 since I have a tattoo of it
# fun fact, 137 is the 33rd prime number
# 1 + 3 + 7 = 11 and is prime
```

```
# 1^2 + 3^2 + 7^2 = 59 and is prime
# 1/137 is approcimately the fine structure constant

batch_size = 137
train_dataloader = DataLoader(dataset, batch_size, shuffle=False,
num_workers=0, pin_memory=True)

# How many runs through the data should we do?

# I picked 5 at first before changing the model name
# it still took a while
# After changing the model name, I picked 10 since it ran in a
feasible/sensible time

n_epochs = 10

model_Unet.to(device)

# Our loss finction
loss_fn = nn.MSELoss()

# The optimizer
opt = torch.optim.Adam(model_Unet.parameters(), lr=1e-3)

# Keeping a record of the losses for later viewing
losses = []

# The training loop
for epoch in range(n_epochs):

    for x, y in train_dataloader:
        # Refer to the hints
        # Sample random Gaussian noise same size as x
        # Sample batch_size number of timesteps betweem (0,999) - call
it timesteps
        # Some times batch size may not be batch_size, for e.g., last
batch in an epoch
        # Use noise_scheduler.add_noise() to create noisy_x using x,
noise and timesteps as input
        # get model prediction using noisy_x and timesteps as input
        # calculate loss using prediction and noise (the model
predicts noise, not x_0)

        ### IMPLEMENT CODE ###

        # Again, I basically just copied the code from the previous
training cell above in part 1
        # it works

        x = x.to(device)
```

```
        noise = torch.randn_like(x).to(device)
        timesteps = torch.randint(0, 1000, (x.shape[0],),
device=device)

        noisy_x = noise_scheduler.add_noise(x, noise, timesteps)

        pred = model_Unet(noisy_x, timesteps).sample  # Extract the
predicted noise tensor
        loss = loss_fn(pred, noise)

        opt.zero_grad()
        loss.backward()
        opt.step()



        # Store the loss for later
        losses.append(loss.item())

    # Print our the average of the loss values for this epoch:
    avg_loss = sum(losses[-
len(train_dataloader):])/len(train_dataloader)
    print(f'Finished epoch {epoch + 1}. Average loss for this epoch:
{avg_loss:05f}')
```

```
Finished epoch 1. Average loss for this epoch: 0.037752
Finished epoch 2. Average loss for this epoch: 0.022278
Finished epoch 3. Average loss for this epoch: 0.020129
Finished epoch 4. Average loss for this epoch: 0.019438
Finished epoch 5. Average loss for this epoch: 0.018448
Finished epoch 6. Average loss for this epoch: 0.018108
Finished epoch 7. Average loss for this epoch: 0.017640
Finished epoch 8. Average loss for this epoch: 0.017466
Finished epoch 9. Average loss for this epoch: 0.017203
Finished epoch 10. Average loss for this epoch: 0.017032
```

```
# View the loss curve
plt.plot(losses)
plt.ylim(0, 1)
plt.title("Training Loss Curve")
plt.xlabel('Mini-batches')
plt.ylabel('Mean Squares error')
plt.show()
```



Sampling 20 pts

Generate 64 images and display them in a grid of 8x8 like we did in the 1st half of the homework.

```
Get DDPM noise_scheduler.timesteps #these are the values of t from 999
to 0
get noise_pred = model(x,t).sample #starting with x_T get the noise
prediction
x = noise_scheduler.step(noise_pred, t, x)[0] #get x_{t-1} with
noise_pred, t, and x_t as inputs

# Display the 64 generated images in a grid
### IMPLEMENT CODE ###

# I just copied the code from the previous plotting cell above in part
1
# it works still


b = torch.linspace(0, 1, n_steps)
a = 1 - b
abar = torch.cumprod(a, dim=0).view(-1, 1, 1, 1).to(device)

x = torch.randn(64, 1, 32, 32).to(device)
```

```
step_history = [x.detach().cpu()]

# Get diffusion timesteps
timesteps = noise_scheduler.timesteps  # These should be from 999 to 0

for t in timesteps:
    with torch.no_grad():
        noise_pred = model_Unet(x, t).sample  # Predict noise at time
t

    # Compute x_{t-1} using the scheduler step function
    x = noise_scheduler.step(noise_pred, t, x)[0]
    step_history.append(x.detach().cpu())

# Visualizing the final images
final_images = x.detach().cpu()
grid = torchvision.utils.make_grid(final_images, nrow=8,
normalize=True)

plt.figure(figsize=(8,8))
plt.imshow(grid.permute(1, 2, 0), cmap='gray')
plt.axis('off')
plt.title('DDPM Generated Samples')
plt.show()
```

DDPM Generated Samples

```
# This is the end of the assignment
# I spent longer on this assignment than I did on the paper
presentaton
# I hated this assignment but I feel like it did a good job of helping
me understand how industry code gets made
# ...
```