ConditionalDDPM

May 4, 2024

0.1 Setup

Similar to the previous projects, we will need some code to set up the environment.

First, run this cell that loads the autoreload extension. This allows us to edit .py source files and re-import them into the notebook for a seamless editing and debugging experience.

[1]:
```
%load_ext autoreload
%autoreload 2
```

0.1.1 Google Colab Setup

Run the following cell to mount your Google Drive. Follow the link and sign in to your Google account (the same account you used to store this notebook!).

[2]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

Then enter your path of the project (for example, /content/drive/MyDrive/ConditionalDDPM)

[3]:
```
%cd /home/alex/Downloads/ConditionalDDPM
```

```
/home/alex/Downloads/ConditionalDDPM
```

```
/home/alex/anaconda3/envs/239as/lib/python3.9/site-
packages/IPython/core/magics/osm.py:417: UserWarning: using dhist requires you
to install the `pickleshare` library.
  self.shell.db['dhist'] = compress_dhist(dhist)[-100:]
```

We will use GPUs to accelerate our computation in this notebook. Go to `Runtime > Change runtime type` and set `Hardware accelerator` to `GPU`. This will reset Colab. **Rerun the top cell to mount your Drive again.** Run the following to make sure GPUs are enabled:

[4]:
```
# set the device
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

if torch.cuda.is_available():
  print('Good to go!')
else:
  print('Please set GPU via the downward triangle in the top right corner.')
```

```
Good to go!
```

0.2 Conditional Denoising Diffusion Probabilistic Models

In the lectures, we have learnt about Denoising Diffusion Probabilistic Models (DDPM), as presented in the paper Denoising Diffusion Probabilistic Models. We went through both the training process and test sampling process of DDPM. In this project, you will use conditional DDPM to generate digits based on given conditions. The project is inspired by the paper Classifier-free Diffusion Guidance, which is a following work of DDPM. You are required to use MNIST dataset and the GPU device to complete the project.

(It will take about 20~30 minutes (10 epochs) if you are using the free-version Google Colab GPU. Typically, realistic digits can be generated after around 2~5 epochs.)

0.2.1 What is a DDPM?

A Denoising Diffusion Probabilistic Model (DDPM) is a type of generative model inspired by the natural diffusion process. In the example of image generation, DDPM works in two main stages:

- Forward Process (Diffusion): It starts with an image sampled from the dataset and gradually adds noise to it step by step, until it becomes completely random noise. In implementation, the forward diffusion process is fixed to a Markov chain that gradually adds Gaussian noise to the data according to a variance schedule $\beta_1, ..., \beta_T$.
- Reverse Process (Denoising): By learning how the noise was added on the image step by step, the model can do the reverse process: start with random noise and step by step, remove this noise to generate an image.

0.2.2 Training and sampling of DDPM

As proposed in the DDPM paper, the training and sampling process can be concluded in the following steps:

Here we still use the example of image generation.

Algorithm 1 shows the training process of DDPM. Initially, an image $\mathbf{x}_0$ is sampled from the data distribution $q(\mathbf{x}_0)$, i.e. the dataset. Then a time step t is randomly selected from a uniform distribution across the predifined number of steps T.
A noise ϵ which has the same shape of the image is sampled from a standard normal distribution. According to the equation (4) in the DDPM paper and the new notation: $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$, $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, we can get an intermediate state of the diffusion process: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon$. The model takes the $\mathbf{x}_t$ and t as inputs, and predict a noise, i.e. $\epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, t)$. The optimization of the model is done by minimize the difference between the sampled noise and the model's prediction of noise.

Algorithm 2 shows the sampling process of DDPM, which is the complete procedure for generating an image. This process starts from noise x_T sampled from a standard normal distribution, and then uses the trained model to iteratively apply denoising for each time step from T to 1.

0.2.3 How to control the generation output?

As you may find, the vanilla DDPM can only randomly generate images which are sampled from the learned distribution of the dataset, while in some cases, we are more interested in controlling the content of generated images. Previous works mainly use an extra trained classifier to guide the diffusion model to generate specific images (Dhariwal & Nichol (2021)). Ho et al. proposed the Classifier-free Diffusion Guidance, which proposes a novel training and sampling method to achieve the conditional generation without extra models besides the diffusion model. Now let's see how it modify the training and sampling pipeline of DDPM.

Algorithm 1: Conditional training The training process is shown in the picture below. Some notations are modified in order to follow DDPM.

Compared with the training process of vanilla DDPM, there are several modifications.

- In the training data sampling, besides the image $\mathbf{x}_0$, we also sample the condition $\mathbf{c}_0$ from the dataset (usually the class label).
- There's a probabilistic step to randomly discard the conditions, training the model to generate data both conditionally and unconditionally. Usually we just set the one-hot encoded label as all -1 to discard the conditions.
- When optimizing the model, the condition $\mathbf{c}_0$ is an extra input.

Algorithm 2: Conditional sampling Below is the sampling process of conditional DDPM.

Compared with the vanilla DDPM, the key modification is in step 4. Here the algorithm computes a corrected noise estimation, $\tilde{\epsilon}_t$, balancing between the conditional prediction $\epsilon_\theta(\mathbf{x}_t, \mathbf{c}, t)$ and the unconditional prediction $\epsilon_\theta(\mathbf{x}_t, t)$. The corrected noise $\tilde{\epsilon}_t$ is then used to update $\mathbf{x}_t$ in step 5. **Here we follow the setting of DDPM paper and define $\sigma_t = \sqrt{\beta_t}$.**

0.2.4 Conditional generation of digits

Now let's practice it! You will first asked to design a denoising network, and then complete the training and sampling process of this conditional DDPM. In this project, by default, we resize all images to a dimension of 28 × 28 and utilize one-hot encoding for class labels.

First we define a configuration class `DMConfig`. This class contains all the settings of the model and experiment that may be useful later.

[5]:

```python
from dataclasses import dataclass, field
from typing import List, Tuple
@dataclass
class DMConfig:
    '''
    Define the model and experiment settings here
    '''
    input_dim: Tuple[int, int] = (28, 28) # input image size
    num_channels: int = 1                 # input image channels
    condition_mask_value: int = -1        # unconditional condition mask value
    num_classes: int = 10                 # number of classes in the dataset
    T: int = 400                              # diffusion and denoising steps
    beta_1: float = 1e-4                      # variance schedule
    beta_T: float = 2e-2
    mask_p: float = 0.1                       # unconditional condition drop ratio
    num_feat: int = 128                       # feature size of the UNet model
    omega: float = 2.0                        # conditional guidance weight

    batch_size: int = 256                     # training batch size
    epochs: int = 10                          # training epochs
    learning_rate: float = 1e-4               # training learning rate
    multi_lr_milestones: List[int] = field(default_factory=lambda: [20]) #␣
 ↪learning rate decay milestone
    multi_lr_gamma: float = 0.1               # learning rate decay ratio
```

Then let's prepare and visualize the dataset:

[6]:

```
from utils import make_dataloader
from torchvision import transforms
import torchvision.utils as vutils
import matplotlib.pyplot as plt

# Define the data preprocessing and configuration
transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
config = DMConfig()

# Create the train and test dataloaders
train_loader = make_dataloader(transform = transform, batch_size = config.
 ↪batch_size, dir = './data', train = True)
test_loader = make_dataloader(transform = transform, batch_size = config.
 ↪batch_size, dir = './data', train = False)

# Visualize the first 100 images
dataiter = iter(train_loader)
images, labels = next(dataiter)
images_subset = images[:100]
grid = vutils.make_grid(images_subset, nrow = 10, normalize = True, padding=2)
plt.figure(figsize=(6, 6))
plt.imshow(grid.numpy().transpose((1, 2, 0)))
plt.axis('off')
plt.show()
```

1. Denoising network (4 points) The denoising network is defined in the file `ResUNet.py`. We have already provided some potentially useful blocks, and you will be asked to complete the class `ConditionalUnet`.

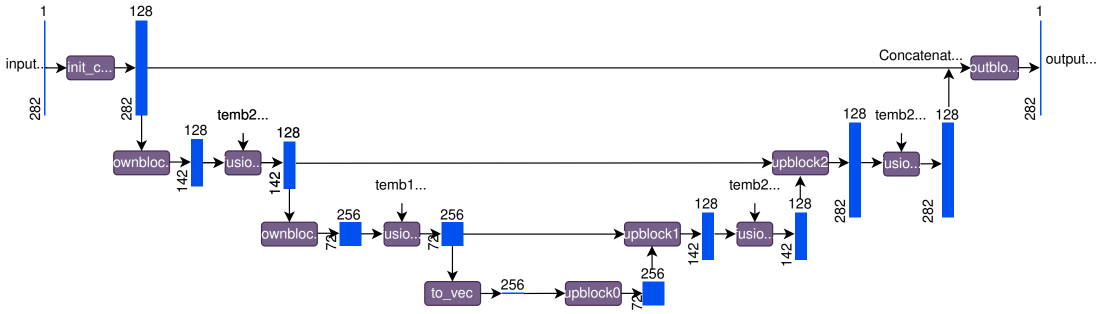
Some hints:

- **Please consider just using 2 down blocks and 2 up blocks. Using more blocks may improve the performance, while the training and sampling time may increase. Feel free to do some extra experiments in the creative exploring part later.**
- **An example structure of Conditional UNet is shown in the next cell. Here the initialization argument `n_feat` is set as 128. We provide all the potential useful components in the `__init__` function. The simplest way to construct the network is to complete the `forward` function with these components**
- **You can design your own network and add any blocks. Feel free to modifiy or even remove the provided blocks or layers. You are also free to change the way of adding the time step and condition.**

[7]:

```
# Example structure of Conditional UNet
from IPython.core.display import SVG
SVG(filename='./pics/ConUNet.svg')
```

[7]:



Now let's check your denoising network using the following code.

[8]:

```
from ResUNet import ConditionalUnet
import torch
model = ConditionalUnet(in_channels = 1, n_feat = 128, n_classes = 10).
  ↪to(device)
x = torch.randn((256,1,28,28)).to(device)
t = torch.randn((256,1,1,1)).to(device)
c = torch.randn((256,10)).to(device)
x_out = model(x,t,c)
assert x_out.shape == (256,1,28,28)
print('Output shape:', model(x,t,c).shape)
print('Dimension test passed!')
```

```
Output shape: torch.Size([256, 1, 28, 28])
Dimension test passed!
```

Before proceeding, please remember to normalize the time step t to the range 0-1 before inputting it into the denoising network for the next part of the project. It will help the network have a more stable output.

2. Conditional DDPM With the correct denoising network, we can then start to build the pipeline of a conditional DDPM. You will be asked to complete the `ConditionalDDPM` class in the file `DDPM.py`.

2.1 Variance schedule (3 points) Let's first prepare the variance schedule β_t along with other potentially useful constants. You are required to complete the `ConditionalDDPM.scheduler` function in `DDPM.py`.

Given the starting and ending variances β_1 and β_T, the function should output one dictionary containing the following terms:

`beta_t`: variance of time step t_s, which is linearly interpolated between β_1 and β_T.

`sqrt_beta_t`: $\sqrt{\beta_t}$

`alpha_t`: $\alpha_t = 1 - \beta_t$

`oneover_sqrt_alpha`: $\frac{1}{\sqrt{\alpha_t}}$

`alpha_t_bar`: $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$

`sqrt_alpha_bar`: $\sqrt{\bar{\alpha}_t}$

`sqrt_oneminus_alpha_bar`: $\sqrt{1 - \bar{\alpha}_t}$

We set $\beta_1 = 1e - 4$ and $\beta_T = 2e - 2$. Let's check your solution!

[9]:

```
from DDPM import ConditionalDDPM
import torch
torch.set_printoptions(precision=8)
config = DMConfig(beta_1 = 1e-4, beta_T = 2e-2)
ConDDPM = ConditionalDDPM(dmconfig = config)
schedule_dict = ConDDPM.scheduler(t_s = torch.tensor(77)) # We use a specific␣
 ↪time step (77) to check your output
assert torch.abs(schedule_dict['beta_t'] - 0.003890) <= 1e-5
assert torch.abs(schedule_dict['sqrt_beta_t'] - 0.062374) <= 1e-5
assert torch.abs(schedule_dict['alpha_t'] - 0.996110) <= 1e-5
assert torch.abs(schedule_dict['oneover_sqrt_alpha'] - 1.001951) <= 1e-5
assert torch.abs(schedule_dict['alpha_t_bar'] - 0.857414) <= 1e-5
assert torch.abs(schedule_dict['sqrt_oneminus_alpha_bar'] - 0.377606) <= 1e-5
print('All tests passed!')
```

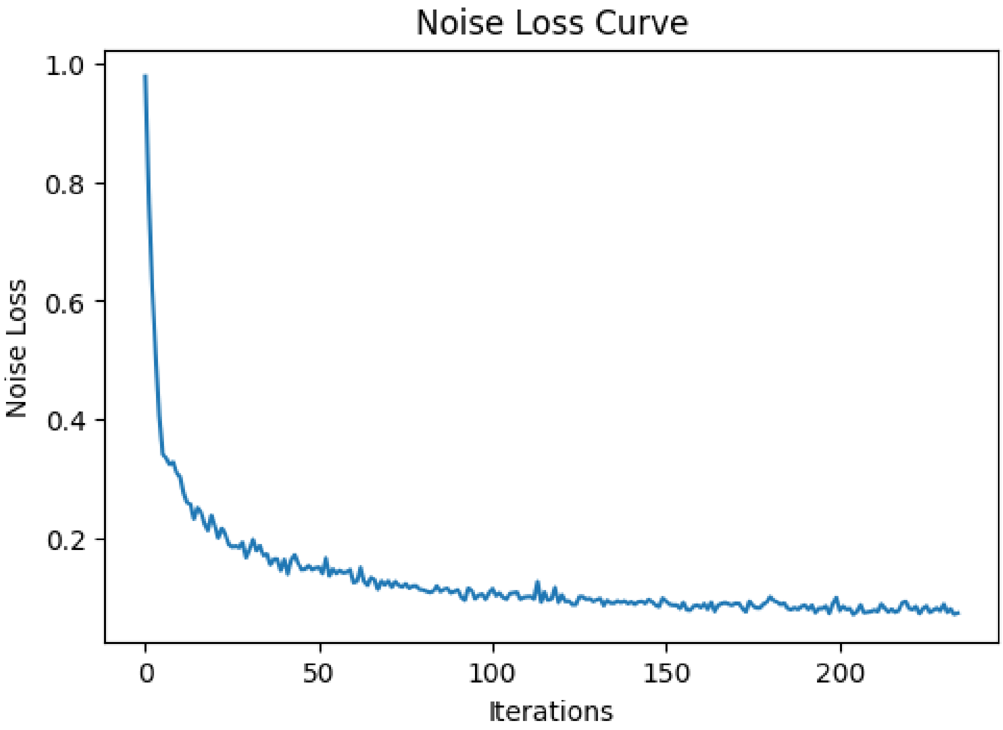
```
All tests passed!
```

2.2 Training process (5 points) Recall the training algorithm we discussed above:

You will need to complete the `ConditionalDDPM.forward` function in the `DDPM.py` file. Then you can use the function `utils.check_forward` to test if it's working properly. The model will be trained for one epoch in this checking process. It should take around 2 min and return one curve showing a decreasing loss trend if your `ConditionalDDPM.forward` function is correct.

[71]:

```
from utils import check_forward
config = DMConfig()
model = check_forward(train_loader, config, device)
```

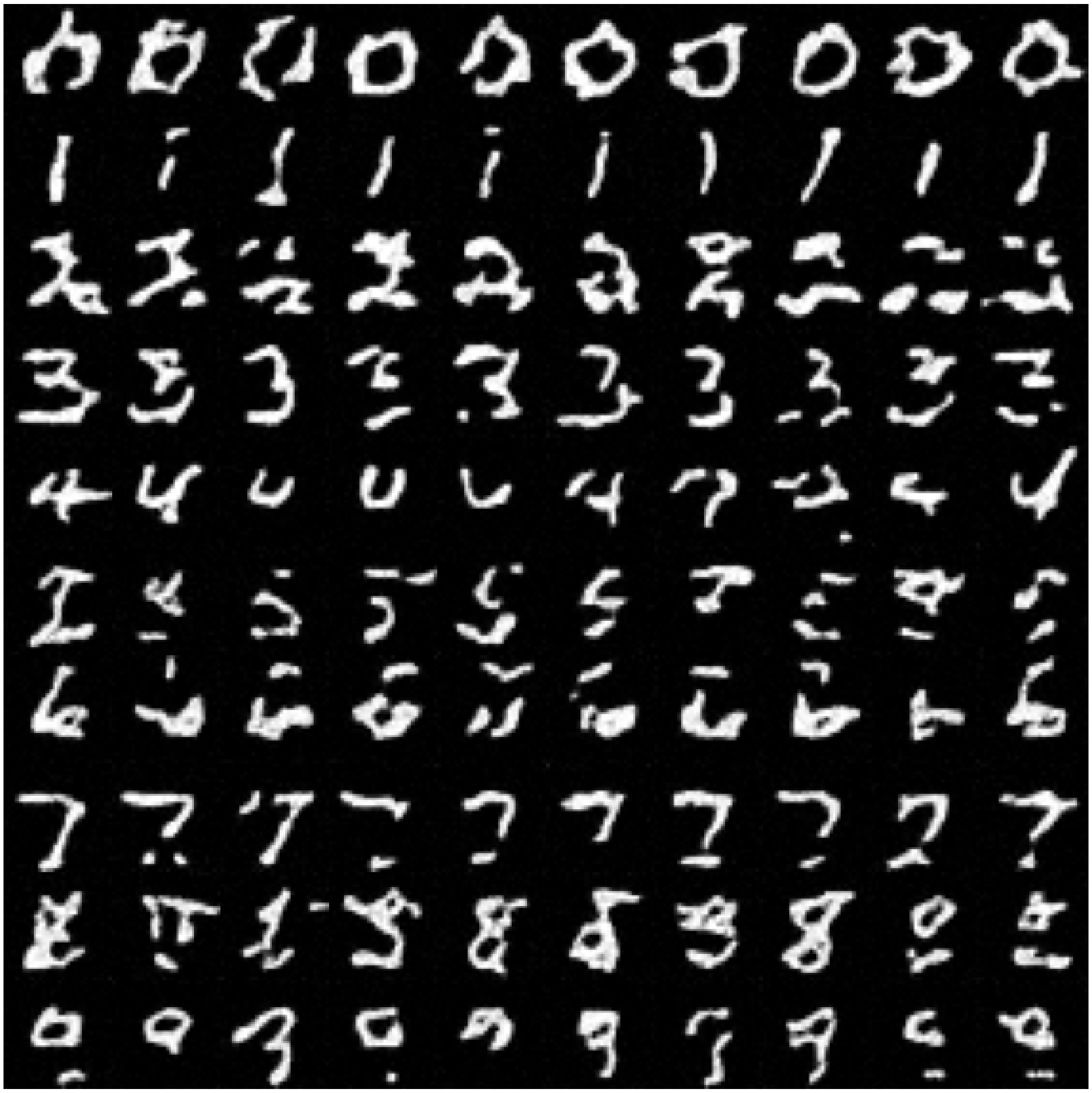


2.3 Sampling process (5 points)

Now you are required to complete the `ConditionalDDPM.sample` function using the sampling process we mentioned above.

In the following cell, we will use the given `utils.check_sample` function to check the correctness. With the trained model in 2.2, the model should be able to generate some super-rough digits (you may not even see them as digits). The sampling process should take about 1 minute.

[73]:
```
from utils import check_sample
config = DMConfig()
fig = check_sample(model, config, device)
```



2.4 Full training (5 points) As you might notice, the images generated are imperfect since the model trained for only one epoch has not yet converged. To improve the model's performance, we should proceed with a complete cycle of training and testing. You can utilize the provided `solver` function in this part.

Let's recall all model and experiment configurations:

[75]:
```
train_config = DMConfig()
print(train_config)
```

```
DMConfig(input_dim=(28, 28), num_channels=1, condition_mask_value=-1,
num_classes=10, T=400, beta_1=0.0001, beta_T=0.02, mask_p=0.1, num_feat=128,
omega=2.0, batch_size=256, epochs=10, learning_rate=0.0001,
multi_lr_milestones=[20], multi_lr_gamma=0.1)
```

Then we can use function `utils.solver` to train the model. You should also input your own experiment name, e.g. `your_exp_name`. The best-trained model will be saved as `./save/your_exp_name/best_checkpoint.pth`. Furthermore, for each training epoch, one generated image will be stored in the directory `./save/your_exp_name/images`.

[76]:

```python
from utils import solver
solver(dmconfig = train_config,
       exp_name = 'vanilla',
       train_loader = train_loader,
       test_loader = test_loader)
```

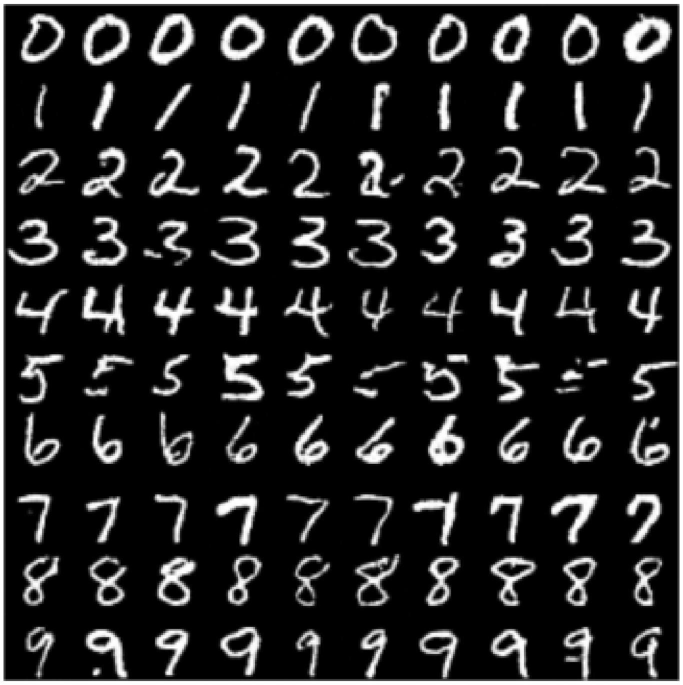
```
epoch 1/10


train: train_noise_loss = 0.1523 test: test_noise_loss = 0.0859
epoch 2/10


train: train_noise_loss = 0.0761 test: test_noise_loss = 0.0712
epoch 3/10


train: train_noise_loss = 0.0671 test: test_noise_loss = 0.0656
epoch 4/10


train: train_noise_loss = 0.0625 test: test_noise_loss = 0.0599
epoch 5/10


train: train_noise_loss = 0.0595 test: test_noise_loss = 0.0610
epoch 6/10


train: train_noise_loss = 0.0577 test: test_noise_loss = 0.0563
epoch 7/10


train: train_noise_loss = 0.0562 test: test_noise_loss = 0.0549
epoch 8/10


train: train_noise_loss = 0.0546 test: test_noise_loss = 0.0543
epoch 9/10
```

```
train: train_noise_loss = 0.0536 test: test_noise_loss = 0.0527
epoch 10/10


train: train_noise_loss = 0.0528 test: test_noise_loss = 0.0522
```

Now please show the image that you believe has the best generation quality in the following cell.

[80]:

```python
# ==================================================== #
# YOUR CODE HERE:
#   Among all images generated in the experiment,
#   show the image that you believe has the best generation quality.
#   You may use tools like matplotlib, PIL, OpenCV, ...
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Path to the image file
image_path = './save/vanilla/images/generate_epoch_10.png'

# Load the image
img = mpimg.imread(image_path)

# Display the image
plt.imshow(img)
plt.axis('off')  # Hide axes
plt.show()

# ==================================================== #
```



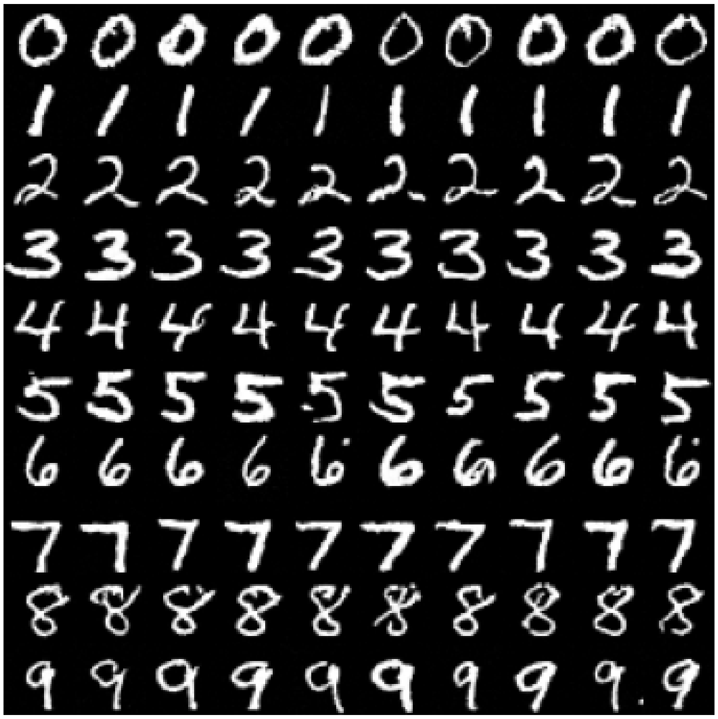
2.5 Exploring the conditional guidance weight (3 points) The generated images from the previous training-sampling process is using the default conditional guidance weight $\omega = 2$. Now with the best checkpoint, please try at least 3 different ω values and visualize the generated images. You can use the provided function `sample_images` to get a combined image each time.

[82]:

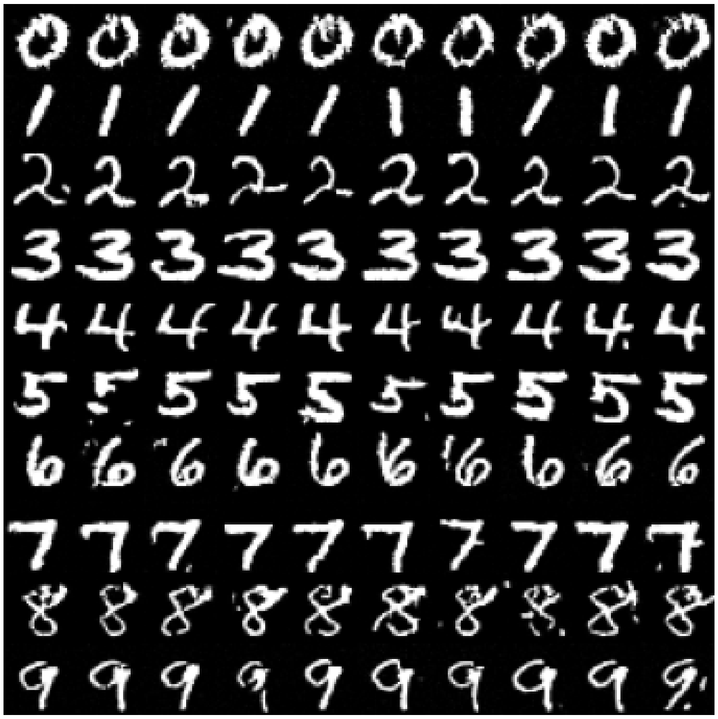
```
from utils import sample_images
import matplotlib.pyplot as plt
import os
# ===================================================== #
# YOUR CODE HERE:
#   Try at least 3 different conditional guidance weights and visualize it.
#   Example of using a different omega value:
#       sample_config = DMConfig(omega = ?)
#       fig = sample_images(config = sample_config, checkpoint_path =␣
 ↪path_to_your_checkpoint)

current_directory = "/home/alex/Downloads/ConditionalDDPM"
# Relative path to the image file
checkpoint_path = "save/vanilla/best_checkpoint.pth"
# Full path to the image file
full_checkpoint_path = os.path.join(current_directory, checkpoint_path)

print('The Figure Generate with Omega = 5:')
```

```
sample_config = DMConfig(omega = 5)
fig1 = sample_images(config = sample_config, checkpoint_path = full_checkpoint_path)
plt.imshow(fig1)
plt.axis('off')  # Turn off axis
plt.show()
print('\n')

print('The Figure Generate with Omega = 15:')
sample_config = DMConfig(omega = 15)
fig2 = sample_images(config = sample_config, checkpoint_path = full_checkpoint_path)
plt.imshow(fig2)
plt.axis('off')  # Turn off axis
plt.show()
print('\n')

print('The Figure Generate with Omega = 25:')
sample_config = DMConfig(omega = 25)
fig3 = sample_images(config = sample_config, checkpoint_path = full_checkpoint_path)
plt.imshow(fig3)
plt.axis('off')  # Turn off axis
plt.show()
print('\n')


# ==================================================== #
```

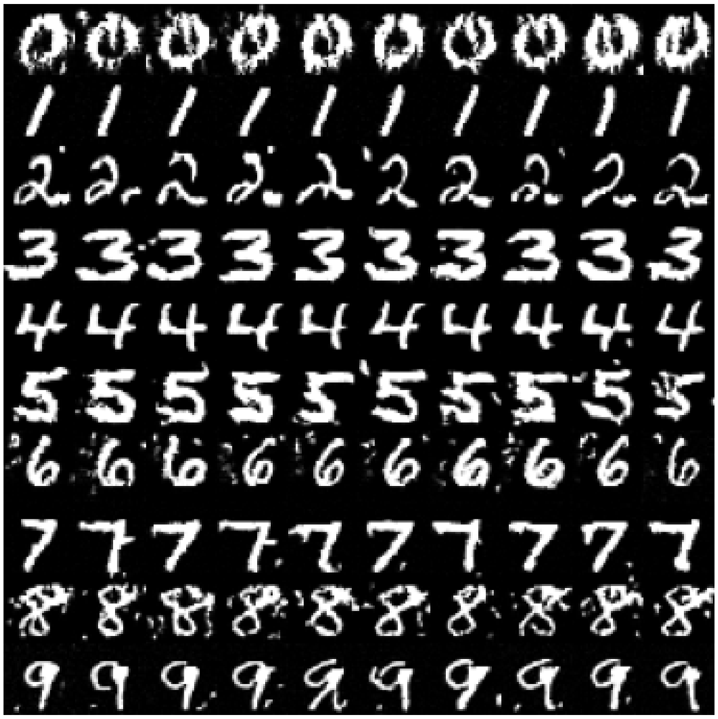
```
The Figure Generate with Omega = 5:
```



The Figure Generate with Omega = 15:



The Figure Generate with Omega = 25:



Inline Question: Based on your experiment, discuss how the conditional guidance weight affects the quality and diversity of generation.

Your answer: As the weight increases, although the quality of the generated images improves and becomes more realistic, it obviously reduces the diversity.

2.6 Customize your own model (5 points) Now let's experiment by modifying some hyperparameters in the config and costomizing your own model. You should at least change one defalut setting in the config and train a new model. Then visualize the generation image and discuss the effects of your modifications.

Hint: Possible changes to the configuration include, but are not limited to, the number of diffusion steps T**, the unconditional condition drop ratio** $mask\_p$**, the feature size** num_feat**, the beta schedule, etc.**

First you should define and print your modified config. Please state all the changes you made to the DMConfig class, i.e. `DMConfig(T=?, num_feat=?, ...)`.

[83]:
```
# ==================================================== #
# YOUR CODE HERE:
#   Your new configuration:
#   train_config_new = DMConfig(...)
```

```
train_config_new = DMConfig(input_dim=(28, 28), num_channels=1,␣
 ↪condition_mask_value=-1, num_classes=10, T=1000, beta_1=0.001, beta_T=0.05,␣
 ↪mask_p=0.3, num_feat=256, omega=2.0, batch_size=256, epochs=10,␣
 ↪learning_rate=0.0001, multi_lr_milestones=[20], multi_lr_gamma=0.1)

# ==================================================== #
print(train_config_new)
```

```
DMConfig(input_dim=(28, 28), num_channels=1, condition_mask_value=-1,
num_classes=10, T=1000, beta_1=0.001, beta_T=0.05, mask_p=0.3, num_feat=256,
omega=2.0, batch_size=256, epochs=10, learning_rate=0.0001,
multi_lr_milestones=[20], multi_lr_gamma=0.1)
```

Then similar to 2.4, use `solver` funtion to complete the training and sampling process.

[84]:

```
from utils import solver
solver(dmconfig = train_config_new,
       exp_name = 'opt_version',
       train_loader = train_loader,
       test_loader = test_loader)
```

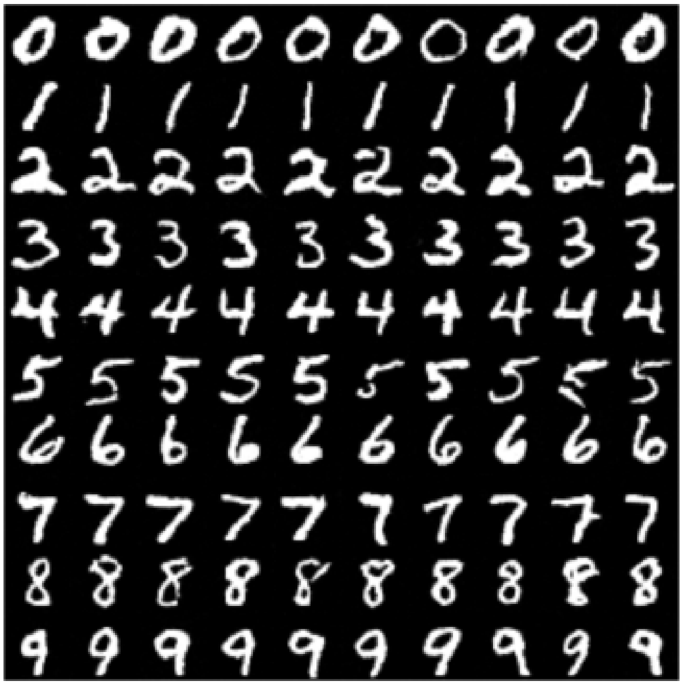
```
epoch 1/10


train: train_noise_loss = 0.0726 test: test_noise_loss = 0.0389
epoch 2/10


train: train_noise_loss = 0.0280 test: test_noise_loss = 0.0272
epoch 3/10


train: train_noise_loss = 0.0253 test: test_noise_loss = 0.0242
epoch 4/10


train: train_noise_loss = 0.0232 test: test_noise_loss = 0.0231
epoch 5/10


train: train_noise_loss = 0.0221 test: test_noise_loss = 0.0212
epoch 6/10


train: train_noise_loss = 0.0215 test: test_noise_loss = 0.0201
epoch 7/10
```

```
train: train_noise_loss = 0.0207 test: test_noise_loss = 0.0203
epoch 8/10


train: train_noise_loss = 0.0202 test: test_noise_loss = 0.0210
epoch 9/10


train: train_noise_loss = 0.0198 test: test_noise_loss = 0.0195
epoch 10/10


train: train_noise_loss = 0.0190 test: test_noise_loss = 0.0219
```

Finally, show one image that you think has the best quality.

[85]:

```python
# ==================================================== #
# YOUR CODE HERE:
#   Among all images generated in the experiment,
#   show the image that you believe has the best generation quality.
#   You may use tools like matplotlib, PIL, OpenCV, ...

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Path to the image file
image_path = './save/opt_version/images/generate_epoch_9.png'

# Load the image
img_opt = mpimg.imread(image_path)

# Display the image
plt.imshow(img_opt)
plt.axis('off')  # Hide axes
plt.show()


# ==================================================== #
```



Inline Question: Discuss the effects of your modifications after you compare the generation performance under different configurations.

Your answer: After changing the difussion T, mask_p, num_feat, beta schedule parameters. The quality of generation is improved and the diversity of generation remain the same than before the modifications.

0.3 ResUNet.py

[ ]:
```
import torch
import torch.nn as nn
import math
class ResConvBlock(nn.Module):
    '''
    Basic residual convolutional block
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(x1)
        if self.in_channels == self.out_channels:
            out = x + x2
        else:
            out = x1 + x2
        return out / math.sqrt(2)


class UnetDown(nn.Module):
    '''
    UNet down block (encoding)
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        layers = [ResConvBlock(in_channels, out_channels), nn.MaxPool2d(2)]
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)


class UnetUp(nn.Module):
    '''
    UNet up block (decoding)
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        layers = [
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResConvBlock(out_channels, out_channels),
            ResConvBlock(out_channels, out_channels),
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x, skip):
        x = torch.cat((x, skip), 1)
        x = self.model(x)
        return x
```

```
class EmbedBlock(nn.Module):
    '''
    Embedding block to embed time step/condition to embedding space
    '''
    def __init__(self, input_dim, emb_dim):
        super().__init__()
        self.input_dim = input_dim
        layers = [
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
        ]
        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        # set embedblock untrainable
        for param in self.layers.parameters():
            param.requires_grad = False
        x = x.view(-1, self.input_dim)
        return self.layers(x)

class FusionBlock(nn.Module):
    '''
    Concatenation and fusion block for adding embeddings
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
    def forward(self, x, t, c):
        h,w = x.shape[-2:]
        return self.layers(torch.cat([x, t.repeat(1,1,h,w), c.repeat(1,1,h,w)], dim = 1))

class ConditionalUnet(nn.Module):
    def __init__(self, in_channels, n_feat = 128, n_classes = 10):
        super().__init__()

        self.in_channels = in_channels
        self.n_feat = n_feat
        self.n_classes = n_classes
        # embeddings
        self.timeembed1 = EmbedBlock(1, 2*n_feat)
        self.timeembed2 = EmbedBlock(1, 1*n_feat)
        self.conditionembed1 = EmbedBlock(n_classes, 2*n_feat)
        self.conditionembed2 = EmbedBlock(n_classes, 1*n_feat)

        # down path for encoding
        self.init_conv = ResConvBlock(in_channels, n_feat)
        self.downblock1 = UnetDown(n_feat, n_feat)
        self.downblock2 = UnetDown(n_feat, 2 * n_feat)
        self.to_vec = nn.Sequential(nn.AvgPool2d(7), nn.GELU())


        # up path for decoding
        self.upblock0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feat, 2 * n_feat, 7, 7),
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )
        self.upblock1 = UnetUp(4 * n_feat, n_feat)
        self.upblock2 = UnetUp(2 * n_feat, n_feat)
        self.outblock = nn.Sequential(
            nn.Conv2d(2 * n_feat, n_feat, 3, 1, 1),
            nn.GroupNorm(8, n_feat),
            nn.ReLU(),
            nn.Conv2d(n_feat, self.in_channels, 3, 1, 1),
        )

        # fusion blocks
        self.fusion1 = FusionBlock(3 * self.n_feat, self.n_feat)
        self.fusion2 = FusionBlock(6 * self.n_feat, 2 * self.n_feat)
        self.fusion3 = FusionBlock(3 * self.n_feat, self.n_feat)
        self.fusion4 = FusionBlock(3 * self.n_feat, self.n_feat)

    def forward(self, x, t, c):
        '''
        Inputs:
            x: input images, with size (B,1,28,28)
            t: input time steps, with size (B,1,1,1)
            c: input conditions (one-hot encoded labels), with size (B,10)
        '''
        t, c = t.float(), c.float()

        # time step embedding
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1) # 256
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1) # 128
        # condition embedding
        cemb1 = self.conditionembed1(c).view(-1, self.n_feat * 2, 1, 1) # 256
        cemb2 = self.conditionembed2(c).view(-1, self.n_feat, 1, 1) # 128

        # ==================================================== #
        # YOUR CODE HERE:
        #   Define the process of computing the output of a
        #   this network given the input x, t, and c.
        #   The input x, t, c indicate the input image, time step
        #   and the condition respectively.
        # A potential format is shown below, feel free to use your own ways to design it.
        # down0 =
        # down1 =
        # down2 =
        # up0 =
        # up1 =
        # up2 =
        # out = self.outblock(torch.cat((up2, down0), dim = 1))
        # ==================================================== #
        down0 = self.init_conv(x)
        down1 = self.fusion1(self.downblock1(down0),temb2,cemb2)
        down2 = self.fusion2(self.downblock2(down1),temb1, cemb1)
        vec = self.to_vec(down2)
        up0 = self.upblock0(vec)
        up1 = self.fusion3(self.upblock1(up0,down2),temb2,cemb2)
        up2 = self.fusion4(self.upblock2(up1,down1),temb2, cemb2)
        out = self.outblock(torch.cat((up2, down0), dim = 1))

        return out
```

0.4 DDPM.py

[ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from ResUNet import ConditionalUnet
from utils import *

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

class ConditionalDDPM(nn.Module):
    def __init__(self, dmconfig):
        super().__init__()
        self.dmconfig = dmconfig
        self.loss_fn = nn.MSELoss()
        self.network = ConditionalUnet(1, self.dmconfig.num_feat, self.dmconfig.num_classes)

    def scheduler(self, t_s):
        beta_1, beta_T, T = self.dmconfig.beta_1, self.dmconfig.beta_T, self.dmconfig.T
        # ==================================================== #
        # YOUR CODE HERE:
        #   Inputs:
        #       t_s: the input time steps, with shape (B,1).
        #   Outputs:
        #       one dictionary containing the variance schedule
        #       $\beta_t$ along with other potentially useful constants.

        # Linear interpolation of beta_t
        all_beta_t = beta_1 + (beta_T - beta_1) * torch.arange(0, T+1,device=t_s.device) / (T - 1)
        beta_t = all_beta_t[t_s -1]

        # Compute other constants
        sqrt_beta_t = torch.sqrt(beta_t)
        alpha_t = 1 - beta_t
        oneover_sqrt_alpha = 1 / torch.sqrt(alpha_t)


        all_alpha_t = 1 - all_beta_t
        # Compute cumulative products of alpha_t
        alpha_t_bar = torch.cumprod(all_alpha_t, dim=0)[t_s -1]
        sqrt_alpha_bar = torch.sqrt(alpha_t_bar)
        sqrt_oneminus_alpha_bar = torch.sqrt(1 - alpha_t_bar)



        # ==================================================== #
        return {
            'beta_t': beta_t,
            'sqrt_beta_t': sqrt_beta_t,
            'alpha_t': alpha_t,
            'sqrt_alpha_bar': sqrt_alpha_bar,
            'oneover_sqrt_alpha': oneover_sqrt_alpha,
            'alpha_t_bar': alpha_t_bar,
            'sqrt_oneminus_alpha_bar': sqrt_oneminus_alpha_bar
        }

    def forward(self, images, conditions):
        T = self.dmconfig.T
        noise_loss = None
        # ==================================================== #
        # YOUR CODE HERE:
        #   Complete the training forward process based on the
        #   given training algorithm.
        #   Inputs:
        #       images: real images from the dataset, with size (B,1,28,28).
        #       conditions: condition labels, with size (B). You should
        #                   convert it to one-hot encoded labels with size (B,10)
        #                   before making it as the input of the denoising network.
        #   Outputs:
        #       noise_loss: loss computed by the self.loss_fn function  .
        x_0 = images
        c = conditions
        B = images.shape[0]
        p_uncond = self.dmconfig.mask_p

        c_0 = F.one_hot(c, num_classes = 10) #(B,10)
        mask = torch.bernoulli((torch.zeros_like(c)+p_uncond).to(device)).view(B,1)
        c_masked = c_0 * (1 - mask) + mask * self.dmconfig.condition_mask_value

        t = torch.randint(1, T+1, (B,)).to(device)  # t ~ Uniform(0, n_T)

        sqrt_alpha_bar = self.scheduler(t)['sqrt_alpha_bar'].to(device)
        sqrt_one_minus_alpha_bar = self.scheduler(t)['sqrt_oneminus_alpha_bar'].to(device)

        eps = torch.randn_like(x_0).to(device)
        # This is the x_t, which is sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
        x_t = sqrt_alpha_bar.view(-1,1,1,1) * x_0 + sqrt_one_minus_alpha_bar.view(-1,1,1,1) * eps
        eps_theta = self.network(x_t, t/T, c_masked)
        # return MSE between added noise, and our predicted noise
        noise_loss = self.loss_fn(eps_theta, eps)

        # ==================================================== #

        return noise_loss

    def sample(self, conditions, omega):
        T = self.dmconfig.T
        X_t = None
        # ==================================================== #
        # YOUR CODE HERE:
        #   Complete the training forward process based on the
        #   given sampling algorithm.
        #   Inputs:
        #       conditions: condition labels, with size (B). You should
        #                   convert it to one-hot encoded labels with size␣
↪(B,10)
        #                   before making it as the input of the denoising␣
↪network.
        #       omega: conditional guidance weight.
        #   Outputs:
        #       generated_images

        B = conditions.size(0)
        c = conditions
        # Start by sampling noise from a normal distribution
        # X_t = torch.randn(batch_size, 1, 28, 28)  # Step 1: initialize X_T␣
↪from N(0, I)
        X_t = torch.randn(B, self.dmconfig.num_channels, self.dmconfig.
↪input_dim[0], self.dmconfig.input_dim[1]).to(device)


        # Convert conditions to one-hot encoded labels if they are not already
        if c.ndim < 2 or c.size(1) != self.dmconfig.num_classes:
           c = F.one_hot(c, num_classes=self.dmconfig.num_classes).float()

        with torch.no_grad():
          for t in reversed(range(1, T+1)):  # Step 2: loop backwards from T to␣
↪1

              # Get the variance and other constants for time t
              t_is = torch.full((B,), t).to(device).view(B, 1)

              # Sample random noise
              z = torch.randn_like(X_t).to(device) if t > 1 else torch.
↪zeros_like(X_t).to(device)

              alpha_t = self.scheduler(t_is)['alpha_t'].to(device)
              sqrt_oneminus_alpha_bar = self.
↪scheduler(t_is)['sqrt_oneminus_alpha_bar'].to(device)
              oneover_sqrt_alpha = self.scheduler(t_is)['oneover_sqrt_alpha'].
↪to(device)
              sqrt_beta_t = self.scheduler(t_is)['sqrt_beta_t'].to(device)

              eps = self.network(X_t, t_is/T, c)
```

```
            c_uncond = torch.full_like(c, self.dmconfig.condition_mask_value).
↪to(device)
            eps_uncond = self.network(X_t, t_is/T, c_uncond)
            # Corrected noise using classifier-free guidance
            eps_hat = (1 + omega) * eps - omega * eps_uncond

            X_t = oneover_sqrt_alpha.view(-1,1,1,1) * (X_t - (1- alpha_t).
↪view(-1,1,1,1) * eps_hat / sqrt_oneminus_alpha_bar.view(-1,1,1,1))
            X_t += sqrt_beta_t.view(-1,1,1,1) * z

      # ==================================================== #
      generated_images = (X_t * 0.3081 + 0.1307).clamp(0,1) # denormalize the␣
↪output images
      return generated_images
```