ConditionalDDPM

May 5, 2025

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.

[3]:
```
%load_ext autoreload
%autoreload 2
```

0.1.1 Google Colab Setup

If you are not using Colab, please just skip this step.

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 /content/drive/MyDrive/Graduate/ECE239_TA/ConditionalDDPM/skeleton
```

We will use GPUs to accelerate our computation in this notebook.

If you are using Colab, 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!')
```

```
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.

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:

[5]:
```python
from IPython.display import Image
Image(filename='pics/DDPM.png', width=800, height=200)
```

[5]:

Algorithm 1 Training

repeat
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
  $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  Take gradient descent step on
    $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar\alpha_t}\mathbf{x}_0 + \sqrt{1-\bar\alpha_t}\boldsymbol{\epsilon}, t) \right\|^2$
until converged

Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar\alpha_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

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: $q(\mathbf{x}_t|\mathbf{x}_0) = N(\mathbf{x}_t; \sqrt{\bar\alpha_t}\mathbf{x}_0, (1-\bar\alpha_t)\mathbf{I})$, $\alpha_t := 1 - \beta_t$ 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 minimizing 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 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.

[6]:
```python
from IPython.display import Image
Image(filename='pics/ConDDPM_1.png', width=800, height=240)
```

[6]:

Algorithm 1 Joint training a diffusion model with classifier-free guidance

Require: p_{uncond}: probability of unconditional training

repeat
- $(\mathbf{x}_0, \mathbf{c}_0) \sim q(\mathbf{x}_0, \mathbf{c}_0)$ ▷ Sample data with conditioning from the dataset
- $\mathbf{c}_0 \leftarrow \emptyset$ with probability p_{uncond} ▷ Randomly discard conditioning to train unconditionally
- $t \sim \text{Uniform}(\{1, ..., T\})$ ▷ Sample time steps
- $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\boldsymbol{\epsilon}$ ▷ Corrupt data to the sampled time steps
- Take gradient step on $\nabla_\theta \left\|\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, \mathbf{c}_0, t) - \boldsymbol{\epsilon}\right\|^2$ ▷ Optimization of denoising model

until converged

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.

[7]:
```python
from IPython.display import Image
Image(filename='pics/ConDDPM_2.png', width=500, height=250)
```

[7]:

Algorithm 2 Conditional sampling with classifier-free guidance

Require: w: guidance weight

Require: c: conditioning information for conditional sampling

1: $\mathbf{x}_T \sim \mathcal{N}(0, I)$

2: **for** $t = T, \dots, 1$ **do**

3: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = 0$

4: $\quad \tilde{\boldsymbol{\epsilon}}_t = (1 + w)\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, \mathbf{c}, t) - w\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$

5: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\tilde{\boldsymbol{\epsilon}}_t\right) + \sigma_t \mathbf{z}$

6: **end for**

7: **return** $\mathbf{x}_0$

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. Also, please remember to normalize the time step t to the range 0-1 before inputting it into the denoising network as it will help the network have a more stable output.

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

[8]:
```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                    # condition drop ratio
    num_feat: int = 64                    # basic 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:

[9]:
```
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()
```

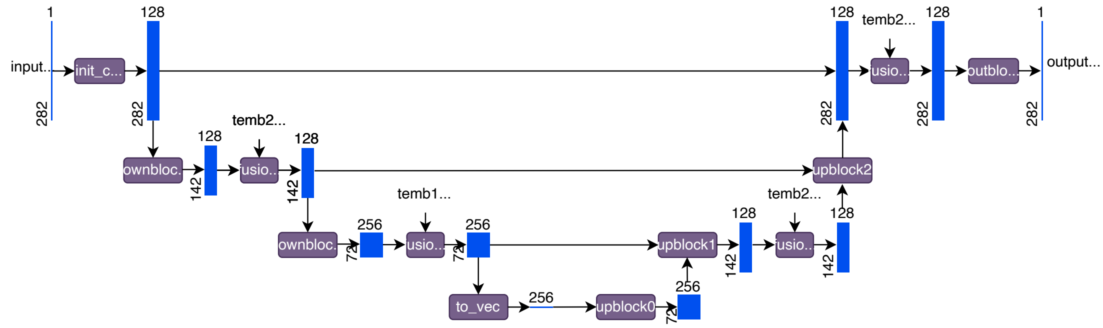
1. Denoising network (6 points) The denoising network is defined in the file `ResUNet.py`. We have already provided some potentially useful layers or blocks, and you will be asked to complete the class `ConditionalUnet`.

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.
- **MODEL DESIGNING IS AN ART: You do not have to use this given structure. You can add/delete any blocks, or even design your own network from scratch. You are also free to change the way of adding the time step and condition.**

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

[10]:



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

```
[11]: 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!
```

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 (4 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!

[12]:
```python
from DDPM import ConditionalDDPM
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 schedule_dict['beta_t'] - 0.003890 <= 5e-6
assert schedule_dict['sqrt_beta_t'] - 0.062374 <= 5e-6
assert schedule_dict['alpha_t'] - 0.996110 <= 5e-6
assert schedule_dict['oneover_sqrt_alpha'] - 1.001951 <= 5e-6
assert schedule_dict['alpha_t_bar'] - 0.857414 <= 5e-6
assert schedule_dict['sqrt_oneminus_alpha_bar'] - 0.377606 <= 5e-6
print('All tests passed!')
```

```
All tests passed!
```

2.2 Training process (6 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 1 min and return one curve showing a decreasing loss trend if your `ConditionalDDPM.forward` function is correct.

[17]:
```python
from IPython.display import Image
Image(filename='pics/ConDDPM_1.png', width=800, height=240)
```

[17]:

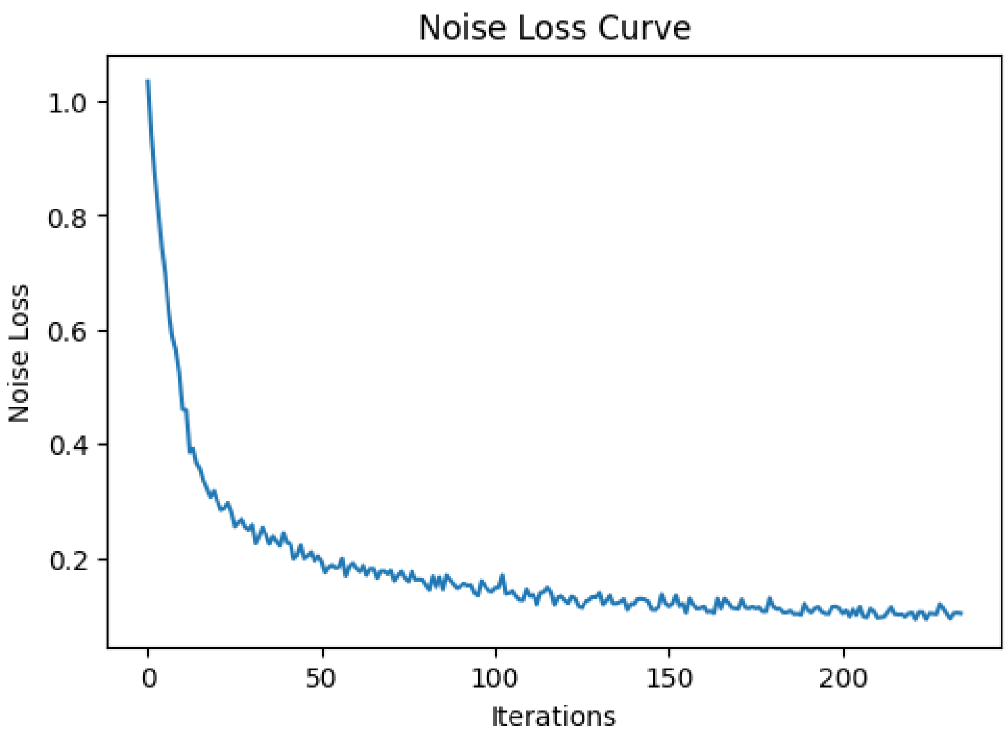
Algorithm 1 Joint training a diffusion model with classifier-free guidance

Require: p_{uncond}: probability of unconditional training

1: **repeat**
2: $(\mathbf{x}_0, \mathbf{c}_0) \sim q(\mathbf{x}_0, \mathbf{c}_0)$ ▷ Sample data with conditioning from the dataset
3: $\mathbf{c}_0 \leftarrow \emptyset$ with probability p_{uncond} ▷ Randomly discard conditioning to train unconditionally
4: $t \sim \text{Uniform}(\{1, ..., T\})$ ▷ Sample time steps
5: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
6: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\boldsymbol{\epsilon}$ ▷ Corrupt data to the sampled time steps
7: Take gradient step on $\nabla_\theta \left\|\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, \mathbf{c}_0, t) - \boldsymbol{\epsilon}\right\|^2$ ▷ Optimization of denoising model
8: **until** converged

[18]:

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



2.3 Sampling process (6 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 around 30s.

[19]:

```python
from IPython.display import Image
Image(filename='pics/ConDDPM_2.png', width=500, height=250)
```

[19]:

Algorithm 2 Conditional sampling with classifier-free guidance

Require: w: guidance weight

Require: c: conditioning information for conditional sampling

1: $\mathbf{x}_T \sim \mathcal{N}(0, I)$

2: **for** $t = T, \ldots, 1$ **do**

3: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = 0$

4: $\quad \tilde{\boldsymbol{\epsilon}}_t = (1 + w)\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, \mathbf{c}, t) - w\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$

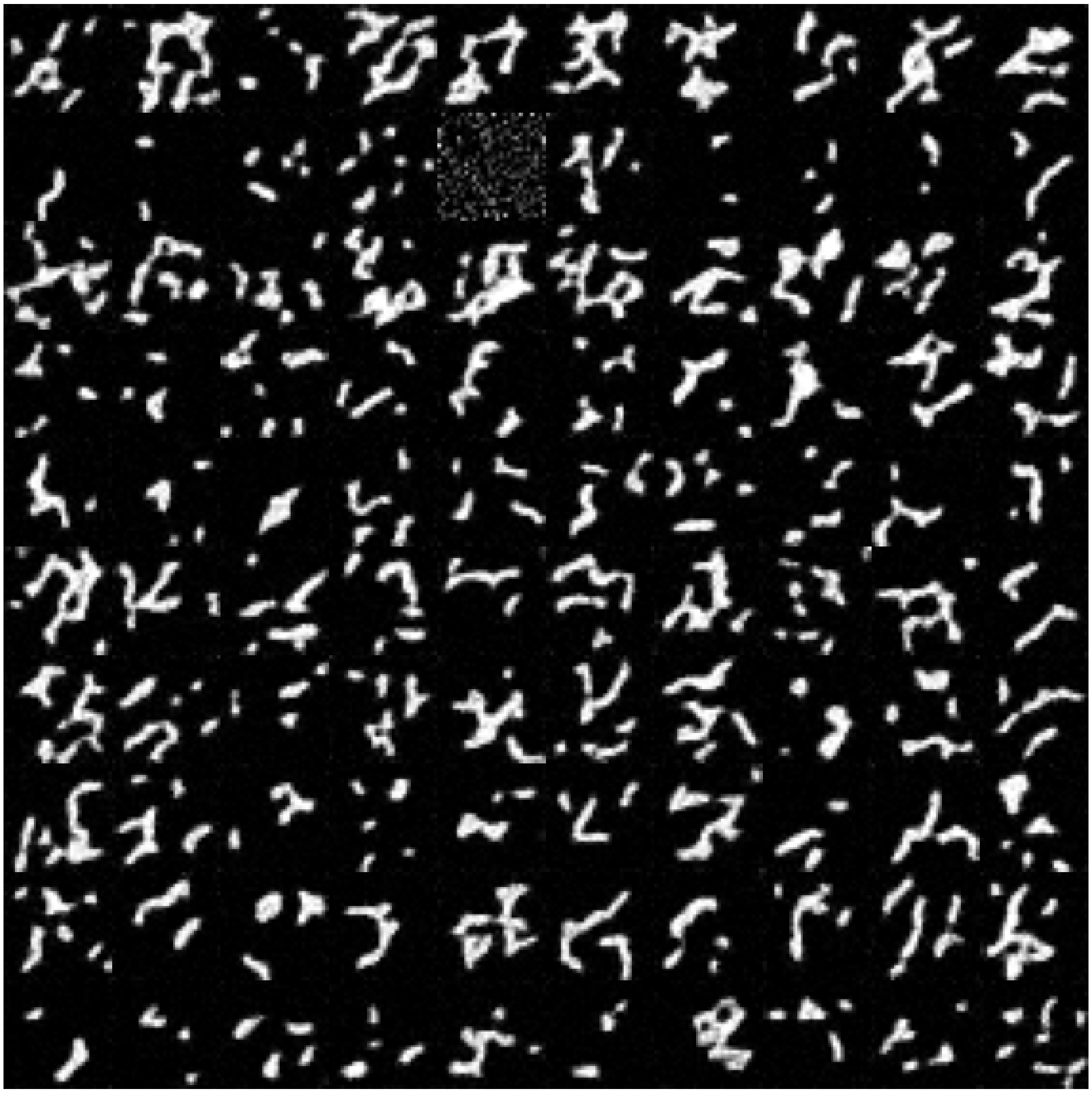
5: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\tilde{\boldsymbol{\epsilon}}_t\right) + \sigma_t \mathbf{z}$

6: **end for**

7: **return** $\mathbf{x}_0$

[35]:

```
from utils import check_sample
config = DMConfig()
fig = check_sample(model, config, device)
```



2.4 Full training (8 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 performance, we should proceed with a complete cycle of training and testing. You can utilize a provided `solver` function in this part.

Let's refresh all configurations:

[36]:
```
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=64,
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` as a validation.

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

[37]:
```
from utils import solver
solver(dmconfig = train_config,
       exp_name = 'test_3',
       train_loader = train_loader,
       test_loader = test_loader)
```

```
epoch 1/10


train: train_noise_loss = 0.1909 test: test_noise_loss = 0.1088
epoch 2/10


train: train_noise_loss = 0.0901 test: test_noise_loss = 0.0835
epoch 3/10


train: train_noise_loss = 0.0787 test: test_noise_loss = 0.0770
epoch 4/10


train: train_noise_loss = 0.0722 test: test_noise_loss = 0.0739
epoch 5/10


train: train_noise_loss = 0.0682 test: test_noise_loss = 0.0713
epoch 6/10


train: train_noise_loss = 0.0666 test: test_noise_loss = 0.0645
epoch 7/10


train: train_noise_loss = 0.0640 test: test_noise_loss = 0.0666
epoch 8/10


train: train_noise_loss = 0.0624 test: test_noise_loss = 0.0612
epoch 9/10
```

```
train: train_noise_loss = 0.0604 test: test_noise_loss = 0.0640
epoch 10/10
```

```
train: train_noise_loss = 0.0598 test: test_noise_loss = 0.0626
```

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

[38]:

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

from IPython.display import Image, display

display(Image(filename='./save/test_3/images/generate_epoch_8.png'))

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

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.

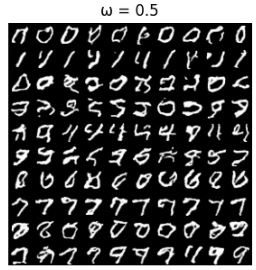
[13]:
```
from utils import sample_images
import matplotlib.pyplot as plt
# ===================================================== #
# YOUR CODE HERE:
#   Try at least 3 different conditional guidance weights and visualize it.
#   Example of using a different omega value:
#       sample_config = DMConfig(omega = ?)
```

```python
#        fig = sample_images(config = sample_config, checkpoint_path = path_to_your_checkpoint)

# Checkpoint path
checkpoint_path = "./save/test_2/best_checkpoint.pth"

torch.serialization.add_safe_globals([DMConfig])

# w values to test
omegas = [-1, 0.0, 0.5, 1.0, 2.0, 5.0, 10.0]

# Prepare test
fig, axes = plt.subplots(
    nrows=len(omegas),
    ncols=1,
    figsize=(6, 3 * len(omegas)),
    constrained_layout=True
)

# Loop over  , sample and plot
for ax, w in zip(axes, omegas):
    config = DMConfig(omega=w)
    grid = sample_images(config=config, checkpoint_path=checkpoint_path)
    ax.imshow(grid)
    ax.set_title(f"  = {w}")
    ax.axis("off")

plt.show()

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

ω = -1

ω = 0.0

ω = 0.5



ω = 1.0

ω = 2.0

ω = 5.0

ω = 10.0

Inline Question: Based on your experiment, discuss how the conditional guidance weight affects the quality and diversity of generation. (1 point)

Your answer:

The conditional guidance weight has a very large affect on the quality and diversity of generation in a few ways (summarized in the table below):

Guidance Weight	Quality & Sharpness	Diversity	Class Fidelity	Notes
–1.0	Low	Very high	Low	highly rely on the unconditional network. so poor quality
0 – 0.5	Low (fuzzy, soft strokes)	Very high	Low	Poor digit legibility
1 – 2	Moderate (clear but natural)	Moderate	High	Sweet spot—legible yet varied handwriting styles
5	High (very sharp, bold edges)	Low	Very high	Stereotyped, grid-like artifacts begin to appear
10	Very high (extreme sharpness)	Very low	Extremely high	Severe mode collapse; samples look almost identical

Overall - low conditional guidance weight (0.0-1.0) leads to increased digit diversity but low fidelity, as you increase w your fidelity increases substantially but your class diversity decreases proportionally.

2.6 Customize your own model (5 points) Now let's experiment by modifying some hyper-parameters 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=?, ...)`.

[14]:
```
# train_config_new = DMConfig()
# print(train_config_new)

# default T = 400, mask_p = 0.1, num_feat = 64, beta_1 = 1e-4, beta_T= 2e-2
print(f"default config: {DMConfig()}")
train_config_new = DMConfig(T = 200, mask_p = 0.2, num_feat = 128, beta_1 =␣
↪1e-3, beta_T= 2e-1 )
print(f"Modified config: {train_config_new}")
```

```
default 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=64, omega=2.0, batch_size=256, epochs=10,
learning_rate=0.0001, multi_lr_milestones=[20], multi_lr_gamma=0.1)
Modified config: DMConfig(input_dim=(28, 28), num_channels=1,
condition_mask_value=-1, num_classes=10, T=200, beta_1=0.001, beta_T=0.2,
mask_p=0.2, 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 similar to 2.4, use `solver` funtion to complete the training and sampling process.

[15]:
```python
from utils import solver
solver(dmconfig = train_config_new,
       exp_name = 'modified_env',
       train_loader = train_loader,
       test_loader = test_loader)
```

```
epoch 1/10


train: train_noise_loss = 0.0956 test: test_noise_loss = 0.0441
epoch 2/10


train: train_noise_loss = 0.0410 test: test_noise_loss = 0.0379
epoch 3/10


train: train_noise_loss = 0.0348 test: test_noise_loss = 0.0347
epoch 4/10


train: train_noise_loss = 0.0313 test: test_noise_loss = 0.0315
epoch 5/10


train: train_noise_loss = 0.0302 test: test_noise_loss = 0.0300
epoch 6/10


train: train_noise_loss = 0.0288 test: test_noise_loss = 0.0285
epoch 7/10


train: train_noise_loss = 0.0279 test: test_noise_loss = 0.0325
epoch 8/10
```

```
train: train_noise_loss = 0.0267 test: test_noise_loss = 0.0276
epoch 9/10


train: train_noise_loss = 0.0256 test: test_noise_loss = 0.0245
epoch 10/10


train: train_noise_loss = 0.0252 test: test_noise_loss = 0.0274
```

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

[16]:
```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, ...

from IPython.display import Image, display

display(Image(filename='./save/modified_env/images/generate_epoch_10.png'))

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

Inline Question: Discuss the effects of your modifications after you compare the generation performance under different configurations. (1 point)

Your answer:

Previous config: T = 400, mask_p = 0.1, num_feat = 64, beta_1 = 1e-4, beta_T= 2e-2

New config: T = 200, mask_p = 0.2, num_feat = 128, beta_1 = 1e-3, beta_T= 2e-1

As you can see, what we did are to decrease the num of timesteps but increase the features of U-Net and increase the beta_T. As the MNIST dataset is relatively simple to general rgb images, so decreasing T won't affect too much, but increase the num_features will enhance the network representation, and increase beta_T will make network better ability to generalization (ie. diversity)