KRISH PATEL DDPM

ECE239AS LAB 2

Q1) a) $P(x_1, x_2, x_3, x_4) = p(x_1)p(x_2|x_1)p(x_3|x_2, x_1)p(x_4|x_1, x_2, x_3)$

however, this can be simplified to

$p(x_1)\, p(x_2|x_1)\, p(x_3|x_2)\, p(x_4|x_3)$.

b) $x_1 \sim N(0, 6^2)$ $\quad x_t|x_{t-1} \sim N(x_{t-1}, 6^2)$ $\quad t = 2, 3, 4$

var $x_1 = 6^2$ var $x_2 = 2 \cdot 6^2$ var $x_3 = 3 \cdot 6^2$ var $x_4 = 4 \cdot 6^2$

covariance matrix =

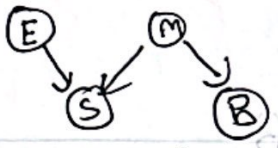
$$\Sigma = \begin{bmatrix} 6^2 & 6^2 & 6^2 & 6^2 \\ 6^2 & 2 \cdot 6^2 & 2 \cdot 6^2 & 2 \cdot 6^2 \\ 6^2 & 2 \cdot 6^2 & 3 \cdot 6^2 & 3 \cdot 6^2 \\ 6^2 & 2 \cdot 6^2 & 3 \cdot 6^2 & 4 \cdot 6^2 \end{bmatrix}$$

c) Taking the inverse of this matrix, $\Sigma \Sigma^{-1} = I$

we get

$$\Sigma^{-1} = \frac{1}{6^2} \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

d) The zeroes are related in the following manner: the values are positive on the diagonal, where $i = j$ where x_i and x_j are adjacent, the values are -1. However when $x_i \neq x_j$ and x_i and x_j are not adjacent, they are zero.



2) a) $P(S, -e \mid -m, -b) \rightarrow$ ~~Using~~

↳ Expanding $\frac{P(-e, -S, -m, -b)}{P(-m, -b)}$

$$= \frac{P(-e)\,P(-S \mid -e, -m)\,P(-m)\,P(-b \mid -m)}{P(-m)\,P(-b \mid m)} = \frac{0.6 \times (0.9)^2}{(0.9)^2} = 0.54$$

b) $P(+m \mid +b, +e, +s) = \frac{P(+m, +b, +e, +s)}{P(+b, +e, +s)}$

$$= \frac{P(+e)\,P(+m)\,P(+b \mid +m)\,P(+s \mid +e, +m)}{P(+e)\,(0.4)(0.1) + (0.8)(0.1)(0.4)(0.9)}$$

$$= \frac{0.04}{0.04 + 0.08 \times 0.36}$$

c) i) Need to show $P(B \mid M, E, S) = P(B \mid M)$ to prove independence.

expanding LHS $= \frac{P(E)\,P(M)\,P(B \mid M)\,P(S \mid E, M)}{P(M, E, S)}$ — (1)

$= \frac{P(E)\,P(M)\,P(B \mid M)\,P(S \mid E, M)}{P(E)\,P(M)\,P(S \mid E, M)} = P(B \mid M) = RHS.$

Proof for $E \perp\!\!\!\perp M$ $\quad P(E, M) = \sum_B \sum_S P(E)\,P(M)\,P(B \mid M)\,P(S \mid E, M)$

$P(E)\,P(M) \sum_B P(B \mid M) \cdot \sum_S P(S \mid E, M)$

$\sum_B P(B \mid M) = 1$, $\sum_S P(S \mid E, M) = 1$

thus $B \perp\!\!\!\perp \{E, S\} \mid M$, $M(B) = M$

$= P(E) \cdot P(M)$

Using (1) $\quad \frac{P(E, M)\,P(B \mid M)\,P(S \mid E, M)}{P(M, E, S)} = \frac{P(S, M, E)\,P(B \mid M)}{P(M, E, S)}$

Hence proved.

2c) ii) If we can show

$$P(E, B \mid M, S) = P(E \mid M, S)\, P(B \mid M, S)$$

then we can prove $E \perp\!\!\!\perp B \mid \{M, S\}$

$$P(E, B \mid M, S) = \frac{P(E)\, P(M)\, P(B \mid M)\, P(S \mid E, M)}{P(M, S)}$$

$P(E, M, S)$ as $E \perp\!\!\!\perp M$

thus, the term is equal to $P(E \mid M, S)\, P(B \mid M)$ ②

= also, $B \perp\!\!\!\perp S \mid M$, thus

$② = P(E \mid M, S)\, P(B \mid M, S)$ which is the RHS.

iii) $E \perp\!\!\!\perp M \mid S$ based on the assumption.

according to (ii) & , $E \perp\!\!\!\perp B \mid \{M, S\}$ thus $M(E) = \{M, S\}$

iv) Generally, the markov Blanket of a node in a Bayesian networds contain parent, childdren, and other parents of its children. In this case, for M, both the markov blanket only contain its parents and children based on the depth of this network.

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.

```
%load_ext autoreload
%autoreload 2
```

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!).

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

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

```
import os

# List files in the current directory
files = os.listdir('.')
print(files)

# Change directory
os.chdir("drive/MyDrive/Lab3-Diffusion/")

# List files in the new directory
files = os.listdir('.')
print(files)
```

```
['.config', 'drive', 'sample_data']
['utils.py', 'project3_Diffusion.pdf', '.DS_Store', 'pics',
'__pycache__', 'data', 'ResUNet.py', 'DDPM.py', 'save',
'ConditionalDDPM.ipynb']
```

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:

```
# 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!
```

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.)

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 $\beta_1, \ldots, \beta_T$.
- 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.

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 x_0 is sampled from the data distribution $q(\mathrm{x}_0)$, 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:

$q(\mathrm{x}_t \vee \mathrm{x}_0) = N(\mathrm{x}_t; \sqrt{\acute{\alpha}_t}\mathrm{x}_0, (1-\acute{\alpha}_t)I)$, $\alpha_t := 1-\beta_t$ and $\acute{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, we can get an intermediate state of the diffusion process: $\mathrm{x}_t = \sqrt{\acute{\alpha}_t}\mathrm{x}_0 + \sqrt{(1-\acute{\alpha}_t)}\epsilon$. The model takes the x_t and t as inputs, and predict a noise, i.e. $\epsilon_\theta(\sqrt{\acute{\alpha}_t}\mathrm{x}_0 + \sqrt{(1-\acute{\alpha}_t)}\epsilon, t)$. 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.

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 x_0, we also sample the condition 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 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 $\epsilon_\theta(\mathrm{x}_t, \mathrm{c}, t)$ and the unconditional prediction $\epsilon_\theta(\mathrm{x}_t, t)$. The corrected noise $\tilde{\epsilon}_t$ is then used to update x_t in step 5. **Here we follow the setting of DDPM paper and define $\sigma_t = \sqrt{\beta_t}$.**

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.

```
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:

```
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.**

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

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

```
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!')
```

```
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
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`: $\acute{\alpha}_t = \prod_{s=1}^{t} \alpha_s$

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

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

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

```
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!')

Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
```

```
hello3
All tests passed!
```

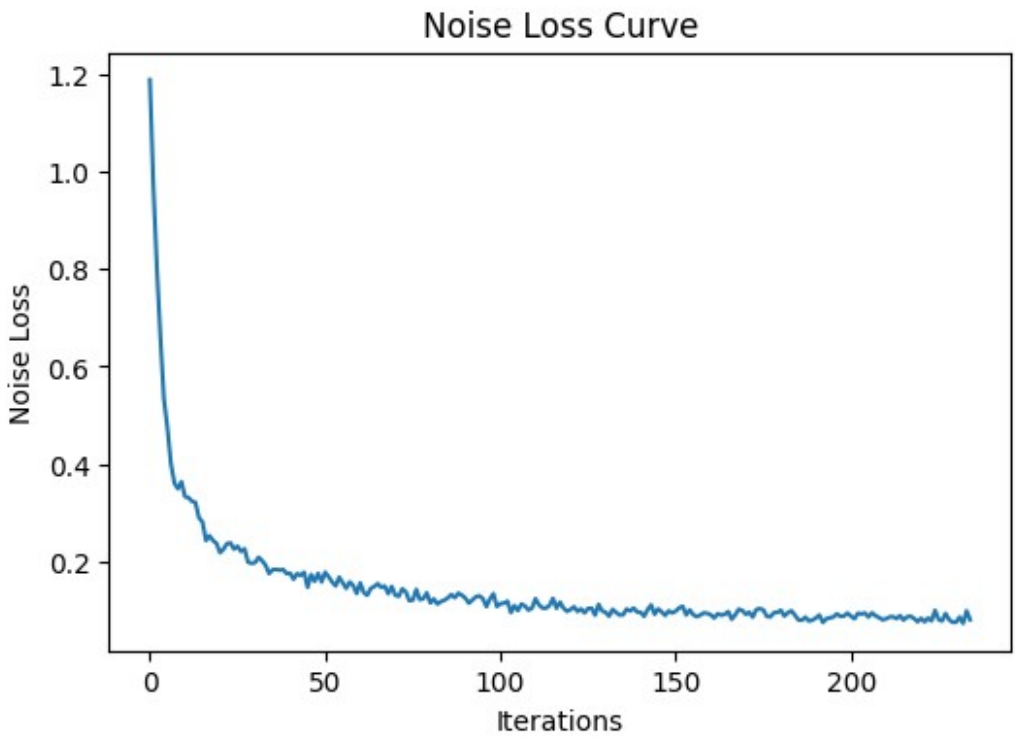
2.2 Training process (5 points)

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.

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

```
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
```



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.

```
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:

```
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`.

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

```
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
epoch 1/10


train: train_noise_loss = 0.1530 test: test_noise_loss = 0.0832
epoch 2/10


train: train_noise_loss = 0.0767 test: test_noise_loss = 0.0728
epoch 3/10


train: train_noise_loss = 0.0671 test: test_noise_loss = 0.0646
epoch 4/10
```

```
train: train_noise_loss = 0.0633 test: test_noise_loss = 0.0585
epoch 5/10


train: train_noise_loss = 0.0598 test: test_noise_loss = 0.0603
epoch 6/10


train: train_noise_loss = 0.0583 test: test_noise_loss = 0.0562
epoch 7/10


train: train_noise_loss = 0.0561 test: test_noise_loss = 0.0570
epoch 8/10


train: train_noise_loss = 0.0549 test: test_noise_loss = 0.0557
epoch 9/10


train: train_noise_loss = 0.0543 test: test_noise_loss = 0.0550
epoch 10/10


train: train_noise_loss = 0.0535 test: test_noise_loss = 0.0531
```

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

```
# ================================================== #
# 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
from PIL import Image

image_path = "./save/krishpatel/images/generate_epoch_10.png"
image = Image.open(image_path)
plt.imshow(image)
plt.axis('off')
plt.show()
# ================================================== #
```

2.5 Exploring the conditional guidance weight (3 points)

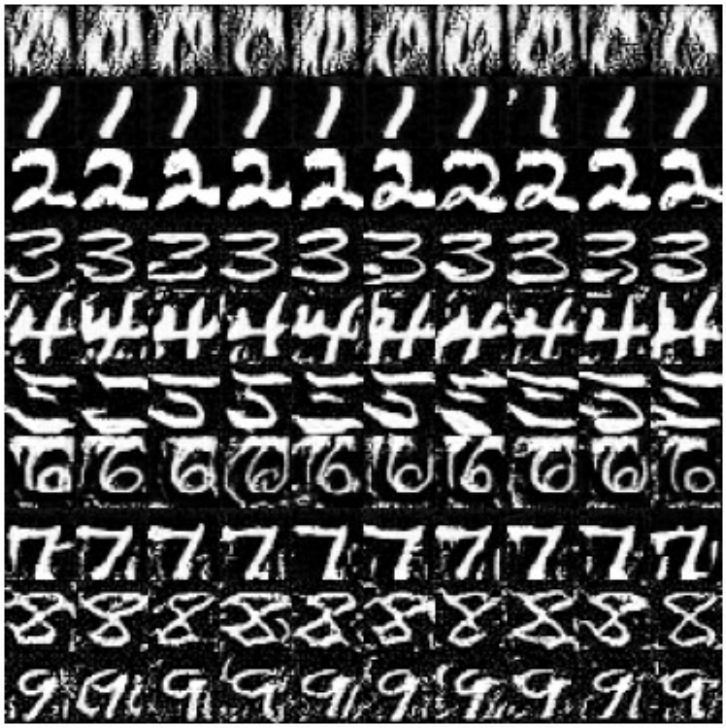
The generated images from the previous training-sampling process is using the default conditional guidance weight $\omega=2$. Now with the best checkpoint, please try at least 3 different ω values and visualize the generated images. You can use the provided function `sample_images` to get a combined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

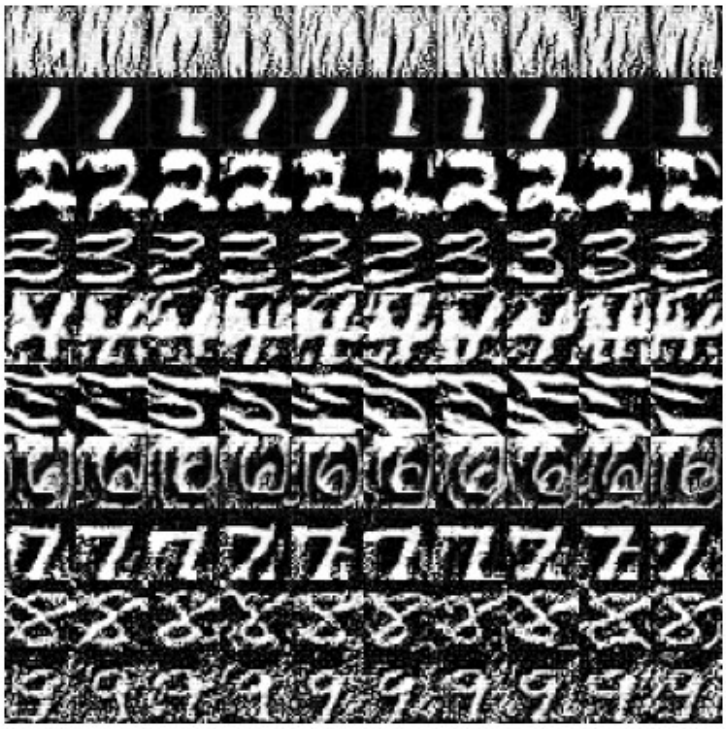
```
from utils import sample_images

checkpoint_path = "./save/krishpatel/best_checkpoint.pth"
omega1 = 0
omega2 = 10
omega3 = 30
omega4 = 50

fig = sample_images(config = DMConfig(omega = omega1), checkpoint_path
= checkpoint_path)
fig2 = sample_images(config = DMConfig(omega = omega2),
checkpoint_path = checkpoint_path)
fig3 = sample_images(config = DMConfig(omega = omega3),
checkpoint_path = checkpoint_path)
fig4 = sample_images(config = DMConfig(omega = omega4),
checkpoint_path = checkpoint_path)

plt.imshow(fig)
plt.axis('off')
plt.show()
```

```
plt.imshow(fig2)
plt.axis('off')
plt.show()

plt.imshow(fig3)
plt.axis('off')
plt.show()

plt.imshow(fig4)
plt.axis('off')
plt.show()
```

```
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
```

```
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
```





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

Your answer: I noticed that when I increased the conditional guidance weight, the numbers in the images became clearer and more saturated. This makes sense because the model is pushed

to create more distinct images based on the given class. However, when I set the weight too high, like at omega equals 30 or 50, the images became overly saturated.

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=?, ...)`.

```
# ================================================== #
# YOUR CODE HERE:
#   Your new configuration:
#   train_config_new = DMConfig(...)
train_config_new = DMConfig(T=1000, num_feat=256, omega=8)

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

```
DMConfig(input_dim=(28, 28), num_channels=1, condition_mask_value=-1,
num_classes=10, T=1000, beta_1=0.0001, beta_T=0.02, mask_p=0.1,
num_feat=256, omega=8, 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.

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

```
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
Hello
hello2
hello3
```

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=?, ...)`.

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

train_config_new = DMConfig(T=900, num_feat=256, omega=8)
# ================================================== #
print(train_config_new)
```

```
DMConfig(input_dim=(28, 28), num_channels=1, condition_mask_value=-1,
num_classes=10, T=900, beta_1=0.0001, beta_T=0.02, mask_p=0.1,
num_feat=256, omega=8, 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.

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

```
epoch 1/10


train: train_noise_loss = 0.1229 test: test_noise_loss = 0.0592
epoch 2/10


train: train_noise_loss = 0.0519 test: test_noise_loss = 0.0446
epoch 3/10
```

```
train: train_noise_loss = 0.0453 test: test_noise_loss = 0.0447
epoch 4/10


train: train_noise_loss = 0.0419 test: test_noise_loss = 0.0409
epoch 5/10


train: train_noise_loss = 0.0398 test: test_noise_loss = 0.0373
epoch 6/10


train: train_noise_loss = 0.0380 test: test_noise_loss = 0.0383
epoch 7/10


train: train_noise_loss = 0.0371 test: test_noise_loss = 0.0356
epoch 8/10


train: train_noise_loss = 0.0358 test: test_noise_loss = 0.0349
epoch 9/10


train: train_noise_loss = 0.0355 test: test_noise_loss = 0.0352
epoch 10/10


train: train_noise_loss = 0.0350 test: test_noise_loss = 0.0348
```

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

```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
from PIL import Image
image_path = "./save/krishpatel_new/images/generate_epoch_10.png"
image = Image.open(image_path)
plt.imshow(image)
plt.axis('off')
plt.show()
```

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

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

Your answer: Boosting the number of time steps, features, and the conditional guidance weight resulted in generating crisper and more distinct images compared to earlier attempts. This upgrade makes sense because increasing the number of time steps and features helps create clearer images with finer details. It allows the model to better preserve intricate patterns by adding and removing noise over a longer period. Furthermore, as demonstrated earlier, increasing the conditional guidance weight leads to producing more prominent and recognizable numerical images. However, due to the large omega, the zeroes and twos turned out to be more blurry.