Assignment 2-2
DDPM Practice

IMVFX Autumn
November 23, 2023

Sample Code :
IMVFX_HW2_DDPM.ipynb - Colaboratory (google.com)
Colab tutorial :
IMVFX_Google_Colab_Tutorial.ipynb – Colaboratory
DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ
https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Outline

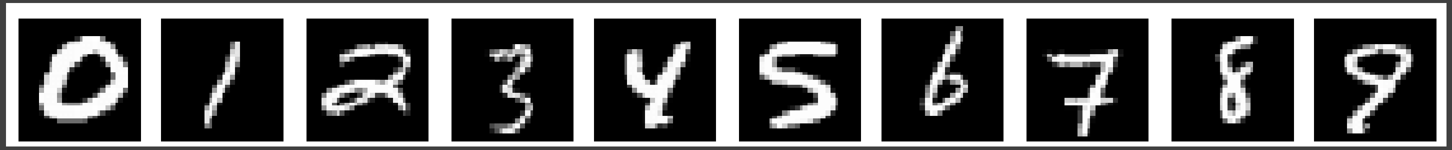
- Introduction
- DDPM
- The structure of DDPM
- Requirements
- Reminder
- Submission
- Reference

Introduction

In this homework, you are going to use DDPM (Denoising Diffusion Probabilistic Models) to generate images.

Use the dataset below to train the model:

- MNIST dataset (60000 images): Google drive link
- Anime Face dataset (21376 images): Google drive link



DDPM

1. DDPM is invented by Jonathan Ho et al. in 2020.
2. Include forward and backward process.

- Forward process: Add random noise into images based on the current step t.
- Backward process: Estimate the noise that was added into the images.

3. The algorithm of DDPM shown as below.

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 $\mathbf{x}_0$

The structure of DDPM

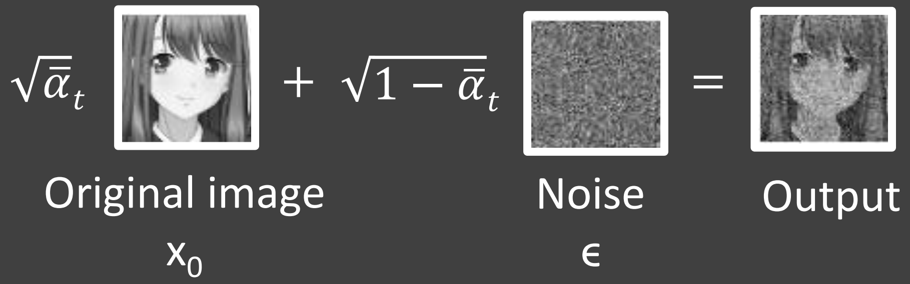
Algorithm 1 Training

1: **repeat**
2: $\quad \mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $\quad t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4: $\quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: $\quad$ Take gradient descent step on

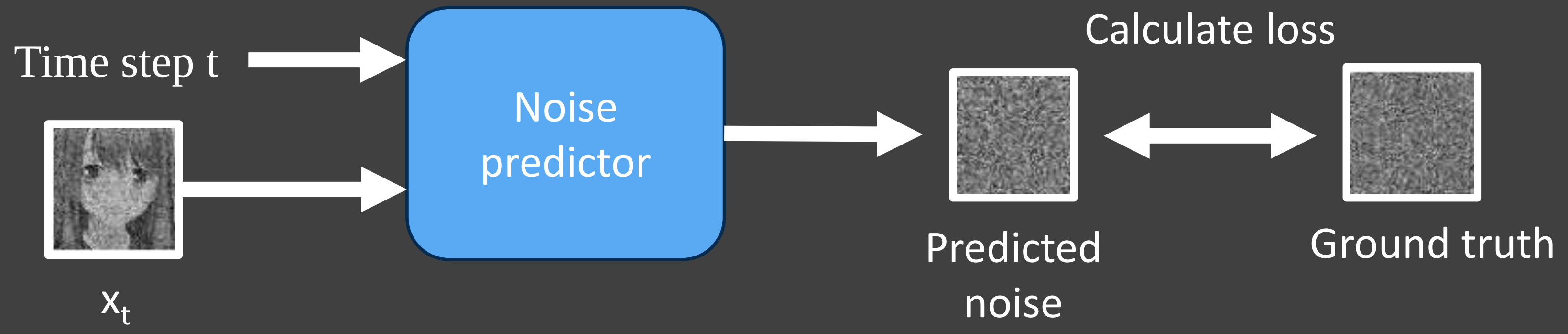
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\underline{\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}}, t) \right\|^2$$

6: **until** converged

Forward process



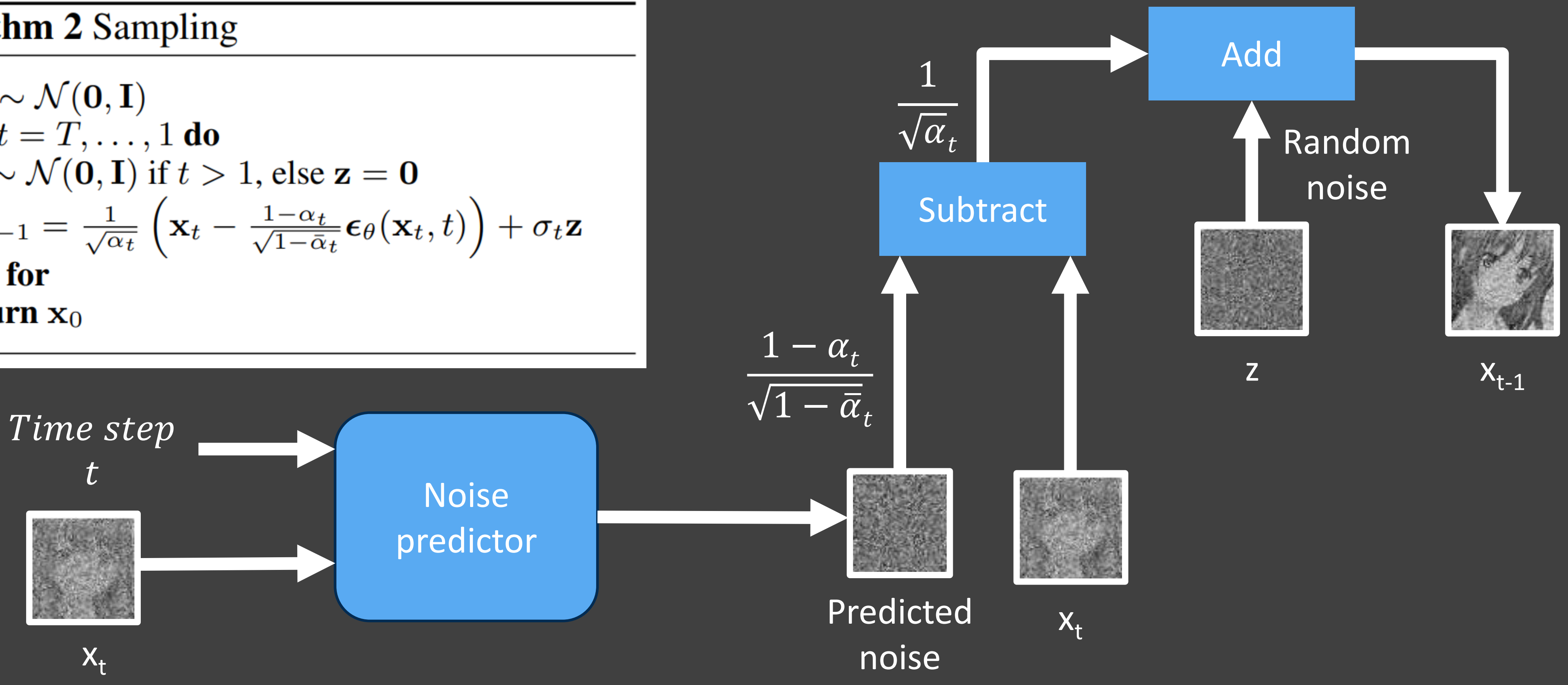
Backward process



The structure of DDPM

Algorithm 2 Sampling

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



For implementation B.1-1

Steps for training DDPM on the MNIST dataset

1. Set up the parameters
2. Load the data
3. Build DDPM model
4. Training

For implementation B.1-1

Step 1. Set up the parameters

Set up the parameters for training.

You can change the parameters to improve your training result. (ex. Set the lr = 0.0005, batch_size = 256)

```
# Root directory for the MNIST dataset
dataset_path = f"{workspace_dir}/mnist_dataset"

# The path to save the model
model_store_path = f"{workspace_dir}/mnist.pt"

# Batch size during training
batch_size = 128

# Number of training epochs
n_epochs = 30

# Learning rate for optimizers
lr = 0.001

# Number of the forward steps
n_steps = 1000

# Initial beta
start_beta = 1e-4

# End beta
end_beta = 0.02

# Getting device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")

# List to keep track of loss
loss_list = []
```

For implementation B.1-1

Step 2. Load the data

Then we can load the data by ImageFolder() and make the data loader by DataLoader().

You can utilize transformation functions for data augmentation.

```
# Load the data, convert it to grayscale, and then normalize it to the range of [-1, 1]
dataset = ImageFolder(root=dataset_path, transform=Compose([
        Grayscale(),
        ToTensor(),
        Lambda(lambda x: (x - 0.5) * 2)]
))

# Make the data loader
dataloader = DataLoader(dataset, batch_size, shuffle=True)
```

For implementation B.1-1

Step 3. Build the DDPM model

In the forward process, you need to add the noise to the images.

```
def forward(self, x0, t, eta=None):
    n, channel, height, width = x0.shape
    alpha_bar = self.alpha_bars[t]

    if eta is None:
        eta = torch.randn(n, channel, height, width).to(self.device)

    noise = alpha_bar.sqrt().reshape(n, 1, 1, 1) * x0 + (1 - alpha_bar).sqrt().reshape(n, 1, 1, 1) * eta
    return noise
```

In the backward process, you need to design a noise predictor to predict the noise that was added to the images during the forward process.

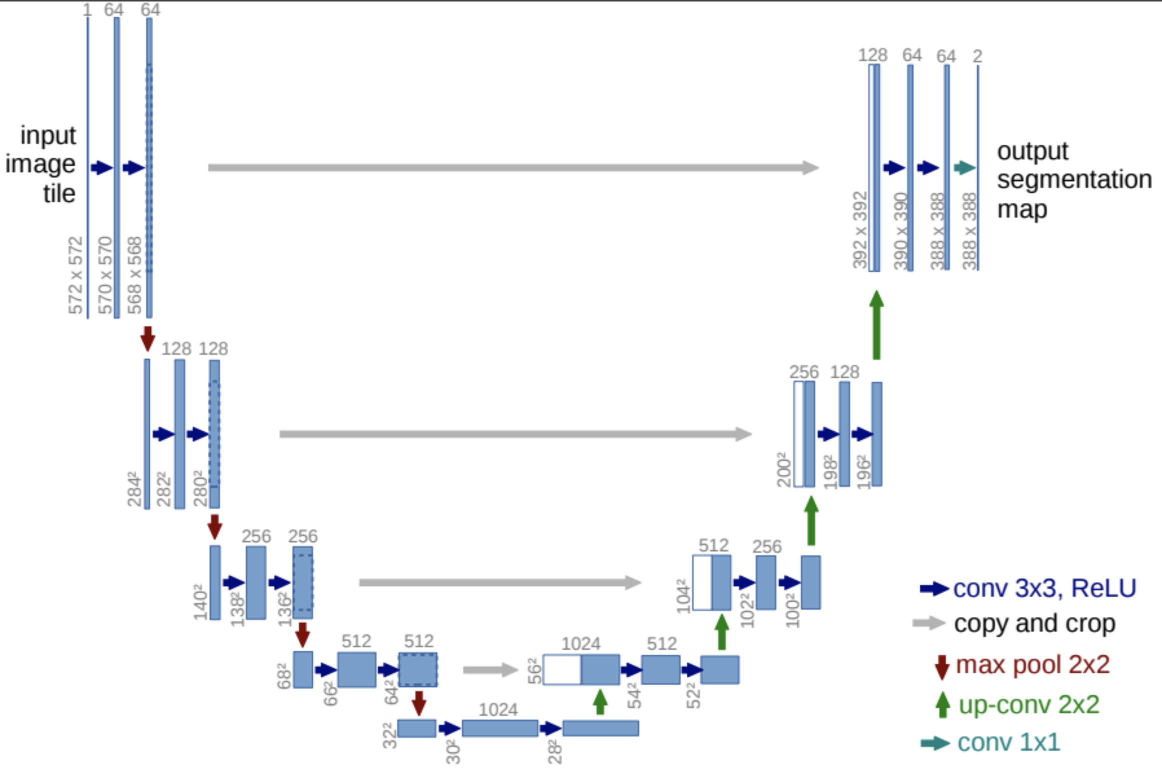
```
def backward(self, x, t):
    return self.noise_predictor(x, t)
```

For implementation B.1-1

Step 3. Build the DDPM model

The noise predictor should be constructed with a U-Net-like architecture.

The basic U-Net is shown below.



From: U-Net

For implementation B.1-1

Step 3. Build the DDPM model

The architecture of the noise predictor in sample code is very simple, and you can design your own.

Please note that you need to input the time embedding into the U-Net block.

You can refer to the link for information about positional encoding.

```
[ ] # Create the time embedding
    def time_embedding(n, d):
        embedding = torch.zeros(n, d)
        wk = torch.tensor([1 / 10000 ** (2 * j / d) for j in range(d)])
        wk = wk.reshape((1, d))
        t = torch.arange(n).reshape((n, 1))
        embedding[:, ::2] = torch.sin(t * wk[:, ::2])
        embedding[:, 1::2] = torch.cos(t * wk[:, ::2])
        return embedding
```

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right),$$
$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$

For implementation B.1-1

Step 4. Training

Then we can start training.

First, we need to pass parameters to initialize the trainer.

We use Adam() as optimizer to update the weight of network and use MSELoss() to compute the mean squared error.

```
trainer(ddpm_mnist, dataloader, n_epochs=n_epochs, optim=Adam(ddpm_mnist.parameters(), lr),
        loss_funciton=nn.MSELoss(), device=device, model_store_path=model_store_path)
```

For implementation B.1-1

Step 4. Training

In each training iteration, we perform the following steps:

1. Load data from the data loader.

```
x0 = batch[0].to(device)
n = len(x0)
```

2. Pick random noise for each of the images in the batch.

```
eta = torch.randn_like(x0).to(device)
t = torch.randint(0, n_steps, (n,)).to(device)
```

3. Compute the noisy image according to x0 and the time step t.

```
noises = ddpm(x0, t, eta)
```

4. Get model estimation of noise based on the images and the time step.

```
eta_theta = ddpm.backward(noises, t.reshape(n, -1))
```

5. Calculate the MSE loss between the injected noise and the predicted noise.

```
loss = loss_funciton(eta_theta, eta)
```

6. Initialize the optimizer's gradient and then update the network's weights.

```
optim.zero_grad()
loss.backward()
optim.step()
```

7. Aggregate the loss values from each iteration to compute the loss value for an epoch.

```
epoch_loss += loss.item() * len(x0) / len(dataloader.dataset)
```

For implementation B.1-1

Step 4. Training

Remember to save the model weight when you are training.

You can use torch.save(model, PATH), torch.save(model.state_dict(), PATH) for saving the model weight.

When you want to evaluate data or train on pretrained weight, you can use torch.load(PATH), model.load_state_dict(torch.load(PATH)).

Example on pytorch.org.

For implementation B.1-2, B.1-3

Plot the loss and save images

For plot loss:

Save the loss values when you are training, you can save them in a list.

Then use the function in matplotlib.pyplot to plot the loss.

For save images:

You can generate images and the GIF of the diffusion process by generate_new_images() in sample code.

```
[ ]  images = generate_new_images(
             ddpm_mnist,
             n_samples=100,
             device=device,
             gif_name="mnist.gif"
         )
```

Then save the result by matplotlib.pyplot.savefig().

Design the training process of DDPM for the Anime Face dataset

You need to train the diffusion model to generate grayscale images or color ones by the Anime Face dataset.

If you choose to implement the color images generation model, you will get additional bonus.

Please note that **the size of the input images should be 64*64**.

Before designing the architecture of the noise predictor, it is important to have a solid understanding of the principles underlying the torch.nn library and how to use it.

For implementation B.1-5, B.1-6

Plot the loss and save images for the Anime Face dataset

The process is same as implementation B.1-2, B.1-3 but for the Anime Face dataset.

Requirement - Implementation

Implementation(40%):

✓ For the MNIST dataset:

B.1-1 Train a DDPM and generate the images. (Please use image size 28*28) (5%)

B.1-2 Plot the loss value of DDPM versus training iterations. (Please upload the image to E3) (5%)

B.1-3 Store your generated image in 5x5 grid. (Please upload the image to E3) (5%)

Requirement - Implementation

Implementation(40%):

✓ For the Anime Face dataset

B.1-4 Train a DDPM and generate grayscale or color images (Please use image size 64*64) (15%)

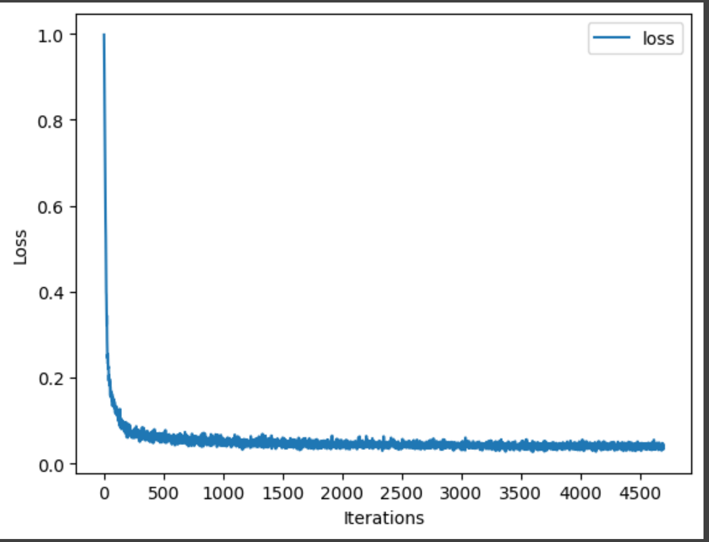
- If you generate only grayscale images, you will get only 10%
- If you generate color images, you will get 15%

B.1-5 Plot the loss value of DDPM versus training iterations (Please upload the image to E3) (5%)

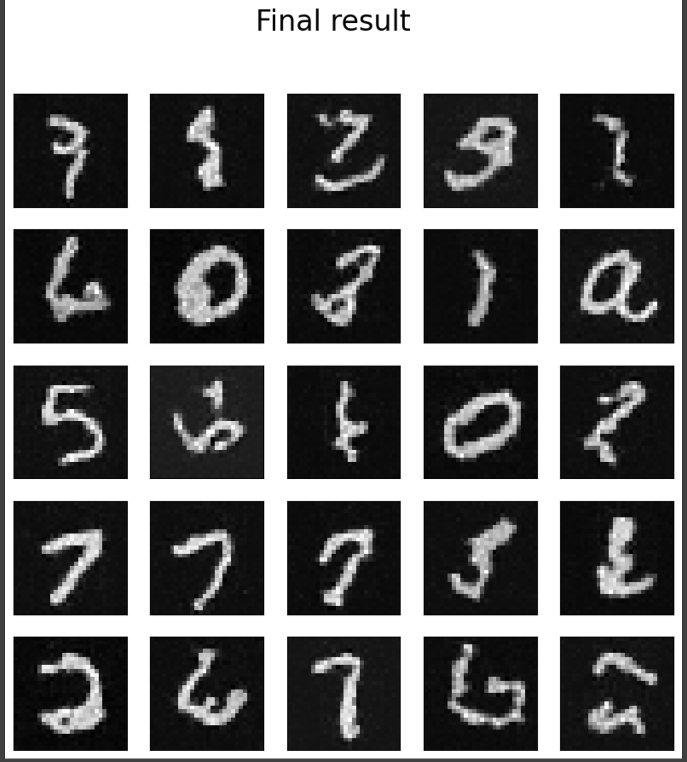
B.1-6 Store your generated image in 5x5 grid. (Please upload the image to E3) (5%)

Requirement - Implementation

Example for B.1-2

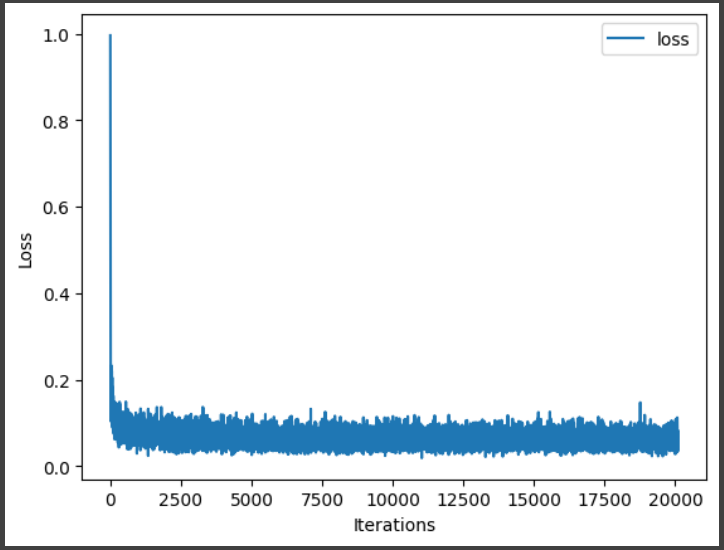


Example for B.1-3



Requirement - Implementation

Example for B.1-5



Example for B.1-6

Requirement - Report

2. Report(15%): You can write in Chinese or English.

B.2-1 Please provide a brief introduction about your experiments on the MNIST and Anime Face dataset, including details such as setting of hyperparameter, data augmentation techniques used, network structure, etc. (5%)

B.2-2 Comparing the generation quality between DCGAN and DDPM: Compare the resolution, level of detail, and diversity of generated images. You can assess them using metrics such as FID, IS, or subjective evaluations. Encourage writing more about the experiment you want to discuss. (10%)

Reminder

- If you refer any code from GitHub or other open source, you have to properly cite the source and comment on codes belonging to the open source. Otherwise, you will get a penalty of 20 points or more.
- You should work on all the given images.
- It takes much time to train the models, so better to start your work early.
- Feel free to modify any code provided by TA or write the code yourself.

Submission

1. Your python source code (.py or .ipynb)
2. Your report (Named as report_<your student ID>.pdf)
3. A image generated by the model trained on the MNIST dataset in a 5*5 grid. (Named as result_mnist.jpg)
4. A plot of loss values for the model trained on the MNIST dataset. (Named as loss_mnist.jpg)
5. A image generated by the model trained on the Anime Face dataset in a 5*5 grid. (Named as result_anime.jpg)
6. A plot of loss values for the model trained on the Anime Face dataset. (Named as loss_anime.jpg)
7. A README describing how to run your code. (Named as readme.txt)

Zip all the files above to <your student ID>_hw2_2.zip and upload the zip file to E3 before the deadline.

Reference

DDPM implementation
https://github.com/lucidrains/denoising-diffusion-pytorch
https://medium.com/mlearning-ai/enerating-images-with-ddpms-a-pytorch-implementation-cef5a2ba8cb1
https://pytorch.org/docs/stable/index.html

Data source
https://github.com/teavanist/MNIST-JPG
https://www.kaggle.com/datasets/soumikrakshit/anime-faces

Paper of DDPM
https://arxiv.org/pdf/2006.11239.pdf