IMVFX Assignment 2-2 DDPM Practice

資工科碩 312551080 紀軒宇

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%)

- MNIST
  - hyperparameter
    - batch_size : 128
      batch size while training
    - n_epochs : 100
      training epochs
    - lr : 0.001
      learning rate
    - n_steps : 1000
      forward steps
    - start_beta : 1e-4
      parameter for DDPM
    - end_beta : 0.02
      parameter for DDPM
    - time_embedding_dim : 256
      parameter for time_embedding() function
  - network structure

```
DDPM(
(noise_predictor): UNet(
  (time_step_embedding): Embedding(1000, 256)
  (time_step_encoder1): Sequential(
    (0): Linear(in_features=256, out_features=1, bias=True)
    (1): SiLU()
    (2): Linear(in_features=1, out_features=1, bias=True)
  )
  (block1): Sequential(
    (0): LayerNorm((1, 28, 28), eps=1e-05, elementwise_affine=True)
    (1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.2)
  )
  (down1): Conv2d(8, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (time_step_encoder2): Sequential(
    (0): Linear(in_features=256, out_features=8, bias=True)
    (1): SiLU()
    (2): Linear(in_features=8, out_features=8, bias=True)
```

```
)
(block2): Sequential(
  (0): LayerNorm((8, 14, 14), eps=1e-05, elementwise_affine=True)
  (1): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): LeakyReLU(negative_slope=0.2)
)
(down2): Conv2d(16, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(time_step_encoder3): Sequential(
  (0): Linear(in_features=256, out_features=16, bias=True)
  (1): SiLU()
  (2): Linear(in_features=16, out_features=16, bias=True)
)
(block3): Sequential(
  (0): LayerNorm((16, 7, 7), eps=1e-05, elementwise_affine=True)
  (1): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): LeakyReLU(negative_slope=0.2)
)
(down3): Sequential(
  (0): Conv2d(32, 32, kernel_size=(2, 2), stride=(1, 1))
  (1): LeakyReLU(negative_slope=0.2)
  (2): Conv2d(32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(time_step_encoder_mid): Sequential(
  (0): Linear(in_features=256, out_features=32, bias=True)
  (1): SiLU()
  (2): Linear(in_features=32, out_features=32, bias=True)
)
(block_mid): Sequential(
  (0): LayerNorm((32, 3, 3), eps=1e-05, elementwise_affine=True)
  (1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): LeakyReLU(negative_slope=0.2)
)
(up1): Sequential(
  (0): ConvTranspose2d(32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (1): LeakyReLU(negative_slope=0.2)
  (2): ConvTranspose2d(32, 32, kernel_size=(2, 2), stride=(1, 1))
)
(time_step_encoder4): Sequential(
  (0): Linear(in_features=256, out_features=64, bias=True)
```

```
        (1): SiLU()
        (2): Linear(in_features=64, out_features=64, bias=True)
      )
      (block4): Sequential(
        (0): LayerNorm((64, 7, 7), eps=1e-05, elementwise_affine=True)
        (1): Conv2d(64, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): LeakyReLU(negative_slope=0.2)
      )
      (up2): ConvTranspose2d(16, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (time_step_encoder5): Sequential(
        (0): Linear(in_features=256, out_features=32, bias=True)
        (1): SiLU()
        (2): Linear(in_features=32, out_features=32, bias=True)
      )
      (block5): Sequential(
        (0): LayerNorm((32, 14, 14), eps=1e-05, elementwise_affine=True)
        (1): Conv2d(32, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): LeakyReLU(negative_slope=0.2)
      )
      (up3): ConvTranspose2d(8, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (time_step_encoder6): Sequential(
        (0): Linear(in_features=256, out_features=16, bias=True)
        (1): SiLU()
        (2): Linear(in_features=16, out_features=16, bias=True)
      )
      (block6): Sequential(
        (0): LayerNorm((16, 28, 28), eps=1e-05, elementwise_affine=True)
        (1): Conv2d(16, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): LeakyReLU(negative_slope=0.2)
        (4): LayerNorm((8, 28, 28), eps=1e-05, elementwise_affine=True)
        (5): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (7): LeakyReLU(negative_slope=0.2)
      )
      (final_layer): Conv2d(8, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
)
```

- loss function
  - MSE Loss

- other techniques used
    - Adam optimizer

- Anime Face dataset
    - hyperparameter
        - batch_size : 128
          batch size while training
        - n_epochs : 100
          training epochs
        - lr : 0.0005
          learning rate
        - n_steps : 1000
          forward steps
        - start_beta : 1e-4
          parameter for DDPM
        - end_beta : 0.02
          parameter for DDPM
        - time_embedding_dim : 256
          parameter for time_embedding() function
    - network structure

```
DDPM(
  (noise_predictor): UNet(
    (time_step_embedding): Embedding(1000, 256)
    (time_step_encoder1): Sequential(
      (0): Linear(in_features=256, out_features=1, bias=True)
      (1): SiLU()
      (2): Linear(in_features=1, out_features=1, bias=True)
    )
    (block1): Sequential(
      (0): LayerNorm((3, 64, 64), eps=1e-05, elementwise_affine=True)
      (1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2)
    )
    (down1): Conv2d(32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (time_step_encoder2): Sequential(
      (0): Linear(in_features=256, out_features=32, bias=True)
      (1): SiLU()
      (2): Linear(in_features=32, out_features=32, bias=True)
    )
    (block2): Sequential(
      (0): LayerNorm((32, 32, 32), eps=1e-05, elementwise_affine=True)
```

```
    (1): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.2)
  )
  (down2): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (time_step_encoder3): Sequential(
    (0): Linear(in_features=256, out_features=64, bias=True)
    (1): SiLU()
    (2): Linear(in_features=64, out_features=64, bias=True)
  )
  (block3): Sequential(
    (0): LayerNorm((64, 16, 16), eps=1e-05, elementwise_affine=True)
    (1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.2)
  )
  (down3): Sequential(
    (0): Conv2d(128, 128, kernel_size=(2, 2), stride=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(128, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
  (time_step_encoder_mid): Sequential(
    (0): Linear(in_features=256, out_features=128, bias=True)
    (1): SiLU()
    (2): Linear(in_features=128, out_features=128, bias=True)
  )
  (block_mid): Sequential(
    (0): LayerNorm((128, 7, 7), eps=1e-05, elementwise_affine=True)
    (1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.2)
  )
  (up1): Sequential(
    (0): ConvTranspose2d(128, 128, kernel_size=(5, 5), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): ConvTranspose2d(128, 128, kernel_size=(2, 2), stride=(1, 1))
  )
  (time_step_encoder4): Sequential(
    (0): Linear(in_features=256, out_features=256, bias=True)
    (1): SiLU()
    (2): Linear(in_features=256, out_features=256, bias=True)
  )
```

```
    (block4): Sequential(
      (0): LayerNorm((256, 16, 16), eps=1e-05, elementwise_affine=True)
      (1): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2)
    )
    (up2): ConvTranspose2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (time_step_encoder5): Sequential(
      (0): Linear(in_features=256, out_features=128, bias=True)
      (1): SiLU()
      (2): Linear(in_features=128, out_features=128, bias=True)
    )
    (block5): Sequential(
      (0): LayerNorm((128, 32, 32), eps=1e-05, elementwise_affine=True)
      (1): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2)
    )
    (up3): ConvTranspose2d(32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (time_step_encoder6): Sequential(
      (0): Linear(in_features=256, out_features=64, bias=True)
      (1): SiLU()
      (2): Linear(in_features=64, out_features=64, bias=True)
    )
    (block6): Sequential(
      (0): LayerNorm((64, 64, 64), eps=1e-05, elementwise_affine=True)
      (1): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2)
      (4): LayerNorm((32, 64, 64), eps=1e-05, elementwise_affine=True)
      (5): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (7): LeakyReLU(negative_slope=0.2)
    )
    (final_layer): Conv2d(32, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
)
```

- loss function
    - MSE Loss
- other techniques used
    - Adam optimizer
    - CosineAnnealing learning rate scheduler

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%)

2.2.1 生成圖片品質及多樣性

生成圖片品質部分，直接使用最後的權重分別產生 25 張圖片做比較，可以發現由 DCGAN 生成的圖片在面部細節部分明顯相較 DDPM 要好。

DCGAN

DDPM

生成圖片多樣性部分，我使用 FID (Frechet Inception Distance score) 作為 metric 做比較，實作部分我使用 GitHub 上用於計算 FID 的套件[1]，以本次作業提供的 anime_face 資料集作為真實資料的分布，並分別以 DCGAN 和 DDPM 產生 1000 張圖片去和 anime_face 資料集做比較， FID 越小即代表其與資料及分布越接近，實驗結果分別如下：

DCGAN

```
(imvfx) E:\ML\IMVFX>python -m pytorch_fid hw2-1\anime_exp hw2-2\anime_face_dataset\data
100%|████████████████████████████████████████| 20/20 [00:03<00:00,  5.28it/s]
100%|███████████████████████████████████████| 428/428 [00:40<00:00, 10.54it/s]
FID:  47.09879323545138
```

DDPM

```
(imvfx) E:\ML\IMVFX>python -m pytorch_fid hw2-2\anime_exp hw2-2\anime_face_dataset\data
100%|████████████████████████████████████████| 20/20 [00:03<00:00,  5.01it/s]
100%|███████████████████████████████████████| 428/428 [00:37<00:00, 11.53it/s]
FID:  75.47974676402893
```

從結果中可以明顯看出，由 DCGAN 生成的圖片相較 DDPM 有更好的多樣性。

2.2.2 向量 z 如何影響生成結果

作業 2-1 中有討論過關於向量 z 如何影響 DCGAN 的生成結果，當時利用兩隨機生成的向量 $z1$, $z2$ ，以及用這兩個內插的多個 z 向量作為 generator 的輸出，觀察輸出圖片的變化，最後我的結論為「向量 z 作為 generator 的輸出結果，z 的內容會影響生成影像的特徵，輸入 z 就猶如一個圖片特徵的表示，並且輸出會隨其 $z1, z2$ 所佔的比例而有一個連續性的過度表現」，所以我想看在 DDPM 中觀察與 DCGAN 有何不同。

DCGAN

DCGAN 中的結果沿用作業 2_1 中結果，結果如下：

DDPM

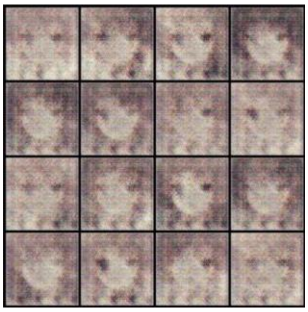
DDPM 使用與上述方法中提到的方式，得到得結果如下：

從結果中可以看到，由兩向量 $z1, z2$ 內插出的多個向量 z 在輸入 DDPM 後得到的圖片中，無法觀察到與於 DCGAN 中觀察到的過渡現象，舉第一個 row 中的圖片為例，第一個 row 中的第一個圖片與第二個圖片相比，頭髮顏色明顯改變，但是接下來兩張圖片髮色就幾乎相同，整體的變化不像 DCGAN 中呈現的線性過渡。

2.2.3 生成圖片隨 epoch 的變化

這個部分，我們分別比較訓練過程中訓練結果隨 epoch 的變化，因為兩模型架構完全不同，所以比較部分的 epoch 可能兩邊不相同，這邊僅對整個過程的變化做比較。

DCGAN



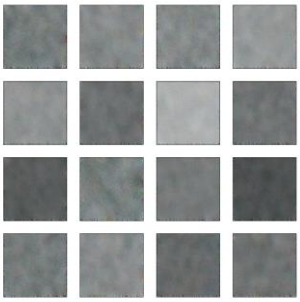
epoch 1

epoch 20

epoch 50

epoch 200(end)

DDPM



epoch 1

epoch 20

epoch 50

epoch 100(end)

觀察這兩模型間的訓練過程，觀察到在一開始時，DCGAN 已生成許多有些許臉部特徵雛形的圖片，相較 DDPM 就只是生成一些充滿雜訊的圖片，在訓練個數十個 epoch 後，DCGAN 生成的臉部特徵部分線條雖然相較一開始更好了，但線條部分還不明顯，DDPM 的部分則是已經可以生成有明顯輪廓的臉部特徵，但生成的圖片色調都相似，統整上述觀察，可以發現 DCGAN 的訓練過程比較像先建構一個臉部的雛型的特徵，接著在一步一步細緻化這些特徵，DDPM 相較之下就比較像將一張充滿噪音的圖片一步一步降噪。

Reference:

[1] FID score for PyTorch (https://github.com/mseitzer/pytorch-fid)