DLCV hw2 電機碩一 R11921100 溫威領

Problem 1: GAN

1. Please print the model architecture of method A and B.

Generator architecture of Model A:

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Discriminator architecture of Model A:

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Implementation details of Model A:

For Generator

Input size: 100x1x1 / batch size: 256 / epochs: 600

Optimizer: Adam / learning rate: 1e-4 / Loss function: BCELoss

Mainly used ConvTranspose2d to generate images

For Discriminator

Input size: 3x64x64 / batch size: 256 / epochs: 600

Optimizer: Adam / learning rate: 1e-4 / Loss function: BCELoss

Generator architecture of Model B:

```
Generator(
  (main): Sequential(
    (0): Upsample(scale_factor=4.0, mode=nearest)
    (1): Conv2d(100, 256, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): ReLU(inplace=True)
    (4): Dropout(p=0.2, inplace=False)
    (5): PixelShuffle(upscale_factor=2)
    (6): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): Dropout(p=0.2, inplace=False)
    (10): PixelShuffle(upscale_factor=2)
    (11): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (12): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): Dropout(p=0.2, inplace=False)
    (15): PixelShuffle(upscale_factor=2)
    (16): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (17): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): ReLU(inplace=True)
    (19): Dropout(p=0.2, inplace=False)
    (20): PixelShuffle(upscale_factor=2)
    (21): Conv2d(32, 3, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (22): Tanh()
  )
)
```

Discriminator architecture of Model B:

```
Discriminator(
  (main): Sequential(
    (0): ParametrizedConv2d(
      3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
      (parametrizations): ModuleDict(
        (weight): ParametrizationList(
          (0): _SpectralNorm()
        )
      )
    )
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): ParametrizedConv2d(
      64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
      (parametrizations): ModuleDict(
        (weight): ParametrizationList(
          (0): _SpectralNorm()
        )
      )
    )
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): ParametrizedConv2d(
      128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
      (parametrizations): ModuleDict(
        (weight): ParametrizationList(
          (0): _SpectralNorm()
        )
      )
    )
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): ParametrizedConv2d(
      256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
      (parametrizations): ModuleDict(
        (weight): ParametrizationList(
          (0): _SpectralNorm()
        )
      )
    )
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
```

```
        (11): ParametrizedConv2d(
            512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False
            (parametrizations): ModuleDict(
                (weight): ParametrizationList(
                    (0): _SpectralNorm()
                )
            )
        )
        (12): Sigmoid()
    )
)
```

Implementation details of Model B:

Used soft and noisy labels (0~0.33, 0.7~1.03)

For Generator

Input size: 100x1x1 / batch size: 256 / epochs: 300

Optimizer: Adam / learning rate: 1e-4 / Loss function: BCELoss

Add dropout and mainly used Conv2d and pixelsuffle to generate images

For Discriminator

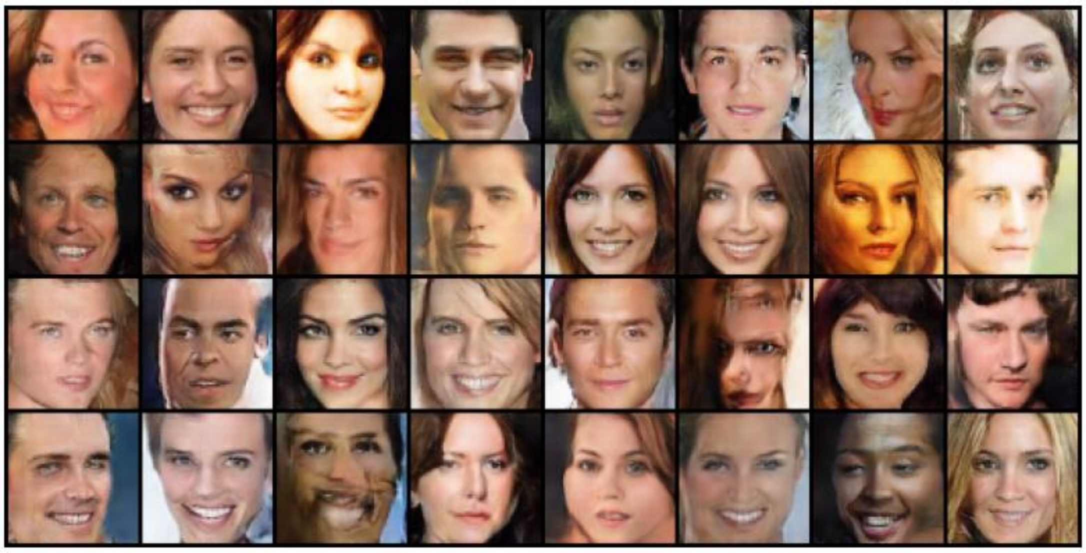
Input size: 3x64x64 / batch size: 256 / epochs: 600

Optimizer: SGD / learning rate: 1e-3 / momentum: 0.5 / Loss function: BCELoss

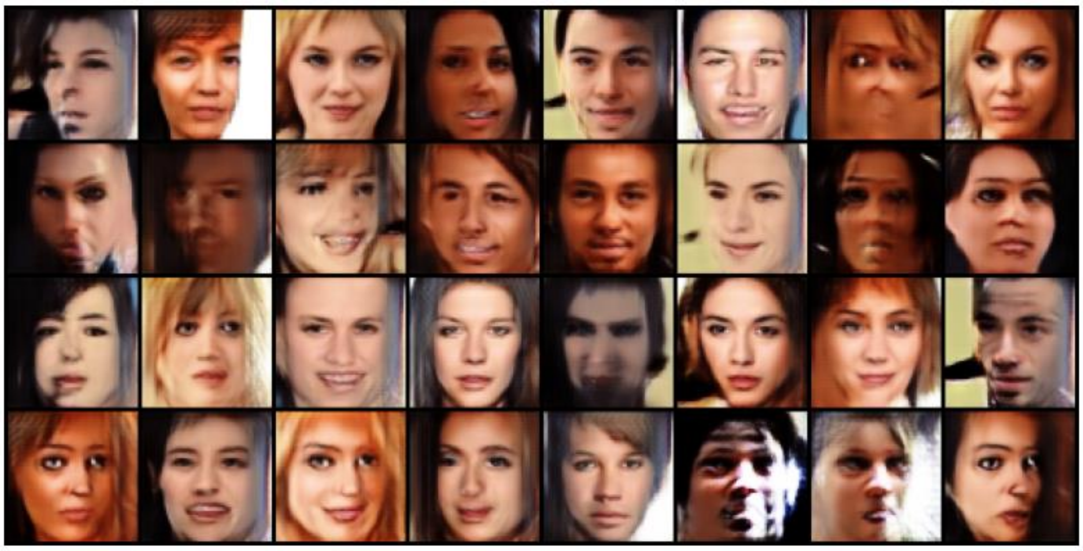
Used SpectralNorm in discriminator

2. Please show the first 32 generated images of both method A and B then discuss the difference between method A and B.

Generated by Model A

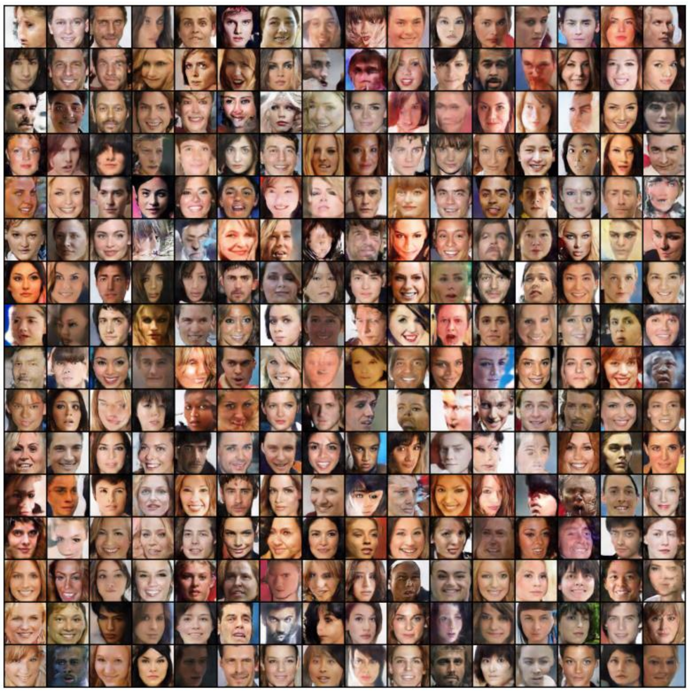


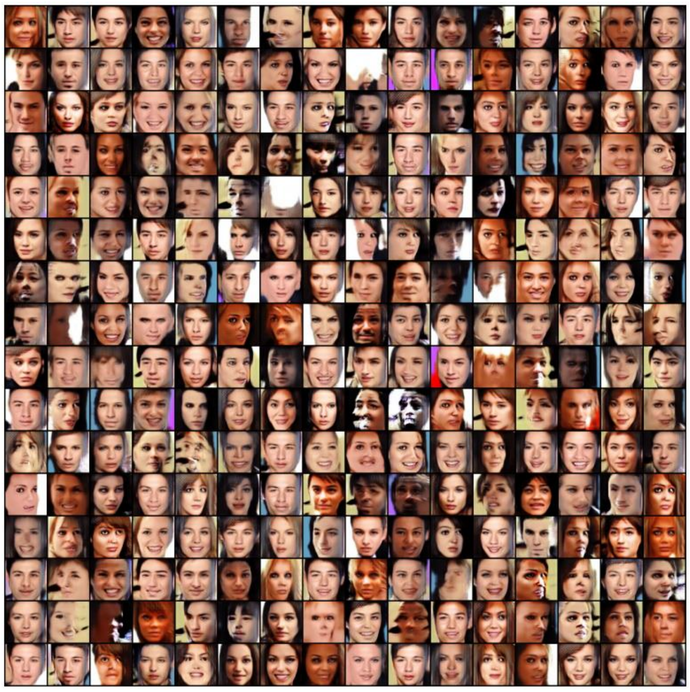
Generated by Model B



3. Please discuss what you've observed and learned from implementing GAN.

在實作 DCGAN 時，model A 採用 pytorch 官方 DCGAN 教學，主要使用 ConvTranspose2d 來產生圖片，效果不錯，訓練越久效果越好。而 model B 主要使用 Conv2d 與 pixelshuffle 加上作業簡報所提供的 tips，如 soft and noisy labels...等，原先非常容易產生 mode collapse，產生出幾乎一樣的臉，雖然 face_recog 都 100%，但 FID 慘不忍睹，之後加入在 discriminator 中加入 SpectralNorm 以及把 discriminator 的 optimizer 改成 SGD 後有稍微好轉，但產生的臉仍有些一致，如下圖所示，左圖為 model A 產生的圖片，右圖為 model B 產生的圖片，可以看出 model A 和 B 的結果有些微差異。





Experiment results: model A, face_recog: 91.3%, fid: 23.72 / model B, face_recog: 94.60%, fid: 35.30

code reference: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Problem 2: Diffusion models

1. Please show your model architecture and describe your implementation details.

Diffusion Model architecture:

```
DDPM(
  (nn_model): ContextUnet(
    (init_conv): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
    (down1): UnetDown(
      (model): Sequential(
        (0): ResidualConvBlock(
          (conv1): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
          (conv2): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
        )
        (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
    )
    (down2): UnetDown(
      (model): Sequential(
        (0): ResidualConvBlock(
          (conv1): Sequential(
            (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
        )
        (conv2): Sequential(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): GELU(approximate=none)
        )
      )
      (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
  )
  (to_vec): Sequential(
    (0): AvgPool2d(kernel_size=7, stride=7, padding=0)
    (1): GELU(approximate=none)
  )
  (timeembed1): EmbedFC(
    (model): Sequential(
      (0): Linear(in_features=1, out_features=256, bias=True)
      (1): GELU(approximate=none)
      (2): Linear(in_features=256, out_features=256, bias=True)
    )
  )
  (timeembed2): EmbedFC(
    (model): Sequential(
      (0): Linear(in_features=1, out_features=128, bias=True)
      (1): GELU(approximate=none)
      (2): Linear(in_features=128, out_features=128, bias=True)
    )
  )
  (contextembed1): EmbedFC(
    (model): Sequential(
      (0): Linear(in_features=10, out_features=256, bias=True)
      (1): GELU(approximate=none)
      (2): Linear(in_features=256, out_features=256, bias=True)
    )
  )
  (contextembed2): EmbedFC(
    (model): Sequential(
      (0): Linear(in_features=10, out_features=128, bias=True)
      (1): GELU(approximate=none)
      (2): Linear(in_features=128, out_features=128, bias=True)
    )
  )
```

```
    (up0): Sequential(
      (0): ConvTranspose2d(256, 256, kernel_size=(7, 7), stride=(7, 7))
      (1): GroupNorm(8, 256, eps=1e-05, affine=True)
      (2): ReLU()
    )
    (up1): UnetUp(
      (model): Sequential(
        (0): ConvTranspose2d(512, 128, kernel_size=(2, 2), stride=(2, 2))
        (1): ResidualConvBlock(
          (conv1): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
          (conv2): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
        )
        (2): ResidualConvBlock(
          (conv1): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
          (conv2): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
        )
      )
    )
    (up2): UnetUp(
      (model): Sequential(
        (0): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
        (1): ResidualConvBlock(
          (conv1): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
          (conv2): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
        )
        (2): ResidualConvBlock(
          (conv1): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
          (conv2): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): GELU(approximate=none)
          )
        )
      )
    )
    (out): Sequential(
      (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): GroupNorm(8, 128, eps=1e-05, affine=True)
      (2): ReLU()
      (3): Conv2d(128, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
  )
)
```

Implementation details:

Input image size: 3x28x28 / batch size: 128 / epochs: 300

Optimizer: Adam / learning rate: 1e-4 / Loss function: MSELoss

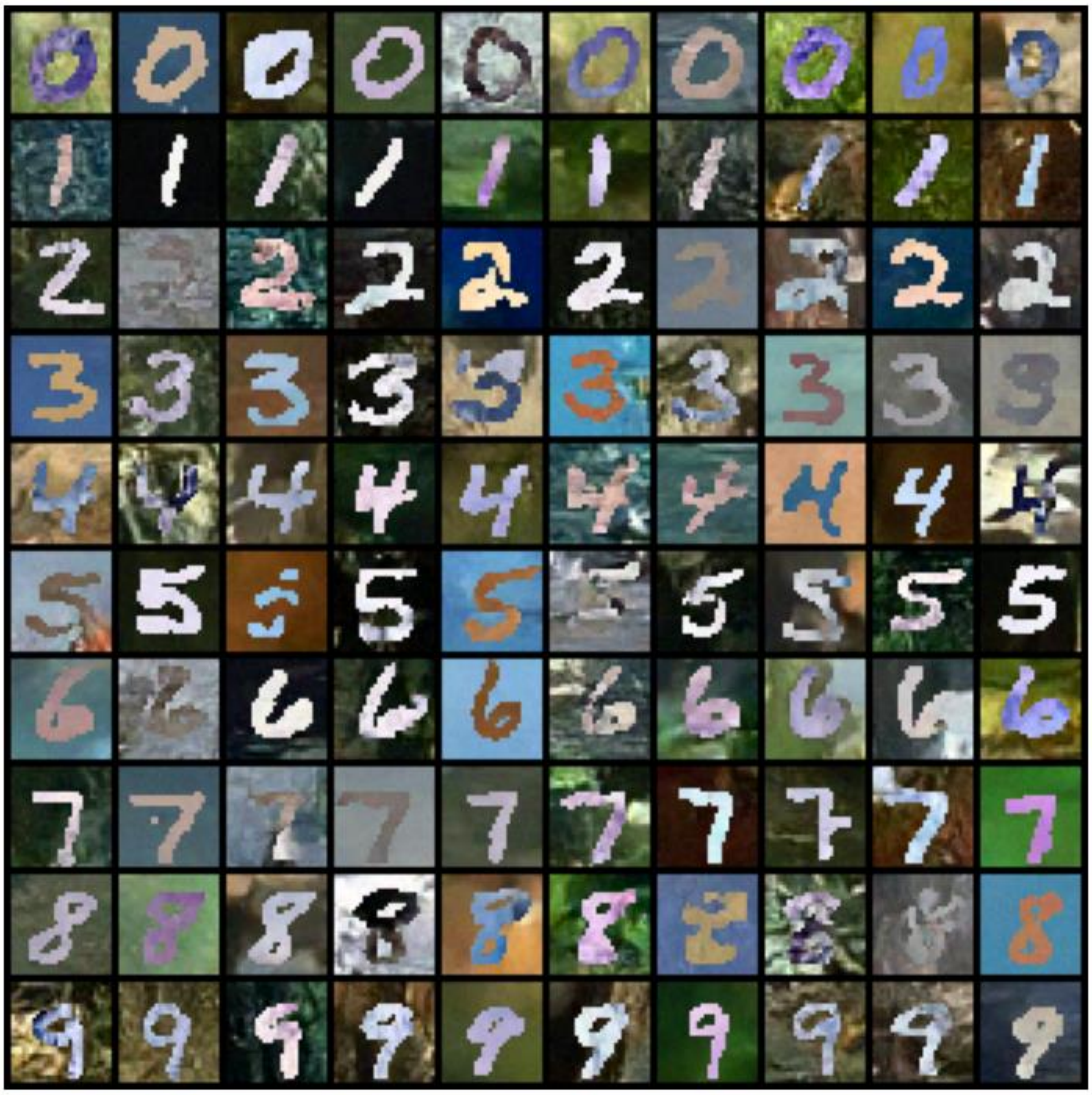
Max time step: 500 / betas: (1e-4, 0.02) / sample guide w: 1

follow the guidance sampling scheme described in *'Classifier-Free Diffusion Guidance'*

'Classifier-Free Diffusion Guidance', paper link: https://arxiv.org/pdf/2207.12598.pdf

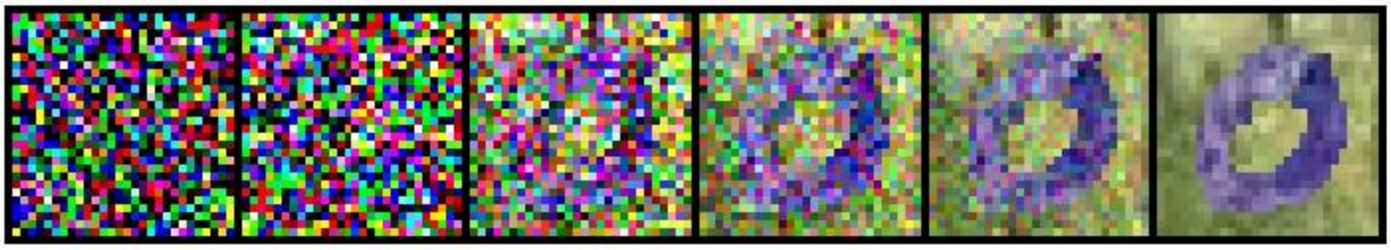
code reference: https://github.com/TeaPearce/Conditional_Diffusion_MNIST

2. Please show 10 generated images for each digit (0-9) in your report. You can put all 100 outputs in one image with columns indicating different noise inputs and rows indicating different digits.



3. Visualize total six images in the reverse process of the first “0” in your grid in (2) with different time steps.

T = 500, 300, 100, 60, 40, 0



4. Please discuss what you’ve observed and learned from implementing conditional diffusion model.

T_max=1000 比 T_max=500 效果好但訓練和產生圖片的時間較久，此外 model 較大效果也較好。

使用*'Classifier-Free Diffusion Guidance'*中的 w 值越大，產生的圖片越標準，但多樣性會降低。

Experiment results: digit_classifier acc 99.6%

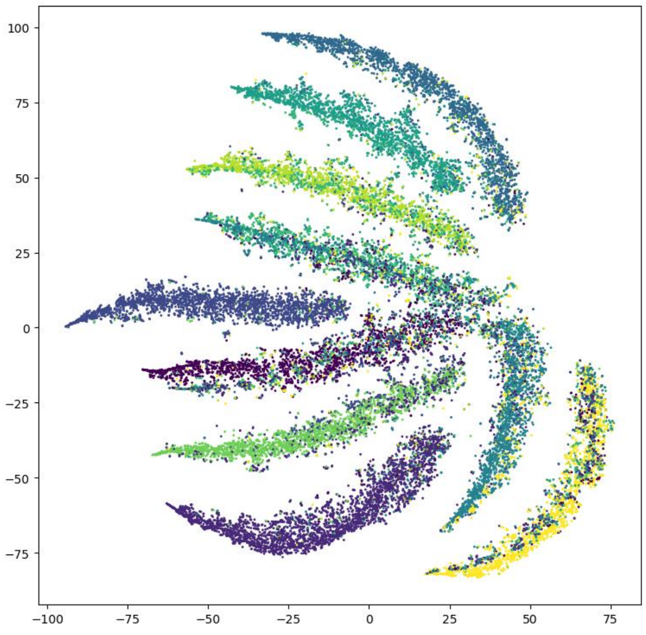
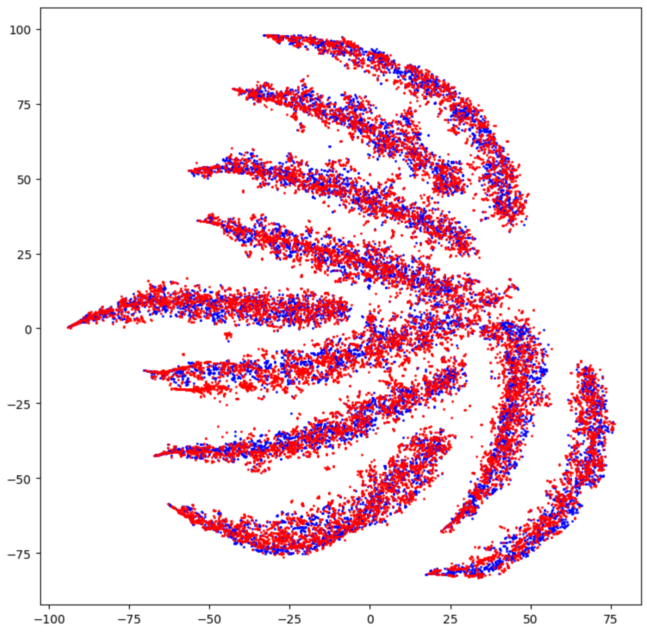
Problem 3: DANN

1. Please create and fill the table with the following format in your report:

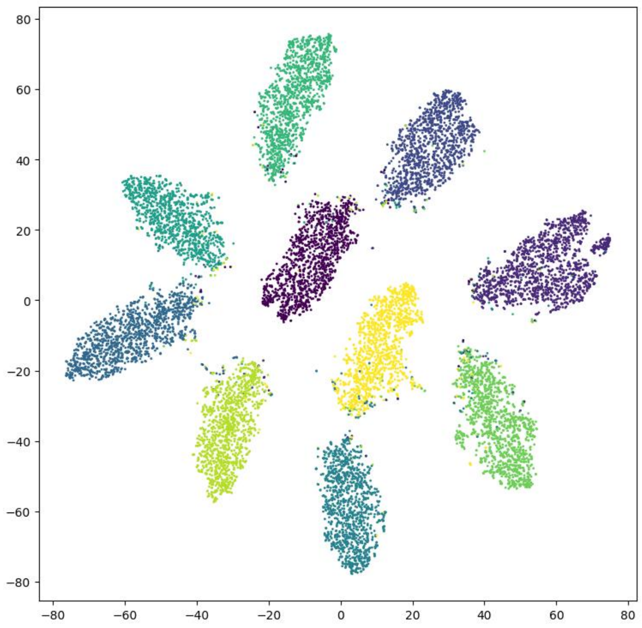
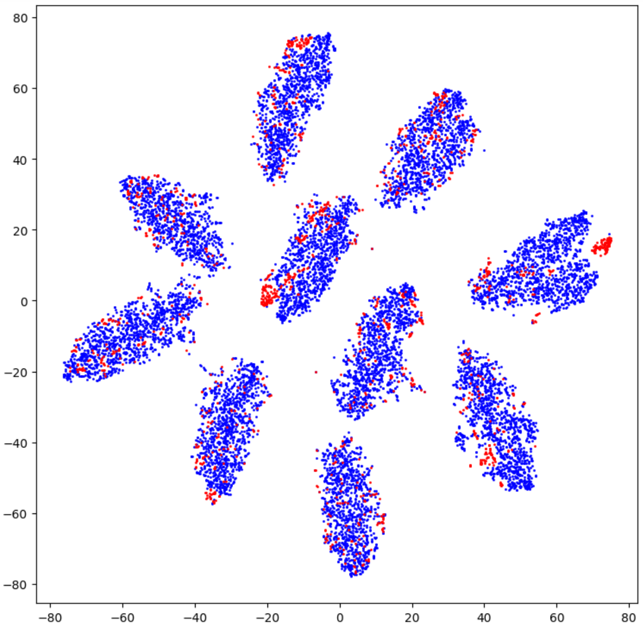
	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	28.80%	77.35%
Adaptation (DANN)	**56.16%**	**93.88%**
Trained on target	92.14%	98.79%

2. Please visualize the latent space of DANN by mapping the validation images to 2D space with t-SNE. For each scenario, you need to plot two figures which are colored by digit class (0-9) and by domain, respectively.

MNIST-M → SVHN

MNIST-M → USPS

3. Please describe the implementation details of your model and discuss what you've observed and learned from implementing DANN.

DANN, Feature Extractor model architecture:

```
FeatureExtractor(
  (conv): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU()
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): ReLU()
    (19): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
)
```

DANN, Label Predictor model architecture:

```
LabelPredictor(
  (layer): Sequential(
    (0): Linear(in_features=512, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

DANN, Domain Classifier model architecture:

```
DomainClassifier(
  (layer): Sequential(
    (0): Linear(in_features=512, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
        (2): ReLU()
        (3): Linear(in_features=512, out_features=512, bias=True)
        (4): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU()
        (6): Linear(in_features=512, out_features=512, bias=True)
        (7): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (8): ReLU()
        (9): Linear(in_features=512, out_features=512, bias=True)
        (10): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (11): ReLU()
        (12): Linear(in_features=512, out_features=1, bias=True)
    )
)
```

Implementation details:

For each model

Input image size: 3x32x32 / batch size: 256 / epochs : 200 / lamb: 0.3

For Feature Extractor & Label Predictor

Optimizer: Adam / learning rate: 1e-3 / Loss function: CrossEntropyLoss

For Domain Classifier

Optimizer: Adam / learning rate: 1e-3 / Loss function: BCEWithLogitsLoss

Observed:

使用只在 source train dataset (mnistm)上訓練的 classifier 來預測 target validation dataset (svhn、usps)的正確率皆是最低的，即 lower bound，儘管都是數字資料集，但 source 和 target 兩者 domain 不盡相同，所以無法完整正確分類。直接使用 target train dataset 訓練來預測 target validation dataset 正確率最高，即 upper bound，因 train 和 validation 兩者 domain 相同，所以幾乎能分類正確。而 DANN 雖然只在 source train dataset (mnistm)上訓練，但多加了 domain classifier 來擬和分類時 source 和 target 的 latent space domain，讓 target dataset 在分類時也能盡量像 source dataset 一樣被正確分類，雖然正確率無法像 upper bound 相同，但在 target dataset 沒有 label 的情況下，DANN 是提升分類正確率的一種方法。

code reference: https://github.com/TeaPearce/Conditional_Diffusion_MNIST