NTU DLCV (Autumn, 2022) HW2 Report

R10921069 沈郁鈞

Problem 1

1

METHOD A (DCGAN)

Generator

```
Generator(
  (l1): Sequential(
    (0): Linear(in_features=100, out_features=8192, bias=False)
    (1): BatchNorm1d(8192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (l2_5): Sequential(
    (0): Sequential(
      (0): ConvTranspose2d(512, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), outpu
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), outpu
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (3): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_pac
    (4): Tanh()
  )
)
```

Discriminatior

```
Discriminator(
  (ls): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (3): Sequential(
      (0): Conv2d(128, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (4): Sequential(
      (0): Conv2d(256, 512, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
    (6): Sigmoid()
  )
)
```

METHOD B (SNGAN)

Generator

```
Generator(
  (l1): Sequential(
    (0): Linear(in_features=100, out_features=8192, bias=False)
    (1): BatchNorm1d(8192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (l2_5): Sequential(
    (0): Sequential(
      (0): ConvTranspose2d(512, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), outpu
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), outpu
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (3): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_pa
    (4): Tanh()
  )
)
```

Discriminatior

```
Discriminator(
  (ls): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): LeakyReLU(negative_slope=0.2)
    )
    (3): Sequential(
      (0): Conv2d(128, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): LeakyReLU(negative_slope=0.2)
    )
    (4): Sequential(
      (0): Conv2d(256, 512, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): LeakyReLU(negative_slope=0.2)
    )
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
  )
)
```

2

METHOD A (DCGAN)

Method B (SNGAN)

Difference of Implementation details between method A & B

- Perform Spectral Normalization on each weight layer on the **Discriminator** of DCGAN.
- Remove **all** Batch Normalization layers on the **Discriminator** of DCGAN.
- Remove the `Sigmoid()` activation on the **Discriminator** of DCGAN.
- Change loss function from `BCELoss()` to `BCEWithLogitsLoss()` for training SNGAN.

3

Difference of sampled images between Method A & B

- Images with complete and clear face contour are more on method B (SNGAN) than on method A (DCGAN).
- Images with facial features like eyes, nose and lips are more recognizable on method B (SNGAN) than method A (DCGAN).

Experiment results of 2 methods

Methods \ Evaluation Metrics	FID	Face Recognition Accuracy
Method A (DCGAN)	38.68	91.4% (914/1000)
Method B (SNGAN)	25.66	92.2% (922/1000)

DIFFICULTIES

- Performance of GAN is really unstable during training. For example, when training DCGAN, the FID value could suddenly jump up from about 40 to more than 100.
- Training each GAN could take several days. However, we are not sure whether we could luckily pick the most suitable GAN (taking less GPU memory and have better performance). Thus, it may take 1~2 weeks to find for the best solutions.

Problem 2

1

```
UNet_conditional(
  (inc): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): GroupNorm(1, 64, eps=1e-05, affine=True)
      (2): GELU(approximate=none)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): GroupNorm(1, 64, eps=1e-05, affine=True)
    )
  )
  (down1): Down(
    (maxpool_conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 64, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 64, eps=1e-05, affine=True)
        )
      )
      (2): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
    )
    (emb_layer): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=128, bias=True)
    )
  )
  (sa1): SelfAttention(
    (mha): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=128, out_features=128, bias=Tr
    )
    (ln): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
    (ff_self): Sequential(
      (0): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=128, out_features=128, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=128, out_features=128, bias=True)
    )
  )
  (down2): Down(
    (maxpool_conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
      (2): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (1): GroupNorm(1, 256, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (4): GroupNorm(1, 256, eps=1e-05, affine=True)
        )
      )
    )
    (emb_layer): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=256, bias=True)
    )
  )
  (sa2): SelfAttention(
    (mha): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=Tr
```

```
      )
      (ln): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (ff_self): Sequential(
        (0): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
        (1): Linear(in_features=256, out_features=256, bias=True)
        (2): GELU(approximate=none)
        (3): Linear(in_features=256, out_features=256, bias=True)
      )
    )
    (down3): Down(
      (maxpool_conv): Sequential(
        (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (1): DoubleConv(
          (double_conv): Sequential(
            (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
            (1): GroupNorm(1, 256, eps=1e-05, affine=True)
            (2): GELU(approximate=none)
            (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
            (4): GroupNorm(1, 256, eps=1e-05, affine=True)
          )
        )
        (2): DoubleConv(
          (double_conv): Sequential(
            (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
            (1): GroupNorm(1, 256, eps=1e-05, affine=True)
            (2): GELU(approximate=none)
            (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
            (4): GroupNorm(1, 256, eps=1e-05, affine=True)
          )
        )
      )
      (emb_layer): Sequential(
        (0): SiLU()
        (1): Linear(in_features=256, out_features=256, bias=True)
      )
    )
    (sa3): SelfAttention(
      (mha): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=Tr
      )
      (ln): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (ff_self): Sequential(
        (0): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
        (1): Linear(in_features=256, out_features=256, bias=True)
        (2): GELU(approximate=none)
        (3): Linear(in_features=256, out_features=256, bias=True)
      )
    )
    (bot1): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(1, 256, eps=1e-05, affine=True)
        (2): GELU(approximate=none)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): GroupNorm(1, 256, eps=1e-05, affine=True)
      )
    )
    (bot3): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(1, 256, eps=1e-05, affine=True)
        (2): GELU(approximate=none)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): GroupNorm(1, 256, eps=1e-05, affine=True)
      )
    )
    (up1): Up(
      (up): Upsample(scale_factor=2.0, mode=bilinear)
      (conv): Sequential(
        (0): DoubleConv(
          (double_conv): Sequential(
            (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
            (1): GroupNorm(1, 512, eps=1e-05, affine=True)
            (2): GELU(approximate=none)
            (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
            (4): GroupNorm(1, 512, eps=1e-05, affine=True)
          )
        )
        (1): DoubleConv(
          (double_conv): Sequential(
```

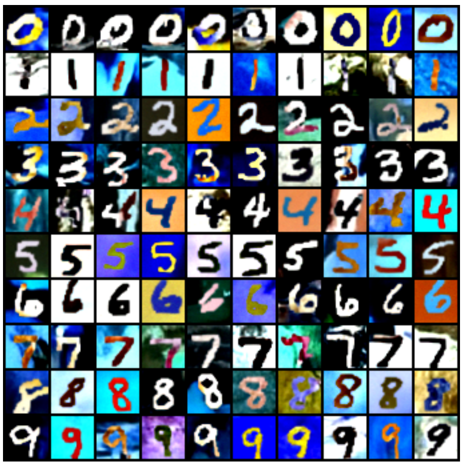
```
          (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (1): GroupNorm(1, 256, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
    )
    (emb_layer): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=128, bias=True)
    )
  )
  (sa4): SelfAttention(
    (mha): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=128, out_features=128, bias=Tr
    )
    (ln): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
    (ff_self): Sequential(
      (0): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=128, out_features=128, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=128, out_features=128, bias=True)
    )
  )
  (up2): Up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (1): GroupNorm(1, 256, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (4): GroupNorm(1, 256, eps=1e-05, affine=True)
        )
      )
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
          (4): GroupNorm(1, 64, eps=1e-05, affine=True)
        )
      )
    )
    (emb_layer): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=64, bias=True)
    )
  )
  (sa5): SelfAttention(
    (mha): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=64, out_features=64, bias=True
    )
    (ln): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
    (ff_self): Sequential(
      (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=64, out_features=64, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=64, out_features=64, bias=True)
    )
  )
  (up3): Up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
          (1): GroupNorm(1, 64, eps=1e-05, affine=True)
```

```
          (1): GroupNorm(1, 64, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 64, eps=1e-05, affine=True)
        )
      )
    )
    (emb_layer): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=64, bias=True)
    )
  )
  (sa6): SelfAttention(
    (mha): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=64, out_features=64, bias=True
    )
    (ln): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
    (ff_self): Sequential(
      (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=64, out_features=64, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=64, out_features=64, bias=True)
    )
  )
  (outc): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
  (label_emb): Embedding(10, 256)
)
```

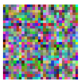
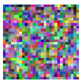
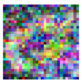
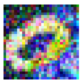
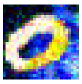
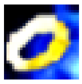
Implementation Details

- I use U-Net as for the main model architecture.
  - New layers & activation functions for U-Net:
    - Layer Normalization
    - Group Normalization
    - SiLU (Sigmoid Linear Unit)
    - GELU (Gaussian Error Linear Unit)
  - Techniques:
    - Self-Attention (Multi head attention)

2



3

t=0	t=120	t=240	t=360	t=480	t=600
					

4

THOUGHTS

- The example step provided in slide is 1000. However, taking 600 steps is sufficient for image quality performance.
- The image generation method is from random noisy images to denoise step by step.
- It's not time consuming rather than training a general GAN (Problem 1).

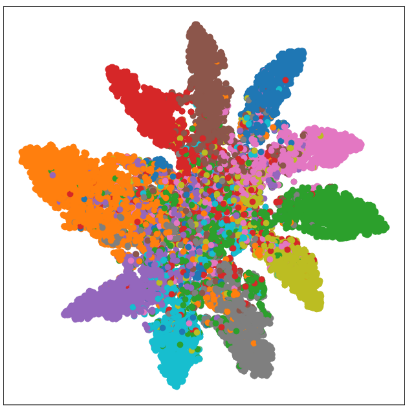
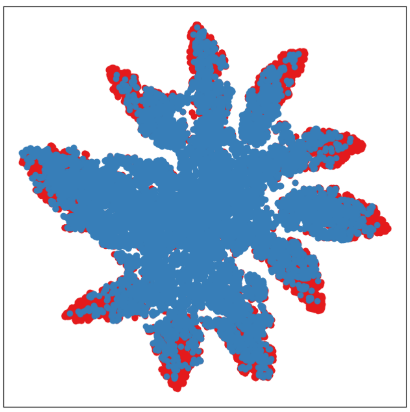
Problem 3

1

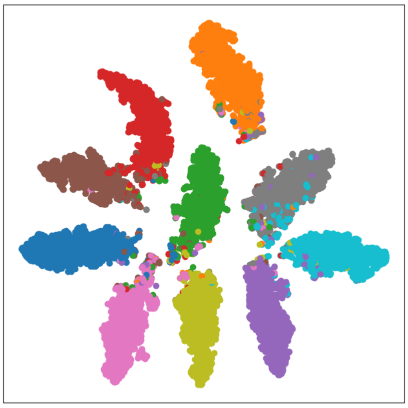
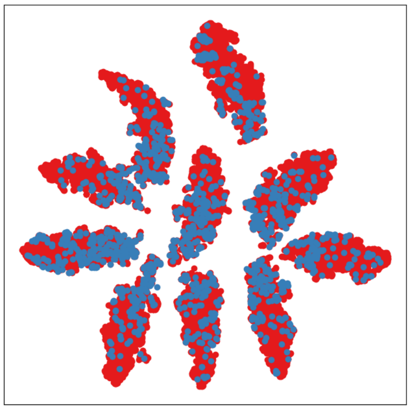
	MNIST-M → SVHN	MNIST-M → USPS
Trained on Source	19.159%	50.645%
Adapation (DANN)	64.506%	86.425%
Trained on TARGET	96.038%	98.658%

2

MNIST-M → SVHN

By Class	By Domain
	

MNIST-M → USPS

By Class	By Domain
	

3

Implementation Details

- For normal tasks (directly training on source / target domain), I use ResNet34 in torchvision for the main architecture.
- For DANN task, I use a VGG-like architecture for feature extractor, and MLP layers for Label Predictor and Domain Classifier. (Please refer to `p3_test.py` for detailed architecture)

Observation

- Scenario **MNIST-M → USPS** has better performance than scenario **MNIST-M → SVHN** on all scenarios.
  - I guess the main reason is both MNIST-M and USPS are hand-written digits dataset, while SVHN is printed digit dataset.
  - Another guess is that USPS dataset is much more smaller than other two datasets, so it may not cover wider hand-written digits conditions.