DLCV HW2 Report

資工所 呂兆凱 R11922098

GAN

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

method A : (DCGAN)

Discriminator

```
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()
  )
)
```

Generator

```
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()
  )
)
```

method B : (SAGAN)

Discriminator

```
Discriminator(
(l4): Sequential(
  (0): SpectralNorm(
    (module): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
  (1): LeakyReLU(negative_slope=0.1)
)
(l1): Sequential(
  (0): SpectralNorm(
    (module): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
  (1): LeakyReLU(negative_slope=0.1)
)
(l2): Sequential(
  (0): SpectralNorm(
    (module): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
  (1): LeakyReLU(negative_slope=0.1)
)
(l3): Sequential(
  (0): SpectralNorm(
    (module): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
  (1): LeakyReLU(negative_slope=0.1)
)
(last): Sequential(
  (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
)
(attn1): Self_Attn(
  (query_conv): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
  (key_conv): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
  (value_conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
  (softmax): Softmax(dim=-1)
)
(attn2): Self_Attn(
  (query_conv): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
```

```
    (key_conv): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
)
```

Generator

```
Generator(
  (l4): Sequential(
    (0): SpectralNorm(
      (module): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (l1): Sequential(
    (0): SpectralNorm(
      (module): ConvTranspose2d(128, 512, kernel_size=(4, 4), stride=(1, 1))
    )
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (l2): Sequential(
    (0): SpectralNorm(
      (module): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (l3): Sequential(
    (0): SpectralNorm(
      (module): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (last): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): Tanh()
  )
  (attn1): Self_Attn(
    (query_conv): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
```

```
  (softmax): Softmax(dim=-1)
)
(attn2): Self_Attn(
  (query_conv): Conv2d(64, 8, kernel_size=(1, 1), stride=(1, 1))
  (key_conv): Conv2d(64, 8, kernel_size=(1, 1), stride=(1, 1))
  (value_conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
  (softmax): Softmax(dim=-1)
)
```

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

method A : (DCGAN)

method B : (SAGAN)

The difference between method A and B :

DCGAN是單純利用兩個CNN來分別當作Generator和Discriminator，而SAGAN則又再加入了pixelwise的Self-Attention機制，可以更準確的對全局的圖像做調整，使圖片可以產生得更為細緻。看圖可以發現 SAGAN 所產出的圖片成果更佳，且其產生的背景也更為清楚明亮。

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

GAN 其實不好訓練，因為我們必須讓 Generator 和 Discriminator 的 loss 一起下降，讓他們一起變好，才可以達到最好的效果，若是有一方過於強勢，都可能導致無法訓練起來的情況。而我們可以透過調整learning rate或是加入 weight_decay 的方式來使兩者共同進步以增進其表現。

Diffusion models

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

model architecture

```
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=False)
     (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=True)
  )
  (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=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=False)
        (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=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)
      )
    )
  )
  (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=True)
  )
  (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=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)
      )
    )
    (2): 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)
      )
    )
  )
  (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=True)
  )
  (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, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (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=False)
    (4): GroupNorm(1, 512, eps=1e-05, affine=True)
  )
)
(bot2): DoubleConv(
  (double_conv): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (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=False)
    (4): GroupNorm(1, 512, eps=1e-05, affine=True)
  )
)
(bot3): DoubleConv(
  (double_conv): Sequential(
    (0): Conv2d(512, 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(
  (up1): Upsample(scale_factor=2.0, mode=bilinear)
  (up2): Upsample(scale_factor=2.5, mode=bilinear)
  (conv): Sequential(
    (0): DoubleConv(
```

```
      (double_conv): Sequential(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (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=False)
        (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=False)
        (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=False)
        (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=True)
  )
  (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(
  (up1): Upsample(scale_factor=2.0, mode=bilinear)
  (up2): Upsample(scale_factor=2.5, mode=bilinear)
  (conv): Sequential(
    (0): 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)
      )
    )
    (1): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(256, 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, 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(
  (up1): Upsample(scale_factor=2.0, mode=bilinear)
  (up2): Upsample(scale_factor=2.5, mode=bilinear)
  (conv): Sequential(
    (0): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(128, 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=False)
        (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)
        (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)
```

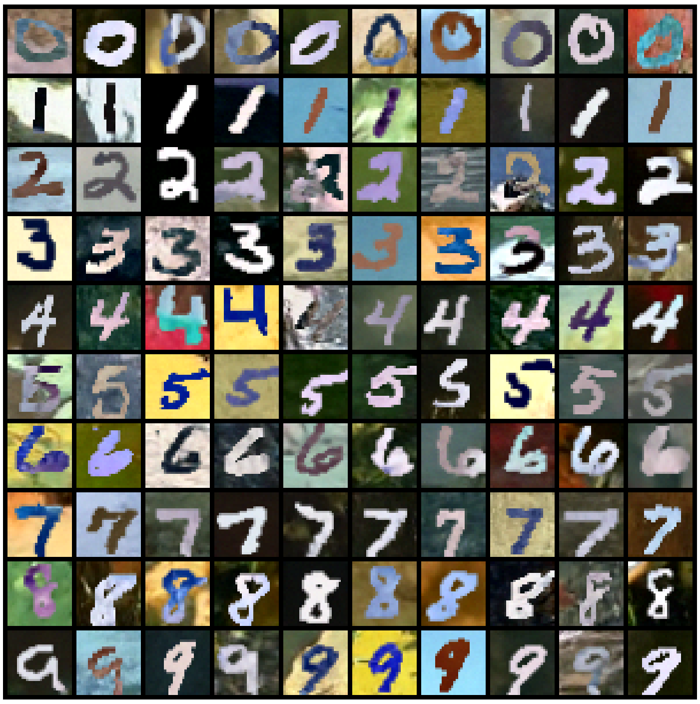
```
      (1): Linear(in_features=64, out_features=64, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=64, out_features=64, bias=True)
    )
  )
  (outc): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
  (label_emb): Embedding(10, 256)
)
```

implementation details

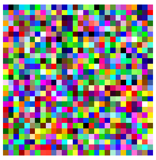
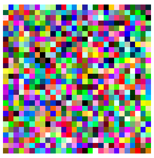
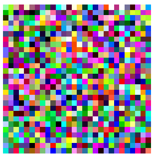
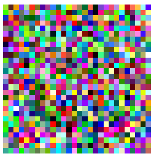
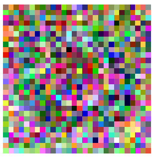
Diffusion 用來對於圖片逐步增加noise，使其漸進變換為一張 noise 圖片；並在訓練過程中使用一個 U-Net 結構的 Autoencoder，來對於在 t 時間的 noise 進行預測並去噪。最後我們就可以使用此U-Net對於隨機亂數去噪產生圖片。

About Parameter : batch_size : 128, learning rate : 3 * 10^{-3}, 使用 AdamW optimizer

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 = 0	t = 200	t = 400	t = 600	t = 800	t = 1000

4. Please discuss what you’ve observed and learned from implementing conditional diffusion model.
認識到除了 GAN 之外的影像生成模型，並且也學到了 diffusion model 的運作原理與實作方式，也覺得的作者想法很有趣，同樣的方法也可以應用在其他領域如 Segmentation、Super Resolution 等。另外也發現雖然 diffusion model 的 sample 速度很慢，但可以生成到品質很好的圖像。

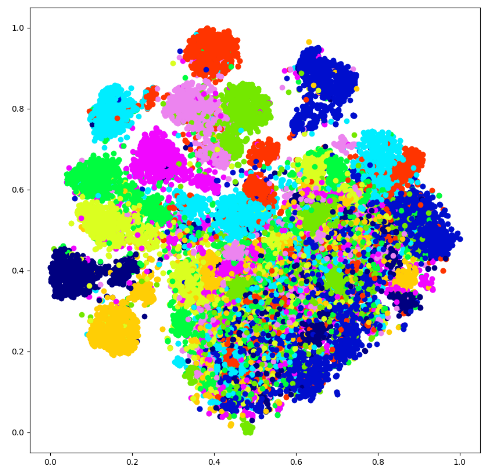
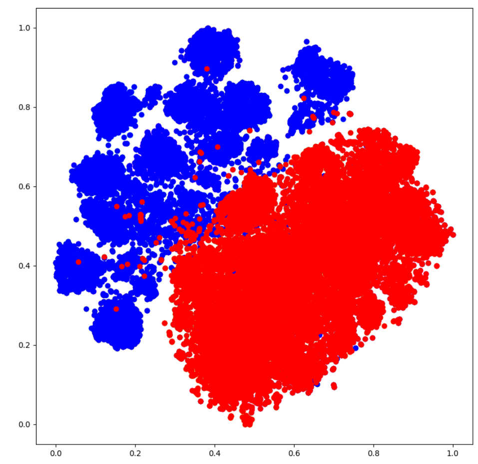
DANN

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

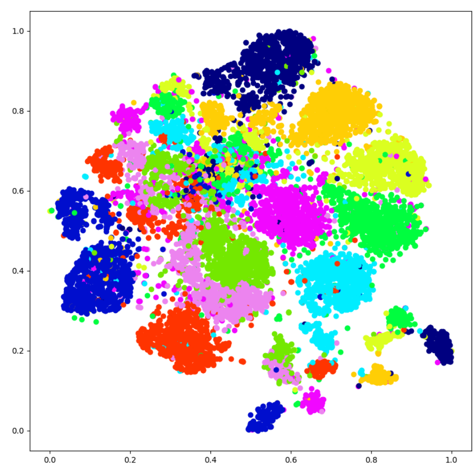
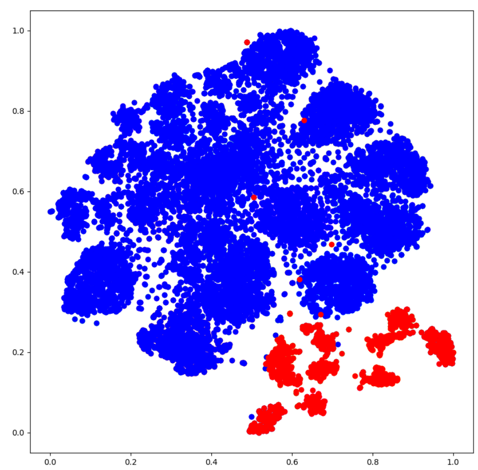
	MNIST-M → SVHN	**MNIST-M → USPS**
Trained on source	39%	80%
Adaptation (DANN)	49%	82%
Trained on target	92%	99%

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

1. MNIST-M → SVHN

By class	By domain
	

2. MNIST-M → USPS

By class	By domain
	

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

DANN model :

```
DANN(
  (feature): Sequential(
    (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 48, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): ReLU(inplace=True)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=768, out_features=384, bias=True)
    (1): BatchNorm1d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=384, out_features=192, bias=True)
    (4): BatchNorm1d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=192, out_features=10, bias=True)
  )
  (discriminator): Sequential(
    (0): Linear(in_features=768, out_features=384, bias=True)
    (1): BatchNorm1d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=384, out_features=2, bias=True)
  )
)
```

the implementation details of your model and discuss what you've observed and learned :

3 components : feature extractor, classifier, discriminator

About Parameter : batch_size : 128, learning rate : 10^{-3}, 使用 Adam optimizer

利用feature extractor取出feature，並將其分別給予classifier和discriminator, 並在discriminator前面加一個GradReverse做梯度反轉。

發現使用 DANN 在 MNIST-M → USPS 的結果上比 MNIST-M → SVHN 的結果來得好很多，因為 USPS 為黑白圖片資料集，而 source domain 為 MNIST-M, MNIST-M 是彩色資料帶有較多資訊，因此可以讓預測在黑白圖片上的效果較好；而若是預測在同樣為彩色資料的SVHN，效果就沒有那麼好。

Reference

- GAN :
  - https://arxiv.org/abs/1511.06434
  - https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
  - https://ithelp.ithome.com.tw/articles/10196257
  - https://github.com/heykeetae/Self-Attention-GAN
  - https://xiaosean.github.io/deep%20learning/computer%20vision/2018-06-15-SAGAN/
- Diffusion models :
  - https://github.com/tcapelle/Diffusion-Models-pytorch?organization=tcapelle&organization=tcapelle
- DANN :
  - https://github.com/fungtion/DANN
  - https://github.com/NaJaeMin92/pytorch_DANN