Hw2_report

陳敬和 R11922166

Problem 1 GAN

1. Print the model architecture of method A and B

- Method A - DCGAN

```
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), output_padding=(1, 1), bias=False)
      (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), output_padding=(1, 1), bias=False)
      (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_padding=(1, 1), bias=False)
      (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_padding=(1, 1))
    (4): Tanh()
  )
)
```

```
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()
  )
)
```

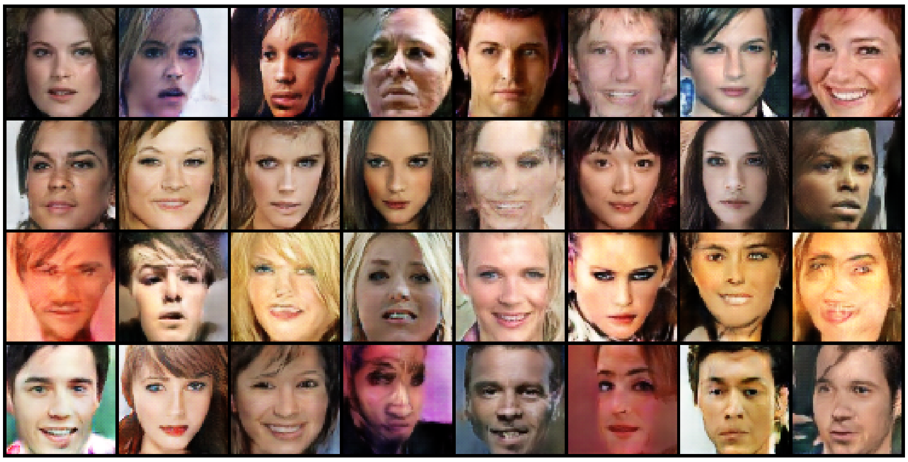
- Method B

```
Generator(
  (l1): Sequential(
    (0): ConvTranspose2d(100, 1024, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (l2_5): Sequential(
    (0): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): ConvTranspose2d(512, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (2): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (3): ConvTranspose2d(128, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
    (4): Tanh()
  )
)
```

```
Discriminator(
  (ls): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): LayerNorm((128, 16, 16), eps=1e-05, elementwise_affine=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): LayerNorm((256, 8, 8), eps=1e-05, elementwise_affine=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): LayerNorm((512, 4, 4), eps=1e-05, elementwise_affine=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
)
```

2. Show the first 32 generated images of both method A and B, and discuss the difference between method A and B

- Method A - DCGAN



- Method B

- Discussion the difference between A and B

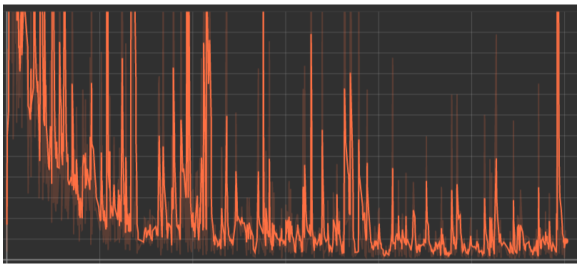
在 face recognition score上，method A 跟 method B的分數大約都在91%上下，也就是說兩個method在生成人臉的準確率十分相似。然而，從FID分數來看兩個method就有些落差，A的FID大約為28，而B的FID大約落在24，表示Method B生成出來的圖片與ground truth較為相似。而從上面圖片中也可以看出，A生成出的人臉圖片有些失真，有扭曲的情況，相較之下B生成的圖片則較為真實。然而，在B中還是有一些失真的圖片出現，可能要透過更換loss function或從model architecture來改善。

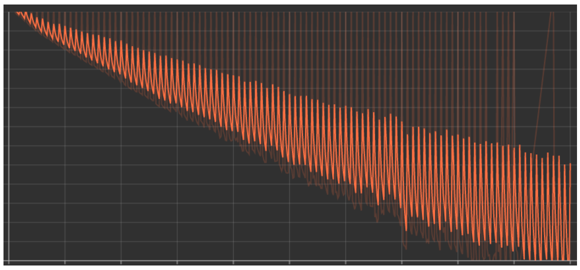
3. Discuss what have observed and learned from implement GAN

原本的DCGAN的效果其實已經很不錯，最佳的結果甚至可以過strong baseline，但是DCGAN在training的過程很不穩定，容易得到很差的結果，loss甚至還可能會直接變nan，也就導致很難train。在此採用SNGAN與WGAN-GP的方式實作，Method B與DCGAN最大的差異就在加入gradient penalty更新discriminator參數，以及在類神經網路每一層都對於其spectral norm 做 normalize，上述兩個改善的方法除了能讓discriminator滿足1-Lipschitz的條件，也能在讓training過程較為穩定。下圖是兩個model在訓練過程中的loss，雖然兩個model loss計算的方式不一樣，但是可以大致看出loss的趨勢，可以觀察到SNGAN在訓練過程中確實比較穩定，在多個epoch後結果也比DCGAN好。

DCGAN discriminator loss

SNGAN discriminator loss





Problem 2 Diffusion Models

1. Print the model architecture and describe your implementation

- 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(
  (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=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(
    (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=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(
    (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=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)
)
```

Epoch	Batch size	Learning rate	optimizer
300	64	3e-4	AdamW

此次模型是使用Unet做model backbone，並且用paper中的training & sampling的algorithm進行實作。在reference的實作中，sampling時會將unconditional noise一起計算，這樣sampling花費的時間過長，遠超過spec所要求的15分鐘，但不計算unconditional noise的效果並不理想。因此最後採折衷方案，在實作時改為有機率不計算unconditional noise，既能減少sampleing的時間又能兼具效果。

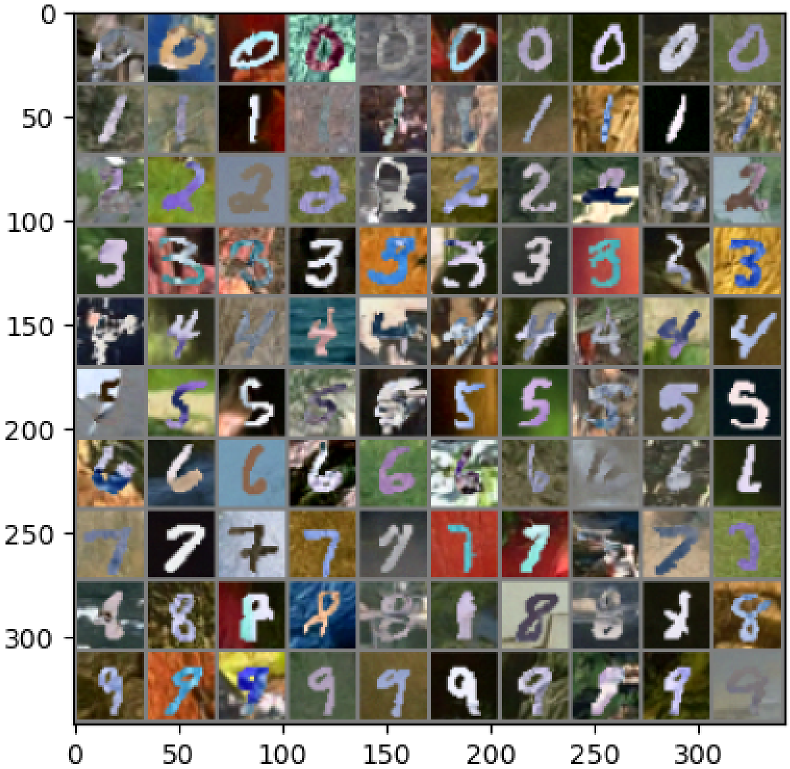
Algorithm 1 Training

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

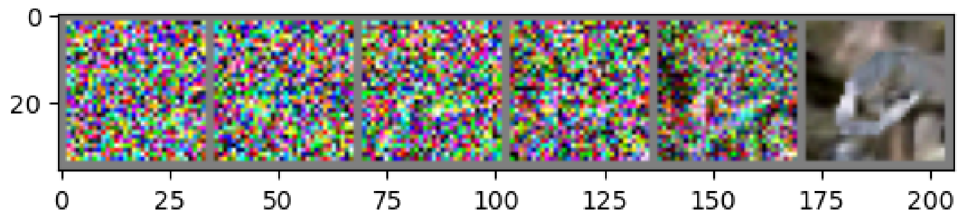
$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t,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

2. Show 10 generated images for each digit



3. Visualize total six images in the reverse of the first “0” with different time steps

From left to right, time step = 0, 200, 400, 600, 800, 1000



4. Discuss what have observed and learned from implement diffusion model

Diffusion model為時下最新的圖像生成工具，最近很火熱的AI作圖都與diffusion model有關。與GAN相比，diffusion model只需要訓練generator，不用訓練discriminator等其他network，loss function簡化許多，大大降低了訓練的難度，在生成圖片上的質量也贏過GAN。而Diffusion model的缺點也很明顯，需要好幾個forward passing做sampling，也導致速度較GAN慢。Diffusion model是近年新興的技術，在查詢資料的過程中有看到許多paper是有關改良sample的研究，還有許多不同領域的應用，相信在未來會有著百家齊放的技術。

Problem 3 DANN

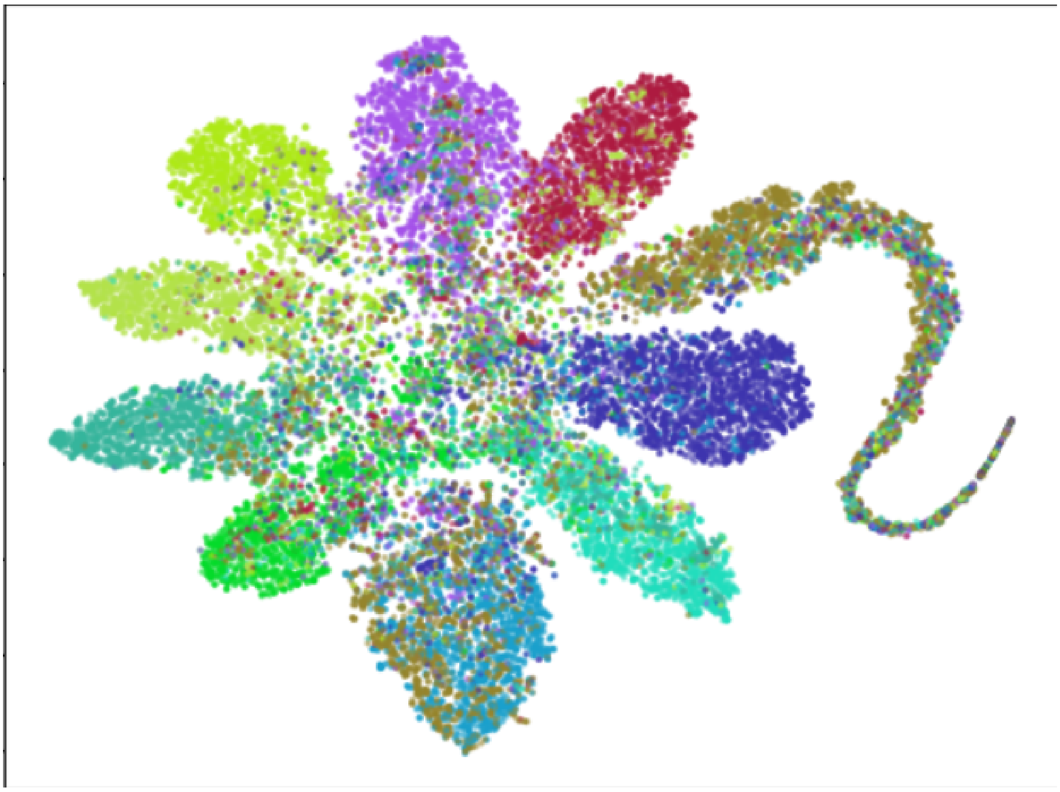
1. Table

	MNIST-M → SVHN	MNISI-M → USPS
Trained on source	33.66%	87.43%
Adaption (DANN)	48.74%	93.68%
Trained on target	90.46%	97.83%

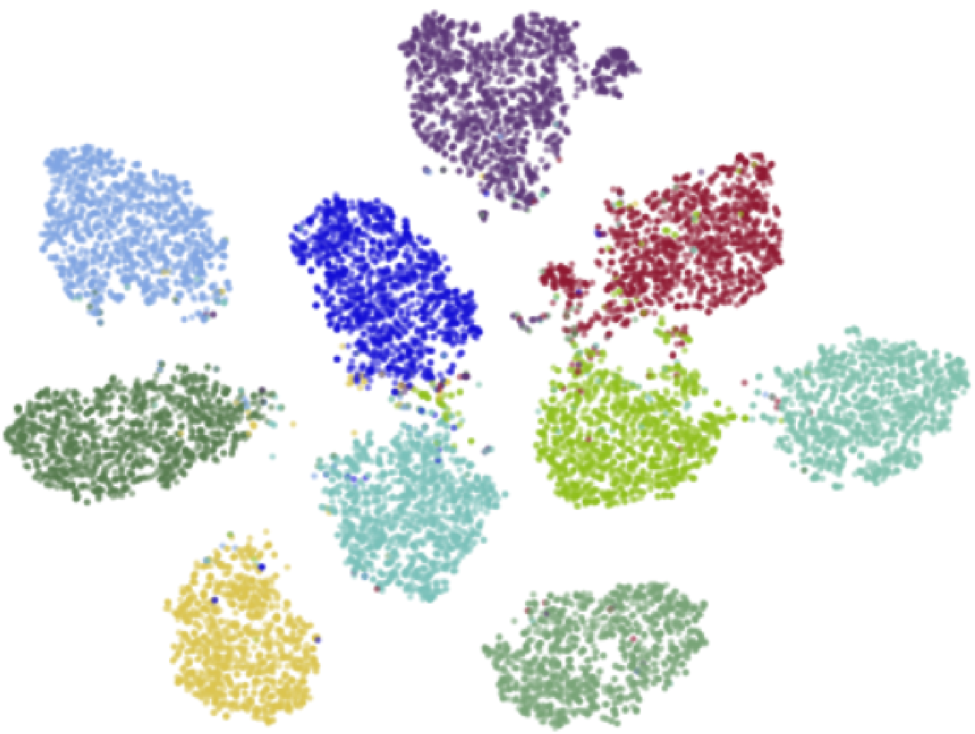
2. Visualize the latent space of DANN

▼ By digit class

- SVHN

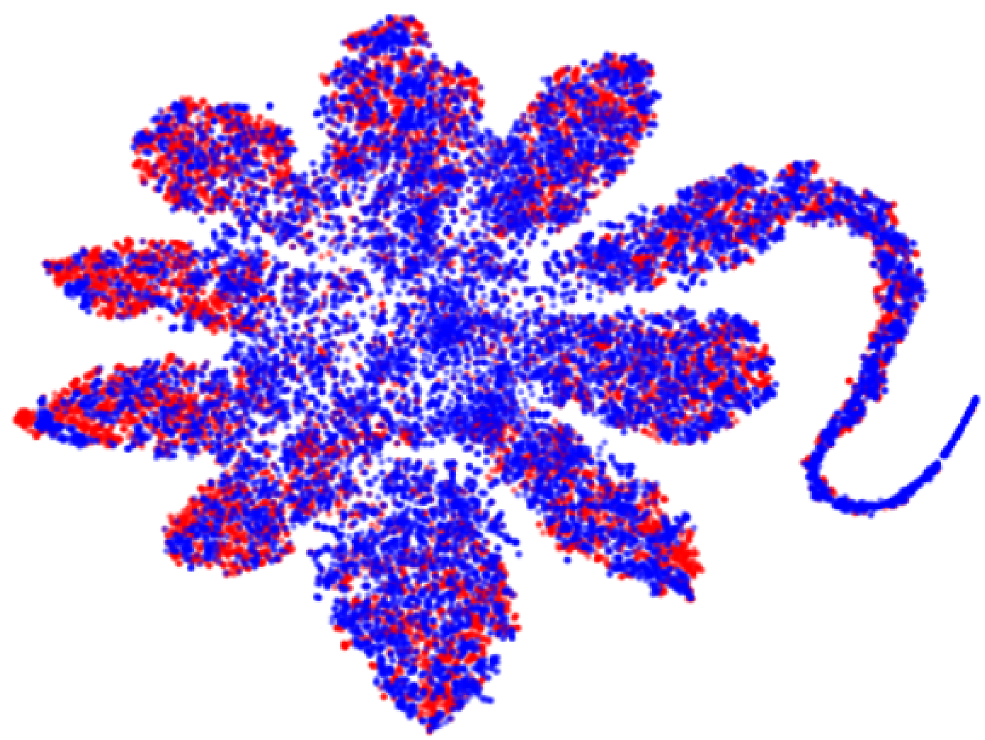


- USPS

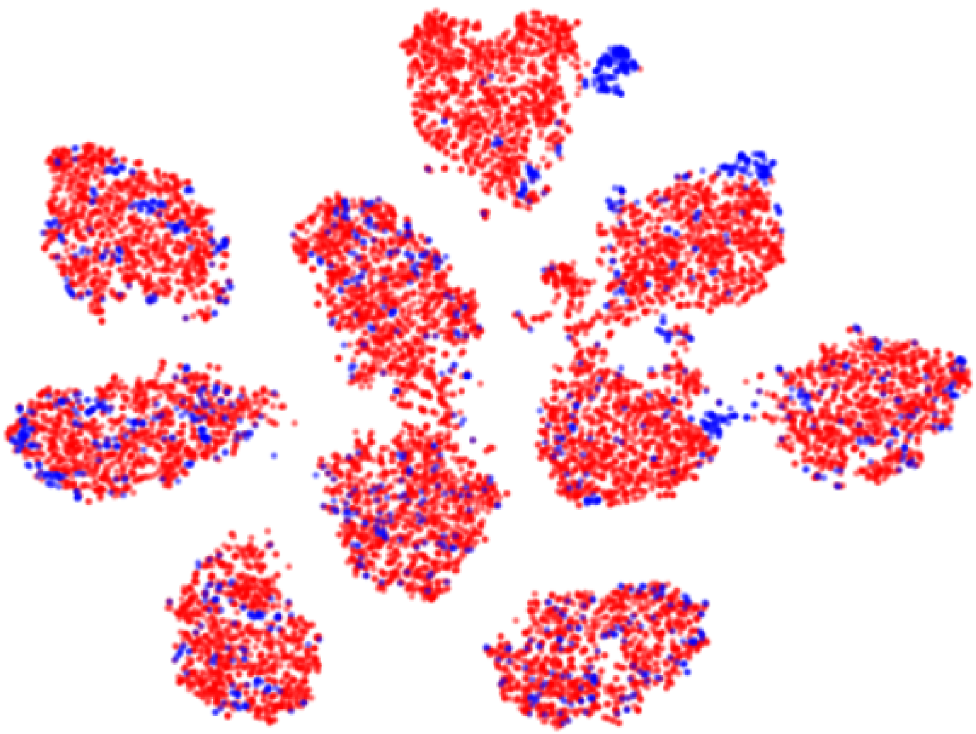


▼ By domain

- SVHN



- USPS



3. Describe the implementation of models and discuss what have observed and learned from implementing DANN

```
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(inplace=True)
    (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(inplace=True)
    (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))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1))
    (15): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
  )
)
```

```
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)
  )
)
```

```
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)
  )
)
```

Epoch	Batch size	Learning rate	optimizer
200	64	3e-4	Adam

在target domain為SVHN的DANN score並不高，但從dataset也可以看出來，SVHN的圖片較模糊，同一張圖片也可能有兩三張digit，效果就沒那麼好；而在USPS的效果就好很多，數字單一且清楚，且與MNIST-M相似，adaption score高也就不意外。

而在reference的model中，在原先feature extractor架構中做了五次的max pooling，在訓練USPS dataset時的成效還不錯，但在訓練SVHN時成效就沒那麼好。如上述所說，USPS的圖片較清楚，因此在做多次pooling沒什麼太大影像；但SVHN就不一樣，就資料特性而言，做太多次的pooling會讓細節丟失，導致feature extractor抓不到feature，訓練成就也就沒那麼好，adaption score沒辦法突破40%。因此在實作時將feature extractor架構稍微調整一下，減少max pooling的次數，於USPS沒什麼大的影響，但在SVHN訓練時就能將adaption score有效提升，adaption score會落在45%以上。

Reference

1. 李弘毅老師機器學習2021 HW6-GAN https://colab.research.google.com/github/ga642381/ML2021-Spring/blob/main/HW06/HW06.ipynb#scrollTo=F0I1jRd6HFmm
2. ：https://github.com/dome272/Diffusion-Models-pytorch
3. 李宏毅老師機器學習2021 HW11-Domain Adaptationhttps://colab.research.google.com/github/ga642381/ML2021-Spring/blob/main/HW11/HW11_ZH.ipynb#scrollTo=3uw2eP09z-pD