

Homework #3

Deep Learning for Computer Vision

Problem 1: VAE (30%)

1. (5%)

```
Encoder()
  (encode): Sequential(
    (0): Conv_Block(
      (block): Sequential(
        (0): Conv2d(3, 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): LeakyReLU(negative_slope=0.2, inplace=True)
      )
    )
    (1): Conv_Block(
      (block): Sequential(
        (0): Conv2d(64, 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): LeakyReLU(negative_slope=0.2, inplace=True)
      )
    )
    (2): Conv_Block(
      (block): Sequential(
        (0): Conv2d(128, 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): LeakyReLU(negative_slope=0.2, inplace=True)
      )
    )
    (3): Conv_Block(
      (block): Sequential(
        (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
      )
    )
  )
```

```

        )
    )
(4): Conv_Block(
    (block): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
)
(5): Conv_Block(
    (block): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
)
)
(meanGen): Linear(in_features=2048, out_features=128, bias=True)
(logvarGen): Linear(in_features=2048, out_features=128, bias=True)
)

```

Decoder(

```

    (preprocess): Sequential(
        (0): Linear(in_features=128, out_features=2048, bias=True)
        (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
    )
    (decode): Sequential(
        (0): Tran_Block(
            (block): Sequential(
                (0): ConvTranspose2d(2048, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
                (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
                (2): ReLU(inplace=True)
            )
        )
        (1): Tran_Block(
            (block): Sequential(
                (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
            )
        )
    )
)
```

```
(1):  BatchNorm2d(512,      eps=1e-05,      momentum=0.1,      affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
(2): Tran_Block(
  (block): Sequential(
    (0): 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(inplace=True)
    )
  )
(3): Tran_Block(
  (block): Sequential(
    (0): 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(inplace=True)
    )
  )
(4): Tran_Block(
  (block): Sequential(
    (0): 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(inplace=True)
    )
  )
(5): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(6): Tanh()
)
)
```

Model implementation details:

在本題中，使用的 Encoder 和 Decoder 採用了幾乎完全對稱的架構。

Encoder 主要利用了 Conv_Block(Conv2d, BatchNorm2d, LeakyReLU) 和 Linear module。

先將原 Image 經過六個 Conv_Block 產生 feature，然後利用兩組 Linear module，分別產生 mean vector 和 logvar vector。

為助理解，架構流程如下圖所示：

```
class Conv_Block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Conv_Block, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2, inplace = True)
        )
    def forward(self, input):
        output = self.block(input)
        return output

class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.encode = nn.Sequential(
            Conv_Block(3, 64, 4, 2, 1),
            Conv_Block(64, 128, 4, 2, 1),
            Conv_Block(128, 256, 4, 2, 1),
            Conv_Block(256, 512, 4, 2, 1),
            Conv_Block(512, 1024, 4, 2, 1),
            Conv_Block(1024, 2048, 4, 2, 1)
        )
        self.meanGen = nn.Linear(2048, 128)
        self.logvarGen = nn.Linear(2048, 128)
    def forward(self, input):
        batch, channel, width, height = input.shape
        feature = self.encode(input)
        feature = feature.view(batch, -1)
        mean, logvar = self.meanGen(feature), self.logvarGen(feature)
        return mean, logvar
```

Decoder 主要利用了 Tran_Block(ConvTranspose2d, BatchNorm2d, ReLU) 和 Linear module(Linear, BatchNorm1d, ReLU)。

先將自 z latent space sample 出來的 noise 經過 Linear module，再經過五個 Tran_Block。最後再經過 ConvTranpose2d 和 Tanh 產生 reconstructed image。

為助理解，架構流程如下圖所示：

```
class Tran_Block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Tran_Block, self).__init__()
        self.block = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True)
        )
    def forward(self, input):
        output = self.block(input)
        return output

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.preprocess = nn.Sequential(
            nn.Linear(128, 2048),
            nn.BatchNorm1d(2048),
            nn.ReLU(True)
        )
        self.decode = nn.Sequential(
            Tran_Block(2048, 1024, 4, 2, 1),
            Tran_Block(1024, 512, 4, 2, 1),
            Tran_Block(512, 256, 4, 2, 1),
            Tran_Block(256, 128, 4, 2, 1),
            Tran_Block(128, 64, 4, 2, 1),
            nn.ConvTranspose2d(64, 3, 4, 2, 1),
            nn.Tanh()
        )
    def forward(self, input):
        feature = self.preprocess(input)
        feature = feature.view(feature.shape[0], -1, 1, 1)
        image = self.decode(feature)
        return image
```

Training implementation details:

本題 VAE 一共 train 了 100 個 epochs, 使用 Adam optimizer, fix learning rate 為 $1e-4$, 並且使用了 RandomHorizontalFlip 的 data augmentation 技巧, batch_size 為 32, 使用 MSE Loss 和 KL Loss。

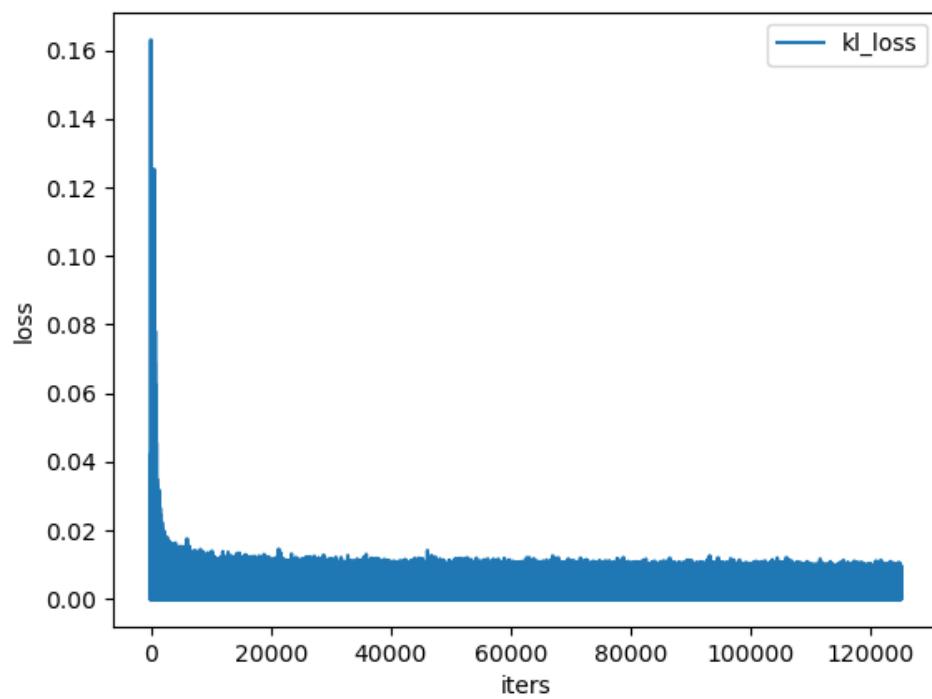
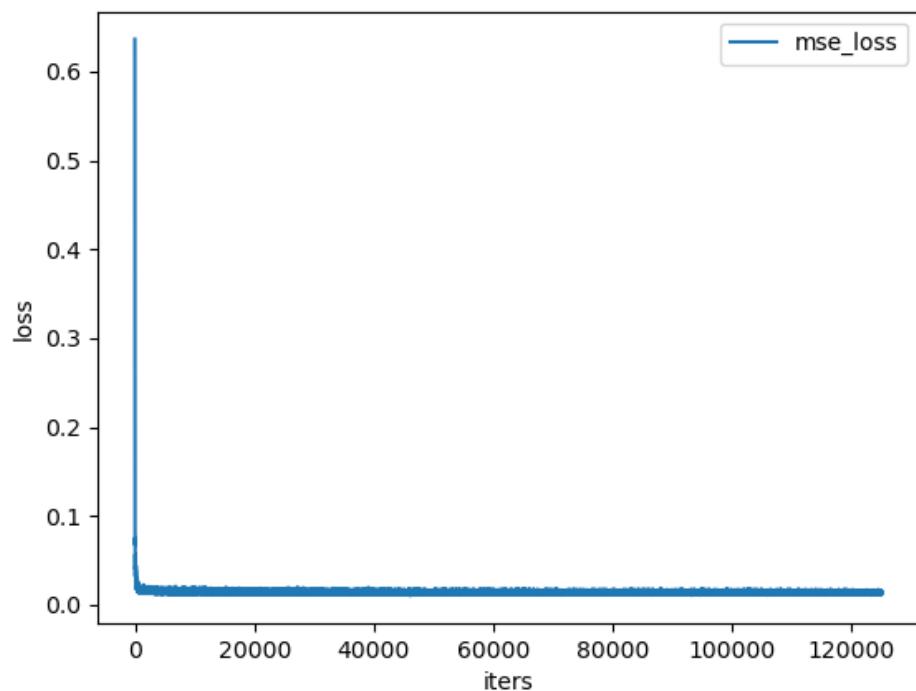
其餘的部分都照原 paper 或是助教所提供的 tips 設定。唯一比較特別的是 KL Loss 的 Lambda 設定公式如下：

```
def get_KL_Lambda(mse_loss):
    KL_Lambda = 1e-5
    if mse_loss < 0.015:
        KL_Lambda /= (1.0 * (mse_loss ** 2))
    return KL_Lambda

def KL_loss(mean, logvar):
    return torch.mean(-0.5 * (1 + logvar - mean ** 2 - torch.exp(logvar)))
```

利用 mse_loss 來決定 Lambda 的大小,主要是想讓 model 先學到一定的 image reconstruct 能力,再去學習產生 normal distribution noise 的能力。

2. (5%)



3. (5%)

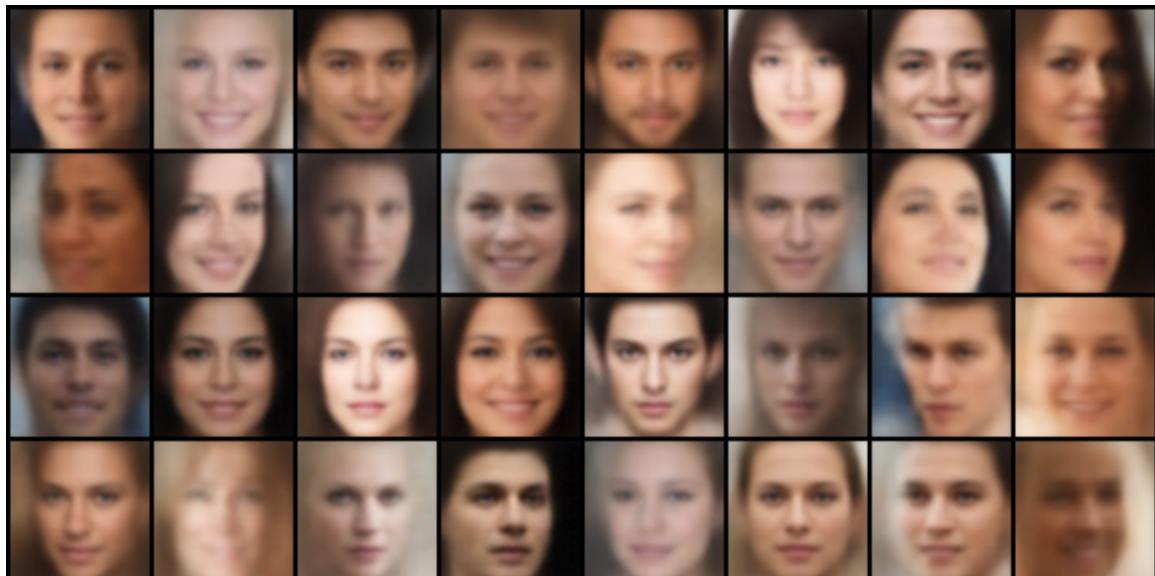
top: testing images / middle: reconstructed images / bottom: MSE



0.016, 0.012, 0.012, 0.012, 0.008, 0.016, 0.010, 0.014, 0.020, 0.012

此處的 MSE 是利用 `nn.MSELoss(reduction = ‘mean’)` 計算出來的，因為型別是 `tensor`，所以 scale 較助教所提供的範例更小。

4. (5%)

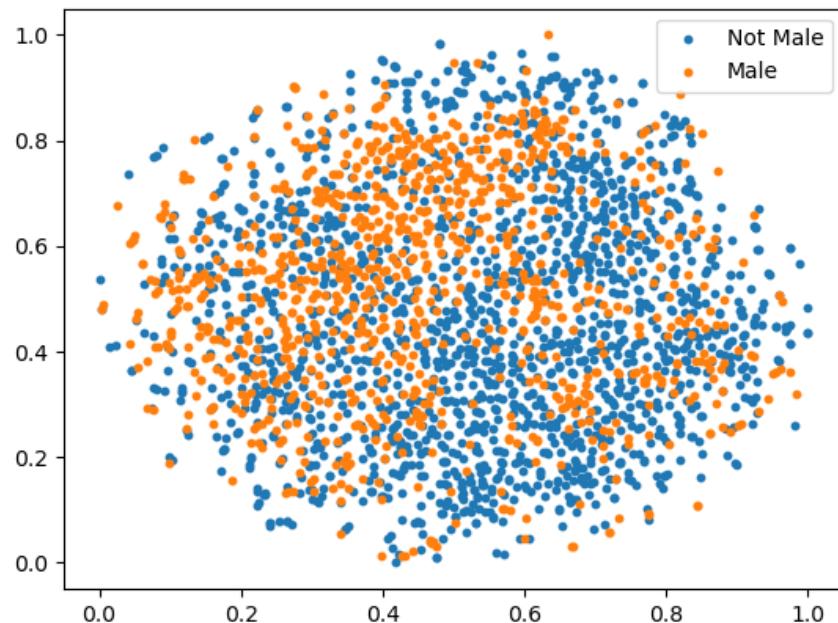


5. (5%)

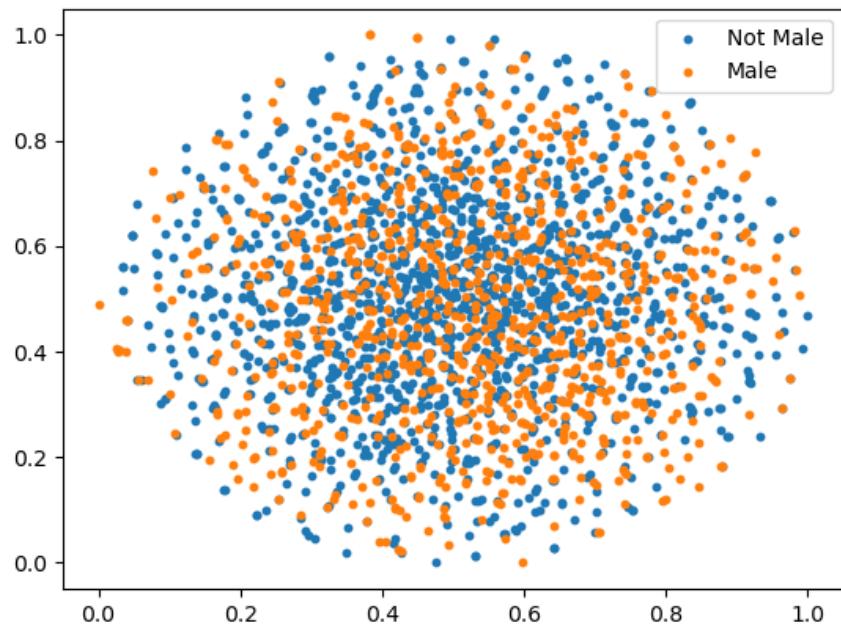
此題所選擇的 attribute 為 Male or Female。

觀察到比較特別的地方是，在不同的 training epoch 中，tsne 將此 attribute 區分的 distribution 有些微的不同，實際圖如下：

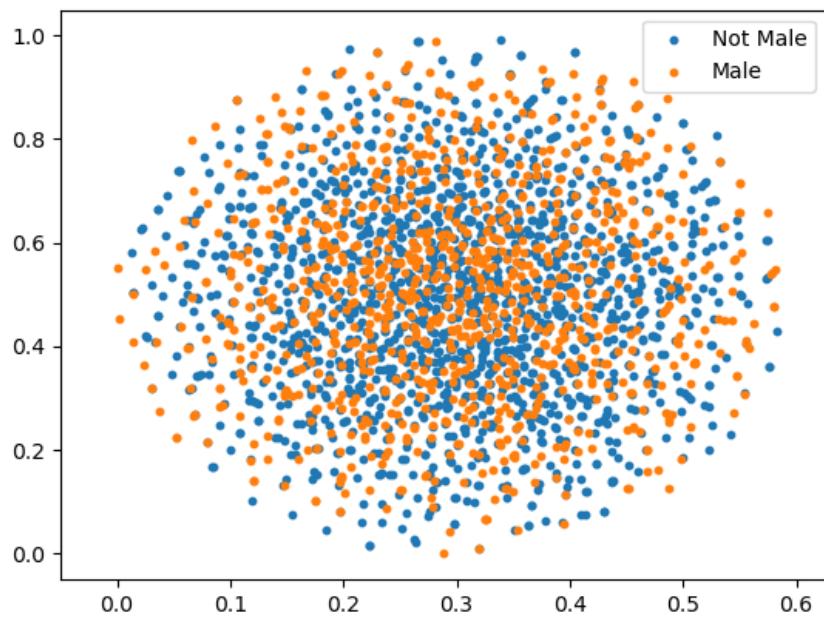
Early stage:



Middle stage:



Final stage:



6. (5%)

從 VAE 所產生的 reconstructed image 來看，可以發現其主要 focus 在人臉所面向的方向、膚色、五官以及背景顏色的部分；雖然有一定程度上的相似，但轉換前後還是有些明顯的差異，比如生理性別發生轉換、頭髮的呈現落差較大、鬍子細節被忽略等，而且整體的影像較為模糊。

而從 randomly generated image 來看，可以發現 VAE 所產生的影像多樣性非常廣，每張圖都好像真有其人，且幾乎不會有重複的人臉出現，不過仍保有上述影像較模糊的缺點。

最後從 tsne 於不同階段的呈現上來看，發現在 early stage 時，雖然整體 noises 分佈有類似於 normal distribution，但若個別來看，就不會有那麼自然的分佈，就像是助教於解說影片中所提到的結果；不過到了 training middle stage 時，即可觀察到 attribute 的有與無都分別有偏向 normal distribution 的走勢，已經不太能分辨兩者了；到 final stage 時，此點特徵更為明顯，圖中的 tsne 點都逐漸向中間靠攏。

在這次的實作中，覺得學到最特別的點在於，若要 model 學到很多種的 task（比如：reconstruction 的能力、產生圖具多樣性的能力等），就需要有個權衡重要性的 ranking，像是此題會先要求 mse_loss (reconstruction 能力) 達到一定的標準，再去要求 model 學習多樣性的變化 (KL Loss)，且此 KL Loss 還是會依照 mse_loss 產生變化的。

Problem 2: GAN (20%)

1. (5%)

Generator(

```
(decode): Sequential(  
    (0): Tran_Block(  
        (block): Sequential(  
            (0): 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(inplace=True)  
        )  
    )  
    (1): Tran_Block(  
        (block): Sequential(  
            (0): 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(inplace=True)  
        )  
    )  
    (2): Tran_Block(  
        (block): Sequential(  
            (0): 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(inplace=True)  
        )  
    )  
    (3): Tran_Block(  
        (block): Sequential(  
            (0): 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(inplace=True)  
        )  
    )  
    (4): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))  
    (5): Tanh()  
)
```

)

Discriminator(

(encode): Sequential(

(0): Conv_Block(

(block): Sequential(

(0): Conv2d(3, 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): LeakyReLU(negative_slope=0.2, inplace=True)

)

)

(1): Conv_Block(

(block): Sequential(

(0): Conv2d(64, 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): LeakyReLU(negative_slope=0.2, inplace=True)

)

)

(2): Conv_Block(

(block): Sequential(

(0): Conv2d(128, 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): LeakyReLU(negative_slope=0.2, inplace=True)

)

)

(3): Conv_Block(

(block): Sequential(

(0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)

(2): LeakyReLU(negative_slope=0.2, inplace=True)

)

)

(4): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))

(5): Sigmoid()

)

)

Model implementation details:

在本題中，使用的 Generator 和 Discriminator 採用了幾乎完全對稱的架構，且和上題 VAE 的 model 有很多相同的地方。

Generator 主要利用了 Tran_Block(ConvTranspose2d, BatchNorm2d, ReLU)。

先將 noise 經過四個 Tran _Block，然後再經過一個 ConvTranspose2d 以及最後的 Tanh，產生 fake image。

為助理解，架構流程如下圖所示：

```
class Tran_Block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Tran_Block, self).__init__()
        self.block = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True)
        )
    def forward(self, input):
        output = self.block(input)
        return output

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.decode = nn.Sequential(
            Tran_Block(128, 512, 4, 1, 0),
            Tran_Block(512, 256, 4, 2, 1),
            Tran_Block(256, 128, 4, 2, 1),
            Tran_Block(128, 64, 4, 2, 1),
            nn.ConvTranspose2d(64, 3, 4, 2, 1),
            nn.Tanh()
        )
    def forward(self, input):
        feature = input.view(input.shape[0], -1, 1, 1)
        image = self.decode(feature)
        return image
```

Discriminator 主要利用了 Conv_Block(Conv2d, BatchNorm2d, LeakyReLU)。

先將 image 經過四個 Conv_Block，然後再經過 Conv2d，以及最後的 Sigmoid，產生 real or fake 的 prediction。

為助理解，架構流程如下圖所示：

```
class Conv_Block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Conv_Block, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2, inplace = True)
        )
    def forward(self, input):
        output = self.block(input)
        return output

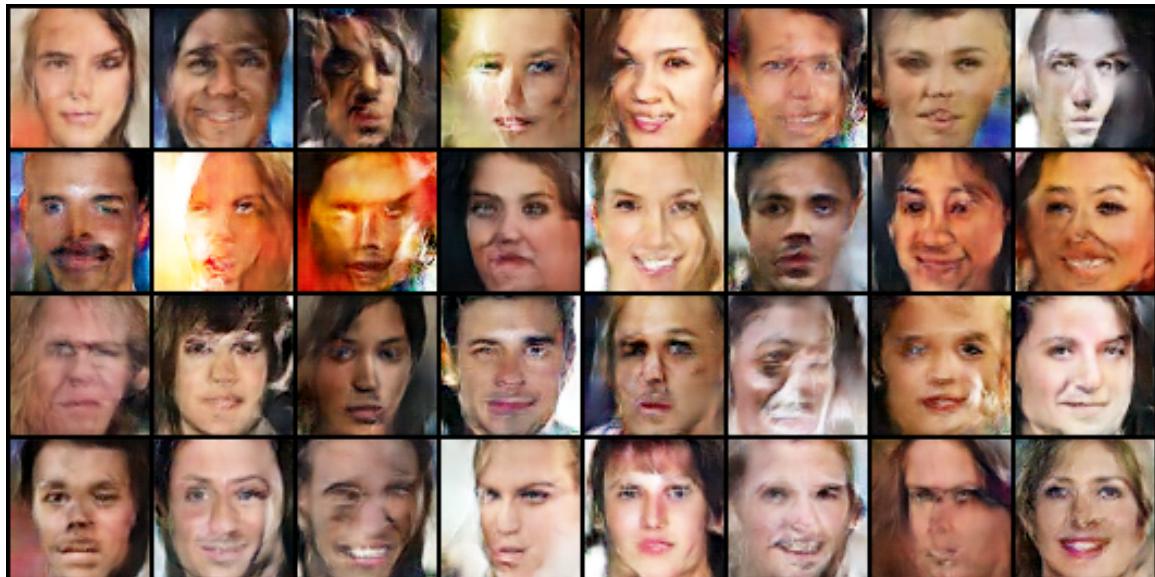

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.encode = nn.Sequential(
            Conv_Block(3, 64, 4, 2, 1),
            Conv_Block(64, 128, 4, 2, 1),
            Conv_Block(128, 256, 4, 2, 1),
            Conv_Block(256, 512, 4, 2, 1),
            nn.Conv2d(512, 1, 4, 1, 0),
            nn.Sigmoid()
        )
    def forward(self, input):
        batch, channel, width, height = input.shape
        prediction = self.encode(input).view(batch, 1)
        return prediction
```

Training implementation details:

本題 GAN 一共 train 了 100 個 epochs，Generator 和 Discriminator 皆使用了 Adam optimizer，且 fix learning rate 皆為 $1e-4$ ，並且使用了 RandomHorizontalFlip 的 data augmentation 技巧，batch_size 為 64，使用 BCE Loss。

主要的 Training 流程即是在每個 epoch 先訓練 Generator，再訓練 Discriminator，其餘的部分照的是原 paper 的設定或做微調整。

2. (5%)



3. (5%)

從 randomly generated image 來看，可以看到 GAN 所產生的圖較為清晰，不過仍有許多缺點，像是細節部分較為粗糙，圖像較易變形、扭曲，且較不真實，有時顏色呈現較不自然、五官位移。

在這次實作中，有種打開眼界的感覺，發現在 training 的過程中，可以利用這種 adversarial 的對抗式方式，讓多個 model 互相學習、教學相長，彼此越來越厲害，對我來說是一種很新奇的方式。不過也有很多要注意的地方，像是先訓練哪個 model、兩個 model 的規模等級（參數量）可能不宜相差太大，否則沒有辦法達到一個訓練的平衡狀態。

4. (5%)

就我實作所產生的結果來比較 VAE 和 GAN 兩種方法，發現兩個方法極端對立，VAE 所產生的圖較為真實卻模糊，而 GAN 所產生的圖較為虛假但清晰，可能是兩種訓練方法中，model 彼此注重的部分不一樣（比如說 Generator 和 Discriminator 在訓練過程中，Generator 所學到可以騙過 Discriminator 的部分跟我們人眼認為的真實不太一樣；而 VAE 較注重在降低 MSE Loss，所以才會有較模糊但又真實、較 general 的影像產生），而有此結果的產生。

Problem 3: DANN (35%)

1. (3%)

Source, Target	usps, mnist-m	mnist-m, svhn	svhn, usps
Accuracy	15.17%	41.79%	54.16%

2. (3+7%)

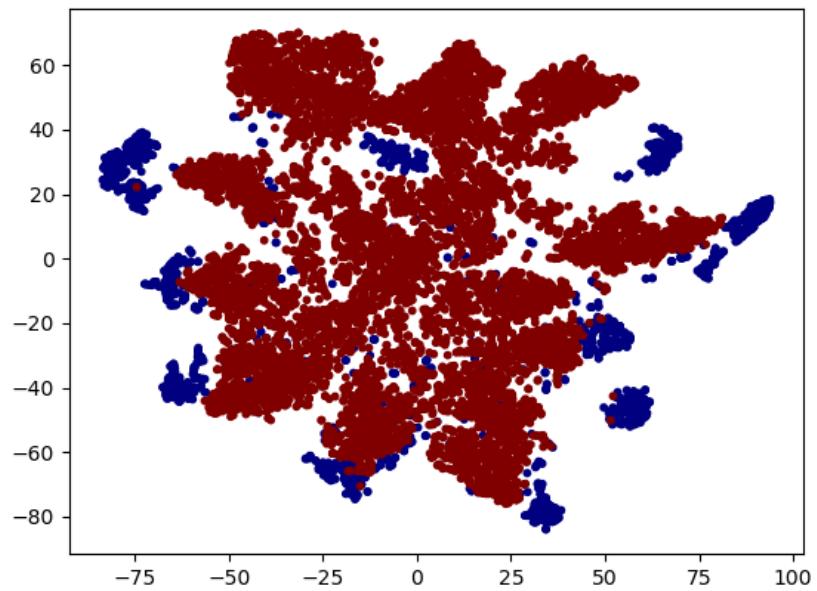
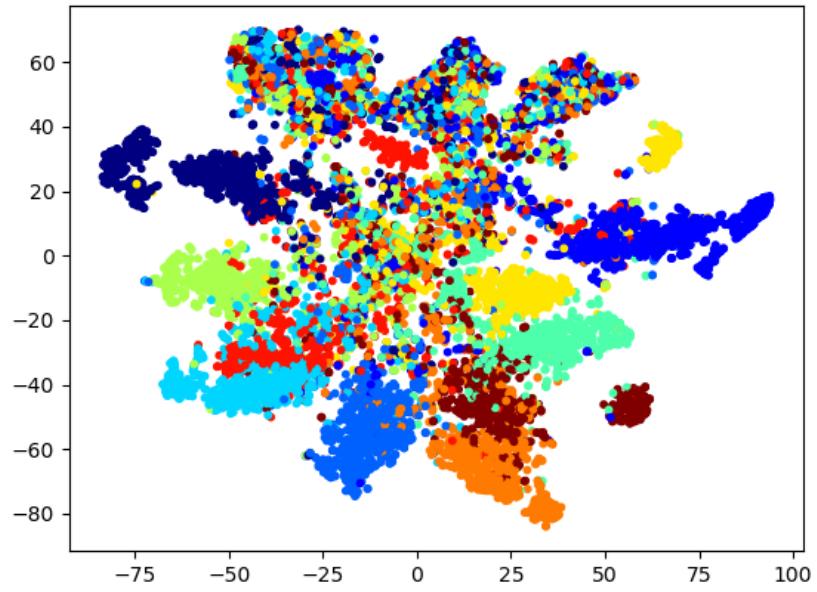
Source, Target	usps, mnist-m	mnist-m, svhn	svhn, usps
Accuracy	42.49%	47.30%	57.89%

3. (3%)

Source, Target	usps, mnist-m	mnist-m, svhn	svhn, usps
Accuracy	87.02%	83.51%	95.46%

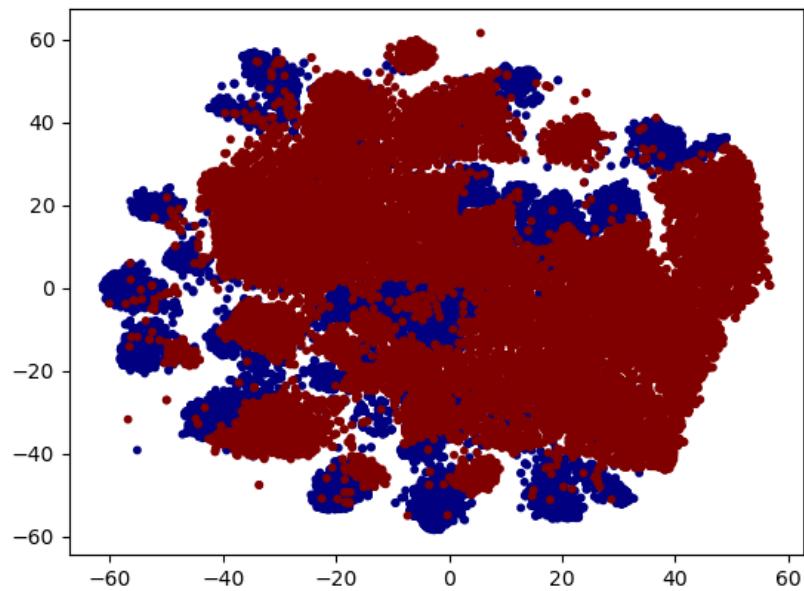
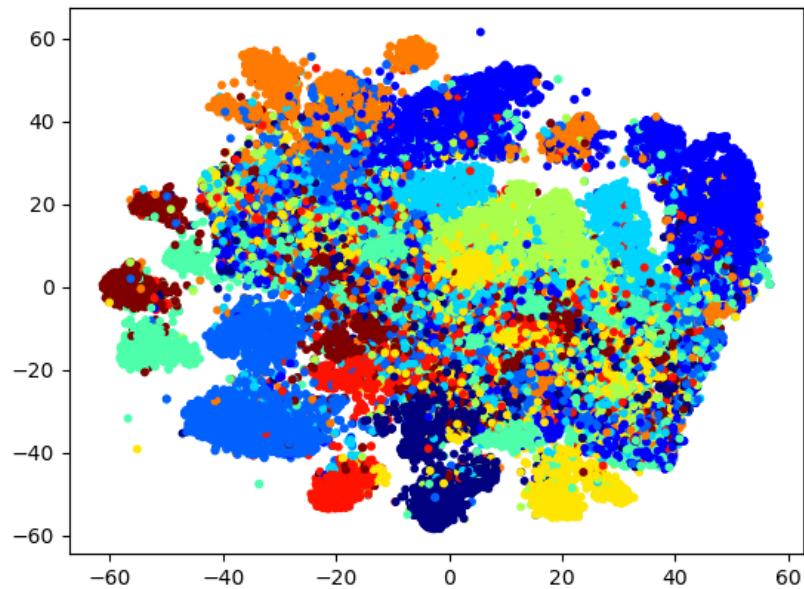
4. (6%)

source, target: usps, mnist-m
top: different digit classes / bottom: different domains



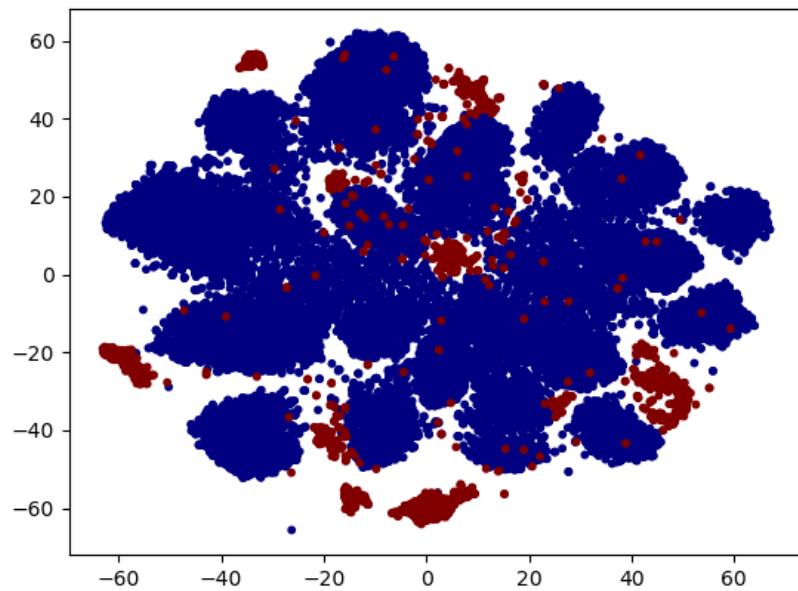
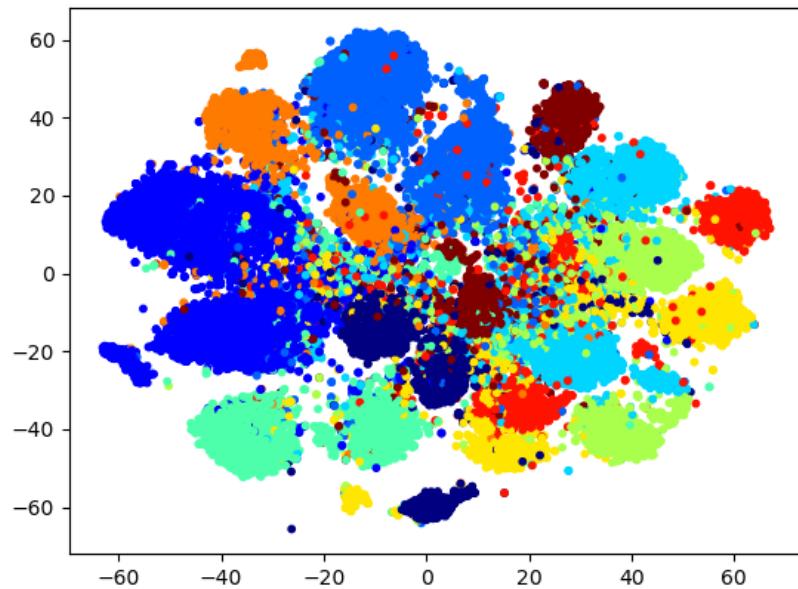
source (blue color) / target (red color)

source, target: mnist-m, svhn
top: different digit classes / bottom: different domains



source (blue color) / target (red color)

source, target: svhn, usps
top: different digit classes / bottom: different domains



source (blue color) / target (red color)

5. (6%)

```
Extractor()
(extract): Sequential(
    (0): Conv_Block(
        (block): Sequential(
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): ReLU(inplace=True)
        )
    )
    (1): Conv_Block(
        (block): Sequential(
            (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): ReLU(inplace=True)
        )
    )
    (2): Conv_Block(
        (block): Sequential(
            (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): ReLU(inplace=True)
        )
    )
    (3): Conv_Block(
        (block): Sequential(
            (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): ReLU(inplace=True)
        )
    )
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
)
```

```
Predictor(  
    (preprocess): Linear_Block(  
        (block): Sequential(  
            (0): Linear(in_features=512, out_features=1024, bias=True)  
            (1): ReLU(inplace=True)  
        )  
    )  
    (predict): Sequential(  
        (0): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=1024, out_features=512, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (1): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=512, out_features=256, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (2): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=256, out_features=128, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (3): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=128, out_features=64, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (4): Linear(in_features=64, out_features=10, bias=True)  
        (5): LogSoftmax(dim=1)  
    )  
)
```

```
Classifier(  
    (preprocess): Linear_Block(  
        (block): Sequential(  
            (0): Linear(in_features=512, out_features=1024, bias=True)  
            (1): ReLU(inplace=True)  
        )  
    )  
    (predict): Sequential(  
        (0): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=1024, out_features=512, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (1): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=512, out_features=256, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (2): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=256, out_features=128, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (3): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=128, out_features=64, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (4): Linear(in_features=64, out_features=2, bias=True)  
        (5): LogSoftmax(dim=1)  
    )  
)
```

Model implementation details:

在本題中，使用的 Predictor 和 Classifier 採用了幾乎完全的架構。

Extractor 主要利用了 Conv_Block(Conv2d, BatchNorm2d, ReLU) 和 MaxPool2d。

先將 image 經過四個 Conv_Block，然後再經過一個 MaxPool2d 產生 features。

為助理解，架構流程如下圖所示：

```
class Conv_Block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Conv_Block, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True)
        )
    def forward(self, input):
        output = self.block(input)
        return output

class Extractor(nn.Module):
    def __init__(self):
        super(Extractor, self).__init__()
        self.extract = nn.Sequential(
            Conv_Block(3, 64, 3, 2, 1),
            Conv_Block(64, 128, 3, 2, 1),
            Conv_Block(128, 256, 3, 2, 1),
            Conv_Block(256, 512, 3, 2, 1),
            nn.MaxPool2d(2)
        )
    def forward(self, input):
        batch, channel, width, height = input.shape
        features = self.extract(input).view(batch, -1)
        return features
```

Predictor 主要利用了 Linear_Block(Linear, ReLU) 和 Linear module 以及 LogSoftmax module。

先將 feautures 經過四個 Linear_Block，然後再經過一個 Linear module，以及最後的 LogSoftmax module，產生 class 的 prediction。

為助理解，架構流程如下圖所示：

```
class Linear_Block(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Linear_Block, self).__init__()
        self.block = nn.Sequential(
            nn.Linear(in_channels, out_channels),
            nn.ReLU(True)
        )
    def forward(self, input):
        output = self.block(input)
        return output

class Predictor(nn.Module):
    def __init__(self):
        super(Predictor, self).__init__()
        self.preprocess = Linear_Block(512, 1024)
        self.predict = nn.Sequential(
            Linear_Block(1024, 512),
            Linear_Block(512, 256),
            Linear_Block(256, 128),
            Linear_Block(128, 64),
            nn.Linear(64, 10),
            nn.LogSoftmax(dim = 1)
        )
    def forward(self, input):
        input = self.preprocess(input)
        prediction = self.predict(input)
        return prediction
```

Classifier 主要利用了上述所提過的 Linear_Block(Linear, ReLU) 和 Linear module 以及 LogSoftmax module。

先將 feautures 經過四個 Linear_Block，然後再經過一個 Linear module，以及最後的 LogSoftmax module，產生 domain 的 prediction。如上述所提，Classifier 和 Predictor 的架構相當類似，不過為了要混淆 Extractor 所產生 feature 的 domain，Classifier 會將 input 先經過一次 Gradient Reverse Layer。

為助理解，架構流程如下圖所示：

```
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.preprocess = Linear_Block(512, 1024)
        self.predict = nn.Sequential(
            Linear_Block(1024, 512),
            Linear_Block(512, 256),
            Linear_Block(256, 128),
            Linear_Block(128, 64),
            nn.Linear(64, 2),
            nn.LogSoftmax(dim = 1)
        )
    def forward(self, input, lambda_term):
        reverse_input = Gradient_Reverse_Layer.apply(input, lambda_term)
        reverse_input = self.preprocess(reverse_input)
        classification = self.predict(reverse_input)
        return classification

class Gradient_Reverse_Layer(Function):
    @staticmethod
    def forward(ctx, input, lambda_term):
        ctx.lambda_term = lambda_term
        return input.view_as(input)

    @staticmethod
    def backward(ctx, grad_output):
        output = grad_output.neg() * ctx.lambda_term
        return output, None
```

Training implementation details:

本題 DANN 一共 train 了 20 個 epochs，Extractor, Predictor 和 Classifier 皆使用了 Adam optimizer，且 fix learning rate 皆為 $1e-3$ ，並無使用 data augmentation 的技巧，batch_size 設為 usps: 64, mnist-m: 256, svhn: 256，使用 NLL Loss。

主要的 Training 流程即是照原 paper 的設定或做微調整。比較不同的是對影像 transform 的方式，會將影像進行 Normalize([0.5], [0.5]) 的轉換，且如果影像中有包含彩色影像，則會將其轉換成灰階影像。

在此題中有從 training set 中切出 30% 作為 validation set，然後利用 validation 結果較好的 model 來做 testing。

6. (7%)

從 tsne 的結果圖來看，可以發現不論是三組中的哪組 task，DANN 的方法皆有達到混淆 feature 的目的。先從區分 class 的 tsne 結果來看，可以看到至少有不少部分的 label 被以相同的顏色標註出來，顯示該堆 feature 是有被分到同一類的，且 source 的區分較開，target 則較位於中間區塊，相當合理。再搭配相對應的 domain tsne 結果對照來看，可以發現在各個顏色的 feature 堆中，都的確有參雜著 source 和 target 的 feature。不過不同 task 間 data 量有一定的差距，顯示起來不一定那麼美觀，有些部份重疊被蓋住了，而且跑 tsne 結果跑比較久。

在這次的 implement 中，學習到 transform 在 data 的 initialize 上也佔了很重要的角色，往往因為不同的 transform 處理方式，就讓 training 過程非常不穩、甚至 train 不太起來。在 data imbalance 比較嚴重的時候，可以利用 zip 以及

不同 batch size 的方法，來使得大部分的 data 都可以被 model 看過，或是避免不同步的訓練。也學習到 Transfer learning 的特別之處，利用現有 label 的 data 來學習沒有 label 的 data 相關的 task。

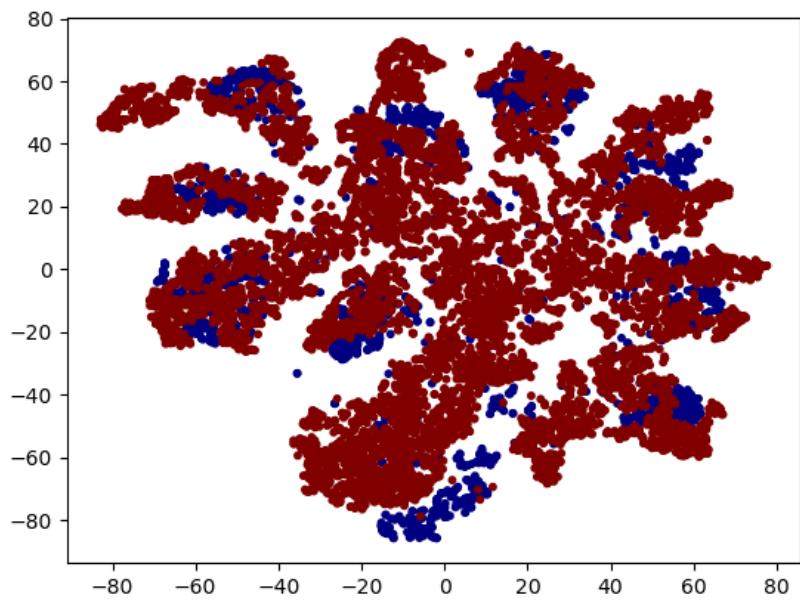
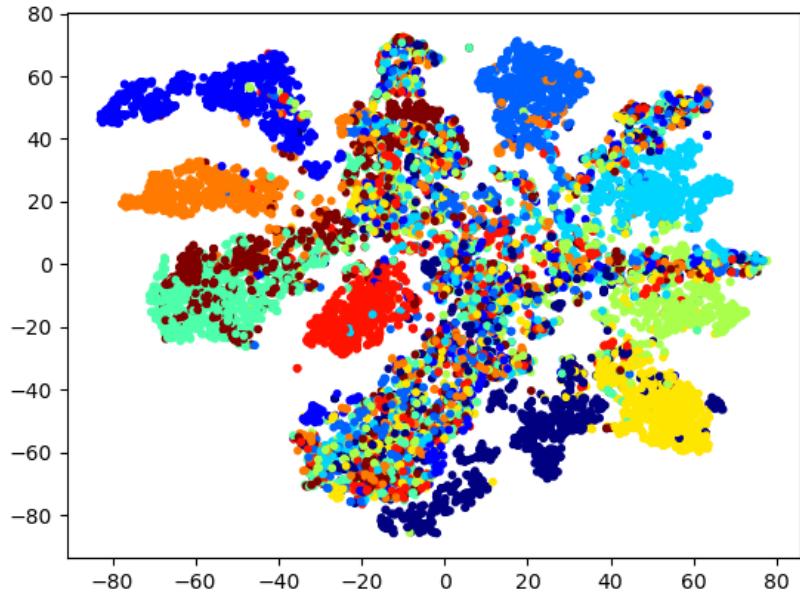
Problem 4: Improved UDA model (35%)

1. (6+10%)

Source, Target	usps, mnist-m	mnist-m, svhn	svhn, usps
Accuracy	45.94%	53.78%	60.03%

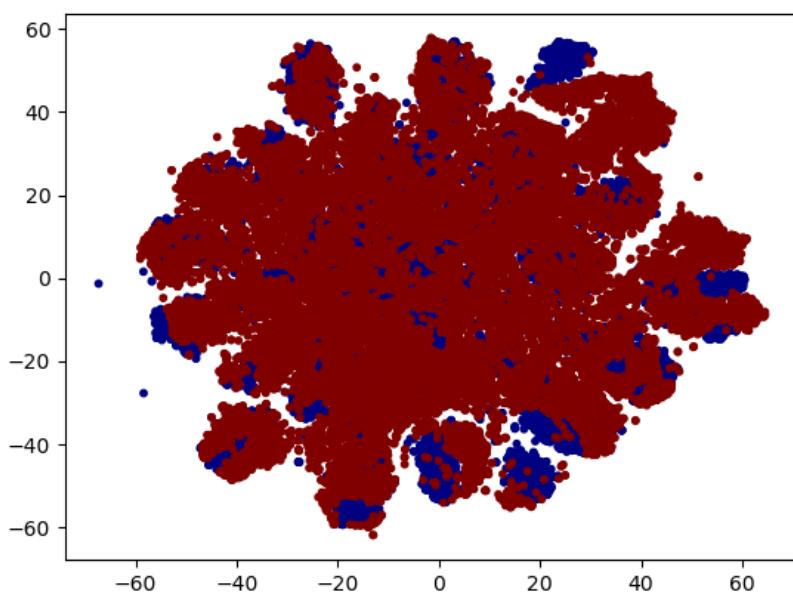
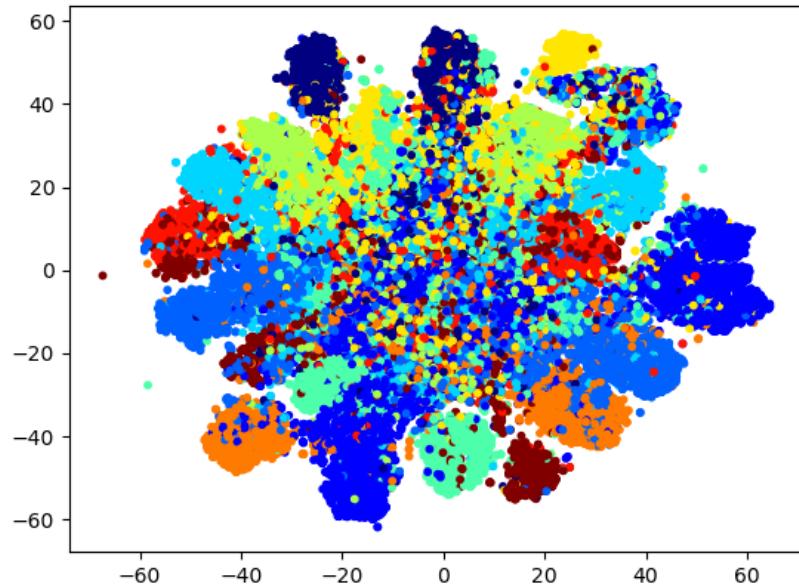
2. (6%)

source, target: usps, mnist-m
top: different digit classes / bottom: different domains



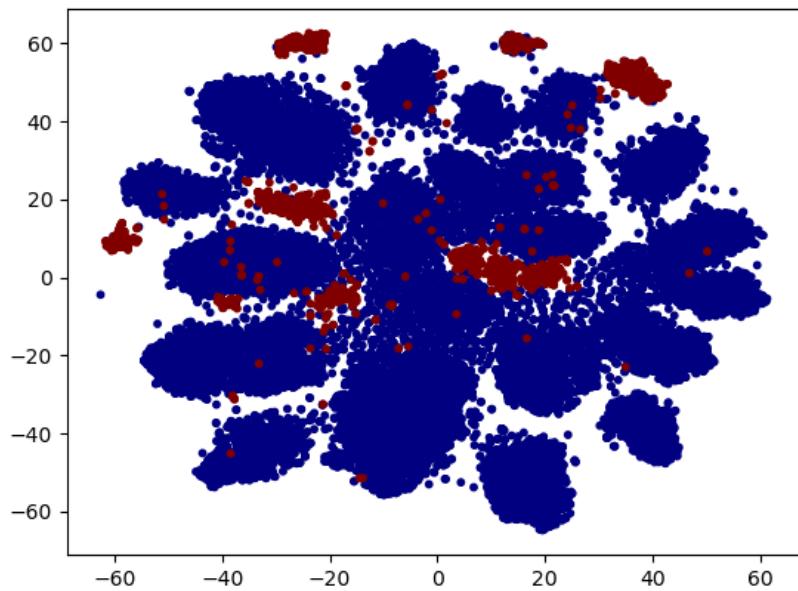
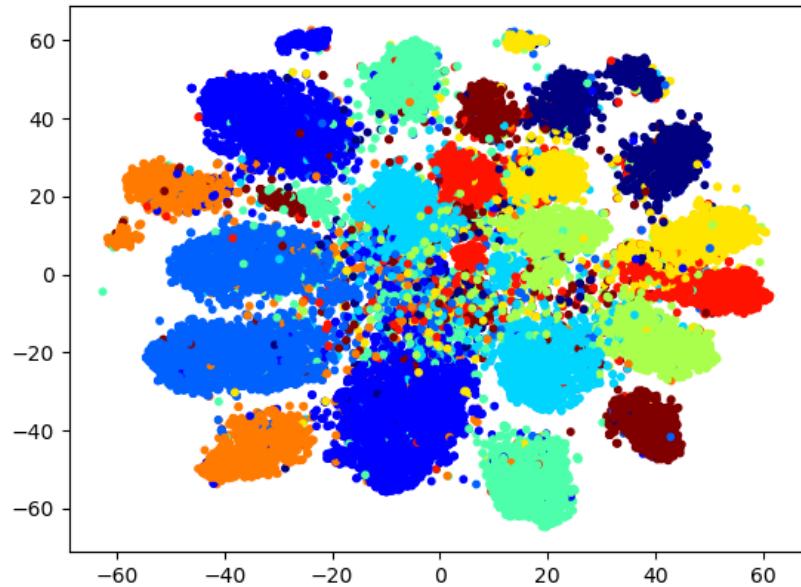
source (blue color) / target (red color)

source, target: mnist-m, svhn
top: different digit classes / bottom: different domains



source (blue color) / target (red color)

source, target: svhn, usps
top: different digit classes / bottom: different domains



source (blue color) / target (red color)

3. (6%)

```
Extractor_pretrain(  
    (extract): Sequential(  
        (0): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (1): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (2): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (3): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
)
```

```
Classifier(  
    (preprocess): Linear_Block(  
        (block): Sequential(  
            (0): Linear(in_features=512, out_features=1024, bias=True)  
            (1): ReLU(inplace=True)  
        )  
    )  
    (classify): Sequential(  
        (0): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=1024, out_features=512, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (1): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=512, out_features=256, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (2): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=256, out_features=128, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (3): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=128, out_features=64, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (4): Linear(in_features=64, out_features=10, bias=True)  
        (5): LogSoftmax(dim=1)  
    )  
)
```

```
Extractor_train(  
    (extract): Sequential(  
        (0): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (1): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (2): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (3): Conv_Block(  
            (block): Sequential(  
                (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
                (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,  
track_running_stats=True)  
                (2): ReLU(inplace=True)  
            )  
        )  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
)
```

```
Discriminator(  
    (preprocess): Linear_Block(  
        (block): Sequential(  
            (0): Linear(in_features=512, out_features=1024, bias=True)  
            (1): ReLU(inplace=True)  
        )  
    )  
    (classify): Sequential(  
        (0): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=1024, out_features=512, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (1): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=512, out_features=256, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (2): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=256, out_features=128, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (3): Linear_Block(  
            (block): Sequential(  
                (0): Linear(in_features=128, out_features=64, bias=True)  
                (1): ReLU(inplace=True)  
            )  
        )  
        (4): Linear(in_features=64, out_features=2, bias=True)  
        (5): LogSoftmax(dim=1)  
    )  
)
```

Model implementation details:

在本題中，一共使用了三種 model（四個），分別為 Extractor（包含 Extractor_pretrain, Extractor_train ），Classifier 和 Discriminator。上述三種 model 皆和前幾題中所提到時的架構、參數完全相同，控制此變因主要是想要利用方法和 model 的 training 方式，來達到 improve 的效果，所以 model 架構的部分就不在此重複提了。

不過為助理解，還是附上架構圖如下圖所示：

```
class Conv_Block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Conv_Block, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True),
        )
    def forward(self, input):
        return self.block(input)

class Linear_Block(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Linear_Block, self).__init__()
        self.block = nn.Sequential(
            nn.Linear(in_channels, out_channels),
            nn.ReLU(True),
        )
    def forward(self, input):
        return self.block(input)
```

```

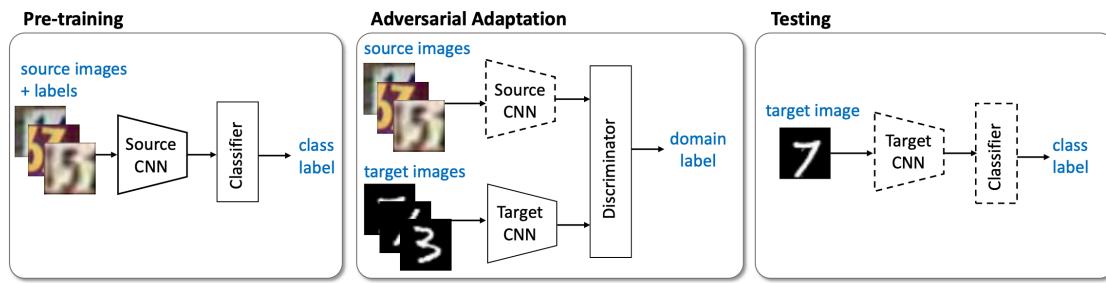
class Extractor(nn.Module):
    def __init__(self):
        super(Extractor, self).__init__()
        self.extract = nn.Sequential(
            Conv_Block(3, 64, 3, 2, 1),
            Conv_Block(64, 128, 3, 2, 1),
            Conv_Block(128, 256, 3, 2, 1),
            Conv_Block(256, 512, 3, 2, 1),
            nn.MaxPool2d(2)
        )
    def forward(self, input):
        batch, channel, width, height = input.shape
        features = self.extract(input).view(batch, -1)
        return features

class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.preprocess = Linear_Block(512, 1024)
        self.classify = nn.Sequential(
            Linear_Block(1024, 512),
            Linear_Block(512, 256),
            Linear_Block(256, 128),
            Linear_Block(128, 64),
            nn.Linear(64, 10),
            nn.LogSoftmax(dim = 1)
        )
    def forward(self, input):
        features = self.preprocess(input)
        prediction = self.classify(features)
        return prediction

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.preprocess = Linear_Block(512, 1024)
        self.classify = nn.Sequential(
            Linear_Block(1024, 512),
            Linear_Block(512, 256),
            Linear_Block(256, 128),
            Linear_Block(128, 64),
            nn.Linear(64, 2),
            nn.LogSoftmax(dim = 1)
        )
    def forward(self, input):
        features = self.preprocess(input)
        prediction = self.classify(features)
        return prediction

```

Training implementation details:



本題的 improve 方法是實作 ADDA 的 paper，詳細訓練流程如上圖所示，虛線表示 freeze。

Training 階段主要分成了兩部分（pretrain, train）：

在 pretrain 階段時，會先 pretrain 一個 train 在 source 上的 Extractor_pretrain 和 Classifier 一共兩個 model（皆為 10 epochs, Adam optimizer, fix learning rate 為 1e-3，無使用 data augmentation 技巧, batch_size 仍為 usps: 64, mnist-m: 256, svhn: 256，亦使用 NLL Loss）。

在 train 階段時，會將 pretrain 好的 Extractor_pretrain 作 freeze，然後訓練 Extractor_train 所產生的 feature 用以騙過同時訓練的 adversarial partner: Discriminator（兩個 model 皆 train 20 epochs, Adam optimizer, 無使用 data augmentation, batch_size 仍為 usps: 64, mnist-m: 256, svhn: 256，亦使用 NLL Loss）。最終希望達到 source 和 target 經過各自的 Extractor 所產生的 feature 是無法被分辨的，以達到 Transfer learning 的效果。

在最後的 test 階段，是以原先 train 好的 Classifier 來進行測試，以評估 Extractor_train 取出的 feature 能否被 train 在 source 上的 Classifier 區分。

上述主要的 Training 流程即是照原 paper 的設定。

以下為主要不同或有做額外調整之處：

和前一題一樣，主要做的調整是對影像的 transform 方式，但這次的調整前提在於「如果影像有 source 或 target 是來自 svhn dataset」，則轉換至灰階，且無做其他額外的 Normalize 轉換。

而和前一題不同的地方在於，針對不同的 task 有去做不同的 fix learning rate 設定，如下圖所示：

```
if source == 'mnistm' and target == 'svhn':  
    # 1e-3 for mnistm -> svhn  
    lr = 1e-3  
elif source == 'svhn' and target == 'usps':  
    # 1e-5 for svhn -> usps  
    lr = 1e-5  
elif source == 'usps' and target == 'mnistm':  
    # 1e-4 for usps -> mnistm  
    lr = 1e-4
```

其餘的部分皆和前一題相同。

在此題中有從 training set 中切出 30% 作為 validation set，然後利用 validation 結果較好的 model 來做 testing。

4. (7%)

從 tsne 的結果圖來看，可以發現不論是三組中的哪組 task，ADDA 的方法皆有達到比 DANN 更好的混淆 feature 效果，中間的混淆部分相對比較稀疏，不僅可以看到更多部分的 label 被以相同的顏色標註出來，顯示該堆 feature 是有被分到同一類的。而且對應的 domain tsne 結果顯示 source 和 target 的 feature 更完好的被混合再一起，更難被輕易區分。仍舊保有 source 區分較開，而 target 相較位於中間區塊的這項特徵。

在這次的 improve implement 中，發現在不同 task 上調整 learning rate 的大小有極大的差異，可能是因為不同 task 之間的距離差異不同，所以需要的學習步伐也要隨之調整；且仍舊發現 transform 在 data 的 initialize 上佔了極重要的角色，往往因為不同的 transform 處理方式，就讓 training 過程相差甚異、甚至 train 不太起來。而且覺得 test 階段的想法很特別，用原先 train 在 source 上的 Classifier 來進行 target domain 的 feature 分類，so clever!

URL:

<https://codertw.com/程式語言/585654/>

<https://arxiv.org/pdf/1505.07818.pdf>

<https://arxiv.org/pdf/1702.05464.pdf>

<https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>