# Homework #3

**December 1, 2020**

# Problem 1: VAE

1. Print the network architecture and describe implementation details.

```
VAE(
  (encoder): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
  )
  (fc1): Linear(in_features=4096, out_features=512, bias=True)
  (fc2): Linear(in_features=4096, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=4096, bias=True)
  (decoder): 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): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): Tanh()
  )
)
```

VAE architecture 如上圖。首先先將 input image normalize 到 [-1, 1]，在將其透過 encoder (4 層 Conv2d layer) 輸出成 (256, 4, 4) 的 tensor；接著將其 flatten 後透過兩種 fully connected layer，預測出 mean 和 logvar；再 random sample 出 100 維的 vector，利用前面預測出的 mean 和 logvar，reparameterize 出 latent vector ($z = \mu + \epsilon \times e^{0.5\ logvar}$)；透過 fc layer 還原為 4096 維，unflatten 為 (256, 4, 4)，最後通過 decoder 產生 reconstructed image。

Loss function 選用 reconstruction MSE 和 KL divergence 的組合，$\lambda_{KL}$ 取 5e-6，Optimizer 使用 Adam(lr=1e-4, betas=(0.5, 0.999))，batch size 取 64。

2. plot the learning curve

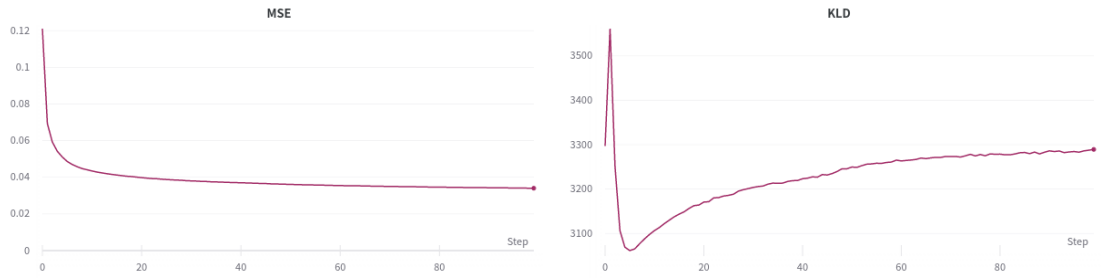
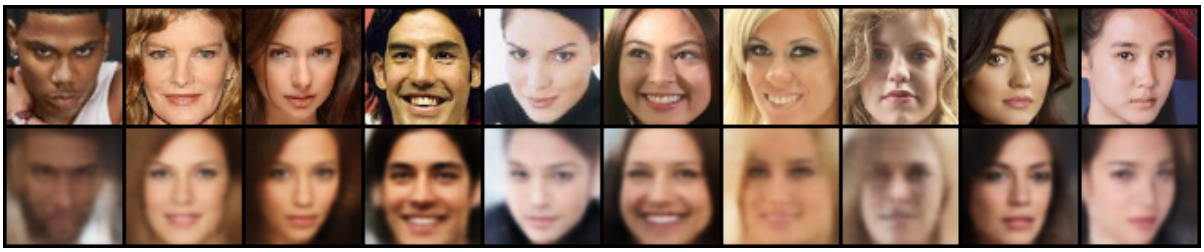



Fig 1.2

3. Plot 10 testing images and their reconstructed results (reconstructed images and MSE) from your model.



Reconstructed image

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0521 | 0.0404 | 0.0215 | 0.0500 | 0.0257 | 0.0309 | 0.0267 | 0.0422 | 0.0276 | 0.0259 |

MSE

4. Plot 32 random generated images from your model.

Fig 1.4

5. Visualize the latent space by mapping the latent vectors of the test images to 2D space (by tSNE) and color them with respect to an attribute (genter).

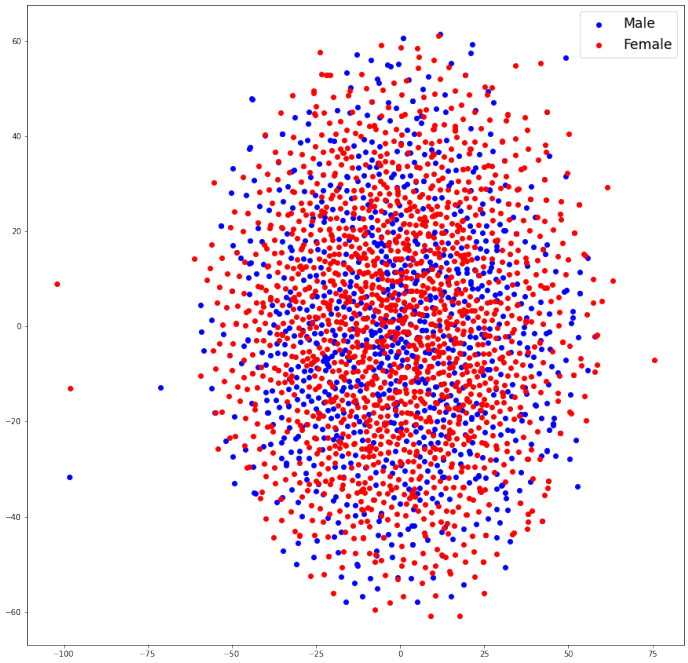



Fig 1.4

6. Discuss what you've observed and learned from implementing VAE.

- VAE 產生的圖較為模糊
- $\lambda_{KL}$ 會影響 training loss，5e-6 的 MSE 較 1e-5 小，KLD 較 1e-5 大。
- 從 Fig 1.4 可看出兩類別的分佈相似
- 大概 train 20 個 epoch 就有不錯的成果

# Problem 2: GAN

1. Print the network architecture and describe implementation details.

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=
        False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=
        False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=
        False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
        )
    (13): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 100, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(100, 200, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(200, 400, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(400, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(400, 800, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(800, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(800, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

選擇實做 DCGAN，Generator 的 input 為 100 維的 random noise (z)，輸出為生成的圖片，Discriminator 輸入為生成的圖片和原圖，輸出使用 Sigmoid function，使用 BCELoss 當作 Loss function，train Generator 時要讓 D(G(z)) 與 1 (real label) 接近，導致 Dicriminator 分辨錯誤，而 train Discriminator 要讓 D(G(z)) 與 0 (fake label) 接近，D(real image) 與 1 (real label) 接近，讓 Generator 生成出的照片可以分類成 fake，真實的照片分類為 real。Optimizer 使用 Adam(lr=2e-4, betas=(0.5, 0.999))，batch size 取 128，weight initialization 使用 Normal distribution with mean = 0 and stdev=0.02。

2. Plot 32 random images generated from your model.

Fig 2.2

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

- 在這組 dataset 上，Generator 沒有比 Discriminator 難 train，因此選擇讓兩個 update 的次數一樣
- 可以使用 RandomHorizontalFlip 增加訓練資料
- 大概在更新 3000 次之後五官就已經很明顯了，但臉仍會扭曲，直到 7000 次後，結果都差不多不怎麼改變

(a) 3000 iters (b) 7000 iters

4. Compare the difference between image generated by VAE and GAN.

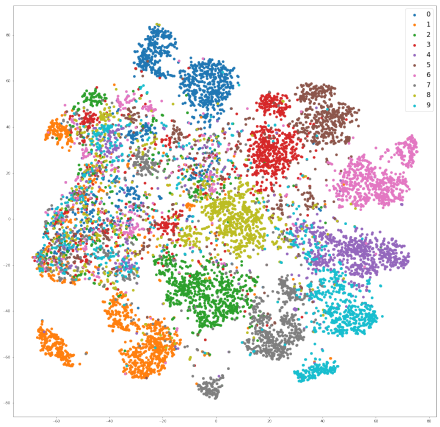
- GAN 產生的圖較為清晰
- GAN 的 loss function 可能因為沒有 MSE Loss，造成臉比起 VAE 容易歪掉
- GAN 沒有用 KLD 讓 input noise 與 image data 的分佈相像，造成比較容易生成不像臉的圖

# Problem 3: DANN

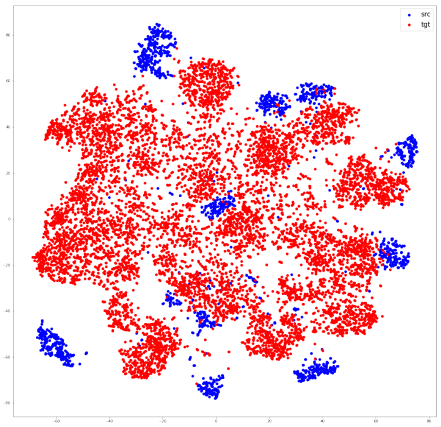
1-3. Compute the accuracy on target domain.

| | USPS → MNIST-M | MNIST-M → SVHN | SVHN → USPS |
|---|---|---|---|
| Trained on source | 3226/10000 (32.26%) | 13326/26032 (51.19%) | 1382/2007 (68.86%) |
| Adaptation (DANN/Improved) | 5098/10000 (50.98%) | 14688/26032 (56.42%) | 1385/2007 (69.01%) |
| Trained on target | 9844/10000 (98.44%) | 24376/26032 (93.64%) | 1958/2007 (97.56%) |

4. Visualize the latent space by mapping the testing images to 2D space (with t-SNE) and use different colors to indicate data of (a) different digit classes 0-9 and (b) different domains (source/target).

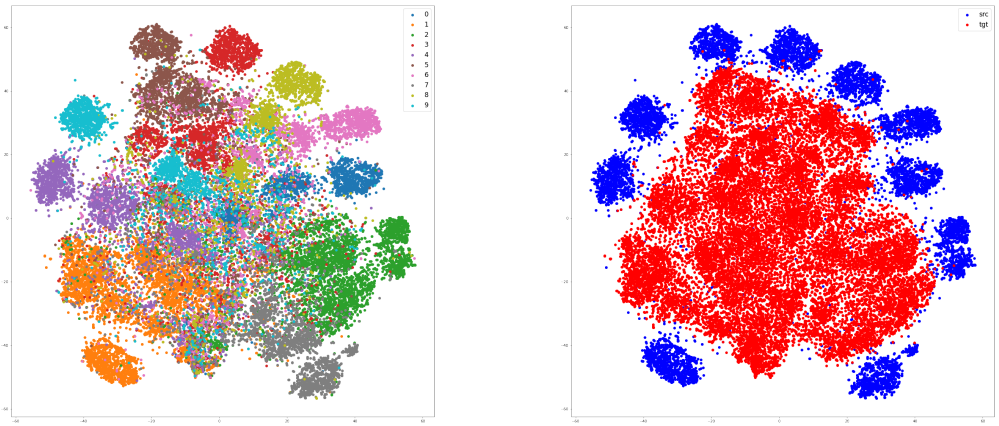


(c)



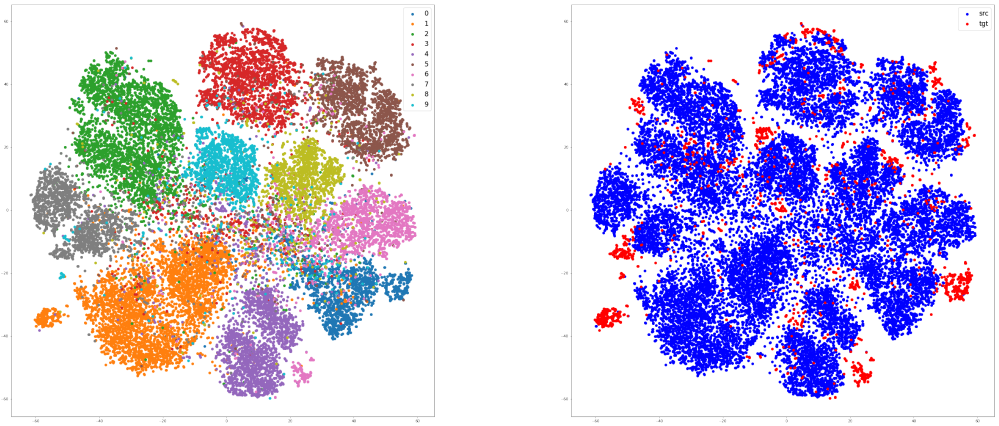
(d)

USPS → MNIST-M



(a) (b)

MNIST-M → SVHN



(a) (b)

SVHN → USPS

## 5. Describe the architecture & implementation detail.

```
DANN(
  (feature_extractor_conv): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
    (4): Conv2d(64, 50, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): ReLU(inplace=True)
  )
```

```
  (feature_extractor_fc): Sequential(
    (0): Linear(in_features=800, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): ReLU(inplace=True)
  )
  (class_classifier): Sequential(
    (0): Linear(in_features=512, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=100, out_features=10, bias=True)
    (4): LogSoftmax()
  )
  (domain_classifier): Sequential(
    (0): Linear(in_features=512, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=100, out_features=2, bias=True)
    (4): LogSoftmax()
  )
)
```

forward 會先通過 feature_extractor_conv 輸出維度為 (50, 4, 4)，flatten 後通過 feature_extractor_fc 產生 512 維的 feature extractor，再通過 class_classifier 產生 0-9 的機率預測，以及通過 reverse gradient layer 再通過 domain_classifier 產生兩個 domain 的機率預測；Loss function 為兩個 classifier output 的 Cross Entropy Loss。
Optimizer 使用 Adam(lr=1e-3, betas=(0.9, 0.999))，batch size 取 32。

6. Discuss what you've observed and learned from implementing DANN.

1. 從 t-SNE 圖來看，可以看到 source 的點都有很好的分群，而 target 的點除了 SVHN → USPS 之外，其他分群結果並沒有很好，可能是因為從少的 data 轉置多的所導致，或是從背景比較簡單到複雜導致

2. 對 source data 使用 colorjitter 做 data augmentation，可以幫助 MNISTM → SVHN 從 49% 進步到 56%，但其他的並沒有進步

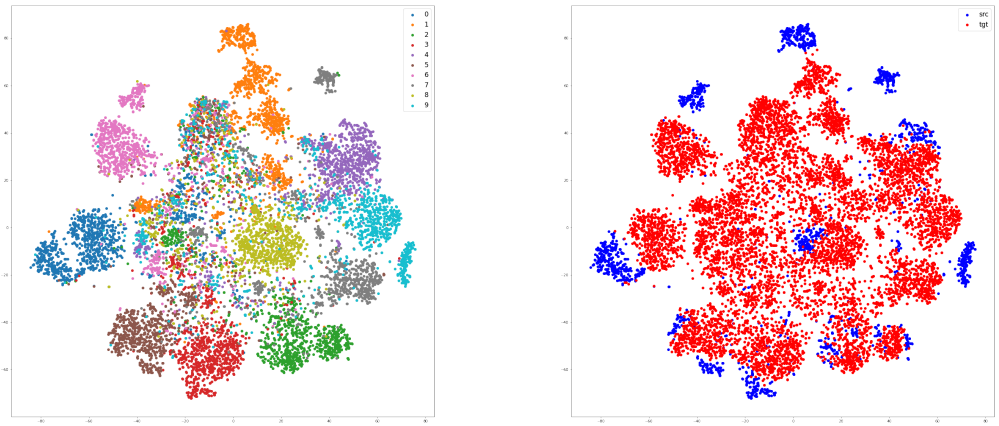
3. SVHN → USPS 只 train 在 source 上就已經表現的很好了，可能是從複雜轉簡單的緣故

# Problem 4: UDA

1. Compute the accuracy on target domain, while the model is trained on source and target domain.

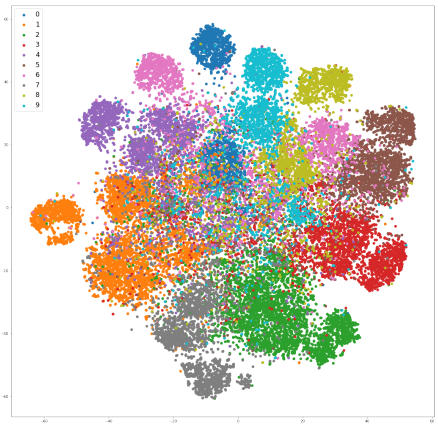
| | USPS → MNIST-M | MNIST-M → SVHN | SVHN → USPS |
|---|---|---|---|
| Trained on source | 3090/10000 (30.90%) | 13511/26032 (51.90%) | 1396/2007 (69.56%) |
| Adaptation (DANN/Improved) | 5956/10000 (59.56%) | 15499/26032 (59.54%) | 1508/2007 (75.14%) |
| Trained on target | 9864/10000 (98.64%) | 24476/26032 (94.02%) | 1963/2007 (97.81%) |

2. Visualize the latent space by mapping the testing images to 2D space (with t-SNE) and use different colors to indicate data of (a) different digit classes 0-9 and (b) different domains (source/target).

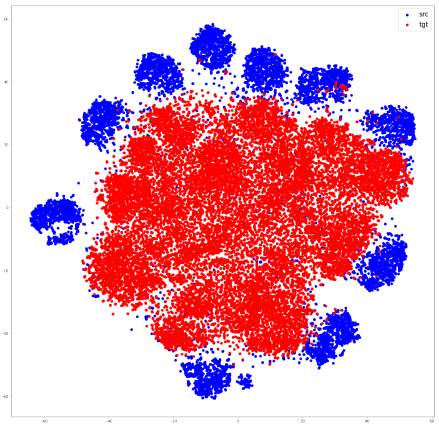


(a) (b)

USPS → MNIST-M

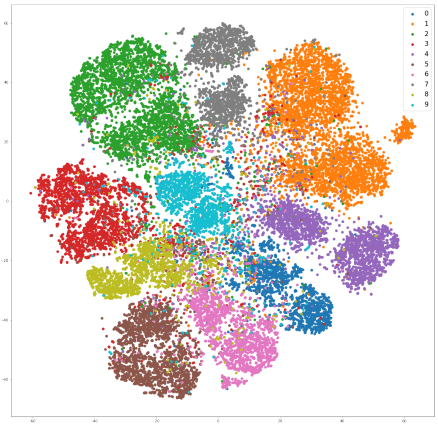


(a)

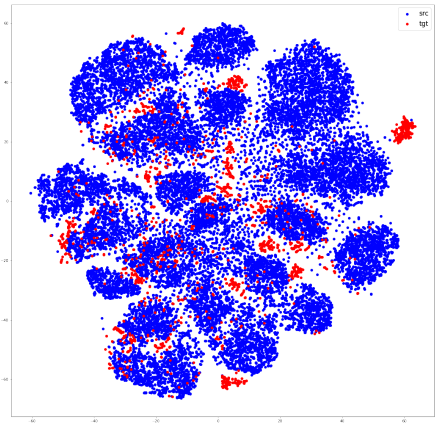


(b)

MNIST-M → SVHN



(a)



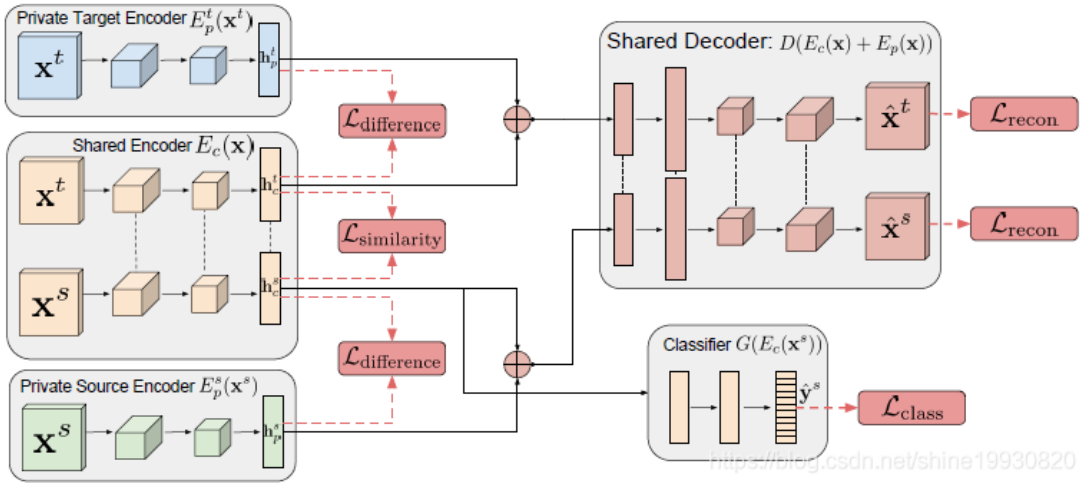
(b)

SVHN → USPS

3. Describe the architecture & implementation detail.

```
DSN(
  (source_encoder_conv): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
    (4): Conv2d(64, 50, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): ReLU(inplace=True)
  )
```

```
  (source_encoder_fc): Sequential(
    (0): Linear(in_features=800, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): ReLU(inplace=True)
  )
  (target_encoder_conv): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
    (4): Conv2d(64, 50, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): ReLU(inplace=True)
  )
  (target_encoder_fc): Sequential(
    (0): Linear(in_features=800, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): ReLU(inplace=True)
  )
  (shared_encoder_conv): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
    (4): Conv2d(64, 50, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): ReLU(inplace=True)
  )
  (shared_encoder_fc): Sequential(
    (0): Linear(in_features=800, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): ReLU(inplace=True)
  )
  (shared_encoder_pred_class): Sequential(
    (0): Linear(in_features=512, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=100, out_features=10, bias=True)
  )
  (shared_encoder_pred_domain): Sequential(
    (0): Linear(in_features=512, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=100, out_features=2, bias=True)
  )
  (shared_decoder_fc): Sequential(
    (0): Linear(in_features=512, out_features=800, bias=True)
    (1): ReLU(inplace=True)
  )
  (shared_decoder_conv): Sequential(
    (0): ConvTranspose2d(50, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
        )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(64, 64, kernel_size=(5, 5), stride=(1, 1), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
        )
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(1, 1), bias=False)
    (10): Tanh()
  )
)
```




DSN architecture

選擇實做 DSN，為方便與 DANN 比較，因此 encoder 與 classifier 都使用與 DANN 相同的架構；shared encoder 會提取共同特徵，private encoder 會提取個別 domain 特有特徵，classifier 會把共有特徵當 input 輸出各種類的機率，decoder 會把共有特徵和私有特徵和當作 input 輸出 reconstructed image。

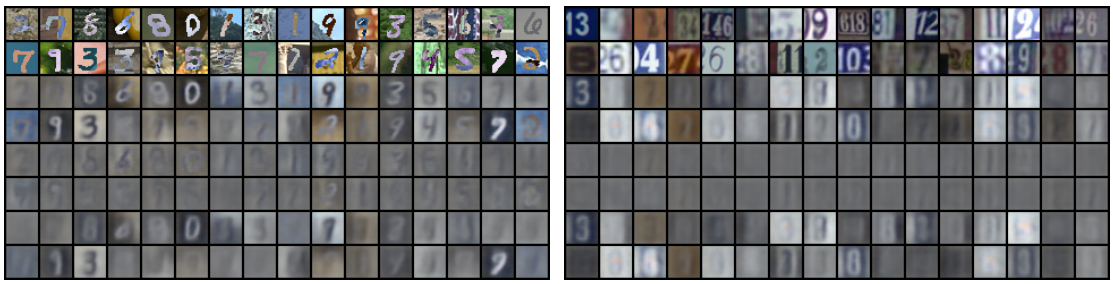
Loss function 為 $\mathcal{L} = \mathcal{L}_{Task} + \alpha\mathcal{L}_{recon} + \beta\mathcal{L}_{different} + \gamma\mathcal{L}_{similarity}$，選擇 $\alpha = 0.015$，$\beta = 0.075$，$\gamma = 0.25$；Optimizer 使用 Adam(lr=1e-3, betas=(0.9, 0.999))，batch size 取 32。

4. Discuss what you've observed and learned from implementing your improved UDA model.

   1. 在 Adaptation 上表現都比 DANN 好，從 t-SNE 分群上也比較明顯
   2. MNISTM → SVHN 在原本的 data 上比較難 train，導致 accuracy 只有 45%，需要加上 colorjitter，正確率才會比較高
   3. 下面的圖為 reconstructed image 和原圖，1-2 層為原圖，3-4 層為 private + shared 重構圖，5-6 層為 shared 重構圖，7-8 層為 private 重構圖，可以看到用 shared 去重構基本上都能看到模糊的數字

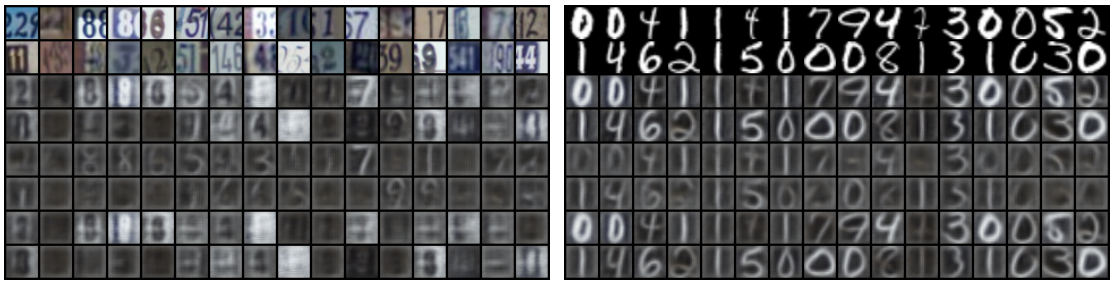
(a) src (b) tgt

USPS → MNIST-M



(a) src (b) tgt

MNIST-M → SVHN



(a) src (b) tgt

SVHN → USPS

# Reference

1. VAE ( https://github.com/sksq96/pytorch-vae/blob/master/vae.py)

2. DCGAN ( https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html )

3. DANN ( https://github.com/fungtion/DANN )

4. DSN ( https://github.com/fungtion/DSN )