DLCV – HW2

台科資工 M11015Q02 柯元豪

Problem 1

1. Show your model architecture in your report and describe implementation details.

DCGAN：

```
Generator(
  (net): 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(
  (net): 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()
```

```
  )
)
```

Implementation details:

```
## Hyper parameters ##
Image size : 64
Channel size : 3
Laten vector size : 100
Generator feature map size : 64
Discriminator feature map size : 64
Generator learning rate : 0.0002
Discriminator learning rate : 0.0004
Optimizer(Adam) beta1 : 0.5 ,beta2 : 0.999
Training epochs : 600
Batch size : 128
######################

## data augmentation ##
transform = transforms.Compose([
    transforms.Resize((image_size,image_size)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
 ])
######################

## other details ##
# during training
1. 每個 epoch 最後會用 geneartor 生成 1000 筆圖片計算 inception score
2. 判斷 epoch 數 > 100 (實驗數次後的經驗 >100 後表現才過 base line) 且 IS > 2.15 才存 model (後續會在加上 FID 做篩選)
# save images
torchvision.utils.save_image(image,path,normaize=True)
######################
```

2. Save the 1000 generated images in the assigned folder path for evaluation, and show the first 32 images in your report.

3. FID & IS Record

FID (Fréchet inception distance)	IS (Inception score)
27.4632	2.1450

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

這次實作前有先去看助教提供的 Tips，看到建議 train DCGAN 後，便照著 paper 敘述的架構把模型架好開始 train，在沒做任何參數與架構調整前，我發現這個 DCGAN 在 20 個 epoch 內就能做得不錯，生成的圖片還原度也蠻高的，但將圖片輸出測試 FID 時卻離 baseline 很遠，這時便了解用肉眼不一定能確定 performance 的好壞。

在嘗試各種實驗後，最終我採用了: training data 做 augmentation、batch size 設 128 並將 Discriminator 的 learning rate 調比 Generator 高一些、生成圖片時做 normalize，以這樣的機制結合 DCGAN 的初始架構跑了 600 個 epochs，生成的影像 FID、IS 便能達到上表所述。

而實驗中也學習到: 在 GAN 任務加上 data augmentation，雖然會讓模型學習速度稍微變慢，

但同樣也能像分類任務一樣使模型學得更好；batch size 與 learning rate 有很大的關聯，模型在較大的 batch size 時，因為每批學習的資料量較多，需要較小的 learning rate，以免學太快，因此在調參時，這兩個參數也會一起做調整；而 Disciminator 的 learning rate 稍微調比 Generator 高則是參考 Reference[2]，實作的 performance 也的確有進步。

Problem 2

1. Show your model architecture in your report and describe implementation details.

ACGAN:

```
Generator(
  (net): Sequential(
    (0): ConvTranspose2d(100, 224, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(224, 112, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(112, 56, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(56, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(56, 28, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(28, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(28, 3, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (net): Sequential(
    (0): Conv2d(3, 28, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Conv2d(28, 56, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(56, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Dropout(p=0.5, inplace=False)
    (7): Conv2d(56, 112, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
```

```
    (8): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): LeakyReLU(negative_slope=0.2, inplace=True)
    (10): Dropout(p=0.5, inplace=False)
    (11): Conv2d(112, 224, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (12): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): LeakyReLU(negative_slope=0.2, inplace=True)
    (14): Dropout(p=0.5, inplace=False)
    (15): Conv2d(224, 28, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (16): LeakyReLU(negative_slope=0.2, inplace=True)
    (17): Dropout(p=0.5, inplace=False)
  )
  (fc_real_fake): Linear(in_features=28, out_features=1, bias=True)
  (fc_classes): Linear(in_features=28, out_features=10, bias=True)
  (sigmoid): Sigmoid()
  (softmax): Softmax(dim=None)
)
```

Implementation details:

```
## Hyper parameters ##
Image size : 28
Channel size : 3
Laten vector size : 100
Generator feature map size : 28
Discriminator feature map size : 28
Generator learning rate : 0.0002
Discriminator learning rate : 0.00025
Optimizer(Adam) beta1 : 0.5 ,beta2 : 0.999
Training epochs : 500
Batch size : 256
#######################

## data augmentation ##
transform = transforms.Compose([
    transforms.Resize((image_size,image_size)),
    transforms.ToTensor(),
    transforms.Normalize([0. 5, 0.5, 0.5], [0.5, 0.5, 0.5])
 ])
#######################

## other details ##
# during training
```

1. 每個 epoch 最後會用 geneartor 生成 1000 筆圖片 (class[0-9] 各 100 張) 做 validation (use Classifier model)
2. 儲存 Validation Accuracy 最高的 model
3. Real fake classifier loss: BCELoss, Classes classifier loss: CrossEntropyLoss

How do I input the class labels into the model

1. DCGAN 生成 noise 方法相同先生成 normal 的 noise (shape 為(size,laten vector size))
2. 生成 label one-hot vector (shape 為(size, num_classes))
3. 將 lable 的 one-hot vector 直接取代掉 noise 的 [0: num_classes]
4. 以此方法讓 noise 帶著 label 的資訊，作為後續 Generator 的 input

```
noises = np.random.normal(0, 1, (size, lv_size))
label_arr = np.zeros((size, num_classes))
label_arr[:, i] = 1
noises[:, :num_classes] = label_arr
noises = torch.from_numpy(noises)
noises = noises.resize_(size, lv_size, 1, 1).float().to(device)
# save images
torchvision.utils.save_image(image,path,normaize=True)
#######################
```

2. Accuracy record

Accuracy (1000 output images) : 82%

3. Show 10 images for each digit (0-9) in your report.

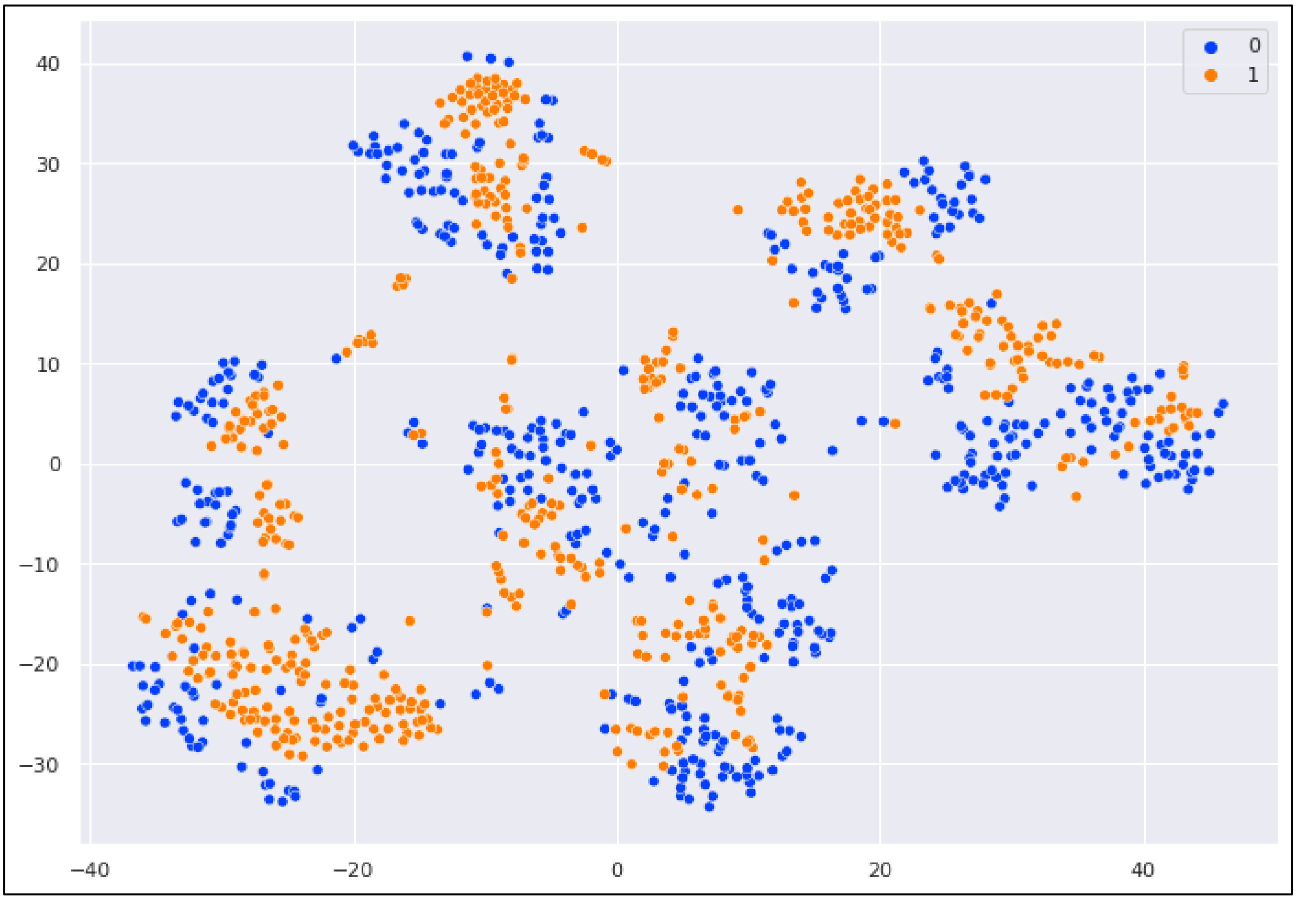
Problem 3

1. Accuracy record

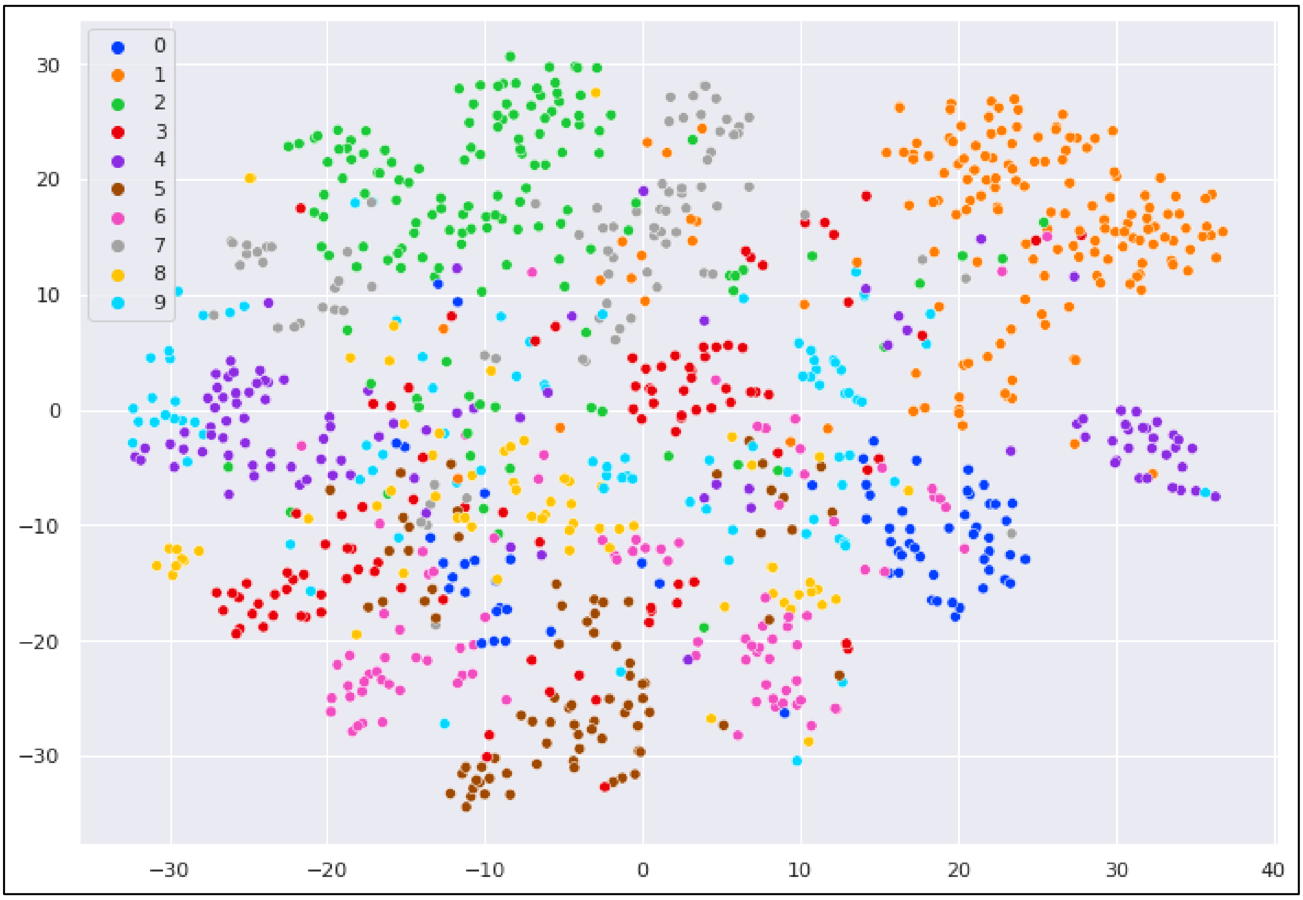
	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
Trained on source	73.44%	53.65%	15.14%
Adaptation (DANN/Improved)	85.70%	59.74%	34.11%
Trained on target	95.68%	93.60%	95.62%

2. Visualize the latent space by mapping the testing images to 2D space with t-SNE

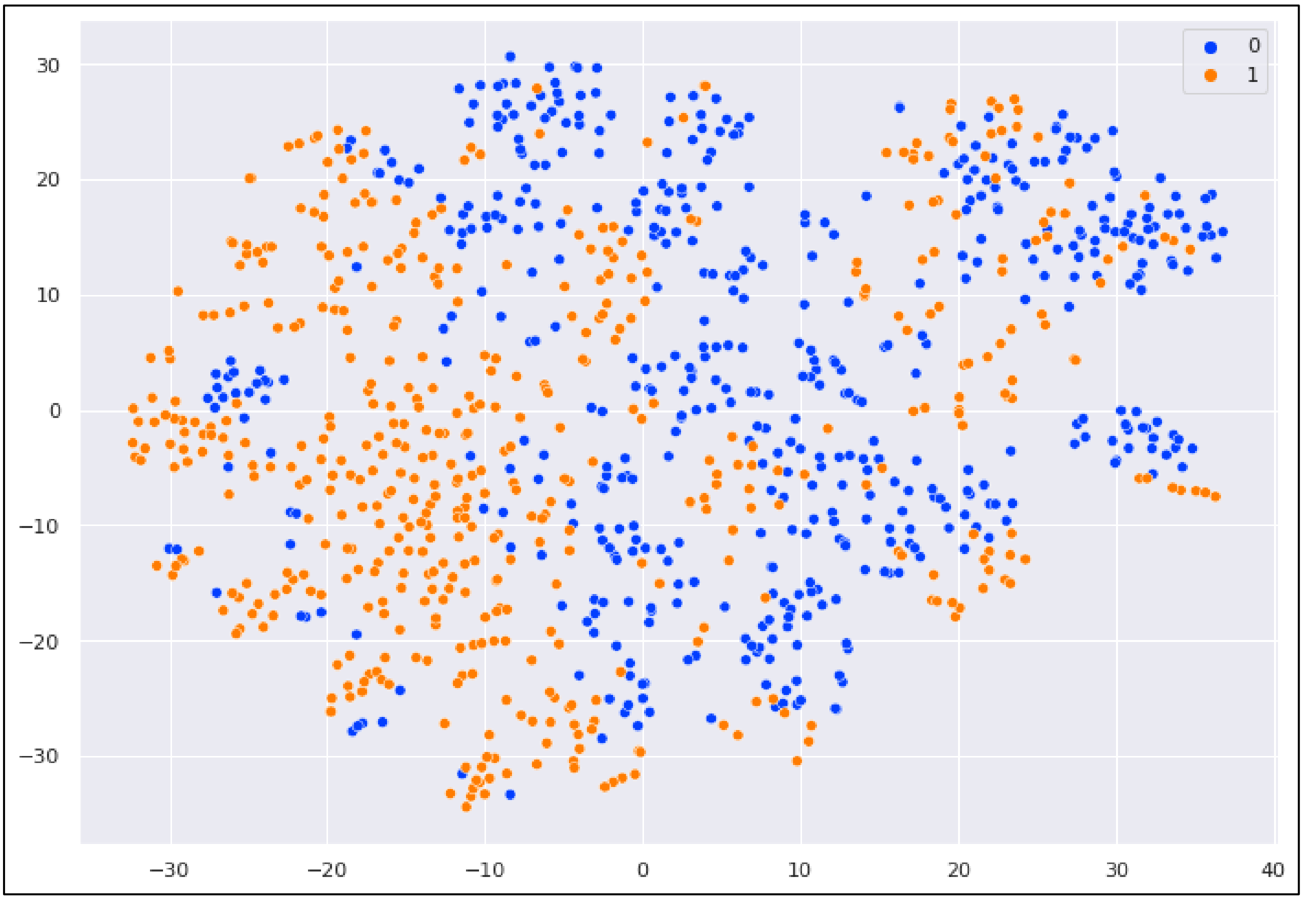
(MNIST-M → USPS)



(SVHN → MNIST-M)

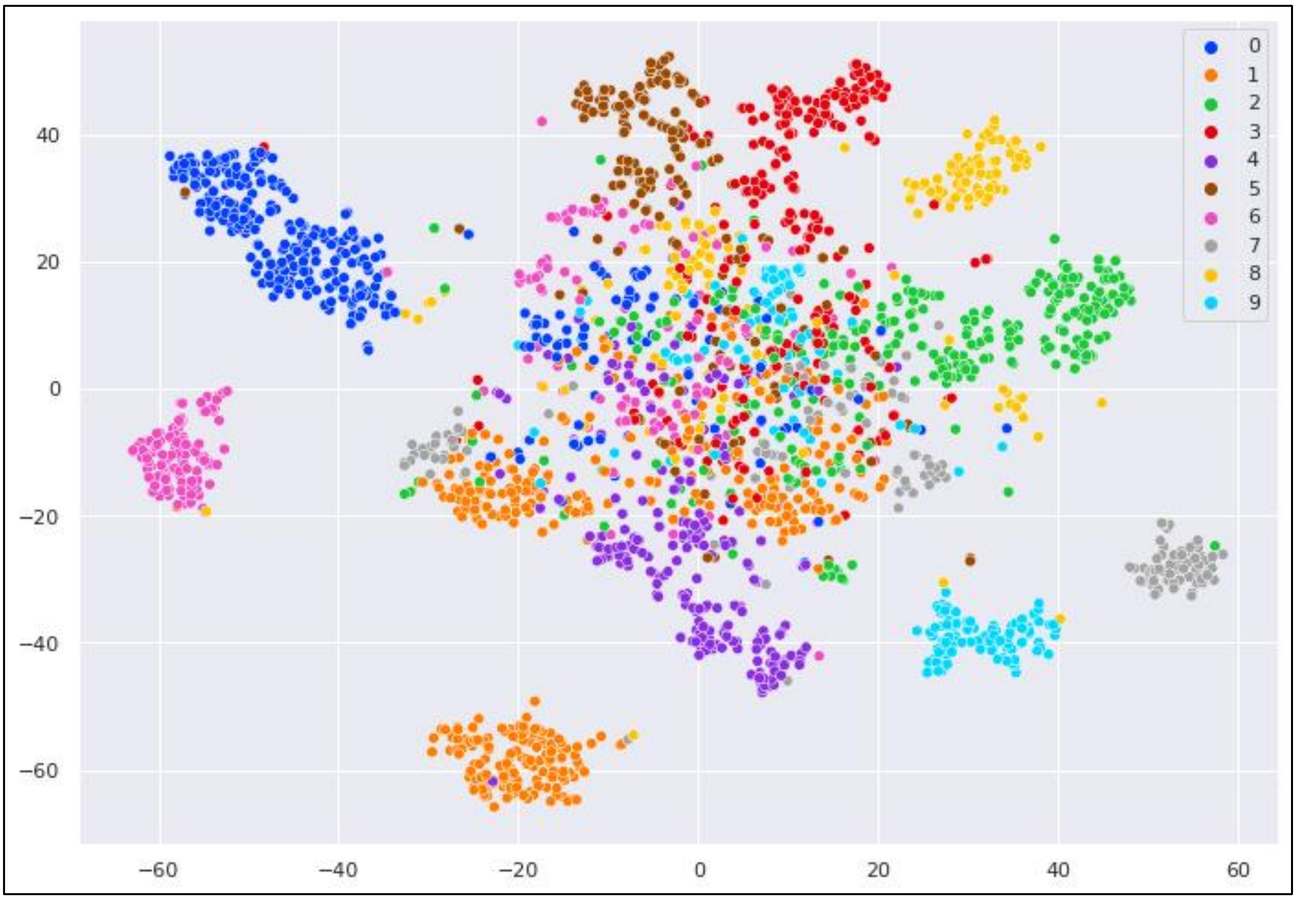


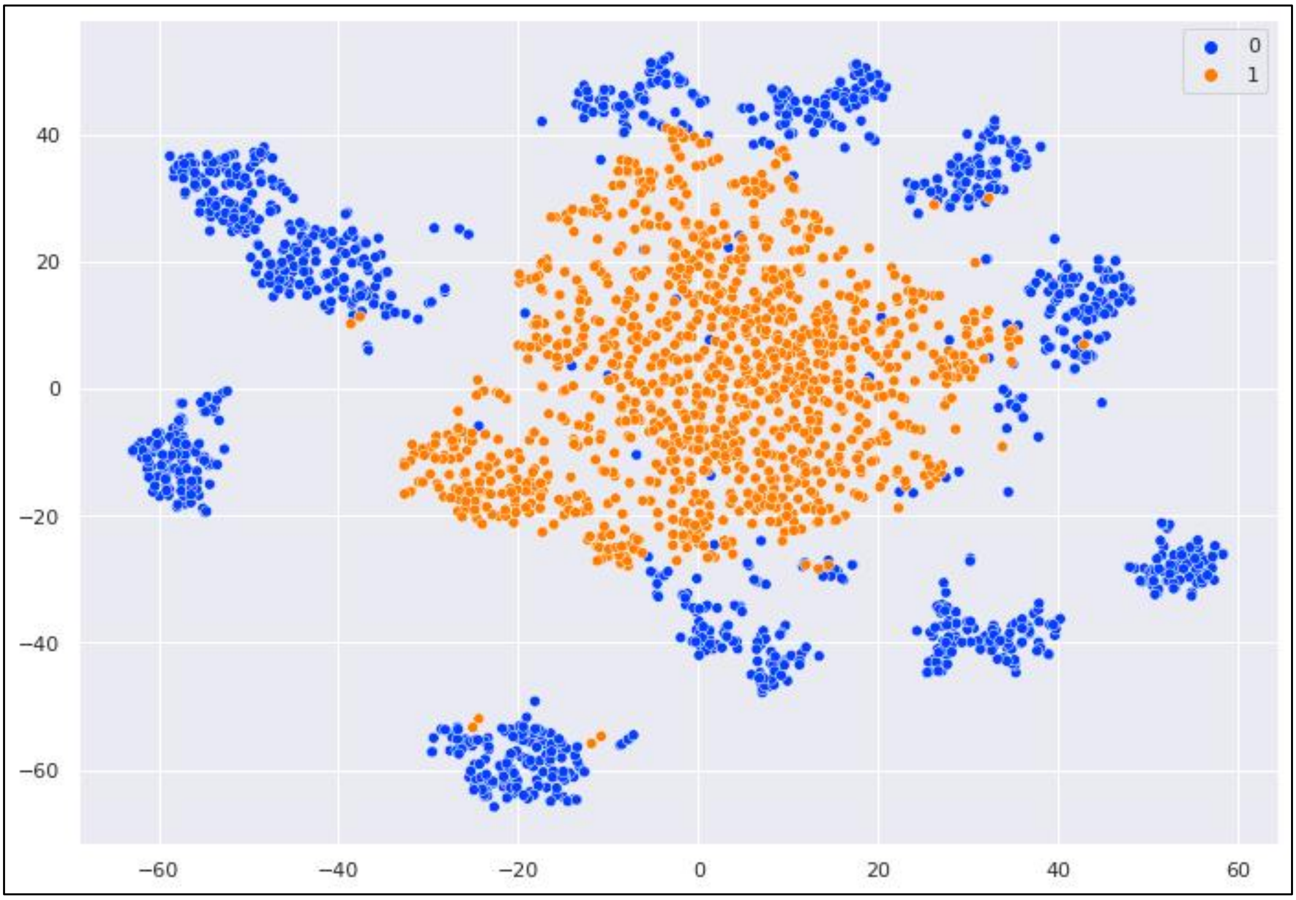
0
1
2
3
4
5
6
7
8
9
30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30
40



0
1
30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30
40

(USPS → SVHN)


0
1
2
3
4
5
6
7
8
9
40
20
0
−20
−40
−60
−60 −40 −20 0 20 40 60


0
1
40
20
0
−20
−40
−60
−60 −40 −20 0 20 40 60

3. <u>Describe the implementation details of your model and discuss what you' ve observed and learned from implementing DANN.</u>

Implementation details

DANN model(image size = 28)

```
DANN(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 28, kernel_size=(4, 4), stride=(1, 1))
    (1): BatchNorm2d(28, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Conv2d(28, 56, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (5): BatchNorm2d(56, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.2, inplace=True)
    (7): Dropout(p=0.5, inplace=False)
    (8): Conv2d(56, 112, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (9): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Dropout(p=0.5, inplace=False)
    (12): Conv2d(112, 224, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (13): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): LeakyReLU(negative_slope=0.2, inplace=True)
    (15): Dropout(p=0.5, inplace=False)
    (16): Conv2d(224, 448, kernel_size=(4, 4), stride=(1, 1))
    (17): BatchNorm2d(448, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): LeakyReLU(negative_slope=0.2, inplace=True)
    (19): Dropout(p=0.5, inplace=False)
  )
  (classes_classifier): Sequential(
    (0): Linear(in_features=448, out_features=224, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=224, out_features=10, bias=True)
  )
  (domain_classifier): Sequential(
    (0): Linear(in_features=448, out_features=224, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=224, out_features=112, bias=True)
    (3): LeakyReLU(negative_slope=0.2)
    (4): Linear(in_features=112, out_features=1, bias=True)
    (5): Sigmoid()
```

```
    )
  )
```

DANN model(image size = 64)

```
  DANN_64x(
    (feature_extractor): Sequential(
      (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Dropout2d(p=0.5, inplace=False)
      (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (6): ReLU(inplace=True)
      (7): Dropout2d(p=0.5, inplace=False)
      (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (10): ReLU(inplace=True)
      (11): Dropout2d(p=0.5, inplace=False)
      (12): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (13): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (14): ReLU(inplace=True)
      (15): Dropout2d(p=0.5, inplace=False)
      (16): Conv2d(512, 1024, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (17): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (18): ReLU(inplace=True)
      (19): Dropout2d(p=0.5, inplace=False)
      (20): Conv2d(1024, 2048, kernel_size=(4, 4), stride=(1, 1))
      (21): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (22): ReLU(inplace=True)
    )
    (classes_classifier): Sequential(
      (0): Linear(in_features=2048, out_features=1024, bias=True)
      (1): ReLU(inplace=True)
      (2): Linear(in_features=1024, out_features=512, bias=True)
      (3): ReLU(inplace=True)
      (4): Linear(in_features=512, out_features=256, bias=True)
      (5): ReLU(inplace=True)
      (6): Linear(in_features=256, out_features=128, bias=True)
      (7): ReLU(inplace=True)
      (8): Linear(in_features=128, out_features=10, bias=True)
    )
    (domain_classifier): Sequential(
      (0): Linear(in_features=2048, out_features=1024, bias=True)
```

```
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=1024, out_features=512, bias=True)
    (3): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Linear(in_features=512, out_features=256, bias=True)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Linear(in_features=256, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
## Hyper parameters ##
Image size(original) : 28
Image size(64x) : 64
Channel size : 3
Number of classes : 10
Optimizer(Adam) Learning rate : 1e-3
Optimizer(Adam) Weight Decay : 3e-4
Training epochs : 100
Batch size : 128
######################

## data augmentation ##
transform = transforms.Compose([
    transforms.Resize((image_size,image_size)),
    transforms.ToTensor(),
    transforms.Normalize([0. 5], [0.5])
 ])
######################

## other details ##
# during training
1. Feature Extractor 與 Domain Classifier 之間的 Gradient Reverse Layer 參考 Reference[6]
的方法實作
2. Loss = Classes Classifier loss /
      + 0.5 * Source Domain Classifier loss + 0.5 * Target Domain Classifier loss
3. 使用 target domain training set 作為 validation data 評估模型好壞與儲存標準
######################
```

Discussion

看 DANN 的架構時，覺得跟 ACGAN 的 Discriminator 很像，感覺只是將 Feature extractor、Classes Classifier(Label Predictor)、Domain Classifier 劃分的更清楚，且多了 Gradient Reverse Layer 希望讓 Feature Extractor 學習到後面所提取的特徵可以混淆 Domain Classifier，同時又能讓 Classes Classifier 預測的準。

實作的過程起初是非常順利的，因為前兩個 Scenario 都是訓練資料夠多且 Domain 差異性不會太大，因此模型一架好就能過 baseline，但跑第三個 Scenario 時就碰到瓶頸了，因為訓練資料只有數千筆卻要預測將近三萬筆的資料，且 USPS 與 SVHN 的差異性也相對前兩個情境大，透過 TSNE 視覺化結果能看出，模型並沒辦法有效的混淆這兩個 Domain 的 data，因此做了各種模型架構、參數的 fine tune，但 Accuracy 極限只能到 23%，最後不得已便為了這個 Scenario 再建一顆 image size 為 64 的 Model，並將 input image resize 到 64 做 training，由於 image size 變大，模型架構變得更複雜、參數更多，雖然還是沒能有效的混淆兩個 Domain，但特徵提取器提取出來的特徵能更有效的辨識類別，因此 performance 有些許進步。

透過這次實驗過程對 DANN 這種 UDA model 有了更深的了解，也從中學習到訓練資料量與 Domain 差異對於訓練 UDA 模型的影響有多大，如何 Fine tune model 與解決訓練遇到的問題比起建構好模型來得困難許多，若能更有效的混淆不同 Domain 的 data，那模型的表現一定能有明顯的進步。

Bonus

1. Accuracy record

	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
Original model	85.70%	59.74%	34.11%
Improved model	87.64%	62.01%	57.33%

2. Briefly describe implementation details of your model and discuss what you've observed and learned from implementing your improved UDA model.

Implementation details

模型架構、參數、訓練過程皆不變，三個 improved model 都加上 data augmentation

```
############## Data augmentation ##############
transform = transforms.Compose([
    transforms.Resize((img_size,img_size)),
    transforms.AutoAugment(AutoAugmentPolicy.SVHN),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])
MNIST-M → USPS:
    Both two training data used it
SVHN → MNIST-M:
    Only MNIST-M training data used it
USPS → SVHN:
    Only USPS training data used it
###############################################
```

Discussion

在實作 original model 時就有試過數種 transformation，但效果都不是很好，後來在和同學討論時，才知道 torchvision.transforms 內建有個 AutoAugment()的方法，其中有針對數個 Dataset 客製化的 Augmentations，仔細看過官方文件後，發現 **AutoAugmentPolicy.SVHN** 可以讓圖片轉換得像是 SVHN 的風格 (如下圖[1])，因為 SVHN 的圖片幾乎都是數字不在中心，

或是有歪斜、雜訊的問題。實際在 Scenario (USPS → SVHN)的 USPS training data 加上這個 augmentation 時，Performance 上升非常多 (34.11%→57.33%)。

因此便嘗試在(SVHN → MNIST-M)的 MNIST-M training data 加上、(MNIST-M → USPS)兩個 Domain 的 training data 也加上，希望藉此讓模型學習時將 Training data 轉換成相近的風格。在不改變模型架構、參數及其他 Training detail 的情況下做實驗，結果也證實以此方法在一定程度上能讓 Training data 改善 Domain 差異性過大導致模型學習不易的問題。

透過實作這個 improved 版本，讓我了解到 DANN 在 Domain 之間差異大時雖然訓練難度會增加，但透過 data augmentation 等方法，嘗試去減少 Domain 間的差異，訓練出來的模型 Performance 也能達到一定程度的進步。

圖 (1)

Reference:

DCGAN paper[1]

10 Lessons I Learned Training GANs for one Year[2]

ACGAN paper[3]

Understanding ACGANs with code[PyTorch][4]

DANN paper[5]

https://github.com/fungtion/DANN[6]

Torchvision.transforms[7]

Collaborators:

M11015Q12 黃柏翰