Homework #2 Report

Deep Learning for Computer Vision

資工碩一　張凱庭　R10922178

Problem 1

1. GAN implementation

Generator architecture

```
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 architecture

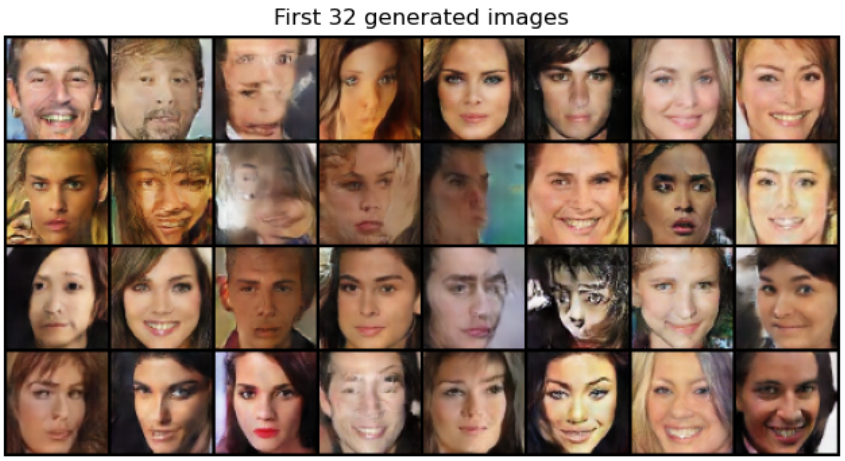
```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05,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

Hyperparameters	
Epochs	100
Optimizer	Adam

Learning rate	0.0002
Batch size	64
Noise vector dimension	100x1x1

In this problem the DCGAN architecture was implemented. The generator mainly contain 4 ConvTranspose2d block, which will take in a $z = (100, 1, 1)$ noise vector to generate images. The output from the final tangent layer range from $(-1, 1)$, this output was then added by 1 and multiplied by 122.5 result in range from $(0, 255)$. The discriminator also mainly contain 4 Conv2d block, which will take in a $(3, 28, 28)$ images. The output from the final sigmoid layer range from $(0, 1)$ as this is a binary classification, to discriminate between real and fake images.

2. **First 32 generated images**



3. **Fréchet inception distance (FID) and Inception score**

Metric	
Fréchet inception distance	26.206736355457565
Inception score	2.1085

4. **Discuss what you've observed and learned from implementing GAN.**
   a. Different random seeds could have a huge impact on the performance. For reproducibility I just fixed the random seed and saved the noise sample for reproduction.
   b. As ganhacks suggested, adjust the objective function to maximize $log(D(G(Z)))$ could avoid early vanishing gradients.
   c. During training it is hard to keep track of 2 metrics(FID and IS), so I just monitor the IS score for choosing epochs.

d. After the first epoch the generator can already create an image with a rough face shape, and after 3500 iterations the face is clear enough.

Problem 2

1. ACGAN implementation:

Generator architecture

```
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=(1, 1), stride=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Discriminator architecture

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), 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=(2, 2), 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)
  )
  (adv): Sequential(
    (0): Linear(in_features=8192, out_features=1, bias=True)
    (1): Sigmoid()
  )
  (aux): Sequential(
    (0): Linear(in_features=8192, out_features=10, bias=True)
    (1): Softmax(dim=1)
  )
)
```

Hyperparameters

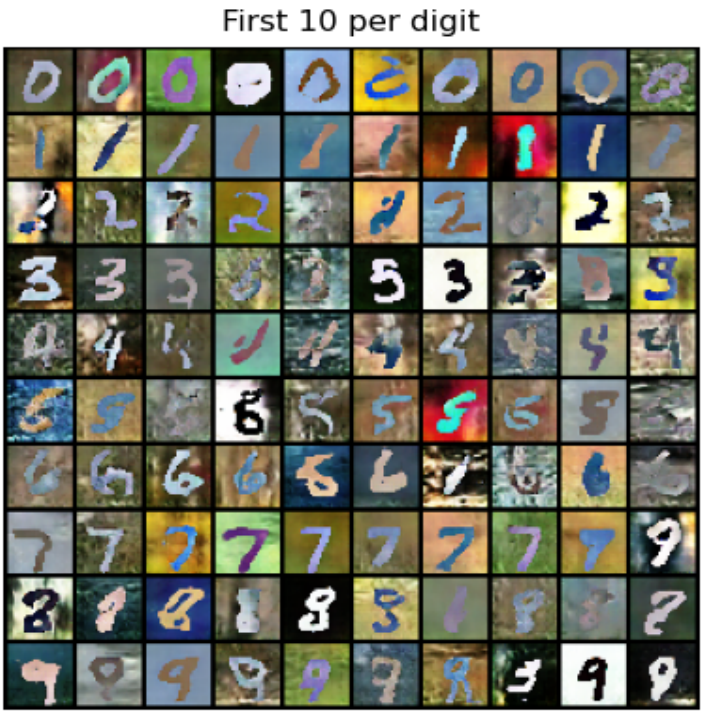
Epochs	42
Optimizer	Adam
Learning rate	0.0002
Batch size	125
Noise vector dimension	100x1x1

The model architecture is pretty similar to DCGAN, except that the discriminator has 2 outputs. The output from the adversarial layer (adv) is to detect if the image is real or fake as the DCGAN model. Another output from the auxiliary layer (aux) is to produce the image probability distribution over the class labels. Here the noise vector is a $(100,\ 1,\ 1)$ dimension vector, the first 10 dimension was a one-hot vector which denoted the class of the vector. In this way we expect the generator to create images according to the class label.

2. **Accuracy**

Accuracy	0.875

3. **10 images for each digit (0-9)**



Problem 3

1. **Accuracy**

Domain	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
Source only	0.6393	0.4604	0.2219
DANN	0.8241	0.5078	0.3421
Target only	0.9606	0.9791	0.9174

2. Domain Adaptation Implementation

Network architecture

```
DANN(
  (feature): Sequential(
    (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 48, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): ReLU(inplace=True)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=768, out_features=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=100, bias=True)
    (4): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=100, out_features=10, bias=True)
  )
  (discriminator): Sequential(
    (0): Linear(in_features=768, 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)
  )
)
```

In this problem the DANN model was implemented, and the model is trained with source domain and target domain. The source data go through all 3 layers, and the target data only go through the feature and discriminator layer. Our goal is to confuse the model with different domains, to do that the gradient reversal layer was applied before the feature goes through the discriminator. During training the model is first updated with source batch then updated with target batch.Since the 2 different domains don't have the exact same batch number, I just cycle through the shorter one, so we can make use of all the data from the larger domain.

Hyperparameters:

Domain	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN

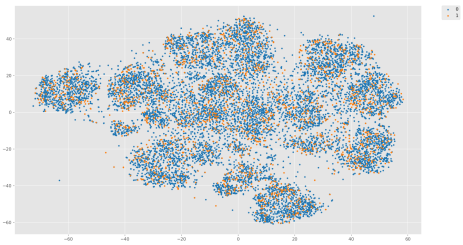
Epoch	18	7	28
Optimizer	SGD	SGD	SGD
Lr	0.001	0.001	0.001
Batch size	64	64	64

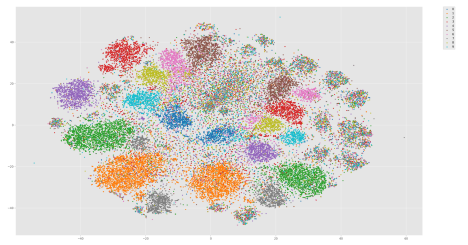
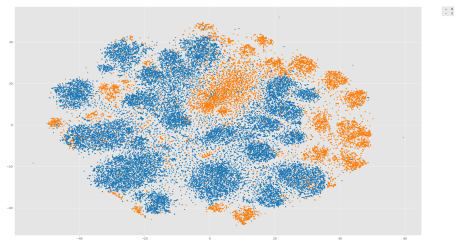
Data preprocessing:

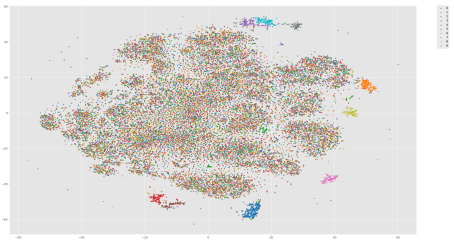
	MNIST-M → USPS		SVHN → MNIST-M		USPS → SVHN	
	MNIST-M	USPS	SVHN	MNIST-M	USPS	SVHN
Normalization	o	o	o	o	o	o
Colorjitter	x	x	x	x	o	x

Normalization was applied to all 3 dataset and using $mean = (0.5, 0.5, 0.5)$ and $std = (0.5, 0.5, 0.5)$ as normalization parameters. Colorjitter was only applied to USPS as a source domain, since it only got 7,291 pictures which is rather small than SVHN and MNIST-M. The parameters for colorjitter was: $(brightness = 0.5, contrast = 0.5, saturation = 0.5)$

3. **t-SNE visualization**

MNIST-M → USPS	
	
3.1(a)different digit classes 0-9	3.1(b)different domain

SVHN → MNIST-M	
	
3.2(a)different digit classes 0-9	3.2(b)different domain

USPS → SVHN	
	
3.3(a)different digit classes 0-9	3.3(b)different domain

4. **Discover**
    a. When implementing domain adaptation, the size of the source dataset really matters.Take "**USPS → SVHN"** for example, the source dataset (USPS) only has 7291 images while the target dataset(SVHN) has 73,257 images. The accuracy of this pair is 0.26 which is significantly lower than the other two groups(without any data augmentation).
    b. To deal with the rather small dataset problem, I applied "transform.colorjitter" to the **USPS** dataset while training the "**USPS → SVHN"**. The accuracy improved from 0.26 to 0.34 and successfully passed the baseline. (3 groups are all trained with the same architecture)
    c. The network architecture of source-only and target-only is exactly the same as the DANN except that the discriminator layer is removed. Using this architecture, all 3 domains have accuracy over 90%.
    d. From the t-SNE visualization we can tell that the better the domain adaptation was done, the features extracted from the model should be more mixed. As Figure 3.1(b), the accuracy is 0.84 and all the source and target points were well mixed. But as Figure 3.3(c) shows, the accuracy is 0.34 and the target domain is separated from the source domain.

Reference

1. DCGAN: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
2. GAN hacks: soumith/ganhacks: starter from "How to Train a GAN?"
3. ACGAN implementation:
    3.1. clvrai/ACGAN-PyTorch
    3.2. kimhc6028/acgan-pytorch
4. DANN implementation:
    4.1. wogong/pytorch-dann
    4.2. CuthbertCai/pytorch_DANN