2021 DLCV
HW2
R09942171 電信所碩一 黃繼綸

Problem 1: GAN (no collaborators)

1. Ref : https://github.com/heykeetae/Self-Attention-GAN

I used the Self-Attention GAN to solve this question. The implementation details are as follow :

Training steps : 500000 steps

Learning rate : lr_G : 0.0001, lr_D : 0.0004

Data augmentation : RandomHorizontalFilp

Optimizer : Adam with beta(0.0, 0.9)(both G and D)

Batch_size : 32

Latent_dim : 128

Loss Function:

Fisrt, I used hinge loss to train a pre-trained model. Next, I changed the loss function to WGAN-GP loss to improve the performance.

The model architectures are shown below:

Generator：

```
Generator_SAGAN(
  (l4): Sequential(
    (0): SpectralNorm(
      (module): 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()
  )
  (l1): Sequential(
    (0): SpectralNorm(
      (module): 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()
  )
  (l2): Sequential(
    (0): SpectralNorm(
      (module): 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()
  )
  (l3): Sequential(
    (0): SpectralNorm(
      (module): 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()
  )
  (last): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): Tanh()
  )
  (attn1): Self_Attn(
    (query_conv): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
  (attn2): Self_Attn(
    (query_conv): Conv2d(64, 8, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(64, 8, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
)
```

Discriminator

```
Discriminator_SAGAN(
  (l4): Sequential(
    (0): SpectralNorm(
      (module): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): LeakyReLU(negative_slope=0.1)
  )
  (l1): Sequential(
    (0): SpectralNorm(
      (module): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): LeakyReLU(negative_slope=0.1)
  )
  (l2): Sequential(
    (0): SpectralNorm(
      (module): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): LeakyReLU(negative_slope=0.1)
  )
  (l3): Sequential(
    (0): SpectralNorm(
      (module): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (1): LeakyReLU(negative_slope=0.1)
  )
  (last): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
  )
  (attn1): Self_Attn(
    (query_conv): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
  (attn2): Self_Attn(
    (query_conv): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
)
```

2. (Random Seed : 7414)

3.

4.

FID	IS
25.1	2.137

5. I used lots of GAN architectures to try to solve this question, such as DCGAN, StyleGAN, StyleGAN2, WGAN-GP and SAGAN.
First, I used **DCGAN** to solve this problem. But even converge, DCGAN can not generate realistic images.
Secondly, I combined the **DCGAN with WGAN-GP loss**, and the result improved a lot. But the result still cannot pass the baseline.
Thus, I tried to use **StyleGAN and StyleGAN2** which are the recently popular models to solve this problem.
But the result still cannot pass the baseline, too. Finally, I surveyed related papers, and I found the images generated by **SAGAN** are amazing. Therefore, I implemented this model, and used it to solve the problem.
In this problem, I found that hyperparameters tuning is very important to train a good GAN. Even if I make a slight tuning during my implementation, it will cause a significant change in performance. Therefore, **the observation of the training metrics and sample results** are the key points when training GAN.

	FID	IS
DCGAN	35.3	1.79
StyleGAN	32	1.87
StyleGAN2	23.1	1.98
WGAN-GP	32.3	1.83
SAGAN	**25.1**	**2.137**

Problem2 : ACGAN (no collaborators)

1. Ref : https://github.com/eriklindernoren/PyTorch-GAN

In the model design, I embedded the num_labels into a latent space which is 128 dimensions.

In the training phase, I first randomly generated the labels which is used to fit noise into the model together.

During training netG, the generated labels and noises are fitted into the generator together. Discriminator should both classify the classes and determine whether the generated image is real or not.

During training netD, there are two steps. The first step is to fit the real image and the real label together into the discriminator. The second step is to fit the fake images generated by netG and the fake labels together into the discriminator.

n_epochs : 200

lr_decay : 100 (Start to linearly decay lr)

lr : 0.0002

latent_dim : 100

random_seed : 1004

optimizer : Adam with beta(0.5, 0.999)

The model architectures are shown below:

Generator :

```
Generator(
  (label_emb): Embedding(10, 100)
  (l1): Sequential(
    (0): Linear(in_features=100, out_features=6272, bias=True)
  )
  (conv_blocks): Sequential(
    (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (1): Upsample(scale_factor=2.0, mode=nearest)
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Upsample(scale_factor=2.0, mode=nearest)
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): Tanh()
  )
)
```

Discriminator :

```
Discriminator(
  (conv_blocks): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Dropout2d(p=0.25, inplace=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Dropout2d(p=0.25, inplace=False)
    (6): BatchNorm2d(32, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Dropout2d(p=0.25, inplace=False)
    (10): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (11): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (12): LeakyReLU(negative_slope=0.2, inplace=True)
    (13): Dropout2d(p=0.25, inplace=False)
    (14): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
  )
  (adv_layer): Sequential(
    (0): Linear(in_features=512, out_features=1, bias=True)
  )
  (aux_layer): Sequential(
    (0): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

2.

3.

4.

Accuracy	99.80%

5.

Problem3 : DANN (no collaborators)

1.

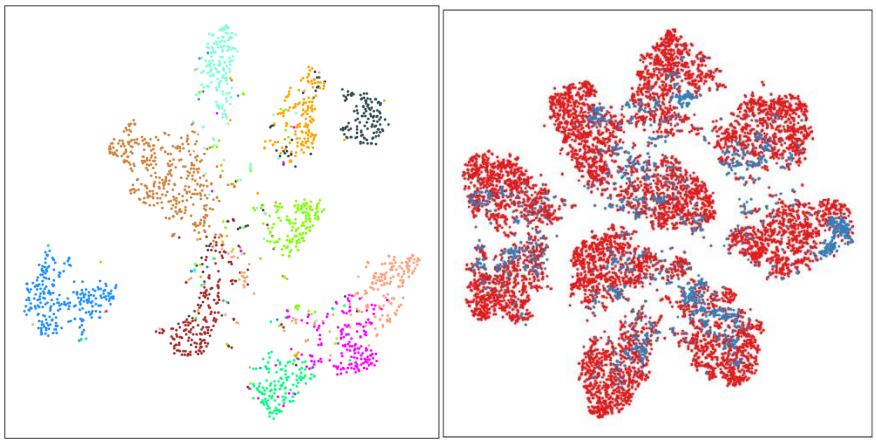
2.

3.

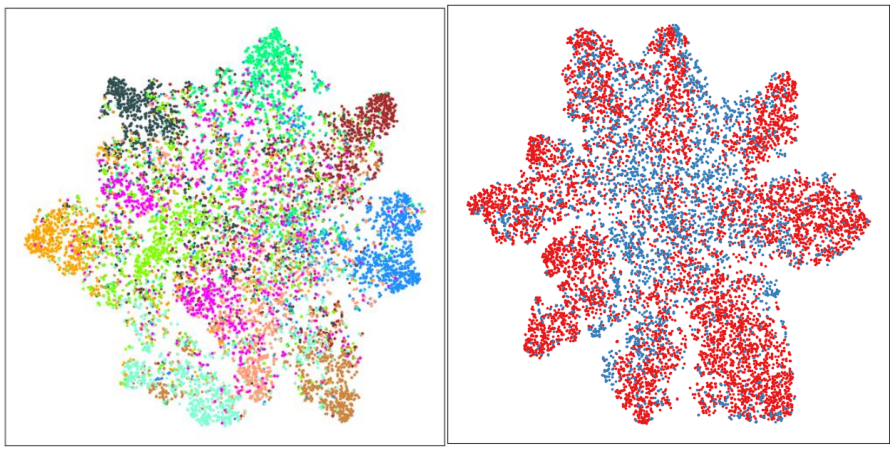
	MNIST-M -> USPS	SVHN -> MNIST-M	USPS -> SVHN
Trainined on Soruce	74.24%	42.89%	19.00%
Adaptation(DANN)	85.20%	47.57%	29.04%
Trained on target	98.25%	90.44%	96.81%

4. T-SNE results

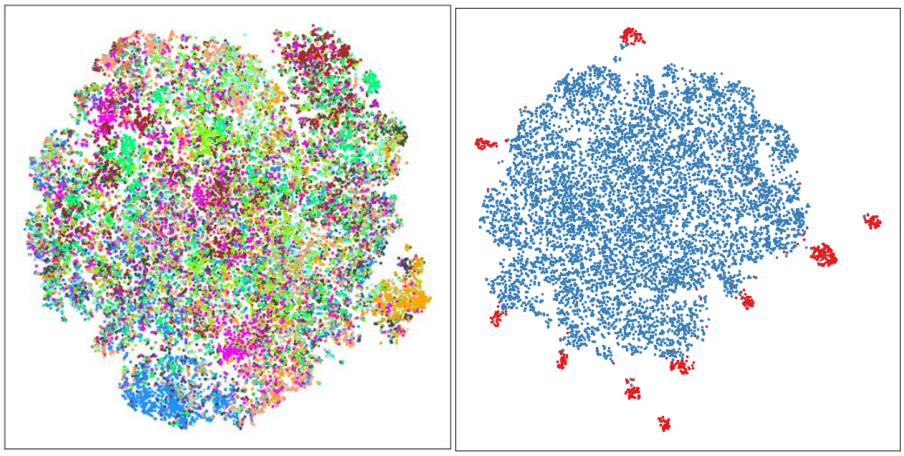
MNIST-M -> USPS



SVHN -> MNIST-M



USPS -> SVHN



5. Ref : https://github.com/fungtion/DANN

In my observation, if the source domain is not similar to the target domain, the result of domain adaptation will be bad(e.g. USPS -> SVHN). However, if the source domain is a little similar to the target domain, the result of domain adaptation will be acceptable(e.g. MNIST-M -> USPS). **In training DANN, The accuracy of the domain classifier is an important metric**. If the accuracy of the domain classifier is always 100%, you should check whether my code may be wrong or not.

Implementation details:

n_epcohs : 200

lr_decay : 50 (lr * 0.412 every 50 epochs)

batch_size : 256

lr : 0.0001

loss function : Crossentropy loss

optimizer : Adam

Bonus : Adversarial Discriminative Domain Adaptation(ADDA) (no collaborators)

1.

	MNIST-M -> USPS	SVHN -> MNIST-M	USPS -> SVHN
DANN	85.20%	47.57%	29.04%
ADDA	89.89%	50.76%	30.01%

2. Ref : https://github.com/corenel/pytorch-adda

The learning rate of the target encoder and the critic are very sensitive to the result. After trial and error, the best result I got were that the learning rate of the **target encoder** and the **critic** are **0.05** and **0.000007**, respectively. In addition, **the ability of the source encoder and classifier are also very important**. During training the target encoder, you should ensure that your source encoder and classifier are powerful enough. Otherwise, whatever you tuned the hyperparameters, you will get a bad result.

Implementation details :

Source_training_epochs : 100

Domain_training_epochs : 150

lr_tartget_encoder : 0.05

lr_critic : 0.000007

Optimizer : Adam with beta(0.5, 0.9) both target_encoder and critic

Batch_size : 64