

DLCV HW2 Report

R11921008 羅恩至

Problem1

(5%) Please print the model architecture of method A and B.

Model A (DCGAN)

Generator:

```
Generator(  
  (11): Sequential(  
    (0): Linear(in_features=100, out_features=8192, bias=False)  
    (1): BatchNorm1d(8192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (2): ReLU()  
  )  
  (12_5): Sequential(  
    (0): Sequential(  
      (0): ConvTranspose2d(512, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)  
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (2): ReLU()  
    )  
    (1): Sequential(  
      (0): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)  
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (2): ReLU()  
    )  
    (2): Sequential(  
      (0): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)  
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (2): ReLU()  
    )  
    (3): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1))  
    (4): Tanh()  
  )  
)
```

Fig. 1. The generator architecture of Model A

Discriminator:

```
Discriminator(  
  (1s): Sequential(  
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
    (1): LeakyReLU(negative_slope=0.2)  
    (2): Sequential(  
      (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (2): LeakyReLU(negative_slope=0.2)  
    )  
    (3): Sequential(  
      (0): Conv2d(128, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (2): LeakyReLU(negative_slope=0.2)  
    )  
    (4): Sequential(  
      (0): Conv2d(256, 512, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (2): LeakyReLU(negative_slope=0.2)  
    )  
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))  
    (6): Sigmoid()  
  )  
)
```

Fig. 2. The discriminator architecture of Model A

Model B (SNGAN):

Generator:

```
Generator(  
  (11): Sequential(  
    (0): Linear(in_features=150, out_features=8192, bias=False)  
    (1): BatchNorm1d(8192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (2): ReLU()  
  )  
  (12_5): Sequential(  
    (0): Sequential(  
      (0): Upsample(scale_factor=2.0, mode=nearest)  
      (1): ReflectionPad2d((2, 2, 2, 2))  
      (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (3): ReLU()  
    )  
    (1): Sequential(  
      (0): Upsample(scale_factor=2.0, mode=nearest)  
      (1): ReflectionPad2d((2, 2, 2, 2))  
      (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (3): ReLU()  
    )  
    (2): Sequential(  
      (0): Upsample(scale_factor=2.0, mode=nearest)  
      (1): ReflectionPad2d((2, 2, 2, 2))  
      (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      (3): ReLU()  
    )  
    (3): Upsample(scale_factor=2.0, mode=nearest)  
    (4): ReflectionPad2d((2, 2, 2, 2))  
    (5): Conv2d(64, 3, kernel_size=(5, 5), stride=(1, 1))  
    (6): Tanh()  
  )  
)
```

Fig. 3. The generator architecture of Model B

Discriminator:

```
Discriminator(  
  (1s): Sequential(  
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
    (1): LeakyReLU(negative_slope=0.1)  
    (2): Sequential(  
      (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
      (1): LeakyReLU(negative_slope=0.1)  
    )  
    (3): Sequential(  
      (0): Conv2d(128, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
      (1): LeakyReLU(negative_slope=0.1)  
    )  
    (4): Sequential(  
      (0): Conv2d(256, 512, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))  
      (1): LeakyReLU(negative_slope=0.1)  
    )  
    (5): Conv2d(512, 8, kernel_size=(1, 1), stride=(4, 4))  
    (6): Sigmoid()  
  )  
)
```

Fig. 4. The discriminator architecture of Model B

(5%) Please show the first 32 generated images of both method A and B then discuss the difference between method A and B.



Fig. 5. First 32 generated images of Model A (FID = 33.58)



Fig. 6. First 32 generated images of Model B (FID = 26.15)

From the above 2 figures, it shows that both models (DCGAN and SNGAN) generate good images. The difference is that the distorted face generated from model A seems to be more distorted than the distorted from model B (e.g. image at row 2 column 2 in model A is worse than image at row 3 column 1 form in B), and the images generated by model A seems slightly darker, including some dark image like the image at row 4 column 8. As in model B, there are some image with white background (like some image in row 2), which covered some face contours. Overall, the images generated from model B is better.

(5%) Please discuss what you've observed and learned from implementing GAN.

(1) Hyperparameters

The hyperparameters are shown below:

Image size	3*64*64
Batch size	128
Generator Optimizer	Adam
Discriminator Optimizer	Adam
Loss function	BinaryCrossEntropyLoss
Learning rate	0.0001
Epoch	270

Table. 1. The implementation details of SNGAN

(2) Observation

While training GAN, I had observed that if we train this task by just using baseline model DCGAN, after trained about 200 to 300 epochs, it could have a fine FID and accuracy output, which is 33.58 and 97.1 % face accuracy. As in SNGAN, I just implemented spectral norm on each layer in discriminator, other architectures and hyperparameters are same as DCGAN, and the FID result improved slightly to about 26, but face accuracy decreased about 94 %.

I had tried WGAN as well. But during the training step, it seemed that the model was not able to generate good faces, since the generated were blurry and the facial features of many images were indistinguishable, even trained for many epochs. So the FID result of WGAN was much worse than DCGAN and SNGAN, which is over 100.

Thus in conclusion, SNGAN seems to be a better method on this task.

Problem 2

(5%) Please show your model architecture and describe your implementation details.

Model Architecture:

```
DDPM(  
  (nn_model): ContextUnet(  
    (init_conv): ResidualConvBlock(  
      (conv1): Sequential(  
        (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (2): GELU(approximate=none)  
      )  
      (conv2): Sequential(  
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (2): GELU(approximate=none)  
      )  
    )  
  )  
)
```

```

(down1): UnetDown(
  (model): Sequential(
    (0): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
)
(down2): UnetDown(
  (model): Sequential(
    (0): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
)

```

```

(to_vec): Sequential(
  (0): AvgPool2d(kernel_size=7, stride=7, padding=0)
  (1): GELU(approximate=none)
)
(timeembed1): EmbedFC(
  (model): Sequential(
    (0): Linear(in_features=1, out_features=256, bias=True)
    (1): GELU(approximate=none)
    (2): Linear(in_features=256, out_features=256, bias=True)
  )
)
(timeembed2): EmbedFC(
  (model): Sequential(
    (0): Linear(in_features=1, out_features=128, bias=True)
    (1): GELU(approximate=none)
    (2): Linear(in_features=128, out_features=128, bias=True)
  )
)
(contextembed1): EmbedFC(
  (model): Sequential(
    (0): Linear(in_features=10, out_features=256, bias=True)
    (1): GELU(approximate=none)
    (2): Linear(in_features=256, out_features=256, bias=True)
  )
)
(contextembed2): EmbedFC(
  (model): Sequential(
    (0): Linear(in_features=10, out_features=128, bias=True)
    (1): GELU(approximate=none)
    (2): Linear(in_features=128, out_features=128, bias=True)
  )
)

```

```

(up0): Sequential(
  (0): ConvTranspose2d(256, 256, kernel_size=(7, 7), stride=(7, 7))
  (1): GroupNorm(8, 256, eps=1e-05, affine=True)
  (2): ReLU()
)
(up1): UnetUp(
  (model): Sequential(
    (0): ConvTranspose2d(512, 128, kernel_size=(2, 2), stride=(2, 2))
    (1): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
    (2): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
  )
)
(up2): UnetUp(
  (model): Sequential(
    (0): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
    (1): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
    (2): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): GELU(approximate=none)
      )
    )
  )
)
(out): Sequential(
  (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): GroupNorm(8, 128, eps=1e-05, affine=True)
  (2): ReLU()
  (3): Conv2d(128, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
(loss_mse): MSELoss()
)

```

Fig. 7. Model architecture of diffusion model

Implementation details:

Image size	3*28*28
Batch size	128
Model architecture	U-Net
Number of features in U-Net	128
Generator Optimizer	Adam
Loss function	MSELoss
Learning rate	0.0001
Epoch	30
Total time step	400

Table. 2. The implementation details of diffusion model

In diffusion model, the backbone model is U-Net, and apply MSELoss between added noise and predicted noise as the criterion loss.

(5%) Please show 10 generated images for each digit (0-9) in your report. You can put all 100 outputs in one image with columns indicating different noise inputs and rows indicating different digits.



Fig. 8. 10 generated images for each digit (The columns indicate different digits, and the rows indicate different noise inputs)

(5%) Visualize total six images in the reverse process of the first “0” in your grid in (2) with different time steps.

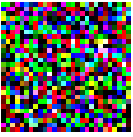
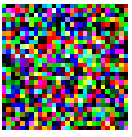
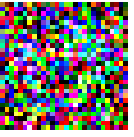
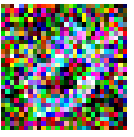


Time step	0	100	200	300	380	400
Image						

Table. 3. Six images of first “0” with different time steps

(5%) Please discuss what you’ve observed and learned from implementing conditional diffusion model.

Since diffusion models are a new class of generative model, so I had done some survey on github repos and websites before solving this problem. In this problem, since the dataset we use are mnistm, the image size is not large, so using simpler U-Net model is enough for this task. Finally I set the number of features into U-Net to 128. During training, the loss drops fast to about 0.06 for a few epochs, and it took about 2~3 minutes to finish training one epoch. As in evaluation step, according to the goal, the model had to generate 1000 images (100 images for each digit in this task) within 15 minutes, so the time step for diffusing and reversing images must not be too large to run overtime. I finally set the time step to 400 to let 1000 images generated in time (about 12~13 minutes) and the generated images quality were also well enough to pass the strong baseline.

Problem 3

(5%) Please create and fill the table with the following format in your report:

	Mnistm to SVHN	Mnistm to USPS
Trained on Source	0.3613	0.7819
Adaptation(DANN)	0.6627	0.9625
Trained on Target	0.9279	0.9901

Table. 4. The upper/lower bound and DANN results

(8%) Please visualize the latent space of DANN by mapping the validation images to 2D space with t-SNE. For each scenario, you need to plot two figures which are colored by digit class (0-9) and by domain, respectively.



Fig. 9. t-SNE plot of mnistm to svhn by class (left)

Fig. 10. t-SNE plot of mnistm to svhn by domain (right)



Fig. 11. t-SNE plot of mnistm to usps by class (left)

Fig. 12. t-SNE plot of mnistm to usps by domain (right)

From the t-SNE plots above, we can see that in mnistm to svhn, the domain of the two does not have large overlapped area, indicating that the domain gap still exists. And in mnistm to usps, the datas from two domain mixed better. Different classes separated clearly in both datasets.

(10%) Please describe the implementation details of your model and discuss what you've observed and learned from implementing DANN.

1. Data Augmentation

Since the domain gap is larger between mnistm and svhn, so in this task I implemented data augmentation on this task. I tried SVHN AutoAugment policy, small degree of rotation (5 or 10 degree) and RandomAffine to make the training dataset larger and increase the data diversity.

As in the task of training mnistm to usps, since the domain gap is closer and is easier to train, so I just changed the dimension of usps dataset from 1 to 3 via `Transform.GrayScale(3)` method.

2. Hyperparameters

The hyperparameters are shown below (all same on 2 tasks)

Image size	3*32*32
Batch size	128
Optimizer	Adam
Domain Classifier Loss Criterion	BCEWithLogitsLoss
Feature Extractor and Label Predictor Loss Criterion	CrossEntropyLoss
Learning rate	0.001
Lambda	$(2/(1 + e^{-10 * \text{epoch} / \text{total epoch}})) - 1$

Table. 5. The implementation details of DANN

I set the lambda to be dynamic with different epoch on gradient reversal layer, and the result is better than setting lambda to a fixed value.

3. Semi-Supervised learning

In this problem, I also implemented semi-supervising to make the model stronger. I treated the target training data as unlabeled training data. As training process goes, if the test accuracy is good enough, then let the label predictor to generate pseudo label on target training dataset. Set the probability threshold to 0.99, and added the images into training set with the probability of the pseudo label over that threshold. With semi-supervised learning applied, in task mnistm to svhn, the validation accuracy increased from 0.54 to 0.66, and also in task mnistm to usps, the validation accuracy increased from 0.87 to 0.96. It concluded that semi-supervising is quite effective in both tasks. (The result is shown in Table. 6. below)

Task	Without semi-supervising	With semi-supervising
Mnistm to svhn	0.5488	0.6627
Mnistm to usps	0.8746	0.9625

Table. 6. Training results with/without semi-supervising used