DLCV HW2 Report

電機四 劉知穎 B07901039

Collaborator: B07303024 李品樺

Problem 1: GAN

1. Describe model architecture and implementation details

```
Generator(
  (l1): Sequential(
    (0): ConvTranspose2d(100, 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(inplace=True)
  )
  (l2_5): Sequential(
    (0): Sequential(
      (0): 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(inplace=True)
    )
    (1): 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)
    )
    (2): Sequential(
      (0): 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(inplace=True)
    )
    (3): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): Tanh()
  )
)
```

Figure 1: DCGAN generator architecture

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

Figure 2: DCGAN discriminator architecture

The model architecture of generator is shown in Figure 1. A 100-dim Gaussian random noise is shaped into (100, 1, 1) to input the generator. After tangent activation function, the output of generator is in the range [-1, 1]. Add the output by 1 and dividing by 2 to make the output in the range [0, 1]. Finally, use torchvision.utils.save_image to unnormalize to the range [0, 255] and save generated images.

The model architecture of discriminator is shown in Figure 2. The input is tensor of shape (3, 64, 64) and the output is a single value in the range [0, 1] indicating whether the input is real or fake.

Both the optimizers of generator and discriminator are Adam with an initial learning rate 0.0002, beta1 0.9, and beta2 0.999. The learning rate is cut half by a

scheduler every 40 epochs. The total number of training epoch is 120. I resize the input images to (64, 64), which is the original size, and normalize them by mean (0.5, 0.5, 0.5) and standard deviation (0.5, 0.5, 0.5). The loss function is binary cross entropy loss.

2. Plot 32 randomly generated images

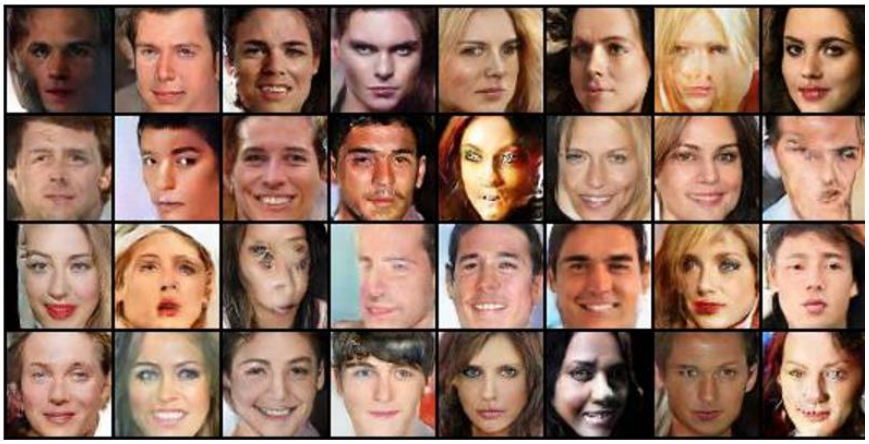


Figure 3: 32 randomly generated images

3. Evaluate 1000 randomly generated images

   FID = 25.3777

   IS = 2.0290

4. Discuss what you've observed and learned from implementing GAN
   a. Using ReLU in the generator and LeakyReLU in the discriminator perform better than using LeakyReLU both in the generator and the discriminator
   b. The generator and the discriminator should have comparable "strength". To let the generator to be stronger, I had tried to use Adam optimizer for the generator and SGD optimizer for the discriminator (a tip suggested in [1]). However, the discriminator failed to predict correct labels. Though the generator loss is lower in this case, it can't generate real-like images.
   c. I think weight initialization and substituting first linear layer of the generator with transpose convolution layer are the two most important tips. After adding these two tips and some subtle optimization modifications, FID score improved from 51.45 to 30.25.
   d. Adding horizontally flip can increase training dataset. However, the quality of generated images, and their FID, IS score have no obvious difference.
   e. Tuning the update times of the generator and the discriminator can make clearer images. In this case, I update the discriminator once then update the generator twice. The FID score improved from 30.25 to 25.377. However, if the generator is updated three times, the discriminator will fail to make reasonable predictions, and thus the quality of generated images will be low.

Problem 2: ACGAN

1. Describe model architecture and implementation details

```
Generator_64(
  (to_embedding): Linear(in_features=10, out_features=20, bias=True)
  (l1): Sequential(
    (0): ConvTranspose2d(120, 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(inplace=True)
  )
  (l2_5): Sequential(
    (0): Sequential(
      (0): 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(inplace=True)
    )
    (1): 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)
    )
    (2): Sequential(
      (0): 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(inplace=True)
    )
    (3): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): Tanh()
  )
)
```

Figure 4: ACGAN generator architecture

```
Discriminator_64(
  (ls): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Sequential(
      (0): Conv2d(64, 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): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(128, 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): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (5): Conv2d(512, 11, kernel_size=(4, 4), stride=(1, 1))
    (6): Sigmoid()
  )
)
```

Figure 5: ACGAN discriminator architecture

The model architecture of generator is shown in Figure 4. A 10-dim one-hot vector and 100-dim noise gaussian random vector shaped into (100, 1, 1) are fed into the generator. A linear layer with input dim 10, output dim 20 converts the one-hot label into 20-dim label embedding. The label embedding is concatenated with the noise vector and fed into the remaining transpose convolution layers. After tangent activation function, the output of generator is in the range [-1, 1]. Add the output by 1 and dividing by 2 to make the output in the range [0, 1].
The model architecture of discriminator is shown in Figure 5. The input is tensor of shape (3, 64, 64) and the output is a single value in the range [0, 1] indicating whether the input is real or fake and a 10-dim vector in the range [0, 1] representing the prediction of class labels.
Both the optimizers of generator and discriminator are Adam with an initial learning rate 0.0001, beta1 0.5, and beta2 0.999. The total number of training epoch is 10. I resize the input images to (64, 64), and normalize them by mean (0.5, 0.5, 0.5) and standard deviation (0.5, 0.5, 0.5). The loss function is binary

cross entropy loss for real/fake prediction and cross entropy loss for class prediction.

4. Evaluation the generated images by digit classifier
   Average accuracy = 0.881
5. Show 100 generated images

Figure 6: 100 randomly generated images

Discussion:

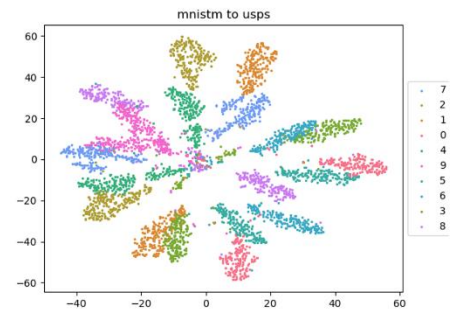
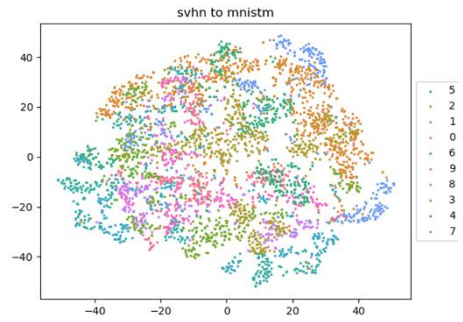
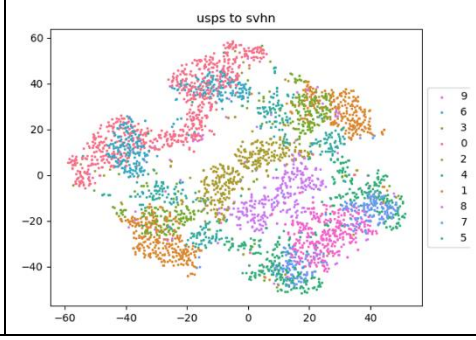
a. It's hard for the model to learn the label information if feeding the label by a scalar (0 to 9). Converting the label into one-hot vector and add a small linear layer to transform label information into a vector is an effective approach.
b. Substituting linear layers other than the label embedding layer into convolution / transpose convolution layers generate clearer images.
c. The stability of training is weak. Though I experimented with different learning rates and generator updating times, the model still collapsed around 15 epochs.

Problem 3: DANN

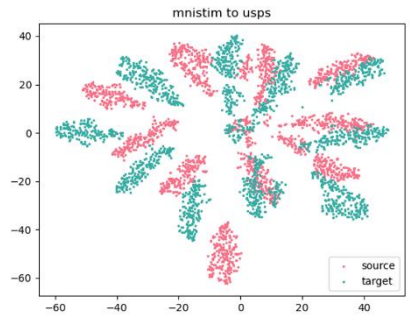
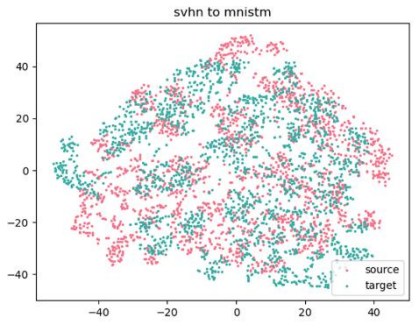
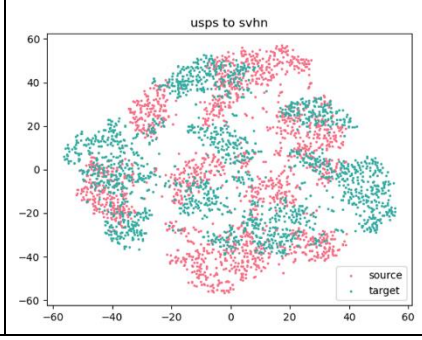
1-3.

	mnistm -> usps	svhn -> mnistm	usps -> svhn
Trained on source	0.8555	0.4848	0.1507
Adaptation	0.9322	0.5161	0.2853
Trained on target	0.9711	0.9835	0.9343

4. Visualize the latent space
    a. Different digit classes 0-9

mnistm -> usps	svhn -> mnistm	usps -> svhn
		

    b. Different domains

mnistm -> usps	svhn -> mnistm	usps -> svhn
		

In the case “mnistm to usps”, different digits have separate clusters, but the cluster of target and source domains aren’t completely overlapping. On the other hand, in other two cases, different digits don’t have obvious clusters, and it’s hard to tell whether the distributions of the same digit of different domains are overlapping.

5. Describe the implementation details of your model and discuss what you’ve observed and learned from implementing DANN

```
DANN_small(
  (feature_extractor): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(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)
    )
    (1): Sequential(
      (0): Conv2d(64, 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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (3): ReLU(inplace=True)
    )
    (2): 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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (3): ReLU(inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, 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): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(1024, 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)
    )
    (5): AdaptiveAvgPool2d(output_size=1)
  )
  (domain_classifier): Sequential(
    (0): Linear(in_features=1024, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=1, bias=True)
    (3): Sigmoid()
  )
  (label_classifier): Sequential(
    (0): Linear(in_features=1024, out_features=1024, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=1024, out_features=1024, bias=True)
    (4): ReLU()
    (5): Linear(in_features=1024, out_features=10, bias=True)
    (6): Sigmoid()
```

Figure 7: Model architecture for “svhn to mnistm” and “mnistm to usps”

```
DANN_svhn1(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(32, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): Dropout2d(p=0.5, inplace=False)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (domain_classifier): Sequential(
    (0): Linear(in_features=2352, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=100, out_features=1, bias=True)
    (4): Sigmoid()
  )
  (label_classifier): Sequential(
    (0): Linear(in_features=2352, out_features=100, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (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()
    (6): Linear(in_features=100, out_features=10, bias=True)
    (7): Sigmoid()
  )
)
```

Figure 8: Model architecture for “usps to svhn”

The model architecture for the cases “svhn to mnistm” and “mnistm to usps” is shown in Figure 7. The input of the model is an image of shape (3, 64, 64), and the output of the model is a scalar indicating the prediction of domain and a 10-dim vector representing the logits of digit classes. After the feature extractor, the latent vector is of dimension 1024. The latent vector is passed through a reversed gradient

layer to get reversed latent vector. The original latent vector is inputted into the label classifier, and the reversed latent vector is inputted into the domain classifier.
The model architecture of the case “usps to svhn” is shown in Figure 8. The input of the model is and image of shape (3, 28, 28). The other details are similar with the model for “svhn to mnistm” and “mnistm to usps”. I trained 4 models with similar architecture as shown in Figure 8 (changing number of convolution kernels, strides, and dimensions of linear layers), and ensemble them by voting their predictions to get the final result.
The optimizer Adam with an initial learning rate 0.0002, beta1 0.9, and beta2 0.999. The total number of training epoch is 20 for “mnistm to usps” and “svhn to mnistm”, and 30 for “usps to svhn”. After observing the learning curve, I evaluated when the model started overfitting. I decided to save the last epoch of training for my final model. I normalize them by mean (0.5, 0.5, 0.5) and standard deviation (0.5, 0.5, 0.5). The loss function is binary cross entropy loss for domain prediction and cross entropy loss for class prediction.
The “mnistm to usps” and “svhn to mnistm” are relatively easy. A strong enough feature extractor trained only on source images can pass baseline on target domain without domain adaptation. After adding domain adaptation, the result further improved. However, the “usps to svhn” case is really hard, because the gap between two domains is big. I found out that the big model used in the previous two cases doesn’t work in this case, and the small feature extractor performs better. In addition, the dropout in feature extractor is important. Without dropout, the accuracy decreases to 0.1839.

Reference

1. https://github.com/soumith/ganhacks