HW2 Image Generation and UDA

GINM11 R11944004 李勝維

Problem 1: GAN

1. Please print the model architecture of method A and B.

Method A, DCGAN

The code is adapted from official pytorch tutorial: [DCGAN Tutorial]

Generator:

```
DCGAN_G(
  (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): LeakyReLU(negative_slope=0.2)
    (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): LeakyReLU(negative_slope=0.2)
    (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): LeakyReLU(negative_slope=0.2)
    (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): LeakyReLU(negative_slope=0.2)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Discriminator:

```
DCGAN_D(
  (net): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2)
    (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)
    (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)
    (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)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Method B, SNGAN

Generator: (identical to method A)

```
DCGAN_G(
  (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): LeakyReLU(negative_slope=0.2)
    (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): LeakyReLU(negative_slope=0.2)
    (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): LeakyReLU(negative_slope=0.2)
    (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): LeakyReLU(negative_slope=0.2)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Discriminator:(almost identical to method A, except spectrum normalization is added to each convolution layer)

```
SNGAN_D(
  (net): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2)
    (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)
    (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)
    (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)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

2. Please show the first 32 generated images of both method A and B then discuss the difference between method A and B.

Method A (FID = 29.50)

Method B (FID = 25.99)

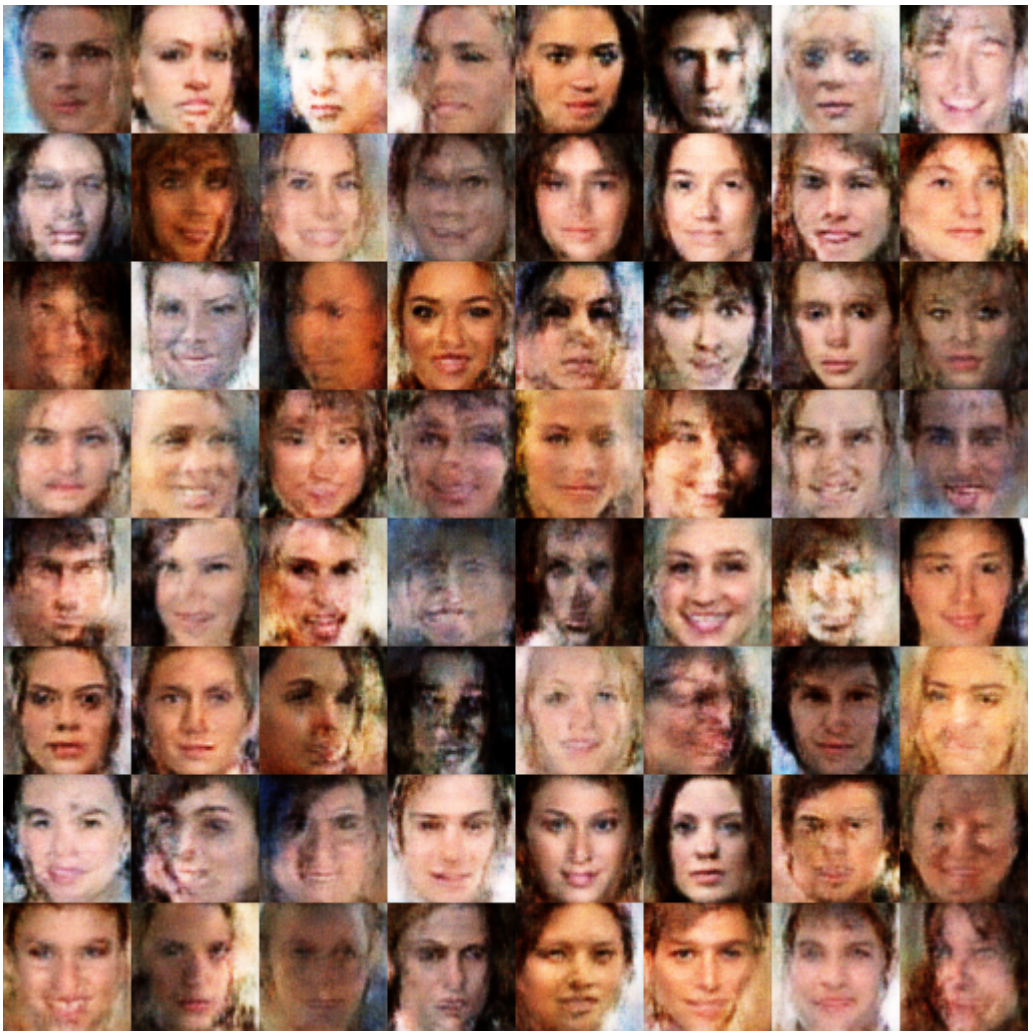
The difference between method A and method B:

I choose to implement DCGAN as my method A and SNGAN as my method B.
The main difference between DCGAN and SNGAN is that SNGAN applies spectrum normalization on each layer of discriminator (i.e., the generator is identical).
The reason for SNGAN using spectrum normalization is to make the discriminator a 1-Lipschitz function, which makes the optimization goal from minimizing KL divergence to minimizing wasserstein distance. Thus, SNGAN has a more stable training curve.
※ **Spectrum normalization:** Divides the weight of each discriminator layer by its largest singular value

3. Please discuss what you've observed and learned from implementing GAN.

I've implemented DRAGAN, WGAN-gp, WGAN-div, DCGAN and SNGAN for this problem.
The characteristics of DRAGAN, WGAN-gp and WGAN-div are very similar: They all have a very stable training curve, but produce blurry images.

Wasserstein distance during training (stable is better)



Blurry images produced by WGAN family

Meanwhile, DCGAN has a very unstable training curve, I have to tune it several times until it produces reasonable images; SNGAN combines the best of both worlds: It has a stable training curve, and produces sharp images. Thus my final choice for model B is SNGAN.

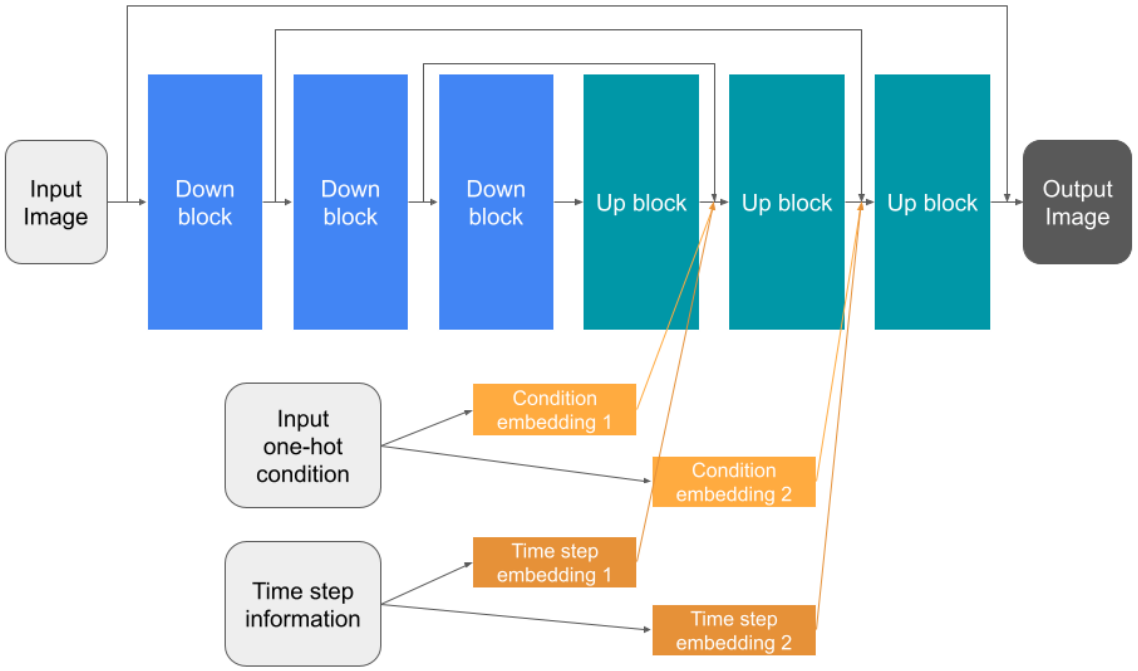
Problem 2: Diffusion Model

1. Please show your model architecture and describe your implementation details.

The code is adapted from [TeaPearce/Conditional_Diffusion_MNIST]

I chose conditional UNet as my denoising model.

Model Architecture Figure (the pytorch-generated text description is way too long):



The UNet consists of three main components:

1. Down block (Encoder): I use three blocks to downsample the input, where each block consists of two layers of convolution with GELU activation.
2. Up block (Decoder): I use three blocks to upsample the input, where each block consists of one transpose convolution, two convolution layers with GELU activation.
3. Embedding blocks: I use four different embedding blocks to embed the conditions and time-step information, where each block consists of two FC layers. Then I combine the input before decoder by the form of:
   decoder_input = concat(encoder_output, skip) * condition_embed + time_embed.

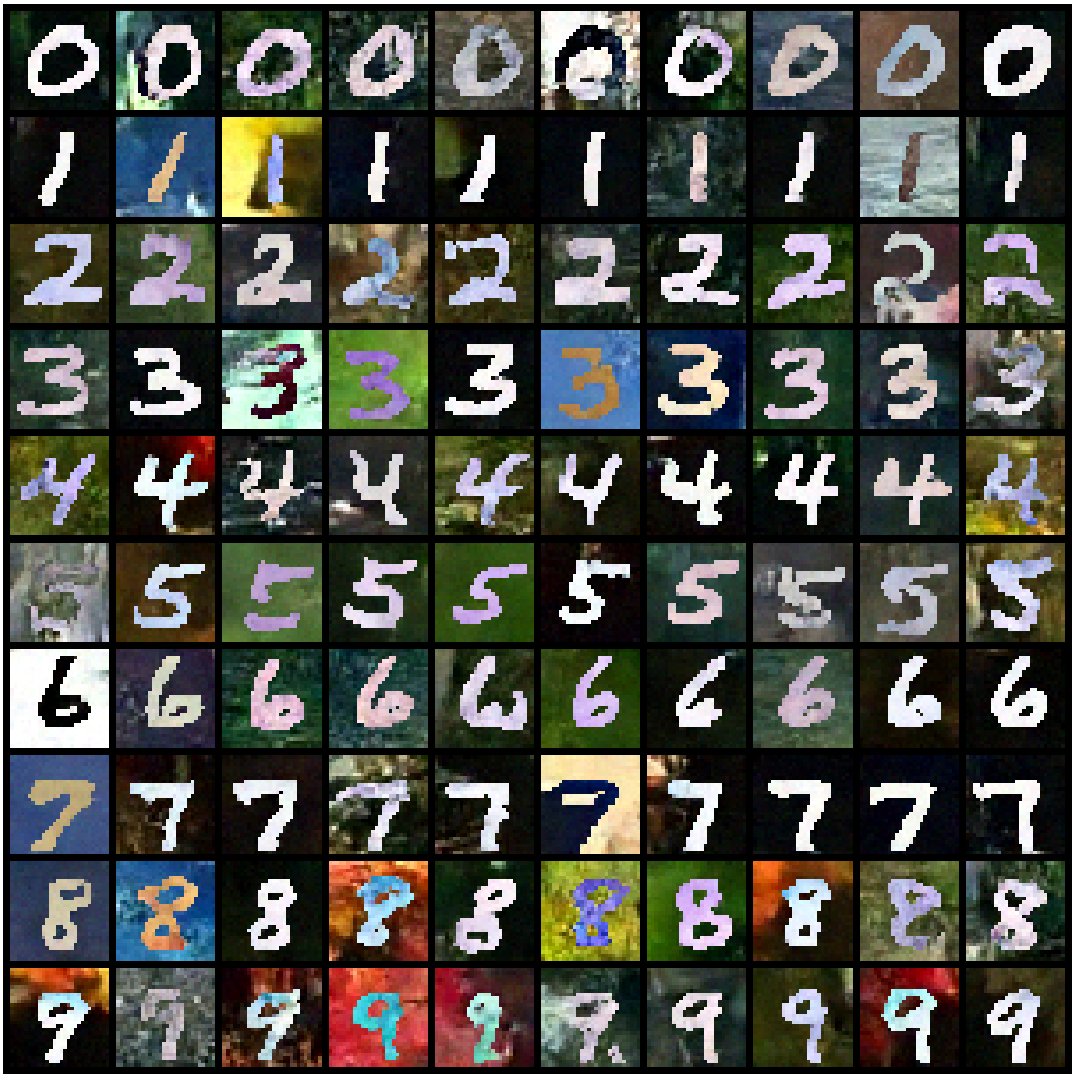
As for the diffusion model system, I follow the original procedure introduced in Denoising Diffusion Probabilistic Models, while with minor modifications to accommodate conditional generation:

1. During training, Randomly drops(masks) the condition. This technique enables the model to do unconditional generation. Thus enables the next technique.
2. During inference, sample twice with the same set of noise. One time with condition inputs and one time without condition inputs, while mixing the two outputs after every step of the denoising procedure. This approach makes the generated images more diverse. (the "guided_w" parameter decides the weight of mixing)

Training Details

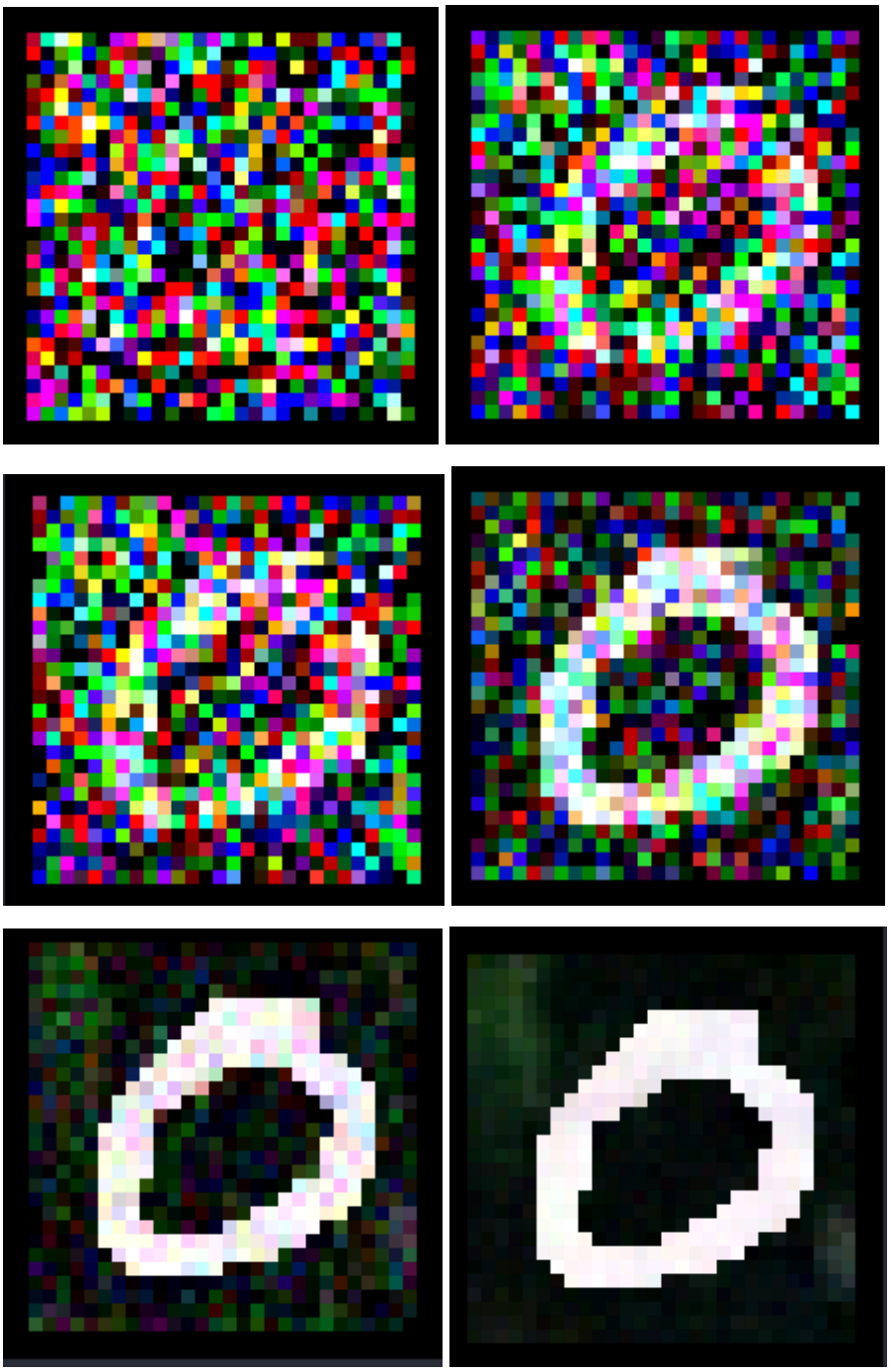
- Loss: MSE between noisy and denoised result
- Diffusion steps: 500
- Optimizer: Adam with learning rate = 0.0001 and linear lr decay
- Auto mixed precision training for faster training

2. 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.



3. Visualize a total of six images in the reverse process of the **first "0"** in your grid in (2) **with different time steps**.

(Ordered from left to right, then from top to bottom)



4. Please discuss what you've observed and learned from implementing conditional diffusion model.

During the training of diffusion models, I found out that diffusion models are more stable compared to GANs. More particularly, Diffusion models can converge with different sizes of model, different diffusion steps, or even different ways of embedding the conditions.

On the other hand, diffusion models' inference speed are way slower compared to GAN due to the nature of its iterative process. But the speed-to-stability trade-off is worth it in my opinion.

After implementing the conditional diffusion model, I've learnt details about how to train the model and to embed the conditions, which is a valuable experience. Also, I've tried to implement the latest conditional diffusion model approach: "classifier free diffusion guidance," which enables a new way to utilize conditions.

Problem 3: UDA

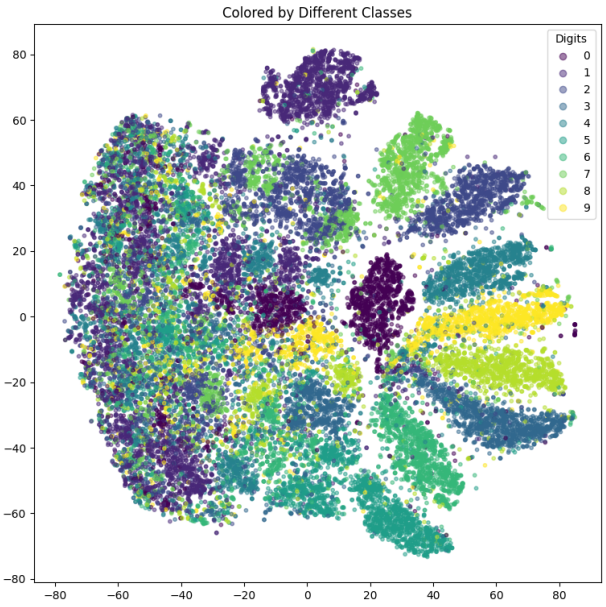
1. Please create and fill the table with the following format **in your report**:

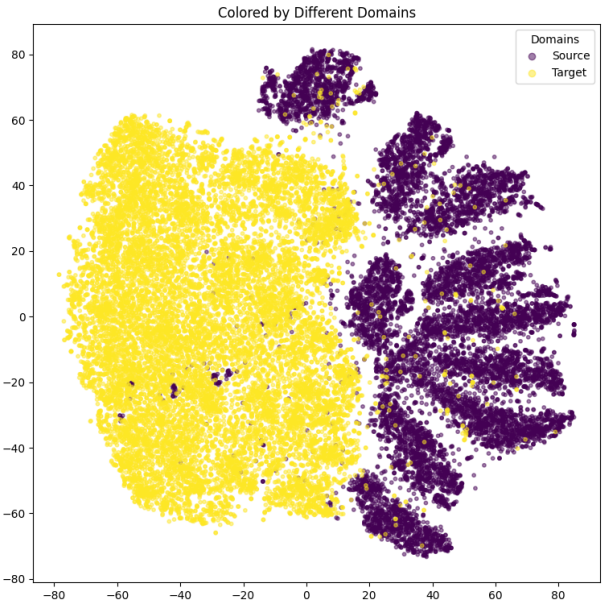
	MNISM → SVHN (Acc)	MNIST-M → USPS (Acc)
Trained on source	0.19233706508687143	0.2167463631465517
DANN	0.4943035186001133	0.9025537634408602
Trained on target	0.9146044041671608	0.9708378232758621

2. 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.

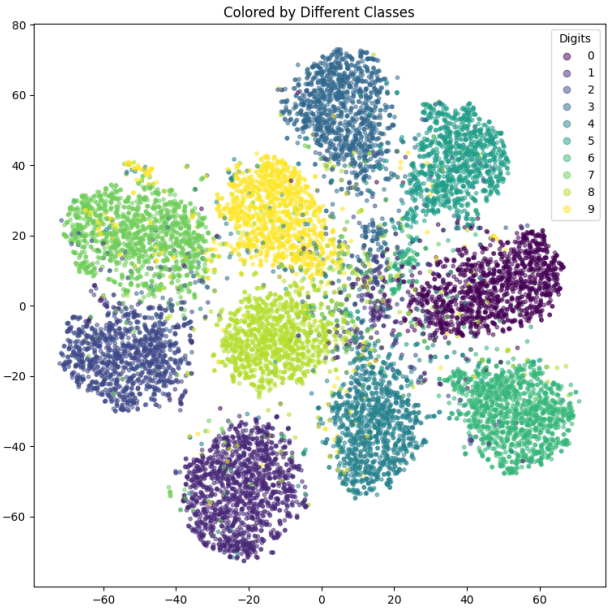
(The left figure is colored by digit classes, and the right figure is colored by domains)

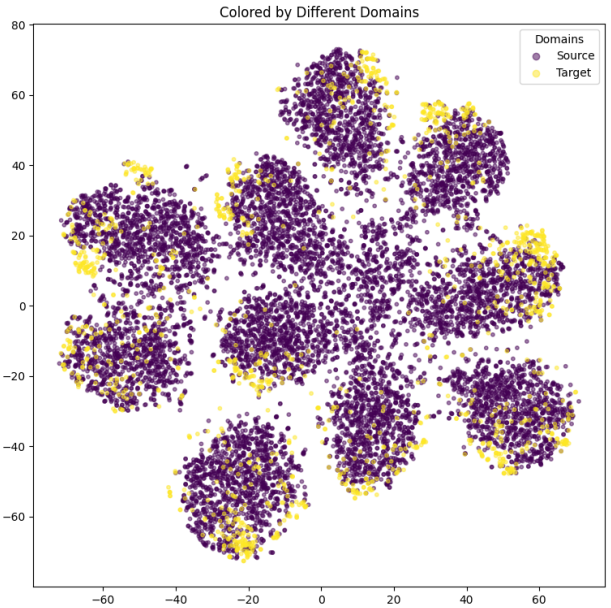
MNISTM → SVHN:





MNISTM → USPS:





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

Implementation details:

The code is adapted from [NaJaeMin92/pytorch_DANN]
I use the same architecture for both experiments.

Feature extractor:

```
FeatureExtractor(
  (conv): 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()
    (3): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 64, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): ReLU()
    (8): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  )
)
```

Label Predictor:

```
LabelPredictor(
  (l_clf): Sequential(
    (0): Linear(in_features=128, out_features=1024, bias=True)
    (1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=1024, out_features=256, bias=True)
    (4): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=10, bias=True)
  )
)
```

Domain Classifier

```
DomainClassifier(
  (d_clf): Sequential(
    (0): Linear(in_features=128, out_features=1024, bias=True)
    (1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=1024, out_features=256, bias=True)
    (4): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

Training Details

- loss function for label prediction: cross entropy
- loss function for domain classification: binary cross entropy
- optimizer: Adam with learning rate = 0.0003 and 0.003 for USPS and SVHN respectively
- lr scheduling:

$$\mu_p = \frac{\mu_0}{(1 + \alpha \cdot p)^\beta},$$

I use the formula described in the original DANN paper (Domain-Adversarial Training of Neural Networks), with alpha=10, beta=0.75

Observations and Findings

By comparing the performance of DANN and the “trained on target” setting, the model generalized well on both MNIST-M and USPS. Thus performs better and almost reaches the upper bound. On the other hand, the transfer performance of DANN is very low compared to the “trained on target” setting.
We can also observe this on the figures. The feature representation of MNIST-M → USPS is fusioned very well, while that of MNIST-M → SVHN is still easily separable. In conclusion, we can clearly see that the domain “gap” between MNIST-M and SVHN is way larger compared to that between MNIST-M and USPS.