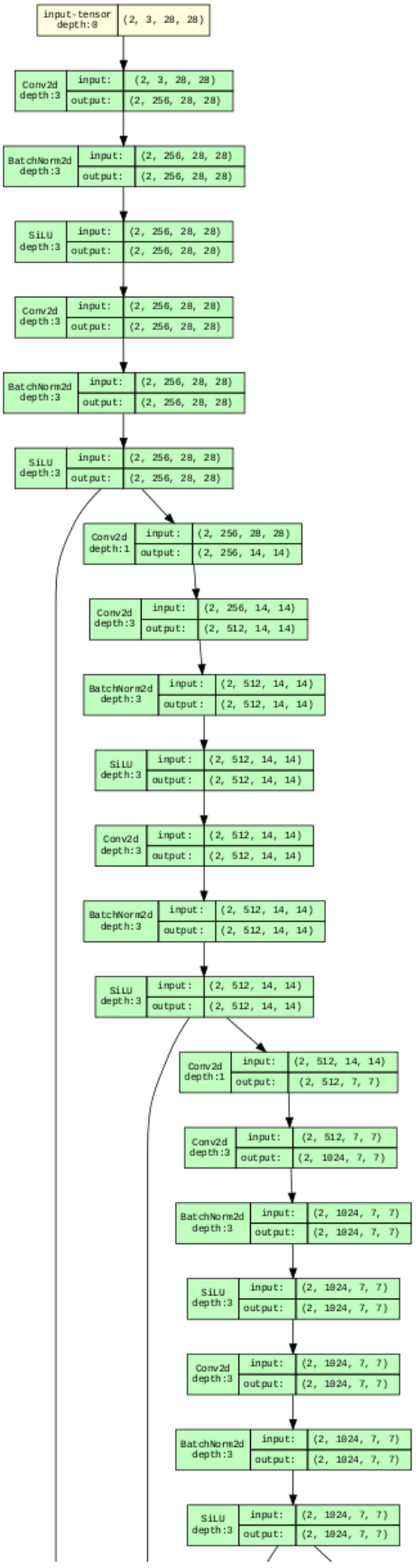
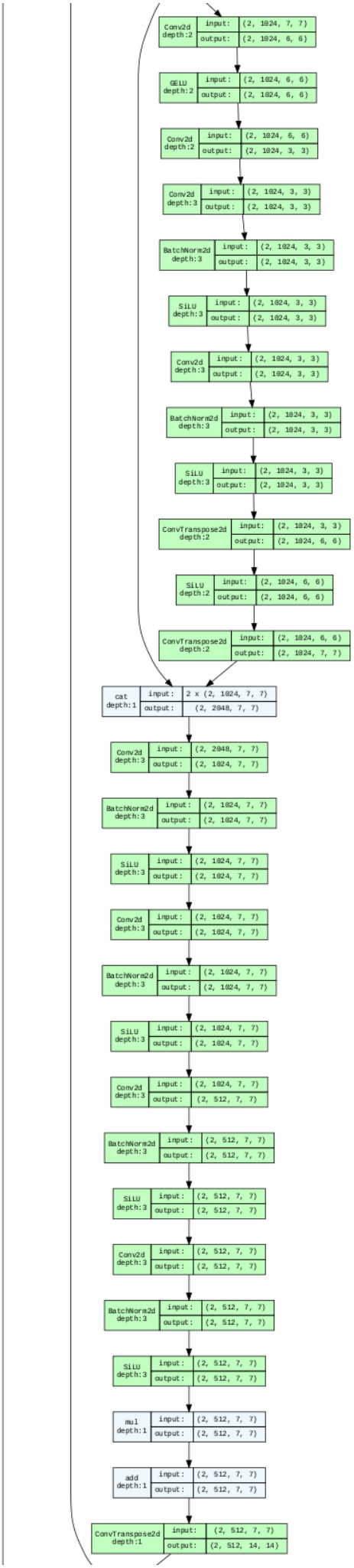
DLCV HW1 REPORT

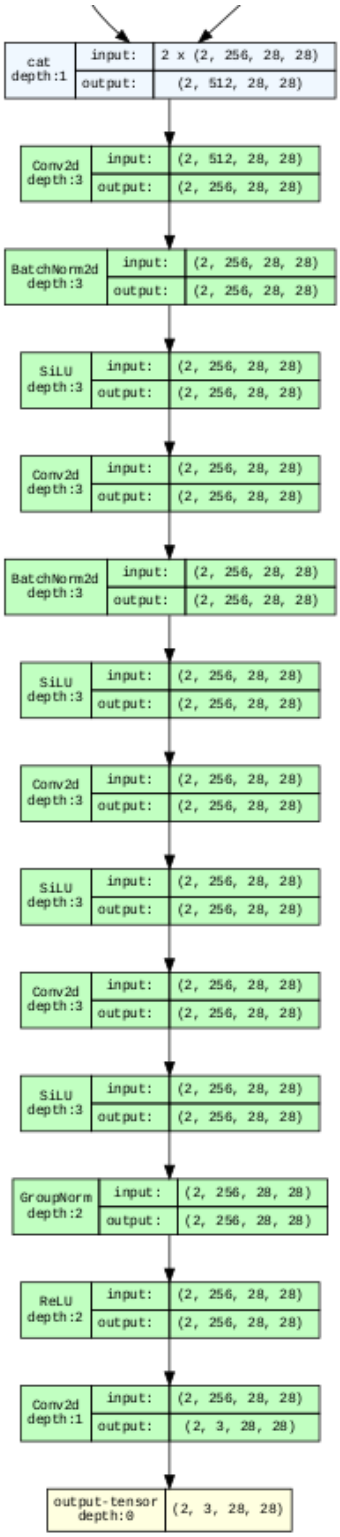
r12942046 薛仁豪

Problem 1

1. Draw the network architecture




Conv2d depth:2 input: (2, 1024, 7, 7) output: (2, 1024, 6, 6)
GELU depth:2 input: (2, 1024, 6, 6) output: (2, 1024, 6, 6)
Conv2d depth:2 input: (2, 1024, 6, 6) output: (2, 1024, 3, 3)
Conv2d depth:3 input: (2, 1024, 3, 3) output: (2, 1024, 3, 3)
BatchNorm2d depth:3 input: (2, 1024, 3, 3) output: (2, 1024, 3, 3)
SiLU depth:3 input: (2, 1024, 3, 3) output: (2, 1024, 3, 3)
Conv2d depth:3 input: (2, 1024, 3, 3) output: (2, 1024, 3, 3)
BatchNorm2d depth:3 input: (2, 1024, 3, 3) output: (2, 1024, 3, 3)
SiLU depth:3 input: (2, 1024, 3, 3) output: (2, 1024, 3, 3)
ConvTranspose2d depth:2 input: (2, 1024, 3, 3) output: (2, 1024, 6, 6)
SiLU depth:2 input: (2, 1024, 6, 6) output: (2, 1024, 6, 6)
ConvTranspose2d depth:2 input: (2, 1024, 6, 6) output: (2, 1024, 7, 7)
cat depth:1 input: 2 x (2, 1024, 7, 7) output: (2, 2048, 7, 7)
Conv2d depth:3 input: (2, 2048, 7, 7) output: (2, 1024, 7, 7)
BatchNorm2d depth:3 input: (2, 1024, 7, 7) output: (2, 1024, 7, 7)
SiLU depth:3 input: (2, 1024, 7, 7) output: (2, 1024, 7, 7)
Conv2d depth:3 input: (2, 1024, 7, 7) output: (2, 1024, 7, 7)
BatchNorm2d depth:3 input: (2, 1024, 7, 7) output: (2, 1024, 7, 7)
SiLU depth:3 input: (2, 1024, 7, 7) output: (2, 1024, 7, 7)
Conv2d depth:3 input: (2, 1024, 7, 7) output: (2, 512, 7, 7)
BatchNorm2d depth:3 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
SiLU depth:3 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
Conv2d depth:3 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
BatchNorm2d depth:3 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
SiLU depth:3 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
mul depth:1 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
add depth:1 input: (2, 512, 7, 7) output: (2, 512, 7, 7)
ConvTranspose2d depth:1 input: (2, 512, 7, 7) output: (2, 512, 14, 14)



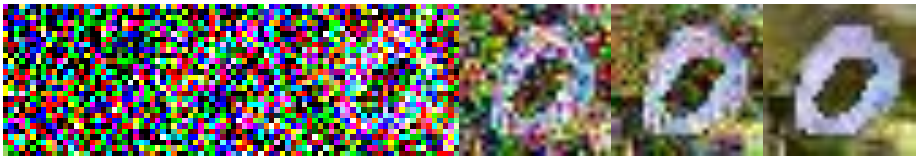
The architecture of my model consists of three encoding blocks, one bottleneck block, and two decoding blocks. In the encoding blocks, Conv2d is employed with a larger kernel size and stride to reduce the image dimensions. On the other hand, the decoding blocks utilize ConvTranspose2d to increase the image size. BatchNorm and SiLU are

applied in all layers except for the output layer, which employs GroupNorm and ReLU. The model implementation and sampling process are referenced from the following GitHub repository:

TeaPearce/Conditional_Diffusion_MNIST: Conditional diffusion model to generate MNIST. Minimal script. Based on 'Classifier-Free Diffusion Guidance'. (github.com)

2. Show Generated Images

3. Six Images in the Reverse Process



t=0 t=400 t=600 t=700 t=750 t=800

4. Observation

Given that the training data comprises solely digits, there's no necessity for an overly complex model. A shallow but efficient network is more suitable for this task. Moreover, the number of time steps significantly impacts the results. If the time steps are too few, some outputs may become saturated.

Problem 2

1. Face Images with Different eta

It appears that the generated images exhibit a gradual transformation as we fine-tune the value of eta. However, there are images undergo a drastic shift and become entirely distinct.

2. Interpolation

Spherical Linear Interpolation

Linear Interpolation

The outcome of the linear interpolation appears peculiar and incorrect. I attribute this anomaly to the model learning to sample from a Gaussian distribution rather than a uniform one, resulting in our latent space adopting a spherical shape.

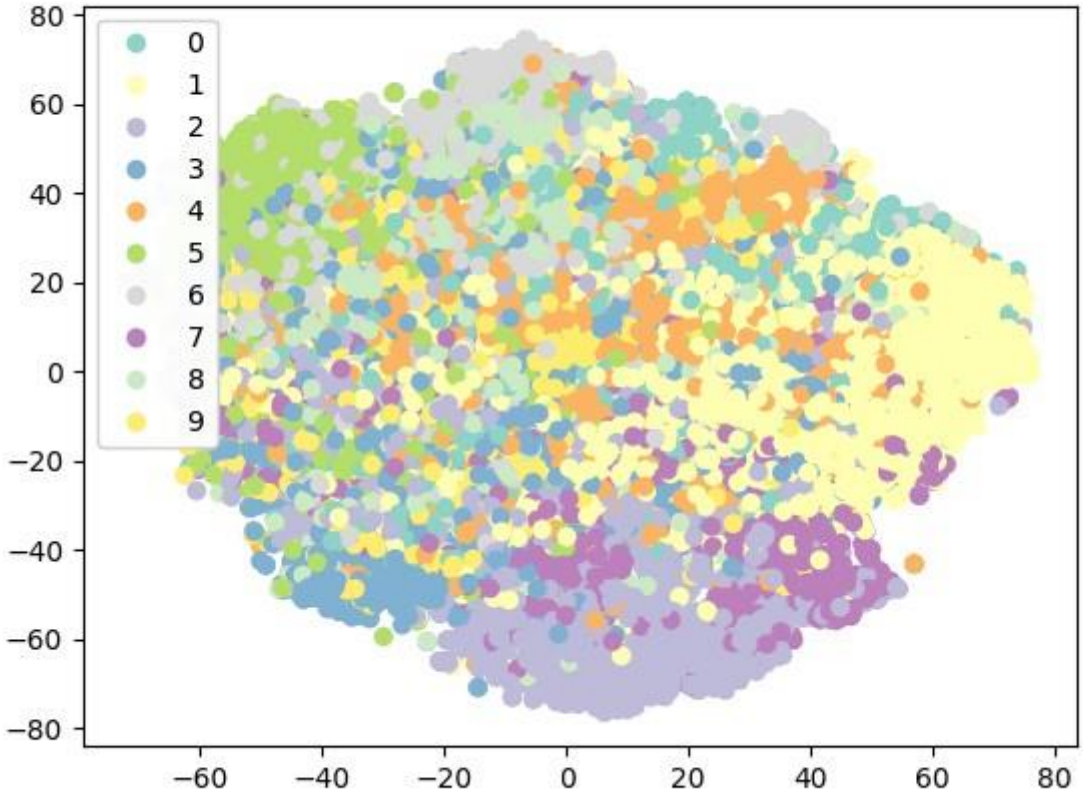
Problem 3

1. Accuracy of Different Settings

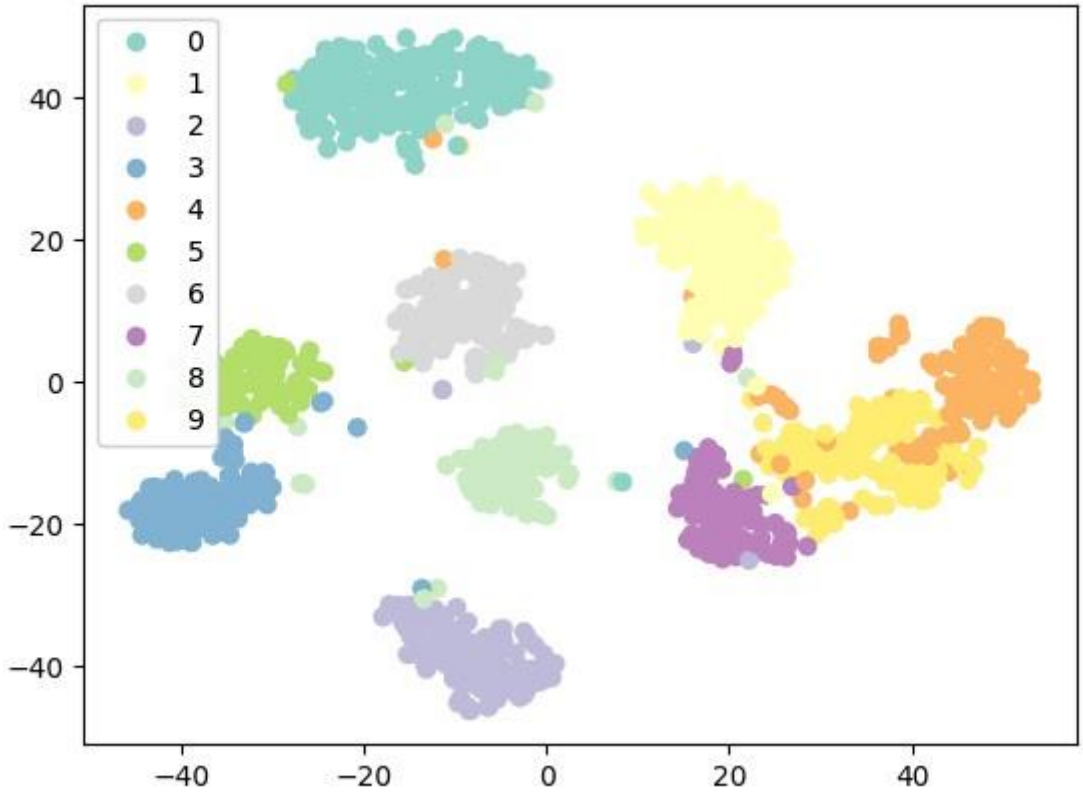
	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	**0.3653**	**0.7937**
Adaptation (DANN)	**0.4129**	**0.9043**
Trained on target	**0.9389**	**0.9872**

2. t-SNE by Digit Classes or by Domain

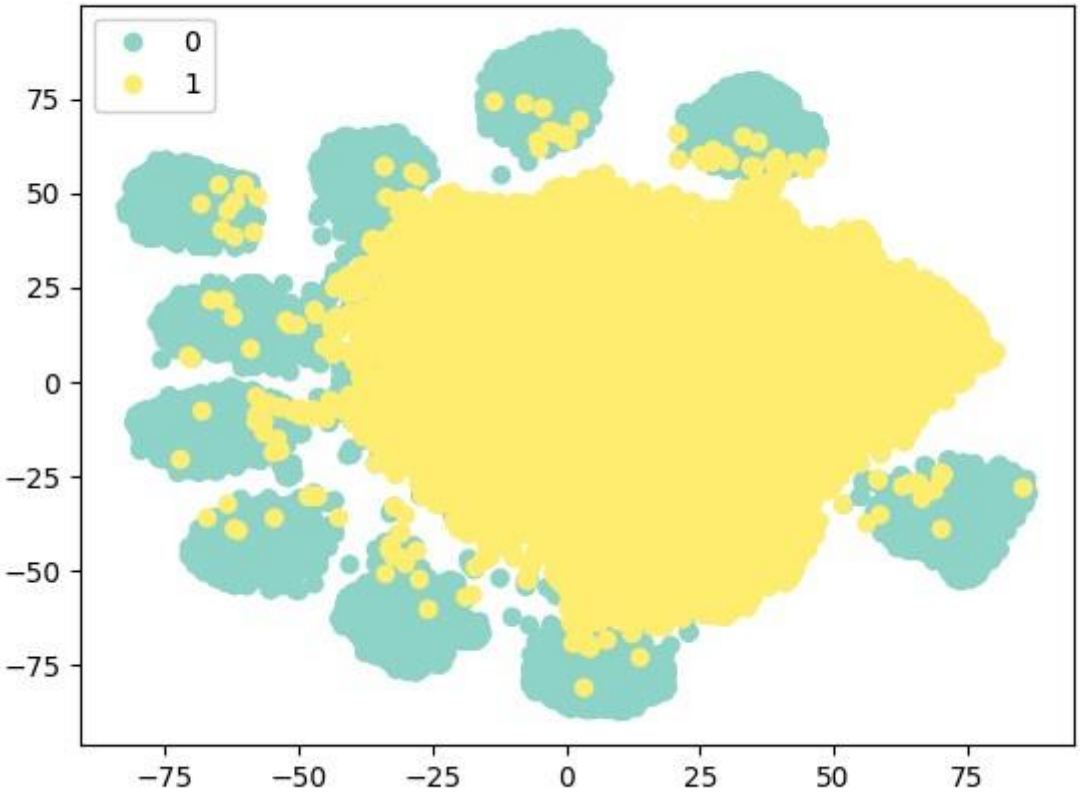
svhn by class



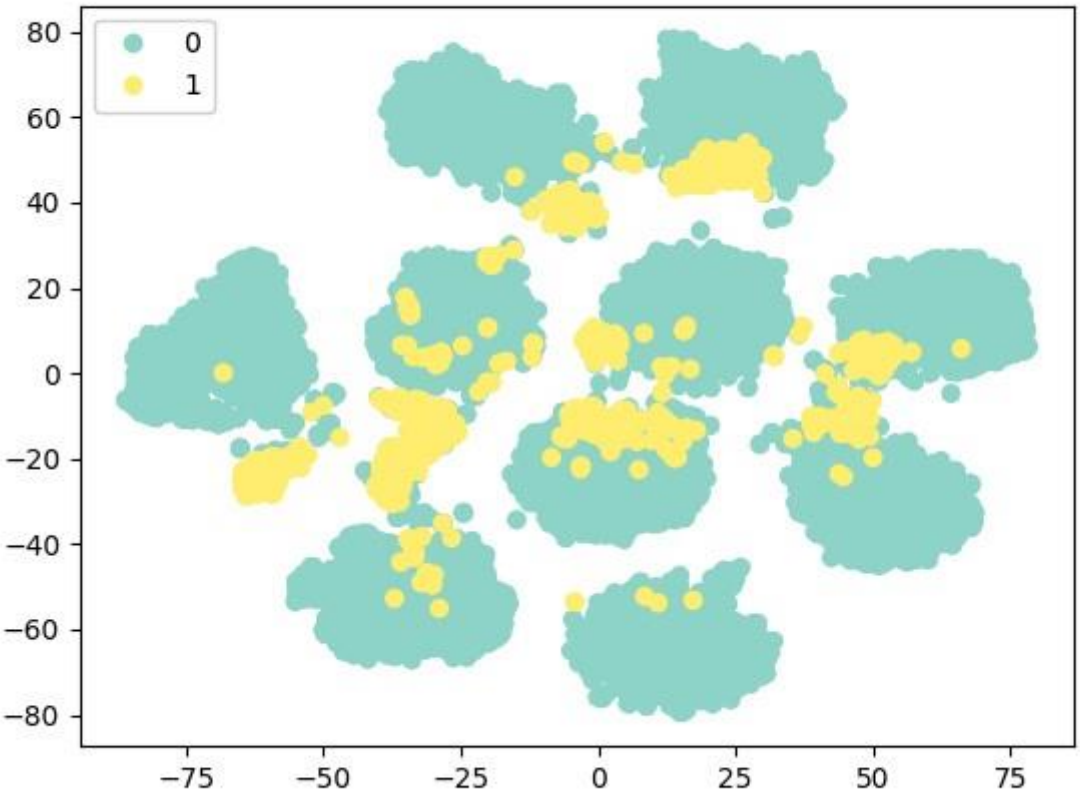
usps by class



mnistm to svhn by domain



mnistm to usps by domain



3. Implementation Detail

The model implementation and sampling process are referenced from the following GitHub repository:

NaJaeMin92/pytorch_DANN: PyTorch implementation of DANN (github.com)

The training procedure adheres to the standard DANN training protocol. The overall training spans 40 epochs, commencing with an initial learning rate of 0.01, which undergoes a 0.1 decay at the 20th epoch. Furthermore, the feature extractor, responsible for extracting features from images, comprises 8 convolution layers. As for the classifier, it encompasses 1 fully connected layer, and the discriminator integrates 3 fully connected layers.

During implementation, I observed that augmenting the depth and width of the feature extractor proves to be significantly more beneficial than augmenting the parameter count of the classifier and discriminator.