DLCV HW2

Problem 1

Please print the model architecture of method A and B

model A

discrminator

```
(model): Sequential(
  (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (3): ReLU(inplace=True)
  (4): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (5): ReLU(inplace=True)
  (6): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (7): ReLU(inplace=True)
  (8): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (9): Sigmoid()
)
```

generator

```
(model): Sequential(
  (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): ReLU(inplace=True)
  (2): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (3): ReLU(inplace=True)
  (4): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (5): ReLU(inplace=True)
  (6): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (7): ReLU(inplace=True)
  (8): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (9): Tanh()
)
```

model B

discrminator

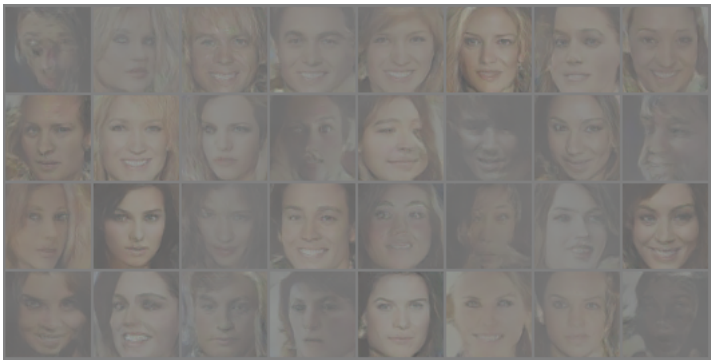
```
(model): Sequential(
  (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (1): LeakyReLU(negative_slope=0.2, inplace=True)
  (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (4): LeakyReLU(negative_slope=0.2, inplace=True)
  (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, inplace=True)
  (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, inplace=True)
  (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (12): Sigmoid()
)
```

generator

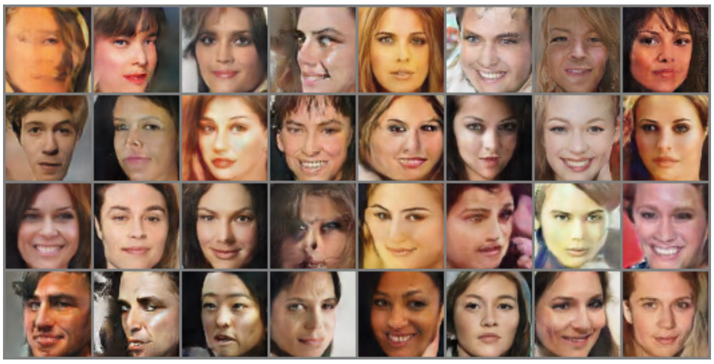
```
(model): Sequential(
  (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): ReLU(inplace=True)
  (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): ReLU(inplace=True)
  (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): ReLU(inplace=True)
  (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (13): Tanh()
)
```

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

model A



model B



In model A, we can see that it has learned some face feature but the color is darker. In other word, the distribution in model A has more bias than model B. In variance perspective, model A and model B isn't too different.

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

- GAN is hard to train. If we only change random seed, the result may get different.
- Add BatchNorm layer and leakyReLU can improve the model performance
- Afer first two epoch, the model can generate rough face shape and the model generate clear face after 100 epoch.

Problem 2

Please print your model architecture and describe your implementation details

model architecture(UNet with attention layer)

```
condition_UNet(
  (pos_embedding): position_embedding()
  (label_embedding): Embedding(10, 256)
  (inc): basic_block(
    (conv): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): GroupNorm(1, 64, eps=1e-05, affine=True)
      (2): GELU(approximate=none)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): GroupNorm(1, 64, eps=1e-05, affine=True)
    )
  )
  (down1): down(
    (conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): basic_block(
        (conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 64, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 64, eps=1e-05, affine=True)
        )
      )
      (2): basic_block(
        (conv): Sequential(
          (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
    )
    (pos_embedding): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=128, bias=True)
    )
  )
  (sa1): attention(
    (m_attention): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=128, out_features=128, bias=True)
    )
    (norm): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
    (fc): Sequential(
      (0): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=128, out_features=128, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=128, out_features=128, bias=True)
    )
  )
  (down2): down(
    (conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): basic_block(
        (conv): Sequential(
          (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
      (2): basic_block(
        (conv): Sequential(
          (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 256, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 256, eps=1e-05, affine=True)
        )
      )
    )
    (pos_embedding): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=256, bias=True)
    )
  )
```

```
(sa2): attention(
  (m_attention): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
  )
  (norm): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
  (fc): Sequential(
    (0): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
    (1): Linear(in_features=256, out_features=256, bias=True)
    (2): GELU(approximate=none)
    (3): Linear(in_features=256, out_features=256, bias=True)
  )
)
(down3): down(
  (conv): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): basic_block(
      (conv): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(1, 256, eps=1e-05, affine=True)
        (2): GELU(approximate=none)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): GroupNorm(1, 256, eps=1e-05, affine=True)
      )
    )
    (2): basic_block(
      (conv): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(1, 256, eps=1e-05, affine=True)
        (2): GELU(approximate=none)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): GroupNorm(1, 256, eps=1e-05, affine=True)
      )
    )
  )
  (pos_embedding): Sequential(
    (0): SiLU()
    (1): Linear(in_features=256, out_features=256, bias=True)
  )
)
(sa3): attention(
  (m_attention): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=256, out_features=256, bias=True)
  )
  (norm): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
  (fc): Sequential(
    (0): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
    (1): Linear(in_features=256, out_features=256, bias=True)
    (2): GELU(approximate=none)
    (3): Linear(in_features=256, out_features=256, bias=True)
  )
)
(bot1): basic_block(
  (conv): Sequential(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): GroupNorm(1, 256, eps=1e-05, affine=True)
    (2): GELU(approximate=none)
    (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (4): GroupNorm(1, 256, eps=1e-05, affine=True)
  )
)
(bot2): basic_block(
  (conv): Sequential(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): GroupNorm(1, 256, eps=1e-05, affine=True)
    (2): GELU(approximate=none)
    (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (4): GroupNorm(1, 256, eps=1e-05, affine=True)
  )
)
```

```
  (up1): up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): basic_block(
        (conv): Sequential(
          (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 512, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 512, eps=1e-05, affine=True)
        )
      )
      (1): basic_block(
        (conv): Sequential(
          (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 256, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 128, eps=1e-05, affine=True)
        )
      )
    )
    (pos_embedding): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=128, bias=True)
    )
  )
  (sa4): attention(
    (m_attention): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=128, out_features=128, bias=True)
    )
    (norm): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
    (fc): Sequential(
      (0): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=128, out_features=128, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=128, out_features=128, bias=True)
    )
  )
  (up2): up(
    (up): Upsample(scale_factor=2.0, mode=bilinear)
    (conv): Sequential(
      (0): basic_block(
        (conv): Sequential(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): basic_block(
        (conv): Sequential(
          (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(1, 128, eps=1e-05, affine=True)
          (2): GELU(approximate=none)
          (3): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (4): GroupNorm(1, 64, eps=1e-05, affine=True)
        )
      )
    )
    (pos_embedding): Sequential(
      (0): SiLU()
      (1): Linear(in_features=256, out_features=64, bias=True)
    )
  )
  (sa5): attention(
    (m_attention): MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=64, out_features=64, bias=True)
    )
    (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
      (1): Linear(in_features=64, out_features=64, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=64, out_features=64, bias=True)
    )
  )
  (outc): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
)
```

optimizer : Adam

learning rate : 1e-4

loss function : smooth L1 loss

epoch : 39

batch size : 16

timestep : 400

beta start : 0.0001

beta end : 0.02

Please show 10 generated images for each digit (0-9) in your report

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

t=0

t=80

t=160

t=240

t=320

t=400

Please discuss what you’ve observed and learned from implementing conditional diffusion model

- timestep may affect a lot in diffusion model.
- Too large timestep may let the diffusion process probagate more error. Too small timestep is hard to train. Hence, we need to find a suitable timestep.
- Add attention layer to UNet may improve the model performance.

Problem 3

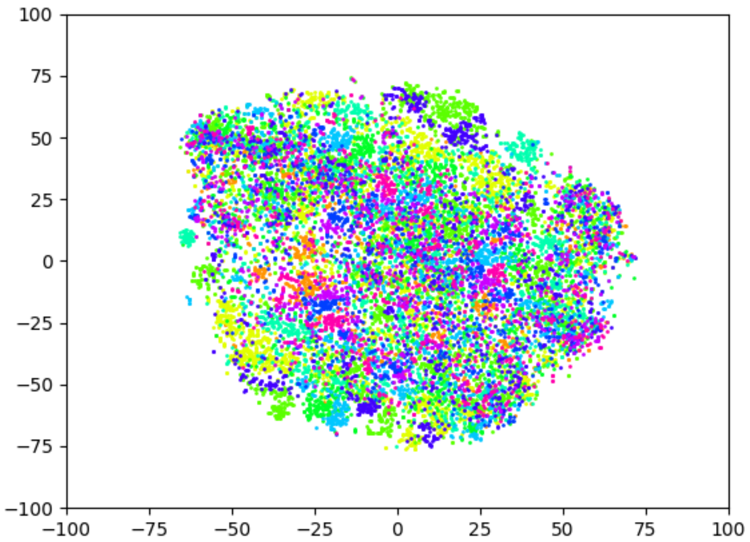
Please create and fill the table with the following format in your report

	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	0.3729	0.6982
Adaptation (DANN)	0.4460	0.8017
Trained on target	0.9295	0.9838

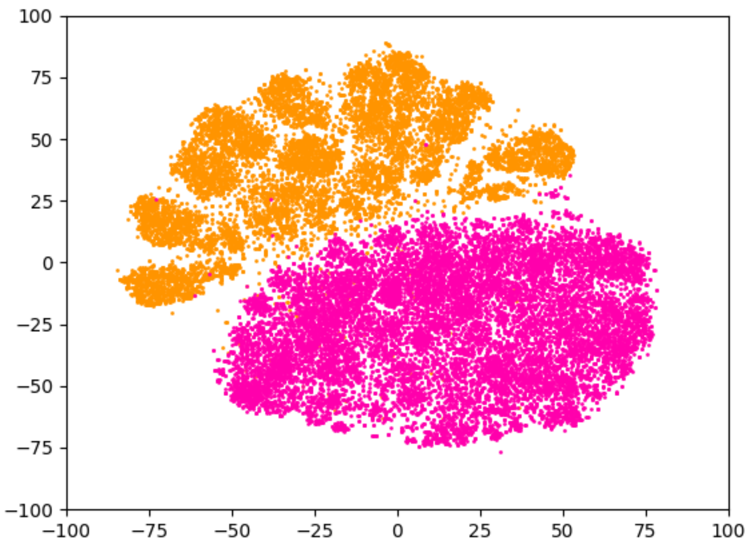
Please visualize the latent space of DANN by mapping the validation images to 2D space with t-SNE

MNIST-M → SVHN

by class

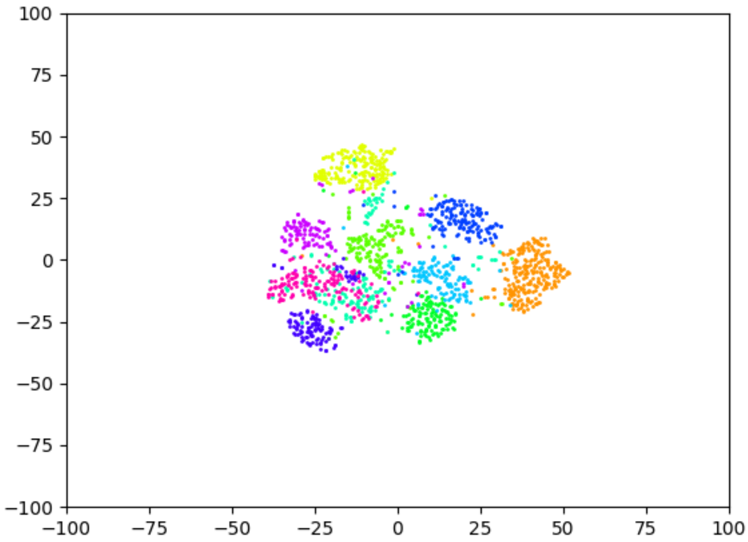


by domain

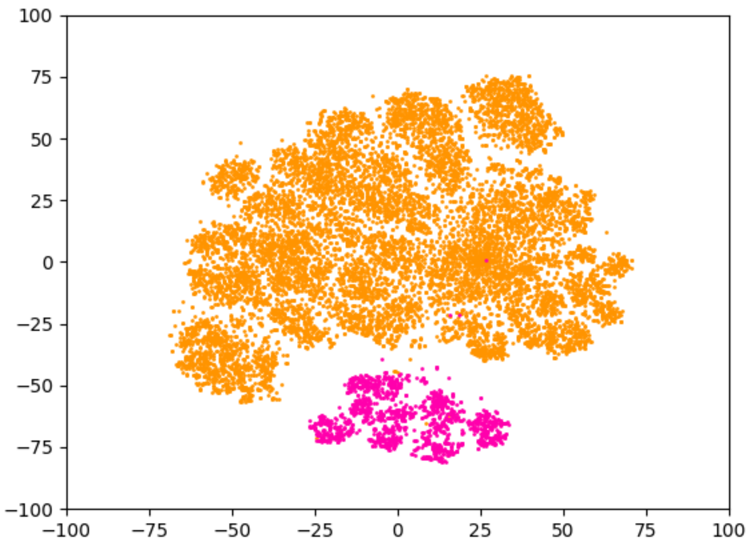


MNIST-M → SVHN

by class



by domain



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

model architecture

```
DANN(
  (feature): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(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)
    (4): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Dropout2d(p=0.5, inplace=False)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): ReLU(inplace=True)
    (9): Flatten(start_dim=1, end_dim=-1)
  )
  (class_classifier): Sequential(
    (0): Linear(in_features=2048, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (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)
  )
  (domain_classifier): Sequential(
    (0): Linear(in_features=2048, out_features=100, bias=True)
    (1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=100, out_features=2, bias=True)
  )
)
```

optimizer : Adam
learning rate : 1e-4
loss function : cross entropy loss
batch size : 64

- In DANN, target data performance is unstable. Sometime training accuracy improve but target accuracy decrease.
- Data size in this task play a crucial role so i implement colorjitter to USPS dataset.
- In this problem, I have learned inverse gradient trick which I didn’t use before.

Reference

DCGAN :
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html (https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)
diffusion model :
https://colab.research.google.com/drive/1sjy9odlSSy0RBVgMTgP7s99NXsqglsUL?usp=sharing

(https://colab.research.google.com/drive/1sjy9odlSSy0RBVgMTgP7s99NXsqglsUL?usp=sharing)

https://huggingface.co/blog/annotated-diffusion

(https://huggingface.co/blog/annotated-diffusion)

DANN :
https://github.com/fungtion/DANN (https://github.com/fungtion/DANN)