DLCV 2022 – HW2

Name : 周宇玄
Student ID : R10525104

Problem 1.:

1-A.

```
Generator(
  (main): 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)
  )
  (out): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): Tanh()
  )
)
Discriminator(
  (main): 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()
  )
)
```

1-B.

```
Generator(
  (conv_block): 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): ConvTranspose2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ReLU(inplace=True)
    (5): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): ConvTranspose2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ConvTranspose2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU(inplace=True)
    (15): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (16): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ConvTranspose2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (21): Tanh()
  )
)
Discriminator(
  (conv_block): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ConvTranspose2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.2, inplace=True)
    (7): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (8): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ConvTranspose2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): LeakyReLU(negative_slope=0.2, inplace=True)
    (12): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (13): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ConvTranspose2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): LeakyReLU(negative_slope=0.2, inplace=True)
    (17): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
    (18): Sigmoid()
  )
)
```

2-A.

2-B.

2. Discuss the difference between method A and B:

The B model consist of A model plus res-concept, and we can see that images generated by both models have same problem which is too dark, so we will think that there’s no big difference between method A and B by observe.

But by the test, we know that even both model’s face-recognition scores are very close (around 90%), but FID score have big different (28 and 23), so we can know that even though the generated images don't seem to be too big difference, but A model has smaller FID with val. face data, which mean A model is better than B model in this case.

3.

I learned that universal formulas are not always “universal”, although this is a matter of course, but before implement GAN model, I really didn’t think that some universal skills they are not always universal.

For example, I use normalize in almost every model I build, and always set by “[[0.2673, 0.5345, 0.8018],[0.4558, 0.5698, 0.6838]]”, just because this parameter always has good effect on the models which I build, but in this problem it’s not.

I spend a lot of time trying to figure out how to break the strong baseline, by take

other augmentation, change model architecture, … etc. at last I find that only I need is change normalize's parameter to [0.5, 0.5, 0.5], just a simple idea but I never think about it. So if I have to discuss what I've observed and learned from implementing GAN, I'll say that I observed and learned some universal skills they are not always universal.

Problem 2.:

1.

```
DDPM(
  (nn_model): ContextUnet(
    (init_conv): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(3, 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)
      )
    )
    (down1): UnetDown(
      (model): Sequential(
        (0): ResidualConvBlock(
          (conv1): 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)
          )
          (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)
      )
    )
    (down2): UnetDown(
      (model): Sequential(
        (0): ResidualConvBlock(
          (conv1): 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): GELU(approximate=none)
          )
          (conv2): Sequential(
            (0): Conv2d(512, 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): 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=512, bias=True)
    (1): GELU(approximate=none)
    (2): Linear(in_features=512, out_features=512, bias=True)
  )
)
(timeembed2): 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)
  )
)
(contextembed1): EmbedFC(
  (model): Sequential(
    (0): Linear(in_features=10, out_features=512, bias=True)
    (1): GELU(approximate=none)
    (2): Linear(in_features=512, out_features=512, bias=True)
  )
)
(contextembed2): 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)
  )
)
(up0): Sequential(
  (0): ConvTranspose2d(512, 512, kernel_size=(7, 7), stride=(7, 7))
  (1): GroupNorm(8, 512, eps=1e-05, affine=True)
  (2): ReLU()
)
(up1): UnetUp(
  (model): Sequential(
    (0): ConvTranspose2d(1024, 256, kernel_size=(2, 2), stride=(2, 2))
    (1): ResidualConvBlock(
      (conv1): 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)
      )
      (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)
      )
    )
    (2): ResidualConvBlock(
      (conv1): 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)
```

```
        )
        (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)
        )
      )
    )
  )
  (up2): UnetUp(
    (model): Sequential(
      (0): ConvTranspose2d(512, 256, kernel_size=(2, 2), stride=(2, 2))
      (1): ResidualConvBlock(
        (conv1): 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)
        )
        (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)
        )
      )
      (2): ResidualConvBlock(
        (conv1): 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)
        )
        (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)
        )
      )
    )
  )
  (out): Sequential(
    (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): GroupNorm(8, 256, eps=1e-05, affine=True)
    (2): ReLU()
    (3): Conv2d(256, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
)
(loss_mse): MSELoss()
)
```

1. Describe your implementation details:

In implement, I use eps=0.00001, momentum=0.1 be my Hyperparameter, and choose active function=GELU.

This model refer to Conditional Diffusion paper, model can be divided into three parts, which is “UnetUp and UnetDown”, “EmbedFC”, and “in-out part”, use UnetDown and UnetUP is correspond to downsample and upsample layer in the paper model that hw2_intro provided, those make the feature on the source can be extract on different size. Similarly, EmbedFC is correspond to middle layer in paper model, it make reconstruction more progressive (means retain more complete features), and in-out part is used to match the in-out size we defined.

But I think the key point of training is Algorithm 1 and Algorithm 2 that didn’t shows on model print, which is used to guess epsilon value and sampling from the

reverse process, those two algorithm make model become powerful and stable.

2.

3.

T = 0, 200, 400, 600, 800, 1000,

4.

Diffusion models is a model that amazes me, before I learned this model I know that noise is a kind of data-augmentation, but I didn’t think about make noise into training and use reverse to make model learn more well, I learned and I observed that data-augmentation could use in different way like become a training skill, I think maybe the other data-augmentation could be training skill too and search about it then find “rotation” is a good training skill too, so I think the biggest

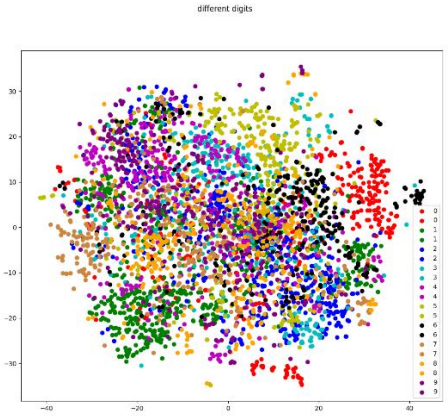
thing about what I learned is that know about different thinking in data-augmentation tech.

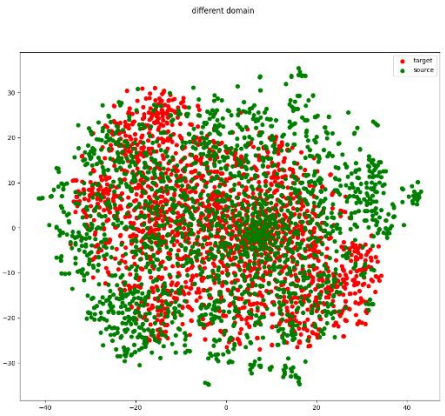
Problem 3.:

1.

	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	50.9 %	83.7 %
Adaptation (DANN)	52.6 %	85.1 %
Trained on target	97.2 %	97.8 %

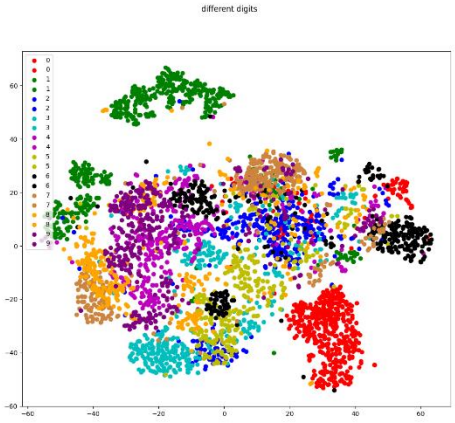
2-MNIST-M->SVHN.

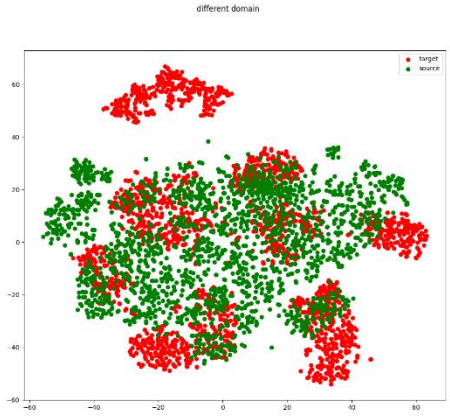




By class By domain

2-MNIST-M->USPS.

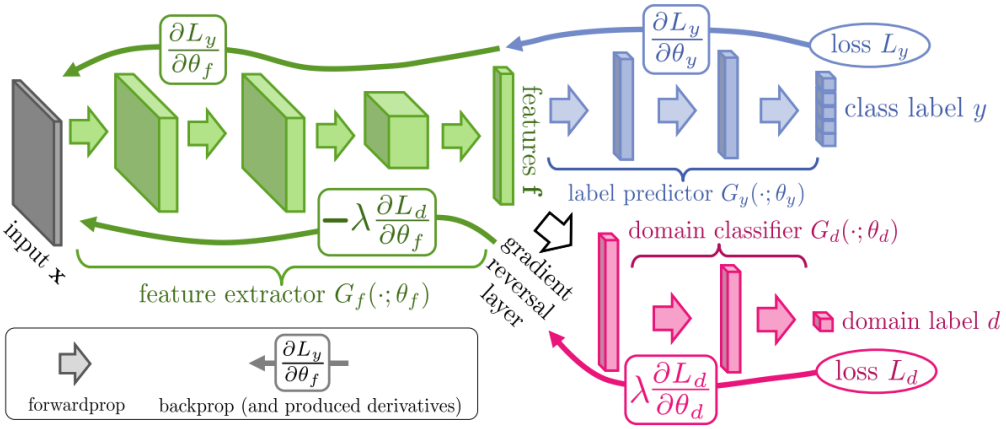




By class By domain

3.

My model architecture is refer to DANN model paper which provide on hw2_intro



In the paper, loss function use SGD, but after test, I find that NLLLOSS get better result of those dataset, and optimizer didn't mention in paper, I choose Adam be my optimizer this time, lr I test by 0.01, 0.001, 0.0001, 0.0002. result shows that 0.001 has best effect.

I observe that "flip" or "rotation" is not effect well (even make result worse) on this model, I think the reason is that different dataset make the difference become a bad element, and I observe that Lower bound is not much worse than DANN, before test I think lower bound will be much worse than DANN.

I learned that more variation between datasets than I thought, and if we want to

overcome this issue we should not use augmentation make dataset become sundry, instead, data augmentation should make the two datasets have close conditions, like light, distributed, color domain, …etc.

Reference :

Hw2_intro

Pytorch-fid :

https://github.com/mseitzer/pytorch-fid

DCGAN :

https://arxiv.org/abs/1511.06434 DCGAN

https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html#dcgan-tutorial

Diffusion models :

https://arxiv.org/abs/2006.11239

https://github.com/hojonathanho/diffusio

n

https://arxiv.org/abs/1610.09585

https://github.com/TeaPearce/Conditional_Diffusion_MNIST

DANN :

https://arxiv.org/pdf/1505.07818.pdf

http://sites.skoltech.ru/compvision/projects/grl/