DLCV hw2

B09901062 黃宥翔

Problem1

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

model A

```
DCGAN_Generator(
  (tconv1): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv2): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv4): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv5): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
)
DCGAN_Discriminator(
  (conv1): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
)
```

model B

```
WGANGP_Generator(
  (tconv1): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv2): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv4): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv5): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
)
WGANGP_Discriminator(
  (l1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (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)
    )
    (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)
    )
    (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)
    )
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
  )
)
```

(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

Method B

Difference between DCGAN and WGANGP:

WGAN is proposed to handle DCGAN's problem of mode collapse and vanishing gradient. Instead of BCEloss that may cause vanishing gradient, the loss is calculated by expected value of discriminator's output to real images and the expected value of discriminator's output to fake images. And Gradient penalty is added for

regularization of discriminator gradient.
Sounds that WGAN will be better than DCGAN, but in this hw, I can't get WGAN done, the fid score won't drop. So I use model A and ensemble instead.

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

You can compare different architectures or describe some difficulties during training, etc.

I'll talk about the problems I faced when training GAN.
When I was training DCGAN initially, the fid score dropped very fast and passed the strong baseline. However, the face recognition score didn't. So I switched to WGANGP, but it somehow didn't work out. Both the fid score and face recognition score failed passing strong baseline.
So finally, I decide to use two different DCGAN model, one with higher face recognition score, the other with lower fid score, to ensemble, and somehow passed the strong baseline.

Problem 2

(1) Please print your model architecture and describe your implementation details.

Model Architecture (I just use print(model))

```
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()
    )
```

Implementation Detail:

I referenced this github repo :

https://github.com/TeaPearce/Conditional_Diffusion_MNIST

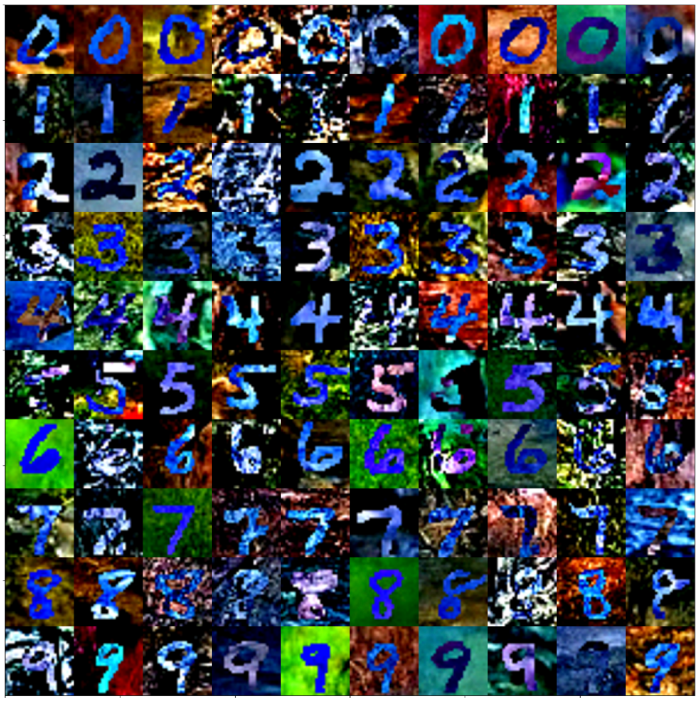
It use DDPM as model.

The conditioning roughly follows the method described in Classifier-Free Diffusion Guidance. The model infuses **timestep embeddings** te and **context embeddings** ce with the **U-Net activations at a certain layer** aL, via,

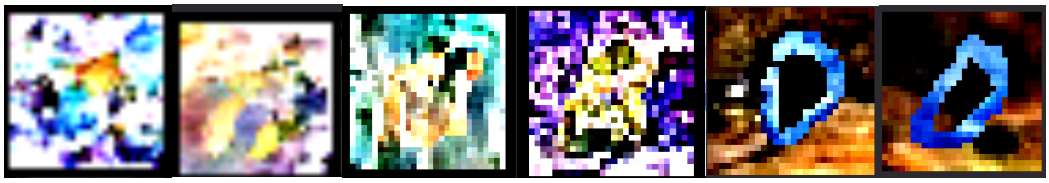
$$aL+1=ceaL+te.$$

in the upsampling process.

(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. [see the below example]



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



Timestep from 0 / 100 / 200 / 300 / 400 / 500

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

I found it very cool that its implementation logic is totally different from the generative model before (like GAN or VAE). So at first I have to do more research and code reading. Also, the concept of conditioning is what I have never met before, so I think I really learn a lot (and viewing the result is satisfying too).

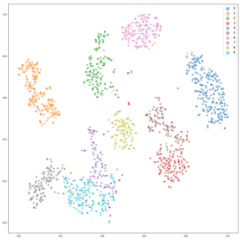
Problem3

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

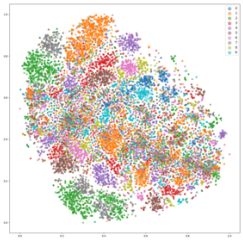
	MNIST-M > SVHN	MNIST-M > USPS
Trained on source	0.3767	0.7285
Adaptation (DANN)	0.5436	0.8177
Trained on target	0.9320	0.9589

(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.

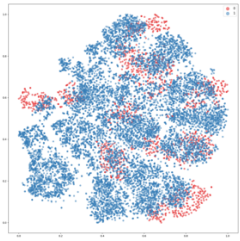
- Note that you need to plot the figures of both **2 scenarios**, so **4 figures** in total.



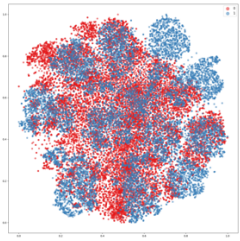
USPS by class.



SVHN by class



USPS by domain(target: red / source: blue)



SVHN by domain(target: red / source: blue)

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

I implement DANN with feature extractor, class classifier and domain classifier.

Feature extractor is formed by 2 conv layers

Class classifier is formed by 3 fully connected layers

Domain classifier is formed by 2 fully connected layers and is fed with reversed feature output.

Optimizer is Adam.

And data are normalized without further augmentation.

What I learned from implementing DANN is that dataset with similar style (like MNIST-M and USPS) may have better adaptation, and dataset with different style (like MNIST-M and svhn, one is picture, the other is handwriting) may have poor adaptation.