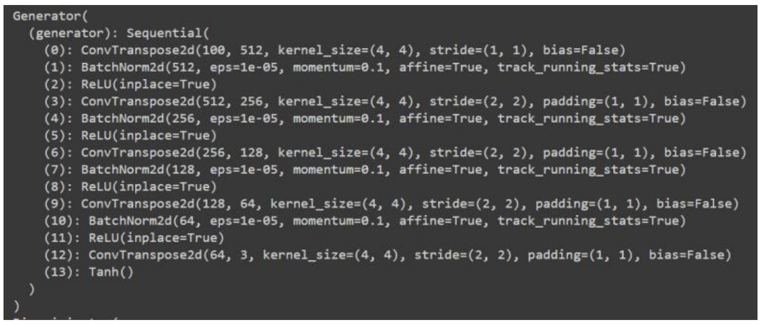
DLCV HW1

工科海洋四 B08505048 劉名凱

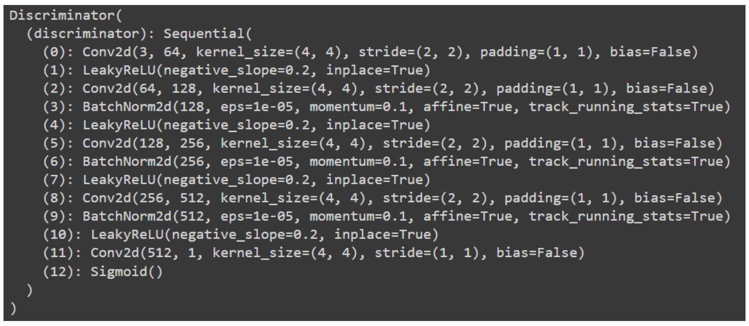
Problem 1

(1)

Method1

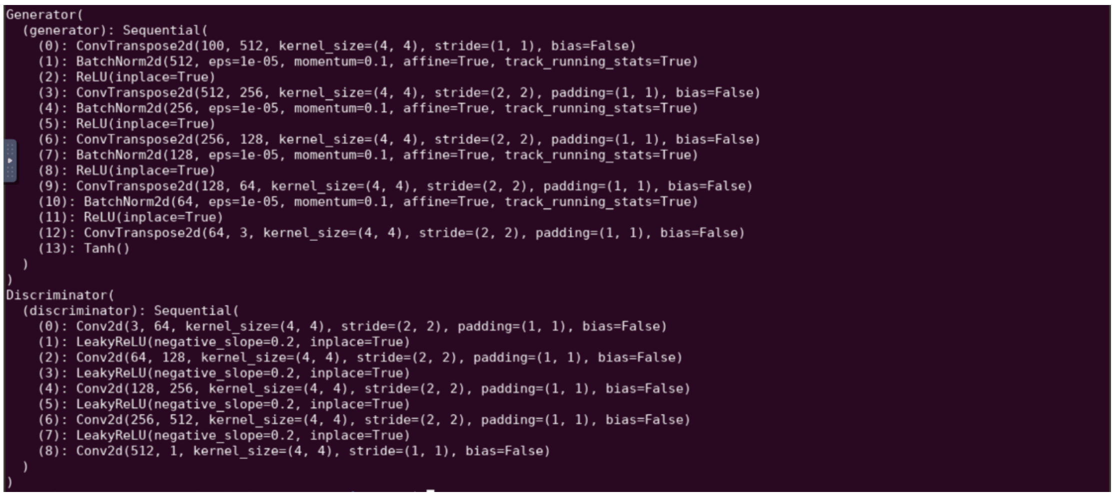


```
Generator(
  (generator): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 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()
  )
)
```



```
Discriminator(
  (discriminator): 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()
  )
)
```

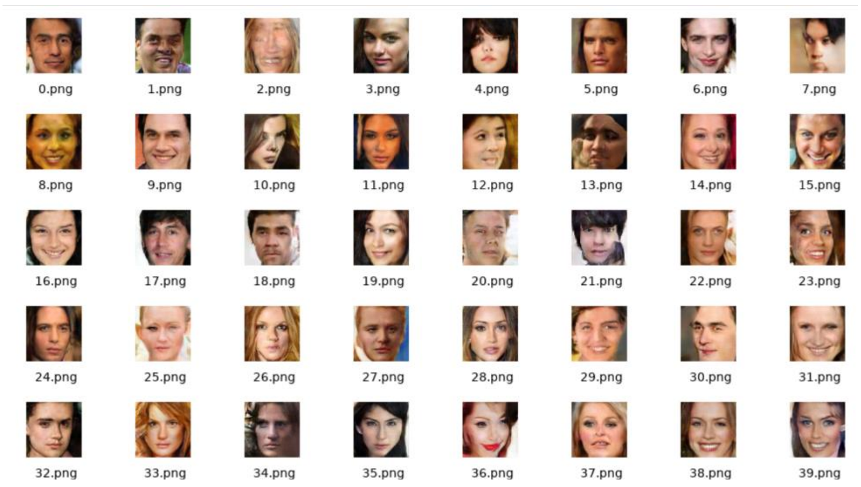
Method2



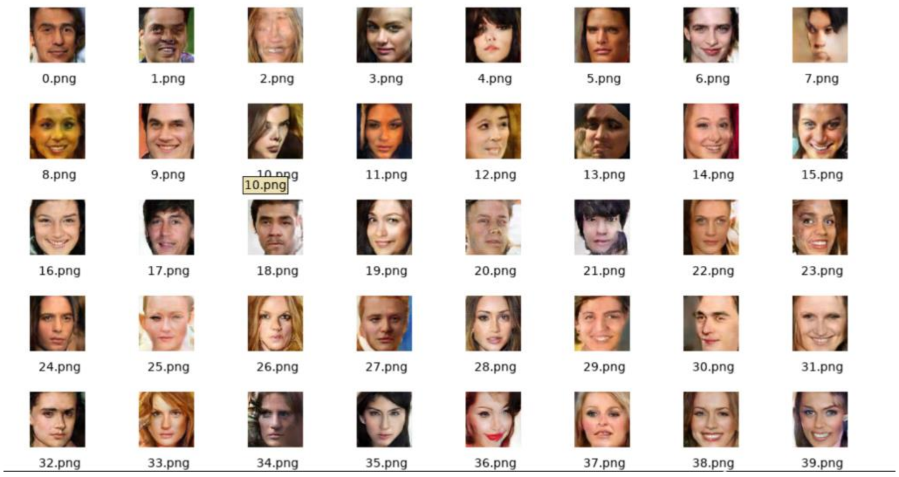
```
Generator(
  (generator): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 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()
  )
)
Discriminator(
  (discriminator): 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): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
)
```

(2)

Method 1 (DCGAN)



Method 2



Difference:

Method 1: I followed the network architecture and training procedure in the DCGAN paper. In generator, I used Transposed Convolution to upsample feature maps and used ReLU as nonlinearity function; I also used Batch Normalization between Transposed Convolution and ReLU. In discriminator, I used convolution to downsample feature maps and used LeakyReLU with slope=-0.2. I also used Batch Noramlization in discriminator.

For the loss function, both generator and discriminator use Binary Cross Entropy loss. Optimizer is Adam with betas = (0.5, 0.999), as mentioned in the paper.

Method 2: The network architecture is pretty much the same as method 1 (DCGAN). However, I removed Batch Normalization layer and Sigmoid layer in discriminator due to the assertion in the paper "*Improved Training of Wasserstein GANs*" that batch normalization and sigmoid are not sutiable with geadient penalty. For the loss function, I used Earth Mover Distance proposed in the paper "*Wasserstein GAN*", combining gradient penalty proposed in the paper "*Improved Training of Wasserstein GANs*".

The main difference between two method is the loss function. Though loss function in method 2 often result much better result, my DCGAN performed better. I believe the reason is I didn't follow the network architecture in paper "*Improved Training of Wasserstein GANs*" which implements Residual Block and Layer Normalization.

(3)

At first, I thought generator plays a more significant role in GAN, since generator is responsible for generating images. After serveral training experience, I observed that in the final stage of training of some failed cases, the discriminator fail to distinguish the difference between fake images and real images. As a result, the generator cease to improve anymore, which I believe is the main reason why method 2 doesn't performed well (fid =)

Besides, I found batch size significanly affect the quality of generated images. If batch size = 64, ouput images look like a mess. Batch size = 64 yields a better result. Thererfore, I set batch size = 64 for all the training I conducted. I also found that Critic iteration proposed in the paper "*Wasserstein GAN*" will ruin my DCGAN, everytime I implement the critic iteration in training processes, the quality of generated images would become worse and worse. I don't know why this would happen, but I guess that in our case (image_size = 64x64), discriminator does not need additional training iteration to successfully distinguish fake images and real images.

Problem 2

(1)

Architecture:

```
GaussianDiffusion(
  (model): Unet(
    (init_conv): Conv2d(3, 64, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
    (emb): Embedding(10, 256)
    (time_mlp): Sequential(
      (0): SinusoidalPosEmb()
      (1): Linear(in_features=64, out_features=256, bias=True)
      (2): GELU(approximate=none)
      (3): Linear(in_features=256, out_features=256, bias=True)
    )
    (downs): ModuleList(
      (0): ModuleList(
        (0): Resblock(
          (block1): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (block2): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (fc): Sequential(
            (0): SiLU()
            (1): Linear(in_features=256, out_features=128, bias=True)
          )
          (res_conv): Identity()
        )
        (1): Resblock(
          (block1): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (block2): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (fc): Sequential(
            (0): SiLU()
            (1): Linear(in_features=256, out_features=128, bias=True)
          )
          (res_conv): Identity()
        )
        (2): Residual(
          (fn): PreNorm(
            (fn): LinearAttention(
              (to_qkv): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
              (to_out): Sequential(
                (0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
                (1): LayerNorm()
              )
            )
            (norm): LayerNorm()
          )
        )
        (3): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      )
      (1): ModuleList(
        (0): Resblock(
          (block1): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (block2): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (fc): Sequential(
            (0): SiLU()
            (1): Linear(in_features=256, out_features=128, bias=True)
          )
          (res_conv): Identity()
        )
        (1): Resblock(
          (block1): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (block2): Block(
            (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
            (nonlinearity): SiLU()
          )
          (fc): Sequential(
            (0): SiLU()
            (1): Linear(in_features=256, out_features=128, bias=True)
          )
```

```
            (res_conv): Identity()
        )
        (2): Residual(
            (fn): PreNorm(
                (fn): LinearAttention(
                    (to_qkv): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
                    (to_out): Sequential(
                        (0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
                        (1): LayerNorm()
                    )
                )
                (norm): LayerNorm()
            )
        )
        (3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (2): ModuleList(
        (0): Resblock(
            (block1): Block(
                (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=256, bias=True)
            )
            (res_conv): Identity()
        )
        (1): Resblock(
            (block1): Block(
                (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=256, bias=True)
            )
            (res_conv): Identity()
        )
        (2): Residual(
            (fn): PreNorm(
                (fn): LinearAttention(
                    (to_qkv): Conv2d(128, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
                    (to_out): Sequential(
                        (0): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
                        (1): LayerNorm()
                    )
                )
                (norm): LayerNorm()
            )
        )
        (3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
    (3): ModuleList(
        (0): Resblock(
            (block1): Block(
                (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=512, bias=True)
            )
            (res_conv): Identity()
        )
        (1): Resblock(
            (block1): Block(
                (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=512, bias=True)
            )
            (res_conv): Identity()
        )
        (2): Residual(
      (fn): PreNorm(
        (fn): LinearAttention(
          (to_qkv): Conv2d(256, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (to_out): Sequential(
            (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
            (1): LayerNorm()
          )
        )
        (norm): LayerNorm()
      )
    )
    (3): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
)
(ups): ModuleList(
  (0): ModuleList(
    (0): Resblock(
      (block1): Block(
        (conv): Conv2d(768, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (block2): Block(
        (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (fc): Sequential(
        (0): SiLU()
        (1): Linear(in_features=256, out_features=1024, bias=True)
      )
      (res_conv): Conv2d(768, 512, kernel_size=(1, 1), stride=(1, 1))
    )
    (1): Resblock(
      (block1): Block(
        (conv): Conv2d(768, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (block2): Block(
        (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (fc): Sequential(
        (0): SiLU()
        (1): Linear(in_features=256, out_features=1024, bias=True)
      )
      (res_conv): Conv2d(768, 512, kernel_size=(1, 1), stride=(1, 1))
    )
    (2): Residual(
      (fn): PreNorm(
        (fn): LinearAttention(
          (to_qkv): Conv2d(512, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (to_out): Sequential(
            (0): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1))
            (1): LayerNorm()
          )
        )
        (norm): LayerNorm()
      )
    )
    (3): Sequential(
      (0): Upsample(scale_factor=2.0, mode=nearest)
      (1): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
  )
  (1): ModuleList(
    (0): Resblock(
      (block1): Block(
        (conv): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (block2): Block(
        (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (fc): Sequential(
        (0): SiLU()
        (1): Linear(in_features=256, out_features=512, bias=True)
      )
      (res_conv): Conv2d(384, 256, kernel_size=(1, 1), stride=(1, 1))
    )
    (1): Resblock(
      (block1): Block(
        (conv): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (block2): Block(
        (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (norm): GroupNorm(8, 256, eps=1e-05, affine=True)
        (nonlinearity): SiLU()
      )
      (fc): Sequential(
        (0): SiLU()
        (1): Linear(in_features=256, out_features=512, bias=True)
      )
      (res_conv): Conv2d(384, 256, kernel_size=(1, 1), stride=(1, 1))
```

```
        )
        (2): Residual(
            (fn): PreNorm(
                (fn): LinearAttention(
                    (to_qkv): Conv2d(256, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
                    (to_out): Sequential(
                        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
                        (1): LayerNorm()
                    )
                )
                (norm): LayerNorm()
            )
        )
        (3): Sequential(
            (0): Upsample(scale_factor=2.0, mode=nearest)
            (1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        )
    )
    (2): ModuleList(
        (0): Resblock(
            (block1): Block(
                (conv): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=256, bias=True)
            )
            (res_conv): Conv2d(192, 128, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): Resblock(
            (block1): Block(
                (conv): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 128, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=256, bias=True)
            )
            (res_conv): Conv2d(192, 128, kernel_size=(1, 1), stride=(1, 1))
        )
        (2): Residual(
            (fn): PreNorm(
                (fn): LinearAttention(
                    (to_qkv): Conv2d(128, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
                    (to_out): Sequential(
                        (0): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
                        (1): LayerNorm()
                    )
                )
                (norm): LayerNorm()
            )
        )
        (3): Sequential(
            (0): Upsample(scale_factor=2.0, mode=nearest)
            (1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        )
    )
    (3): ModuleList(
        (0): Resblock(
            (block1): Block(
                (conv): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=128, bias=True)
            )
            (res_conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): Resblock(
            (block1): Block(
                (conv): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (block2): Block(
                (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
            )
            (fc): Sequential(
                (0): SiLU()
```

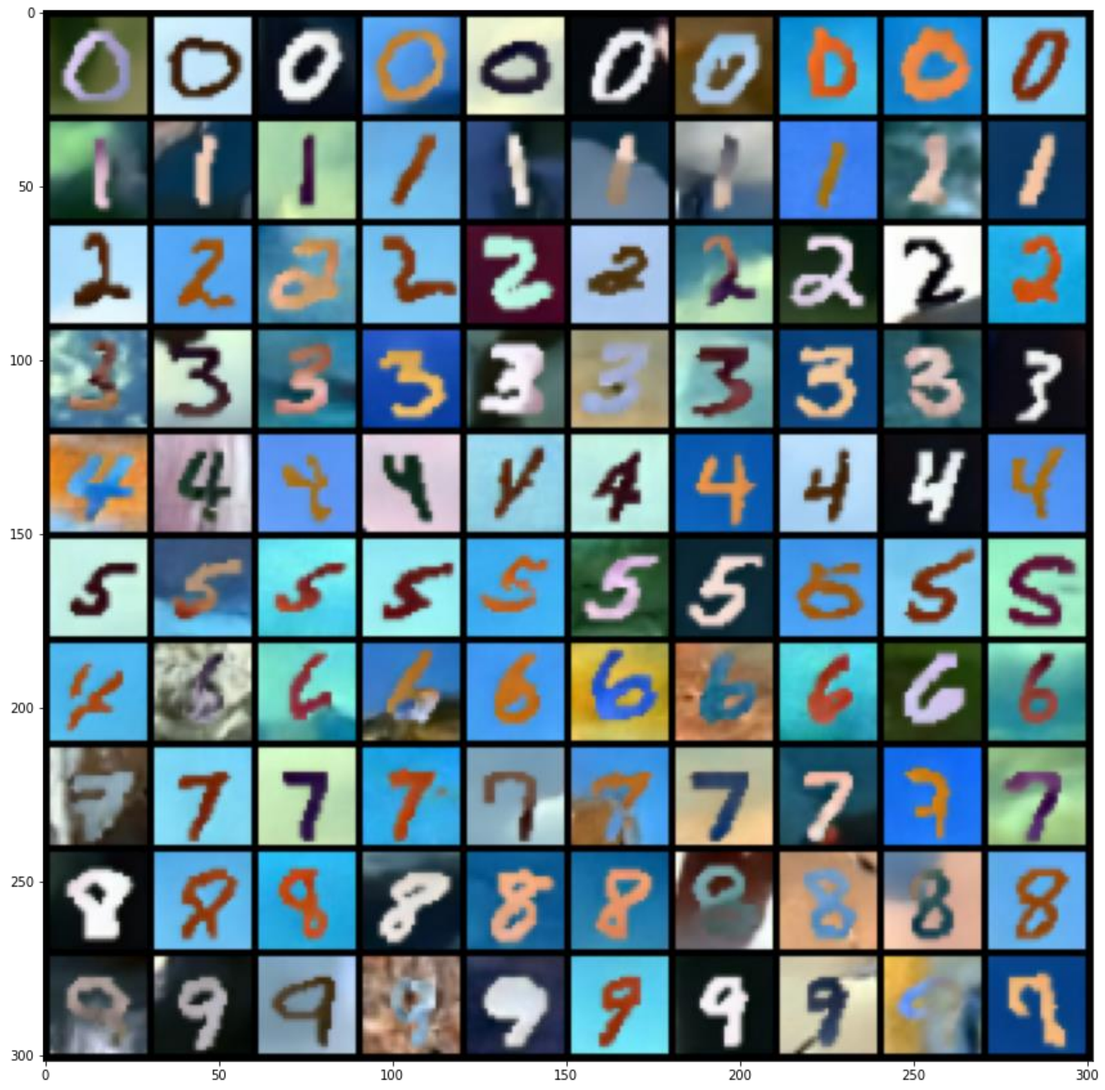
```
                    (1): Linear(in_features=256, out_features=128, bias=True)
                  )
                  (res_conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
                )
                (2): Residual(
                  (fn): PreNorm(
                    (fn): LinearAttention(
                      (to_qkv): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
                      (to_out): Sequential(
                        (0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
                        (1): LayerNorm()
                      )
                    )
                    (norm): LayerNorm()
                  )
                )
                (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              )
            )
            (mid_block1): Resblock(
              (block1): Block(
                (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
              )
              (block2): Block(
                (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
              )
              (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=1024, bias=True)
              )
              (res_conv): Identity()
            )
            (mid_attn): Residual(
              (fn): PreNorm(
                (fn): Attention(
                  (to_qkv): Conv2d(512, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
                  (to_out): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1))
                )
                (norm): LayerNorm()
              )
            )
            (mid_block2): Resblock(
              (block1): Block(
                (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
              )
              (block2): Block(
                (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 512, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
              )
              (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=1024, bias=True)
              )
              (res_conv): Identity()
            )
            (final_res_block): Resblock(
              (block1): Block(
                (conv): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
              )
              (block2): Block(
                (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (norm): GroupNorm(8, 64, eps=1e-05, affine=True)
                (nonlinearity): SiLU()
              )
              (fc): Sequential(
                (0): SiLU()
                (1): Linear(in_features=256, out_features=128, bias=True)
              )
              (res_conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
            )
            (final_conv): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
          )
        )
```

Above architecture is modified version of source code right here:

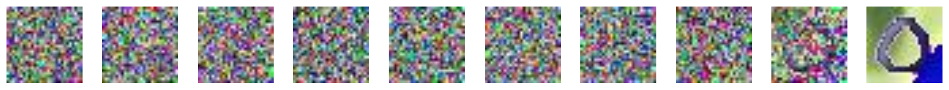
https://github.com/lucidrains/denoising-diffusion-pytorch

The original source code does not support conditional denoising, so I modified U-net into conditional version. I combine time step embedding in the U-net with label embedding (add them together). The label for each digit is 10-dimension vector with each entried equal to digits, e.g. [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] for digit 1. Then use MSE loss to compare the difference between generated digits and the ground truth digits. Batch size = 64, using Adam optimizer. Finally, the time step is set to 1000.

(2)



(3)



t = 100 200 300 400 500 600 700 800 900 1000

The first digits in (2), though it looks a little bit different

(4)

Although the denoising diffusion model is so complicated that I don't fully understand even right now, I did observer several things. The main observation is that denoising model is lot easier to train (at least for my case) compared to GAN. After I carefully follow the instruction in the original Denoising Diffusion Model paper, the quality of generated digits become perfect (the model can generate recognizable digit according to the label) only after few hours.

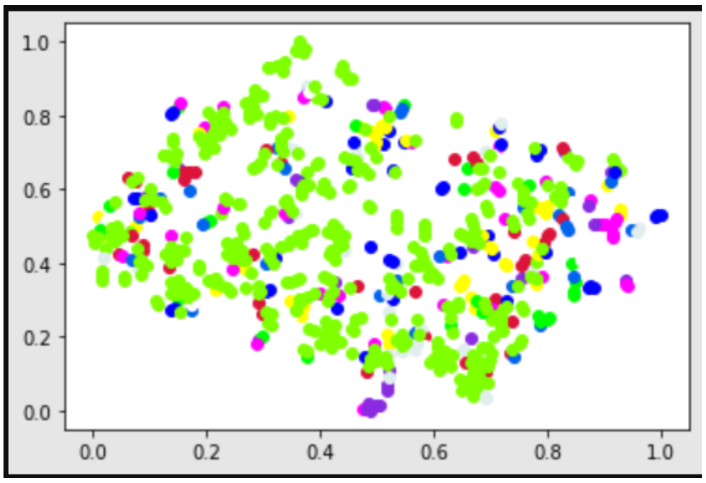
The second observation is that though denoising outperformed GAN, the time to generate images takes much longer than GAN. In problem 1, GAN can generate 1000 images in 10 seconds. On the contrary, the denoising model takes 10 minutes to generate same number of images.

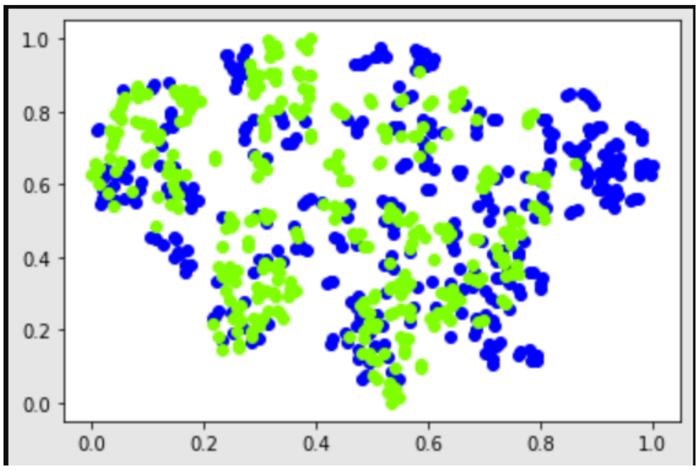
Problem 3

(1)

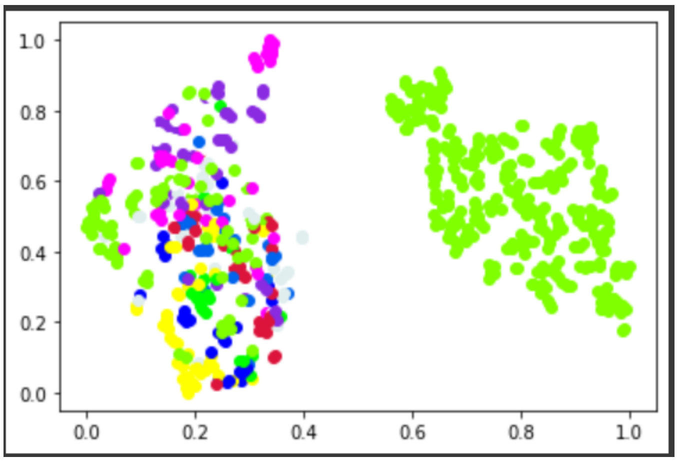
	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	0.3323	0.9
Adaptation (DANN)	0.5134	0.7634
Trained on target	0.8298	0.9832

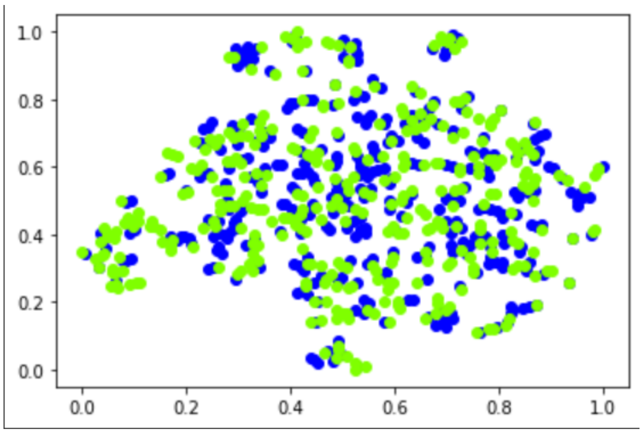
(2) class SVHN domain





Class USPS domain





SVHN DANN does not perform well on classifying, while the USPS DANN not performed well on Domain classifying

(3)

I followed the network architecture proposed in the original DANN paper. The feature extractor consist of Convolution layer and ReLU function. While the class classifier and domain classifier both only use linear layer and ReLU. (domain classifier last layer is Sigmoid in order to map output between 0 and 1) The batch size is set to 60, and the learning rate is 0.00001. The optimizer is RAdam.

The main observation is that when I train a classifier on Mnistm images and test it on USPS images, the classifier yields a 0.9 accuracy, which is better than DANN's result (0.7634). I believe this is because USPS is highly similar to Mnistm dataset. Both dataset's images consist of only one digit, unlike SVHN dataset. The SVHN dataset consist of more complex images than Mnistm, so a classifier which only trained on Mnistm images yields only 0.33 accuracy. When implementing DANN, the accuracy for SVHN increase to 0.5134, which is a huge process and prove the effectiveness of DANN.