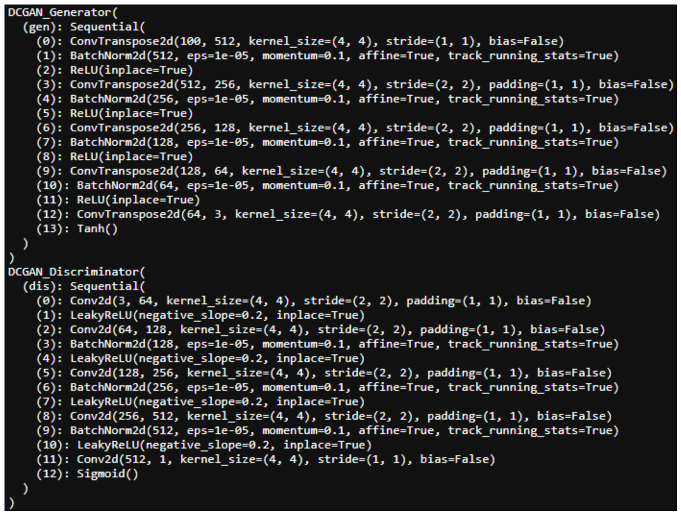
Deep Learning for Computer Vision HW2

Pin-Han, Huang: R10946003

October 30, 2022

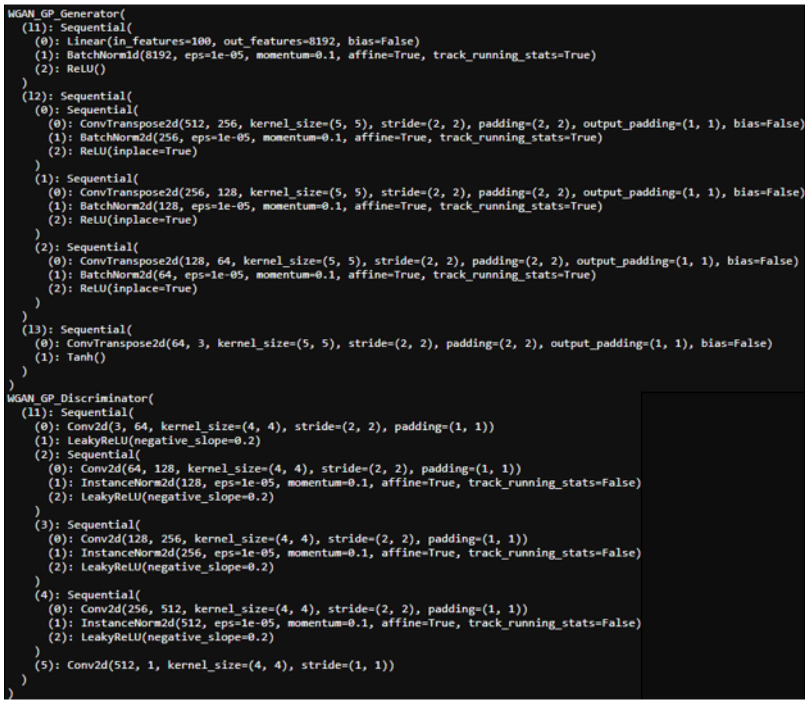
1 Problem1: GAN

1. **<u>Print the model architecture of method A</u>**



```
DCGAN_Generator(
  (gen): 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()
  )
)
DCGAN_Discriminator(
  (dis): 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()
  )
)
```

Figure 1: Model A: DCGAN model architecture.



```
WGAN_GP_Generator(
  (l1): Sequential(
    (0): Linear(in_features=100, out_features=8192, bias=False)
    (1): BatchNorm1d(8192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (l2): Sequential(
    (0): Sequential(
      (0): ConvTranspose2d(512, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (2): Sequential(
      (0): ConvTranspose2d(128, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (l3): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), output_padding=(1, 1), bias=False)
    (1): Tanh()
  )
)
WGAN_GP_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): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (3): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (4): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (1): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1))
  )
)
```

Figure 2: Model B: WGAN-GP model architecture.

2. **<u>Show the first 32 generated images of model A and B.</u>**

   - The images generated by model A is more clear than model B does in terms of good case. Both models perform badly in some images.

3. **<u>Discuss what you've observed and learned from implementing GAN.</u>**

   - Model A is based on DCGAN trained 200 epochs with batch size = 64, and label smoothing. In my experiments, smaller batch size, train longer and perform label smoothing helps to improve performance. Particularly, label smoothing mitigates the model

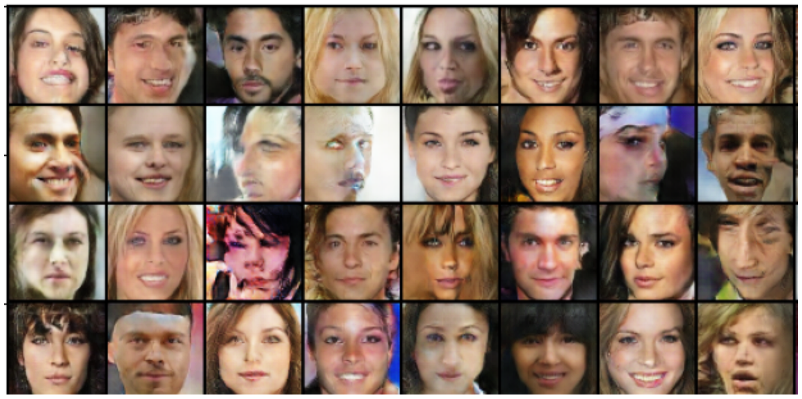


Figure 3: DCGAN generated images.

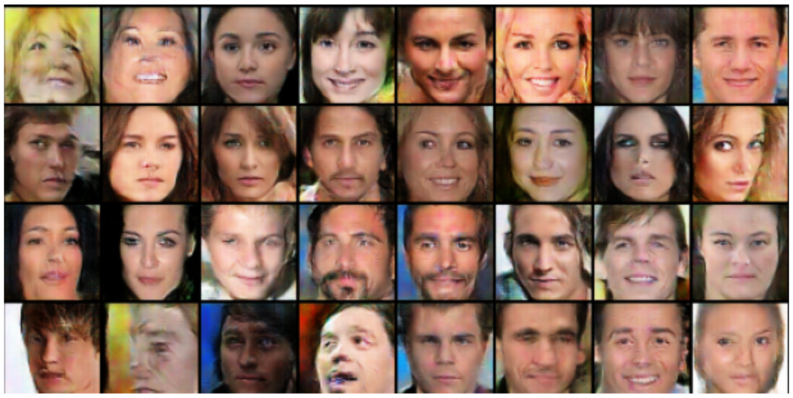


Figure 4: WGAN-GP generated images.

training difference between generator and discriminator. Model B is based on WGAN-GP, which assumes the critic to be a Lipschiz continuous function and limits the $l2-$norm of gradient near 1. Gradient penalty serves as a substitution to the weight clipping in WGAN.

2 Problem2: Conditional DDPM

1. <u>Print the model architecture</u>

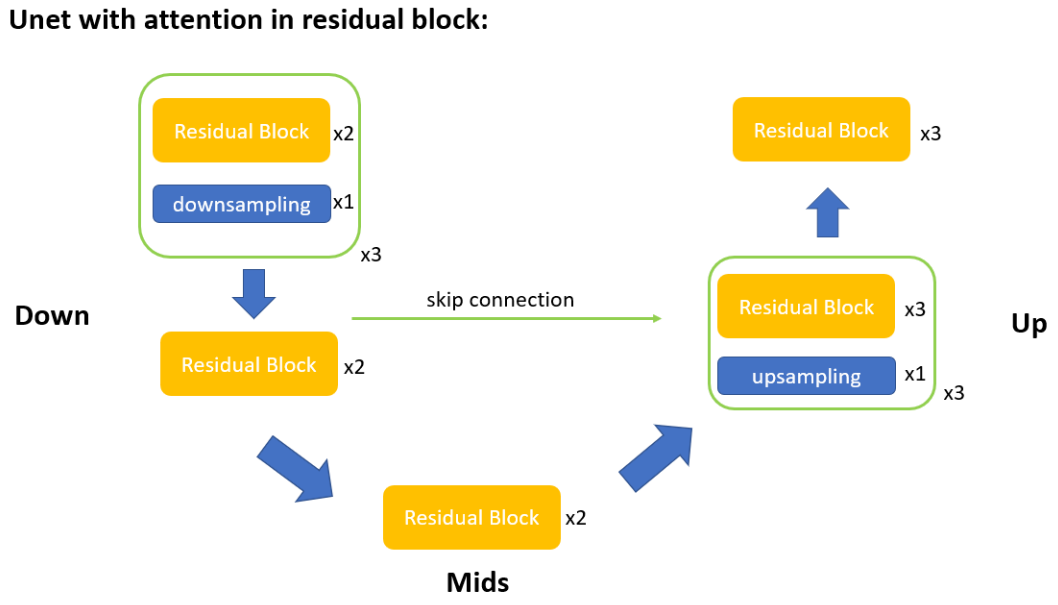


Figure 5: U-Net model with attention blocks.

- Conditional DDPM is performed with U-Net model where the encoding parts contains 8 residual blocks and 3 downsampling, the latent part contains 2 residual blocks, the decoder parts contains 12 residual blocks and 3 upsampling. Attention block is included in some residual blocks.

2. **Show 10 generated images for each digits.**

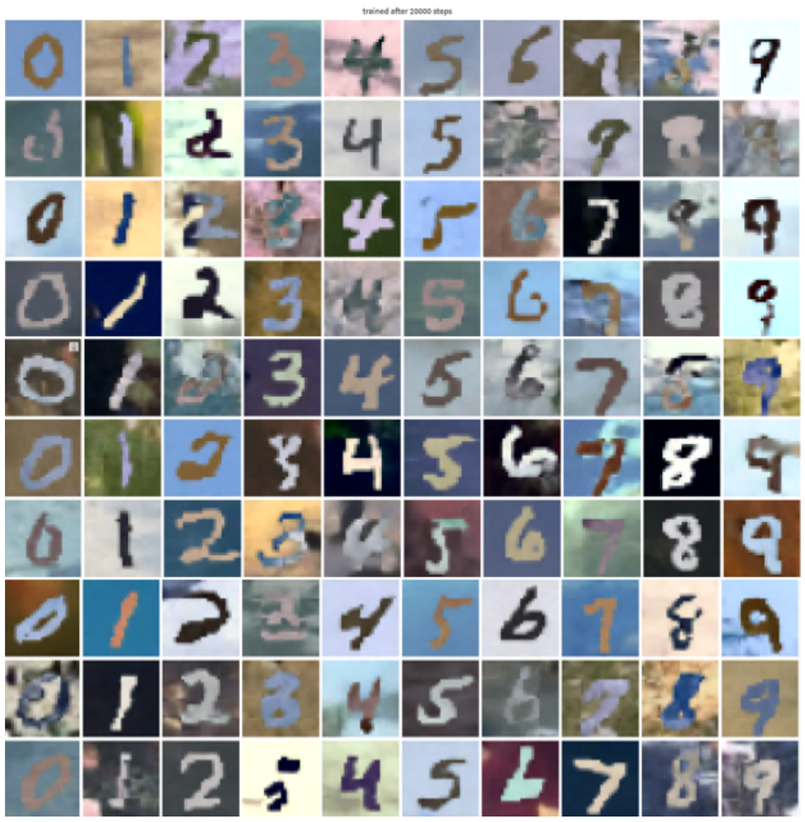


Figure 6: 10 generated images for each digits.

3. **Visualize total six images in the reverse process of 0 digit.**

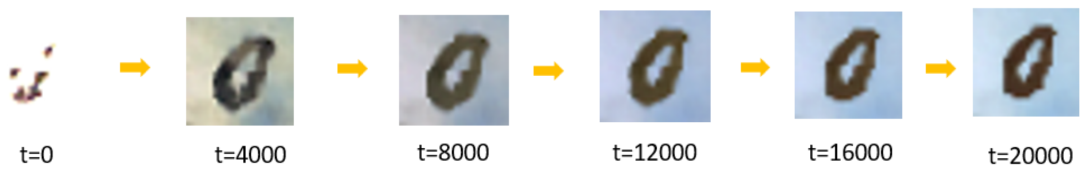


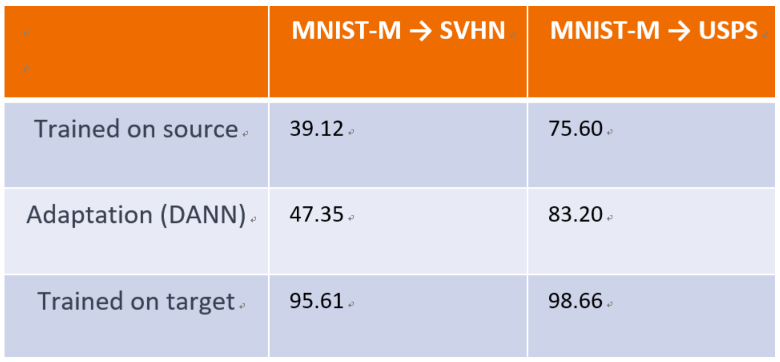
Figure 7: Digit zero at six time steps.

4. **Discuss what you've observed and learned from conditional DDPM.**

   - Conditional DDPM relies on using the given label to generate images. The model structure contains embedding of classes and is different from the original DDPM model. The model and time steps for conditional DDPM should be modified for different data, which in MNIST case, the model and time steps is relatively small.

3 Problem3: DANN

1. **DANN experiment results.**



	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	39.12	75.60
Adaptation (DANN)	47.35	83.20
Trained on target	95.61	98.66

Figure 8: DANN experiment results.

2. **Visualize the latent space of images to 2d space via t-SNE.**

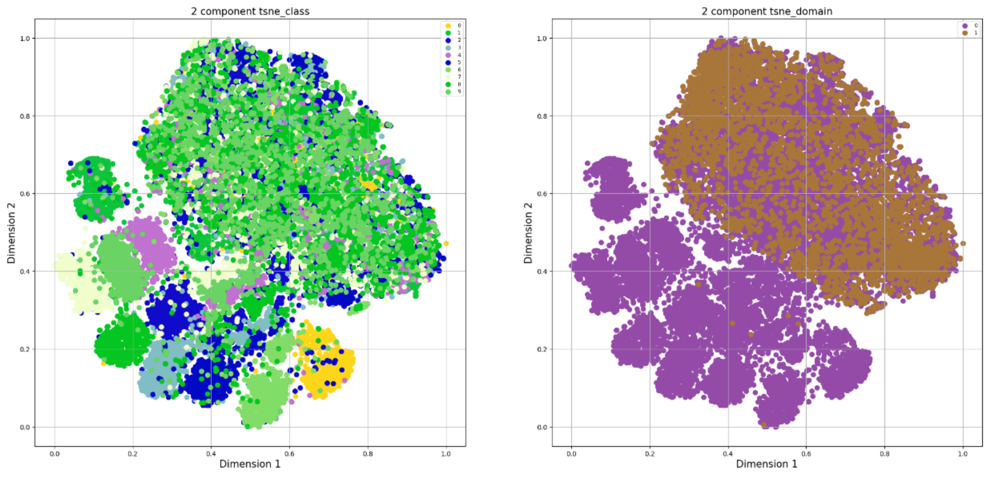


Figure 9: SVHN t-SNE results.

3. **Discuss what you've observed and learned from DANN.**

   - DANN is a transfer learning algorithm composed of three parts, including feature extractor, label classifier, and domain discriminator. The aim is to align two different data distributions into one. The feature extractor I perform is VGG-like form, while the label classifier and domain discriminator are simply MLPs.

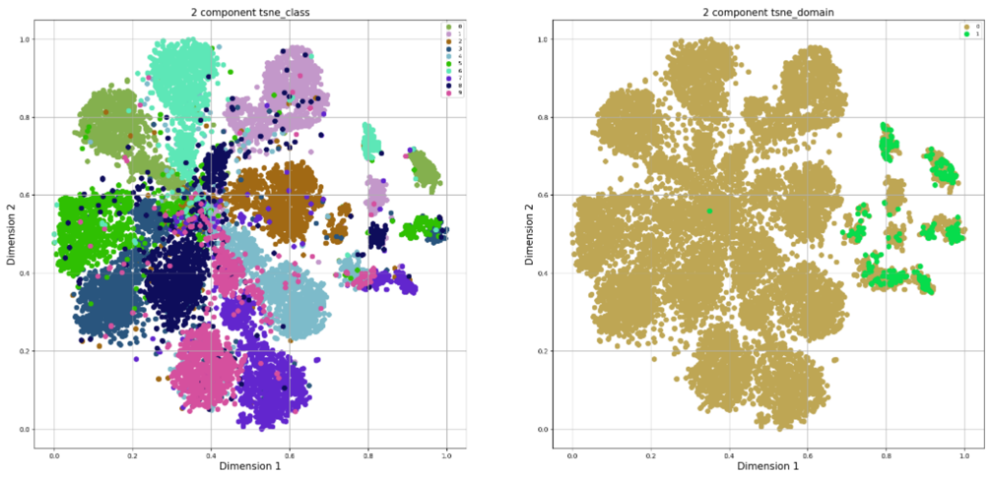


Figure 10: USPS t-SNE results.

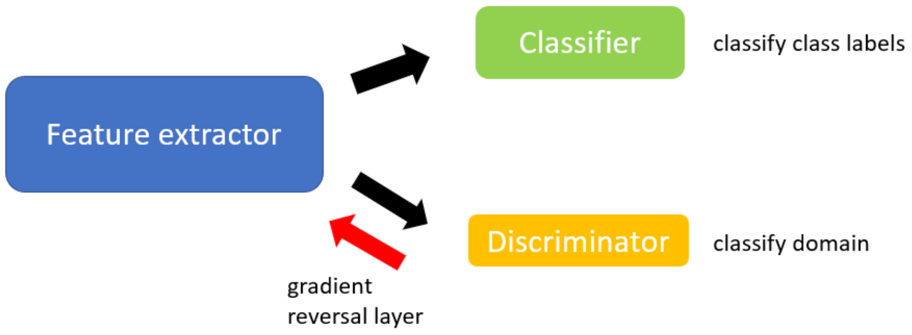


Figure 11: DANN training.