# DLCV Hw2 Report

R10942198 林仲偉

# Problem 1: Report (15%)

1. (5%) Please print the model architecture of method A and B.

Method A: DCGAN – Generator:

```
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)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

# Problem 1: Report (15%)

Method A: DCGAN – Discriminator:

```
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()
  )
)
```

# Problem 1: Report (15%)

Method B: Improved GAN (**WGAN-GP**) – Generator

```
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)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

# Problem 1: Report (15%)

Method B: Improved GAN (**WGAN-GP**) – Discriminator:

```
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): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (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): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (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): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
)
```

# Problem 1: Report (15%)

2. (5%) Please show the first 32 generated images of both method A (DCGAN) and method B (WGAN-GP) then discuss the difference between method A and B.

Method A (DCGAN)

Method B (WGAN-GP)

Difference:

(1) The last layer of discriminator in DCGAN is sigmoid layer, which is not used in WGAN.

(2) WGAN uses InstanceNorm2d() as normalization, while DCGAN uses BatchNorm2d().

(3) The loss of DCGAN is binary cross entropy (which contains log computation). The loss of WGAN-GP is the mean of discriminator outputs.

(4) WGAN uses gradient penalty to constrain the parameter of discriminator, while DCGAN has no constraint for its discriminator.

Ref: https://zhuanlan.zhihu.com/p/25071913

# Problem 1: Report (15%)

3. (5%) Please discuss what you've observed and learned from implementing GAN.

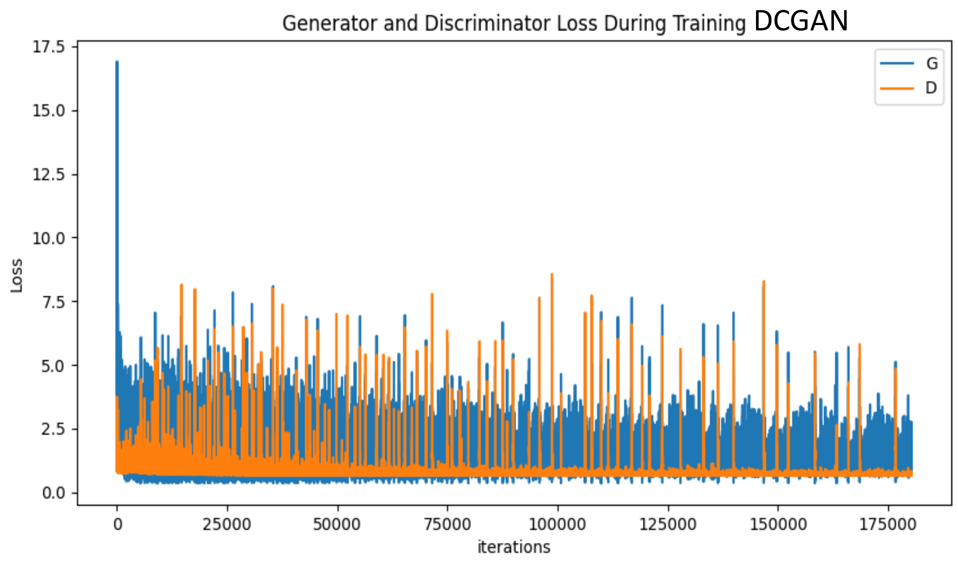
(1) In DCGAN, the loss curve is not very stable. The loss of discriminator will suddenly increase.

(2) I use 2 tricks during training GAN. Both of the tricks make discriminator task more difficult and prevent the discriminator being too strong.

- Soft labeling (0.9 for true image and 0.1 for fake image)
- Adding noise (0.01*normal distribution) when training discriminator.

(3) The performance of adversarial networks is not necessary better when the training step increases.

(4) Remember to use png to save file instead of jpg.



# Problem 2: Report (20%)

1. (5%) Please print your model architecture and describe your implementation details.

```
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)
      )
    )
```

The architecture of Unet-based DDPM

```
    (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))
)
```

Ref: [https://github.com/TeaPearce/Conditional_Diffusion_MNIST/blob/main/script.py](https://github.com/TeaPearce/Conditional_Diffusion_MNIST/blob/main/script.py)

# Problem 2: Report (20%)

## Some implementation details:

**ResidualConvBlock():**

- It contains 2 convolutional layers which are composed of regular convolution, batch norm layer and GELU().
- For the last convolutional layer in this block, the input and output channel number are the same. Therefore the residual is guaranteed to be computed.

**UnetDown():**

- Up-sampling block of Unet. This block is composed of ResidualConvBlock().

**UnetUp():**

- Up-sampling block of Unet. This block is composed of ResidualConvBlock().
- Like Unet: the feature of corresponding UnetDown() layer is concatenated during the up-sampling.

**EmbedFC():**

- Context & time embeddings. The embeddings are created by one layer fully connected network with activation function GELU().
- It is added and multiplied with the feature during the up-sampling of last two Unet layers.

Ref: https://github.com/TeaPearce/Conditional_Diffusion_MNIST/blob/main/script.py

# Problem 2: Report (20%)

## Some implementation details:

**ddpm_schedules():**

- Computes the necessary terms for DDPM sampling  and training process.
- Training:

```
x_t = (self.sqrtab[_ts, None, None, None] * x + self.sqrtmab[_ts, None, None, None] * noise)
```

$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$

- sampling:

```
x_i=(self.oneover_sqrta[i] * (x_i-eps * self.mab_over_sqrtmab[i]) + self.sqrt_beta_t[i]*z)
```

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$$

**DDPM():**

- Main module to connect all of the blocks.

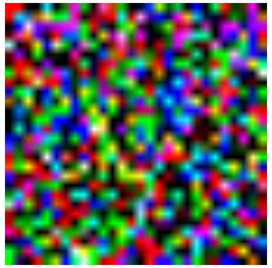
Ref: https://github.com/TeaPearce/Conditional_Diffusion_MNIST/blob/main/script.py, course slides

# Problem 2: Report (20%)

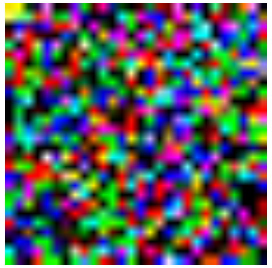
2. (5%) Please show 10 generated images **for each digit (0-9)** in your report. You can put all 100 outputs in one image with columns indicating different noise inputs and rows indicating different digits.

# Problem 2: Report (20%)

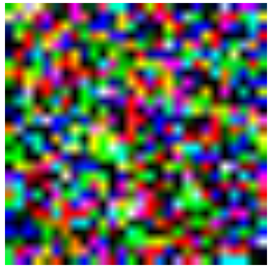
3. (5%) Visualize total six images in the reverse process of the first “0” in your grid in (2) with different time steps.



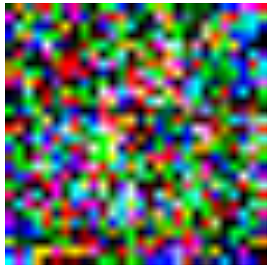
t=0



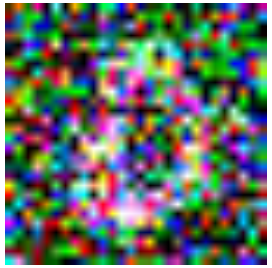
t=80



t=160



t=240



t=320

t=399

t=400

# Problem 2: Report (20%)

4. (5%) Please discuss what you've observed and learned from implementing conditional diffusion model.

- Diffusion model takes really a long inference time to generate image.
- During sampling process, the value of $\epsilon_\theta$ is determined by guidance w. When w is larger, the generated images contain less gray-scale-like images and has more diversity.

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$$

(Ref: Classifier-Free Diffusion Guidance)

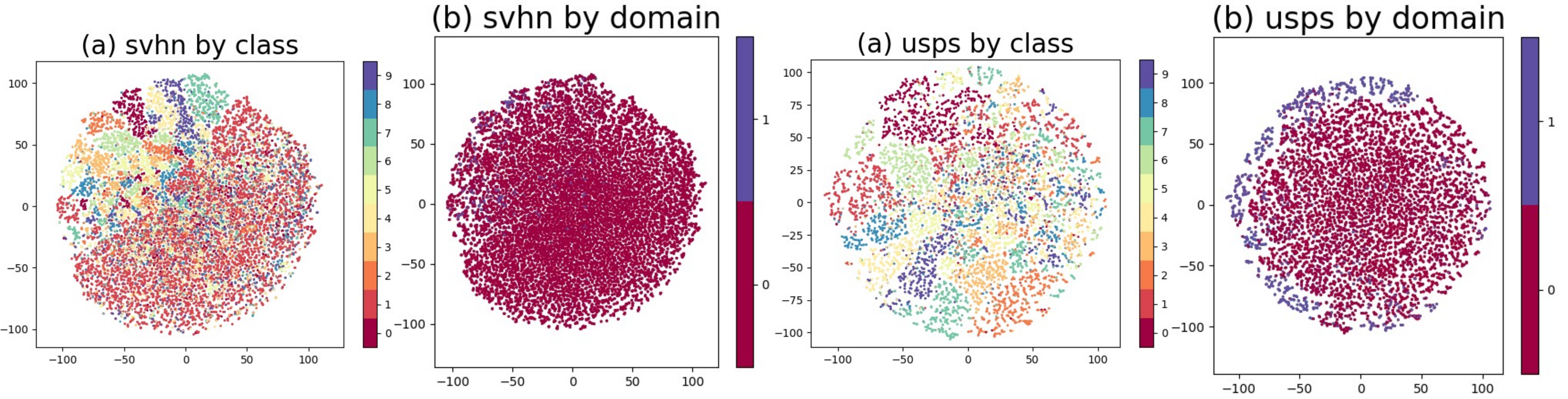
# Problem 3: Report (23%)

1. (5%) Please create and fill the table with the following format **in your report**:

| | MNIST-M → SVHN | MNIST-M → USPS |
|---|---|---|
| Trained on source | 0.337 | 0.760 |
| Adaptation (DANN) | 0.496 | 0.826 |
| Trained on target | 0.915 | 0.983 |

# Problem 3: Report (23%)

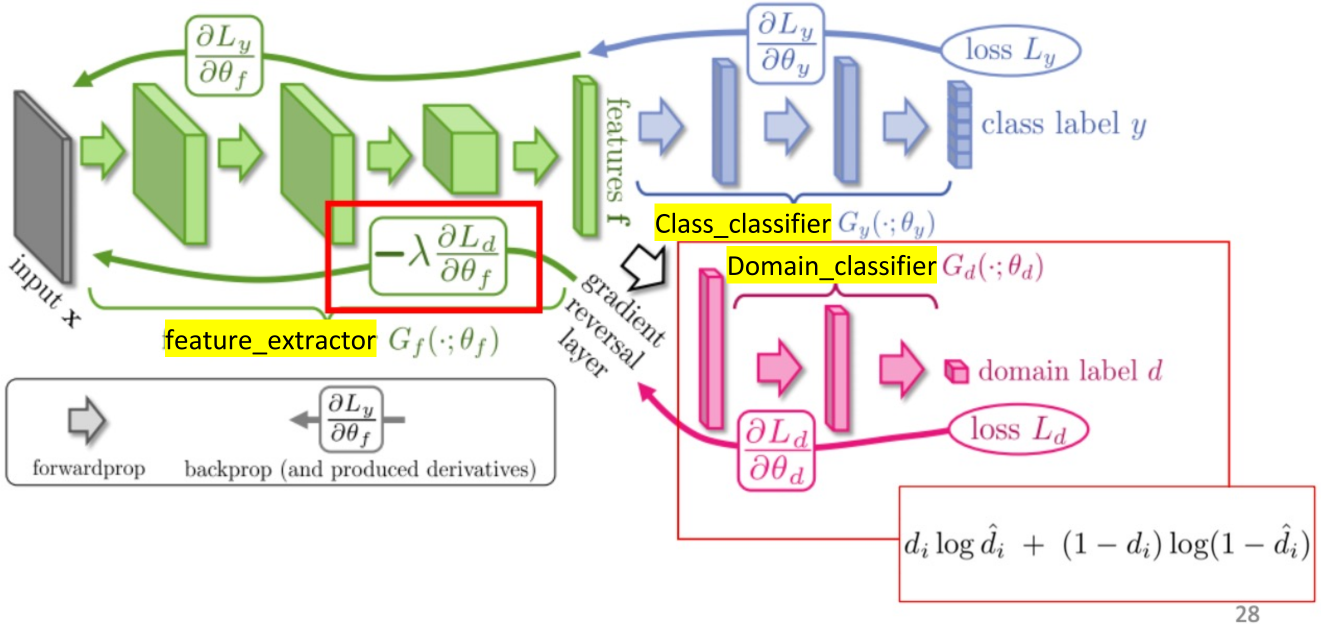
2. (8%) Please visualize the latent space of DANN by mapping the ***validation*** images to 2D space with t-SNE. For each scenario, you need to plot two figures which are colored **by digit class (0-9)** and **by domain**, respectively.



# Problem 3: Report (23%)

3. (10%) Please describe the implementation details of your model and discuss what you’ve observed and learned from implementing DANN.

```
CNNModel(
  (feature): Sequential(
    (f_conv1): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (f_bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (f_pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (f_relu1): ReLU(inplace=True)
    (f_conv2): Conv2d(64, 50, kernel_size=(5, 5), stride=(1, 1))
    (f_bn2): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (f_drop1): Dropout2d(p=0.5, inplace=False)
    (f_pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (f_relu2): ReLU(inplace=True)
  )
  (class_classifier): Sequential(
    (c_fc1): Linear(in_features=800, out_features=100, bias=True)
    (c_bn1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (c_relu1): ReLU(inplace=True)
    (c_drop1): Dropout2d(p=0.5, inplace=False)
    (c_fc2): Linear(in_features=100, out_features=100, bias=True)
    (c_bn2): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (c_relu2): ReLU(inplace=True)
    (c_fc3): Linear(in_features=100, out_features=10, bias=True)
    (c_softmax): LogSoftmax(dim=None)
  )
  (domain_classifier): Sequential(
    (d_fc1): Linear(in_features=800, out_features=100, bias=True)
    (d_bn1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (d_relu1): ReLU(inplace=True)
    (d_fc2): Linear(in_features=100, out_features=2, bias=True)
    (d_softmax): LogSoftmax(dim=1)
  )
)
```




Ref. Y. C. Wang. DLCV week 6 course slides p.28

# Problem 3: Report (23%)

## Some implementation details:

**self.feature: Feature extractor**

- It is composed of 2 blocks of convolutional layer, batch norm layer, max pooling layer, dropout layer, and ReLU().
- Output: 50 x 4 x 4 feature vector.

**self.class_classifier: Classifier**

- It is composed of 2 blocks of fully connected layer, batch norm layer, dropout layer, and ReLU().
- Output: Softmax.

**self.domain_classifier: Maximize domain confusion**

- It is composed of fully connected layer, batch norm layer, ReLU(), and fully connected layer.
- Output: Softmax.

**ReverseLayerF(): Gradient reversal layer**

- Compute reversal feature with given feature and coefficient.
- The output is then forwarded to domain classifier, and the gradient is back propagated to feature extractor.

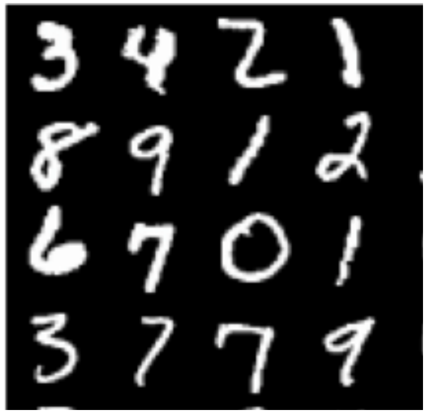
# Problem 3: Report (23%)

## Observed and learned from implementing DANN:

1. DANN contains adversarial module (the gradient reverse layer), so training for longer time doesn't imply better performance.
2. The prediction result of DANN model on usps dataset is better than svhn dataset. The possible reason is that mnist-m is more similar to usps dataset than svhn. (usps dataset is just the gray-scale version of mnist-m, but svhn dataset is the real version of mnist-m dataset.)

svhn dataset

mnist-m dataset



usps dataset