DLCV HW2
r10921a36 石子仙

- # Part 1

The part1 code is revise from my homework in machine learning 2021 which is lectured by Prof. Hung-Yi Lee.

1. (5%) Please print the model architecture of method A and B.
   You can use “print(model)” directly.

DCGAN

```
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_5): 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()
    )
    (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()
    )
    (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()
```

```
        )
        (3): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2),
output_padding=(1, 1))
        (4): Tanh()
    )
)
Discriminator(
    (ls): Sequential(
        (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        (1): LeakyReLU(negative_slope=0.2)
        (2): Sequential(
            (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
            (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=(5, 5), stride=(2, 2), padding=(2, 2))
            (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=(5, 5), stride=(2, 2), padding=(2, 2))
            (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))
        (6): Sigmoid()
    )
)
```

WGAN

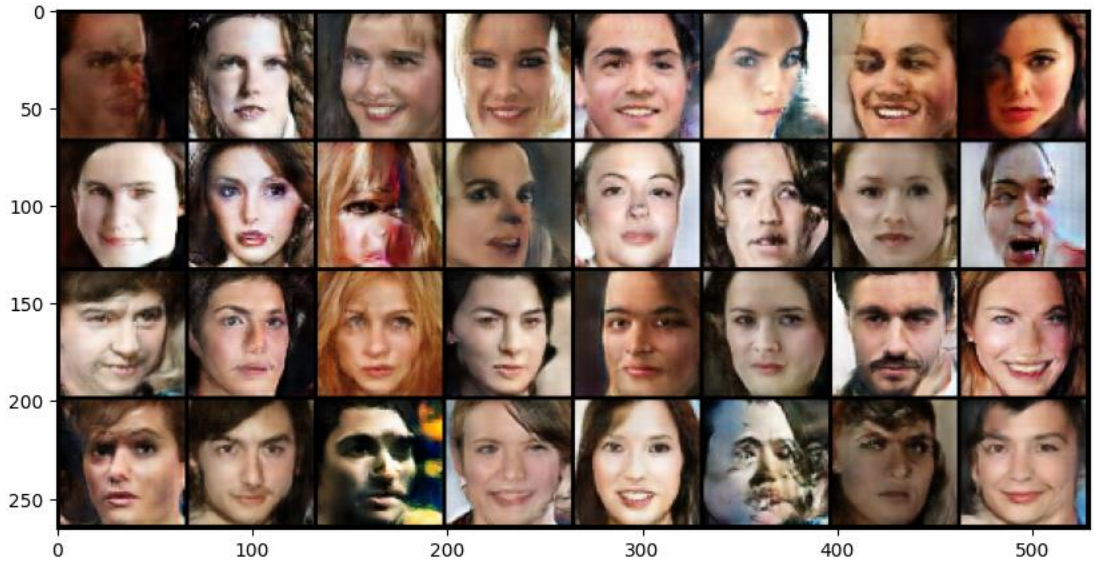
```
Generator(
    (l1): Sequential(
        (0): Linear(in_features=100, out_features=8192, bias=False)
```

```
      (1): BatchNorm1d(8192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
   )
   (l2_5): 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()
      )
      (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()
      )
      (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()
      )
      (3): ConvTranspose2d(64, 3, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2),
output_padding=(1, 1))
      (4): Tanh()
   )
)
Discriminator(
   (ls): Sequential(
      (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
      (1): LeakyReLU(negative_slope=0.2)
      (2): Sequential(
         (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
         (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

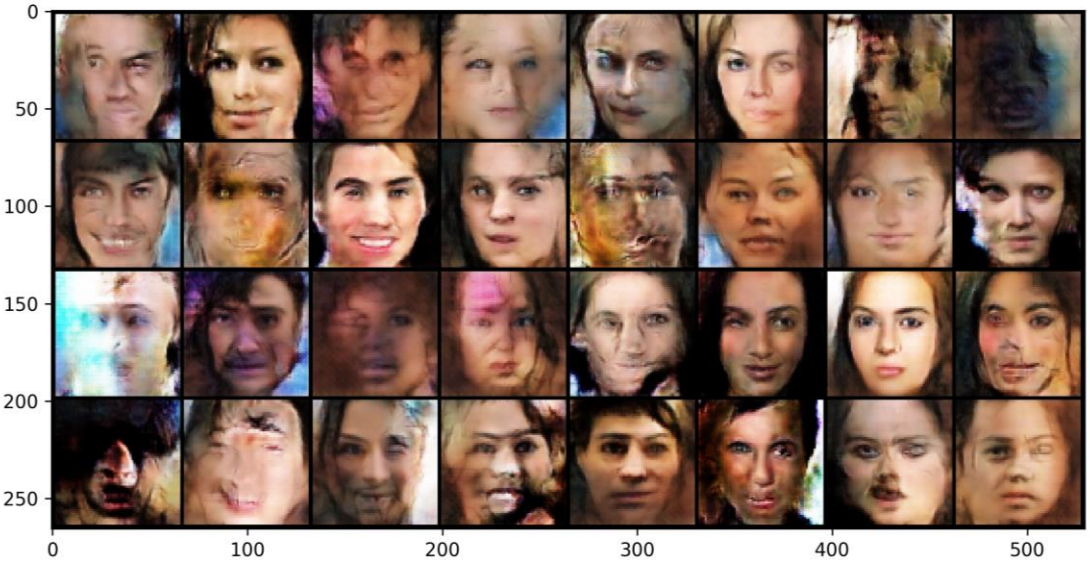
```
track_running_stats=True)
          (2): LeakyReLU(negative_slope=0.2)
        )
        (3): Sequential(
          (0): Conv2d(128, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): LeakyReLU(negative_slope=0.2)
        )
        (4): Sequential(
          (0): Conv2d(256, 512, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
          (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. (5%) Please show the first 32 generated images of both method A and B then discuss the difference between method A and B.

   DCGAN



   WGAN



3. (5%) Please discuss what you've observed and learned from implementing GAN. You can compare different architectures or describe some difficulties during training, etc.

I think the training process of GAN is not very stable and not that easy, sometime I change just batch size then the output became really bad. Or the loss function will become very big or small, or stuck in some place when training too long. I also learned that the models that be thought powerful GAN are not always good at every domain. Beside DCGAN, I've tried WGAN and SNGAN, which is not good as DCGAN. Training GAN is not an easy thing.

● Part 2

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

```
DDPM(
  (nn_model): ContextUnet(
    (init_conv): ResidualConvBlock(
      (conv1): Sequential(
        (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): GELU(approximate=none)
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
            (1): BatchNorm2d(128, 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(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
                (2): GELU(approximate=none)
              )
              (conv2): Sequential(
                (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (1): BatchNorm2d(128, 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(128, 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)
    )
)
(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=256, bias=True)
        (1): GELU(approximate=none)
        (2): Linear(in_features=256, out_features=256, bias=True)
    )
)
(timeembed2): EmbedFC(
    (model): Sequential(
        (0): Linear(in_features=1, out_features=128, bias=True)
        (1): GELU(approximate=none)
        (2): Linear(in_features=128, out_features=128, bias=True)
    )
)
(contextembed1): 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)
    )
)
(contextembed2): EmbedFC(
    (model): Sequential(
        (0): Linear(in_features=10, out_features=128, bias=True)
        (1): GELU(approximate=none)
        (2): Linear(in_features=128, out_features=128, bias=True)
    )
)
(up0): Sequential(
```

```
        (0): ConvTranspose2d(256, 256, kernel_size=(7, 7), stride=(7, 7))
        (1): GroupNorm(8, 256, eps=1e-05, affine=True)
        (2): ReLU()
      )
      (up1): UnetUp(
        (model): Sequential(
          (0): ConvTranspose2d(512, 128, kernel_size=(2, 2), stride=(2, 2))
          (1): ResidualConvBlock(
            (conv1): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
            (conv2): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
          )
          (2): ResidualConvBlock(
            (conv1): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
            (conv2): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
          )
        )
      )
      (up2): UnetUp(
```

```
        (model): Sequential(
          (0): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
          (1): ResidualConvBlock(
            (conv1): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (2): GELU(approximate=none)
            )
            (conv2): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
          )
          (2): ResidualConvBlock(
            (conv1): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
            (conv2): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
              (2): GELU(approximate=none)
            )
          )
        )
      )
      (out): Sequential(
        (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): GroupNorm(8, 128, eps=1e-05, affine=True)
        (2): ReLU()
        (3): Conv2d(128, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
    )
```

```
    (loss_mse): MSELoss()
```

The code refers from open source. The model is implemented with UNet architecture, the denoise step is set to 400, learning rate 0.0001, the optimizer is Adam.

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. [see the below example]

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

t	0	80	160	240	320	400
image						

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

I think the implementing of conditional diffusion model needs fully understand the model architecture and how it works. It is the most difficult part in this homework for me, I also learned that the larger feature and the step, the better performance we get.

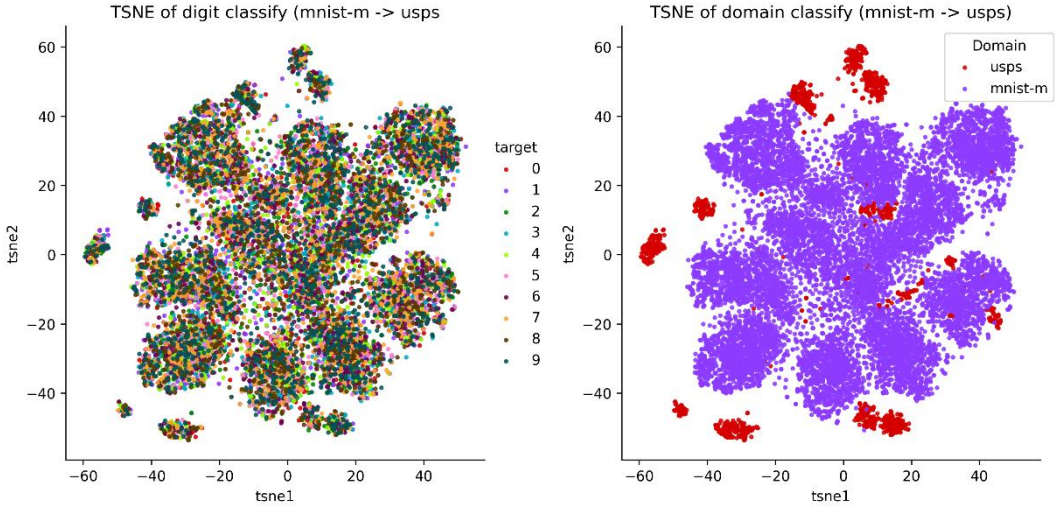
● Part 3

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

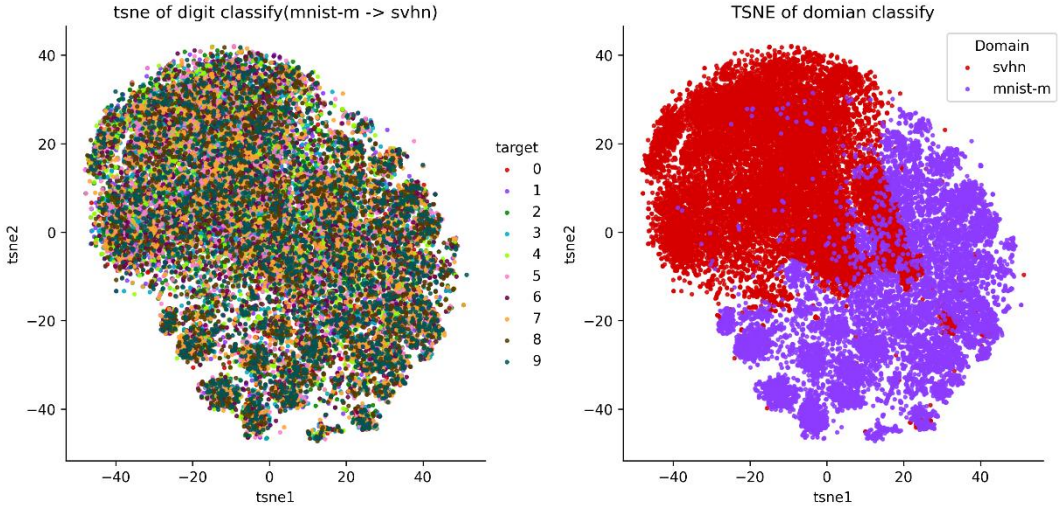
	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	0.4348	0.7695
Adaptation (DANN)	0.5280	0.8185
Trained on target	0.9175	0.9866

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

- MNIST-M -> USPS



- MNIST-M -> SVHN

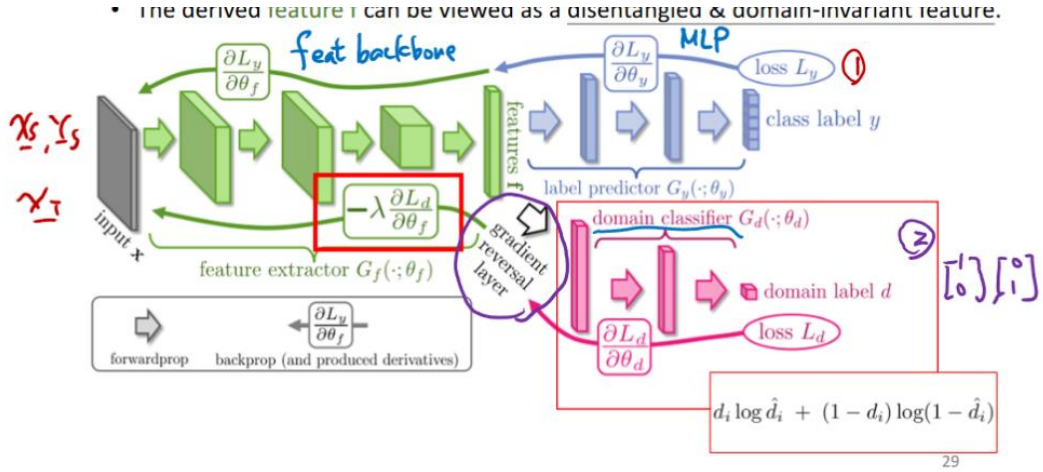


I thought my visual result is not good. But I checked the visualize code and my inference result of validation set, I didn't see anything wrong.

3. (10%) Please describe the implementation details of your model and discuss

what you've observed and learned from implementing DANN.

The code refers from open source. The feature extractor's architecture is simply two layers of convolution. The DANN architecture is similar to the architecture of the lecture note. (figure below)



For what I've learned, I think the DANN is a simple but powerful and useful model. Each component of the model is not complicated as I thought, but it has a lot of help. And during the implement, I found that if I want to split the DANN to three part, training the three different modules at the same time, it will have bad performance comparing to putting them into one model. I'm wondering why it happened, but I couldn't find it for now.

Reference:

- Part1:
  1. Prof. Hung-Yi Lee ML 2021 material, https://speech.ee.ntu.edu.tw/~hylee/ml/2021-spring.php
- Part2:
  1. Github repo, https://github.com/TeaPearce/Conditional_Diffusion_MNIST
- Part3:
  1. Github repo, https://github.com/CuthbertCai/pytorch_DANN