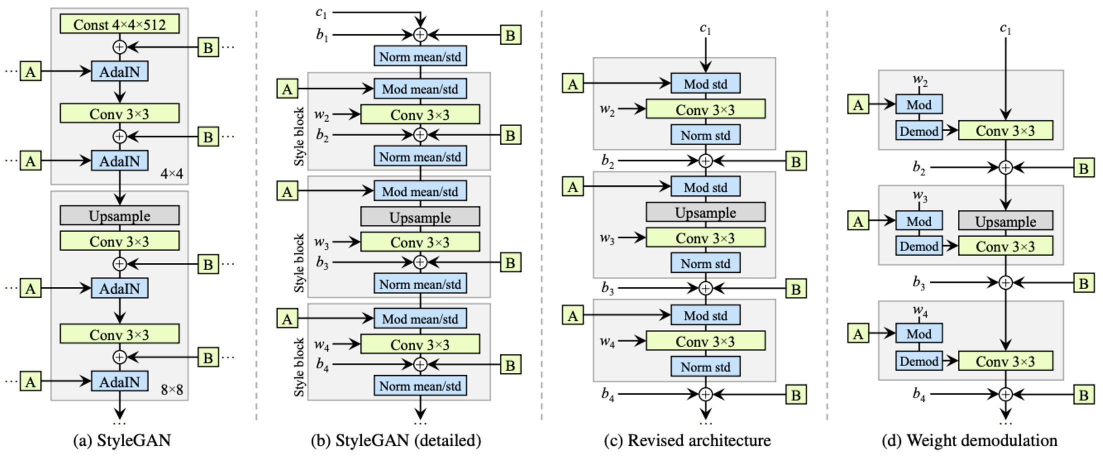
DLCV HW2 Report

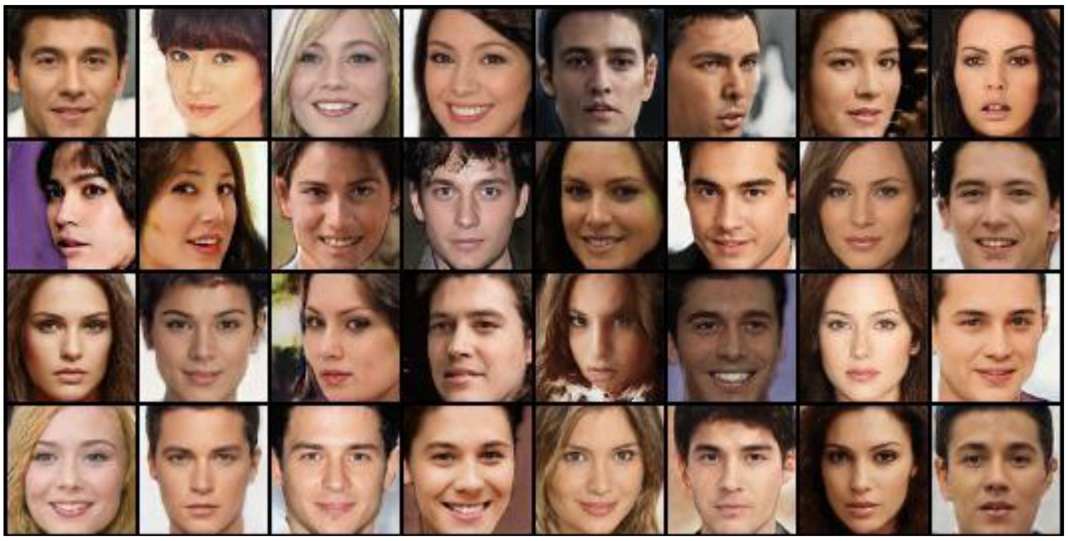
B07502071 陳志臻

P1-GAN

1. Model architecture (I put it in the end of report)



2. First 32 images



3&4. Results

```
FID:  28.158576252243705
Inception Score: 2.001824013042993
```

5. Implementing GAN

I chose to use the stylegan2 model architecture. I trained my model for 68000 steps on colab pro with P100 GPU for about 22 hours. Besides, I added one attention-layer in my training. According to my collaborators' experience, if I didn't use the attention-layer, it would be hard to pass the original baseline. We also found that if the model architecture is too big, it would be hard to be trained well.

P2-ACGAN

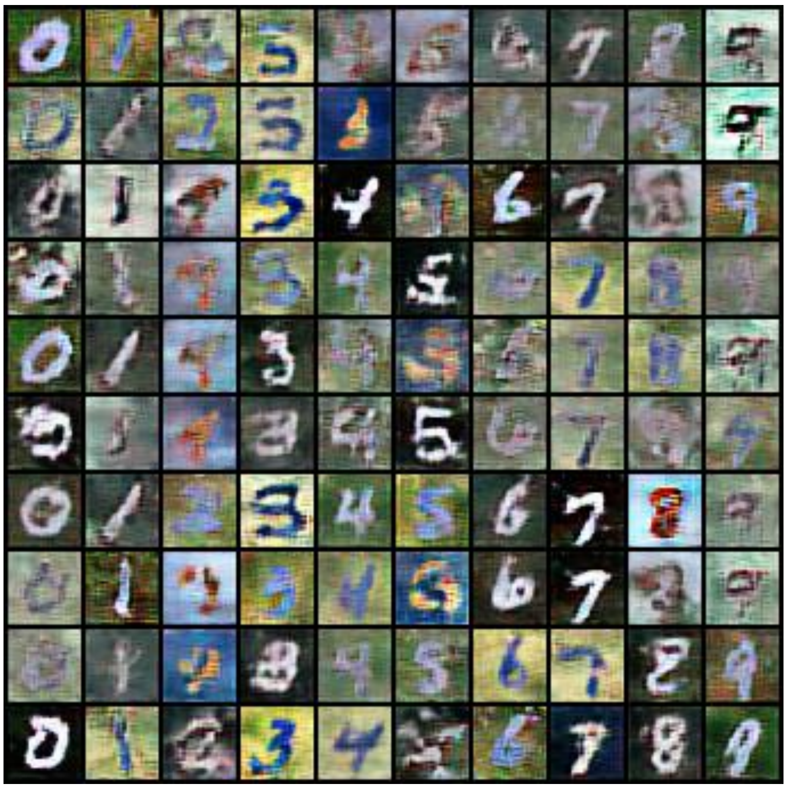
1. Model architecture

```
Generator(
  (label_emb): Embedding(10, 100)
  (l1): Sequential(
    (0): Linear(in_features=100, out_features=8192, bias=True)
  )
  (conv_blocks): Sequential(
    (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (1): Upsample(scale_factor=2.0, mode=nearest)
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True,
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Upsample(scale_factor=2.0, mode=nearest)
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True,
track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): Tanh()
  )
)

Discriminator(
  (conv_blocks): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Dropout2d(p=0.25, inplace=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Dropout2d(p=0.25, inplace=False)
    (6): BatchNorm2d(32, eps=0.8, momentum=0.1, affine=True,
track_running_stats=True)
    (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Dropout2d(p=0.25, inplace=False)
```

```
        (10): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True,
    track_running_stats=True)
        (11): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (12): LeakyReLU(negative_slope=0.2, inplace=True)
        (13): Dropout2d(p=0.25, inplace=False)
        (14): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True,
    track_running_stats=True)
      )
      (adv_layer): Sequential(
        (0): Linear(in_features=512, out_features=1, bias=True)
        (1): Sigmoid()
      )
      (aux_layer): Sequential(
        (0): Linear(in_features=512, out_features=10, bias=True)
        (1): Softmax(dim=None)
      )
    )
```

2. Accuracy : 82.5%
3. Sample output

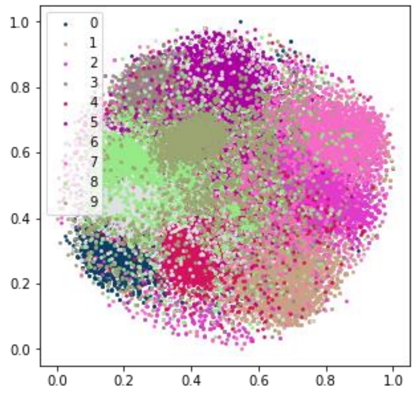
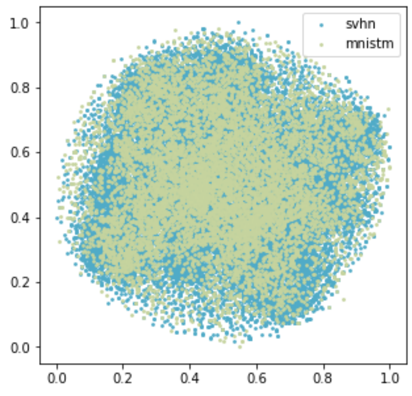
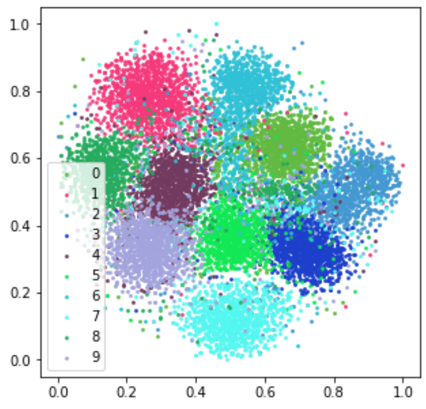
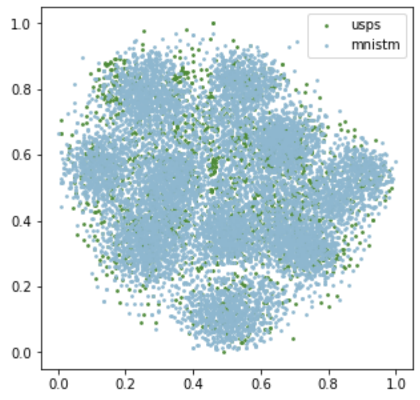
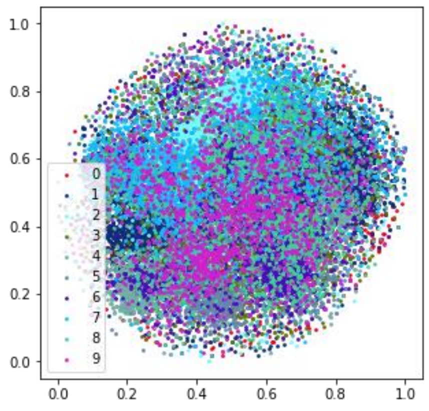
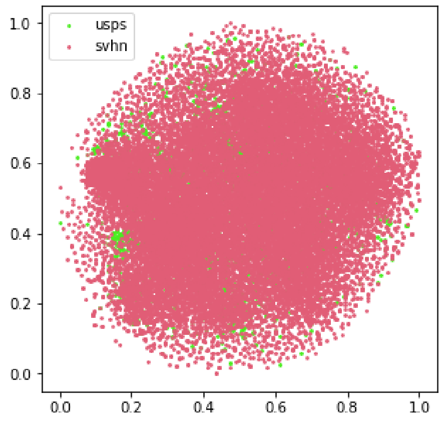


P3-DANN

1. Accuracy

	SVHN-Mnistm	Mnistm-USPS	USPS-SVHN
Trained on source	48.91%	75.49%	25.64%
Adaption	57.99%	88.19%	31.63%
Trained on target	97.58%	96.71%	89.64%

2. Visualization t-sne

	
Svhn-mnistm class0-9	Svhn-mnistm domain
	
Mnistm-Usps class0-9	Mnistm-Usps domain
	
Usps-Svhn class0-9	Usps-Svhn domain

3. Observation of DANN

   First, I found that it’s possible to pass the simple baseline just training on source dataset for SVHN-Mnistm and Mnistm-USPS。Look at the table above, we implemented t-sne visualization on target domain, and the best one is Mnistm-Usps, which has 0.8819% of accuracy. The second one is Svhn-mnistm, which has 0.5799% of accuracy, and the worst one is Usps-Svhn which has 0.3163%. The results of accuracy is same with the pictures. I think the most important reason for the difference is the amount of data for both source domain and target domain. In conclusion, the amount of source domain data should not be much more than target domain.

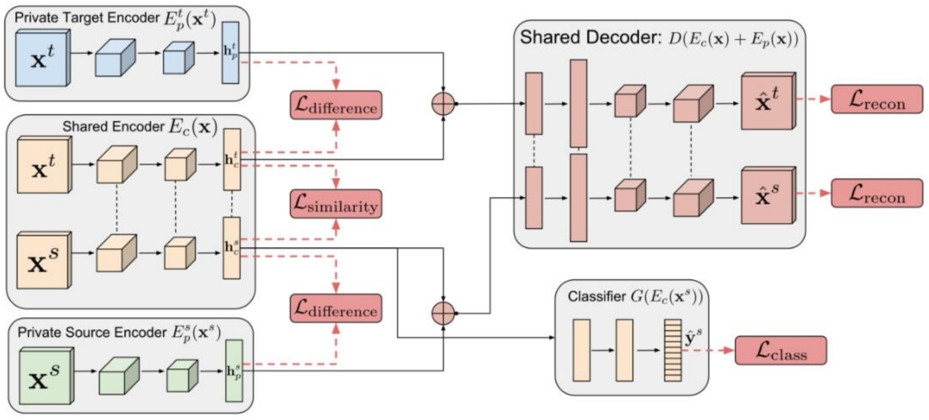
Bonus

1. Improved model: DSN

	SVHN-Mnistm	Mnistm-USPS	USPS-SVHN
Original model	57.99%	88.19%	31.63%
Improved model	62.28%	88.34%	42.82%

2. Observation of DSN

   I implemented the DSN model structure as my improved model. When I was training DANN, it was almost convergent when I had trained it for 14000 iterations with batch size 32. However, when I was training DSN, it converged much slower than DANN. I trained it for 29000 iterations to get a higher performance. In the training of USPS-SVHN, DSN got a obviously higher accuracy than DANN. However, it is hard for DSN to get a higher accuracy in training Mnistm-USPS, which has the most amount of data.


Private Target Encoder $E_p^t(\mathbf{x}^t)$
$\mathbf{x}^t$
$\mathbf{h}_p^t$
Shared Encoder $E_c(\mathbf{x})$
$\mathbf{x}^t$
$\mathbf{h}_c^t$
$\mathbf{x}^s$
$\mathbf{h}_c^s$
Private Source Encoder $E_p^s(\mathbf{x}^s)$
$\mathbf{x}^s$
$\mathbf{h}_p^s$
$\mathcal{L}_{\text{difference}}$
$\mathcal{L}_{\text{similarity}}$
$\mathcal{L}_{\text{difference}}$
Shared Decoder: $D(E_c(\mathbf{x})+E_p(\mathbf{x}))$
$\hat{\mathbf{x}}^t$
$\hat{\mathbf{x}}^s$
$\mathcal{L}_{\text{recon}}$
$\mathcal{L}_{\text{recon}}$
Classifier $G(E_c(\mathbf{x}^s))$
$\hat{\mathbf{y}}^s$
$\mathcal{L}_{\text{class}}$

DSN

References

https://github.com/lucidrains/stylegan2-pytorch
https://drive.google.com/file/d/1fnF-QsiQeKaxF-HbvFiGtzHF_Bf3CzJu/view
https://github.com/kai860115/DLCV2020-FALL/tree/main/hw3/dann
https://github.com/kai860115/DLCV2020-FALL/tree/main/hw3/dsn
https://github.com/eriklindernoren/PyTorch-GAN#auxiliary-classifier-gan
https://github.com/fungtion/DSN

Collaborators

B08902134 曾揚哲
B08502141 石旻翰

Stylegan2 model architecture

Generator

```
Generator(
  (initial_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (blocks): ModuleList(
    (0): GeneratorBlock(
      (to_style1): Linear(in_features=512, out_features=512, bias=True)
      (to_noise1): Linear(in_features=1, out_features=512, bias=True)
      (conv1): Conv2DMod()
      (to_style2): Linear(in_features=512, out_features=512, bias=True)
      (to_noise2): Linear(in_features=1, out_features=512, bias=True)
      (conv2): Conv2DMod()
      (activation): LeakyReLU(negative_slope=0.2, inplace=True)
      (to_rgb): RGBBlock(
        (to_style): Linear(in_features=512, out_features=512, bias=True)
        (conv): Conv2DMod()
        (upsample): Sequential(
          (0): Upsample(scale_factor=2.0, mode=bilinear)
          (1): Blur()
        )
      )
    )
    (1): GeneratorBlock(
      (upsample): Upsample(scale_factor=2.0, mode=bilinear)
      (to_style1): Linear(in_features=512, out_features=512, bias=True)
      (to_noise1): Linear(in_features=1, out_features=256, bias=True)
      (conv1): Conv2DMod()
      (to_style2): Linear(in_features=512, out_features=256, bias=True)
      (to_noise2): Linear(in_features=1, out_features=256, bias=True)
      (conv2): Conv2DMod()
      (activation): LeakyReLU(negative_slope=0.2, inplace=True)
      (to_rgb): RGBBlock(
        (to_style): Linear(in_features=512, out_features=256, bias=True)
        (conv): Conv2DMod()
        (upsample): Sequential(
          (0): Upsample(scale_factor=2.0, mode=bilinear)
          (1): Blur()
        )
      )
    )
    (2): GeneratorBlock(
      (upsample): Upsample(scale_factor=2.0, mode=bilinear)
      (to_style1): Linear(in_features=512, out_features=256, bias=True)
      (to_noise1): Linear(in_features=1, out_features=128, bias=True)
      (conv1): Conv2DMod()
      (to_style2): Linear(in_features=512, out_features=128, bias=True)
      (to_noise2): Linear(in_features=1, out_features=128, bias=True)
      (conv2): Conv2DMod()
      (activation): LeakyReLU(negative_slope=0.2, inplace=True)
      (to_rgb): RGBBlock(
        (to_style): Linear(in_features=512, out_features=128, bias=True)
        (conv): Conv2DMod()
        (upsample): Sequential(
          (0): Upsample(scale_factor=2.0, mode=bilinear)
          (1): Blur()
        )
      )
    )
    (3): GeneratorBlock(
      (upsample): Upsample(scale_factor=2.0, mode=bilinear)
      (to_style1): Linear(in_features=512, out_features=128, bias=True)
      (to_noise1): Linear(in_features=1, out_features=64, bias=True)
      (conv1): Conv2DMod()
      (to_style2): Linear(in_features=512, out_features=64, bias=True)
      (to_noise2): Linear(in_features=1, out_features=64, bias=True)
      (conv2): Conv2DMod()
      (activation): LeakyReLU(negative_slope=0.2, inplace=True)
      (to_rgb): RGBBlock(
        (to_style): Linear(in_features=512, out_features=64, bias=True)
        (conv): Conv2DMod()
        (upsample): Sequential(
          (0): Upsample(scale_factor=2.0, mode=bilinear)
          (1): Blur()
        )
      )
    )
```

```
        (4): GeneratorBlock(
          (upsample): Upsample(scale_factor=2.0, mode=bilinear)
          (to_style1): Linear(in_features=512, out_features=64, bias=True)
          (to_noise1): Linear(in_features=1, out_features=32, bias=True)
          (conv1): Conv2DMod()
          (to_style2): Linear(in_features=512, out_features=32, bias=True)
          (to_noise2): Linear(in_features=1, out_features=32, bias=True)
          (conv2): Conv2DMod()
          (activation): LeakyReLU(negative_slope=0.2, inplace=True)
          (to_rgb): RGBBlock(
            (to_style): Linear(in_features=512, out_features=32, bias=True)
            (conv): Conv2DMod()
          )
        )
      )
      (attns): ModuleList(
        (0): None
        (1): None
        (2): None
        (3): None
        (4): Sequential(
          (0): Residual(
            (fn): PreNorm(
              (fn): LinearAttention(
                (nonlin): GELU()
                (to_q): Conv2d(64, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
                (to_kv): DepthWiseConv2d(
                  (net): Sequential(
                    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64, 
bias=False)
                    (1): Conv2d(64, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
                  )
                )
                (to_out): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
              )
              (norm): ChanNorm()
            )
          )
```

```
            (1): Residual(
                (fn): PreNorm(
                    (fn): Sequential(
                        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1))
                        (1): LeakyReLU(negative_slope=0.2, inplace=True)
                        (2): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
                    )
                    (norm): ChanNorm()
                )
            )
        )
    )
)
Discriminator
Discriminator(
    (blocks): ModuleList(
        (0): DiscriminatorBlock(
            (conv_res): Conv2d(3, 64, kernel_size=(1, 1), stride=(2, 2))
            (net): Sequential(
                (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (1): LeakyReLU(negative_slope=0.2, inplace=True)
                (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (3): LeakyReLU(negative_slope=0.2, inplace=True)
            )
            (downsample): Sequential(
                (0): Blur()
                (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
            )
        )
        (1): DiscriminatorBlock(
            (conv_res): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2))
            (net): Sequential(
                (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (1): LeakyReLU(negative_slope=0.2, inplace=True)
                (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
                (3): LeakyReLU(negative_slope=0.2, inplace=True)
            )
            (downsample): Sequential(
      (0): Blur()
      (1): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    )
  )
  (2): DiscriminatorBlock(
    (conv_res): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2))
    (net): Sequential(
      (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
      (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (downsample): Sequential(
      (0): Blur()
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    )
  )
  (3): DiscriminatorBlock(
    (conv_res): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2))
    (net): Sequential(
      (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
      (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (downsample): Sequential(
      (0): Blur()
      (1): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    )
  )
  (4): DiscriminatorBlock(
    (conv_res): Conv2d(512, 512, kernel_size=(1, 1), stride=(2, 2))
    (net): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
      (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): LeakyReLU(negative_slope=0.2, inplace=True)
    )
```

```
      (downsample): Sequential(
        (0): Blur()
        (1): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      )
    )
    (5): DiscriminatorBlock(
      (conv_res): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1))
      (net): Sequential(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): LeakyReLU(negative_slope=0.2, inplace=True)
        (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): LeakyReLU(negative_slope=0.2, inplace=True)
      )
    )
  )
  (attn_blocks): ModuleList(
    (0): Sequential(
      (0): Residual(
        (fn): PreNorm(
          (fn): LinearAttention(
            (nonlin): GELU()
            (to_q): Conv2d(64, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (to_kv): DepthWiseConv2d(
              (net): Sequential(
                (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
                (1): Conv2d(64, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
              )
            )
            (to_out): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
          )
          (norm): ChanNorm()
        )
      )
      (1): Residual(
        (fn): PreNorm(
          (fn): Sequential(
            (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1))
```

```
                    (1): LeakyReLU(negative_slope=0.2, inplace=True)
                    (2): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
                  )
                  (norm): ChanNorm()
                )
              )
            )
            (1): None
            (2): None
            (3): None
            (4): None
            (5): None
          )
          (quantize_blocks): ModuleList(
            (0): None
            (1): None
            (2): None
            (3): None
            (4): None
            (5): None
          )
          (final_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (flatten): Flatten()
          (to_logit): Linear(in_features=2048, out_features=1, bias=True)
        )
```