

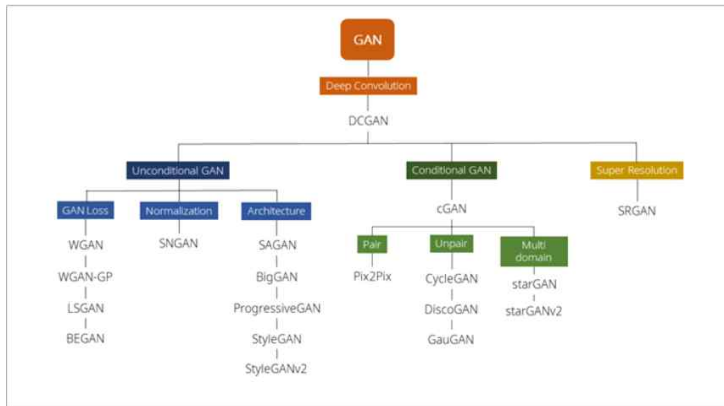
StyleGAN


A Style-Based Generator Architecture for Generative Adversarial Networks

GAN을 기반으로 한 이미지 합성 기술은 PGGAN 등을 포함하여 지속적으로 발전하고 있습니다. 그러나 Generator를 통한 이미지 합성 과정은 여전히 block box로 여겨지며, 이로 인해 합성되는 이미지의 특징 (얼굴형, 표정, 헤어스타일 등)을 조절하기가 매우 어렵다는 한계가 있어 퀄리티가 낮은 부자연스러운 이미지를 생성하게 됩니다.




GAN 종류

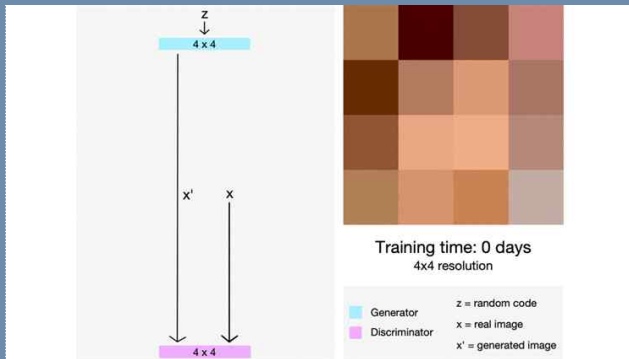




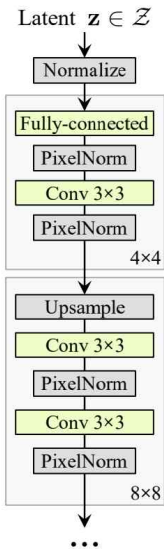
StyleGAN은 GAN에서 파생된 알고리즘 중 하나로 2019
CVPR에 발표된 NVIDIA의 새 논문인 "A Style-Based
Generator Architecture for Generative Adversarial Networks"
입니다.



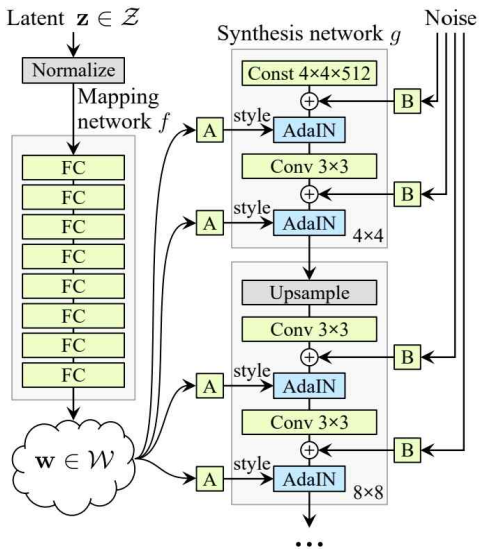




PGGAN은 StyleGAN의 baseline이 되는 모델로 저화질에서 시작해 점점 레이어를 쌓아가며, 고화질 이미지를 생성합니다. 하지만 대부분의 모델과 마찬가지로 생성된 이미지의 특징을 제어하는 기능이 매우 제한적 인데 StyleGAN은 이러한 문제점을 개선하며 PGGAN의 Progressive Growing을 적용해 고해상도 이미지를 생성합니다.

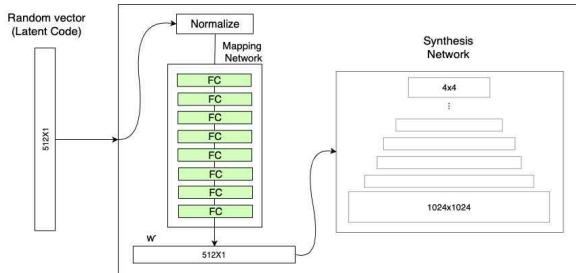


(a) Traditional



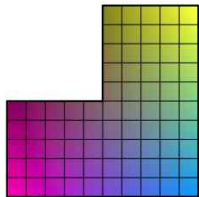
(b) Style-based generator

Mapping Network

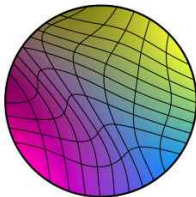


baseline인 PGGAN의 문제는 세부적인 attribute를 조절하지 못한다는 것입니다. 이 단점을 해결하기 위해 latent sapce Z(Gaussian)에서 sampling한 z 를 intermediate latent space W 의 w 로 mapping해주는 8-layer MLP로 구성된 non-linear mapping network를 사용합니다.

Disentanglement studies



(a) Distribution of features in training set



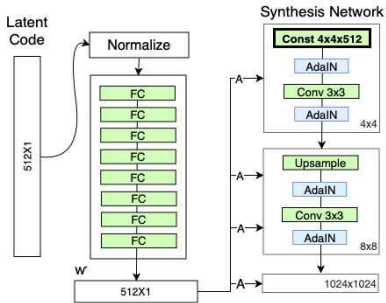
(b) Mapping from \mathcal{Z} to features



(c) Mapping from \mathcal{W} to features

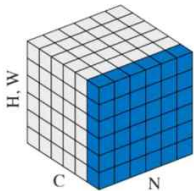
b 는 기존의 전통적인 방식인 gaussian distribution에서 sampling 한 latent vector z 입니다. 이 그림에서 보면 보라색과 노란색을 interpolation 할 때 가운데 점인 파란색으로 되게 되는데 이는 entangle하게 될 수 밖에 없습니다. 기존의 Gaussian 분포에서 추출하는 방식은 mapping이 되면서 nonlinear하게 변하게 됩니다. 즉, 특징들이 얹히고 얹혀서 변화가 된다는 것입니다. 각 특징을 잃어버리게 된다는 것인데 본 논문에서 제안하는 방식 latent vector z 를 fully connected 네트워크를 거쳐 매핑한 w space를 보면 그림 c 처럼 각 색이 잘 분리되어 있는 것을 볼 수 있습니다. 본 논문에서 제안하는 Mapping network 방식은 linear하게 각 특징들이 Disentangle하게 됨을 알 수 있습니다.

Synthesis Network

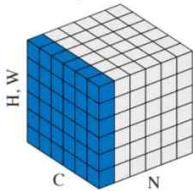


synthesis network는 4x4부터 1024x1024 resolution feature map을 만드는 총 9개의 style block으로 이루어져있습니다. 마지막에는 RGB로 바꿔주는 layer가 있어 이미지 channel에 맞춰줍니다. synthesis network는 4x4x512 tensor에서 부터 시작해 하나의 style block당 두번의 convolution을 진행합니다. Style block의 input은 전 style block의 output인 featuremap입니다.

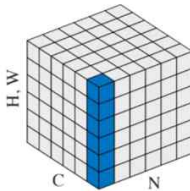
Batch Norm



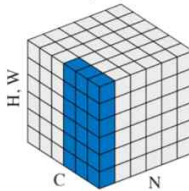
Layer Norm



Instance Norm

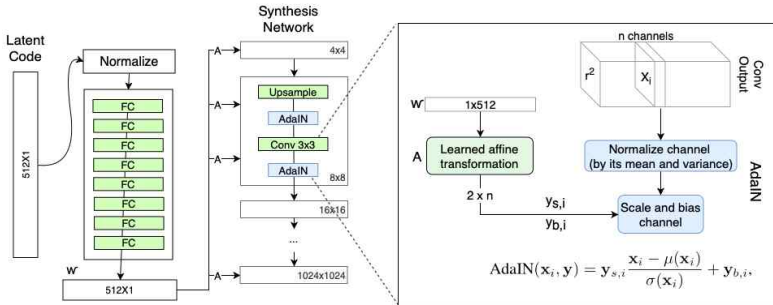


Group Norm



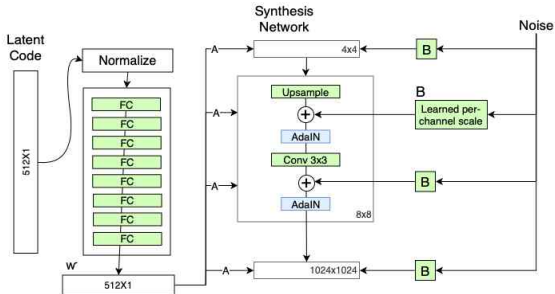
AdaIN은 content 이미지 x에 style 이미지 y의 스타일을 입힐 때 사용하는 normalization으로, style transfer에 거의 공식적으로 사용됩니다.

N개의 배치에서 각각의 이미지를 각 채널에 대해 정규화 실행

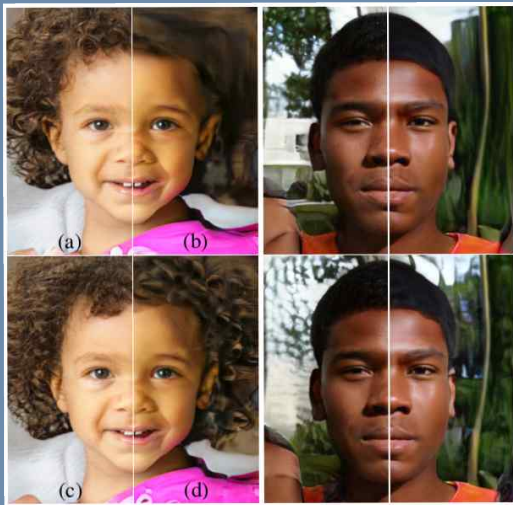


mapping network로 만들어진 w 는 affine transform을 거쳐 AdaIN의 style scaling factor와 style bias factor의 역할을 하는 style vector($2 \times N$)로 변화하게 됩니다.
정규화된 content 이미지는 이에 맞춰 scaling되고 bias가 더해집니다.

Stochastic variation



synthesis network의 각 layer마다 random noise를 추가하였습니다. 이렇게 stochastic한 정보를 따로 추가해주면 더욱 사실적인 이미지를 생성하게 될 뿐 아니라, input latent vector는 이미지의 중요한 정보(성별, 인종, 헤어스타일 등)를 표현하는 데에만 집중할 수 있게 되고 이를 조절하는 것도 더욱 쉬워집니다.

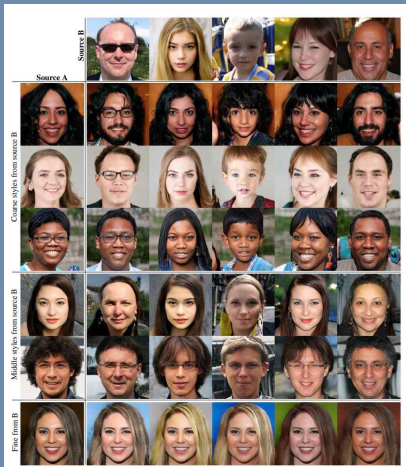


- a는 noise를 모든 layer에 적용한 것
 - b는 noise를 적용 안한 것
- c는 64x64~1024x1024 레이어(fine layer)에만 노이즈를 적용한 것
- d는 4x4~32x32 layer(coarse layer)에만 노이즈를 적용한 것

progressive 레이어를 적절히 활용할 경우

1. Coarse(굵직한 특징) - 8 해상도까지 (* 4x4 layer ~ 8x8 layer)- 포즈, 일반적인 헤어스타일, 얼굴형 등에 영향
2. Middle(중간 특징) - 16부터 32 해상도까지 (* 16x16 layer ~ 32x32 layer) - 자세한 얼굴 특징, 헤어스타일, 눈 뜨고/감음 등에 영향
3. Fine(자세한 특징) - 64부터 1024(* 64x64 layer ~ 1024x1024 layer) 해상도까지 - 눈, 머리, 피부 등의 색 조합과 미세한 특징 등에 영향

Style mixing



mixing regularization

Mixing regularization	Number of latents during testing			
	1	2	3	4
E 0%	4.42	8.22	12.88	17.41
50%	4.41	6.10	8.71	11.61
F 90%	4.40	5.11	6.88	9.03
100%	4.83	5.17	6.63	8.40

원래대로라면 하나의 latent code z

만 가지고 학습을 해야하는데 네트워크 정규화 효과를 노리기 위하여 여러 개의 z 를 사용하였습니다.

예를 들어 2개를 사용한다고 하면 z_1 과 z_2 를 mapping network에 통과시켜 w_1 과 w_2 를 만듭니다.

이후에 크로스오버 포인트를 설정해, w_1 을 크로스오버 이전에 적용을 시킨 뒤, w_2 이후에 적용을 시킵니다. 이런 방법을 사용하면 다양한 스타일이 섞여 네트워크의 regularization 효과를 얻을 수 있습니다.

Disentanglement measurement method

본 논문에서는 disentanglement를 측정할 수 있는 두가지 지표를 제안합니다.

1. Perceptual Path Length

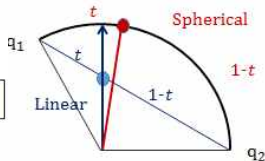
두개의 vector를 interpolation할 때 부드럽게 바뀌는지 측정하는 지표입니다.

disentanglement를 비교하기 위해 Z space에서 sampling한 경우와 W space에서 sampling한 경우로 나누었습니다.

z,w를 각각 다른 방식으로 interpolation합니다.

두 개의 z1,z2로 생성한 이미지를 VGG16을 통과시켜 embedding시킨 뒤, 두 embedding vector의 차이를 계산하는 것입니다.

$$l_Z = \mathbb{E} \left[\frac{1}{\epsilon^2} d(G(\text{slerp}(\mathbf{z}_1, \mathbf{z}_2; t)), G(\text{slerp}(\mathbf{z}_1, \mathbf{z}_2; t + \epsilon))) \right]$$
$$l_W = \mathbb{E} \left[\frac{1}{\epsilon^2} d(g(\text{lerp}(f(\mathbf{z}_1), f(\mathbf{z}_2); t)), g(\text{lerp}(f(\mathbf{z}_1), f(\mathbf{z}_2); t + \epsilon))) \right]$$



2. linear separability

latent space에서 attribute가 얼마나 선형적으로 분류될 수 있는지를 판단하는 것입니다.

이를 위해 간단한 선형 분류기를 학습한 뒤 엔트로피를 계산해서 latent vector가 얼마나 linear한 subspace에 존재하는지를 확인하는 것입니다.

latent space point를 분류할 수 있는 하나의 선형 분류기(SVM)을 학습합니다. 그 후 latent vector가 선형 분류기에서 얼마나 떨어져있는지 알 수 있고 conditional entropy 값을 구할 수 있습니다.

여기서 entropy 값은 하나의 입력 벡터 X 가 주어졌을 때 그때의 true class에 대한 entropy 값을 측정할 수 있습니다.

다시 말해 하나의 이미지 data가 특정 클래스로 정확히 분류되기 위해 해당하는 feature가 얼마나 부족한지에 대한 정보를 알 수 있으며 이는 값이 낮을수록 이상적인 image임을 나타냅니다.

FFHQ Face Data Set



```
import torch
from tqdm import tqdm
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from PIL import Image
import math

from torch.utils.data import DataLoader
from torchvision import datasets, transforms, utils

%matplotlib inline
```

```

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()

        self.from_rgbs = nn.ModuleList([
            SConv2d(3, 16, 1),
            SConv2d(3, 32, 1),
            SConv2d(3, 64, 1),
            SConv2d(3, 128, 1),
            SConv2d(3, 256, 1),
            SConv2d(1, 512, 1),
            SConv2d(1, 512, 1),
            SConv2d(1, 512, 1)
        ])

        self.convs = nn.ModuleList([
            ConvBlock(16, 32, 3, 1),
            ConvBlock(32, 64, 3, 1),
            ConvBlock(64, 128, 3, 1),
            ConvBlock(128, 256, 3, 1),
            ConvBlock(256, 512, 3, 1),
            ConvBlock(512, 512, 3, 1),
            ConvBlock(512, 512, 3, 1),
            ConvBlock(512, 512, 3, 1),
            ConvBlock(512, 512, 3, 1, 4, 0)
        ])

        self.fc = nn.Linear(512, 1)

        self.n_layer = 9

    def forward(self, image,
                step = 0,
                alpha = 1):
        for i in range(step, -1, -1):
            layer_index = self.n_layer - i - 1

            if i == step:
                result = self.from_rgbs[layer_index](image)

            if i == 0:
                res_var = result.var(0, unbiased=False) + 1e-8
                res_std = torch.sqrt(res_var)

                mean_std = res_std.mean().expand(result.size(0), 1, 4, 4)
                result = torch.cat([result, mean_std, 1])

            result = self.convs[layer_index](result)

            if i > 0:
                result = nn.functional.interpolate(result, scale_factor=0.5, mode='bilinear',
                                                    align_corners=False)

            if i == step and 0 <= alpha < 1:
                result_next = self.from_rgbs[layer_index + 1](image)
                result_next = nn.functional.interpolate(result_next, scale_factor=0.5,
                                                        mode='bilinear', align_corners=False)

                result = alpha * result + (1 - alpha) * result_next

        result = result.squeeze(2).squeeze(2)
        result = self.fc(result)
        return result

```

```
class StyleBasedGenerator(nn.Module):
```

```
    def __init__(self, n_fc, dim_latent, dim_input):
        super().__init__()
        self.fcns = Intermediate_Generator(n_fc, dim_latent)
        self.convs = nn.ModuleList([
            Early_StyleConv_Block(512, dim_latent, dim_input),
            StyleConv_Block(512, 512, dim_latent),
            StyleConv_Block(512, 512, dim_latent),
            StyleConv_Block(512, 512, dim_latent),
            StyleConv_Block(512, 256, dim_latent),
            StyleConv_Block(256, 128, dim_latent),
            StyleConv_Block(128, 64, dim_latent),
            StyleConv_Block(64, 32, dim_latent),
            StyleConv_Block(32, 16, dim_latent)
        ])
        self.to_rgbs = nn.ModuleList([
            SConv2d(512, 3, 1),
            SConv2d(512, 3, 1),
            SConv2d(512, 3, 1),
            SConv2d(512, 3, 1),
            SConv2d(256, 3, 1),
            SConv2d(128, 3, 1),
            SConv2d(64, 3, 1),
            SConv2d(32, 3, 1),
            SConv2d(16, 3, 1)
        ])

    def forward(self, latent_z,
                step = 0,
                alpha=1,
                noise=None,
                mix_steps=[]):
        if type(latent_z) != type([]):
            print('You should use list to package your latent_z')
            latent_z = [latent_z]

        if (len(latent_z) != 2 and len(mix_steps) == 0) or type(mix_steps) != type([]):
            print('Warning: Style mixing disabled, possible reasons:')
            print('- Invalid number of latent vectors')
            print('- Invalid parameter type: mix_steps')
            mix_steps = []

        latent_w = [self.fcns(latent) for latent in latent_z]
        batch_size = latent_w[0].size(0)

        result = 0
        current_latent = 0

        for i, conv in enumerate(self.convs):
            if i in mix_steps:
                current_latent = latent_w[i]
            else:
                current_latent = latent_w[0]

            if i > 0 and step > 0:
                result_upsample = nn.functional.interpolate(result, scale_factor=2, mode='bilinear',
                                                             align_corners=False)
                result = conv(result_upsample, current_latent, noise[i])
            else:
                result = conv(current_latent, noise[i])

            if i == step:
                result = self.to_rgbs[i](result)

            if i > 8 and 0 <= alpha < 1:
                result_prev = self.to_rgbs[i - 1](result_upsample)
                result = alpha * result + (1 - alpha) * result_prev

            break

        return result
```

```
class Intermediate_Generator(nn.Module):

    def __init__(self, n_fc, dim_latent):
        super().__init__()
        layers = [PixelNorm()]
        for i in range(n_fc):
            layers.append(SLinear(dim_latent, dim_latent))
            layers.append(nn.LeakyReLU(0.2))

        self.mapping = nn.Sequential(*layers)

    def forward(self, latent_z):
        latent_w = self.mapping(latent_z)
        return latent_w
```

```
class FC_A(nn.Module):

    def __init__(self, dim_latent, n_channel):
        super().__init__()
        self.transform = SLinear(dim_latent, n_channel * 2)

        self.transform.linear.bias.data[:n_channel] = 1
        self.transform.linear.bias.data[n_channel:] = 0

    def forward(self, w):

        style = self.transform(w).unsqueeze(2).unsqueeze(3)
        return style
```



```

class AdaIn(nn.Module):

    def __init__(self, n_channel):
        super().__init__()
        self.norm = nn.InstanceNorm2d(n_channel)

    def forward(self, image, style):
        factor, bias = style.chunk(2, 1)
        result = self.norm(image)
        result = result * factor + bias
        return result

```

```

class StyleConv_Block(nn.Module):

    def __init__(self, in_channel, out_channel, dim_latent):
        super().__init__()

        self.style1 = FC_A(dim_latent, out_channel)
        self.style2 = FC_A(dim_latent, out_channel)

        self.noise1 = quick_scale(Scale_B(out_channel))
        self.noise2 = quick_scale(Scale_B(out_channel))

        self.adain = AdaIn(out_channel)
        self.lrelu = nn.LeakyReLU(0.2)

        self.conv1 = SConv2d(in_channel, out_channel, 3, padding=1)
        self.conv2 = SConv2d(out_channel, out_channel, 3, padding=1)

    def forward(self, previous_result, latent_w, noise):

        result = self.conv1(previous_result)

        result = result + self.noise1(noise)
        result = self.adain(result, self.style1(latent_w))
        result = self.lrelu(result)
        result = self.conv2(result)
        result = result + self.noise2(noise)
        result = self.adain(result, self.style2(latent_w))
        result = self.lrelu(result)

        return result

```

```

import os
os.environ['CUDA_VISIBLE_DEVICES']='1, 2'
n_gpu          = 1 #모델 훈련에 사용되는 GPU 수
device         = torch.device('cuda:0') #테서를 만들고 저장하는 기본 장치

learning_rate  = {128: 0.0015, 256: 0.002, 512: 0.003, 1024: 0.003} #훈련의 다른 단계에서 학습률 나타내는 딕셔너리
batch_size_1gpu = {4: 128, 8: 128, 16: 64, 32: 32, 64: 16, 128: 16} #훈련의 다른 단계에서 배치 크기를 나타내는 딕셔너리
mini_batch_size_1 = 8 #최소 배치 크기
batch_size     = {4: 256, 8: 256, 16: 128, 32: 64, 64: 32, 128: 16} #훈련의 다른 단계에서 배치 크기를 나타내는 딕셔너리
mini_batch_size = 8 #최소 배치 크기
batch_size_4gpus = {4: 512, 8: 256, 16: 128, 32: 64, 64: 32} #
mini_batch_size_4 = 16 #
batch_size_8gpus = {4: 512, 8: 256, 16: 128, 32: 64} #
mini_batch_size_8 = 32 #
n_fc              = 8 #완전 연결 매핑 네트워크의 레이어 수
dim_latent        = 512 #잠재 공간의 차원
dim_input         = 4 #생성기의 첫 번째 레이어 크기
n_sample          = 120000 #단일 레이어를 훈련하는 데 사용할 샘플 수
DGR               = 1 #Generator를 훈련시키기 전에 Discriminator를 몇 번 훈련시킬 것인가?
n_show_loss       = 40 #n_show_loss번 반복 마다 손실이 기록
step              = 1 # 학습을 시작할 레이어
max_step          = 8 # 이미지의 최대 해상도는 (max_step + 2)^2
style_mixing      = [] # 두 번째 스타일을 사용하여 평가할 레이어
image_folder_path = './dataset/'
save_folder_path  = '/kaggle/working/'

```

```

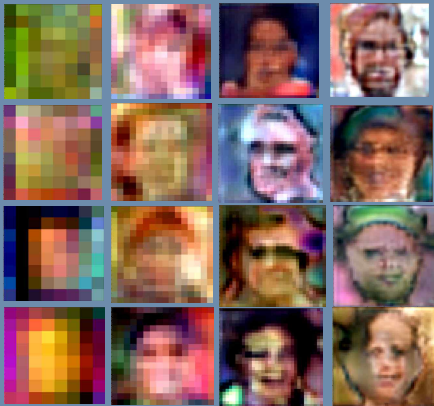
generator      = StyleBased_Generator(n_fc, dim_latent, dim_input).to(device)
discriminator   = Discriminator().to(device)
g_optim        = optim.Adam([
    {'params': generator.convs.parameters(),
     'lr'      : 0.001
    }, {
    'params': generator.to_rgbs.parameters(),
    'lr'      : 0.001
    }], lr=0.001, betas=(0.0, 0.99))
g_optim.add_param_group({
    'params': generator.fcs.parameters(),
    'lr'      : 0.001 * 0.01,
    'mul'      : 0.01
})
d_optim        = optim.Adam(discriminator.parameters(), lr=0.001, betas=(0.0, 0.99))
dataset        = datasets.ImageFolder('/kaggle/input/ffhq-face-data-set')

if is_continue:
    if os.path.exists('checkpoint/trained.pth'):
        # Load data from last checkpoint
        print('Loading pre-trained model...')
        checkpoint = torch.load('checkpoint/trained.pth')
        generator.load_state_dict(checkpoint['generator'])
        discriminator.load_state_dict(checkpoint['discriminator'])
        g_optim.load_state_dict(checkpoint['g_optim'])
        d_optim.load_state_dict(checkpoint['d_optim'])
        step, startpoint, used_sample, alpha = checkpoint['parameters']
        d_losses = checkpoint.get('d_losses', [float('inf')])
        g_losses = checkpoint.get('g_losses', [float('inf')])
    else:
        print('No pre-trained model detected, restart training...')

if is_train:
    generator.train()
    discriminator.train()
    d_losses, g_losses = train(generator, discriminator, g_optim, d_optim, dataset, step, startpoint,
                               used_sample, d_losses, g_losses, alpha)
else:
    pass

```

생성된 FAKE 이미지



컴퓨터 성능 부족으로 64x64 이미지 까지만 생성

감사합니다.