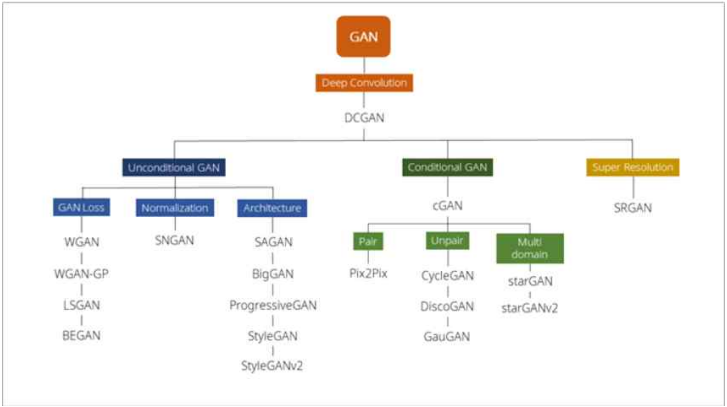
StyleGAN

A Style-Based Generator Architecture for Generative Adversarial Networks

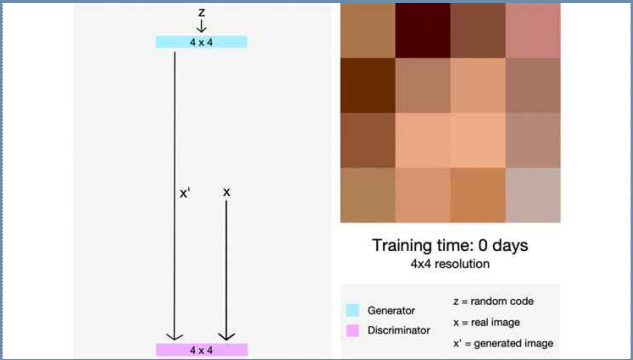
GANを基盤としたイメージ合成技術はPGGANなどを含めて持続的に発展しています。しかし、Generatorを通じたイメージ合成過程は依然としてblock boxと見なされ、これによって合成されるイメージの特徴(顔型、表情、ヘアスタイルなど)を調節するのが非常に難しいという限界があり、クオリティの低い不自然なイメージを生成します。

GANの種類

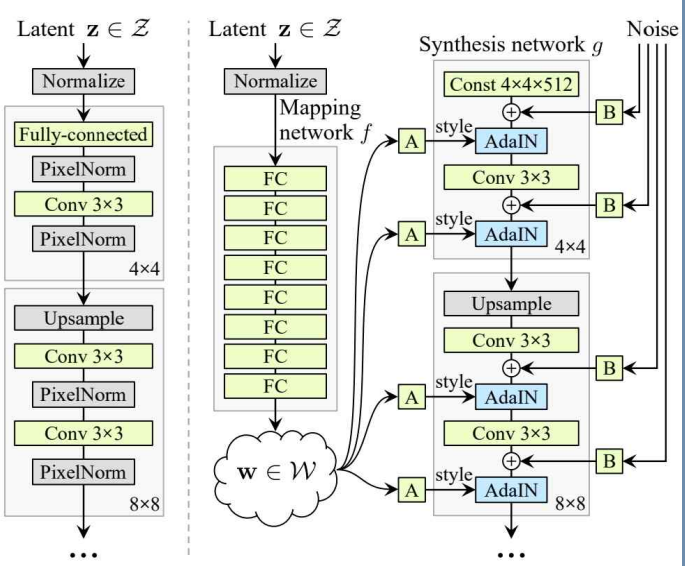

GAN
Deep Convolution
DCGAN
Unconditional GAN
Conditional GAN
Super Resolution
GAN Loss
Normalization
Architecture
cGAN
SRGAN
WGAN
SNGAN
SAGAN
Pair
Unpair
Multi domain
WGAN-GP
BigGAN
Pix2Pix
CycleGAN
starGAN
LSGAN
ProgressiveGAN
DiscoGAN
starGANv2
BEGAN
StyleGAN
GauGAN
StyleGANv2

StyleGANはGANから派生されたアルゴリズムで、2019CVPRで発表されたNVIDIAの新しい論文「A Style-Based Generator Architecture for Generative Adversarial Networks」です。


Source B
Source A
Coarse styles from source B



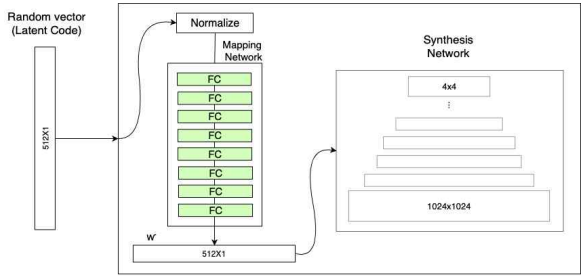
PGGANはStyleGANのbaselineになるモデルで、低画質から始まってどんどんレイヤーを積み重ね、高画質イメージを生成します。しかし、ほとんどのモデルのように生成されたイメージの特徴を制御する機能が制限的ですが、StyleGANはこのような問題点を改善し、PGGANのProgressive Growingを適用して高解像度イメージを生成します。



(a) Traditional

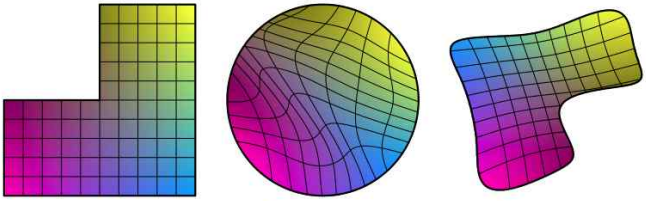
(b) Style-based generator

Mapping Network



baselineであるPGGANの問題は詳細なattributeを調節できないということです。この短所を解決するために、latent sapce Z(Gaussian)でsampingしたzをintermediate latent space Wのwにmappingする8-layer MLPで構成されたnon-linear mapping Networkを使用します。

Disentanglement studies



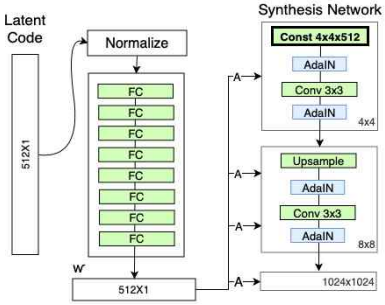
(a) Distribution of features in training set

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

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

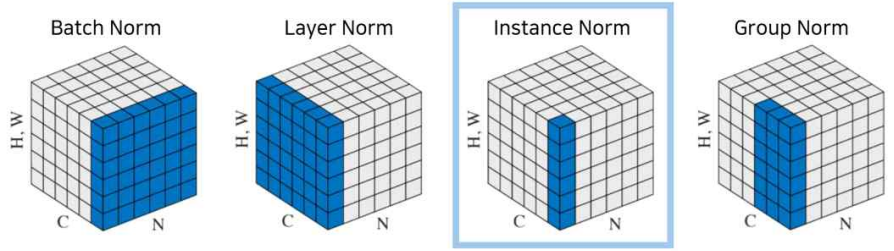
(b)は既存の方式であるgaussian distributionでsamplingしたlatent vector zです。この絵を見ると紫色と黄色をinterpolationする時、真ん中の点である青色になりますが、これはentangleになるしかありません。既存のGaussian分布から抽出する方法は、mappingされながらnonlinearに変化します。つまり、特徴が絡み合って変化するということです。各特徴を失ってしまうということですが、本論文で提案する方式latent vector zをfully connected network を通じてmappingしたw spaceを見ると、(c)のように色がよく分離されていることが分かります。本論文で提案するMapping network方式は、linearに各特徴がDisentangleになることが分かります。

Synthesis Network



synthesis networkは、4x4から1024x1024 resolution feature mapを作る計9つのstyle blockで構成されています。最後にはRGBに変えてくれるlayerがあり、イメージchannelに合わせてくれます。
synthesis networkは、4x4x512 tensorから始まり、1つのstyle blockあたり2回のconvolutionを行います。Style blockのinputは、前のstyle blockのoutputであるfeaturemapです。

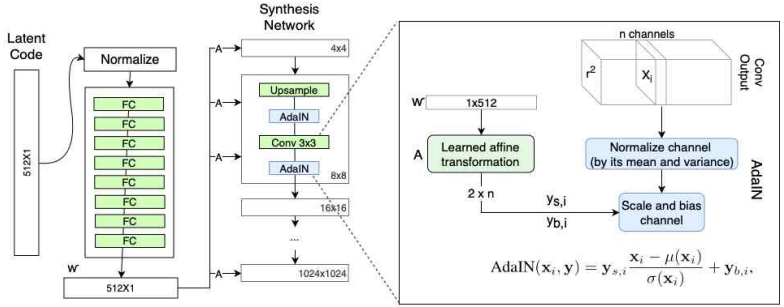
AdaIN



AdaINは、contentイメージxにstyleイメージyのスタイルを付ける時に使用するnormalizationで、styletransferにほぼ公式的に使用されます。

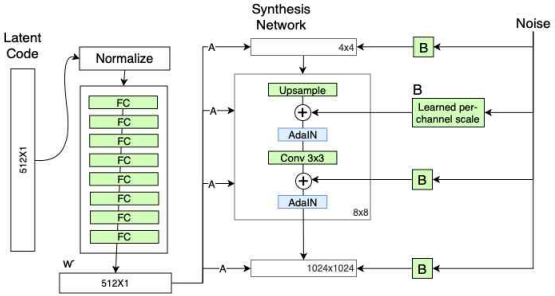
N個の配置でそれぞれのイメージを各チャンネルに対して正規化実行

AdaIN

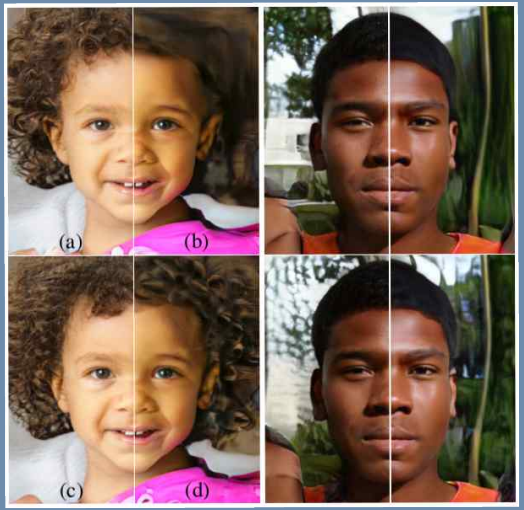


mapping networkで作られたwは、affine transformを通じてAdAINのstyle scaling factorとstyle bias factor役割をするstyle vector(2*N)に変化します。
正規化されたcontentイメージはこれに合わせてscalingされ、biasが加わります。

Stochastic variation



synthesis networkの各layerごとにrandom noiseを追加しました。このようにstochasticな情報を追加すると、よりリアルなイメージを生成するだけでなく、input latent vectorはイメージの重要な情報(性別、人種、ヘアスタイルなど)を表現することだけに集中できるようになり、これを調節することもより容易になります。

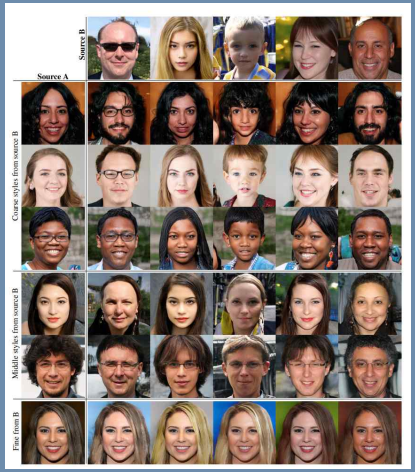


- (a)はnoiseをすべてのlayerに適用
- (b)はnoiseを適用していない
- (c)は64x64~1024x1024layer(fine layer)にのみnoiseを適用
- (d)は4x4~32x32layer(coarse layer)にのみノイズを適用

progressive layerを適切に活用する場合

1. Coarse(太い特徴) - 8解像度まで(4x4layer~8x8layer) - ポーズ、一般的なヘアスタイル、顔型などに影響
2. Middle(中間特徴) - 16から32解像度まで(16x16layer~32x32layer) - 詳しい顔の特徴、ヘアスタイル、目を開ける/閉じるなどに影響
3. Fine(詳しい特徴) - 64から1024解像度まで(64x64 layer ~ 1024x1024 layer) - 目、髪、肌などの色の組み合わせと微細な特徴などに影響

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**

元通りならひとつのlatent code zだけで学習しなければなりませんが、network正規化効果のために複数のzを使用しました。たとえば、2つを使用すると、z1とz2をmapping networkに通過させてw1とw2を作ります。
以後にcross over pointを設定し、w1をcross over以前に適用させた後、w2を以降に適用させます。
このような方法を使用すると、多様なスタイルが混ざってnetworkのregularization効果を得ることができます。

Disentanglement measurement method

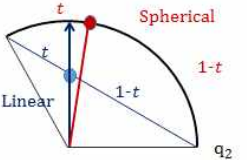
本論文では、disentanglementを測定できる2つの指標を提案します。

1. Perceptural Path Length

2つのvectorをinterpolationする時、柔らかく変わるかを測定する指標です。
disentanglementを比較するためにZ spaceでsamplingした場合とW spaceでsamplingした場合に分けました。
z、wをそれぞれ異なる方式でinterpolationします。
二つのz1、z2で生成したイメージをVGG16を通過させてembeddingさせた後、2つのembedding vectorの違いを計算することです。

$$l_{\mathcal{Z}} = \mathbb{E}\left[\frac{1}{\epsilon^2} d\big(G(\mathrm{slerp}(\mathbf{z}_1, \mathbf{z}_2; t)), G(\mathrm{slerp}(\mathbf{z}_1, \mathbf{z}_2; t+\epsilon))\big)\right]$$

$$l_{\mathcal{W}} = \mathbb{E}\left[\frac{1}{\epsilon^2} d\big(g(\mathrm{lerp}(f(\mathbf{z}_1), f(\mathbf{z}_2); t)), g(\mathrm{lerp}(f(\mathbf{z}_1), f(\mathbf{z}_2); t+\epsilon))\big)\right]$$



Disentanglement measurement method

2. linear separability

latent spaceでattributeがどれだけ線形的に分類できるかを判断することです。
このために簡単な線形分類器を学習した後、エントロピーを計算してlatent vectorがどれほどlinearなsubspaceに存在するかを確認します。
latent space pointを分類できる1つの線形分類器(SVM)を学習します。その後、latent vectorが線形分類器からどれだけ離れているかが分かり、conditional entropyが得られます。
ここでentropyは、1つの入力vector Xが与えられたときのtrue classのentropyを測定することができます。

つまり、1つのイメージdataが特定クラスに正確に分類されるために、該当するfeatureがどれほど不足しているかについての情報が分かり、これは値が低いほど理想的なイメージであることを示します。

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(3, 512, 1),
            SConv2d(3, 512, 1),
            SConv2d(3, 512, 1),
            SConv2d(3, 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(513, 512, 3, 1, 4, 0)
        ])
        self.fc = SLinear(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 StyleBased_Generator(nn.Module):

    def __init__(self, n_fc, dim_latent, dim_input):
        super().__init__()
        self.fcs    = 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.fcs(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[1]
            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 > 0 and 0 <= alpha < 1:
                    result_prev = self.to_rgbs[i - 1](result_upsample)
                    result = alpha * result + (1 - alpha) * result_prev

                break

        return result
```

```python
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
```

```python
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
```

```python
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
```

```python
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') #텐서를 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/'
```

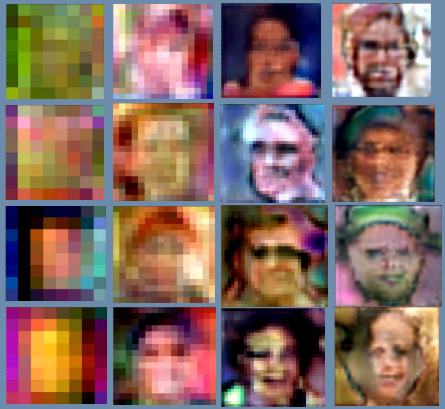
```
generator      = StyleBased_Generator(n_fc, dim_latent, dim_input).to(device)
discriminator  = Discriminator().to(device)
g_optim        = optim.Adam([{
    'params': generator.convs.parameters(),
    'lr'    : 0.001
}, {
    'params': generator.to_rgbs.parameters(),
    'lr'    : 0.001
}], lr=0.001, 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
```

生成されたFAKEイメージ



gpu性能不足により64x64イメージまで生成