# 3.4-B 模式生成：可以反映地理空间尺度的 SytleGAN

在研究城市地理空间模式时，对于高分辨率航拍（遥感）影像而言，如果直接应用原始的生成对抗网络（GAN），即使使用具有更好收敛效果的 WGAN，通常生成图像也很难达到理想效果。这主要因为地物对象具有不同的尺度（尺寸），应考虑有与SIFT（Scale-Invariant Feature Transform）（见\* 图像特征提取与动态街景视觉感知\*一章）算法所体现的尺度空间（scale space），能够改进生成数据（图像）的质量，也能够进一步验证尺度对城市地理空间模式的影响。

GAN 得以快速的发展，被广泛的应用于不同的领域，反应有尺度空间（粒度，图像分辨率）的模型有 PGGAN（Progressive growing of GANs）[1] ，及基于 PGGAN 模型发展的 StyleGAN（Style-Based Generator Architecture for Generative Adversarial Networks）系列，例如 StyleGAN[2]、StyleGAN2[3]、StyleGAN3[4]，及基于 StyleGAN 系列用于火星影像生成的 MarsGAN[5] 等。此次实验以 StyleGAN（1）为基础，结合作者提供的代码[6]（TensorFlow 版本），用迁移的 Pytorch 版本[7]计算，从地理空间数据视角解释 StyleGAN，并用于高分辨率航拍影像的生成、亦可用于模式变化及土地覆盖类型（Land Cover,LC）、DEM 等生成实验。

## 3.4.1 StyleGAN 阐释和 NAIP 航拍影像数据生成

### 3.4.1.1 迁移 StyleGAN

为了方便实验，将 StyleGAN（ Pytorch 版本） 迁移至 USDA 库，位于sda.migrated\_project.stylegan模块之下。迁移库之前需要验证原有库是否运行正常，测试无误后才可迁移。迁移过程中，通常需要根据未来操作模式进行对应的调整，例如 原 StyleGAN 使用[argparse](https://docs.python.org/3/library/argparse.html)配置部分参数。如果未来在 Jupyter[Lab] Notebook 下运行代码（例如用 CoLab 实验），则调整参数输入方式均为 [yacs](https://github.com/rbgirshick/yacs)库提供的属性字典方式，基本同 usda\_utils.AttrDict()提供的方法，但是增加了更多的功能，方便参数调用、合并、可变（mutable）和不可变（immutable）切换等；同时，需要调整模块、类、函数等调入的路径，配置\_\_init\_\_.py文件，保证 USDA 库下可以调用 StyleGAN 代码工具；如果为了增加代码的弹性，例如可以单独调用 StyleGAN 的或网络单独的块（block）或层（layer），如网络块InputBlock、GSynthesisBlock、DiscriminatorTop和DiscriminatorBlock等，如单独层PixelNormLayer、Upscale2d、Downscale2d、EqualizedLinear、EqualizedConv2d、NoiseLayer、StyleMod、LayerEpilogue、BlurLayer、View、StddevLayer和Truncation等，则可以将其包括在\_\_init\_\_.py下。

下述代码为迁移 StyleGAN 至 USDA 库，从 USDA 库调用 StyleGAN 生成 NAIP 航拍影像数据的方式。影像数据分辨率为 512，数量为 20,000张（影像样本数据生成方法具体查看*3.4-A 模式生成：从聚类模式特征到生成对抗网络和计算分析工具的建构*一章）。

最终计算在 CoLab 中结合 Google driver（个人云存储空间和文件共享平台） 完成。

# IPython extension to reload modules before executing user code.  
%load\_ext autoreload   
# Reload all modules (except those excluded by %aimport) every time before executing the Python code typed.  
%autoreload 2   
from usda.migrated\_project.stylegan import Stylegan\_train  
import usda.migrated\_project.stylegan as stylegan  
import usda.migrated\_project.stylegan.data as sgan\_data   
import usda.migrated\_project.stylegan.extracted\_funcs as sgan\_gadgets  
import usda.migrated\_project.stylegan.models as sgan\_model  
from usda.maths import plot\_single\_function  
  
import torch  
import math  
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
import random  
import torchvision

超参数（Hyperparameter）的配置尽量选择原作者已经实验验证可行的参数，如需调整参数，则以原参数为参照调试至满足不同分析目的，或为适应不同 GPU 算力要求。

sgan=Stylegan\_train()  
  
sgan.opt.output\_dir=r'I:\model\_ckpts\styleGAN4naip\sgan'  
sgan.opt.dataset.img\_dir=r'I:\data\NAIP4StyleGAN\naip\_1024'   
sgan.opt.dataset.folder=False  
sgan.opt.dataset.resolution=1024  
sgan.opt.structure='linear'  
sgan.opt.model.gen.mapping\_layers = 8 # Mapping network 部分全连接层数量配置，默认为 8（为 StyleGAN 作者参数配置）；StyleGAN代码 Pytorch 版作者默认配置为 4  
  
sgan.opt.sched.batch\_sizes = [128, 128, 128, 64, 32, 16, 8, 4, 2] # StyleGAN代码 Pytorch 版作者配置，（batches for oen 1080Ti with 11G memory）  
sgan.opt.epochs=[40,80,80,80,80,160,160,160]   
  
sgan.configuration()  
sgan.build\_dataset()  
sgan.init\_network()

2023-04-16 07:34:36 INFO: Using 1 GPUs.  
2023-04-16 07:34:36 INFO: Training on NVIDIA GeForce RTX 2080 with Max-Q Design.  
  
2023-04-16 07:34:39 INFO: Training from scratch...

sgan.train()

2023-04-16 07:34:54 INFO: Starting the training process ...   
  
2023-04-16 07:34:54 INFO: Currently working on depth: 1  
2023-04-16 07:34:54 INFO: Current resolution: 4 x 4  
2023-04-16 07:34:54 INFO: Epoch: [1]  
2023-04-16 07:36:01 INFO: Elapsed: [0:01:07] Step: 1 Batch: 1 D\_Loss: 714.139893 G\_Loss: 0.103811  
2023-04-16 07:36:06 INFO: Elapsed: [0:01:11] Step: 3 Batch: 3 D\_Loss: 409.393951 G\_Loss: 0.000000  
2023-04-16 07:36:11 INFO: Elapsed: [0:01:16] Step: 6 Batch: 6 D\_Loss: 412.592224 G\_Loss: 0.000000  
2023-04-16 07:36:19 INFO: Elapsed: [0:01:24] Step: 9 Batch: 9 D\_Loss: 468.190460 G\_Loss: 0.000314  
2023-04-16 07:36:26 INFO: Elapsed: [0:01:31] Step: 12 Batch: 12 D\_Loss: 371.070770 G\_Loss: 0.000432  
2023-04-16 07:36:37 INFO: Elapsed: [0:01:42] Step: 15 Batch: 15 D\_Loss: 270.685150 G\_Loss: 0.002868  
2023-04-16 07:36:51 INFO: Elapsed: [0:01:56] Step: 18 Batch: 18 D\_Loss: 380.751556 G\_Loss: 0.000240  
2023-04-16 07:37:05 INFO: Elapsed: [0:02:10] Step: 21 Batch: 21 D\_Loss: 278.914978 G\_Loss: 0.000945  
2023-04-16 07:37:12 INFO: Elapsed: [0:02:17] Step: 24 Batch: 24 D\_Loss: 249.346146 G\_Loss: 0.010437  
2023-04-16 07:37:18 INFO: Time taken for epoch: 0:02:23  
  
2023-04-16 07:37:21 INFO: Saving the model to: I:\model\_ckpts\styleGAN4naip\sgan\models\GAN\_GEN\_0\_1.pth  
  
2023-04-16 07:37:31 INFO: Saving the model to: I:\model\_ckpts\styleGAN4naip\sgan\models\GAN\_GEN\_SHADOW\_0\_1.pth  
  
2023-04-16 07:37:31 INFO: Epoch: [2]  
2023-04-16 07:38:42 INFO: Elapsed: [0:03:47] Step: 26 Batch: 1 D\_Loss: 297.541382 G\_Loss: 0.000000  
2023-04-16 07:38:47 INFO: Elapsed: [0:03:53] Step: 28 Batch: 3 D\_Loss: 260.714508 G\_Loss: 0.002044  
2023-04-16 07:38:55 INFO: Elapsed: [0:04:01] Step: 31 Batch: 6 D\_Loss: 238.747452 G\_Loss: 1.679931  
2023-04-16 07:39:02 INFO: Elapsed: [0:04:07] Step: 34 Batch: 9 D\_Loss: 198.448822 G\_Loss: 6.064939  
2023-04-16 07:39:07 INFO: Elapsed: [0:04:12] Step: 37 Batch: 12 D\_Loss: 206.848328 G\_Loss: 7.724971  
2023-04-16 07:39:14 INFO: Elapsed: [0:04:19] Step: 40 Batch: 15 D\_Loss: 248.303726 G\_Loss: 2.320208  
2023-04-16 07:39:22 INFO: Elapsed: [0:04:27] Step: 43 Batch: 18 D\_Loss: 230.896790 G\_Loss: 1.446756  
2023-04-16 07:39:29 INFO: Elapsed: [0:04:35] Step: 46 Batch: 21 D\_Loss: 193.671387 G\_Loss: 5.729475  
2023-04-16 07:39:34 INFO: Elapsed: [0:04:39] Step: 49 Batch: 24 D\_Loss: 161.947311 G\_Loss: 2.598769  
2023-04-16 07:39:38 INFO: Time taken for epoch: 0:02:06  
  
2023-04-16 07:39:38 INFO: Epoch: [3]  
...

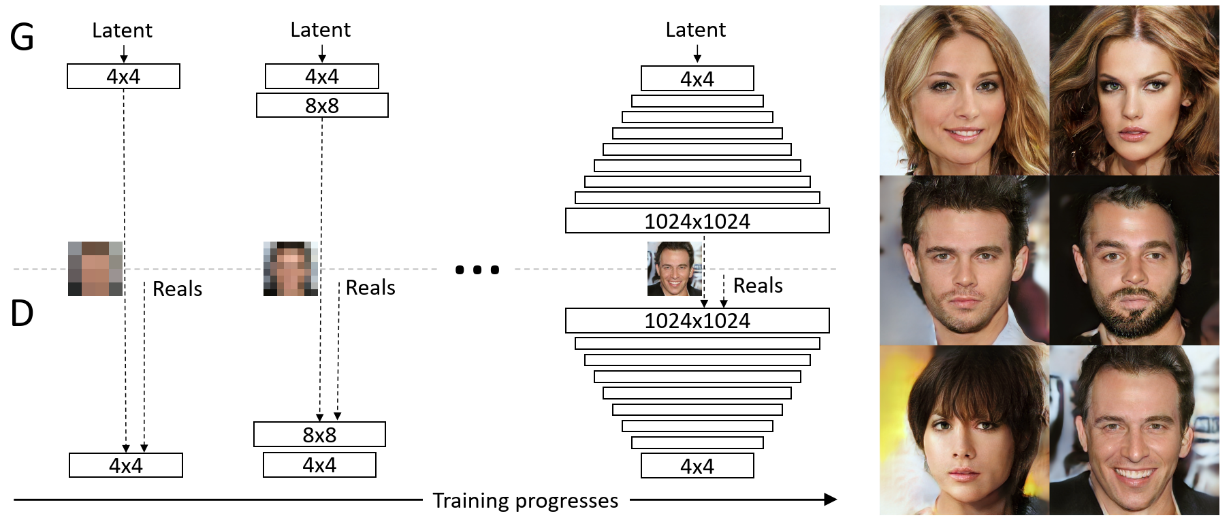
例举了各深度训练，开始和结束轮次生成图像。从各深度训练对应的生成图像可以清晰辨别出图像尺度空间（粒度，分辨率）的作用，及各深度基于上一深度训练结果的变化，虽然当前深度为上一深度像素单元的4倍，但各深度开始生成的图像与上一深度类似，即扩展的4倍栅格单元数值之间基本无差异，随着训练的进行，新扩展的栅格单元数值逐渐向真实图像当前深度的分布靠拢，扩展的4倍栅格单元数值之间产生差异变化。从生成结果可判断，使用 StyleGAN 模型训练高分辨率航拍影像能够获得真实影像数据分布，从而生成逼真的“假”图像。那么推断将其用于不同特定地貌、具有不同特征城市模式样本的训练是可行的。

| 深度（depth）（像素，resolution） | 开始 | 结束 |
| --- | --- | --- |
| 0 （4） | 0\_1\_1 （深度/depth\_轮次/epoch\_批次/batch） | 0\_40\_144 |
| 1 （8） | 1\_1\_1 | 1\_80\_144 |
| 2 (16) | 2\_1\_1 | 2\_80\_144 |
| 3 (32) | 3\_1\_1 | 3\_80\_288 |
| 4 (64) | 4\_1\_1 | 4\_80\_567 |
| 5 (128) | 5\_1\_1 | 5\_20\_1134 |
| 6 (256) | 6\_1\_1 | 6\_7\_1 |
| 7 (512) | 待 | 待 |

### 3.4.1.2 应用 StyleGAN 于地理空间数据分析上的阐释

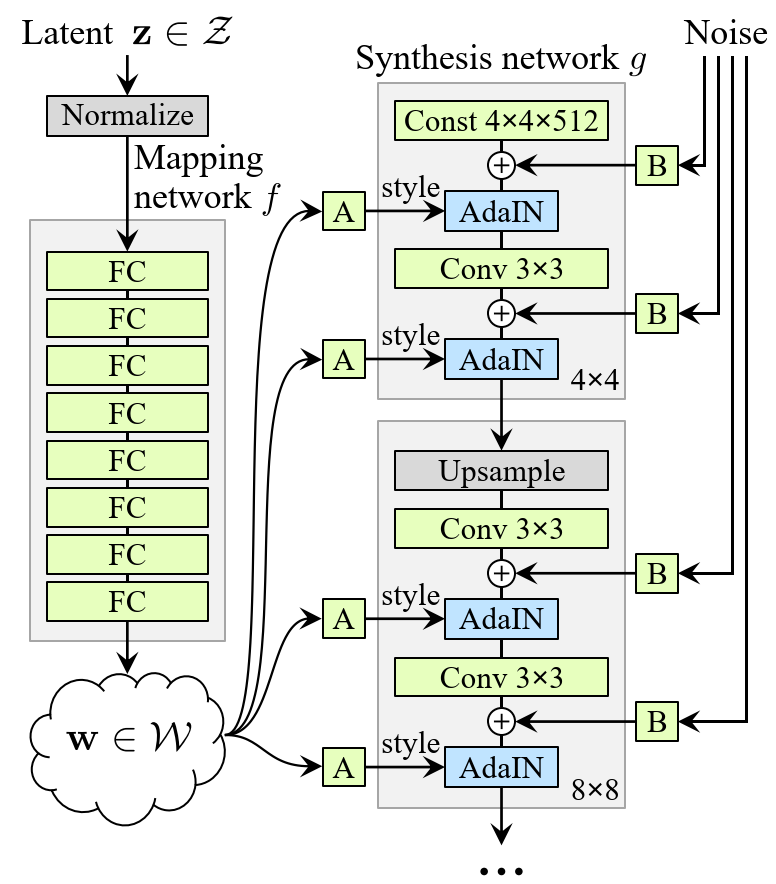
基于 StyleGAN 作者 Karras, T.等人的研究论文[2]和StyleGAN 的PyTroch版本代码[7]， 从地理空间数据分析方向阐释 StyleGAN 架构和关键细节。为了契合应用 StyleGAN 于地理空间数据分析的意义，调整了个别名词表述的方式，例如将“风格（style）”表述为“尺度空间”；对应不同分辨率的风格表述为尺度空间“深度（depth）”；对人脸的风格特征描述，例如人脸姿态、形状、发型特征、纹理和色彩等，转化为对地物或相关地理空间数据表达对象的描述，例如建筑、街道、植被和水体等；相应，不同分辨率的特征描述转化为不同尺度空间深度或不同空间尺度下模式结构特征的描述。

PGGAN 网络示意图[1]说明 PGGAN 训练从具有 低空间分辨率的的生成器和判别器开始，并随着训练的进行，逐渐向和中添加高分辨率的层（从图左至右），从而提高生成图像的空间分辨率。整个过程，所有图层都保持可训练状态。为在空间分辨率上运行的卷积层。虽然 PGGAN 能够生成高分辨率高品质的图像，但是不能分析不同空间分辨率的特征分布，也不能控制生成图像的样式趋向，这将限制基于尺度深度变化地理空间模式结构特征分析，及限制生成可控有意义的地理空间模式用于规划参考。



SytleGAN 则借助构建的Mapping network层（下图[2]）将各个空间尺度对应的地理空间特征分解开来（即从各个深度表征的图像特征相互纠缠（ entangle）到尽量相互独立的解缠过程（disentangle）），可以分析不同空间尺度特征和控制不同深度对生成（图像）空间模式的影响。

因为增加了Mapping network层，因此 StyleGAN 的网络总共包括两个部分，Mapping network，（MN）映射网络和Synthesis network，（SN）合成网络。



#### 1) MN 映射网络

MN 输入为属于隐藏（潜在）空间（latent space）的隐藏特征（编码）（latent code）（），即为 512 维的噪声向量。 通过 MN 的非线性映射网络f:Z→W （为8层全连接层（fully connected layer，FC，也称为 Dense layer）的多层感知机（ multi-layer perceptron，MLP ））映射到w**（**w**∈W）**AA

AfA之间不同深度的特征纠缠，即表征地物对象不同深度的分布特征会尽量分离。

因为 NAIP 航拍影像的高空分辨率为 1m，样本制作采样的单元大小为512，因此对应到空间尺度深度，深度7（512）的实际地理空间分辨率为1m，由对应逆序计算，深度6（256）的地理空间分辨率为4m， 深度5（128）为8m，深度4（64）为16m，深度3（32）为32m，深度2（16）为64m，深度1（8）为128m，至深度0（4）为256m。那么，

1. 对于空间尺度深度7，可以表征的地物尺寸约m，对象有道路划线、垃圾桶等城市设施、人等；
2. 对于深度6，可以表征的地物尺寸约m，对象有车辆、单株植被等；
3. 深度5，可以表征的地物尺寸约m，对象有小径、单株植被、建筑顶的空调系统等；
4. 深度4，可以表征的地物尺寸约m，对象有独栋住宅等建筑、成片的植被、城市道路等；
5. 深度3，可以表征的地物尺寸约m，对象有较大的建筑体、适中的停车场、街头绿地等；
6. 深度2，可以表征的地物尺寸约m，对象有大型建筑、室外活动场地、成片的绿地等；
7. 深度1，可以表征的地物尺寸约m，对象有小块街区、小块绿地或林冠等；
8. 深度0，可以表征的地物尺寸约m，对象有体育场馆等特大建筑、中等街区、中等绿地或活动空间等。

通过sgan\_model.GMapping()方法调出 MN 映射网络，因为配置了mapping\_layers参数为 8，因此总共有8个全连接层EqualizedLinear，该层的定义由class EqualizedLinear(nn.Module)类调用torch.nn.functional.linear(input, weight, bias=None) → Tensor实现。激活函数使用的为LeakyReLU。

mapping\_net=sgan\_model.GMapping()  
mapping\_net

GMapping(  
 (map): Sequential(  
 (pixel\_norm): PixelNormLayer()  
 (dense0): EqualizedLinear()  
 (dense0\_act): LeakyReLU(negative\_slope=0.2)  
 (dense1): EqualizedLinear()  
 (dense1\_act): LeakyReLU(negative\_slope=0.2)  
 (dense2): EqualizedLinear()  
 (dense2\_act): LeakyReLU(negative\_slope=0.2)  
 (dense3): EqualizedLinear()  
 (dense3\_act): LeakyReLU(negative\_slope=0.2)  
 (dense4): EqualizedLinear()  
 (dense4\_act): LeakyReLU(negative\_slope=0.2)  
 (dense5): EqualizedLinear()  
 (dense5\_act): LeakyReLU(negative\_slope=0.2)  
 (dense6): EqualizedLinear()  
 (dense6\_act): LeakyReLU(negative\_slope=0.2)  
 (dense7): EqualizedLinear()  
 (dense7\_act): LeakyReLU(negative\_slope=0.2)  
 )  
)

训练深度不同，空间尺度大小不同，输入神经元数量（input\_size）不同，为了保持不同深度趋于等价学习率的效果，调整（归一化） MN 层权重，深度较浅时（趋向 深度0），具有较大的学习率；深度较深时（趋向 深度8），学习率降低。调整的方式上，通过配置use\_wscale参数，固定init\_std或w\_mul，而变化另一个，两种方式结果相同（即结果值同）。在类EqualizedLinear中对应调整权重大小的位置分别为self.weight = torch.nn.Parameter(torch.randn(output\_size, input\_size) \* init\_std)和F.linear(x, self.weight \* self.w\_mul, bias)。

从类EqualizedLinear中提取归一化学习率的代码，定义为equalized\_lr()函数，演示数值变化如下：

def equalized\_lr(input\_size,gain=np.sqrt(2),use\_wscale=True,lrmul=0.8):  
 he\_std=gain \* input\_size \*\* (-0.5) # He init  
 if use\_wscale:  
 init\_std = 1.0 / lrmul  
 w\_mul = he\_std \* lrmul  
 else:  
 init\_std = he\_std / lrmul  
 w\_mul = lrmul   
   
 print(f'input size={input\_size};\tinit\_std={init\_std};\tw\_mul={w\_mul}')   
  
resolution=[pow(2,n) for n in range(2,11)]   
for res in resolution:  
 equalized\_lr(res,use\_wscale=True)  
print('-'\*50)  
for res in resolution:  
 equalized\_lr(res,use\_wscale=False)

input size=4; init\_std=1.25; w\_mul=0.5656854249492381  
input size=8; init\_std=1.25; w\_mul=0.40000000000000013  
input size=16; init\_std=1.25; w\_mul=0.28284271247461906  
input size=32; init\_std=1.25; w\_mul=0.20000000000000007  
input size=64; init\_std=1.25; w\_mul=0.14142135623730953  
input size=128; init\_std=1.25; w\_mul=0.10000000000000003  
input size=256; init\_std=1.25; w\_mul=0.07071067811865477  
input size=512; init\_std=1.25; w\_mul=0.05000000000000002  
input size=1024; init\_std=1.25; w\_mul=0.03535533905932738  
--------------------------------------------------  
input size=4; init\_std=0.8838834764831844; w\_mul=0.8  
input size=8; init\_std=0.6250000000000001; w\_mul=0.8  
input size=16; init\_std=0.4419417382415922; w\_mul=0.8  
input size=32; init\_std=0.31250000000000006; w\_mul=0.8  
input size=64; init\_std=0.2209708691207961; w\_mul=0.8  
input size=128; init\_std=0.15625000000000003; w\_mul=0.8  
input size=256; init\_std=0.11048543456039805; w\_mul=0.8  
input size=512; init\_std=0.07812500000000001; w\_mul=0.8  
input size=1024; init\_std=0.05524271728019903; w\_mul=0.8

下面构建了数据集和数据加载器，提取一批样本用于实验演示。输入的对象为gan\_input，批大小为128，维度为512，均值为0，方差为1标准正态分布的噪声向量，out 。

batch\_size=128  
num\_workers=4  
data\_loader=sgan\_data.get\_data\_loader(sgan.dataset,batch\_size=batch\_size,num\_workers=num\_workers)  
images=next(iter(data\_loader))  
print(images.shape)

torch.Size([128, 3, 1024, 1024])

ngpu=1  
device = torch.device("cuda:0" if (torch.cuda.is\_available() and ngpu > 0) else "cpu")  
  
latent\_size=sgan.opt.model.gen.latent\_size  
print(f'latent\_size={latent\_size}')  
gan\_input=torch.randn(images.shape[0], latent\_size) # .to(device)   
print(gan\_input.shape)

latent\_size=512  
torch.Size([128, 512])

fake\_samples=mapping\_net(gan\_input)  
print(fake\_samples.shape)

torch.Size([128, 512])

#### 2) SN 合成网络

* 常量输入

SN 的作用是生成图像，不同于 PGGAN 网络是给每一深度的子网络都喂入和，是习得对应深度 MN 映射网络仿射变换的结果，用于控制尺度空间（不同深度）的特征；是转化后的随机噪声，用于丰富生成图像的细节。SN 的输入移除了噪声向量（latent code），替代为常量输入（值为1），维度为，如下代码。

const=torch.nn.Parameter(torch.ones(1, 512, 4, 4)).expand(batch\_size,-1,-1,-1)  
print(f'const:{const.shape}')

const:torch.Size([128, 512, 4, 4])

* 注入噪声向量

在 AdaIN 层之前添加噪声向量，代码如下：

class NoiseLayer(nn.Module):  
 """adds noise. noise is per pixel (constant over channels) with per-channel weight"""  
  
 def \_\_init\_\_(self, channels):  
 super().\_\_init\_\_()  
 self.weight = nn.Parameter(torch.zeros(channels))  
 self.noise = None  
  
 def forward(self, x, noise=None):  
 if noise is None and self.noise is None:  
 noise = torch.randn(x.size(0), 1, x.size(2), x.size(3), device=x.device, dtype=x.dtype)  
 elif noise is None:  
 # here is a little trick: if you get all the noise layers and set each  
 # modules .noise attribute, you can have pre-defined noise.  
 # Very useful for analysis  
 noise = self.noise  
 x = x + self.weight.view(1, -1, 1, 1) \* noise  
 return x

当为 SN 网络输入层时，x为常量const，后续其它层则为对应深度卷积（上采样）后的图像。噪声置入的方式为训练的权重缩放噪声向量后与x相加。nn.Parameter方法将噪声权重值转换为可训练的参数并注册到模型参数中，其初始值为0，因此为了演示实验，在该章开始*迁移 StyleGAN*部分训练了几个轮次，因此噪声向量的值已经发生了改变，这里以输入层部分为例，通过sgan.style\_gan.gen方法提取 网络，用G.state\_dict()查看网络全部参数名称和对应的值，找到噪声权重名称（路径），通过G.g\_synthesis.init\_block.epi2.top\_epi.noise.weight方法提取噪声权重。

G=sgan.style\_gan.gen  
G.state\_dict().keys()

odict\_keys(['g\_mapping.map.dense0.weight', 'g\_mapping.map.dense0.bias', 'g\_mapping.map.dense1.weight', 'g\_mapping.map.dense1.bias', 'g\_mapping.map.dense2.weight', 'g\_mapping.map.dense2.bias', 'g\_mapping.map.dense3.weight', 'g\_mapping.map.dense3.bias', 'g\_mapping.map.dense4.weight', 'g\_mapping.map.dense4.bias', 'g\_mapping.map.dense5.weight', 'g\_mapping.map.dense5.bias', 'g\_mapping.map.dense6.weight', 'g\_mapping.map.dense6.bias', 'g\_mapping.map.dense7.weight', 'g\_mapping.map.dense7.bias', 'g\_synthesis.init\_block.const', 'g\_synthesis.init\_block.bias', 'g\_synthesis.init\_block.epi1.top\_epi.noise.weight', 'g\_synthesis.init\_block.epi1.style\_mod.lin.weight', 'g\_synthesis.init\_block.epi1.style\_mod.lin.bias', 'g\_synthesis.init\_block.conv.weight', 'g\_synthesis.init\_block.conv.bias', 'g\_synthesis.init\_block.epi2.top\_epi.noise.weight', 'g\_synthesis.init\_block.epi2.style\_mod.lin.weight', 'g\_synthesis.init\_block.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.0.conv0\_up.weight', 'g\_synthesis.blocks.0.conv0\_up.bias', 'g\_synthesis.blocks.0.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.0.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.0.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.0.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.0.conv1.weight', 'g\_synthesis.blocks.0.conv1.bias', 'g\_synthesis.blocks.0.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.0.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.0.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.1.conv0\_up.weight', 'g\_synthesis.blocks.1.conv0\_up.bias', 'g\_synthesis.blocks.1.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.1.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.1.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.1.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.1.conv1.weight', 'g\_synthesis.blocks.1.conv1.bias', 'g\_synthesis.blocks.1.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.1.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.1.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.2.conv0\_up.weight', 'g\_synthesis.blocks.2.conv0\_up.bias', 'g\_synthesis.blocks.2.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.2.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.2.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.2.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.2.conv1.weight', 'g\_synthesis.blocks.2.conv1.bias', 'g\_synthesis.blocks.2.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.2.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.2.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.3.conv0\_up.weight', 'g\_synthesis.blocks.3.conv0\_up.bias', 'g\_synthesis.blocks.3.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.3.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.3.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.3.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.3.conv1.weight', 'g\_synthesis.blocks.3.conv1.bias', 'g\_synthesis.blocks.3.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.3.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.3.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.4.conv0\_up.weight', 'g\_synthesis.blocks.4.conv0\_up.bias', 'g\_synthesis.blocks.4.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.4.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.4.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.4.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.4.conv1.weight', 'g\_synthesis.blocks.4.conv1.bias', 'g\_synthesis.blocks.4.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.4.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.4.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.5.conv0\_up.weight', 'g\_synthesis.blocks.5.conv0\_up.bias', 'g\_synthesis.blocks.5.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.5.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.5.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.5.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.5.conv1.weight', 'g\_synthesis.blocks.5.conv1.bias', 'g\_synthesis.blocks.5.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.5.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.5.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.6.conv0\_up.weight', 'g\_synthesis.blocks.6.conv0\_up.bias', 'g\_synthesis.blocks.6.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.6.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.6.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.6.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.6.conv1.weight', 'g\_synthesis.blocks.6.conv1.bias', 'g\_synthesis.blocks.6.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.6.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.6.epi2.style\_mod.lin.bias', 'g\_synthesis.blocks.7.conv0\_up.weight', 'g\_synthesis.blocks.7.conv0\_up.bias', 'g\_synthesis.blocks.7.conv0\_up.intermediate.kernel', 'g\_synthesis.blocks.7.epi1.top\_epi.noise.weight', 'g\_synthesis.blocks.7.epi1.style\_mod.lin.weight', 'g\_synthesis.blocks.7.epi1.style\_mod.lin.bias', 'g\_synthesis.blocks.7.conv1.weight', 'g\_synthesis.blocks.7.conv1.bias', 'g\_synthesis.blocks.7.epi2.top\_epi.noise.weight', 'g\_synthesis.blocks.7.epi2.style\_mod.lin.weight', 'g\_synthesis.blocks.7.epi2.style\_mod.lin.bias', 'g\_synthesis.to\_rgb.0.weight', 'g\_synthesis.to\_rgb.0.bias', 'g\_synthesis.to\_rgb.1.weight', 'g\_synthesis.to\_rgb.1.bias', 'g\_synthesis.to\_rgb.2.weight', 'g\_synthesis.to\_rgb.2.bias', 'g\_synthesis.to\_rgb.3.weight', 'g\_synthesis.to\_rgb.3.bias', 'g\_synthesis.to\_rgb.4.weight', 'g\_synthesis.to\_rgb.4.bias', 'g\_synthesis.to\_rgb.5.weight', 'g\_synthesis.to\_rgb.5.bias', 'g\_synthesis.to\_rgb.6.weight', 'g\_synthesis.to\_rgb.6.bias', 'g\_synthesis.to\_rgb.7.weight', 'g\_synthesis.to\_rgb.7.bias', 'g\_synthesis.to\_rgb.8.weight', 'g\_synthesis.to\_rgb.8.bias', 'truncation.avg\_latent'])

G=sgan.style\_gan.gen  
noise\_weight=G.g\_synthesis.init\_block.epi2.top\_epi.noise.weight  
print(noise\_weight.shape)  
noise\_weight[:10]

torch.Size([512])  
  
  
  
  
  
tensor([ 0.0422, 0.0046, 0.0530, 0.0048, 0.0055, 0.0790, -0.0131, 0.0242,  
 -0.0042, 0.0362], device='cuda:0', grad\_fn=<SliceBackward0>)

x=const.to(device)   
noise = torch.randn(x.size(0), 1, x.size(2), x.size(3), device=x.device, dtype=x.dtype)  
x = x + noise\_weight.view(1, -1, 1, 1) \* noise  
print(x.shape)  
x[0,0,:]

torch.Size([128, 512, 4, 4])  
  
  
  
  
  
tensor([[0.9824, 1.0417, 1.0030, 1.0089],  
 [0.9722, 1.0714, 1.0557, 1.0131],  
 [1.0221, 0.9826, 0.9829, 0.9618],  
 [0.9928, 0.9859, 1.0202, 0.9527]], device='cuda:0',  
 grad\_fn=<SliceBackward0>)

* 深度混合正则化（style mixing regularization）

为了降低 SytleGAN 生成器各深度表征特征的相关性，采用了混合正则化的训练技巧。在训练过程中，随机选择两个，latents\_in和latents2，通过 MN 映射网络mapping\_net(g\_mapping)对应得到两个，为dlatents\_in和dlatents2，给定随机混合概率mixing\_prob，在任意随机深度位置mixing\_cutoff，交换前后两段向量。

为演示混合正则化，提取class Generator(nn.Module)类中的代码，定义为depth\_mixing\_regularization()函数，通过下述打印信息可以观察算法逻辑。

def depth\_mixing\_regularization(mapping\_net,dlatents\_in,latents\_in,resolution,depth,mixing\_prob=0.9):   
 latents2 = torch.randn(latents\_in.shape).to(latents\_in.device)  
 dlatents2 = mapping\_net(latents2)  
 num\_layers = (int(np.log2(resolution)) - 1) \* 2   
   
 print(f'num\_layers={num\_layers}')  
 layer\_idx = torch.from\_numpy(np.arange(num\_layers)[np.newaxis, :, np.newaxis]).to(latents\_in.device)  
 print(f'layer\_idx:\n{layer\_idx}')  
 cur\_layers = 2 \* (depth + 1)  
 print(f'cur\_layers={cur\_layers}')  
 mixing\_cutoff = random.randint(1,cur\_layers) if random.random() < mixing\_prob else cur\_layers  
 print(f'mixing\_cutoff={mixing\_cutoff}')  
   
 dlatents\_in = dlatents\_in.unsqueeze(1).expand(-1, num\_layers, -1)   
 dlatents2 = dlatents2.unsqueeze(1).expand(-1, num\_layers, -1)  
   
 print(f'dlatents\_in shape={dlatents\_in.shape},dlatents2 shape={dlatents2.shape}')  
 dlatents\_in = torch.where(layer\_idx < mixing\_cutoff, dlatents\_in, dlatents2)  
 print(f'mixed dlatents\_in shape={dlatents\_in.shape}')  
   
 return dlatents\_in  
   
dlatents\_in=depth\_mixing\_regularization(mapping\_net=mapping\_net,dlatents\_in=fake\_samples,latents\_in=gan\_input,resolution=1024,depth=3)

num\_layers=18  
layer\_idx:  
tensor([[[ 0],  
 [ 1],  
 [ 2],  
 [ 3],  
 [ 4],  
 [ 5],  
 [ 6],  
 [ 7],  
 [ 8],  
 [ 9],  
 [10],  
 [11],  
 [12],  
 [13],  
 [14],  
 [15],  
 [16],  
 [17]]], dtype=torch.int32)  
cur\_layers=8  
mixing\_cutoff=3  
dlatents\_in shape=torch.Size([128, 18, 512]),dlatents2 shape=torch.Size([128, 18, 512])  
mixed dlatents\_in shape=torch.Size([128, 18, 512])

* AdaIN (adaptive instance normalization)（自适应实例归一化）

AdaIN 层是一个在生成对抗网络和风格化领域中应用非常广泛的归一化层，在风格编码任务中可以替换BatchNorm2d（批归一化层）优化效果，其公式为：，其中部分通过torch.nn.InstanceNorm2d（实例标准化）方法实现；缩放因子w经过定义类class EqualizedLinear(nn.Module)可学习的仿射变换后，通过[-1, 2, x.size(1)] + (x.dim() - 2) \* [1]形状，即[-1, 2, 512, 1, 1]，生成转化为缩放因子和偏差因子。代码如下：

class StyleMod(nn.Module):  
 def \_\_init\_\_(self, latent\_size, channels, use\_wscale):  
 super(StyleMod, self).\_\_init\_\_()  
 self.lin = EqualizedLinear(latent\_size,  
 channels \* 2,  
 gain=1.0, use\_wscale=use\_wscale)  
  
 def forward(self, x, latent):  
 style = self.lin(latent) # style => [batch\_size, n\_channels\*2]  
  
 shape = [-1, 2, x.size(1)] + (x.dim() - 2) \* [1]  
 style = style.view(shape) # [batch\_size, 2, n\_channels, ...]  
 x = x \* (style[:, 0] + 1.) + style[:, 1]  
 return x

演示 AdaIN 层，直接通过sgan\_model.StyleMod方法调用StyleMod类，传入噪声向量x时，先执行InstanceNorm2d再传入。

dlatents\_in\_range=dlatents\_in[:, 0:2]  
print(f'dlatents\_in\_range shape={dlatents\_in\_range.shape}')  
  
stylemod=sgan\_model.StyleMod(latent\_size=512, channels=512, use\_wscale=True)  
  
m=torch.nn.InstanceNorm2d(512)  
x=m(x)  
x\_stylemod=stylemod(x.to(dlatents\_in\_range.device),dlatents\_in\_range[:, 0])  
print(x\_stylemod.shape)

dlatents\_in\_range shape=torch.Size([128, 2, 512])  
torch.Size([128, 512, 4, 4])

* 卷积上采样

对分辨率为128及其以上深度的图像使用逆卷积ConvTranspose2d方法，对于分辨率小于128深度图像使用最近邻上采样，对应的类为class Upscale2d(nn.Module)。

upscale2d=sgan\_model.Upscale2d(factor=2, gain=1)  
upscale2d\_x=upscale2d(x\_stylemod)  
upscale2d\_x.shape

torch.Size([128, 512, 8, 8])

* 中的截断（truncation）

对于训练数据分布明显的低密度区域，很难学习，因此从截断或紧缩的采样空间提取隐藏向量往往会提高平均图像质量。首先计算的质心，公式为。然后将给定的与中心的偏差（deviation）缩放为，式中ψ<1，表示压缩倍数。

截断对应定义的类为class Truncation(nn.Module)，代码如下：

class Truncation(nn.Module):  
 def \_\_init\_\_(self, avg\_latent, max\_layer=8, threshold=0.7, beta=0.995):  
 super().\_\_init\_\_()  
 self.max\_layer = max\_layer  
 self.threshold = threshold  
 self.beta = beta  
 self.register\_buffer('avg\_latent', avg\_latent)  
  
 def update(self, last\_avg):  
 self.avg\_latent.copy\_(self.beta \* self.avg\_latent + (1. - self.beta) \* last\_avg)  
  
 def forward(self, x):  
 assert x.dim() == 3  
 interp = torch.lerp(self.avg\_latent, x, self.threshold)  
 do\_trunc = (torch.arange(x.size(1)) < self.max\_layer).view(1, -1, 1).to(x.device)  
 return torch.where(do\_trunc, interp, x)

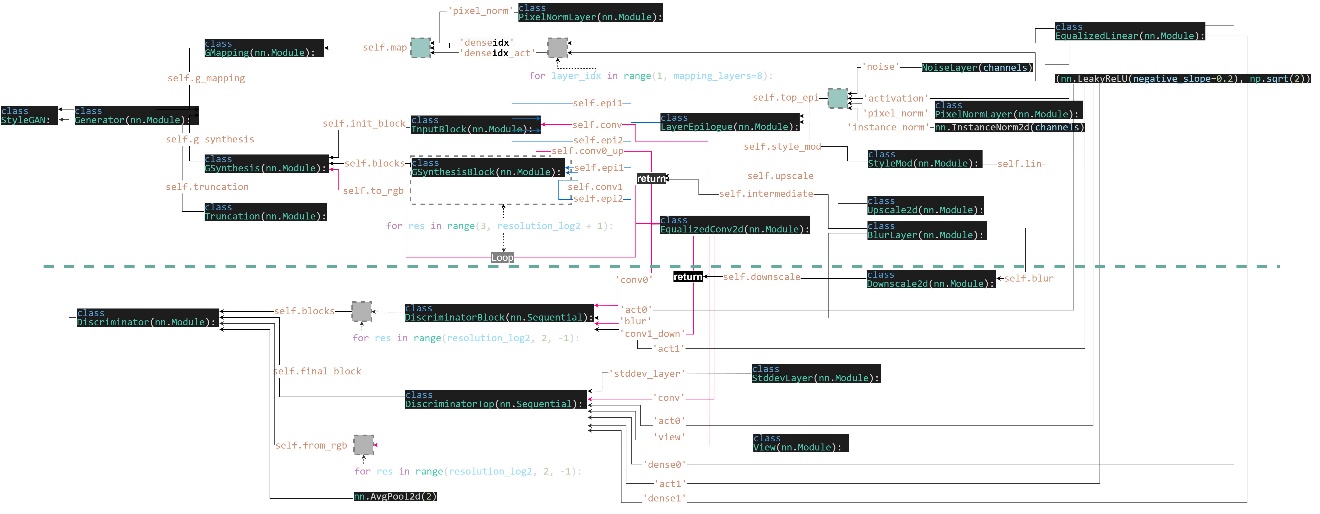
其中avg\_latent注册在模型参数的缓冲区，允许PyTorch跟踪并被模型保存，但是不会在训练过程中使用 SGD 学习更新，但可自定义更新，例如上述update方法。从模型中提取avg\_latent参数值用于演示 的截断逻辑如下。

avg\_latent=G.truncation.avg\_latent  
print(f'avg\_latent shape={avg\_latent.shape}')  
dlatents\_in=dlatents\_in.to(avg\_latent.device)  
interp=torch.lerp(avg\_latent, dlatents\_in, 0.7)  
max\_layer=8  
do\_trunc=(torch.arange(dlatents\_in.size(1)) <max\_layer).view(1, -1, 1).to(dlatents\_in.device)  
print(f'do\_trunc=\n{do\_trunc}')  
trunc\_dlatents\_in=torch.where(do\_trunc, interp, dlatents\_in)  
print(f'trunc\_dlatents\_in shape={trunc\_dlatents\_in.shape}')

avg\_latent shape=torch.Size([512])  
do\_trunc=  
tensor([[[ True],  
 [ True],  
 [ True],  
 [ True],  
 [ True],  
 [ True],  
 [ True],  
 [ True],  
 [False],  
 [False],  
 [False],  
 [False],  
 [False],  
 [False],  
 [False],  
 [False],  
 [False],  
 [False]]], device='cuda:0')  
trunc\_dlatents\_in shape=torch.Size([128, 18, 512])

#### 3) 对应 StyleGAN 网络的代码结构图

Karras, T. 在论文中给出了 StyleGAN 网络简图，可以用于理解网络的基本结构，但是不容易对应到代码细节的书写上和对已有代码结构细节的理解上，于是基于 StyleGAN 的 PyTorch 版本[7]，梳理 StyleGAN 网络对应的代码类和函数如下：



该作者书写代码规范并具有弹性，StyleGAN 包括和两个网络，并对应到各自的类Generator和Discriminator。Generator含三个部分self.g\_mapping、self.g\_synthesis和self.truncation，对应3个定义的类，以此类推。其中定义self.g\_synthesis时，以块（block）的方式组织代码，有self.init\_block和self.blocks，方便更深层级的组织、调用和条件编写。将前文对的解释，对照到该图对应的类或函数上，可以辅助阅读源代码。

## 3.4.2 用训练的模型生成航拍遥感影像及尺度空间深度特征控制

### 3.4.2.1 用训练的模型生成航拍遥感影像

训练时保存了多个类型网络模型文件，、及各自的优化器等，在应用训练的模型生成图像时仅使用网络。首先构建同训练时相同的网络，配置有相同的参数，读取训练好的模型更新参数后，用生成的噪声向量生成图像。在 USDA 库中定义G\_imgs类用于由训练好的网络模型生成图像实验。噪声向量生成部分如下代码：

with torch.no\_grad():  
 point = torch.randn(1, latent\_size)  
 point = (point / point.norm()) \* (latent\_size \*\* 0.5)  
 ss\_image = gen(point, depth=out\_depth, alpha=1)  
 # color adjust the generated image:  
 ss\_image = adjust\_dynamic\_range(ss\_image)

生成的数据位于[-1,1]之间，需要经过定义的adjust\_dynamic\_range函数将其映射到[0,1]区间，转化的方法如下：

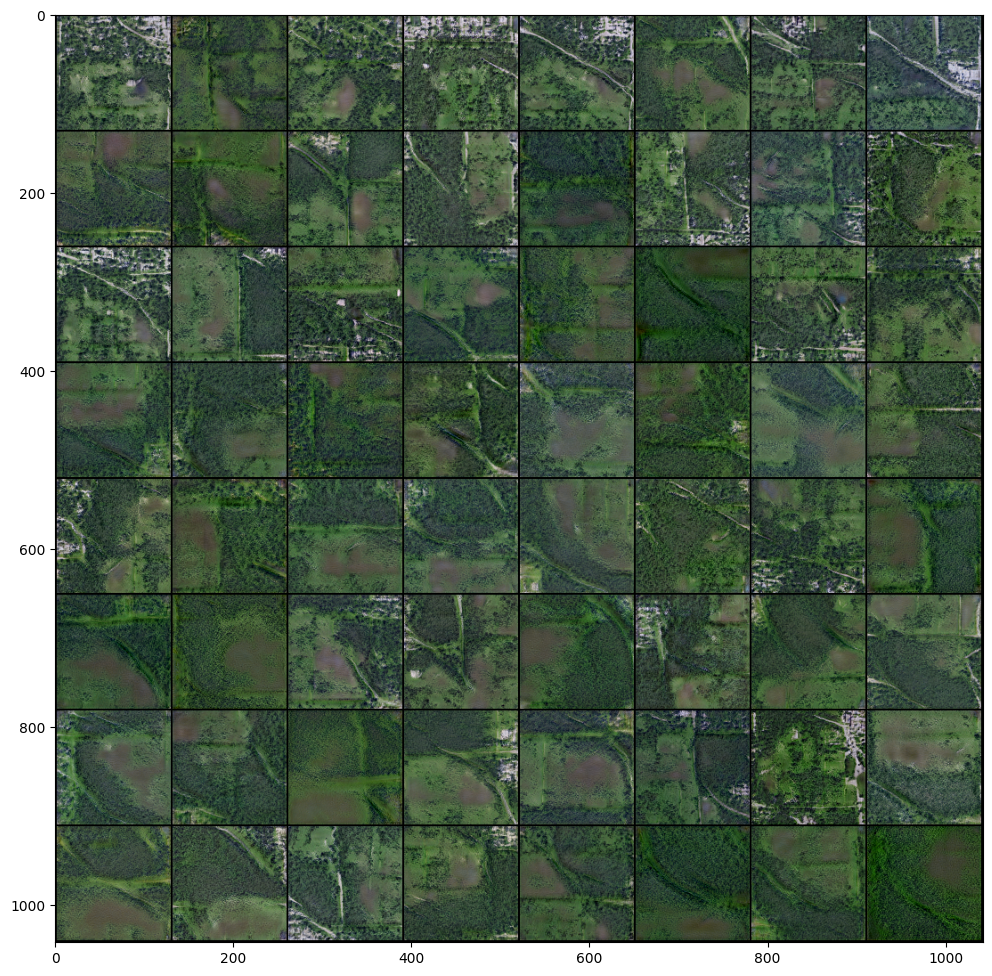
def adjust\_dynamic\_range(data, drange\_in=(-1, 1), drange\_out=(0, 1)):  
 """  
 adjust the dynamic colour range of the given input data  
 :param data: input image data  
 :param drange\_in: original range of input  
 :param drange\_out: required range of output  
 :return: img => colour range adjusted images  
 """  
 if drange\_in != drange\_out:  
 scale = (np.float32(drange\_out[1]) - np.float32(drange\_out[0])) / (  
 np.float32(drange\_in[1]) - np.float32(drange\_in[0]))  
 bias = (np.float32(drange\_out[0]) - np.float32(drange\_in[0]) \* scale)  
 data = data \* scale + bias  
 return torch.clamp(data, min=0, max=1)

下面生成了 64 个影像，较之 WGAN 网络有更好的表现。

g\_model\_path=r'I:\model\_ckpts\StyleGAN\_trained\_model\_8\GAN\_GEN\_6\_1.pth'  
pretrained\_G=stylegan.G\_imgs(g\_model\_path)  
pretrained\_G.opt.num\_samples=64  
g\_imgs=pretrained\_G.generate\_imgs()

Creating generator object ...  
Loading the generator weights from: I:\model\_ckpts\StyleGAN\_trained\_model\_8\GAN\_GEN\_6\_1.pth  
Generating scale synchronized images ...  
  
  
100%|████████████████████████████████████████████████████████████████████████| 64/64 [00:23<00:00, 2.76it/s]

g\_imgs\_strack=torch.stack(g\_imgs).squeeze(dim=1)  
grid\_img=torchvision.utils.make\_grid(g\_imgs\_strack, nrow=8)  
  
fig, ax = plt.subplots(figsize=(10,10))  
ax.imshow(grid\_img.permute(1, 2, 0))  
fig.tight\_layout()  
plt.show()



### 3.4.2.2 截断空间特征分析

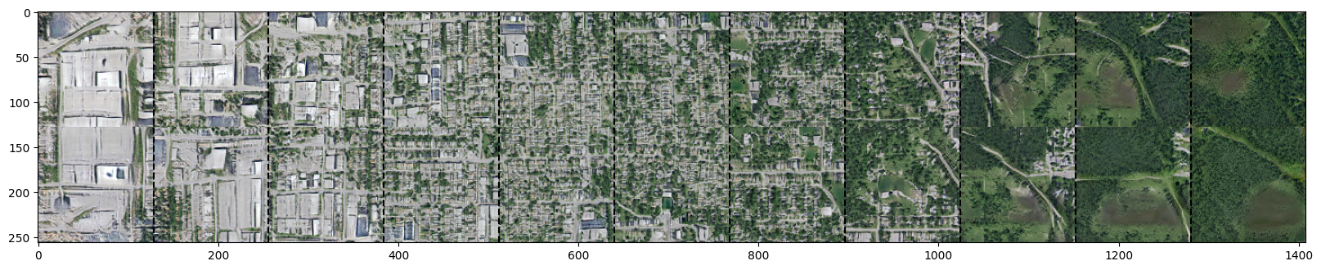
当  ψ*↦0*  *w(*ψ*=0)* ψ趋向-1（调换了方向），或1时，显现两个截然相反特征的影像信息，为大体量建筑布局和植被森林分布。趋向于0的位置为这两类特征的融合。下述代码定义了11个值 ψ $截断的方法可以有目的性的控制两类极端特征的融合分布，对于航拍影像而言为建筑和植被的融合程度，及建筑体量和植被破碎化程度的控制。

截断核心代码如下：

with torch.no\_grad():  
 latents\_np = np.stack([np.random.RandomState(seed).randn(latent\_size) for seed in self.opt.seeds])  
 latents = torch.from\_numpy(latents\_np.astype(np.float32))  
 dlatents = self.gen.g\_mapping(latents).detach().numpy()   
 dlatent\_avg = self.gen.truncation.avg\_latent.numpy()   
  
 canvas = Image.new('RGB', (w \* len(self.opt.psis), h \* len(self.opt.seeds)), 'white')  
 for row, dlatent in enumerate(list(dlatents)):  
 row\_dlatents = (dlatent[np.newaxis] - dlatent\_avg) \* np.reshape(self.opt.psis, [-1, 1, 1]) + dlatent\_avg  
 row\_dlatents = torch.from\_numpy(row\_dlatents.astype(np.float32))  
 row\_images = self.gen.g\_synthesis(row\_dlatents, depth=self.opt.out\_depth, alpha=1)  
 for col, image in enumerate(list(row\_images)):  
 image = adjust\_dynamic\_range(image)  
 image = image.mul(255).clamp(0, 255).byte().permute(1, 2, 0).numpy()  
 canvas.paste(Image.fromarray(image, 'RGB'), (col \* w, row \* h))

pretrained\_G\_trunc=stylegan.G\_truncation\_imgs(g\_model\_path)  
pretrained\_G\_trunc.opt.out\_depth=5  
pretrained\_G\_trunc.opt.psis=[-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1]  
trunc\_imgs=pretrained\_G\_trunc.draw\_truncation\_trick\_figure()  
  
fig, ax = plt.subplots(figsize=(20, 10))  
ax.imshow(trunc\_imgs)  
for x in list(range(128,1408,128)):  
 plt.axvline(x=x, color = 'k', linestyle='--')  
plt.show()

Creating generator object ...  
Loading the generator weights from: I:\model\_ckpts\StyleGAN\_trained\_model\_8\GAN\_GEN\_6\_1.pth



### 3.4.2.3 地理空间数据尺度空间深度融合可行性

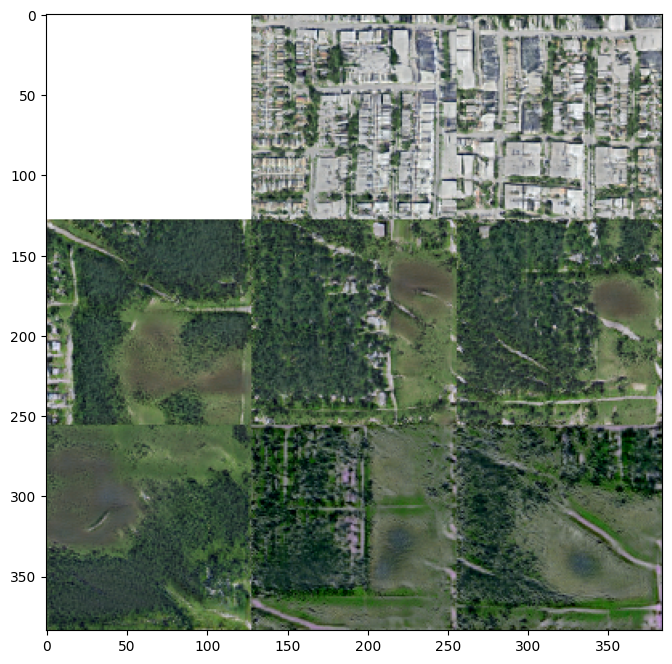
GAN、DCGAN、WGAN、StyleGAN 等众多生成对抗网络所训练的数据集通常为，人脸，如[Celeb-A Faces dataset](http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html)，[FFHQ](https://github.com/DmitryUlyanov/texture_nets)等；[CIFAR-10](https://www.cs.toronto.edu/~kriz/cifar.html)中的各类分类数据，例如鸟类、各类动物、各类交通工具等；及风景、漫画类。用 StyleGAN 于地理空间数据生成影像等各类信息数据，同样可以利用“解缠”尺度空深度信息融合不同特征影像，例如下述实验，通过截断，配置ψ为-0.5和0.7获取不同分布变换后的特征影像，一类偏于建筑，一类偏于植被自然，通过融合不同深度[(0,5),(5,10)]区间，生成了建筑隐于丛林特征的影像。其核心代码如下：

with torch.no\_grad():  
 latent\_size = self.gen.g\_mapping.latent\_size  
 src\_latents\_np = np.stack([np.random.RandomState(seed).randn(latent\_size, ) for seed in self.opt.src\_seeds])  
 dst\_latents\_np = np.stack([np.random.RandomState(seed).randn(latent\_size, ) for seed in self.opt.dst\_seeds])   
  
 src\_latents = torch.from\_numpy(src\_latents\_np.astype(np.float32))  
 dst\_latents = torch.from\_numpy(dst\_latents\_np.astype(np.float32))  
  
  
 src\_dlatents = self.gen.g\_mapping(src\_latents)   
 dst\_dlatents = self.gen.g\_mapping(dst\_latents)   
  
 #------------------------------------------------------------------  
 dlatent\_avg = self.gen.truncation.avg\_latent.numpy()   
  
 src\_dlatents = src\_dlatents.detach().numpy()   
 dst\_dlatents = dst\_dlatents.detach().numpy()   
  
 src\_dlatents = (src\_dlatents - dlatent\_avg) \* np.reshape(self.opt.src\_psis, [-1, 1,1]) + dlatent\_avg  
 dst\_dlatents = (dst\_dlatents - dlatent\_avg) \* np.reshape(self.opt.dst\_psis, [-1, 1,1]) + dlatent\_avg  
  
 src\_dlatents = torch.from\_numpy(src\_dlatents.astype(np.float32))  
 dst\_dlatents = torch.from\_numpy(dst\_dlatents.astype(np.float32))  
 #------------------------------------------------------------------  
  
 src\_images = self.gen.g\_synthesis(src\_dlatents, depth=self.opt.out\_depth, alpha=1)  
 dst\_images = self.gen.g\_synthesis(dst\_dlatents, depth=self.opt.out\_depth, alpha=1)  
  
 src\_dlatents\_np = src\_dlatents.numpy()  
 dst\_dlatents\_np = dst\_dlatents.numpy()  
 canvas = Image.new('RGB', (w \* (n\_col + 1), h \* (n\_row + 1)), 'white')  
 for col, src\_image in enumerate(list(src\_images)):  
 src\_image = adjust\_dynamic\_range(src\_image)  
 src\_image = src\_image.mul(255).clamp(0, 255).byte().permute(1, 2, 0).numpy()  
 canvas.paste(Image.fromarray(src\_image, 'RGB'), ((col + 1) \* w, 0))  
 for row, dst\_image in enumerate(list(dst\_images)):  
 dst\_image = adjust\_dynamic\_range(dst\_image)  
 dst\_image = dst\_image.mul(255).clamp(0, 255).byte().permute(1, 2, 0).numpy()  
 canvas.paste(Image.fromarray(dst\_image, 'RGB'), (0, (row + 1) \* h))  
  
 row\_dlatents = np.stack([dst\_dlatents\_np[row]] \* n\_col)  
 row\_dlatents[:, self.opt.style\_ranges[row]] = src\_dlatents\_np[:, self.opt.style\_ranges[row]]  
 row\_dlatents = torch.from\_numpy(row\_dlatents)  
  
 row\_images = self.gen.g\_synthesis(row\_dlatents, depth=self.opt.out\_depth, alpha=1)  
 for col, image in enumerate(list(row\_images)):  
 image = adjust\_dynamic\_range(image)  
 image = image.mul(255).clamp(0, 255).byte().permute(1, 2, 0).numpy()  
 canvas.paste(Image.fromarray(image, 'RGB'), ((col + 1) \* w, (row + 1) \* h))

pretrained\_G\_mixing=stylegan.G\_depth\_mixing\_imgs(g\_model\_path)  
pretrained\_G\_mixing.opt.out\_depth=5  
pretrained\_G\_mixing.opt.src\_seeds=[730, 904]   
pretrained\_G\_mixing.opt.dst\_seeds=[387, 190]  
pretrained\_G\_mixing.opt.style\_ranges=[(0,5),(5,10)]  
pretrained\_G\_mixing.opt.src\_psis=[-0.5,-0.5]  
pretrained\_G\_mixing.opt.dst\_psis=[0.7,0.7]  
  
mixing\_imgs=pretrained\_G\_mixing.draw\_depth\_mixing\_figure()

Creating generator object ...  
Loading the generator weights from: I:\model\_ckpts\StyleGAN\_trained\_model\_8\GAN\_GEN\_6\_1.pth

fig, ax = plt.subplots(figsize=(8, 8))  
ax.imshow(mixing\_imgs)  
plt.show()



注释（Notes）：

① argparse，（<https://docs.python.org/3/library/argparse.html>）。

② yacs，（<https://github.com/rbgirshick/yacs>）。

③ eleb-A Faces dataset，（<http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>）。

④ FFHQ，（<https://github.com/DmitryUlyanov/texture_nets>）。

⑤ CIFAR-10，（<https://www.cs.toronto.edu/~kriz/cifar.html>）。

参考文献（References）:

[1] Karras, T., Aila, T., Laine, S. & Lehtinen, J. Progressive Growing of GANs for Improved Quality, Stability, and Variation. (2017).

[2] Karras, T., Laine, S. & Aila, T. A Style-Based Generator Architecture for Generative Adversarial Networks. (2018).

[3] Karras, T. et al. Analyzing and Improving the Image Quality of StyleGAN. (2019).

[4] Karras, T. et al. Alias-Free Generative Adversarial Networks. https://nvlabs.github.io/stylegan3.

[5] MarsGAN, <https://github.com/kheyer/MarsGAN>

[6] StyleGAN — Official TensorFlow Implementation, <https://github.com/NVlabs/stylegan>

[7] StyleGAN.pytorch, <https://github.com/huangzh13/StyleGAN.pytorch>