资料来源：www.gwylab.com

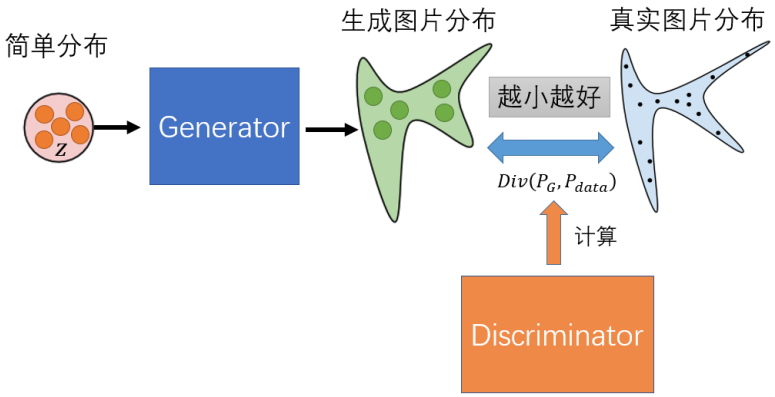
StyleGAN---基于样式的生成对抗网络学习笔记

——更新于 2019 年 11 月 21 日

目 录

前 言

先简单回顾一下前面说到的 GANs 在做的事情，用一句话来说就是，在判别器的帮助下，生成器实现了从简单分布到复杂分布的转换，如下图所示。

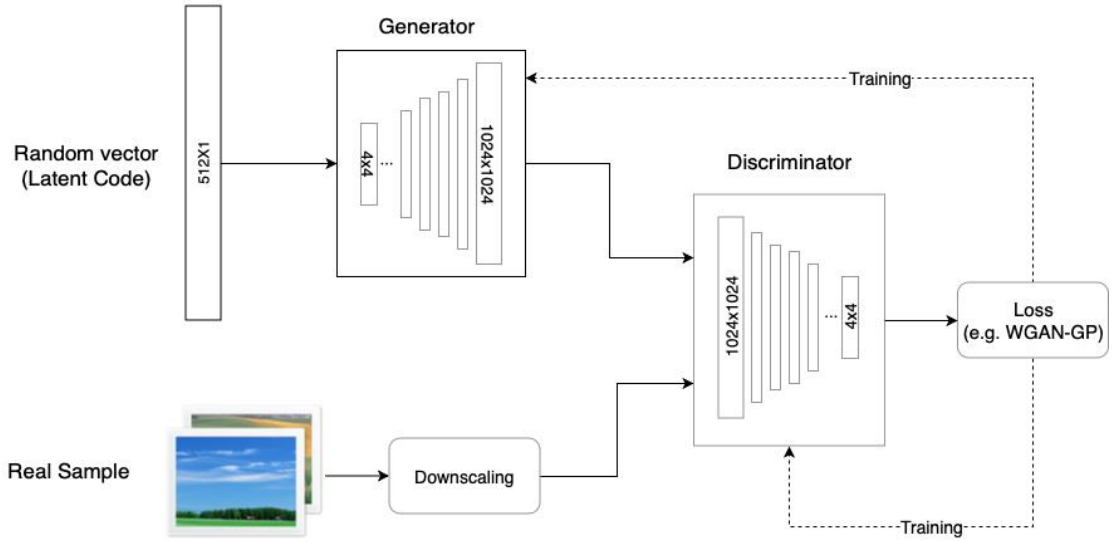


现在我们要做一个有点挑战性的事情，让 GANs 能生成高质量的大图像（1024*1024）。为什么说这是一件有挑战性的事情呢，因为在一个彩色的 1024*1024 大小的图片空间中，总共存在的图片样本点为$256^{1024*1024*3}$个（图片深度为 3，像素值为 0-255），而现在我们要让 GANs 学会在这样一个庞大的图片空间中找出真实图片的分布区域，并且建立一个从简单分布到这一复杂分布的转换，任务量无疑是巨大的。

很显然，我们不能考虑直接实现这样的转换，一个比较自然的想法是由小至大，逐级生成。也就是，先使用非常低分辨率的图像（如：4×4）开始训练生成器和判别器，并且逐次增加一个更高分辨率的网络层，直到最后能产生一个具有丰富细节的高清图片。基于这一想法，ProGAN（渐进式生成对抗网络）诞生了。

第一章 StyleGAN 原理介绍

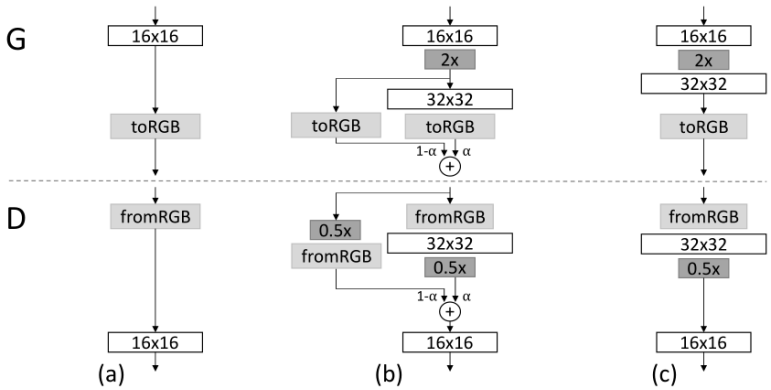
1.1 StyleGAN 的前身——ProGAN



上图是 ProGAN 的网络结构图。这项技术首先通过学习即使在低分辨率图像中也可以显示的基本特征，来创建图像的基本部分，并且随着分辨率的提高和时间的推移，学习越来越多的细节。低分辨率图像的训练不仅简单、快速，而且有助于更高级别的训练，因此，整体的训练也就更快。

特别值得注意的是，上图 Generator 中的网络结构不是指的从 4*4 网络连接到 8*8 网络，再连接到 16*16 网络依次输出，而是指的从 4*4 网络变化到 8*8 网络，再变化到 16*16 网络。也就是说，Generator 内部的网络只有一个，但是在训练过程中网络的结构是在动态变化的。事实上，前面那种依次连接的网络模型叫做 StackGAN，但是 StackGAN 不适合用来做超清图片生成，因为会特别慢。

不过，ProGAN 网络结构的动态变化是如何做到的呢？因为如果从 4×4 的输出直接变为 8×8 的输出的话，网络层数的突变会造成 GANs 原有参数失效，导致急剧不稳定，这会影响模型训练的效率。为了解决这一问题，PGGAN 提出了一种平滑过渡技术。



如上图所示，当把生成器和判别器的分辨率加倍时，会平滑地增大新的层。我们以从 16 × 16 像素的图片转换到 32 × 32 像素的图片为例。在转换（b）过程中，把在更高分辨率上操作的层视为一个残缺块，权重 α 从 0 到 1 线性增长。当 α 为 0 的时候，相当于图

(a),当 α 为 1 的时候，相当于图(c)。所以，在转换过程中，生成样本的像素，是从 16x16 到 32x32 转换的。同理，对真实样本也做了类似的平滑过渡，也就是，在这个阶段的某个训练 batch，真实样本是: $\mathrm{X} = \mathrm{X}_{16\mathrm{pixel}} * (1-\alpha) + \mathrm{X}_{32\mathrm{pixel}} * \alpha$。

上图中的 2× 和 0.5× 指利用最近邻卷积和平均池化分别对图片分辨率加倍和折半。toRGB 表示将一个层中的特征向量投射到 RGB 颜色空间中，fromRGB 正好是相反的过程；这两个过程都是利用 1 × 1 卷积。当训练判别器时，插入下采样后的真实图片去匹配网络中的当前分辨率。在分辨率转换过程中，会在两张真实图片的分辨率之间插值，类似于将两个分辨率结合到一起用生成器输出。详细的过程可以参见 ProGAN 论文。

上述就是 ProGAN 的模型设计介绍了，接下来我们反思一下 ProGAN 有什么缺陷。由于 ProGAN 是逐级直接生成图片，我们没有对其增添控制，我们也就无法获知它在每一级上学到的特征是什么，这就导致了它控制所生成图像的特定特征的能力非常有限。换句话说，这些特性是互相关联的，因此尝试调整一下输入，即使是一点儿，通常也会同时影响多个特性。

我们希望有一种更好的模型，能让我们控制住输出的图片是长什么样的，也就是在生成图片过程中每一级的特征，要能够特定决定生成图片某些方面的表象，并且相互间的影响尽可能小。于是，在 ProGAN 的基础上，StyleGAN 作出了进一步的改进与提升。

1.2 StyleGAN 架构解读

StyleGAN 首先重点关注了 ProGAN 的生成器网络，它发现，渐进层的一个潜在的好处是，如果使用得当，它们能够控制图像的不同视觉特征。层和分辨率越低，它所影响的特征就越粗糙。简要将这些特征分为三种类型:

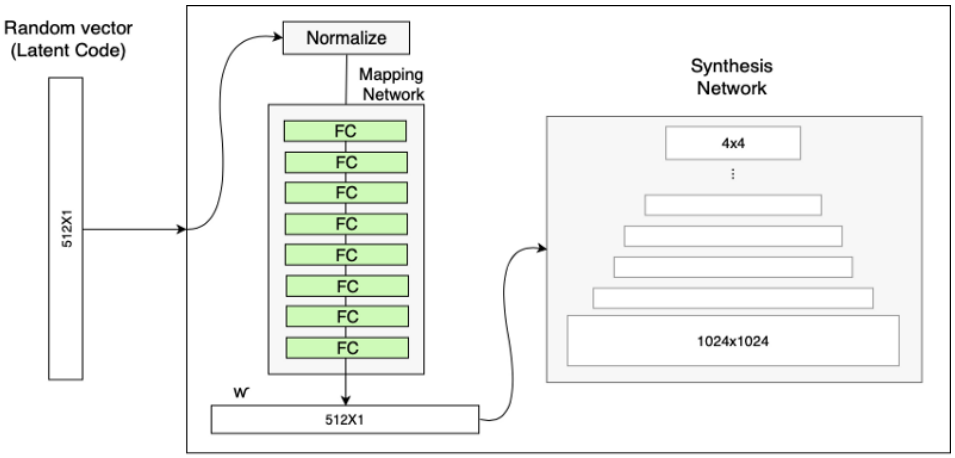
1、粗糙的——分辨率不超过8^2，影响姿势、一般发型、面部形状等;

2、中等的——分辨率为16^2至32^2，影响更精细的面部特征、发型、眼睛的睁开或是闭合等;

3、高质的——分辨率为64^2到1024^2，影响颜色（眼睛、头发和皮肤）和微观特征;

然后，StyleGAN 就在 ProGAN 的生成器的基础上增添了很多附加模块以实现样式上更细微和精确的控制。

1.2.1 映射网络



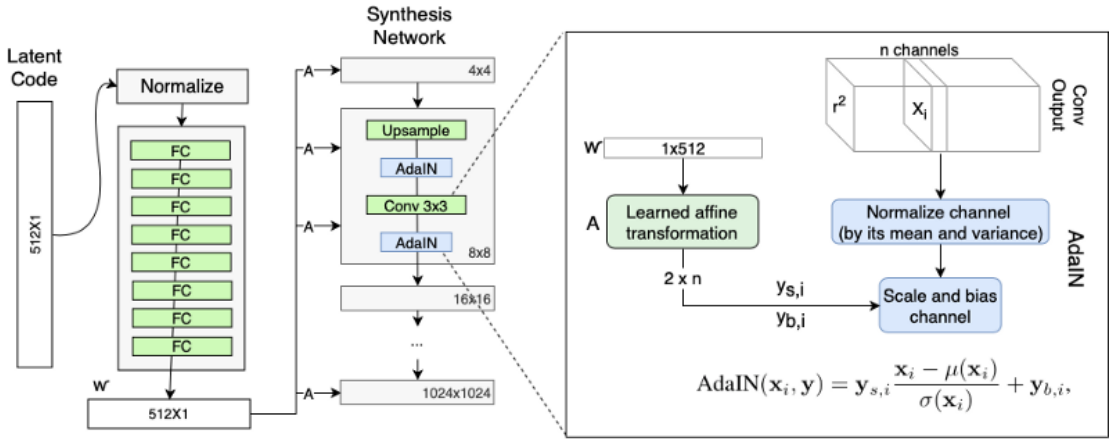
StyleGAN 的第一点改进是，给 Generator 的输入加上了由 8 个全连接层组成的 Mapping Network，并且 Mapping Network 的输出W'与输入层（512×1）的大小相同。

添加 Mapping Network 的目标是将输入向量编码为中间向量，并且中间向量后续会传给生成网络得到 18 个控制向量，使得该控制向量的不同元素能够控制不同的视觉特征。为何要加 Mapping Network 呢？因为如果不加这个 Mapping Network 的话，后续得到的 18 个控制向量之间会存在特征纠缠的现象——比如说我们想调节 8*8 分辨率上的控制向量（假设它能控制人脸生成的角度），但是我们会发现 32*32 分辨率上的控制内容（譬如肤色）也被改变了，这个就叫做特征纠缠。所以 Mapping Network 的作用就是为输入向量的特征解缠提供一条学习的通路。

为何 Mapping Network 能够学习到特征解缠呢？简单来说，如果仅使用输入向量来控制视觉特征，能力是非常有限的，因此它必须遵循训练数据的概率密度。例如，如果黑头发的人的图像在数据集中更常见，那么更多的输入值将会被映射到该特征上。因此，该模型无法将部分输入（向量中的元素）映射到特征上，这就会造成特征纠缠。然而，通过使用另一个神经网络，该模型可以生成一个不必遵循训练数据分布的向量，并且可以减少特征之间的相关性。

映射网络由 8 个全连接层组成，它的输出W'与输入层（512×1）的大小相同。

1.2.2 样式模块（AdaIN）

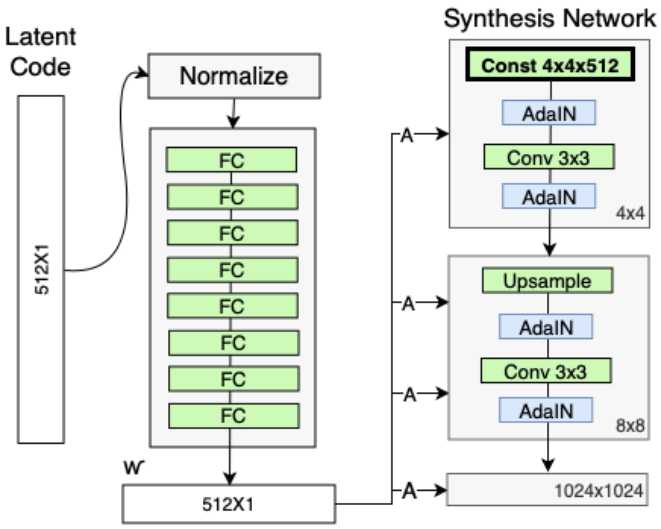


StyleGAN 的第二点改进是，将特征解缠后的中间向量W'变换为样式控制向量，从而参与影响生成器的生成过程。

生成器由于从 4*4，变换到 8*8，并最终变换到 1024*1024，所以它由 9 个生成阶段组成，而每个阶段都会受两个控制向量（A）对其施加影响，其中一个控制向量在 Upsample 之后对其影响一次，另外一个控制向量在 Convolution 之后对其影响一次，影响的方式都采用 AdaIN（自适应实例归一化）。因此，中间向量W'总共被变换成 18 个控制向量（A）传给生成器。

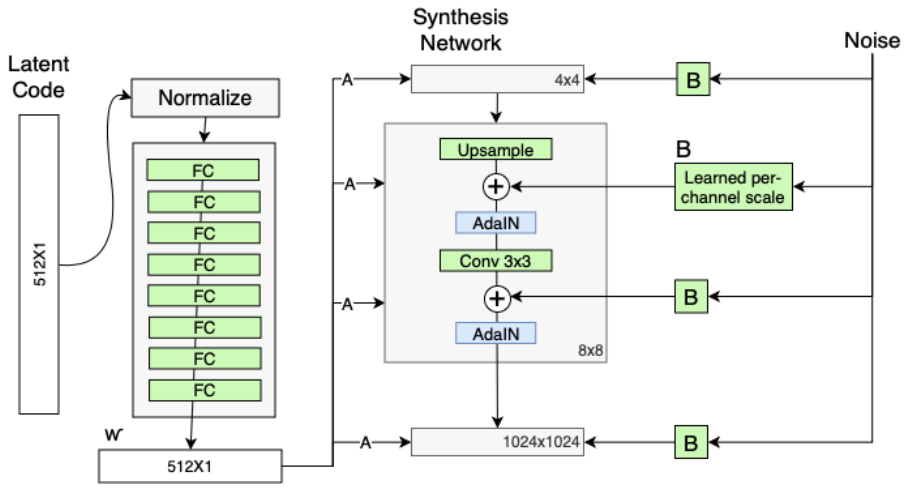
其中 AdaIN 的具体实现过程如上右图所示：将W'通过一个可学习的仿射变换（A，实际上是一个全连接层）扩变为放缩因子$\mathrm{y}_{s,i}$与偏差因子$\mathrm{y}_{b,i}$，这两个因子会与标准化之后的卷积输出做一个加权求和，就完成了一次W'影响原始输出x_i的过程。而这种影响方式能够实现样式控制，主要是因为它让W'（即变换后的$\mathrm{y}_{s,i}$与$\mathrm{y}_{b,i}$）影响图片的全局信息（注意标准化抹去了对图片局部信息的可见性），而保留生成人脸的关键信息由上采样层和卷积层来决定，因此W'只能够影响到图片的样式信息。

1.2.3 删除传统输入



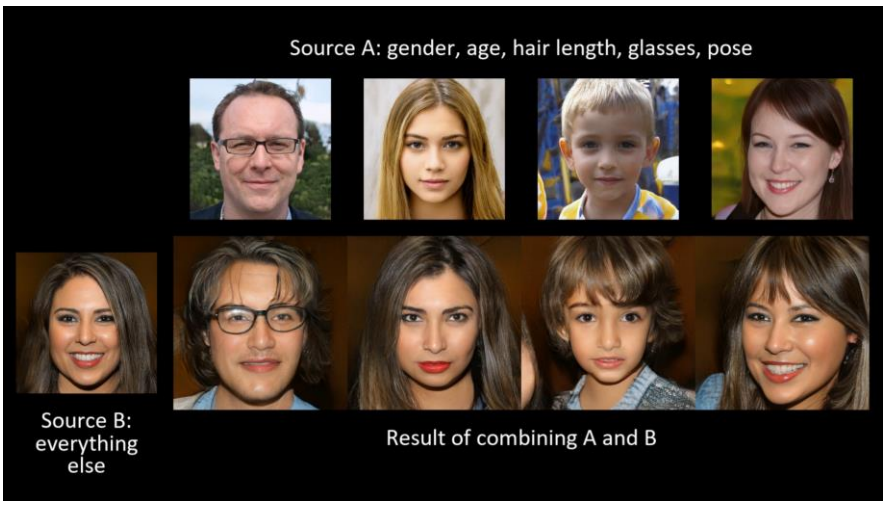
既然 StyleGAN 生成图像的特征是由W'和 AdaIN 控制的，那么生成器的初始输入可以被忽略，并用常量值替代。这样做的理由是，首先可以降低由于初始输入取值不当而生成出一些不正常的照片的概率(这在 GANs 中非常常见)，另一个好处是它有助于减少特征纠缠，对于网络在只使用W'不依赖于纠缠输入向量的情况下更容易学习。

1.2.4 随机变化



人们的脸上有许多小的特征，可以看作是随机的，例如：雀斑、发髻线的准确位置、皱纹、使图像更逼真的特征以及各种增加输出的变化。将这些小特征插入 GAN 图像的常用方法是在输入向量中添加随机噪声。为了控制噪声仅影响图片样式上细微的变化，StyleGAN 采用类似于 AdaIN 机制的方式添加噪声，即在 AdaIN 模块之前向每个通道添加一个缩放过的噪声，并稍微改变其操作的分辨率级别特征的视觉表达方式。加入噪声后的生成人脸往往更加逼真与多样。

1.2.5 样式混合



StyleGAN 生成器在合成网络的每个级别中使用了中间向量，这有可能导致网络学习到这些级别是相关的。为了降低相关性，模型随机选择两个输入向量，并为它们生成了中间向量W'。然后，它用第一个输入向量来训练一些网络级别，然后（在一个随机点中）切换到另一个输入向量来训练其余的级别。随机的切换确保了网络不会学习并依赖于一个合成网络级别之间的相关性。

虽然它并不会提高所有数据集上的模型性能，但是这个概念有一个非常有趣的副作用——它能够以一种连贯的方式来组合多个图像（视频请查看原文）。该模型生成了两个图像 A 和 B，然后通过从 A 中提取低级别的特征并从 B 中提取其余特征再组合这两个图像，这样能生成出混合了 A 和 B 的样式特征的新人脸。

1.2.6 在 W 中的截断技巧

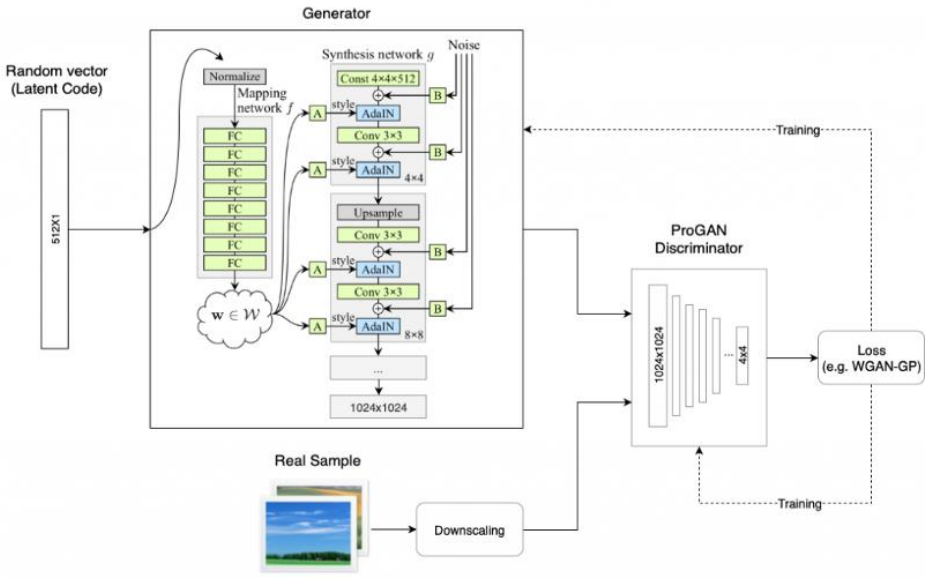
在生成模型中的一个挑战，是处理在训练数据中表现不佳的地方。这导致了生成器无法学习和创建与它们类似的图像（相反，它会创建效果不好的图像）。为了避免生成较差的图像，StyleGAN 截断了中间向量W'，迫使它保持接近“平均”的中间向量（上图左 4）。

对模型进行训练之后，通过选择多个随机的输入，用映射网络生成它们的中间向量，并计算这些向量的平均值，从而生成“中间向量”的平均值W'_{avg}。当生成新的图像时，不用直接使用映射网络的输出，而是将值W'转换为$W'_{new} = W'_{avg} + \boldsymbol{\Psi}(W' - W'_{avg})$，其中$\boldsymbol{\Psi}$的值定义了图像与“平均”图像的差异量（以及输出的多样性）。有趣的是，在仿射转换块之前，通过对每个级别使用不同的$\boldsymbol{\Psi}$，模型可以控制每个级别上的特征值与平均特征值的差异量。

1.2.7 微调超参数

StyleGAN 的另外一个改进措施是更新几个网络超参数，例如训练持续时间和损失函数，并将图片最接近尺度的缩放方式替换为双线性采样。

综上，加入了一系列附加模块后得到的 StyleGAN 最终网络模型结构图如下：



上述就是 StyleGAN 的完整模型的介绍了。不得不说，不论是在理论方法上，还是工程实践上，StyleGAN 都是一篇具有突破性的论文，它不仅可以生成高质量的和逼真的图像，而且还可以对生成的图像进行较好的控制和理解。

第二章 StyleGAN 代码解读

这一章将对 StyleGAN 的代码进行非常细致的分析和解读。一方面有助于对 StyleGAN 的架构和原理有更深的认识，另一方面是觉得 AdaIN 的思想很有价值，希望把它写代码的技巧学习下来，以后应该在 GANs 中会有很多能用得上的地方（其它 paper 里挺多出现了 AdaIN 的地方）。含有中文注释的代码可以在这里获得。

2.1 StyleGAN 代码架构总览

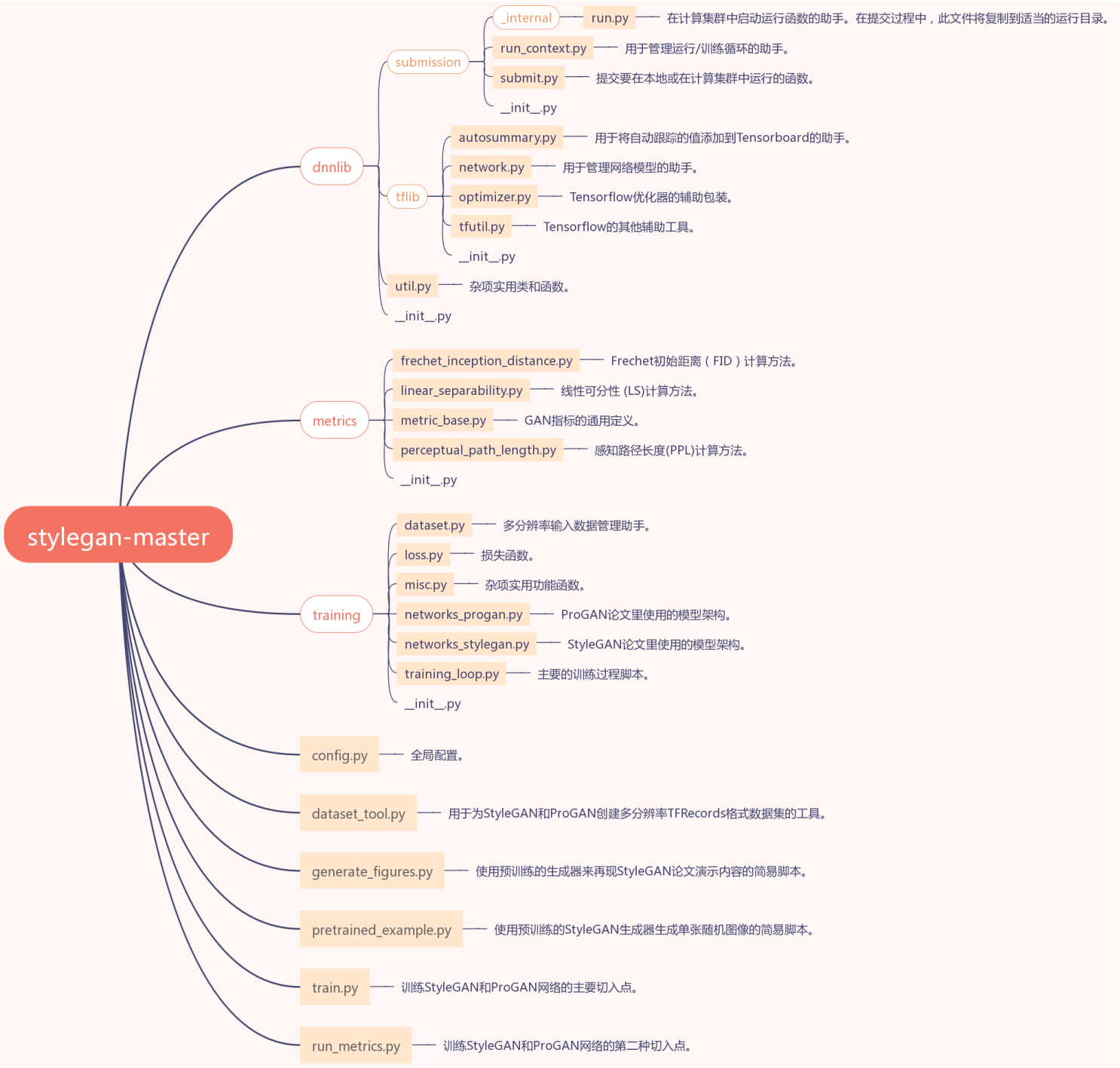


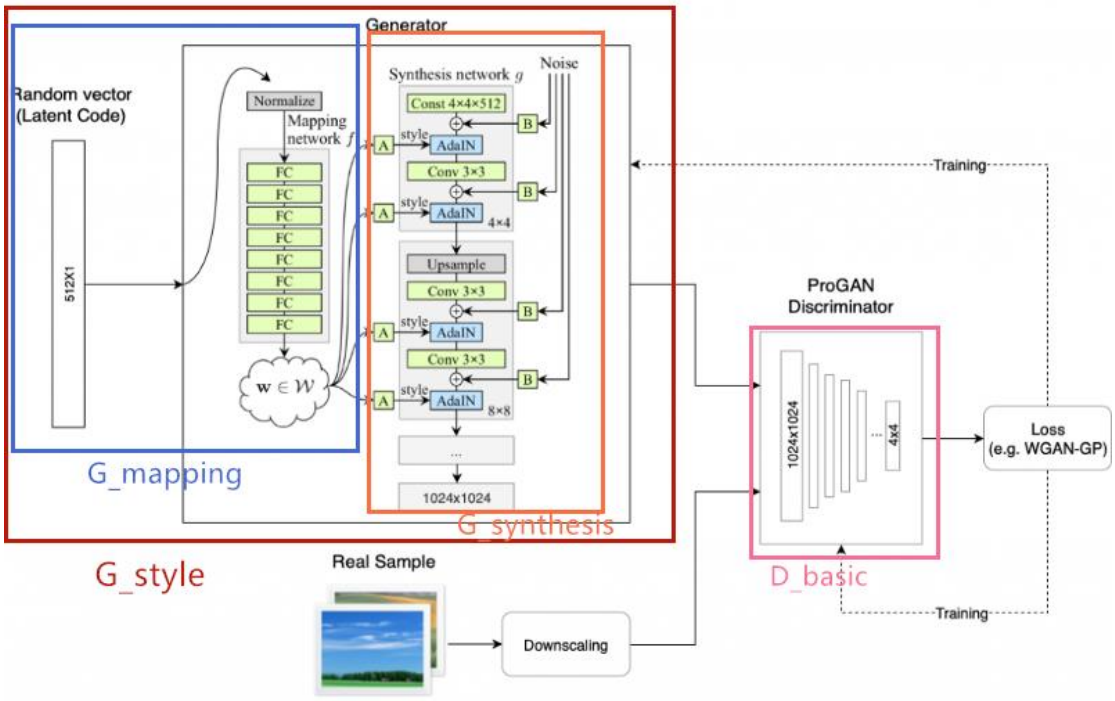
图 2.1 StyleGAN 官方代码架构

如图 2.1 所示，StyleGAN 代码的封装与解耦做的非常细致，可见作者的代码功底是非常扎实的。简单来说，在 dnnlib 文件夹下封装了日志提交工具、tensorflow 环境配置与网络处理工具以及一些杂项实用类函数，这个文件夹尽量不要去动；在 metrics 文件夹下定义了许多指标计算方法，包括 FID、LS、PPL 指标以及一些 GANs 的通用指标计算方法；而 training 文件夹是需重点关注的部分，里面包含了数据处理、模型架构、损失函数和训练过程等基于 StyleGAN 的核心内容，在接下来的笔记中也会重点对这一部分进行细致讲解；最后，在主目录下，有一些全局配置、功能展示和运行接口的代码，其中 train.py 值得细读一下，它是训练 StyleGAN 网络的主要切入点。

在接下来的笔记中，将从三个部分解读 StyleGAN 的代码，分别是：模型架构、损失函数和训练过程。至于其它部分的代码，由于我并不是特别关注，就不再赘述了。

2.2 网络架构代码解读

StyleGAN 的网络架构全都写在 training/networks_stylegan.py 下，主要包括四个组成部分（代码 302 行-659 行）：G_style()，G_mapping()，G_synthesis()和 D_basic()。



如上图所示，G_style 表示整个生成器的网络架构，它由两个子网络组成，分别是映射网络 G_mapping 和合成网络 G_synthesis；然后 D_basic 表示整个判别器的网络架构，它沿用了 ProGAN 中的模型设计。

2.2.1 G_style 网络

G_style 网络位于代码 302-379 行。在 G_style 中定义的组件包括：参数验证->设置子网络->设置变量->计算映射网络输出->更新 W 的移动平均值->执行样式混合正则化->应用截断技巧->计算合成网络输出。其中设置子网络就是调用构建 G_mapping 和 G_synthesis 的过程，两个子网络的定义将在下两节介绍。

· G_style 输入参数（line303-315）

```
def G_style(
    latents_in,                                 # 第一个输入：Z码向量 [minibatch, latent_size].
    labels_in,                                  # 第二个输入：条件标签 [minibatch, label_size].
    truncation_psi          = 0.7,              # 截断技巧的样式强度乘数。 None = disable.
    truncation_cutoff       = 8,                # 要应用截断技巧的层数。 None = disable.
    truncation_psi_val      = None,             # 验证期间要使用的truncation_psi的值。
    truncation_cutoff_val   = None,             # 验证期间要使用的truncation_cutoff的值。
    dlatent_avg_beta        = 0.995,            # 在训练期间跟踪W的移动平均值的衰减率。 None = disable.
    style_mixing_prob       = 0.9,              # 训练期间混合样式的概率。 None = disable.
    is_training             = False,            # 网络正在接受训练？ 这个选择可以启用和禁用特定特征。
    is_validation           = False,            # 网络正在验证中？ 这个选择用于确定truncation_psi的值。
    is_template_graph       = False,            # True表示由Network类构造的模板图，False表示实际评估。
    components              = dnnlib.EasyDict(),# 子网络的容器。调用时候保留。
    **kwargs):                                  # 子网络的参数们 (G_mapping 和 G_synthesis)。
```

输入参数包括 512 维的 Z 码向量和条件标签（在 FFHQ 数据集上没有使用标签），和一些可选的参数，包括截断参数、计算移动平均值 W 时的参数、样式混合参数、网络状态参数和子网络的参数们等。

· 参数验证（line318-330）

```
# 参数验证。
assert not is_training or not is_validation        # 不能同时出现训练/验证状态
assert isinstance(components, dnnlib.EasyDict)     # components作为EasyDict类，后续被用来装下synthesis和mapping两个网络
if is_validation:    # 把验证期间要使用的truncation_psi_val和truncation_cutoff_val值赋过来（默认是None），也就是验证期不使用截断
    truncation_psi = truncation_psi_val
    truncation_cutoff = truncation_cutoff_val
if is_training or (truncation_psi is not None and not tflib.is_tf_expression(truncation_psi) and truncation_psi == 1):
    truncation_psi = None  # 训练期间或截断率为1时，不使用截断
if is_training or (truncation_cutoff is not None and not tflib.is_tf_expression(truncation_cutoff) and truncation_cutoff <= 0):
    truncation_cutoff = None  # 训练期间或截断层为0时，不使用截断
if not is_training or (dlatent_avg_beta is not None and not tflib.is_tf_expression(dlatent_avg_beta) and dlatent_avg_beta == 1):
    dlatent_avg_beta = None  # 非训练期间或计算平均W时的衰减率为1时，不使用衰减
if not is_training or (style_mixing_prob is not None and not tflib.is_tf_expression(style_mixing_prob) and style_mixing_prob <= 0):
    style_mixing_prob = None  # 非训练期间或样式混合的概率小于等于0时，不使用样式混合
```

对输入参数进行验证，主要是对网络状态和其对截断率、W 平均值衰减率和样式混合概率的关系之间进行验证。

· 设置子网络（line333-338）

```
# 设置子网络。
if 'synthesis' not in components:  # 载入合成网络
    components.synthesis = tflib.Network('G_synthesis', func_name=G_synthesis, **kwargs)
num_layers = components.synthesis.input_shape[1]  # num_layers = 18
dlatent_size = components.synthesis.input_shape[2]  # dlatent_size = (18,512)
if 'mapping' not in components:  # 载入映射网络
    components.mapping = tflib.Network('G_mapping', func_name=G_mapping, dlatent_broadcast=num_layers, **kwargs)
```

直接使用 tflib.Network()类（充当参数化网络构建功能的便捷包装，提供多种实用方法并方便地访问输入/输出/权重）创建两个子网络，子网络的内容在后面的函数（func_name = G_synthesis 或 func_name = G_mapping）中被定义。

· 设置变量（line341-342）

```
# 设置变量。
lod_in = tf.get_variable('lod', initializer=np.float32(0), trainable=False)
    # 初始化为0。lod的定义式为：lod = resolution_log2 - res，其中resolution_log2（=10）表示最终分辨率级别，res表示当前层对应的分辨率级别（2-10）。
dlatent_avg = tf.get_variable('dlatent_avg', shape=[dlatent_size], initializer=tf.initializers.zeros(), trainable=False)  # 人脸平均值
```

设置两个变量 lod_in 和 dlatent_avg。前者决定当前处在第几层分辨率，即 lod=resolution_log2–res（其中 res 表示当前层对应的分辨率级别（2-10））；后者决定截断操作的基准，即生成人脸的 dlatent 码的平均值。

· 计算映射网络输出（line345）

```
# 计算映射网络输出。
dlatents = components.mapping.get_output_for(latents_in, labels_in, **kwargs)
```

得到映射网络的输出，即中间向量W'。

· 更新 W 的移动平均值（line348-353）

```
# 更新w的移动平均值。
if dlatent_avg_beta is not None:
    with tf.variable_scope('DlatentAvg'):
        batch_avg = tf.reduce_mean(dlatents[:, 0], axis=0)   # 找到新batch的dlatent平均值
        update_op = tf.assign(dlatent_avg, tflib.lerp(batch_avg, dlatent_avg, dlatent_avg_beta))
            # 把batch的dlatent平均值朝着总dlatent平均值以dlatent_avg_beta步幅靠近，作为新的人脸dlatent平均值
        with tf.control_dependencies([update_op]):
            dlatents = tf.identity(dlatents)  # 这个赋值语句似乎没有什么意义，但是可以确保update_op操作完成？
```

把 batch 的 dlatent 平均值朝着总 dlatent 平均值以 dlatent_avg_beta 步幅靠近，作为新的人脸 dlatent 平均值，即 dlatent_avg。

· 执行样式混合正则化（line356-366）

```
# 执行样式混合正则化。
if style_mixing_prob is not None:
    with tf.name_scope('StyleMix'):
        latents2 = tf.random_normal(tf.shape(latents_in))
        dlatents2 = components.mapping.get_output_for(latents2, labels_in, **kwargs)  # 用来做样式混合的随机中间向量
        layer_idx = np.arange(num_layers)[np.newaxis, :, np.newaxis]   # 层的索引：[[[0],[1],[2],[3],[4]...[17]]]
        cur_layers = num_layers - tf.cast(lod_in, tf.int32) * 2  # 当前层等于总层数减去lod_in的两倍，因为每个分辨率对应两层
        mixing_cutoff = tf.cond(
            tf.random_uniform([], 0.0, 1.0) < style_mixing_prob,   # 如果随机值小于样式混合的概率，则从1到当前层随机选一个层，否则保留原层
            lambda: tf.random_uniform([], 1, cur_layers, dtype=tf.int32),
            lambda: cur_layers)
        dlatents = tf.where(tf.broadcast_to(layer_idx < mixing_cutoff, tf.shape(dlatents)), dlatents, dlatents2)
                # 对于1到mixing_cutoff层保留dlatents的值，其余层采用dlatents2的值替换
```

样式混合正则化其实很好理解，就是随机创建一个新的潜码，这个潜码以一定概率与原始潜码交换某一部分，对于交换后的混合潜码，其生成的图片也要能够逼真，这就是样式混合正则化的实现。

· 应用截断技巧（line369-374）

```
# 应用截断技巧。
if truncation_psi is not None and truncation_cutoff is not None:
    with tf.variable_scope('Truncation'):
        layer_idx = np.arange(num_layers)[np.newaxis, :, np.newaxis]  # 层的索引：[[[0],[1],[2],[3],[4]...[17]]]
        ones = np.ones(layer_idx.shape, dtype=np.float32)  # ones:[[[1],[1],[1],[1],[1]...[1]]]
        coefs = tf.where(layer_idx < truncation_cutoff, truncation_psi * ones, ones)
                # 截断的步幅，需要截断的层步幅为truncation_psi，否则为1
        dlatents = tflib.lerp(dlatent_avg, dlatents, coefs)
                # 截断，用平均脸dlatent_avg朝着当前脸dlatents以coefs步幅靠近，最后得到的结果取代当前脸dlatents
```

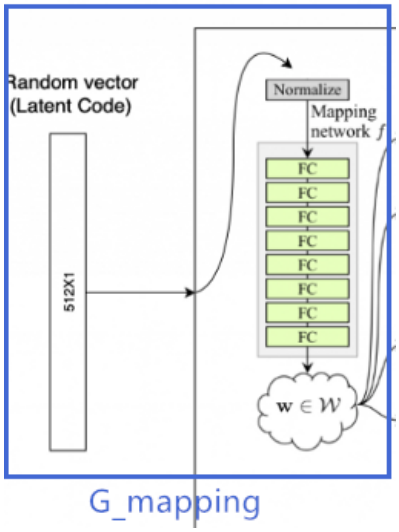
截断是指，用平均脸 dlatent_avg 朝着当前脸 dlatents 以 coefs 步幅靠近，得到的结果就是截断的 dlatents。

· 计算合成网络输出（line377-379）

```
# 计算合成网络输出。
with tf.control_dependencies([tf.assign(components.synthesis.find_var('lod'), lod_in)]):
    images_out = components.synthesis.get_output_for(dlatents, force_clean_graph=is_template_graph, **kwargs)
return tf.identity(images_out, name='images_out')  # 返回生成的图片
```

将截断的 dlatents 传给 G_synthesis 网络进行合成，得到的结果就是整个生成网络 G_style 的输出结果。

2.2.2 G_mapping 网络



G_mapping 网络位于代码 384-435 行。如上图所示，G_mapping 网络实现了从初始生成码到中间向量的映射过程。在 G_mapping 中定义的组件包括：输入->连接标签->归一化潜码->映射层->广播->输出。其中映射层由 8 个全连接层组成。

· G_mapping 输入参数（line385-398）

```
latents_in,                         # 第一个输入：Z码向量 [minibatch, latent_size].
labels_in,                          # 第二个输入：条件标签 [minibatch, label_size].
latent_size             = 512,      # 潜在向量（Z）维度。
label_size              = 0,        # 标签尺寸，0表示没有标签。
dlatent_size            = 512,      # 解缠后的中间向量（W）维度。
dlatent_broadcast       = None,     # 将解缠后的中间向量（W）输出为[minibatch, dlatent_size]或[minibatch, dlatent_broadcast, dlatent_size]格式。
mapping_layers          = 8,        # 映射网络的层数。
mapping_fmaps           = 512,      # 映射层中的特征图维度。
mapping_lrmul           = 0.01,     # 映射层的学习率变化率。
mapping_nonlinearity    = 'lrelu',  # 激活函数：'relu'，'lrelu'。
use_wscale              = True,     # 启用均等的学习率？
normalize_latents       = True,     # 在将潜在向量（Z）馈送到映射层之前对其进行归一化？
dtype                   = 'float32',# 用于激活和输出的数据类型。
**_kwargs):                         # 忽略无法识别的关键字参数。
```

输入参数包括 512 维的 Z 码向量和条件标签（在 FFHQ 数据集上没有使用标签），和一些可选的参数，包括初始向量 Z 参数、中间向量 W 参数、映射层设置、激活函数设置、学习率设置以及归一化设置等。

· 网络输入（line403-407）

```
# 输入
latents_in.set_shape([None, latent_size])
labels_in.set_shape([None, label_size])
latents_in = tf.cast(latents_in, dtype)
labels_in = tf.cast(labels_in, dtype)
x = latents_in
```

处理好 latent 的大小和格式后，其值赋给 x，即用 x 标识网络的输入。

· 连接标签（line410-414）

```
# 嵌入标签并将其与潜码连接起来。
if label_size:  # 原始StyleGAN是无条件训练集，这部分不用考虑
    with tf.variable_scope('LabelConcat'):
        w = tf.get_variable('weight', shape=[label_size, latent_size], initializer=tf.initializers.random_normal())
        y = tf.matmul(labels_in, tf.cast(w, dtype))
        x = tf.concat([x, y], axis=1)
```

原始 StyleGAN 是无标签训练集，这部分不会被调用。

· 归一化潜码（line417-418）

```
# 归一化潜码。
if normalize_latents:
    x = pixel_norm(x)
```

pixel_norm()（line239-242）

```
# 逐像素特征向量归一化。
def pixel_norm(x, epsilon=1e-8):
    with tf.variable_scope('PixelNorm'):
        epsilon = tf.constant(epsilon, dtype=x.dtype, name='epsilon')
        return x * tf.rsqrt(tf.reduce_mean(tf.square(x), axis=1, keepdims=True) + epsilon)  # x = x/√(x2_avg+ε)
```

逐像素归一化的实现方式为：$x = x/\sqrt{\frac{1}{N}\sum_{i=0}^{N-1} x_i^2 + \epsilon}$，其中$\epsilon = 10^{-8}$。

为何要使用 pixel_norm 呢? Pixel norm，它是 local response normalization 的变种，具有避免生成器梯度爆炸的作用。Pixel norm 沿着 channel 维度做归一化（axis=1），这样归一化的一个好处在于，feature map 的每个位置都具有单位长度。这个归一化策略与作者设计的 Generator 输出有较大关系，注意到 Generator 的输出层并没有 Tanh 或者 Sigmoid 激活函数。

· 映射层（line421-426）

```
# 映射层。
for layer_idx in range(mapping_layers):
    with tf.variable_scope('Dense%d' % layer_idx):
        fmaps = dlatent_size if layer_idx == mapping_layers - 1 else mapping_fmaps
            # 除去最后一层输出维度是中间向量的维度dlatent_size，其他层输出都是映射特征图的维度mapping_fmaps
        x = dense(x, fmaps=fmaps, gain=gain, use_wscale=use_wscale, lrmul=mapping_lrmul)  # 全连接层
        x = apply_bias(x, lrmul=mapping_lrmul)  # 添加偏置
        x = act(x)  # 添加激活函数
```

构建了 mapping_layers 层映射层，每个映射层有三个组成部分：全连接层 dense()、偏置函数 apply_bias()和激活函数 act()。

1）全连接层 dense()（line 154-159）

```
# 全连接层

def dense(x, fmaps, **kwargs):
    if len(x.shape) > 2:
        x = tf.reshape(x, [-1, np.prod([d.value for d in x.shape[1:]])])  # 如果x的维度超过2，把高于2的维度的值连乘起来作为第二维度的值？？
    w = get_weight([x.shape[1].value, fmaps], **kwargs)  # 创建一个[512,512]的全连接层
    w = tf.cast(w, x.dtype)
    return tf.matmul(x, w)
```

dense()函数中首先将输出全部展平以计算输出维度，然后调用 get_weight()创建全连接层 w，最后返回 x 与 w 的矩阵相乘的结果，作为 dense()层的输出。

get_weight()（line 135-149）

```
def get_weight(shape, gain=np.sqrt(2), use_wscale=False, lrmul=1):
    fan_in = np.prod(shape[:-1])  # fan_in表示某卷积核参数个数(h*w*fmaps_in)或dense层输入节点数目fmaps_in
    he_std = gain / np.sqrt(fan_in)  # He公式为：0.5*n*var(w)=1 , so: std(w)=sqrt(2)/sqrt(n)=gain/sqrt(fan_in)

    # 均衡学习率以及自定义学习率变化率。
    if use_wscale:
        init_std = 1.0 / lrmul
        runtime_coef = he_std * lrmul
    else:
        init_std = he_std / lrmul  # 获得scale后的归一化值
        runtime_coef = lrmul  # 网络也实时scale

    # 创建变量。
    init = tf.initializers.random_normal(0, init_std)  # He的初始化方法能够确保网络初始化的时候，随机初始化的参数不会大幅度地改变输入信号的强度。
    return tf.get_variable('weight', shape=shape, initializer=init) * runtime_coef  # StyleGAN中不仅限初始状态scale而且是实时scale
```

get_weight()函数是用来创建卷积层或完全连接层，且获取权重张量的函数。

值得注意的是，get_weight()采用了 He 的初始化方法。He 的初始化方法能够确保网络初始化的时候，随机初始化的参数不会大幅度地改变输入信号的强度。StyleGAN 中不仅限初始状态 scale 而且也是实时 scale。

2）添加偏置 apply_bias()（line 213-218）

```
# 对给定的激活张量施加偏差。

def apply_bias(x, lrmul=1):
    b = tf.get_variable('bias', shape=[x.shape[1]], initializer=tf.initializers.zeros()) * lrmul
    b = tf.cast(b, x.dtype)
    if len(x.shape) == 2:
        return x + b
    return x + tf.reshape(b, [1, -1, 1, 1])   # 偏差应该是只对于第二维有效？？（Dense层中已把高于2的维度的值连乘起来作为第二维度的值）
```

对给定的激活张量施加偏差。

3）激活函数 act()（line 400）

```
act, gain = {'relu': (tf.nn.relu, np.sqrt(2)), 'lrelu': (leaky_relu, np.sqrt(2))}[mapping_nonlinearity]
```

激活函数采用 mapping_nonlinearity 的值，StyleGAN 中选用'lrelu'，且增益值为$\sqrt{2}$。

注意这儿的 gain 是一个增益值，增益值是指的非线性函数稳态时输入幅度与输出幅度的比值，通常被用来乘在激活函数之后使激活函数更加稳定。常见的增益值包括：Sigmoid 推荐的增益值是 1，Relu 推荐的增益值是$\sqrt{2}$，LeakyRelu 推荐的增益值是$\sqrt{\frac{2}{1+negative_slope^2}}$。

· 广播（line429-431）

```
# 广播。
if dlatent_broadcast is not None:
    with tf.variable_scope('Broadcast'):   # 这儿只定义了简单的复制扩充，广播前x的维度是(?,512),广播后x的维度是(?,18,512)
        x = tf.tile(x[:, np.newaxis], [1, dlatent_broadcast, 1])
```

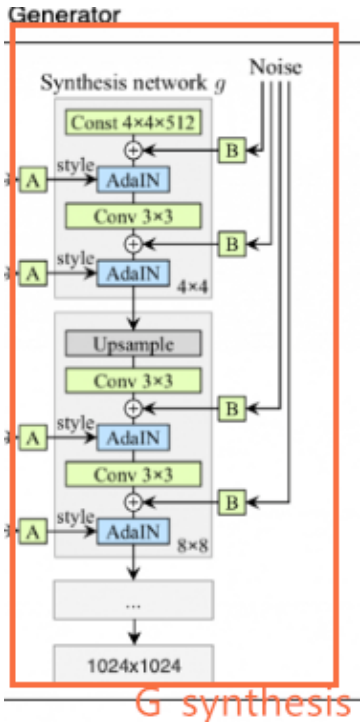
这儿只定义了简单的复制扩充，广播前 x 的维度是(?,512)，广播后 x 的维度是(?,18,512)。

· 输出（line 434-435）

```
# 输出。
assert x.dtype == tf.as_dtype(dtype)
return tf.identity(x, name='dlatents_out')
```

广播后的中间向量，就是 G_mapping 网络的最终输出。

2.2.3 G_synthesis 网络



G_synthesis 网络位于代码 441-560 行。如上图所示，G_synthesis 网络实现了从广播得到的中间向量到生成图片的合成过程。在 G_synthesis 中定义的组件包括：预处理->主要输入->噪音输入->★每层层末的调制功能->早期层(4*4)结构->剩余层的 block 块->★网络增长变换过程->输出。其中，每层层末的调制功能是指的在卷积之后，融入噪音与样式控制的过程（上图的⊕与 AdaIN 过程）；网络增长变换过程是指的在训练时期，合成网络的架构动态增长，以生成更高分辨率图片的过程。上述两个内容是重点值得学习的部分。

· **G_synthesis 输入参数（line441-462）**

```
dlatents_in,                        # 输入：解缠的中间向量 (W) [minibatch, num_layers, dlatent_size].
dlatent_size        = 512,          # 解缠的中间向量 (W) 的维度。
num_channels        = 3,            # 输出颜色通道数。
resolution          = 1024,         # 输出分辨率。
fmap_base           = 8192,         # 特征图的总数目，这儿取8192因为512*(18-2)=8192。
fmap_decay          = 1.0,          # 当分辨率翻倍时以log2降低特征图，这儿指示降低的速率。
fmap_max            = 512,          # 在任何层中特征图的最大数量。
use_styles          = True,         # 启用样式输入？
const_input_layer   = True,         # 第一层是常数？
use_noise           = True,         # 启用噪音输入？
randomize_noise     = True,         # True表示每次都随机化噪声输入（不确定），False表示从变量中读取噪声输入。
nonlinearity        = 'lrelu',      # 激活函数: 'relu', 'lrelu'
use_wscale          = True,         # 启用均等的学习率？
use_pixel_norm      = False,        # 启用逐像素特征向量归一化？
use_instance_norm   = True,         # 启用实例规一化？
dtype               = 'float32',    # 用于激活和输出的数据类型。
fused_scale         = 'auto',       # True = 融合卷积+缩放，False = 单独操作，'auto'= 自动决定。
blur_filter         = [1,2,1],      # 重采样激活时应用的低通卷积核（Low-pass filter）。None表示不过滤。
structure           = 'auto',       # 'fixed' = 无渐进式增长，'linear' = 人类可读，'recursive' = 有效，'auto' = 自动选择。
is_template_graph   = False,        # True表示由Network类构造的模板图，False表示实际评估。
force_clean_graph   = False,        # True表示构建一个在TensorBoard中看起来很漂亮的干净图形，False表示默认设置。
**kwargs):                          # 忽略无法识别的关键字参数。
```

输入参数包括 512 维的中间向量(W)和输出图片的分辨率及通道，和一些可选的参数，包括各层特征图的设置、样式/网络起始值/噪音设置、激活函数设置、数据处理设置以及网络增长架构的设置等。

· **预处理（line464-474）**

```
resolution_log2 = int(np.log2(resolution))  # 计算分辨率是2的多少次方
assert resolution == 2**resolution_log2 and resolution >= 4   # 分辨率需要大于等于32，因为训练从学习生成32*32的图片开始
def nf(stage): return min(int(fmap_base / (2.0 ** (stage * fmap_decay))), fmap_max)
    # nf()返回在第stage层中特征图的数量——当stage<=4时，特征图数量为512；当stage>4时，每多一层特征图数量就减半。
def blur(x): return blur2d(x, blur_filter) if blur_filter else x  # 对图片进行滤波模糊操作，有利于降噪
if is_template_graph: force_clean_graph = True
if force_clean_graph: randomize_noise = False
if structure == 'auto': structure = 'linear' if force_clean_graph else 'recursive'  # 依据force_clean_graph选择架构为'linear'或'recursive'
act, gain = {'relu': (tf.nn.relu, np.sqrt(2)), 'lrelu': (leaky_relu, np.sqrt(2))}[nonlinearity]  # 激活函数
num_layers = resolution_log2 * 2 - 2  # 因为每个分辨率有两层，所以层数为：分辨率级别(10)*2-2=18
num_styles = num_layers if use_styles else 1  # 样式层数
images_out = None
```

预处理部分除了进一步细化网络配置以外，还定义了两个函数——nf()返回在第 stage 层中特征图的数量；blur()对图片进行滤波模糊操作，有利于降噪，其中 blur 的函数实现方式为 blur2d()。

blur2d ()（line 96-106）

```
def blur2d(x, f=[1,2,1], normalize=True):
    with tf.variable_scope('Blur2D'):
        @tf.custom_gradient
        def func(x):  # 定义一个函数，返回模糊后的特征值以及该函数的梯度计算式grad()
            y = _blur2d(x, f, normalize)  # 模糊后的特征值y
            @tf.custom_gradient
            def grad(dy):  # func函数的梯度计算式
                dx = _blur2d(dy, f, normalize, flip=True)  # d_func(x)/d_x=_blur2d(d_x)
                return dx, lambda ddx: _blur2d(ddx, f, normalize)
                # func的梯度(即grad)用_blur2d()去近似，grad的梯度也用_blur2d()去近似(对于一阶导和二阶导都用_blur2d()作近似，可能是因为_blur2d()函数的结果(模糊)等效为目的(变模糊))。
            return y, grad
        return func(x)
```

在 blur2d()里定义了模糊的返回函数为_blur2d()，同时 blur2d()的一阶导和二阶导也被定义了出来，都是直接使用_blur2d()函数作为近似。

_blur2d ()（line 22-49）

```
def _blur2d(x, f=[1,2,1], normalize=True, flip=False, stride=1):
    assert x.shape.ndims == 4 and all(dim.value is not None for dim in x.shape[1:])
    assert isinstance(stride, int) and stride >= 1

    # 处理滤波器核f。
    f = np.array(f, dtype=np.float32)
    if f.ndim == 1:
        f = f[:, np.newaxis] * f[np.newaxis, :]
    assert f.ndim == 2
    if normalize:
        f /= np.sum(f)
    if flip:
        f = f[::-1, ::-1]
    f = f[:, :, np.newaxis, np.newaxis]
    f = np.tile(f, [1, 1, int(x.shape[1]), 1])

    # 无操作=>提前退出。
    if f.shape == (1, 1) and f[0, 0] == 1:
        return x

    # 使用depthwise_conv2d卷积。
    orig_dtype = x.dtype
    x = tf.cast(x, tf.float32)  # tf.nn.depthwise_conv2d()不支持fp16
    f = tf.constant(f, dtype=x.dtype, name='filter')
    strides = [1, 1, stride, stride]
    x = tf.nn.depthwise_conv2d(x, f, strides=strides, padding='SAME', data_format='NCHW')
        # 这个depthwise_conv2d卷积挺有意思。普通的卷积，我们对卷积核每一个out_channel的两个通道分别和输入的两个通道做卷积相加，
        # 得到feature map的一个channel，而depthwise_conv2d卷积，我们对每一个对应的in_channel，分别卷积生成两个out_channel，
        # 所以获得的feature map的通道数量可以用in_channel* channel_multiplier来表达，这个channel_multiplier，就可以理解为卷积核的第四维。
        # 参见博客：https://blog.csdn.net/mao_xiao_feng/article/details/78003476。
    x = tf.cast(x, orig_dtype)
    return x
```

_blur2d ()的实现主要有两个部分，第一个部分是对于卷积核的处理，包括维度的规范和归一化处理；第二个部分是模糊的实现，即用卷积核对 x 实行 depthwise_conv2d 卷积。

*注意：depthwise_conv2d 与普通的卷积有些不同。普通的卷积对卷积核每一个 out_channel 的两个通道分别和输入的两个通道做卷积相加，得到 feature map 的一个 channel，而 depthwise_conv2d 卷积对每一个对应的 in_channel，分别卷积生成两个 out_channel，所以获得的 feature map 的通道数量可以用 in_channel * channel_multiplier 来表达，这个 channel_multiplier，就可以理解为卷积核的第四维。参见博客：https://blog.csdn.net/mao_xiao_feng/article/details/78003476。*

· 主要输入（line 477-479）

```
# 主要输入。
dlatents_in.set_shape([None, num_styles, dlatent_size])  # dlatents_in是通过广播得到的中间向量，维度是(?,18,512)
dlatents_in = tf.cast(dlatents_in, dtype)
lod_in = tf.cast(tf.get_variable('lod', initializer=np.float32(0), trainable=False), dtype)
    # lod_in是一个指定当前输入分辨率级别的参数，规定lod = resolution_log2 - res
```

主要输入除了 dlatents_in 之外，还有一个 lod_in 参数。lod_in 是一个指定当前输入分辨率级别的参数，规定 lod = resolution_log2 – res。lod_in 在递归构建模型架构的部分中被使用。

· 创建噪音（line 482-487）

```
# 创建噪音。
noise_inputs = []
if use_noise:
    for layer_idx in range(num_layers):
        res = layer_idx // 2 + 2  # [2,2,3,3,…,10,10]
        shape = [1, use_noise, 2**res, 2**res]  # 不同层的噪音shape从[1,1,4,4]一直到[1,1,1024,1024]
        noise_inputs.append(tf.get_variable('noise%d' % layer_idx, shape=shape, initializer=tf.initializers.random_normal(), trainable=False))  # 随机初始化噪音
```

最初创建噪音时，只是依据对应层的分辨率创建对应的 shape，然后随机初始化即为一个噪音。

· ★层末调制（含 AdaIN，line490-501）

```
# ★每一层最后需要做的事情。
def layer_epilogue(x, layer_idx):
    if use_noise:
        x = apply_noise(x, noise_inputs[layer_idx], randomize_noise=randomize_noise)  # 应用噪音
    x = apply_bias(x)  # 应用偏置
    x = act(x)  # 应用激活函数
    if use_pixel_norm:
        x = pixel_norm(x)  # 逐像素归一化
    if use_instance_norm:
        x = instance_norm(x)  # 实例归一化
    if use_styles:
        x = style_mod(x, dlatents_in[:, layer_idx], use_wscale=use_wscale)  # 样式调制，AdaIN
    return x
```

层末调制，是在每个 block 的卷积之后对特征的处理，包含 6 种（可选）内容：应用噪音 apply_noise()、应用偏置 apply_bias()、应用激活函数 act()、逐像素归一化 pixel_norm()、实例归一化 instance_norm()和样式调制（AdaIN）style_mod()。其中 apply_bias()、act()与 pixel_norm()在前文中已提及过，下面将不再赘述。

1）apply_noise()（line 270-278）

```
def apply_noise(x, noise_var=None, randomize_noise=True):
    assert len(x.shape) == 4  # NCHW
    with tf.variable_scope('Noise'):
        if noise_var is None or randomize_noise:  # 添加随机噪音
            noise = tf.random_normal([tf.shape(x)[0], 1, x.shape[2], x.shape[3]], dtype=x.dtype)
        else:  # 添加指定噪音
            noise = tf.cast(noise_var, x.dtype)
        weight = tf.get_variable('weight', shape=[x.shape[1].value], initializer=tf.initializers.zeros())  # 噪音的权重
        return x + noise * tf.reshape(tf.cast(weight, x.dtype), [1, -1, 1, 1])
```

应用噪音，直接将噪音加在特征 x 上就行了，注意按 channel 叠加。

2）instance_norm()（line 247-256）

```
# 实例规一化。
def instance_norm(x, epsilon=1e-8):
    assert len(x.shape) == 4  # NCHW
    with tf.variable_scope('InstanceNorm'):
        orig_dtype = x.dtype
        x = tf.cast(x, tf.float32)
        x -= tf.reduce_mean(x, axis=[2,3], keepdims=True)  # 实例归一化仅对HW做归一化，所以axis是2,3
        epsilon = tf.constant(epsilon, dtype=x.dtype, name='epsilon')
        x *= tf.rsqrt(tf.reduce_mean(tf.square(x), axis=[2,3], keepdims=True) + epsilon)
        x = tf.cast(x, orig_dtype)  # x = (x-x_mean)/√(x2_avg+ε)
        return x
```

实例归一化是一个在生成模型中应用非常广泛的归一化方式，它的主要特点是仅对特征的 HW（高和宽）维度做归一化，对图像的风格影响明显。

3）★style_mod ()（line 261-265）

```
def style_mod(x, dlatent, **kwargs):
    with tf.variable_scope('StyleMod'):
        style = apply_bias(dense(dlatent, fmaps=x.shape[1]*2, gain=1, **kwargs))  # 仿射变换A（通过全连接层将dlatent扩大一倍）
        style = tf.reshape(style, [-1, 2, x.shape[1]] + [1] * (len(x.shape) - 2))  # 扩大后的dlatent转换为放缩因子y_s,i和偏置因子y_b,i
        return x * (style[:,0] + 1) + style[:,1]  # 对卷积后的x执行样式调制（自适应实例归一化）
```

样式控制（AdaIN）的代码只有 3 行。第 1 行是仿射变化 A，它通过全连接层将 dlatent 扩大一倍；第 2 行将扩大后的 dlatent 转换为放缩因子$y_{s,i}$和偏置因子$y_{b,i}$；第 3 行是将这两个因子对卷积后的 x 实施自适应实例归一化。

· 早期层结构（line 504-514）

```
# 早期的层。
with tf.variable_scope('4x4'):
    if const_input_layer:  # 合成网络的起点是否为固定常数，StyleGAN中选用固定常数。
        with tf.variable_scope('Const'):
            x = tf.get_variable('const', shape=[1, nf(1), 4, 4], initializer=tf.initializers.ones())  # 初始为常数变量，shape为(1,512,4,4)
            x = layer_epilogue(tf.tile(tf.cast(x, dtype), [tf.shape(dlatents_in)[0], 1, 1, 1]), 0)  # 第0层的层末调制
    else:
        with tf.variable_scope('Dense'):
            x = dense(dlatents_in[:, 0], fmaps=nf(1)*16, gain=gain/4, use_wscale=use_wscale)  # 调整增益值以匹配ProGAN的官方实现（ProGAN的初始起点不是常数，而就是latent）
            x = layer_epilogue(tf.reshape(x, [-1, nf(1), 4, 4]), 0)
    with tf.variable_scope('Conv'):
        x = layer_epilogue(conv2d(x, fmaps=nf(1), kernel=3, gain=gain, use_wscale=use_wscale), 1)  # 第1层为卷积层，添加层末调制
```

由于 StyleGAN 的网络结构随训练进行是动态变化的，所以代码中定义了训练最开始时的网络结构，即 4*4 分辨率的生成网络。StyleGAN 的生成起点选用维度为(1,512,4,4)的常量，通过一个卷积层（conv2d）得到了通道数为 nf(1)（即 512 维）的特征图。

· conv2d（line 164-168）

```
# 卷积层
def conv2d(x, fmaps, kernel, **kwargs):
    assert kernel >= 1 and kernel % 2 == 1
    w = get_weight([kernel, kernel, x.shape[1].value, fmaps], **kwargs)
    w = tf.cast(w, x.dtype)
    return tf.nn.conv2d(x, w, strides=[1,1,1,1], padding='SAME', data_format='NCHW')  # 简单的全卷积
```

conv2d 通过简单的卷积实现，将 x 的通道数由 x.shape[1]变为 fmaps，而 x 的大小不变。

· 剩余层的 block 块（line 517-527）

```
# 为剩余层构建block块。
def block(res, x):  # res从3增加到resolution_log2；这些层被写在函数里方便网络需要时再创建。
    with tf.variable_scope('%dx%d' % (2**res, 2**res)):
        with tf.variable_scope('Conv0_up'):  # 第2,4,6…,16层为上采样层；上采样之后会加一个模糊滤波以降噪。
            x = layer_epilogue(blur(upscale2d_conv2d(x, fmaps=nf(res-1), kernel=3, gain=gain, use_wscale=use_wscale, fused_scale=fused_scale)), res*2-4)
        with tf.variable_scope('Conv1'):  # 第3,5,7…,17层为卷积层
            x = layer_epilogue(conv2d(x, fmaps=nf(res-1), kernel=3, gain=gain, use_wscale=use_wscale), res*2-3)
        return x
def torgb(res, x): # res从2增加到resolution_log2；这个函数实现特征图到RGB图像的转换。
    lod = resolution_log2 - res
    with tf.variable_scope('ToRGB_lod%d' % lod):
        return apply_bias(conv2d(x, fmaps=num_channels, kernel=1, gain=1, use_wscale=use_wscale))  # ToRGB是通过一个简单卷积实现的
```

StyleGAN 将剩余分辨率的网络层封装成了 block 函数，方便在训练过程中依据输入的 res（由 lod_in 计算出来）实时构建及调整网络的架构。其中每个 block 都包括了一个上采样层（upscaled_conv2d）和一个卷积层，上采样层后置滤波处理与层末调制，卷积层后置层末调制。另外在训练过程中网络需实时输出图片，StyleGAN 中定义了 torgb()函数，负责将对应分辨率的特征图转换为 RGB 图像。

· upscaled_conv2d()（line 174-191）

```
def upscale2d_conv2d(x, fmaps, kernel, fused_scale='auto', **kwargs):
    assert kernel >= 1 and kernel % 2 == 1
    assert fused_scale in [True, False, 'auto']
    if fused_scale == 'auto':
        fused_scale = min(x.shape[2:]) * 2 >= 128  # x的高和宽≥64的话使用融合，否则不使用融合

    # 不融合=>直接调用各个操作。
    if not fused_scale:
        return conv2d(upscale2d(x), fmaps, kernel, **kwargs)

    # 融合=>使用tf.nn.conv2d_transpose()同时执行两个操作。
    w = get_weight([kernel, kernel, x.shape[1].value, fmaps], **kwargs)  # 创建一个卷积层w，shape为[kernel, kernel, fmaps_in, fmaps_out]
    w = tf.transpose(w, [0, 1, 3, 2])  # 将w转成[kernel, kernel, fmaps_out, fmaps_in]，后面tf函数中卷积核输出通道在前（tf坑太多。。）
    w = tf.pad(w, [[1,1], [1,1], [0,0], [0,0]], mode='CONSTANT')  # 对w进行填充，在两个kernel的第0维和最后一维分别填充0，w的shape变为[kernel+2, kernel+2, fmaps_in, fmaps_out]
    w = tf.add_n([w[1:, 1:], w[:-1, 1:], w[1:, :-1], w[:-1, :-1]])  # 填充区域求和两次，非填充区域求和四次，w的shape不变。为什么要对卷积核做这样的处理呢？？
    w = tf.cast(w, x.dtype)
    os = [tf.shape(x)[0], fmaps, x.shape[2] * 2, x.shape[3] * 2]  # 即output_shape，定义反卷积需输出的维度（高宽扩大两倍）
    return tf.nn.conv2d_transpose(x, w, os, strides=[1,1,2,2], padding='SAME', data_format='NCHW')
        # 最终的反卷积输出，可以验算一下：特征x的维度是[Num,fmaps_in,H,W],卷积核w的维度是[kernel+2, kernel+2, fmaps_out, fmaps_in],strides为[1,1,2,2],
        # 这三者做反卷积的输出维度就是[Num,fmaps_out,Hx2,Wx2]，刚好就是os的shape。
```

upscaled_conv2d 利用 tf.nn.conv2d_transpose 反卷积操作实现将特征图放大一倍。其中一个值得注意的操作是，其卷积核被轻微平移了四次并对自身做了叠加，这样或许对于提取特征有帮助，但我查阅不到相关资料证明这一点。

· **★网络增长变换过程（line 530-556）**

StyleGAN 的生成网络需具备动态变换的能力，代码中定义了三种结构组合方式，分别是：固定结构、线性结构与递归结构。

1）固定结构（line 530-533）

```
# 固定结构：简单高效，但不支持渐进式增长。
if structure == 'fixed':
    for res in range(3, resolution_log2 + 1):  # res从3增加到resolution_log2
        x = block(res, x)  # 相当于直接构建了一个1024*1024分辨率的生成器网络
    images_out = torgb(resolution_log2, x)
```

固定结构构建了直达 1024*1024 分辨率的生成器网络，简单高效但不支持渐进式增长。

2）线性结构（line 536-544）

```
# ★线性结构：简单但效率低下。
if structure == 'linear':
    images_out = torgb(2, x)
    for res in range(3, resolution_log2 + 1):  # res从3增加到resolution_log2
        lod = resolution_log2 - res
        x = block(res, x)
        img = torgb(res, x)
        images_out = upscale2d(images_out)  # 通过upscale2d()构建上采样层，将当前分辨率放大一倍
        with tf.variable_scope('Grow_lod%d' % lod):
            images_out = tflib.lerp_clip(img, images_out, lod_in - lod)  # 依靠含大小值裁剪的线性插值实现图片放大，相当于在过渡阶段实现平滑过渡
```

线性结构构建了 upscale2d()的上采样层，能将当前分辨率放大一倍。另外在不同分辨率间变换时，线性结构采用了含大小值裁剪的线性插值，实现了不同分辨率下的平滑过渡。

upscale2d()（line 108-118）

```
def upscale2d(x, factor=2):
    with tf.variable_scope('Upscale2D'):
        @tf.custom_gradient
        def func(x):  # 定义一个函数，返回上采样后的特征值以及该函数的梯度计算式grad()
            y = _upscale2d(x, factor)
            @tf.custom_gradient
            def grad(dy):  # func函数的梯度计算式与blur2d()写法类似，对于此种函数具有通用性
                dx = _downscale2d(dy, factor, gain=factor**2)  # 上采样的增益值取放大倍数的平方
                return dx, lambda ddx: _upscale2d(ddx, factor)
            return y, grad
        return func(x)
```

在 upscale2d()里定义了上采样的返回函数为_upscale2d()，同时 upscale2d()的一阶导和二阶导也被定义了出来，都是直接使用_upscale2d()函数作为近似。

_upscale2d()（line 51-68）

```
def _upscale2d(x, factor=2, gain=1):
    assert x.shape.ndims == 4 and all(dim.value is not None for dim in x.shape[1:])
    assert isinstance(factor, int) and factor >= 1

    # 运用增益值。
    if gain != 1:
        x *= gain

    # 无操作=>提前退出。
    if factor == 1:
        return x

    # 上采用使用tf.tile()，下面是骚操作。
    s = x.shape
    x = tf.reshape(x, [-1, s[1], s[2], 1, s[3], 1])  # 把特征的高维度上的每个像素再封装一层，特征的宽维度上的每个像素再封装一层
    x = tf.tile(x, [1, 1, 1, factor, 1, factor])  # 像素copy一下
    x = tf.reshape(x, [-1, s[1], s[2] * factor, s[3] * factor])  # 整合进原始维度下，相当于高和宽放大了factor倍，扩展的像素值是一样的
    return x
```

_upscale2d()的实现使用了 tf.tile()的骚操作，其实很简单，就是复制扩展了像素值。总而言之，线性结构的实现简单但效率低下。

3）递归结构（line 547-556）

```
# ★递归结构：复杂但高效。
if structure == 'recursive':
    def cset(cur_lambda, new_cond, new_lambda):
        return lambda: tf.cond(new_cond, new_lambda, cur_lambda)  # 返回一个函数，依据是否满足new_cond决定返回new_lambda函数还是cur_lambda函数
    def grow(x, res, lod):
        y = block(res, x)
        img = lambda: upscale2d(torgb(res, y), 2**lod)
        img = cset(img, (lod_in > lod), lambda: upscale2d(tflib.lerp(torgb(res, y), upscale2d(torgb(res - 1, x)), lod_in - lod), 2**lod))
            # 如果输入层数lod_in超过当前层lod的话（但同时小于lod+1），实现从lod对应分辨率到lod_in对应分辨率的扩增，采用线性插值；否则按lod处理。
        if lod > 0: img = cset(img, (lod_in < lod), lambda: grow(y, res + 1, lod - 1))
            # 如果lod_in小于lod且不是最后一层的话（也就是前者的res超过后者的res），表明可以进入到下一级分辨率上了，此时res+1，lod-1
        return img()
    images_out = grow(x, 3, resolution_log2 - 3)  # res一开始为3，lod一开始为resolution_log2 - res，利用递归就可以构建res从3增加到resolution_log2的全部架构
```

递归结构下定义了递归函数 grow()，使得只需要调用一次 grow()就能够实现所有分辨率层的构建。它的实现逻辑主要是：比较 lod_in 和 lod 的关系——当 lod_in 超过 lod 时，在同一层级上实现分辨率的线性扩增；当 lod_in 小于 lod 且不是最后一层时，跳转到下一级的分辨率层上。

· 输出（line 558-559）

```
assert images_out.dtype == tf.as_dtype(dtype)
return tf.identity(images_out, name='images_out')  # 输出
```

网络的最终输出为之前结构中得到的 images_out。

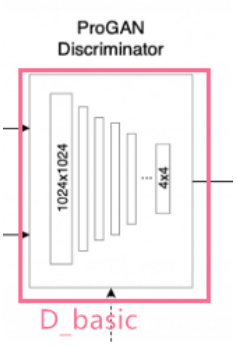
· G_style 架构总览

最后，通过一张 G_style 的完整网络架构图，让我们对各个层的名称、参数量、输入维度和输出维度有更具体的理解。

Gs	Params	OutputShape	WeightShape
latents_in	-	(?, 512)	-
labels_in	-	(?, 0)	-
lod	-	()	-
dlatent_avg	-	(512,)	-
G_mapping/latents_in	-	(?, 512)	-
G_mapping/labels_in	-	(?, 0)	-
G_mapping/PixelNorm	-	(?, 512)	-
G_mapping/Dense0	262656	(?, 512)	(512, 512)
G_mapping/Dense1	262656	(?, 512)	(512, 512)
G_mapping/Dense2	262656	(?, 512)	(512, 512)
G_mapping/Dense3	262656	(?, 512)	(512, 512)
G_mapping/Dense4	262656	(?, 512)	(512, 512)
G_mapping/Dense5	262656	(?, 512)	(512, 512)
G_mapping/Dense6	262656	(?, 512)	(512, 512)
G_mapping/Dense7	262656	(?, 512)	(512, 512)
G_mapping/Broadcast	-	(?, 18, 512)	-
G_mapping/dlatents_out	-	(?, 18, 512)	-
Truncation	-	(?, 18, 512)	-
G_synthesis/dlatents_in	-	(?, 18, 512)	-
G_synthesis/4x4/Const	534528	(?, 512, 4, 4)	(512,)
G_synthesis/4x4/Conv	2885632	(?, 512, 4, 4)	(3, 3, 512, 512)
G_synthesis/ToRGB_lod8	1539	(?, 3, 4, 4)	(1, 1, 512, 3)
G_synthesis/8x8/Conv0_up	2885632	(?, 512, 8, 8)	(3, 3, 512, 512)
G_synthesis/8x8/Conv1	2885632	(?, 512, 8, 8)	(3, 3, 512, 512)
G_synthesis/ToRGB_lod7	1539	(?, 3, 8, 8)	(1, 1, 512, 3)
G_synthesis/Upscale2D	-	(?, 3, 8, 8)	-
G_synthesis/Grow_lod7	-	(?, 3, 8, 8)	-
G_synthesis/16x16/Conv0_up	2885632	(?, 512, 16, 16)	(3, 3, 512, 512)
G_synthesis/16x16/Conv1	2885632	(?, 512, 16, 16)	(3, 3, 512, 512)
G_synthesis/ToRGB_lod6	1539	(?, 3, 16, 16)	(1, 1, 512, 3)
G_synthesis/Upscale2D_1	-	(?, 3, 16, 16)	-
G_synthesis/Grow_lod6	-	(?, 3, 16, 16)	-
G_synthesis/32x32/Conv0_up	2885632	(?, 512, 32, 32)	(3, 3, 512, 512)
G_synthesis/32x32/Conv1	2885632	(?, 512, 32, 32)	(3, 3, 512, 512)
G_synthesis/ToRGB_lod5	1539	(?, 3, 32, 32)	(1, 1, 512, 3)
G_synthesis/Upscale2D_2	-	(?, 3, 32, 32)	-
G_synthesis/Grow_lod5	-	(?, 3, 32, 32)	-
G_synthesis/64x64/Conv0_up	1442816	(?, 256, 64, 64)	(3, 3, 512, 256)
G_synthesis/64x64/Conv1	852992	(?, 256, 64, 64)	(3, 3, 256, 256)
G_synthesis/ToRGB_lod4	771	(?, 3, 64, 64)	(1, 1, 256, 3)
G_synthesis/Upscale2D_3	-	(?, 3, 64, 64)	-
G_synthesis/Grow_lod4	-	(?, 3, 64, 64)	-
G_synthesis/128x128/Conv0_up	426496	(?, 128, 128, 128)	(3, 3, 256, 128)
G_synthesis/128x128/Conv1	279040	(?, 128, 128, 128)	(3, 3, 128, 128)
G_synthesis/ToRGB_lod3	387	(?, 3, 128, 128)	(1, 1, 128, 3)
G_synthesis/Upscale2D_4	-	(?, 3, 128, 128)	-
G_synthesis/Grow_lod3	-	(?, 3, 128, 128)	-
G_synthesis/256x256/Conv0_up	139520	(?, 64, 256, 256)	(3, 3, 128, 64)
G_synthesis/256x256/Conv1	102656	(?, 64, 256, 256)	(3, 3, 64, 64)
G_synthesis/ToRGB_lod2	195	(?, 3, 256, 256)	(1, 1, 64, 3)
G_synthesis/Upscale2D_5	-	(?, 3, 256, 256)	-
G_synthesis/Grow_lod2	-	(?, 3, 256, 256)	-
G_synthesis/512x512/Conv0_up	51328	(?, 32, 512, 512)	(3, 3, 64, 32)
G_synthesis/512x512/Conv1	42112	(?, 32, 512, 512)	(3, 3, 32, 32)
G_synthesis/ToRGB_lod1	99	(?, 3, 512, 512)	(1, 1, 32, 3)
G_synthesis/Upscale2D_6	-	(?, 3, 512, 512)	-
G_synthesis/Grow_lod1	-	(?, 3, 512, 512)	-
G_synthesis/1024x1024/Conv0_up	21056	(?, 16, 1024, 1024)	(3, 3, 32, 16)
G_synthesis/1024x1024/Conv1	18752	(?, 16, 1024, 1024)	(3, 3, 16, 16)
G_synthesis/ToRGB_lod0	51	(?, 3, 1024, 1024)	(1, 1, 16, 3)
G_synthesis/Upscale2D_7	-	(?, 3, 1024, 1024)	-
G_synthesis/Grow_lod0	-	(?, 3, 1024, 1024)	-
G_synthesis/images_out	-	(?, 3, 1024, 1024)	-
G_synthesis/lod	-	()	-
G_synthesis/noise0	-	(1, 1, 4, 4)	-
G_synthesis/noise1	-	(1, 1, 4, 4)	-
G_synthesis/noise2	-	(1, 1, 8, 8)	-
G_synthesis/noise3	-	(1, 1, 8, 8)	-
G_synthesis/noise4	-	(1, 1, 16, 16)	-
G_synthesis/noise5	-	(1, 1, 16, 16)	-
G_synthesis/noise6	-	(1, 1, 32, 32)	-
G_synthesis/noise7	-	(1, 1, 32, 32)	-
G_synthesis/noise8	-	(1, 1, 64, 64)	-
G_synthesis/noise9	-	(1, 1, 64, 64)	-
G_synthesis/noise10	-	(1, 1, 128, 128)	-
G_synthesis/noise11	-	(1, 1, 128, 128)	-
G_synthesis/noise12	-	(1, 1, 256, 256)	-
G_synthesis/noise13	-	(1, 1, 256, 256)	-
G_synthesis/noise14	-	(1, 1, 512, 512)	-
G_synthesis/noise15	-	(1, 1, 512, 512)	-
G_synthesis/noise16	-	(1, 1, 1024, 1024)	-
G_synthesis/noise17	-	(1, 1, 1024, 1024)	-
images_out	-	(?, 3, 1024, 1024)	-
---	---	---	---
Total	26219627		

G_style 完整网络架构图

2.2.4 D_basic 网络



D_basic 网络位于代码 566-661 行。如上图所示，D_basic 网络实现区分合成图片与真实图片的功能，沿用了 ProGAN 判别器的架构，基本上是生成器反过来的样子。在 D_basic 中定义的组件包括：预处理->构建 block 块->★网络增长变换过程->标签计算->输出。其中，网络增长变换过程是指的在训练时期，随着生成图片的分辨率提升，判别网络的架构动态增长的过程。另外，标签计算是指在训练集使用了含标签数据时，会将标签值与判别分数的乘积作为最终判别网络的输出值。

· **D_basic 输入参数（line565-582）**

```
images_in,                          # 第一个输入：图片 [minibatch, channel, height, width].
labels_in,                          # 第二个输入：标签 [minibatch, label_size].
num_channels        = 1,            # 输入颜色通道数。 根据数据集覆盖。
resolution          = 32,           # 输入分辨率。 根据数据集覆盖。
label_size          = 0,            # 标签的维数，0表示没有标签。根据数据集覆盖。
fmap_base           = 8192,         # 特征图的总数目，这儿取8192因为512*(18-2)=8192。
fmap_decay          = 1.0,          # 当分辨率翻倍时以log2降低特征图，这儿指示降低的速率。
fmap_max            = 512,          # 在任何层中特征图的最大数量。
nonlinearity        = 'lrelu',      # 激活函数：'relu', 'lrelu'。
use_wscale          = True,         # 启用均等的学习率？
mbstd_group_size    = 4,            # 小批量标准偏差层的组大小，0表示禁用。
mbstd_num_features  = 1,            # 小批量标准偏差层的特征数量。
dtype               = 'float32',    # 用于激活和输出的数据类型。
fused_scale         = 'auto',       # True = 融合卷积+缩放，False = 单独操作，'auto'= 自动决定。
blur_filter         = [1,2,1],      # 重采样激活时应用的低通卷积核（Low-pass filter）。None表示不过滤。
structure           = 'auto',       # 'fixed' = 无渐进式增长，'linear' = 人类可读，'recursive' = 有效，'auto' = 自动选择。
is_template_graph   = False,        # True表示由Network类构造的模板图，False表示实际评估。
**kwargs):                          # 忽略无法识别的关键字参数。
```

输入参数包括图片、标签以及这二者的相关配置，和一些可选的参数，包括各层特征图的设置、激活函数设置、小批量标准偏差层设置、数据处理设置以及网络增长架构的设置等。

· **预处理（line584-596）**

```
resolution_log2 = int(np.log2(resolution))  # 计算分辨率是2的多少次方
assert resolution == 2**resolution_log2 and resolution >= 4  # 分辨率需要大于等于32，因为训练从学习生成32*32的图片开始
def nf(stage): return min(int(fmap_base / (2.0 ** (stage * fmap_decay))), fmap_max)
    # nf()返回在第stage层中特征图的数量——当stage<4时，特征图数量为512；当stage>4时，每多一层特征图数量就减半。
def blur(x): return blur2d(x, blur_filter) if blur_filter else x  # 对图片进行滤波模糊操作，有利于降噪
if structure == 'auto': structure = 'linear' if is_template_graph else 'recursive'  # 依据is_template_graph选择架构为'linear'或'recursive'
act, gain = {'relu': (tf.nn.relu, np.sqrt(2)), 'lrelu': (leaky_relu, np.sqrt(2))}[nonlinearity]  # 激活函数
# 输入处理
images_in.set_shape([None, num_channels, resolution, resolution])
labels_in.set_shape([None, label_size])
images_in = tf.cast(images_in, dtype)
labels_in = tf.cast(labels_in, dtype)
lod_in = tf.cast(tf.get_variable('lod', initializer=np.float32(0.0), trainable=False), dtype)  # 输入的分辨率级别，lod = resolution_log2 - res
scores_out = None  # 输出分数
```

预处理主要包括细化网络配置、输入数据的处理以及输出数据的定义，包含内容与 G_synthesis 中的预处理过程类似。

· **构建 block 块（line 599-618）**

```
# 构建block块。
def fromrgb(x, res):  # res从2增加到resolution_log2；这个函数实现RGB图像到特征图的转换。
    with tf.variable_scope('FromRGB_lod%d' % (resolution_log2 - res)):
        return act(apply_bias(conv2d(x, fmaps=nf(res-1), kernel=1, gain=gain, use_wscale=use_wscale)))  # 简单卷积实现，并应用激活函数
def block(x, res):  # res从2增加到resolution_log2；这些层被写在函数里方便网络需要时再创建。
    with tf.variable_scope('%dx%d' % (2**res, 2**res)):
        if res >= 3:  # 8x8分辨率及以上
            with tf.variable_scope('Conv0'):
                x = act(apply_bias(conv2d(x, fmaps=nf(res-1), kernel=3, gain=gain, use_wscale=use_wscale)))  # 构建一个卷积层
            with tf.variable_scope('Conv1_down'):
                x = act(apply_bias(conv2d_downscale2d(blur(x), fmaps=nf(res-2), kernel=3, gain=gain, use_wscale=use_wscale, fused_scale=fused_scale)))  # 构建一个下采样层
        else:  # 4x4分辨率，得到判别分数scores_out
            if mbstd_group_size > 1:
                x = minibatch_stddev_layer(x, mbstd_group_size, mbstd_num_features)  # 构建一个小偏量标准偏差层
            with tf.variable_scope('Conv'):
                x = act(apply_bias(conv2d(x, fmaps=nf(res-1), kernel=3, gain=gain, use_wscale=use_wscale)))  # 卷积
            with tf.variable_scope('Dense0'):
                x = act(apply_bias(dense(x, fmaps=nf(res-2), gain=gain, use_wscale=use_wscale)))  # 全连接
            with tf.variable_scope('Dense1'):
                x = apply_bias(dense(x, fmaps=max(label_size, 1), gain=1, use_wscale=use_wscale))  # 全连接
        return x
```

在训练过程中网络需实时处理图片，StyleGAN 中定义了 fromrgb()函数，负责将对应分辨率的 RGB 图像转换为特征图。

在 block 函数中，当分辨率不低于 8*8 时，一个 block 包含一个卷积层和一个下采样层（conv2d_downscale2d）；而当分辨率为最开始的 4*4 时，一个 block 包含一个小批量标准偏差层（minibatch_stddev_layer）、一个卷积层和两个全连接层。

conv2d_downscale2d()（line 193-208）

```
def conv2d_downscale2d(x, fmaps, kernel, fused_scale='auto', **kwargs):
    assert kernel >= 1 and kernel % 2 == 1
    assert fused_scale in [True, False, 'auto']
    if fused_scale == 'auto':
        fused_scale = min(x.shape[2:]) >= 128

    # 不融合=>直接调用各个操作。
    if not fused_scale:
        return downscale2d(conv2d(x, fmaps, kernel, **kwargs))

    # 融合=>使用tf.nn.conv2d_transpose()同时执行两个操作。
    w = get_weight([kernel, kernel, x.shape[1].value, fmaps], **kwargs)  # 创建一个卷积层w，shape为[kernel, kernel, fmaps_in, fmaps_out]
    w = tf.pad(w, [[1,1], [1,1], [0,0], [0,0]], mode='CONSTANT')  # 对w进行填充，在两个kernel的第0维和最后一维分别填充0，w的shape变为[kernel+2, kernel+2, fmaps_in, fmaps_out]
    w = tf.add_n([w[1:, 1:], w[:-1, 1:], w[1:, :-1], w[:-1, :-1]]) * 0.25  # 填充区域求和两次，非填充区域求和四次，最后再除以4，w的shape不变。为什么要对卷积核做这样的处理呢？？
    w = tf.cast(w, x.dtype)
    return tf.nn.conv2d(x, w, strides=[1,1,2,2], padding='SAME', data_format='NCHW')
        # 最终的卷积输出，可以验算一下：特征x的维度是[Num,fmaps_in,H,W],卷积核w的维度是[kernel+2, kernel+2, fmaps_in, fmaps_out],strides为[1,1,2,2],
        # 这三者做卷积的输出维度是[Num,fmaps_out,H//2,W//2]
```

conv2d_downscale2d 利用 tf.nn.conv2d 卷积操作实现将特征图缩小一倍。其中一个值得注意的操作是，其卷积核被轻微平移了四次并对自身做了叠加然后取平均值，这样或许对于提取特征有帮助，但我查阅不到相关资料证明这一点。

minibatch_stddev_layer()（line 283-296）

```
# 小批量标准偏差。
def minibatch_stddev_layer(x, group_size=4, num_new_features=1):
    with tf.variable_scope('MinibatchStddev'):
        group_size = tf.minimum(group_size, tf.shape(x)[0])     # 小批量一定能被group_size整除（或小于group_size）。
        s = x.shape                                             # [NCHW]  输入shape.
        y = tf.reshape(x, [group_size, -1, num_new_features, s[1]//num_new_features, s[2], s[3]])   # [GMncHW] 将小批量拆分为M个大小为G的组。将通道拆分为n个c个通道的组。
        y = tf.cast(y, tf.float32)                              # [GMncHW] 转换成FP32。
        y -= tf.reduce_mean(y, axis=0, keepdims=True)           # [GMncHW] 按组减去均值。
        y = tf.reduce_mean(tf.square(y), axis=0)                # [MncHW]  按组计算方差。
        y = tf.sqrt(y + 1e-8)                                   # [MncHW]  按组计算标准方差。
        y = tf.reduce_mean(y, axis=[2,3,4], keepdims=True)      # [Mn111]  在特征图和像素上采取平均值。
        y = tf.reduce_mean(y, axis=[2])                         # [Mn11] 将通道划分为c个通道组。
        y = tf.cast(y, x.dtype)                                 # [Mn11]  转换回原始的数据类型。
        y = tf.tile(y, [group_size, 1, s[2], s[3]])             # [NnHW]  按组和像素进行复制。
        return tf.concat([x, y], axis=1)                        # [NCHW]  添加为新的特征图。
```

在 4*4 分辨率的 block 中，构建了一个小批量标准偏差层，将特征图标准化处理，这样能让判别网络收敛得更快。

· **★网络增长变换过程（line 621-650）**

StyleGAN 的判别网络需具备动态变换的能力，代码中定义了三种结构组合方式，分别是：固定结构、线性结构与递归结构。

1）固定结构（line 621-625）

```
# 固定结构：简单高效，但不支持渐进式增长。
if structure == 'fixed':
    x = fromrgb(images_in, resolution_log2)  # 将输入图片转换为特征x
    for res in range(resolution_log2, 2, -1):
        x = block(x, res)  # 相当于直接构建了一个从1024*1024分辨率降到4*4分辨率的下采样网络
    scores_out = block(x, 2)  # 输出为判别分数
```

固定结构构建了直达 1024*1024 分辨率的判别器网络，简单高效但不支持渐进式增长。

2）线性结构（line 628-638）

```
# 线性结构：简单但效率低下。
if structure == 'linear':
    img = images_in
    x = fromrgb(img, resolution_log2)  # 将输入图片转换为特征x
    for res in range(resolution_log2, 2, -1):  # res从resolution_log2降低到3
        lod = resolution_log2 - res
        x = block(x, res)
        img = downscale2d(img)  # 通过downscale2d()构建下采样层，将当前分辨率缩小一倍
        y = fromrgb(img, res - 1)
        with tf.variable_scope('Grow_lod%d' % lod):
            x = tflib.lerp_clip(x, y, lod_in - lod)  # 依靠含大小值裁剪的线性插值实现图片缩小，相当于在过渡阶段实现平滑过渡
    scores_out = block(x, 2)
```

线性结构构建了 downscale2d()的下采样层，能将当前分辨率缩小一倍。另外在不同分辨率间变换时，线性结构采用了含大小值裁剪的线性插值，实现了不同分辨率下的平滑过渡。

downscale2d()（line 120-130）

```
def downscale2d(x, factor=2):
    with tf.variable_scope('Downscale2D'):
        @tf.custom_gradient
        def func(x):  # 定义一个函数，返回下采样后的特征值以及该函数的梯度计算式grad()
            y = _downscale2d(x, factor)
            @tf.custom_gradient
            def grad(dy):  # func函数的梯度计算式与blur2d()写法类似，对于此种函数具有通用性
                dx = _upscale2d(dy, factor, gain=1/factor**2)  # 下采样的增益值取放大倍数平方的倒数
                return dx, lambda ddx: _downscale2d(ddx, factor)
            return y, grad
        return func(x)
```

在 downscale2d()里定义了下采样的返回函数为_downscale2d()，同时 downscale2d()的一阶导和二阶导也被定义了出来，都是直接使用_downscale2d ()函数作为近似。

_downscale2d()（line 70-90）

```
def _downscale2d(x, factor=2, gain=1):
    assert x.shape.ndims == 4 and all(dim.value is not None for dim in x.shape[1:])
    assert isinstance(factor, int) and factor >= 1

    # 卷积核大小为2x2 => 下采用采用_blur2d()
    if factor == 2 and x.dtype == tf.float32:
        f = [np.sqrt(gain) / factor] * factor
        return _blur2d(x, f=f, normalize=False, stride=factor)

    # 运用增益值。
    if gain != 1:
        x *= gain

    # 无操作=>提前退出。
    if factor == 1:
        return x

    # 下采样时采用平均池化。
    # 注意：需要tf_config ['graph_options.place_pruned_graph'] = True 才能正常工作。
    ksize = [1, 1, factor, factor]
    return tf.nn.avg_pool(x, ksize=ksize, strides=ksize, padding='VALID', data_format='NCHW')
```

_downscale2d()中，如果卷积核大小为 2*2，则直接返回_blur2d()的结果；否则采用平均池化的方式实现下采样。

3）递归结构（line 641-650）

```
# 递归结构：复杂但高效。
if structure == 'recursive':  # 注意判别器在训练时是输入图片先进入lod最小的层，但是构建判别网络时是lod从大往小构建，所以递归的过程是与生成器相反的。
    def cset(cur_lambda, new_cond, new_lambda):
        return lambda: tf.cond(new_cond, new_lambda, cur_lambda)  # 返回一个函数，依据是否满足new_cond决定返回new_lambda函数还是cur_lambda函数
    def grow(res, lod):
        x = lambda: fromrgb(downscale2d(images_in, 2**lod), res)  # 先暂时将下采样函数赋给x
        if lod > 0: x = cset(x, (lod_in < lod), lambda: grow(res + 1, lod - 1))
            # 非第一层时，如果输入层数lod_in小于当前层lod的话，表明可以进入到下一级分辨率上了，将grow()赋给x；否则x还是保留为下采样函数。
        x = block(x(), res); y = lambda: x  # x执行一次自身的函数，构建出一个block，并将结果赋给y（以函数的形式）
        if res > 2: y = cset(y, (lod_in > lod), lambda: tflib.lerp(x, fromrgb(downscale2d(images_in, 2**(lod+1)), res - 1), lod_in - lod))
            # 非最后一层时，如果输入层数lod_in大于当前层lod的话，表明需要进行插值操作，将lerp()赋给y；否则y还是保留为之前的操作。
        return y()
    scores_out = grow(2, resolution_log2 - 2)  # 构建判别网络时是lod从大往小构建，所以一开始的lod输入为8
```

递归结构下定义了递归函数 grow()，使得只需要调用一次 grow()就能够实现所有分辨率层的构建。它的实现逻辑比较复杂，请参见代码注释；值得注意的是，构建判别网络时是 lod 从大往小构建，所以递归的过程是与生成器相反的。

· 标签计算（line 653-655）

```
# 标签条件来自“哪种GAN训练方法实际上会收敛？”
if label_size:
    with tf.variable_scope('LabelSwitch'):
        scores_out = tf.reduce_sum(scores_out * labels_in, axis=1, keepdims=True)
```

如果使用了标签的话，将标签值与判别分数的乘积作为最终判别网络的输出值。

· 输出（line 657-659）

```
assert scores_out.dtype == tf.as_dtype(dtype)
scores_out = tf.identity(scores_out, name='scores_out')
return scores_out  # 输出
```

网络的最终输出为 scores_out。

· D_basic 架构总览

最后，通过一张 D_basic 的完整网络架构图，让我们对各个层的名称、参数量、输入维度和输出维度有更具体的理解。

D	Params	OutputShape	WeightShape
images_in	-	(?, 3, 1024, 1024)	-
labels_in	-	(?, 0)	-
lod	-	()	-
FromRGB_lod0	64	(?, 16, 1024, 1024)	(1, 1, 3, 16)
1024x1024/Conv0	2320	(?, 16, 1024, 1024)	(3, 3, 16, 16)
1024x1024/Conv1_down	4640	(?, 32, 512, 512)	(3, 3, 16, 32)
Downscale2D	-	(?, 3, 512, 512)	-
FromRGB_lod1	128	(?, 32, 512, 512)	(1, 1, 3, 32)
Grow_lod0	-	(?, 32, 512, 512)	-
512x512/Conv0	9248	(?, 32, 512, 512)	(3, 3, 32, 32)
512x512/Conv1_down	18496	(?, 64, 256, 256)	(3, 3, 32, 64)
Downscale2D_1	-	(?, 3, 256, 256)	-
FromRGB_lod2	256	(?, 64, 256, 256)	(1, 1, 3, 64)
Grow_lod1	-	(?, 64, 256, 256)	-
256x256/Conv0	36928	(?, 64, 256, 256)	(3, 3, 64, 64)
256x256/Conv1_down	73856	(?, 128, 128, 128)	(3, 3, 64, 128)
Downscale2D_2	-	(?, 3, 128, 128)	-
FromRGB_lod3	512	(?, 128, 128, 128)	(1, 1, 3, 128)
Grow_lod2	-	(?, 128, 128, 128)	-
128x128/Conv0	147584	(?, 128, 128, 128)	(3, 3, 128, 128)
128x128/Conv1_down	295168	(?, 256, 64, 64)	(3, 3, 128, 256)
Downscale2D_3	-	(?, 3, 64, 64)	-
FromRGB_lod4	1024	(?, 256, 64, 64)	(1, 1, 3, 256)
Grow_lod3	-	(?, 256, 64, 64)	-
64x64/Conv0	590080	(?, 256, 64, 64)	(3, 3, 256, 256)
64x64/Conv1_down	1180160	(?, 512, 32, 32)	(3, 3, 256, 512)
Downscale2D_4	-	(?, 3, 32, 32)	-
FromRGB_lod5	2048	(?, 512, 32, 32)	(1, 1, 3, 512)
Grow_lod4	-	(?, 512, 32, 32)	-
32x32/Conv0	2359808	(?, 512, 32, 32)	(3, 3, 512, 512)
32x32/Conv1_down	2359808	(?, 512, 16, 16)	(3, 3, 512, 512)
Downscale2D_5	-	(?, 3, 16, 16)	-
FromRGB_lod6	2048	(?, 512, 16, 16)	(1, 1, 3, 512)
Grow_lod5	-	(?, 512, 16, 16)	-
16x16/Conv0	2359808	(?, 512, 16, 16)	(3, 3, 512, 512)
16x16/Conv1_down	2359808	(?, 512, 8, 8)	(3, 3, 512, 512)
Downscale2D_6	-	(?, 3, 8, 8)	-
FromRGB_lod7	2048	(?, 512, 8, 8)	(1, 1, 3, 512)
Grow_lod6	-	(?, 512, 8, 8)	-
8x8/Conv0	2359808	(?, 512, 8, 8)	(3, 3, 512, 512)
8x8/Conv1_down	2359808	(?, 512, 4, 4)	(3, 3, 512, 512)
Downscale2D_7	-	(?, 3, 4, 4)	-
FromRGB_lod8	2048	(?, 512, 4, 4)	(1, 1, 3, 512)
Grow_lod7	-	(?, 512, 4, 4)	-
4x4/MinibatchStddev	-	(?, 513, 4, 4)	-
4x4/Conv	2364416	(?, 512, 4, 4)	(3, 3, 513, 512)
4x4/Dense0	4194816	(?, 512)	(8192, 512)
4x4/Dense1	513	(?, 1)	(512, 1)
scores_out	-	(?, 1)	-
---	---	---	---
Total	23087249		

D_basic 完整网络架构图

2.3 损失函数代码解读

StyleGAN 的损失函数写在 training/loss.py 下，包括了三种损失：WGAN/WGAN-GP 损失，Hinge 判别损失，以及 StyleGAN 倡导的 logistic 损失。最终 StyleGAN 选择了添加了简单梯度惩罚项的 Logistic 损失。

2.3.1 WGAN、WGAN-GP 损失

· **Loss_G（line 26-32）**

```
def G_wgan(G, D, opt, training_set, minibatch_size): # pylint: disable=unused-argument
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    labels = training_set.get_random_labels_tf(minibatch_size)
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    loss = -fake_scores_out
    return loss  # Loss_G = -D(G(z))
```

生成器希望假样本的得分越高越好，故$\mathrm{Loss}_G = -\mathrm{D}(\mathrm{G}(\mathrm{z}))$。

· **Loss_D 不含梯度惩罚（line 34-48）**

```
def D_wgan(G, D, opt, training_set, minibatch_size, reals, labels, # pylint: disable=unused-argument
    wgan_epsilon = 0.001):  # ε项的值， \epsilon_{drift}.

    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = fp32(D.get_output_for(reals, labels, is_training=True))
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = fake_scores_out - real_scores_out

    with tf.name_scope('EpsilonPenalty'):
        epsilon_penalty = autosummary('Loss/epsilon_penalty', tf.square(real_scores_out))
    loss += epsilon_penalty * wgan_epsilon
    return loss  # Loss_D = D(G(z)) - D(x) + ε·D(x)^2
```

判别器希望假样本的得分越低越好同时真样本的得分越高越好，并且通过正则项限制真样本得分不宜过高否则易导致 G 失去学习的梯度，故$\mathrm{Loss}_D = \mathrm{D}(\mathrm{G}(\mathrm{z})) - \mathrm{D}(\mathrm{x}) + \varepsilon \cdot \mathrm{D}(\mathrm{x})^2$。

· **Loss_D 含梯度惩罚（line 50-78）**

```
def D_wgan_gp(G, D, opt, training_set, minibatch_size, reals, labels, # pylint: disable=unused-argument
    wgan_lambda     = 10.0,     # 梯度惩罚项的权重。
    wgan_epsilon    = 0.001,    # ε项的值, \epsilon_{drift}.
    wgan_target     = 1.0):     # 梯度幅度的目标值，即满足1-lipschitz范式所以梯度的模得小于等于1。

    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = fp32(D.get_output_for(reals, labels, is_training=True))
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = fake_scores_out - real_scores_out

    with tf.name_scope('GradientPenalty'):
        mixing_factors = tf.random_uniform([minibatch_size, 1, 1, 1], 0.0, 1.0, dtype=fake_images_out.dtype)  # alpha
        mixed_images_out = tflib.lerp(tf.cast(reals, fake_images_out.dtype), fake_images_out, mixing_factors)
            # 惩罚区的样本为xp = alpha * x + (1 - alpha) * G(z)
        mixed_scores_out = fp32(D.get_output_for(mixed_images_out, labels, is_training=True))  # 惩罚区样本的判别值D(xp)
        mixed_scores_out = autosummary('Loss/scores/mixed', mixed_scores_out)
        mixed_loss = opt.apply_loss_scaling(tf.reduce_sum(mixed_scores_out))
        mixed_grads = opt.undo_loss_scaling(fp32(tf.gradients(mixed_loss, [mixed_images_out])[0]))  # 惩罚区样本的梯度∇T
        mixed_norms = tf.sqrt(tf.reduce_sum(tf.square(mixed_grads), axis=[1,2,3]))  # 惩罚区样本梯度∇T的模||∇T||
        mixed_norms = autosummary('Loss/mixed_norms', mixed_norms)
        gradient_penalty = tf.square(mixed_norms - wgan_target)  # 惩罚项为(||∇T||-1)^2
    loss += gradient_penalty * (wgan_lambda / (wgan_target**2))

    with tf.name_scope('EpsilonPenalty'):
        epsilon_penalty = autosummary('Loss/epsilon_penalty', tf.square(real_scores_out))
    loss += epsilon_penalty * wgan_epsilon
    return loss  # Loss_D = D(G(z)) - D(x) + η·(||∇T||-1)^2 + ε·D(x)^2
```

WGAN-GP 中的判别损失多增添了一项梯度惩罚项，它的作用是让判别函数尽量符合1-lipschitz范数限制，即梯度的模始终小于1，这样才能让判别器的求解结果逼近Wasserstein距离，但是WGAN-GP并未达到严格的1-lipschitz范数限制，真正实现这一限制的是SNGAN中的谱归一化方法。最终WGAN-GP的判别损失为：

$$\mathrm{Loss}_D = \mathrm{D}\big(\mathrm{G}(\mathrm{z})\big) - \mathrm{D}(\mathrm{x}) + \eta \cdot (||\nabla_T|| - 1)^2 + \varepsilon \cdot \mathrm{D}(\mathrm{x})^2$$

2.3.2 Hinge 判别损失

· Loss_D 不含梯度惩罚（line 84-92）

```
def D_hinge(G, D, opt, training_set, minibatch_size, reals, labels): # pylint: disable=unused-argument
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = fp32(D.get_output_for(reals, labels, is_training=True))
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = tf.maximum(0., 1.+fake_scores_out) + tf.maximum(0., 1.-real_scores_out)
    return loss  # Loss_D = max(0,1+D(G(z))) + max(0,1-D(x))
```

Hinge 损失也是非常经典的一种损失，它限制判别函数的值只处在(-1,1)间有效，其表达式为：$\mathrm{Loss}_D = \max(0,1 + \mathrm{D}(\mathrm{G}(\mathrm{z}))) + \max(0,1 - \mathrm{D}(\mathrm{x}))$。

· Loss_D 含梯度惩罚（line 94-117）

```
def D_hinge_gp(G, D, opt, training_set, minibatch_size, reals, labels, # pylint: disable=unused-argument
    wgan_lambda     = 10.0,     # 梯度惩罚项的权重。
    wgan_target     = 1.0):     # 梯度幅度的目标值。

    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = fp32(D.get_output_for(reals, labels, is_training=True))
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = tf.maximum(0., 1.+fake_scores_out) + tf.maximum(0., 1.-real_scores_out)

    with tf.name_scope('GradientPenalty'):
        mixing_factors = tf.random_uniform([minibatch_size, 1, 1, 1], 0.0, 1.0, dtype=fake_images_out.dtype)
        mixed_images_out = tflib.lerp(tf.cast(reals, fake_images_out.dtype), fake_images_out, mixing_factors)
        mixed_scores_out = fp32(D.get_output_for(mixed_images_out, labels, is_training=True))
        mixed_scores_out = autosummary('Loss/scores/mixed', mixed_scores_out)
        mixed_loss = opt.apply_loss_scaling(tf.reduce_sum(mixed_scores_out))
        mixed_grads = opt.undo_loss_scaling(fp32(tf.gradients(mixed_loss, [mixed_images_out])[0]))
        mixed_norms = tf.sqrt(tf.reduce_sum(tf.square(mixed_grads), axis=[1,2,3]))
        mixed_norms = autosummary('Loss/mixed_norms', mixed_norms)
        gradient_penalty = tf.square(mixed_norms - wgan_target)
    loss += gradient_penalty * (wgan_lambda / (wgan_target**2))
    return loss  # Loss_D = max(0,1+D(G(z))) + max(0,1-D(x)) + η·(||∇T||-1)^2
```

带有 GP 的 Hinge 损失添加的梯度惩罚项与 WGAN-GP 中的完全一致，都是逼迫所有位于惩罚区域中的样本梯度都尽量逼近 1（或不超过 1），最终 Hinge-GP 的判别损失为：

$$\mathrm{Loss}_D = \max(0,1 + \mathrm{D}(\mathrm{G}(\mathrm{z}))) + \max(0,1 - \mathrm{D}(\mathrm{x})) + \eta \cdot (||\nabla_T|| - 1)^2$$

2.3.3 StyleGAN 提倡损失——Logistic 损失

· Loss_G 饱和逻辑斯谛（line 124-130）

```python
def G_logistic_saturating(G, D, opt, training_set, minibatch_size):  # pylint: disable=unused-argument
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    labels = training_set.get_random_labels_tf(minibatch_size)
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    loss = -tf.nn.softplus(fake_scores_out)
    return loss  # Loss_G = -log(exp(D(G(z))) + 1)
```

选用饱和逻辑斯谛损失时，$\mathrm{Loss}_G = -\log(\exp(\mathrm{D}(\mathrm{G}(\mathrm{z}))) + 1)$

· Loss_G 非饱和逻辑斯谛（line 132-138）

```python
def G_logistic_nonsaturating(G, D, opt, training_set, minibatch_size): # pylint: disable=unused-argument
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    labels = training_set.get_random_labels_tf(minibatch_size)
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    loss = tf.nn.softplus(-fake_scores_out)
    return loss  # Loss_G = log(exp(-D(G(z))) + 1)
```

选用非饱和逻辑斯谛损失时，$\mathrm{Loss}_G = \log(\exp(-\mathrm{D}(\mathrm{G}(\mathrm{z}))) + 1)$

· Loss_D 不含梯度惩罚（line 140-149）

```python
def D_logistic(G, D, opt, training_set, minibatch_size, reals, labels): # pylint: disable=unused-argument
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = fp32(D.get_output_for(reals, labels, is_training=True))
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = tf.nn.softplus(fake_scores_out)
    loss += tf.nn.softplus(-real_scores_out)
    return loss  # Loss_D = log(exp(D(G(z))) + 1) + log(exp(-D(x)) + 1)
```

不含梯度惩罚时，$\mathrm{Loss}_D = \log(\exp(\mathrm{D}(\mathrm{G}(\mathrm{z}))) + 1) + \log(\exp(-\mathrm{D}(\mathrm{x})) + 1)$

· Loss_D 含梯度惩罚（line 151-176）

```python
def D_logistic_simplegp(G, D, opt, training_set, minibatch_size, reals, labels, r1_gamma=10.0, r2_gamma=0.0): # pylint: disable=unused-argument
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = fp32(D.get_output_for(reals, labels, is_training=True))
    fake_scores_out = fp32(D.get_output_for(fake_images_out, labels, is_training=True))
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = tf.nn.softplus(fake_scores_out)
    loss += tf.nn.softplus(-real_scores_out)

    if r1_gamma != 0.0:
        with tf.name_scope('R1Penalty'):
            real_loss = opt.apply_loss_scaling(tf.reduce_sum(real_scores_out))  # 惩罚区(来自真实样本)的判别值D(x_real)
            real_grads = opt.undo_loss_scaling(fp32(tf.gradients(real_loss, [reals])[0]))  # 惩罚区样本的梯度∇T_real
            r1_penalty = tf.reduce_sum(tf.square(real_grads), axis=[1,2,3])  # 惩罚项为∑(∇T_real^2)
            r1_penalty = autosummary('Loss/r1_penalty', r1_penalty)
        loss += r1_penalty * (r1_gamma * 0.5)

    if r2_gamma != 0.0:
        with tf.name_scope('R2Penalty'):
            fake_loss = opt.apply_loss_scaling(tf.reduce_sum(fake_scores_out))  # 惩罚区(来自生成样本)的判别值D(x_fake)
            fake_grads = opt.undo_loss_scaling(fp32(tf.gradients(fake_loss, [fake_images_out])[0]))  # 惩罚区样本的梯度∇T_fake
            r2_penalty = tf.reduce_sum(tf.square(fake_grads), axis=[1,2,3])  # 惩罚项为∑(∇T_fake^2)
            r2_penalty = autosummary('Loss/r2_penalty', r2_penalty)
        loss += r2_penalty * (r2_gamma * 0.5)
    return loss  # Loss_D = log(exp(D(G(z))) + 1) + log(exp(-D(x)) + 1) + r1_gamma*0.5*∑(∇T_real^2) + r2_gamma*0.5*∑(∇T_fake^2)
```

含梯度惩罚时，可能受到梯度惩罚的样本有两种：第一种是来自真实分布的样本，其梯度绝对值过高时会受到惩罚；第二种是来自生成分布的样本，其梯度绝对值过高时也会受到惩罚。StyleGAN 中默认保留了对真实样本的梯度惩罚（$\mathrm{r1}_{gamma} = 10$），而舍弃了对生成样本的梯度惩罚（$\mathrm{r2}_{gamma} = 0$）。含有梯度惩罚的判别损失为：

$$\mathrm{Loss}_D = \log(\exp(\mathrm{D}(\mathrm{G}(\mathrm{z}))) + 1) + \log(\exp(-\mathrm{D}(\mathrm{x})) + 1) + \mathrm{r1}_{gamma} * 0.5 * \sum \nabla_{\mathrm{T}_{real}}{}^2 + \mathrm{r2}_{gamma} * 0.5 * \sum \nabla_{\mathrm{T}_{fake}}{}^2$$

最终，StyleGAN 官方代码采用的损失为 G_logistic_nonsaturating 与 D_logistic_simplegp。

2.4 训练过程代码解读

StyleGAN 的训练调用接口写在 train.py 下，训练过程写在 training/training_loop.py 下。本节主要讲述在这两个代码文件中分别定义了哪些训练设置，以及简要说明 training_loop 的训练流程，从而方便我们在自定义训练时做出更合适的设置。

· **train.py 主要设置（line 21-55）**

```
desc           = 'sgan'                                                 # 包含在结果子目录名称中的描述字符串。
train          = EasyDict(run_func_name='training.training_loop.training_loop')  # 训练过程设置。
G              = EasyDict(func_name='training.networks_stylegan.G_style')        # 生成网络架构设置。
D              = EasyDict(func_name='training.networks_stylegan.D_basic')        # 判别网络架构设置。
G_opt          = EasyDict(beta1=0.0, beta2=0.99, epsilon=1e-8)          # 生成网络优化器设置。
D_opt          = EasyDict(beta1=0.0, beta2=0.99, epsilon=1e-8)          # 判别网络优化器设置。
G_loss         = EasyDict(func_name='training.loss.G_logistic_nonsaturating')    # 生成损失设置。
D_loss         = EasyDict(func_name='training.loss.D_logistic_simplegp', r1_gamma=10.0)  # 判别损失设置。
dataset        = EasyDict()                                             # 数据集设置，在后文确认。
sched          = EasyDict()                                             # 训练计划设置，在后文确认。
grid           = EasyDict(size='4k', layout='random')                   # setup_snapshot_image_grid()相关设置。
metrics        = [metric_base.fid50k]                                   # 指标方法设置。
submit_config = dnnlib.SubmitConfig()                                   # dnnlib.submit_run()相关设置。
tf_config      = {'rnd.np_random_seed': 1000}                           # tflib.init_tf()相关设置。

# 数据集。
desc += '-ffhq';     dataset = EasyDict(tfrecord_dir='ffhq');                train.mirror_augment = True
#desc += '-ffhq512';  dataset = EasyDict(tfrecord_dir='ffhq', resolution=512); train.mirror_augment = True
#desc += '-ffhq256';  dataset = EasyDict(tfrecord_dir='ffhq', resolution=256); train.mirror_augment = True
#desc += '-celebahq'; dataset = EasyDict(tfrecord_dir='celebahq');            train.mirror_augment = True
#desc += '-bedroom';  dataset = EasyDict(tfrecord_dir='lsun-bedroom-full');   train.mirror_augment = False
#desc += '-car';      dataset = EasyDict(tfrecord_dir='lsun-car-512x384');    train.mirror_augment = False
#desc += '-cat';      dataset = EasyDict(tfrecord_dir='lsun-cat-full');       train.mirror_augment = False

# GPU数量。
#desc += '-1gpu'; submit_config.num_gpus = 1; sched.minibatch_base = 4; sched.minibatch_dict = {4: 128, 8: 128, 16: 128, 32: 64, 64: 32, 128: 16, 256: 8, 512: 4}
#desc += '-2gpu'; submit_config.num_gpus = 2; sched.minibatch_base = 8; sched.minibatch_dict = {4: 256, 8: 256, 16: 128, 32: 64, 64: 32, 128: 16, 256: 8}
#desc += '-4gpu'; submit_config.num_gpus = 4; sched.minibatch_base = 16; sched.minibatch_dict = {4: 512, 8: 256, 16: 128, 32: 64, 64: 32, 128: 16}
desc += '-8gpu'; submit_config.num_gpus = 8; sched.minibatch_base = 32; sched.minibatch_dict = {4: 512, 8: 256, 16: 128, 32: 64, 64: 32}

# 默认设置。
train.total_kimg = 25000
sched.lod_initial_resolution = 8
sched.G_lrate_dict = {128: 0.0015, 256: 0.002, 512: 0.003, 1024: 0.003}
sched.D_lrate_dict = EasyDict(sched.G_lrate_dict)
```

在训练 StyleGAN 的初始接口 train.py 下定义了一些主要的设置，包括生成网络和判别网络各自的架构、优化和损失的设置，以及训练计划、数据集和 GPU 的设置等。配置完成之后，通过调用 dnnlib.submit_run(**kwargs)就能进入到 StyleGAN 的训练过程中。

在我们自定义训练时，通常需要手动调整数据集设置（名称和分辨率）、GPU 设置（GPU 数量和 batch 大小（它取决于 GPU 缓存大小））以及默认设置（总迭代数和学习率），而网络架构的设置则视情况而定，一般不建议修改。

· **training/training_loop.py 训练计划设置（line 56-71）**

```
def training_schedule(
    cur_nimg,
    training_set,
    num_gpus,
    lod_initial_resolution  = 4,        # 开始训练时使用的图像分辨率。
    lod_training_kimg       = 600,      # 在将分辨率提高一倍之前要显示数千个真实图像。
    lod_transition_kimg     = 600,      # 淡入新图层时要显示数千个真实图像。
    minibatch_base          = 16,       # 小批量大小的最大值，在GPU之间平均分配。
    minibatch_dict          = {},       # 特定分辨率的重写。
    max_minibatch_per_gpu   = {},       # 每个GPU的特定分辨率的小批量大小的最大值。
    G_lrate_base            = 0.001,    # 生成器的学习率。
    G_lrate_dict            = {},       # 特定分辨率的重写。
    D_lrate_base            = 0.001,    # 判别器的学习率。
    D_lrate_dict            = {},       # 特定分辨率的重写。
    lrate_rampup_kimg       = 0,        # 学习率提升的持续时间。
    tick_kimg_base          = 160,      # 训练过程中生成快照的默认间隔。
    tick_kimg_dict          = {4: 160, 8:140, 16:120, 32:100, 64:80, 128:60, 256:40, 512:30, 1024:20}):  # 特定分辨率的重写。
```

在 training_schedule()函数中可以定义一些训练计划的设置，内容如上图所示。

· **★training/training_loop.py 训练过程设置（line 113-139）**

```
def training_loop(
    submit_config,
    G_args                  = {},       # 生成网络的设置。
    D_args                  = {},       # 判别网络的设置。
    G_opt_args              = {},       # 生成网络优化器设置。
    D_opt_args              = {},       # 判别网络优化器设置。
    G_loss_args             = {},       # 生成损失设置。
    D_loss_args             = {},       # 判别损失设置。
    dataset_args            = {},       # 数据集设置。
    sched_args              = {},       # 训练计划设置。
    grid_args               = {},       # setup_snapshot_image_grid()相关设置。
    metric_arg_list         = [],       # 指标方法设置。
    tf_config               = {},       # tflib.init_tf()相关设置。
    G_smoothing_kimg        = 10.0,     # 生成器权重的运行平均值的半衰期。
    D_repeats               = 1,        # G每迭代一次训练判别器多少次。
    minibatch_repeats       = 4,        # 调整训练参数前要运行的minibatch的数量。
    reset_opt_for_new_lod   = True,     # 引入新层时是否重置优化器内部状态（例如Adam时刻）？
    total_kimg              = 15000,    # 训练的总长度，以成千上万个真实图像为统计。
    mirror_augment          = False,    # 启用镜像增强？
    drange_net              = [-1,1],   # 将图像数据馈送到网络时使用的动态范围。
    image_snapshot_ticks    = 1,        # 多久导出一次图像快照？
    network_snapshot_ticks  = 10,       # 多久导出一次网络模型存储？
    save_tf_graph           = False,    # 在tfevents文件中包含完整的TensorFlow计算图吗？
    save_weight_histograms  = False,    # 在tfevents文件中包括权重直方图？
    resume_run_id           = None,     # 运行已有ID或载入已有网络pkl以从中恢复训练，None = 从头开始。
    resume_snapshot         = None,     # 要从哪恢复训练的快照的索引，None = 自动检测。
    resume_kimg             = 0.0,      # 在训练开始时给定当前训练进度。影响报告和训练计划。
    resume_time             = 0.0):     # 在训练开始时给定统计时间。影响报告。
```

training_loop()包含最详细的训练设置，它也是在训练过程中被实时调用的函数。training_loop()既有由 train.py 传递过来的主要设置的参数（上图前 11 行），也包含一些更精细的训练过程设置，具体内容如上图所示。其中有三个比较常用的选项（上图红线部分），第一个是 network_snapshot_ticks，它决定多久导出一次网络模型存储；第二个是 resume_run_id，它可以导入之前存储的模型以继续训练；第三个是 rusume_kimg，它可以自定义当前的训练进度和状态，一般在微调模型时被使用。

- **★training/training_loop.py 训练过程流程（line 142-276）**

训练的具体流程如下图所示，主要包括 8 个部分：初始化 dnnlib 和 TensorFlow->载入训练集->构建网络->构建计算图与优化器->设置快照图像网格->建立运行目录->训练->保存最终结果。

由于这部分代码是 StyleGAN 作者依据自身的需求编写的，学习价值不是很大，我就不再细述了，感兴趣的读者可以自行阅读掌握。

```
# 初始化dnnlib和TensorFlow。
ctx = dnnlib.RunContext(submit_config, train)
tflib.init_tf(tf_config)

# 载入训练集。
training_set = dataset.load_dataset(data_dir=config.data_dir, verbose=True, **dataset_args)

# 构建网络。
with tf.device('/gpu:0'):
    if resume_run_id is not None:
        network_pkl = misc.locate_network_pkl(resume_run_id, resume_snapshot)
        print('Loading networks from "%s"...' % network_pkl)
        G, D, Gs = misc.load_pkl(network_pkl)
    else:
        print('Constructing networks...')
        G = tflib.Network('G', num_channels=training_set.shape[0], resolution=training_set.shape[1], label_size=training_set.label_size, **G_args)
        D = tflib.Network('D', num_channels=training_set.shape[0], resolution=training_set.shape[1], label_size=training_set.label_size, **D_args)
        Gs = G.clone('Gs')
G.print_layers(); D.print_layers()
# 构建计算图与优化器
print('Building TensorFlow graph...')
with tf.name_scope('Inputs'), tf.device('/cpu:0'):
    lod_in          = tf.placeholder(tf.float32, name='lod_in', shape=[])
    lrate_in        = tf.placeholder(tf.float32, name='lrate_in', shape=[])
    minibatch_in    = tf.placeholder(tf.int32, name='minibatch_in', shape=[])
    minibatch_split = minibatch_in // submit_config.num_gpus
    Gs_beta         = 0.5 ** tf.div(tf.cast(minibatch_in, tf.float32), G_smoothing_kimg * 1000.0) if G_smoothing_kimg > 0.0 else 0.0

G_opt = tflib.Optimizer(name='TrainG', learning_rate=lrate_in, **G_opt_args)
D_opt = tflib.Optimizer(name='TrainD', learning_rate=lrate_in, **D_opt_args)
for gpu in range(submit_config.num_gpus):
    with tf.name_scope('GPU%d' % gpu), tf.device('/gpu:%d' % gpu):
        G_gpu = G if gpu == 0 else G.clone(G.name + '_shadow')
        D_gpu = D if gpu == 0 else D.clone(D.name + '_shadow')
        lod_assign_ops = [tf.assign(G_gpu.find_var('lod'), lod_in), tf.assign(D_gpu.find_var('lod'), lod_in)]
        reals, labels = training_set.get_minibatch_tf()
        reals = process_reals(reals, lod_in, mirror_augment, training_set.dynamic_range, drange_net)
        with tf.name_scope('G_loss'), tf.control_dependencies(lod_assign_ops):
            G_loss = dnnlib.util.call_func_by_name(G=G_gpu, D=D_gpu, opt=G_opt, training_set=training_set, minibatch_size=minibatch_split, **G_loss_args)
        with tf.name_scope('D_loss'), tf.control_dependencies(lod_assign_ops):
            D_loss = dnnlib.util.call_func_by_name(G=G_gpu, D=D_gpu, opt=D_opt, training_set=training_set, minibatch_size=minibatch_split, reals=reals, labels=labels, **D_loss_args)
        G_opt.register_gradients(tf.reduce_mean(G_loss), G_gpu.trainables)
        D_opt.register_gradients(tf.reduce_mean(D_loss), D_gpu.trainables)
G_train_op = G_opt.apply_updates()
D_train_op = D_opt.apply_updates()

Gs_update_op = Gs.setup_as_moving_average_of(G, beta=Gs_beta)
with tf.device('/gpu:0'):
    try:
        peak_gpu_mem_op = tf.contrib.memory_stats.MaxBytesInUse()
    except tf.errors.NotFoundError:
        peak_gpu_mem_op = tf.constant(0)
# 设置快照图像网格
print('Setting up snapshot image grid...')
grid_size, grid_reals, grid_labels, grid_latents = misc.setup_snapshot_image_grid(G, training_set, **grid_args)
sched = training_schedule(cur_nimg=total_kimg*1000, training_set=training_set, num_gpus=submit_config.num_gpus, **sched_args)
grid_fakes = Gs.run(grid_latents, grid_labels, is_validation=True, minibatch_size=sched.minibatch//submit_config.num_gpus)
# 建立运行目录
print('Setting up run dir...')
misc.save_image_grid(grid_reals, os.path.join(submit_config.run_dir, 'reals.png'), drange=training_set.dynamic_range, grid_size=grid_size)
misc.save_image_grid(grid_fakes, os.path.join(submit_config.run_dir, 'fakes%06d.png' % resume_kimg), drange=drange_net, grid_size=grid_size)
summary_log = tf.summary.FileWriter(submit_config.run_dir)
if save_tf_graph:
    summary_log.add_graph(tf.get_default_graph())
if save_weight_histograms:
    G.setup_weight_histograms(); D.setup_weight_histograms()
metrics = metric_base.MetricGroup(metric_arg_list)
# 训练
print('Training...\n')
ctx.update('', cur_epoch=resume_kimg, max_epoch=total_kimg)
maintenance_time = ctx.get_last_update_interval()
cur_nimg = int(resume_kimg * 1000)
cur_tick = 0
tick_start_nimg = cur_nimg
prev_lod = -1.0
while cur_nimg < total_kimg * 1000:
    if ctx.should_stop(): break

    # 选择训练参数并配置训练操作。
    sched = training_schedule(cur_nimg=cur_nimg, training_set=training_set, num_gpus=submit_config.num_gpus, **sched_args)
    training_set.configure(sched.minibatch // submit_config.num_gpus, sched.lod)
    if reset_opt_for_new_lod:
        if np.floor(sched.lod) != np.floor(prev_lod) or np.ceil(sched.lod) != np.ceil(prev_lod):
            G_opt.reset_optimizer_state(); D_opt.reset_optimizer_state()
    prev_lod = sched.lod

    # 进行训练。
    for _mb_repeat in range(minibatch_repeats):
        for _D_repeat in range(D_repeats):
            tflib.run([D_train_op, Gs_update_op], {lod_in: sched.lod, lrate_in: sched.D_lrate, minibatch_in: sched.minibatch})
            cur_nimg += sched.minibatch
        tflib.run([G_train_op], {lod_in: sched.lod, lrate_in: sched.G_lrate, minibatch_in: sched.minibatch})

    # 每个tick执行一次维护任务。
    done = (cur_nimg >= total_kimg * 1000)
    if cur_nimg >= tick_start_nimg + sched.tick_kimg * 1000 or done:
        cur_tick += 1
        tick_kimg = (cur_nimg - tick_start_nimg) / 1000.0
        tick_start_nimg = cur_nimg
        tick_time = ctx.get_time_since_last_update()
        total_time = ctx.get_time_since_start() + resume_time

        # 报告进度。
        print('tick %-5d kimg %-8.1f lod %-5.2f minibatch %-4d time %-12s sec/tick %-7.1f sec/kimg %-7.2f maintenance %-6.1f gpumem %-4.1f' % (
            autosummary('Progress/tick', cur_tick),
            autosummary('Progress/kimg', cur_nimg / 1000.0),
            autosummary('Progress/lod', sched.lod),
            autosummary('Progress/minibatch', sched.minibatch),
            dnnlib.util.format_time(autosummary('Timing/total_sec', total_time)),
            autosummary('Timing/sec_per_tick', tick_time),
            autosummary('Timing/sec_per_kimg', tick_time / tick_kimg),
            autosummary('Timing/maintenance_sec', maintenance_time),
            autosummary('Resources/peak_gpu_mem_gb', peak_gpu_mem_op.eval() / 2**30)))
        autosummary('Timing/total_hours', total_time / (60.0 * 60.0))
        autosummary('Timing/total_days', total_time / (24.0 * 60.0 * 60.0))

        # 保存快照。
        if cur_tick % image_snapshot_ticks == 0 or done:
            grid_fakes = Gs.run(grid_latents, grid_labels, is_validation=True, minibatch_size=sched.minibatch//submit_config.num_gpus)
            misc.save_image_grid(grid_fakes, os.path.join(submit_config.run_dir, 'fakes%06d.png' % (cur_nimg // 1000)), drange=drange_net, grid_size=grid_size)
        if cur_tick % network_snapshot_ticks == 0 or done or cur_tick == 1:
            pkl = os.path.join(submit_config.run_dir, 'network-snapshot-%06d.pkl' % (cur_nimg // 1000))
            misc.save_pkl((G, D, Gs), pkl)
            metrics.run(pkl, run_dir=submit_config.run_dir, num_gpus=submit_config.num_gpus, tf_config=tf_config)

        # 更新摘要和RunContext。
        metrics.update_autosummaries()
        tflib.autosummary.save_summaries(summary_log, cur_nimg)
        ctx.update('%.2f' % sched.lod, cur_epoch=cur_nimg // 1000, max_epoch=total_kimg)
        maintenance_time = ctx.get_last_update_interval() - tick_time

# 保存最终结果。
misc.save_pkl((G, D, Gs), os.path.join(submit_config.run_dir, 'network-final.pkl'))
summary_log.close()

ctx.close()
```

第三章 StyleGAN 模型修改与拓展

前两章介绍了很多关于 StyleGAN 的模型原理与实现细节，而通常我们在自己去应用新的模型时需要对它做一些修改与拓展，以实现个性化的功能。这一章将记录和介绍一些我学到的部分 trick，虽然不难，但能够帮助我们在使用 StyleGAN 时拥有更加灵活的手段和方法。当然，更重要的应该是先有想法，然后再找 trick 去实现这些想法。我的一些想法被记录在了网站：www.seeprettyface.com/research_notes.html 中，虽然都比较简单，但是这种实践对提升自己的工程能力还是有一定帮助的。

3.1 如何修改 StyleGAN 架构

在第二章中已经对 StyleGAN 的网络做了细致的介绍，如果对 tensorflow 掌握的熟练的话，tf.nn 下有一大堆好用（但不好记。。。）的网络层或结构，依据自己的需要去对原始架构做删改或增添就可以了。

还有一种应用方法是将 StyleGAN 的一些组件扒出来用在别的模型架构上，目前 AdaIN 在生成模型中的应用是非常广泛的。我的感觉，当被学习的图片服从一个很强的集中分布模式时（譬如人脸图，猫图，Waifu 图等），AdaIN 就特别适合应用在这种模型上。

AdaIN 还有一种妙用，就是用来做样式迁移（Style Transfer）。举个例子，在我的 Github 上展示的所有生成器，如黄种人生成器、网红脸生成器、明星脸生成器、超模脸生成器以及萌娃脸生成器等，都是用样式迁移（代码上基于 Finetune 编写）制作出来的，即当新数据与原数据服从同一种分布但仅具有样式的不同时，就可以基于 AdaIN 实现这样的风格转换。AdaIN 实现样式迁移的效率非常高，本站提供的所有人脸生成器都是用单卡 1080Ti 制作的，训练时间不超过 1 天。

3.2 如何拓展 StyleGAN 组件

通常来说，StyleGAN 最有价值的部分是生成器，它可以提供各种各样的生成图片给我们。但有时我们希望生成的图片能被定制，或者有所筛选，于是可以在生成器初始端添加一个 latents 的处理模块，或者在生成器末端添加一个图片的筛选模块等。

假设我们现在想给 latents 添加一个线性变换矩阵以实现 latents 的微调，就可以构建一个 create_variable_for_generator()函数来创建一个对 latents 进行线性变换的模块：

```
def create_variable_for_generator(name, initial_dlatent, batch_size, model_scale=18):

    # 添加属性变换矩阵
    xs = np.zeros([17, 18, 512], dtype=np.float32)
    xs[0] = np.load('latent_directions/angle_horizontal.npy')
    xs[1] = np.load('latent_directions/angle_vertical.npy')
    xs[2] = np.load('latent_directions/height.npy')
    xs[3] = np.load('latent_directions/width.npy')
    xs[4] = np.load('latent_directions/smile.npy')
    xs[5] = np.load('latent_directions/face_shape.npy')
    xs[6] = np.load('latent_directions/gender.npy')
    xs[7] = np.load('latent_directions/eyes_open.npy')

    xs[8] = np.load('latent_directions/age.npy')
    xs[9] = np.load('latent_directions/beauty.npy')
    xs[10] = np.load('latent_directions/emotion_angry.npy')
    xs[11] = np.load('latent_directions/emotion_disgust.npy')
    xs[12] = np.load('latent_directions/emotion_easy.npy')
    xs[13] = np.load('latent_directions/emotion_fear.npy')
    xs[14] = np.load('latent_directions/emotion_happy.npy')
    xs[15] = np.load('latent_directions/emotion_sad.npy')
    xs[16] = np.load('latent_directions/emotion_surprise.npy')

    # 属性编辑向量的权重
    weights = [1.0, 1.0, 0.02, 0.02, 1.0, 0, 0, 0.2, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]

    x = tf.constant(xs, dtype=tf.float32, shape=(17, model_scale, 512))
    w = tf.get_variable('learnable_w', shape=(17), dtype='float32', initializer=tf.zeros_initializer())
    y = tf.get_variable('dlatent_variable', shape=(batch_size, model_scale, 512), dtype='float32', initializer=tf.constant_initializer(initial_dlatent))
    for i in range(17):
        y = y + weights[i] * w[i] * x[i]

    return y
```

上述函数创建了一个模块，在给 initial_dlatents 添加了 17 个编辑向量的处理后，得到新的 latent 作为返回值。如果想要将这个模块融入进 Generator 中，只需在创建 Generator 时将其写在 custom_inputs 参数中即可。

```
class Generator:
    def __init__(self, model, batch_size, randomize_noise=False):
        self.batch_size = batch_size

        self.initial_dlatents = np.zeros((self.batch_size, 18, 512))
        model.components.synthesis.run(self.initial_dlatents,
                                       randomize_noise=randomize_noise, minibatch_size=self.batch_size,
                                       custom_inputs=[partial(create_variable_for_generator, batch_size=batch_size),
                                                      partial(create_stub, batch_size=batch_size)],
                                       structure='fixed')
```

上图便是在 Generator 初始化时调用了 create_variable_for_generator()函数以创建扩增后的模型输入，注意另外一个函数 create_stub()会返回一个值为 0 的 tf.constant 常量，可以理解为是生成器输入的但无效的标签。

注意，这份包含 Generator 类的代码来自官方的额外补充工作中，这一部分讲述的技巧也是从这份代码中学到的，在此特别感谢 NVIDIA 团队的无私开源与分享。

3.3 如何指定仅拓展组件的参数更新而固定 Generator 参数

当添加了新的组件后，训练策略也会有一些改变。有时我们会希望 Generator 的参数固定不动，模型仅去更新新添加组件的参数。一种简单的方法是，optimizer.minimize()函数中有一个可选参数 var_list，它可以指定需要更新的参数有哪些（如下图所示）。

```
vars_to_optimize = vars_to_optimize if isinstance(vars_to_optimize, list) else [vars_to_optimize]
optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
min_op = optimizer.minimize(self.loss, var_list=[vars_to_optimize])
self.sess.run(tf.variables_initializer(optimizer.variables()))
```

因此，我们只需将指定组件的参数作为 var_list，然后交给 optimizer 去更新就可以了。

3.4 拓展组件无法梯度反向传播的解决方法

有时我们在模型后端加上了类别分类器或者姿态提取器，但是通常它们有可能不是用 tensorflow 写的，这样在反向传播时这些模块的梯度就无法计算，训练就会出问题。一种非常简单的解决方法是使用@tf.custom_gradient 装饰器为这些模块自定义一阶导或二阶导，类似于下面这种写法：

```
def blur2d(x, f=[1,2,1], normalize=True):
    with tf.variable_scope('Blur2D'):
        @tf.custom_gradient
        def func(x):  # 定义一个函数，返回模糊后的特征值以及该函数的梯度计算式grad()
            y = _blur2d(x, f, normalize)  # 模糊后的特征值y
            @tf.custom_gradient
            def grad(dy):  # func函数的梯度计算式
                dx = _blur2d(dy, f, normalize, flip=True)  # d_func(x)/d_x=_blur2d(d_x)
                return dx, lambda ddx: _blur2d(ddx, f, normalize)
                # func的梯度(即grad)用_blur2d()去近似，grad的梯度也用_blur2d()去近似(对于一阶导和二阶导都用_blur2d()作近似，可能是因为_blur2d()函数的结果(模糊)等效为目的(变模糊))。
            return y, grad
        return func(x)
```

3.5 StyleGAN 衍生论文介绍

StyleGAN 的应用和商业潜力是非常大的，最后推荐两篇基于 StyleGAN 的偏应用方向的论文：

3.5.1 StyleGAN-Embedder

这篇 paper 主要关于如何用 StyleGAN 做图像编码、图像变换与风格融合。

- Arxiv 地址：https://arxiv.org/pdf/1904.03189.pdf
- 译文地址：http://www.gwylab.com/pdf/stylegan-embedder_chs.pdf

3.5.2 StyleGAN-Wearing

这篇 paper 主要关于如何用 StyleGAN 做换装应用。

- Arxiv 地址：https://arxiv.org/pdf/1908.08847.pdf
- 译文地址：http://www.gwylab.com/pdf/stylegan-wearing_chs.pdf

· 致谢及引用

1. 第一章 图片来源

第一章原理介绍的图片引用自下列网站：
https://towardsdatascience.com/explained-a-style-based-generator-architecture-for-gans-generating-and-tuning-realistic-6cb2be0f431?spm=a2c4e.11153940.blogcont686373.15.99143440XmYsTe

本资料仅用来学习，请不要用于商业用途。