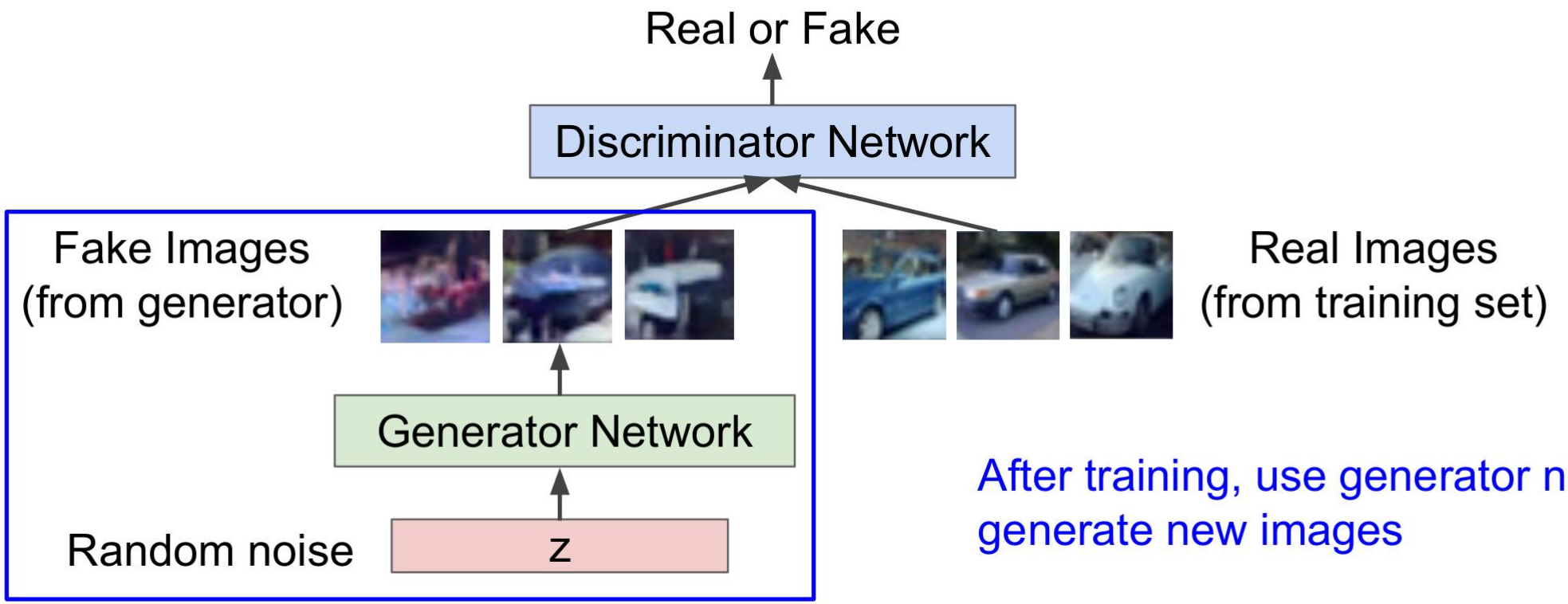
Training GANs: Two-player game

Ian Goodfellow et al., “Generat
Adversarial Nets”, NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

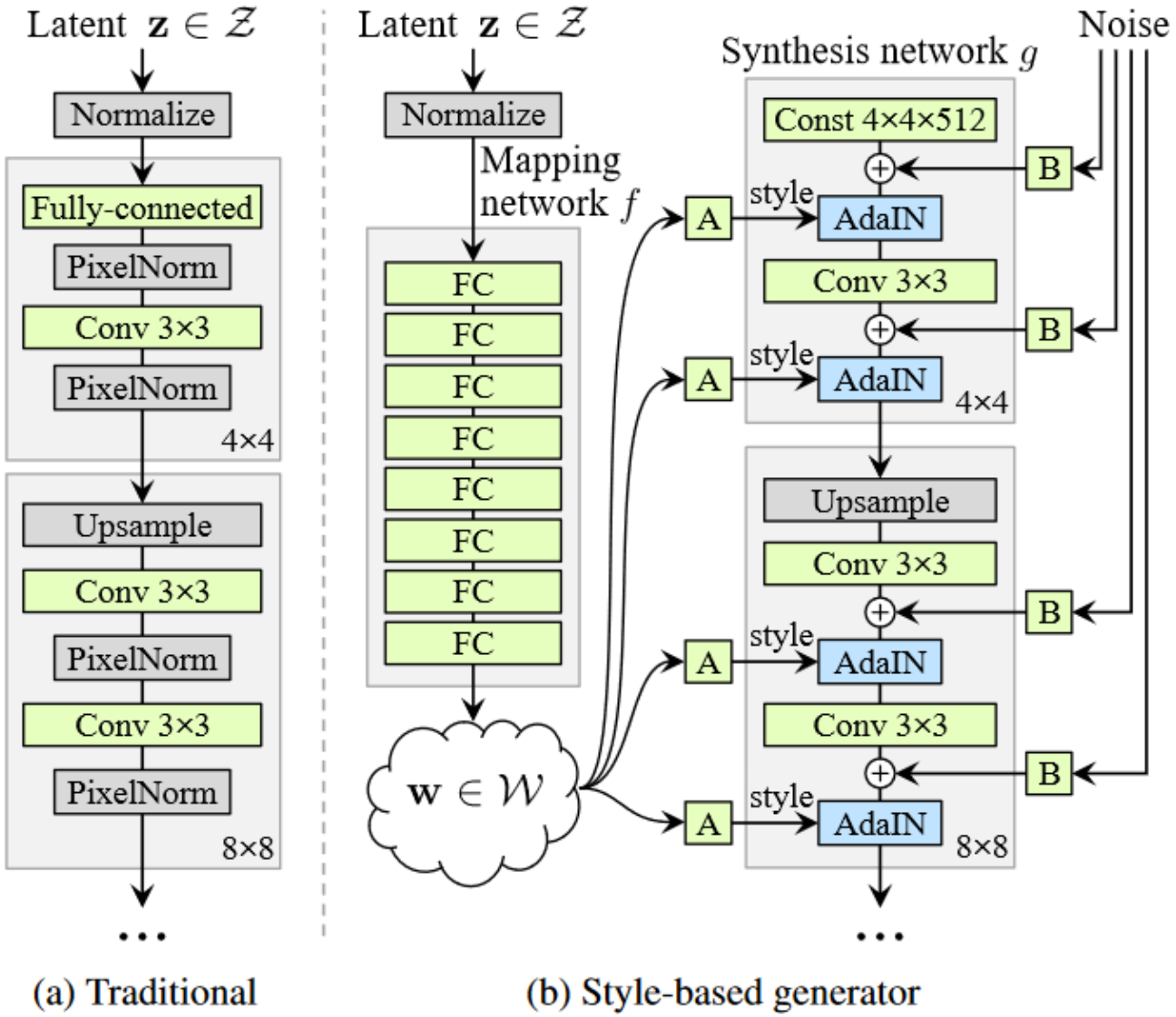


After training, use generator network to generate new images

StyleGAN中的“Style”是指数据集中人脸的主要属性，比如人物的姿态等信息，包括了脸型上面的表情、人脸朝向、发型等等，还包括纹理细节上的人脸肤色、人脸光照等方方面面。

StyleGAN 用风格（style）来影响人脸的姿态、身份特征等，用噪声（noise）来影响头发丝、皱纹、肤色等细节部分

模型架构



当传统的生成器只通过输入层输入latent z时，首先将输入映射到一个中间潜在空间W，然后在每个卷积层通过自适应实例标准化(AdaIN)控制生成器。高斯噪声是在每次卷积之后，在进行非线性变化之前添加的。其中，‘ A ’表示学习到的仿射变换，‘ B ’将学习到的每通道缩放因子应用于噪声输入。 **Mapping network** f由8层组成，**Synthesis network** g由18层组成，每个分辨率(4^2~ 1024^2)有2层。最后一层的输出使用单独的1 × 1卷积转换为RGB。

- **Mapping network : 用于将 latent code z 转换成为 w （利用w来实现style的作用）**
- **Synthesis network : 用于生成图像**

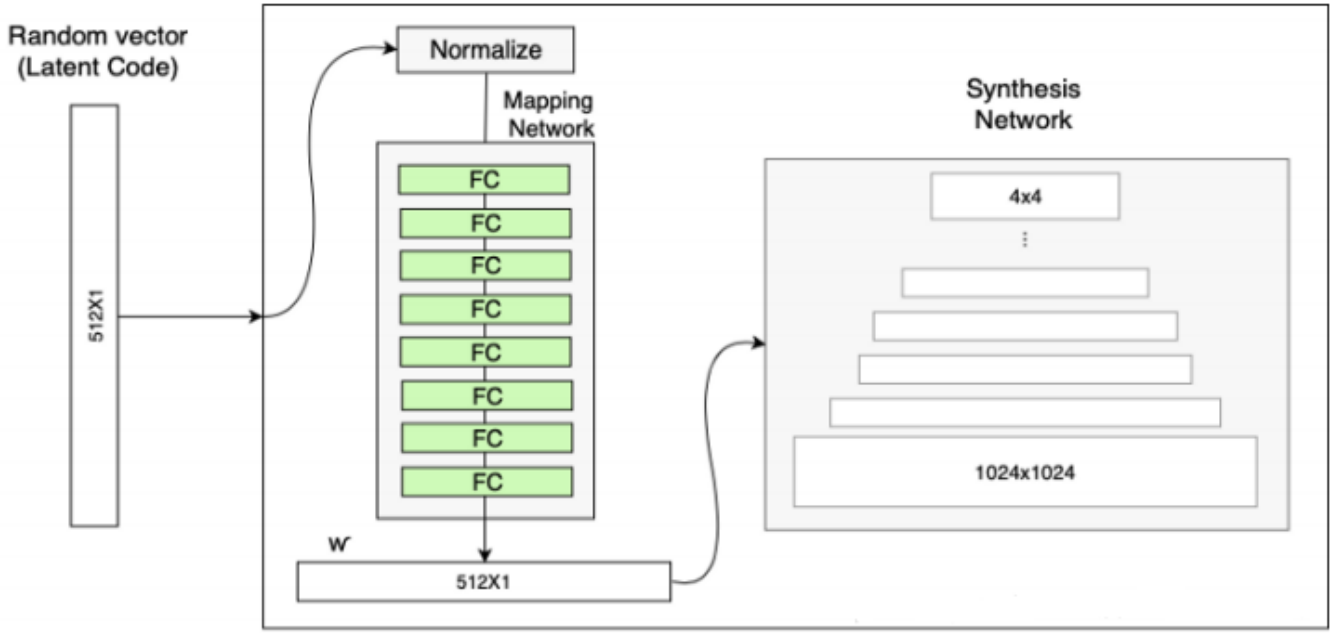
Mapping network

作用：为输入向量z的特征解耦提供一条学习通道

由于z是符合均匀分布或者高斯分布的随机变量，所以变量之间的耦合性比较大。

比如特征：头发长度和男子气概，如果按照z的分布来说，那么这两个特征之间就会存在交缠紧密的联系，头发短了你的男子气概会降低或者增加，但其实现实情况来说，短发男子、长发男子都可以有很强的男子气概。

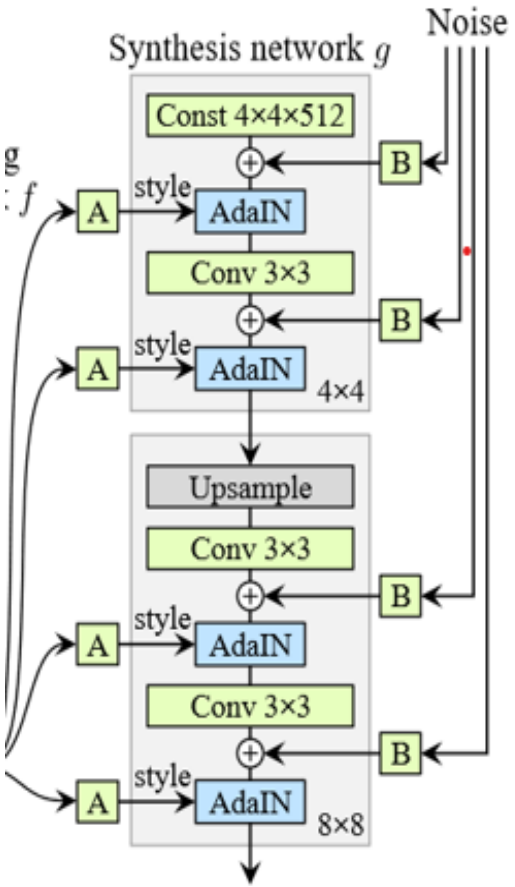
需要将latent code z进行解耦，才能更好的后续操作，来改变其不同特征。




```python
class G_mapping(nn.Module):
    """映射网络"""
    def __init__(self):
        super().__init__()
        self.func = nn.Sequential(
            nn.Linear(512,512,bias=True),
            nn.LeakyReLU(512,True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

            nn.Linear(512, 512, bias=True),
            nn.LeakyReLU(512, True),

        )
        self.pixel_norm = PixelNorm()

        for m in self.modules():
            if isinstance(m, nn.Linear):
                m.weight.data.normal_(0,1)
                m.bias.data.zero_()

    def forward(self, x):
        x = self.pixel_norm(x)
        out = self.func(x)
        return out
```

Synthesis block

$$\mathrm{AdaIN}(x_i, y) = \sigma(y)\left(\frac{x_i - \mu(x_i)}{\sigma(x_i)}\right) + \mu(y)$$



up_sample->noise->adaIN->conv->noise->adaIN为一个block

```
x = self.const_input  # 可学习常量Noise [1,512,4,4]
x = x + self.bias.view(1, -1, 1, 1)  # 加上偏执

x = self.addNoise_01(x, self.noise_inputs[0])
x = self.adaIN_01(x, dlatent[:, 0])

x = self.conv_0(x)

x = self.addNoise_02(x, self.noise_inputs[1])
x = self.adaIN_02(x,  dlatent[:, 1])

x = self.up_1(x)

x = self.addNoise_11(x, self.noise_inputs[2])
x = self.adaIN_11(x, dlatent[:, 1])

x = self.conv_1(x)

x = self.addNoise_12(x, self.noise_inputs[3])
x = self.adaIN_12(x, dlatent[:, 3])
```

```
def __init__(self):
    super().__init__()
    # 噪声初始化
    self.noise_inputs = []
    self.noise_inputs.append(torch.randn((1,1,4,4)))
    self.noise_inputs.append(torch.randn((1,1,4,4)))
    self.noise_inputs.append(torch.randn((1,1,8,8)))
    self.noise_inputs.append(torch.randn((1,1,8,8)))
    self.noise_inputs.append(torch.randn((1,1,16,16)))
    self.noise_inputs.append(torch.randn((1,1,16,16)))
```

noise的维度在模型初始化的时候就已经计算好

每个阶段把上采样或者卷积过后的x直接加上对应尺寸大小和维度的noise

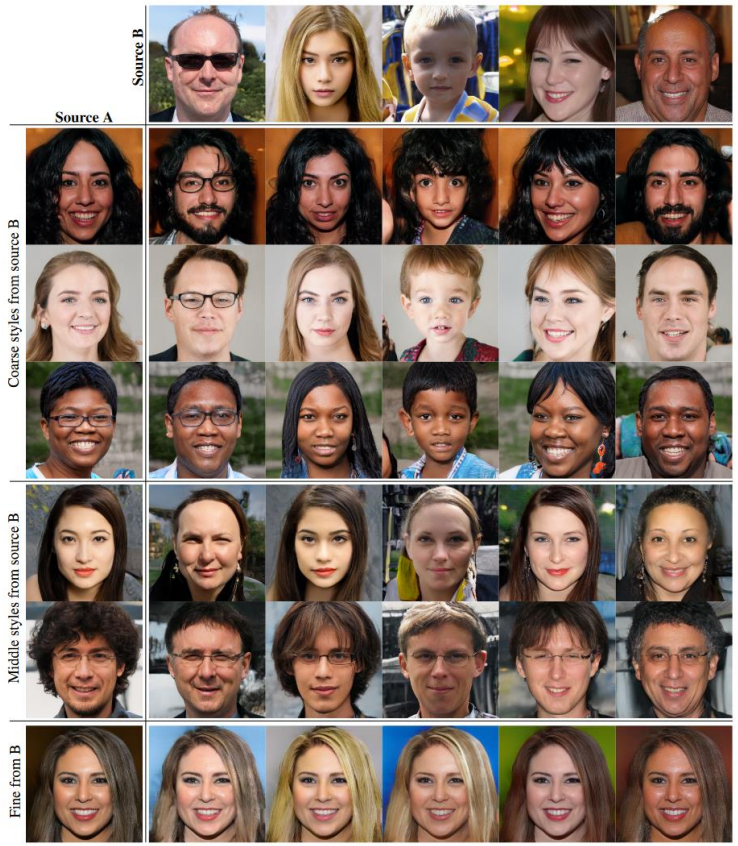
```
x = self.addNoise_01(x, self.noise_inputs[0])
```

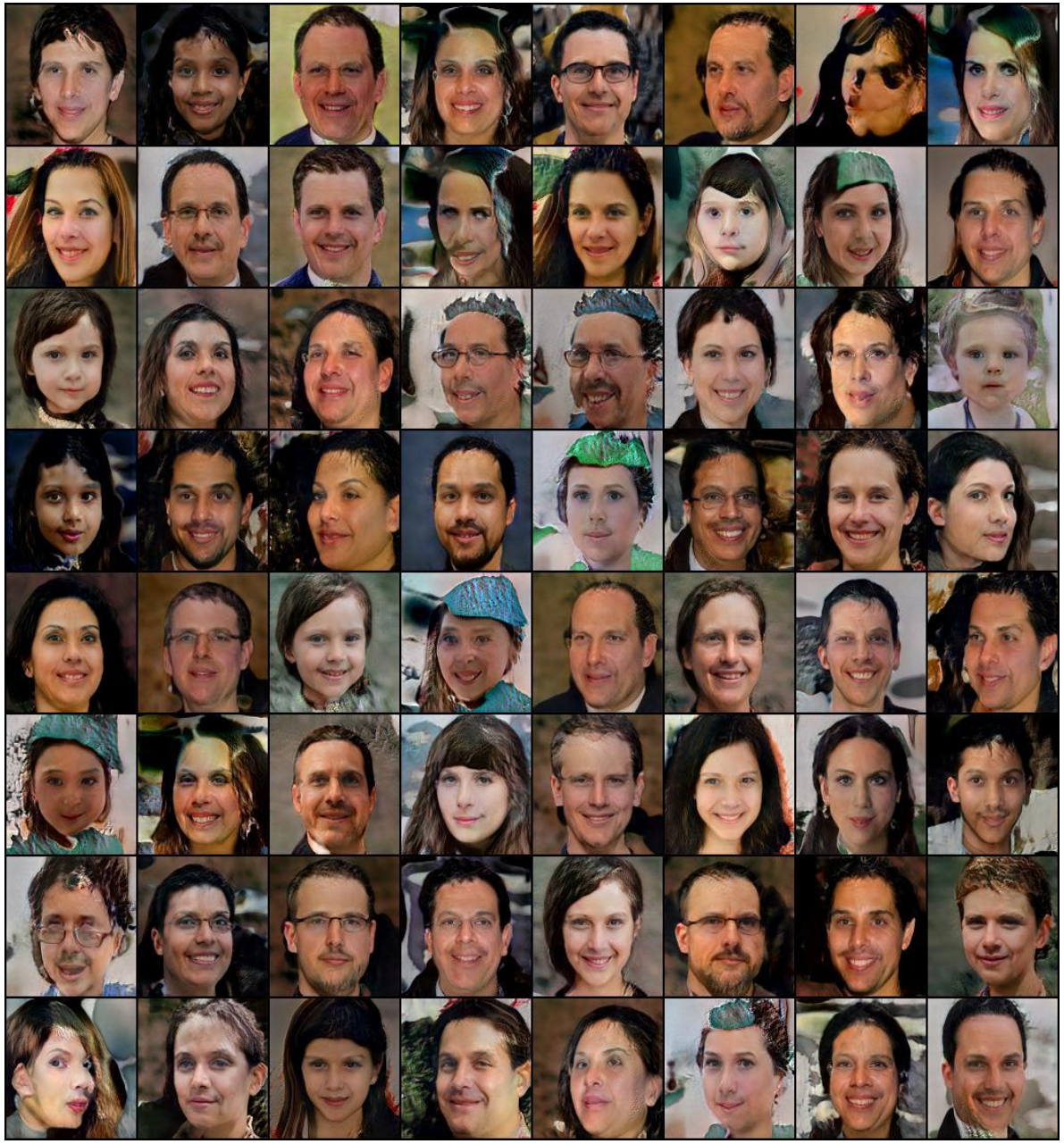
```
class ApplyNoise(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.weight = nn.Parameter(torch.zeros(channels))

    def forward(self, x, noise):
        if noise is None:
            noise = torch.randn(x.size(0), 1, x.size(2), x.size(3), device=x.device, dtype=x.dtype)
        return x + self.weight.view(1, -1, 1, 1) * noise
```


做到可学习是因为把noise和可学习参数相乘之后再加上x的

```python
dlatents1, num_layers = self.mapping(latents1)

dlatents1 = dlatents1.unsqueeze(1)
dlatents1 = dlatents1.expand(-1, int(num_layers), -1)

# Add mixing style mechanism.
with torch.no_grad():
    latents2 = torch.randn(latents1.shape).to(latents1.device)
    dlatents2, num_layers = self.mapping(latents2)
    dlatents2 = dlatents2.unsqueeze(1)
    dlatents2 = dlatents2.expand(-1, int(num_layers), -1)
    cur_layers = num_layers
    mix_layers = num_layers
    if np.random.random() < self.style_mixing_prob:
        mix_layers = np.random.randint(1, cur_layers)
    for i in range(num_layers):
        if i >= mix_layers:
            dlatents1[:, i, :] = dlatents2[:, i, :]
```

作者在训练的时候用了两个隐变量z1、z2产生w1、w2，用于不同阶段的Style。这样做的好处就是能如果只使用一个w，那么网络中各个阶段的样式可能会产生关联(因为AdaIN的参数就是这个w的仿射变换结果)，为了防止这种情况作者使用了两个w拼接的方法。





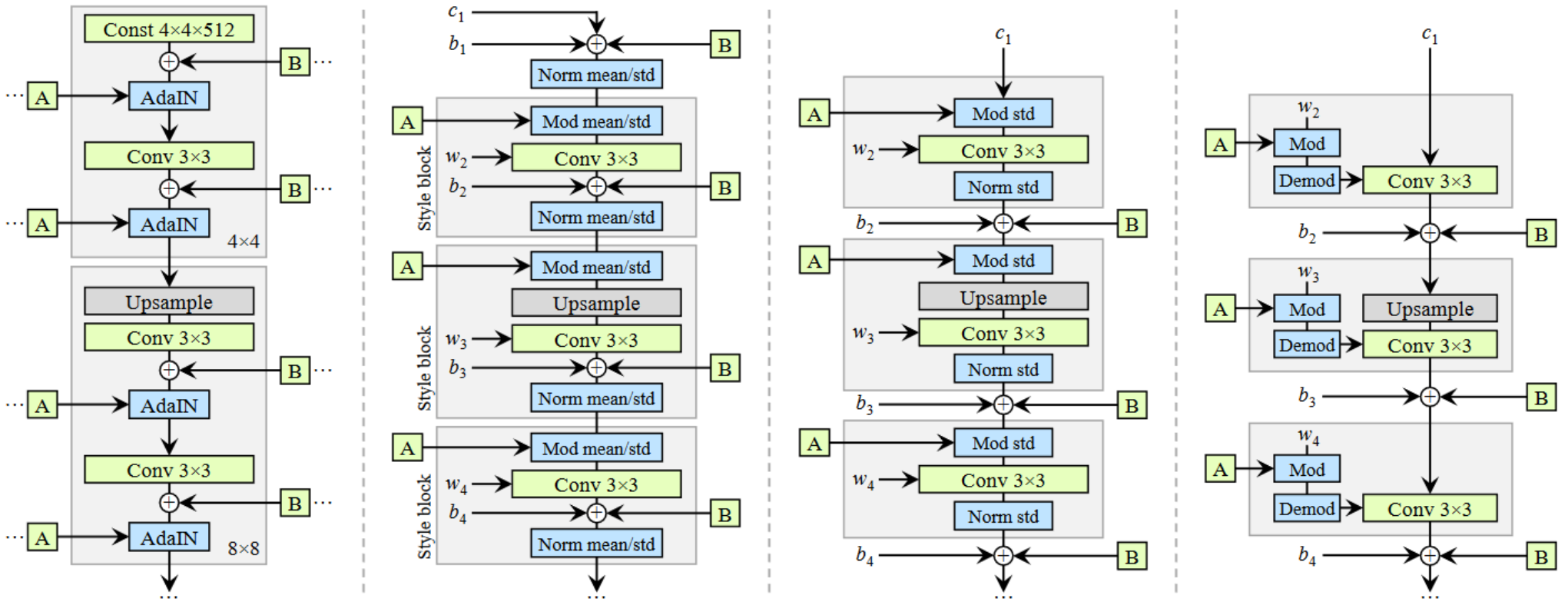
水滴状伪影 （blob-shaped artifacts）

作者将问题归结为AdaIN操作，该操作分别对每个特征图的均值和方差进行归一化处理，从而可能破坏在特征相对大小中发现的任何信息。我们假设液滴伪影是生成器故意通过实例归一化来隐藏信号强度信息的结果：通过创建一个强大的、局部的、主导统计的尖峰，生成器可以像在其他地方一样有效地缩放信号。假设得到了以下发现的支持：当标准化步骤从生成器中移除时，水滴状伪影完全消失。

同一特征在不同分辨率的特征图上都有所表现，在从4x4向1024x1024生长的过程中，该特征在不同层级间彼此有前后继承或关联关系，而AdaIN操作破坏了这种继承或关联关系

StyleGAN生成图像的特征信息在不同分辨率的特征图之间逐级传递时，把液滴伪影作为一个核心特征，该核心特征统领其他面部特征从4x4向1024x1024逐级进化和精细化。由于液滴伪影的信号强度足够强大，一般的面部特征的信号表达被压制，因而面部特征的细节可以悄悄地越过归一化操作而到达下一级特征图。

Figure 15. An example of the importance of the droplet artifact in StyleGAN generator. We compare two generated images, one successful and one severely corrupted. The corresponding feature maps were normalized to the viewable dynamic range using instance normalization. For the top image, the droplet artifact starts forming in 64^2 resolution, is clearly visible in 128^2, and increasingly dominates the feature maps in higher resolutions. For the bottom image, 64^2 is qualitatively similar to the top row, but the droplet does not materialize in 128^2. Consequently, the facial features are stronger in the normalized feature map. This leads to an overshoot in 256^2, followed by multiple spurious droplets forming in subsequent resolutions. Based on our experience, it is rare that the droplet is missing from StyleGAN images, and indeed the generator fully relies on its existence.

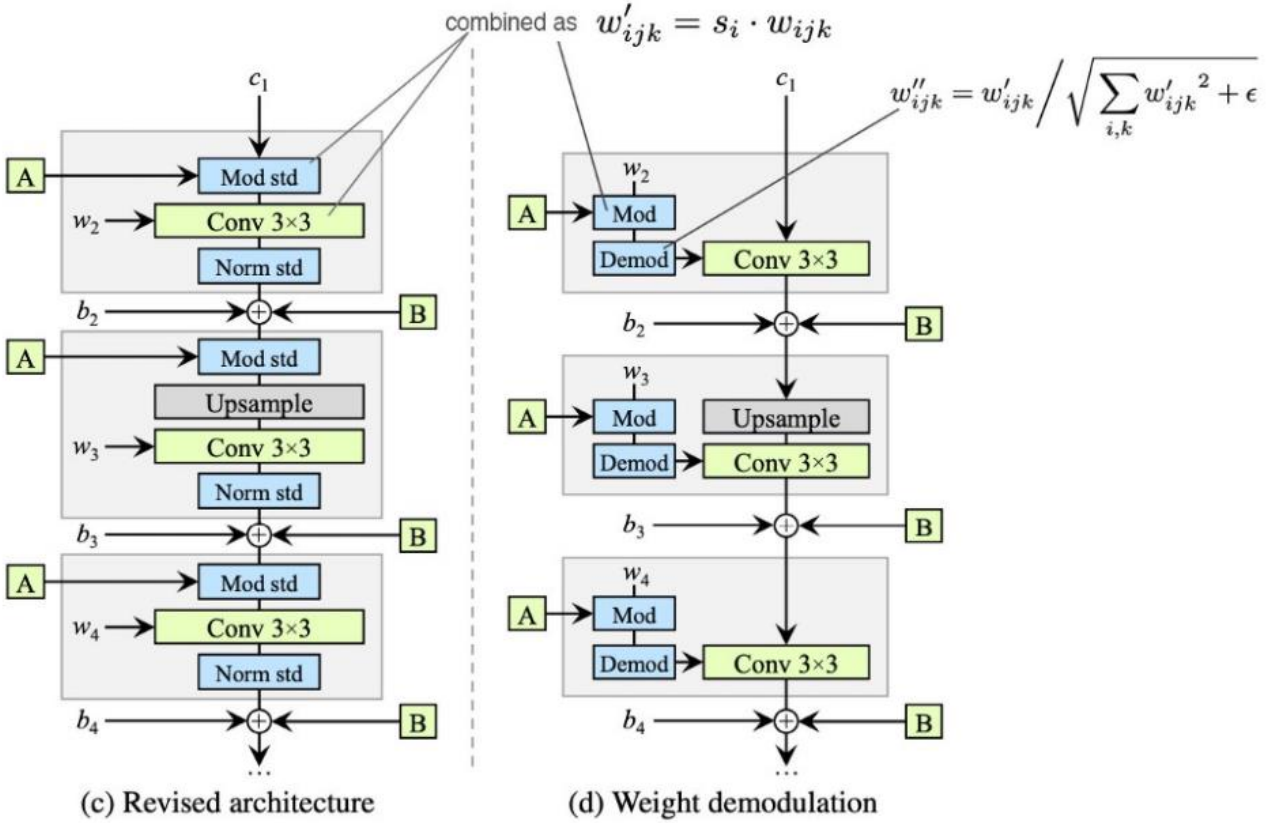


（a）初始StyleGAN，其中A表示由W学习得到的产生某种风格的仿射变换，B为噪声广播操作。
（b）具有完整细节的同一图。在这里，我们打破了AdaIN的显式标准化，然后进行调制，两者都在每个特征图的均值和标准差上操作。我们还对学习到的权重（w）、偏差（b）和常量输入（c）进行了标注，并重新绘制了灰色方框，使得每个方框有一个样式是激活的。激活函数(泄漏ReLU)总是在添加偏置后立即应用。
（c）我们对原架构作了几处改动，这些改动在主要案文中是合理的。我们在开始时去除一些冗余操作，将b和B的添加移动到一个样式的外部活动区域，并且只调整每个特征图的标准差。
（d）修改后的架构能够用“解调”操作代替实例归一化，将其应用于与每个卷积层相关的权重。

- **移除最开始的数据处理**
- **在标准化特征时取消均值**
- **将noise模块在外部style模块添加**

weight demodulation

在样式混合（style mixing）中，容易将某个特征放大一个数量级或更多，而在去除了Adain后，便无法有效控制这个问题（因为移除了mean），但style mixing又是stylegan的亮点，如何能够在保留style mixing的同时有效地解决特征放大问题呢？这便是weight demodulation设计的原因。



实例归一化（AdaIN）的目的是从卷积输出特征图的统计中本质上去除s的影响。

通过相应权重的L2范数对输出进行缩放。随后的归一化旨在将输出恢复到单位标准差。如果我们将("解调")每个输出特征映射j缩放1 / σ j，就可以实现这一点。

```python
class Conv2DMod(nn.Module):
    def __init__(self, in_chan, out_chan, kernel, demod=True, stride=1, dilation=1, eps = 1e-8, **kwargs):
        super().__init__()
        self.filters = out_chan
        self.demod = demod
        self.kernel = kernel
        self.stride = stride
        self.dilation = dilation
        self.weight = nn.Parameter(torch.randn((out_chan, in_chan, kernel, kernel)))
        self.eps = eps
        nn.init.kaiming_normal_(self.weight, a=0, mode='fan_in', nonlinearity='leaky_relu')

    def _get_same_padding(self, size, kernel, dilation, stride):
        return ((size - 1) * (stride - 1) + dilation * (kernel - 1)) // 2

    def forward(self, x, y):
        b, c, h, w = x.shape

        w1 = y[:, None, :, None, None]
        w2 = self.weight[None, :, :, :, :]
        weights = w2 * (w1 + 1)

        if self.demod:
            d = torch.rsqrt((weights ** 2).sum(dim=(2, 3, 4), keepdim=True) + self.eps)
            weights = weights * d

        x = x.reshape(1, -1, h, w)

        _, _, *ws = weights.shape
        weights = weights.reshape(b * self.filters, *ws)

        padding = self._get_same_padding(h, self.kernel, self.dilation, self.stride)
        x = F.conv2d(x, weights, padding=padding, groups=b)

        x = x.reshape(-1, self.filters, h, w)
        return x
```

基于输入的样式，调制操作对卷积的每个输入特征图的尺度进行调整，这可以通过调整卷积权重的尺度而替代性地予以实现

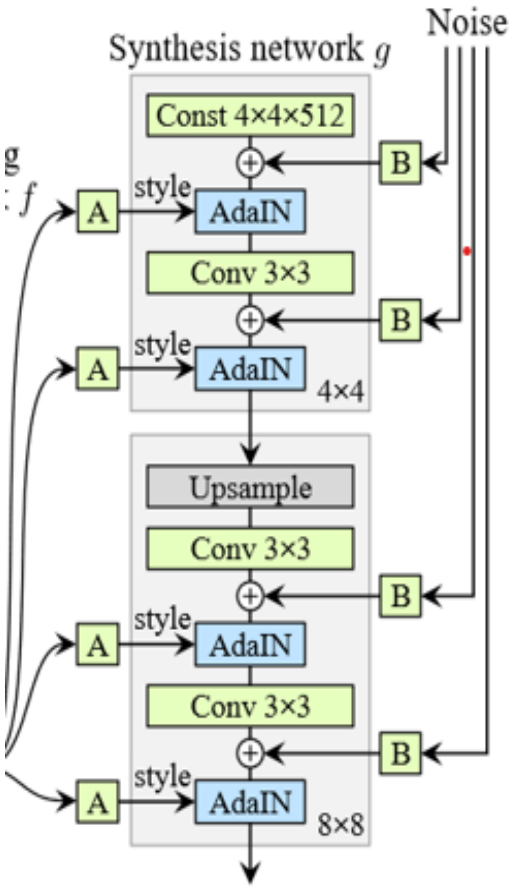
$$w'_{ijk} = s_i \cdot w_{ijk},$$

$$w''_{ijk} = w'_{ijk} \Big/ \sqrt{\sum_{i,k} {w'_{ijk}}^2 + \epsilon},$$

输出的尺度用对应权重的L_2范式进行调整。随后的归一化操作，其目的是将输出复原到单位标准差。

Synthesis block

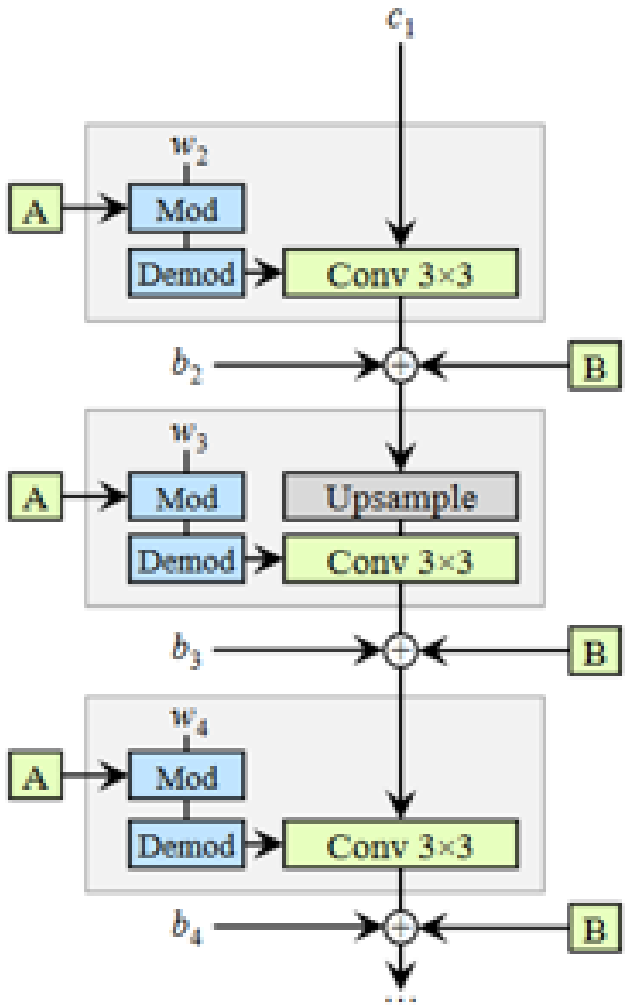
$$\mathrm{AdaIN}(x_i, y) = \sigma(y)\left(\frac{x_i - \mu(x_i)}{\sigma(x_i)}\right) + \mu(y)$$



up_sample->noise->adaIN->conv->noise->adaIN为一个block

```
x = self.const_input  # 可学习常量Noise [1,512,4,4]
x = x + self.bias.view(1, -1, 1, 1)  # 加上偏执

x = self.addNoise_01(x, self.noise_inputs[0])
x = self.adaIN_01(x, dlatent[:, 0])

x = self.conv_0(x)

x = self.addNoise_02(x, self.noise_inputs[1])
x = self.adaIN_02(x,  dlatent[:, 1])

x = self.up_1(x)

x = self.addNoise_11(x, self.noise_inputs[2])
x = self.adaIN_11(x, dlatent[:, 1])

x = self.conv_1(x)

x = self.addNoise_12(x, self.noise_inputs[3])
x = self.adaIN_12(x, dlatent[:, 3])
```

```python
class GeneratorBlock(nn.Module):
    def __init__(self, latent_dim, input_channels, filters, upsample = True, upsample_rgb = True, rgba = False):
        super().__init__()
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=False) if upsample else None

        self.to_style1 = nn.Linear(latent_dim, input_channels)
        self.to_noise1 = nn.Linear(1, filters)
        self.conv1 = Conv2DMod(input_channels, filters, 3)

        self.to_style2 = nn.Linear(latent_dim, filters)
        self.to_noise2 = nn.Linear(1, filters)
        self.conv2 = Conv2DMod(filters, filters, 3)

        self.activation = leaky_relu()
        self.to_rgb = RGBBlock(latent_dim, filters, upsample_rgb, rgba)

    def forward(self, x, prev_rgb, istyle, inoise):
        if exists(self.upsample):
            x = self.upsample(x)

        inoise = inoise[:, :x.shape[2], :x.shape[3], :]
        noise1 = self.to_noise1(inoise).permute((0, 3, 2, 1))
        noise2 = self.to_noise2(inoise).permute((0, 3, 2, 1))

        style1 = self.to_style1(istyle)
        x = self.conv1(x, style1)
        x = self.activation(x + noise1)

        style2 = self.to_style2(istyle)
        x = self.conv2(x, style2)
        x = self.activation(x + noise2)

        rgb = self.to_rgb(x, prev_rgb, istyle)
        return x, rgb
```



(d) Weight demodulation

Path length regularization

在生成人脸的同时，我们希望能够控制人脸的属性，不同的latent code能得到不同的人脸，当确定latent code变化的具体方向时，该方向上不同的大小应该对应了图像上某一个具体变化的不同幅度。为了达到这个目的，设计了 Path length regularization。

我们希望鼓励W中的一个固定大小的步骤导致图像中的非零、固定大小的变化。我们可以通过进入图像空间中的随机方向并观察相应的w个梯度来经验地衡量偏离这个理想的程度。这些梯度不管是w还是图像空间方向，都应该具有接近相等的长度，表明从潜在空间到图像空间的映射是有条件的

在单个w∈W时，生成元映射g（w）：W →Y的局部度量标度性质由Jacobian矩阵Jw =∂g（w）/∂w捕获

$$\mathbb{E}_{\mathbf{w},\mathbf{y}\sim\mathcal{N}(0,\mathbf{I})}\left(\left\|\mathbf{J}_{\mathbf{w}}^{T}\mathbf{y}\right\|_{2}-a\right)^{2},$$

其中y是像素强度服从正态分布的随机图像，w∈f（z），其中z服从正态分布。，在高维情况下，当Jw在任意w处正交(上升到全球范围)时，这个先验被最小化。一个正交矩阵保持长度且不引入任何维数的压缩。

生成器为了避免总是被判别器惩罚，在整个潜码空间区域的布局上，对应于好图像的潜码会逐渐形成一个开阔平稳的平原，而对应于破损图像的潜码会形成犬牙交错、高低起伏的褶皱山区。在褶皱山区内部，生成器的映射平滑度是非常差的，即：在潜码空间上一个小的扰动就可能带来非常大的输出图像的改变，这在整体上导致PPL分数偏高（表明生成图像的质量较差）。

```python
def G_logistic_ns_pathreg(G, D, opt, training_set, minibatch_size, pl_minibatch_shrink=2, pl_decay=0.01, pl_weight=
    _ = opt
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    labels = training_set.get_random_labels_tf(minibatch_size)
    fake_images_out, fake_dlatents_out = G.get_output_for(latents, labels, is_training=True, return_dlatents=True)
    fake_scores_out = D.get_output_for(fake_images_out, labels, is_training=True)
    loss = tf.nn.softplus(-fake_scores_out) # -log(sigmoid(fake_scores_out))

    # Path length regularization.
    # 路径长度正则化运算
    with tf.name_scope('PathReg'):

        # Evaluate the regularization term using a smaller minibatch to conserve memory.
        # 评估正则化项，使用较小的小批量以节省内存
        if pl_minibatch_shrink > 1:
            pl_minibatch = minibatch_size // pl_minibatch_shrink
            pl_latents = tf.random_normal([pl_minibatch] + G.input_shapes[0][1:])
            pl_labels = training_set.get_random_labels_tf(pl_minibatch)
            fake_images_out, fake_dlatents_out = G.get_output_for(pl_latents, pl_labels, is_training=True, return_d

        # Compute |J*y|.
        # 计算|J*y|
        pl_noise = tf.random_normal(tf.shape(fake_images_out)) / np.sqrt(np.prod(G.output_shape[2:]))
        pl_grads = tf.gradients(tf.reduce_sum(fake_images_out * pl_noise), [fake_dlatents_out])[0]
        pl_lengths = tf.sqrt(tf.reduce_mean(tf.reduce_sum(tf.square(pl_grads), axis=2), axis=1))
        pl_lengths = autosummary('Loss/pl_lengths', pl_lengths)

        # Track exponential moving average of |J*y|.
        # 跟踪|J*y|的指数移动平均值
        with tf.control_dependencies(None):
            pl_mean_var = tf.Variable(name='pl_mean', trainable=False, initial_value=0.0, dtype=tf.float32)
        pl_mean = pl_mean_var + pl_decay * (tf.reduce_mean(pl_lengths) - pl_mean_var)
        pl_update = tf.assign(pl_mean_var, pl_mean)

        # Calculate (|J*y|-a)^2.
        # 计算(|J*y|-a)^2
        with tf.control_dependencies([pl_update]):
            pl_penalty = tf.square(pl_lengths - pl_mean)
            pl_penalty = autosummary('Loss/pl_penalty', pl_penalty)

        # Apply weight.
        # 应用权重（计算正则化项）
        #
        # Note: The division in pl_noise decreases the weight by num_pixels, and the reduce_mean
        # in pl_lengths decreases it by num_affine_layers. The effective weight then becomes:
        #
        # gamma_pl = pl_weight / num_pixels / num_affine_layers
        # = 2 / (r^2) / (log2(r) * 2 - 2)
        # = 1 / (r^2 * (log2(r) - 1))
        # = ln(2) / (r^2 * (ln(r) - ln(2))
        #
        reg = pl_penalty * pl_weight
```

y是随机图像，其像素（数值）强度服从正态分布；w ~ f(z)（w是z的函数），z 是正态分布的。

由于y = g(w)，则 Δ y = ∂ g(w)/∂ w ·Δw，即：Δ y = Jw·Δw同时，由于Jw是正交矩阵，JwT·Jw = E （E 为单位矩阵），因此：JwT·Δ y = Δw。当 y 是随机图像，其像素（数值）强度服从正态分布时，w 也表现为服从正态分布。也就是说，w 向量的微小扰动，表现为图像 y 的微小变化，二者的变化模式趋同。

若基于训练数据得出的Jw在任何 w 向量处都正交时，则公式（4）所表达的先验（期望）是最小的，这意味着此时 w 向量的微小扰动，所带来的图像 y 的变化是最小的，即：生成器的映射平滑度为最优。

为了避免雅克比矩阵的显式计算，我们使用恒等式：

$$\mathbf{J}_{\mathbf{w}}^{T}\mathbf{y} = \nabla_{\mathbf{w}}(g(\mathbf{w}) \cdot \mathbf{y})$$

其中，哈密顿算子使用标准的反向传播算法即可高效计算。在优化过程中，常数 a 被动态地设置为长度的长期指数移动平均值，使得优化过程可以仅凭自己就能找到合适的全局尺度。也就是说，公式（4）中的常数 a 引用的是当前优化周期得到的欧式距离长度的平均值，减去 a 之后，则公式（4）所表达的先验（期望）则接近于一个均值为 0 的正态分布，该正态分布越是呈现出一种窄而高的尖峰时，则生成器映射的期望方差越小，生成的图像质量越稳定。

Lab of Computer vision and Numerical optimization