Notes: "A Style-Based Generator Architecture for Generative Adversarial Networks"

Sun

May 18, 2020

1 Notes:

1.1 Summary:

1. Borrowing ideas from style transfer literature, they propose a new type generator
2. This new generator deals with a lot of the issues that GANs traditionally have: the images are sharp, cover more of the base distribution, and better latent codes
3. A huge improvement over GANs of the past; the successor, StyleGAN2, is still one of the best generative models available today (May 2020).

1.2 High level:

GANs approximate a given distribution by playing a minmax game that rewards the generator for generating realistic outputs. However, there are a few fallbacks to this. Traditional GANs fail to accurately generate rare samples / fail to capture diverse distributions, and are difficult to work with when you want to generate images similar to a given one.

StyleGAN[1] (SG) sidesteps these issues by some clever tricks that increase the expressiveness of GANs, in addition to a more natural scheme of generation. We now inject noise to each layer of the GAN, producing a stochastic affect that has distinct results depending on where the noise is injected.

In addition, SG introduces a "style" feature, which you can mix and match styles, producing images with mixed attributes.

1.3 Medium level:

A traditional GAN uses a random seed of fixed dimension as the basal input. Usually this seed is a small vector composed of noise sampled from independent Gaussians. SG deviates from this: the input into the model is now a learned constant. A latent code, $\mathbf{z}$, is sampled, then mapped through an 8-layer fully connected network. The output of this is a set of *style* attributes, two of which you use per upsample block.

This creates an interesting affect — the early styles decide the highest level attributes — glasses, gender, facial positioning, and expression, while the "higher" styles control lower and lower level attributes, going from eye status (open, closed, squinting), skin color, hair color, lip color, etc....

In addition, twice per upsampling block, we also introduce noise, denoted B in the figure below. Intuitively, the addition of explicit noise into the model corrosponds to actualization of low level stochastic variables (such as freckle positioning, individual hair strand positioning, eye glint, wrinkles, background, etc...). The AdaIN block is a modified version of BatchNorm (BN) that is good for style transfer; it will be covered in the next section. At a high level, it is a method to make the activations of a given content image match the activations in the same layer of a style image.

[1]https://arxiv.org/abs/1812.04948

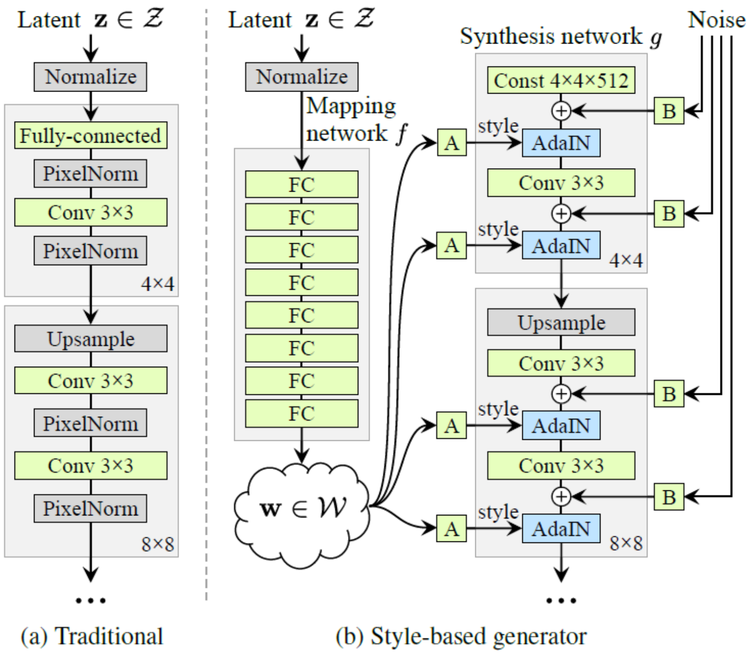

Figure 1: Proposed generator.

The noise itself has a bit more nuance to it: these low-level variables can be changed commonly without our perception of them, as long as they follow the correct distribution. A traditional generator must internalize the generation of these stochastic variations of low-level attributes: something that wastes network capacity, and something that can't even be done effectively (repetitive artifacts are common in traditional GAN generations). By externalizing the generation of these random effects, the architecture frees additional network capacity to focus on things that we *do* perceive.

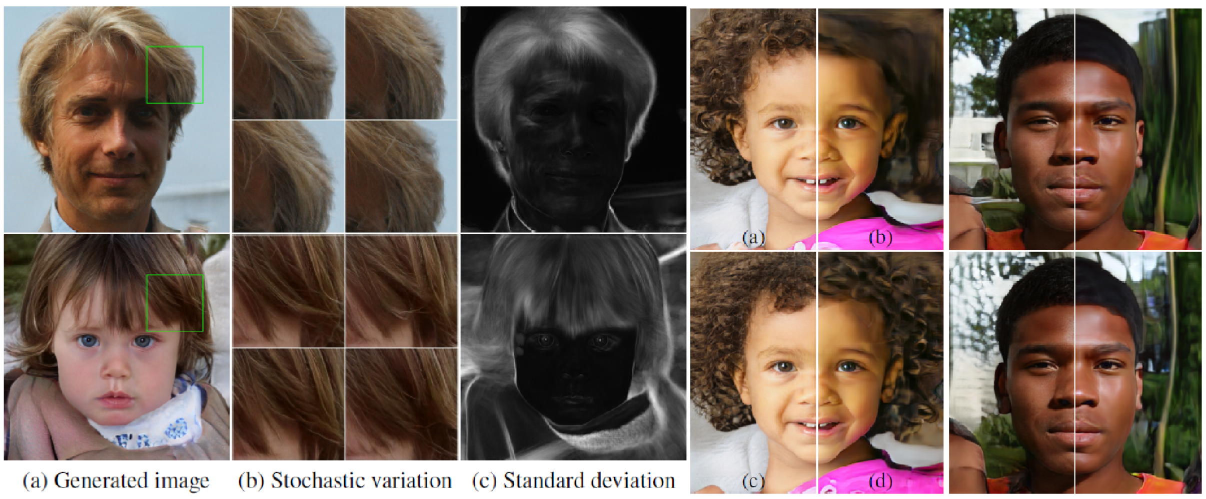

Figure 2: Examples of stochastic variation. **Left:** (a): Two generated images. (b): 8 zoom-ins with different realizations of input noise. Effect of noise at different layers. **Right:** (a): Noise is applied to all layers. (b): No noise. (c): Noise in fine layers only. (d): Noise in coarse layers only. You can empirically see that the more fine noise brings out more fine details; fine curls of hair, more percise background, skin pores, etc... In all, we can see the stochastic input noise only affects attributes that are naturally random to begin with.

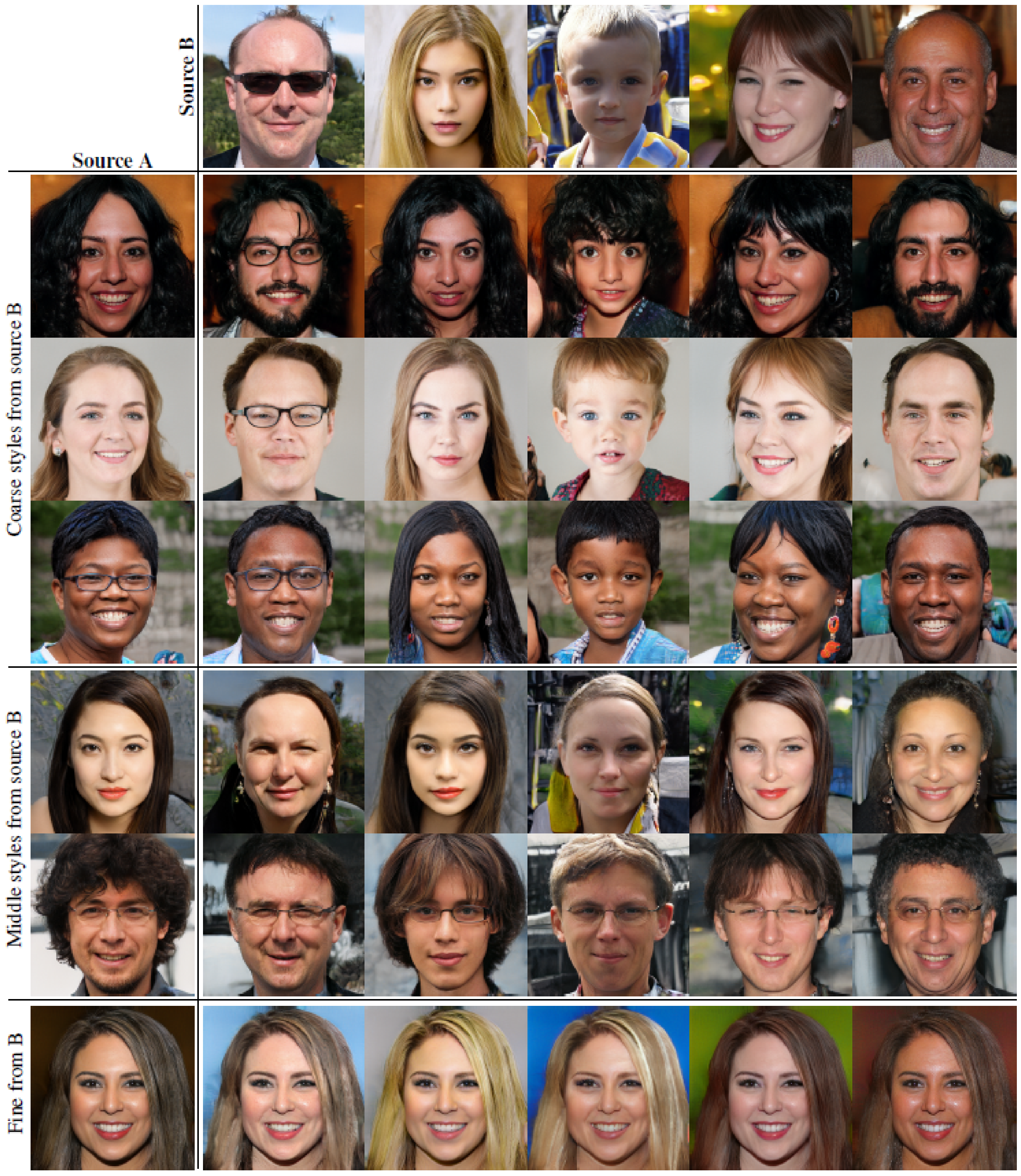

Figure 3: Two sets of images with varying latent codes. As denoted on the left side, different styles were taken from different images. We can see that, empirically, low-level style attributes decide the “base” of the image, and as you get “higher”, the effects become more and more low level.

1.4 Low level:

1.4.1 AdaIN:

As you may have noticed in the chart, AdaIN[2] is rather important here; it's actually the only thing connecting the model to the latent code, $\mathbf{z}$, or the style vector, $w \in \mathcal{W}$

Background: Batch normalization (BN) does as it sounds: it normalizes the input data given a batch. More specifically, given a batch $x \in \mathbb{R}^{N \times C \times H \times W}$:

$$\mathrm{BN}(x) = \gamma \left(\frac{x - \mu(x)}{\sigma(x)} \right) + \beta \tag{1}$$

where $\gamma, \beta \in \mathbb{R}^C$ are parameters learned from the data; $\mu(x), \sigma(x) \in \mathbb{R}^C$ are the mean and standard deviation calculated across batch size and spatial dimensions independently for each feature channel.

However, it's more useful than just that: BN can be used in style transfer. Naively, if transferring from a source domain to a target domain, source statistics should not be used for BN. Intuitively, if you are transferring from source to target, the statistics of your output image (mean, std, variance) should roughly equal that of the target domain.

Instance normalization: IN is the next extension to BN: merely calculate BN, except independently for each sample.

Conditional instance normalization: CIN came after. For each *style*, s,

$$\mathrm{CIN}(x, s) = \gamma^s \left(\frac{x - \mu(x)}{\sigma(x)} \right) + \beta^s \tag{2}$$

During training, a style image, together with the label of the style, is random chosen from a $fixed$ set of styles. The content image is then processed by a style transfer network, in which, for each layer, the corresponding γ^s and β^s are used.

AdaIN: The final extension we care about is AdaIN itself. If CIN normalizes an input to the style specified by affine parameters, is it possible to adapt it to arbitrarily given styles by using adaptive affine transformations?

Unlike BN, IN, and CIN, AdaIN has no learnable parameters. Instead, adaptively compute the parameters from the style input.

$$\mathrm{AdaIN}(x, y) = \sigma(y) \left(\frac{x - \mu(x)}{\sigma(x)} \right) + \mu(y) \tag{3}$$

Intuitively, consider a feature map channel that detects brushstrokes of a certain style. A style image with these kinds of strokes will produce a high average activation on this feature map. The output produced by AdaIN will have the same average activation for this feature map, while preserving the spatial structure of the content image. The brushstroke feature map can then be projected to image-space by using a feed-forward decoder.

AdaIN simply transfers the channel-wise mean and variance, which then directly leads to style transfer on a learned network.

How does StyleGAN use this? Twice per upsample block, we project the generated *style* ($w \in \mathcal{W}$) using some affine transformation such that the dimensionality of the style is twice that of the number of feature maps. We then input this projected style, y, into the AdaIN layer, combined with the generated feature map.

This corresponds to something quite smart: We take a latent code, $\mathbf{z}$, nonlinearly project it to be a style, w, then use this style to generate by-channel mean and variance parameters, which then scale and bias each given feature map. That, in turn, corresponds to realizing a random style, w, and then artificially generating these mean and variance parameters while actually generating the image this style would really correspond to.

[2] https://arxiv.org/abs/1703.06868

1.4.2 Why it works:

Lets consider why this works better than a traditional GAN. Immediately, we have a few reasons.

1. Traditional GANs *only* get to see the latent code in one layer, and the rest of the output must build iteratively on top of that. This means that the content of the entire end-image must be present within the latent code, and the model has no choice but to either immediately use or destroy the information; the model cannot easily say "oh, I will use this information later for the low-level details"; intuitively, we can see the latent code must be composed of mostly high-level details, and the model will implicitly fill in the rest.

   StyleGAN has no such restriction: the affine transformations applied to the Style prior to input to the AdaIN layer *can* selectively use the information within the style $w \in \mathcal{W}$. Ideally, the model would only receive high-level attributes at the very beginning, and the transformations would slowly introduce lower-level details. This allows for more varied information to be present in the style w.

2. It can be argued that styles are naturally more expressive than explicitly telling the model what to do: the model generates a wide base from the learned constant, then simply iteratively modifies it to fit a goal style.

1.4.3 Implementation

Some parts of the implementation are pure genius.

1. Upsampling! Instead of dealing with tricky, devlishing, disasteruous, good-for-nothing Conv2DTranspose layers, just upsample using some differentiable algorithm and blur. This makes a lot of sense: nearby values are very very likely to be similar anyways, by using a C2DT layer, you force the NN to learn a modified version of the identity (it's not quite the identity, but close values are very similar). NNs are, of course, bad at mapping the identity without residual connections, and this entirely side-steps the issue.

2. The 8-layer dense network that projects $\mathbf{z}$ to w. Intuitively, if we simply mapped $\mathbf{z}$ to styles, the generator itself (and the limited capacity of the mapping from $\mathbf{z}$ to a given style, y) would have to learn the support of the base distribution we are trying to approximate (to prevent "obviously" false samples from being generated). in SG, the 8-layer network assumes this role. The authors note that, if you add this part to even a regular GAN, the FID scores improve.

3. Style Mixing. A set percentage of images (hyperparameter) are generated using 2 style codes, not 1. This encourages the model to be able to generate more diverse samples by explicitly generating samples that might not be generated otherwise. For example, if the style-mapping network did not generate long-haired-males due to the fact that it could not do so without high loss, the style-mixing would force the model to do it anyways, assuming the model was generating long-haired-females.

1.4.4 Conclusion:

In all, this paper was a very good one, and I highly recommend y'all go check it out yourselves, same with the AdaIN paper; they put a lot of things better than I can.

Up next week: StyleGAN2.