

# Anime Generation

## **Group Member and Contributions:**

Wenjie Zhang (5677291) :

Wgan-div Experiment, Abstract, Results and Discussion

Qi Le (5674954) :

StyleGAN Experiment, Introduction, Background, Results and Discussion, Conclusion

John Dunlap (5354238):

Introduction, Background

## **Github Link:**

<https://github.com/zhan7867/CSCI-5523-Project>.git

## Abstract

In this project, the Wasserstein Gan - div(Wgan-div) model and the Style Generative Adversarial Network(StyleGAN) are compared with each other in different aspects. Our group modified the two models and analyzed the changes in the report. For the Wgan-div model, two trials, which are the Wgan-div model and the Wgan-div model modified with noise, have been run for 100 epochs with 1115 batches in each epoch. For the StyleGAN model, the official loss functions were replaced with Wgan-div loss functions by our group and compared with the original StyleGAN model. All the trials were running with 71313 anime images with the 64\*64 resolution. Finally, we recommend the StyleGAN model for the generation of images for its high quality.

## Introduction

The project being conducted is based on the Generative Adversarial Networks (GANs) model. GAN is a class of machine learning frameworks that are designed by Ian Goodfellow and his colleagues in 2014[1]. The generative model, which is an unsupervised learning task that involves discovering and learning the regularities or patterns in input data, is framed as a supervised learning problem with two sub-models by GANs and the core idea of a GAN is based on the "indirect" training through the discriminator, which itself is also being updated dynamically[2]. One sub-model of it is a generator model, which has been trained to generate new examples and another sub-model is a discriminator model, which has been used to classify the examples as either real or fake. We have several strategies to improve the discriminator. One technique is adversarial training, which is used to improve the robustness of the discriminator by combining adversarial attacker (generator) and discriminator in the training phase[5]. However, in this project, we will only train the two sub-models, discriminator and generator, together until the goal of the loss function are reached. Wgan-div model and StyleGAN model, which are used in the project are the modified GANs models.

GANs model is widely used in the computer science area. An example of the application of the GANs model is Image-to-Image Translation [4]. This model can be used to change the style of the image. The Cartoon Characters Generation, which can be used to create new anime characters, is another example of the application of the GANs model.

The purpose of this project is to gain an understanding of the structure and code of the advanced GANs models. This understanding is developed through the careful reading of the paper and the step-by-step debugging of the source code. The traditional GAN model, Wgan-div model, and the StyleGAN model are compared with each other in this report. After becoming familiar with the Wgan-div and StyleGAN models, our goal is to focus on the generator of the models and modify the Wgan-div model and the StyleGAN model with some parts we learned from the advanced GAN model. Finally, the newly generated anime images would be analyzed.

## Background

Compared to the traditional GAN model, the loss function would be the biggest change for the Wgan-div model.

$$\min_G \max_D \mathbb{E}_{G(z) \sim P_g} [D(G(z))] - \mathbb{E}_{x \sim P_r} [D(x)] - k \mathbb{E}_{\hat{x} \sim P_u} [\|\nabla_{\hat{x}} D(\hat{x})\|^p],$$

*Equation 1. Loss Function of Wgan-div*

$$-\mathbb{E}_{G(z) \sim P_g} [D(G(z))]$$

*Equation 2. Loss Function of Generator of Wgan-div*

$$\mathbb{E}_{G(z) \sim P_g} [D(G(z))] - \mathbb{E}_{x \sim P_r} [D(x)] - k \mathbb{E}_{\hat{x} \sim P_u} [\|\nabla_{\hat{x}} D(\hat{x})\|^p]$$

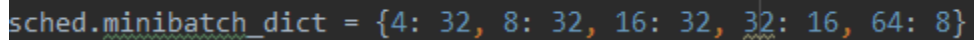
*Equation 3. Loss Function of Discriminator of Wgan-div*

The Wgan-div model uses Wasserstein distance instead of the traditional distance. If we represent  $P_r$  as the real sample distribution and  $P_g$  as the sample distribution generated by the generator,  $W(P_r, P_g)$ , which is the result of the Wasserstein distance, can represent the minimal cost of moving  $P_r$  to  $P_g$  under the optimal path. With the Wasserstein distance, the Wgan-div model can generate a continuous loss function to evaluate the quality of the model and avoid the gradient instability that exists in the traditional GAN models [6]. Furthermore, it can get rid of problems like mode collapse, and provide meaningful learning curves useful for debugging and hyperparameter searches [3].

Compared to the traditional GAN model, the StyleGAN has 4 main changes in the new style-based generator [7]. The first change is that the mapping network consisting of 8 fully connected layers is added [Appendix A, Figure 1], and the size of the output of Mapping Network  $W'$  is the same as that of the input layer ( $512 \times 1$ ). The second change is that the adaptive instance normalization is added [Appendix A, Figure 2] to transform the  $W'$  vector into a style control vector and influence the generation process of the generator. The third change is that the traditional latent point input is removed [Appendix A, Figure 3]. Instead of taking a point from the latent space as input, a constant  $4 \times 4 \times 512$  constant value becomes the input of the synthesis network. The fourth change is that the scaled noise is added to each resolution lever of the synthesis network [Appendix A, Figure 4].

## Results and Discussion

For StyleGAN, we tried many environment combinations, the correct environment would be: Cuda 10.0, Cudnn 7.4, Python 3.7, and tensorflow-gpu 1.13.1. Since the limited performance of our computer, we restrict the resolution of the image to 64\*64 and the mini-batch sizes for training the generator are reduced to a quarter of the minimum configuration. Since the resolution of the image is 64\*64, according to the formula in the StyleGAN, we have 10 layers in the mapping network and 5 blocks in the synthesis network.



```
sched.minibatch_dict = {4: 32, 8: 32, 16: 32, 32: 16, 64: 8}
```

*Figure 1. Mini-batch size for each layer. For example, mini-batch size would be 32 for the 4\*4 resolution layer*

StyleGAN source code from NVidia is complicated and the encapsulation and decoupling of StyleGAN code are very meticulous. We first tried to figure out the frame of the code and how the author implemented the architecture. For a couple of files in the StyleGAN code, the training folder is the part that needs to be focused on. The training folder contains the core content based on StyleGAN such as data processing, model architecture, loss function, and training process.

StyleGAN's network architecture is all written under training/networks\_StyleGAN.py, which mainly includes four components (line 305-664 of code): G\_style(), G\_mapping(), G\_synthesis() and D\_basic(). As shown in the figure above, G\_style represents the network architecture of the entire generator. It consists of two sub-networks, the mapping network G\_mapping and the synthesis network G\_synthesis; then D\_basic represents the network architecture of the entire discriminator, which follows the model design in ProGAN.

The G\_style network is located on lines 305-382 of code. The components defined in G\_style include: parameter verification -> set subnet -> set variable -> calculate mapping network output -> update the moving average of W -> perform style hybrid regularization -> apply truncation technique -> calculate synthetic network Output. Among them, setting the subnet is to call the process of building G\_mapping and G\_synthesis.

The G\_mapping network is located on lines 387-438 of code. The G\_mapping network has realized the process of mapping the code to the intermediate vector. The components defined in G\_mapping include: input -> connection label -> normalization latent code -> mapping layer -> broadcast -> output. The mapping layer is composed of 8 fully connected layers.

The G\_synthesis network is located at code 443-564. The G\_synthesis network realizes the synthesis process from the intermediate vector obtained by the broadcast to the generated image. The components defined in G\_synthesis include: preprocessing -> main input -> noise input -> modulation function at the end of each layer -> early layer (4\*4) structure -> remaining layer block -> network growth transformation process -> output. Among them, the modulation function at the end of each layer refers to the process of incorporating noise and style control

after convolution; the network growth transformation process refers to the dynamic growth of the synthetic network architecture during the training period to generate higher resolution images.

The D\_basic network is located on lines 569-664 of code. The D\_basic network realizes the function of distinguishing synthesized images from real images. In D\_basic, the components defined include: preprocessing -> building block -> network growth transformation process -> label calculation -> output. Among them, the network growth transformation process refers to the process of determining the dynamic growth of the network architecture as the resolution of the generated image increases during the training period.

A detailed explanation of the important part of the code that was used in this project could be found in Appendix B, C, and D.

After constructing the Wgan-div model and understanding the structure of the StyleGAN model, we found that there are significant differences between these two models.

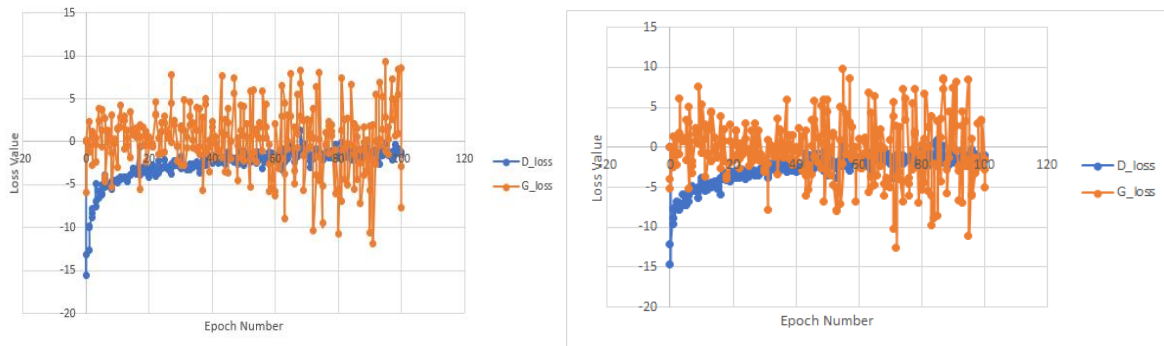
The first difference would be the structure of the generator. Our Wgan-div model contains five convolutional layers in the generator and the simpler network would connect to a more complex network until the model outputs the result. When we are trying to generate high-resolution images, this model has a big weakness, which is the long program running time. But in the generator of StyleGAN, although there are many layers, there is only one network inside the Generator. Furthermore, StyleGAN uses a smooth transition technology[Appendix C] to handle the dynamically changing of the structure of the network during the training process.

The second difference is the normalization of the parameters in the model. The problem of difficulty in training often occurs when training GANs, because after each parameter iteration, the distribution of the output data of the previous network will change because the data have been calculated by the previous network. This phenomenon will cause difficulty for the learning of the next network. We need normalization to keep the input of each layer of the network in the same distribution during the GANs training process. In the Wgan-div model, the data would be normalized during each convolutional layer in the generator. In the StyleGAN model, the normalization almost happens in every step, such as each sampling layer, each layer in the synthesis network. Pixel\_norm() and instance\_norm() [Appendix C] would be two important normalization steps. Pixel\_norm() is a variant of local response normalization and has the effect of avoiding the explosion of the generator gradient. Pixel norm is normalized along the channel dimension (axis=1). One advantage of such normalization is that each position of the feature map has a unit length. This normalization strategy has a greater relationship with the generator output designed by the author. Instance\_norm() Instance normalization is a normalization method that is widely used in generative models and its main feature is that it only normalizes the height and width dimensions of features, which has a significant impact on the style of the image.



*Figure 2. Effect of noise inputs at different layers of our generator. (a) Noise is applied to all layers. (b) No noise. From 'A Style-Based Generator Architecture for Generative Adversarial Networks'*

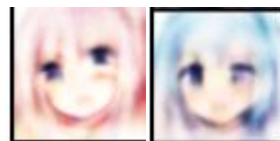
The third difference is the noise. From the Figure 2 above, we can observe the significant difference between the generated images with noise applied to all layers and the no noise. In the Wgan-div model, the noise only being generated on the input vector. But in the StyleGAN model, 10 learned per-channel scaled noises are connected to the synthesis network.



*Figure 3 . The changing of generator loss and discriminator loss along with the increasing epoch number up to 100 epochs (Left, without noise). The changing of generator loss and discriminator loss along with the increasing epoch number up to 100 epochs (Right, with noise)*

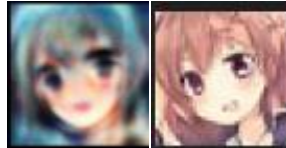


*Figure 4. The image generated by the Wgan-div model (without noise)*



*Figure 5. The image generated by the Wgan-div model (with noise)*

Since the noise affects the quality of the images a lot, our group implements the Wgan-div model by adding the noise on every layer of the generator to see if the quality of the image increases. Both trials have been run for 100 epochs and 1115 batches with 64 batch size for each epoch. The results show in Figures 3,4 and 5. The results are similar. The main reason is that the noise added to each layer of the StyleGAN generator is scaled and transformed by learnable scaling parameters. The parameters are changed by the backpropagation and this operation would slightly change the visual expression of the features. So, a simple addition of noise does not change the quality of the generated image.



*Figure 6. The image generated by the Wgan-div model vs The image generated by the StyleGAN model*

Each anime faces generated in the above two images has the resolution of 64\*64 pixels. By comparing these two images in Figure 6, we can easily find out that the quality of the image generated by the StyleGAN model is much better than the quality of the image generated by the Wgan-div model in two aspects: details and decoupling. The right image has clear details, such as eyes, hair and face and the different parts of face are independent with each other, which give us the vivid image.

```
# Linear structure: simple but inefficient.
if structure == 'linear':
    images_out = torgb(2, x)
    for res in range(3, resolution_log2 + 1):
        lod = resolution_log2 - res
        x = block(res, x)
        img = torgb(res, x)
        images_out = upscale2d(images_out)
        with tf.variable_scope('Grow_lod%d' % lod):
            images_out = tf.nn.lerp_clip(img, images_out, lod_in - lod)
```

*Figure 7. Linear structure of each block in the generator of the StyleGAN. Detailed information of functions can be found in Appendix*

The main reason for the different details is that the StyleGan model is more complex than the WGan-div model. In each block in the generator of the StyleGAN, the intermediate variables would go through many steps (Figure 7) excluding the simple convolutional layer. The intermediate variables would first go into the block() function, which contains blur() and upscale2d\_conv2d() function. The upscale2d\_conv2d() function could double the resolution of the intermediate variables. It adopts linear interpolation with size value clipping to achieve smooth transition under different resolutions. In the blur() function, the first part is the processing of the convolution kernel, including dimension specification and normalization; the second part is the realization of blur - the convolution kernel uses depthwise\_conv2d convolution

to change the intermediate variables. Moreover, depthwise\_conv2d convolution is slightly different from ordinary convolution. Ordinary convolution performs convolution and addition on the two channels of each out\_channel of the convolution kernel and the two channels of the input to obtain a channel of the feature map. The depthwise\_conv2d convolution generates two out\_channel for each corresponding in\_channel and we can use in\_channel \* channel\_multiplier to express the number of channels of the feature map.

```
# Things to do at the end of each layer.
def layer_epilogue(x, layer_idx):
    if use_noise:
        x = apply_noise(x, noise_inputs[layer_idx], randomize_noise=randomize_noise)
    x = apply_bias(x)
    x = act(x)
    if use_pixel_norm:
        x = pixel_norm(x)
    if use_instance_norm:
        x = instance_norm(x)
    if use_styles:
        x = style_mod(x, dlatents_in[:, layer_idx], use_wscales=use_wscales)
    return x
```

*Figure 8. Layer\_epilogue in each block in the generator of the StyleGAN. Detailed information of functions can be found in Appendix B, C and D*

Then the intermediate variables would go into the layer\_epilogue() function in Figure 8. Six steps are processed in the layer\_epilogue(), which causes the decoupling difference. The important thing is that the noise and the style modulation are applied to the intermediate variables. The intermediate vector  $W'$  after feature unwrapping passes through a learnable affine transformation, which is the fully connected layer and becomes the scaling factor  $ys, i$ , and the deviation factor  $yb, i$ . These two factors will do a weighted summation with the normalized convolution output to complete the process of  $W'$  affecting the original intermediate variables. Moreover, this influence method can achieve style control, mainly because it allows  $W'$  (that is, the transformed  $ys, i$  and  $yb, i$ ) to affect the global information of the picture. The key information that retains the generated face is determined by the upsampling layer and the convolutional layer, so  $W'$  can only affect the style information of the picture. Finally, the intermediate variables would go into the torgb() function and are converted into RGB images.

The StyleGAN uses many tricks in upscaling and convolving the intermediate variables. The image generated by StyleGAN proves that the complex structure and the tricks have a huge impact on the quality of the image. But some tricks are confusing and lack math support.

Since the generators of the two models have huge differences from each other, we tried to modify the loss function of the StyleGAN model to find out if the loss function of the Wgan-div model can be used in the StyleGAN model. The basis for this modification is that the Wasserstein distance directly acts on the distribution of the real images and the generated images.



```

def G_wgan_div(G, D, opt, training_set, minibatch_size):
    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

```

Figure 9. The Implemented Loss Function for StyleGAN Generator using Wgan-div Loss Function

```

def D_wgan_div(G, D, opt, training_set, minibatch_size, reals, labels,
              wgan_div_lambda=2.0,
              wgan_div_epsilon=0.001,
              wgan_div_target=1.0):
    wgan_div_power = 6

    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

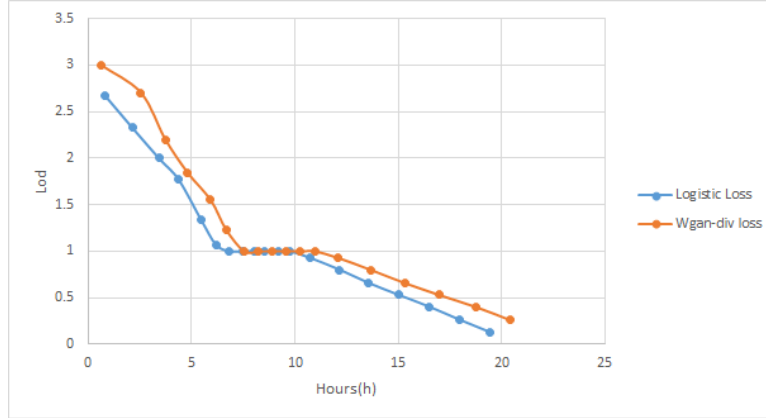
    with tf.name_scope('GradientPenalty'):
        mixing_factors = tf.random_uniform([minibatch_size, 1, 1, 1], 0.0, 1.0, dtype=fake_images_out.dtype)
        mixed_images_out = tf.nn.lerp(tf.cast(reals, fake_images_out.dtype), fake_images_out, mixing_factors)
        mixed_scores_out = fp32(D.get_output_for(mixed_images_out, 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.pow(mixed_norms, wgan_div_power)
        loss += gradient_penalty * (wgan_div_lambda / (wgan_div_target ** 2))

    return loss

```

Figure 10. The Implemented Loss Function for StyleGAN Discriminator using Wgan-div Loss Function

The official loss function for StyleGAN would be logistic loss. We modify the loss function using Wgan-div loss function (training/loss.py), which shows in Figure 9 and 10. In the modified loss function, the distribution of the images is measured by the `get_output_for()` function. Equation(2) is used to minimize the loss function of the generator and equation(3) is used to minimize the loss function of the discriminator.



*Figure 11. The Lod of the StyleGAN vs Running Time. Vertical Axis : Lod (Level of Detail). Horizontal Axis: Time (Hours)*



*Figure 12. Images generated by the StyleGAN with logistic loss function*



*Figure 13. Images generated by the StyleGAN with Wgan-div loss function*

Figures 11, 12, and 13 show the running result StyleGAN using different loss functions. Both trials have been run for about 20 hours, with 32 ticks(4.48 million images). The level of detail is measured by the Perceptual Path Length (metrics/perceptual\_path\_length.py). The smaller the level of detail is, the clearer the images are. We can find out that both loss functions have good performance. Compared to the level of detail of Logistic loss function, the level of detail of Wgan-div loss function is a little higher, which means for StyleGAN, Wgan-div loss function is a good choice but is not the best choice. We can also notice that the level of detail decreases rapidly at the beginning of the program and among 7 to 10 hours, the level of detailing stays the same. Among 10 to 20 hours, the decreasing rate of the level of detail decreases a lot, which means the difficulty of letting the model establish a transformation from simple distribution to the complex distribution increases a lot. When we are trying to generate the high-resolution images, the task would be undoubtedly huge. This reflects the importance of a single network structure in the generator with successively increasing higher resolution network layers in generating high-resolution images.

## Conclusions

In conclusion, this project proceeded as expected. We gain a deep understanding of the architecture and principles of Wgan-div model and StyleGAN model and we improve the coding ability. The ideas of the Wgan-div loss function, AdaIN, mapping network, upscale images, and the tricks of the convolutional kernel are very valuable. The images generated by the Wgan-div model are worse than those of the StyleGAN model because of the huge difference between the structure of the generator. A simple addition of noise to the Wgan-div model does not change the quality of the generated image. However, we obtain similar quality images from the StyleGAN model with original loss function and the StyleGAN model with Wgan-div loss function, which proves that the continuous Wasserstein distance could provide meaningful gradients. Between the Wgan-div model and the StyleGAN model, we recommend using the StyleGAN model to generate the images. With the pre-trained model, people can easily generate high-quality images from the StyleGAN model.

## References

- [1] Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua (2014). Generative Adversarial Networks. Proceedings of the International Conference on Neural Information Processing Systems (NIPS 2014). pp. 2672–2680.
  
- [2] Adaloglou, Nikolas. Vanilla GAN (GANs in computer vision: Introduction to generative learning. theaisummer.com. AI Summer.
  
- [3] Arjovsky, Martin; Chintala, Soumith; Bottou, Léon. Wasserstein GAN. (2017). arXiv:1701.07875.
  
- [4] Isola, Phillip; Zhu, Jun-Yan; Zhou, Tinghui; Efros, Alexei A.; Image-To-Image Translation With Conditional Adversarial Networks.(2017). Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 1125-1134
  
- [5] Liu, Xuanqing; Hsieh, Cho-Jui. Rob-GAN: Generator, Discriminator, and Adversarial Attacker. (2018). arXiv:1807.10454.
  
- [6] Gulrajani, Ishaan; Ahmed, Faruk; Arjovsky, Martin; Dumoulin, Vincent; Courville, Aaron. Improved Training of Wasserstein GANs. (2017). arXiv:1704.00028v3
  
- [7] Karras Tero; Laine, Samuli; Aila, Timo. A Style-Based Generator Architecture for Generative Adversarial Networks. (2018). arXiv:1812.04948

# Appendices

## Appendix A: Basic Structure of StyleGAN

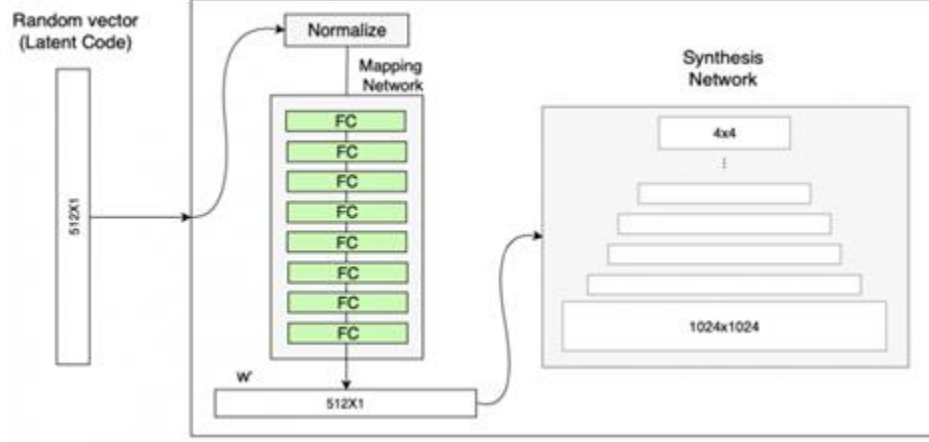


Figure 1. The generator with the Mapping Network (in addition to the ProGAN synthesis network)

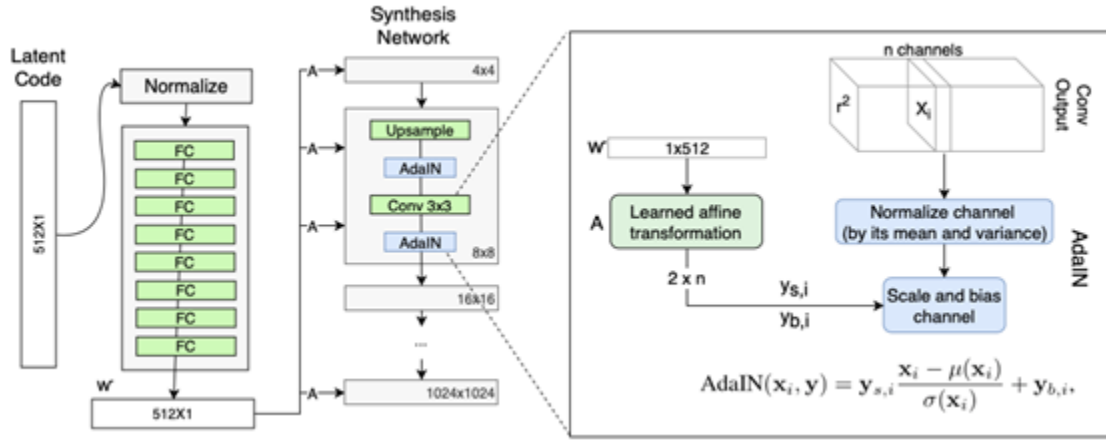


Figure 2. The generator's Adaptive Instance Normalization (AdaIN)

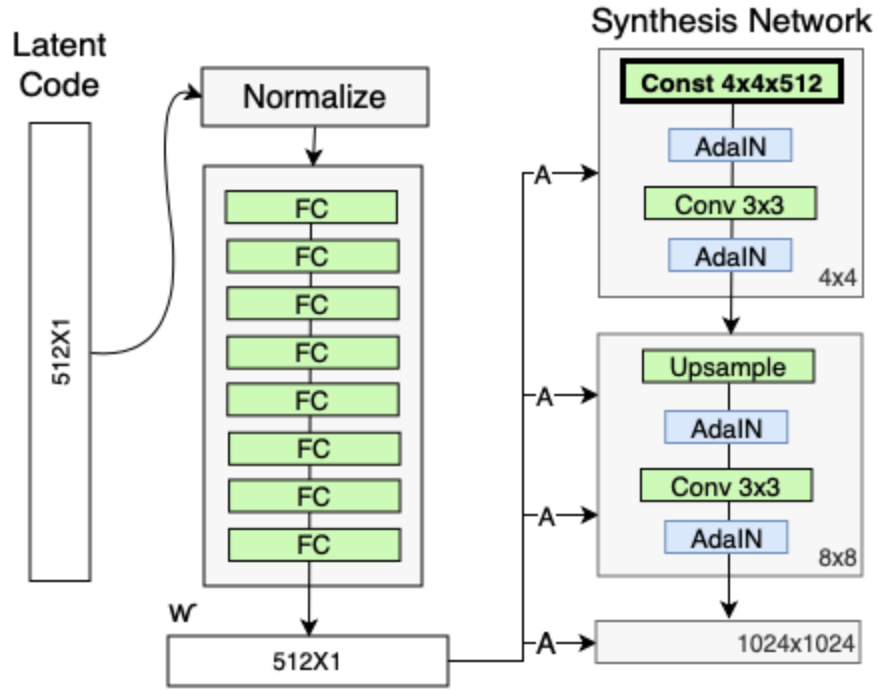


Figure 3. The Synthesis Network input is replaced with a constant input

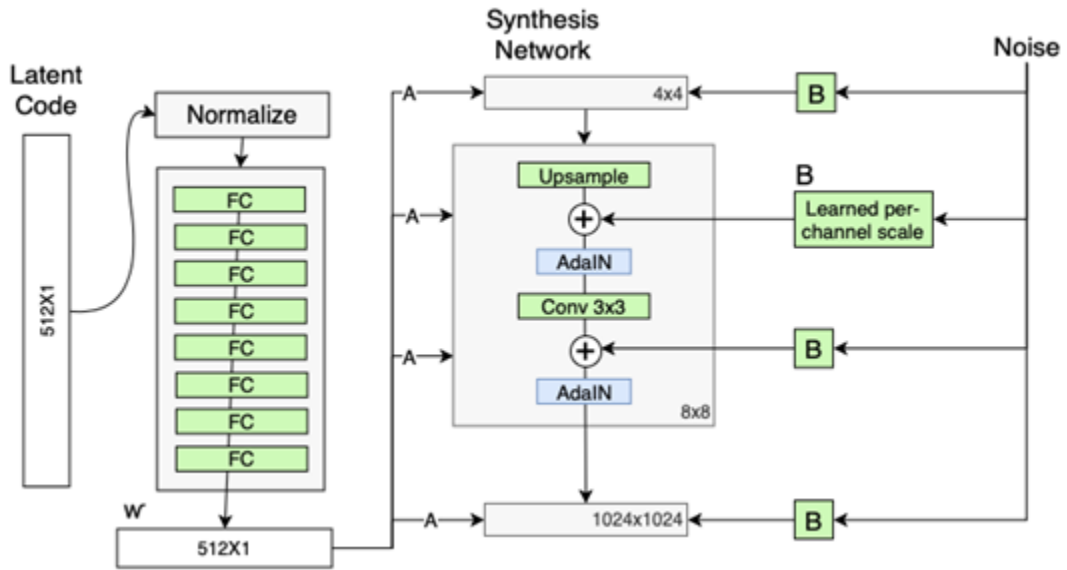


Figure 4. Adding scaled noise to each resolution level of the synthesis network

## Appendix B: Explanation of G\_style

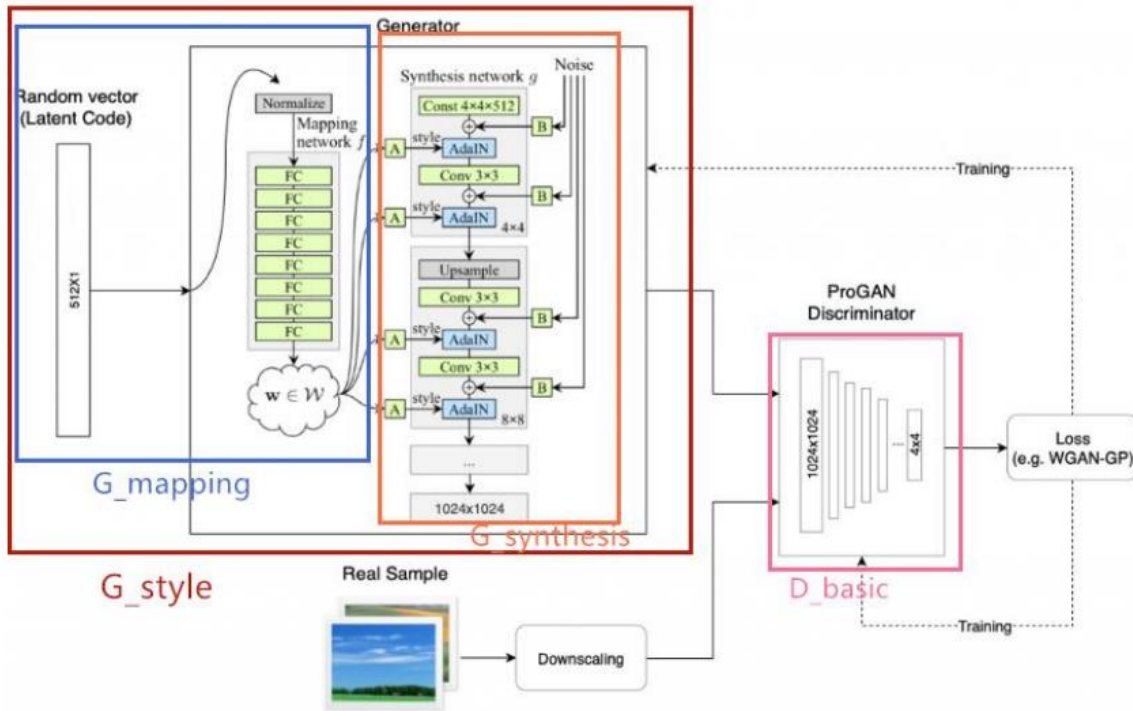


Figure 1. General flow chart of StyleGAN

```
def G_style(
    latents_in,          # First input: Latent vectors (Z) [minibatch, latent_size].
    labels_in,           # Second input: Conditioning labels [minibatch, label_size].
    truncation_psi       = 0.7,      # Style strength multiplier for the truncation trick. None = disable.
    truncation_cutoff    = 0,        # Number of layers for which to apply the truncation trick. None = disable.
    truncation_psi_val   = None,     # Value for truncation_psi to use during validation.
    truncation_cutoff_val = None,     # Value for truncation_cutoff to use during validation.
    dlatent_avg_beta     = 0.995,    # Decay for tracking the moving average of W during training. None = disable.
    style_mixing_prob     = 0.9,     # Probability of mixing styles during training. None = disable.
    is_training          = False,    # Network is under training? Enables and disables specific features.
    is_validation         = False,    # Network is under validation? Chooses which value to use for truncation_psi.
    is_template_graph    = False,    # True = template graph constructed by the Network class, False = actual evaluation.
    components           = dnnlib.EasyDict(), # Container for sub-networks. Retained between calls.
    **kwargs):           # Arguments for sub-networks (G_mapping and G_synthesis).
```

**G\_style input parameters (line 304-317):** Input parameters include 512-dimensional Z code vector and conditional labels

```
# Setup components.
if 'synthesis' not in components:
    components.synthesis = tfli.Network('G_synthesis', func_name=G_synthesis, **kwargs)
num_layers = components.synthesis.input_shape[1]
dlatent_size = components.synthesis.input_shape[2]
if 'mapping' not in components:
    components.mapping = tfli.Network('G_mapping', func_name=G_mapping, dlatent_broadcast=num_layers, **kwargs)
```

**Set up the subnet (line 334-340):** Directly use the `tflib.Network()` class (act as a convenient wrapper for the parameterized network construction function, providing a variety of practical methods And easily access input/output/weight) create two sub-networks.

```
# Setup variables.
lod_in = tf.get_variable('lod', initializer=np.float32(0), trainable=False)
dlatent_avg = tf.get_variable('dlatent_avg', shape=[dlatent_size], initializer=tf.initializers.zeros(), trainable=False)
```

**Set variables (line 342-344):** Set two variables `lod_in` and `dlatent_avg`. The former determines the resolution at the current level, which `lod` equals to `resolution_log2-res` (where `res` represents the resolution level (2-10) corresponding to the current layer); the latter determines the benchmark of the truncation operation, that is, the average value of the `dlatent` codes that generate the face.

```
# Update moving average of W.
if dlatent_avg_beta is not None:
    with tf.variable_scope('DlatentAvg'):
        batch_avg = tf.reduce_mean(dlatents[:, 0], axis=0)
        update_op = tf.assign(dlatent_avg, tflib.lerp(batch_avg, dlatent_avg, dlatent_avg_beta))
        with tf.control_dependencies([update_op]):
            dlatents = tf.identity(dlatents)
```

**Update the moving average of W (line 349-355):** The average value of batch `dlatent` is approached to the average value of total `dlatent` by `dlatent_avg_beta` stride as the new average value of face `dlatent(dlatent_avg)`.

```
# Evaluate synthesis network.
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')
```

**Compute synthesis network output (line 378-381):** Pass the truncated `dlatents` to the `G_synthesis` network for synthesis, and the result obtained is the output result of the entire generation network `G_style`.



## Appendix C: Explanation of G\_mapping

```
# Normalize latents.  
if normalize_latents:  
    x = pixel_norm(x)
```

```
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)
```

**pixel\_norm Function(line 418-420):** The realization of pixel-by-pixel normalization is  $x = x / \sqrt{1/N(\sum_{i=0}^{N-1} x_i^2 + \epsilon)}$ , where  $\epsilon = 10^{-8}$

$$x = x / \sqrt{1/N(\sum_{i=0}^{N-1} x_i^2 + \epsilon)}, \text{ where } \epsilon = 10^{-8}$$

```
# Mapping layers.  
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  
        x = dense(x, fmaps=fmaps, gain=gain, use_wscales=use_wscales, lrmul=mapping_lrmul)  
        x = apply_bias(x, lrmul=mapping_lrmul)  
        x = act(x)
```

**Mapping layer (line 422-428):** Each mapping layer has three components: fully connected layer `dense()`, bias function `apply_bias()` and activation function `act()`.

```
# Fully-connected layer.  
  
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:]])])  
    w = get_weight([x.shape[1].value, fmaps], **kwargs)  
    w = tf.cast(w, x.dtype)  
    return tf.matmul(x, w)
```

**Dense Function(line 152-159):** The `dense()` function first flattens the output to calculate the output dimension, then calls `get_weight()` to create a fully connected layer `w`, and finally returns the result of the matrix multiplication of `x` and `w` as the output of the `dense()` layer.

```

# Get/create weight tensor for a convolutional or fully-connected layer.

def get_weight(shape, gain=np.sqrt(2), use_wscales=False, lrmul=1):
    fan_in = np.prod(shape[:-1])_# [kernel, kernel, fmaps_in, fmaps_out] or [in, out]
    he_std = gain / np.sqrt(fan_in)_# He init

    # Equalized learning rate and custom learning rate multiplier.
    if use_wscales:
        init_std = 1.0 / lrmul
        runtime_coef = he_std * lrmul
    else:
        init_std = he_std / lrmul
        runtime_coef = lrmul

    # Create variable.
    init = tf.initializers.random_normal(0, init_std)
    return tf.get_variable('weight', shape=shape, initializer=init) * runtime_coef

```

**Get\_weight Function(line 133-149):** The get\_weight() function is a function used to create a convolutional layer or a fully connected layer and obtain a weight tensor.

```

# Apply bias to the given activation tensor.

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])

```

**Apply\_bias Function(line 211-218):** Apply a bias to the given activation tensor.

```

# Broadcast.
if dlatent_broadcast is not None:
    with tf.variable_scope('Broadcast'):
        x = tf.tile(x[:, np.newaxis], [1, dlatent_broadcast, 1])

```

**(line 430-433):** The activation function uses the value of mapping\_nonlinearity. 'lrelu' activation function is used in StyleGAN, and the gain value is  $\sqrt{2}$ .

## Appendix C: Explanation of G\_synthesis

```
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)
        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)
    return x
```

**Instance\_norm Function(line 247-256):** Instance Normalization

```
# Noise inputs.
noise_inputs = []
if use_noise:
    for layer_idx in range(num_layers):
        res = layer_idx // 2 + 2
        shape = [1, use_noise, 2**res, 2**res]
        noise_inputs.append(tf.get_variable('noise%d' % layer_idx, shape=shape, initializer=tf.initializers.random_normal(), trainable=False))
```

**Create noise (line 484-490):** When initially creating the noise, just create the corresponding shape according to the resolution of the corresponding layer, and then randomly initialize it.

```
# Things to do at the end of each layer.
def layer_epilogue(x, layer_idx):
    if use_noise:
        x = apply_noise(x, noise_inputs[layer_idx], randomize_noise=randomize_noise)
    x = apply_bias(x)
    x = act(x)
    if use_pixel_norm:
        x = pixel_norm(x)
    if use_instance_norm:
        x = instance_norm(x)
    if use_styles:
        x = style_mod(x, dlatents_in[:, layer_idx], use_wscale=use_wscale)
    return x
```

**Layer end modulation (including AdaIN, line 492-504):** End-of-layer modulation is the processing of features after the convolution of each block, including 6 (optional) contents: apply\_noise()(Apply noise), apply\_bias()(Apply bias), act()(Apply activation function), pixel\_norm()(pixel-by-pixel normalization), instance\_norm()(Instance normalization), style\_mod()(style modulation (AdaIN)).

```
# Style modulation.

def style_mod(x, dlatent, **kwargs):
    with tf.variable_scope('StyleMod'):
        style = apply_bias(dense(dlatent, fmaps=x.shape[1]*2, gain=1, **kwargs))
        style = tf.reshape(style, [-1, 2, x.shape[1]] + [1] * (len(x.shape) - 2))
        return x * (style[:,0] + 1) + style[:,1]
```

**Style control (AdaIN)(line 259-267):** The first line is the affine change A, which doubles the dlatent through the fully connected layer; the second line converts the expanded dlatent into the scaling factor  $y_{s,i}$  and the bias factor  $y_{b,i}$ ; the third line is the two A factor implements adaptive instance normalization to the convolved x.

```
# Convolutional layer.

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')
```

**Conv2d Function(line 162-168):** changing the number of channels of x from x.shape[1] to fmaps, while the size of x remains unchanged.

```
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

    # Not fused => call the individual ops directly.
    if not fused_scale:
        return conv2d(upscale2d(x), fmaps, kernel, **kwargs)

    # Fused => perform both ops simultaneously using tf.nn.conv2d_transpose().
    w = get_weight([kernel, kernel, x.shape[1].value, fmaps], **kwargs)
    w = tf.transpose(w, [0, 1, 3, 2]) # [kernel, kernel, fmaps_out, fmaps_in]
    w = tf.pad(w, [[1,1], [1,1], [0,0], [0,0]], mode='CONSTANT')
    w = tf.add_n([w[1:, 1:], w[:-1, 1:], w[1:, :-1], w[:-1, :-1]])
    w = tf.cast(w, x.dtype)
    os = [tf.shape(x)[0], fmaps, x.shape[2] * 2, x.shape[3] * 2]
    return tf.nn.conv2d_transpose(x, w, os, strides=[1,1,2,2], padding='SAME', data_format='NCHW')
```

**Upscale2d\_conv2d Function(line 174-191):** Use tf.nn.conv2d\_transpose deconvolution operation to double the feature map.

```
# Linear structure: simple but inefficient.
if structure == 'linear':
    images_out = torgb(2, x)
    for res in range(3, resolution_log2 + 1):
        lod = resolution_log2 - res
        x = block(res, x)
        img = torgb(res, x)
        images_out = upscale2d(images_out)
        with tf.variable_scope('Grow_lod%d' % lod):
            images_out = tflib.lerp_clip(img, images_out, lod_in - lod)
```

**Linear Structure(line 540-549)**

## Appendix D: Explanation of D\_basic

```
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

    # Not fused => call the individual ops directly.
    if not fused_scale:
        return downscale2d(conv2d(x, fmaps, kernel, **kwargs))

    # Fused => perform both ops simultaneously using tf.nn.conv2d().
    w = get_weight([kernel, kernel, x.shape[1].value, fmaps], **kwargs)
    w = tf.pad(w, [[1,1], [1,1], [0,0], [0,0]], mode='CONSTANT')
    w = tf.add_n([w[1:, 1:], w[:-1, 1:], w[1:, :-1], w[:-1, :-1]]) * 0.25
    w = tf.cast(w, x.dtype)
    return tf.nn.conv2d(x, w, strides=[1,1,2,2], padding='SAME', data_format='NCHW')
```

**Conv2d\_downscale2d Function(line 193-208):** Use tf.nn.conv2d convolution operation to reduce the feature map by one time. The convolution kernel is slightly translated four times and superimposed on itself and then average.s