Analyzing and Improving the Image Quality of StyleGAN

Table of Contents

SUMMARY OF PAPER

GAN

Generative methods, notably Generative Adversarial Networks (GANs) [7], continually advance in image resolution and quality [10, 13, 2]. StyleGAN [11] is the leading high-resolution image synthesis method, known for its reliability across diverse datasets. This paper aims to rectify characteristic artifacts in StyleGAN-generated images and improve overall image quality.

StyleGAN

StyleGAN's distinguishing feature [11] lies in its unconventional generator architecture. Rather than directly inputting latent code $z \in Z$ into the network, a mapping network f transforms it into an intermediate latent code $w \in W$. This w controls the synthesis network g via adaptive instance normalization (AdaIN) [9, 4, 6, 3], supported by additional random noise maps for stochastic variation. This design choice results in a less entangled intermediate latent space W, the focus of the analysis from the synthesis network's perspective.

Key Issues

Characteristic artifacts observed in StyleGAN-generated images [7] stem from two primary causes. Blob-like artifacts result from a design flaw in the generator's architecture, addressed by redesigning the normalization process. Artifacts related to progressive growth during training [10] are tackled through an alternative design without altering network topology, revealing lower-than-expected effective image resolution and justifying a capacity increase.

Perceptual Path Length

Quantitative analysis of image quality remains a challenge. Metrics like Frechet inception distance (FID) [8] and Precision and Recall (P&R) [14, 12] mainly focus on textures rather than shapes [5]. Perceptual path length (PPL) [11] correlates with shape consistency and stability. Regularizing the synthesis network to favor smoother mappings significantly enhances image quality. The study also reduces the frequency of regularizations to manage computational costs without compromising effectiveness.

Blob-Shaped Artifacts

The prevalent blob-shaped artifacts resembling water droplets found in StyleGAN-generated images by attributing their presence to the AdaIN operation, which normalizes feature map statistics and potentially disrupts feature magnitude information. Through the proposal of a redesigned architecture, specifically shifting bias and noise operations and introducing a demodulation technique replacing instance normalization within convolutional layers, the study successfully removes these artifacts while maintaining control over image generation.

Architecture Change

The progressive growing technique leads to a strong preference for specific locations in the generated details, causing them to appear stuck or disjointed within the image. This issue arises because each resolution serves as the output resolution temporarily, forcing maximal frequency details generation, which in turn results in excessively high frequencies in intermediate layers, compromising the model's shift-invariance. To address this, the paper explores alternative network architectures, examining skip connections, residual networks, and hierarchical methods. Through extensive comparisons and evaluations using FID and PPL metrics, they identify that skip connections in the generator significantly improve PPL scores across configurations, while a residual discriminator network exhibits clear benefits for FID. Ultimately, adopting a skip generator and a residual discriminator without progressive growing (corresponding to configuration E) notably enhances FID and PPL scores, demonstrating a more effective network design for image generation.

IMPORTANCE OF PAPER

The paper "Analyzing and Improving the Image Quality of StyleGAN," published in 2020 at the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) holding the top rank (1) on research.com signifies its eminent position and influence within the academic community, emerged as a pivotal contribution in the realm of generative modeling. The Impact Score of 63.10 further attests to the significance of the research published within the conference, reflecting the widespread impact and citation of the papers presented at CVPR. Led by authors from NVIDIA, a prominent institution in graphics processing and AI, this paper stood out by dissecting and addressing critical limitations in StyleGAN, a leading generative model for high-resolution image synthesis. By identifying inherent artifacts in StyleGAN-generated images and proposing innovative solutions, such as redesigning normalization techniques and exploring alternative network architectures, the research significantly advanced the understanding and enhancement of image quality in generative models. Its presence in a prestigious conference, coupled with its impactful contributions and subsequent high citation count of 4527, underscored its immediate recognition and lasting influence within the computer vision community, marking it as a standout and influential work in the field at the time of publication.

IMPROVEMENT OF STANDARD GAN ALGORITHM

StyleGAN, a significant evolution from the traditional Generative Adversarial Network (GAN), brings forth a huge amount of enhancements and innovations. Unlike standard GANs that lack fine-grained control over specific image features, StyleGAN introduces a latent space that allows for independent manipulation of various attributes, and more precise and targeted image generation. This architecture employs a progressive growing mechanism that enables the generation of high-resolution images by incrementally adding layers, both quality and stability throughout the training process. Additionally, the network redesign and normalization modifications within StyleGAN comprehend inherent artifacts observed in GAN-generated images, like mode collapse and the presence of consistent visual anomalies, resulting in a more stable and coherent image generation process.

The standard GAN algorithm faces limitations in generating high-resolution, diverse, and realistic images due to challenges related to training instability, mode collapse, and lack of control over specific features. StyleGAN mitigates these shortcomings by introducing a progressive growing mechanism, allowing for a smoother training process and higher-resolution image generation. By incorporating latent style vectors, StyleGAN empowers users to control individual aspects of the generated images, enabling the creation of diverse and realistic outputs. Moreover, the introduction of perceptual path length (PPL) as a novel evaluation metric in StyleGAN offers a more complex understanding of image quality by focusing on smoothness and perceived visual fidelity, overcoming the traditional Fréchet Inception Distance (FID) metric used in standard GANs.

StyleGAN over comes the standard GAN framework by not only addressing the shortcomings but also introducing new quality, diversity, and control in image generation. Its progressive growing architecture and novel techniques for latent space manipulation enable the synthesis of high-resolution images with improved stability and fidelity. The redesigned normalization methods and modifications in network architecture significantly reduce artifacts and anomalies, providing more consistent and visually appealing results. By introducing PPL as a metric that captures the smoothness and perceptual quality of generated images, StyleGAN surpasses the limitations of traditional evaluation methods, offering a more comprehensive assessment of image quality and providing a groundbreaking framework for the generation of high-quality, realistic images.

RECENT DEVELOPMENTS

There is one higher version of this proposed paper which is StyleGAN 3 which is proposed in 2021 [15]. The limitations of conventional Generative Adversarial Networks (GANs) in generating images are due to their heavy reliance on absolute pixel coordinates, resulting in details seemingly fixed to specific locations instead of conforming naturally to depicted objects. It identifies careless signal processing within the network as a cause of aliasing issues, proposing small architectural changes that prevent unwanted information from compromising the hierarchical synthesis process. These modifications yield networks to StyleGAN2 in performance metrics but significantly differ in internal representations while achieving full equivariance to translation and rotation, even at subpixel scales. Such improvements hold promise for generating more consistent and predictable content, particularly beneficial for applications in video and animation [15].

REPETITION OF AUTHOR WORK

Dataset used in research paper

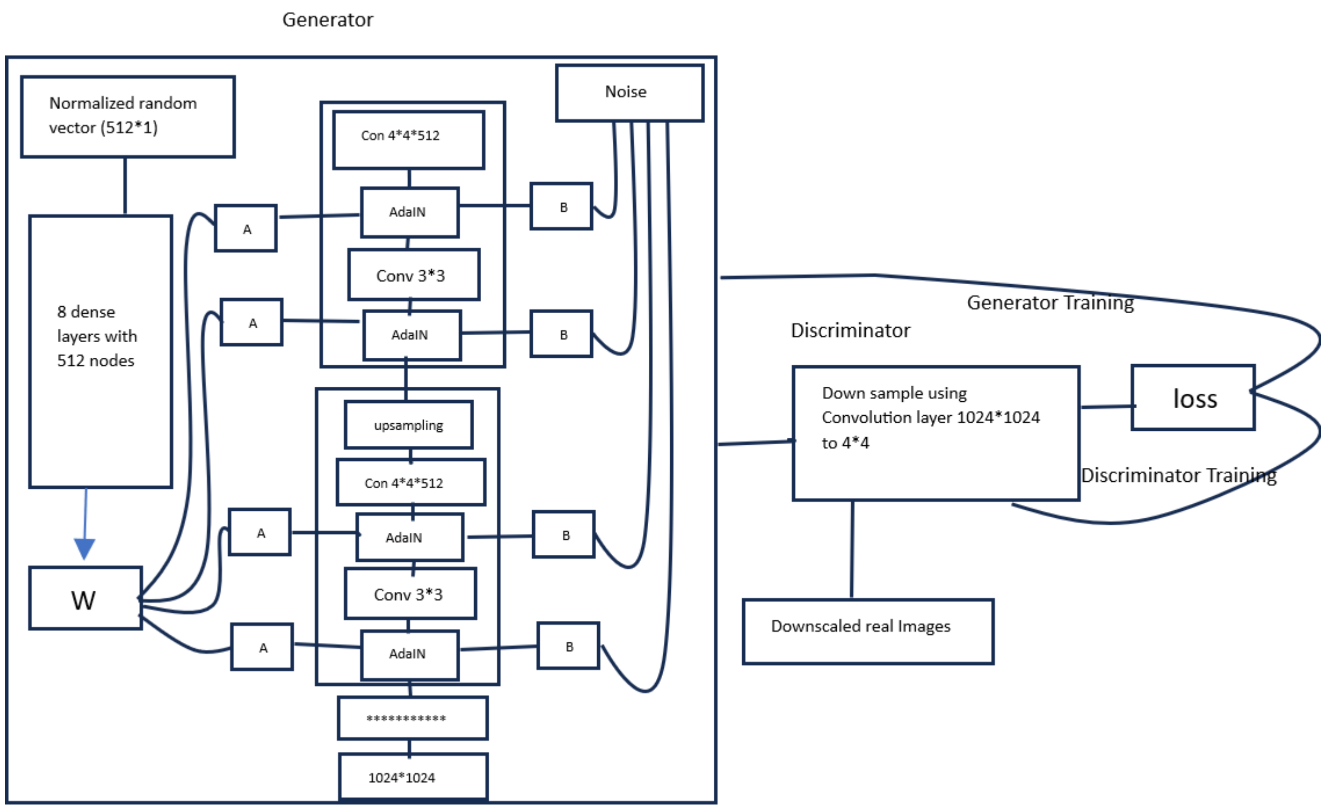
The LSUN Car dataset is a collection of images specifically focused on cars. These images have a resolution of 512x384 pixels.

Dataset used in the current paper

The "Synthetic Faces v1.0.0" dataset on Kaggle comprises 1,000 images of synthetic faces, each with a resolution of 512x512 pixels. This dataset serves as a resource for researchers and practitioners in computer vision and machine learning, providing synthetic facial images for various applications such as facial recognition, generative modeling, and image analysis tasks. The high-resolution nature of the images allows for detailed exploration and experimentation in facial feature analysis, while the synthetic generation offers controlled data for training and testing algorithms. This dataset, available on Kaggle, caters to those interested in utilizing synthetic facial data to advance research and development in the field of machine learning and computer vision.

https://www.kaggle.com/datasets/defileroff/comic-faces-paired-synthetic

Style GAN

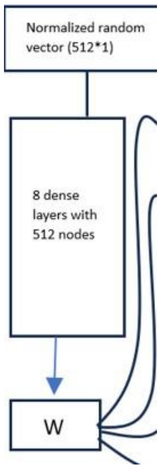


A complete architecture of Style GAN

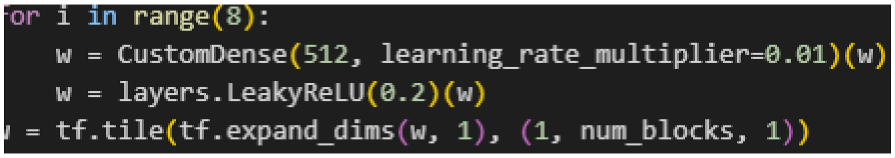
Generator

Mapping Network

The Mapping function defines a mapping network that takes a latent vector z as the input shape of batch and latent size 512, applies 8 layers of equalized dense units with LeakyReLU activations, and produces an intermediate latent representation w. The purpose of this network is to transform the input latent vector into a higher-level latent space that can be further utilized in the generation process.


Normalized random vector (512*1)
8 dense layers with 512 nodes
W

Mapping Network

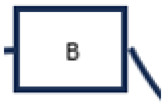


```
or i in range(8):
    w = CustomDense(512, learning_rate_multiplier=0.01)(w)
    w = layers.LeakyReLU(0.2)(w)
 = tf.tile(tf.expand_dims(w, 1), (1, num_blocks, 1))
```

Source code for Mapping Network

Noise Layer

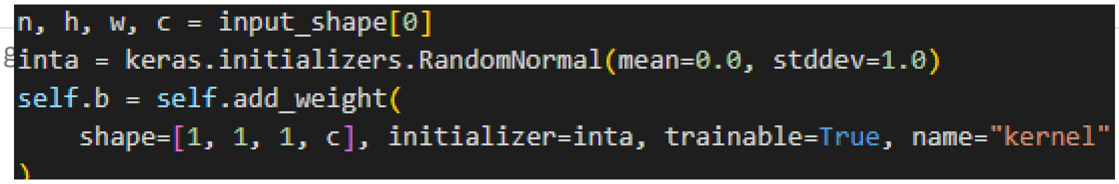
The custom noise layer neural network. where the standard devotion of all weights falls from 0 to 1 which helps to initialize the weights in a way that allows for effective learning during the trainingprocess and varies with the bias based on channels.



output = input + bias * noise.

Noise layer

Source code for the Noise Layer



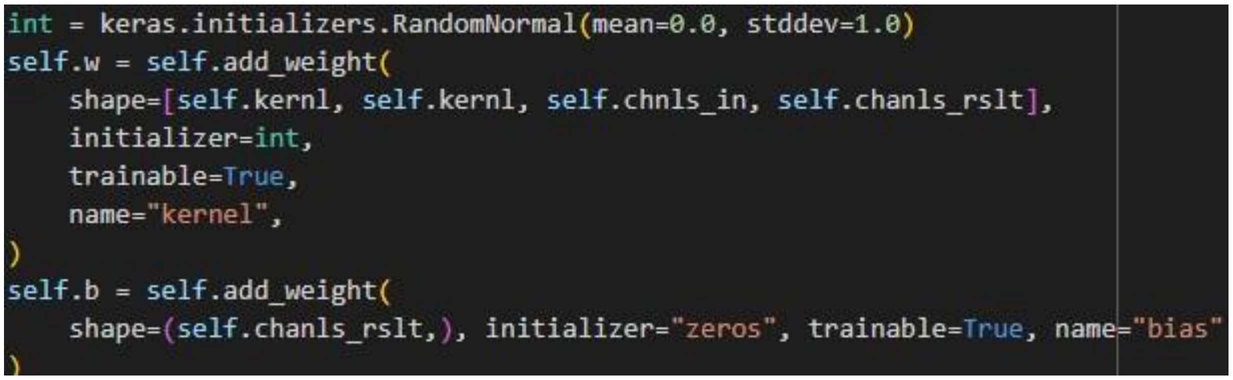
```
n, h, w, c = input_shape[0]
inta = keras.initializers.RandomNormal(mean=0.0, stddev=1.0)
self.b = self.add_weight(
    shape=[1, 1, 1, c], initializer=inta, trainable=True, name="kernel"
)
```

Convolution Layer

Build a custom convolution layer where the standard devotion of all weights falls from 0 to 1 which helps to initialize the weights in a way that allows for effective learning during the training process and bias is initialized as zero. The scale factor is calculated as the square root of the gain divided by the fadeing value, where fade is the product of kernel size, kernel size, and input channels.

Output = Convolution network (scale factor * weight) + bias.

Custom convolution Layer



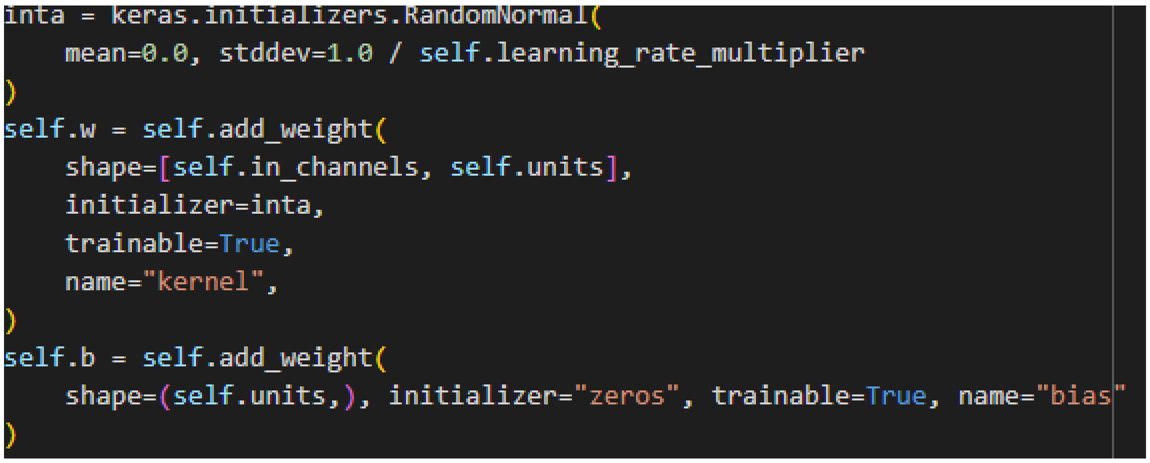
```
int = keras.initializers.RandomNormal(mean=0.0, stddev=1.0)
self.w = self.add_weight(
    shape=[self.kernl, self.kernl, self.chnls_in, self.chanls_rslt],
    initializer=int,
    trainable=True,
    name="kernel",
)
self.b = self.add_weight(
    shape=(self.chanls_rslt,), initializer="zeros", trainable=True, name="bias"
)
```

Source code for Custom convolution Layer

Dense Layer

Build a custom dense layer where the standard devotion of all weights falls from 0 to 1 which helps to initialize the weights in a way that allows for effective learning during the training processand bias is initialized as zero. and the output is controlled by the learning rate which is multipliedby the learning rate.

Output = channel of color (it depends on the fade of color) * weight +bias



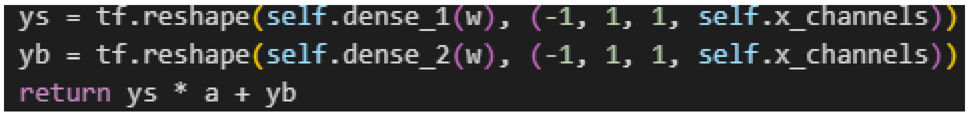
```
inta = keras.initializers.RandomNormal(
    mean=0.0, stddev=1.0 / self.learning_rate_multiplier
)
self.w = self.add_weight(
    shape=[self.in_channels, self.units],
    initializer=inta,
    trainable=True,
    name="kernel",
)
self.b = self.add_weight(
    shape=(self.units,), initializer="zeros", trainable=True, name="bias"
)
```

Source code for Dense Layer

Adaptive Instance Normalization

The adaptive Instance Normalization layer is used to adjust the style of an input feature map (x) based on the style parameters. The input feature vector from noise and style vector are sent to two different custom dense works and reshaped which results to give two styling parameters. Scaling the input using these two parameters adjusts the style of the input feature map based on the learnedstyle parameters.

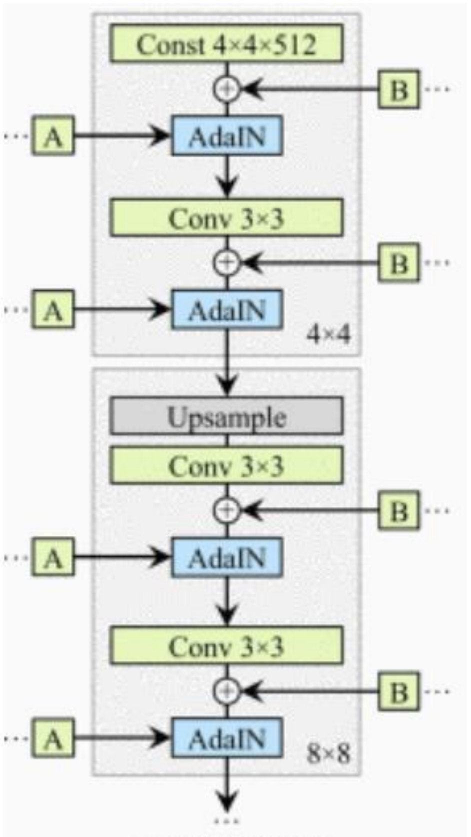
Adaptive instant normalization



```
ys = tf.reshape(self.dense_1(w), (-1, 1, 1, self.x_channels))
yb = tf.reshape(self.dense_2(w), (-1, 1, 1, self.x_channels))
return ys * a + yb
```

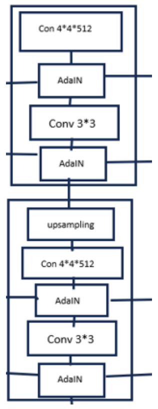
Source code for adaptive instant normalization

Progressive growth of the network

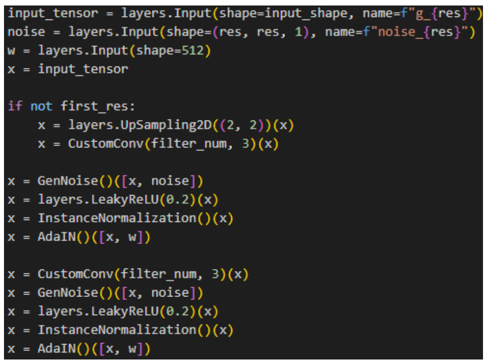


(a) StyleGAN

At each stage, a noise input layer is created to introduce random variations into the generator's output. This noise input is specific to the current resolution. An RGB conversion layer is created to convert the activation of the generator block to RGB output. This layer is responsible for generating the final image at the current resolution. If the current stage is not the base stage, up- sampling and a convolution layer are applied to increase the resolution of the generator's output. This allows the generator to generate more detailed images as the resolution increases. Noise is added to the current flow. This helps introduce additional variations and stochasticity to the generated images. LeakyReLU activation is applied to introduce non-linearity and help the network learn more complex patterns and structures. Instance normalization is applied to normalize the activations, which can help with stabilizing training and improving the quality of generated images. Adaptive Instance Normalization is applied using the activation and the input latent vector w. AdaIN dynamically adjusts the normalization parameters based on the input latent vector, allowingthe network to control style and appearance based on the input.



Progressive growth of the network

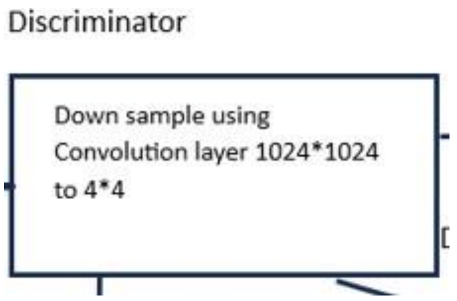


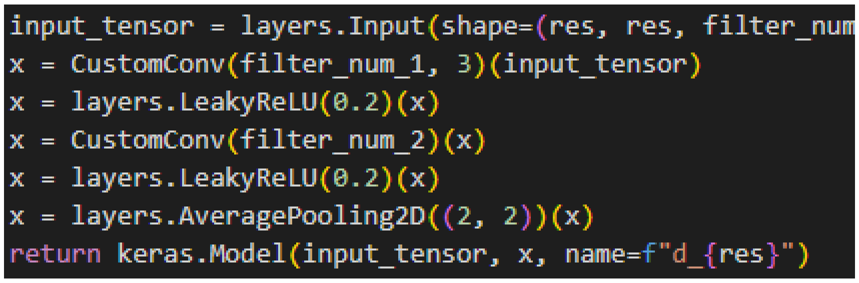
```
input_tensor = layers.Input(shape=input_shape, name=f"g_{res}")
noise = layers.Input(shape=(res, res, 1), name=f"noise_{res}")
w = layers.Input(shape=512)
x = input_tensor

if not first_res:
    x = layers.UpSampling2D((2, 2))(x)
    x = CustomConv(filter_num, 3)(x)

x = GenNoise()([x, noise])
x = layers.LeakyReLU(0.2)(x)
x = InstanceNormalization()(x)
x = AdaIN()([x, w])

x = CustomConv(filter_num, 3)(x)
x = GenNoise()([x, noise])
x = layers.LeakyReLU(0.2)(x)
x = InstanceNormalization()(x)
x = AdaIN()([x, w])
```

Source code for Progressive growth of the network

Discriminator

The discriminator block is applied from higher resolution to lower resolution. In the first stage, it takes the input tensor calculates the average standard deviation across groupedsamples in the input tensor and then replicates it to match the shape of the input tensor, an equalized convolution layer, LeakyReLU activation, flattening, two equalized dense layers with LeakyReLU activations, and a final equalized dense layer to output a single value representing the decision of real or fake. In the higher resolution, the input is passed to a series of custom convolution layers and activationfunctions and down samples the resolution using average pooling.



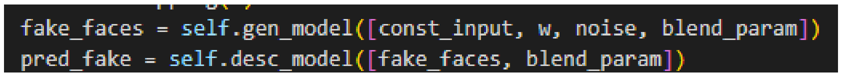


```
input_tensor = layers.Input(shape=(res, res, filter_num
x = CustomConv(filter_num_1, 3)(input_tensor)
x = layers.LeakyReLU(0.2)(x)
x = CustomConv(filter_num_2)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.AveragePooling2D((2, 2))(x)
return keras.Model(input_tensor, x, name=f"d_{res}")
```

Source code for the discriminator

GAN

It is responsible for growing discriminators and generators concerning the target resolution. In training face the generator generates fake images using the generator, computes the generator loss based on the Wasserstein loss, and calculates and applies gradients to update the generator's trainable weights. It then computes the discriminator loss based on the Wasserstein loss, gradient penalty, and drift loss, and applies gradients to update the discriminator's trainable weights which control the discriminator decisions. Model inference by generating images based on input style codes, latent vectors, noise, batch size,and alpha values. It uses the generator to generate images and applies and transforms the pixels ofthe image to the valid range.

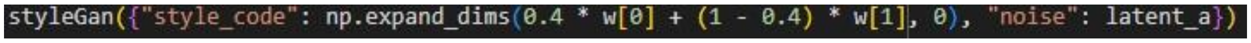


```
fake_faces = self.gen_model([const_input, w, noise, blend_param])
pred_fake = self.desc_model([fake_faces, blend_param])
```

GAN Architecture

Style mixing

Mix two style vectors using linear interpolation. This will create a style-mixed mapping network and an additional dimension. Input this to generated StyleGan weights to create a new style.A mixed style code is computed by linearly interpolating between the style codes and using blend. This interpolation combines the styles of the two vectors and introduces a new dimensionrepresenting the mixed style. The mixed-style vector and expanded noise tensors are used as inputs to the model, generating mixed images. These images reflect the combined style of the original two images, allowing for the exploration of new visual styles.

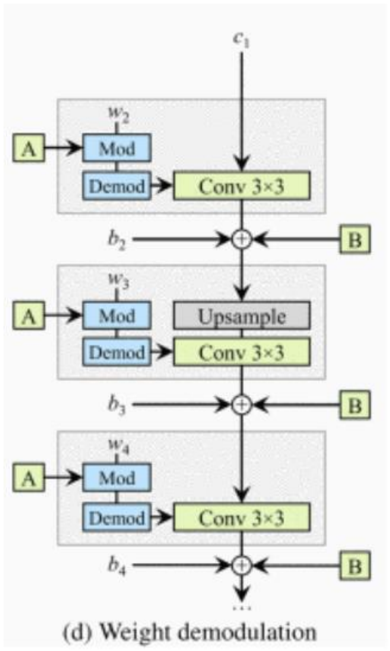


```
styleGan({"style_code": np.expand_dims(0.4 * w[0] + (1 - 0.4) * w[1], 0), "noise": latent_a})
```

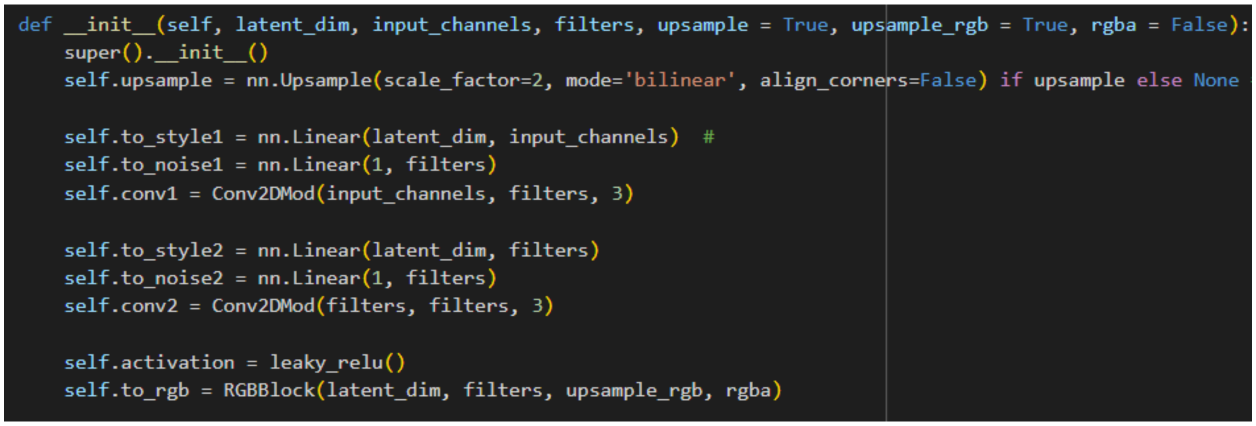
Style Mixing

Style GAN 2

Generator

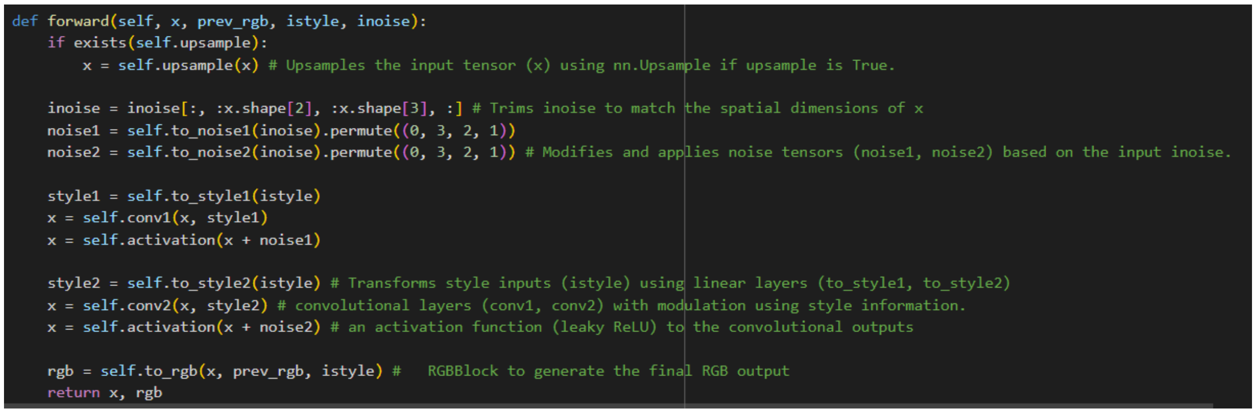


The GeneratorBlock class is constructed to provide image generation within a neural network. It's designed to handle crucial components essential for this task. The initialization process begins by configuring the block's structure based on specified parameters. It sets up a series of layers and functionalities: starting with a potential upsampling layer, followed by style mapping using linear transformations from the latent space, noise injection layers, convolutional layers with modulation, activation function, and finally, a block for generating the RGB output. These components collectively enable the GeneratorBlock to process latent space information, apply modulation and noise injection, perform convolutions, and produce the final RGB output necessary for image generation within the neural network.

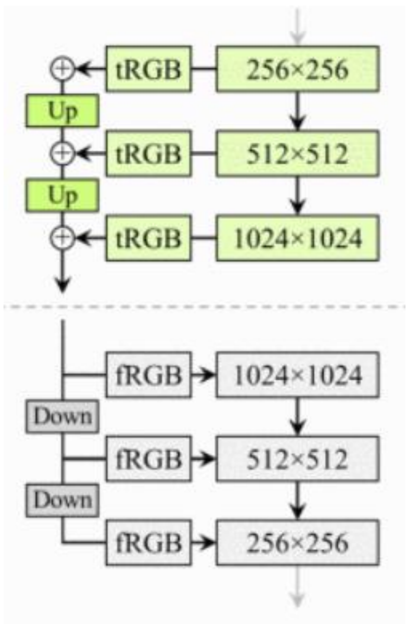


```
def __init__(self, latent_dim, input_channels, filters, upsample = True, upsample_rgb = True, rgba = False):
    super().__init__()
    self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=False) if upsample else None

    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)
```

The forward method in the GeneratorBlock class has a sequence of crucial operations within a Generative Adversarial Network (GAN). It begins by potentially upsampling the input tensor to enhance spatial resolution and then incorporates noise through linear transformations, reshaping it to match the tensor's dimensions. Subsequently, stylistic information undergoes processing via linear layers to generate style vectors, utilized in modulation operations within convolutional layers. These convolutions, along with an activation function, introduce non-linearity and feature enhancement. Finally, the method employs an RGBBlock to combine processed tensors with previous RGB information and stylistic details, ultimately generating the final RGB output, producing meaningful image representations essential for GAN-based image generation.

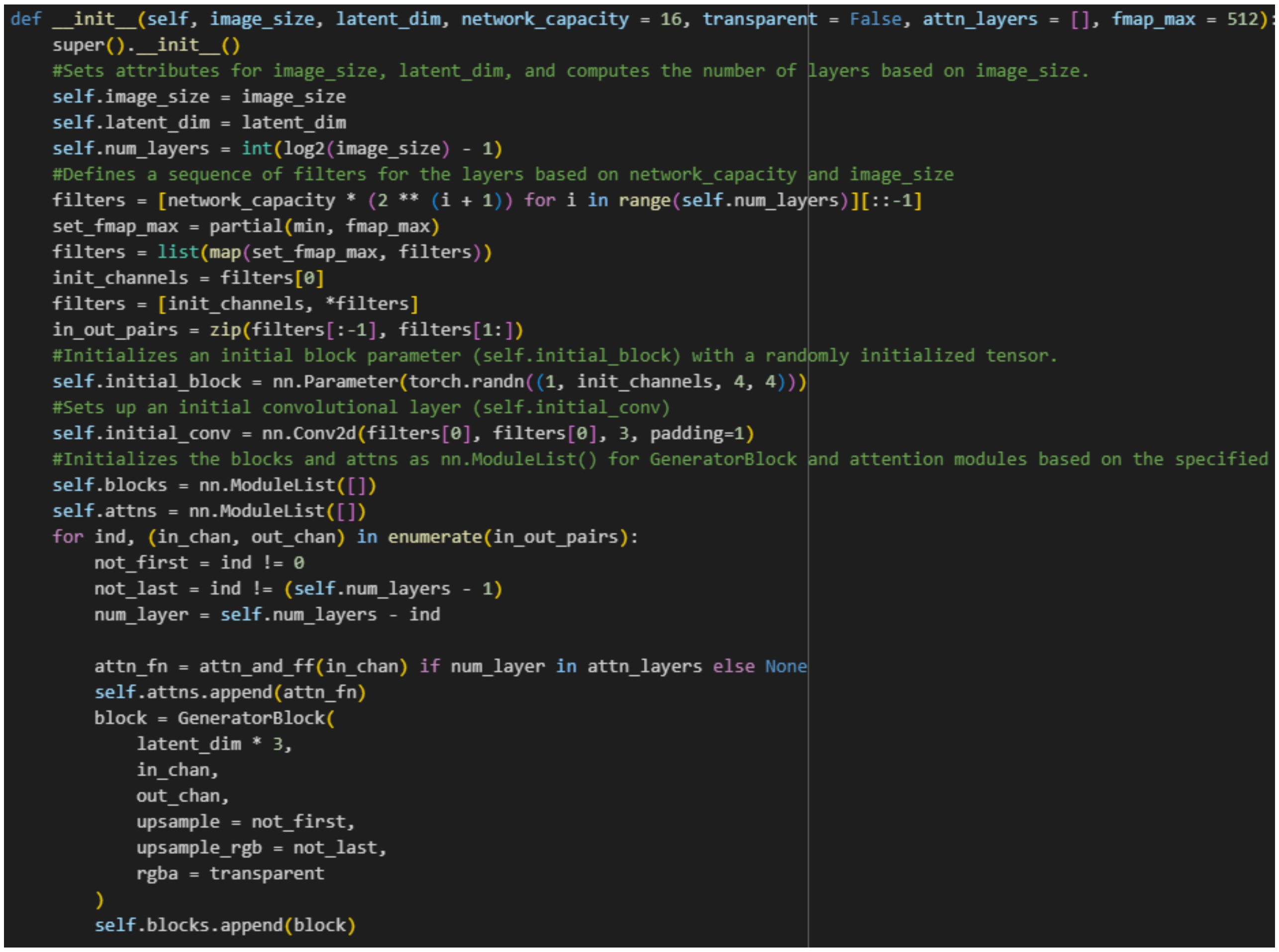


```python
def forward(self, x, prev_rgb, istyle, inoise):
    if exists(self.upsample):
        x = self.upsample(x) # Upsamples the input tensor (x) using nn.Upsample if upsample is True.

    inoise = inoise[:, :x.shape[2], :x.shape[3], :] # Trims inoise to match the spatial dimensions of x
    noise1 = self.to_noise1(inoise).permute((0, 3, 2, 1))
    noise2 = self.to_noise2(inoise).permute((0, 3, 2, 1)) # Modifies and applies noise tensors (noise1, noise2) based on the input inoise.

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

    style2 = self.to_style2(istyle) # Transforms style inputs (istyle) using linear layers (to_style1, to_style2)
    x = self.conv2(x, style2) # convolutional layers (conv1, conv2) with modulation using style information.
    x = self.activation(x + noise2) # an activation function (leaky ReLU) to the convolutional outputs

    rgb = self.to_rgb(x, prev_rgb, istyle) #   RGBBlock to generate the final RGB output
    return x, rgb
```



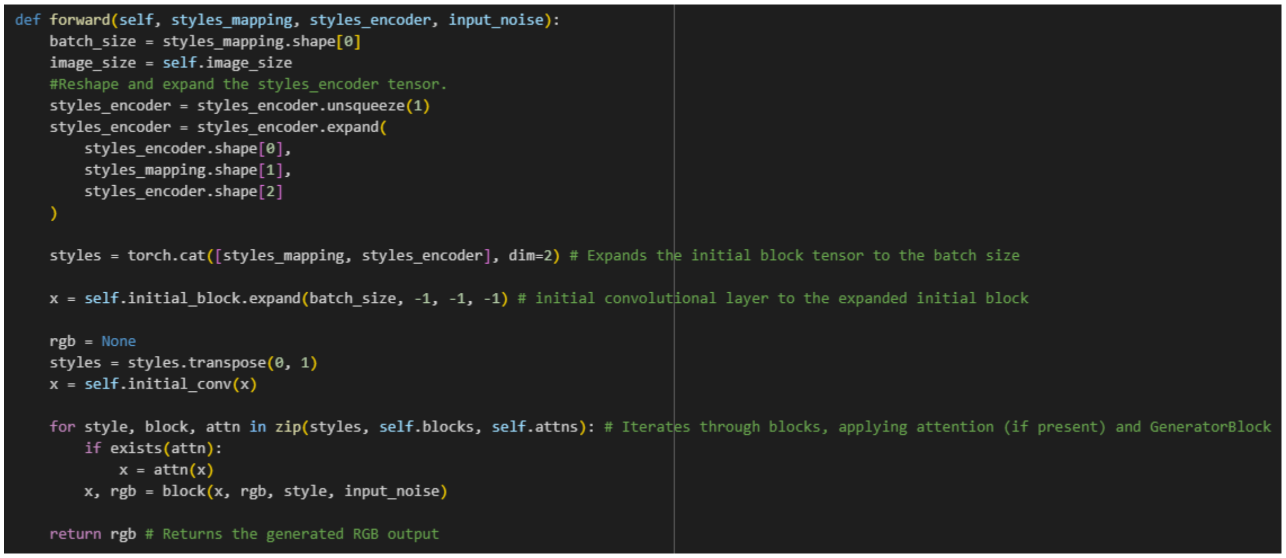
(b) Input/output skips

The initialization method within the Generator class is responsible for setting up the architecture components. It starts by storing essential parameters like image size, latent dimension, network capacity, and attention layers. It calculates a sequence of filter sizes based on the network capacity and image size, configuring the initial block and convolutional layer accordingly. For each layer in the network, it initializes potential attention modules based on specified attention layers and creates GeneratorBlock instances, considering input and output channels along with flags for upsampling and RGBA representation. These instances and attention modules are stored within the generator, forming the foundational structure for subsequent image generation in the GAN.



```python
def __init__(self, image_size, latent_dim, network_capacity = 16, transparent = False, attn_layers = [], fmap_max = 512):
    super().__init__()
    #Sets attributes for image_size, latent_dim, and computes the number of layers based on image_size.
    self.image_size = image_size
    self.latent_dim = latent_dim
    self.num_layers = int(log2(image_size) - 1)
    #Defines a sequence of filters for the layers based on network_capacity and image_size
    filters = [network_capacity * (2 ** (i + 1)) for i in range(self.num_layers)][::-1]
    set_fmap_max = partial(min, fmap_max)
    filters = list(map(set_fmap_max, filters))
    init_channels = filters[0]
    filters = [init_channels, *filters]
    in_out_pairs = zip(filters[:-1], filters[1:])
    #Initializes an initial block parameter (self.initial_block) with a randomly initialized tensor.
    self.initial_block = nn.Parameter(torch.randn((1, init_channels, 4, 4)))
    #Sets up an initial convolutional layer (self.initial_conv)
    self.initial_conv = nn.Conv2d(filters[0], filters[0], 3, padding=1)
    #Initializes the blocks and attns as nn.ModuleList() for GeneratorBlock and attention modules based on the specified
    self.blocks = nn.ModuleList([])
    self.attns = nn.ModuleList([])
    for ind, (in_chan, out_chan) in enumerate(in_out_pairs):
        not_first = ind != 0
        not_last = ind != (self.num_layers - 1)
        num_layer = self.num_layers - ind

        attn_fn = attn_and_ff(in_chan) if num_layer in attn_layers else None
        self.attns.append(attn_fn)
        block = GeneratorBlock(
            latent_dim * 3,
            in_chan,
            out_chan,
            upsample = not_first,
            upsample_rgb = not_last,
            rgba = transparent
        )
        self.blocks.append(block)
```

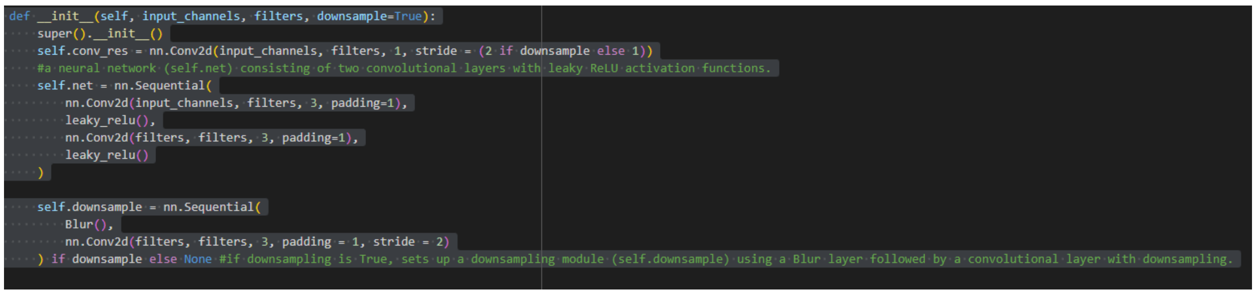
The forward method in the Generator class serves as the engine for creating RGB images by processing input styles, encoded information, and noise through a sequence of blocks and potential attention mechanisms. Initially, it reshapes and combines the styles and encoder information, preparing them for subsequent operations. Through an iterative process, it expands the initial block tensor to fit the batch size and applies the initial convolutional layer. Then, in a loop iterating over styles, blocks, and possible attention modules, it systematically modulates the input data using GeneratorBlock instances, applying attention if specified, and generating RGB outputs. This orchestrated approach culminates in the production of the final RGB image output, signifying the culmination of the generator's transformative process on the input data, encapsulating modulation, and synthesis at various stages to generate coherent visual representations.



```python
def forward(self, styles_mapping, styles_encoder, input_noise):
    batch_size = styles_mapping.shape[0]
    image_size = self.image_size
    #Reshape and expand the styles_encoder tensor.
    styles_encoder = styles_encoder.unsqueeze(1)
    styles_encoder = styles_encoder.expand(
        styles_encoder.shape[0],
        styles_mapping.shape[1],
        styles_encoder.shape[2]
    )

    styles = torch.cat([styles_mapping, styles_encoder], dim=2) # Expands the initial block tensor to the batch size

    x = self.initial_block.expand(batch_size, -1, -1, -1) # initial convolutional layer to the expanded initial block

    rgb = None
    styles = styles.transpose(0, 1)
    x = self.initial_conv(x)

    for style, block, attn in zip(styles, self.blocks, self.attns): # Iterates through blocks, applying attention (if present) and GeneratorBlock
        if exists(attn):
            x = attn(x)
        x, rgb = block(x, rgb, style, input_noise)

    return rgb # Returns the generated RGB output
```

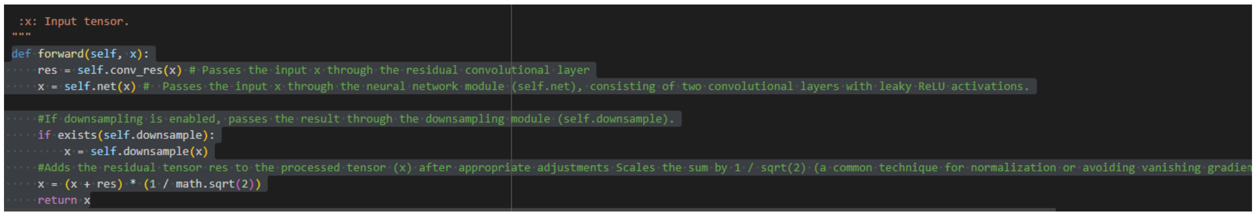
Discriminator

This initialization process revolves around critical parameters like input channels and filter specifications. It begins by establishing a residual convolutional layer (self.conv_res) that processes input features, potentially incorporating downsampling based on the provided downsample flag. Subsequently, it constructs a convolutional neural network (self.net) via nn.Sequential, composed of two consecutive convolutional layers with 3x3 kernels followed by leaky ReLU activation functions, enabling effective feature extraction and introducing non-linearity. Additionally, a downsampling module (self.downsample) is conditionally set up, employing a Blur layer and a 3x3 convolutional layer for spatial dimension reduction, an operation crucial for feature extraction within the discriminator network.

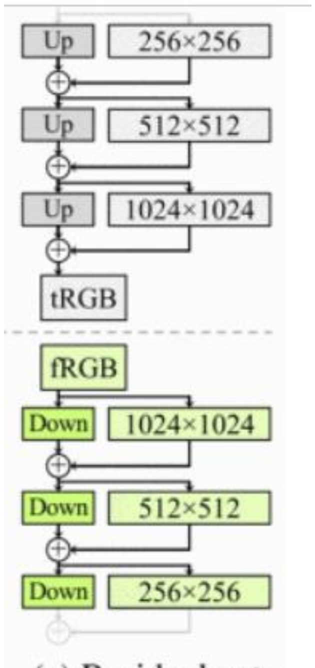


```python
def __init__(self, input_channels, filters, downsample=True):
    super().__init__()
    self.conv_res = nn.Conv2d(input_channels, filters, 1, stride = (2 if downsample else 1))
    #a neural network (self.net) consisting of two convolutional layers with leaky ReLU activation functions.
    self.net = nn.Sequential(
        nn.Conv2d(input_channels, filters, 3, padding=1),
        leaky_relu(),
        nn.Conv2d(filters, filters, 3, padding=1),
        leaky_relu()
    )

    self.downsample = nn.Sequential(
        Blur(),
        nn.Conv2d(filters, filters, 3, padding = 1, stride = 2)
    ) if downsample else None #if downsampling is True, sets up a downsampling module (self.downsample) using a Blur layer followed by a convolutional layer with downsampling.
```

The forward method within the DiscriminatorBlock class orchestrates the transformation of input tensors by sequentially processing them through essential components. Initially, the input tensor undergoes convolutional operations via the residual convolutional layer (self.conv_res). Simultaneously, the neural network module (self.net), comprised of two convolutional layers with leaky ReLU activations, extracts intricate features from the input. Optionally, if enabled, the downsampling module (self.downsample) reduces spatial dimensions in the tensor. The method adeptly integrates the residual tensor with the processed tensor using a residual connection, scaled by 1 / sqrt(2) to maintain stability and mitigate gradient issues. This comprehensive process encapsulates hierarchical feature extraction, spatial manipulation, and residual connections vital for discerning between real and generated images within the discriminator network, culminating in the transformed output tensor.

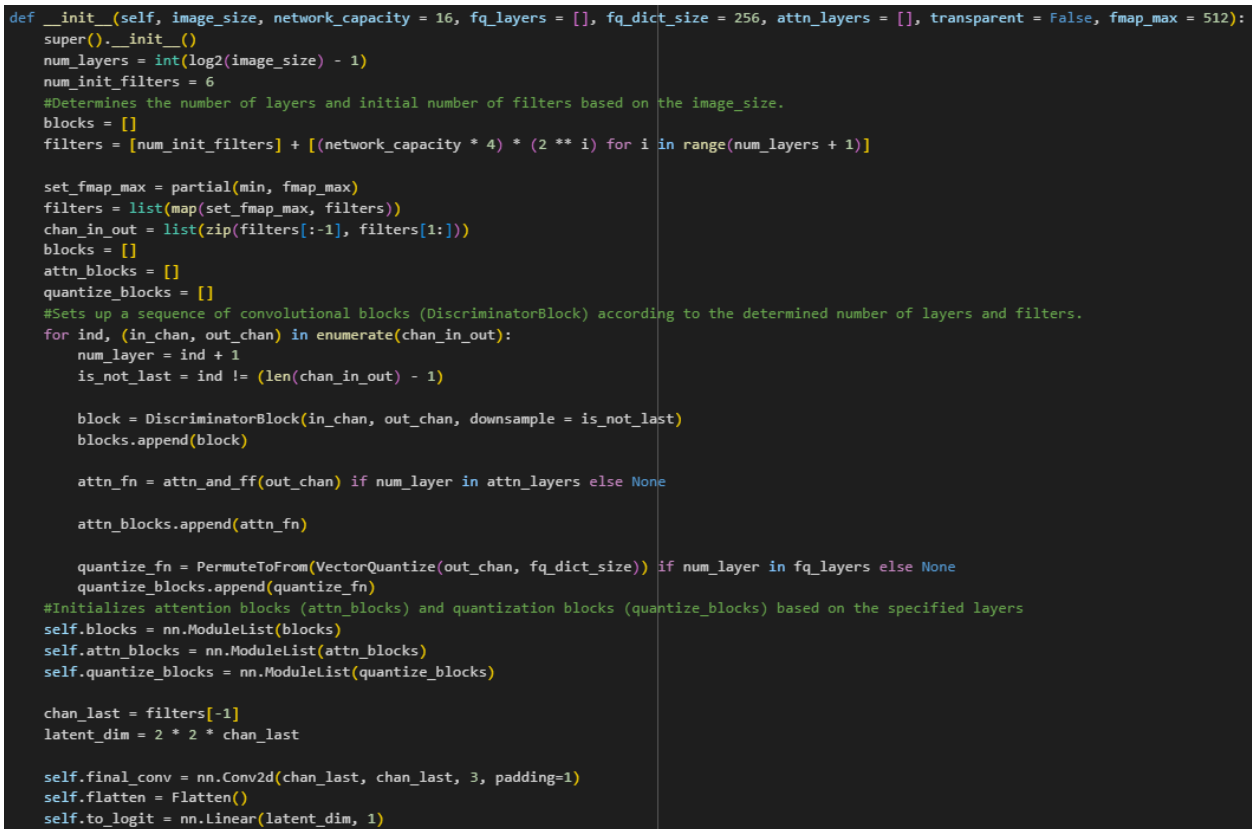


```
 :x: Input tensor.
"""
def forward(self, x):
    res = self.conv_res(x) # Passes the input x through the residual convolutional layer
    x = self.net(x) #  Passes the input x through the neural network module (self.net), consisting of two convolutional layers with leaky ReLU activations.

    #If downsampling is enabled, passes the result through the downsampling module (self.downsample).
    if exists(self.downsample):
        x = self.downsample(x)
    #Adds the residual tensor res to the processed tensor (x) after appropriate adjustments Scales the sum by 1 / sqrt(2) (a common technique for normalization or avoiding vanishing gradien
    x = (x + res) * (1 / math.sqrt(2))
    return x
```



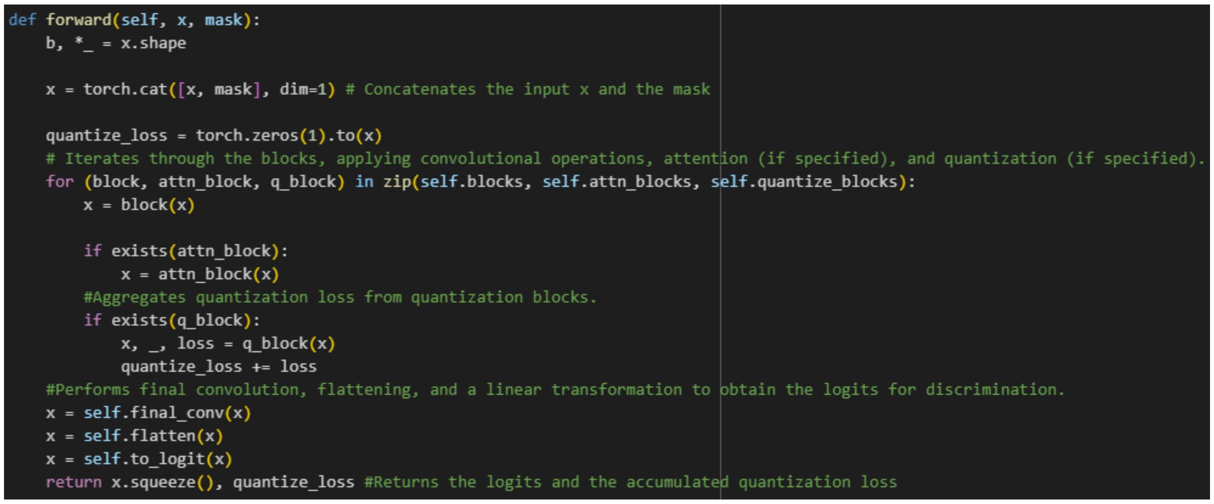
(c) Residual nets

The discriminator first establishes the network's architecture by determining the number of layers based on the input image size and initializing an initial number of filters accordingly. Subsequently, it constructs a sequence of convolutional blocks (DiscriminatorBlock instances) following the determined layer count and filter sizes. Additionally, this method handles attention mechanisms by setting up attention blocks using attn_and_ff functions, as specified in attn_layers, and initializes quantization blocks utilizing VectorQuantize and PermuteToFrom functions for layers listed in fq_layers, along with the specified dictionary size for quantization. Moreover, it organizes these blocks into ModuleLists (self.blocks, self.attn_blocks, and self.quantize_blocks) to manage the discriminator's architecture. Finally, the method defines essential layers like the final convolutional layer (self.final_conv), a flattening operation (self.flatten), and a linear transformation (self.to_logit) responsible for generating logits used in the discrimination process.



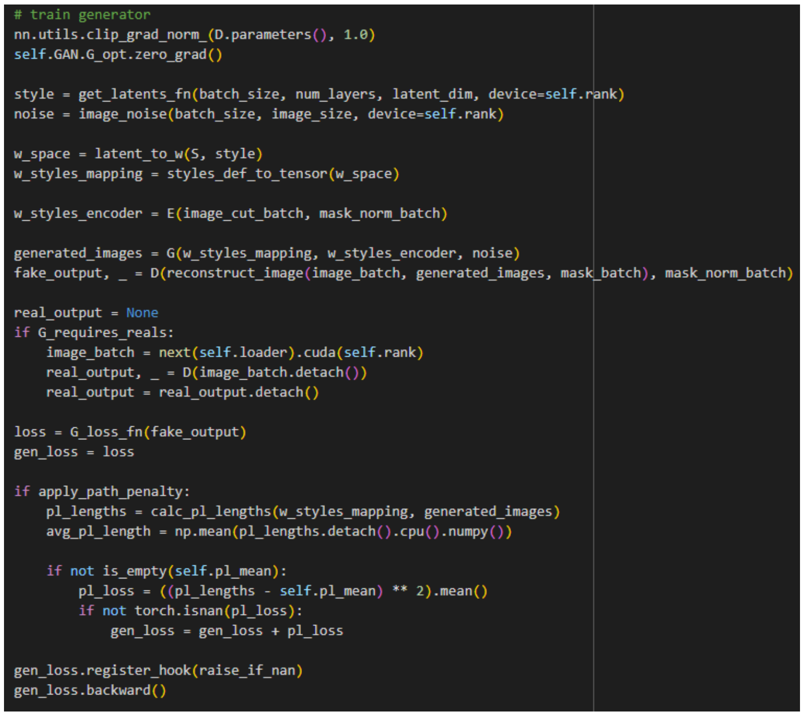
```python
def __init__(self, image_size, network_capacity = 16, fq_layers = [], fq_dict_size = 256, attn_layers = [], transparent = False, fmap_max = 512):
    super().__init__()
    num_layers = int(log2(image_size) - 1)
    num_init_filters = 6
    #Determines the number of layers and initial number of filters based on the image_size.
    blocks = []
    filters = [num_init_filters] + [(network_capacity * 4) * (2 ** i) for i in range(num_layers + 1)]

    set_fmap_max = partial(min, fmap_max)
    filters = list(map(set_fmap_max, filters))
    chan_in_out = list(zip(filters[:-1], filters[1:]))
    blocks = []
    attn_blocks = []
    quantize_blocks = []
    #Sets up a sequence of convolutional blocks (DiscriminatorBlock) according to the determined number of layers and filters.
    for ind, (in_chan, out_chan) in enumerate(chan_in_out):
        num_layer = ind + 1
        is_not_last = ind != (len(chan_in_out) - 1)

        block = DiscriminatorBlock(in_chan, out_chan, downsample = is_not_last)
        blocks.append(block)

        attn_fn = attn_and_ff(out_chan) if num_layer in attn_layers else None

        attn_blocks.append(attn_fn)

        quantize_fn = PermuteToFrom(VectorQuantize(out_chan, fq_dict_size)) if num_layer in fq_layers else None
        quantize_blocks.append(quantize_fn)
    #Initializes attention blocks (attn_blocks) and quantization blocks (quantize_blocks) based on the specified layers
    self.blocks = nn.ModuleList(blocks)
    self.attn_blocks = nn.ModuleList(attn_blocks)
    self.quantize_blocks = nn.ModuleList(quantize_blocks)

    chan_last = filters[-1]
    latent_dim = 2 * 2 * chan_last

    self.final_conv = nn.Conv2d(chan_last, chan_last, 3, padding=1)
    self.flatten = Flatten()
    self.to_logit = nn.Linear(latent_dim, 1)
```

The forward function in the Discriminator class is the processing of input tensors by sequentially passing them through defined blocks, attending to specific layers if attention mechanisms are specified, and incorporating loss of processes where applicable. It iterates through each block, executing convolutional operations and accumulating potential losses from specified blocks. The method then conducts final convolutional, flattening, and linear transformation operations, producing logits crucial for discrimination while concurrently aggregating the loss. This comprehensive approach encapsulates the entire process, providing both discriminative output and loss in a singular function call, enabling an assessment of the discriminator's functionality.



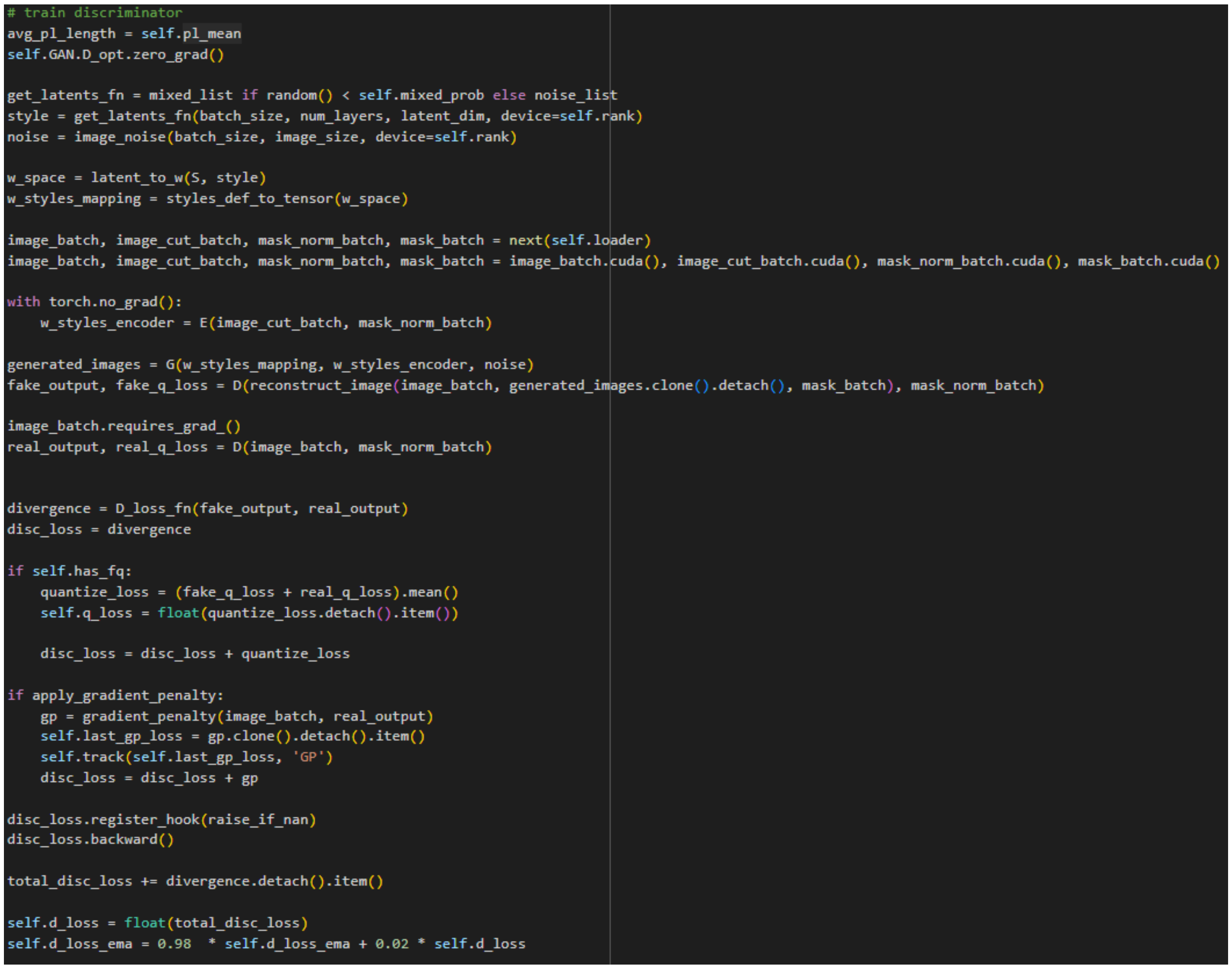
```
def forward(self, x, mask):
    b, *_ = x.shape

    x = torch.cat([x, mask], dim=1) # Concatenates the input x and the mask

    quantize_loss = torch.zeros(1).to(x)
    # Iterates through the blocks, applying convolutional operations, attention (if specified), and quantization (if specified).
    for (block, attn_block, q_block) in zip(self.blocks, self.attn_blocks, self.quantize_blocks):
        x = block(x)

        if exists(attn_block):
            x = attn_block(x)
        #Aggregates quantization loss from quantization blocks.
        if exists(q_block):
            x, _, loss = q_block(x)
            quantize_loss += loss
    #Performs final convolution, flattening, and a linear transformation to obtain the logits for discrimination.
    x = self.final_conv(x)
    x = self.flatten(x)
    x = self.to_logit(x)
    return x.squeeze(), quantize_loss #Returns the logits and the accumulated quantization loss
```

The generator training begins by preparing inputs for the generator, including style vectors and noise. These vectors are used to generate fake images through the generator network, which are then evaluated by the discriminator to calculate a loss. This loss is subsequently used for backpropagation to update the generator's weights to improve its ability to produce realistic images. Optionally, it evaluates real images through the discriminator as well. Additionally, there's an optional step for path length regularization, contributing to the generator's loss computation if conditions are met. Overall, this snippet showcases a crucial part of the GAN training loop, wherein the generator learns to produce more realistic images by optimizing its parameters based on the discriminator's feedback and the predefined loss function.



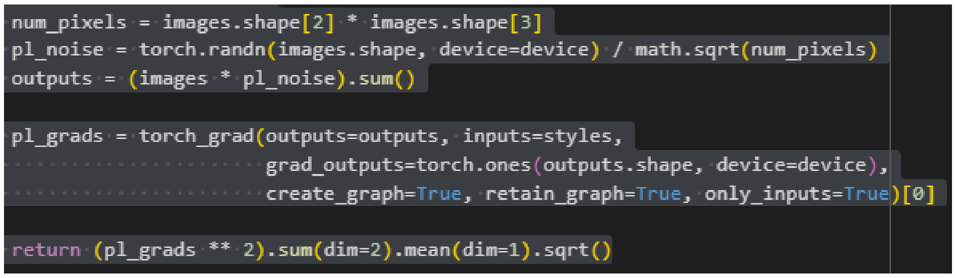
```
# train generator
nn.utils.clip_grad_norm_(D.parameters(), 1.0)
self.GAN.G_opt.zero_grad()

style = get_latents_fn(batch_size, num_layers, latent_dim, device=self.rank)
noise = image_noise(batch_size, image_size, device=self.rank)

w_space = latent_to_w(S, style)
w_styles_mapping = styles_def_to_tensor(w_space)

w_styles_encoder = E(image_cut_batch, mask_norm_batch)

generated_images = G(w_styles_mapping, w_styles_encoder, noise)
fake_output, _ = D(reconstruct_image(image_batch, generated_images, mask_batch), mask_norm_batch)

real_output = None
if G_requires_reals:
    image_batch = next(self.loader).cuda(self.rank)
    real_output, _ = D(image_batch.detach())
    real_output = real_output.detach()

loss = G_loss_fn(fake_output)
gen_loss = loss

if apply_path_penalty:
    pl_lengths = calc_pl_lengths(w_styles_mapping, generated_images)
    avg_pl_length = np.mean(pl_lengths.detach().cpu().numpy())

    if not is_empty(self.pl_mean):
        pl_loss = ((pl_lengths - self.pl_mean) ** 2).mean()
        if not torch.isnan(pl_loss):
            gen_loss = gen_loss + pl_loss

gen_loss.register_hook(raise_if_nan)
gen_loss.backward()
```

The process initiates by zeroing out the gradients in the discriminator optimizer. It then proceeds to prepare the inputs for the discriminator, including style vectors and noise, before generating fake images through the generator network using these inputs. These generated images, alongside real images, are evaluated by the discriminator to compute a loss, typically measuring the discrepancy between the discriminator's predictions for real and generated images. Optional components, such as loss or gradient penalty , will contribute to the overall discriminator loss. The computed loss is used for backpropagation, updating the discriminator's weights to improve its ability to distinguish between real and generated images. Additionally, various tracking and monitoring mechanisms are implemented, including the calculation of losses, gradient penalties, and exponential moving averages (EMAs) of losses for monitoring purposes throughout the training process.



```
# train discriminator
avg_pl_length = self.pl_mean
self.GAN.D_opt.zero_grad()

get_latents_fn = mixed_list if random() < self.mixed_prob else noise_list
style = get_latents_fn(batch_size, num_layers, latent_dim, device=self.rank)
noise = image_noise(batch_size, image_size, device=self.rank)

w_space = latent_to_w(S, style)
w_styles_mapping = styles_def_to_tensor(w_space)

image_batch, image_cut_batch, mask_norm_batch, mask_batch = next(self.loader)
image_batch, image_cut_batch, mask_norm_batch, mask_batch = image_batch.cuda(), image_cut_batch.cuda(), mask_norm_batch.cuda(), mask_batch.cuda()

with torch.no_grad():
    w_styles_encoder = E(image_cut_batch, mask_norm_batch)

generated_images = G(w_styles_mapping, w_styles_encoder, noise)
fake_output, fake_q_loss = D(reconstruct_image(image_batch, generated_images.clone().detach(), mask_batch), mask_norm_batch)

image_batch.requires_grad_()
real_output, real_q_loss = D(image_batch, mask_norm_batch)


divergence = D_loss_fn(fake_output, real_output)
disc_loss = divergence

if self.has_fq:
    quantize_loss = (fake_q_loss + real_q_loss).mean()
    self.q_loss = float(quantize_loss.detach().item())

    disc_loss = disc_loss + quantize_loss

if apply_gradient_penalty:
    gp = gradient_penalty(image_batch, real_output)
    self.last_gp_loss = gp.clone().detach().item()
    self.track(self.last_gp_loss, 'GP')
    disc_loss = disc_loss + gp

disc_loss.register_hook(raise_if_nan)
disc_loss.backward()

total_disc_loss += divergence.detach().item()

self.d_loss = float(total_disc_loss)
self.d_loss_ema = 0.98  * self.d_loss_ema + 0.02 * self.d_loss
```

The Perceptual path length is calculated by generating normalized random noise proportional to the size of the images and computes the outputs by element-wise multiplication with the generated images. Utilizing PyTorch's autograd, determines the gradients of these outputs concerning the style vectors, isolating the impact of style changes on the image generation process. The function subsequently calculates the path lengths by computing the squared norms of these gradients along the style vectors, summing across vector dimensions, calculating the mean across samples, and finally taking the square root. This calculated metric characterizes the smoothness of style variations in generating images, offering a means of regularization within the PGGAN training process.



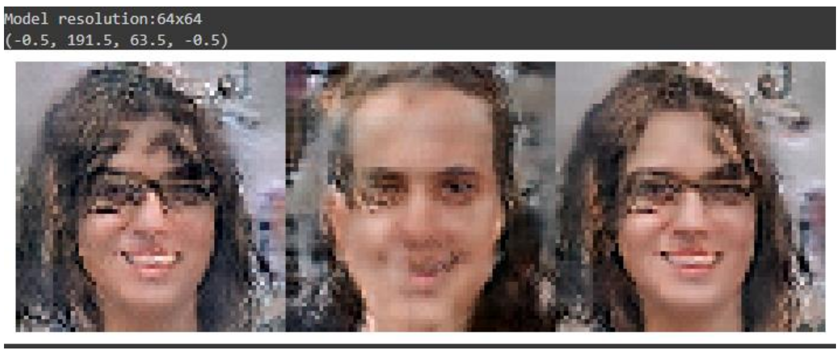
```
num_pixels = images.shape[2] * images.shape[3]
pl_noise = torch.randn(images.shape, device=device) / math.sqrt(num_pixels)
outputs = (images * pl_noise).sum()

pl_grads = torch_grad(outputs=outputs, inputs=styles,
                      grad_outputs=torch.ones(outputs.shape, device=device),
                      create_graph=True, retain_graph=True, only_inputs=True)[0]

return (pl_grads ** 2).sum(dim=2).mean(dim=1).sqrt()
```

ANALYZING RESULTS AND BETTERMENT

Style GAN

The training is performed over 100 epochs, and within each epoch, the data is divided into batches. For batch of input data and corresponding target values and performing a single optimization step on the model using that batch of data. It updates the model's parameters based on the calculated loss and the Adam optimizer algorithm. It trains the discriminator and generator models simultaneously using batches of real and fake images along with their respective labels. With a learning rate of 0.0001, sigmoid activation, a kernel size of 3, and a stride of 2. In this setting, the generated images showed less accuracy. Setting with a learning rate of 0.0003, LeakyReLU activation, a kernel size of 4, and a stride of 3. In this setting, the generated images exhibited improved clarity and were more recognizable as digits. The higher learning rate and theuse of the LeakyReLU activation function contributed to this enhancement in image quality. The evaluation also considered the training time required for each setting. The setting with a learning rate of 0.0003, LeakyReLU activation, and a larger kernel size took longer to train, approximately 8 hours. However, this additional training time resulted in the generation of clearer and more visually appealing images. Based on the analysis, it can be concluded that the second set with a learning rate of 0.0003, LeakyReLU activation, a kernel size of 4, and a stride of 3 worked better in terms of generating clearer and more distinct digit images. This configuration resulted in improved performance and yielded more satisfying results.

The StyleGAN model was employed to generate portraits of 64x64 resolution images based on the latent space. Due to the relatively low resolution, the generated images may exhibit certain limitations in terms of fine details and overall image quality. This can be attributed to the inherent constraints of working with a lower resolution, which can limit the ability to capture intricate facial features and nuances. Despite the limitations imposed by the lower resolution, the style mixing technique employed in the project yielded interesting results. By mixing the styles of the first and second images with the last image, the model was able to generate hybrid portraits that combine characteristics from different styles. Future iterations of the project could explore higher-resolution images with increased computational power to further enhance the level of detail and realism in the generated faces.



```
Model resolution:64x64
(-0.5, 191.5, 63.5, -0.5)
```

64*64 Face image style mixing

Resolution	Type	Loss
4x4	gen_loss	3.0761
4x4	disc_loss	-4.5660
8x8	gen_loss	3.0813
8x8	disc_loss	-4.3405
16x16	gen_loss	4.0706
16x16	disc_loss	-5.0096
32x32	gen_loss	5.9122
32x32	disc_loss	-7.8018
64x64	gen_loss	13.0625
64x64	disc_loss	-17.7234

Table of Loss in Generating Image

Style GNA 2

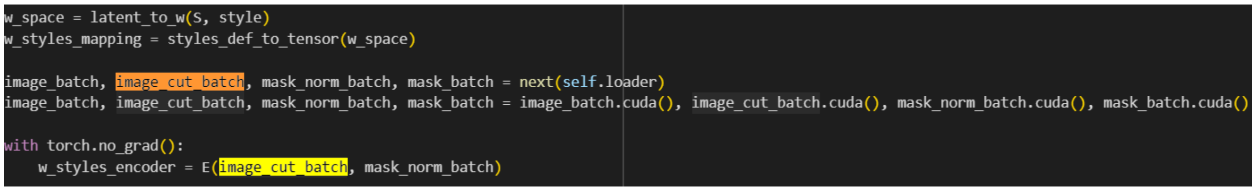
The training process involves changing the training parameters, through grid search algorithm, to comprehend their impact on model performance. Parameters such as 'image_size' are reduced to identify the resolution trade-offs between details and computational expense. 'network_capacity' changes to achieve the balance between model complexity and learning capability. The 'learning_rate' (2e-4) is fine-tuned to understand its influence on the stability and speed of model convergence during training. Parameters like 'trunc_psi' and 'aug_types' are modified to observe their impact on image diversity and model robustness. The results from this grid search approach help dissect how variations in each parameter impact the training process, convergence behavior, image quality, and overall model robustness, leading to a more informed understanding of parameter sensitivities and their effects on the model's performance.

Generator al influence over the training for image generation by 'Upsample' operation with a scale factor of 2.0 and 'bilinear' mode enhances image resolution by increasing spatial dimensions, augmenting image details. The 'Linear' layers, including 'to_style1', 'to_style2', 'to_noise1', and 'to_noise2', manipulate style and noise inputs, altering their dimensional representation to match the dataset characteristics. 'Conv2DMod' operations ('conv1', 'conv2') extract and transform features, contributing significantly to learned representations and image texture. The 'LeakyReLU' activation function with a negative slope of 0.2 introduces non-linearity, facilitating convergence by mitigating gradient vanishing issues. The 'RGBBlock' manages the final RGB output, adjusting input features via 'to_style' and 'conv' layers to ensure realistic color distribution. Additionally, the 'Blur' operation applied after upsampling smoothens feature maps, reducing noise and enhancing visual appeal. Collectively, these parameters orchestrate the model's ability to generate high-quality, diverse, and realistic images by manipulating feature representations, influencing visual fidelity, and augmenting the overall image-generation process. Adjusting these parameters allows for fine-tuning the model to produce images that closely resemble the dataset while enhancing the overall quality of the generated outputs.

The image classification in the discriminator block influences the training process by convolution layer of parameters with a kernel size of (1, 1) and a stride of (2, 2) performs downsampling, reducing the spatial dimensions of the feature maps by a factor of 2, extracting higher-level features, and enabling the network to learn more abstract representations of the input images. The 'net' Sequential module consists of two sets of Conv2d layers followed by LeakyReLU activations. The convolutional layers with kernel sizes of (3, 3) and padding of (1, 1) aim to extract hierarchical features from the input images. The LeakyReLU activation function introduces non-linearity and helps the network converge by preventing gradient saturation. The 'downsample' Sequential module utilizes the 'Blur' operation to smooth feature maps and reduce noise before further downsampling with a Conv2d layer, facilitating the extraction of more informative features at a lower resolution. Each parameter adjustment within this DiscriminatorBlock influences the network's ability to extract and process hierarchical features, aiding in the classification of images with improved accuracy and efficiency during training.

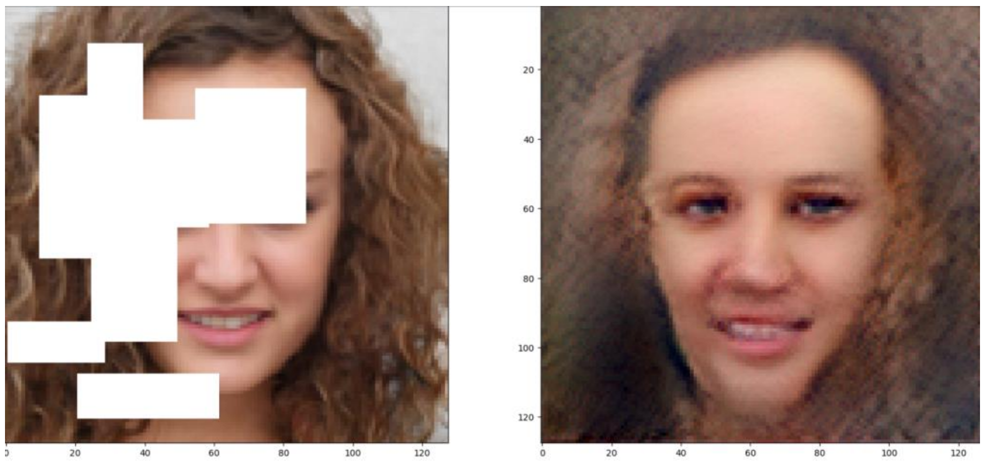
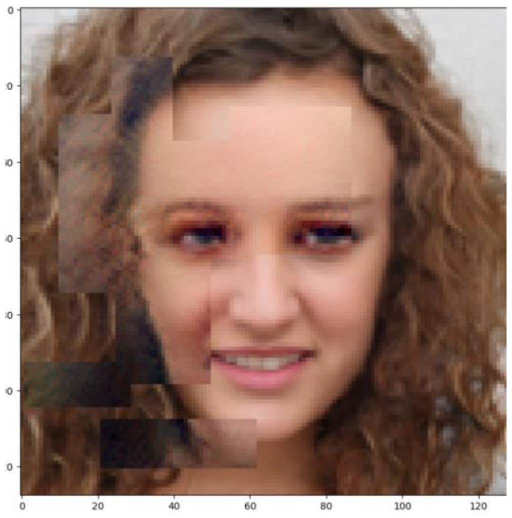
Betterment

Applying style encoder only specific position of the image which gives the image style mixing specific to the particular features like hair, eyes, etc.



```
w_space = latent_to_w(S, style)
w_styles_mapping = styles_def_to_tensor(w_space)

image_batch, image_cut_batch, mask_norm_batch, mask_batch = next(self.loader)
image_batch, image_cut_batch, mask_norm_batch, mask_batch = image_batch.cuda(), image_cut_batch.cuda(), mask_norm_batch.cuda(), mask_batch.cuda()

with torch.no_grad():
    w_styles_encoder = E(image_cut_batch, mask_norm_batch)
```

Segment extracts batches of image data and associated masks from a data loader. Within a torch.no_grad() block, an encoder model (likely denoted as E) is utilized to generate latent style representations, denoted as w_styles_encoder, from modified images (image_cut_batch) and their corresponding normalized masks (mask_norm_batch). This process aims to derive latent representations based on specific masked regions in the images, used in for image manipulation in particular region.

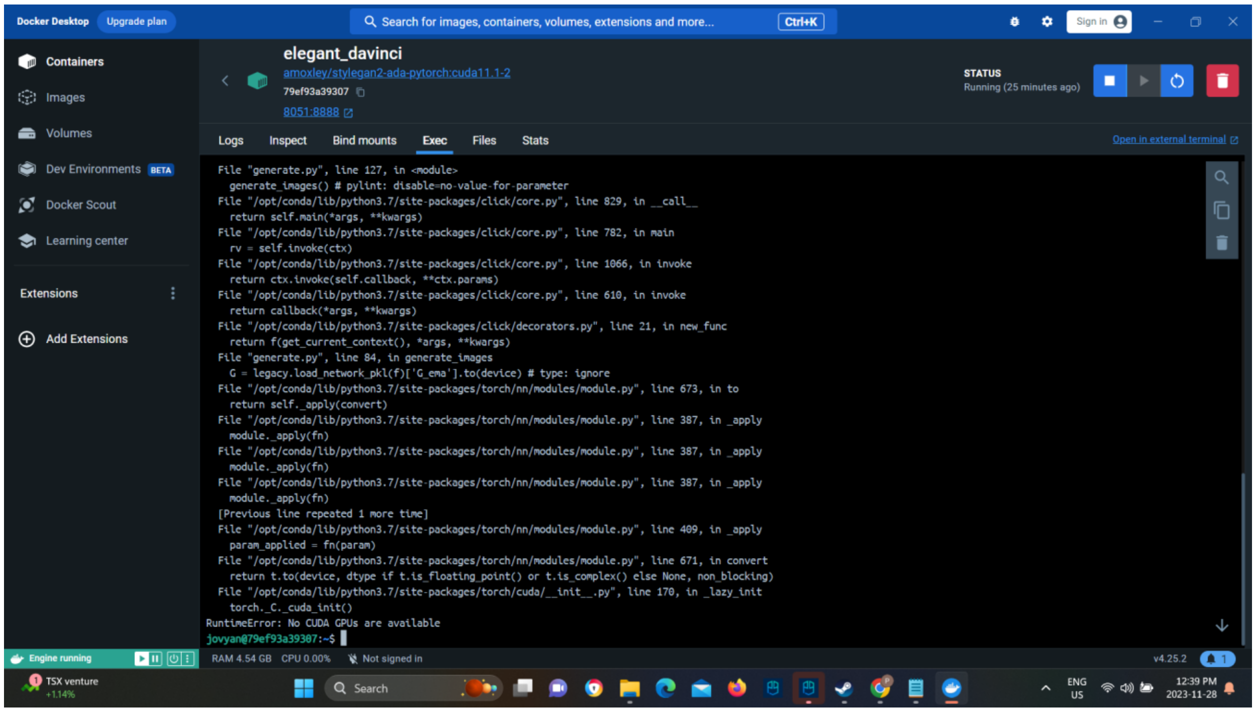
Mixing styles only on specific features of the image.

Type	Loss
gen_loss	80.828
disc_loss	15.6491
gen_loss	19.7928
disc_loss	5.2244
gen_loss	9.0001
disc_loss	4.3516
gen_loss	5.2214
disc_loss	2.1398

gen_loss	2.2847
disc_loss	0.6669

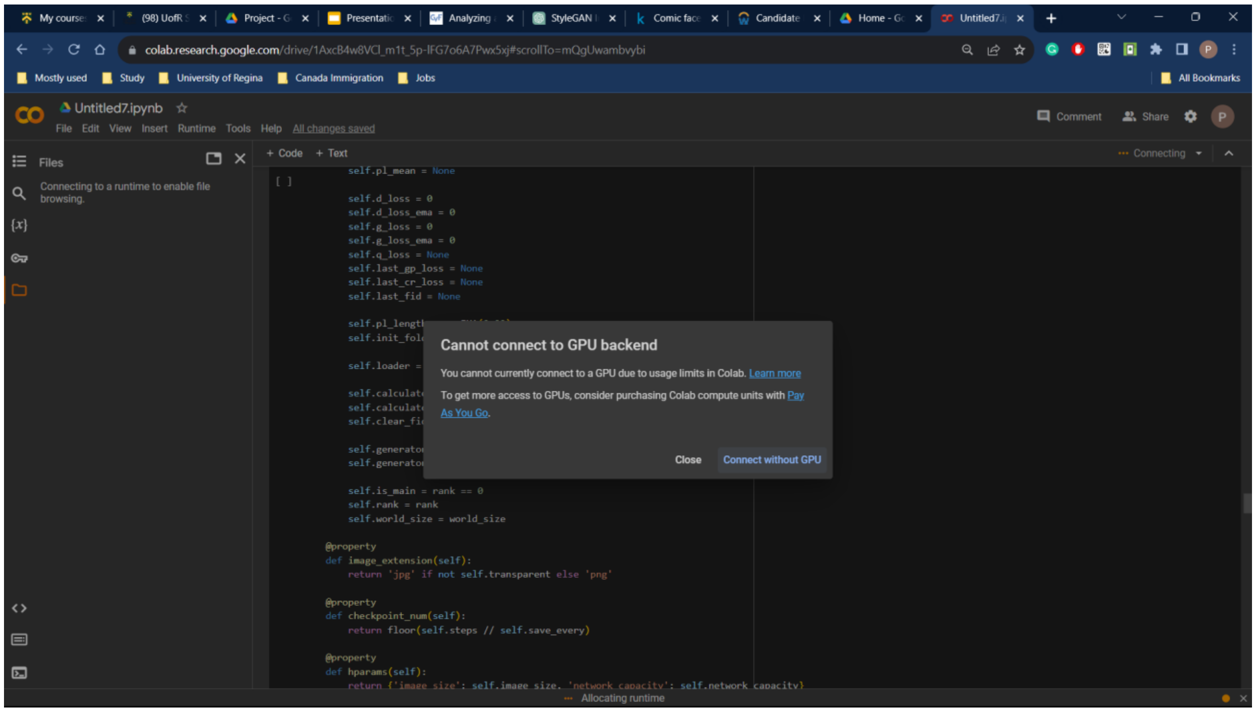
Table of Loss in generating image

LIMITATIONS



```
File "generate.py", line 127, in <module>
  generate_images() # pylint: disable=no-value-for-parameter
File "/opt/conda/lib/python3.7/site-packages/click/core.py", line 829, in __call__
  return self.main(*args, **kwargs)
File "/opt/conda/lib/python3.7/site-packages/click/core.py", line 782, in main
  rv = self.invoke(ctx)
File "/opt/conda/lib/python3.7/site-packages/click/core.py", line 1066, in invoke
  return ctx.invoke(self.callback, **ctx.params)
File "/opt/conda/lib/python3.7/site-packages/click/core.py", line 610, in invoke
  return callback(*args, **kwargs)
File "/opt/conda/lib/python3.7/site-packages/click/decorators.py", line 21, in new_func
  return f(get_current_context(), *args, **kwargs)
File "generate.py", line 84, in generate_images
  G = legacy.load_network_pkl(f)['G_ema'].to(device) # type: ignore
File "/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py", line 673, in to
  return self._apply(convert)
File "/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py", line 387, in _apply
  module._apply(fn)
File "/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py", line 387, in _apply
  module._apply(fn)
File "/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py", line 387, in _apply
  module._apply(fn)
[Previous line repeated 1 more time]
File "/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py", line 409, in _apply
  param_applied = fn(param)
File "/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py", line 671, in convert
  return t.to(device, dtype if t.is_floating_point() or t.is_complex() else None, non_blocking)
File "/opt/conda/lib/python3.7/site-packages/torch/cuda/__init__.py", line 170, in _lazy_init
  torch._C._cuda_init()
RuntimeError: No CUDA GPUs are available
jovyan@79ef93a39307:~$
```

Usage of Less no of images in Data set due to lack of computation power



```
        self.pl_mean = None

        self.d_loss = 0
        self.d_loss_ema = 0
        self.g_loss = 0
        self.g_loss_ema = 0
        self.q_loss = None
        self.last_gp_loss = None
        self.last_cr_loss = None
        self.last_fid = None

        self.pl_lengt
        self.init_fol

        self.loader =

        self.calculat
        self.calculat
        self.clear_fi

        self.generato
        self.generato

        self.is_main = rank == 0
        self.rank = rank
        self.world_size = world_size

    @property
    def image_extension(self):
        return 'jpg' if not self.transparent else 'png'

    @property
    def checkpoint_num(self):
        return floor(self.steps // self.save_every)

    @property
    def hparams(self):
        return {'image_size': self.image_size, 'network_capacity': self.network_capacity}
```

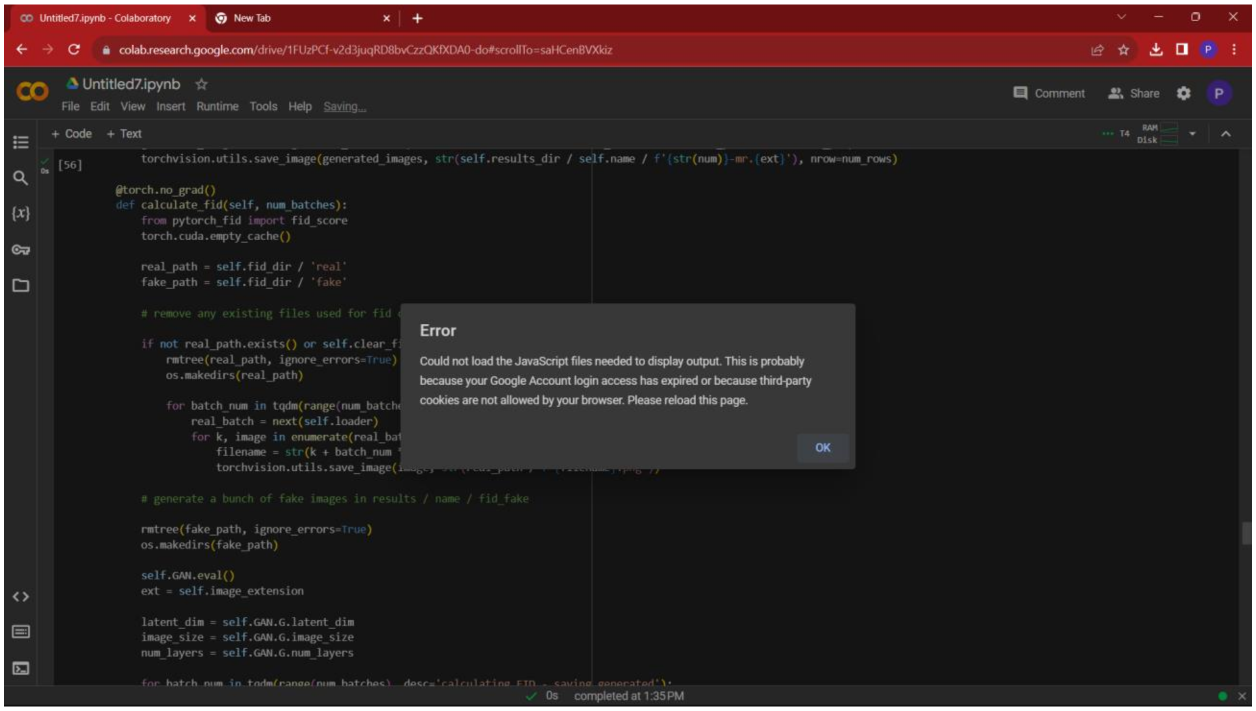
Cannot connect to GPU backend

You cannot currently connect to a GPU due to usage limits in Colab. Learn more

To get more access to GPUs, consider purchasing Colab compute units with Pay As You Go.

Close   Connect without GPU

Can’t able to achieve higher resolution due to less computational power



```
        torchvision.utils.save_image(generated_images, str(self.results_dir / self.name / f'{str(num)}-mr.{ext}'), nrow=num_rows)

    @torch.no_grad()
    def calculate_fid(self, num_batches):
        from pytorch_fid import fid_score
        torch.cuda.empty_cache()

        real_path = self.fid_dir / 'real'
        fake_path = self.fid_dir / 'fake'

        # remove any existing files used for fid

        if not real_path.exists() or self.clear_f
            rmtree(real_path, ignore_errors=True)
            os.makedirs(real_path)

            for batch_num in tqdm(range(num_batche
                real_batch = next(self.loader)
                for k, image in enumerate(real_bat
                    filename = str(k + batch_num
                    torchvision.utils.save_image(

        # generate a bunch of fake images in results / name / fid_fake

        rmtree(fake_path, ignore_errors=True)
        os.makedirs(fake_path)

        self.GAN.eval()
        ext = self.image_extension

        latent_dim = self.GAN.G.latent_dim
        image_size = self.GAN.G.image_size
        num_layers = self.GAN.G.num_layers

        for batch_num in tqdm(range(num_batches), desc='calculating FID - saving generated'):
```

Error

Could not load the JavaScript files needed to display output. This is probably because your Google Account login access has expired or because third-party cookies are not allowed by your browser. Please reload this page.

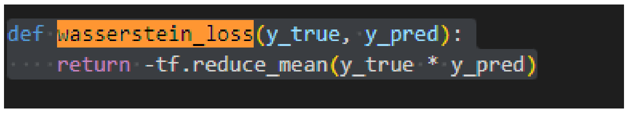
The browser can’t abl3 handle the output as it generates huge amount of output

IMPROVEMENTS ON THE PROPOSED WORK

Due to large variations in the data set of Face images, the discriminator may suffer from high sensitivity to the data to avoid this a Lipschitz constraint is added to prevent the discriminator too judgy from small changes in the input. Which also prevents unstable training. This a regularization technique that helps to keep the discriminator's gradients smooth and prevents them from exploding or vanishing during training. The generator receives more reliable and consistent feedback from the discriminator. This is achieved by adding a gradient penalty which is computed using the Wasserstein loss and the gradients of the discriminator's output with respect to the interpolated Face images (real Face images and fake Face images) and drift loss which is computed by taking the mean squared value of the discriminator loss. Both these values can be controlled by hyperparameters.

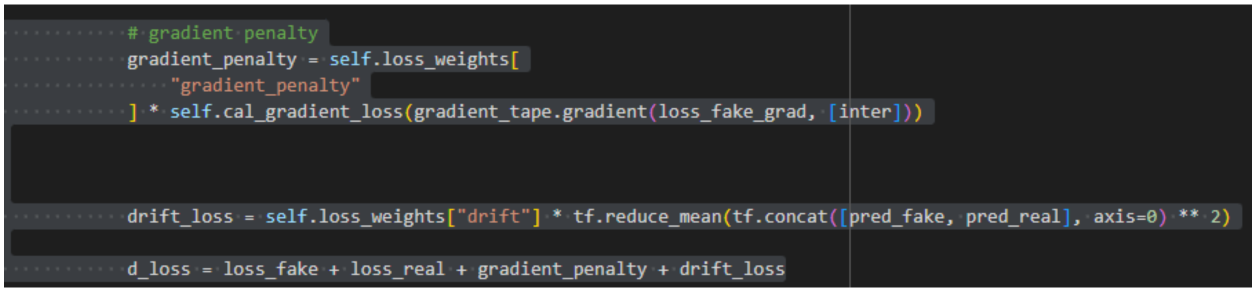
IMPLEMENTATION

The Wasserstein loss is calculated by taking the negative mean of the element-wise product of y_true and y_pred. This loss function is fundamental in WGANs as it helps in minimizing the difference between the probability distributions of real and generated samples, facilitating the stabilization of GAN training by providing a more meaningful and smoother gradient signal for the generator and discriminator networks.



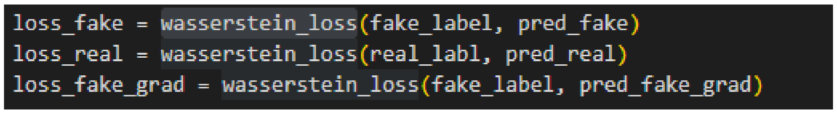
```
def wasserstein_loss(y_true, y_pred):
    return -tf.reduce_mean(y_true * y_pred)
```

The gradient_penalty component aims to enforce the Lipschitz constraint on the discriminator network by penalizing the gradients of the discriminator's output concerning interpolated points between real and generated samples. This penalty is weighted and added to the overall discriminator loss. Additionally, the drift_loss term aims to prevent discriminator predictions from drifting away from zero by computing the mean of the squared values of the concatenated discriminator predictions. The final discriminator loss, d_loss, is composed of various elements, including losses related to fake and real samples (loss_fake and loss_real, respectively), the gradient penalty, and the drift loss, providing a comprehensive metric used to update the discriminator during GAN training. Overall, these loss components contribute to guiding the discriminator's learning process and improving the stability and convergence of the GAN model.



```
            # gradient penalty
            gradient_penalty = self.loss_weights[
                "gradient_penalty"
            ] * self.cal_gradient_loss(gradient_tape.gradient(loss_fake_grad, [inter]))



            drift_loss = self.loss_weights["drift"] * tf.reduce_mean(tf.concat([pred_fake, pred_real], axis=0) ** 2)

            d_loss = loss_fake + loss_real + gradient_penalty + drift_loss
```

Computing three Wasserstein's losses: loss_fake, loss_real, and loss_fake_grad. These losses are computed based on the Wasserstein distance which is a metric used to quantify the dissimilarity between probability distributions. Wasserstein loss is utilized to optimize the discriminator by measuring the discrepancy between the model's predictions for fake and real samples against their respective labels (fake_label and real_label). The Wasserstein loss for the gradient of the discriminator's output with respect to interpolated points (pred_fake_grad) is also computed, likely for the gradient penalty calculation in the GAN's training process.



```
loss_fake = wasserstein_loss(fake_label, pred_fake)
loss_real = wasserstein_loss(real_labl, pred_real)
loss_fake_grad = wasserstein_loss(fake_label, pred_fake_grad)
```

REFERENCES

[1] Carl Bergstrom and Jevin West. Which face is real? http://www.whichfaceisreal.com/learn.html, Accessed November 15, 2019. 1

[2] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. CoRR, abs/1809.11096, 2018. 1

[3] Vincent Dumoulin, Ethan Perez, Nathan Schucher, Florian Strub, Harm de Vries, Aaron Courville, and Yoshua Bengio. Feature-wise transformations. Distill, 2018. https://distill.pub/2018/feature-wise-transformations. 1

[4] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. CoRR, abs/1610.07629, 2016. 1

[5] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accu

[6] Golnaz Ghiasi, Honglak Lee, Manjunath Kudlur, Vincent Dumoulin, and Jonathon Shlens. Exploring the structure of a real-time, arbitrary neural artistic stylization network. CoRR, abs/1705.06830, 2017. 1

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. In NIPS, 2014. 1, 5

[8] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In Proc. NIPS, pages 6626–6637, 2017. 1

[9] Xun Huang and Serge J. Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. CoRR, abs/1703.06868, 2017. 1

[10] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. CoRR, abs/1710.10196, 2017. 1, 5

[11] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In Proc. CVPR, 2018. 1, 2, 4, 5

[12] Tuomas Kynka¨anniemi, Tero Karras, Samuli Laine, Jaakko ¨ Lehtinen, and Timo Aila. Improved precision and recall metric for assessing generative models. In Proc. NeurIPS, 2019. 1, 2, 4

[13] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. CoRR, abs/1802.05957, 2018. 1, 5, 6

[14] Mehdi S. M. Sajjadi, Olivier Bachem, Mario Lucic, Olivier Bousquet, and Sylvain Gelly. Assessing generative models via precision and recall. CoRR, abs/1806.00035, 2018. 1

[15] Karras, T., Aittala, M., Laine, S., Härkönen, E., Hellsten, J., Lehtinen, J. and Aila, T., 2021. Alias-free generative adversarial networks. *Advances in Neural Information Processing Systems*, *34*, pp.852-863.