Assignment 3
Computer Vision (ELL793)

Braj Raj Nagar (2022AIB2682)
Sangam Kumar (2022AIB2671)

1. **Style GAN**
   1) StyleGAN model is loaded from github repo of official PyTorch implementation (https://github.com/NVlabs/stylegan2-ada-pytorch.git).
      Architecture of Style GAN is as followed

```
Generator((synthesis): SynthesisNetwork(
  (b4): SynthesisBlock((conv1): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b8): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b16): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b32): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b64): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b128): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer())    )
  (b256): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b512): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
    (torgb): ToRGBLayer((affine): FullyConnectedLayer()))
  (b1024): SynthesisBlock((conv0): SynthesisLayer(
      (affine): FullyConnectedLayer())
    (conv1): SynthesisLayer((affine): FullyConnectedLayer())
```

```
      (torgb): ToRGBLayer((affine): FullyConnectedLayer()))  )
  (mapping): MappingNetwork(
    (fc0): FullyConnectedLayer()
    (fc1): FullyConnectedLayer()
    (fc2): FullyConnectedLayer()
    (fc3): FullyConnectedLayer()
    (fc4): FullyConnectedLayer()
    (fc5): FullyConnectedLayer()
    (fc6): FullyConnectedLayer()
    (fc7): FullyConnectedLayer()))
```

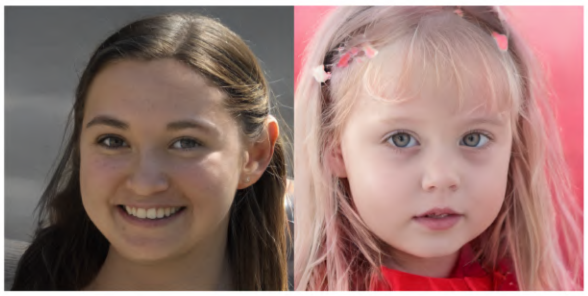
Discriminator((b1024): DiscriminatorBlock(

```
    (fromrgb): Conv2dLayer()
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer())
  (b512): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer()  )
  (b256): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer()  )
  (b128): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer())
  (b64): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer())
  (b32): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer())
  (b16): DiscriminatorBlock(
    (conv0): Conv2dLayer()
    (conv1): Conv2dLayer()
    (skip): Conv2dLayer())
    (b8): DiscriminatorBlock(
      (conv0): Conv2dLayer()
      (conv1): Conv2dLayer()
      (skip): Conv2dLayer())
    (b4): DiscriminatorEpilogue(
      (mbstd): MinibatchStdLayer()
      (conv): Conv2dLayer()
      (fc): FullyConnectedLayer()
      (out): FullyConnectedLayer()))
```

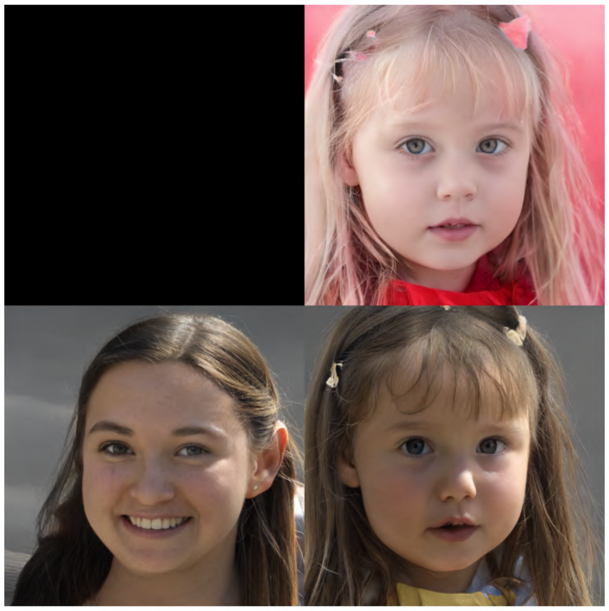
weights of 'b1024.conv1' layer
Feature map size is (32, 32, 3, 3)

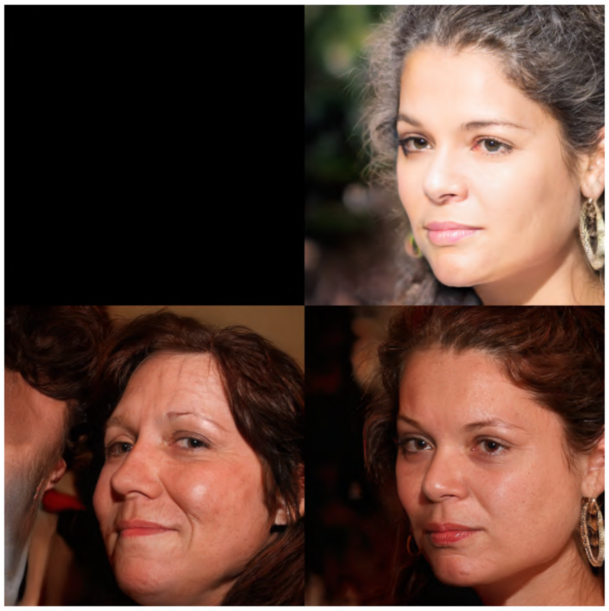
```
[[[[ 0.5074812   1.1998049   0.5371145 ]
   [ 0.44401515  2.6050704  -0.01086236]
   [ 1.2395022  -0.16365209  0.64544326]]

  [[-0.8664978   0.61592245 -1.0916802 ]
   [-0.2837225  -1.4518836  -1.1321101 ]
   [-0.40520248 -0.3041015  -1.3851835 ]]

  [[ 0.08943254  0.05449752 -0.6377199 ]
   [-1.0361853  -0.57320553  1.1334028 ]
   [-0.03746326  0.49329662 -0.63504237]]

  ...

   [[-0.3158307   0.37060362  0.44279686]
   [ 1.0288624  -2.0849771  -0.9286323 ]
   [-2.0945804   2.378132   -0.5208342 ]]]]
```

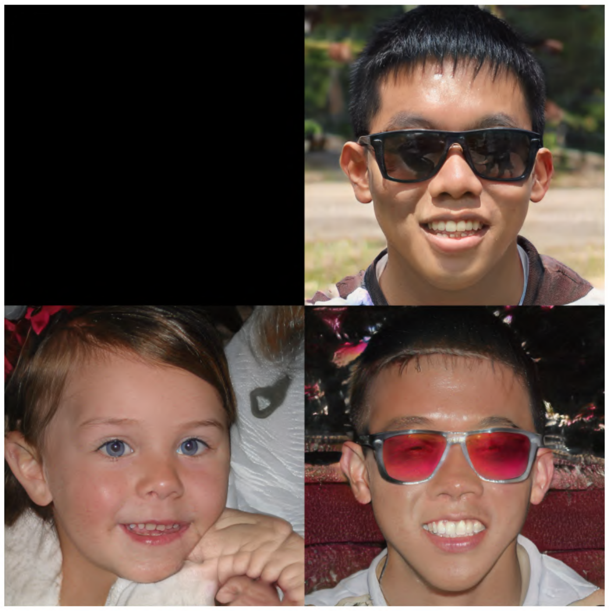
Outputs generated at layer ‘b1024.conv1’ with two random latent vectors:

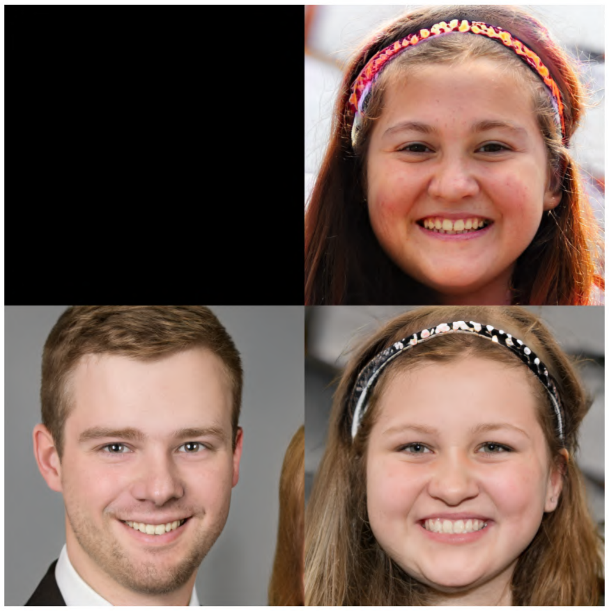


2) Here are the some images generated by style mixing of two randomly generated latent vectors for each example. Style mixing is done on row's image by mixing the features of column's images at synthesis blocks 'b4', 'b8', 'b16', and 'b32'.


Style mixed image 1


Style mixed image 2


Style mixed image 3


Style mixed image 4

3) weights of 'b512.conv1' layer
Feature map size is (64, 64, 3, 3)

```
[[[[ 1.16834259e+00  6.47364929e-02 -1.62330151e+00]
   [ 5.47506034e-01 -3.50682735e+00 -1.92343891e+00]
   [ 3.15828562e-01 -3.07402873e+00  6.48242056e-01]]

  [[-2.09949493e+00  1.39875984e+00  1.61090040e+00]
   [-1.06046192e-01  1.39851928e+00  2.94626999e+00]
   [-7.51548290e-01 -1.10379672e+00 -6.13248467e-01]]

  [[-4.80560303e+00 -8.48222256e-01 -5.03653705e-01]
   [ 2.38746107e-01 -1.05673420e+00  2.71434450e+00]
   [-1.04808331e+00 -1.20017910e+00  1.75444543e+00]]

  ...

  [[ 2.74589396e+00 -3.88662434e+00 -3.84220505e+00]
   [ 2.77123904e+00 -3.34387493e+00 -2.06560826e+00]
   [ 3.55051041e+00 -3.71368790e+00 -1.11968637e-01]]

  [[-1.31480908e+00  1.23471677e+00 -2.94538945e-01]
   [-1.84274209e+00 -8.76590490e-01  1.01717182e-01]
   [-4.85633537e-02 -1.44607460e+00  1.04567516e+00]]

  [[-5.05499172e+00 -3.77770519e+00  4.11394000e-01]
   [-1.86704111e+00  5.95886111e-02 -1.99898267e+00]
   [ 2.43037248e+00 -9.88720298e-01 -1.45790532e-01]]]]
```

I have tried to generate output not only ‘b512’ synthesis block but I each synthesis block from ‘b4’ to ‘b1024’ and here’s the generated output images

Output of ‘b4’ synthesis block

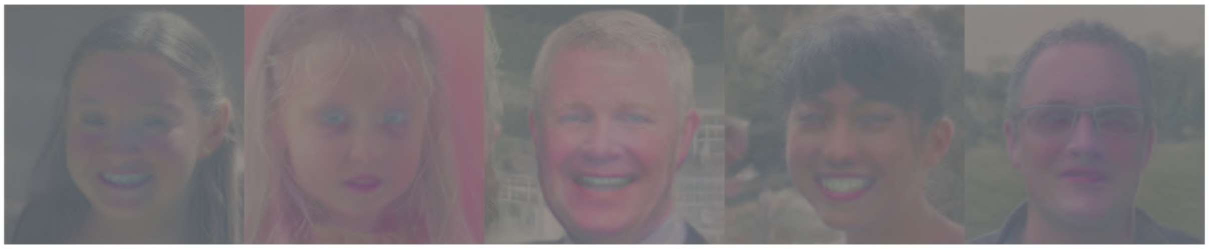
Output of ‘b8’ synthesis block

Output of ‘b16’ synthesis block

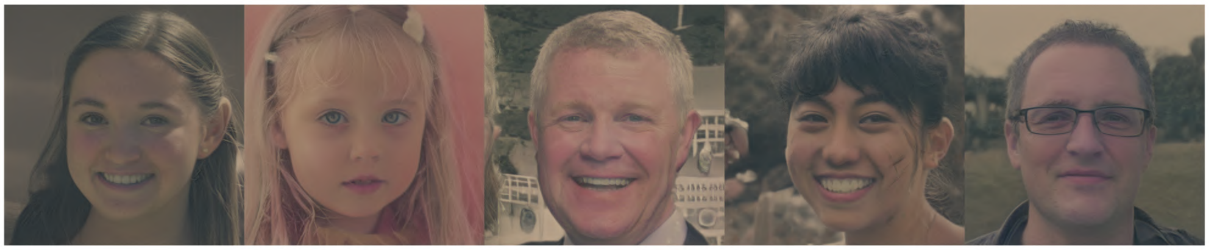
Output of ‘b32’ synthesis block

Output of ‘b64’ synthesis block

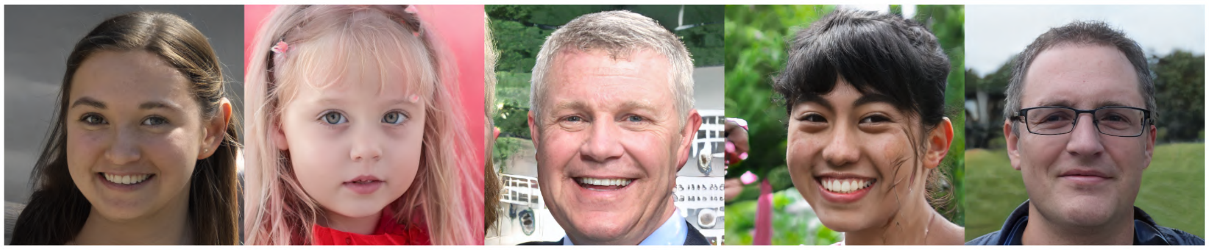
Output of ‘b128’ synthesis block



Output of ‘b256’ synthesis block



Output of ‘b512’ synthesis block



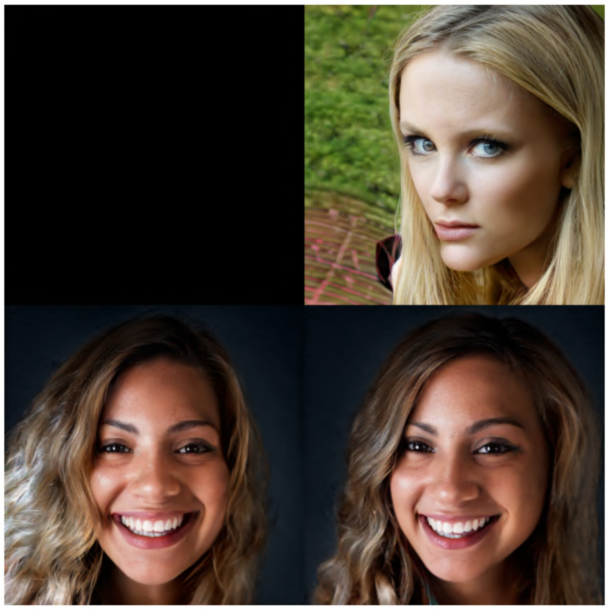
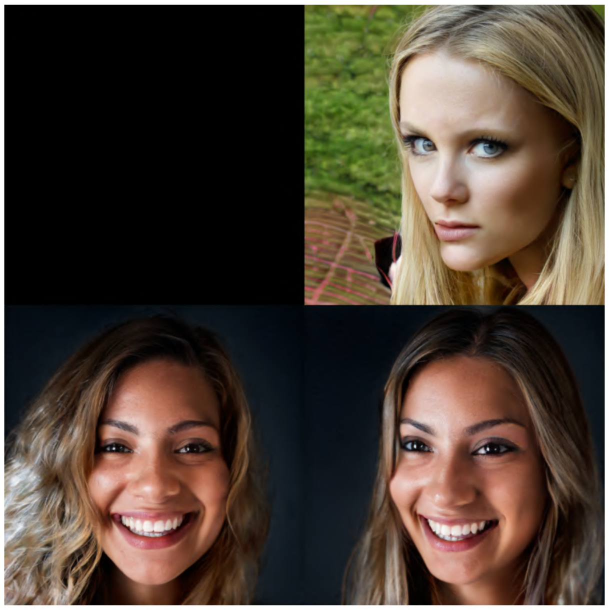
Output of ‘b1024’ synthesis block

As you can see that at the initial synthesis blocks model learns low level features and as the generated tensor are passed through later synthesis blocks model learns high level features. the deeper blocks generate larger and more complex features of the image, such as textures and patterns, while the shallower blocks generate simpler features such as edges and shapes.

4) Style mixing is a concept in StyleGAN (Generative Adversarial Networks) that allows for the creation of new images by mixing styles of different images during the generation process. It allows for greater control over the generated images, as users can selectively control which parts of the generated images are influenced by which styles.
In StyleGAN, the "style" of an image refers to the mapping of latent variables to the different layers of the generator network. The generator network is composed of several convolutional layers, each of which learns to create a specific feature of the image. The style of the image determines how each of these layers will create the features of the image.

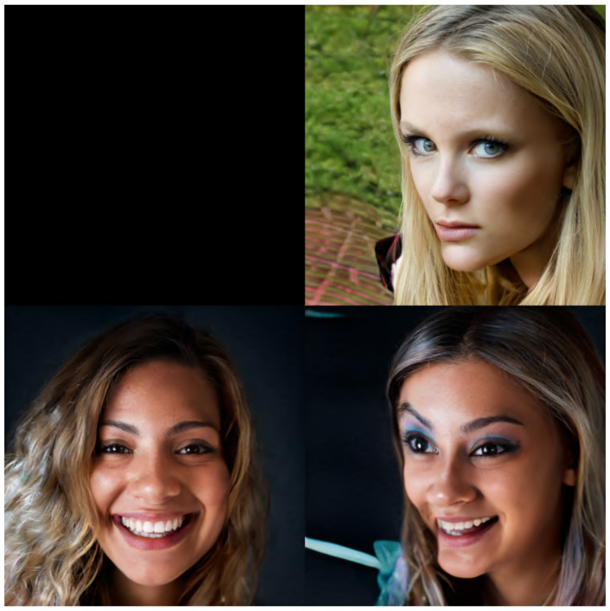
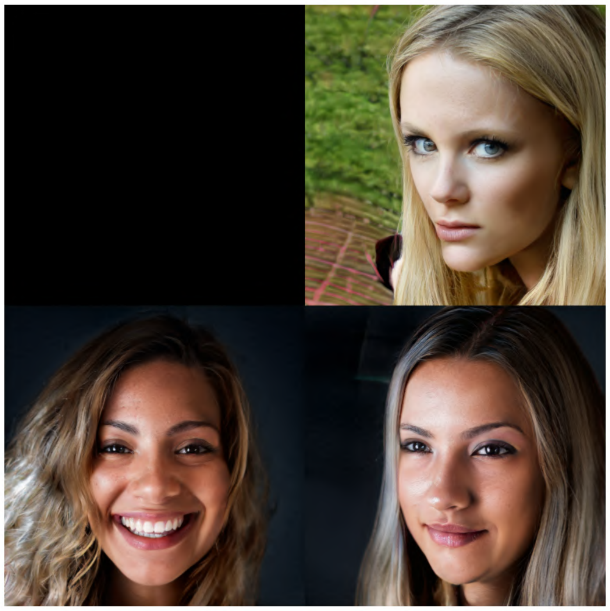
Style mixing works by taking two or more latent vectors (the input to the generator network) and using different parts of each latent vector to control the styles of different layers in the generator network. Specifically, the first part of one latent vector is used to control the styles of the lower layers of the network, while the second part of another latent vector is used to control the styles of the higher layers of the network. The resulting image is a combination of these styles, resulting in a unique image that incorporates the features of both input images.
Style mixing differs from traditional style transfer methods in that it is not limited to transferring the style of one image onto another. Instead, it allows for the creation of new images that incorporate the styles of multiple input images. Additionally, it provides greater control over the generated images, allowing users to selectively control which parts of the generated images are influenced by which styles.
The effect of style mixing on the generated images is that it creates images that are more diverse and visually interesting than those generated by traditional style transfer methods. It allows for the creation of images that combine the best features of multiple input images, resulting in unique and compelling images that are visually striking.

5) The depth of the chosen layer in StyleGAN can have a significant effect on the quality of the generated images when using style mixing. In general, using deeper layers tends to produce higher quality images.
The reason for this is that deeper layers in the generator network tend to capture more abstract level features of the image. These layers are responsible for generating more complex and detailed parts of the image, such as textures and patterns, while earlier layers tend to focus on more basic features such as edges and shapes.
When using style mixing, the lower layers of the network control more basic features of the generated image, while higher layers control more complex features. If the lower layers are used exclusively in style mixing, the resulting image will likely be simple and lacking in detail. On the other hand, if only higher layers are used, the image may be too complex and chaotic.
Therefore, to achieve a balance between simplicity and complexity, it is recommended to use a mix of lower and higher layers in style mixing. However, deeper layers are generally preferred as they can capture more complex and detailed features of the image, resulting in higher quality generated images.
It is also worth noting that the optimal layer depth for style mixing may vary depending on the specific dataset and the intended application.
To explain the theory mentioned above here's some examples of style mixing at different depth of synthesis blocks (layers). In each example features of column image is mixed with row image. Let's start mixing features at deeper layers of model and examin the results.

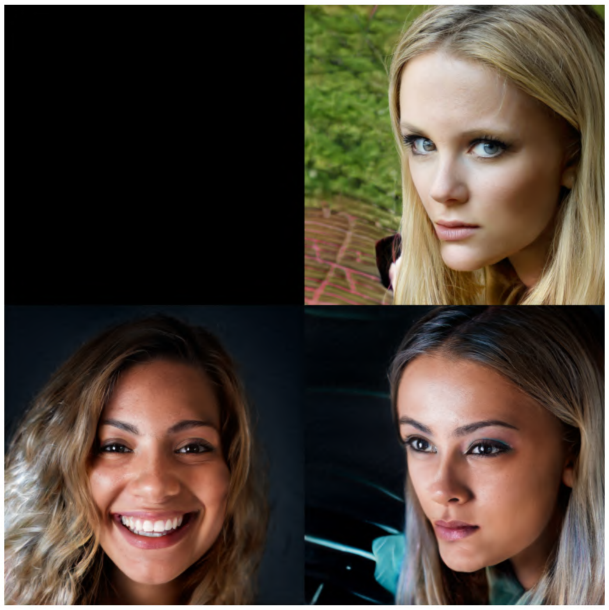
Style mixing at 'b4' synthesis block Style mixing at 'b8' synthesis block

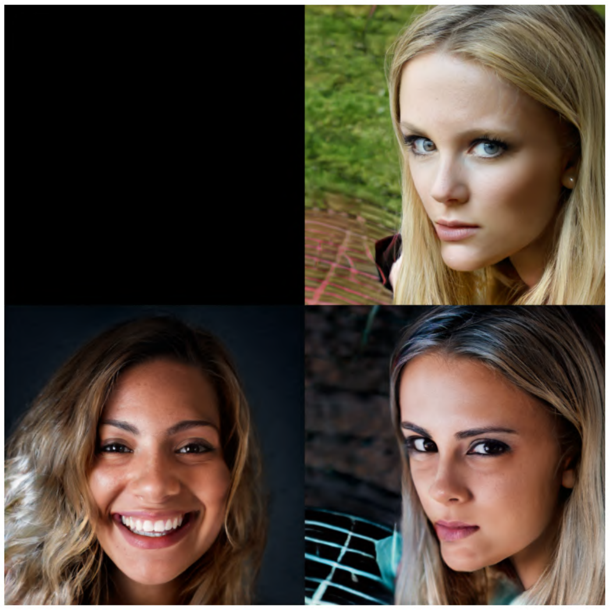
'b4' block mostly captures the orientation and some geometrical features of column image but 'b8 block captures some texture feature also.

Style mixing at 'b4', 'b8' synthesis block Style mixing at 'b8', 'b16' synthesis block

'b4', 'b8' captures more orientation and texture feature compare to let alone 'b4' and 'b8' synthesis blocks. But as we skip the 'b4', 'b8' and do the style mixing at 'b8', 'b16' synthesis block texture features are less visible from column image.

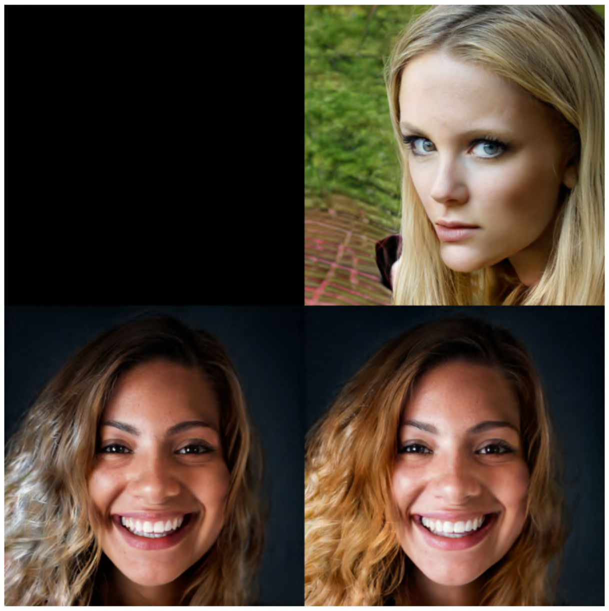

Style mixing at ‘b4’, ‘b8’, ‘b16’ synthesis block

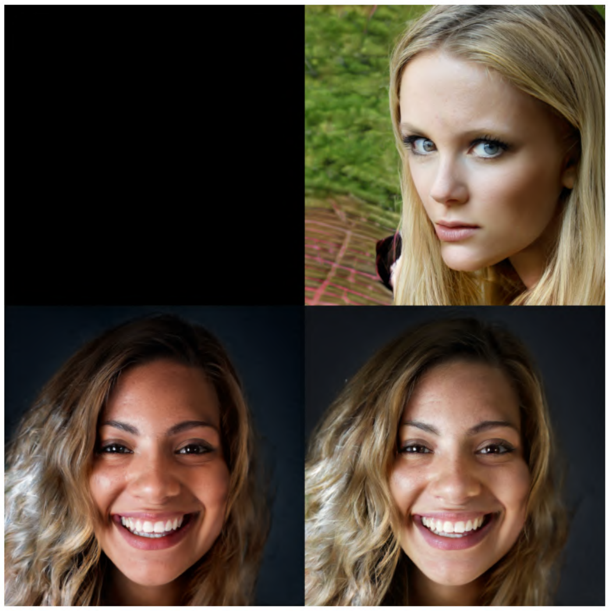

Style mixing at ‘b4’, ‘b8’, ‘b16’, ‘b32’ synthesis block

As we style mix more at deeper blocks we start to get much beter texture and orientation mixing of column image into row image. For example in last image model even able to mix the eyes orientation.

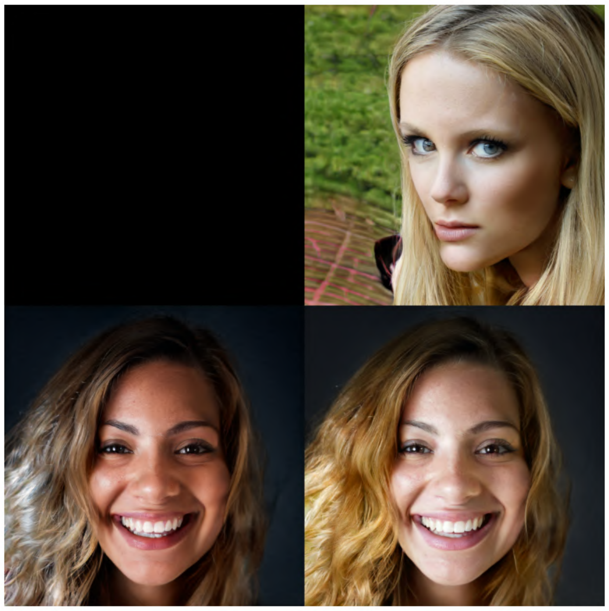
This imply that deeper layers in the generator network tend to capture and mix more abstract level features of the image. Such as textures, patterns and orientation.

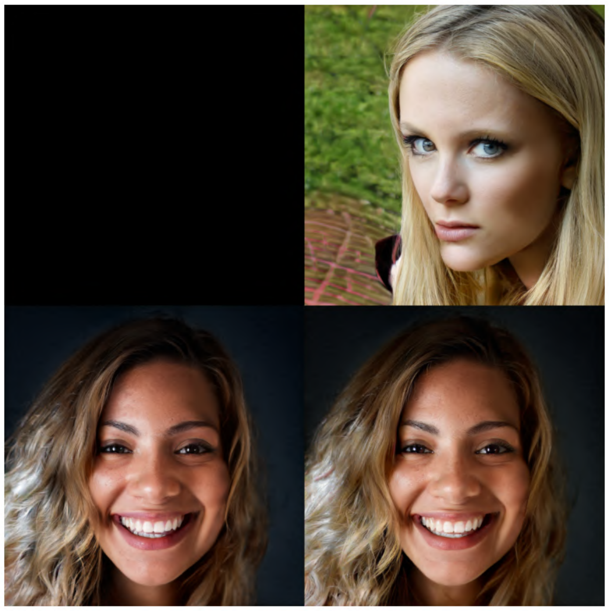
But what if we style mix at shallow blocks, here’s some results of mixing features at shallow blocks

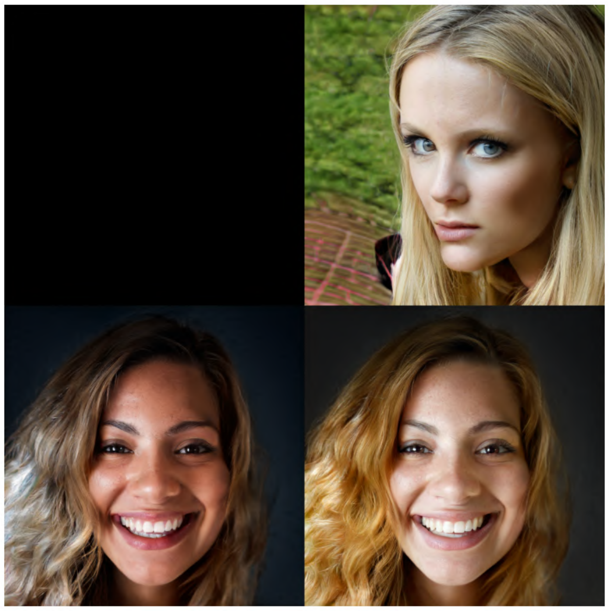

Style mixing at ‘b128’ synthesis block

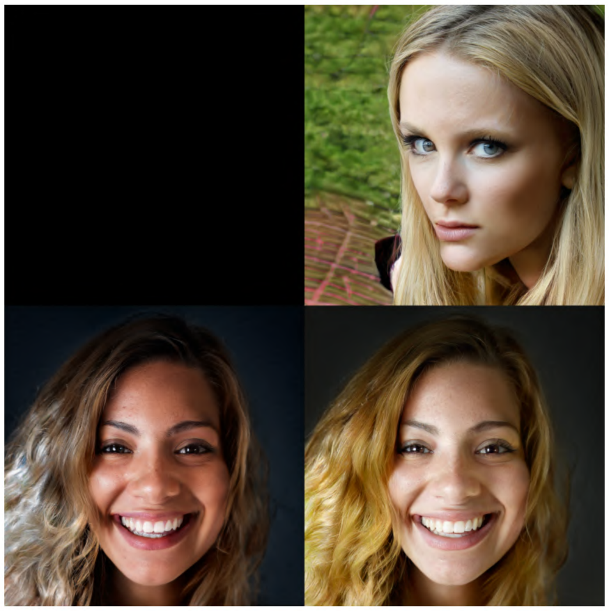

Style mixing at ‘b256’ synthesis block

‘b128’ block mix the light and bightness of colum image with row image. ‘B256’ block mix the colour. But none of the block mixes texture or geometical features.


Style mixing at ‘b128’ and ‘b256’ synthesis block

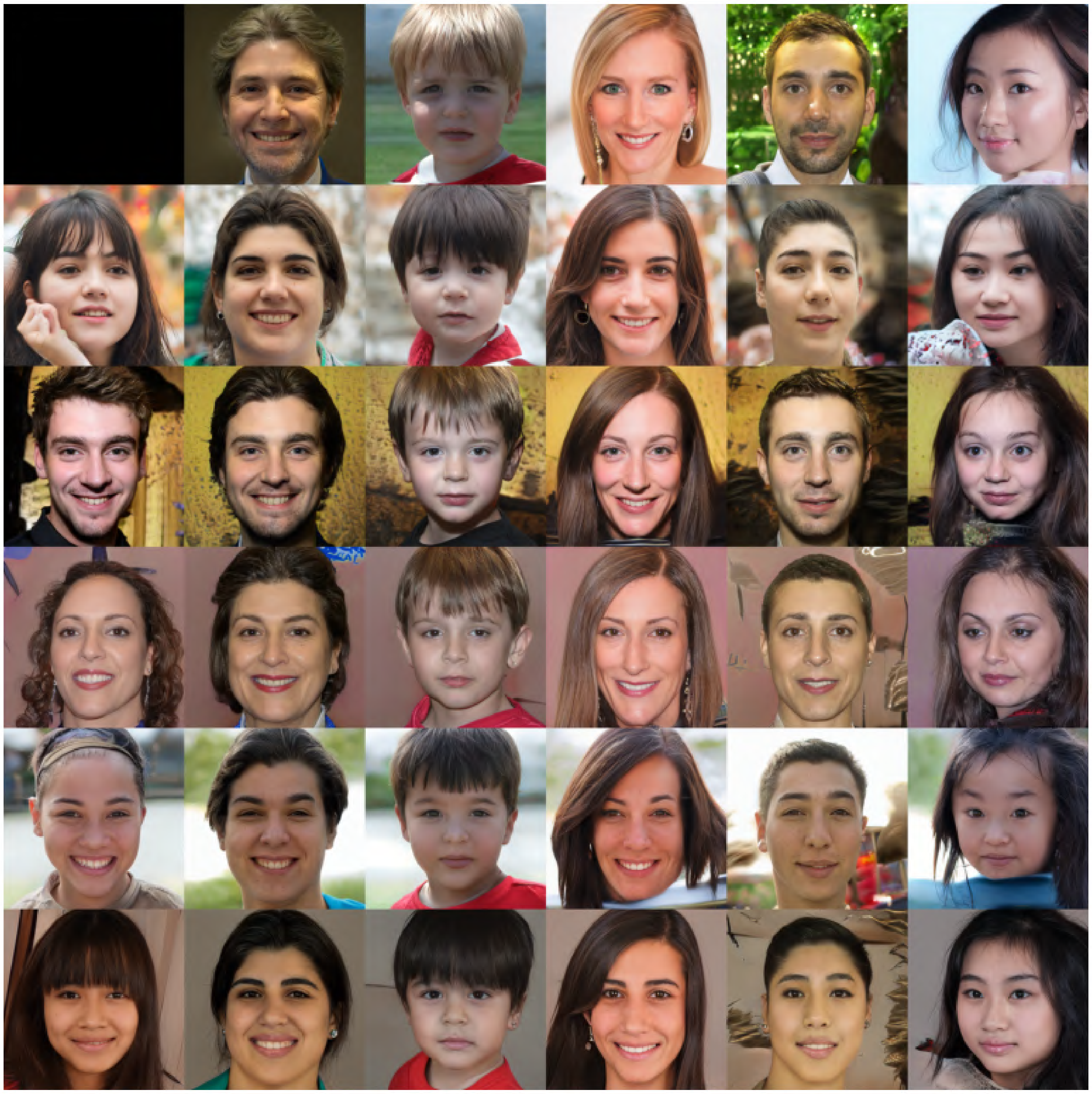

Style mixing at ‘b512’ and ‘b1024’ synthesis block


Style mixing at ‘b128’, ‘b256’ and ‘b512’ synthesis block

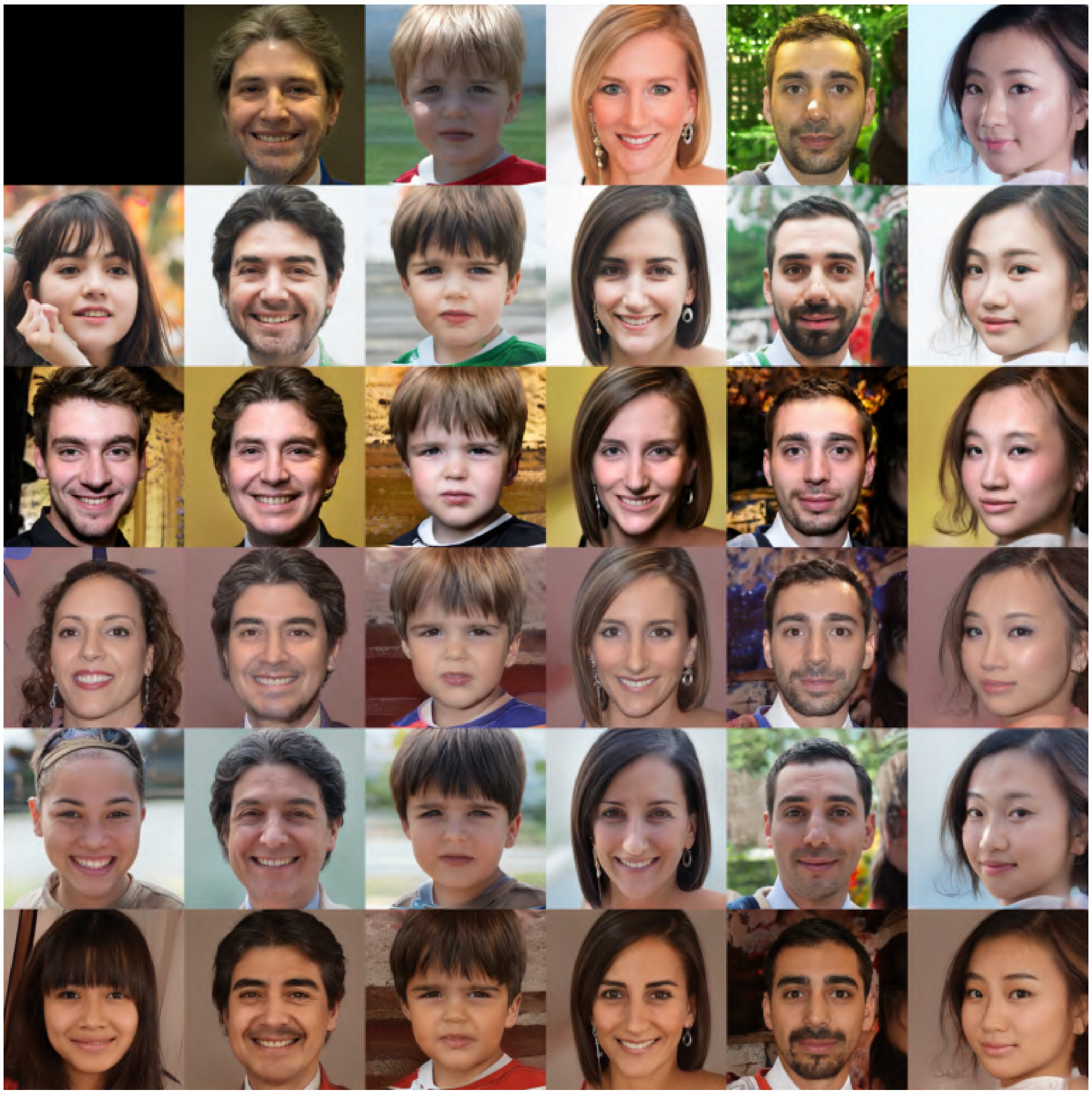

Style mixing at ‘b128’, ‘b256’, ‘b512’ and ‘b1024’ synthesis block

It is evident from all the images above that shallow synthesis blocks (‘b128’ to ‘b1024’) capture and mixes basic level features such as lighting, brightness, contrast, colours etc. but doesn’t captures the abstract level features such as texture, pattern etc.

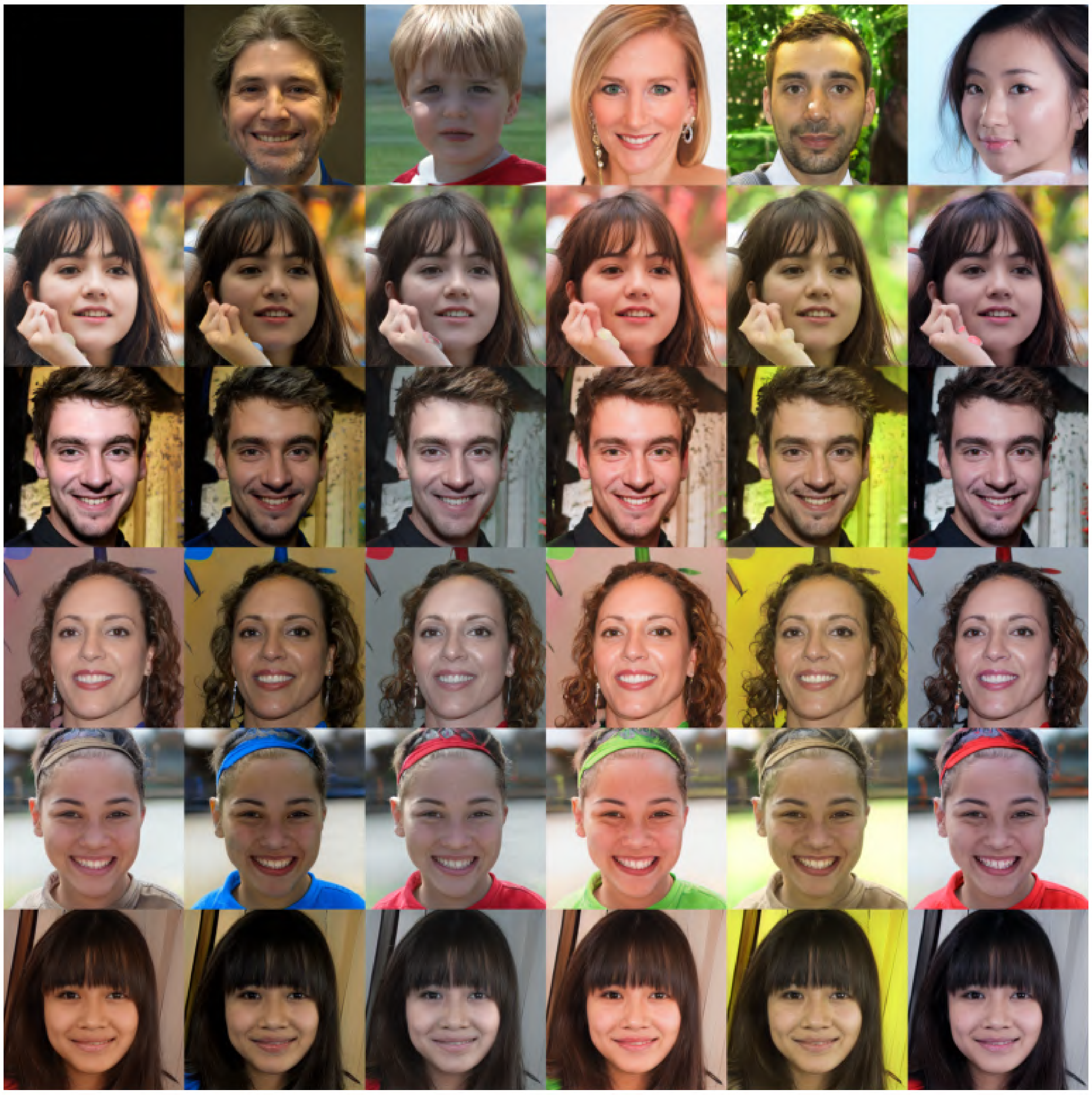
6) style mixing with more than two latent vectors



Style mixing at ‘b8’, ‘b16’, ‘b256’, ‘b512’ synthesis blocks
‘b8’, ‘b16’ mixes Texture and Pattern, ‘b256’, ‘b512’ mixes brightness and colour types of features from colum images to row images.



Style mixing at deeper layers ‘b4’, ‘b8’, ‘b16’, ‘b32’ synthesis blocks
Mixing the texture, pattern and orientational level features of colum images into row images

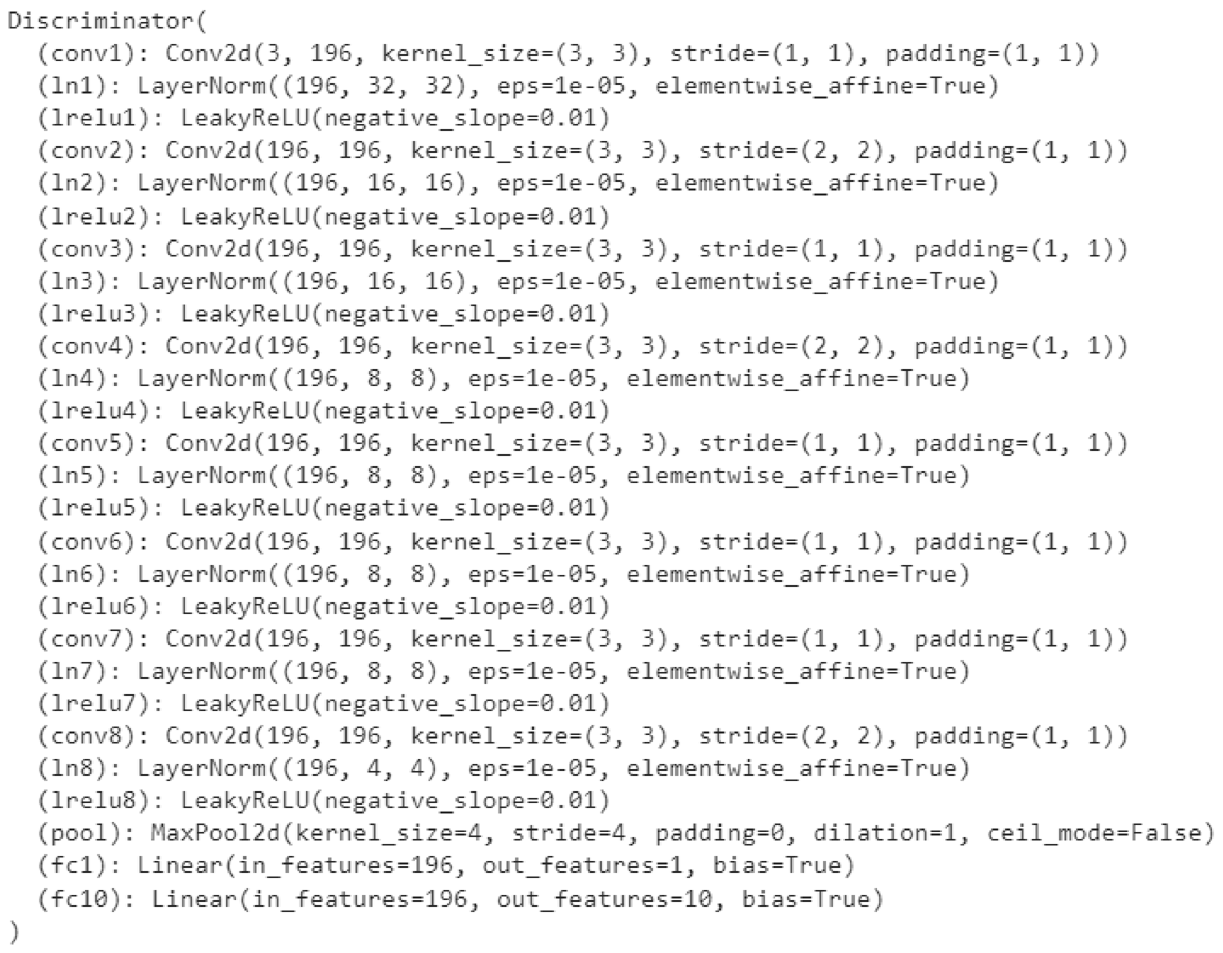


Style mixing at ‘b128’, ‘b256’, ‘b512’ and ‘b1024’ synthesis blocks
Mixing the lighting, brightness, colour, shadow, contrast type of features of colum images into row images

2. GAN from scratch for CIFAR-10

In this task, we are asked to implement a Generative Adversarial Network (GAN) from scratch using PyTorch for the CIFAR-10 dataset.
A GAN is a type of deep neural network architecture that consists of two parts: a generator network and a discriminator network. The generator network takes in random noise as input and generates fake images, while the discriminator network takes in both real and fake images and tries to distinguish between them. The goal of the generator network is to generate images that are indistinguishable from real images, while the goal of the discriminator network is to correctly identify fake images as fake and real images as real. The two networks are trained in adversarial manner, where the generator tries to fool the discriminator and the discriminator tries to correctly identify fake images.
To implement this GAN, we are asked to use CNN models for the generator and discriminator networks. Convolutional Neural Networks (CNNs) are commonly used for image recognition tasks and are well-suited for processing images.

DISCRIMINATOR

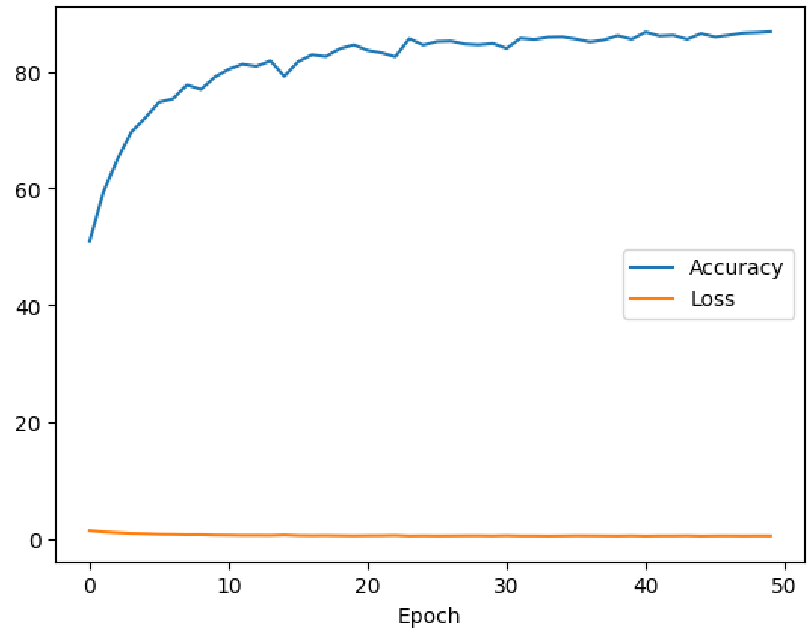


```
Discriminator(
  (conv1): Conv2d(3, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (ln1): LayerNorm((196, 32, 32), eps=1e-05, elementwise_affine=True)
  (lrelu1): LeakyReLU(negative_slope=0.01)
  (conv2): Conv2d(196, 196, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (ln2): LayerNorm((196, 16, 16), eps=1e-05, elementwise_affine=True)
  (lrelu2): LeakyReLU(negative_slope=0.01)
  (conv3): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (ln3): LayerNorm((196, 16, 16), eps=1e-05, elementwise_affine=True)
  (lrelu3): LeakyReLU(negative_slope=0.01)
  (conv4): Conv2d(196, 196, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (ln4): LayerNorm((196, 8, 8), eps=1e-05, elementwise_affine=True)
  (lrelu4): LeakyReLU(negative_slope=0.01)
  (conv5): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (ln5): LayerNorm((196, 8, 8), eps=1e-05, elementwise_affine=True)
  (lrelu5): LeakyReLU(negative_slope=0.01)
  (conv6): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (ln6): LayerNorm((196, 8, 8), eps=1e-05, elementwise_affine=True)
  (lrelu6): LeakyReLU(negative_slope=0.01)
  (conv7): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (ln7): LayerNorm((196, 8, 8), eps=1e-05, elementwise_affine=True)
  (lrelu7): LeakyReLU(negative_slope=0.01)
  (conv8): Conv2d(196, 196, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (ln8): LayerNorm((196, 4, 4), eps=1e-05, elementwise_affine=True)
  (lrelu8): LeakyReLU(negative_slope=0.01)
  (pool): MaxPool2d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=196, out_features=1, bias=True)
  (fc10): Linear(in_features=196, out_features=10, bias=True)
)
```

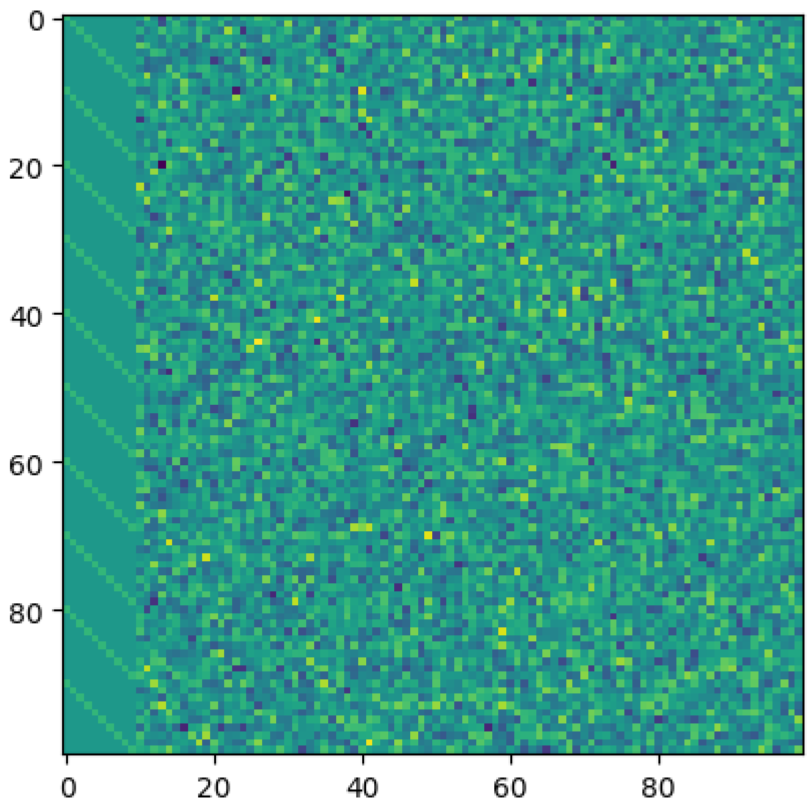
GENERTATOR

```
Generator(
  (fc1): Linear(in_features=100, out_features=3136, bias=True)
  (bn0): BatchNorm1d(3136, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu0): ReLU()
  (conv1): ConvTranspose2d(196, 196, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (bn1): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU()
  (conv2): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): ReLU()
  (conv3): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): ReLU()
  (conv4): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu4): ReLU()
  (conv5): ConvTranspose2d(196, 196, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (bn5): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu5): ReLU()
  (conv6): Conv2d(196, 196, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn6): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu6): ReLU()
  (conv7): ConvTranspose2d(196, 196, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (bn7): BatchNorm2d(196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu7): ReLU()
  (conv8): Conv2d(196, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (tanh): Tanh()
)
```

First we train the discrimniator model to check its accuracy to discriminate between fake images and real images .



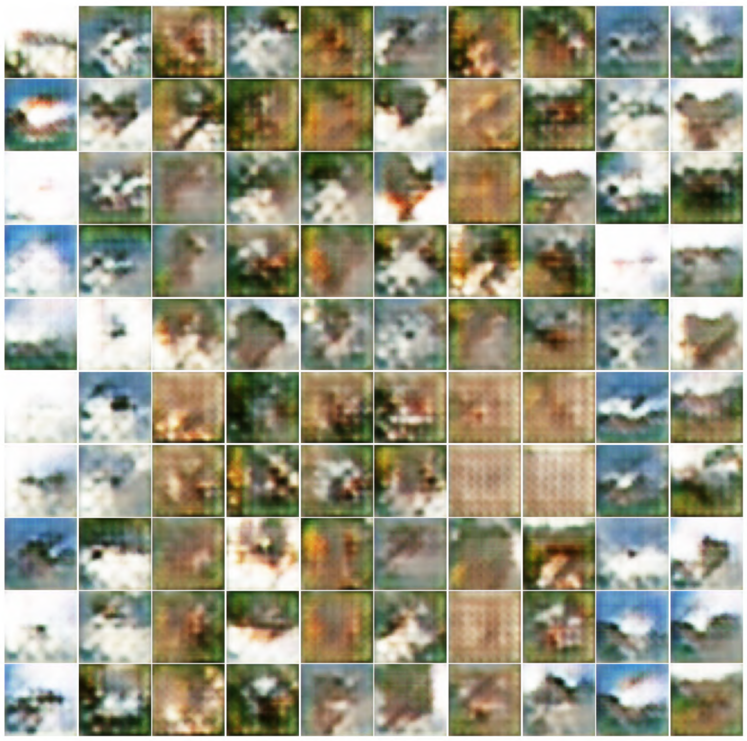
Then we started with a random noise creation which sent into the Generator.



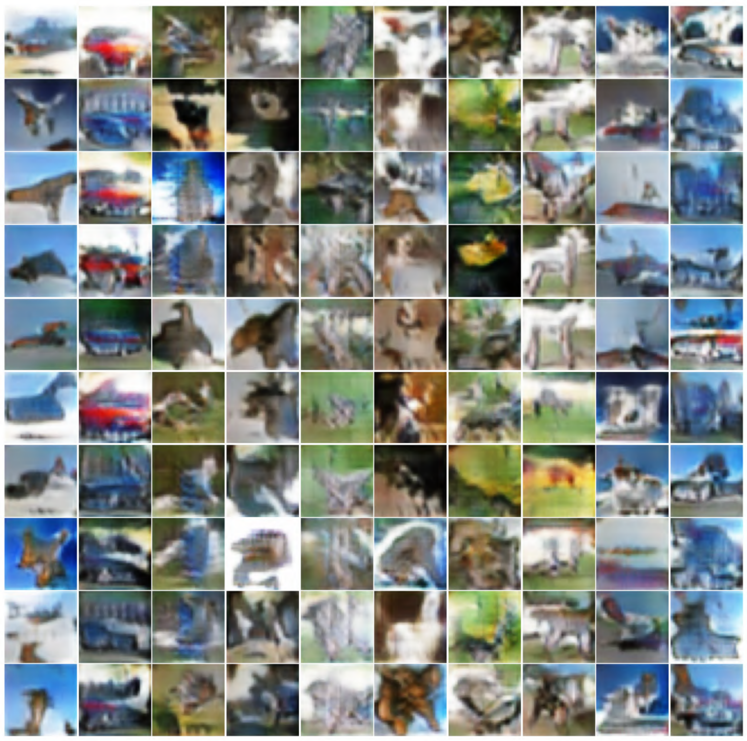
We are also asked to optimize the learning rate, which is a hyper-parameter that determines how quickly the network learns. Choosing the right learning rate is important for getting good results, as setting it too high can cause the network to diverge, while setting it too low can cause the network to converge too slowly or get stuck in local minima. A common approach is to use a learning rate scheduler, which gradually reduces the learning rate over time as the network converges.
So we choose are learning rate as 0.0002 and decrease it while training after 15 epochs.

Images during training:

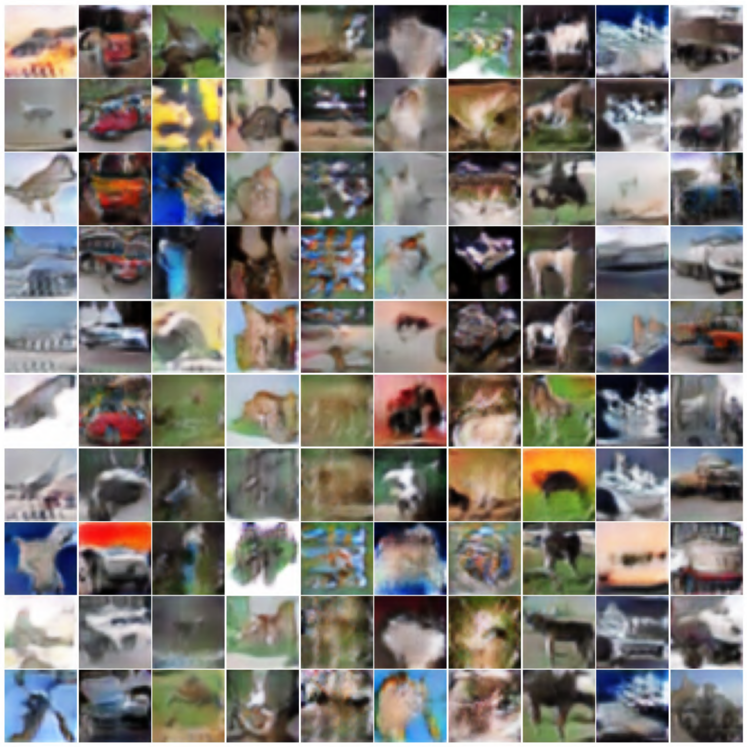
At 0th epoch:



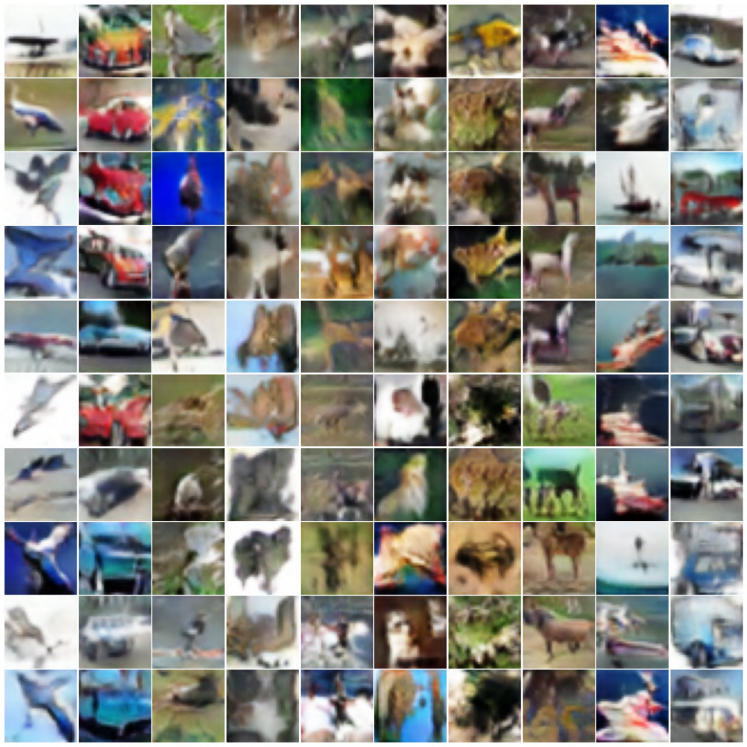
At 10th epoch:



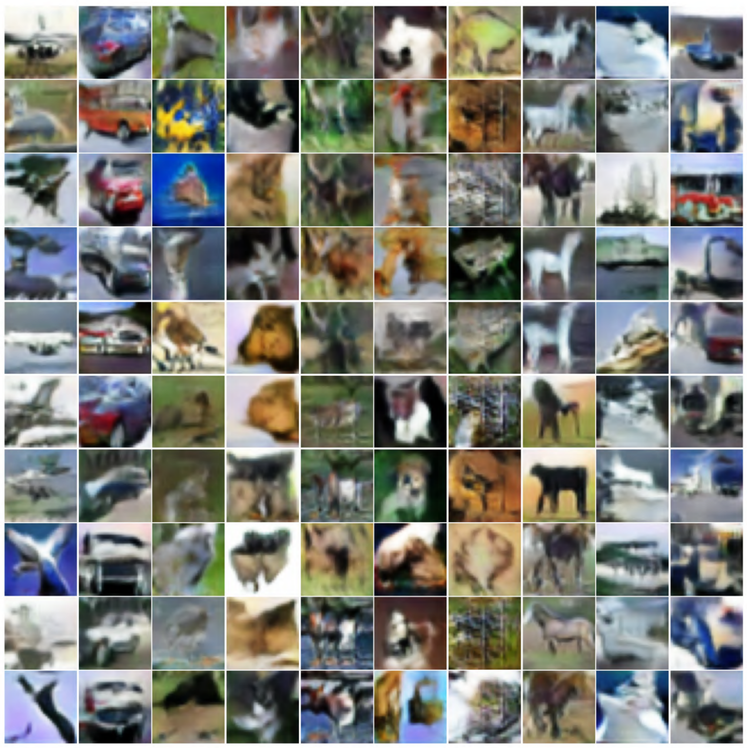
At 20th epoch:



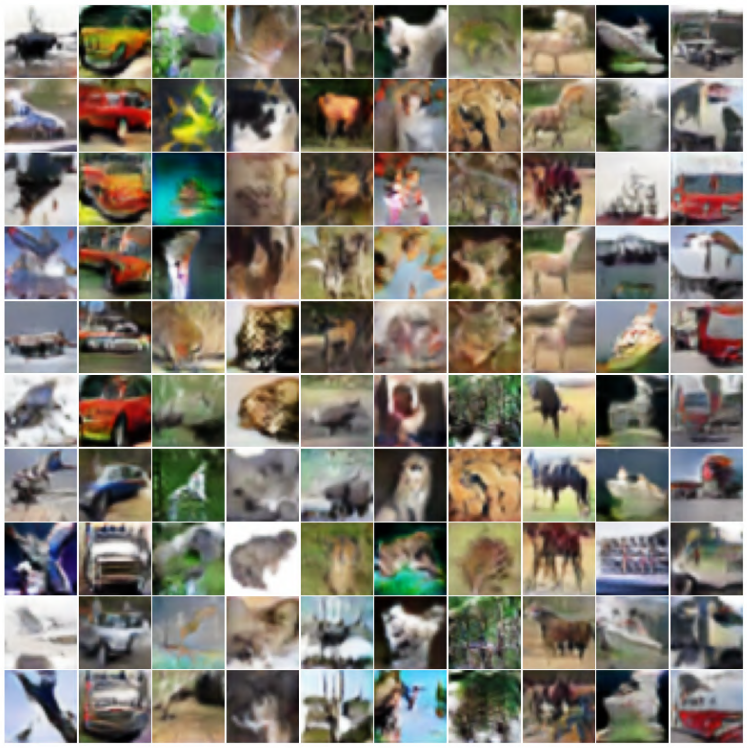
At 30th epoch:



At 40th epoch:

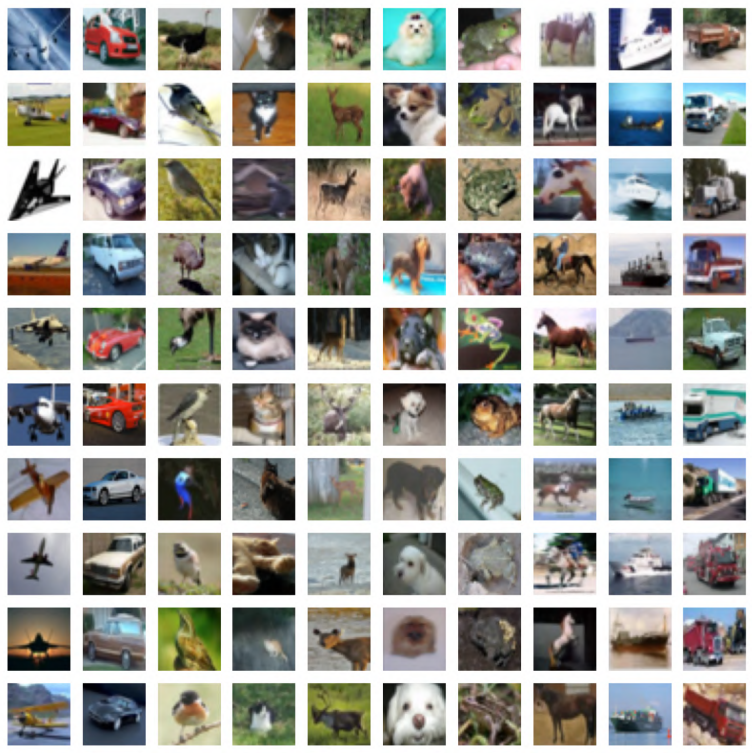


At 50th epoch:.



REAL IMAGES vs GAN IMAGES

Real Images



GAN images