COIMBATORE INSTITUTE OF TECHNOLOGY COIMBATORE

RECORD

NAME	**: SARIKA M**
ROLL NO	**: 1934040**
SUBJECT NAME	**: GENERATIVE AI LAB**
SUBJECT CODE	**: 19MAMEL07**

Ex.no: 01 **Image to Image Generation**

Code:

```
from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
import numpy as np
import glob
import imageio
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time
import os
from tqdm import tqdm
from IPython import display
```

```
img_path = 'image_ds'
```

```
from PIL import Image

dataset = []
data_size = 43
amplification = 5
for i in range(1, data_size+1):
  for j in range(amplification):
    image = Image.open(img_path + '{}.jpg'.format(i))
    img = image.resize((512,512))                    # Reshaping the images to 512,512 size
    dataset.append((np.asarray(img)-127.5)/127.5)      # Conversion of image to numpy array and
Normalizing
```

```
BATCH_SIZE = 4
BUFFER_SIZE = 60000
train_dataset =
tf.data.Dataset.from_tensor_slices(dataset).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

```
# Generator

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(32 * 32 * 1024, use_bias=False, input_shape=(100,)))  # Fix the input
shape here
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((32, 32, 1024)))
    assert model.output_shape == (None, 32, 32, 1024)

    model.add(layers.Conv2DTranspose(512, (3, 3), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 64, 64, 512)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(256, (3, 3), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 128, 128, 256)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(128, (3, 3), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 256, 256, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (3, 3), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 512, 512, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(3, (3, 3), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 512, 512, 3)

    return model

gen = make_generator_model()

# Discriminator

def make_discriminator_model():
  model = tf.keras.Sequential()
  model.add(layers.Conv2D(64, (3,3), strides = (2,2), padding='same', input_shape = [512,512,3]))

  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))

  model.add(layers.Conv2D(128, (3,3), strides = (2,2), padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))

  model.add(layers.Conv2D(256, (3,3), strides = (2,2), padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))

  model.add(layers.Conv2D(512, (3,3), strides = (2,2), padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))

  model.add(layers.Flatten())
  model.add(layers.Dense(64))
  model.add(layers.Dense(1))

  return model

dcrm = make_discriminator_model()
```

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits = True)

def discriminator_loss(real_output, fake_output):
  real_loss = cross_entropy(tf.ones_like(real_output), real_output)
  fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
  total_loss = real_loss + fake_loss
  return total_loss

def generator_loss(fake_output):
  return cross_entropy(tf.ones_like(fake_output), fake_output)
```

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
checkpoint_dir = 'E:\semester_notest_assignment\Sem9\GenAI\P1\Checkpoint'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer = generator_optimizer,
discriminator_optimizer = discriminator_optimizer, generator = gen, discriminator = dcrm)
```

```
# Training

EPOCHS = 500
```

```
noise_dim = 100
num_examples_to_generate = 10

seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

```
@tf.function
def train_step(images):
  noise = tf.random.normal([BATCH_SIZE, noise_dim])

  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    generated_images = gen(noise, training = True)
    real_output = dcrm(images, training = True)
    fake_output = dcrm(generated_images, training =True)

    gen_loss = generator_loss(fake_output)
    disc_loss = discriminator_loss(real_output, fake_output)

  gradients_of_generator = gen_tape.gradient(gen_loss, gen.trainable_variables)
  gradients_of_discriminator = disc_tape.gradient(disc_loss, dcrm.trainable_variables)

  generator_optimizer.apply_gradients(zip(gradients_of_generator, gen.trainable_variables))
  discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
dcrm.trainable_variables))
```

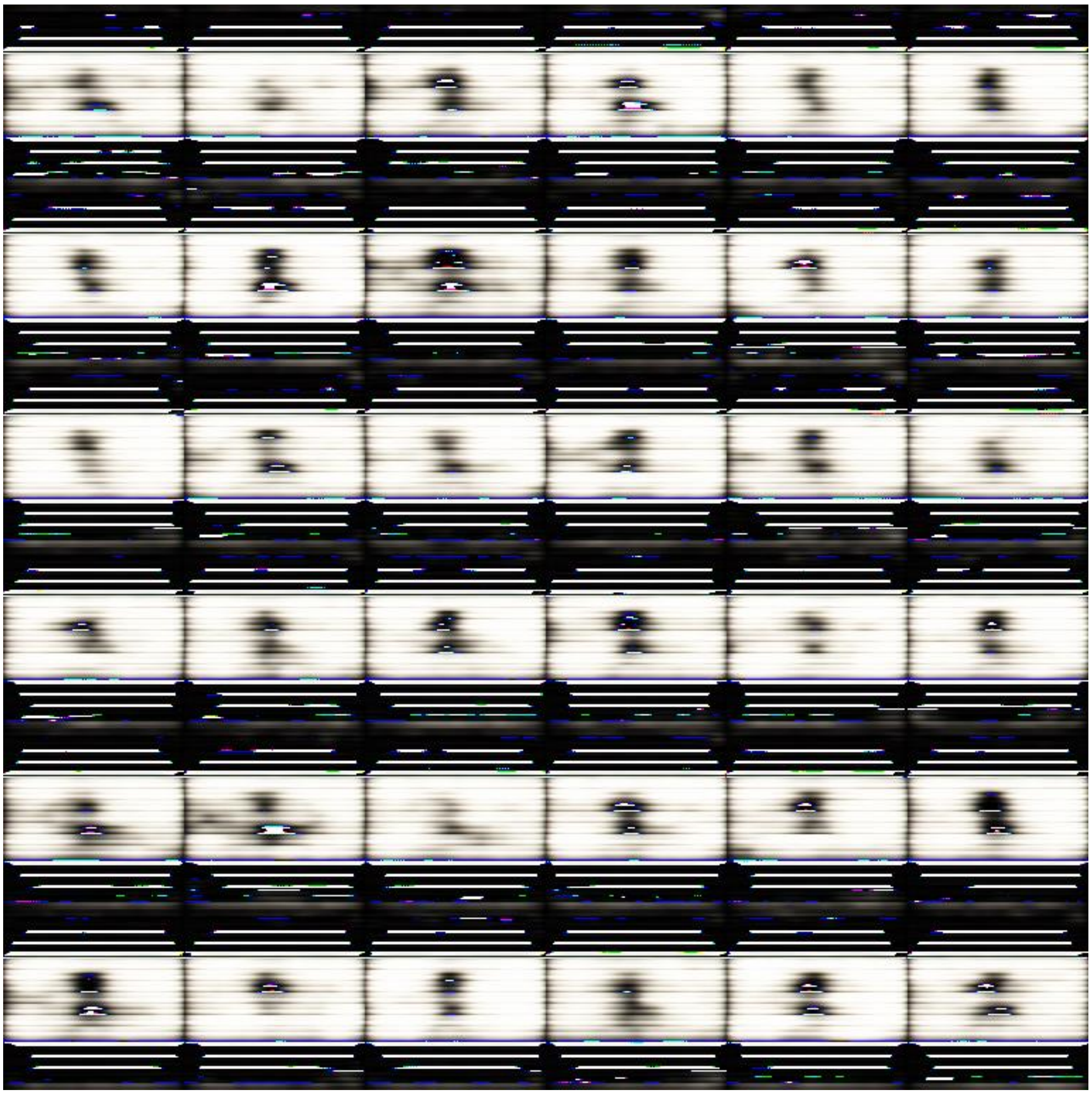
```
def generate_and_save_img(model, epoch, test_input):
  predictions = model(test_input, training = False)

  fig = plt.figure(figsize = (10,10))

  for i in range(predictions.shape[0]):
    plt.subplot(4,4, i+1)
    plt.imshow(np.array((predictions[i, :, :, :]*127.5 + 127.5), np.int32))
    plt.axis('off')

  plt.savefig('results_2/image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()
```

```
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
```

```
    train_step(image_batch)

  display.clear_output(wait=True)
  generate_and_save_img(gen, epoch+1, seed)

  if(epoch+1)%5==0:
    gen.save(checkpoint_dir)

  print('Time for epoch {} is {} secs'.format(epoch+1, time.time()-start))

 display.clear_output(wait=True)
 generate_and_save_img(gen, epochs, seed)
```

```
train(train_dataset, EPOCHS)
```

Output:



Ex.no: 02 **Style Transfer**

Code:

```
import numpy as np
import tensorflow as tf

from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.applications.vgg19 import preprocess_input

from tensorflow.keras.applications.vgg19 import VGG19
from tensorflow.keras.models import Model

import tensorflow.keras.backend as K

from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
```

```
def load_and_preprocess_image(image_path, target_size=(256, 256)):
    image = load_img(image_path, target_size=target_size)
    image = img_to_array(image)
    image = np.expand_dims(image, axis=0)
    image = preprocess_input(image)
    return image
```

```
content_image =
load_and_preprocess_image('/content/drive/MyDrive/DS&Op/content_images/1.png')
style_image = load_and_preprocess_image('/content/drive/MyDrive/DS&Op/style_images/1.png')
```

```
base_model = VGG19(weights='imagenet', include_top=False)

# Extract output from intermediate layers for style representation
style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'block5_conv1']
content_layer = 'block5_conv2'
style_outputs = [base_model.get_layer(layer).output for layer in style_layers]
model = Model(inputs=base_model.input, outputs=style_outputs)
```

```
def gram_matrix(input_tensor):
    assert K.ndim(input_tensor) == 4
    channels = int(input_tensor.shape[-1])
    a = tf.reshape(input_tensor, (-1, channels))
    n = tf.shape(a)[0]
```

```python
    gram = tf.matmul(a, a, transpose_a=True)
    return gram / tf.cast(n, tf.float32)
```

```python
def style_loss(style, generated):
    return K.sum(K.square(gram_matrix(style) - gram_matrix(generated))) / (4.0 * (style.shape[0] **
2) * (style.shape[1] ** 2) * (style.shape[2] ** 2))
```

```python
def content_loss(content, generated):
    return K.sum(K.square(generated - content))
```

```python
def total_variation_loss(image):
    a = K.square(image[:, :-1, :-1, :] - image[:, 1:, :-1, :])
    b = K.square(image[:, :-1, :-1, :] - image[:, :-1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

```python
alpha = 1.0  # Content weight
beta = 0.01  # Style weight
gamma = 0.01 # Total Variation weight (optional)
```

```python
def total_loss(content_image, style_image, generated_image):
    content_features = model(content_image)
    style_features = model(style_image)
    generated_features = model(generated_image)

    content_loss_value = content_loss(content_features[-1], generated_features[-1])

    style_loss_value = 0.0
    for i in range(len(style_layers)):
        style_loss_value += style_loss(style_features[i], generated_features[i])

    total_loss = (alpha * content_loss_value) + (beta * style_loss_value)

    # Add total variation loss (optional)
    if gamma > 0.0:
        total_loss += gamma * total_variation_loss(generated_image)

    return total_loss
```

```python
generated_image = tf.Variable(content_image, dtype=tf.float32)

# Choose the number of iterations for optimization
iterations = 100
```

```
optimizer = Adam(learning_rate=2.0, beta_1=0.99, epsilon=1e-1)
```

```
def deprocess_image(image):
    image = image.reshape((256, 256, 3))
    # Undo the preprocessing normalization
    image[:, :, 0] += 103.939
    image[:, :, 1] += 116.779
    image[:, :, 2] += 123.68
    # Convert from BGR to RGB
    image = image[:, :, ::-1]
    # Clip values to [0, 255] range
    image = np.clip(image, 0, 255).astype('uint8')
    return image
```

```
# Perform style transfer
for i in range(iterations):
    with tf.GradientTape() as tape:
        loss = total_loss(content_image, style_image, generated_image)

    gradients = tape.gradient(loss, generated_image)
    optimizer.apply_gradients([(gradients, generated_image)])
    generated_image.assign(tf.clip_by_value(generated_image, 0.0, 255.0))

    # Deprocess the generated_image tensor
    generated_image_o = deprocess_image(generated_image.numpy())

    # Clip values to [0, 255] range and convert to uint8
    generated_image_o = np.clip(generated_image_o, 0, 255).astype('uint8')
```

```
# Deprocess the generated_image tensor
generated_image = deprocess_image(generated_image.numpy())

# Clip values to [0, 255] range and convert to uint8
generated_image = np.clip(generated_image, 0, 255).astype('uint8')

# Display the generated image
plt.imshow(generated_image)
plt.axis('off')
plt.show()

# Save the generated image
plt.imsave('/content/drive/MyDrive/DS&Op/Result_ST_GAN/generated_image.jpg',
generated_image)
```

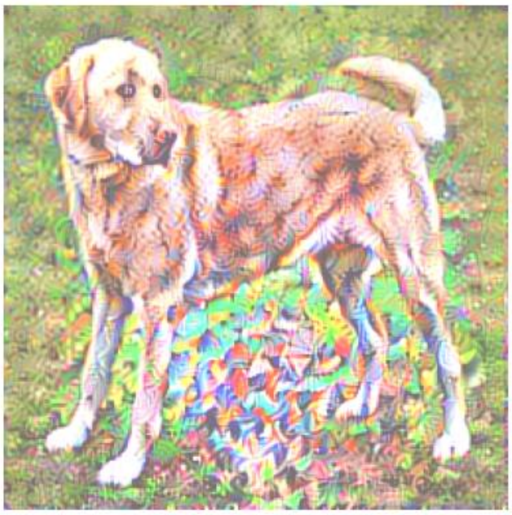
Output:

Content Image:

Style Image:

Epoch 1:

Epoch 1000:



Ex.no: 03 **Generate new faces using stylegan**

Code:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import OrderedDict
```

```python
import torchvision
import matplotlib.pyplot as plt
import pickle
import numpy as np
import os
import IPython
```

```python
class MyLinear(nn.Module):
    """Linear layer with equalized learning rate and custom learning rate multiplier."""
    def __init__(self, input_size, output_size, gain=2**(0.5), use_wscale=False, lrmul=1, bias=True):
        super().__init__()
        he_std = gain * input_size**(-0.5) # He init
        # Equalized learning rate and custom learning rate multiplier.
        if use_wscale:
            init_std = 1.0 / lrmul
            self.w_mul = he_std * lrmul
        else:
            init_std = he_std / lrmul
            self.w_mul = lrmul
        self.weight = torch.nn.Parameter(torch.randn(output_size, input_size) * init_std)
        if bias:
            self.bias = torch.nn.Parameter(torch.zeros(output_size))
            self.b_mul = lrmul
        else:
            self.bias = None

    def forward(self, x):
        bias = self.bias
        if bias is not None:
            bias = bias * self.b_mul
        return F.linear(x, self.weight * self.w_mul, bias)
```

```python
class MyConv2d(nn.Module):
    """Conv layer with equalized learning rate and custom learning rate multiplier."""
    def __init__(self, input_channels, output_channels, kernel_size, gain=2**(0.5), use_wscale=False,
lrmul=1, bias=True,
                intermediate=None, upscale=False):
        super().__init__()
        if upscale:
            self.upscale = Upscale2d()
        else:
            self.upscale = None
        he_std = gain * (input_channels * kernel_size ** 2) ** (-0.5) # He init
        self.kernel_size = kernel_size
        if use_wscale:
            init_std = 1.0 / lrmul
            self.w_mul = he_std * lrmul
        else:
            init_std = he_std / lrmul
            self.w_mul = lrmul
        self.weight = torch.nn.Parameter(torch.randn(output_channels, input_channels, kernel_size,
kernel_size) * init_std)
        if bias:
            self.bias = torch.nn.Parameter(torch.zeros(output_channels))
            self.b_mul = lrmul
        else:
            self.bias = None
        self.intermediate = intermediate

    def forward(self, x):
        bias = self.bias
        if bias is not None:
            bias = bias * self.b_mul

        have_convolution = False
        if self.upscale is not None and min(x.shape[2:]) * 2 >= 128:
            # this is the fused upscale + conv from StyleGAN, sadly this seems incompatible with the non-
fused way
            # this really needs to be cleaned up and go into the conv...
            w = self.weight * self.w_mul
            w = w.permute(1, 0, 2, 3)
            # probably applying a conv on w would be more efficient. also this quadruples the weight
(average)?!
            w = F.pad(w, (1,1,1,1))
            w = w[:, :, 1:, 1:]+ w[:, :, :-1, 1:] + w[:, :, 1:, :-1] + w[:, :, :-1, :-1]
            x = F.conv_transpose2d(x, w, stride=2, padding=(w.size(-1)-1)//2)
            have_convolution = True
        elif self.upscale is not None:
            x = self.upscale(x)

        if not have_convolution and self.intermediate is None:
            return F.conv2d(x, self.weight * self.w_mul, bias, padding=self.kernel_size//2)
        elif not have_convolution:
            x = F.conv2d(x, self.weight * self.w_mul, None, padding=self.kernel_size//2)

        if self.intermediate is not None:
            x = self.intermediate(x)
        if bias is not None:
            x = x + bias.view(1, -1, 1, 1)
        return x
```

```
class NoiseLayer(nn.Module):
    """adds noise. noise is per pixel (constant over channels) with per-channel weight"""
    def __init__(self, channels):
        super().__init__()
        self.weight = nn.Parameter(torch.zeros(channels))
        self.noise = None

    def forward(self, x, noise=None):
        if noise is None and self.noise is None:
            noise = torch.randn(x.size(0), 1, x.size(2), x.size(3), device=x.device, dtype=x.dtype)
        elif noise is None:
            # here is a little trick: if you get all the noiselayers and set each
            # modules .noise attribute, you can have pre-defined noise.
            # Very useful for analysis
            noise = self.noise
        x = x + self.weight.view(1, -1, 1, 1) * noise
        return x
```

```
class StyleMod(nn.Module):
    def __init__(self, latent_size, channels, use_wscale):
        super(StyleMod, self).__init__()
        self.lin = MyLinear(latent_size,
                            channels * 2,
                            gain=1.0, use_wscale=use_wscale)

    def forward(self, x, latent):
        style = self.lin(latent) # style => [batch_size, n_channels*2]
        shape = [-1, 2, x.size(1)] + (x.dim() - 2) * [1]
        style = style.view(shape)  # [batch_size, 2, n_channels, ...]
        x = x * (style[:, 0] + 1.) + style[:, 1]
        return x
```

```
class StyleMod(nn.Module):
    def __init__(self, latent_size, channels, use_wscale):
        super(StyleMod, self).__init__()
        self.lin = MyLinear(latent_size,
                            channels * 2,
                            gain=1.0, use_wscale=use_wscale)

    def forward(self, x, latent):
        style = self.lin(latent) # style => [batch_size, n_channels*2]
        shape = [-1, 2, x.size(1)] + (x.dim() - 2) * [1]
        style = style.view(shape)  # [batch_size, 2, n_channels, ...]
        x = x * (style[:, 0] + 1.) + style[:, 1]
        return x
```

```
class BlurLayer(nn.Module):
    def __init__(self, kernel=[1, 2, 1], normalize=True, flip=False, stride=1):
        super(BlurLayer, self).__init__()
        kernel=[1, 2, 1]
        kernel = torch.tensor(kernel, dtype=torch.float32)
        kernel = kernel[:, None] * kernel[None, :]
        kernel = kernel[None, None]
        if normalize:
            kernel = kernel / kernel.sum()
        if flip:
            kernel = kernel[:, :, ::-1, ::-1]
        self.register_buffer('kernel', kernel)
        self.stride = stride

    def forward(self, x):
        # expand kernel channels
        kernel = self.kernel.expand(x.size(1), -1, -1, -1)
        x = F.conv2d(
            x,
            kernel,
            stride=self.stride,
            padding=int((self.kernel.size(2)-1)/2),
            groups=x.size(1)
        )
        return x
```

```
def upscale2d(x, factor=2, gain=1):
    assert x.dim() == 4
    if gain != 1:
        x = x * gain
    if factor != 1:
        shape = x.shape
        x = x.view(shape[0], shape[1], shape[2], 1, shape[3], 1).expand(-1, -1, -1, factor, -1, factor)
        x = x.contiguous().view(shape[0], shape[1], factor * shape[2], factor * shape[3])
    return x

class Upscale2d(nn.Module):
    def __init__(self, factor=2, gain=1):
        super().__init__()
        assert isinstance(factor, int) and factor >= 1
        self.gain = gain
        self.factor = factor
    def forward(self, x):
        return upscale2d(x, factor=self.factor, gain=self.gain)
```

```
class G_mapping(nn.Sequential):
    def __init__(self, nonlinearity='lrelu', use_wscale=True):
        act, gain = {'relu': (torch.relu, np.sqrt(2)),
                     'lrelu': (nn.LeakyReLU(negative_slope=0.2), np.sqrt(2))}[nonlinearity]
        layers = [
            ('pixel_norm', PixelNormLayer()),
            ('dense0', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense0_act', act),
            ('dense1', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense1_act', act),
            ('dense2', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense2_act', act),
            ('dense3', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense3_act', act),
            ('dense4', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense4_act', act),
            ('dense5', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense5_act', act),
            ('dense6', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense6_act', act),
            ('dense7', MyLinear(512, 512, gain=gain, lrmul=0.01, use_wscale=use_wscale)),
            ('dense7_act', act)
        ]
        super().__init__(OrderedDict(layers))

    def forward(self, x):
        x = super().forward(x)
        # Broadcast
        x = x.unsqueeze(1).expand(-1, 18, -1)
        return x
```

```
class LayerEpilogue(nn.Module):
    """Things to do at the end of each layer."""
    def __init__(self, channels, dlatent_size, use_wscale, use_noise, use_pixel_norm,
use_instance_norm, use_styles, activation_layer):
        super().__init__()
        layers = []
        if use_noise:
            layers.append(('noise', NoiseLayer(channels)))
        layers.append(('activation', activation_layer))
        if use_pixel_norm:
            layers.append(('pixel_norm', PixelNorm()))
        if use_instance_norm:
            layers.append(('instance_norm', nn.InstanceNorm2d(channels)))
        self.top_epi = nn.Sequential(OrderedDict(layers))
        if use_styles:
            self.style_mod = StyleMod(dlatent_size, channels, use_wscale=use_wscale)
        else:
            self.style_mod = None
    def forward(self, x, dlatents_in_slice=None):
        x = self.top_epi(x)
        if self.style_mod is not None:
            x = self.style_mod(x, dlatents_in_slice)
        else:
            assert dlatents_in_slice is None
        return x
```

```
class InputBlock(nn.Module):
    def __init__(self, nf, dlatent_size, const_input_layer, gain, use_wscale, use_noise, use_pixel_norm,
use_instance_norm, use_styles, activation_layer):
        super().__init__()
        self.const_input_layer = const_input_layer
        self.nf = nf
        if self.const_input_layer:
            # called 'const' in tf
            self.const = nn.Parameter(torch.ones(1, nf, 4, 4))
            self.bias = nn.Parameter(torch.ones(nf))
        else:
            self.dense = MyLinear(dlatent_size, nf*16, gain=gain/4, use_wscale=use_wscale) # tweak gain
to match the official implementation of Progressing GAN
        self.epi1 = LayerEpilogue(nf, dlatent_size, use_wscale, use_noise, use_pixel_norm,
use_instance_norm, use_styles, activation_layer)
        self.conv = MyConv2d(nf, nf, 3, gain=gain, use_wscale=use_wscale)
        self.epi2 = LayerEpilogue(nf, dlatent_size, use_wscale, use_noise, use_pixel_norm,
use_instance_norm, use_styles, activation_layer)

    def forward(self, dlatents_in_range):
        batch_size = dlatents_in_range.size(0)
        if self.const_input_layer:
            x = self.const.expand(batch_size, -1, -1, -1)
            x = x + self.bias.view(1, -1, 1, 1)
        else:
            x = self.dense(dlatents_in_range[:, 0]).view(batch_size, self.nf, 4, 4)
        x = self.epi1(x, dlatents_in_range[:, 0])
        x = self.conv(x)
        x = self.epi2(x, dlatents_in_range[:, 1])
        return x
```

```
class GSynthesisBlock(nn.Module):
    def __init__(self, in_channels, out_channels, blur_filter, dlatent_size, gain, use_wscale, use_noise,
use_pixel_norm, use_instance_norm, use_styles, activation_layer):
        # 2**res x 2**res # res = 3..resolution_log2
        super().__init__()
        if blur_filter:
            blur = BlurLayer(blur_filter)
        else:
            blur = None
        self.conv0_up = MyConv2d(in_channels, out_channels, kernel_size=3, gain=gain,
use_wscale=use_wscale,
                                 intermediate=blur, upscale=True)
        self.epi1 = LayerEpilogue(out_channels, dlatent_size, use_wscale, use_noise, use_pixel_norm,
use_instance_norm, use_styles, activation_layer)
        self.conv1 = MyConv2d(out_channels, out_channels, kernel_size=3, gain=gain,
use_wscale=use_wscale)
        self.epi2 = LayerEpilogue(out_channels, dlatent_size, use_wscale, use_noise, use_pixel_norm,
use_instance_norm, use_styles, activation_layer)

    def forward(self, x, dlatents_in_range):
        x = self.conv0_up(x)
        x = self.epi1(x, dlatents_in_range[:, 0])
        x = self.conv1(x)
        x = self.epi2(x, dlatents_in_range[:, 1])
        return x
```

```
class G_synthesis(nn.Module):
    def __init__(self,
        dlatent_size        = 512,          # Disentangled latent (W) dimensionality.
        num_channels        = 3,            # Number of output color channels.
        resolution          = 1024,         # Output resolution.
        fmap_base           = 8192,         # Overall multiplier for the number of feature maps.
        fmap_decay          = 1.0,          # log2 feature map reduction when doubling the resolution.
        fmap_max            = 512,          # Maximum number of feature maps in any layer.
        use_styles          = True,         # Enable style inputs?
        const_input_layer   = True,         # First layer is a learned constant?
        use_noise           = True,         # Enable noise inputs?
        randomize_noise     = True,         # True = randomize noise inputs every time (non-
deterministic), False = read noise inputs from variables.
        nonlinearity        = 'lrelu',      # Activation function: 'relu', 'lrelu'
        use_wscale          = True,         # Enable equalized learning rate?
        use_pixel_norm      = False,        # Enable pixelwise feature vector normalization?
        use_instance_norm   = True,         # Enable instance normalization?
        dtype               = torch.float32,  # Data type to use for activations and outputs.
        blur_filter         = [1,2,1],      # Low-pass filter to apply when resampling activations. None = no
filtering.
        ):

        super().__init__()
        def nf(stage):
            return min(int(fmap_base / (2.0 ** (stage * fmap_decay))), fmap_max)
        self.dlatent_size = dlatent_size
        resolution_log2 = int(np.log2(resolution))
        assert resolution == 2**resolution_log2 and resolution >= 4

        act, gain = {'relu': (torch.relu, np.sqrt(2)),
                     'lrelu': (nn.LeakyReLU(negative_slope=0.2), np.sqrt(2))}[nonlinearity]
        num_layers = resolution_log2 * 2 - 2
        num_styles = num_layers if use_styles else 1
        torgbs = []
        blocks = []
        for res in range(2, resolution_log2 + 1):
            channels = nf(res-1)
            name = '{s}x{s}'.format(s=2**res)
            if res == 2:
                blocks.append((name,
                               InputBlock(channels, dlatent_size, const_input_layer, gain, use_wscale,
                                      use_noise, use_pixel_norm, use_instance_norm, use_styles, act)))

            else:
                blocks.append((name,
                               GSynthesisBlock(last_channels, channels, blur_filter, dlatent_size, gain,
use_wscale, use_noise, use_pixel_norm, use_instance_norm, use_styles, act)))
            last_channels = channels
        self.torgb = MyConv2d(channels, num_channels, 1, gain=1, use_wscale=use_wscale)
        self.blocks = nn.ModuleDict(OrderedDict(blocks))

    def forward(self, dlatents_in):
        # Input: Disentangled latents (W) [minibatch, num_layers, dlatent_size].
        # lod_in = tf.cast(tf.get_variable('lod', initializer=np.float32(0), trainable=False), dtype)
        batch_size = dlatents_in.size(0)
        for i, m in enumerate(self.blocks.values()):
            if i == 0:
                x = m(dlatents_in[:, 2*i:2*i+2])
            else:
                x = m(x, dlatents_in[:, 2*i:2*i+2])
        rgb = self.torgb(x)
        return rgb
```

```
with torch.no_grad():
    imgs = g_all(latents)
    imgs = (imgs.clamp(-1, 1)+1)/2.0  # normalization to 0~1 range
imgs = imgs.cpu()

imgs = torchvision.utils.make_grid(imgs, nrow=nb_cols)

plt.figure(figsize=(15,6))
plt.imshow(imgs.permute(1,2,0).detach().numpy())
plt.axis('off')
plt.show()
```

Output:

Ex.no: 04 **Image-Image conversion using pix2pix**

Code:

```
import tensorflow as tf
import os
import pathlib
import time
import datetime
from matplotlib import pyplot as plt
from IPython import display
```

```
_URL = f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/{dataset_name}.tar.gz'


path_to_zip = tf.keras.utils.get_file(
    fname=f"{dataset_name}.tar.gz",
    origin=_URL,
    extract=True)

path_to_zip  = pathlib.Path(path_to_zip)

PATH = path_to_zip.parent/dataset_name
```

```
def load(image_file):
  # Read and decode an image file to a uint8 tensor
  image = tf.io.read_file(image_file)
  image = tf.io.decode_jpeg(image)

  # Split each image tensor into two tensors:
  # - one with a real building facade image
  # - one with an architecture label image
  w = tf.shape(image)[1]
  w = w // 2
  input_image = image[:, w:, :]
  real_image = image[:, :w, :]

  # Convert both images to float32 tensors
  input_image = tf.cast(input_image, tf.float32)
  real_image = tf.cast(real_image, tf.float32)

  return input_image, real_image
inp, re = load(str(PATH / 'train/100_8399_to_8397.jpg'))
# Casting to int for matplotlib to display the images
plt.figure()
plt.imshow(inp / 255.0)
plt.figure()
plt.imshow(re / 255.0)
```

```
# The facade training set consist of 400 images
BUFFER_SIZE = 400
# The batch size of 1 produced better results for the U-Net in the original pix2pix experiment
BATCH_SIZE = 1
# Each image is 256x256 in size
IMG_WIDTH = 256
IMG_HEIGHT = 256
```

```
def resize(input_image, real_image, height, width):
  input_image = tf.image.resize(input_image, [height, width],
                                method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
  real_image = tf.image.resize(real_image, [height, width],
                               method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

  return input_image, real_image
```

```
def random_crop(input_image, real_image):
  stacked_image = tf.stack([input_image, real_image], axis=0)
  cropped_image = tf.image.random_crop(
      stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])

  return cropped_image[0], cropped_image[1]
```

```
# Normalizing the images to [-1, 1]
def normalize(input_image, real_image):
  input_image = (input_image / 127.5) - 1
  real_image = (real_image / 127.5) - 1

  return input_image, real_image
```

```
@tf.function()
def random_jitter(input_image, real_image):
  # Resizing to 286x286
  input_image, real_image = resize(input_image, real_image, 286, 286)

  # Random cropping back to 256x256
  input_image, real_image = random_crop(input_image, real_image)

  if tf.random.uniform(()) > 0.5:
    # Random mirroring
    input_image = tf.image.flip_left_right(input_image)
    real_image = tf.image.flip_left_right(real_image)

  return input_image, real_image
```

```
plt.figure(figsize=(6, 6))
for i in range(4):
  rj_inp, rj_re = random_jitter(inp, re)
  plt.subplot(2, 2, i + 1)
  plt.imshow(rj_inp / 255.0)
  plt.axis('off')
plt.show()
```

```python
def load_image_train(image_file):
  input_image, real_image = load(image_file)
  input_image, real_image = random_jitter(input_image, real_image)
  input_image, real_image = normalize(input_image, real_image)

  return input_image, real_image
```

```python
def load_image_test(image_file):
  input_image, real_image = load(image_file)
  input_image, real_image = resize(input_image, real_image,
                                   IMG_HEIGHT, IMG_WIDTH)
  input_image, real_image = normalize(input_image, real_image)

  return input_image, real_image
```

```python
train_dataset = tf.data.Dataset.list_files(str(PATH / 'train/*.jpg'))
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)
```

```python
try:
  test_dataset = tf.data.Dataset.list_files(str(PATH / 'test/*.jpg'))
except tf.errors.InvalidArgumentError:
  test_dataset = tf.data.Dataset.list_files(str(PATH / 'val/*.jpg'))
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

```python
def downsample(filters, size, apply_batchnorm=True):
  initializer = tf.random_normal_initializer(0., 0.02)

  result = tf.keras.Sequential()
  result.add(
      tf.keras.layers.Conv2D(filters, size, strides=2, padding='same',
                             kernel_initializer=initializer, use_bias=False))

  if apply_batchnorm:
    result.add(tf.keras.layers.BatchNormalization())

  result.add(tf.keras.layers.LeakyReLU())

  return result
```

```
def upsample(filters, size, apply_dropout=False):
  initializer = tf.random_normal_initializer(0., 0.02)

  result = tf.keras.Sequential()
  result.add(
    tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
                                    padding='same',
                                    kernel_initializer=initializer,
                                    use_bias=False))

  result.add(tf.keras.layers.BatchNormalization())

  if apply_dropout:
      result.add(tf.keras.layers.Dropout(0.5))

  result.add(tf.keras.layers.ReLU())

  return result
```

```
up_model = upsample(3, 4)
up_result = up_model(down_result)
print (up_result.shape)
def Generator():
  inputs = tf.keras.layers.Input(shape=[256, 256, 3])

  down_stack = [
    downsample(64, 4, apply_batchnorm=False),  # (batch_size, 128, 128, 64)
    downsample(128, 4),  # (batch_size, 64, 64, 128)
    downsample(256, 4),  # (batch_size, 32, 32, 256)
    downsample(512, 4),  # (batch_size, 16, 16, 512)
    downsample(512, 4),  # (batch_size, 8, 8, 512)
    downsample(512, 4),  # (batch_size, 4, 4, 512)
    downsample(512, 4),  # (batch_size, 2, 2, 512)
    downsample(512, 4),  # (batch_size, 1, 1, 512)
  ]

  up_stack = [
    upsample(512, 4, apply_dropout=True),  # (batch_size, 2, 2, 1024)
    upsample(512, 4, apply_dropout=True),  # (batch_size, 4, 4, 1024)
    upsample(512, 4, apply_dropout=True),  # (batch_size, 8, 8, 1024)
    upsample(512, 4),  # (batch_size, 16, 16, 1024)
    upsample(256, 4),  # (batch_size, 32, 32, 512)
    upsample(128, 4),  # (batch_size, 64, 64, 256)
    upsample(64, 4),  # (batch_size, 128, 128, 128)
  ]

  initializer = tf.random_normal_initializer(0., 0.02)
  last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                         strides=2,
                                         padding='same',
                                         kernel_initializer=initializer,
                                         activation='tanh')  # (batch_size, 256, 256, 3)

  x = inputs

  # Downsampling through the model
  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])

  # Upsampling and establishing the skip connections
  for up, skip in zip(up_stack, skips):
    x = up(x)
    x = tf.keras.layers.Concatenate()([x, skip])

  x = last(x)

  return tf.keras.Model(inputs=inputs, outputs=x)
```

```
def generator_loss(disc_generated_output, gen_output, target):
 gan_loss = loss_object(tf.ones_like(disc_generated_output), disc_generated_output)

 # Mean absolute error
 l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

 total_gen_loss = gan_loss + (LAMBDA * l1_loss)

 return total_gen_loss, gan_loss, l1_loss
```

```
def Discriminator():
  initializer = tf.random_normal_initializer(0., 0.02)

  inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
  tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

  x = tf.keras.layers.concatenate([inp, tar])  # (batch_size, 256, 256, channels*2)

  down1 = downsample(64, 4, False)(x)  # (batch_size, 128, 128, 64)
  down2 = downsample(128, 4)(down1)  # (batch_size, 64, 64, 128)
  down3 = downsample(256, 4)(down2)  # (batch_size, 32, 32, 256)

  zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3)  # (batch_size, 34, 34, 256)
  conv = tf.keras.layers.Conv2D(512, 4, strides=1,
                                kernel_initializer=initializer,
                                use_bias=False)(zero_pad1)  # (batch_size, 31, 31, 512)

  batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

  leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

  zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu)  # (batch_size, 33, 33, 512)

  last = tf.keras.layers.Conv2D(1, 4, strides=1,
                                kernel_initializer=initializer)(zero_pad2)  # (batch_size, 30, 30, 1)

  return tf.keras.Model(inputs=[inp, tar], outputs=last)
```

```
disc_out = discriminator([inp[tf.newaxis, ...], gen_output], training=False)
plt.imshow(disc_out[0, ..., -1], vmin=-20, vmax=20, cmap='RdBu_r')
plt.colorbar()
def discriminator_loss(disc_real_output, disc_generated_output):
  real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

  generated_loss = loss_object(tf.zeros_like(disc_generated_output), disc_generated_output)

  total_disc_loss = real_loss + generated_loss

  return total_disc_loss
```

```
def generate_images(model, test_input, tar):
  prediction = model(test_input, training=True)
  plt.figure(figsize=(15, 15))
```

```
 display_list = [test_input[0], tar[0], prediction[0]]
 title = ['Input Image', 'Ground Truth', 'Predicted Image']

 for i in range(3):
   plt.subplot(1, 3, i+1)
   plt.title(title[i])
   # Getting the pixel values in the [0, 1] range to plot.
   plt.imshow(display_list[i] * 0.5 + 0.5)
   plt.axis('off')
 plt.show()
```

```
@tf.function
def train_step(input_image, target, step):
 with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
   gen_output = generator(input_image, training=True)

   disc_real_output = discriminator([input_image, target], training=True)
   disc_generated_output = discriminator([input_image, gen_output], training=True)

   gen_total_loss, gen_gan_loss, gen_l1_loss = generator_loss(disc_generated_output, gen_output,
target)
   disc_loss = discriminator_loss(disc_real_output, disc_generated_output)

 generator_gradients = gen_tape.gradient(gen_total_loss,
                                         generator.trainable_variables)
 discriminator_gradients = disc_tape.gradient(disc_loss,
                                              discriminator.trainable_variables)

 generator_optimizer.apply_gradients(zip(generator_gradients,
                                         generator.trainable_variables))
 discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                             discriminator.trainable_variables))

 with summary_writer.as_default():
   tf.summary.scalar('gen_total_loss', gen_total_loss, step=step//1000)
   tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=step//1000)
   tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=step//1000)
   tf.summary.scalar('disc_loss', disc_loss, step=step//1000)
```

```
def fit(train_ds, test_ds, steps):
 example_input, example_target = next(iter(test_ds.take(1)))
```

```
  start = time.time()

  for step, (input_image, target) in train_ds.repeat().take(steps).enumerate():
    if (step) % 1000 == 0:
      display.clear_output(wait=True)

      if step != 0:
        print(f'Time taken for 1000 steps: {time.time()-start:.2f} sec\n')

      start = time.time()

      generate_images(generator, example_input, example_target)
      print(f"Step: {step//1000}k")

    train_step(input_image, target, step)

    # Training step
    if (step+1) % 10 == 0:
      print('.', end='', flush=True)


    # Save (checkpoint) the model every 5k steps
    if (step + 1) % 5000 == 0:
      checkpoint.save(file_prefix=checkpoint_prefix)
# Run the trained model on a few examples from the test set
for inp, tar in test_dataset.take(5):
  generate_images(generator, inp, tar)
```

```
# Restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

```
%load_ext tensorboard
%tensorboard --logdir {log_dir}
```

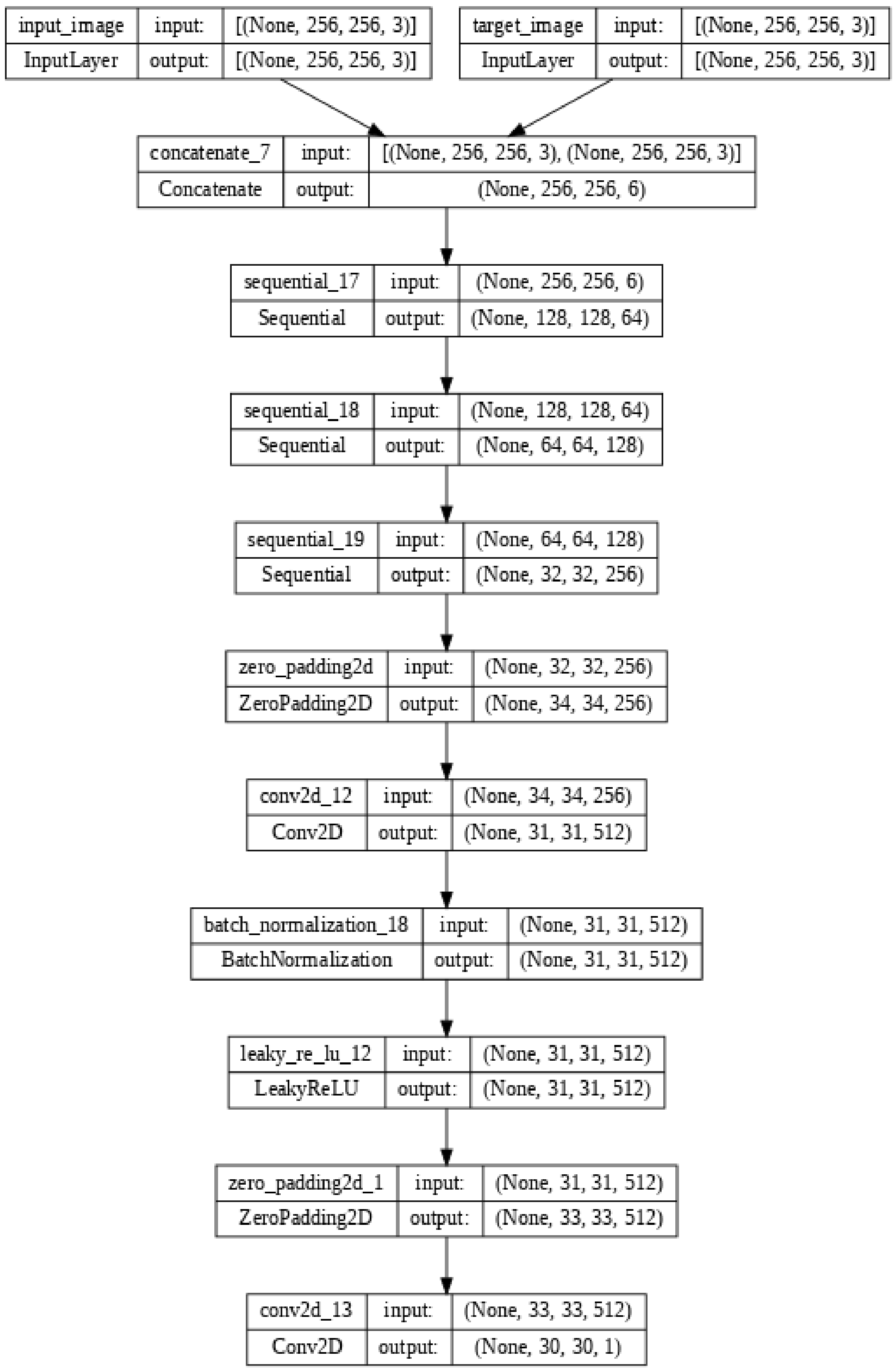
```
fit(train_dataset, test_dataset, steps=4000)
```

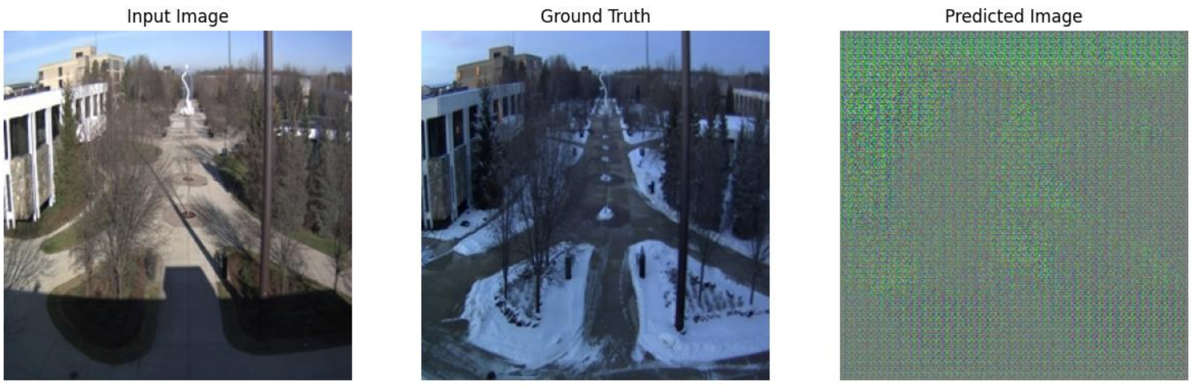
```
log_dir="logs/"

summary_writer = tf.summary.create_file_writer(
  log_dir + "fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
```

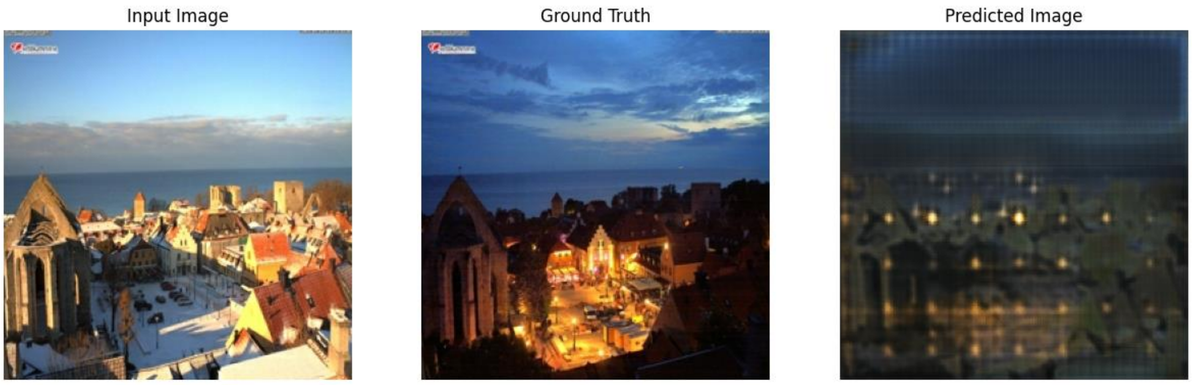
```
# Run the trained model on a few examples from the test set
for inp, tar in test_dataset.take(5):
```

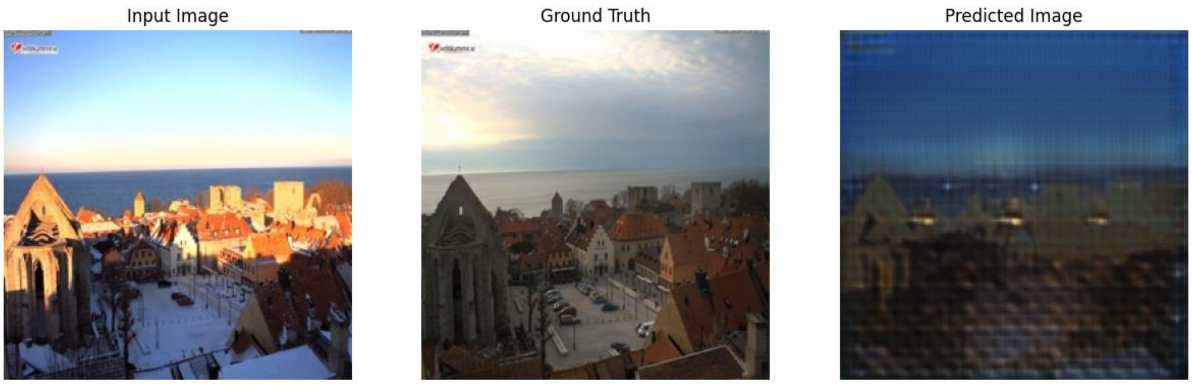
```
generate_images(generator, inp, tar)
```

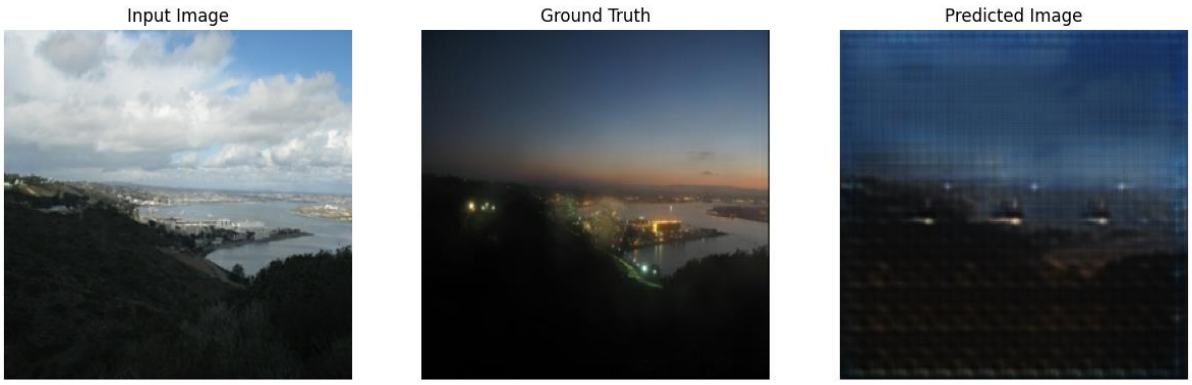
Output:


input_image | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
target_image | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
concatenate_7 | input: | [(None, 256, 256, 3), (None, 256, 256, 3)]
Concatenate | output: | (None, 256, 256, 6)
sequential_17 | input: | (None, 256, 256, 6)
Sequential | output: | (None, 128, 128, 64)
sequential_18 | input: | (None, 128, 128, 64)
Sequential | output: | (None, 64, 64, 128)
sequential_19 | input: | (None, 64, 64, 128)
Sequential | output: | (None, 32, 32, 256)
zero_padding2d | input: | (None, 32, 32, 256)
ZeroPadding2D | output: | (None, 34, 34, 256)
conv2d_12 | input: | (None, 34, 34, 256)
Conv2D | output: | (None, 31, 31, 512)
batch_normalization_18 | input: | (None, 31, 31, 512)
BatchNormalization | output: | (None, 31, 31, 512)
leaky_re_lu_12 | input: | (None, 31, 31, 512)
LeakyReLU | output: | (None, 31, 31, 512)
zero_padding2d_1 | input: | (None, 31, 31, 512)
ZeroPadding2D | output: | (None, 33, 33, 512)
conv2d_13 | input: | (None, 33, 33, 512)
Conv2D | output: | (None, 30, 30, 1)









Ex.no: 05 Autoencoders - Recreate a new image

Code:

```
# http://www.cs.columbia.edu/CAVE/databases/pubfig/download/lfw_attributes.txt
ATTRS_NAME = "lfw_attribute.txt"
```

```

# http://vis-www.cs.umass.edu/lfw/lfw-deepfunneled.tgz
IMAGES_NAME = "lfw-deepfunneled.tgz"
```

```
# http://vis-www.cs.umass.edu/lfw/lfw.tgz
RAW_IMAGES_NAME = "lfw.tgz"
```

```
def decode_image_from_raw_bytes(raw_bytes):
    img = cv2.imdecode(np.asarray(bytearray(raw_bytes), dtype=np.uint8), 1)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    return img
```

```
def load_lfw_dataset(
        use_raw=False,
        dx=80, dy=80,
        dimx=45, dimy=45):

    # Read attrs
    df_attrs = pd.read_csv(ATTRS_NAME, sep='\t', skiprows=1)
    df_attrs = pd.DataFrame(df_attrs.iloc[:, :-1].values, columns=df_attrs.columns[1:])
    imgs_with_attrs = set(map(tuple, df_attrs[["person", "imagenum"]].values))

    # Read photos
    all_photos = []
    photo_ids = []

    # tqdm in used to show progress bar while reading the data in a notebook here, you can change
    # tqdm_notebook to use it outside a notebook
    with tarfile.open(RAW_IMAGES_NAME if use_raw else IMAGES_NAME) as f:
        for m in tqdm.tqdm_notebook(f.getmembers()):
            # Only process image files from the compressed data
            if m.isfile() and m.name.endswith(".jpg"):
                # Prepare image
                img = decode_image_from_raw_bytes(f.extractfile(m).read())

                # Crop only faces and resize it
                img = img[dy:-dy, dx:-dx]
                img = cv2.resize(img, (dimx, dimy))

                # Parse person and append it to the collected data
                fname = os.path.split(m.name)[-1]
                fname_splitted = fname[:-4].replace('_', ' ').split()
                person_id = ' '.join(fname_splitted[:-1])
                photo_number = int(fname_splitted[-1])
                if (person_id, photo_number) in imgs_with_attrs:
                    all_photos.append(img)
                    photo_ids.append({'person': person_id, 'imagenum': photo_number})

    photo_ids = pd.DataFrame(photo_ids)
    all_photos = np.stack(all_photos).astype('uint8')

    # Preserve photo_ids order!
    all_attrs = photo_ids.merge(df_attrs, on=('person', 'imagenum')).drop(["person", "imagenum"],
axis=1)

    return all_photos, all_attrs
```

```
import numpy as np
import pandas as pd
import tarfile, zipfile
import tqdm
import cv2
import os
X, attr = load_lfw_dataset(use_raw=True, dimx=32, dimy=32)
```

```
X = X.astype('float32') / 255.0 - 0.5
```

```
import matplotlib.pyplot as plt
def show_image(x):
    plt.imshow(np.clip(x + 0.5, 0, 1))
from sklearn.model_selection import train_test_split
X_train, X_test = train_test_split(X, test_size=0.1, random_state=42)
```

```
from keras.layers import Dense, Flatten, Reshape, Input, InputLayer
from keras.models import Sequential, Model

def build_autoencoder(img_shape, code_size):
    # The encoder
    encoder = Sequential()
    encoder.add(InputLayer(img_shape))
    encoder.add(Flatten())
    encoder.add(Dense(code_size))

    # The decoder
    decoder = Sequential()
    decoder.add(InputLayer((code_size,)))
    decoder.add(Dense(np.prod(img_shape))) # np.prod(img_shape) is the same as 32*32*3, it's more
generic than saying 3072
    decoder.add(Reshape(img_shape))

    return encoder, decoder
```

```
# Same as (32,32,3), we neglect the number of instances from shape
IMG_SHAPE = X.shape[1:]
encoder, decoder = build_autoencoder(IMG_SHAPE, 32)

inp = Input(IMG_SHAPE)
code = encoder(inp)
reconstruction = decoder(code)

autoencoder = Model(inp,reconstruction)
autoencoder.compile(optimizer='adamax', loss='mse')

print(autoencoder.summary())
```

```
history = autoencoder.fit(x=X_train, y=X_train, epochs=20,
                validation_data=[X_test, X_test])
```

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
def visualize(img,encoder,decoder):
    """Draws original, encoded and decoded images"""
    # img[None] will have shape of (1, 32, 32, 3) which is the same as the model input
    code = encoder.predict(img[None])[0]
    reco = decoder.predict(code[None])[0]
    plt.subplot(1,3,1)
    plt.title("Original")
    show_image(img)

    plt.subplot(1,3,2)
    plt.title("Code")
    plt.imshow(code.reshape([code.shape[-1]//2,-1]))

    plt.subplot(1,3,3)
    plt.title("Reconstructed")
    show_image(reco)
    plt.show()

for i in range(5):
    img = X_test[i]
    visualize(img,encoder,decoder)
```

```
plt.subplot(1,4,1)
show_image(X_train[0])
plt.subplot(1,4,2)
show_image(apply_gaussian_noise(X_train[:1],sigma=0.01)[0])
plt.subplot(1,4,3)
show_image(apply_gaussian_noise(X_train[:1],sigma=0.1)[0])
plt.subplot(1,4,4)
show_image(apply_gaussian_noise(X_train[:1],sigma=0.5)[0])
```

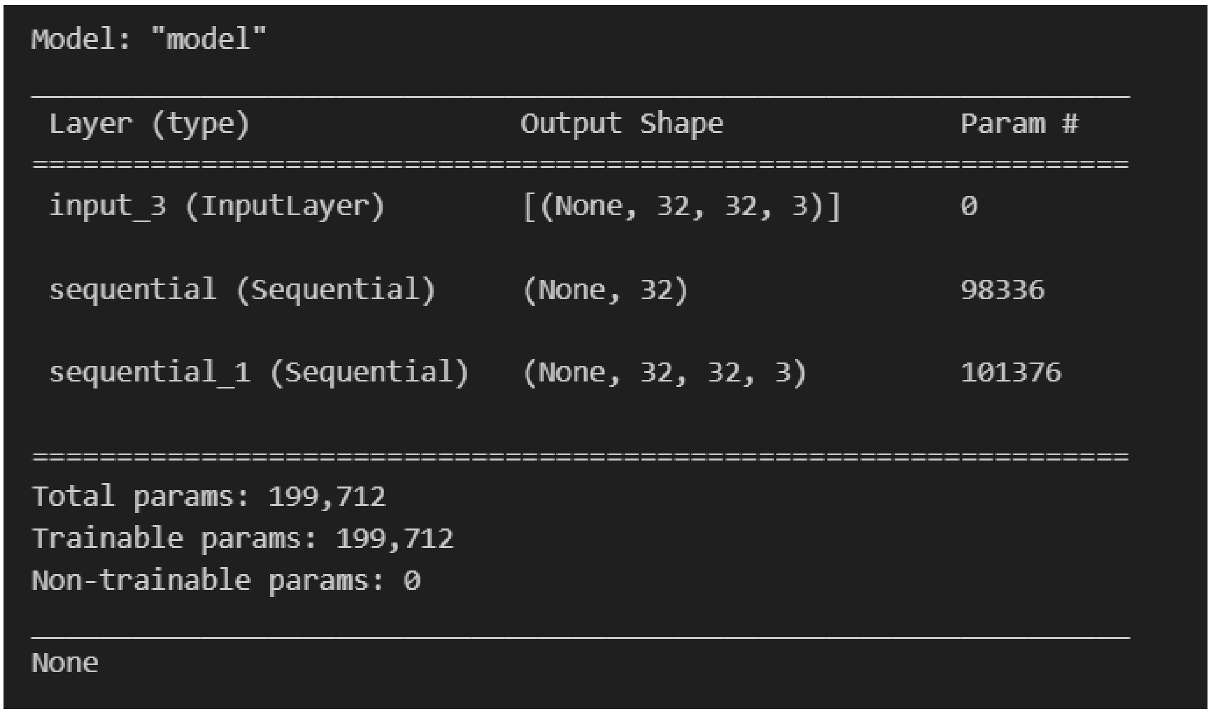
```
code_size = 100

# We can use bigger code size for better quality
encoder, decoder = build_autoencoder(IMG_SHAPE, code_size=code_size)

inp = Input(IMG_SHAPE)
code = encoder(inp)
reconstruction = decoder(code)

autoencoder = Model(inp, reconstruction)
autoencoder.compile('adamax', 'mse')

for i in range(25):
    print("Epoch %i/25, Generating corrupted samples..."%(i+1))
    X_train_noise = apply_gaussian_noise(X_train)
    X_test_noise = apply_gaussian_noise(X_test)
```

```
# We continue to train our model with new noise-augmented data
autoencoder.fit(x=X_train_noise, y=X_train, epochs=1,
                validation_data=[X_test_noise, X_test])
```

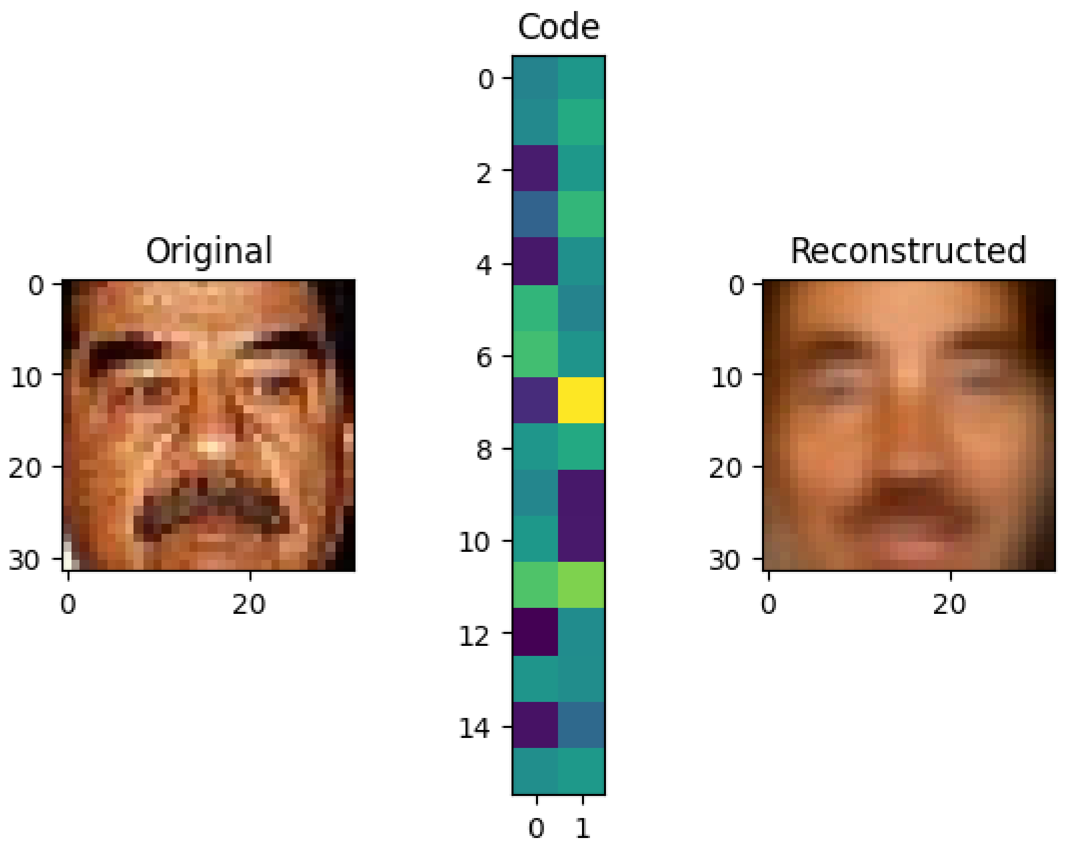
```
def apply_gaussian_noise(X, sigma=0.1):
    noise = np.random.normal(loc=0.0, scale=sigma, size=X.shape)
    return X + noise
```

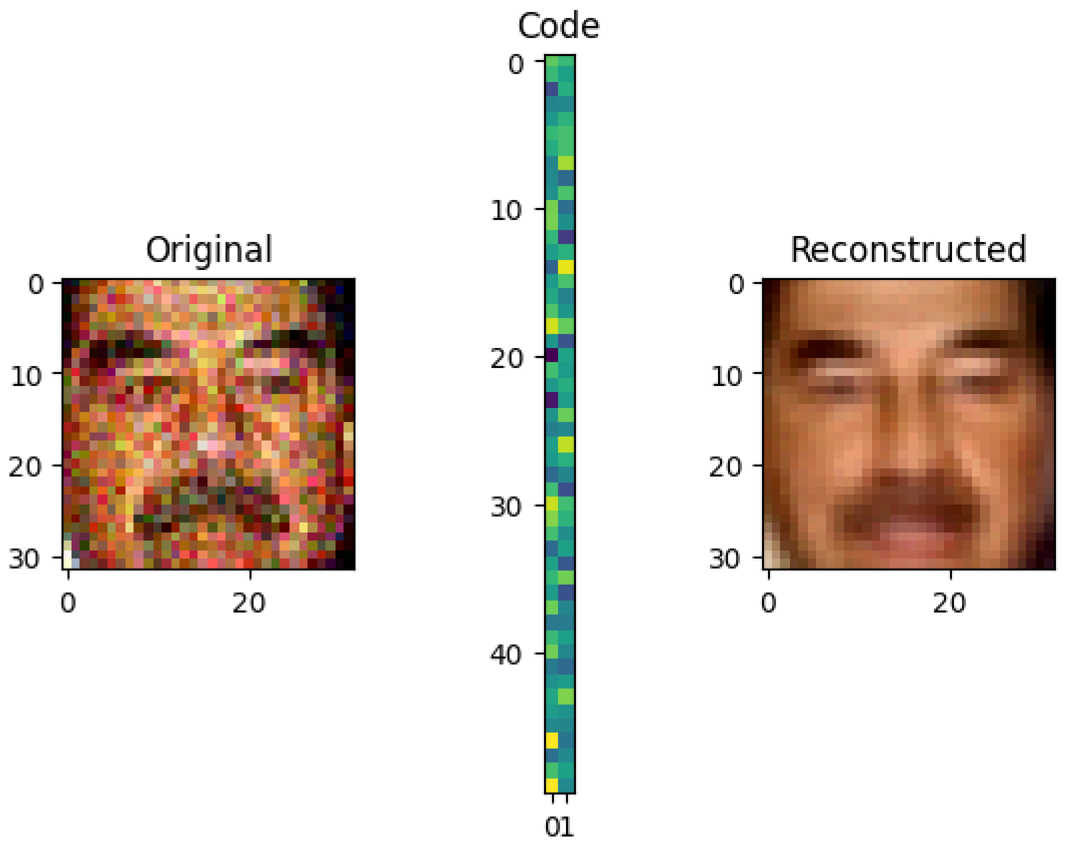
```
X_test_noise = apply_gaussian_noise(X_test)
for i in range(5):
    img = X_test_noise[i]
    visualize(img,encoder,decoder)
```

Output:



```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 32, 32, 3)]       0

 sequential (Sequential)     (None, 32)                98336

 sequential_1 (Sequential)   (None, 32, 32, 3)         101376

=================================================================
Total params: 199,712
Trainable params: 199,712
Non-trainable params: 0
_________________________________________________________________
None
```


Original
Code
Reconstructed
0
10
20
30
0
20
0
2
4
6
8
10
12
14
0
1


Original
Code
Reconstructed
0
10
20
30
0
20
0
10
20
30
40
01

Ex.no: 06 **Morphing and clone one face to another**

Code:

```
import cv2
import dlib
import numpy as np
import matplotlib.pyplot as plt
```

```
image_1 = cv2.imread("/content/Driver Gosling.jpg")
image_1 = cv2.cvtColor(image_1,cv2.COLOR_BGR2RGB)
image_1_gray = cv2.cvtColor(image_1,cv2.COLOR_BGR2GRAY)
mask = np.zeros_like(image_1_gray)

image_2 = cv2.imread("/content/Pat Bateman.jpg")
image_2 = cv2.cvtColor(image_2,cv2.COLOR_BGR2RGB)
image_2_gray = cv2.cvtColor(image_2,cv2.COLOR_BGR2GRAY)
```

```
plt.imshow(image_1)
```

```
plt.imshow(image_2)
```

```
def extract_index_nparray(nparray):
    index = None
    for num in nparray[0]:
        index = num
        break
    return index
```

```
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("/content/shape_predictor_68_face_landmarks.dat")
```

```
height, width, channels = image_2.shape
image_2_new_face = np.zeros((height, width, channels), np.uint8)
```

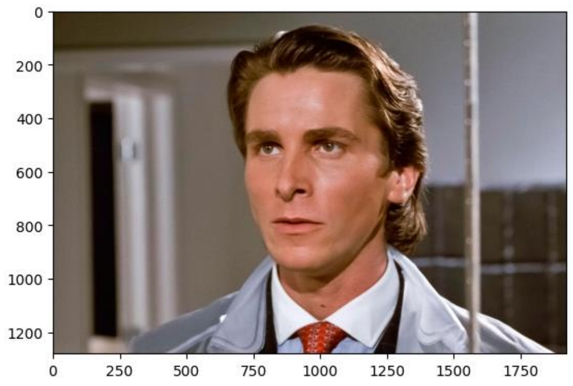
```
# Face 1
faces = detector(image_1_gray)
for face in faces:
    landmarks = predictor(image_1_gray, face)
    landmarks_points = []
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y
        landmarks_points.append((x, y))

    points = np.array(landmarks_points, np.int32)
    convexhull = cv2.convexHull(points)
    cv2.fillConvexPoly(mask, convexhull, 255)

    face_image_1 = cv2.bitwise_and(image_1, image_1, mask=mask)

    rect = cv2.boundingRect(convexhull)
    subdiv = cv2.Subdiv2D(rect)
    subdiv.insert(landmarks_points)
    triangles = subdiv.getTriangleList()
    triangles = np.array(triangles, dtype=np.int32)

    indexes_triangles = []
    for t in triangles:
        pt1 = (t[0], t[1])
        pt2 = (t[2], t[3])
        pt3 = (t[4], t[5])


        index_pt1 = np.where((points == pt1).all(axis=1))
        index_pt1 = extract_index_nparray(index_pt1)

        index_pt2 = np.where((points == pt2).all(axis=1))
        index_pt2 = extract_index_nparray(index_pt2)

        index_pt3 = np.where((points == pt3).all(axis=1))
        index_pt3 = extract_index_nparray(index_pt3)

        if index_pt1 is not None and index_pt2 is not None and index_pt3 is not None:
            triangle = [index_pt1, index_pt2, index_pt3]
            indexes_triangles.append(triangle)
```

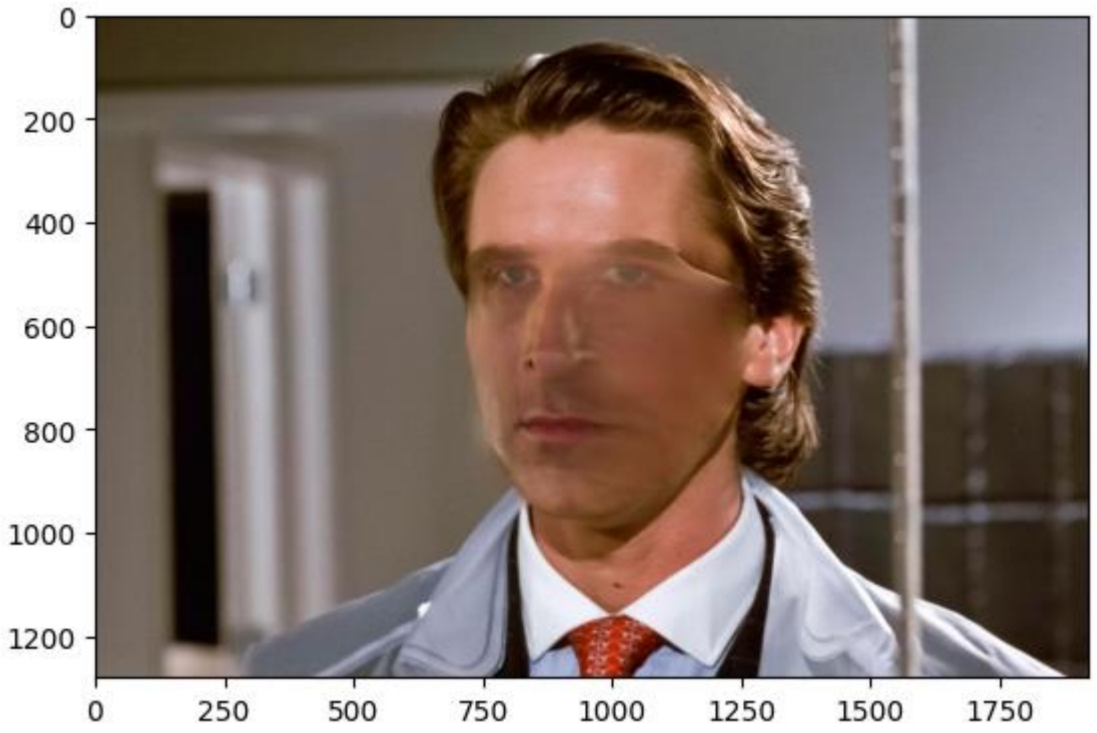
```
faces2 = detector(image_2_gray)
for face in faces2:
    landmarks = predictor(image_2_gray, face)
    landmarks_points2 = []
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y
        landmarks_points2.append((x, y))
    points2 = np.array(landmarks_points2, np.int32)
    convexhull2 = cv2.convexHull(points2)
```

```
lines_space_mask = np.zeros_like(image_1_gray)
lines_space_new_face = np.zeros_like(image_2)



for triangle_index in indexes_triangles:
    tr1_pt1 = landmarks_points[triangle_index[0]]
    tr1_pt2 = landmarks_points[triangle_index[1]]
    tr1_pt3 = landmarks_points[triangle_index[2]]
    triangle1 = np.array([tr1_pt1, tr1_pt2, tr1_pt3], np.int32)



    rect1 = cv2.boundingRect(triangle1)
    (x, y, w, h) = rect1
    cropped_triangle = image_1[y: y + h, x: x + w]
    cropped_tr1_mask = np.zeros((h, w), np.uint8)



    points = np.array([[tr1_pt1[0] - x, tr1_pt1[1] - y],
                       [tr1_pt2[0] - x, tr1_pt2[1] - y],
                       [tr1_pt3[0] - x, tr1_pt3[1] - y]], np.int32)

    cv2.fillConvexPoly(cropped_tr1_mask, points, 255)

    cv2.line(lines_space_mask, tr1_pt1, tr1_pt2, 255)
    cv2.line(lines_space_mask, tr1_pt2, tr1_pt3, 255)
    cv2.line(lines_space_mask, tr1_pt1, tr1_pt3, 255)
    lines_space = cv2.bitwise_and(image_1, image_1, mask=lines_space_mask)

    tr2_pt1 = landmarks_points2[triangle_index[0]]
    tr2_pt2 = landmarks_points2[triangle_index[1]]
    tr2_pt3 = landmarks_points2[triangle_index[2]]
    triangle2 = np.array([tr2_pt1, tr2_pt2, tr2_pt3], np.int32)



    rect2 = cv2.boundingRect(triangle2)
    (x, y, w, h) = rect2
    cropped_tr2_mask = np.zeros((h, w), np.uint8)

    points2 = np.array([[tr2_pt1[0] - x, tr2_pt1[1] - y],
                        [tr2_pt2[0] - x, tr2_pt2[1] - y],
                        [tr2_pt3[0] - x, tr2_pt3[1] - y]], np.int32)

    cv2.fillConvexPoly(cropped_tr2_mask, points2, 255)

    points = np.float32(points)
    points2 = np.float32(points2)
    M = cv2.getAffineTransform(points, points2)
    warped_triangle = cv2.warpAffine(cropped_triangle, M, (w, h))
    warped_triangle = cv2.bitwise_and(warped_triangle, warped_triangle, mask=cropped_tr2_mask)

    img2_new_face_rect_area = image_2_new_face[y: y + h, x: x + w]
    img2_new_face_rect_area_gray = cv2.cvtColor(img2_new_face_rect_area,
cv2.COLOR_BGR2GRAY)
    _, mask_triangles_designed = cv2.threshold(img2_new_face_rect_area_gray, 1, 255,
cv2.THRESH_BINARY_INV)
    warped_triangle = cv2.bitwise_and(warped_triangle, warped_triangle,
mask=mask_triangles_designed)

    img2_new_face_rect_area = cv2.add(img2_new_face_rect_area, warped_triangle)
    image_2_new_face[y: y + h, x: x + w] = img2_new_face_rect_area
```

```
img2_face_mask = np.zeros_like(image_2_gray)
img2_head_mask = cv2.fillConvexPoly(img2_face_mask, convexhull2, 255)
img2_face_mask = cv2.bitwise_not(img2_head_mask)



img2_head_noface = cv2.bitwise_and(image_2, image_2, mask=img2_face_mask)
result = cv2.add(img2_head_noface, image_2_new_face)

(x, y, w, h) = cv2.boundingRect(convexhull2)
center_face2 = (int((x + x + w) / 2), int((y + y + h) / 2))

seamlessclone = cv2.seamlessClone(result, image_2, img2_head_mask, center_face2,
cv2.NORMAL_CLONE
```

Output:





Ex.no: 07 Change the facial expression using VAE

Code:

```python
import matplotlib.pyplot as plt
import os
import glob
import pandas as pd
import random
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data_utils
from copy import deepcopy
from torch.autograd import Variable
from tqdm import tqdm
from pprint import pprint
from PIL import Image
from sklearn.model_selection import train_test_split
import os
import opendatasets as od
import pickle
```

```python
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print('Training on',DEVICE)
```

```python
DATASET_PATH ="/content/lfw-dataset/lfw-deepfunneled/lfw-deepfunneled"
ATTRIBUTES_PATH = "/content/lfw-attributes/lfw_attributes.txt"
```

```python
dataset = []
for path in glob.iglob(os.path.join(DATASET_PATH, "**", "*.jpg")):
    person = path.split("/")[-2]
    dataset.append({"person":person, "path": path})

dataset = pd.DataFrame(dataset)
#too much Bush
dataset = dataset.groupby("person").filter(lambda x: len(x) < 25 )
dataset.head(10)
plt.figure(figsize=(20,10))
for i in range(20):
    idx = random.randint(0, len(dataset))
    img = plt.imread(dataset.path.iloc[idx])
    plt.subplot(4, 5, i+1)
    plt.imshow(img)
    plt.title(dataset.person.iloc[idx])
    plt.xticks([])
    plt.yticks([])
plt.tight_layout()
plt.show()
```

```
def fetch_dataset(dx=80,dy=80, dimx=45,dimy=45):

    df_attrs = pd.read_csv(ATTRIBUTES_PATH, sep='\t', skiprows=1,)
    df_attrs = pd.DataFrame(df_attrs.iloc[:,:-1].values, columns = df_attrs.columns[1:])

    photo_ids = []
    for dirpath, dirnames, filenames in os.walk(DATASET_PATH):
        for fname in filenames:
            if fname.endswith(".jpg"):
                fpath = os.path.join(dirpath,fname)
                photo_id = fname[:-4].replace('_',' ').split()
                person_id = ' '.join(photo_id[:-1])
                photo_number = int(photo_id[-1])
                photo_ids.append({'person':person_id,'imagenum':photo_number,'photo_path':fpath})

    photo_ids = pd.DataFrame(photo_ids)
    df = pd.merge(df_attrs,photo_ids,on=('person','imagenum'))

    assert len(df)==len(df_attrs),"lost some data when merging dataframes"

    all_photos = df['photo_path'].apply(imageio.imread)\
                                .apply(lambda img:img[dy:-dy,dx:-dx])\
                                .apply(lambda img: np.array(Image.fromarray(img).resize([dimx,dimy])) )

    all_photos = np.stack(all_photos.values).astype('uint8')
    all_attrs = df.drop(["photo_path","person","imagenum"],axis=1)

    return all_photos,all_attrs
```

```
data, attrs = fetch_dataset()
```

```
#45,45
IMAGE_H = data.shape[1]
IMAGE_W = data.shape[2]
N_CHANNELS = 3
```

```
data = np.array(data / 255, dtype='float32')
X_train, X_val = train_test_split(data, test_size=0.2, random_state=42)
```

```
X_train = torch.FloatTensor(X_train)
X_val = torch.FloatTensor(X_val)
```

```
class VAE(nn.Module):
   def __init__(self):
      super(VAE, self).__init__()
      self.fc1 = nn.Linear(45*45*3, 1500)
      self.fc21 = nn.Linear(1500, dim_z)
      self.fc22 = nn.Linear(1500, dim_z)
      self.fc3 = nn.Linear(dim_z, 1500)
      self.fc4 = nn.Linear(1500, 45*45*3)
      self.relu = nn.LeakyReLU()

   def encode(self, x):
      x = self.relu(self.fc1(x))
      return self.fc21(x), self.fc22(x)

   def reparameterize(self, mu, logvar):
      std = torch.exp(0.5 *logvar)
      eps = torch.randn_like(std)
      return eps.mul(std).add_(mu)

   def decode(self, z):
      z = self.relu(self.fc3(z)) #1500
      return torch.sigmoid(self.fc4(z))

   def forward(self, x):
      mu, logvar = self.encode(x)
      z = self.reparameterize(mu, logvar)
      z = self.decode(z)
      return z, mu, logvar

def loss_vae_fn(x, recon_x, mu, logvar):
   BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
   KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
   return BCE + KLD
```

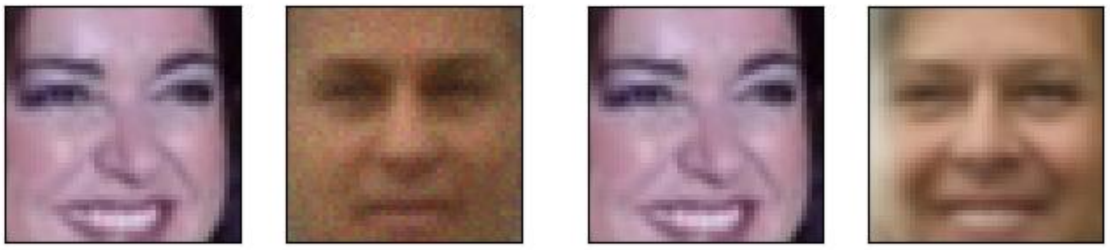
```
def get_batch(data, batch_size=64):
   total_len = data.shape[0]
   for i in range(0, total_len, batch_size):
      yield data[i:min(i+batch_size,total_len)]
```

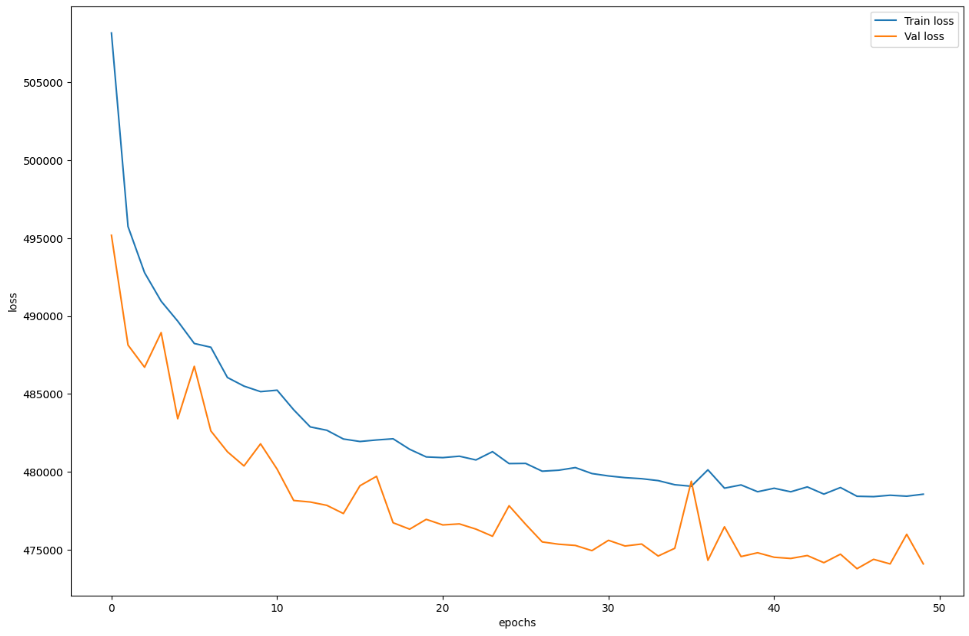
```
def plot_gallery(images, h, w, n_row=3, n_col=6, with_title=False, titles=[]):
    plt.figure(figsize=(1.5 * n_col, 1.7 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        try:
            plt.imshow(images[i].reshape((h, w, 3)), cmap=plt.cm.gray, vmin=-1, vmax=1,
interpolation='nearest')
            if with_title:
                plt.title(titles[i])
            plt.xticks(())
            plt.yticks(())
        except:
            pass
```

```
def fit_epoch_vae(model, train_x, optimizer, batch_size, is_cnn=False):
    running_loss = 0.0
    processed_data = 0

    for inputs in get_batch(train_x,batch_size):
        inputs = inputs.view(-1, 45*45*3)
        inputs = inputs.to(DEVICE)
        optimizer.zero_grad()

        decoded,mu,logvar, = model(inputs)
        outputs = decoded.view(-1, 45*45*3)
        outputs = outputs.to(DEVICE)

        loss = loss_vae_fn(inputs,outputs,mu,logvar)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.shape[0]
        processed_data += inputs.shape[0]

    train_loss = running_loss / processed_data
    return train_loss

def eval_epoch_vae(model, x_val, batch_size):
    running_loss = 0.0
    processed_data = 0
    model.eval()
    for inputs in get_batch(x_val,batch_size=batch_size):
        inputs = inputs.view(-1, 45*45*3)
        inputs = inputs.to(DEVICE)

        with torch.set_grad_enabled(False):
            decoded,mu,logvar = model(inputs)
            outputs = decoded.view(-1, 45*45*3)
            loss = loss_vae_fn(inputs,outputs,mu,logvar)
            running_loss += loss.item() * inputs.shape[0]
            processed_data += inputs.shape[0]

    val_loss = running_loss / processed_data

    #draw
    with torch.set_grad_enabled(False):
        pic = x_val[3]
        pic_input = pic.view(-1, 45*45*3)
        pic_input = pic_input.to(DEVICE)
        decoded,mu,logvar = model(inputs)
        pic_output = decoded[0].view(-1, 45*45*3).squeeze()
        pic_output = pic_output.to("cpu")
        pic_input = pic_input.to("cpu")
        plot_gallery([pic_input, pic_output],45,45,1,2)

    return val_loss

def train_vae(train_x, val_x, model, epochs=10, batch_size=32, lr=0.001):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    history = []
    log_template = "\nEpoch {ep:03d} train_loss: {t_loss:0.4f} val_loss: {val_loss:0.4f}"

    with tqdm(desc="epoch", total=epochs) as pbar_outer:
        for epoch in range(epochs):
            train_loss = fit_epoch_vae(model,train_x,optimizer,batch_size)
            val_loss = eval_epoch_vae(model,val_x,batch_size)
            print("loss: ", train_loss)

            history.append((train_loss,val_loss))

            pbar_outer.update(1)
            tqdm.write(log_template.format(ep=epoch+1, t_loss=train_loss, val_loss=val_loss))
    return history
```

```
history_vae = train_vae(X_train, X_val, model_vae, epochs=50, batch_size=128, lr=0.001)
```

Output:





Ex.no: 08 **Image based Stable - Diffusion models**

Code:

```
%%capture
!pip install diffusers["torch"] transformers ftfy accelerate?
```

```
import torch
import requests
from PIL import Image
from io import BytesIO
from diffusers import StableDiffusionImg2ImgPipeline
```

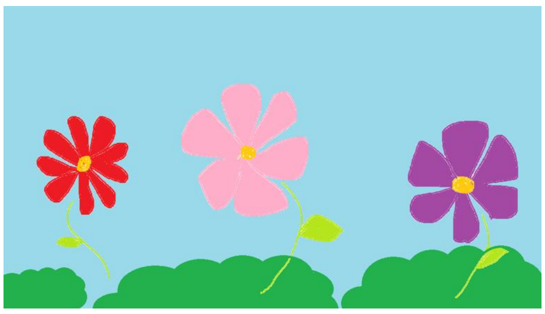
```
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print("You are on",DEVICE)
```

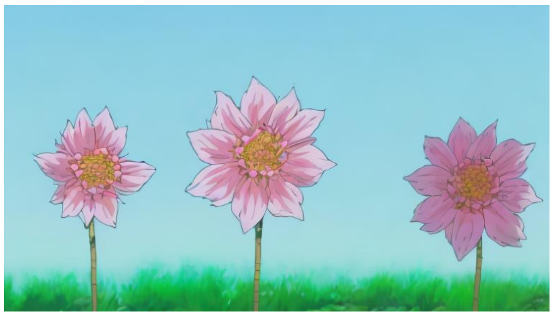
```
device = "cuda"
pipe = StableDiffusionImg2ImgPipeline.from_pretrained("nitrosocke/Ghibli-Diffusion",
torch_dtype=torch.float16, use_safetensors=True).to(device)
```

```
file_path = "flower.png"
init_image = Image.open(file_path).convert("RGB")
init_image.thumbnail((768, 768))
init_image
```

```
prompt = "ghibli style, a new variety of flower"
generator = torch.Generator(device=device).manual_seed(1024)
image = pipe(prompt=prompt, image=init_image, strength=0.75, guidance_scale=7.5,
generator=generator).images[0]
image
```

Output:





Ex.no: 09 **Model scope convert a given text into video generation**

Code:

```
!pip install diffusers transformers accelerate
```

```
import torch
from diffusers import DiffusionPipeline, DPMSolverMultistepScheduler
from diffusers.utils import export_to_video
```

```
pipe = DiffusionPipeline.from_pretrained("damo-vilab/text-to-video-ms-
1.7b",torch_dtype=torch.float16,variant="fp16")
```

```
pipe.scheduler=DPMSolverMultistepScheduler.from_config(pipe.scheduler.config)
pipe.enable_model_cpu_offload()
```

```
prompt="Monkey riding a horse in the sea"
video_frames=pipe(prompt,num_inference_steps=25).frames
video_path=export_to_video(video_frames)
video_name=video_path.replace('\tmp','')
print('Name:',video_name)
torch.cuda.empty_cache()
```

Output:

Ex.no: 10 **Langchain - Proverb for a given input query**

Code:

```
from langchain.document_loaders import PyPDFLoader, OnlinePDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import Pinecone
from sentence_transformers import SentenceTransformer
from langchain.chains.question_answering import load_qa_chain
import pinecone
```

```
loader = PyPDFLoader("cti-guide.pdf")
data = loader.load()
```

```
text_splitter=RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=0)
docs=text_splitter.split_documents(data)
```

```
os.environ["HUGGINGFACEHUB_API_TOKEN"] =
"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"
PINECONE_API_KEY = os.environ.get('PINECONE_API_KEY', '63e35379-ae89-437b-8e78-
9795263ff99c')
PINECONE_API_ENV = os.environ.get('PINECONE_API_ENV', 'gcp-starter')
embeddings=HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2')
```

```
# initialize pinecone
pinecone.init(
    api_key=PINECONE_API_KEY,  # find at app.pinecone.io
    environment=PINECONE_API_ENV  # next to api key in console
)
index_name = "studentdb" # put in the name of your pinecone index here
# docsearch=Pinecone.from_texts([t.page_content for t in docs], embeddings,
index_name=index_name)
docsearch = Pinecone.from_existing_index(index_name, embeddings)
```

```
from langchain.llms import LlamaCpp
from langchain.callbacks.manager import CallbackManager
from huggingface_hub import hf_hub_download
from langchain.chains.question_answering import load_qa_chain
```

```
from langchain.llms import HuggingFaceHub
# https://github.com/EleutherAI/gpt-neox
llm=HuggingFaceHub(repo_id="google/flan-t5-xxl", model_kwargs={"temperature":0.5,
"max_length":512})
chain=load_qa_chain(llm, chain_type="stuff")
```

```
query="Proverb related to mercy"
docs=docsearch.similarity_search(query)
chain.run(input_documents=docs, question=query)
```

Output:

```
'The Lord is merciful and gracious, slow to anger and of great kindness.'
```

Ex.no: 11 **Langchain - department class**

Code:

```
!pip install langchain
```

```
!pip install -U sentence-transformers
```

```
!pip install pinecone-client
```

```
pip install ctransformers
```

```
from langchain.document_loaders.csv_loader import CSVLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.llms import CTransformers
from langchain.memory import ConversationBufferMemory
```

```python
from langchain.chains import ConversationalRetrievalChain
from langchain.vectorstores import Pinecone
from langchain.chains.question_answering import load_qa_chain
import pinecone
import sys
import os
```

```python
loader = CSVLoader(file_path="/content/Student DB for Placement.csv", encoding="utf-8",
csv_args={'delimiter': ','})
data = loader.load()
print(data[0])
```

```python
# Split the text into Chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=20)

text_chunks = text_splitter.split_documents(data)
```

```python
print(len(text_chunks))
```

```python
os.environ["HUGGINGFACEHUB_API_TOKEN"] =
"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"
PINECONE_API_KEY = os.environ.get('PINECONE_API_KEY', 'a7d483e9-4ac0-4fee-96d2-
73c1a2c02caa')
PINECONE_API_ENV = os.environ.get('PINECONE_API_ENV', 'us-west4-gcp-free')
```

```python
embeddings = HuggingFaceEmbeddings(model_name = 'sentence-transformers/all-MiniLM-L6-v2')
```

```python
# initialize pinecone
pinecone.init(
    api_key=PINECONE_API_KEY,  # find at app.pinecone.io
    environment=PINECONE_API_ENV  # next to api key in console
)
index_name = "student" # put in the name of your pinecone index here
```

```python
#docsearch=Pinecone.from_texts([t.page_content for t in text_chunks], embeddings,
index_name=index_name)

docsearch = Pinecone.from_existing_index(index_name, embeddings)
```

```python
query="What is the phone number of the student NithishKumaar K P?"
docs=docsearch.similarity_search(query)
```

```python
docs
```

```
from langchain.llms import HuggingFaceHub
```

```
llm=HuggingFaceHub(repo_id="google/flan-t5-xxl", model_kwargs={"temperature":0.5,
"max_length":512})
chain=load_qa_chain(llm, chain_type="stuff")
```

```
query="What is the phone number of the student NithishKumaar K P?"
docs=docsearch.similarity_search(query)
```

```
chain.run(input_documents=docs, question=query)
```

Output:

```
'9486263726'
```

Ex.no: 12 MuseGAN - generate new Voice/Video

Code:

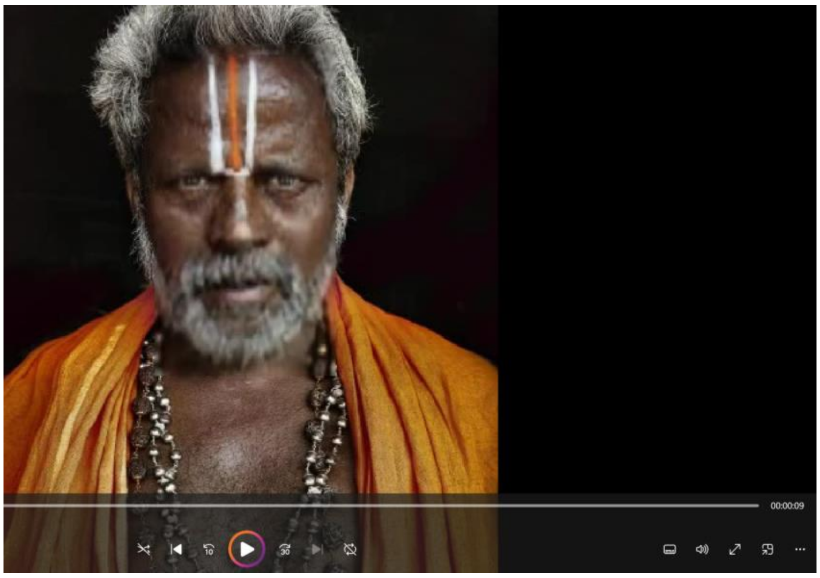
```
!wget https://huggingface.co/spaces/camenduru/one-shot-talking-
face/resolve/main/examples/audio.wav -O /content/audio.wav
```

```
import cv2
from google.colab.patches import cv2_imshow

image = cv2.imread("/content/suriya1.jpg")
cv2_imshow(image)
```

```
!python inference.py --checkpoint_path checkpoints/wav2lip_gan.pth --face "/content/suriya1.jpg" --
audio "/content/audio.wav"
```

```
from IPython.display import HTML
from base64 import b64encode
mp4 = open('/content/Wav2Lip/results/result_voice.mp4','rb').read()
data_url = "data:video/mp4;base64," + b64encode(mp4).decode()
HTML(f"""
<video width="50%" height="50%" controls>
    <source src="{data_url}" type="video/mp4">
</video>""")
```

Output:


00:00:09