ASSIGNMENT -1

1.Create a GAN to generate original artwork or visual designs. The network should learn from a dataset of existing artworks to produce new, unique pieces that mimic the style of the training data.

Aim:

The goal of the provided code is to implement and train a Generative Adversarial Network (GAN) using the CIFAR-10 dataset. The aim is for the Generator to produce images that are so realistic that the Discriminator cannot distinguish them from real images.

Algorithm:

1. Data Preparation:
    - The CIFAR-10 dataset is loaded and preprocessed with transformations such as resizing to 32x32 pixels, normalization, and conversion to tensor format.
    - A DataLoader is used to batch the data for training.
2. Model Definition:
    - The Generator takes a random noise vector (z_dim = 100) and generates an image of shape (3, 32, 32) using several fully connected layers followed by non-linear activations (ReLU and Tanh).
    - The Discriminator takes an image (either real or fake) and classifies it as real (1) or fake (0) using several fully connected layers with Leaky ReLU activations and a final Sigmoid activation.

3.Training Process:

- For each epoch, both the Generator and the Discriminator are trained iteratively.
- Discriminator Training:
    - Forward pass with real images and compute loss (real_loss).
    - Generate fake images using the Generator and compute loss (fake_loss).
    - The total loss (d_loss) is computed as the sum of real_loss and fake_loss. The Discriminator's parameters are updated by backpropagation.
- Generator Training:
    - Generate fake images and compute the loss (g_loss) using the output from the Discriminator.
    - The Generator's parameters are updated to minimize g_loss.

4.Loss Calculation and Optimization:

- The Binary Cross-Entropy loss function (nn.BCELoss) is used to compute the loss.
- Adam optimizer is used for updating the parameters of both networks.

5.Output:

- For each epoch, the loss values for both the Discriminator (D Loss) and the Generator (G Loss) are printed.

Code:

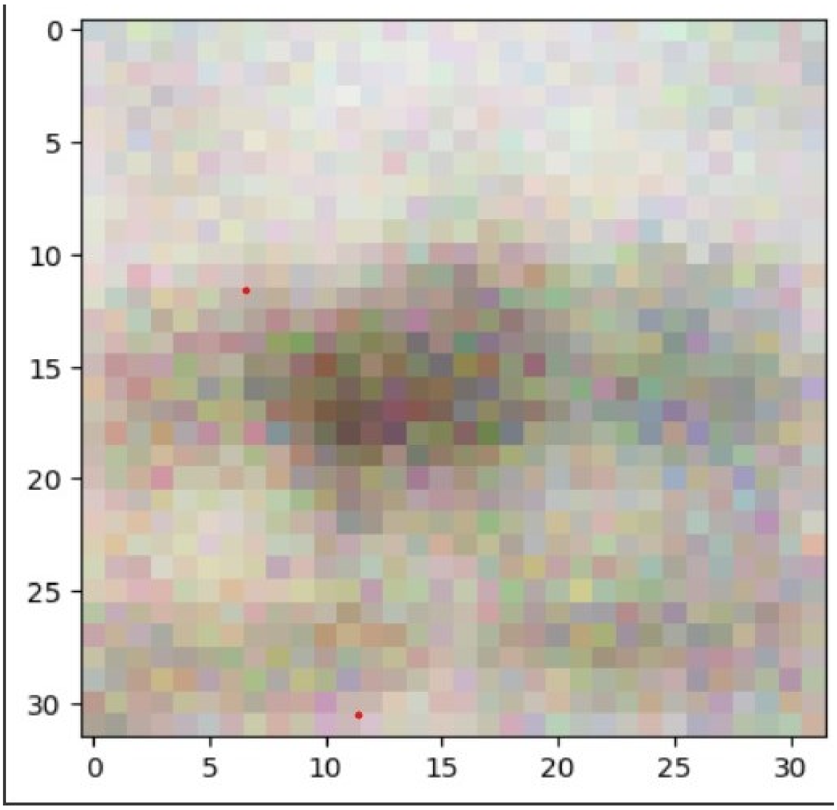
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import torchvision.transforms as transformsdevice = torch.
device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])
dataset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
class Generator(nn.Module):
    def _init_(self, z_dim):
        super(Generator, self)._init_()
        self.model = nn.Sequential(
            nn.Linear(z_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
```

```
            nn.ReLU(),
            nn.Linear(512, 3*32*32),
            nn.Tanh()
    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 3, 32, 32)
        return img
class Discriminator(nn.Module):
    def _init_(self):
        super(Discriminator, self)._init_()
        self.model = nn.Sequential(
            nn.Linear(3*32*32, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()  )
    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return
z_dim = 100
lr = 0.0002
epochs = 100  # Set to 100 for quicker training/testing
batch_size = 64
generator = Generator(z_dim).to(device)
```

```
discriminator = Discriminator().to(device)
optimizer_G = optim.Adam(generator.parameters(), lr=lr)
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr)
criterion = nn.BCELoss()
 range(epochs):
    for real_imgs, _ in dataloader:
        real_imgs = real_imgs.to(device)
        real_labels = torch.ones(real_imgs.size(0), 1).to(device)
        fake_labels = torch.zeros(real_imgs.size(0), 1).to(device)
        optimizer_D.zero_grad()
        real_imgs = real_imgs.view(real_imgs.size(0), -1)
        real_loss = criterion(discriminator(real_imgs), real_labels)
        z = torch.randn(real_imgs.size(0), z_dim).to(device)
        fake_imgs = generator(z)
        fake_loss = criterion(discriminator(fake_imgs.detach()), fake_labels)
        d_loss = real_loss + fake_loss
        d_loss.backward()
        optimizer_D.step()
        optimizer_G.zero_grad()
        g_loss = criterion(discriminator(fake_imgs), real_labels)
        g_loss.backward()
        optimizer_G.step()
 print(f"Epoch {epoch}/{epochs} | D Loss: {d_loss.item()} | G Loss: 
{g_loss.item()}")z_dim = 100
lr = 0.0002
epochs = 100  # Set to 100 for quicker training/testing
batch_size = 64
```

```
generator = Generator(z_dim).to(device)
discriminator = Discriminator().to(device)
optimizer_G = optim.Adam(generator.parameters(), lr=lr)
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr)
criterion = nn.BCELoss()
 range(epochs):
    for real_imgs, _ in dataloader:
        real_imgs = real_imgs.to(device)
        real_labels = torch.ones(real_imgs.size(0), 1).to(device)
        fake_labels = torch.zeros(real_imgs.size(0), 1).to(device)
        optimizer_D.zero_grad()
        real_imgs = real_imgs.view(real_imgs.size(0), -1)
        real_loss = criterion(discriminator(real_imgs), real_labels)
        z = torch.randn(real_imgs.size(0), z_dim).to(device)
        fake_imgs = generator(z)
        fake_loss = criterion(discriminator(fake_imgs.detach()), fake_labels)
        d_loss = real_loss + fake_loss
        d_loss.backward()
        optimizer_D.step()
        optimizer_G.zero_grad()
        g_loss = criterion(discriminator(fake_imgs), real_labels)
        g_loss.backward()
        optimizer_G.step()

    print(f"Epoch {epoch}/{epochs} | D Loss: {d_loss.item()} | G Loss:
{g_loss.item()}")
```

Output:



Result:

The output of the GAN training process includes the loss values of both the Discriminator and the Generator at each epoch. Ideally, over time, the Discriminator's loss stabilizes around 0.5, indicating it is equally likely to classify an image as real or fake, and the Generator's loss decreases, indicating it is generating more realistic images.

2.Develop an autoencoder to transfer artistic styles to images. The model should separate and recombine content and style from different images to produce stylized versions of the original images.

Aim:

The aim is to build an autoencoder that can transfer artistic styles between images by:

1. Separating the content (structural elements) and style (textures, colors, patterns) from different images.
2. Recombining the content of one image with the style of another to create a stylized version.

Algorithm:

1. Input Images: Take two images, one representing the content (original image) and another representing the style (artistic style image).
2. Feature Extraction: Use a convolutional neural network (CNN) as an encoder to extract high-level content features from the content image and style features from the style image.
3. Style Representation: Calculate the Gram matrix of the feature maps from the style image to represent its style.
4. Content Representation: Use the encoded output from the content image to represent the content.
5. Decoder: Design a decoder that takes the content representation and applies the style representation to reconstruct the stylized image.
6. Loss Function: Define a loss function that consists of both content loss (difference between the content of the generated image and the original

content image) and style loss (difference between the style of the generated image and the style image).

7. Training: Train the autoencoder to minimize the combined content and style loss using backpropagation.

Code:

```
import tensorflow as tf

from tensorflow.keras.applications import VGG19

from tensorflow.keras.models import Model

from tensorflow.keras import backend as K

import numpy as np

import matplotlib.pyplot as plt

# Load and preprocess images

def load_and_preprocess_image(path, target_size=(224, 224)):

    img = tf.keras.preprocessing.image.load_img(path, target_size=target_size)

    img = tf.keras.preprocessing.image.img_to_array(img)

    img = np.expand_dims(img, axis=0)

    img = tf.keras.applications.vgg19.preprocess_input(img)

    return tf.convert_to_tensor(img)

content_image_path = 'content_image.jpg'  # Path to content image

style_image_path = 'style_image.jpg'  # Path to style image

content_image = load_and_preprocess_image(content_image_path)

style_image = load_and_preprocess_image(style_image_path)
```

```
# Load VGG19 model
vgg = VGG19(include_top=False, weights='imagenet')
vgg.trainable = False
# Define content and style layers
content_layers = ['block5_conv2']
style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1',
'block5_conv1']
num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
def get_feature_representations(model, content_path, style_path):
    """Helper function to compute content and style feature representations."""
    content_image = load_and_preprocess_image(content_path)
    style_image = load_and_preprocess_image(style_path)
    content_outputs = model(content_image)
    style_outputs = model(style_image)
    content_features = [content_layer[0] for content_layer in
content_outputs[:num_content_layers]]
    style_features = [style_layer[0] for style_layer in
style_outputs[num_content_layers:]]
    return content_features, style_features
def gram_matrix(input_tensor):
    """Calculate Gram matrix to represent style features."""
```

```
    channels = int(input_tensor.shape[-1])
    a = K.reshape(input_tensor, (-1, channels))
    n = K.shape(a)[0]
    gram = K.dot(a, a) / K.cast(n, tf.float32)
    return gram
# Build the model to extract style and content features
outputs = [vgg.get_layer(name).output for name in content_layers +
style_layers]
model = Model([vgg.input], outputs)
model.trainable = False
# Define loss functions
def compute_loss(model, loss_weights, init_image, gram_style_features,
content_features):
    """Compute total loss for the stylized image."""
    style_weight, content_weight = loss_weights
    model_outputs = model(init_image)
    # Compute content loss
    content_output_features = model_outputs[:num_content_layers]
    style_output_features = model_outputs[num_content_layers:]
    content_loss = tf.add_n([tf.reduce_mean((content_features[i] -
content_output_features[i])**2)
      for i in range(num_content_layers)])
```

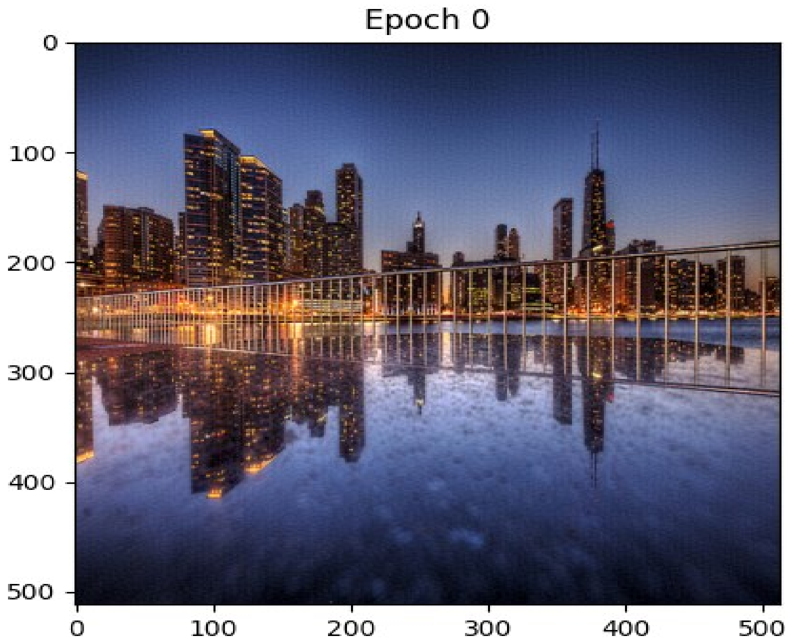
```
    content_loss *= content_weight / num_content_layers

    # Compute style loss

    style_loss = 0

    for target_style, comb_style in zip(gram_style_features,
style_output_features):

        style_loss += tf.reduce_mean((target_style -
gram_matrix(comb_style[0]))**2)

    style_loss *= style_weight / num_style_layers

    total_loss = content_loss + style_loss

    return total_loss

# Optimization settings

optimizer = tf.optimizers.Adam(learning_rate=5.0)

epochs = 1000

loss_weights = (1e-3, 1e4)

content_features, style_features = get_feature_representations(model,
content_image_path, style_image_path)

gram_style_features = [gram_matrix(style_feature) for style_feature in
style_features]

init_image = tf.Variable(content_image, dtype=tf.float32)

# Train the model

for i in range(epochs):

    with tf.GradientTape() as tape:
```

```
        loss = compute_loss(model, loss_weights, init_image,
gram_style_features, content_features)
    grads = tape.gradient(loss, init_image)
    optimizer.apply_gradients([(grads, init_image)])
    if i % 100 == 0:
        print(f"Iteration {i}: loss={loss.numpy()}")
        plt.imshow(tf.keras.preprocessing.image.array_to_img(init_image[0]))
        plt.show()
```

Output:

Content Image:

Style Image:



Result:

After training, the model will produce a stylized version of the content image that combines the structure of the content image with the style of the style image.