A
REPORT
ON

Image-to-image Translation using a VAE-GAN hybrid model

By

M Shruti	**2021A7PS2011P**
Arush Dayal	**2021A7PS0011P**

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (Rajasthan)

(April, 2024)

Introduction

The main objective of the project involved creating a VAE-GAN hybrid model for Image-to-Image Translation task, specifically to convert an impressionist image to a cubist image and vice versa. We have used the papers Style Transfer with GAN and Combining Variational Autoencoders & Generative Adversarial Networks to Improve Image Quality.

A Brief Summary of the paper Style Transfer with GAN

The deep learning revolution has enabled the development of intelligent objects due to the increasing complexity of learning models. One of the most fascinating models is generative adversarial network (GAN), which generates new data that closely resembles the examples used during training. GANs have demonstrated their potential in image-to-image translation and style transfer, which are used for creating works of art. The paper focuses on using GAN concepts for computer vision, specifically defogging, which removes fog from images, restoring them as if they were taken during optimal weather conditions. The paper also adopts an unpaired approach to defogging, attempting to transfer a foggy image to the clear picture without having pairs of foggy and ground truth haze-free images during training. From this paper, we were able to extract crucial information on how to implement our hybrid model using the Pix2Pix model for our image-to-image translation task.

A Brief Summary of the Paper Combining Variational Autoencoders & Generative Adversarial Networks to Improve Image Quality

The paper presents the use of a VAE (Variational AutoEncoder), trained on a combination of training and generated data. The VAE then maps these generated images to better versions of it, a similar process to denoising with slight variations. The use of this additional VAE component combats the issues of mode collapse, lack of proper training data, etc. This is shown to work better than the usual WGANs. From this paper, we were able to learn more about the algorithm for training a VAE-GAN Hybrid model and avoiding mode collapse.

Key Approach and Design Choices

Our end-to-end implementation involves the use of the Pix2Pix model. The Pix2Pix model is a conditional adversarial network that was proposed as a solution for image-to-image translation tasks. These networks learn a mapping from an input image to an output image along with learning the loss function to train this mapping. However, there is a key difference between our model and a vanilla Pix2Pix model - the Pix2Pix

architecture involves a generator that acts as an autoencoder with skip connections, however, we have employed a Variational AutoEncoder (VAE) which learns the latent distribution from which the generator samples.

Generative Adversarial Networks (GAN)

A Generative Adversarial Network (GAN), designed by Ian Goodfellow, is a deep learning architecture that trains two neural networks to compete against each other to generate samples that are not blurry and that have rich representations of features. A GAN includes a Discriminator component, which is simply a classifier that classifies whether the data that comes from the output of the Generator is real or fake. The Generator is that component of the GAN that takes noise which is sampled from a Gaussian distribution and generates images similar to the original dataset. The training loop of the GAN involves generating samples from the generator and training the discriminator to distinguish between the real data and the fake data that the generator produces. The generator learns the representations that would produce images that are as close to the original in order to fool the generator.

Pix2Pix

Pix2Pix is a model within the realm of conditional generative adversarial networks (cGANs), specifically designed for image-to-image translation tasks. It too operates on an architecture comprising a generator and a discriminator, however, the major difference in its implementation is the use of a U-Net architecture as well as a PatchGAN. The generator takes an input image from one domain, and generates corresponding output images in another domain. Concurrently, the discriminator evaluates the realism of the generated images by differentiating between them and real images from the target domain. Training Pix2Pix involves optimizing a combination of adversarial loss, which compels the generator to create indistinguishable images from real ones, and L1 loss, measuring pixel-wise differences between generated and target images to ensure both realism and content similarity. As a conditional GAN, Pix2Pix conditions both the generator and discriminator on input images, enabling precise image translation based on input context.

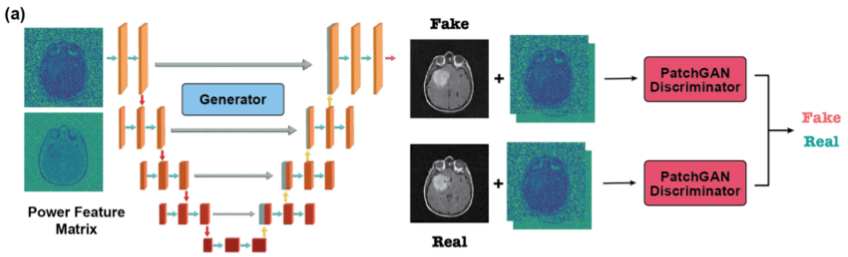


Fig 1: Pix2Pix Architecture

We found that the Pix2Pix model was extremely efficient in performing image-to-image translation tasks. Therefore, beginning with the implementation of the Pix2Pix, we added the functionalities of a VAE (Variational AutoEncoder) by making the encoder compress the real image into a distribution in a latent space (instead of a single point, to avoid overfitting of the data) and adding the reparameterization trick (which allows a gradient path through a non-stochastic node).

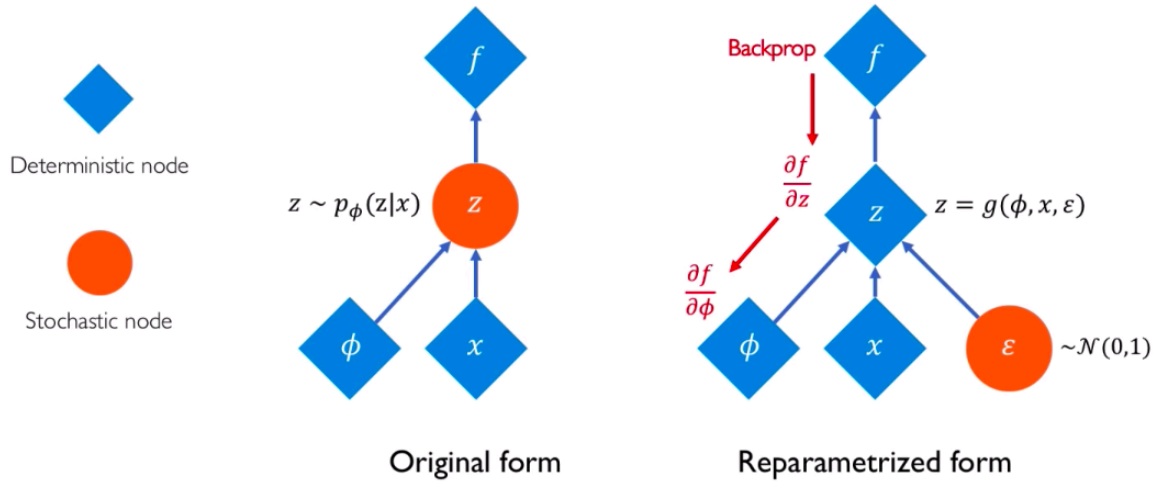


Fig 2: The reparametrization trick in VAE

Key Steps:

The end-to-end implementation of the VAE-GAN hybrid model requires the following key steps:

1. **Dataset Preparation (Data Loading and Augmentation)**

- The dataset used contains pairs of images: one in the Impressionist style and one in the Cubist style as mentioned in the project description. Due to the lack of availability of such a dataset, we had to create the same by passing an image into a Stable Diffusion model and converting it to both the Impressionist and the Cubist style. This was through an online model : https://dezgo.com/image2image
- These base images were taken from multiple subsets of the PeoplesArt dataset, specifically Magic realism, Symbolism, Classicism, Art nouveau, Realism. These pairs of images are loaded using the PIL Library and augmented using various transformations such as random cropping, flipping, affine transformations, brightness/contrast/saturation adjustments and Gaussian/Unsharp mask filtering.
- Further, the num_augmentations variable allows us to set the number of different and random augmentations we want to subject each pair in the dataset to. The function applies the same augmentation on both the input and the output images.

- We have created a dataset with 120 pairs of images, with 80 pairs being the training dataset and the 40 pairs being the validation dataset.

```
'''Loading of the training dataset'''

def load_image(image_path):
    return Image.open(image_path).convert("RGB")

def load_and_augment_dataset(dataset_path, num_augmentations=4): #num_augmentations can be changed depending on task
    augmented_pairs = []

    pair_folders = os.listdir(dataset_path)
    pair_folders.sort()

    for folder in pair_folders:
        folder_path = os.path.join(dataset_path, folder)

        if not os.path.isdir(folder_path):
            continue

        image_files = os.listdir(folder_path)
        image_files.sort()

        image_files = [f for f in image_files if f.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif'))]

        if len(image_files) == 2:
            output_image = load_image(os.path.join(folder_path, image_files[0]))
            input_image = load_image(os.path.join(folder_path, image_files[1]))

            input_image = input_image.resize(output_image.size, resample=Image.Resampling.BILINEAR)

            # Convert input and output images to numpy arrays
            input_image = np.array(input_image, dtype=np.double) / 255.0  # Normalize to range [0, 1]
            output_image = np.array(output_image, dtype=np.float64) / 255.0

            for _ in range(num_augmentations):
                input_image_augmented, output_image_augmented = transform_images(input_image, output_image)

                augmented_pairs.append((input_image_augmented, output_image_augmented))

    return augmented_pairs
```

```
''' Augmentation of the dataset'''

def transform_images(input_image, output_image):

    # Convert numpy arrays to PIL images
    input_image = TF.to_pil_image(input_image)
    output_image = TF.to_pil_image(output_image)

    resize = transforms.Resize(size=(256, 256))
    input_image = resize(input_image)
    output_image = resize(output_image)

    #Randomly crops the pair of images
    i, j, h, w = transforms.RandomCrop.get_params(input_image, output_size=(256, 256))
    input_image = TF.crop(input_image, i, j, h, w)
    output_image = TF.crop(output_image, i, j, h, w)

    #Randomly horizontally flips the pair of images
    if random.random() > 0.5:
        input_image = TF.hflip(input_image)
        output_image = TF.hflip(output_image)

    #Randomly vertically flips the pair of images
    if random.random() > 0.5:
        input_image = TF.vflip(input_image)
        output_image = TF.vflip(output_image)

    #Randomly tils the pair of images
    affine_param = transforms.RandomAffine.get_params(
        degrees = [-180, 180], translate = [0.3,0.3],
        img_size = [520, 520], scale_ranges = [1, 1.3],
        shears = [2,2])
    input_image = TF.affine(input_image,
                            affine_param[0], affine_param[1],
                            affine_param[2], affine_param[3])
    output_image = TF.affine(output_image,
                            affine_param[0], affine_param[1],
                            affine_param[2], affine_param[3])
```

```
#Randomly changes the brightness of the pair of images
brightness_factor = random.uniform(0.8, 1.2)
input_image = ImageEnhance.Brightness(input_image).enhance(brightness_factor)
output_image = ImageEnhance.Brightness(output_image).enhance(brightness_factor)

#Randomly changes the contrast of the pair of images
contrast_factor = random.uniform(0.8, 1.2)
input_image = ImageEnhance.Contrast(input_image).enhance(contrast_factor)
output_image = ImageEnhance.Contrast(output_image).enhance(contrast_factor)

#Randomly changes the saturation of the pair of images
saturation_factor = random.uniform(0.8, 1.2)
input_image = ImageEnhance.Color(input_image).enhance(saturation_factor)
output_image = ImageEnhance.Color(output_image).enhance(saturation_factor)

#Randomly changes the hue of the pair of images
hue_factor = random.uniform(0.8, 1.2)
input_image = ImageEnhance.Sharpness(input_image).enhance(hue_factor)
output_image = ImageEnhance.Sharpness(output_image).enhance(hue_factor)

input_image = np.array(input_image)
output_image = np.array(output_image)

#Randomly adds gaussian blur to the pair of images
if random.random() < 0.25:
    sigma_param = random.uniform(0.1, 0.5)
    input_image = gaussian(input_image, sigma=sigma_param)
    output_image = gaussian(output_image, sigma=sigma_param)

#Randomly adds unsharp mask to the pair of images
if random.random() < 0.25:
    radius_param = random.uniform(0, 2)
    amount_param = random.uniform(0.2, 1)
    input_image = unsharp_mask(input_image, radius=radius_param, amount=amount_param)
    output_image = unsharp_mask(output_image, radius=radius_param, amount=amount_param)

input_image = TF.to_tensor(input_image)
output_image = TF.to_tensor(output_image)

return input_image, output_image
```

2. Model Architecture:

The model architecture involves the use of the Pix2Pix model for Image-to-Image translation. However, the encoder and discriminator blocks along with the training loop are modified in order to combine the Variational AutoEncoder in the implementation. An overview of the model architecture is given below:

Encoder Architecture:

- According to literature on image style transfer, an IAE architecture is optimal to learn latent vectors of the input image and then learn its reconstruction. An IAE uses VGG-16 or 19 architecture for the encoder. Similarly, we have also employed a VGG-like architecture for our encoder.
- A VGG-16 model involves 16 convolutional layers and multiple maxpool layers to reduce dimensions into a latent vector. Due to computational restrictions, we were unable to employ 16 layers. Our encoder consists of 3 blocks. Each block consists of a convolutional layer, which aids in the learning of important features of the image, a batch normalization layer and a maxpooling layer of stride 2 and kernel size 2 by 2 which halves the dimensions of the image.
- The reduced output image is flattened and then passed through a fully connected layer to reduce it to the latent distribution dimensions, thus preparing the vector for mapping to the latent space.

- The vector is sliced into 2 in order to get the mu and log_sigma vectors which are to be used for the reparameterization trick.
- The implementation of this encoder block becomes VAE-like due to its use of the reparameterization trick. After the fully connected layer, the encoder applies this trick to sample latent vectors.
- The sampled latent vector (‘z’) is computed using this trick.
- The final output of the encoder is this sampled vector reshaped to match the expected dimensions as well as the mean and log variance from the reparameterization trick.

```
class Encoder(nn.Module):
    def __init__(self, z_dim=50):
        super(Encoder, self).__init__()
        self.z_dim = z_dim

        #Block 1
        self.conv1_1 = nn.Conv2d(3, 32, 5, stride=1, bias=False)
        self.bn1_1 = nn.BatchNorm2d(32)
        self.conv1_2 = nn.Conv2d(32, 64, 4, stride=1, bias=False)
        self.bn1_2 = nn.BatchNorm2d(64)

        self.maxPool_1 = nn.MaxPool2d(kernel_size=2, stride=2)

        #Block 2
        self.conv2_1 = nn.Conv2d(64, 128, 4, stride=1, bias=False)
        self.bn2_1 = nn.BatchNorm2d(128)
        self.conv2_2 = nn.Conv2d(128, 256, 4, stride=1, bias=False)
        self.bn2_2 = nn.BatchNorm2d(256)

        self.maxPool_2 = nn.MaxPool2d(kernel_size=2, stride=2)

        #Block 3
        self.conv3_1 = nn.Conv2d(256, 512, 4, stride=1, bias=False)
        self.bn3_1 = nn.BatchNorm2d(512)
        self.conv3_2 = nn.Conv2d(512, 512, 1, stride=1, padding='same', bias=True, padding_mode='reflect')
        self.bn3_2 = nn.BatchNorm2d(512)

        self.maxPool_3 = nn.MaxPool2d(kernel_size=2, stride=2)
        #Dropout added because encoder becoming computationally expensive
        self.drop = nn.Dropout2d(p = 0.4)
        #Fully Connected layer.
        self.fc = nn.Linear(240, self.z_dim*2)
```

```
def forward(self, x):

    print("Entering encoder")
    x1_1 = F.leaky_relu(self.bn1_1(self.conv1_1(x)), negative_slope=0.1)
    x1_2 = F.leaky_relu(self.bn1_2(self.conv1_2(x1_1)), negative_slope=0.1)
    x1_3 = self.drop(x1_2)
    x1 = self.maxPool_1(x1_3)
    print("Block 1 complete")

    x2_1 = F.leaky_relu(self.bn2_1(self.conv2_1(x1)), negative_slope=0.1)
    x2_2 = F.leaky_relu(self.bn2_2(self.conv2_2(x2_1)), negative_slope=0.1)
    x2_3 = self.drop(x2_2)
    x2 = self.maxPool_2(x2_3)
    print("Block 2 complete")

    x3_1 = F.leaky_relu(self.bn3_1(self.conv3_1(x2)), negative_slope=0.1)
    x3_2 = F.leaky_relu(self.bn3_2(self.conv3_2(x3_1)), negative_slope=0.1)
    x3_3 = self.drop(x3_2)
    x3 = self.maxPool_3(x3_3)
    print("Block 3 complete")

    '''Flattening and slicing x to be passed into linear fc layer'''
    x_flat = torch.flatten(x3, start_dim=1)
    x_slice = x_flat[:, :240]

    x_fc = self.fc(x_slice)
    z = self.reparameterize(x_fc)
    return z

'''Applying the reparameterization trick to sample latent vector'''
def reparameterize(self, z):
    mu, log_sigma = z[:, :self.z_dim], z[:, self.z_dim:]
    std = torch.exp(log_sigma)
    eps = torch.randn_like(std)
    z_out = mu + eps * std
    return z_out, mu, log_sigma
```

Generator Architecture:

- Our ‘Generator’ class takes the latent vector (‘z’) as input and generates an image.
- It does so by upsampling and transforming the latent vector through a series of transposed convolutional layers. These layers are followed by batch normalization layers.
- The generator passes the latent vector through the series of transposed convolutional layers, performing upsampling and learning to generate features.
- Batch normalization layers, as used in the Encoder class, help stabilize and speed up training. The Leaky ReLU activation functions introduce non-linearity, helping prevent the vanishing gradient problem.
- The generator outputs the generated image after the last convolutional layer, matching the desired output size.

```
'''Generator Class'''

class Generator(nn.Module):
    def __init__(self, z_dim=50):
        super(Generator, self).__init__()
        self.z_dim = z_dim

        self.output_bias = nn.Parameter(torch.zeros(3, 32, 32), requires_grad=True)
        self.deconv1 = nn.ConvTranspose2d(z_dim, 32, 4, stride=1, bias=False)
        self.bn1 = nn.BatchNorm2d(32)
        self.deconv2 = nn.ConvTranspose2d(32, 16, 4, stride=4, padding = 0, bias=False)
        self.bn2 = nn.BatchNorm2d(16)
        self.deconv3 = nn.ConvTranspose2d(16, 8, 4, stride=4,padding = 0, bias=False)
        self.bn3 = nn.BatchNorm2d(8)
        self.deconv4 = nn.ConvTranspose2d(8, 3, 4, stride=4, padding=0, bias=False)
        self.bn4 = nn.BatchNorm2d(3)

    def forward(self, z):
        print("Entering generator")
        z_1 = F.leaky_relu(self.bn1(self.deconv1(z)), negative_slope=0.1)
        z_2 = F.leaky_relu(self.bn2(self.deconv2(z_1)), negative_slope=0.1)
        z_3 = F.leaky_relu(self.bn3(self.deconv3(z_2)), negative_slope=0.1)
        z_4 = F.leaky_relu(self.bn4(self.deconv4(z_3)), negative_slope=0.1)
        print("Leaving Generator")
        return z_4
```

Discriminator Architecture:

- Our ‘Discriminator’ class evaluates the images fed to it and outputs a value representing the likelihood that the image is real.
- It learns the correct classification as a joint distribution of the input image that is to be translated and the target/generated image. This helps the model better learn the relationships between the input and target images.
- The Discriminator takes two images as input and concatenates them along the channel dimension to form a single tensor ‘x’, which is then further processed.
- This tensor undergoes processing through various discriminator layers, including:
    - Convolutional layers

    - Batch normalization
    - Leaky ReLU activation functions, allowing for feature extraction.
- Dropout is applied to prevent overfitting.
- Zero padding is added to maintain spatial dimensions.
- The last convolutional layer produces a single-channel output.
- This output is passed through a sigmoid activation function to convert it into a probability between [0,1].
- Values close to 1 represent real images, while values close to 0 represent generated or fake images.

```
'''Discriminator Class'''

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(6, 64, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False)
        self.zero_pad1 = nn.ZeroPad2d(1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=1, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(256)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.zero_pad2 = nn.ZeroPad2d(1)
        self.last_conv = nn.Conv2d(256, 1, kernel_size=4, stride=1, padding=0)
        self.drop = nn.Dropout2d(p = 0.4)

    def forward(self, inp, tar):

        print("Entering Discriminator")
        x_1 = torch.cat([inp, tar], dim=1)
        x_2 = self.leaky_relu(self.bn1(self.conv1(x_1)))
        x_3 = self.drop(x_2)
        x_4 = self.leaky_relu(self.bn2(self.conv2(x_3)))
        x_5 = self.drop(x_4)
        x_6 = self.leaky_relu((self.conv3(x_5)))
        x_7 = self.drop(x_6)
        x_8 = self.zero_pad1(x_7)
        x_9 = (self.last_conv(x_8))
        x_10 = self.drop(x_9)
        print("Leaving Discriminator")
        return torch.sigmoid(x_10)
```

Major Learning

Some major takeaways from the process of designing this model architecture and implementing it were as follows:

- The Encoder class required us to innovate on the readily available Pix2Pix architecture by adding VAE components to it. This required us to print and match the dimensions of what we needed and what we were getting from our model and make changes along the way. This taught us how to use MaxPool2d and fully connected layers in order to achieve the required goal. Further, adding the reparameterization trick was also tricky since there were issues with how the dimensions were being changed in the layers.

- The Generator class consisted of the Conv2d Transpose layers. This required further insight on how the dimensions are manipulated in this class' layers. We had to figure out how the sizes changed and how we could add or remove different layers to get our desired results.
- The Discriminator class consisted of Conv2d layers. The dimensions of the tensor after being processed through these layers was initially not compatible with the output we needed. We, therefore, had to figure out how to change the kernel size, stride and padding to get our desired output.
- The project also enabled us to explore multiple VAE and GAN architectures to find the most appropriate ones for our task. This helped us increase our knowledge on these topics as well as come up with ideas to innovate on existing architectures.

3. Loss Functions:

Discriminator Loss:

The discriminator loss is minimized to train the discriminator to correctly classify the image pair as (input image, target image) or (input image, generated image). The discriminator loss is simple adversarial loss.

```
out_true = self.D(input_image, output_image)
y_true = torch.ones_like(out_true)
d_real_loss = criterion(out_true, y_true)
```

```
out_fake = self.D(input_image, x_generated)
y_fake = torch.ones_like(out_fake)
d_fake_loss = criterion(out_fake, y_fake)
d_loss = d_real_loss + d_fake_loss
```

Generator-Encoder Loss:

For the generator-encoder loss, we have used the usual loss function used in VAEs and Generators and have combined them. This includes:

KL-Divergence: This term acts as a regularizer of the latent space. The formula is given as follows

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \ln\left(\frac{P(x)}{Q(x)}\right)$$

```
#Calculating KL Divergence between latent and prior distributipn
kl_loss = -0.5 * torch.sum(1+ log_sigma - mu.pow(2) - torch.exp(log_sigma), dim = -1)
kl_loss = torch.mean(kl_loss)
```

Reconstruction Loss: This term improves the quality of the resulting images. The formula is as follows:

$$-\mathbb{E}_{z\sim q(z|x)} log(p_{model}(x|z))$$

```
rec_loss = reconstruction_loss(x_generated, output_image)
```

Adversarial Loss: This loss function is included to train the generator-encoder to produce learnt images resembling the target images, which will fool the discriminator into classifying them as the actual target images. The ge adversarial loss and the discriminator loss "play" the min-max game so that the generator can eventually generate realistic images.

```
g_adv = criterion(out_fake, y_true)
```

```
ge_loss = g_adv + rec_loss + kl_loss
```

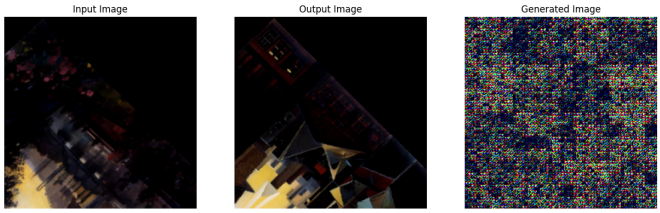
4. **Training Strategy:**

- Our 'Trainer' class comprises the training loop for our model. We have used Adam optimizers for the Generator and the Encoder. The parameters of the models are passed to the optimizers.
- We have used BCELoss for the adversarial loss and MSELoss for the reconstruction loss. We have delved deeper into the loss functions in the above section.
- By optimizing the discriminator and generator separately, our model is trained adversarially. The generator learns to generate images that are indistinguishable from real images by trying to minimize the discriminator's ability to differentiate between real and fake images.
- At the same time, the discriminator is trained to become better at distinguishing between real and fake images. This adversarial process continues until the generator produces images that are realistic.

```
self.D.zero_grad()
# Train discriminator with real images
out_true = self.D(input_image, output_image)
y_true = torch.ones_like(out_true)
d_real_loss = criterion(out_true, y_true)

# Generate fake images
encoder_out = self.E(input_image)
z_fake = encoder_out[0]
mu = encoder_out[1]
log_sigma = encoder_out[2]

#Calculating KL Divergence between latent and prior distributipn
kl_loss = -0.5 * torch.sum(1+ log_sigma - mu.pow(2) - torch.exp(log_sigma), dim = -1)
kl_loss = torch.mean(kl_loss)

x_generated = self.G( z_fake.view(z_fake.size(0), 50, 1, 1))
out_fake = self.D(input_image, x_generated)
y_fake = torch.ones_like(out_fake)
d_fake_loss = criterion(out_fake, y_fake)
d_loss = d_real_loss + d_fake_loss
d_loss.backward(retain_graph=True)
optimizer_d.step()

# Update generator
self.G.zero_grad()
#Train generator to fool discriminator
out_fake = self.D(input_image, x_generated)
rec_loss = reconstruction_loss(x_generated, output_image)
g_adv = criterion(out_fake, y_true)
#Kl div getting tensor loss, need scalar loss.
ge_loss = g_adv + rec_loss + kl_loss
ge_loss.backward()
optimizer_ge.step()

#Computing gradients and backpropagate
d_losses += d_loss.item()
ge_losses += ge_loss.item()
```

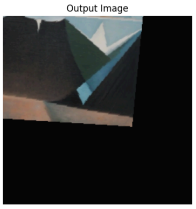
5. Evaluation Metrics:

Due to the lack of computational resources for the required task, we have limited the number of epochs to between 6 and 10, keeping the batch size at 40 consistently and the number of augmentations for each pair of images as 4. We have captured below one of our tries for the same.

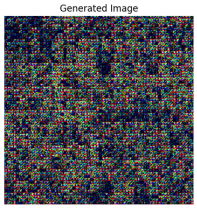


Input Image

Output Image

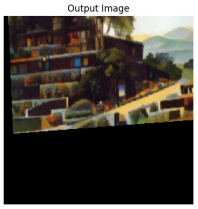


Generated Image

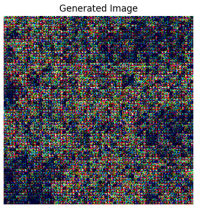


Input Image

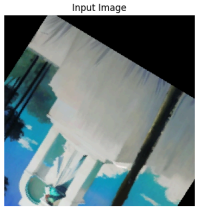
Output Image



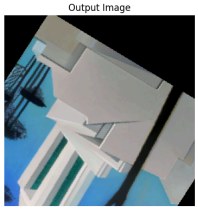
Generated Image



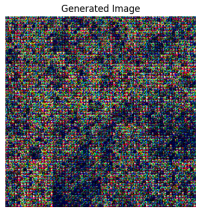
Input Image



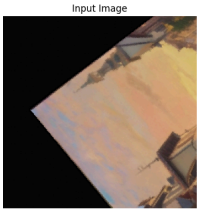
Output Image



Generated Image

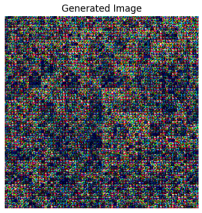


Input Image

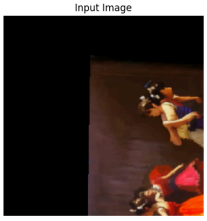


Output Image

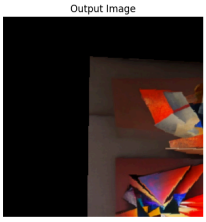
Generated Image



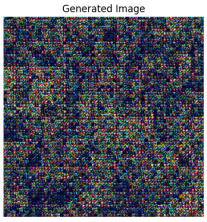
Input Image

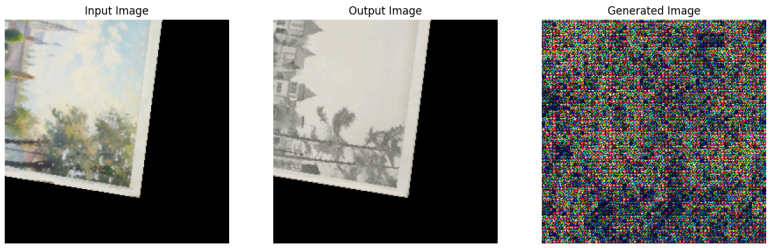


Output Image



Generated Image






```
Training... Epoch: 0, Discriminator Loss: 1.049, Generator Loss: 2.979
 40/210: [>..............................] - ETA 0.0sEntering Discriminator

Training... Epoch: 1, Discriminator Loss: 0.573, Generator Loss: 1.423
 40/210: [>..............................] - ETA 0.0sEntering Discriminator

Training... Epoch: 2, Discriminator Loss: 0.493, Generator Loss: 1.136
 40/210: [>..............................] - ETA 0.0sEntering Discriminator

Training... Epoch: 3, Discriminator Loss: 0.593, Generator Loss: 0.907
 40/210: [>..............................] - ETA 0.0sEntering Discriminator

Training... Epoch: 4, Discriminator Loss: 0.546, Generator Loss: 0.775
 40/210: [>..............................] - ETA 0.0sEntering Discriminator

Training... Epoch: 5, Discriminator Loss: 0.615, Generator Loss: 0.687
 40/210: [>..............................] - ETA 0.0sEntering Discriminator

Training... Epoch: 6, Discriminator Loss: 0.526, Generator Loss: 0.690
```