Super-Resolution Using GANs and Satellite Images

Introduction

Super-resolution is a technique used to enhance the resolution of low-resolution images, making them appear sharper and more detailed. In this project, we leverage Generative Adversarial Networks (GANs) to perform Super-Resolution (SR) on satellite images.

1. Libraries Used

TensorFlow/Keras	• For deep learning model building (e.g., layers, optimizers, loss functions).
OpenCV	• Image preprocessing and augmentation.
h5py	• Handling and processing .h5 format datasets.
Matplotlib	• Visualization of images and training performance.
Scikit-learn	• Data preprocessing (scaling, train-test split).
Joblib	• Saving and loading models.
Glob, OS	• File handling and dataset management.

2. Dataset Handling

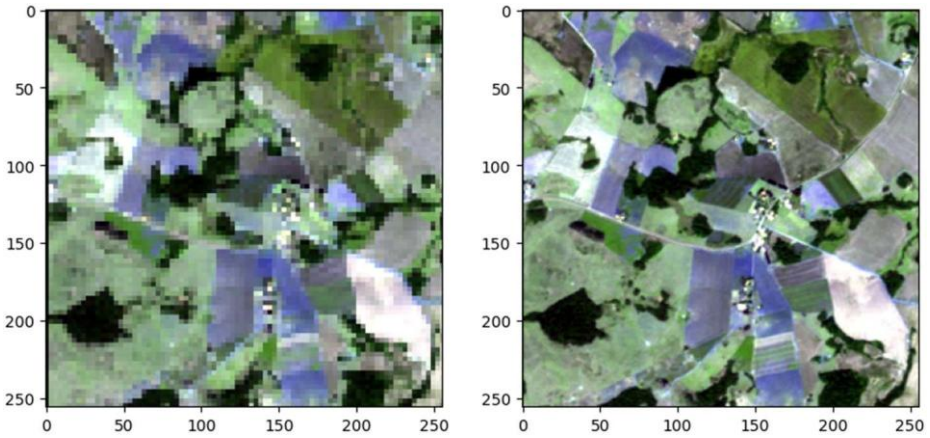
- The dataset is stored in HDF5 format (.h5).
- It contains pairs of low-resolution (LR) images and their corresponding high-resolution (HR) images.
- Images are extracted and pre-processed before being fed into the model.

Data Loading Code:

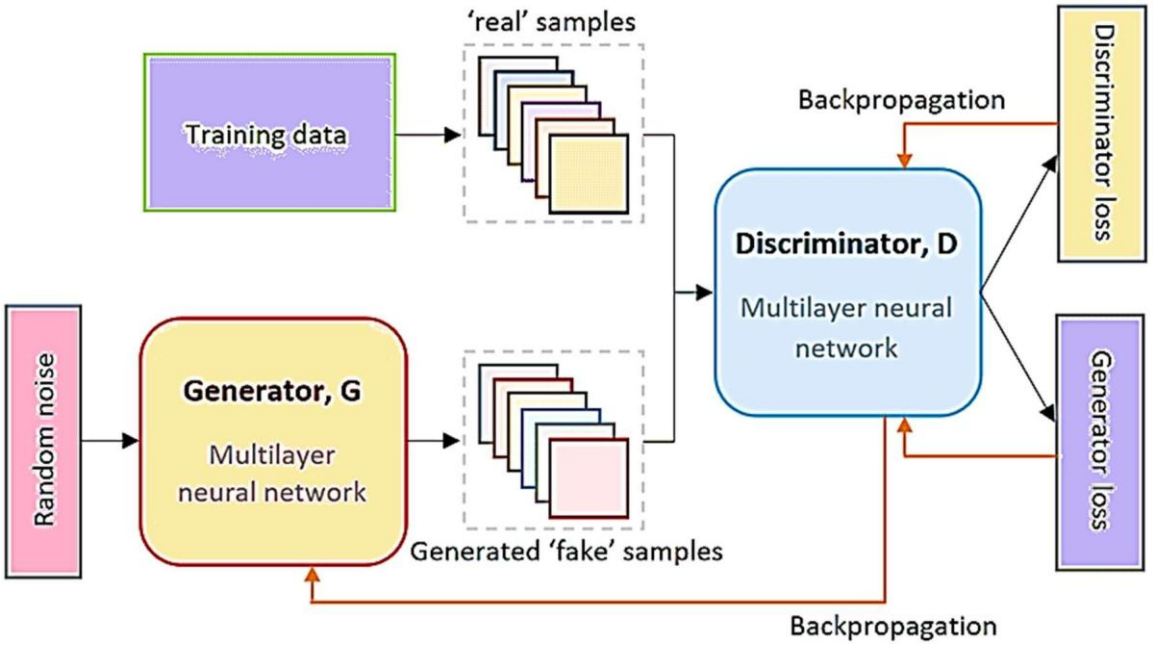
```
import h5py

# Load dataset
file_path = '/content/train.h5'
dataset = h5py.File(file_path, 'r')
X = dataset['x']   # Low-resolution images
y = dataset['y']   # High-resolution images

# Check dataset shape
print(X.shape, y.shape)
```

Dataset Details

Number of Images: 9245 Image Pairs
Image Dimensions: 256*256 pixels
Some sample images are given below:



3. Super-Resolution GAN (SRGAN) Architecture



The Generative Adversarial Network (GAN) architecture, particularly in the context of image-to-image translation tasks like the Pix2Pix model, is comprised of a Generator and a Discriminator that work together in an adversarial manner to improve the quality of generated images. The Generator aims to create realistic images by learning the mapping between input images and their corresponding target images, while the Discriminator attempts to distinguish between real and fake image pairs. This process is guided by a two-loss system: the generator's loss encourages the creation of visually accurate images, and the discriminator's loss helps it effectively classify real and fake images.

The Discriminator model is a deep convolutional neural network (CNN) that takes as input both the source image and the generated image. It outputs a probability indicating whether the image pair is real or fake. The discriminator uses several convolutional layers with increasing filters to extract high-level features from the images. It applies LeakyReLU activation functions and batch normalization to stabilize training. The final output layer uses a sigmoid activation to produce a binary classification (real or fake). During training, the weights of the discriminator are fixed when training the GAN, ensuring that the generator improves while the discriminator remains constant.

The Generator model is designed to convert a source image into a target-like image, effectively learning how to perform image-to-image translation. It follows an encoder-decoder architecture, where the encoder progressively downscales the image and learns feature representations, and the decoder upscales the image to generate the final output. The encoder is composed of several convolutional layers, and the decoder utilizes transposed convolutions (also known as deconvolutions) to upsample the image. Skip connections are used between corresponding layers of the encoder and decoder to preserve spatial information. The final output is generated using a tanh activation to constrain the pixel values to the range [-1, 1].

In training the GAN, the Discriminator and Generator are trained together but in a way that each model's parameters are updated differently. The Discriminator is trained to distinguish between real and fake images using binary cross-entropy loss, while the Generator is trained to create images that minimize the difference between the generated and target images using mean absolute error (MAE). The GAN model is compiled with an Adam optimizer, and the generator's loss is weighted more heavily than the discriminator's to ensure the generator focuses on improving the realism of its generated images. During each epoch of training, the discriminator is trained on both real samples and fake samples generated by the generator. Afterward, the generator is updated based on the feedback from the discriminator. This adversarial process continues through multiple epochs, gradually improving both models.

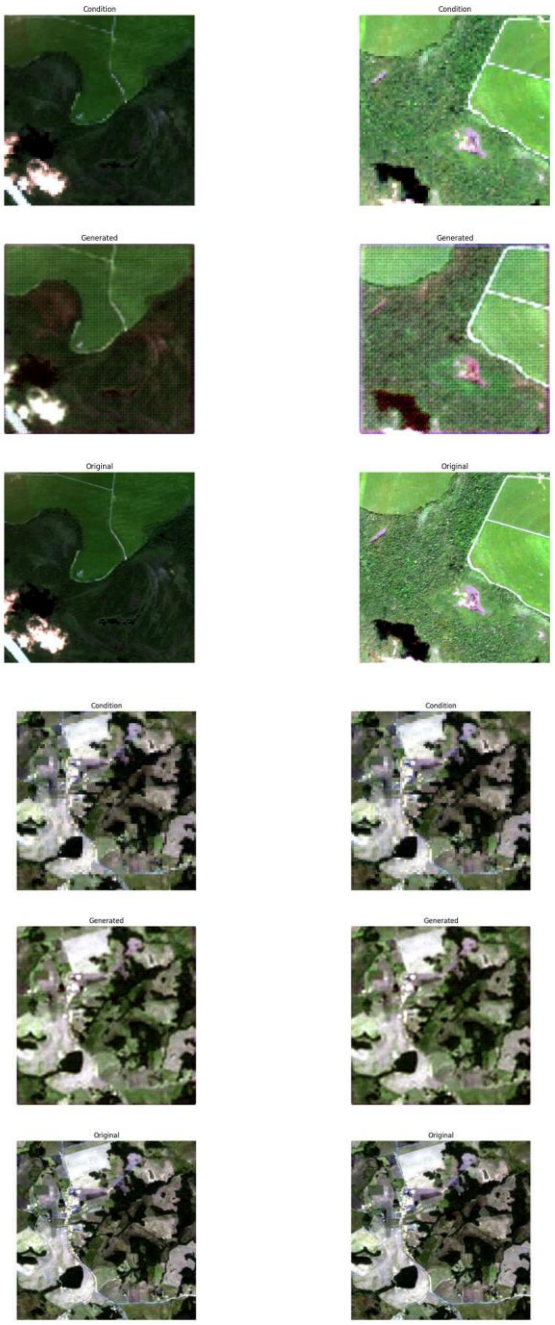
The training process iterates through the dataset for multiple epochs, where for each batch, it first trains the discriminator on real images, then on fake images generated by the generator. Following this, the generator is trained via the GAN model, which uses the generator's output to update the generator's weights. The function tracks the loss for both the discriminator and generator at each step and prints performance metrics. After each epoch, the function summarizes the performance, saves the generator's and discriminator's models, and logs the training process. By repeating this loop for many epochs, the GAN progressively learns to generate increasingly realistic images, improving its performance with each iteration.

4. Training Process
    1. Load dataset and preprocess images.
    2. Initialize Generator and Discriminator.
    3. Train the Discriminator to distinguish real and fake HR images.
    4. Train the Generator using Perceptual Loss:
        (i) Mean Squared Error (MSE) between generated HR and real HR.
        (ii) Adversarial Loss from GAN training.
    5. Iterate for multiple epochs.

5. Results and Evaluation

- The trained Generator produces super-resolved images that are visually closer to ground truth HR images.
- Evaluation metrics:
  (i) PSNR (Peak Signal-to-Noise Ratio)
  (ii) SSIM (Structural Similarity Index)

Sample Results:



[D1 loss: 0.47619977593421936]
[D2 loss: 0.47628289461135864]
[G loss: 2.6246891021728516]

Code Snippet:
Complete Code (GitHUB): https://github.com/krixhnaprasad/Super-Resolution-of-Satellite-Images

Discriminator:

```
def define_discriminator(image_shape):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # source image input
    in_src_image = Input(shape=image_shape)
    # target image input
    in_target_image = Input(shape=image_shape)
    # concatenate images channel-wise
    merged = Concatenate()([in_src_image, in_target_image])
    # C64
    d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
    d = LeakyReLU(alpha=0.2)(d)
    # C128
    d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # C256
    d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # C512
    d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # second last output layer
    d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # patch output
    d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
    patch_out = Activation('sigmoid')(d)
    # define model
    model = Model([in_src_image, in_target_image], patch_out)
    # compile model
    opt = Adam(learning_rate=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
    return model
```

Generator:

```
def define_generator(image_shape):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # image input
    in_image = Input(shape=image_shape)
    # encoder model
    e1 = define_encoder_block(in_image, 64, batchnorm=False)
    e2 = define_encoder_block(e1, 128)
    e3 = define_encoder_block(e2, 256)
    e4 = define_encoder_block(e3, 512)
    e5 = define_encoder_block(e4, 512)
    e6 = define_encoder_block(e5, 512)
    e7 = define_encoder_block(e6, 512)
    # bottleneck, no batch norm and relu
    b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
    b = Activation('relu')(b)
    # decoder model
    d1 = decoder_block(b, e7, 512)
    d2 = decoder_block(d1, e6, 512)
    d3 = decoder_block(d2, e5, 512)
    d4 = decoder_block(d3, e4, 512, dropout=False)
    d5 = decoder_block(d4, e3, 256, dropout=False)
    d6 = decoder_block(d5, e2, 128, dropout=False)
    d7 = decoder_block(d6, e1, 64, dropout=False)
    # output
    g = Conv2DTranspose(image_shape[-1], (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d7)
    out_image = Activation('tanh')(g)
    # define model
    model = Model(in_image, out_image)
    return model
```