Enhancing image Resolution (GAN)

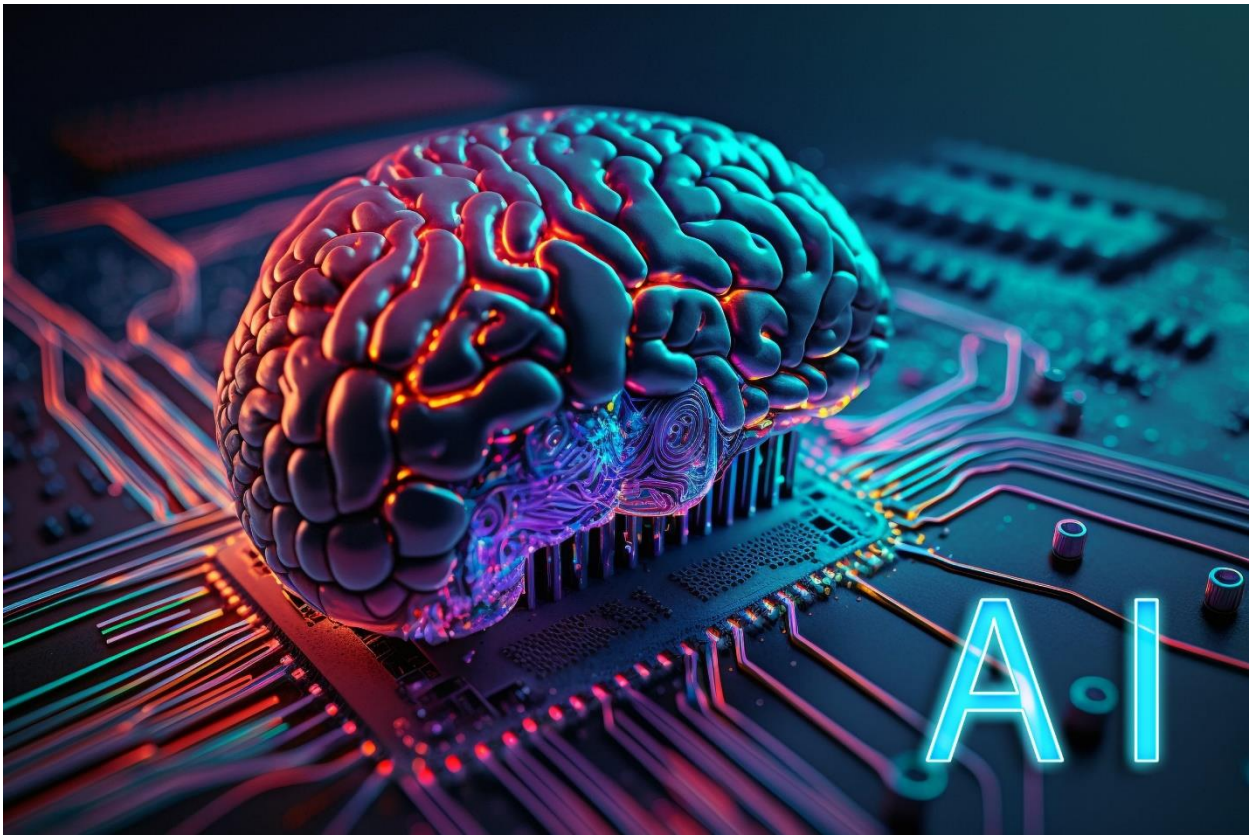
Adnane Mouhmid

Anass El Foughali

Summary

I. Context

In this project, we aim to enhance the resolution of low-quality images using the Enhanced Super-Resolution Generative Adversarial Network (ESRGAN). Image super resolution is a critical task in computer vision with applications in various fields such as medical imaging, satellite imaging, and photography. To facilitate easy access and usability, and implementing it in a web application using Flask (Python framework), that allows users to upload images and receive enhanced high-resolution outputs.



II. Project Objectives

1. Develop an ESRGAN model to upscale low-resolution images to high-resolution images.

2. Train the ESRGAN model using high-quality image datasets.

3. Create a user-friendly web application using Flask for image upload and enhancement.

III. Generative adversarial network (GAN)

Generative Adversarial Networks (GANs) have revolutionized the field of image generation and enhancement. ESRGAN, an advanced variant of GAN, introduces a more sophisticated generator and discriminator architecture, allowing for finer detail and higher-quality image generation. Previous models like SRGAN laid the foundation for ESRGAN by demonstrating the feasibility of GANs for super-resolution tasks.

IV. Model Architecture

a) Generator

The generator model is designed to take low resolution images of shape (128, 128, 3) and upscale them to high resolution images of shape (256, 256, 3). The architecture includes:

1. Initial Convolutional Layer: Captures low-level features from the input image.
2. Residual Blocks: residual blocks enhance feature learning through skip connections and batch normalization.
3. Up sampling Layers: Two up sampling layers double the image resolution sequentially.
4. Final Convolutional Layer: Produces the high-resolution output image.

b) Discriminator

The discriminator model differentiates between real high-resolution images and generated images. It includes:

1. Convolutional Layers: Apply convolution with increasing filter sizes and down sampling.
2. LeakyReLU Activations: Introduce non-linearity.
3. Batch Normalization: Normalize activations to improve training stability.
4. Dense Layer: Outputs the probability of the input image being real.

V. Dataset Preparation

Our training dataset consists of 800 image, in order to train and evaluate or model we need to perform the following transformations to our data:

1. Resizing the train and validation set to a unified size (256x256 pixels)
2. Creating a lower resolution version of each image (128x128 pixels)
3. Augmenting the images and introduce variety to address overfitting

a) Resizing

Resizing ensures all images have a consistent size, which is crucial for feeding into the model, given that it requires input images of a fixed size.

```
import cv2
import os

target_size = (256, 256)
def resize_images(dir_path, output_dir):
    try:
        os.makedirs(output_dir, exist_ok=True)

        for filename in os.listdir(dir_path):
            if filename.endswith('.jpg') or filename.endswith('.png'):
                img_path = os.path.join(dir_path, filename)
                img = cv2.imread(img_path)
                resized_img = cv2.resize(img, target_size, interpolation=cv2.INTER_AREA)
                output_path = os.path.join(output_dir, 'resized_' + filename)
                cv2.imwrite(output_path, resized_img)
    except Exception as e:
        print(f"Error processing images in {dir_path}: {e}")


resize_images(train_data_dir, resized_train_dir)
resize_images(validation_data_dir, resized_valid_dir)
```

b) Creating a low-resolution version of the images

Generating low-resolution versions simulates the input images' degradation, which is essential for training models designed for super-resolution or enhancement tasks.

1. Resize the 256x256 images to a lower resolution of 128x128 pixels. This provides the input data that the model will learn to upscale.
2. Ensure the low-resolution images retain the essential features of the original images despite the reduced size.

```
def create_low_resolution_dataset(input_dir, output_dir, scale_factor):
    os.makedirs(output_dir, exist_ok=True)
    for filename in os.listdir(input_dir):
        if filename.endswith('.jpg') or filename.endswith('.png'):
            img_path = os.path.join(input_dir, filename)
            img = cv2.imread(img_path)
            height, width = img.shape[:2]
            new_height, new_width = height // scale_factor, width // scale_factor
            low_res_img = cv2.resize(img, (new_width, new_height), interpolation=cv2.INTER_CUBIC)
            output_path = os.path.join(output_dir,filename)
            cv2.imwrite(output_path, low_res_img)


scale_factor = 2
create_low_resolution_dataset(resized_train_dir, low_res_train_dir, scale_factor)
create_low_resolution_dataset(resized_valid_dir, low_res_valid_dir, scale_factor)
```

c) Data augmentation

Augmentation introduces variability into the training dataset and helps preventing overfitting. It ensures the model generalizes better to new, unseen data.

1. Rotation: Randomly rotate images within a specified range.
2. Zoom: Apply random zoom-in or zoom-out effects.
3. Horizontal and Vertical Flips: Flip images randomly to introduce variability.
4. Cropping: Randomly crop images.

```python
def random_crop(image, top, left, patch_size=(128,128), target_size= (256,256)):
    top,left= int(top), int(left)
    cropped_image = image[top:top + patch_size[0], left:left + patch_size[1]]
    resized= cv2.resize(cropped_image, target_size)
    return resized

def random_rotation(image, angle,  target_size= (256,256)):
    rows, cols = image.shape[:2]
    rotation_matrix = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1)
    rotated_image = cv2.warpAffine(image, rotation_matrix, (cols, rows))
    resized= cv2.resize(rotated_image, target_size)
    return resized

def random_flip(image, flip_code,  target_size= (256,256)):
    flipped_image = cv2.flip(image, flip_code)
    resized= cv2.resize(flipped_image, target_size)
    return resized

def random_scaling(image, scale_factor,  target_size= (256,256)):
    scaled_image = cv2.resize(image, None, fx=scale_factor, fy=scale_factor, interpolation=cv2.INTER_LINEAR)
    resized= cv2.resize(scaled_image, target_size)
    return resized
```

```python
import random

def augment_image(image, top, left, angle, flip_code, scale_factor, patch_size=(128,128), target_size= (256,256)):
    images= {}
    images["cropped_"]= random_crop(image, top, left, patch_size, target_size)
    images["rotated_"]=random_rotation(image, angle, target_size)
    images["flipped_"]=random_flip(image, flip_code, target_size)
    images["scaled_"]= random_scaling(image, scale_factor, target_size)

    return images
```

```python
max_angle= 360
max_scale_factor= 1.2
min_scale_factor= 0.8
patch_size=(128, 128)

def generate_augmentation_params(image_shape= (256,256,3), patch_size=(128, 128)):
    top = np.random.randint(0, image_shape[0] - patch_size[0])
    left = np.random.randint(0, image_shape[1] - patch_size[1])
    angle = np.random.uniform(-max_angle, max_angle)
    flip_code = np.random.randint(-1, 2)
    scale_factor = np.random.uniform(min_scale_factor, max_scale_factor)
    return top, left, angle, flip_code, scale_factor
```

```python
def augment_images_pair(high_res_dir, low_res_dir, high_res_output_dir, low_res_output_dir):
    if not os.path.exists(high_res_output_dir):
        os.makedirs(high_res_output_dir)
    if not os.path.exists(low_res_output_dir):
        os.makedirs(low_res_output_dir)

    for filename in os.listdir(high_res_dir):
        if filename.endswith(('.png', '.jpg', '.jpeg')):
            high_res_img_path = os.path.join(high_res_dir, filename)
            low_res_img_path = os.path.join(low_res_dir, filename)

            high_res_image = cv2.imread(high_res_img_path)
            low_res_image = cv2.imread(low_res_img_path)

            top, left, angle, flip_code, scale_factor = generate_augmentation_params(high_res_image.shape)

            high_res_augmented = augment_image(high_res_image, top, left, angle, flip_code, scale_factor, patch_size=(128,128), target_size= (256,256))
            low_res_augmented = augment_image(low_res_image, top//2, left//2, angle, flip_code, scale_factor,patch_size=(64,64), target_size= (128,128))


            for key in high_res_augmented.keys():
                high_res_output_path = os.path.join(high_res_output_dir, f"{os.path.splitext(filename)[0]}_{key}{os.path.splitext(filename)[1]}")
                low_res_output_path = os.path.join(low_res_output_dir, f"{os.path.splitext(filename)[0]}_{key}{os.path.splitext(filename)[1]}")

                if high_res_augmented[key] is None or high_res_augmented[key].size == 0:
                    print(f"High-res augmentation {key} failed for image: {high_res_img_path}")
                    continue

                if low_res_augmented[key] is None or low_res_augmented[key].size == 0:
                    print(f"Low-res augmentation {key} failed for image: {low_res_img_path}")
                    continue

                cv2.imwrite(high_res_output_path, high_res_augmented[key])
                cv2.imwrite(low_res_output_path, low_res_augmented[key])
```

```python
augment_images_pair(high_res_dir=resized_train_dir, low_res_dir=low_res_train_dir, high_res_output_dir=augmented_data_high, low_res_output_dir=augmented_data_low)
```

VI. Generating the dataset

We implemented a robust dataset creation pipeline using TensorFlow. Images are loaded and preprocessed using a custom function that reads the image files, decodes them into a tensor format, resizes them, and normalizes the pixel values to the range of [-1, 1]. We then paired corresponding low and high resolution images, ensuring consistency and alignment between the two sets.

This pairing is crucial for training the model to learn the mapping from LR to HR images. The dataset is created from the directory of image paths and batched for efficient loading during training. Additionally.

```python
import tensorflow as tf
import os

lr_image_size = (128, 128)
hr_image_size = (256, 256)
batch_size = 16


def load_image(image_path, target_size):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, target_size)
    image = (image / 127.5) - 1.0
    return image

def load_lr_hr_pair(lr_image_path, hr_image_path):
    lr_image = load_image(lr_image_path, lr_image_size)
    hr_image = load_image(hr_image_path, hr_image_size)
    return lr_image, hr_image

def create_dataset(lr_dir, hr_dir, batch_size):
    lr_image_paths = sorted([os.path.join(lr_dir, img) for img in os.listdir(lr_dir) if img.endswith(('.jpg', '.jpeg', '.png'))])
    hr_image_paths = sorted([os.path.join(hr_dir, img) for img in os.listdir(hr_dir) if img.endswith(('.jpg', '.jpeg', '.png'))])

    dataset = tf.data.Dataset.from_tensor_slices((lr_image_paths, hr_image_paths))
    dataset = dataset.map(lambda lr, hr: load_lr_hr_pair(lr, hr), num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(tf.data.AUTOTUNE)

    return dataset

train_lr_dir = '/content/div2k-dataset/augmented_train_low/all'
train_hr_dir = '/content/div2k-dataset/augmented_train_high/all'


train_dataset = create_dataset(train_lr_dir, train_hr_dir, batch_size)

for lr_batch, hr_batch in train_dataset.take(1):
    print(f'Train LR Batch shape: {lr_batch.shape}')
    print(f'Train HR Batch shape: {hr_batch.shape}')
```

VII. Building the model

The ESRGAN model consists of a generator and a discriminator. The generator enhances low-resolution images to high resolution ones, while the discriminator distinguishes real high resolution images from generated ones.

a) Generator

The generator takes 128x128x3 images and processes them through a convolutional layer followed by LeakyReLU activation function. It uses residual blocks, each with convolutional layers, batch normalization, and LeakyReLU, to learn complex features. After the residual blocks, it upsamples the image to double its resolution and outputs the final 256x256x3 HR image using a tanh activation.

```
def build_generator():
    input = Input(shape=(128, 128, 3))
    x = Conv2D(64, (9, 9), padding='same')(input)
    x = LeakyReLU(alpha=0.2)(x)

    res = x
    for _ in range(16):
        res = residual_block(res)

    res = Conv2D(64, (3, 3), padding='same')(res)
    res = BatchNormalization()(res)
    x = Add()([x, res])

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(64, (3, 3), padding='same')(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(3, (9, 9), padding='same', activation='tanh')(x)

    return Model(inputs=input, outputs=x)
```

b) Discriminator

The discriminator takes 256x256x3 images and processes them through convolutional layers with increasing filters (64 to 512) and strides, followed by batch normalization and LeakyReLU activations. It ends with global average pooling and a dense layer to output the probability of the image being real or generated.

```
def build_discriminator():
    input = Input(shape=(1024,1024, 3))
    x = Conv2D(64, (3, 3), strides=1, padding='same')(input)
    x = LeakyReLU(alpha=0.2)(x)

    for filters in [64, 128, 128, 256, 256, 512, 512]:
        x = Conv2D(filters, (3, 3), strides=2, padding='same')(x)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=0.2)(x)

    x = GlobalAveragePooling2D()(x)
    x = Dense(1, activation='tanh')(x)

    return Model(inputs=input, outputs=x)
```

c) Combined model

The combined model integrates the generator and discriminator.

```python
def build_combined_model(generator, discriminator):
    discriminator.trainable = True
    hr_input = Input(shape=(128, 128, 3))
    generated_hr = generator(hr_input)
    valid = discriminator(generated_hr)
    return Model(inputs=hr_input, outputs=[generated_hr, valid])
```

a) Training process

The training of the ESRGAN model focuses on enhancing the resolution of low-resolution images to high-resolution equivalents through adversarial and perceptual learning techniques.

b)

```python
@tf.function
def train_step(lr_images, hr_images, generator, discriminator, vgg_model, generator_optimizer, discriminator_optimizer):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        sr_images = generator(lr_images, training=True)

        gen_content_loss = content_loss(hr_images, sr_images, vgg_model)
        gen_pixel_loss = pixel_loss(hr_images, sr_images)
        gen_loss = gen_content_loss + 0.01 * gen_pixel_loss

        fake_output = discriminator(sr_images, training=True)
        disc_fake_loss = adversarial_loss(fake_output)
        real_output = discriminator(hr_images, training=True)
        disc_real_loss = discriminator_loss(real_output, fake_output)
        disc_loss = disc_real_loss + disc_fake_loss

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

    return gen_loss, disc_loss

def train(train_dataset, epochs, generator, discriminator, vgg_model, checkpoint_prefix):
    batch_size = 16
    train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)

    for epoch in range(epochs):
        print(f"Starting epoch {epoch+1}/{epochs}")
        for lr_images, hr_images in tqdm(train_dataset):
            gen_loss, disc_loss = train_step(lr_images, hr_images, generator, discriminator, vgg_model, generator_optimizer, discriminator_optimizer)

        print(f'Epoch {epoch+1}, Generator Loss: {gen_loss.numpy()}, Discriminator Loss: {disc_loss.numpy()}')

        if (epoch + 1) % 10 == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)
```

1.Data Preparation: LR-HR image pairs are prepared and preprocessed for training.

2.Training Loop:

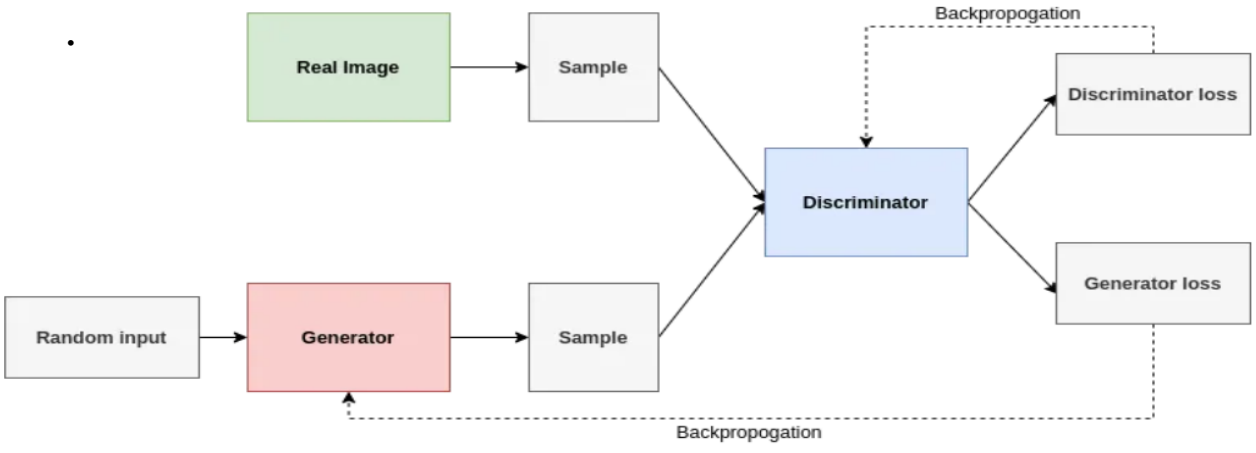
1. LR images are input into the generator to produce SR images.
2. Losses (generator and discriminator) are computed based on SR and HR images.
3. Gradients are calculated and used to update the generator and discriminator parameters.

Optimization: Adam optimizers adjust model weights to minimize losses during training.

Monitoring: Progress is tracked by evaluating generator and discriminator losses regularly.

Model checkpoints are saved periodically for recovery and evaluation.

The ESRGAN training process iteratively refines the generator's ability to produce realistic images by balancing adversarial feedback and perceptual fidelity. This approach improves image resolution effectively, making ESRGAN a valuable tool in image enhancement tasks.



Loss Functions

1.Discriminator Loss: Measures the accuracy in distinguishing real and fake images.

2. Generator Loss: Combines adversarial loss (feedback from the discriminator) and pixel-wise loss (difference from the real high-resolution images).

VIII. Flask Application

a) Overview

Flask is a lightweight and flexible web framework for Python, designed for quick and easy web application development. It provides essential tools without imposing a specific structure, making it ideal for both small and large projects. Flask's simplicity and modularity make it popular among developers.

b) Key Endpoints

1. **/:** render the image upload page.
2. **/upload**: Handles image uploads.
3. **/enhance**: Processes the uploaded image using the ESRGAN model and returns the result.
4. **/result/<image>:** Renders a page to display the result, showing the generated image.
5. **/uploads/<filename>:** Serves the uploaded files from the UPLOAD_FOLDER directory
6. **/generated/<filename>:** Serves the generated (enhanced) files from the GENERATED_FOLDER directory.

IX. Results

To test out model, we used images from our validation data (never seen by the model before) that have been preprocessed (resized and normalized)

Original Image

Enhanced Image

Original Image

Enhanced Image

Original Image

Enhanced Image