University of Moratuwa

Department of Electronic and Telecommunication Engineering

EN3160 – Image Processing and Machine Vision

Super-resolution (NTIRE18)
Project Report

Miranda C.M.C.C. 200396U

Nirushtihan B. 200431B

Contents

1. Introduction of the Problem

We have been tasked with completing one of the challenges in the 2nd *NTIRE* competition. This specific challenge is centered around the restoration of rich details in low-resolution images. The challenge revolves around single image super-resolution, aiming to recover high-frequency details lost in low-resolution images by referencing paired low-resolution and high-resolution images. This problem is complex due to the multitude of potential high-resolution counterparts for each low-resolution image, escalating with the magnification factor. To tackle this challenge, we researched relevant papers and opted to implement the *"Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network"* paper. This paper introduced SRGAN as the solution, and we have implemented the SRGAN model for our approach.

2. Existing Alternative Methods

2.1 Bicubic Interpolation method

Bicubic interpolation is a technique used to estimate pixel values in a higher-resolution image based on a 2x2 neighborhood of pixels from a lower-resolution image. It employs a cubic polynomial function and 16 weighting coefficients to perform the interpolation. Bicubic interpolation is often used for upscaling images, including in super-resolution tasks, to create smoother and more detailed images during the upscaling process.

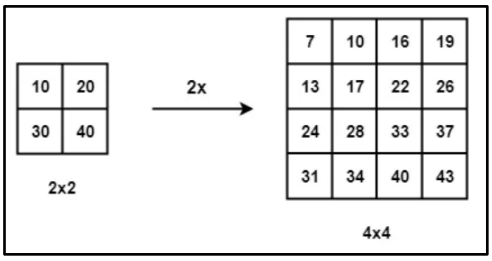


Figure 1: Bicubic Interpolation

2.2 SRResNet method

SRResNet is a deep learning architecture for single-image super-resolution. It uses residual blocks to learn to upscale low-resolution images, focusing on capturing image details. The network is trained to minimize the difference between its output and high-resolution images. It is a foundational model in deep learning-based super-resolution and has been extended to more advanced models like SRGAN.

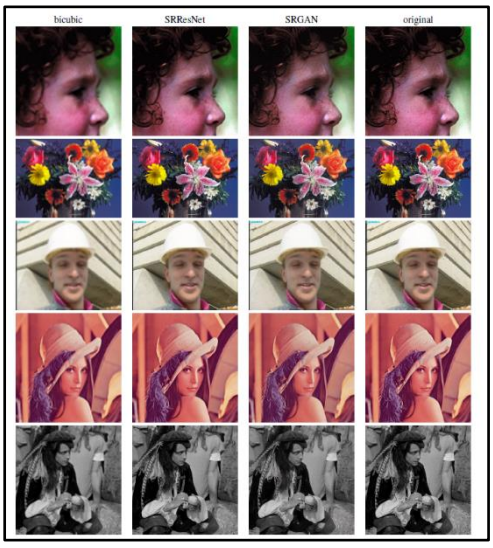


Figure 2: Comparison between different methods

3. Method of Solving the Problem

After conducting thorough research, we have decided to implement SRGAN, a generative adversarial network (GAN) designed for image super-resolution. SRGAN is a pioneering framework capable of generating photo-realistic natural images with a 4x upscaling factor.

In this paper ("***Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network***" by *Christian Ledig, Lucas Theis, Ferenc Husz´ar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi Twitt*er), they introduce a perceptual loss function, which comprises an adversarial loss and a content loss. The adversarial loss guides our solution towards the realm of natural images by using a discriminator network that's trained to distinguish between the super-resolved images and the original photo-realistic images. Additionally, they employ a content loss that draws inspiration from perceptual similarity rather than mere similarity in pixel space. Our deep residual network demonstrates the ability to recover photo-realistic textures from heavily down-sampled images, as demonstrated on public benchmarks.

3.1 Brief method

The research paper on generating '*Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network'* proposes a loss called perceptual loss. This loss evaluates the image quality based on its perceptual quality. An interesting way to do this is by comparing the high-level features of the generated image and the ground truth image. We can obtain these high-level features by passing both of these images through a pre-trained image classification network. (such as a VGG-Net or a ResNet)

The generator architecture is basically a fully convolutional SRRESNET model which is utilized for generating high-quality super-resolution images. The addition of the discriminator model, which acts as an image classifier, is constructed to ensure that the overall architecture adjusts accordingly to the quality of the images and the resulting images are much more optimal. The SRGAN architecture generates plausible-looking natural images with high perceptual quality.

3.2 Method Description

3.2.1 Perceptual Loss

The paper proposes SRGAN which is a GAN- based network optimized for a new perceptual loss. Here they replace the MSE (Mean square error) based content loss with a loss calculated on features maps of the VGG network, which are more invariant to changes in pixel space. MSE based laws focus on pixel-to-pixel comparison, they don't focus on perceptual difference.

Perceptual Loss = Adversarial Loss + Content Loss

- **Content Loss:** Evaluates the image quality based on its perceptual quality.
- **Adversarial Loss:** Measures how well the generator can generate data that is indistinguishable from real data.

The goal is to ensure perceptual similarity between images, rather than a pixel-by-pixel match. This is why the Mean Square Error (MSE) function was replaced with a perceptual loss function. MSE only considers pixel intensity differences, lacking information about the image's content structure. The new approach involves using a pre-trained model, VGG19, to obtain feature maps from both the generated and ground truth images. By comparing these feature maps, the perceptual difference between the images is assessed, ensuring a more accurate evaluation of similarity.

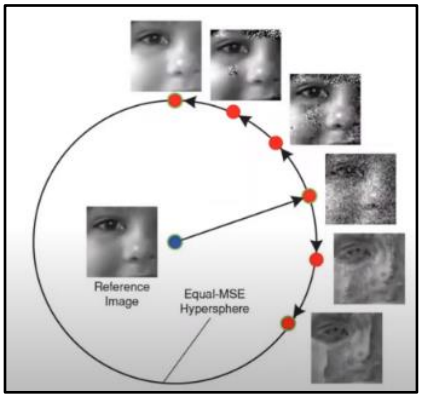


Figure 3: MSE difference between images

Code snippet for Perceptual Loss (Important part)

```
class perceptual_loss(nn.Module):

    def __init__(self, vgg):
        super(perceptual_loss, self).__init__()
        self.normalization_mean = [0.485, 0.456, 0.406]
        self.normalization_std = [0.229, 0.224, 0.225]
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.transform = MeanShift(norm_mean = self.normalization_mean, norm_std =
self.normalization_std).to(self.device)
        self.vgg = vgg
        self.criterion = nn.MSELoss()
    def forward(self, HR, SR, layer = 'relu5_4'):
        ## HR and SR should be normalized [0,1]
        hr = self.transform(HR)
        sr = self.transform(SR)

        hr_feat = getattr(self.vgg(hr), layer)
        sr_feat = getattr(self.vgg(sr), layer)

        return self.criterion(hr_feat, sr_feat), hr_feat, sr_feat
```

This *perceptual_loss* class computes the perceptual loss, which measures the difference in high-level content between the HR and SR images. It does so by extracting features from a specified layer of a pre-trained VGG network and computing the MSE loss between these features, providing a valuable metric for evaluating the quality of super-resolved images.

Code snippet for Total Variation Loss (Important part)

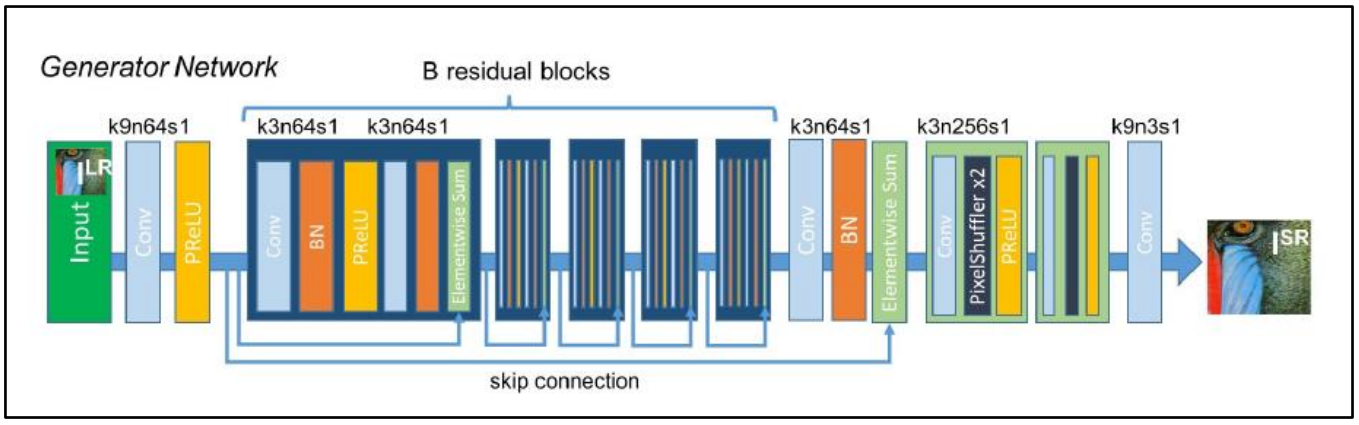
```
class TVLoss(nn.Module):
    def __init__(self, tv_loss_weight=1):
        super(TVLoss, self).__init__()
        self.tv_loss_weight = tv_loss_weight

    def forward(self, x):
        batch_size = x.size()[0]
        h_x = x.size()[2]
```

```
        w_x = x.size()[3]
        count_h = self.tensor_size(x[:, :, 1:, :])
        count_w = self.tensor_size(x[:, :, :, 1:])
        h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
        w_tv = torch.pow((x[:, :, :, 1:] - x[:, :, :, :w_x - 1]), 2).sum()

        return self.tv_loss_weight * 2 * (h_tv / count_h + w_tv / count_w) / batch_size

    @staticmethod
    def tensor_size(t):
        return t.size()[1] * t.size()[2] * t.size()[3]
```

The *TVLoss* class calculates the total variation loss of the input images. This loss encourages smoother images by penalizing sharp intensity transitions. The class allows flexibility in adjusting the influence of TV loss in the overall loss function by providing a customizable weight (*tv_loss_weight*).

3.2.2 Generator

SRGAN requires a dataset comprising pairs of high-resolution and low-resolution images. The generator model takes a low-resolution image as input and is trained to produce a high-resolution image that closely matches the original images. The generator upscales the resolution by a factor of 4.



Initially, the low-resolution input is passed through an initial convolutional layer with a kernel size of 9x9. This layer has 64 feature maps followed by a parametric ReLU layer. Throughout the generator architecture, parametric ReLU is the primary activation function. It was chosen for its effectiveness in tasks involving mapping low-resolution to high-resolution images. The output from the initial P-ReLu block is fed to the subsequent layers.

The generator consists of 16 residual blocks. Each residual block contains two 3x3 convolutional layers with 64 feature maps. Batch normalization follows each convolutional layer, and the P-ReLU layer serves as the activation function. The output of each convolutional layer in the residual block is elementwise summed with the output from the previous step, which involves batch normalization.

After the 16 residual blocks, another convolutional layer follows with a filter size of 3x3 and 64 feature maps, followed by a batch normalization block. The output from this block is elementwise summed with the output from the first P-ReLU block. Subsequently, two 3x3 convolutional layers with 256 feature maps each follow. Each convolutional layer is succeeded by a pixel shuffler block and a P-ReLU block. Finally, there is a convolutional layer with a filter size of 9x9 and 3 feature maps.

Batch normalization is inserted between each layer to stabilize the model's learning process. The pixel shuffler in the generator model plays a crucial role in producing super-resolution images.

This layer increases the image size by 4 times, transforming the feature maps from a low-resolution image to a high-resolution image.

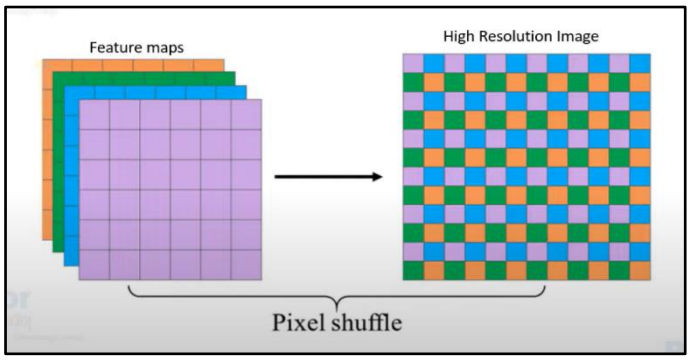


Figure 4: Pixel Shuffler

The Pixel Shuffler is integral in SRGAN's generator, converting channels into height and width, effectively boosting image resolution. By reshaping feature maps, it preserves crucial details, enhancing image quality. This transformation plays a pivotal role in generating high-resolution images from low-resolution inputs.

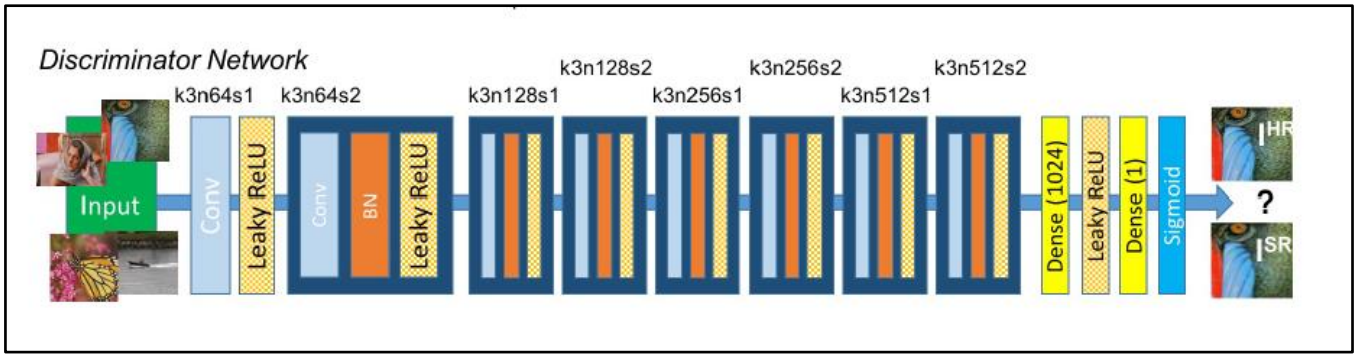
Code snippet for Generator (Important part)

```
class Generator(nn.Module):
    def __init__(self, img_feat=3, n_feats=64, kernel_size=3, num_block=16,
act=nn.PReLU(), scale=4):
        super(Generator, self).__init__()
        self.conv01 = conv(in_channel=img_feat, out_channel=n_feats, kernel_size=9,
BN=False, act=act)
        resblocks = [ResBlock(channels=n_feats, kernel_size=3, act=act) for _ in
range(num_block)]
        self.body = nn.Sequential(*resblocks)
        self.conv02 = conv(in_channel=n_feats, out_channel=n_feats, kernel_size=3,
BN=True, act=None)

        if(scale == 4):
            upsample_blocks = [Upsampler(channel=n_feats, kernel_size=3, scale=2,
act=act) for _ in range(2)]
        else:
            upsample_blocks = [Upsampler(channel=n_feats, kernel_size=3, scale=scale,
act=act)]
        self.tail = nn.Sequential(*upsample_blocks)
        self.last_conv = conv(in_channel=n_feats, out_channel=img_feat, kernel_size=3,
BN=False, act=nn.Tanh())

    def forward(self, x):
        x = self.conv01(x)
        _skip_connection = x

        x = self.body(x)
        x = self.conv02(x)
        feat = x + _skip_connection

        x = self.tail(feat)
        x = self.last_conv(x)

        return x, feat
```

This *Generator* class uses residual blocks and skip connections to learn intricate features from low-resolution inputs and generate high-resolution outputs. The network leverages the power of residual learning to mitigate issues like vanishing gradients during training, enhancing the overall effectiveness of the super-resolution task. The up-sampling blocks help in increasing the resolution of the generated image, making it suitable for tasks like image super-resolution.

3.2.3 Discriminator

Discriminators are currently being trained to distinguish between real high-resolution images and generated high-resolution images from the generator. These discriminators identify disparities and adjust both generator and discriminator weights using backpropagation for optimization. The discriminator functions as an image classifier, determining the authenticity of images fed into it: real high-resolution images from the dataset and high-resolution images generated by the generator.



Two inputs are fed into the discriminator: the first being the output from the generator, and the second being the real high-resolution image sourced from the dataset. The discriminator block consists of an initial convolutional layer with 64 filters (3x3 kernel, stride 1, same padding) and leaky ReLU activation (α=0.2). Following this, there are repeating blocks of convolutional layers, batch normalization, and leaky ReLU activation.

This block comprises 7 convolutional layers. The first layer has a kernel size of 3x3, 64 feature maps, and a stride count of 2. Following this, there are 2 convolutional layers with the same kernel size, where the feature maps double to 128. The stride count is 1 for the first layer and 2 for the second. Similarly, the subsequent 4 layers follow the same pattern: a kernel size of 3x3, 256 and 512 feature maps in pairs, with stride counts of 1 and 2 for each pair, respectively.

The output from these convolutional blocks is passed through a dense layer with 1024 neurons, followed by leaky ReLU activation, and another dense layer with a sigmoid activation function to obtain the probability of classification for real or fake images.

Code snippet for Discriminator (Important part)

```
class Discriminator(nn.Module):

    def __init__(self, img_feat=3, n_feats=64, kernel_size=3, act=nn.LeakyReLU(inplace =
True), num_of_block=3, patch_size=96):
        super(Discriminator, self).__init__()
        self.act = act

        self.conv01 = conv(in_channel=img_feat, out_channel=n_feats, kernel_size=3,
BN=False, act=self.act)
        self.conv02 = conv(in_channel=n_feats, out_channel=n_feats, kernel_size=3,
BN=False, act=self.act, stride=2)

        body = [discrim_block(in_feats=n_feats*(2**i), out_feats=n_feats*(2**(i+1)),
kernel_size=3, act=self.act) for i in range(num_of_block)]
```

```
        self.body = nn.Sequential(*body)

        self.linear_size = ((patch_size//(2**(num_of_block+1)))**2)*(n_feats*(2**
num_of_block))

        tail = []

        tail.append(nn.Linear(self.linear_size, 1024))
        tail.append(self.act)
        tail.append(nn.Linear(1024, 1))
        tail.append(nn.Sigmoid())

        self.tail = nn.Sequential(*tail)

    def forward(self, x):

        x = self.conv01(x)
        x = self.conv02(x)
        x = self.body(x)
        x = x.view(-1, self.linear_size)
        x = self.tail(x)

        return x
```

This *Discriminator* class defines a network that takes an image as input and produces a single probability value indicating the authenticity of the input image. This network is an essential component of a GAN, providing feedback to the generator about the quality of the generated images.

3.2.4 How it works?

During training, both networks progress simultaneously, enhancing their performance progressively. However, post-training, only the generator is essential as it generates the desired high-resolution images autonomously. The discriminator's role is confined to the training period, aiding the model in learning. After successful training, the focus shifts entirely to utilizing the trained generator for high-quality image generation, making the discriminator unnecessary in the operational phase.

3.2.5 Modes of Operation

Within the mode file, three distinct modes govern its functionality. The first mode, termed "train mode," is dedicated to training the model, allowing it to learn and improve its performance over iterations. The second mode, known as "test mode," serves the purpose of evaluating the model's performance on a given dataset. Finally, there is the "test only mode," specifically designed for generating output from the test dataset.

Upon completing the training process, the next step involves testing the trained model. There are two approaches for this evaluation. The first method involves solely obtaining output images from the test dataset, which is facilitated by the "test only" mode. This mode is ideal when the sole requirement is generating images without any additional metrics.

On the other hand, the "test mode" offers a more comprehensive evaluation. It not only generates output images but also calculates the Peak Signal-to-Noise Ratio (PSNR) values for these images. PSNR values provide a quantitative measure of the quality of the generated images concerning the original high-resolution images. Thus, the choice between these modes depends on the specific evaluation needs, allowing for flexibility in assessing the model's performance.

4. Results comparing to a state-of-the-art model.

In the realm of Image Super-Resolution on the challenging DIV2K dataset with 4x upscaling, current advancements are marked by the pioneering work of SRGAN, specifically the methods ESRGAN and RankSRGAN, outlined in the paper **"*Implicit Diffusion Models for Continuous Super-Resolution*"** by *Sicheng Gao, Xuhui Liu, Bohan Zeng, Sheng Xu, Yanjing Li, Xiaoyan Luo, Jianzhuang Liu, XianTong Zhen, and Baochang Zhang*, presented at CVPR 2023. These cutting-edge approaches have set new benchmarks for the field with SRGAN.

According to the comprehensive analysis available on ***paperswithcode.com***, the PSNR (Peak Signal-to-Noise Ratio) values achieved by these methods on the DIV2K dataset stand at an impressive 26.22 for ESRGAN and 26.55 for RankSRGAN. These values signify the exceptional quality of super-resolved images produced by these state-of-the-art models.

As per the research paper we are referencing, it's noted that "Highest PSNR does not necessarily reflect the perceptually better Super Resolution result." Despite this insight, we have chosen PSNR as our comparative metric to assess our model against state-of-the-art counterparts.

In this context, our developed model, which follows the SRGAN framework, has been rigorously evaluated. The results reveal an average PSNR value of 10.89 on the same DIV2K dataset for Image Super-Resolution. While our model demonstrates promising progress, it is evident that there is a considerable performance gap when compared to the benchmark PSNR values achieved by ESRGAN and RankSRGAN.

These findings underscore the challenges that persist in the domain of Image Super-Resolution and highlight the need for further research and innovation to bridge this performance divide and advance the state of the art in this field.

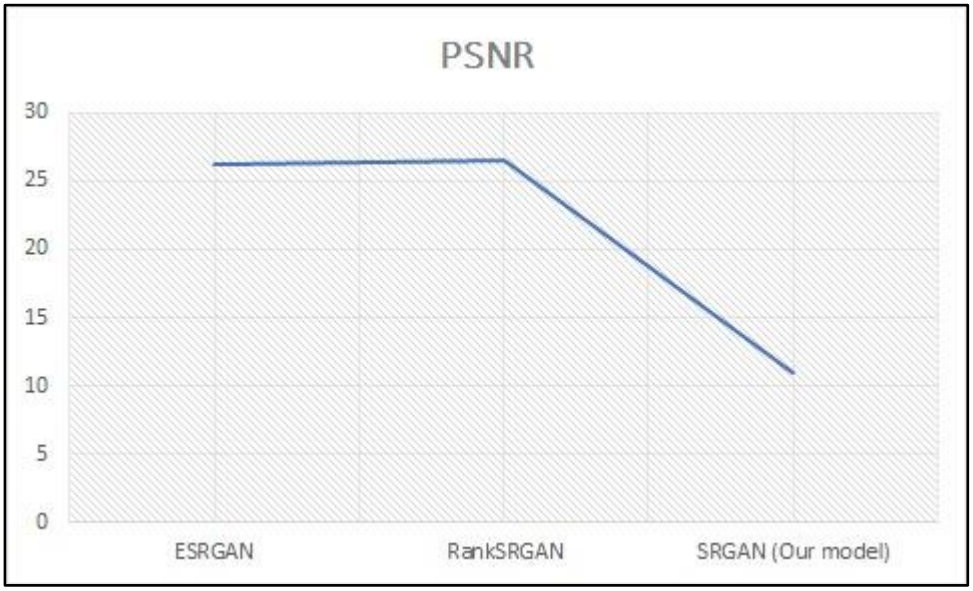


Figure 5: PSNR Values for Different methods

5. Visual Results

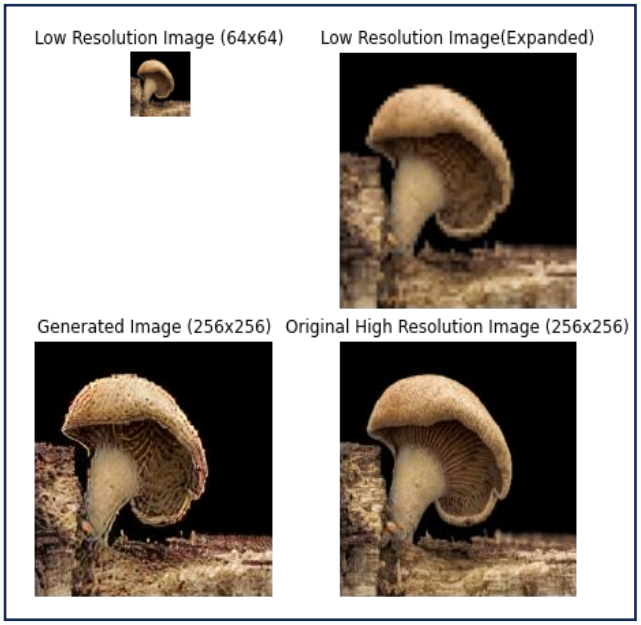


Figure 6: DIV2K Data set image result

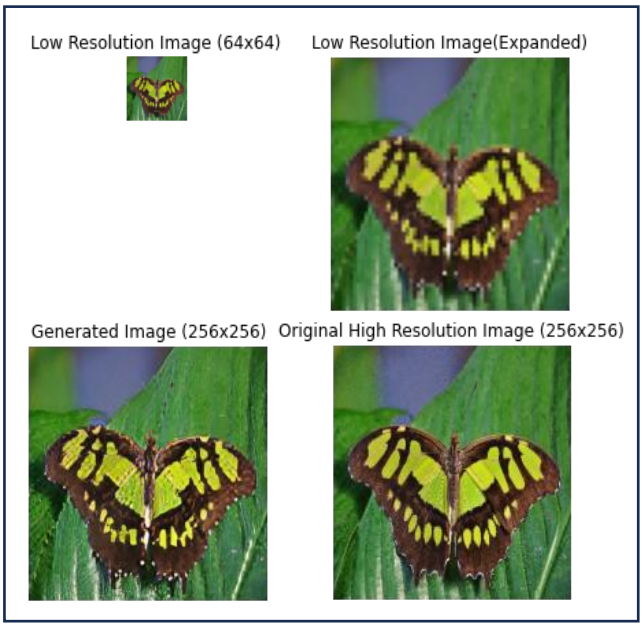


Figure 7: DIV2K Data set image result

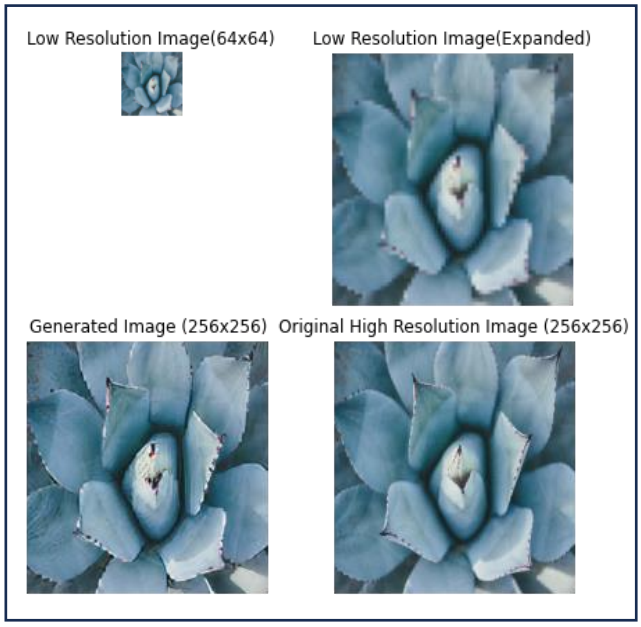


Figure 8: DIV2K Data set image result

Figure 9: Custom image result

Remarks:

The initial dataset comprises the first three images, all sourced from the DIV2K dataset. To start, these images were resized to 64x64 pixels, serving as input images. Subsequently, in the results, they underwent an upscale of 4 times, resulting in a high-resolution of 256x256 pixels. The last image is a custom addition, following the same procedure as the previous three.

The visual outcomes were indeed satisfactory, showcasing the potential of our model. However, our progress was constrained by technical limitations, preventing us from pushing the training to its full potential. With greater training epochs, there's a strong possibility of achieving even more impressive results, surpassing the current level of quality and visual fidelity.

6. Discussion

1. **Challenge:** Limited training epochs might hinder the model from reaching its optimal performance, especially in capturing intricate textures and diverse patterns.
   Solution: Extending Training Iterations.
   Long-term training allows the model to explore the dataset comprehensively, refining its understanding over time. Continuous monitoring of loss functions becomes crucial to identify the point of convergence, addressing the challenge of premature convergence. Patience in training permits the model to learn nuanced patterns and textures, overcoming the challenge of limited exploration within a fixed epoch count.

2. **Challenge:** Sole reliance on PSNR values might overlook finer image details, especially in complex regions. Evaluating based solely on PSNR might not capture the perceptual differences significant to human observers.
   Solution: Enhancing Loss Function Complexity.
   Integrating advanced metrics like SSIM and LPIPS alongside PSNR provides a nuanced evaluation, addressing the challenge of oversimplification in assessment. These metrics offer a holistic view, considering both structural similarity and perceptual differences. Combining metrics with diverse focuses enhances the evaluation process, mitigating the challenge of overlooking intricate details and ensuring a comprehensive assessment.

3. **Challenge:** Artifacts often arise from adversarial training instability, particularly in regions like human faces or intricate backgrounds.
   Solution: Exploring Advanced GAN Variants.
   Advanced GAN variants such as WGAN and LSGAN provide stability and reduce mode collapse, addressing the challenge of unstable training. These variants offer smoother gradient flows, minimizing artifacts in complex areas. Implementation of stable GAN variants improves training robustness, tackling the challenge of artifacts, especially in intricate regions of images.

4. **Challenge:** Simple models might fail to capture intricate textures like fur or detailed patterns, leading to a loss of essential image details.
   Solution: Implementing Complex Model Architectures.
   Complex architectures like DenseNet facilitate the modeling of intricate textures, addressing the challenge of oversimplification. Skip connections in these architectures enable the flow of gradients, ensuring the model captures intricate details effectively. Purposeful complexity in model design ensures the preservation of intricate textures, overcoming the challenge of oversimplification and enhancing the model's ability to represent detailed features.

5. **Challenge:** Artifacts often persist due to the model's inability to focus on specific details within the image, especially in densely textured regions.
   Solution: Precision through Attention Mechanisms.
   Attention mechanisms enable precise focus, addressing the challenge of generalized processing. Self-attention modules allow detailed scrutiny, ensuring artifact-free representations in intricate areas. Multi-scale attention mechanisms enable the model to focus on both macroscopic and microscopic details, overcoming the challenge of overlooking intricate textures and refining the output's precision.

7. Acknowledgement of Resources Used

Initially, our model training relied on the "AMD RYZEN 7 5800H" CPU, processing 800 low-resolution and high-resolution images from the DIV2K dataset across 2000 pre-training epochs and 1000 fine-training epochs. However, this approach proved time-consuming, taking nearly a day to complete.

Recognizing the need for efficiency, we transitioned to leveraging GPU power. By harnessing the capabilities of the NVIDIA GeForce RTX 3060 Laptop GPU (6GB) through the "CUDA 11.8" library, our model training time dramatically reduced to just 5 hours and 33 minutes. Python 3.11.5, within the Anaconda Navigator with Jupyter Notebook, played a pivotal role in this accelerated training process.

8. References

- *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*
- *Implicit Diffusion Models for Continuous Super-Resolution*
- *NTIRE 2018 Challenge on Single Image Super-Resolution: Methods and Results*
- *State-of-the-art comparisons*
- *DIV2K dataset*

9. Source Files

GitHub Link : ***Super resolution (NTIRE18)***