Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

Presented by:

Maryam Bandali - 1861486

Abdolhadi Rezaei- 1837982

SAPIENZA
UNIVERSITÀ DI ROMA

Elective in AI 1

Table of Content

Project Definition:

Implementation of a super-resolution generative adversarial network (SRGAN) using a deep residual network (ResNet) in which a novel perceptual loss using high-level feature maps of the VGG network has been employed

One central unsolved problem: How do we recover the finer texture details when we super-resolve at large upscaling factors?

- Minimizing the mean squared reconstruction error
  - high peak signal-to-noise ratios
  - Lacking high-frequency details
  - perceptually unsatisfying
- generative adversarial network (GAN) for image super-resolution (SR)
  - perceptual loss function which consists of:
    - ***adversarial loss:*** *pushes our solution to the natural image manifold*
    - ***content loss:*** *motivated by perceptual similarity instead of similarity in pixel space*

Introduction

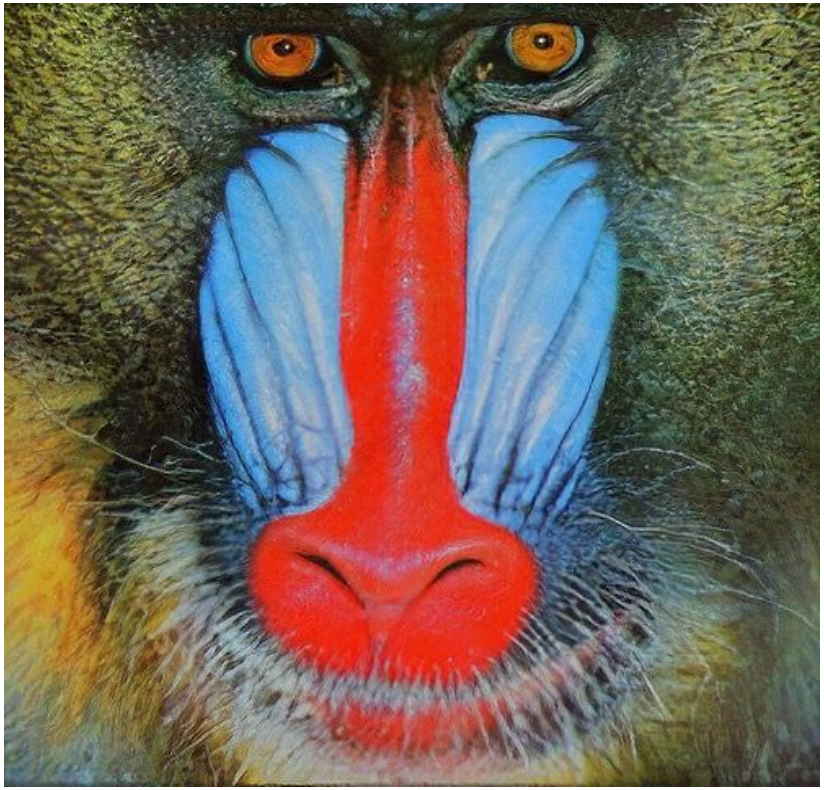
1) Super-resolution problem: estimating a high-resolution (HR) image from its low-resolution (LR)
2) Texture detail in the reconstructed SR images is typically absent
3) The optimization target of supervised SR algorithms is to minimize the mean squared error (MSE) between the recovered HR image and the ground truth
4) Minimizing MSE = Maximizing the peak signal-to-noise ratio (PSNR)
5) The ability of MSE and PSNR to capture high texture detail is very limited
6) Perceptual difference: The recovered image is not photo-realistic

bicubic interpolation
(21.59dB/0.6423)

deep residual network
optimized for MSE
(23.53dB/0.7832)

deep residual generative adversarial
network optimized for a loss more
sensitive to human perception
(21.15dB/0.6868)

original HR image



Super-resolved image
using SRGAN

Original

Some explanations

Image super-resolution

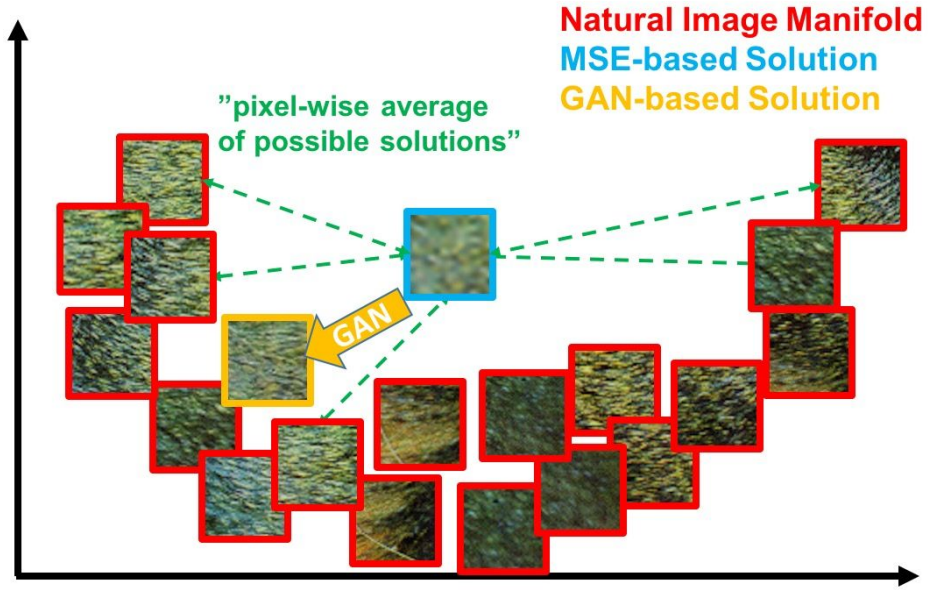
1) Here, the focus is on single image super-resolution (SISR)
2) Prediction-based methods
3) Approaches based on establishing a mapping between low- and high-resolution image information and usually rely on training data.
4) Approaches to reconstruct realistic texture detail while avoiding edge artifacts
5) Neighborhood embedding approaches
6) Convolutional neural network (CNN): excellent performance!

Design of convolutional neural networks

1) Deeper network architectures
   a) Difficult to train
   b) However, increase the network's accuracy (modeling mappings of very high complexity)
2) Training efficiently these deeper networks architecture
   a) Batch normalization
3) Deeper network architectures increase performance for SISR
4) Another powerful design choice that eases the training of deep CNNs
   a) residual blocks
   b) skip-connections
5) Learning upscaling filters is beneficial
   a) Accuracy
   b) speed

Loss functions

1) Minimizing MSE: finding pixel-wise averages
   a) overly-smooth
   b) poor perceptual quality



Paper Contributions
&
Implementation

The proposed approach

The GAN procedure encourages the reconstructions to move towards regions of the search space with high probability of containing photo-realistic images and thus closer to the natural image manifold:

Our goal: To implement the first very deep ResNet architecture using the concept of GANs to form a perceptual loss function for photo-realistic SISR

1) 16 blocks deep ResNet (SRResNet) optimized for MSE
2) Propose SRGAN which is a GAN-based network optimized for a new perceptual loss
    a) Replace the MSE-based content loss with a loss calculated on feature maps of the VGG network

Method

1) In SISR the aim:
   a) to estimate a high-resolution, super-resolved image from a low-resolution input image
2) Our ultimate goal:
   a) to train a generating function G that estimates for a given LR input image its corresponding HR counterpart.
3) We train a generator network as a feed-forward CNN parametrized by θ . Here $\theta = \{W^{1:L} ; b^{1:L}\}$ denotes the weights and biases of a L-layer deep network and is obtained by optimizing a SR-specific loss function
4) We implement a perceptual loss as a weighted combination of several loss components that model distinct desirable characteristics of the recovered SR image.

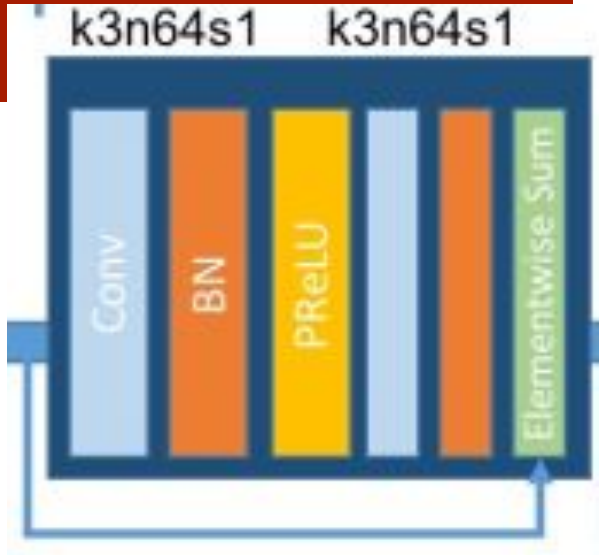
Adversarial network architecture

1) we implement a discriminator network which we optimize in an alternating manner along with the generator network to solve the adversarial min-max problem
2) We train a generative model G with the goal of fooling a discriminator D that is trained to distinguish super-resolved images from real images
3) Our generator learn to create images that are highly similar to real images
    a) difficult to classify by discriminator
    b) In contrast to SR solutions obtained by MSE

Residual Block

The residual block is consists of 2 layers, one non-linearity in the middle which they use ReLU and the x connection is the identity. This method allows us to design deeper networks in order to deal with Much complicated problems and tasks.

```
def residual_block(self,input_layer):
        x = Conv2D(filters = 64, kernel_size = 3, padding =
'same')(input_layer)
        x = BatchNormalization(momentum=0.8)(x)
        x = PReLU()(x)
        x = Conv2D(filters = 64, kernel_size = 3, padding =
'same')(x)
        x = BatchNormalization(momentum=0.8)(x)
        return Add()([input_layer,x])
```



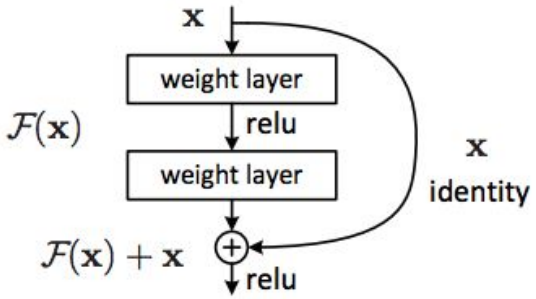
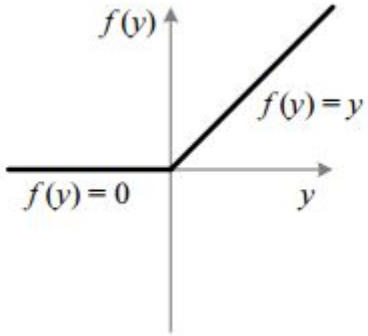


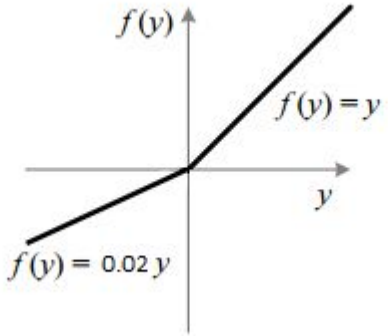
Figure 2. Residual learning: a building block.

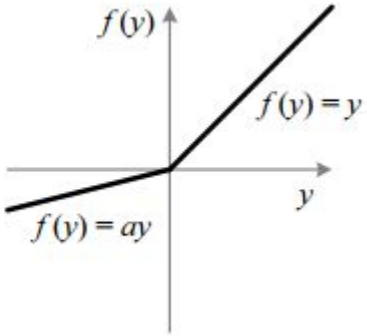
Generator network architecture

1) At the core of our very deep generator network G, there are following components:
   a) B residual blocks with identical layout
      i) Two convolutional layers with small 3×3 kernels and 64 feature maps followed batch-normalization layers and ParametricReLU as the activation function

ReLU







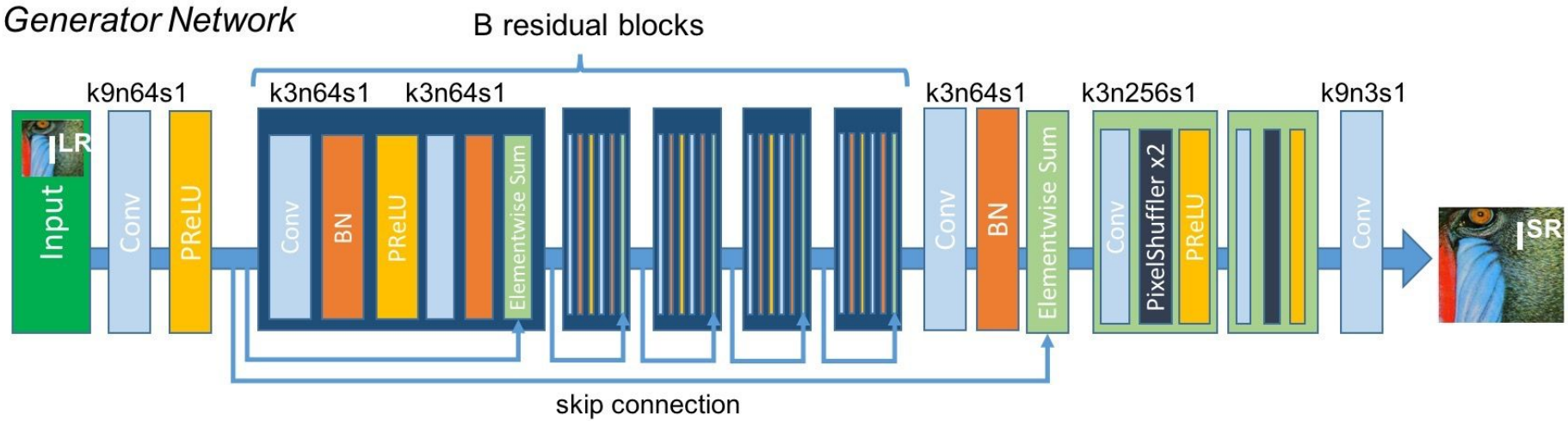
$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

- if a□=0, f becomes ReLU
- if a□>0, f becomes leaky ReLU
- if a□ is a learnable parameter, f becomes PReLU

Batch Normalization

1) Normalize output from activation function.
   a) z= (x-m)/s
2) Multiply normalized output by arbitrary parameter.
   a) z*g
3) Add arbitrary parameter , b , to resulting product.
   a) (z*g) + b

Generator Architecture



Implementation of Generator

```
def build_generator(self,opti_generator,n_blocks = 16):
        input_layer = Input( self.shape_low_reso)
        first_layer = Conv2D(filters =  64, kernel_size = 9,
                             padding =  'same')(input_layer)
        first_layer = PReLU()(first_layer)

        residual_blocks =  self.residual_block(first_layer)

        for _ in range(n_blocks-1):
            residual_blocks =
self.residual_block(residual_blocks)

        output_residual = Conv2D(filters =  64, kernel_size
= 3, padding =  'same')(residual_blocks)
        output_residual =
BatchNormalization(momentum= 0.8)(output_residual)
         output_residual =
Add()([output_residual,first_layer])
        upsample_layer
=self.Upsample_Block(output_residual)
```

```
for _ in range(self.upscale_factor// 2-1):
    upsample_layer =
self.Upsample_Block(upsample_layer)
gen_output = Conv2D(filters =  3, kernel_size =  9,
padding = 'same', activation =
'tanh')(upsample_layer)
gen_model = Model(inputs = input_layer, outputs =
gen_output)
gen_model.compile(loss = 'binary_crossentropy' ,
optimizer = opti_generator)
return gen_model

def Upsample_Block(self,x_in):
        x = Conv2D(filters = 256, kernel_size=3,
padding='same')(x_in)
        x = self.SubpixelConv2D(2)(x)
        return PReLU()(x)
def SubpixelConv2D(self,scale):
        return Lambda(lambda x:
tf.nn.depth_to_space(x, scale))
```

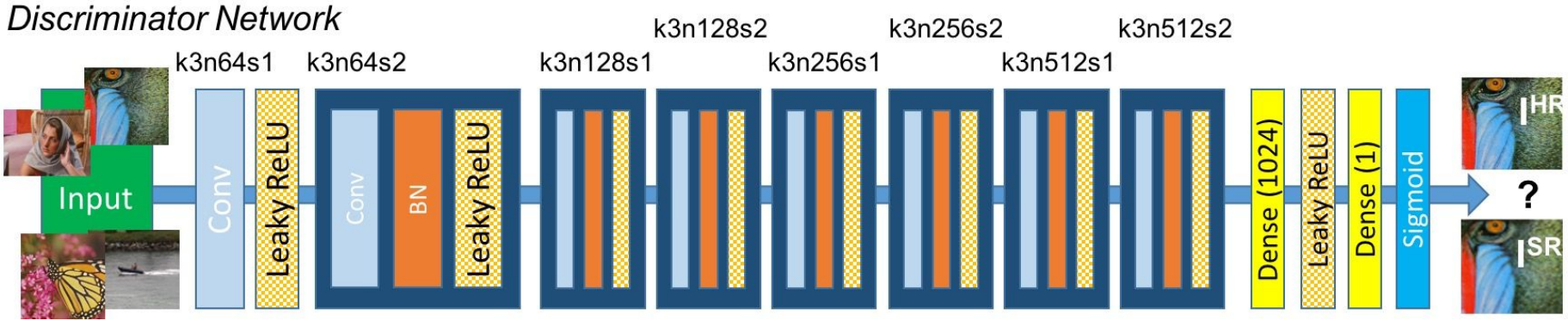
Discriminator network architecture

To discriminate real HR images from generated SR samples we train a discriminator network. The architecture is as follows:

1) use LeakyReLU activation ($\alpha = 0.2$)
2) avoid max-pooling throughout the network
3) It contains eight convolutional layers with an increasing number of 3 × 3 filter kernels, increasing by a factor of 2 from 64 to 512 kernels as in the VGG network

   (Strided convolutions are used to reduce the image resolution each time the number of features is doubled)
4) two dense layers
5) final sigmoid activation function to obtain a probability for sample classification

Implementation of Discriminator



Implementation of Discriminator

```
def build_discriminator (self,opti_discriminator ,n_blocks =
3, n_filters = 64):
        input_layer = Input( self.shape_high_reso)
        discriminator_blocks =
self.disc_block(input_layer,n_filters, False)
        discriminator_blocks =
self.disc_block(discriminator_blocks, n_filters =  128)
        discriminator_blocks =
self.disc_block(discriminator_blocks, n_filters =  256)
        discriminator_blocks =
self.disc_block(discriminator_blocks, n_filters =  512)
        f_layer = Dense(units =  1024)(discriminator_blocks)
        f_layer = LeakyReLU(alpha= 0.2)(f_layer)
        dis_output = Dense(units =  1, activation =
'sigmoid')(f_layer)
        disc_model = Model(inputs = input_layer, outputs =
dis_output)
        disc_model. compile(loss = 'mse', optimizer =
opti_discriminator, metrics = [ 'accuracy'])
        return disc_model
```

```
def disc_block(self,layer, n_filters,
batch_norm = True):
        x = Conv2D(filters = n_filters,
kernel_size = 3, padding = 'same')(layer)
        if batch_norm:
            x =
BatchNormalization(momentum=0.8)(x)
        x = LeakyReLU(alpha=0.2)(x)
        x = Conv2D(filters = n_filters,
kernel_size = 3,
                   strides=2, padding =
'same')(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = LeakyReLU(alpha=0.2)(x)
        return x
```

Perceptual loss function

The loss function defined here is:

1) a loss function that assesses a solution with respect to perceptually relevant characteristics
2) the weighted sum of a content loss and an adversarial loss component

Possible choices for content loss

1) The pixel-wise MSE
    a) pros:
        i) Achieving high PSNR (peak signal-to-noise ratio)
    b) cons:
        i) perceptually unsatisfying solutions with overly smooth textures
2) VGG loss
    a) closer to perceptual similarity
    b) based on the ReLU activation layers of the pre-trained 19 layer VGG network
    c) The euclidean distance between the feature representations of a reconstructed image and the reference image

Content Loss

```
def bulid_vgg(self):
        vgg = VGG19(weights = "imagenet")
        vgg.outputs = [vgg.layers[9].output]
        img = Input(shape = self.shape_high_reso)
        img_features = vgg(img)
        vgg_model = Model(img, img_features)
        for layer in vgg_model.layers:
          layer.trainable = False
        vgg_model.compile(loss = 'mse', optimizer =
Adam(0.0002,0.5),
                          metrics =['acc'])
        return vgg_model
```

To compile VGG19, use **mse** as the loss, **accuracy** as the metrics, and **Adam_optimizer** as the optimizer. Before compiling the network, **disable** the training, as we don't want to train the VGG19 network.

SRGAN Architecture

```
def build_srgan(self,optimizer):
        dis_input = Input(self.shape_high_reso)
        gen_input = Input(self.shape_low_reso)
        generated_high_reso = self.generator(gen_input)#1
        generated_features =
self.vgg(generated_high_reso)#2
        generator_valid =
self.discriminator(generated_high_reso)#3
        gan_model = Model(inputs = [gen_input,
dis_input],outputs = [generator_valid,
generated_features])#4
        for l in
gan_model.layers[-1].layers[-1].layers:
          l.trainable=False #5
        gan_model.compile(loss =
['binary_crossentropy','mse'], loss_weights =
[1e-2,1], optimizer = 'adam')
        return gan_model
```

1) generate fake high resolution images using the generator network
2) extract the features of the fake generated images using the VGG19 network
3) pass the fake images to the discriminator network Use the discriminator network to get the probabilities of the generated high resolution fake images.
4) Finally, create a Keras model, which will be our adversarial model
5) we are making the discriminator network non-trainable because we don't want to train the discriminator network while we train the generator network

We have now successfully implemented the networks. Next, we train the network on the dataset.

Experiments
&
Results

Train the Network

```
model_srgan.train(500, save_interval=50
,batch_size=16)
```

Training the SRGAN network is a two-step process. In the first step, we train the discriminator network. In the second step, we train the adversarial network, which eventually trains the generator network.

Experiment

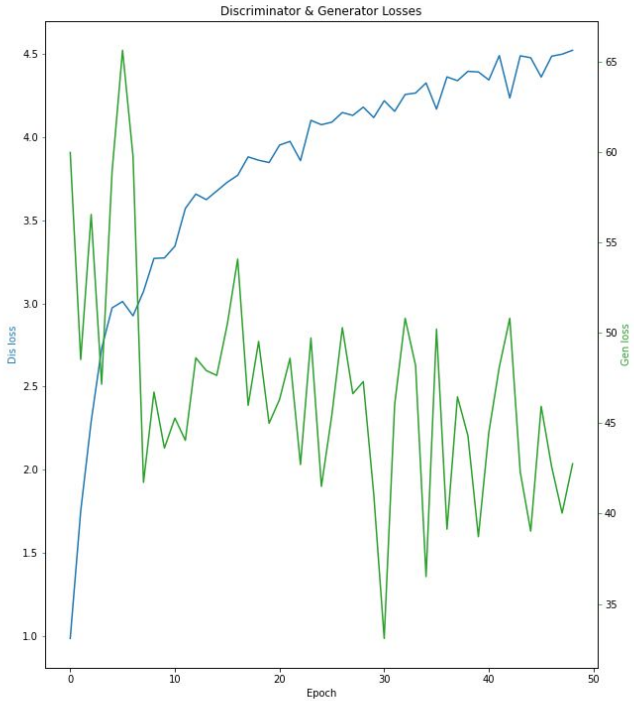
Loss after 50 epochs

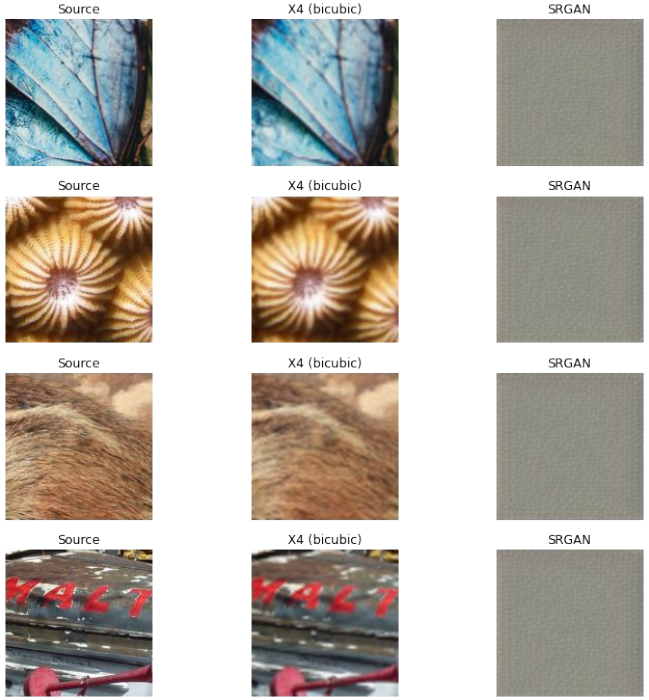
Discriminator Loss:

0.9843276

Generator Loss:

59.97525





Experiment

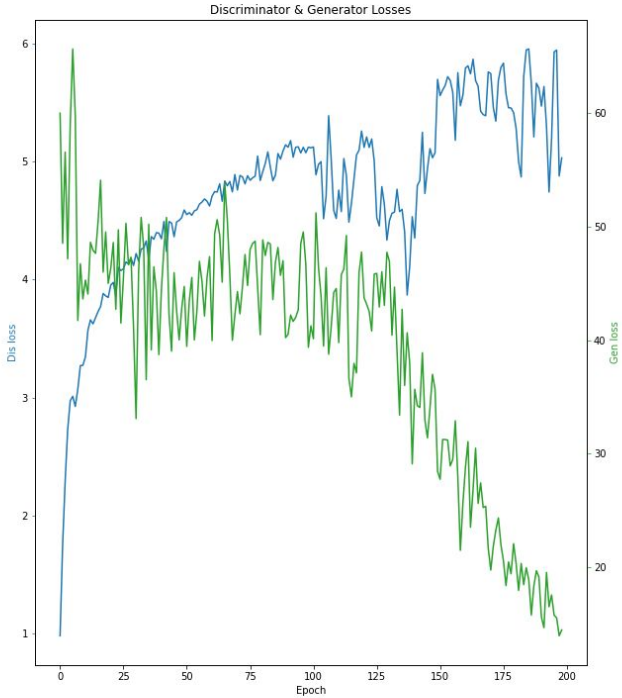
Loss after 200 epochs

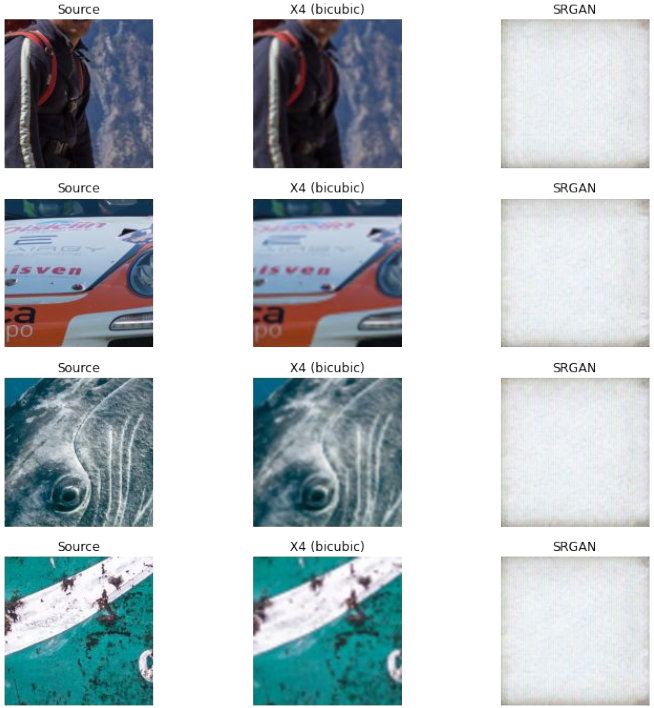
Discriminator Loss:

5.0300584

Generator Loss:

14.499729





Experiment

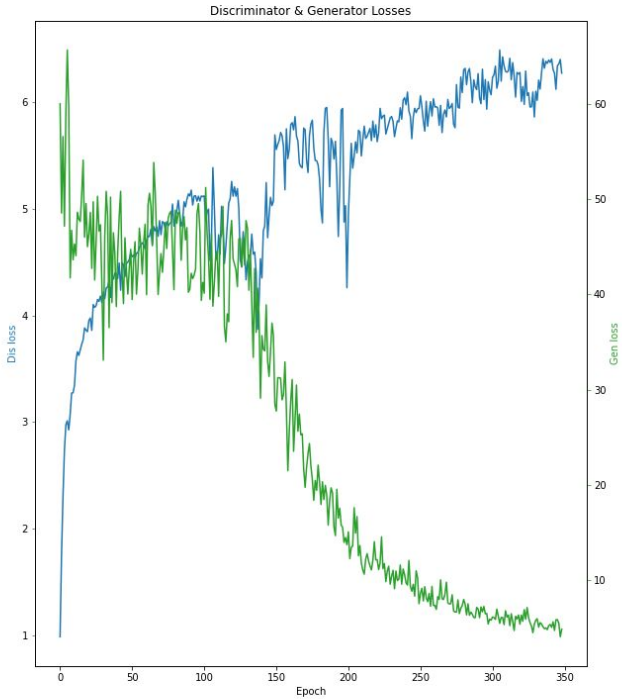
Loss after 350 epochs

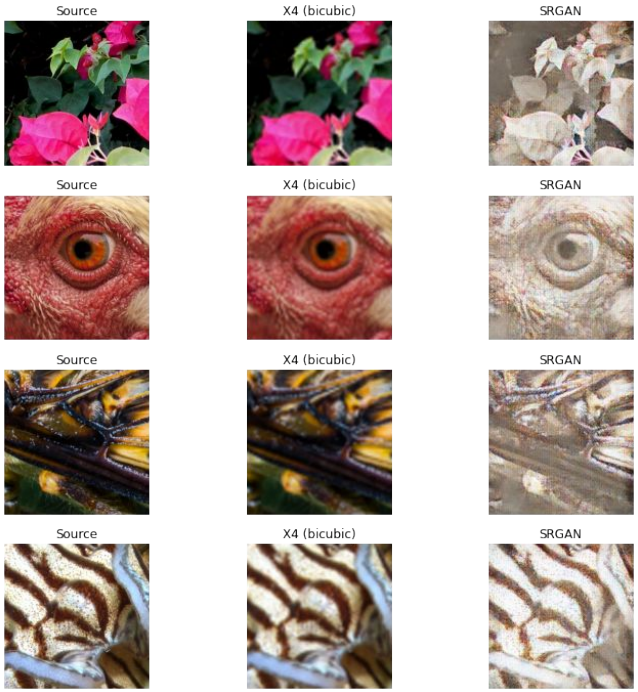
Discriminator Loss:

6.2733507

Generator Loss:

4.888229





Experiment

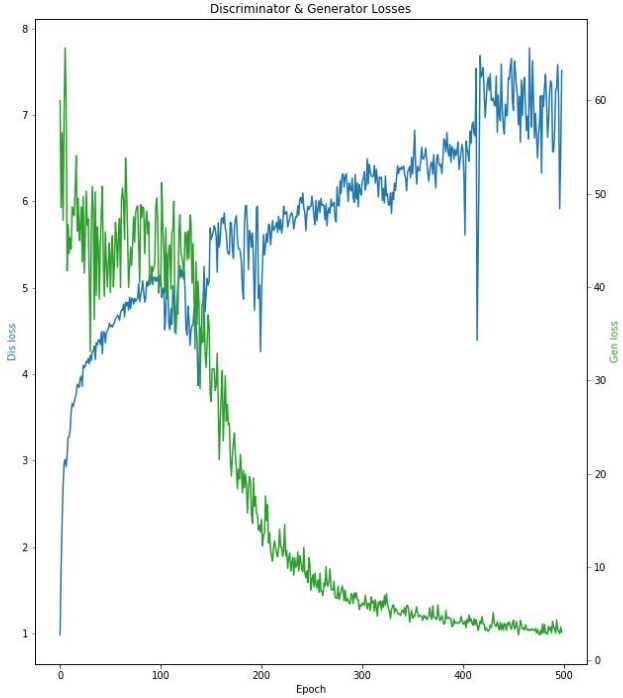
Loss after 500 epochs

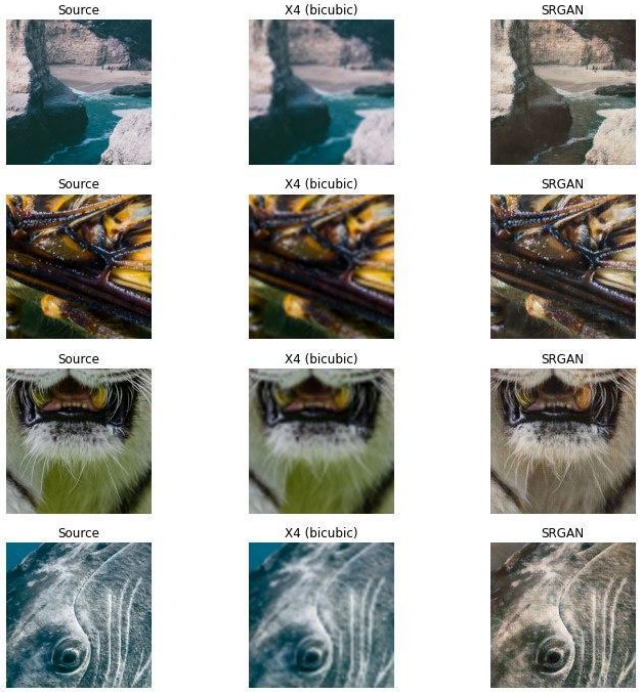
Discriminator Loss:

7.5123367

Generator Loss:

3.0429573





Experiment

```
os.chdir('/content/drive/My Drive/GAN/models/train-500epoch-saturday')
model_srgan.srgan.load_weights('srgan_weights_epoch_500.h5')

os.chdir('/content/drive/My Drive/GAN/models/train-500epoch-new')

model_srgan.train(500, save_interval=50 ,batch_size=16)
```

Experiment(second train)

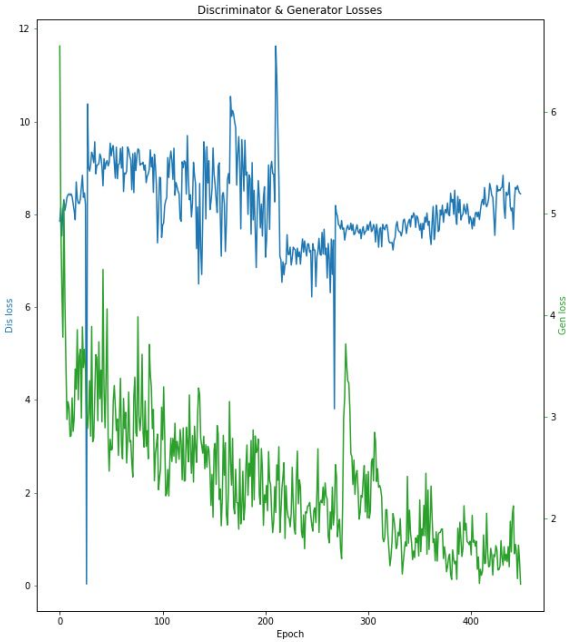
Loss after 450 epochs

Discriminator Loss:

8.4432

Generator Loss:

1.3501143



Thanks for your attention!