Computational Intelligence and Deep Learning

Group Project

Fabio Cognata
Luca Di Giacomo
Giovanni Paolini

Introduction

SRGAN

Nowadays there are still in use devices which have very low capture quality, such as webcams, security cameras and cheap mobile phones. The subjects captured by these devices are often unrecognizable; the project aims to implement image upscaling without losing resolution, while attempting to improve the quality of the picture by restoring the lost details.

Super-Resolution Generative Adversarial Networks (SRGANs) have emerged as a powerful solution for image upscaling, offering a compelling alternative to traditional techniques. SRGAN is capable of recovering details lost in low-resolution images, restoring fine texture details. With the use of a discriminator which distinguishes between the real High-resolution image and the generated ones it is possible to push the generator to create more realistic images. With the possibility of taking a low resolution image to create an output without previous pre-processing, SRGANs are easy to use even though these models often have complex structures, requiring long training with a high number of epochs. Since the output image is generated from the low resolution one, it is possible to create high-resolution images at different scales.
SRGANs have good generalization capabilities, performing well on unseen data, which is very important in real-world applications.

Denoiser

Another problem which arises from the previous real world context is that images captured by such devices often include various types of noise. To produce a clean image from a noisy input, this project used a Denoiser, which is an architecture built to produce a clear image from a distorted input.

The proposed model is an autoencoder model that uses two VGG19 architectures, one complete VGG19 and another built using only a subset of the layers, combined with many convolutional layers and dilated convolutional layers.

By using a pre-trained VGG19 network for feature extraction the architecture benefits from the learning capabilities of the VGG, which can help the Denoiser with capturing meaningful features from the noisy inputs.
This model can be seen as a multi-layered architecture with upsampling and downsampling operations allowing to capture and preserve the hierarchical features in the image data and to understand both low-level details and high-level structures.
The Denoiser leverages a Skip Connection, in particular an Add operation between the processed output of the first VGG19 and the output of the second VGG19 leading to the recovery of fine details lost during the downsampling process.
The downside of this denoising architecture is that it is not capable of performing denoising operations for different sizes of images.

Dataset

This project exploits a dataset created for super resolution purposes, DIV2K dataset: DIVerse 2K resolution high quality images.

This dataset is split in 3 collections:.

- **Train** dataset contains 800 high-resolution images and 800 low-resolution images at different scaling (x2, x3, x4, x8) which were also processed with different interpolation methods such as the bicubic interpolation.
- **Validation** dataset includes 100 high-resolution images and 100 low-resolution images
- **Test** dataset contains 100 low resolution images.

For the scope of the project, only the High Resolution images are used applying augmentation to it to obtain highly variable train data and ground truths.

Image Preprocessing and Augmentation

In order to train the Architectures with such a small amount of data, some strategies were applied:

1. Shuffling the Train and Test sets
2. Slicing the original images into 256x256 dimension portions randomly
3. Applying some transformations such as flip and rotation
4. Rescaling the 256x256 processed slice into a 64x64 image through bicubic interpolation method
5. In case the Denoiser is in use, adding random Gaussian or Poisson noise to the obtained image

Architectures Implementation

- Conv2D: Two Dimensional convolutional layer which applies a set of filters to input data. Extracts spatial features by sliding the filters across the input matrix and performing multiplications and sums across the elements.
- Batch Normalization: Used to improve the stability and convergence of neural networks, it normalizes each batch during the training. This ensures that the input to each layer remains within a reasonable range. This normalization can also accelerate the overall training process and reduce the number of training iterations required.
- Upsampling2D:Used to increase the spatial dimension of the input, enlarging it by replicating or interpolating the existing values, allowing the upscaling of low-resolution images to high-resolution.
- Dilated Conv2D: With the introduction of a dilation factor to the convolution operation, this layer can control the spacing between the kernel elements or the amount of “holes” in the kernel. Instead of sliding the kernel linearly on the input, elements are placed with gaps between them (Gap sizes are determined by the dilation rate). This process has the effect of expanding the kernel’s receptive field. By having a larger receptive field, the network is able to capture more context from the input. Also by increasing the receptive field without adding a number of parameters in the kernel the result is a larger context but with a lower computational cost compared to traditional convolutions with large

kernels. Dilated convolution can capture multi-scale information without the use of pooling or striding operations.

- VGG19: Architecture which extends the original VGG network, used to capture visual features from input images. Composed of 19 layers, 16 convolutional and 3 fully connected. The convolutional layer makes use of a 3x3 filter with a stride of 1. Convolutional layers are stacked one after another allowing the network to learn a hierarchical representation of increasing complexity.

Functions and Operations

- PRELU : Activation function which extends the traditional ReLU, applies a linear transformation to the input, but with additional learnable parameters.
- Leaky RELU: Activation function which extends the traditional ReLU, addresses the issues of “dead neurons”. Instead of mapping negative values to 0, leaky relu introduces a small non-zero slope for negative inputs to help alleviate the issue of dead neurons.
- Sigmoid: Activation function which maps the input values in a 0 to 1 range.
- Huber Loss: Loss function used in image denoising tasks. Huber Loss is capable of retaining edges and fine details, while being less susceptible to outliers compared to Mean Squared Error loss. Instead of squaring the differences between the denoised image and the ground truth image as in MSE, Huber Loss takes the square root of the sum of the squared differences. This square root operation provides a more robust measurement that penalizes large differences less severely than MSE. The denoising model can focus on reducing the absolute differences between the pixel values of the denoised image and the ground truth image. This loss function is particularly effective in preserving sharp edges and fine details in the denoised output.

```
def huber_loss(delta):
    def loss(y_true, y_pred):
        error = y_true - y_pred
        abs_error = tf.abs(error)
        quadratic_error = tf.square(error)
        condition = tf.less(abs_error, delta)
        return tf.where(condition, quadratic_error / 2, delta *
(abs_error - 0.5 * delta))
    return loss
```

- Add: Element wise operation, which takes multiple inputs and performs element-wise addition, combining information from different layers of the network.

Proposed Architecture

SRGAN Architecture

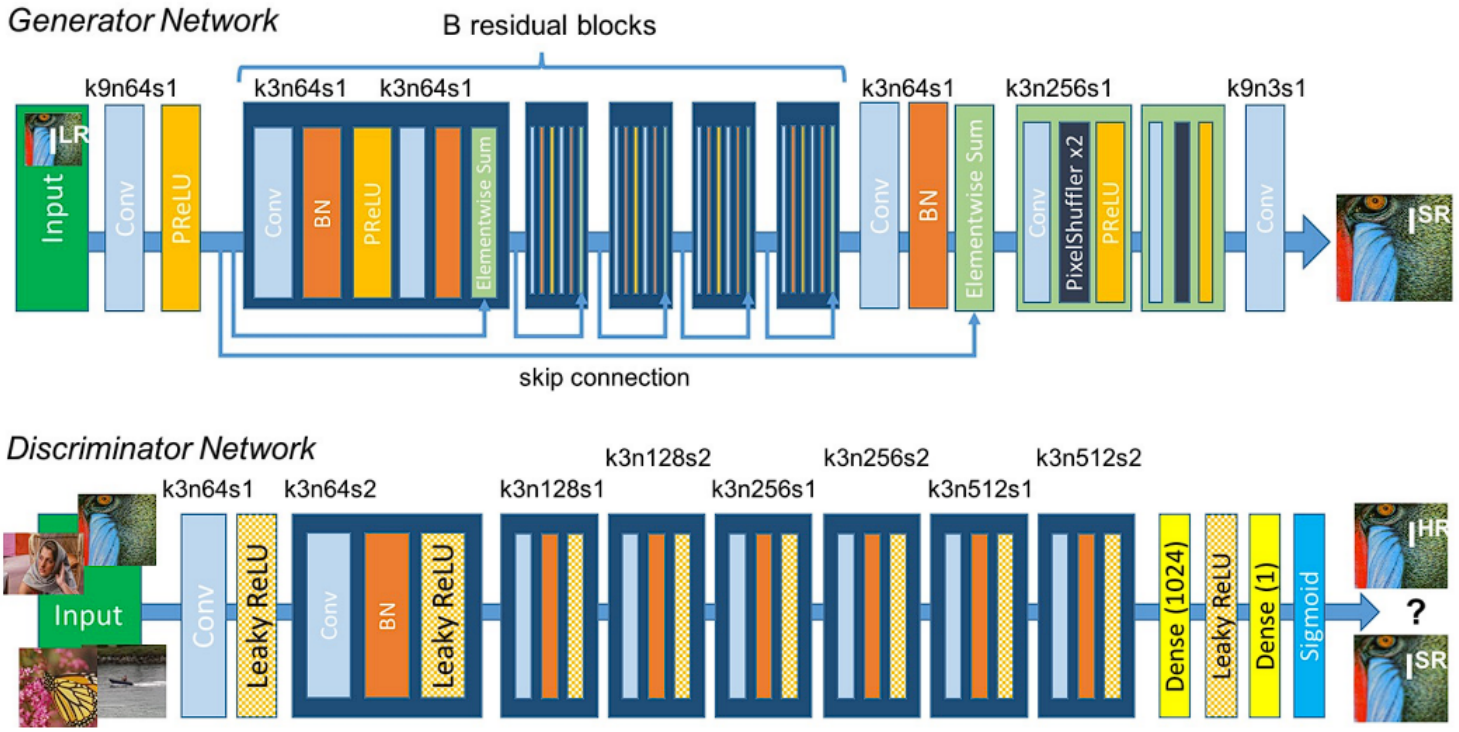
The First Implementation follows the guidelines of the SRGAN Paper , which introduces the architecture below.

The key-points of the Generator architecture are:

- The usage of B Residual blocks , with typically B=16, where each one is composed by a conv2D, a BatchNormalization and a PReLU layers followed by another Conv2D and BatchNormalization Layers.
- The presence of several skip connections
- And that the model levarages two upsampling block builds as in the picture.

While the Discriminator takes advantage of

- Several Discriminator Basic Blocks composed by a Conv2D layer and a BatchNormalization layer with a Leaky Relu activation function.
- A Final Sigmoid Activation Function



Loss Functions

- Content Loss: VGG loss based on the relu activation layers of the pre-trained vgg19 layers, defined as the MSE between the feature maps stopped to the blockX_convY relu-based convolutional block, for our implementation we chose to stop it to the 4th convolution of the 3rd block:

$$l^{SR}_{VGG/i.j} = \frac{1}{W_{i,j}H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

- Adversarial Loss: defined as the maximization of the below formula

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

  or the minimization of a 1-filled tensor with respect to the tensor of probabilities obtained by applying the discriminator to the output of the generator.

$$[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]$$

- Perceptual Loss: It is defined in the paper as the weighted sum between the adversarial and the content losses.

$$l^{SR} = \underbrace{\underbrace{l_{\mathrm{X}}^{SR}}_{\text{content loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

Training Phase:

To handle a complex architecture the chosen optimizer is the Adam Optimizer , particularly useful for huge models trained on big datasets.
The Learning rate and beta have been chosen with an empirical approach , following also the canonical implementations of those types of GAN.

```
common_optimizer = Adam(learning_rate=0.0002, beta_1=0.5)
discriminator_optimizer = Adam(learning_rate=0.0002, beta_1=0.7)
```

```
discriminator.compile(loss='binary_crossentropy',
optimizer=discriminator_optimizer, metrics=['accuracy'])
```

```
adversarial_model.compile(loss=['binary_crossentropy', 'mse'],
loss_weights=[1e-3, 6e-3], optimizer=common_optimizer)
```

to introduce some flexibility on the adversarial loss, we introduced random noise in 1-labeled tensors used as ground truths multiplied by a rescaling factor of 5e-2.

Metrics

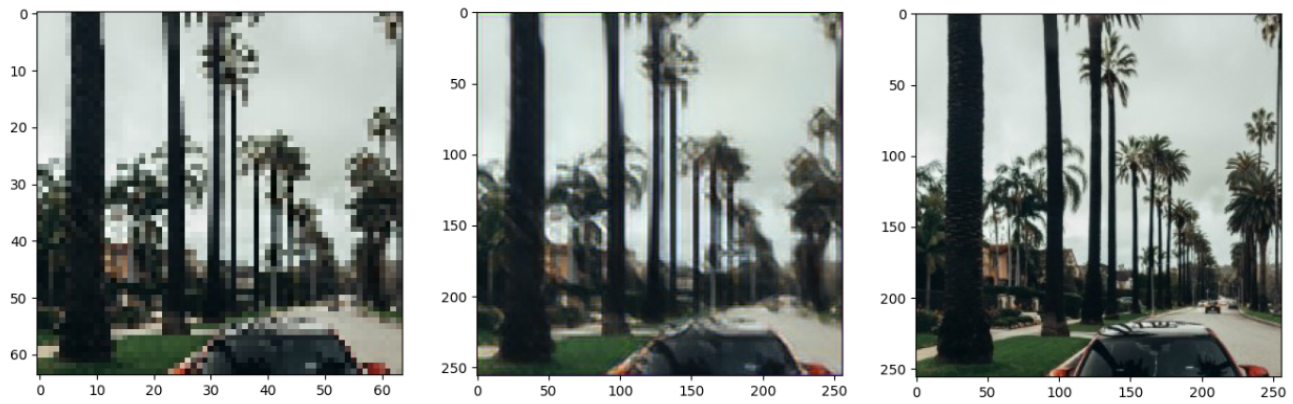
- Peak Signal-to-Noise Ratio (PSNR) : Image quality metric, measures the fidelity of a reconstructed or compressed image compared to its original reference image. Quantifies the level of noise or distortion present in the reconstructed image by calculating the ratio of the maximum power of the original image's pixel (1) over the mean square error (MSE) between the original picture and the generated one.

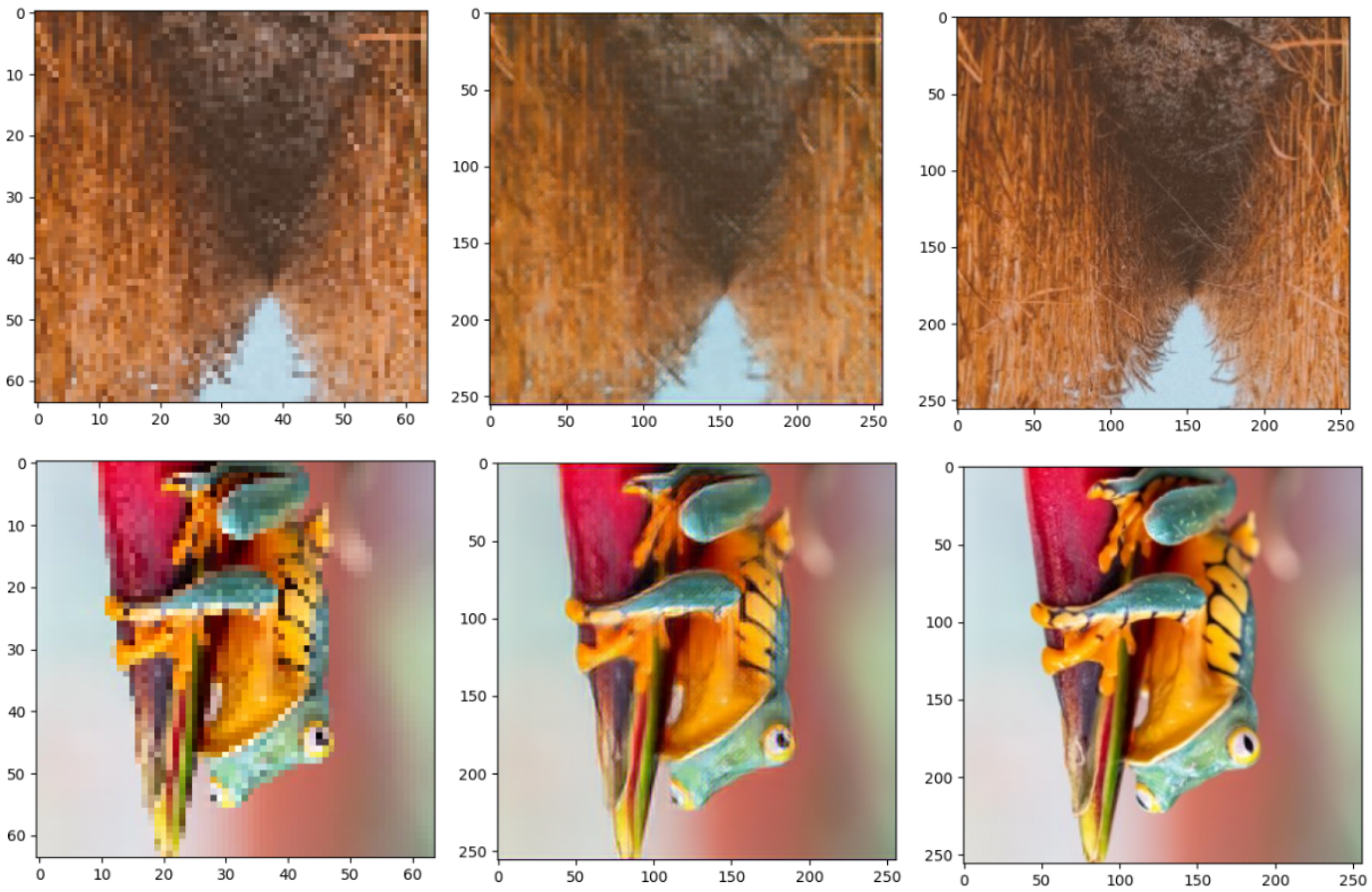
- Structural Similarity Index (SSIM): Image quality metric, evaluates the perceived similarity between the reference image and the generated one. Unlike PSNR, which focuses on pixel-wise difference, SSIM takes also into account the structural information and luminosity to provide a more perceptually relevant quality assessment. To calculate this metric 3 components are used: Luminance, Contrast and structure.

Results

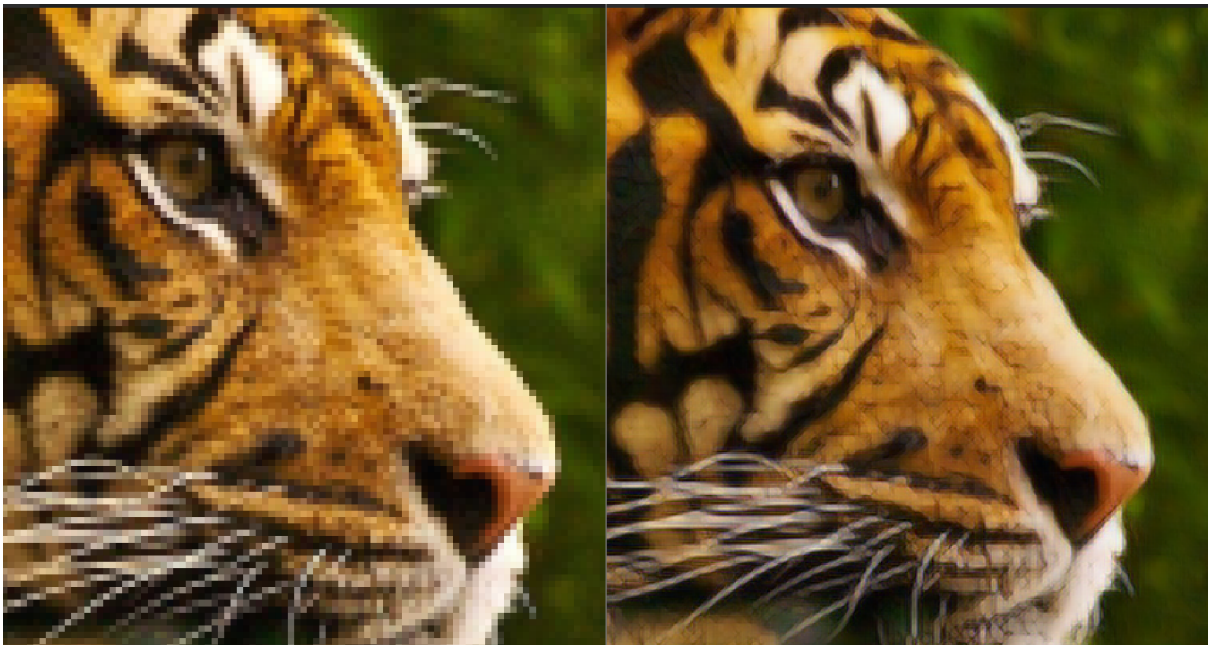
After 500 epochs, the SRGAN has achieved excellent results, compared also with the presented results on the paper relative to a training that lasted approximately 10^5 epochs.

In the following pages, results from the epochs 20, 250 and 500 are presented for different pictures.





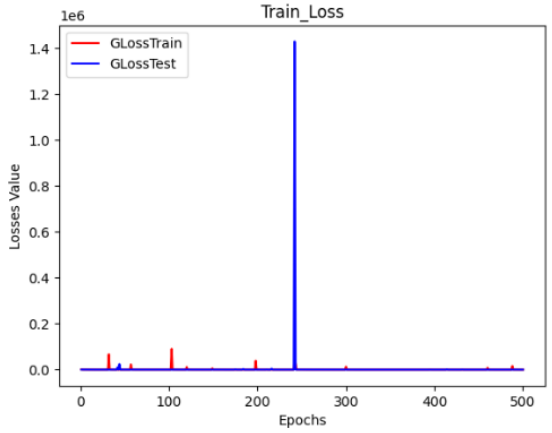
In the following example, the input was a high resolution image instead of a low resolution one, the picture is also shown in more detail in a cropped portion with a zoom of 1000%

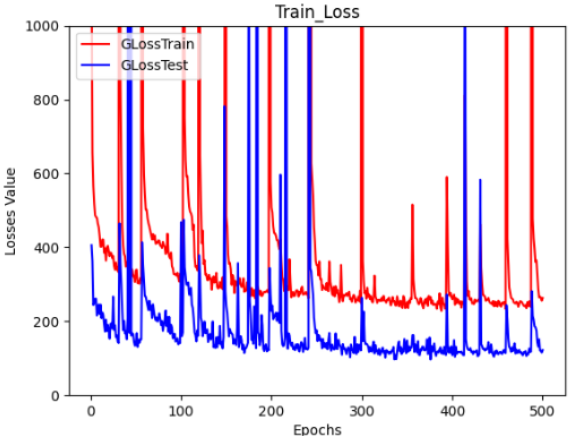


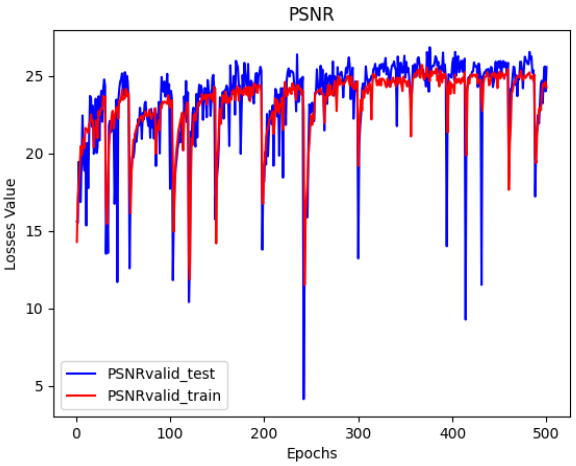
original 500x281 4x generated output (2000x1124)

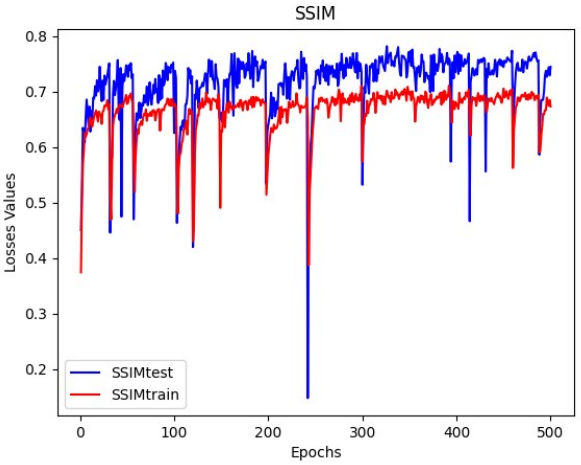
Performances

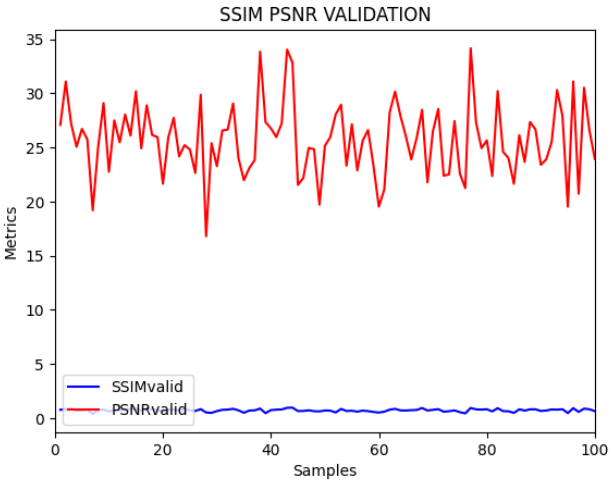
After 500 epochs the Model obtained PSNR: 25.55 SSIM: 0.74 on the test phase











Implementation of Variants

Some alternative choices have been made over the original architecture in order to improve the performances of the training, in particular:

- A Sigmoid function as activation function for the last convolutional layer to keep the range between 0 to 1 for the pixels.
- The alpha initializer of the PReLU is set at 2e-2 to avoid gradient fading also for the first epochs
- A Generator to Discriminator Ratio (gdr) calculated as

$$\frac{Generator\ Loss}{Discriminator\ Loss}$$

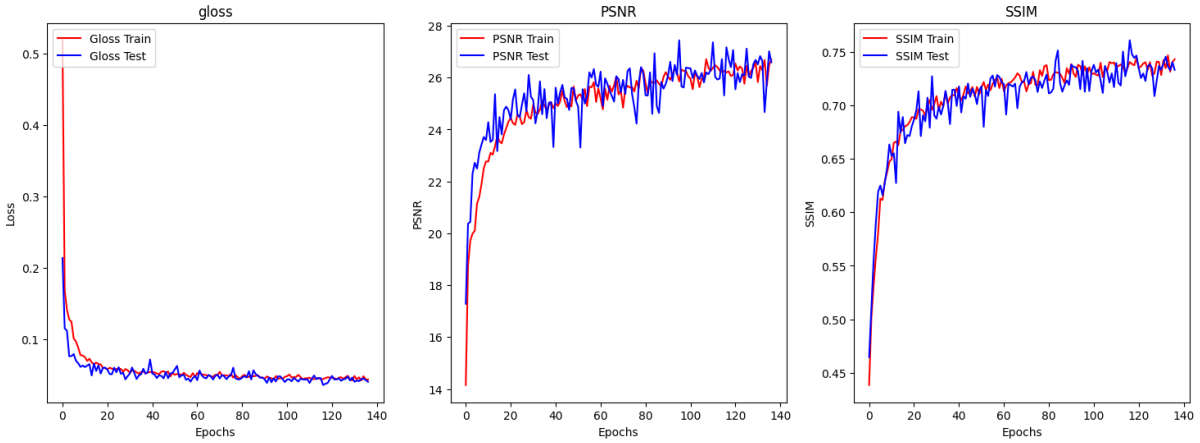
  has been used during training to maintain an equilibrium between the training of the two architectures by stopping the one exceeding its threshold set to match the scales of the losses.

Also, three variants of the content loss are proposed:

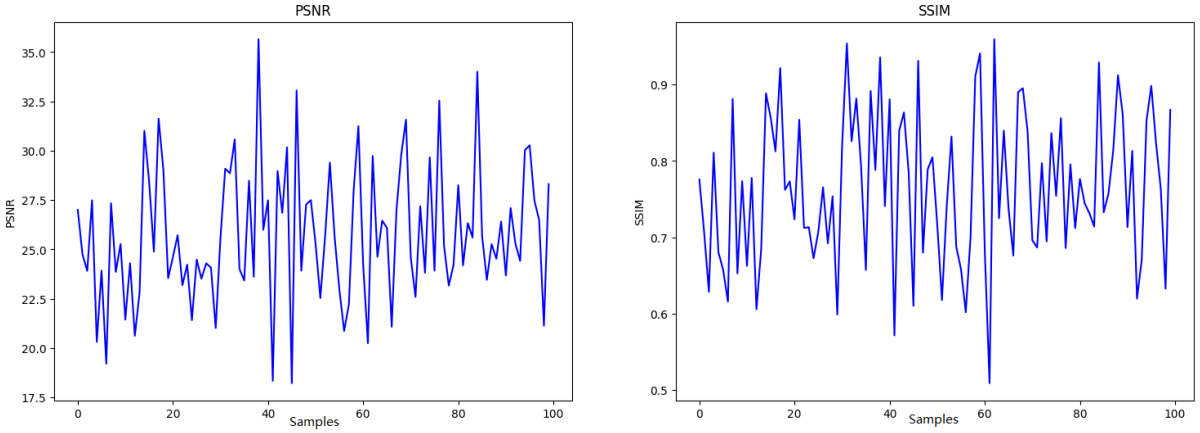
- Pixel-wise MSE: Calculated by taking the mean(squared(y_true - y_pred)) where **y_true** is the tensor of the high resolution image and **y_pred** is the tensor relative to the generated image (rescaled on [0,1] by the sigmoid activation). It was observed having higher performances in terms of metrics but also worse perceptive performances as reported in the original paper.

  For this variant the generator has been given a beta1 of 0.5 to give it a smoother descent, the learning rates have both been set to 2e-4 and the pixel-wise weight has been set to 1e2 to match the order of the vgg loss to have the same ratio between content and adversarial losses w.r.t. the other variants.

  Discriminator stop condition set to gdr > 100 and for generator gdr < 5e-2 (generator loss < 20 * discriminator loss generator is considered to be too well trained w.r.t. the discriminator so its training is stopped) Train over 800 samples + test over 100 samples

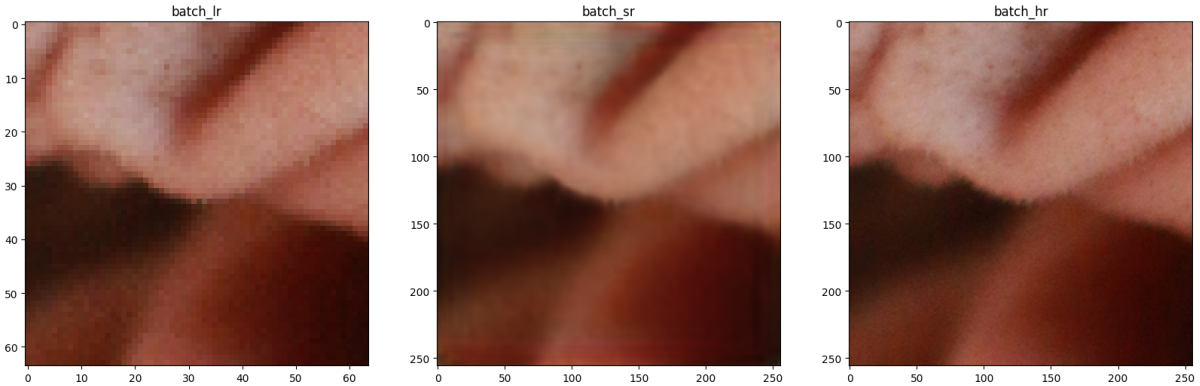


Evaluation over 100 samples:

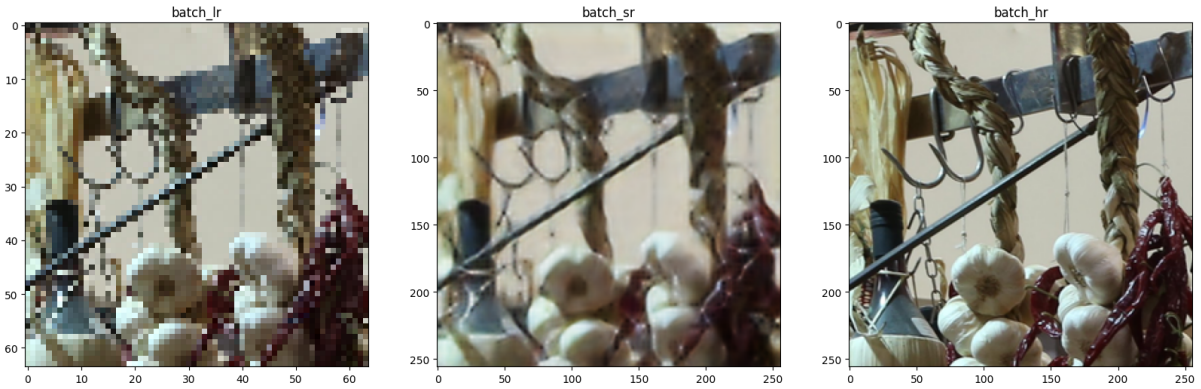


Example of test images generated:

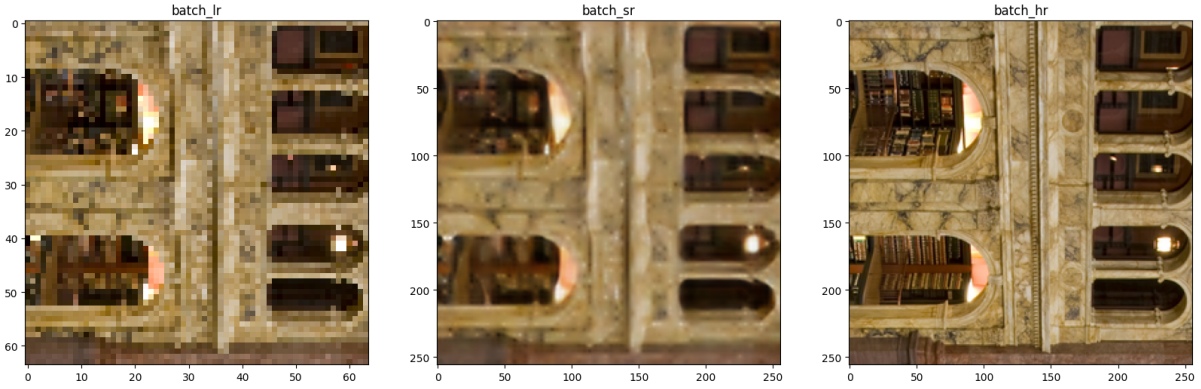
Epoch 20



Epoch 80



Epoch 135

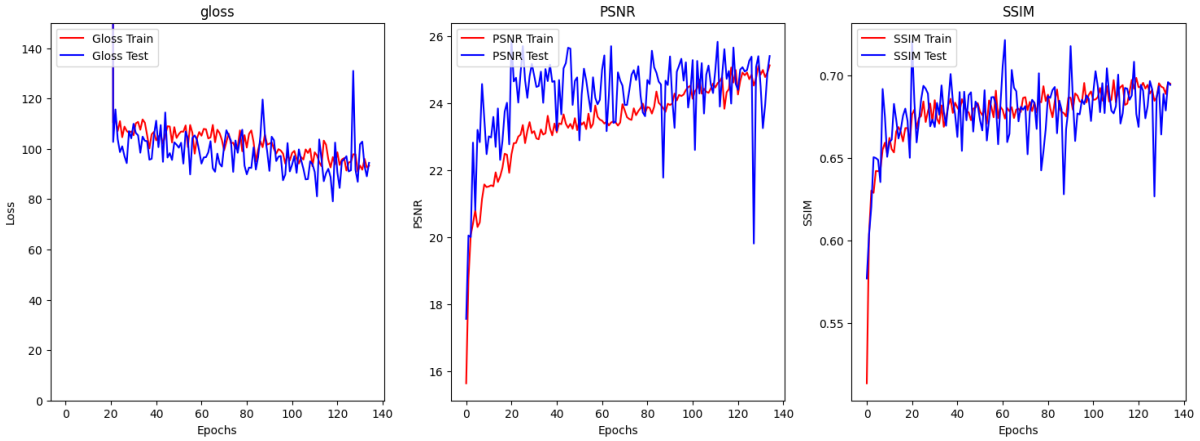


- Vgg Features MSE: Extracts the feature maps of the ground truth image and the ones relative to the generated image, then calculates the mean squared error between them. It was observed to focus on image's details resulting in worse metrics (doesn't optimize directly the MSE between the images) but much preferable results in terms of perception.

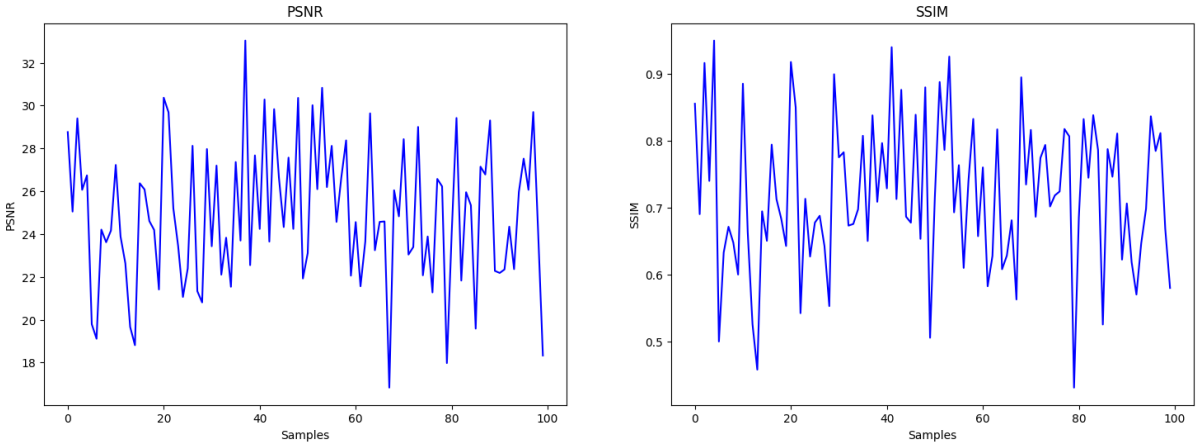
  Hyperparameters selection for this variant are: common learning rate of 2e-4, content loss of 6e-4 (changed after 20th epoch because of the too high loss values given by the selection of block3_conv4 for the vgg feature extraction instead of the block5_conv4 selected in the paper's implementation)

  Discriminator's training stop condition to gdr > 200 and for generator to gdr < 5e-2

  Train over 800 samples + test over 100 samples

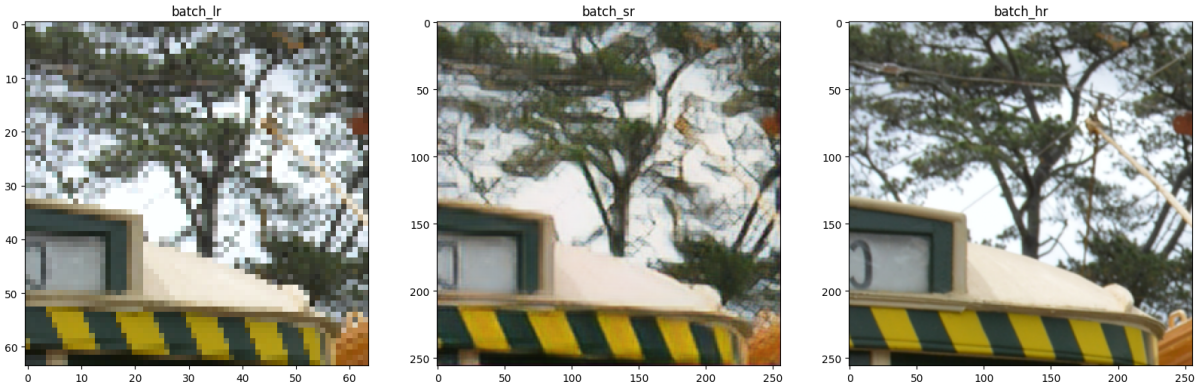


Evaluation over 100 samples:

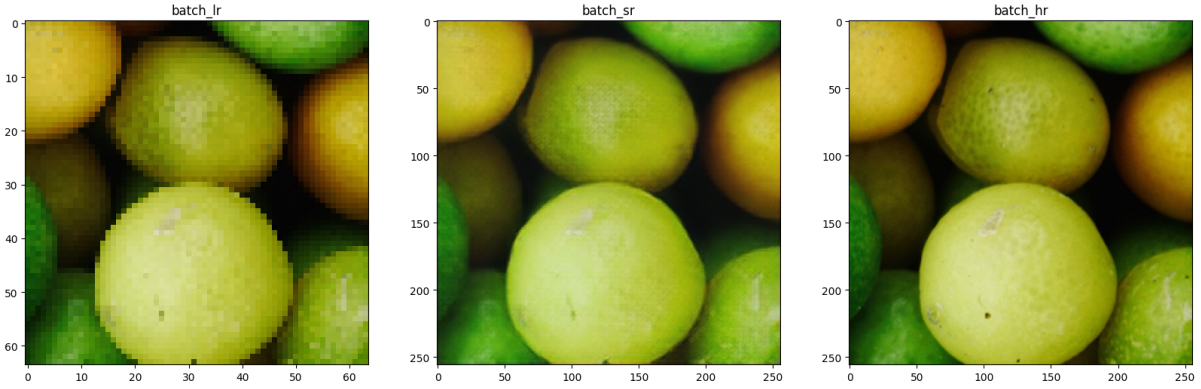


- Example of test images generated:

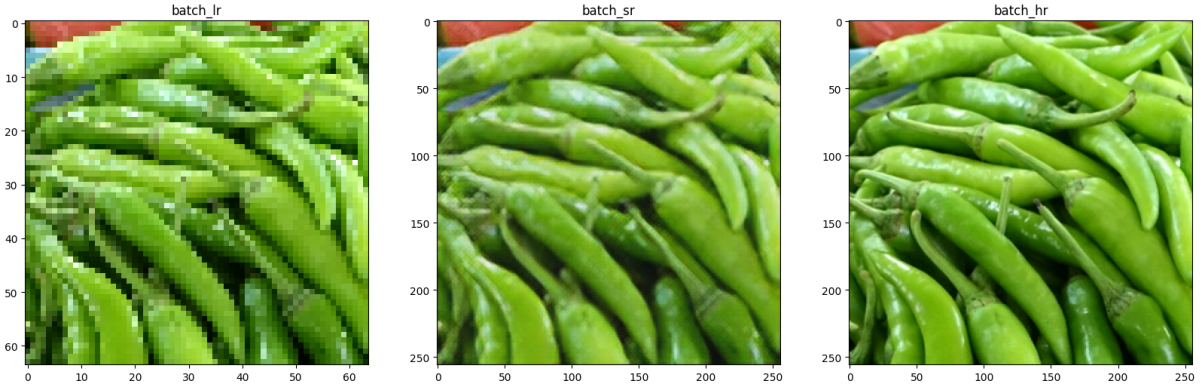
  Epoch 20



  Epoch 80



  Epoch 135



- Mixed: First calculates the pixel-wise MSE on the batch, then also calculates the extracted feature maps MSE and averages the two errors with different weights to get more importance to the VGG loss. The difference in values from Vgg Features MSE is minimal, but with higher peaks on the metrics PSNR and SSIM keeping the same values for the adversarial loss.

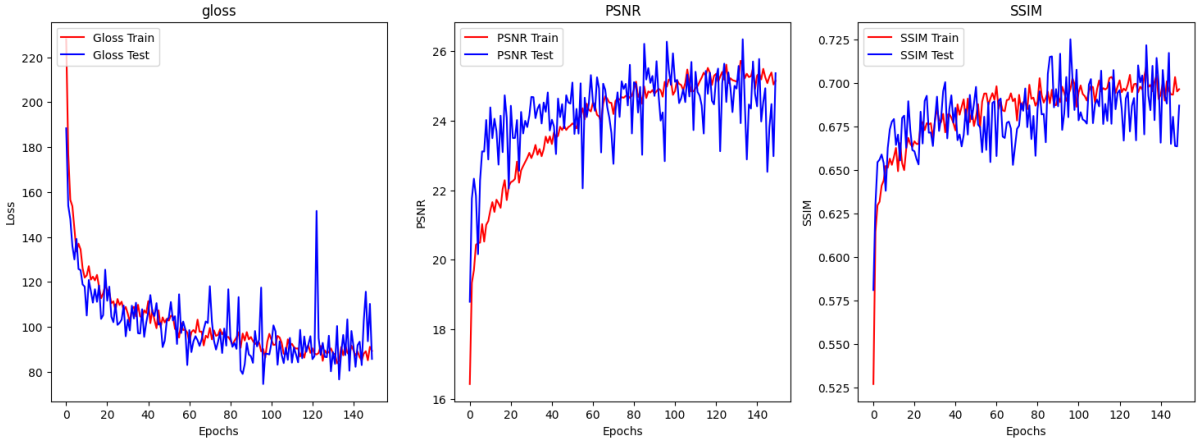
  For this variant of the perceptual loss, the standard learning rate of 2e-4 has been chosen with a beta2 of 0.8 to lower the second order

momentum and keep major memory of previous gradients.

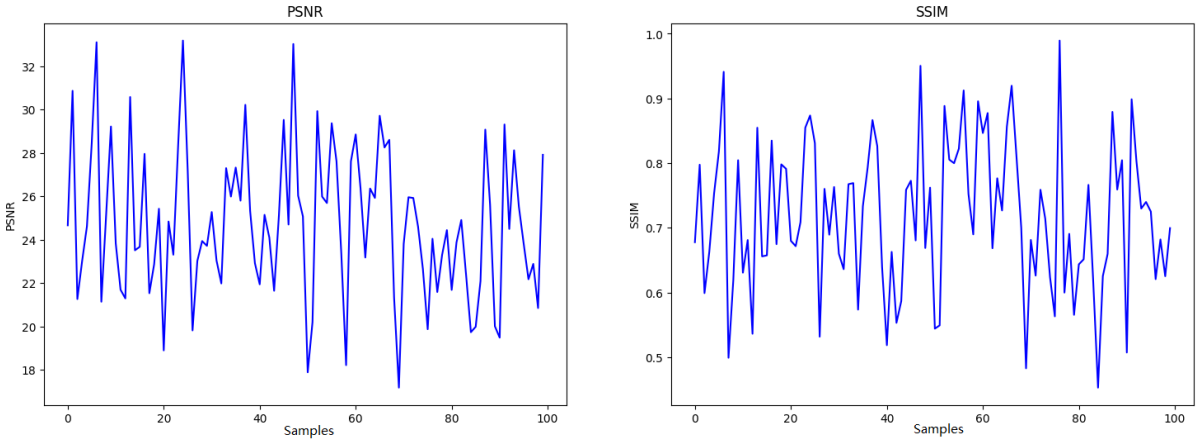
The discriminator stop condition has been set to gdr > 200 because of the higher scale of the error due to the vgg loss summed to the mse loss, while the generator stop condition has been set to gdr < 5e-2.

Pixel-weight of 10 (lower to give more importance to the vgg loss); vgg-weight of 6e-4 (e-1 more w.r.t. the paper to match the block3_conv4 scale of the loss which is higher due to the higher dimensionality 64x64x256), and the discriminator loss has been given a beta1 of 0.7 to match the training speed of the generator

Train over 800 samples + test over 100 samples

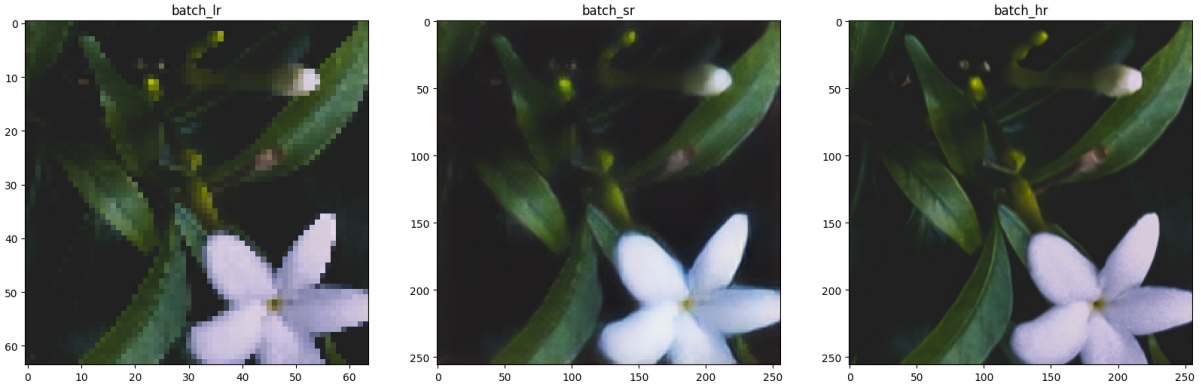


Evaluation over 100 samples:

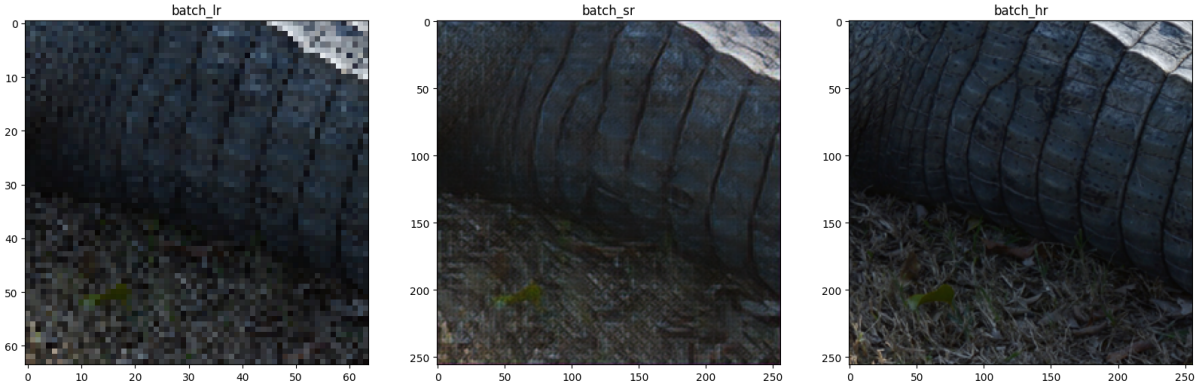


- Example of test images generated:

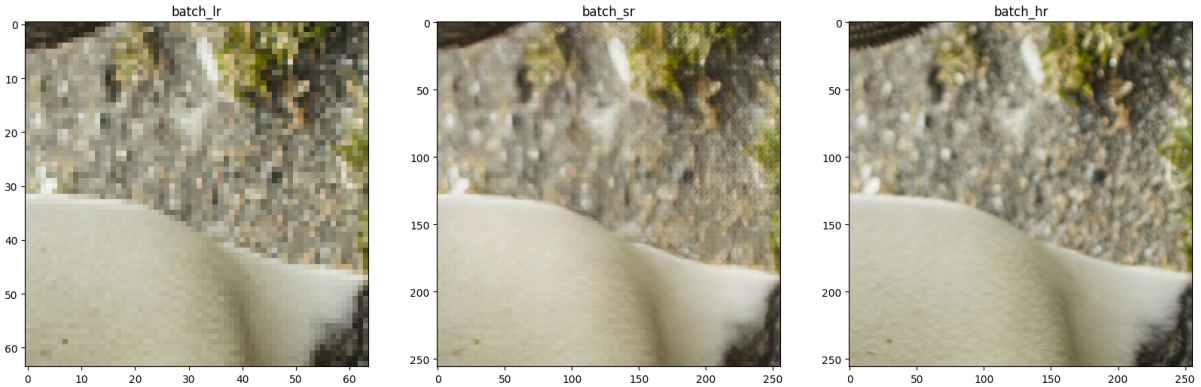
Epoch 20



Epoch 80



Epoch 150



Experiments:

During the development relative to the SRGAN some experiments were tested.
For the proposed implementation 3 possible variations about the handling of the generated images have been tried, due to the missing of one normalization layer such as the sigmoid layer at the end of the Generator:

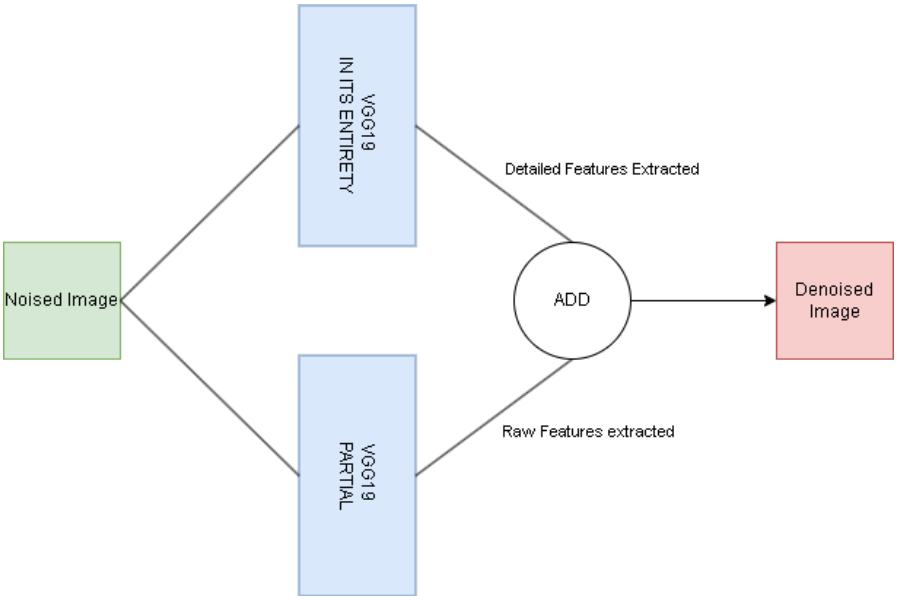
1. Output normalization followed by a clipping image values:
   ```
   def normalize_image(img):
     img = tf.cast(img, tf.float32)
     min_vals = tf.reduce_min(img)
     max_vals = tf.reduce_max(img)
     # Perform min-max scaling
     img = (img - min_vals) / (max_vals - min_vals)
     img = tf.clip_by_value(img, 0, 1)
     return img
   ```

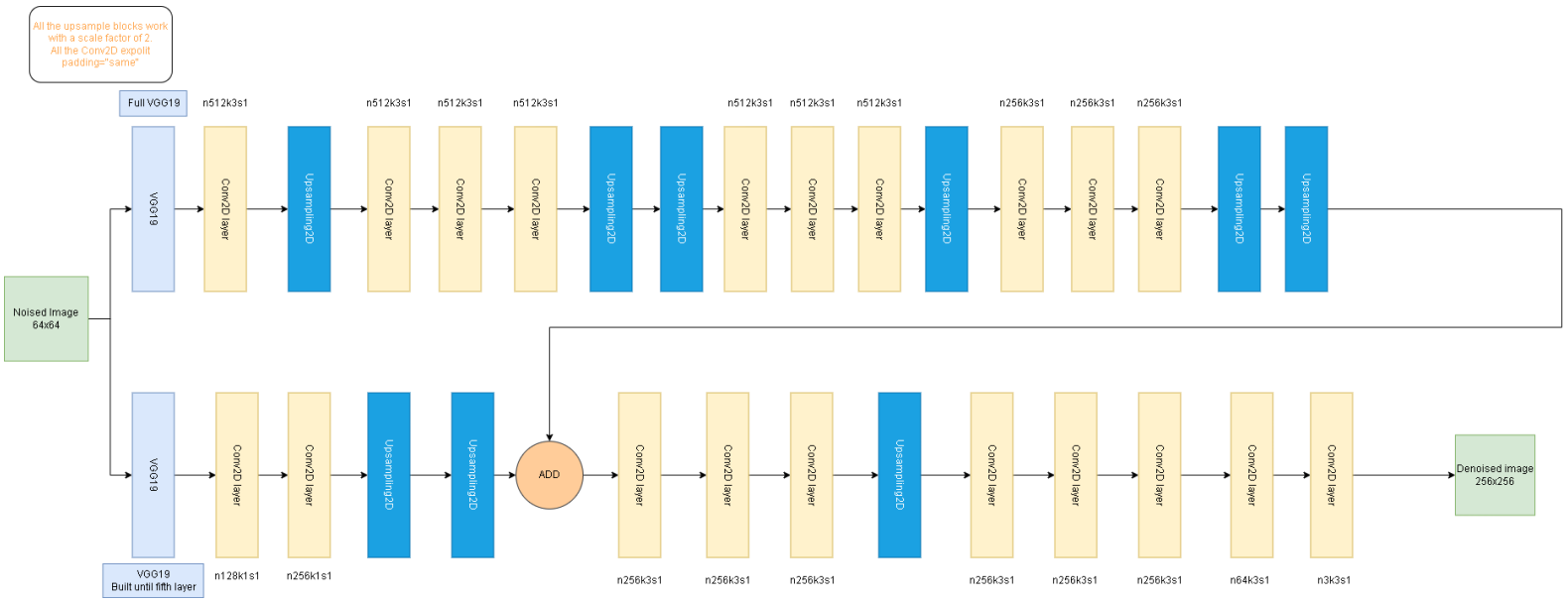
   also tried with a full tensorflow APIs implementation
2. Only output clipping
3. No Normalization

This last attempt gave the best results on 20 epochs

Denoiser Architecture

In the following architecture, before the last 2 conv2D blocks, 3 convolutional blocks also include a dilation parameter of 2.





Metrics and Losses:

The goal of this architecture is to perform denoising, to achieve this MAE was chosen, calculated between the denoised image and the ground truth image.

The weight of the loss function, relative to all the following implementation, was selected empirically, with value $1e^{-3}$.

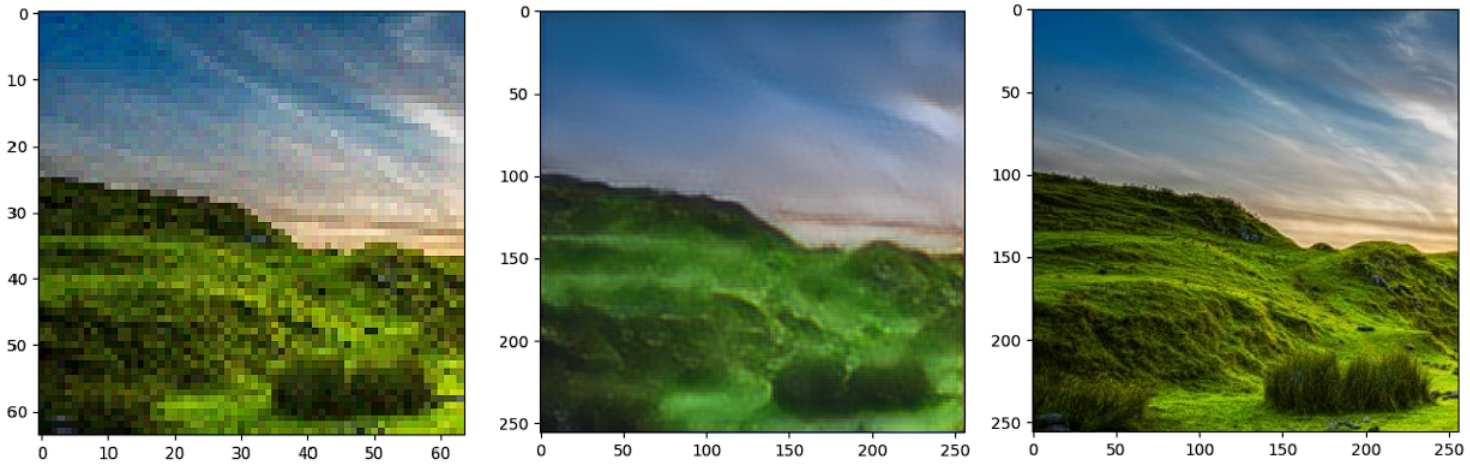
PSNR and SSIM were selected as metrics.

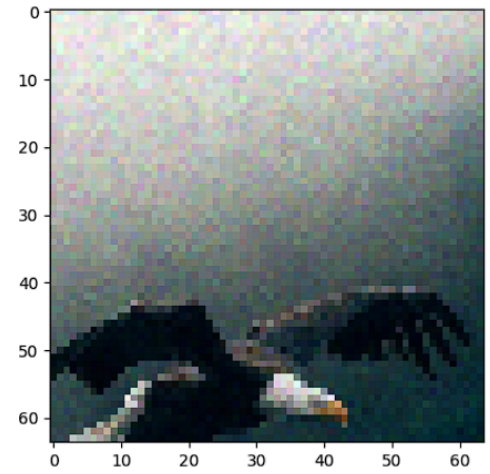
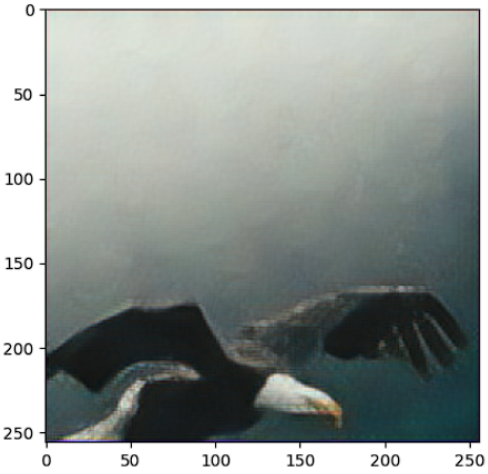
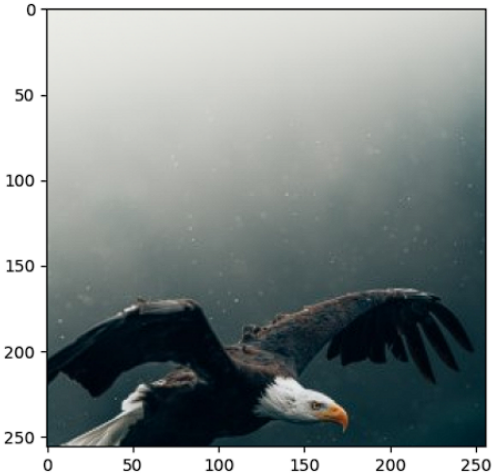
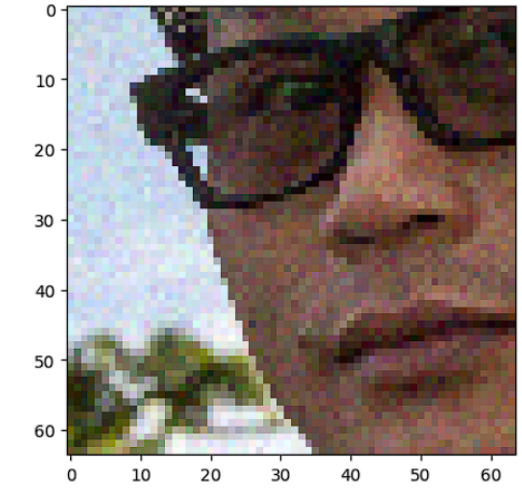
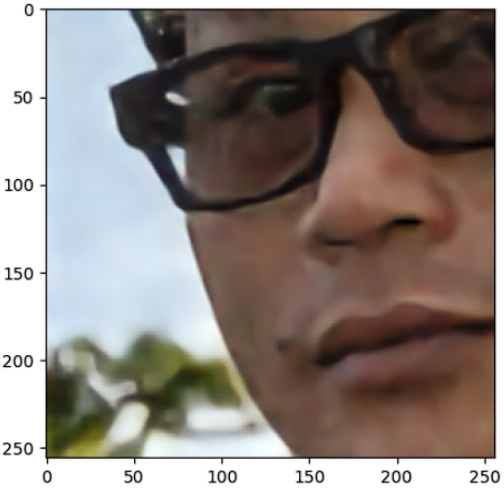
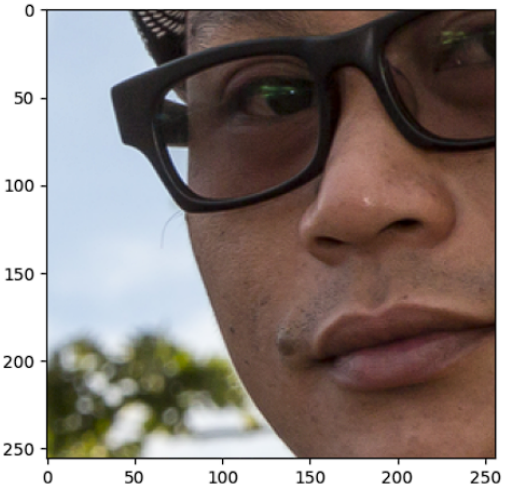
```
common_optimizer = Adam(learning_rate=0.0002, beta_1=0.5)

Model.compile(optimizer=common_optimizer, loss='mae', metrics=[[psnr_metric,
ssim_metric]], loss_weights=1e3)
```

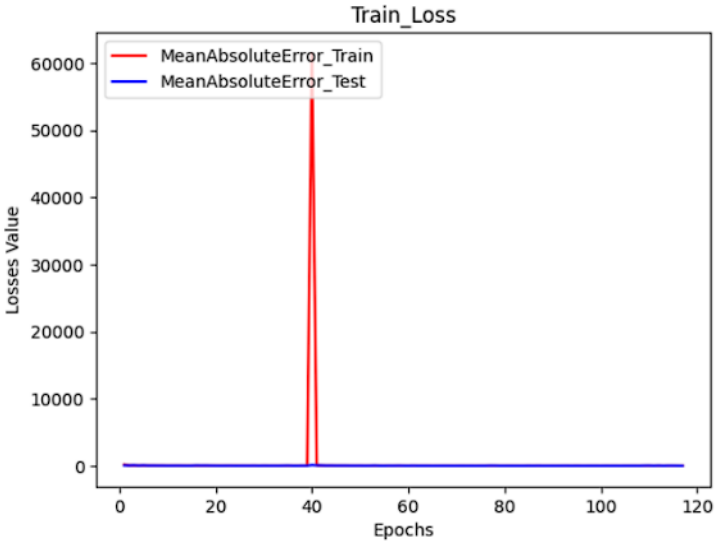
The results are shown below for epochs 20, 60 and 110

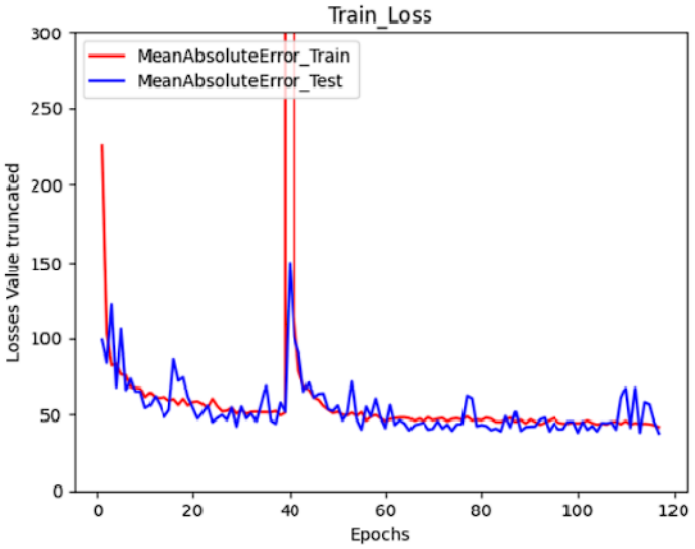
Low Resolution Images Generated Results High Resultion Images

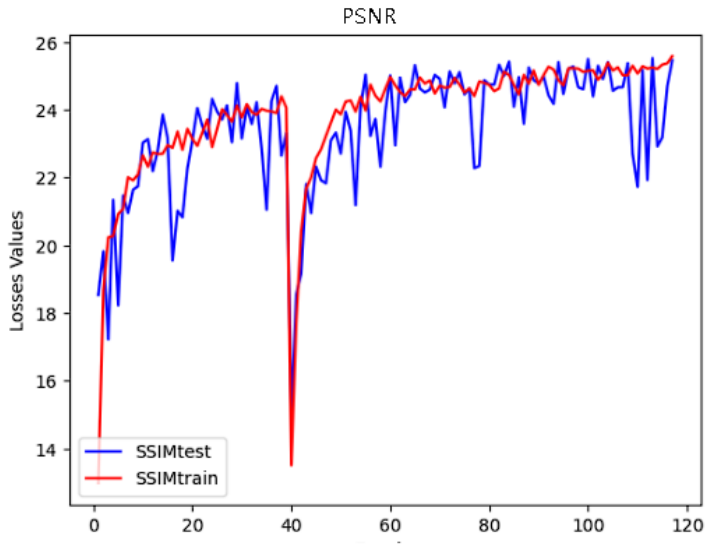


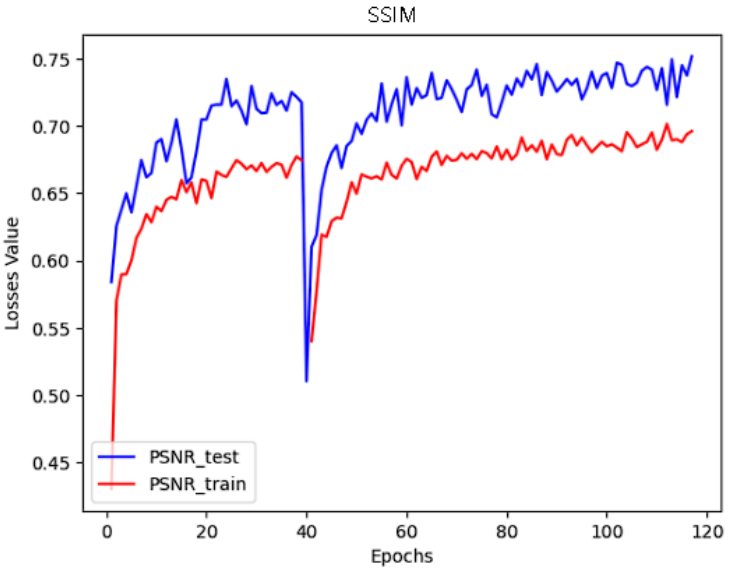







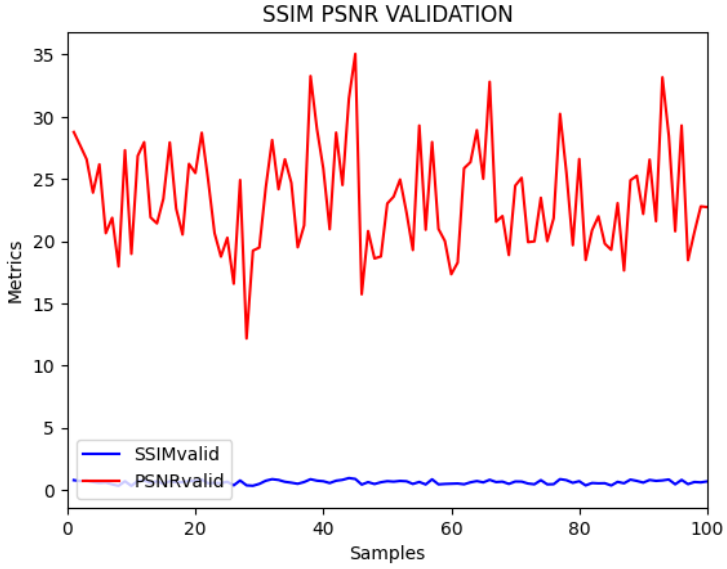
Performances









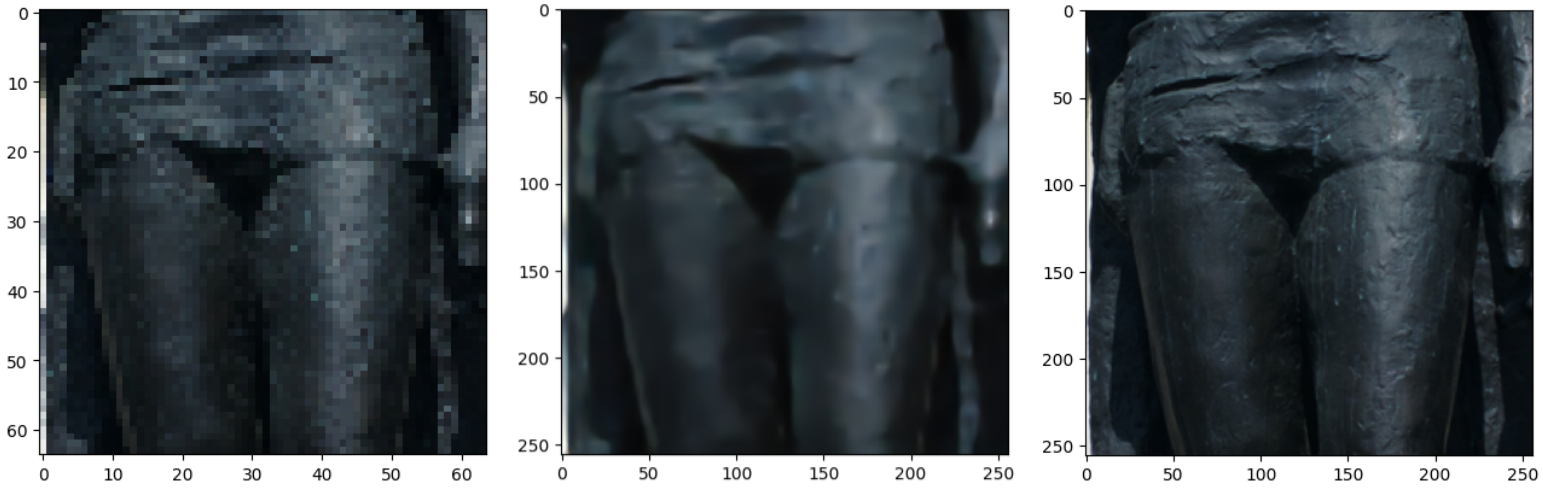


Alternative Implementation

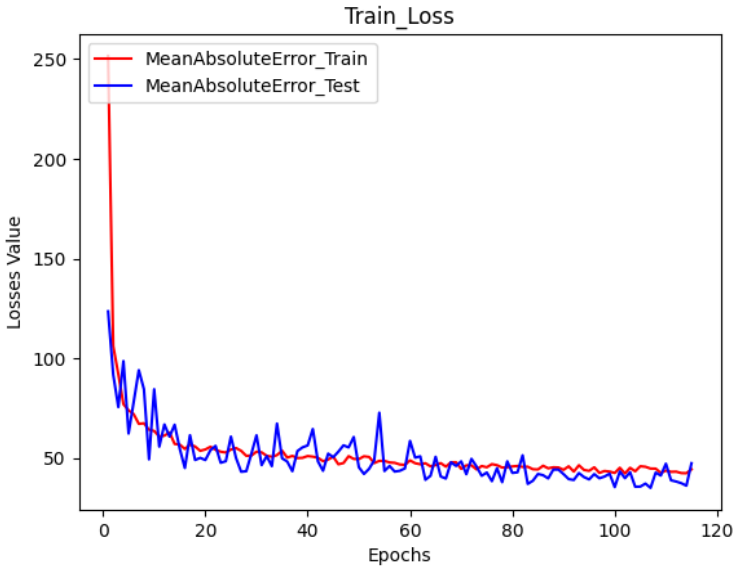
SSIM and PSNR Legends are inverted

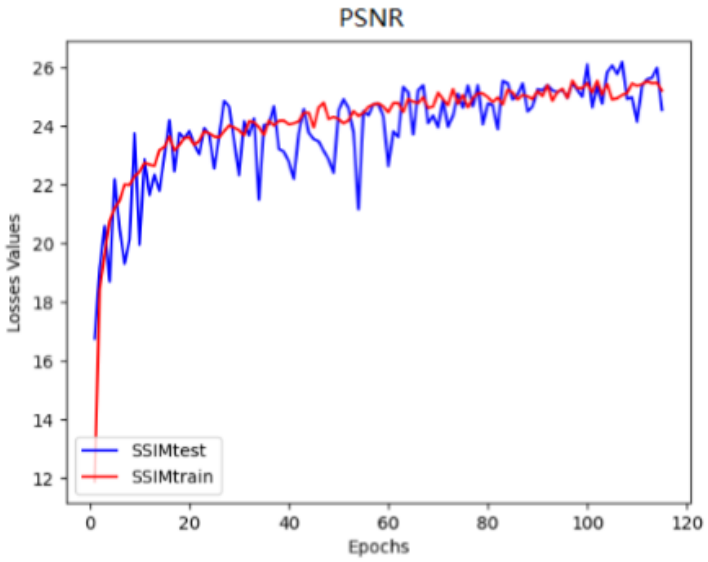
1- The following Neural Network is built with the same architecture as the previous network, using the same loss function, with the only difference being that this one was fine tuned on both the VGG19. The architecture was trained for 120 epochs. (dilationDenoisermseATLAST)

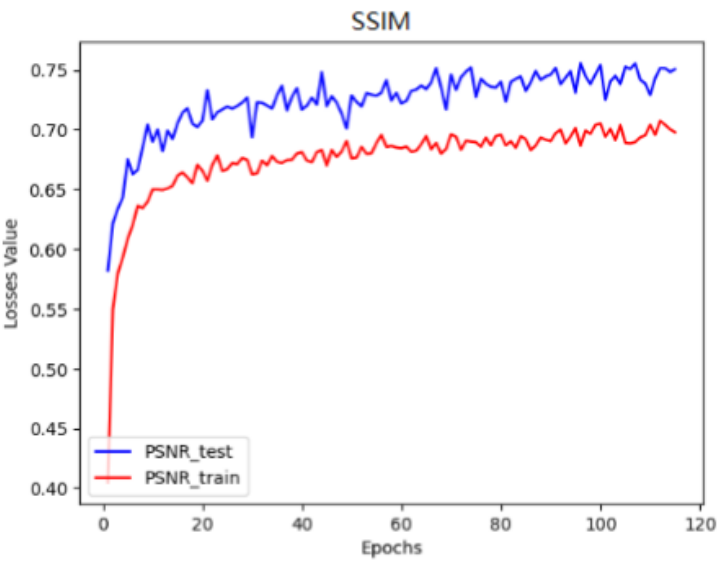
Results:

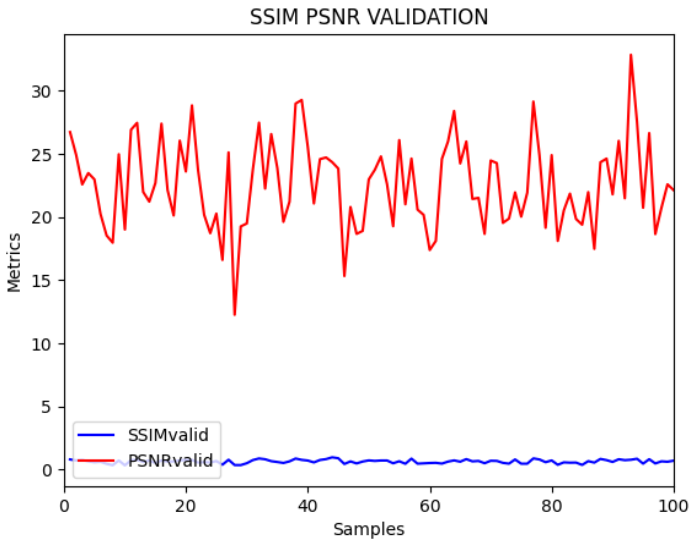


Performances:



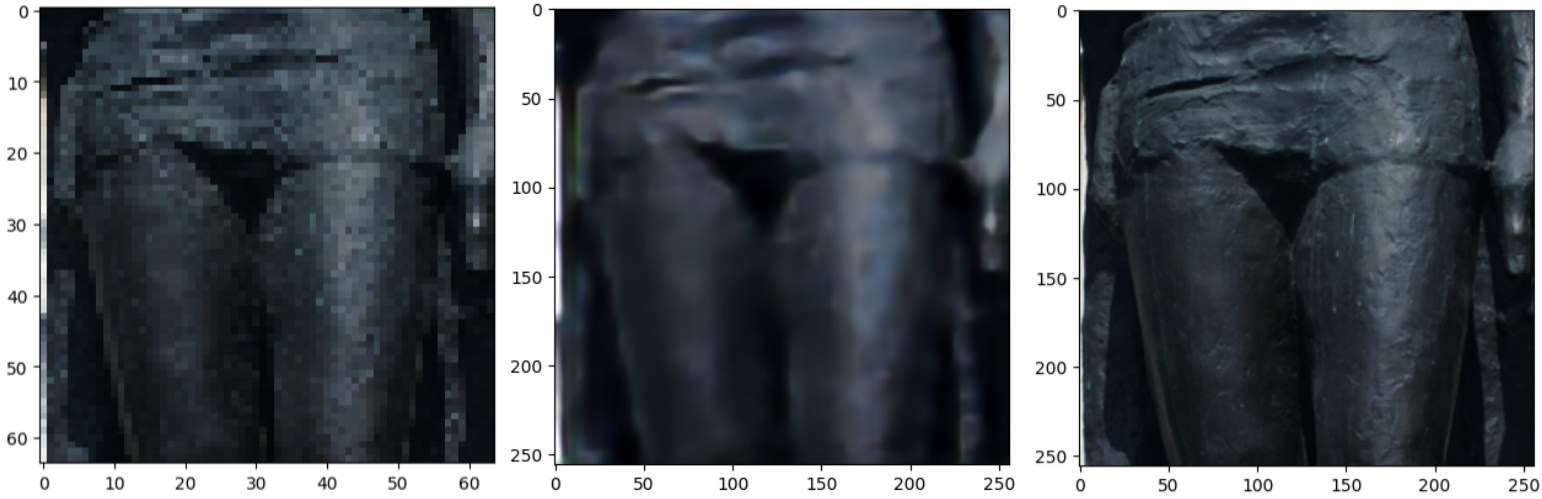




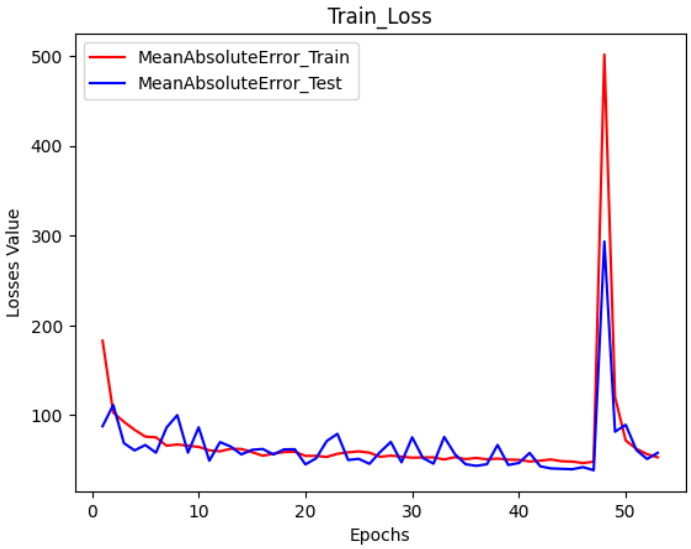


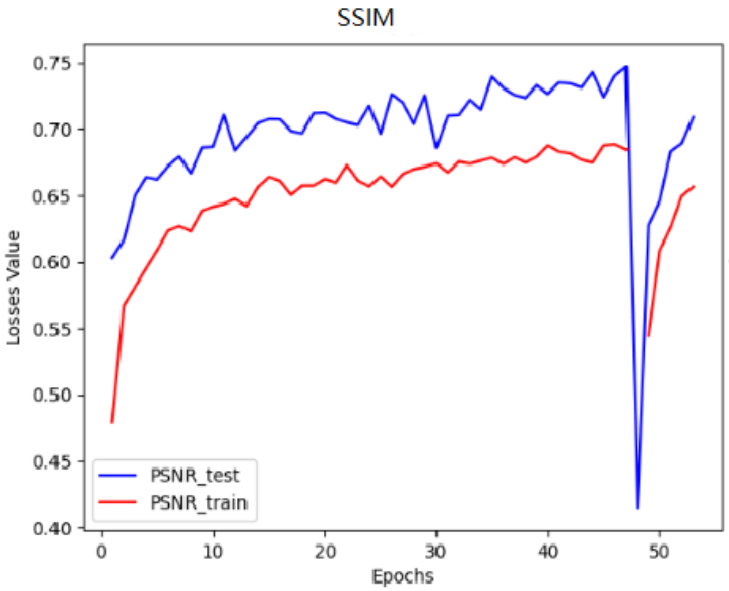
2 - The following implementation is a variation of the above model. This architecture exploits the same loss function , fine tuning was performed on both VGG19 networks. Layers using dilation were added at the beginning and in the middle of the architecture. (dilationDenoisermse)

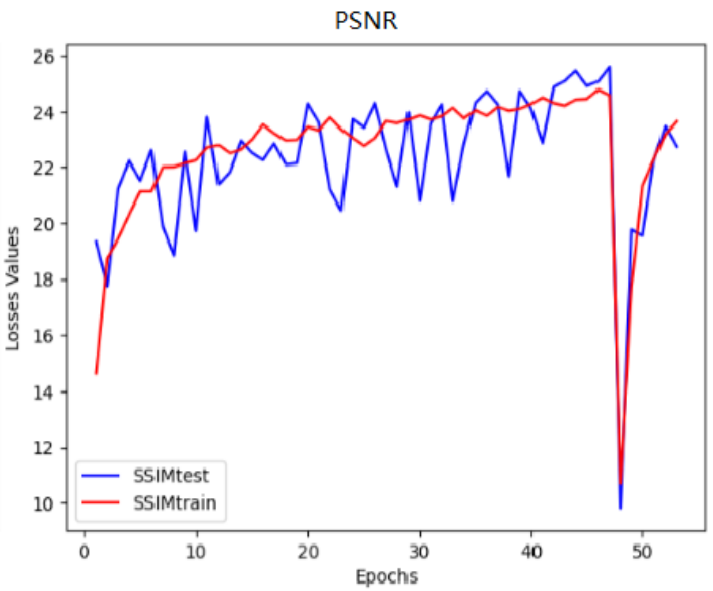
Results:

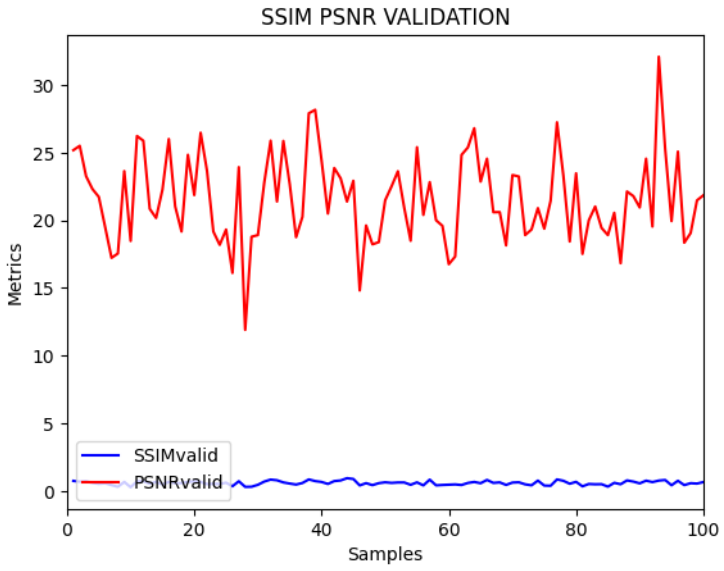


Performances:

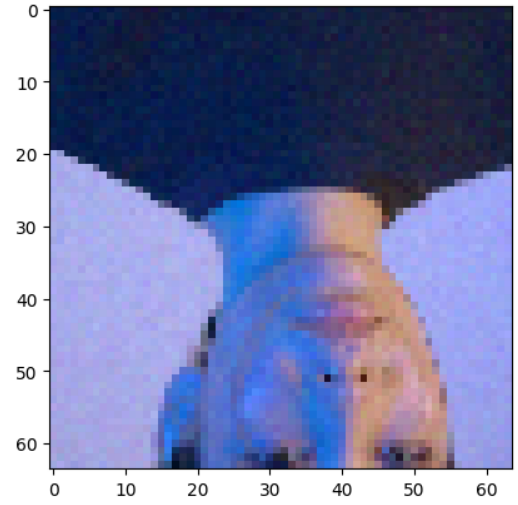

Train_Loss
MeanAbsoluteError_Train
MeanAbsoluteError_Test
Losses Value
Epochs

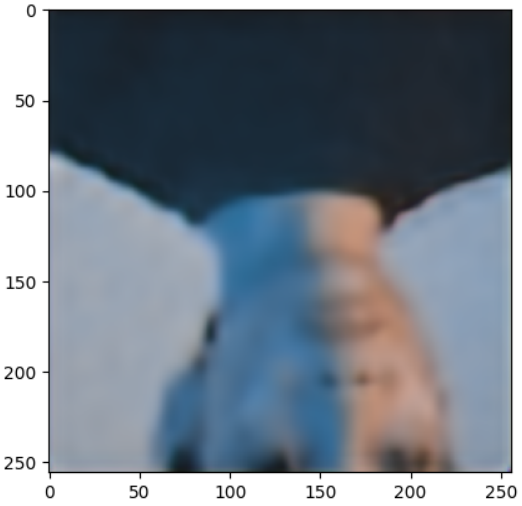

SSIM
PSNR_test
PSNR_train
Losses Value
Epochs

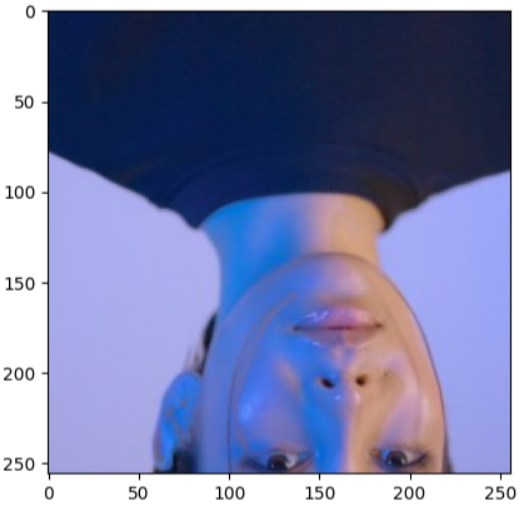

PSNR
SSIMtest
SSIMtrain
Losses Values
Epochs


SSIM PSNR VALIDATION
SSIMvalid
PSNRvalid
Metrics
Samples

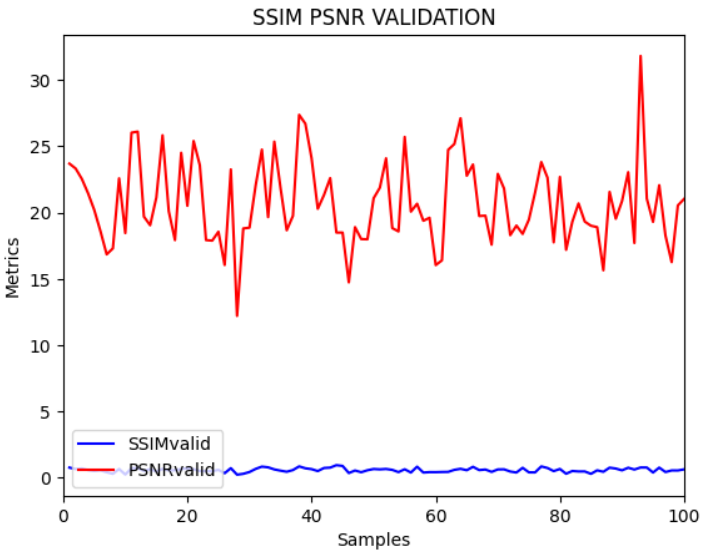
3 - The main difference between this model and the proposed one is the use of a different loss function, Mean Squared Error, which shows good performance with respect to the metrics and there aren’t dilated convolutional layers. The output of the architecture produces images with less contrast in comparison with the previous proposed architecture.
Fine tuning was performed.
This architecture was trained only for 25 epochs. (realmse)

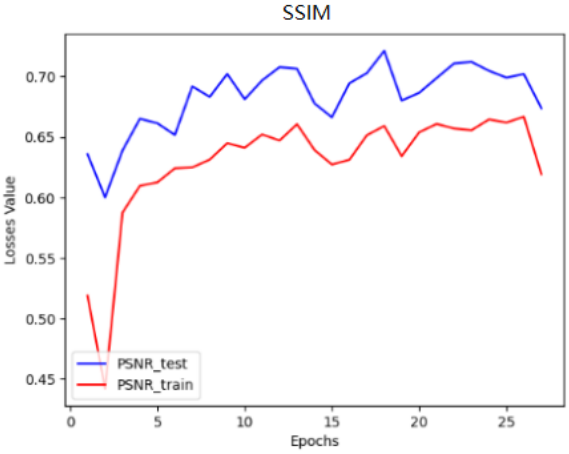


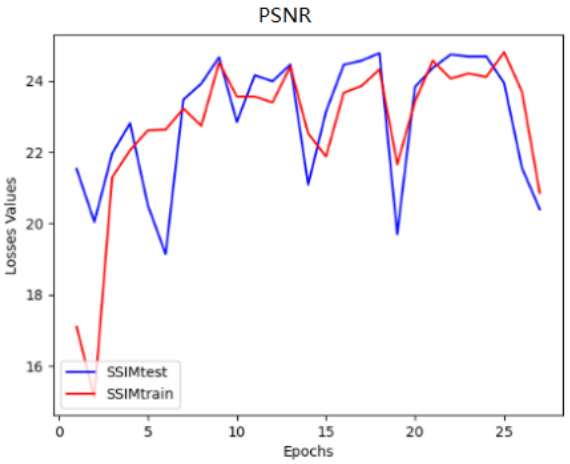


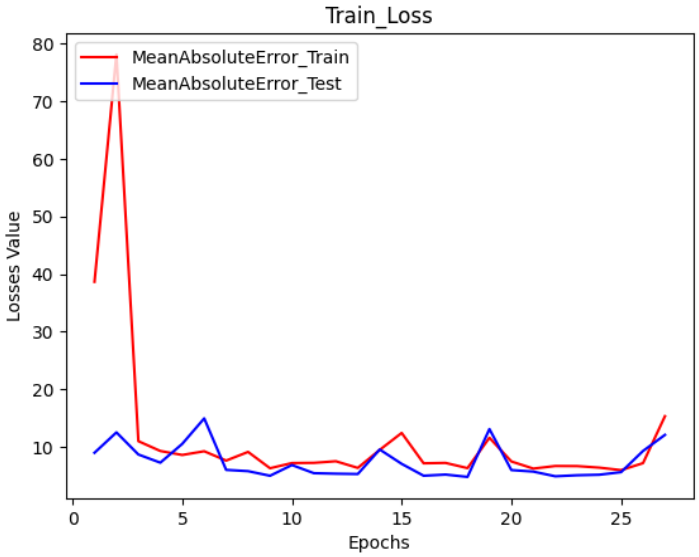


Performances:



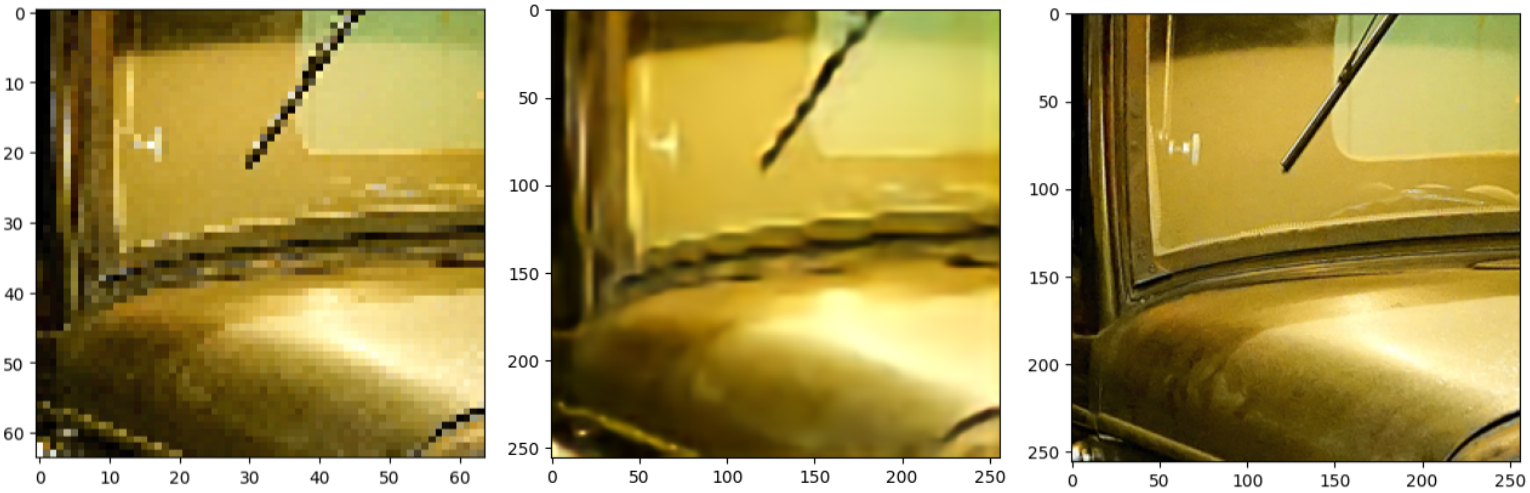




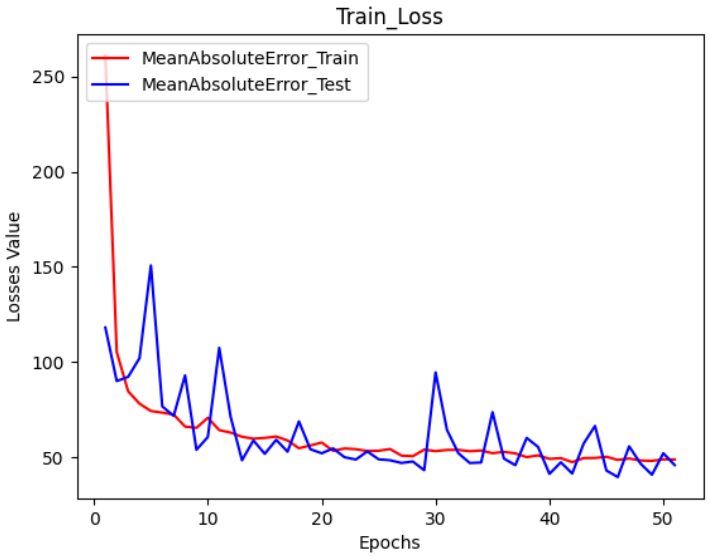


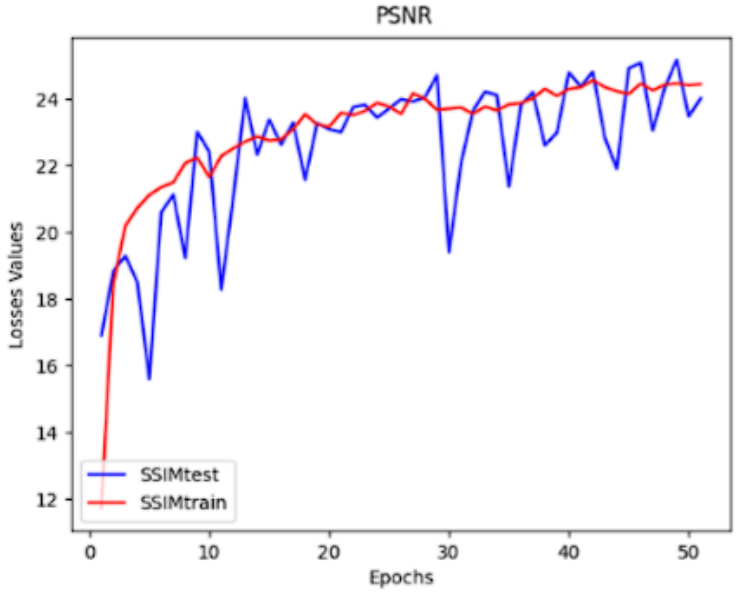
4 - Another implementation was tested, this time using mae without the dilation factor.
Fine tuning was performed on this architecture on both VGG19 networks.
The architecture was trained for 50 epochs. This architecture gave decent outputs, but visually not as good as the architecture implementing the dilation factor. (Denoisermse)

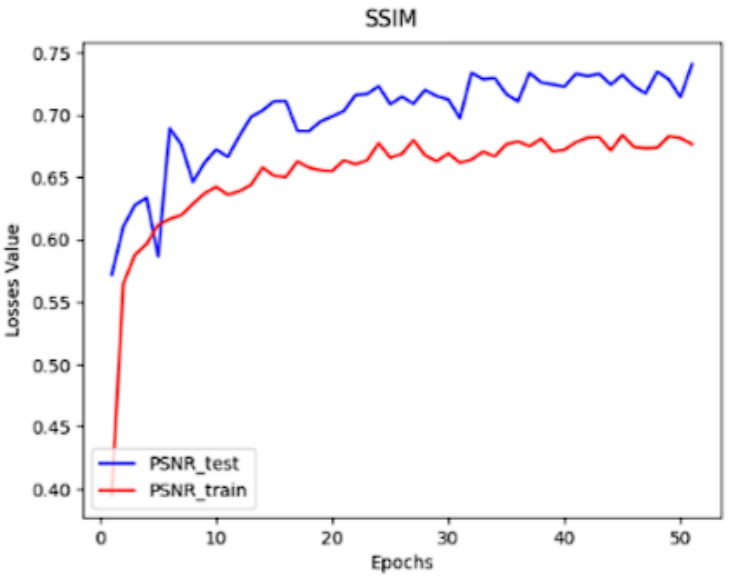
Results:

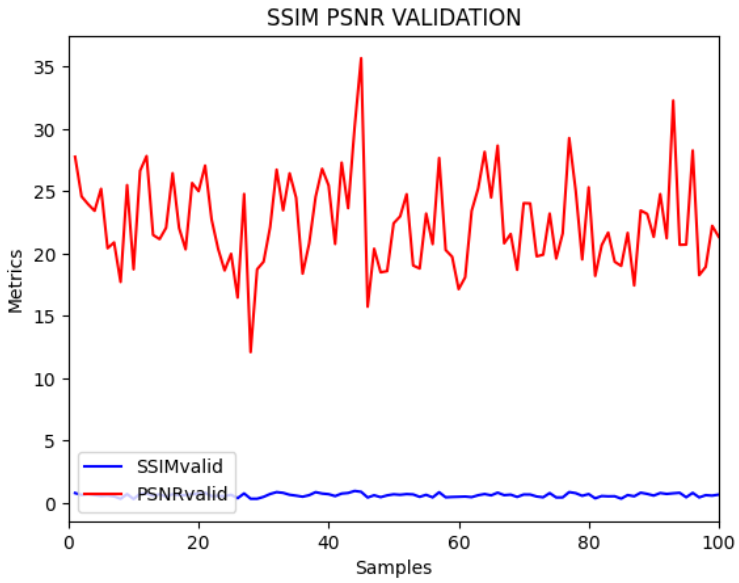


Performances:


Train_Loss
MeanAbsoluteError_Train
MeanAbsoluteError_Test
Losses Value
Epochs

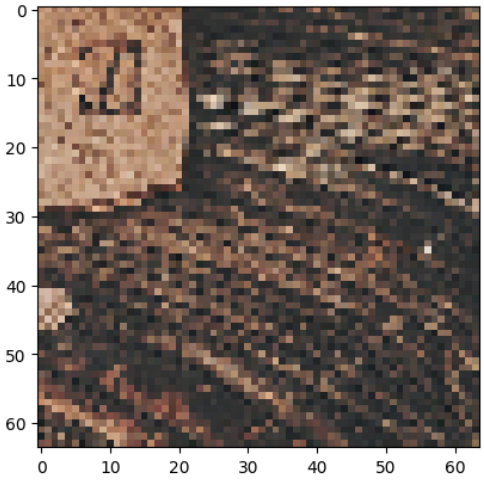
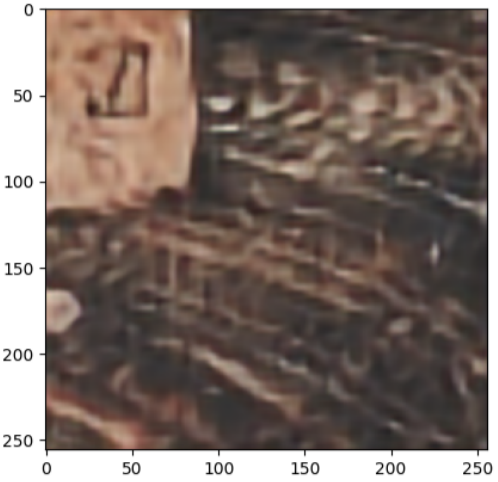
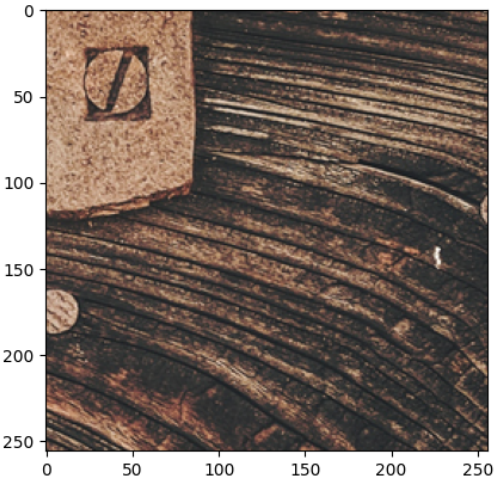

PSNR
SSIMtest
SSIMtrain
Losses Values
Epochs


SSIM
PSNR_test
PSNR_train
Losses Value
Epochs

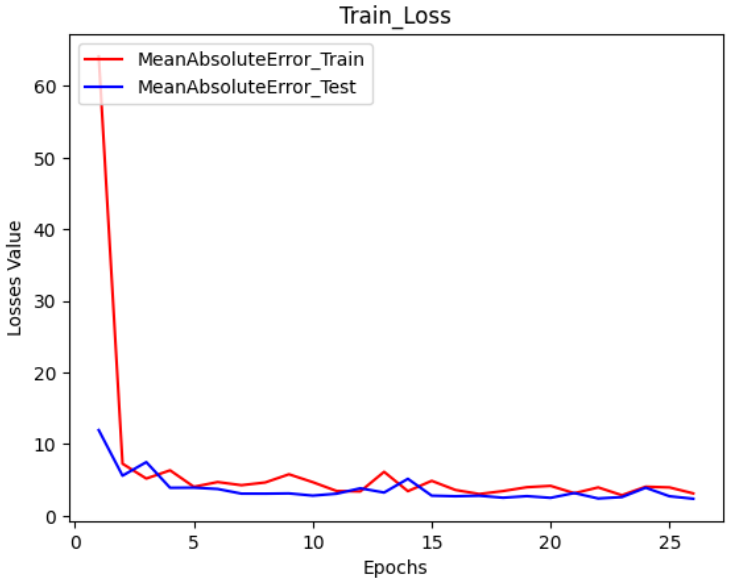

SSIM PSNR VALIDATION
SSIMvalid
PSNRvalid
Metrics
Samples

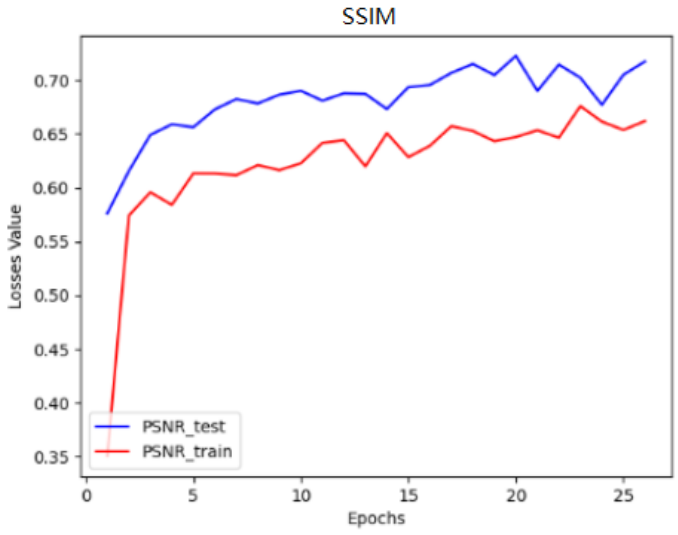
5 - Another approach was using Huber Loss, which seemed a good fit for this task, as described in the “Functions and Operations” section, but the results were still worse than the architecture using the dilation factor (MAE).
Fine tuning was performed on this architecture on both VGG19 networks. The architecture was trained for 20 epochs.

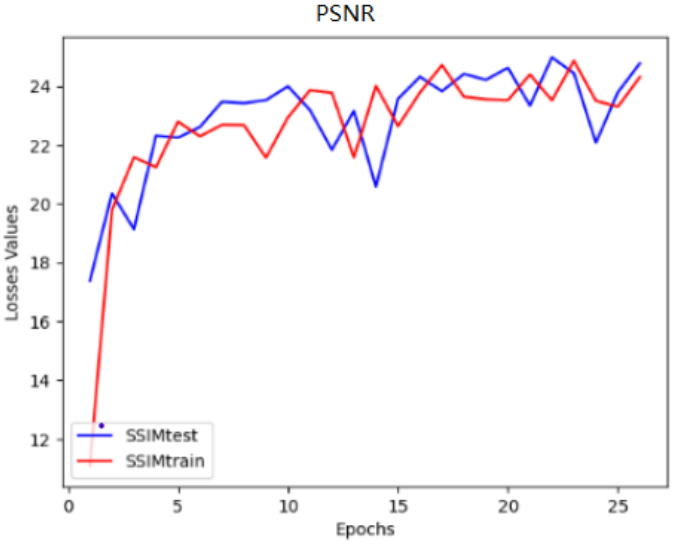
Results:

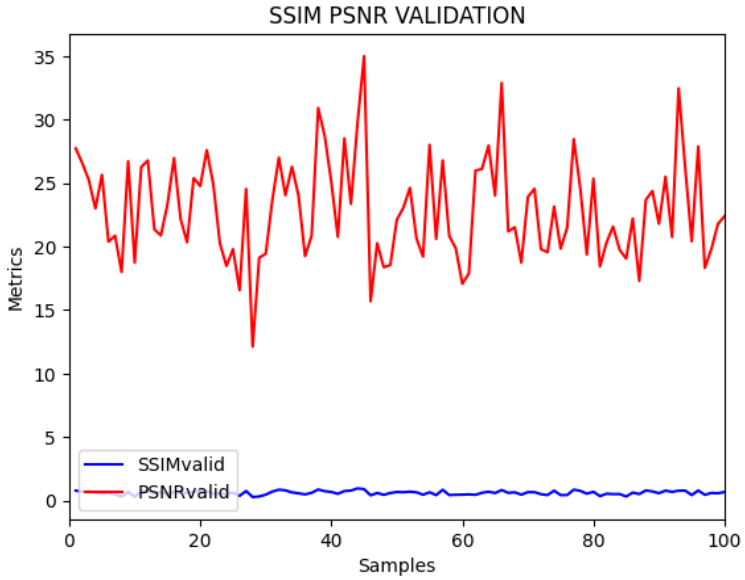




Performances:


Train_Loss
MeanAbsoluteError_Train
MeanAbsoluteError_Test
Losses Value
Epochs

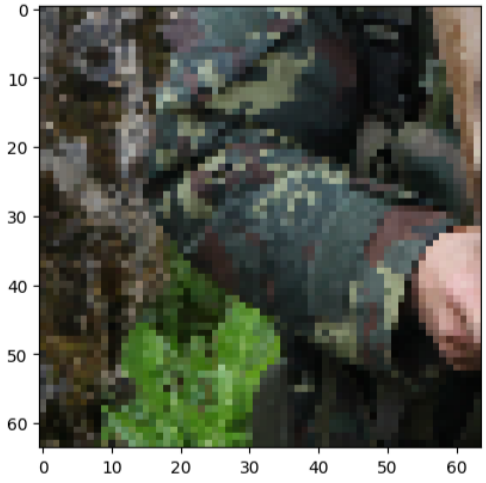

SSIM
PSNR_test
PSNR_train
Losses Value
Epochs

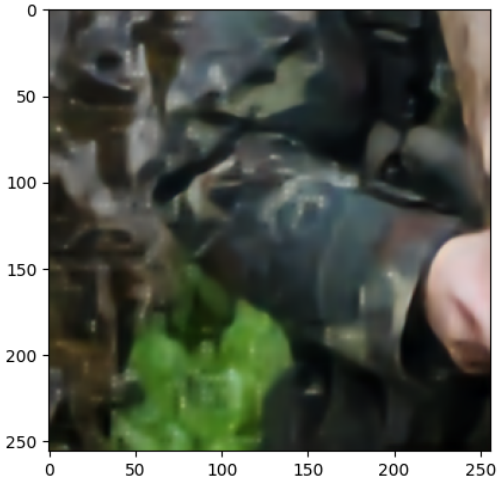

PSNR
SSIMtest
SSIMtrain
Losses Values
Epochs

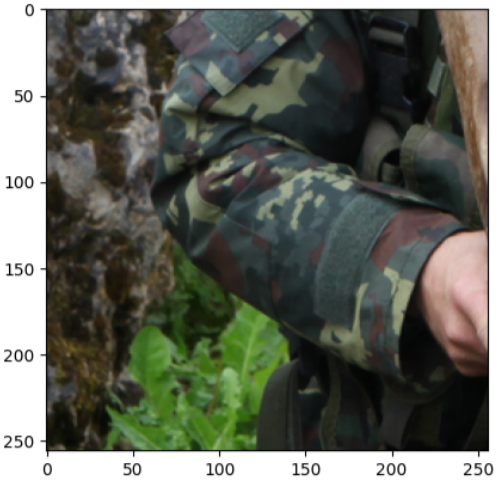

SSIM PSNR VALIDATION
SSIMvalid
PSNRvalid
Metrics
Samples

6 - The peculiarity of this Neural Network is the use of many residual connections, while trying to enhance the performances of the generator, using also the same dilated convolutional layer of the proposed model and the same loss function MAE.
The trainable states of the VGG19 are set to “True” .
The model training lasted 120 epochs, achieving acceptable results.

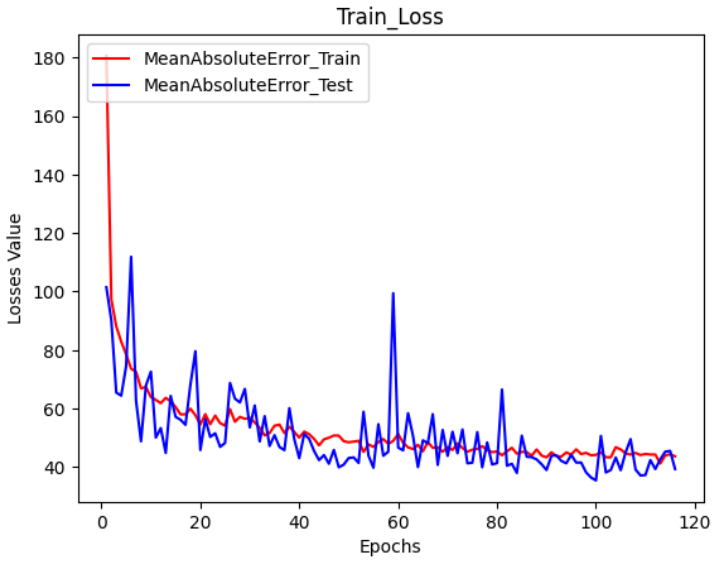
Results:

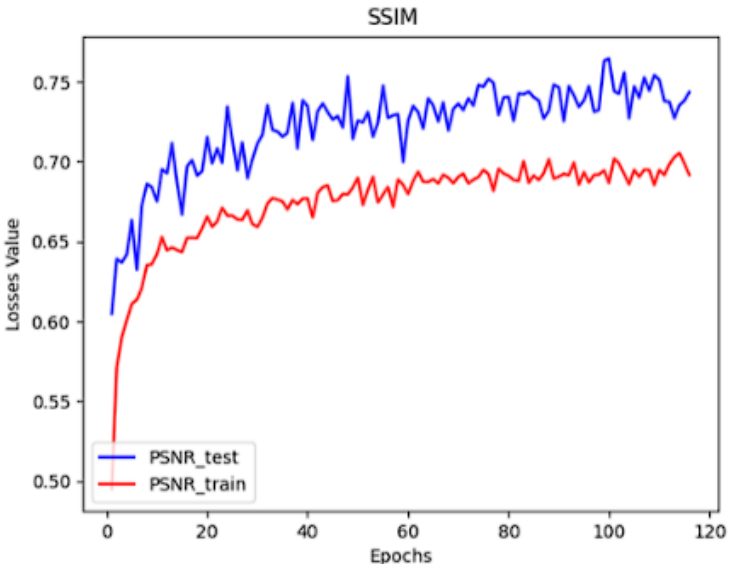


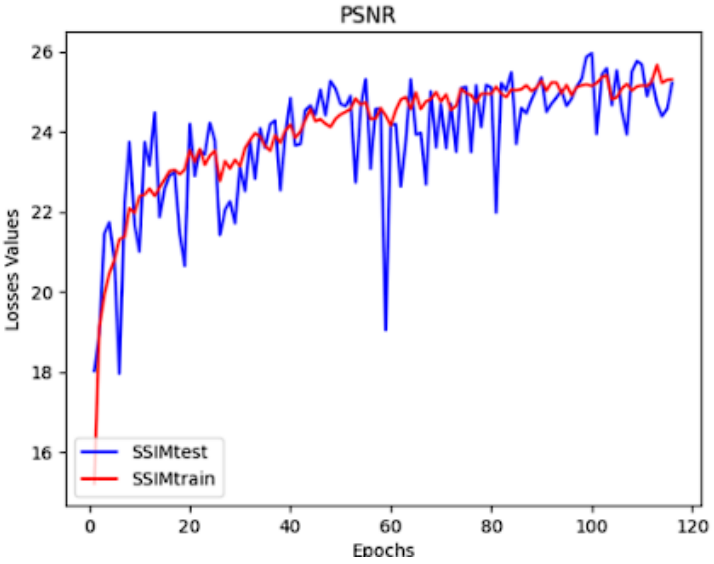


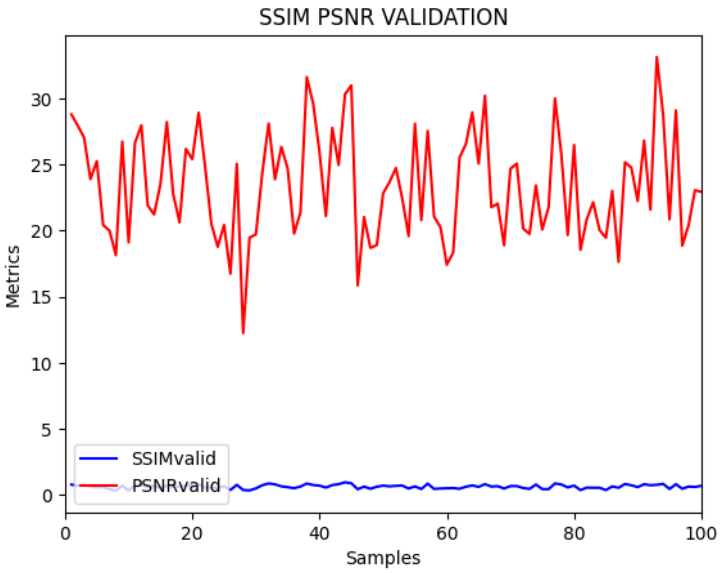


Performances:



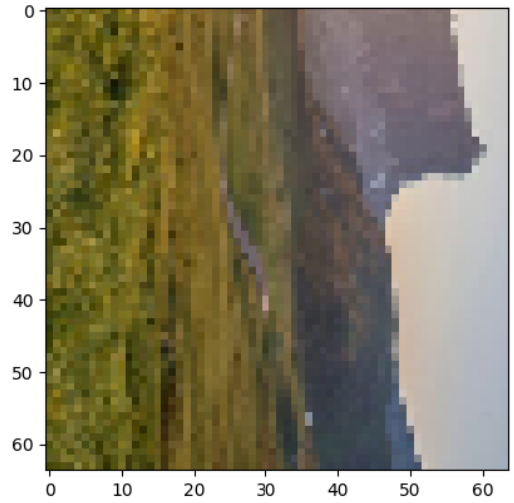
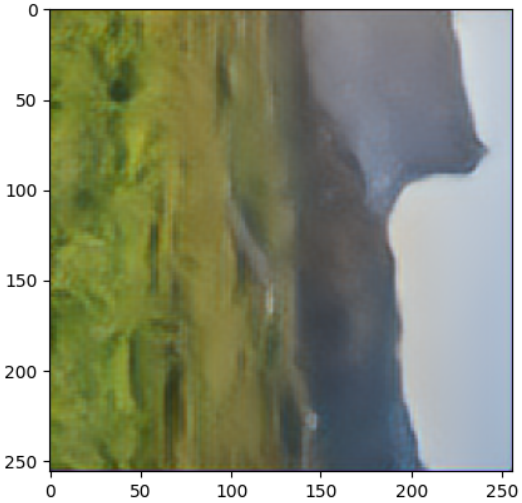
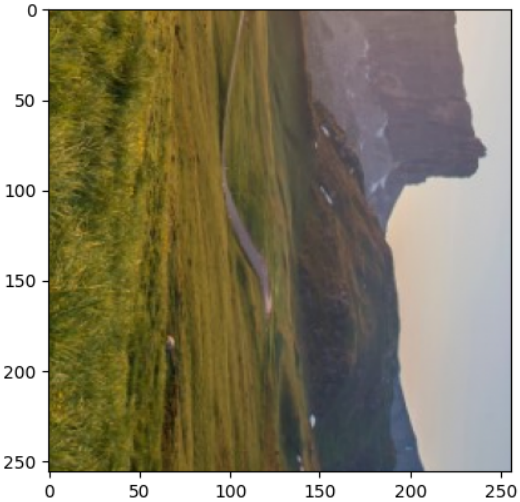
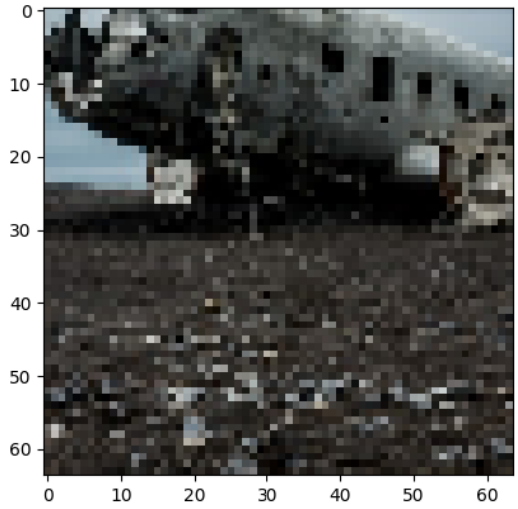
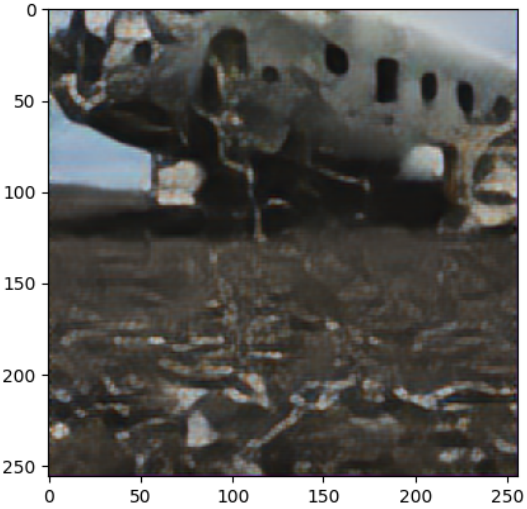
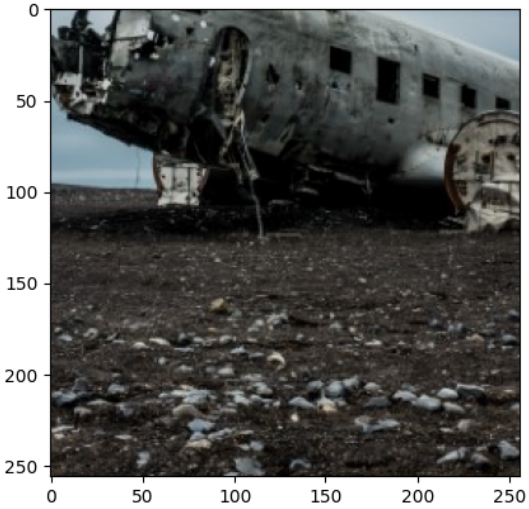
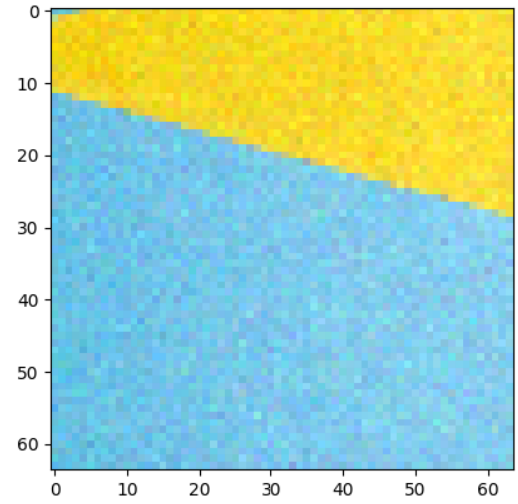
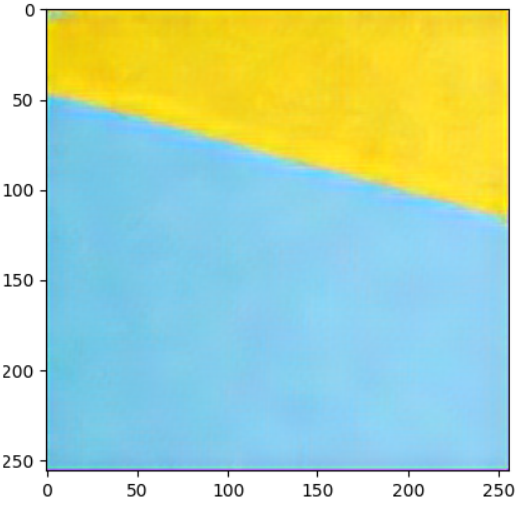
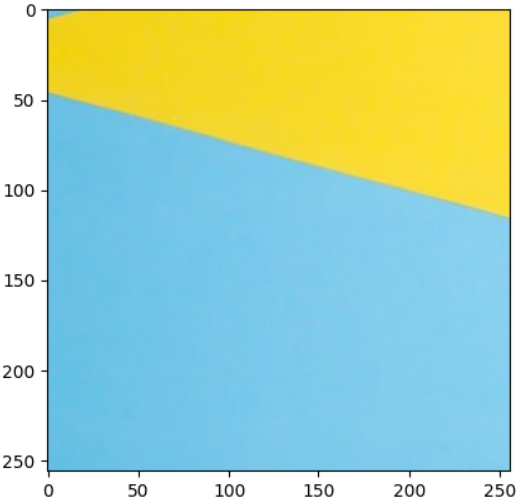




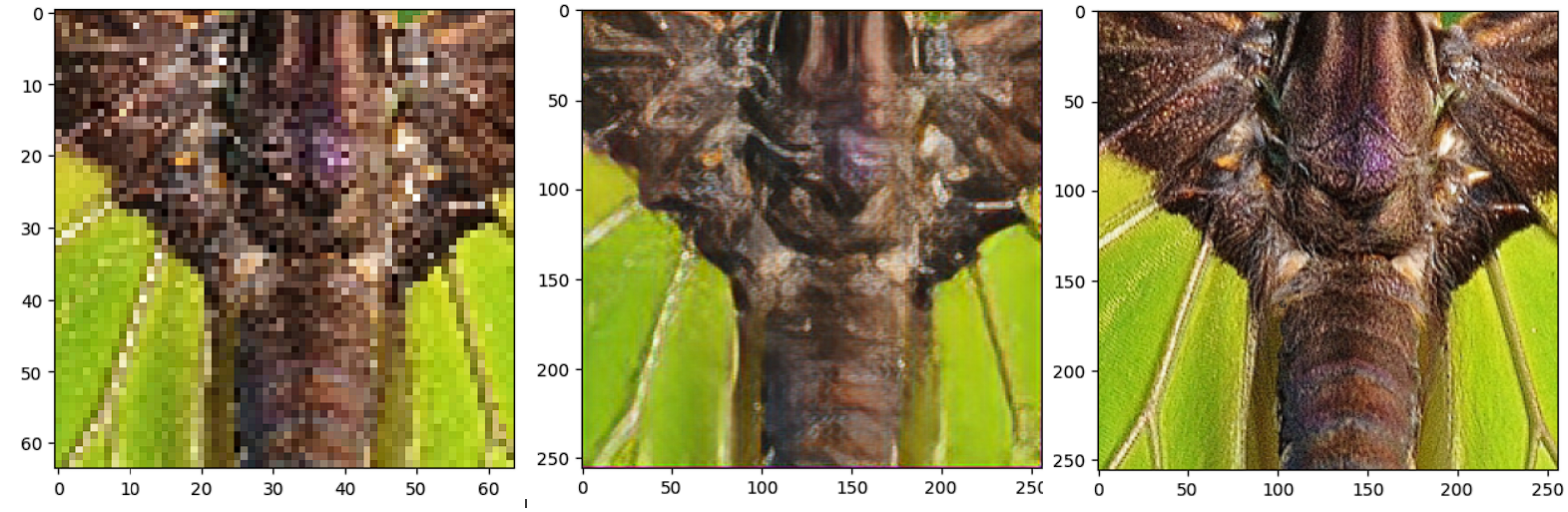


7 - Between the alternative implementations, the potentially best one was found to be an architecture using VGG extractor to calculate the loss in a similar way to the SRGAN perceptual loss.
The outputs were not denoised as well as the results of the proposed model, and this leads also to lower metric values. However it pretends to produce higher detailed and contrasted images even if this can generate some weird artifacts.
The model architecture exploits the same dilated convolutional layers of the proposed one.

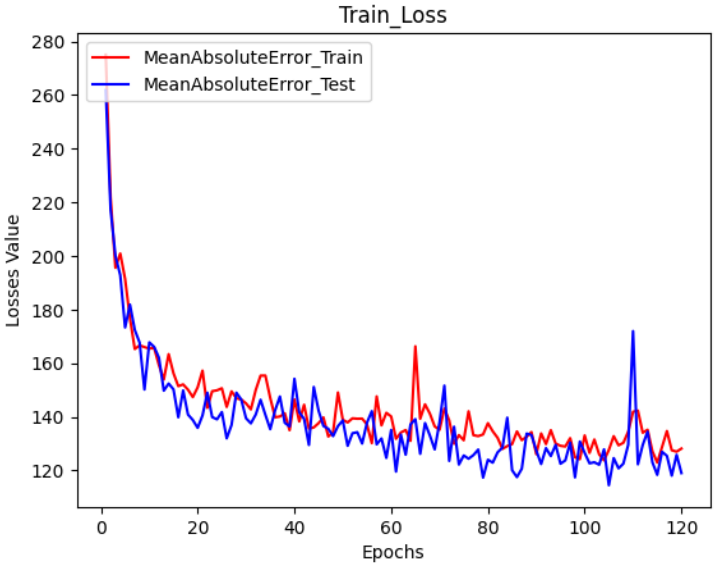
Results:
The following results were produced during the test phase after 20-50-120 epochs.

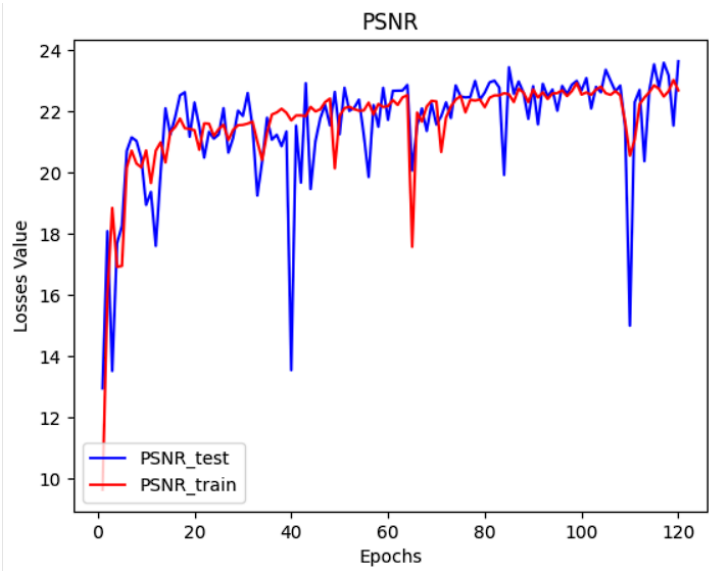










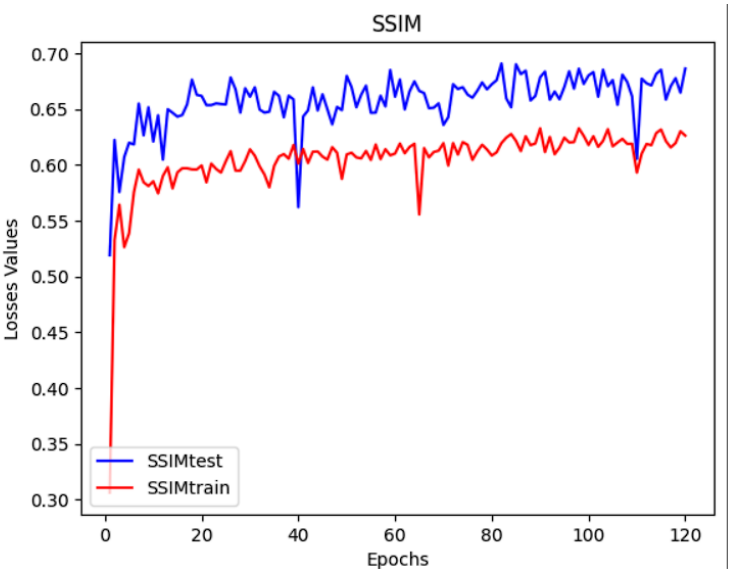
This sequence of images shows the result produced during the validation phase after 120 epochs

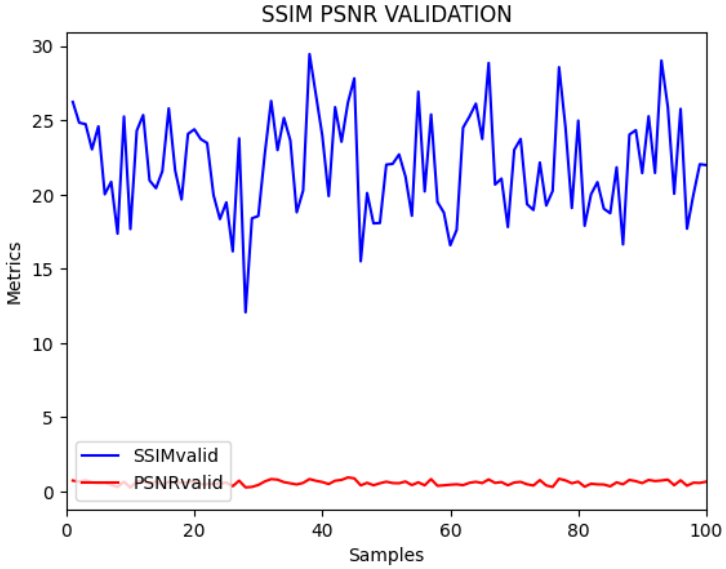


Performance:









Short Overview of the alternative implementations

Implementation	Loss Type	Dilated Convolutional Layer	Fine Tuning	Number of epochs
Proposed	MAE	✅	❌	120
1	MAE	✅	✅	120
2	MAE	✅ (Start and middle)	✅	50
3	MSE	❌	✅	25
4	MAE	❌	✅	50
5	Huber Loss	❌	✅	20
6	MAE	✅	✅	120
7	VGG	✅	❌	120

The shown psnr, ssim and loss are calculated on the test phase.
Even if some architectures obtained good PSNR and SSIM values, the perceptive generated images were not sufficiently good.

Implementation	SSIM- PSNR Epoch 25	SSIM- PSNR Epoch 50	SSIM- PSNR Epoch 120
Proposed	0.71 | 23.95	0.74 | 22.8	0.75 | 25.5
1	0.71 | 23.9	0.72 | 22.8	0.75 | 25.5
2	0.69 | 23.4	0.63 | 19.5	❌
3	0.68 | 22.9	❌	❌
4	0.71 | 23.7	0.71 | 23.5	❌
5	0.7 | 23.78	❌	❌
6	0.72 | 23.75	0.72 | 24.7	0.74 |25.2
7	0.65 | 21.23	0.68 | 21.24	0.68 |23.6

Experiments

During the development of the project, a number of experiments were performed, often following already existing architectures.
Below all the experiments which did not produce satisfactory results are presented.

-SurdCNN

```
def makesurdcnn(input_shape):
# Define the architecture of SuRDCNN

  inp = Input(shape=input_shape)
  model = Sequential()

  # Convolutional layers
  model.add(Conv2D(64, (3, 3), padding='same', input_shape=input_shape))
  model.add(Activation('tanh'))
  for _ in range(2, 21):
      model.add(Conv2D(64, (3, 3), padding='same'))
      model.add(BatchNormalization())
      model.add(Activation('tanh'))

  # Output layer
  model.add(Conv2D(3, (3, 3), padding='same'))
  xoutModel = model(inp)
  xout = keras.layers.subtract([inp, xoutModel])
  return Model(inp,xout)
```

-Another experiment implemented an architecture which used ResNet to perform denoising.

```
def resBlock(input_tensor, num_channels):
  conv1 = Conv2D(num_channels,(3,3),padding='same')(input_tensor)
  relu  = Activation('relu')(conv1)
  conv2 = Conv2D(num_channels,(3,3),padding='same')(relu)
  add   = Add()([input_tensor, conv2])

  output_tensor = Activation('relu')(add)
  return output_tensor

def build_resnet_model(height,width,num_channels,num_res_blocks):
  inp       = Input(shape=(height,width,3))
  conv      = Conv2D (num_channels,(3,3),padding='same')(inp)
  block_out = Activation('relu')(conv)

  for i in np.arange(0,num_res_blocks):
      block_out = resBlock(block_out, num_channels)

  conv_m2   = Conv2D (3,(3,3),padding='same')(block_out)
  add_m2    = Add()([inp, conv_m2])
  model     = Model(inputs =inp , outputs = add_m2)

  return model
```

- Also an autoencoder with the idea of producing an encoding using highly dimensioned kernels (exploiting dilation rate) to produce multiple output, the first trained to get the same encodings from the high resolution and the low resolution image, the second to produce an high resolution image from the produced encoding.

```
def Autoencoder(input_shape=(None, None, 3)):
  inlayer = tf.keras.Input(shape=input_shape)

  x = tf.keras.layers.Conv2D(filters=64, kernel_size=9, padding="same")(inlayer)

  for idx,n_channels in enumerate([64,128,256]):
    x = tf.keras.layers.Conv2D(filters=n_channels, kernel_size=3, padding="same", activation="relu", use_bias=False)(x)
    x = tf.keras.layers.BatchNormalization(momentum=0.8)(x)
    x = tf.keras.layers.Conv2D(filters=n_channels, kernel_size=3, padding="same", activation="relu", use_bias=False)(x)
    x = tf.keras.layers.BatchNormalization(momentum=0.8)(x)
    x = tf.keras.layers.Conv2D(filters=n_channels*2, kernel_size=9, dilation_rate=4)(x)

  encoding = x # first trained output

  for idx,n_channels in enumerate([256,128,64]):
    x = tf.keras.layers.Conv2DTranspose(filters=n_channels*2, kernel_size=9, dilation_rate=4)(x)
    x = tf.keras.layers.Conv2DTranspose(filters=n_channels, kernel_size=3, padding="same", activation="relu", use_bias=False)(x)
    x = tf.keras.layers.BatchNormalization(momentum=0.8)(x)
    x = tf.keras.layers.Conv2DTranspose(filters=n_channels, kernel_size=3, padding="same", activation="relu", use_bias=False)(x)
    x = tf.keras.layers.BatchNormalization(momentum=0.8)(x)

  image = tf.keras.layers.Conv2D(filters=3, kernel_size=9, padding="same", activation="sigmoid")(x)
  return tf.keras.Model(inputs=[inlayer], outputs=[encoding, image])
```

- Lastly a ResNet-based GAN using tanh activations was tested to allow the learning of corrections applying [-1,1] corrections to the [0,1] input image

```python
def Generator(input_shape=(None,None,3), scale_factor: int = 4):
  inlayer = tf.keras.Input(shape=input_shape)

  x = inlayer
  x = tf.keras.layers.Conv2D(filters=64, kernel_size=9, padding="same")(x)
  x = tf.keras.layers.BatchNormalization(momentum=0.8)(x)
  x = tf.keras.layers.PReLU(shared_axes=[1,2], alpha_initializer=tf.keras.initializers.constant(0.2))(x)
  for _ in range(16):
    checkpoint = x
    x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, padding="same")(x)
    x = tf.keras.layers.BatchNormalization(momentum=0.8)(x)
    x = tf.keras.layers.PReLU(shared_axes=[1,2], alpha_initializer=tf.keras.initializers.constant(0.2))(x)
    x = tf.keras.layers.Add()([x, checkpoint])

  x = tf.keras.layers.Conv2D(filters=3, kernel_size=3, padding="same", activation="tanh")(x)
  x = tf.keras.layers.Add()([inlayer, x])

  clean = tf.keras.layers.Conv2D(filters=3,kernel_size=1, activation="sigmoid")(x)
  scaled = tf.keras.layers.UpSampling2D(size=4)(clean)
  return tf.keras.Model(inputs=[inlayer], outputs=[scaled])
```

```python
def Discriminator(extractor, input_shape=(256,256,3)):
  inlayer = tf.keras.Input(shape=input_shape)
  feat_extractor = tf.keras.Sequential(extractor.layers[:])
  feat_extractor.trainable=False

  # Feature Extraction
  x = feat_extractor(tf.keras.applications.vgg19.preprocess_input(inlayer*255.))

  # Real/Fake Classifier
  x = tf.keras.layers.Flatten()(x)
  x = tf.keras.layers.Dense(units=2048)(x)
  x = tf.keras.layers.LeakyReLU(alpha=0.2)(x)
  x = tf.keras.layers.Dense(units=1024)(x)
  x = tf.keras.layers.LeakyReLU(alpha=0.2)(x)
  x = tf.keras.layers.Dense(units=1, activation="sigmoid")(x)

  return tf.keras.Model(inputs=[inlayer], outputs=[x])
```

Folders and Files Structure

The project is split into two main folders:

- SRGANs → contains the 500 epochs trained SRGAN implementation and its variants trained to 135 epochs
    - 500EPOCHS → notebook and .zips containing the models, the metrics, and the plotted results
    - Mixed/MSE/VGG → notebook of the implementation, notebook with the plots, and zip files with the generated images, the .h5 models and the metrics
- Denoiser → contains all the alternative implementations of the Denoiser Architecture.
  For all the implementations the “results” subfolder contains images and a “plots” folder where are located the plots of the performances for the variant.
    - 0-Proposed → the main implementation of the Denoiser Architecture containing the notebook of the implementation and the .zip with models, loss metrics and some plotted results
    - All the following are renamed with their own index corresponding to the one in the table above and contain the notebook of the implementation and the .zip with models, loss metrics and some plotted results