Image Enhancement Using Deep Learning

Bassil Usama El-Oraby

Table of Contents

1) Introduction:

1.1) Abstract:

In this report, we will be taking a look at how to approach the problem of image enhancement using deep learning techniques, Image Enhancement is the process of adjusting digital images so that the results are more suitable for display or further image analysis. For example, you can remove noise, sharpen, or brighten an image, making it easier to identify key features, in this project, I've chosen the SRCNN model for reasons which will be discussed in the report sections below, Our goal is to enhance images of ids in order to be able to extract information from it using OCR Models, a family of deep learning models that can tackle such a problem are super resolution networks, we will be looking at what they are in detail in the following sections.

1.2) Related Work:

Super resolution networks are deep learning models that aim to enhance the quality and resolution of low-resolution images. SISR (Single-Image Super-Resolution) models are models that take a single low-resolution image as input and generate a high-resolution image as output, without using any external information. They usually rely on residual learning, skip connections, and dense connections to improve the performance and stability of the network. Some examples of SISR models developed are:

- EDSR: Enhanced Deep Super-Resolution Network, which removes unnecessary modules in conventional residual networks and expands the model size.
- MDSR: Multi-Scale Deep Super-Resolution System, which can reconstruct high-resolution images of different upscaling factors in a single model.
- SRCNN: Super Resolution Convolutional Neural Network, which learns an end-to-end mapping between low resolution and high resolution images
- ESPCN: Efficient Sub-Pixel Convolutional Neural Network, is a deep learning model for single image super-resolution that uses an efficient sub-pixel convolutional layer to increase the resolution at the end of the network, reducing the computational complexity and memory cost.
- SRResNet: Super Resolution Residual Network, is a deep learning model that uses a residual learning framework for superior performance in image super-resolution. It employs a perceptual VGG loss function to recover fine texture details, offering an improvement over the previously used MSE loss.
- LapSRN: Laplacian Pyramid Super-Resolution Network, is a model that progressively reconstructs the sub-band residuals of high-resolution images from coarse-resolution feature maps.
- VDSR: Very Deep Super Resolution, is a deep learning model for image enlargement that uses a very deep convolutional network with 20 weight layers.
- DRRN: Deep Recursive Residual Network, is a deep learning model for image super-resolution that uses recursive learning and residual learning to handle different scales of images efficiently.

These models will help us in performing the task at hand efficiently, it's up to the goals and constraints required by the task that we will start choosing which model of those to rely on in our work for the rest of the project, we will be assessing these factors in the model selection section below.

2) Entering the pipeline:

2.1) Data inspection and importing:

In this very first stage, I started looking at the data in order to gain some preliminary insights and information and what I found is the following:

1. The data comes in two folders which are: blurred_images and sharped_images, and since this is the whole dataset and given the research i've done which I'll be discussing in the model selection phase, I figured out these two folders represent a full train/val/test sets and their associated target sets (including test , you can even understand the connection given the filenames standard codes "f.Z.fornt.XXXX" in both folders.

2. Dataset consists of 4060 x 2 images divided 4060 each in a folder and they are categorized as follows:

   2.1) 2030 images from "f.h.fornt.5000" to "f.h.fornt.7030" and 2030 similar images with a different naming from "f.v.fornt.5000" to "f.v.fornt.7030" , for the naming convention, from my research according to the super resolution networks context we're working within, this could mean something along the lines of :

   2.1.1- "high" and "very high", meaning different degrees of sharpening or blurring.

   2.1.2- "high" and "very high" relating to the resolution of the images

   2.2) 4060 images are sharped images while the other 4060 images are blurred images

   2.3) the sizes of the images are not standardized; every image has a different width and height values.

   2.4) Some images are way too hazy or too bad of a quality to be considered in our training or testing process.

As for the next step and after an initial data inspection, I started importing the data into 2 lists which we'll be using later for the EDA and train/val/test split stages, also, I've standardized the sizes of all images to 256*256 pixels so as to be able to work with it in later stages of the pipeline (regarding this resizing part, i think it might have not been the best move to resize it to 256*256, as it makes the width too much compact and this may lead to some details loss that could be leading to non-efficient results during the training of the later network), also normalized all images to the range of [0:1] as it leads to better accuracy while training and is better suited for visualizations.

2.2) Exploratory data analysis:

In the second stage of our pipeline, we'll start exploring our data, looking for some deeper insights that could help us in our pre-processing and training stages, the steps for this stage are as follows:

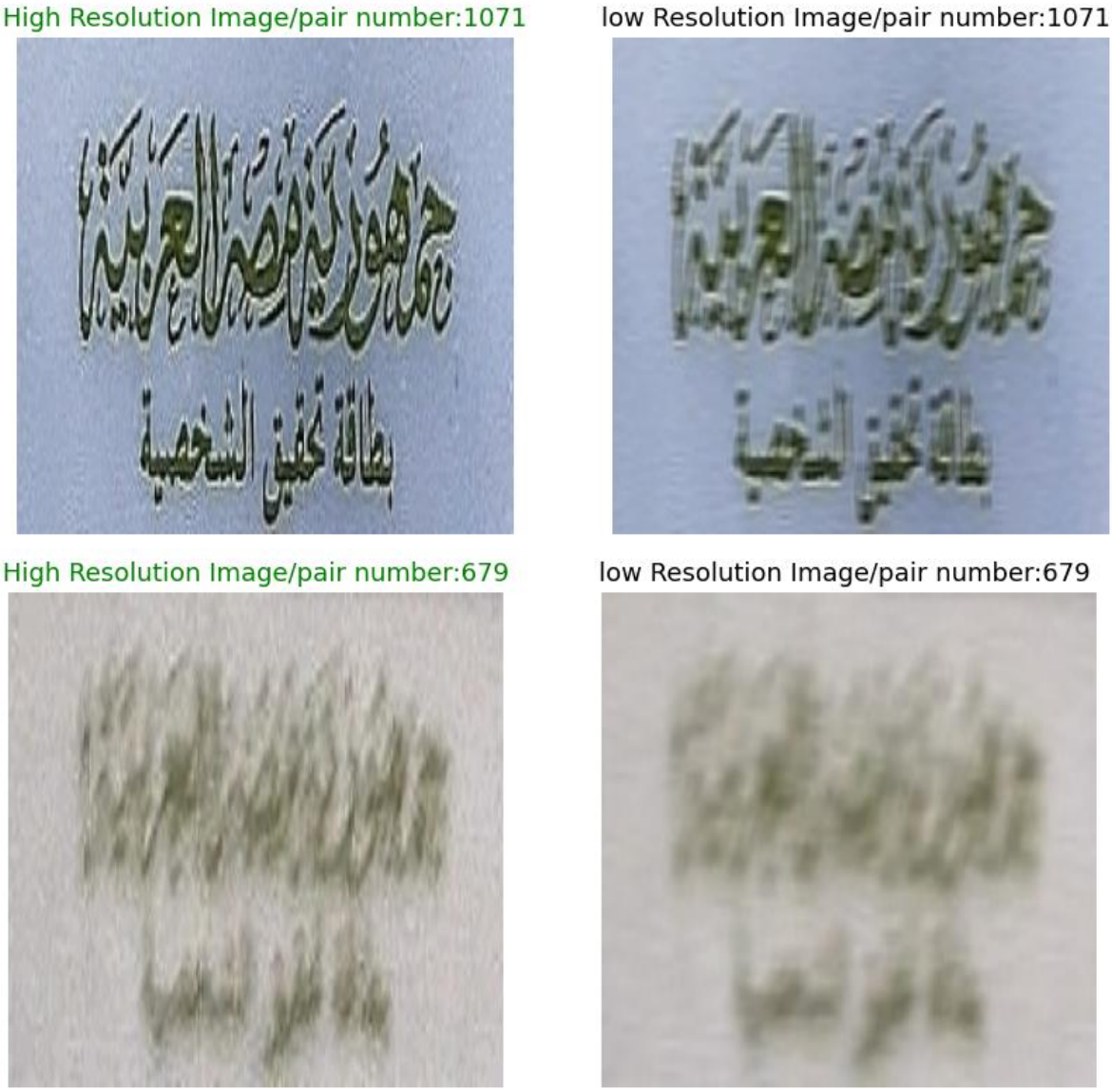
1. Visualization of our dataset:

High Resolution Image/pair number:72

low Resolution Image/pair number:72

High Resolution Image/pair number:82

low Resolution Image/pair number:82

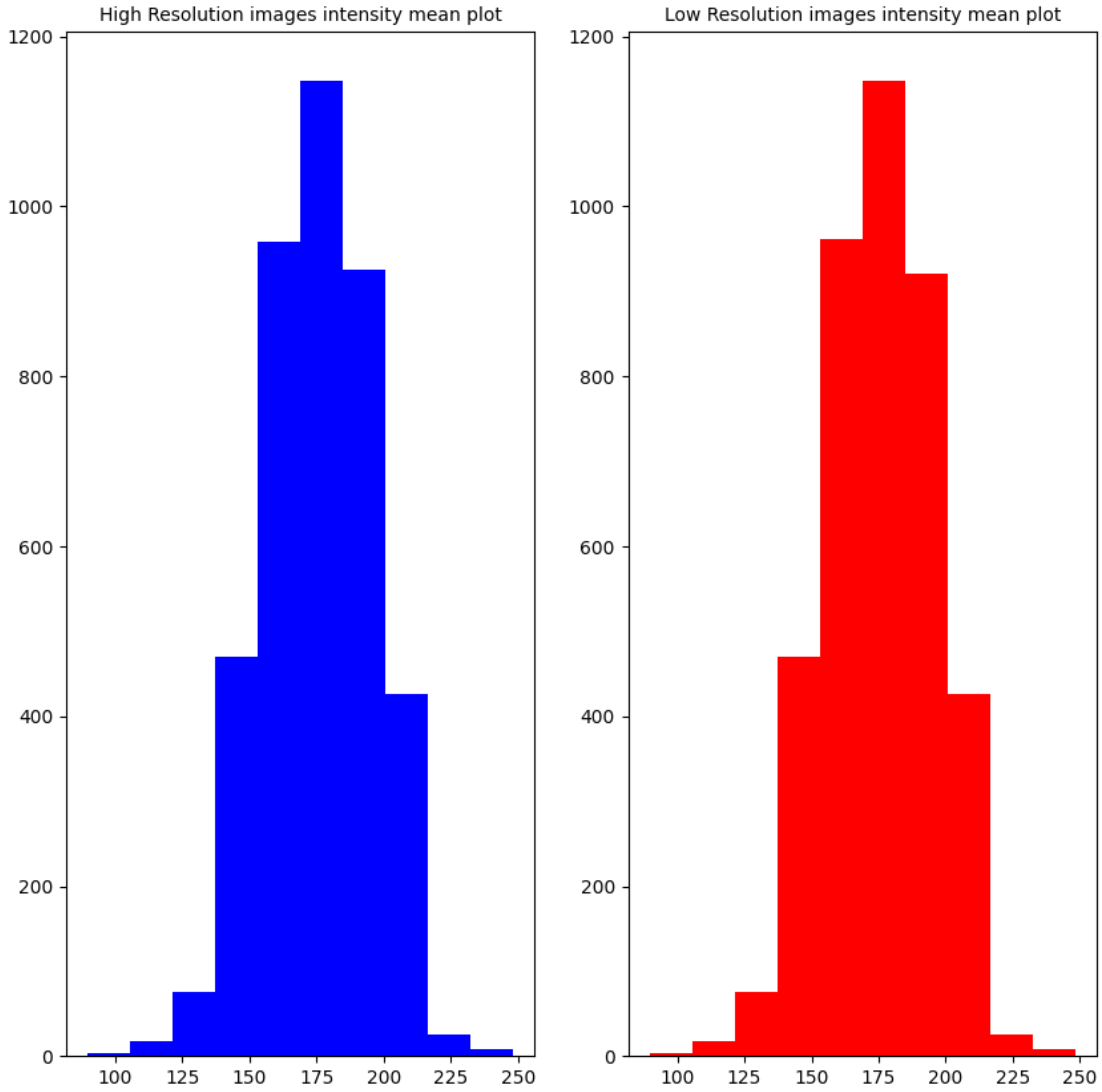


In this dataset, Various kinds of noises are introduced as you can see in the previous visualization of the dataset, including blurring, circular noises, salt and pepper, variations in the background color of the images and also numerous training pairs that need to be thrown away for they are too bad or too good to be used and will degrade the model's training performance, as for the variations in the background color of the images, not sure if this would be considered as a problem during training but surely i've kept an eye on it.

2. Maximum/Minimum/Mean/Median:

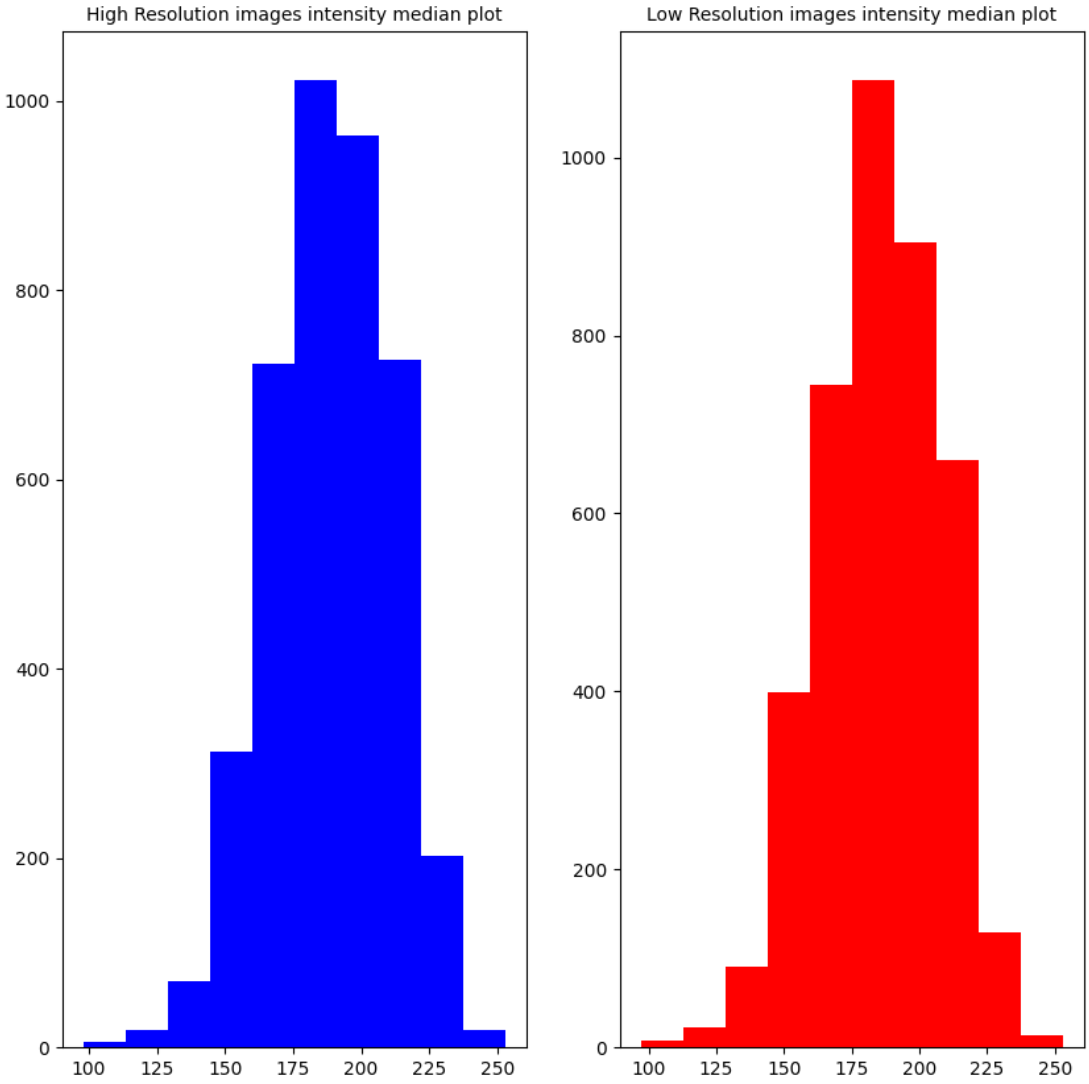
In this step we look at the Maximum, Minimum, Mean & Median values of each image of our dataset:

2.1. Mean intensity plots for HR/LR images:



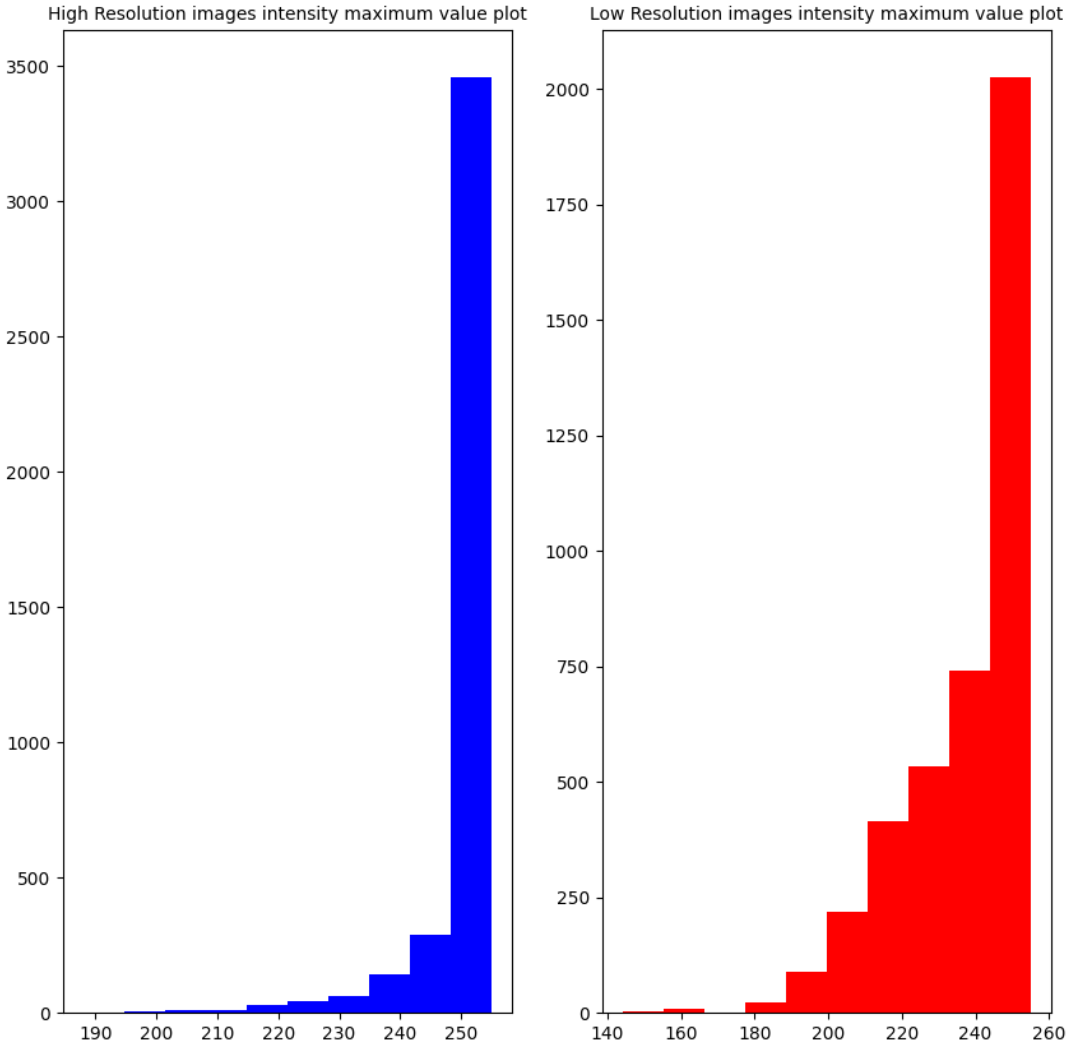
We can see in this plot that the images intensity distribution is somewhat skewed a bit to the upper side of the intensity spectrum [0:255], with a mean of 175.54358 and standard deviation of 20.22926, indicating that the images are more on the bright side.

2.2. Median intensity plots for HR/LR images:

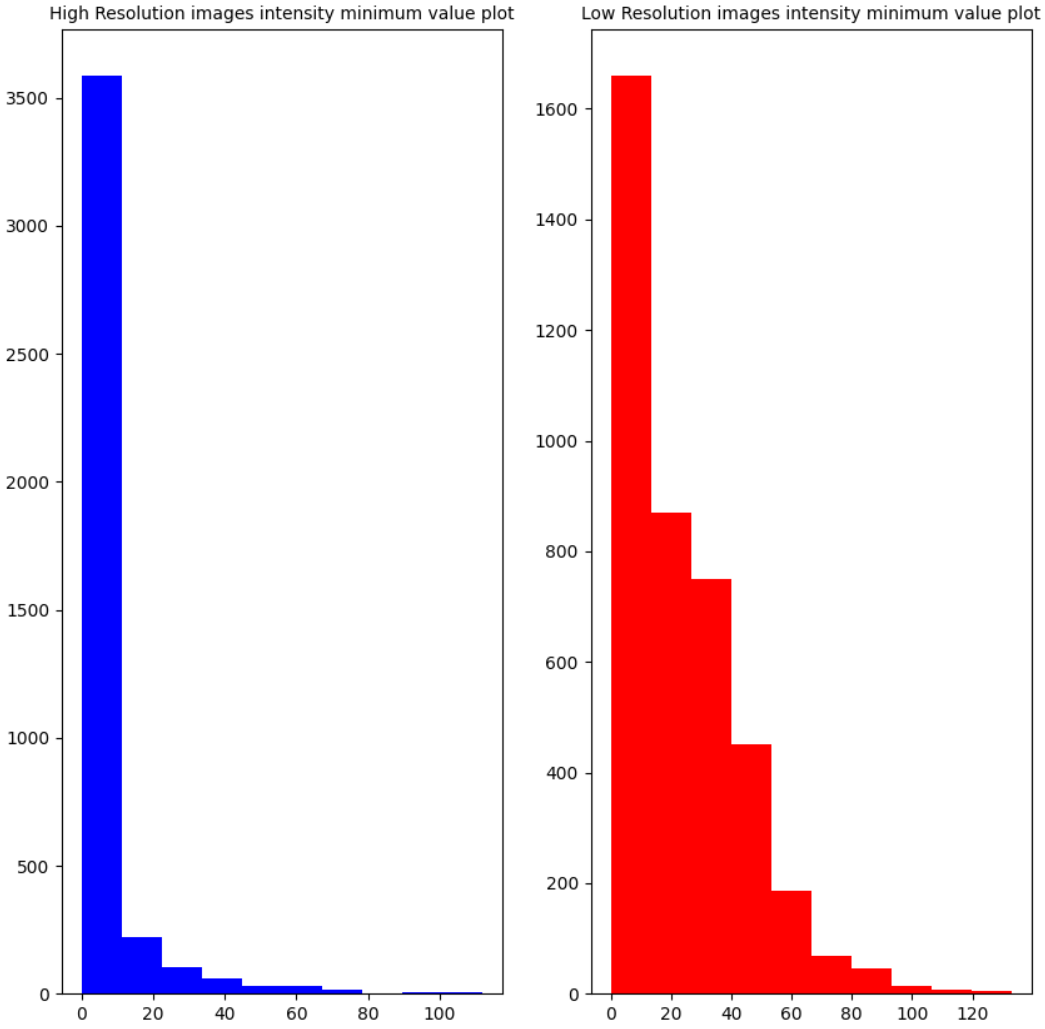


This plot consolidates our previous conclusion that the images are more on the bright side of the intensity spectrum as the median of these images collectively skews the intensities distribution even more to the right side of the graph as it can be seen above.

2.3. Maximum intensity plots for HR/LR images:



2.4. Minimum intensity plots for HR/LR images:



2.3) Train/Val/Test Split:

In this stage, we work on splitting our data into training, validation and testing sets, before going into the details of the sets, I'd like to justify a point I've done during my work, in this stage in my notebook, I've worked with only 855*2 of the 4060*2 images in our dataset, the problem mainly was due to memory usage issue that happens when loading the whole dataset into the network for training as one whole batch, the solution to such a problem is using a custom data generator or a dataloader, simply it **yields** a part of the dataset (a **batch**) to the network for training, and keeps handing it the training set batch by batch, this way we can train the neural network on the whole dataset without running out of RAM memory space and crashing, in the following subpoints i specify the splits I've done :

1. Training Set (X_train, y_train): i took 700*2 images for training.

2. Validation Set (X_val, y_val): took 130*2 images for validation of the model, Keep in mind that the research paper authors had already set a standard and advocated for a set of hyperparameters as we will discuss in the model selection section to be the best for operating the

model, namely the number of channels or kernels in each convolutional layer, the receptive fields of the kernels in each layer, and the number of convolutional layers in the SRCNN model, of course this is up to debate and nothing in the field of ML/DL is definite and maybe if we experimented enough we'd find different results as no one size fits all, but as a general heuristic and given the time interval within which i'm working, i decided to go with the findings of the research paper.

3. Test Set (X_test, y_test): As for the test set, i decided to go with 25 photos.

Also one more thing I'd like to note, as a standard, it is better to start with the train/val/test splits before performing any preprocessing operations, this is due to a fear of the risk of data or information leakage, information leakage is the phenomenon of using information in the model training process that would not be available or reliable at prediction time, leading to invalid or optimistic evaluation of the model performance. Data leakage can occur in various ways, the main way we're trying to avoid here is preprocessing or transforming the data using information from the whole dataset or the test set, which can leak statistical properties or distributional characteristics of the data, but in our case, this is not a problem as the preprocessing we'll be doing does not contain any kind of information mixing between the different samples in our dataset.

2.4) Pre-processing:

In this Stage, we will be going through multiple subpoints, discussing what worked and what didn't, why i've chosen such approaches and how they turned out to be during training and evaluation phase.

In my appproach, i tended to make the label images (high_img array) as clear as possible with a good contrast ratio between the foreground writings and the plain background of the ids, so that the mapping between the LR Images and the HR ones could be as easy as possible for the neural net, and this was clearly seen after performing some preprocessing on the HR set as it reflected in the strong minimization of the cost function of the model.

I've also tried a couple of techniques on the LR Dataset but with no significance in results, mostly what happened was the opposite, a greater degradation in the percepted quality of the output images from the network and in the PSNR and MSE Values.

All the techniques that i tried were as follows:

1) **Denoising:** Images in this dataset had alot of noise, and it had to be removed, i used non-local means denoising for this objective and it worked quite well as you can see in the following visualization, did the same for the LR Images but gave bad results and i specify in the following section about denoising.

2) **Sharpening:** I've tried sharpening the HR Images, but the result was not significant, gave a better output visually for the HR Images but as for the evaluation of the network outputs, the results nor the metrics differed by much, so i decided not to go with it.

3) **Weighted addition:** Tried performing a weighted addition between the sharpened HR Images and the smoothed denoised ones, results were pretty good visually but there were two problems, first the background and even some pixels in the foreground writings overflowed due to the pixels' values exceeding 255 and showing white pixels, and the second is a consequent of the first as the neural network's outputs' exposure was very high in some images due to it learning from the overly skewed Images Histograms towards the very high end of the intensity's spectrum.

4) **Histogram Equalization:** Tried using Contrast Limited Adaptive Histogram Equalization (CLAHE) for Stretching the HR Images Histogram for maybe it could give better results while training by stretching out the histogram of intensities of the image, but it wasn't good either as it dimmed out the image and decreased the differences between the foreground and background which is not the goal.

the 3 techniques which i did not go with, i had code for them in the notebook which I've provided but because you asked for a clean code structure, i removed them as they are of no use but i do have them in my local notebook for your reference if you'd like.

Also, Other techniques that i still did not try but i believe they can enhance the training and evaluation of the model are:

1) **Data Cleaning:** I've noticed that many images in the dataset are either way too bad or way too good to be used in the training process, but given the time constraint, i decided to go with the other approaches.

2) **Removing noise from HR and LR Images in the frequency domain:** I've noticed that some images still have a kind of a repeating noise that is not salt and pepper but a uniform kind of noise, which lead me to the idea that maybe if i transform the images into the frequency domain, maybe i could find a way to suppress these noise components over there and transform the enhanced images back to the spatial domain.

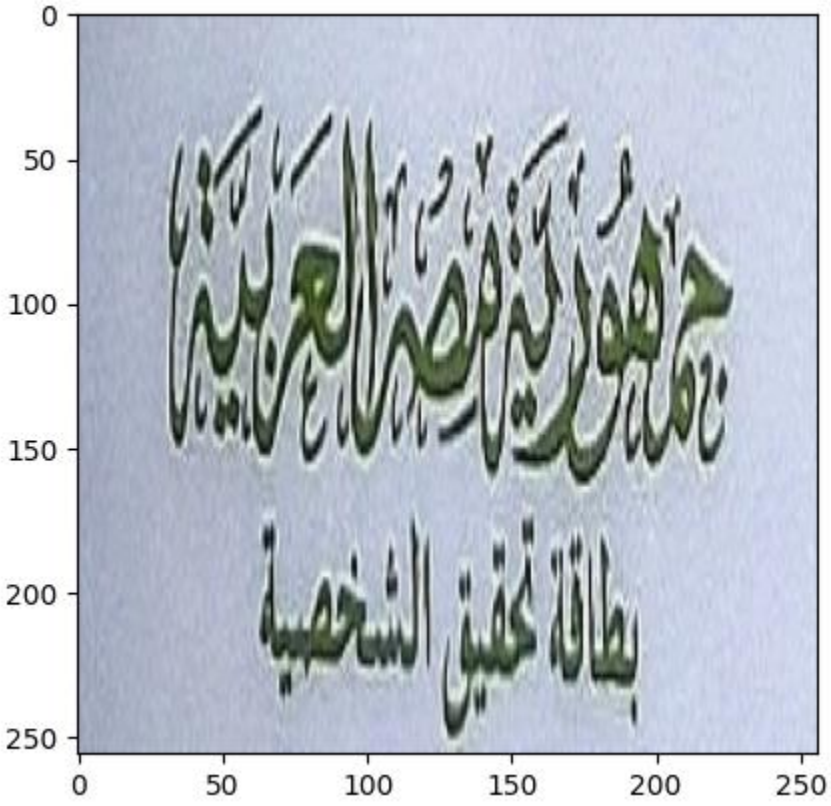
Denoising:

Non-local means denoising is a technique for removing noise from images while preserving the details and textures. It works by comparing small patches of pixels around each pixel and averaging the values of similar patches. This way, the pixels that belong to the same texture or structure are smoothed together, while the pixels that belong to different features are kept distinct.

It looked like this would be a suitable technique to smooth out the noisy HR images and it did work as you will see in the following visualization, also, i've tried performing the same technique on LR Images lest that i can remove the noise off them and keep them only in their blurred form, but it did actually worsen the image, degrading many of the remaining features of the photo that still defined partly the information in the blurred LR images.

Also, i tried using bilateral filtering for smoothing out the noise in the LR Images while trying to preserve the apparent edges in the blurred images but with no significant result also, noise was still there.

Sample “f.h.fornt.5019”:
Before:



After:

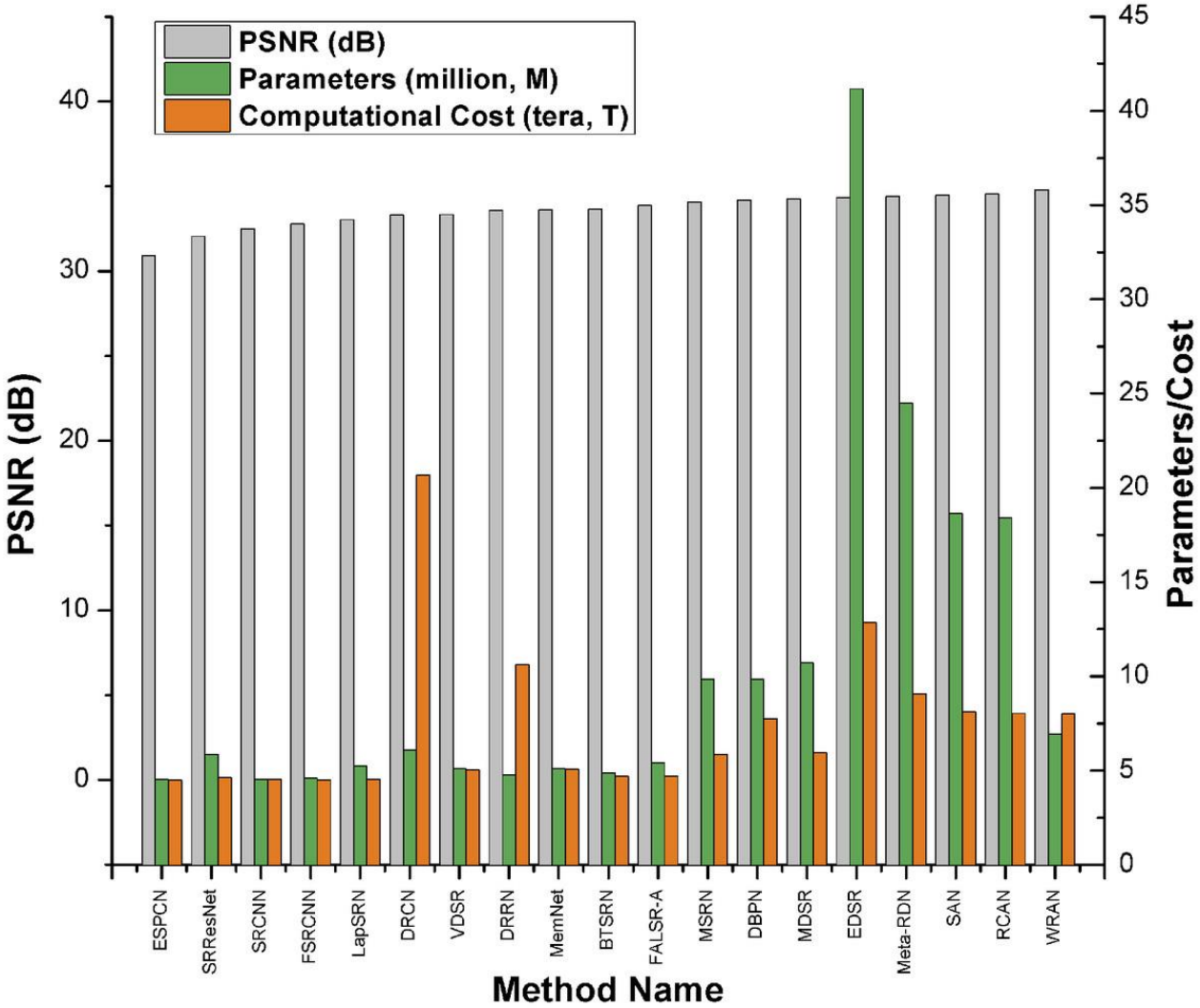
2.5) Model Selection:

In this stage, I started doing my own research on this topic of image enhancement using deep learning methods, what I found mainly are the following families of networks that can solve this problem:

- **Convolutional neural networks (CNNs)**, which are composed of multiple layers of neurons that apply convolution operations to the input image or feature maps. CNNs can learn to extract and combine features from images and perform various tasks, such as image classification, segmentation, detection, etc. CNNs can also be used for image enhancement by learning to map low-quality images to high-quality ones, such as super-resolution, deblurring, denoising, etc.
- **Residual neural networks (ResNets)**, which are a type of CNNs that use skip connections or shortcuts to connect the input and output of some layers. ResNets can overcome the problem of vanishing gradients and improve the performance of deep networks. ResNets can also be used for image enhancement by learning to add residual details to the input image, such as dehazing, deraining, etc.
- **Generative adversarial networks (GANs)**, which are composed of two networks: a generator and a discriminator. The generator tries to produce realistic images that can fool the discriminator, while the discriminator tries to distinguish between real and fake images. GANs can learn to generate high-quality images from low-quality ones or from random noise, such as inpainting, style transfer, colorization, etc.

The goals and constraints with which I was guided and decided to choose which model to use in this project:

- Goals and Constraints:
  - The top priority that I was guided by as stated in the task's description is to build the pipeline till completion and deliver a working solution, so when in my search for a model, I've searched for the simplest model that I can understand and implement quickly, this led me to choosing a model that is a part of the family of the CNNs or ResNets, as I'm more comfortable with such models.
  - Second priority was to get the best results, quite honestly, according to multiple surveys, CNNs as I've chosen later are not bad but do not give the state-of-the-art results as other networks, for example, according to the below graph:



Attention Networks like SAN, RCAN, WRAN do score better results than CNN models on the left part of the graph, yet, as I mentioned before, guided by the top priorities first and foremost, and given the time constraint, I decided to work with what I know best and pretty sure it can give me the good results required for the task.

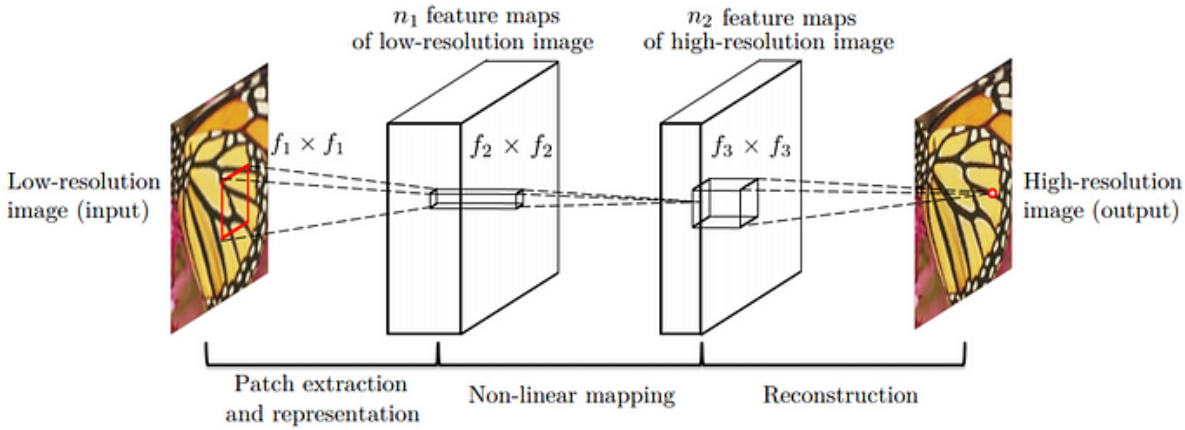
And so, according to the previous justification, I started searching into models that are built on top of CNNs and ResNets if there ever was, what I found was the models which I've mentioned above in section 1.2) Related Work, the best of them being EDSR according to the above graph in terms of outcome, but the worst when it comes to computational complexity and memory usage, nevertheless I started working with this model but I've faced a few implementation issues, so to save time, I resided to a simpler model which is SRCNN(Super Resolution Convolutional Neural Network).

SRCNN:

In this task, I will be implementing the Super Resolution Convolutional Neural Network(SRCNN) by Chao Dong, Chen Change Loy, Kaiming He and Xiaoou Tang.

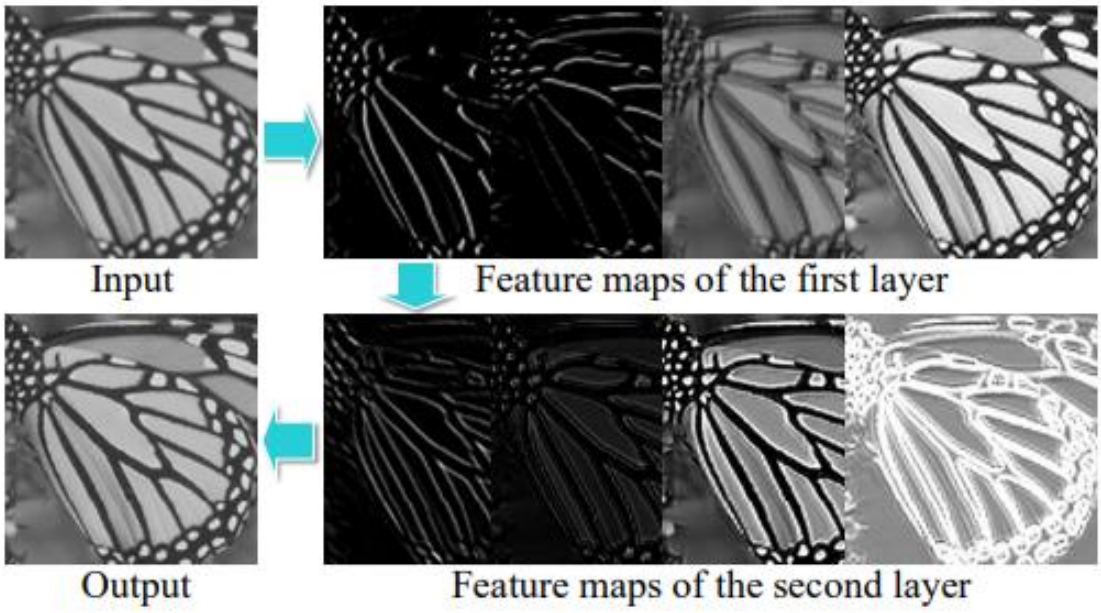
SRCNN proposes a 3-layer CNN for image super-resolution. It is one of the first papers to apply deep neural networks for the task of image super-resolution. The SRCNN architecture is composed of three components: Feature extractor, non-linear mapping, reconstruction. The model is trained to minimize the pixel-wise MSE between the reconstructed and ground truth image.

1. Model Architecture:



The proposed architecture conceptually consists of 3 components: Feature extractor, non-linear mapping, reconstruction. Each is responsible for extracting low-resolution features, mapping into high resolution features, reconstruction. The low-resolution image is an image introduced with blurring and different kinds of noises simulating real situations, let this LR Image be X, with the same size as the high-resolution image, Y. The model aims to learn a mapping F: X->Y.

Conclusively, each component turns out to be represented as one convolution layer, resulting in a 3-layer convolutional neural network, with kernel size 9–1–5. According to the figure below, the intermediate output of each layer seems to contain the necessary information they are expected to compute.



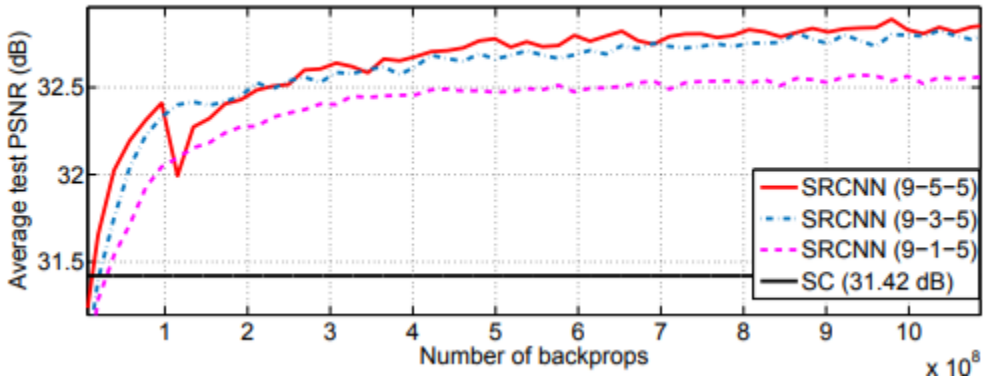
1.2) Loss:

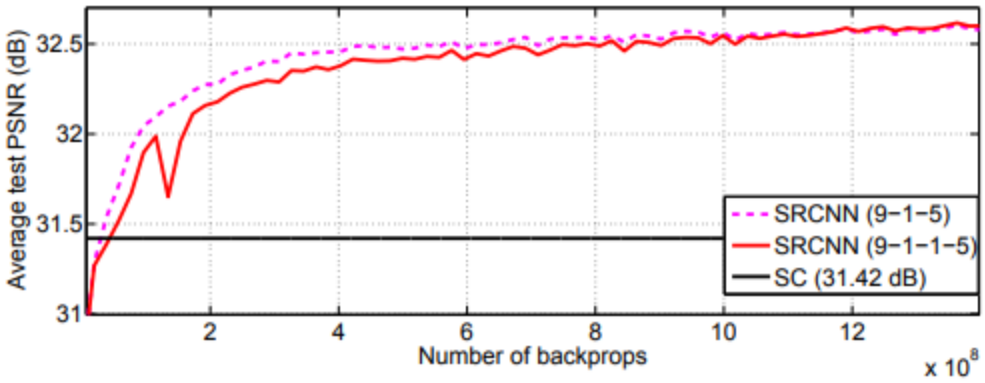
The loss function is defined as the pixel wise MSE(Mean Squared Error) between the reconstructed image F(Y) and the ground truth image X. This will result in training to maximize the PSNR measure.

$$L(\Theta) = \frac{1}{n}\sum_{i=1}^{n} ||F(\mathbf{Y}_i;\Theta) - \mathbf{X}_i||^2,$$

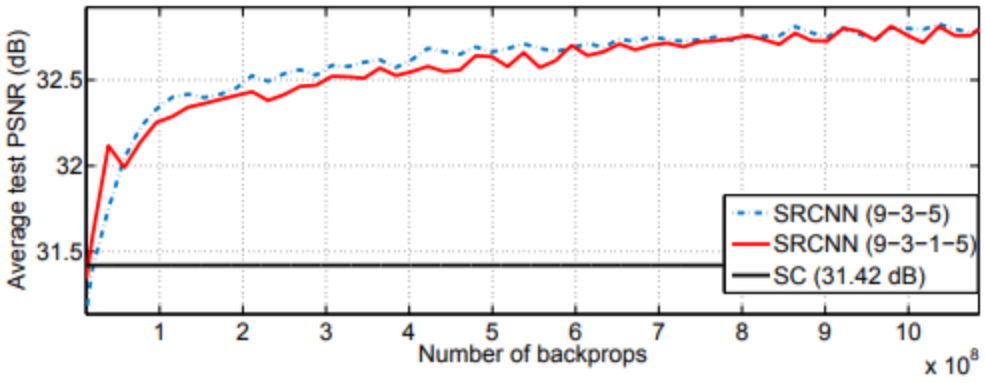
1.3) Experiments:

The paper experiments various hyper-parameter settings to improve performance. The figure below shows how a network channel sizes of 9–1-5 outperforms other settings, and that deeper networks that increase the capability of the non-linear mapping based on the 3-stage methodology of the paper were unnecessary. Although the concept of the proposed model has some known flaws proven to be false by further research in deep learning, the experiments shows the high performance of the proposed 3-stage methodology for SR.

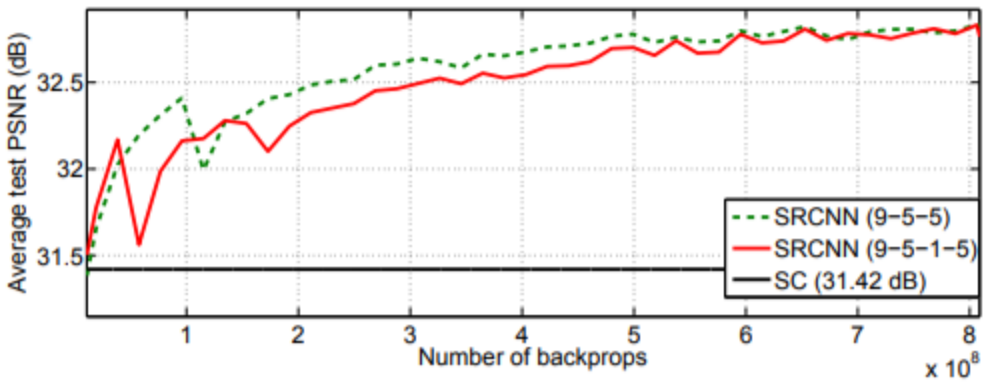




(a) 9-1-5 vs. 9-1-1-5



(b) 9-3-5 vs. 9-3-1-5



(c) 9-5-5 vs. 9-5-1-5

2.6) Model Construction:

In this stage, we'll start building our SRCNN Model using Tensorflow and Keras.

Why did I choose Tensorflow and Keras? for the following reasons:

- Tensorflow and Keras are more accustomed to the industry than PyTorch which is used more heavily on the academic side, and it's because Tensorflow and keras are more simple when it comes to constructing industry-ready models, downside being that they are slower in training compared to PyTorch.

- Most of the projects I've worked on was using TensorFlow and Keras, and so, I decided to go with what I had more experience in.

```
input_img=Input(shape=(256,256,3))
l1=tf.keras.layers.Conv2D(64,9,padding='same',activation='relu') (input_img)
l2=tf.keras.layers.Conv2D(32,1,padding='same',activation='relu') (l1)
l3=tf.keras.layers.Conv2D(3,5,padding='same',activation='relu') (l2)


SRCNN=Model(input_img,l3)
SRCNN.compile(optimizer=tf.keras.optimizers.Adam(0.0008),loss=pixel_MSE)
SRCNN.summary()
plot_model(SRCNN, to_file ='super_res.png',show_shapes=True)
```

In the following part we dissect the above code block line by line, explaining how the model is constructed:

- 1st Line: in the first line, we define the input layer which will be passed to the second layer in our network which is the first hidden layer.

- 2nd Line: in the second line, we start making use of the Keras layer-centric API which is built on top of Tensorflow starting from the 2.0 Version, we utilize a layer which is called "**Conv2D**" layer, This function is used to create a 2D convolution layer, which is a common type of layer for processing images or other grid-like data. A 2D convolution layer applies a set of filters to the input, which can extract features or patterns from the data. The output of the layer is a tensor of the same rank as the input, but with a different number of channels, depending on the number of filters used. in the following subpoints we'll be specifying its parameters one by one:

    - **filters:** This is an integer that specifies the number of output filters in the convolution. Each filter is a tensor of shape (kernel_size, kernel_size, input_channels), where kernel_size is the height and width of the filter, and input_channels is the number of channels in the input. The filters are randomly initialized and learned during training. The output of the layer will have a shape of (output_height, output_width, filters), where output_height and output_width depend on the input shape, the strides, the padding, and the dilation_rate, In this project, I gave it the value of 64 as defined above according to the best practice concluded by the research paper of the model.

    - **kernel_size:** This is an integer or a tuple/list of two integers that specifies the height and width of the 2D convolution window. The convolution window is the region of the input that is multiplied element-wise with the filter and summed up to produce one output value. The kernel_size determines the size of the filter and the receptive field of the layer, which is the area of the input that affects one output value. In this layer, I've defined it to be 9, meaning a 9x9 sized Kernel, this is also in accordance with the research paper findings of best kernel sizes 9-1-5 as stated above in the experimental graphs.

- **padding:** This is a string that can be either "valid" or "same" (case-insensitive). The padding determines how the input is padded with zeros before applying the convolution. If padding is "valid", no padding is applied, and the output shape is smaller than the input shape. If padding is "same", the input is padded such that the output shape is the same as the input shape when strides is (1, 1). The padding affects the output shape and the boundary effects of the layer. In our project, I gave it the value of "same", so that the output tensor would have the same size as the input tensor, keep in mind, that the output shape is already defined by the equation

  $output\_size = \frac{input\_size+2\times padding\_size-dilation\_rate\times(kernel\_size-1)-1}{strides} + 1$, where *input_size* represents the SIZE variable defined in our code that shapes the LR and HR image sizes, *padding_size* is the amount of zero-padding applied to the input, *dilation_rate* is the amount of spacing between the filter elements. If the padding value is "same", the padding size is calculated as:

$$padding\_size = \frac{(output\_size - 1) \times strides + dilation\_rate \times (kernel\_size - 1) + 1 - input\_size}{2}$$

  If the padding value is "valid", the padding size is zero.

- **activation:** This is a string or a callable that specifies the activation function to apply to the output of the layer. The activation function is a nonlinear function that can introduce nonlinearity and complexity to the model. Some common activation functions are "relu", "sigmoid", "tanh", etc. If activation is None, no activation function is applied, and the output is linear. In our case, we'll use the "relu" activation function as it is the proposed function by the research paper and there is a valid justification in general to use relu activations in CNNs, and this is due to that it can cutoff for example negative values occurring from the convolution operations in the previous stages in a convolutional layer, and this is definitely needed as negative values in an image has no meaning.
- **(input_argument):** An input argument which is the previous layer's output or activations is passed as an input to this convolutional layer, in fact what this does is it entwines both of these layers together so that in the model's definition they are always connected in this desired topological order.

- 3rd Line: again after we define the previous layer and store it in l1 variable, we define the second "Conv2D" layer and it's parameters with the same reasoning as the above in the 2nd Line, we connect it with the l1 layer and again store it in a variable l2 to be passed to the next layer.

- 4th Line: Same as above and with this layer we conclude our SRCNN Model definition.

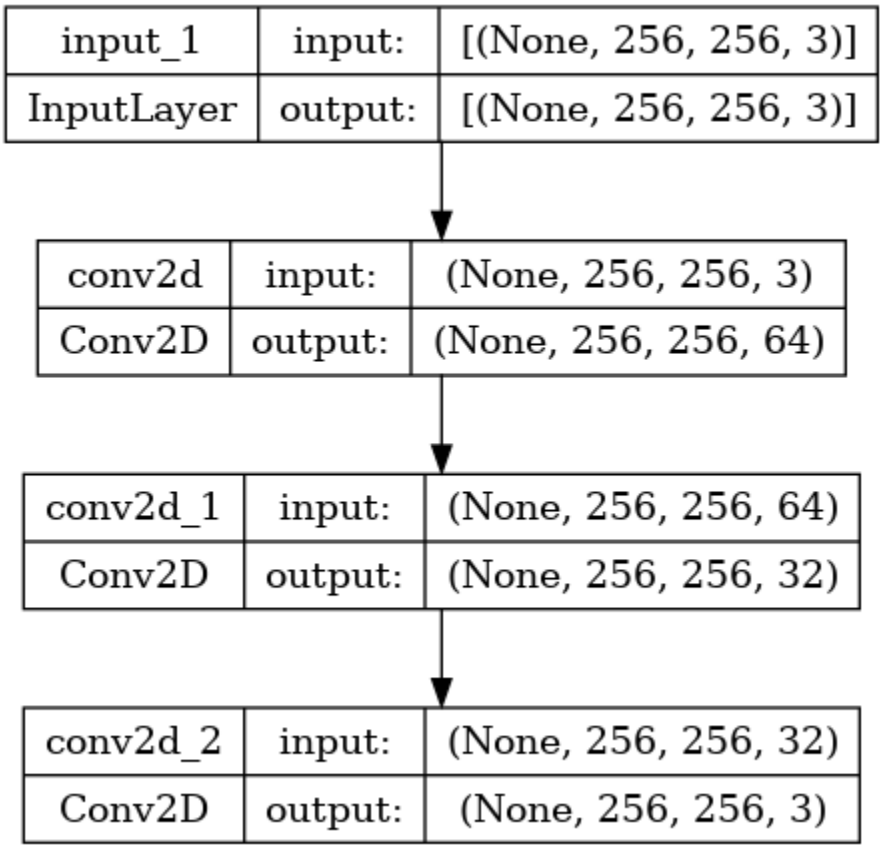
- 5th Line: in this line, we define the tensorflow model using Model() function, passing to it 2 arguments which are the input layer & the output layer, and so we can start using this model for next training step.

- 6th Line: In this line, we start what is called as the "Compilation" of our model, which is used to configure the model for training and evaluation. The compile function takes several arguments, such as optimizer, loss, metrics, etc., that specify how the model will optimize its parameters, measure its performance, and update its gradients. We've defined above the loss function and the optimizer with which we'll be setting the objectives for our model using the loss function definition which is the mean squared error loss function as proposed by the research paper, and we'll be using the ADAM optimizer. One thing I'd like to note here is, such parameters are defined as a result of extensive experimentation and also knowledge of the problem and what functions would suit it best, given more time, I would've done more experimentation on the optimizers side to see if a better algorithm would result in better training performance, but actually as we will see in the following section, ADAM does pretty good on optimizing the model's performance and reaches some fairly good accuracy results, and so I decided to go with it.

- 7th & 8th Lines: in those 2 line, we will be reviewing the summary of the model we've been discussing above, here's the details of our model:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 256, 256, 3)]     0

 conv2d (Conv2D)             (None, 256, 256, 64)      15616

 conv2d_1 (Conv2D)           (None, 256, 256, 32)      2080

 conv2d_2 (Conv2D)           (None, 256, 256, 3)       2403

=================================================================
Total params: 20099 (78.51 KB)
Trainable params: 20099 (78.51 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Out[12]:



2.7) Model Training:

In this stage, we will start training our model and try to get the best results out of it according to the loss and validation metrics defined as we'll see.

```
SRCNN.fit(X_train, y_train, epochs = 40, batch_size = 10,
          validation_data = (X_val, y_val))
```

in this line of code, the **fit()** function is a method of the tf.keras.Model class, which is used to train a model on a given dataset. The fit function takes several arguments, such as the input data, the target labels, the number of epochs, the batch size, the validation data, the callbacks, etc., that specify how the model will learn from the data and evaluate its performance. The fit function also returns a history object that contains the loss and metric values for each epoch. And we specify each argument as the following:

- **Input_data:** So our input data to the model will be X_train, the 700 images which we've defined previously in the train/val/test split. These are the LR images which our model will start trying to map to the corresponding HR label image through finding a mapping between them.
- **Target_labels:** These are the y_train target labels corresponding to X_train images.
- **Epochs:** An epoch, is a traversal over the whole training set, it's an experimental hyperparameter to which I found it best to be around 40 epochs, maybe even less would suffice, such as 30 or 20, as 40 epochs didn't result in a significantly better result than say, 30 epochs.
- **Batch_size:** in this argument, we specify how we divide our training input set into a group of subsets or "batches", working on each batch separately and not on the whole dataset as one batch, and updating the network's weights a number of times equal to *Training_Dataset_size / Batch_size* per epoch.
- **Validation_data:** This parameter receives a tuple of (X_val, y_val) representing the validation subset of the data that we will use to assess what hyperparameters are best for our model and whether we're having a good Bias-Variance ratio or not.

Next, We assess the training performance of our model through the loss and validation error analysis:

```
Epoch 1/40
70/70 [==============================] - 20s 59ms/step - loss: 0.0554 - val_loss: 0.0100
Epoch 2/40
70/70 [==============================] - 4s 52ms/step - loss: 0.0078 - val_loss: 0.0074
Epoch 3/40
70/70 [==============================] - 4s 50ms/step - loss: 0.0060 - val_loss: 0.0062
Epoch 4/40
70/70 [==============================] - 4s 51ms/step - loss: 0.0052 - val_loss: 0.0056
Epoch 5/40
70/70 [==============================] - 4s 51ms/step - loss: 0.0048 - val_loss: 0.0053
Epoch 6/40
70/70 [==============================] - 4s 52ms/step - loss: 0.0046 - val_loss: 0.0050
Epoch 7/40
70/70 [==============================] - 4s 53ms/step - loss: 0.0045 - val_loss: 0.0049
Epoch 8/40
70/70 [==============================] - 4s 53ms/step - loss: 0.0042 - val_loss: 0.0046
Epoch 9/40
```

```
70/70 [==============================] - 4s 53ms/step - loss: 0.0040 - val_loss: 0.0044
Epoch 10/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0042 - val_loss: 0.0044
Epoch 11/40
70/70 [==============================] - 4s 53ms/step - loss: 0.0038 - val_loss: 0.0042
Epoch 12/40
70/70 [==============================] - 4s 53ms/step - loss: 0.0036 - val_loss: 0.0041
Epoch 13/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0035 - val_loss: 0.0041
Epoch 14/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0034 - val_loss: 0.0039
Epoch 15/40
70/70 [==============================] - 4s 56ms/step - loss: 0.0034 - val_loss: 0.0040
Epoch 16/40
70/70 [==============================] - 4s 53ms/step - loss: 0.0033 - val_loss: 0.0037
Epoch 17/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0032 - val_loss: 0.0036
Epoch 18/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0031 - val_loss: 0.0037
Epoch 19/40
70/70 [==============================] - 4s 57ms/step - loss: 0.0031 - val_loss: 0.0035
Epoch 20/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0030 - val_loss: 0.0036
Epoch 21/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0030 - val_loss: 0.0035
Epoch 22/40
70/70 [==============================] - 4s 56ms/step - loss: 0.0029 - val_loss: 0.0034
Epoch 23/40
70/70 [==============================] - 4s 57ms/step - loss: 0.0029 - val_loss: 0.0035
Epoch 24/40
70/70 [==============================] - 4s 57ms/step - loss: 0.0029 - val_loss: 0.0033
Epoch 25/40
70/70 [==============================] - 4s 57ms/step - loss: 0.0029 - val_loss: 0.0034
Epoch 26/40
70/70 [==============================] - 4s 57ms/step - loss: 0.0029 - val_loss: 0.0033
Epoch 27/40
70/70 [==============================] - 4s 58ms/step - loss: 0.0029 - val_loss: 0.0032
Epoch 28/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0028 - val_loss: 0.0034
Epoch 29/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0029 - val_loss: 0.0033
Epoch 30/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0028 - val_loss: 0.0032
Epoch 31/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0028 - val_loss: 0.0032
Epoch 32/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0028 - val_loss: 0.0032
Epoch 33/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0028 - val_loss: 0.0033
Epoch 34/40
70/70 [==============================] - 4s 54ms/step - loss: 0.0027 - val_loss: 0.0032
Epoch 35/40
70/70 [==============================] - 4s 57ms/step - loss: 0.0027 - val_loss: 0.0031
Epoch 36/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0027 - val_loss: 0.0032
Epoch 37/40
70/70 [==============================] - 4s 56ms/step - loss: 0.0027 - val_loss: 0.0031
```

```
Epoch 38/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0028 - val_loss: 0.0032
Epoch 39/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0027 - val_loss: 0.0032
Epoch 40/40
70/70 [==============================] - 4s 55ms/step - loss: 0.0027 - val_loss: 0.0031
```

[10]:

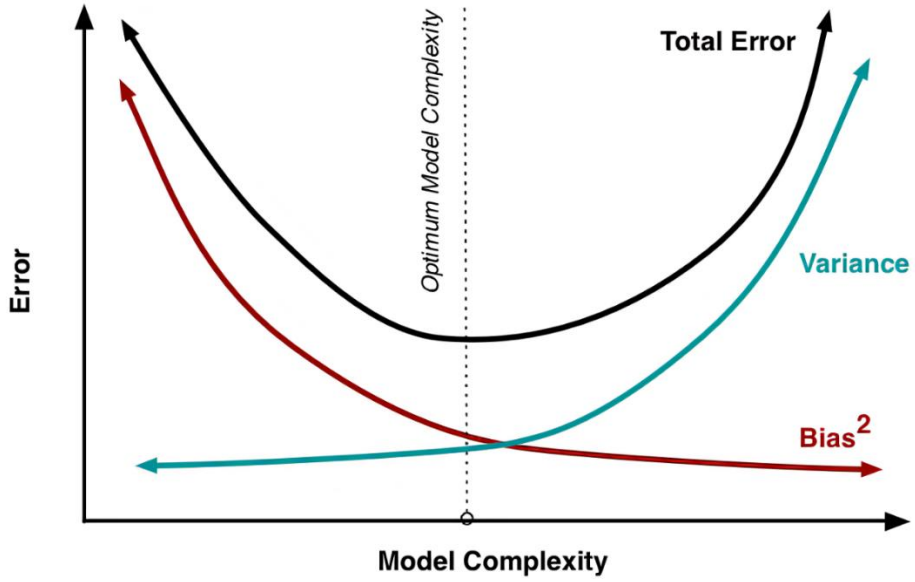
```
<keras.callbacks.History at 0x7a4b85859b70>
```

As we can see from the previous records, We have 40 epochs being executed one after the ot her, annotated from 1/40 till 40/40 epochs declaring completion of the training, we also have 70/70 batches bein g worked on during every epoch, where 70 batches is the result of 700 images / 10 images = 70 batches

4s 55ms/step is a metric that indicates how long it takes to run one training step on a batch of data. A training step is one gradient update, where the model parameters are adjusted based on the loss function and the optimizer. The size of the batch affects the speed and the accuracy of the training. The metric 4s 55ms/step means that it takes 4 seconds and 55 milliseconds to process one batch of data and update the model parameters. This metric depends on several factors, such as the model architecture, the batch size, the hardware, the data preprocessing, the optimization algorithm, etc. Generally, a lower value of this metric means a faster training process, but it does not necessarily mean a better model performance.

As for the loss metric, this loss is based on the pixel_MSE loss function which we've defined and passed to the loss parameter in the compile() function, this function computes the average loss value per epoch after a number of iterations and updates using the batches, the loss is performed on the predictions of the network after the epochs updates to the weights against the true labels of the input images and assessing how much error or by how much far are we from the true labels values. We can see that numerically, the average loss error does converge to a fairly good value, which definitely can be enhanced even more, but as general rule of thumb, if my model reaches a loss error value of 0.00XXXX….. or anything lower, it is considered to be in a good state from the training evaluation perspective.

And for the Val loss metric, same as the above loss metric, it is considered to have fairly good results that can also be improved, but one thing that I must note here is that, from all the training records we're seeing above, it is evident numerically and we can also plot it, that the loss error and validation error values are close to each other, which means that it is definite that there is still room for improvement of the model's performance, this is due to the Bias-Variance ratio concept:



As we can see from the above graph, optimum model complexity occurs when we reach a state numerically where the training error is low (relative to the validation error) and the validation error is also at it's lowest (relative to it's curve plot), and according to the records of the training data above, the loss and val_loss values seem to be still more on the left side of the graph, meaning that we may be somewhat close to a state of underfitting and that we need to train the model more and enhance it's performance so that the errors can diverge out more in terms of their values. If we plot the graphs of these errors we can definitely pinpoint what best number of iterations would suffice, given that all the other hyperparameters are well satisfied.

2.8) Evaluation Metrics:

Metrics to be used for evaluating the output images and the training performance of the model as described by the research paper are as follows:

1) Peak Signal-to-Noise Ratio (PSNR): It is a measure of how well an image can be reconstructed after being corrupted by noise or compression. It is defined as the ratio between the maximum possible power of an image and the power of the noise that affects its quality. The higher the PSNR, the better the quality of the reconstructed image.

2) Mean Squared Error (MSE): The mean squared error is the average of the squared differences between corresponding pixels in the two images, here the two images are the reconstructed and the original HR Image.

Both these metrics aid in giving an idea of whether we're proceeding in a positive or a negative direction while training, but i did do some research into whether there could be other metrics that focused more on the output's visual quality of the image as a main objective function for the model to go after, rather than using the pixel-wise difference comparison between the reconstructed images and the original HR Images. I found that there could be such loss functions that focus on the perceptual quality rather than on the pixel-level differences, but rather focuses on the high-level features of the output image. Such an example is called the perceptual loss or feature loss.

Perceptual loss is defined as the difference between the features extracted from a pretrained network (such as VGG19) for the predicted image and the original HR image. The idea is that the pretrained network can capture the high-level semantic information and the perceptual similarity of the images, rather than the low-level pixel values. Perceptual loss can produce more visually pleasing results with sharper details and fewer artifacts.

Coming to the code:

```
def PSNR(y_true,y_pred):
    mse=tf.reduce_mean( (y_true - y_pred) ** 2 )
    return 20 * log10(1/ (mse ** 0.5))
def log10(x):
    numerator = tf.math.log(x)
    denominator = tf.math.log(tf.constant(10, dtype=numerator.dtype))
    return numerator / denominator
def pixel_MSE(y_true,y_pred):
    return tf.reduce_mean( (y_true - y_pred) ** 2 )
```

Here I've defined three functions, two of which are related to each other, namely "PSNR" & "log10", and then also "pixel_MSE". Now let us dissect them one by one in the following bulletpoints:

- **pixel_MSE:** this function is simply the loss function which we've used while training our model, it simply works on finding the pixel-level difference between each pixel in the HR image and the predicted image and inform us by how much does the predicted pixel diverge from the target pixel, we do this for all the training samples in our training dataset and then we utilize a function in tensorflow called **reduce_mean**, what reduce_mean does is what we've discussed before in section 2.5, namely:

$$L(\Theta) = \frac{1}{n}\sum_{i=1}^{n} ||F(\mathbf{Y}_i;\Theta) - \mathbf{X}_i||^2,$$

- **log10:** this is the log base 10 function and it implements this mathematical function using tensorflow's tf.math class utilities, The function first computes the natural logarithm of x using the tf.math.log function, then divides it by the natural logarithm of 10, which is obtained by passing a constant tensor of value 10 and the same data type as x to the

tf.math.log function. The function returns the result of the division, which is the base-10 logarithm of x.

- **PSNR:** this function implements the Peak Signal-to-Noise Ratio equation and utilizes the log10 function:

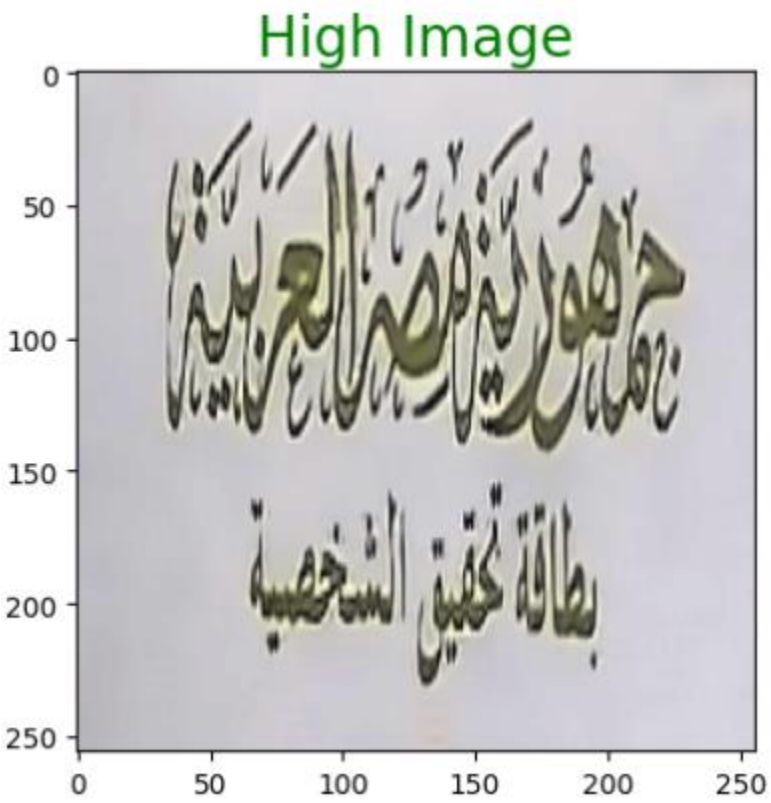
$$PSNR = 20 \log_{10} \left(\frac{MAX_f}{\sqrt{MSE}} \right)$$

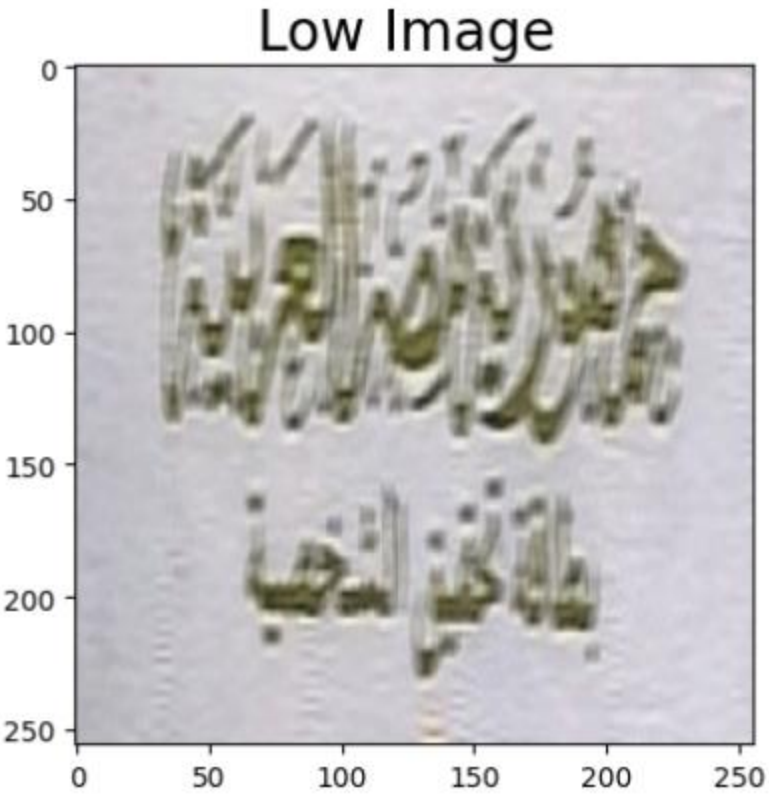
The function first computes the mean squared error (MSE) between y_true and y_pred using the tf.reduce_mean function and the power operator **. Then, it calls the log10 function that was defined in the previous code to calculate the base-10 logarithm of the inverse of the square root of the MSE. The function multiplies the result by 20 and returns it as the PSNR value.

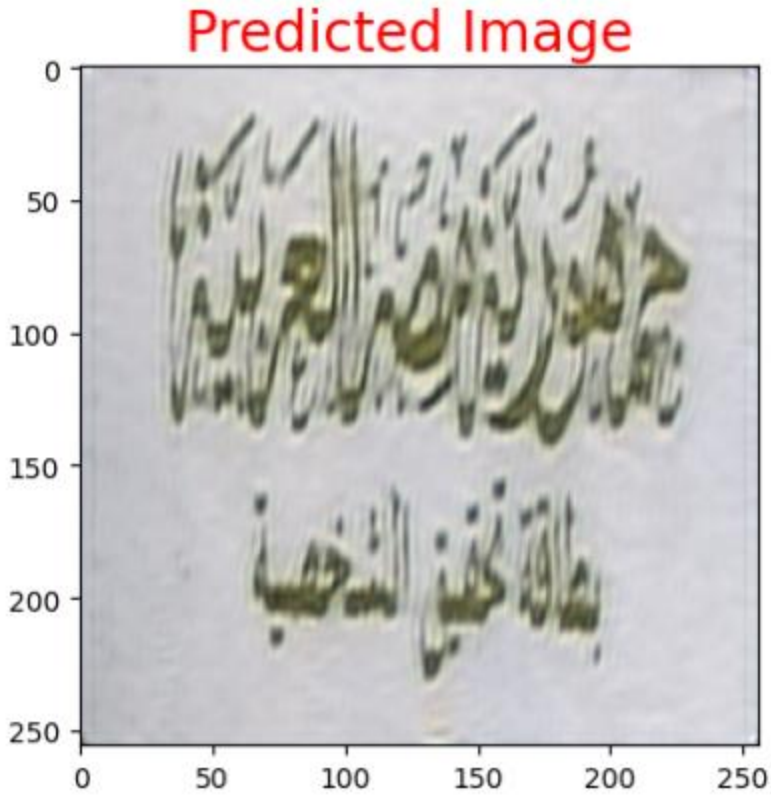
2.9) Model Evaluation:

In this stage, we will be assessing the performance of our model using the test set that we've defined above in section 2.3, here are some of the results:

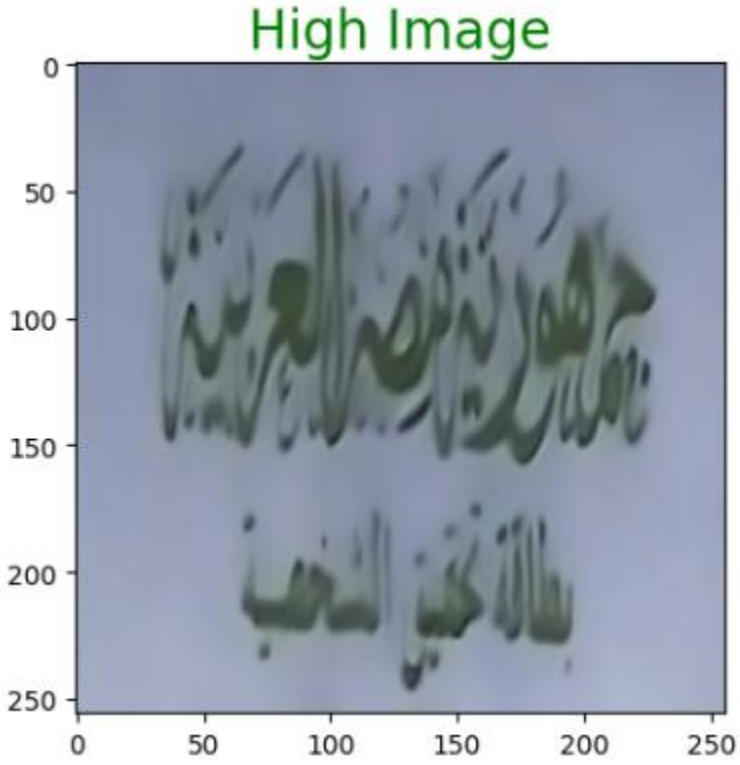
Sample 1:

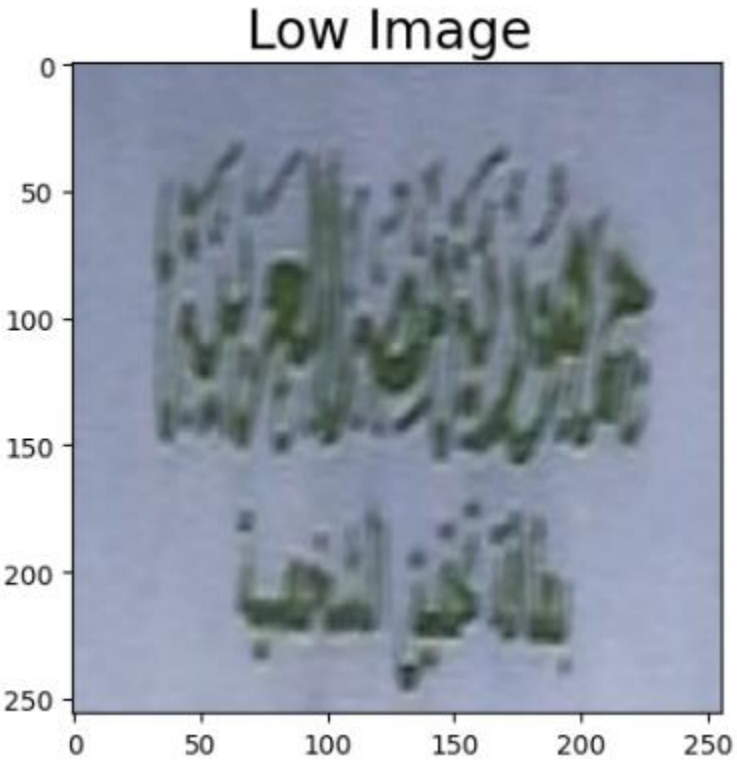


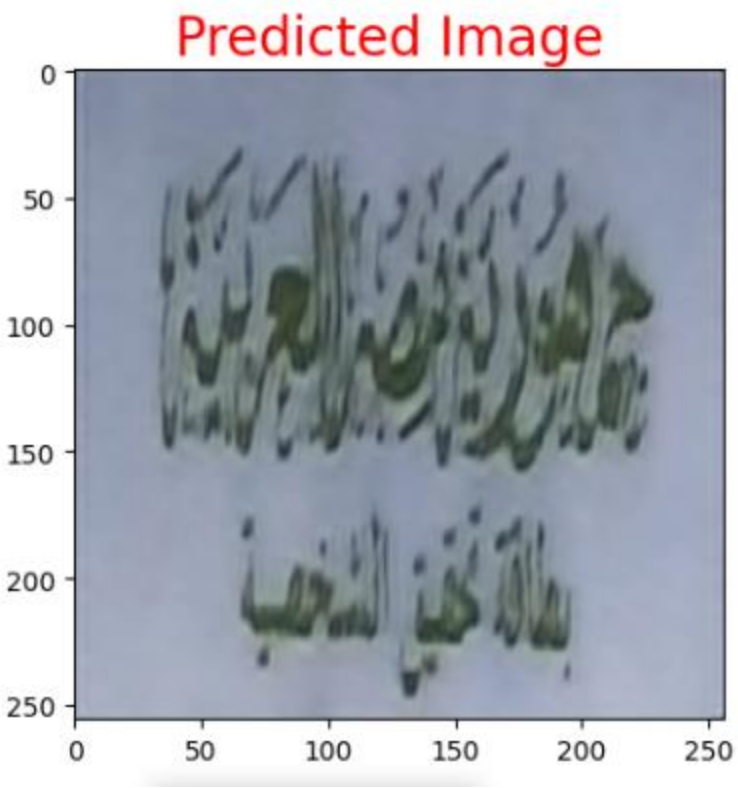

Low Image
0
50
100
150
200
250
0
50
100
150
200
250


Predicted Image
0
50
100
150
200
250
0
50
100
150
200
250

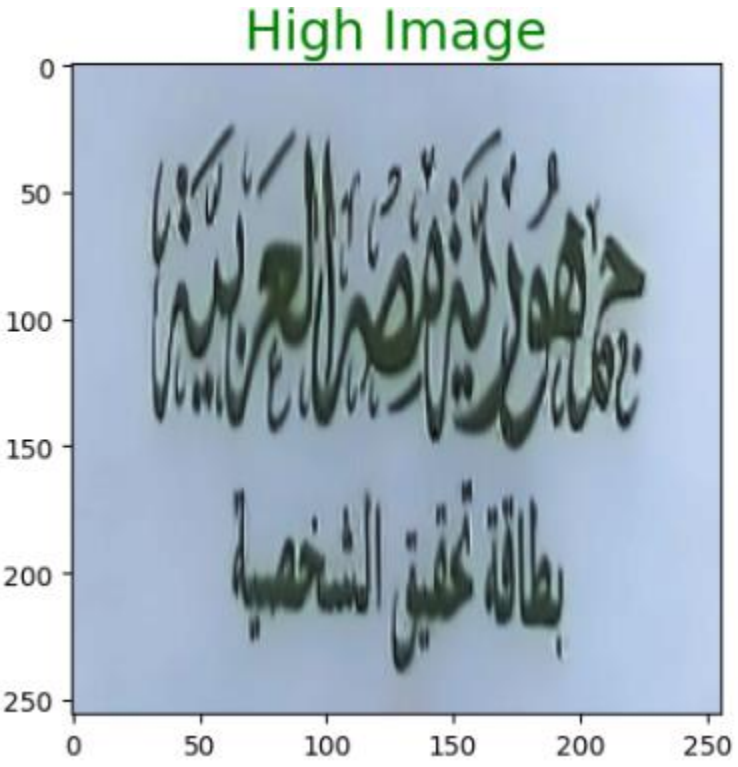
Sample 2:

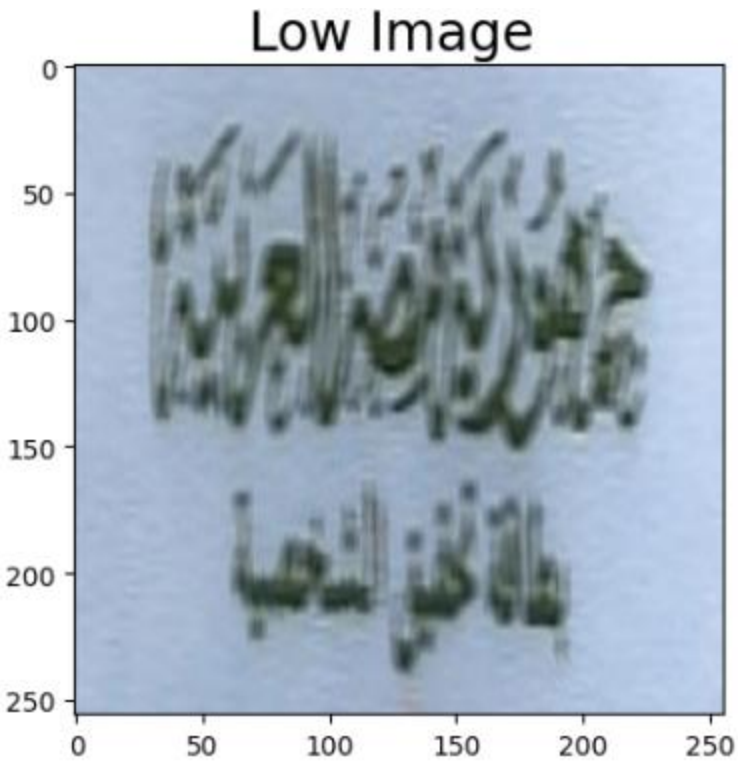

High Image
0
50
100
150
200
250
0
50
100
150
200
250

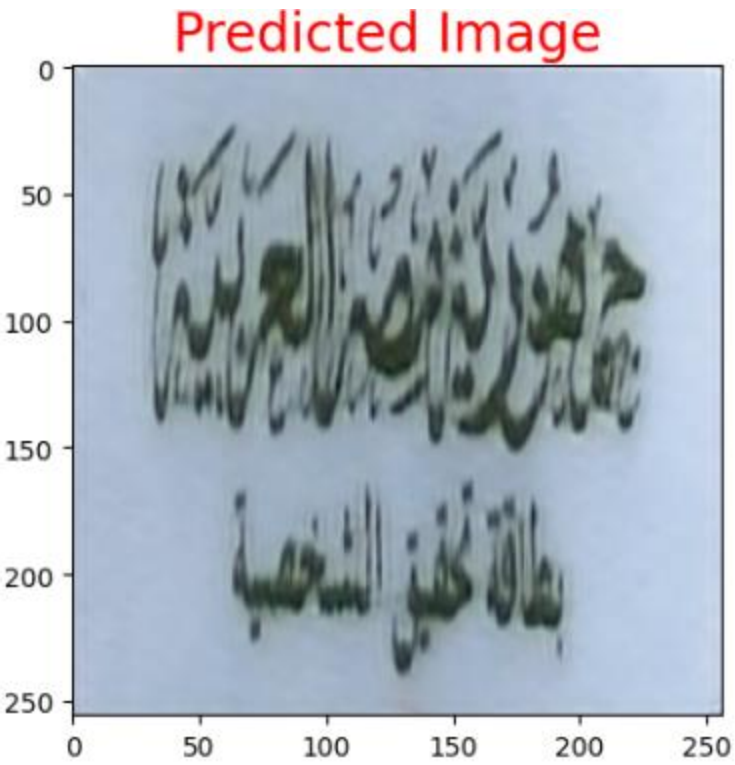

Low Image
0
50
100
150
200
250
0
50
100
150
200
250


Predicted Image
0
50
100
150
200
250
0
50
100
150
200
250

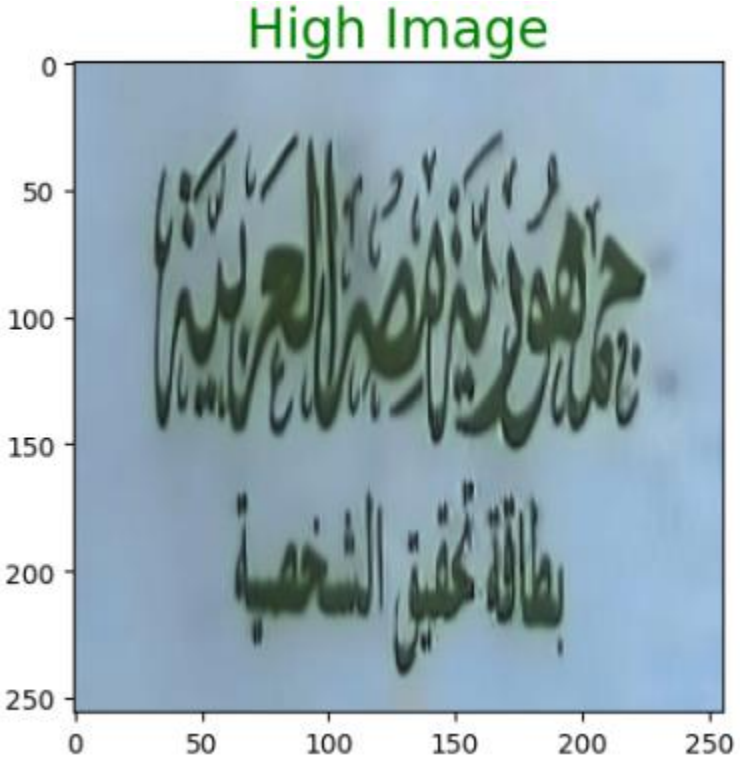
Sample 3:

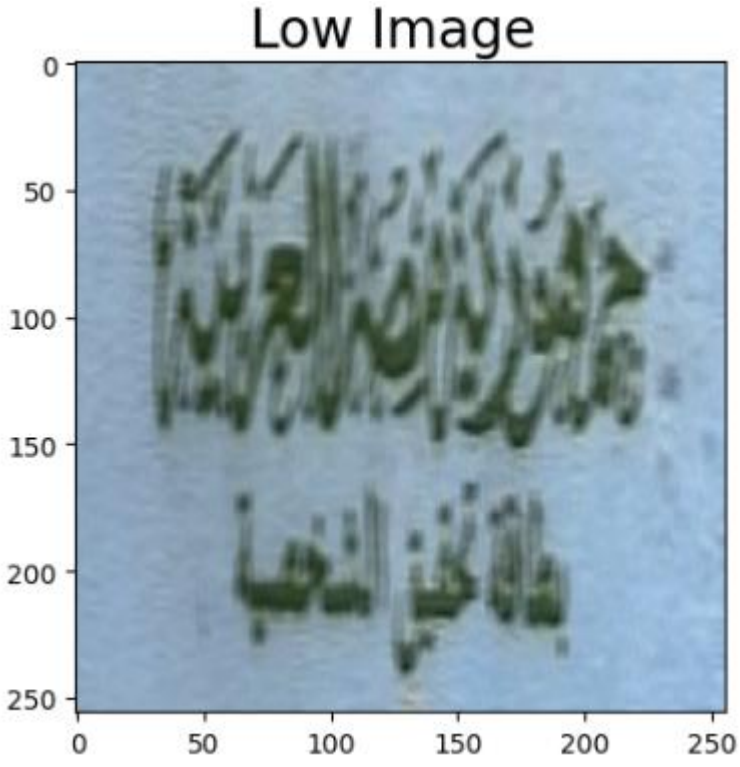

High Image
0
50
100
150
200
250
0
50
100
150
200
250

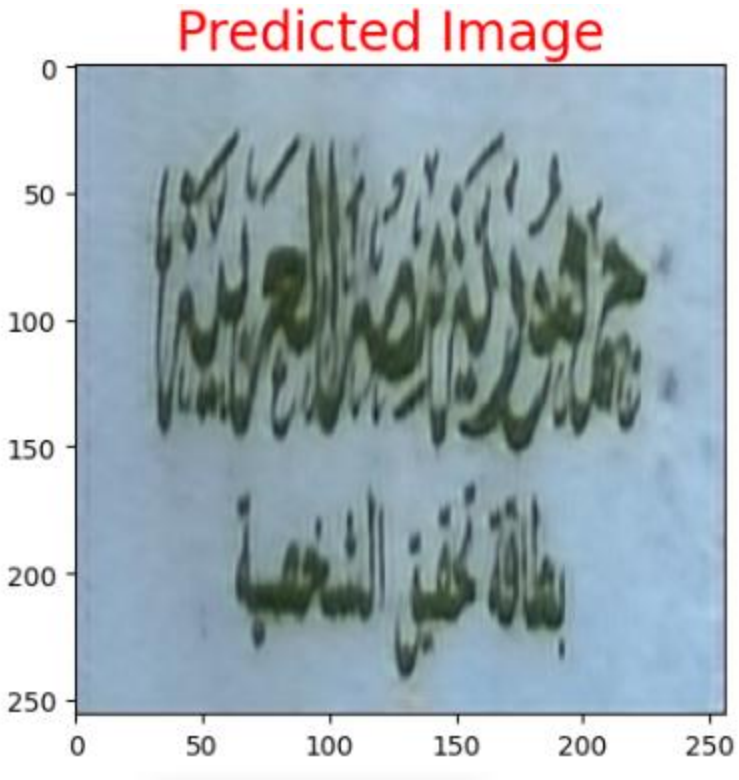

Low Image
0
50
100
150
200
250
0
50
100
150
200
250


Predicted Image
0
50
100
150
200
250
0
50
100
150
200
250

Sample 4:


High Image
0
50
100
150
200
250
0
50
100
150
200
250


Low Image
0
50
100
150
200
250
0
50
100
150
200
250



Mean average of PSNR & MSE Losses:

```
psnr mean average result is: 24.956628799438477
mse mean average error is: 0.0030490679200738 67
```

We can see from the research paper's findings that an average baseline that we can gauge our results relative to is about 31Db approximately for PSNR, therefore, we can conclude that our results are not the best, yet are not the worst, but surely there is room for improvement which we will be discussing in the next section.

2.10) Model and hyperparameters tuning:

In this section, we will discuss the improvements that can be made in order to enhance the prediction performance of our model:

1. **Utilizing the whole dataset:** Deep learning models can benefit from having more examples to learn from. Surely If we use the whole dataset, performance would even improve much better, the problem regarding the RAM usage can be solved by a tensorflow dataset object that would be handed to the model and yield the dataset in batches to it, this would lead to much less occupation of the memory space.

2. **Pre-Processing:** as mentioned before in section 2.4, two techniques that I feel like they can help in enhancing the performance of the training even more is **data cleaning** and **Analyzing noises in the frequency domain.**

3. **Choosing the right model:** Different deep learning models have different strengths and weaknesses, and some may be more suitable for this problem than others. According to the survey we've seen above in section 2.5, there's already some potential in other models with performance that is similar or a bit higher than the current model we're working with. We can try to compare different models, such as convolutional neural networks, ResNets, GANs and Attention Networks, and see which one performs better on our data.

4. **Tuning the hyperparameters:** Deep learning models have many hyperparameters that control the learning process, such as the learning rate, the batch size, the number of epochs, the activation function, the optimizer, and the regularization (in our case we don't need regularization as we're already trying to achieve a more complex model). We can try to tune these hyperparameters using methods such as grid search for example, and find the optimal combination that maximizes the model performance.

3) Conclusion and Final Remarks:

In conclusion, in this project, I have worked on enhancing images quality in order to be fed afterwards to an OCR model, this was accomplished using single image super resolution (SISR) deep learning models, of course there's much room for improvement and ideas are abundant, both from the mathematical side and from the programmatic side.