Modern University for Technology and Information
Faculty of Computers Science & AI
Department of Computer Science

Image Super Resolution Using Deep Convolution Neural Network

Computer Science Graduation Project

Submitted by:

Janna Allah Ayman El-Kayyali

Sief El-Deen Mostafa

Tarek Magdy Ahmed

Amr Shaaban Mahmoud

Mohamed Saad Labib

Adham Tarek Mohamed

Abdelrahman Mohamed Ahmed

Supervised by:

Prof. Eng. Dr. Hafez M.S Abdel-Wahab

T.A Dawood Essam

July 2022

Acknowledgement

We would like to express our gratitude to the **Modern University for Technology and Information** (**M.T.I**) for providing us with all necessary needs of our educational development, and a good environment to learn in throughout our educational years.

Prof. Dr. Olfat Kamel, President of the MTI University who established such a learning environment, providing the most effective facilities for our job to be done.

It is a pleasure to thank those who made this Possible, **Prof. Dr. Nabeel El-Deeb**, and Dean of faculty of Computers Science **Prof. Dr. Mohamed Taher El-Mayah**.

We also would like to express our deepest appreciation to the head of the Information System Department, and our favorite supervisor **Prof. Eng. Hafez M.S Abdel-Wahab** for his guidance, unlimited help and support.

Thanks to **Prof. Dr. Hanafy Ismail**, head of the Computer Science Department; **Prof. Dr. Eman Taha**, head of Basic Science Department; and **Prof. Dr. Hesham El-Deeb**, head of the Artificial Intelligence Department. For their efforts along with our university studies.

As well as the Department staff, **Prof. Dr. Alaa Abd El-Raheem, Prof. Dr. Mahmoud El-Shishtawy, Prof. Dr. Elsayed Bakkar, Prof. Dr. Rasha Saeed**, **Dr. Rania Ahmed** and **Dr. Marwa Rafaey** for teaching us important courses for our computer science degree.

We gratefully acknowledge the assistance, help, effort of all the teaching assistants who played an important role in our studies, with practical demonstrations that helped us to understand and to practice what we have learnt.

Finally, a special thanks to our families who always supported us during our academic life, and we could not do it without their help.

Abstract

Image super-resolution (ISR) is one of the vital image processing methods that improve the resolution of an image in the field of computer vision. In the last two decades, significant progress has been made in the field of super resolution, especially by utilizing deep learning methods. Image super-resolution plays an important role in several fields such as, computer graphics, medical imaging, security, space, and satellite.

The main objective of this project is to enhance and improve the resolution of an image, so it can be used beneficially in the fields mentioned before.

This documentation provides a literature review and techniques of image super-resolution in the perspective of initial classical methods and deep learning methods.

While the Deep learning methods has the best results compared to other traditional techniques. This project mainly discusses seven implemented models and view the results compared to each other.

Six models are based on Convolutional Neural Networks (CNN) which are Image Super-Resolution Using Deep Convolutional Networks (SRCNN), Fast Super-Resolution Convolutional Neural Network (FSRCNN), Residual Dense Network (RDN), Residual Feature Distillation Network for Lightweight Image Super-Resolution (RFDN), and Deep Auto-encoder for Single Image Super-Resolution. The seventh model is based on Generative adversarial network (GAN) which is Super-Resolution GAN (SRGAN). All models are implemented with python.

The result showed that RFDN is the best model objectively and subjectively, while FSRCNN and ESPCN are the fastest models in time prosessing.

Contents

List of Abbreviations:

CNN	Convolutional Neural Network
CPU	Central Processing Unit
GUI	Graphical User Interface.
GPU	Graphic Processing Unit
LR	Learning Rate
SGD	Stochastic Gradient Descent.
ADA-GRAD	Adaptive Gradient Descent.
GAN	Generative adversarial networks
VGG	Visual Geometry Group.
SR	Super Resolution
SISR	Single Image Super Resolution

List of Figures

List of Tables

Chapter 1
Introduction

Chapter 1: Introduction

Image super-resolution (SR), which refers to the process of recovering high resolution (HR) images from low resolution (LR) images, is an important class of image processing techniques in computer vision and image processing. It enjoys a wide range of real-world applications, such as medical imaging surveillance and security. Other than improving image perceptual quality, it also helps to improve other computer vision tasks. In general, this problem is very challenging and inherently ill-posed since there are always multiple HR images corresponding to a single LR image. To address this problem, numerous super-resolution (SR) methods are proposed, including early traditional methods and recent learning-based methods. Traditional methods include interpolation-based methods and regularization-based methods. Recently, a great number of convolutional neural network-based methods have been proposed to address the image SR problem.

1.1 Applications

As mentioned above, Super Resolution has a wide range of real-world applications, such as medical imaging, surveillance and security. In this section super resolution applications examples in several fields are discussed.

1.1.1 Medical diagnosis

Various medical imaging modalities can provide both anatomical information about the human body structure and functional information. However, resolution limitations always degrade the value of medical images in the diagnosis. SR technologies have been used with the key medical imaging modalities, including magnetic resonance imaging (MRI), functional MRI (fMRI), and positron emission tomography (PET). The goal is to increase the resolution of medical images while preserving the true isotropic 3-D imaging. Medical imaging systems can be operated under highly controlled environments, and thus continuous and multi-view images can be easily acquired. Fig. 1.1 shows the SR results on human brain MRI data and a respiratory synchronized PET image, respectively. [1]

Example-based SR for single frames has been also applied in the medical imaging field, by collecting similar images to establish a database. The following example presented in Fig. 1.2 is the reconstructed image of the single MRI image of the knee. The training database was established with a set of five standard images, including computed tomography (CT) and MRI images from various parts of the human body.

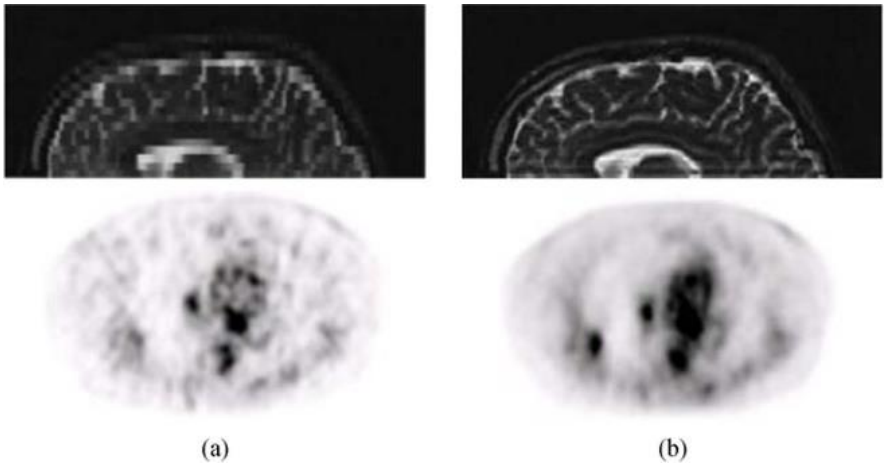


Figure 1.1 The SR results of (a) an MRI image and (b) a PET image. The first column is the original LR images, and the second column shows the corresponding SR.

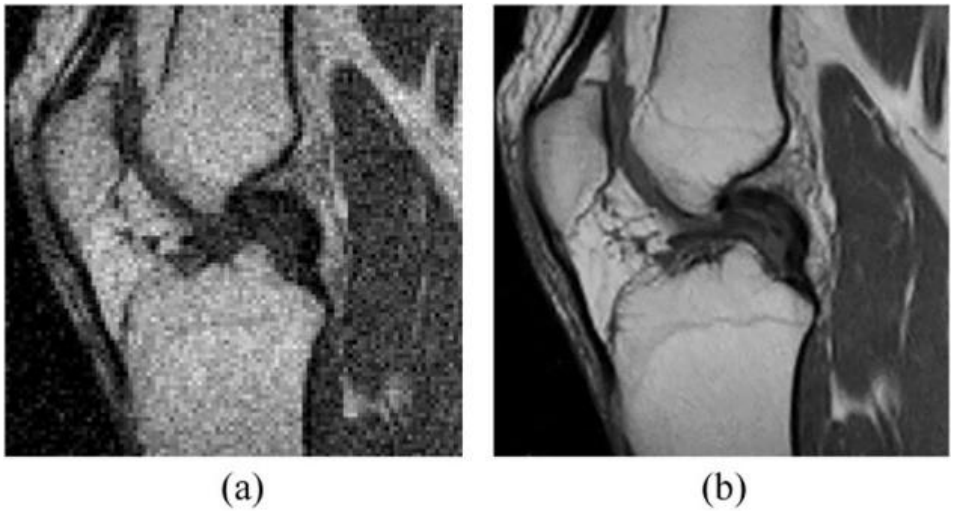


Figure 1.2 The single-frame SR result on the MRI knee image with a magnification factor of 4. (a) The original LR data. (b) The SR result.

1.1.2 Biometric information identification

SR is also important in biometric recognition, including resolution enhancement for faces, fingerprints, and iris images. The resolution of biometric images is pivotal in the recognition and detection process. To deal with the LR observations, a common approach is the development of high quality images from multiple LR images. Based on the redundancy and similarity in the structured features of biometric images, example based single-frame SR with an external database is an effective way of resolution enhancement. Three cases of biometric image reconstruction in Fig. 1.3 are presented. Using SR, the details of the shapes and structural texture are clearly enhanced, while the global structure is effectively preserved, which can improve the recognition ability in the relevant applications. [2]

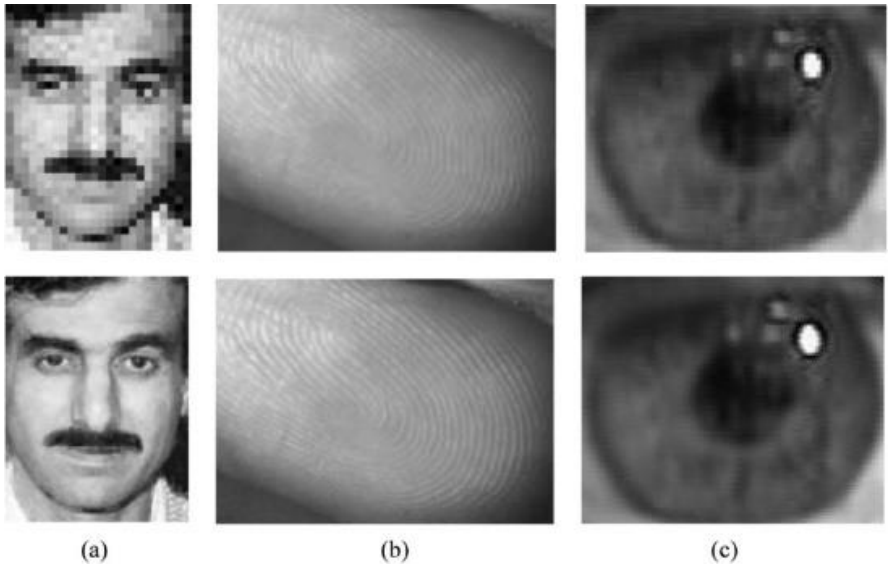


Figure 1.3 The SR results for face, fingerprint , and iris images , respectively. The first row is the LR image, while the second row shows the reconstructed result. (a) Face hallucination, (b) fingerprint reconstruction, and (c) iris reconstruction.

1.1.3 Earth-observation remote sensing

As known, the idea of applying SR techniques to remote sensing imaging has been developed for decades. Though data satisfying the demand for SR are not easy to obtain, there have been a few successful applicable examples for real data. Among them, the resolution of the panchromatic image acquired by SPOT-5 can reach 2.5m through the SR of two 5m images obtained by shifting a double CCD array by half a sampling interval, which was the most successful case. In addition, Shen et al. [3] proposed a MAP algorithm and tested it with moderate-resolution imaging spectroradiometer (MODIS) remote sensing images, as shown in fig. 1.4 Moreover, satellites can acquire multi-temporal or multi-view images for the same area, e.g. Landsat, CBERS, and WorldView-2, and thus provide the possibility for SR. An example is also given in fig. 1.4, which incorporates five angular images provided by the WorldView-2 satellite for SR. SR for the spectral unmixing of fraction images has been widely studied to acquire a finer-resolution map of class labels, and is known as sub-pixel mapping. Researchers have also attempted to apply the example-based methods to remotely sensed image SR. Recently, Skybox Imaging planned to launch a group of 24 small satellites, which can provide real-time “videos” with a submeter resolution using SR techniques. At the moment, SkySat-1 and SkySat-2 have been launched and put into use. By incorporating approximately 20 frames, the ground-based distance (GSD) of the output image can be decreased to 4/5 of the original data. This is a great opportunity to bring SR techniques into our daily life. The main challenges

for remotely-sensed image SR are to overcome the scene changes due to temporal differences, and to adapt the existing methods to massive amounts of observations every day.

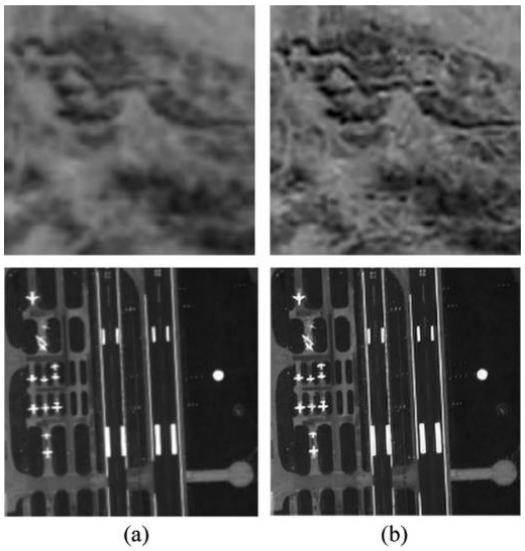


Figure 1.4 The SR reconstruction of remote sensing images: (a) and (b) indicate the LR and HR images, respectively. The first row shows the test on multi-temporal MODIS images with a magnification factor of 2. The second row is the SR example for multi-angle World

1.1.4 Astronomical observation

The physical resolution of astronomical imaging devices limited by system parameters also provides a chance for SR techniques to play a role. Astronomical systems can typically collect a series of images for SR. By improving the resolution of astronomical images, SR can help astronomers with the exploration of outer space. A specific example is shown in fig. 1.5 showing the SR of multiple star images. Satellites are also now being sent into outer space, e.g. the lunar exploration program and the Mars Odyssey mission. The SR can enhance the image resolution, and thus improve the discernibility of small objects on the moon's surface. Beyond this, Hughes and Ramsey used Thermal Emission Imaging System (THEMIS) thermal infrared and visible datasets from different spectral regions to generate an enhanced thermal infrared image of the surface of Mars.[3]

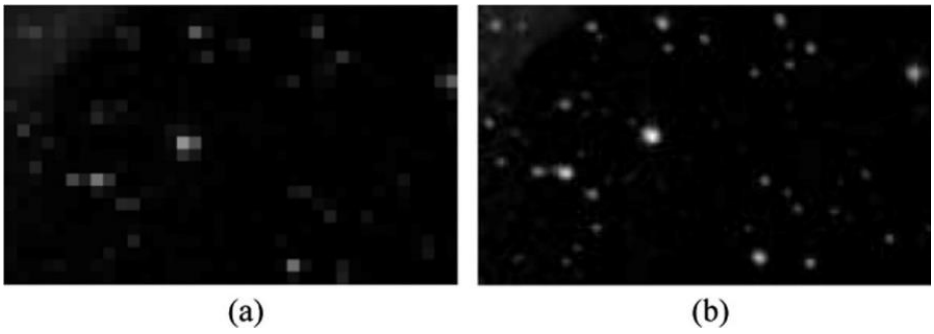


Figure 1.5 SR example of astronomical images: (a) the original LR image and (b) the SR result.

1.1.5 Surveillance

Nowadays, digital video recorder (DVR) devices are everywhere, and they play a significant role in applications such as traffic surveillance and security monitoring. It is, however, impossible for the moment to equip large-scale HR devices. Thus, it is necessary to study image SR techniques. fig. 1.6 gives two examples of SR for the Walk sequence and a UAV surveillance sequence. Although the techniques have developed progressively, the practical use of video SR is still a challenge. Firstly, outdoor video devices are vulnerable to the impact of weather conditions. Moreover, video data usually feature a huge amount of data and complex motion. Some algorithms can deal with the motion outliers, but the computational efficiency limits their application. Compressed video SR has also been a focus. Therefore, Image Super Resolution will help if a specific scene/ frame in the video isn't clear.[2]

Figure 1.6 The SR reconstruction of the Walk sequence (top row) and a UAV surveillance sequence (bottom row) : (a) indicates the reference LR frames, while (b) presents the corresponding reconstruction results.

1.2 Problem Definition

The image SR focuses on the recovery of an HR image from LR image input as and in principle, the LR image I_{xLR} can be represented as the output of the degradation function, as shown in the following equation. [4]

$$I_{xLR} = d\ (I_{yHR,}\ \partial)$$

Where d is the SR degradation function that is responsible for the conversion of HR image to LR image, I_{yHR} is the input HR image (reference image), whereas ∂ depicts the input parameters of the image degradation function. Degradation parameters are usually scaling factor, blur type, and noise. In practice, the degradation process and dependent parameters are unknown, and only LR images are used to get HR images by the SR method. The SR process is responsible for predicting the inverse of the degradation function d, such that $g = d^{-1}$

$$g(I_{xLR}, \delta) = d^{-1}(I_{xHR}) = I_{yE} \approx I_{yHR}$$

Where g is the SR function, δ depicts the input parameters to the function g, and I_{yE} is the estimated HR corresponding to the input I_{xLR} image. It is also worth noticing that the super resolution function, as in the previous equation, is ill-posed, as the function g is a non-injective function; thus, there are infinite possibilities of I_{yE} for which the condition $d(I_{yE}, \partial) = I_{xLR}$ will hold.

To conclude the problem definition, having a low resolution image with low details and quality is the main problem.

1.3 Problem Solution

To solve this problem. The timeline of approaches for super resolution includes

- Early algorithms based on resampling (interpolation).
- Contemporary algorithms based on deep learning models.

1.4 Solution Approach

- Investigation related to convolution approaches
  - Literature review on convolution approaches for super resolution are discussed in chapter 2.
- Recent advances in deep learning
  - Deep learning models for super resolution are discussed in chapter 3, and implemented in chapter 4.

1.5 Project Management

1.5.1 Team Members:

1. Janna Allah Ayman El-Kayyali
2. Sief El-Deen Mostafa
3. Tarek Magdy Ahmed
4. Amr Shaaban Mahmoud
5. Mohamed Saad Labib
6. Adham Tarek Mohamed
7. Abdelrahman Mohamed Ahmed

1.5.2 Task Definition:

The Team tasks can be Summarized as the following

1. **Literature review:** All Team.

2. **Deep Learning Models:**
    - **SRCNN:** Abdelrahman Mohamed Ahmed
    - **FSRCNN:** Abdelrahman Mohamed Ahmed
    - **ESPCN:** Tarek Magdy Ahmed
    - **RDN:** Mohamed Saad Labeeb
    - **RFDN:** Janna Allah Ayman El-Kayyali
    - **Autoencoder:** Amr Shaaban Mahmoud
    - **ESRGAN:** Adham Tarek Mohamed
3. **Similar Applications:** Sief El-Deen Mostafa
4. **Implementation:** All Team.
5. **Datasets:** Sief El-Deen Mostafa
6. **GUI:** Amr Shaaban Mahmoud
7. **Testing & Conclusion:** Sief El-Deen Mostafa
8. **Documentation & Presentation:** All Team.

1.6 Documentation Organization

Chapter One: Introduces Super Resolution, applications of super resolution, problem definition and solution, and project management.

Chapter Two: Introduces a literature review including Super Resolution techniques, and an introduction about deep learning.

Chapter Three: Introduces the related work & similar applications will be pointed out.

Chapter Four: Introduces the implementation of 7 models and the training process. The platforms used and the algorithms for each implementation. Then the Graphical User Interface (GUI) will be presented.

Chapter Five: Testing and evaluation of the system, where some experiments are carried out and the results will be stated.

Chapter Six: The conclusion established based on the results of the experiments done, and the future work.

The following table 1.1, describes the sequence of tasks taken and implemented in this project and how it is handled between the team members.

Table 1.1 Time Schedule.

#	Task Name	Duration	Start	Finish	Assigned to
1	Literature review on SR systems	4 wks	Sun 26-09-21	Sun 24-10-21	All students
2	Hands on tools including setup of the platform	3 wks	Sun 24-10-21	Sun 14-11-21	all students
3	Understanding the images resolution and super resolution	2 wks	Sun 14-11-21	Sun 28-11-21	all students
4	Detailed introduction with good survey on DL methods	3 wks	Sun 28-11-21	Sun 19-12-21	all students
5	DL approaches to solve SR problem	3 wks	Sun 19-12-21	Sun 09-01-22	all students
6	Different architecture and up-sampling operations	3 wks	Sun 09-1-22	Sun 30-01-22	all students
7	Studying of similar application, their features, architecture	3 wks	Sun 30-01-22	Sun 20-02-22	all students
8	Full description and analysis of the application as a whole and each part of it	3 wks	Sun 20-02-22	Sun 13-03-22	all students
9	Collecting the used dataset and using required preprocessing need	3 wks	Sun 13-03-22	Sun 03-04-22	Sief El-Deen
10	Implementation of selected algorithms of the	6 wks	Sun 13-03-22	Sun 24-04-22	Janna, Tarek, Amr, Mohamed,

	required systems on different scales.				Adham, Abdelrahman
11	Testing the DL model, evaluating system performance	4 wks	Sun 03-04-22	Sun 01-05-22	All students
12	GUI Implementation	4 wks	Sun 14-04-22	Sun 08-05-22	Amr
13	Project documentation and presentation	9 wks	Sun 01-05-22	Sun 03-07-22	all students

Chapter 2

Literature Review

Chapter 2: Literature Review

This chapter, discuss the following topics fig.2.1. It includes an overview about super resolution techniques, traditional approaches and advanced approaches. Also it discusses the mechanism of deep learning and how to use deep learning in super resolution.

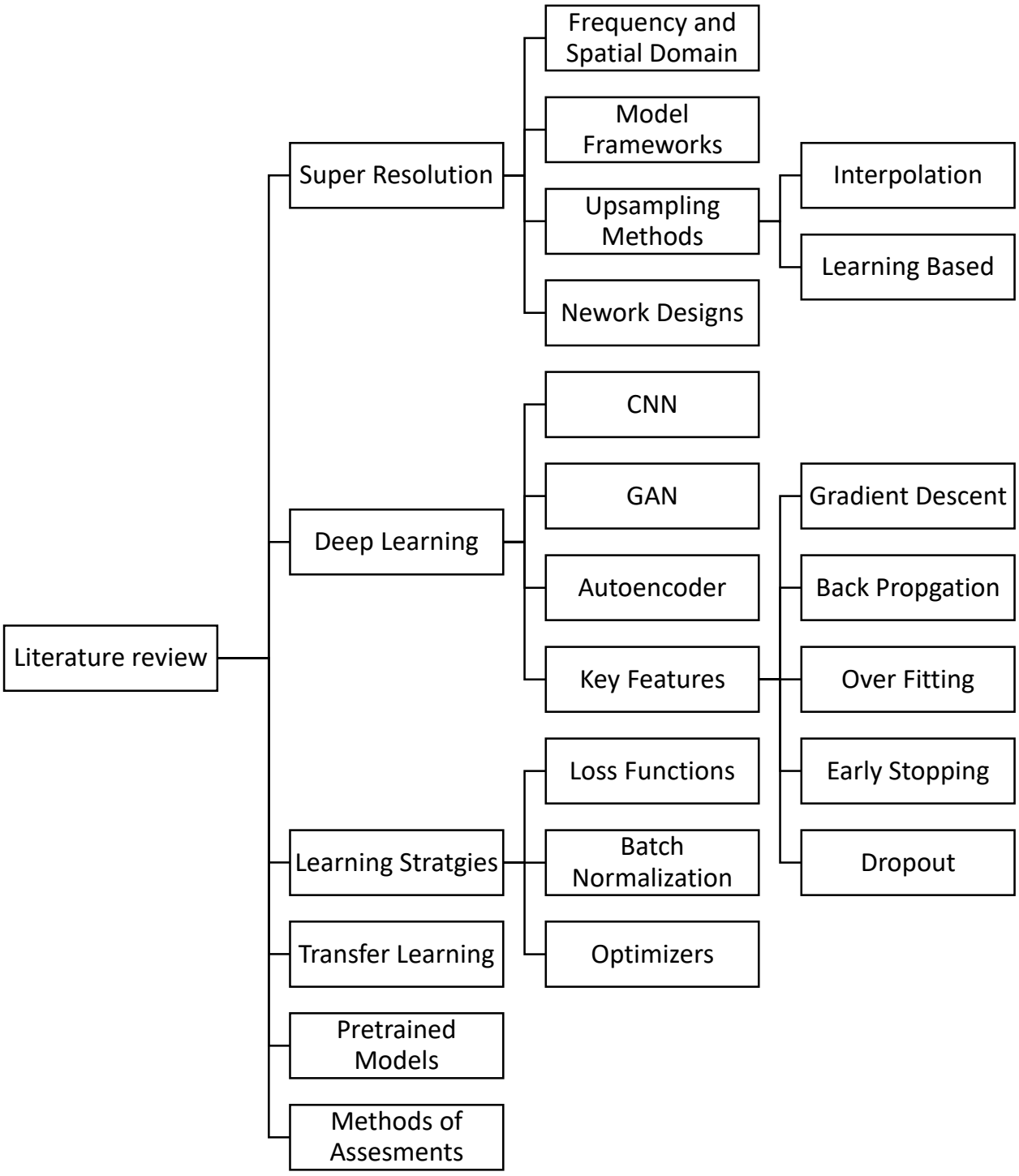


Figure 2.1 CH.2 Map

2.1 Frequency and Spatial Domain for Super Resolution

2.1.1 Frequency Domain

Fourier Series

As the Fourier series is an extension of the periodic function that uses an infinite number of sines and cosines. The Fourier series was originally developed to solve thermal equations, but it later became clear that the same technique could be used to solve a wide variety of mathematical problems, especially those involving linear differential equations with constant coefficients. Fourier series are used today in various fields including electrical engineering, vibration analysis, acoustics, optics, signal processing, image processing, quantum mechanics, and econometrics. The Fourier series in fig 2.2 uses the orthogonality of the sine and cosine functions. Calculating and studying the Fourier series is called harmonic analysis and is very useful when working with any periodic function because it allows the function to be broken down into simple terms that can be used to solve the original problem obtained.

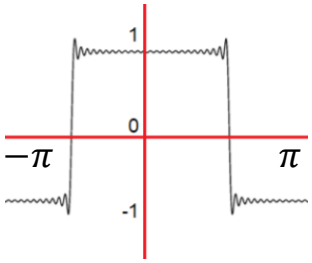


Figure 2.2 Fourier Series

Fourier transform

A Fourier transform is a mathematical transform that decomposes functions depending on space or time into functions depending on spatial frequency or temporal frequency

2D Continuous

$$F(u.v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x.y)e^{-j2\pi(ux-vy)}dxdy \qquad \delta\{f(x.y)\} = F(u.v)$$
$$\delta^{-1}\{F(u.v)\} = f(x.y)$$

$$f(x.y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u.v)e^{j2\pi(ux+vy)}dudv$$

2D Discrete

$$F(u.v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x.y)e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \qquad \text{for } u = 0.1.\ldots.M-1. and\ v = 0.1.\ldots.N-1$$

$$f(x.y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u.v)e^{j2\pi\left(\frac{ux}{M}+\frac{v}{N}\right)} \qquad \text{for } x = 0.1.\ldots.M-1. and\ y = 0.1.\ldots.N-1$$

$Notes \rightarrow u = (u.v) is\ spatial\ frecuency\ and\ direction$

$\rightarrow F(0.0)\ is\ the\ dc\ component\ (ie\ sum\ of\ values)$

Difference between Fourier series and Fourier transform

Fourier series is an extension of the periodic signal as a linear combination of sine and cosine, while the Fourier transform is a process or function used to convert signals in the time domain to the frequency domain. Fourier series is defined for periodic signals and the Fourier transform can be applied to aperiodic signals (without periodicity). As noted above, studying the Fourier series provides the motivation for Fourier transforms.

Filtering in frequency domain

We take the image and get the FFT(fast Fourier transform) of it and multiply it with filter in frequency as in fig. 2.3.

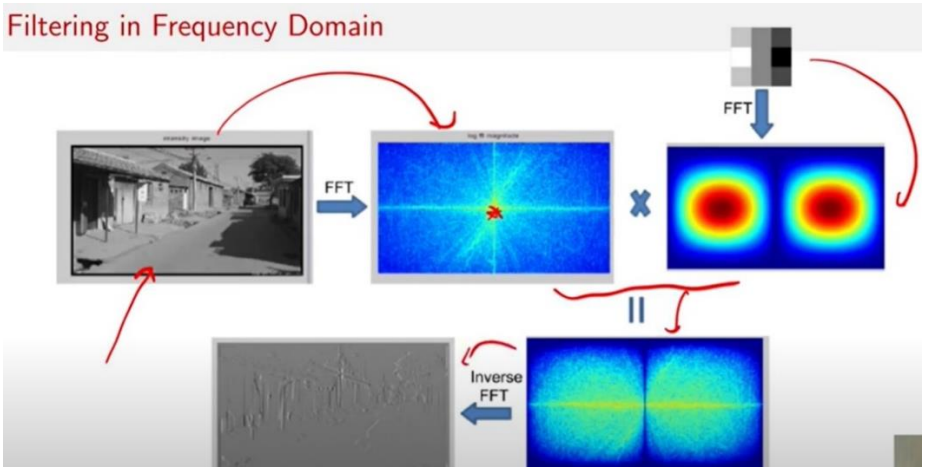


Figure 2.3 Filtering in Frequency Domain

Image Enhancement in Frequency Domain Blurring / Noise Reduction:

- Low-Pass Filter: allows the frequency below the cut off frequency to pass through it (Smoothing the image) in fig. 2.4.
- High-Pass Filter: allows the frequencies above the cut-off frequency to pass through it. (Sharpen the image) in fig. 2.4.
- Band-Pass Filter: allows frequencies within the chosen range through and attenuates frequencies outside of the given range in fig. 2.4.

Figure 2.4 Image Enhancement in Frequency Domain

2.1.2 Spatial Domain

Spatial domain methods refer to the image plane itself and involve the direct manipulation of the pixels in an image.

Filtering in Spatial Domain in fig 2.5

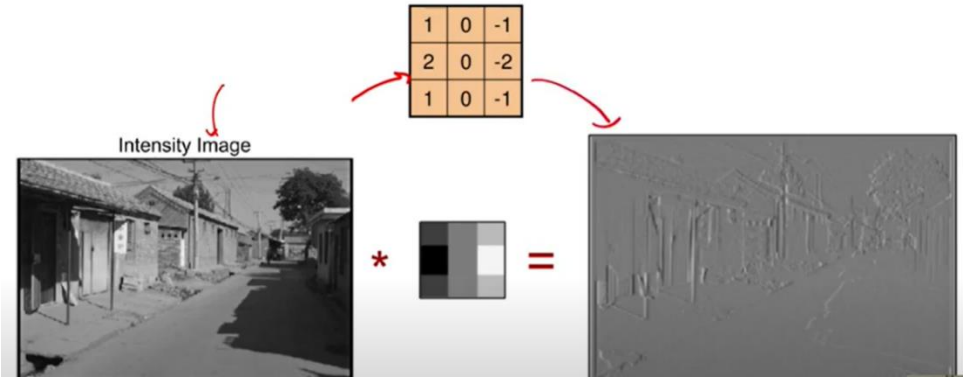


Figure 2.5 Filtering in Spatial Domain

2.2 Model Frameworks

Super resolution main idea is to increase the dimensions of an image, and then fill the empty spaces with meaningful pixel intensities. Similarly, at some point in a CNN network architecture that will be discussed later, few layers are assigned for up-sampling the input image to match the dimensions of the expected output image. This can be achieved in different ways: pre-upsampling, Post-upsampling, Progressive Upsampling, and Iterative Up-and-down Sampling Super Resolution.

In this section we will only discuss the first two techniques (pre and post upsampling super resolution), since they are the most common techniques and used in our models that will be further discussed in chapter 3, and 4. For further information about other techniques check reference. [5]

2.2.1 Pre-Up-Sampling

At the expense of the difficulty of directly learning the mapping from low-dimensional space to high-dimensional space, utilizing traditional upsampling algorithms to obtain higher resolution images and then refining them using deep neural networks is a straightforward solution. Firstly, adopt the pre-upsampling SR framework (as Fig.2.6 shows) and propose the deep learning model to learn an end-to-end mapping from interpolated LR images to HR images. Specifically, the LR images are upsampled to coarse HR images with the desired size using traditional methods (e.g., bicubic interpolation), then deep CNNs are applied on these images for reconstructing high-quality details. Since the most difficult upsampling operation

has been completed, CNNs only need to refine the coarse images, which significantly reduces the learning difficulty. However, the predefined upsampling often introduce side effects (e.g., noise amplification and blurring), and since most operations are performed in high-dimensional space, the cost of time and space is much higher than other frameworks.

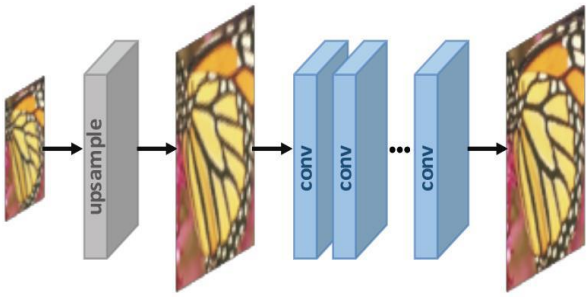


Figure 2.6 Pre-Up-Sampling

2.2.2 Post-Up-Sampling

In order to improve the computational efficiency and make full use of deep learning technology to increase resolution automatically, researchers propose to perform most computation in low-dimensional space by replacing the predefined upsampling with end-to-end learnable layers integrated at the end of the models as in fig. 2.7. In the pioneer works of this framework, namely post-upsampling SR, the LR input images are fed into deep CNNs without increasing resolution, and end-to-end learnable upsampling layers (in the form of deconvolution or sub-pixel convolution is used, instead of using simple bicubic interpolation) are applied at the end of the network. Since the feature extraction process with huge computational cost only occurs in low-dimensional space and the resolution increases only at the end, the computation and spatial complexity are much reduced. Therefore, this framework also has become one of the most mainstream frameworks. These models differ mainly in the learnable upsampling layers, anterior CNN structures and learning strategies.

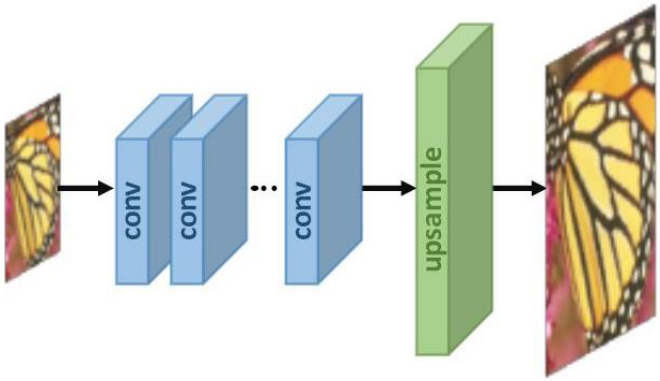


Figure 2.7 Post-Up-Sampling

2.3 Up-sampling Methods

This section discusses upsampling methods as in fig. 2.8. the 5 highlighted methods are discussed in this sections as they are the most common methods and used in our models. For further information about other methods check the reference. [6]

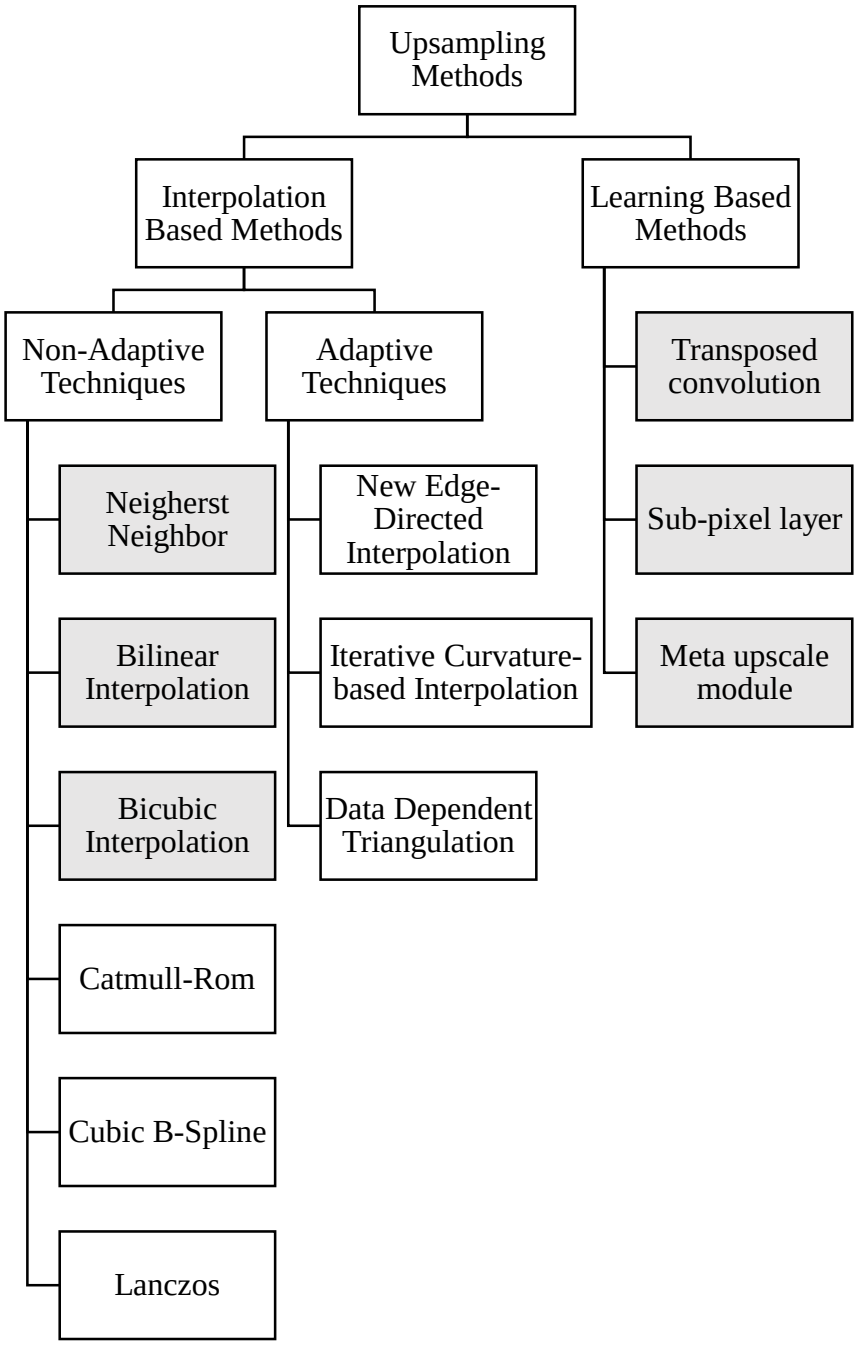


Figure 2.8 Upsampling Methods

2.3.1 Non Adaptive Techniques

Non-adaptive interpolation techniques are based on direct manipulation on pixels instead of considering any feature or content of an image. These techniques follow the same pattern for all pixels and are easy to perform and have less calculation cost. There are various types of non-adaptive techniques like nearest neighbor, bilinear and bicubic...etc.

1. Nearest Neighbor

The Nearest neighbor interpolation is simplest to implement as it only considers one pixel, the closest one to interpolate the point and does not consider the values of neighboring points at all. It requires the least computation and takes least processing time. Using the Nearest Neighbor algorithm, the empty spaces will be filled in with the closest neighboring pixel value as in fig. 2.9. It simply makes each pixel larger by replicating the new pixel. The pixels or dots of color are replicated to create new pixels as the image grows. This interpolation method is very efficient and does not create an anti-aliasing effect. The quality of an image generated using nearest neighbor is very poor as it creates pixilation or edges that break up curves into steps or jagged edges. This is not the suitable interpolation method for enlarging images because it results in blocky images, so it is not used in high quality imaging application which has fast execution time but has difficulty in producing high-quality results.

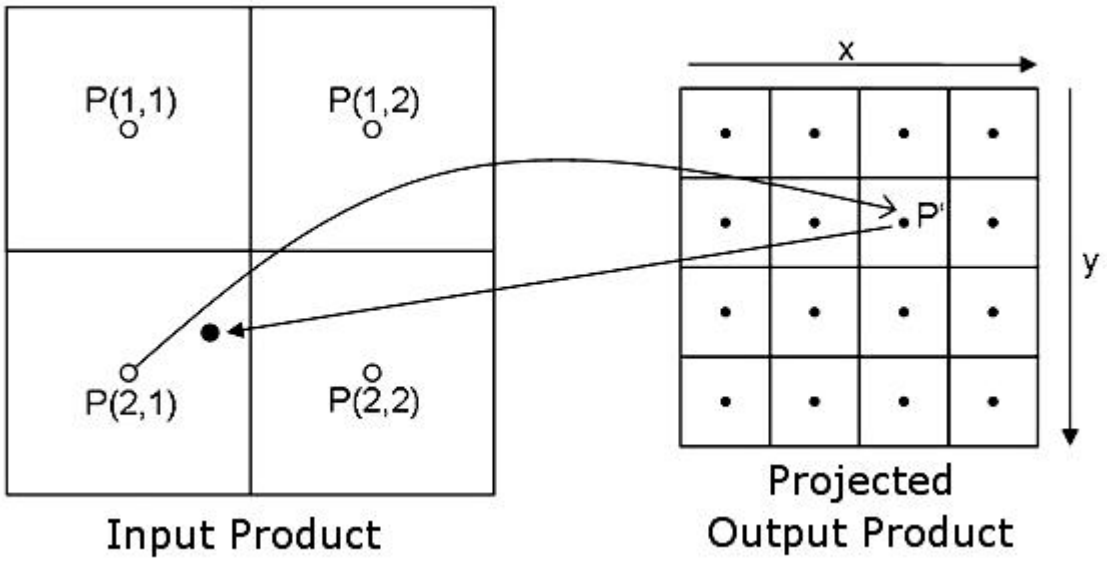


Figure 2.9 Nearest Neighbor

2. Bilinear Interpolation

Bilinear interpolation identifies the four nearby pixels in an image and takes the distance-weighted average of these four pixels to determine new value. Since new a pixel is estimated according to the relative position of neighboring four pixels as in fig. 2.10, results in smoother images than the Nearest Neighbor interpolation. The new image generated using the bilinear interpolation method will have smooth edges compared to the original image. If all the four pixels are equal distance from the computed pixel then the intensity of the new pixel will be simply average of four neighbor pixels. This method will generate the blurring effect in an image. Since it takes more number of pixels for computation than Nearest Neighbor interpolation, it requires more processing time and produces better quality realistic image output.

$$P'(x.y) = P(1.1) \cdot (1-d) \cdot (1-d') + P(1.2) \cdot d \cdot (1-d') + P(2.1) \cdot d' \cdot (1-d') + P(2.2) \cdot d \cdot d'$$

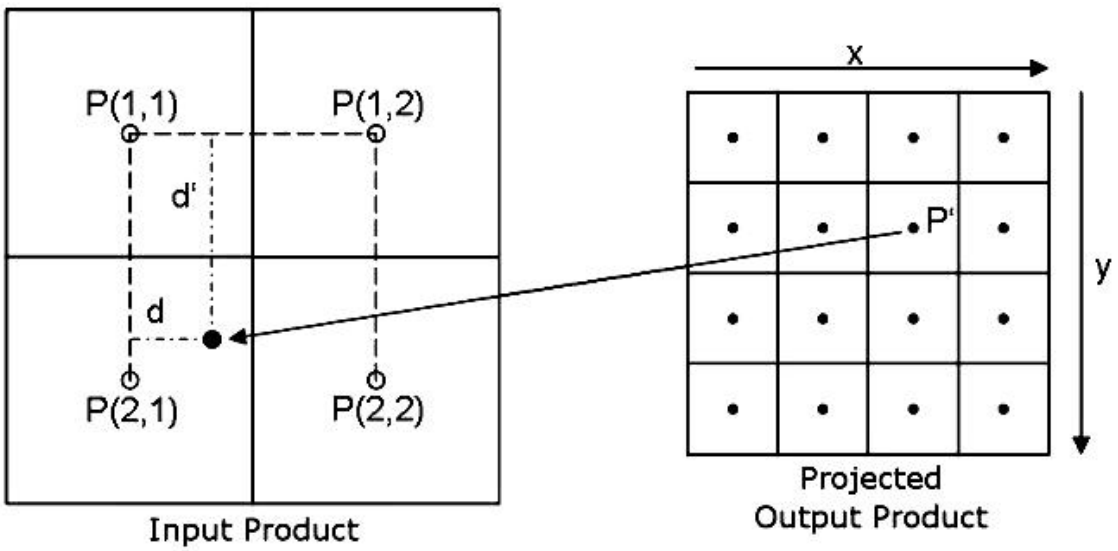
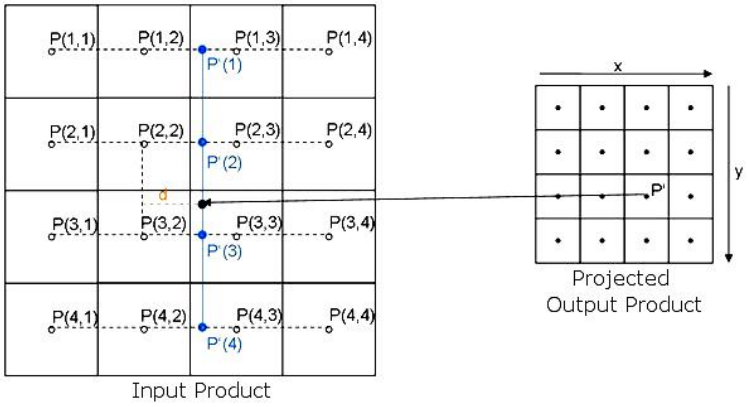


Figure 2.10 Bilinear Interpolation

3. Bicubic Interpolation

Bicubic interpolation is an advanced image interpolation algorithm as it considers four by four adjacent pixels for a sum of 16 pixels as in fig. 2.11. These sixteen pixels are at a different position from the computed pixel, more weight is given to nearby pixels according to its distance. The color intensity of the new pixel will be calculated by these 16 pixels according to their weighted average [7]. Bicubic interpolation generates sharper images than the nearest neighbor and bilinear interpolation methods. This method gives jaggies effect around sharp

boundaries lines, which are more visible with the contrast color interpolated image. Computational time for this method is more as it takes more number of pixels for calculation. Bicubic interpolation algorithm produces an eye pleasing image and it is a standard in the majority of image editing software



$$P'(k) = P(k,1)*\left(4-8(1+d)+5(1+d)^2-(1+d)^3\right)+$$
$$P(k,2)*\left(1-2d^2+d^3\right)+$$
$$P(k,3)*\left(1-2(1-d)^2+(1-d)^3\right)+$$
$$P(k,4)*\left(4-8(2-d)+5(2-d)^2-(2-d)^3\right)$$

Figure 2.11 Bicubic Interpolation

Comparison between the previous techniques

Table 2.1 Difference Between Nearest Neighbor, Bilinear, and Bicubic Interpolation

	Pros	**Cons**
Nearest Neighbour	• Very simple, fast • No new values are calculated by interpolation • Fast, compared to Cubic Convolution resampling	• Some pixels get lost, and others are duplicated • Loss of sharpness
bilinear interpolation	• Extremas are balanced • Image losses sharpness compared to Nearest Neighbour	• Less contrast compared to Nearest Neighbour • New values are calculated which are not present in the input product
bicubic interpolation	• Extremas are balanced • Image is sharper compared to Bi-linear Interpolation	• Less contrast compared to Nearest Neighbour • New values are calculated which are not present in the input product • Slow, compared to Nearest Neighbour resampling

2.3.2 Learning Based Methods

1. Transposed Convolutional:

The transposed Convolutional Layer is also known as the Deconvolutional layer. A deconvolutional layer reverses the operation of a standard convolutional layer. if the output generated through a standard convolutional layer is deconvolved, you get back the original input. The transposed convolutional layer is like the deconvolutional layer in the sense that the spatial dimension generated by both are the same. Transposed convolution doesn't reverse the standard convolution by values, rather by dimensions only as in fig. 2.12.

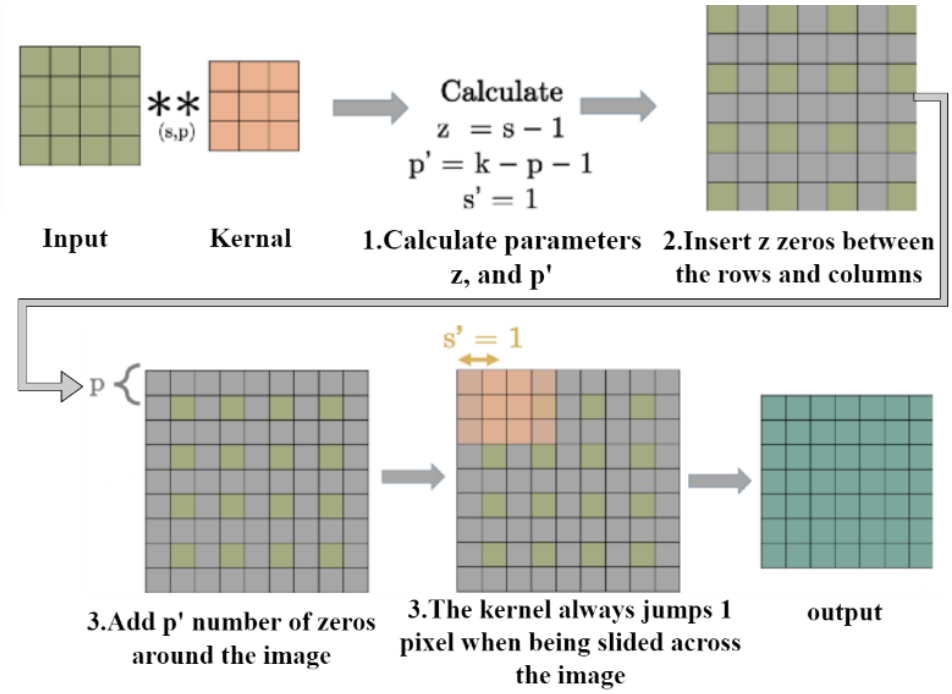


Figure 2.12 Transposed Convolution

A transposed convolutional layer is usually carried out for up-sampling. to generate an output feature map that has a spatial dimension greater than that of the input feature map. Just like the standard convolutional layer, the transposed convolutional layer is also defined by the padding and stride. These values of padding and stride are the one that hypothetically was carried out on the output to generate the input. if you take the output and carry out a standard convolution with stride and padding defined, it will generate the spatial dimension same as that of the input.

Implementing a transposed convolutional layer can be better explained as a 4-step process.

Step 1: Calculate new parameters z and p'.

Step 2: Between each row and columns of the input, insert z number of zeros. This increases the size of the input to (2*i-1) x(2*i-1).

Step 3: Pad the modified input image with p' number of zeros.

Step 4: Carry out standard convolution on the image generated from step 3 with astride length of 1

For a given size of the input (I), kernel (k), padding (p), and stride (s), the size of the output feature map (o) generated is given by:

$$O = (i-1) \times s + k - 2p$$

The table 2.2 below summarizes the two convolutions, standard and transposed:

Table 2.2 Standard and Transposed Convolution Summary

Comparison					
Conv Type	**Operation**	**Zero Insertions**	**Padding**	**Stride**	**Output Size**
Standard	Downsampling	0	P	s	(i+2p-k)/s +1
Transposed	Upsampling	(s-1)	(k-p-1)	1	(i-1)*s+k-2p

2. Sub-pixel Convolution:

The sub-pixel layers another end to-end learnable up sampling layer, performs up sampling by generating a plurality of channels by convolution and then reshaping them. Before understanding sub-pixel convolution, it is necessary to familiar with the concept of sub-pixel. In the camera imaging system, the image data obtained by the camera have been processed by a kind of discretized processing method. Due to the limitation of the light sensor, images are limited to the original pixel resolution, in other words, each pixel on the images represented a small area of color in the real world. In the digital image we saw, pixels and pixels are connected, while in the microscopic world there are numbers of tiny pixels between the two physical pixels. Those tiny pixels are called sub-pixels as in fig. 2.13.

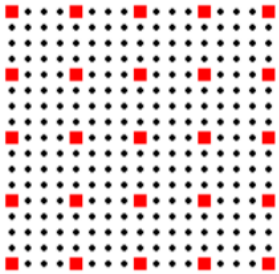


Figure 2.13 Subpixel

Each square area surrounded by four little red squares is the pixel in the imaging plane of the camera, the black dots are sub-pixels. The accuracy of sub-pixels can be adjusted depending on the interpolation between the adjacent pixels. In this way, the mapping from small square areas to big square areas can be implemented through sub-pixel interpolation.

Based on this theory, the sub-pixel convolution method can be used in the SR model to obtain high-resolution images. In general deconvolution operation, we pad the images with zeros and then do the convolution, which can be bad for the result. While performing pixel shuffle at the last layer of the network to recover the LR image does not need padding operation. combining each pixel on multiple-channel feature maps into one r × r square area in the output image. Thus, each pixel on feature maps is equivalent to the sub-pixel on the generated output imageas in fig. 2.14.

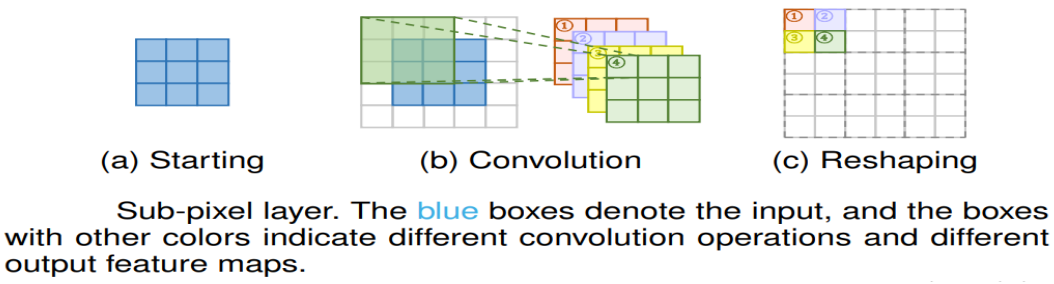


Figure 2.14 Subpixel Layer

Sub-pixel convolution involves two fundamental processes: a general convolutional operation followed by the rearrangement of pixels. The output channel of the last layer must be C × r × r so that the total number of pixels is consistent with the HR image to be obtained. In the ESPCN network that will be further discussed in chapter 3, the interpolation method is implicitly contained in the convolutional layers, it can be learned automatically by the network. Since the convolution operations are implemented on smaller size LR images, the efficiency is much higher.[8]

2.4 Network Designs

Researchers find that better reconstruction performance can be obtained by adding more convolutional layers to increase the receptive field. However, directly stacking the layers will cause Vanishing/exploding gradients and degradation problem. Meanwhile, adding more layers will lead to a Higher training error and more expensive computational cost. So researchers created network designs to avoid those problems. [9]

There are several types of network designs, 3 of them are discussed in this section since they are the most common used in super resolution.

- Residual Learning
- Recursive Learning
- Dense Connections

Other network designs include Multipath Learning, Attention Mechanism, Advanced conclusion, ...etc. are found in this reference. [9]

2.4.1 Residual Networks

To solve the gradient vanishing problem associated with ultra-deep networks, the He et al. proposed a residual learning framework [10] Residual connections are simply connections between a layer and layers after the next. This idea is clearly illustrated in fig 2.15

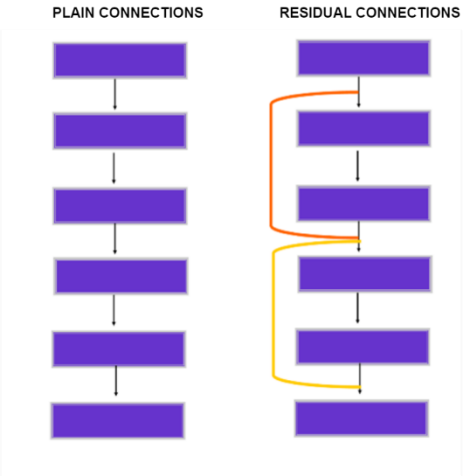


Figure 2.15 Residual Connections

A residual network is formed by stacking several residual blocks together. the training error tends to increase. However, deep **ResNets** are capable of forming an identity function that maps to an activation earlier in the network when a specific layer's activation tends to zero deeper in the network.

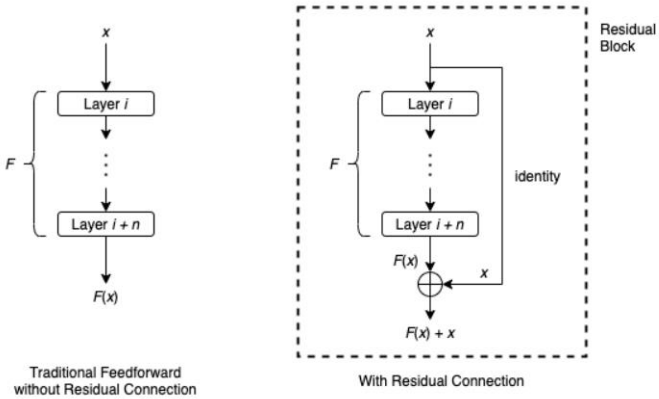


Figure 2.16 Residual Block

A residual block in fig. 2.16 is a stack of layers set in such a way that the output of a layer is taken and added to another layer deeper in the block. The non-linearity is then applied after adding it together with the output of the corresponding layer in the main path. This by-pass connection is known as the shortcut or the skip-connection.

Performance improvement is achieved whenever the extra layers learn some meaningful information from the data. While, the presence of the residual blocks prevents the loss of performance whenever, the activations tend to vanish or explode. One important thing to note here is that the skip connection is applied before the RELU activation as shown in the diagram above. Research has found that this has the best results.

Skip connections

ResNet first introduced the concept of skip connection The fig. 2.17 on the left is stacking convolution layers together one after the other. On the right we still stack convolution layers as before but we now also add the original input to the output of the convolution block. This is called skip connection

Figure 2.17 Skip Connection

The importance of skip connection is that it mitigates the problem of vanishing gradient by allowing this alternate shortcut path for gradient to flow through. They allow the model to learn an identity function which ensures that the higher layer will perform at least as good as the lower layer, and not worse.

2.4.2 Recursive Learning

One of the basic network-based learning strategies is to use the same module for recursively learning high-level features. This method also minimizes the parameters as the strategy is based on the same module being updated recursively, as shown in the following fig. 2.18.

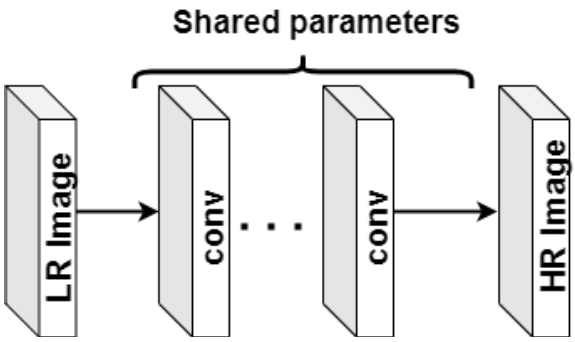


Figure 2.18 Recursive Learning

while reducing the parameters, recursive learning networks can learn the complex representation of the data at the cost of computational performance. Additionally, the increase in computational requirements may result in an exploding or vanishing gradient. Thus, recursive learning is often used in combination with multi- supervision or residual learning for minimizing the risk of exploding or vanishing gradient.[4]

2.4.3 Dense Connections

The dense block utilizes all the features maps generated by the previous layers as inputs and its feature inputs, using dense blocks will increase the reusability of the features while resolving the gradient vanishing problem. Furthermore, the dense connections also minimize the model size by utilizing a small growth rate and enfolding the channels using concatenated input features. Dense connections are used in SR to connect the low-level and high-level features maps for reconstructing a high-quality fine-detailed HR image, as shown in the following fig.2.19 [4]

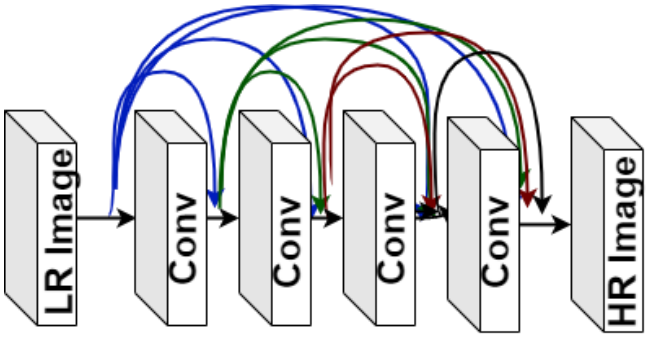


Figure 2.19 Dense Connection

2.5 Deep Learning

Deep learning is the core of this project, where all the implemented models are deep learning. Also deep learning models made a huge progress in super resolution in this section basics of any deep learning model is discussed, there are 3 types of models this project presents (CNN, Autoencoder, and GAN).

2.5.1 Convolutional Neural Network

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics. fig. 2.20 CNN architecture where each layer is discussed briefly in this section.

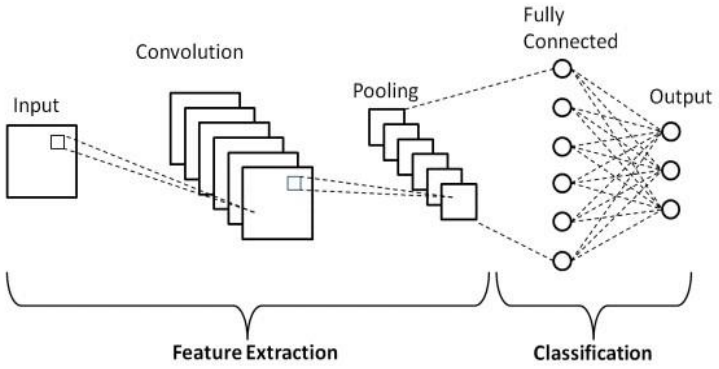


Figure 2.20 CNN Architecture

1. **Convolutional Layers**

Convolutional layers are considered the core building block of CNN architectures as illustrates convolutional layers work as feature extractor and transform the input data by using a patch of locally connecting neuron from the previous layer, the layer will compute a dot product between the region of the neurons in the input layer and the weights to which they locally connected in the output layer.

Convolution is a mathematical term to describe the process of combining two functions to produce a third function. This third function or the output is called a Feature Map, a convolution is the action of using a filter or kernel that is applied to the input. In most cases, the input is an image. Convolutions are executed by sliding the filter or kernel over the input image. This sliding process is a simple matrix multiplication or dot product.

Convolutional layers components:

a. **Kernel**

Depending on the values on the kernel matrix in fig. 2.21, applying kernel produces scalar outputs, convolving with different kernels produces interesting feature maps that can be used to detect different features, Convolution keeps the spatial relationship between pixels by

learning image features over the small segments that passes over on the input image, the next figure shows an example of applying kernel in an image.

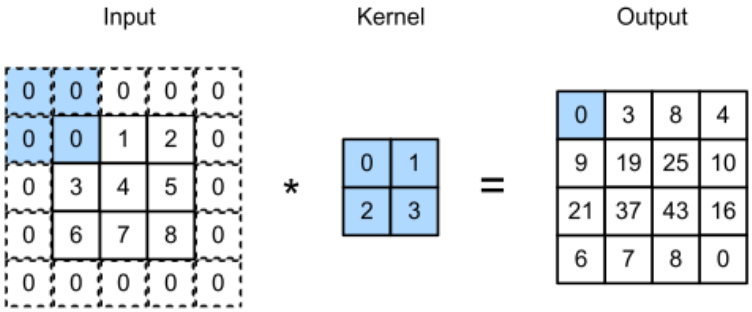


Figure 2.21 Kernel

b. Strides

Stride is the number of pixels shifts over the input matrix as in fig.2.22. When stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on.

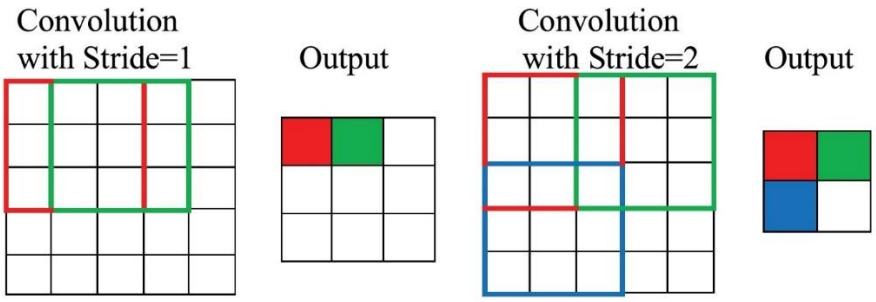


Figure 2.22 Strides

c. Padding

Sometimes filter does not fit perfectly fit the input image. We have two options: Pad the picture with zeros so that it fits as in fig. 2.23. Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.

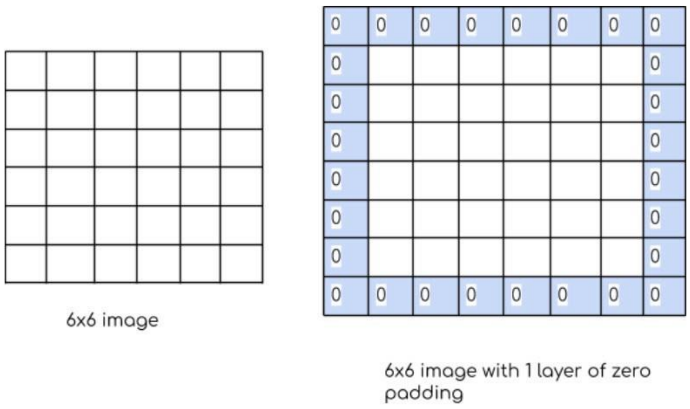


Figure 2.23 Zero Padding

2. Pooling Layer

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

There are two types of Pooling as in fig. 2.24: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that Max Pooling performs a lot better than Average Pooling.

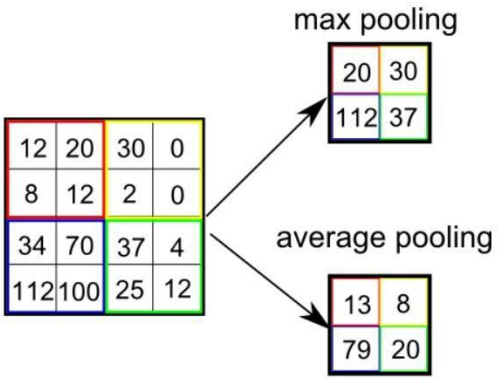


Figure 2.24 Max Pooling and Average Pooling

3. Fully Connected Layer:

Fully Connected layers are those layers where all the inputs from one layer are connected to every activation unit of the next layer as in fig. 2.25. which compiles the data extracted by previous layers to form the final output.

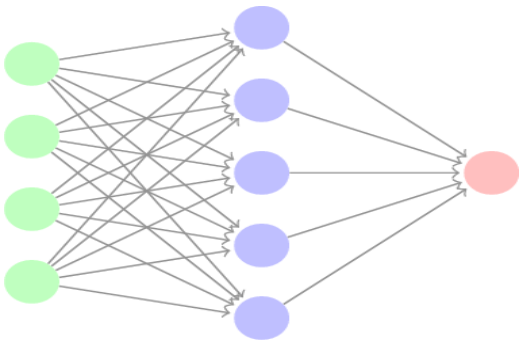


Figure 2.25 Fully Connected Layer

4. Activation Functions

The activation function is a non-linear transformation that we do over the input before sending it to the next layer of neurons or finalizing it as output.

Types of activation functions:

a. Step Function

Step Function is one of the simplest kind of activation functions. In this, we consider a threshold value and if the value of net input say y is greater than the threshold then the neuron is activated.

b. Sigmoid Function

This is a smooth function and is continuously differentiable. The biggest advantage that it has over step and linear function is that it is non-linear. This is an incredibly cool feature of the sigmoid function. This essentially means that when I have multiple neurons having sigmoid function as their activation function the output is nonlinear as well. The function ranges from 0-1 having an S shape.

c. Relu

The ReLU function is the Rectified linear unit. It is the most widely used activation function. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time. If the input is negative it will convert it to zero and the neuron does not get activated.

d. Leaky Relu

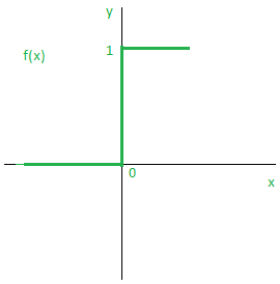
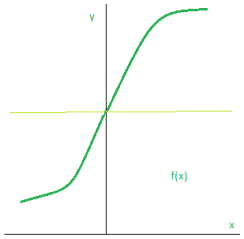
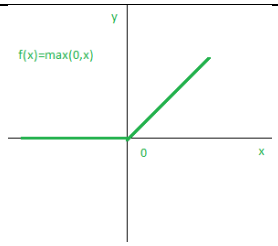
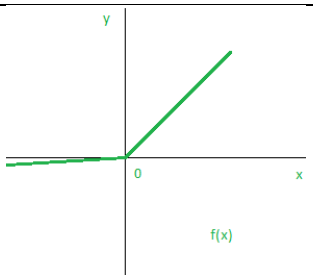
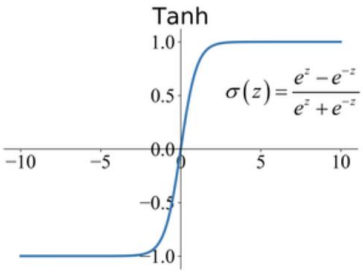
An improved version of the ReLU function. Instead of defining the Relu function as 0 for x less than 0, we define it as a small linear component of x.

e. Tanh

Became preferred over the sigmoid function as it gave better performance for multi-layer neural networks.

The following table 2.2 view the activation functions mathematically and graphically

Table 2.3 Activation Functions

Activation Function	Mathematically	Graphically
Step Function	f(x) = 1 if x >= 0 f(x) = 0 if x <= 0	
Sigmoid Function	$\frac{1}{1+e^{-x}}$	
Relu	f(x) = max (0,x)	
Leaky Relu	f(x) = ax, x < 0 f(x) = x, otherwise	
Tanh	$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	

2.5.2 Generative Adversarial Network

Generative adversarial networks (GAN) represent a shift in architecture design for deep neural networks. This new architecture pits two or more neural networks against each other in

adversarial training to produce generative models. Also its an approach to generative modeling using deep learning methods, such as convolutional neural networks. [11]

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.
It can also be considered as a secondary field of ML algorithms inspired by the brain structure and functionality. In the applications of image identification, speech synthesis, text mining applications by receiving a distinct kind of data that hierarchical models can be built by representing probability distributions. Deep learning dependent on an end to end wireless communication system with conditional GANs using Deep Neural Networks (DNNs) do function of message passing like encoding, decoding, modulation, and demodulation. For this, the right judgement of immediate channel transfer state is required to transfer DNN.

Generative and discriminative models

Machine learning (ML) and deep learning can be described by two terms: generative and discriminative modeling. When discussing the machine learning techniques that most people are familiar with, the thinking of a discriminative modeling technique, such as classification.

The GAN model architecture involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real, from the domain, or fake, generated by the generator model.

- **Generator:** Model that is used to generate new plausible examples from the problem domain.
- **Discriminator:** Model that is used to classify examples as real (from the domain) or fake (generated).

Generative models tackle a more difficult task than analogous discriminative models. It has to model more.

A generative model for images might capture correlations like "things that look like boats are probably going to appear near things that look like water" and "eyes are unlikely to appear on foreheads." These are very complicated distributions.

1. **The Generator Model**

The generator model takes a fixed-length random vector as input and generates a sample in the domain. The vector is drawn from randomly from a Gaussian distribution, and the vector is used to seed the generative process. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution. This vector space is referred to as a latent space, or a vector space comprised of latent variables. Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.

It’s often referred to latent variables, or a latent space, as a projection or compression of a data distribution. That is, a latent space provides a compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples as in fig. 2.26.

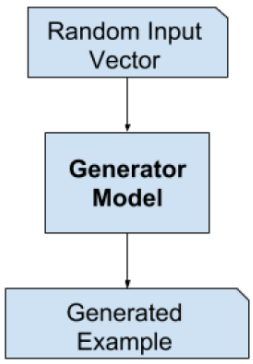


Figure 2.26 Generator Model

2. **The Discriminator Model**

The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated). The real example comes from the training dataset. The generated examples are output by the generator model. The discriminator

is a normal classification model. After the training process, the discriminator model is discarded as we are interested in the generator.

Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data. The most important feature of deep learning is discriminative models that can relate high dimensional sensory input sent to a class of labels.

These generative models based on deep learning impact are lesser because approximation of obstinate probabilistic computation is difficult and leads to the utmost chances of judgement. If deep learning models are applied on generative networks, then the advantage will be that deep learning models are work on big datasets. These datasets are largely dependent on high-end machines and took a long time to do model training and less time for testing.

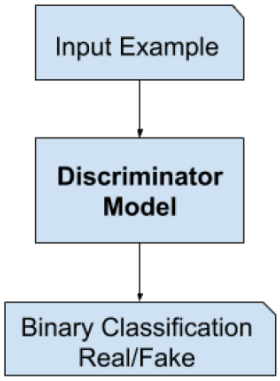


Figure 2.27 Discriminator Model

Algorithm steps:

The GAN working based on three principles:

1. Make the generative model learn, and the data can be generated employing some probabilistic representation.
2. The training of a model is done can be done in any conflicting situation.
3. Using the deep learning neural networks and using the artificial intelligence algorithms for training the complete system.

The basic idea of GAN network deployment is for unsupervised ML techniques but also proved to be better solutions for semi-supervised and reinforcement learning. These factors all together enable GAN networks as comprehensive solutions in many fields such as healthcare, mechanics, banking, etc.

GANs consists of two networks, a Generator G(x), and a Discriminator D(x). They both play an adversarial game where the generator tries to fool the discriminator by generating data similar to those in the training set. The Discriminator tries not to be fooled by identifying fake data from real data. They both work simultaneously to learn and train complex data like audio, video or image files. The generator model generates images from random noise(z) and then learns how to generate realistic images. Random noise which is input is sampled using uniform or normal distribution and then it is fed into the generator which generates an image. The generator output which are fake images and the real images from the training set is fed into the discriminator that learns how to differentiate fake images from real images. The output D(x) is the probability that the input is real. If the input is real, D(x) would be 1 and if it is generated, D(x) should be 0.

The math behind the GAN

The Discriminator and Generator play a two-player minimax game with the value function V(G, D). So, Minimax Objective function is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

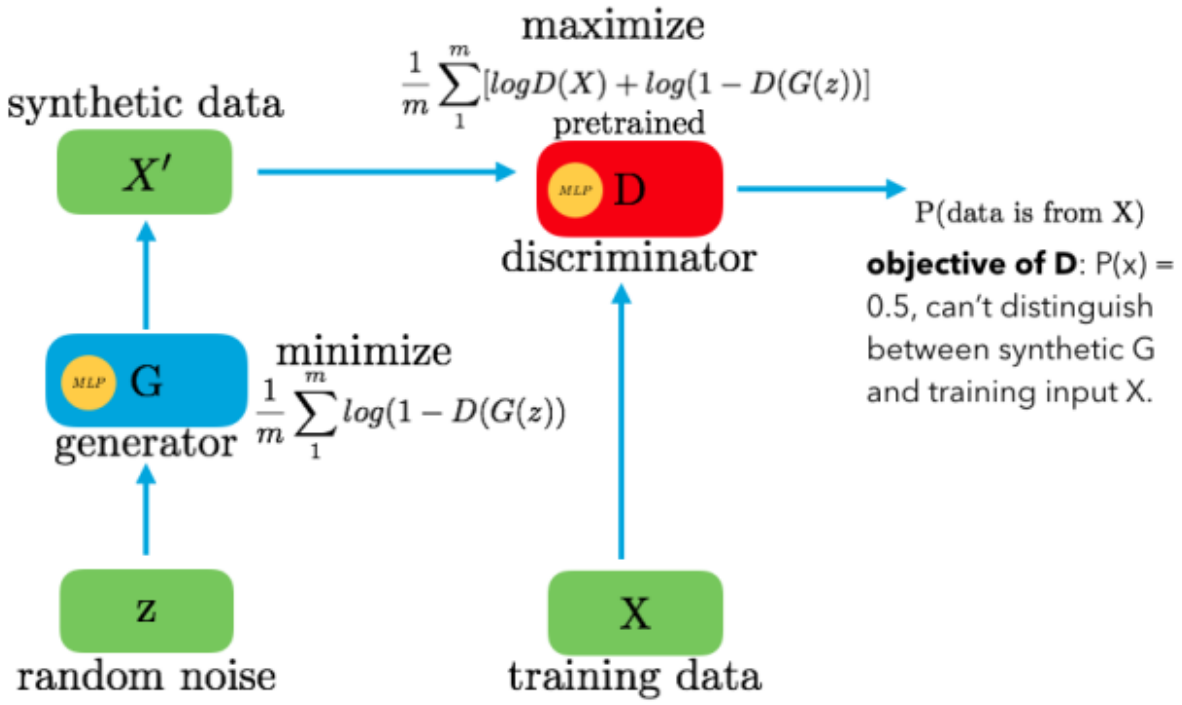


Figure 2.28 Math Behind GAN

D() gives us the probability that the given sample is from training data X. For the Generator, we want to minimize log(1-D(G(z)) i.e. when the value of D(G(z)) is high then D will assume

that G(z) is nothing but X and this makes 1-D(G(z)) very low and we want to minimize it which this even lower. For the Discriminator, we want to maximize D(X) and (1-D(G(z))). So the optimal state of D will be P(x)=0.5. However, we want to train the generator G such that it will produce the results for the discriminator D so that D won't be able to distinguish between z and X.

Now the question is why this is a minimax function. Here, the Discriminator tries to maximize the objective which is V while the Generator tries to minimize it, due to this minimizing/maximizing we get the minimax term. They both learn together by alternating gradient descent.

Training:

Because a GAN contains two separately trained networks, its training algorithm must address two complications:

- GANs must juggle two different kinds of training (generator and discriminator).
- GAN convergence is hard to identify.

The generator and the discriminator have different training processes. So how do we train the GAN as a whole? GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs.
2. The generator trains for one or more epochs.
3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.

We keep the generator constant during the discriminator training phase. As discriminator training tries to figure out how to distinguish real data from fake, it has to learn how to recognize the generator's flaws. That's a different problem for a thoroughly trained generator than it is for an untrained generator that produces random output.

Similarly, we keep the discriminator constant during the generator training phase. Otherwise the generator would be trying to hit a moving target and might never converge. It's this back and forth that allows GANs to tackle otherwise intractable generative problems. We get a toehold in the difficult generative problem by starting with a much simpler classification problem.

Conversely, if you can't train a classifier to tell the difference between real and generated data even for the initial random generator output, you can't get the GAN training started. As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake.

If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction. This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.

2.5.3 Autoencoder

Autoencoder is a type of neural network where the output layer has the same dimensionality as the input layer. In simpler words, the number of output units in the output layer is equal to the number of input units in the input layer. An autoencoder replicates the data from the input to the output in an unsupervised manner and is therefore sometimes referred to as a replicator neural network.

Architecture of autoencoders

An autoencoder consists of three components shown in fig. 2.29:

1. **Encoder:** An encoder is a feedforward, fully connected neural network that compresses the input into a latent space representation and encodes the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
2. **Code:** This part of the network contains the reduced representation of the input that is fed into the decoder.
3. **Decoder:** Decoder is also a feedforward network like the encoder and has a similar structure to the encoder. This network is responsible for reconstructing the input back to the original dimensions from the code

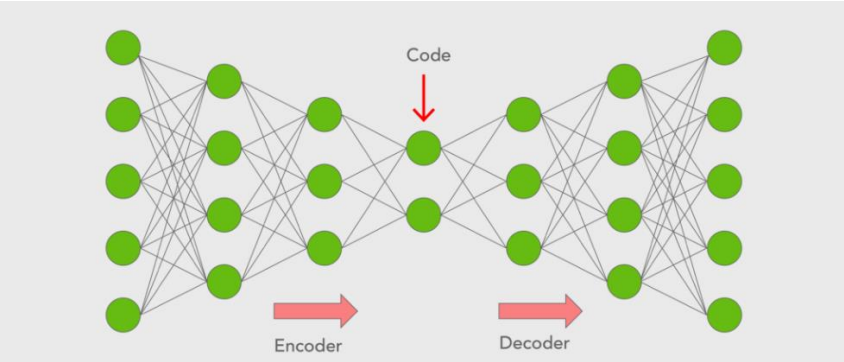


Figure 2.29 Architecture of autoencoders

First, the input goes through the encoder where it is compressed and stored in the layer called Code, then the decoder decompresses the original input from the code. The main objective of the autoencoder is to get an output identical to the input.

Note that the decoder architecture is the mirror image of the encoder. This is not a requirement but it's typically the case. The only requirement is the dimensionality of the input and output must be the same.

Types of autoencoders

There are many types of autoencoders and some of them are mentioned below with a brief description

1. **Convolutional Autoencoder:** Convolutional Autoencoders (CAE) as shown in fig. 2.30 learn to encode the input in a set of simple signals and then reconstruct the input from them. In addition, we can modify the geometry or generate the reflectance of the image by using CAE. In this type of autoencoder, encoder layers are known as convolution layers and decoder layers are also called deconvolution layers. The deconvolution side is also known as up sampling or transpose convolution.

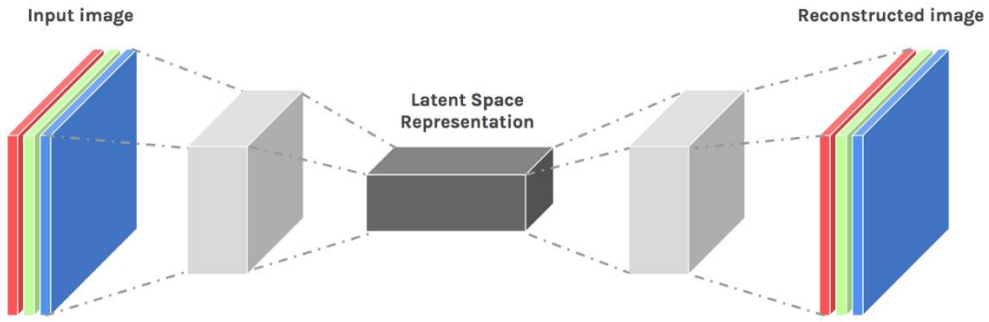


Figure 2.30 Convolutional Autoencoder

2. **Variational Autoencoders:** This type of autoencoder can generate new images just like GANs. Variational autoencoder models tend to make strong assumptions related to the distribution of latent variables. They use a variational approach for latent representation learning, which results in an additional loss component and a specific estimator for the training algorithm called the Stochastic Gradient Variational Bayes estimator. The probability distribution of the latent vector of a variational autoencoder typically matches the training data much closer than a standard autoencoder. As VAEs are much more flexible and customizable in their generation behavior than GANs, they are suitable for art generation of any kind.

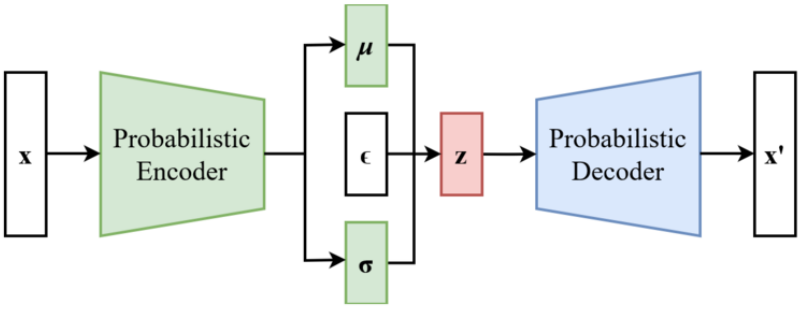


Figure 2.31 Variational Autoencoders

3. **Denoising autoencoders:** Denoising autoencoders add some noise to the input image as shown in fig. 2.32 and learn to remove it. Thus, avoiding copying the input to the output without learning features about the data. These autoencoders take a partially corrupted input while training to recover the original undistorted input. The model learns a vector field for mapping the input data towards a lower-dimensional manifold which describes the natural data to cancel out the added noise. By this means, the encoder will extract the most important features and learn a more robust representation of the data.

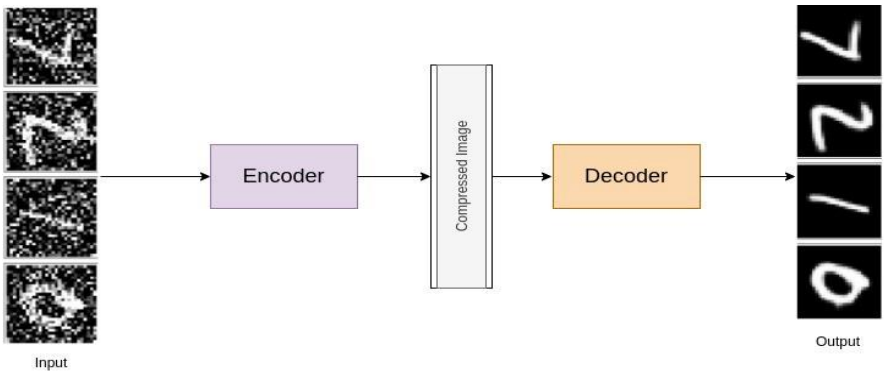


Figure 2.32 Denoising Autoencoders

2.5.4 Key features of deep learning

1. **Gradient Descent**

By adjusting the weights to lower the loss, we are performing gradient descent. This is an 'optimization' problem, Backpropagation is simply the method by which we execute gradient descent, Gradients (also called slope) are the direction of a function at a point, its magnitude signifies how much the function is changing at that point, as shown in figure 2.33.

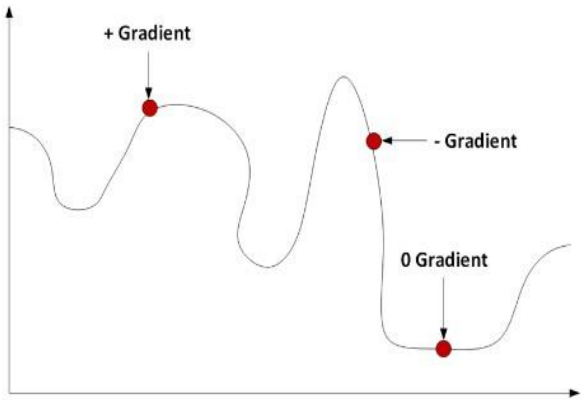


Figure 2.33 Gradient Descent

2. Back Propagation

Backpropagation in neural network is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network as shown in fig. 2.34.

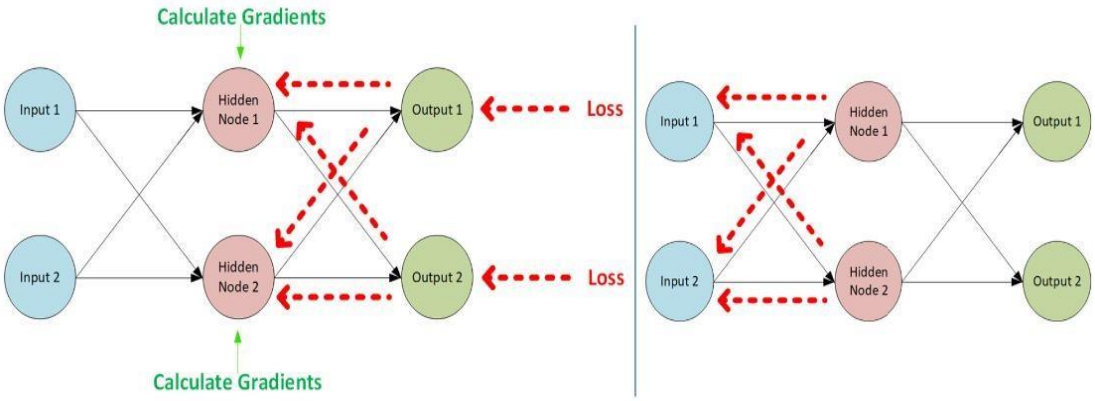


Figure 2.34 Back Propagation

3. Over Fitting

Overfitting occurs when our model fits near perfectly to training data, fig 2.35 shows the different between overfitting and under fitting and good fitting.

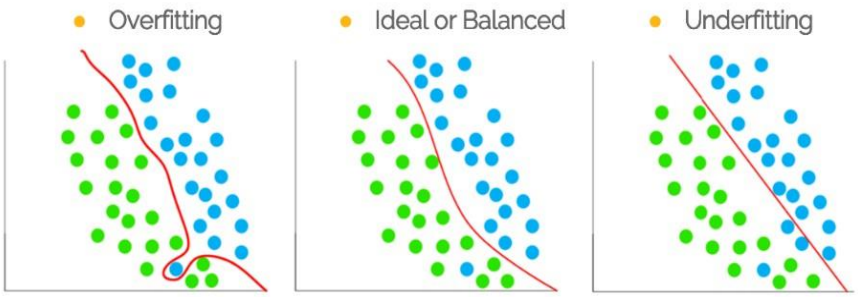


Figure 2.35 Over fitting, Balanced, Under fitting

The high variance of the model performance is an indicator of an overfitting problem. The training time of the model or its architectural complexity may cause the model to overfat. If the model trains for too long on the training data or is too complex, it learns the noise or irrelevant information within the dataset. fig. 2.36 shows signs of overfitting from learning curves.

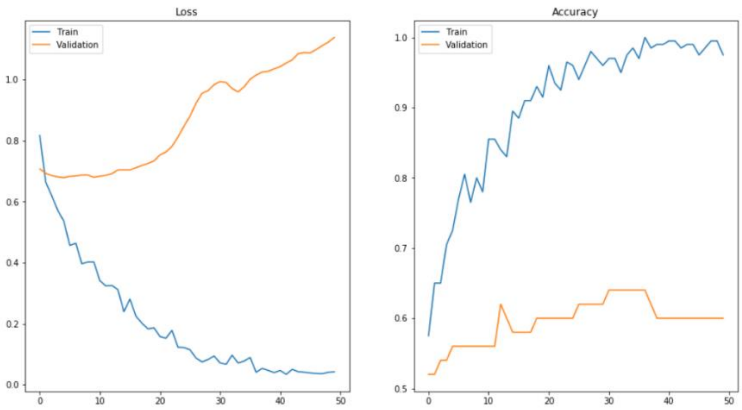


Figure 2.36 Signs of Overfitting

4. Early Stopping

Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset to prevent from overfitting as shown in fig. 2.37.

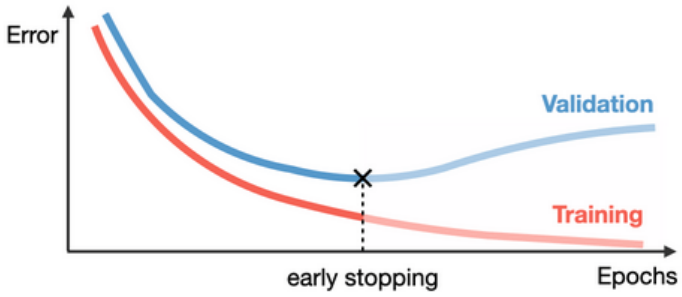


Figure 2.37 Early Stopping

5. Dropout

Dropouts are the regularization technique that is used to prevent overfitting in the model. Dropouts are added to randomly switching some percentage of neurons of the network. When the neurons are switched off the incoming and outgoing connection to those neurons is also switched off as shown in fig. 2.38. This is done to enhance the learning of the model.

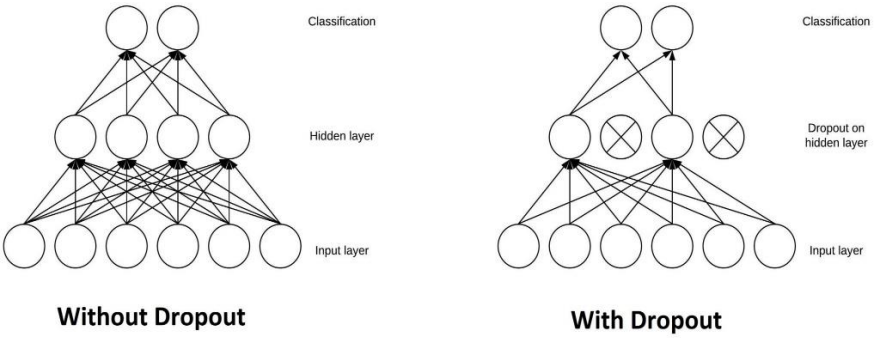


Figure 2.38 Dropout

2.6 Learning Strategy

According to different strategies, the DL-based SISR models can be mainly divided into supervised learning methods and unsupervised learning methods.

Supervised Learning: In supervised learning SISR, researchers compute the reconstruction error between the ground-truth image I_y and the reconstructed image I_{SR}.

$$\Theta_f = \arg \Theta_f \ \min L(I_{SR} , I_y)$$

Alternatively, researchers may sometimes search for a mapping ϕ , such as a pre-trained neural network, to transform the images or image feature maps to some other space and then compute the error:

$$\Theta_f = \arg \Theta_f \ \min L(\phi (ISR) , \phi(Iy))$$

Among them, L is the loss function which is used to minimize the gap between the reconstructed image and ground-truth image. According to different loss functions, the model can achieve different performances. Therefore, an effective loss function is also crucial for SISR.

Unsupervised Learning: In unsupervised learning SISR, the way of evaluation and parameter up-gradation is changing by different unsupervised learning algorithms. For example, ZSSR uses the test image and its downscaling images with the data augmentation methods to build the “training dataset” and then applies the loss function to optimize the model. In CinCGAN, a model consists of two CycleGAN is proposed, where parameters are upgraded through optimizing the generator-adversarial loss, the cycle consistency loss, the identity loss, and the total variation loss together in each cycle. [9]

2.6.1 Loss Functions

For any application in deep learning, the selection of the loss functions is critical, and in SR, these functions are used to measure the error in the reconstruction of HR, which further helps

optimize the model iteratively. In this section, we will briefly introduce several commonly used loss functions. [9]

1. Pixel Loss

This is the simplest and common type of loss function used in training super-resolution networks. which aims to measure the difference between two images on pixel basis so that these two images can converge as close as possible. It mainly includes the L1 loss, Mean Square Error (MSE Loss), and Charbonnier loss (a differentiable variant of L1 loss):

$$\mathcal{L}_{L1}(I_{SR}.I_y) = \frac{1}{hwc}\sum_{i.j.k} \left|I_{SR}^{i.j.k} - I_y^{i.j.k}\right|.$$

$$\mathcal{L}_{MSE}(I_{SR}.I_y) = \frac{1}{hwc}\sum_{i.j.k} \left(I_{SR}^{i.j.k} - I_y^{i.j.k}\right)^2.$$

$$\mathcal{L}_{Char}(I_{SR}.I_y) = \frac{1}{hwc}\sum_{i.j.k} \sqrt{\left(I_{SR}^{i.j.k} - I_y^{i.j.k}\right)^2 + \epsilon^2},$$

where, h, w, and c are the height, width, and the number of channels of the image. ϵ is a numerical stability constant, usually setting to 10^{-3}. Since most mainstream image evaluation indicators are highly correlated with pixel-by-pixel differences, pixel loss is still widely sought after. However, the image reconstructed by this type of loss function usually lacks high-frequency details, so it is difficult to obtain excellent visual effects. Training with pixel loss optimizes PSNR, but doesn't directly optimize the perceptual quality, and hence generates images which might not be pleasing to the human eye.

2. Perceptual Loss

Perceptual loss or content loss tries to match the high-level features in a generated image with a given HR output image. This is achieved by taking a pre-trained network, like VGG, and using the difference of feature outputs between predicted and output images as loss. and can be further expressed as the Euclidean distance between the high-level representations of these two images:

$$\mathcal{L}_{\text{Cont}}\left(I_{SR}.I_y.\phi\right) = \frac{1}{h_l w_l c_l}\sum_{i.j.k}\left(\phi_{(l)}^{i.j.k}(I_{SR}) - \phi_{(l)}^{i.j.k}(I_y)\right).$$

where $\emptyset$ represents the pre-trained classification network and $\emptyset_{(L)}$ (IHQ) represents the high-level representation extracted from the l layer of the network. h_l, w_l, and c_l are the height, width, and the number of channels of the feature map in the lth layer respectively. With this method, the visual effects of these two images can be as consistent as possible. Among them, VGG and ResNet are the most commonly used pre-training classification networks.

3. Adversarial Loss

Used in all GAN-related architectures, adversarial loss helps in fooling the discriminator and generally produces images which have better perceptual quality. ESRGAN model that is discussed in ch.3 adds an extra variant of this by using the relativistic discriminator, and thus instructing the network not only to make fake images more real, but also to make real images look more fake.

$$\mathcal{L}_{\text{Adversarial}}\,(I_x.\,G.\,D) = \sum_{n=1}^{N} -\log D\big(G(I_x)\big)$$

where G(ILQ) is the reconstructed SR image, G and D represent the Generator and the Discriminator, respectively.

2.6.2 Batch normalization

In order to accelerate and stabilize training of deep CNNs, Sergey et al. propose batch normalization (BN) to reduce internal covariate shift of networks. Specifically, they perform normalization for each mini-batch and train two extra transformation parameters for each channel to preserve the representation ability. Since the BN calibrates the intermediate feature distribution and mitigates vanishing gradients, it allows using higher learning rates and being less careful about initialization. Thus this technique is widely used by SR models.

However, Lim et al. argue that the BN loses the scale information of each image and gets rid of range flexibility from networks. So they remove BN and use the saved memory cost (up to 40%) to develop a much larger model, and thus increase the performance substantially. Some other models also adopt this experience and achieve performance improvements.[12]

2.6.3 Optimizers

Optimizers are algorithms or methods used to minimize an error function (loss function) or to maximize the efficiency of production. Optimizers are mathematical functions which are dependent on model's learnable parameters i.e. Weights & Biases. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

1. Gradient Descent (GD)

This is the most basic optimizer that directly uses the derivative of the loss function and learning rate to reduce the loss and achieve the minima. This approach is also adopted in backpropagation in neural networks where the updated parameters are shared between

different layers depending upon when the minimum loss is achieved. It is easy to implement and interpret the results, but it has various issues.

2. **Stochastic Gradient Descent**

This is a changed version of the GD method, where the model parameters are updated on every iteration. It means that after every training sample, the loss function is tested and the model is updated. These frequent updates result in converging to the minima in less time, but it comes at the cost of increased variance that can make the model overshoot the required position.
But an advantage of this technique is low memory requirement as compared to the previous one because now there is no need to store the previous values of the loss functions.

3. **Mini-Batch Gradient Descent**

Another variant of this GD approach is mini-batch, where the model parameters are updated in small batch sizes. It means that after every n batches, the model parameters will be updated and this ensures that the model is proceeding towards minima in fewer steps without getting derailed often. This results in less memory usage and low variance in the model.

4. **Momentum Based Gradient Descent**

Based on the first-order derivative of the loss function, we are back-propagating the gradients. The frequency of updates can be after every iteration, a batch, or at the last, but we are not considering how many updates we have in the parameters.
If this history element is included in the next updates, then it can speed the whole process and this is what momentum means in this optimizer. This history element is like how our mind memorizes things. If you are walking on a street and you cover a pretty large distance, then you will be sure that your destination is some distance ahead and you will increase your speed. This element depends on the previous value, learning rate, and a new parameter called gamma, which controls this history update. The update rule will be something like $w = w – v$, where v is the history element.

5. **Nesterov Accelerated Gradient (NAG)**

The momentum-based GD gave a boost to the currently used optimizers by converging to the minima at the earliest, but it introduced a new problem. This method takes a lot of uturns and oscillates in and out in the minima valley adding to the total time. The time taken is still way too less than normal GD, but this issue also needs a fix and this is done in NAG.

6. **Adagrad**

Till now we are only focusing on how the model parameters are affecting our training, but we haven't talked about the hyper-parameters that are assigned constant value throughout the training. One such important hyper-parameter is learning rate and varying this can change the pace of training.

One disadvantage of this approach is that the learning rate decays aggressively and after some time it approaches zero.

7. **RMSProp**

It is an improvement to the Adagrad optimizer. This aims to reduce the aggressiveness of the learning rate by taking an exponential average of the gradients instead of the cumulative sum of squared gradients. Adaptive learning rate remains intact as now exponential average will punish larger learning rate in conditions when there are fewer updates and smaller rate in a higher number of updates.

8. **Adam**

Adaptive Moment Estimation combines the power of RMSProp (root-mean-square prop) and momentum-based GD. In Adam optimizers, the power of momentum GD to hold the history of updates and the adaptive learning rate provided by RMSProp makes Adam optimizer a powerful method. It also introduces two new hyper-parameters beta1 and beta2 which are usually kept around 0.9 and 0.99 but you can change them according to your use case.

2.7 Transfer Learning and Pretrained Models

In practice, Not any machine can train an entire Convolutional Network from scratch as it is relatively rare to have a dataset of sufficient size as one of the main challenges in machine learning is having the suitable amount of data as for small dataset though we can use a pertained model that was used to be trained on a larger dataset (e.g. image net which contains 1.2 million images with 1000 categories) to exploits its data in training on the smaller datasets, in the next table shown a comparison between the traditional machine learning and transfer learning. [13][14]

The three major transfer learning scenarios look as follows:

1. **ConvNet as fixed feature extractor:**

Take a ConvNet pertained on image net, remove the last fully connected layer (as those layers outputs the 1000 class scores for a different task), then treat the rest of the ConvNet as fixed feature extractor for the new dataset.

2. **Train some layers and leave the other frozen:**

Lower layer refers to general features (problem independent), while higher layers refer to specific features (problem dependent). Here we play that dichotomy by choosing how much we want to adjust the weights of the network (frozen layers do not change during training). Usually, if you have a small dataset and a larger number of parameters you will leave more layers frozen to avoid over fitting. By contrast, if the dataset is large and the number of parameters is small, you can improve your model by training more layers to the new task since over-fitting is not an issue.

3. **Freeze the convolutional base:**

This case corresponds to an extreme situation of the train/freeze trade-off. The main idea is to keep the convolutional base in its original form and then use its outputs to feed the classifier. you are using the pertained model as a fixed feature extraction mechanism, which can be useful if you are short on computational power, your dataset is small and pre-trained model solves a problem very similar to the one you want to solve, fig 2.39 illustrate the transfer learning strategies.

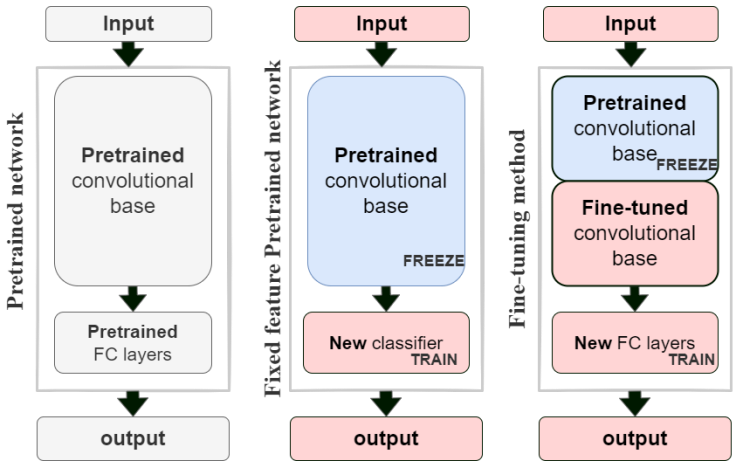


Figure 2.39 Transfer Learning strategies

Pretrained Models

In Machine Learning, a pre-trained model falls under the category of transfer learning. Pre Trained models are machine learning models that are trained, developed and made available by other developers. They are generally used to solve problems based on deep learning and are always trained on a very large dataset. ESRGAN is a one of our implemented models where is a pretrained model and we used transfer learning to implement it. This gave us less time to work on that model.

2.8 Methods of assessments

2.8.1 Subjective methods

In subjective testing a group of people are asked to give their opinion about the quality of each image. There are several types of subjective methods like the following:[15]

1. **Single stimulus categorical rating**

In this method, test images are displayed on a screen for a fixed amount of time, after that, they will disappear from the screen and observers will be asked to rate the quality of them on an abstract scale containing one of the five categories: excellent, good, fair, poor, or bad. All of the test images are displayed randomly. In order to avoid quantization artifacts, some methods use continuous rather than categorical scales

2. **Double stimulus categorical rating**

This method is similar to single stimulus method. However, in this method both the test and reference images are being displayed for a fixed amount of time. After that, images will disappear from the screen and observers will be asked to rate the quality of the test image according to the abstract scale described earlier.

3. **Ordering by force-choice pair-wise comparison**

In this type of subjective assessment, two images of the same scene are being displayed for observers. Afterward, they are asked to choose the image with higher quality. Observers are always required to choose one image even if both images possess no difference. There is no time limit for observers to make the decision. The drawback of this approach is that it requires more trials to compare each pair of conditions

4. **Pair-wise similarity judgment**

As we mentioned before, in force-choice comparison, observers are required to choose one image even if they see no difference between the pair of images. However, in pair-wise similarity judgment observers are asked not only to choose the image with higher quality, but also to indicate the level of difference between them on a continuous scale. One might be tempted to use the raw rating results such as: excellent, good, fair, and etc. for quality scores. However, these rating results are unreliable. One reason for this is that observers are likely to assign different quality scales to each scene and even distortion types

5. **Mean opinion score (MOS)**

MOS is obtained by asking observers to rate images for their quality on a particular scale. such as a scale from 1 to 5 where 1 is bad and 5 is excellent.

2.8.2 Objective methods

Objective methods depend on mathematical model to measure not human testers. There are several types of objective methods like the following.[15]

1. **Mean Square Error (MSE)**

The difference between the Pixel value of one image and the corresponding Pixel value of the other image. The MSE measures the average of the square of the errors

$$MSE = \frac{1}{NM}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}(A(i.j) - B(i.j))^2$$

2. **Peak Signal to Noise Ratio(PSNR)**

PSNR is used to calculate the ratio between the maximum possible signal power and the power of the distorting noise this ratio is used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

3. **Structure Similarity Index Method (SSIM)**

It measures the similarity between two images in 3 aspects (luminance, structure, contrast) SSIM can be expressed through these three terms as:

- L is the luminance (used to compare the brightness between two images)
- C is the contrast (used to differ the ranges between the brightest and darkest region of two images)
- S is the structure (used to compare the local luminance pattern between two images to find the similarity of the images) and α, β and γ are the positive constants.

$$\text{SSIM}(x.y) = [l(x.y)]^{\alpha} \cdot [c(x.y)]^{\beta} \cdot [s(x.y)]^{\gamma}$$

Chapter 3
Related Work

Chapter 3: Related Work

In this chapter, 12 models are discussed fig. 3.1. The preprocessing techniques used for images before being fed into the models are shown. Each model architecture will be illustrated. The hyper parameters used for each model and the dataset on which the models are trained and tested. The 7 highlighted models that have been implemented will be further discussed in chapter 4.

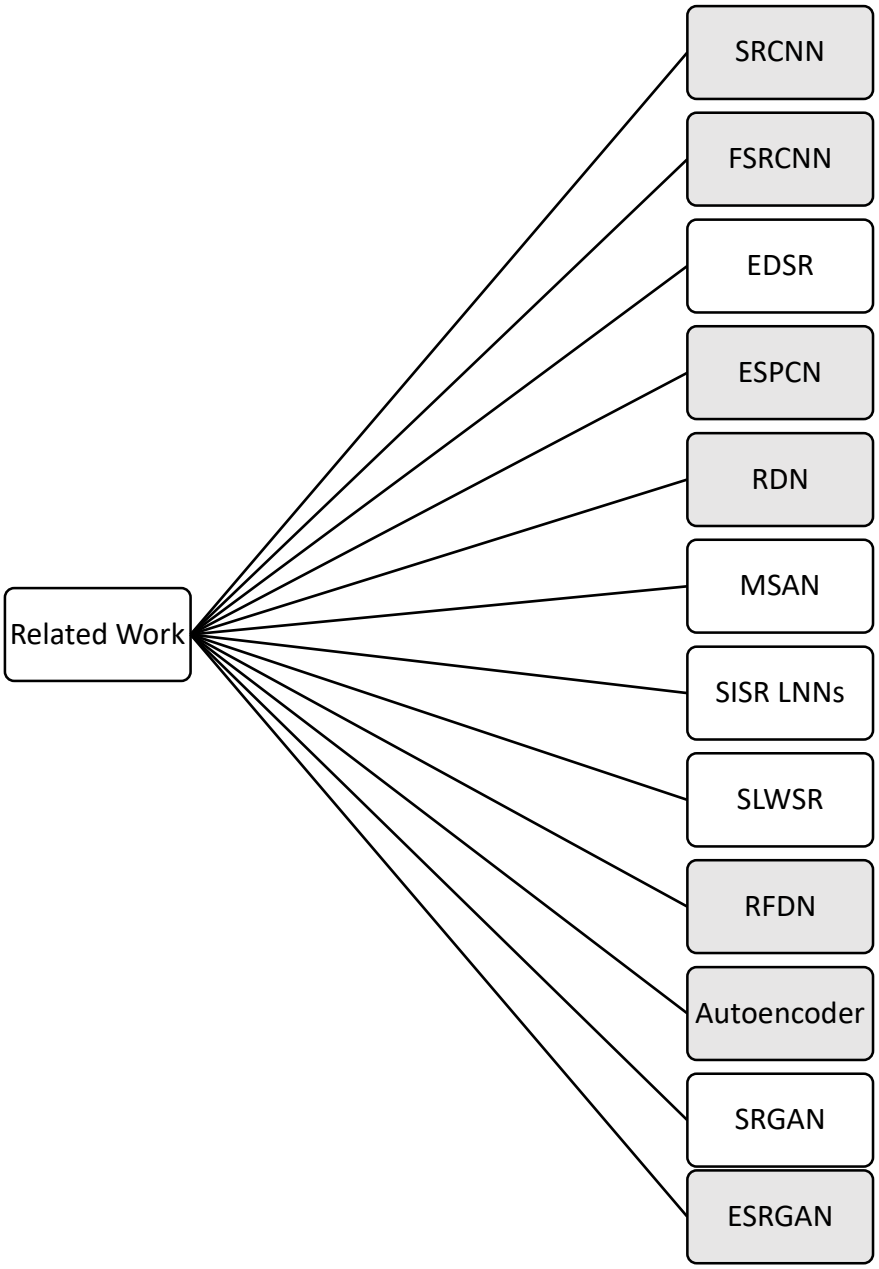


Figure 3.1 CH.3 Related Work Map

3.1 Super Resolution using deep convolution neural network

Super Resolution using deep convolution neural network (SRCNN) [16] was the first deep learning model to outperform traditional ones. It is a convolutional neural network consisting of only 3 convolutional layers shown in fig. 3.2: patch extraction and representation, non-linear mapping and reconstruction.

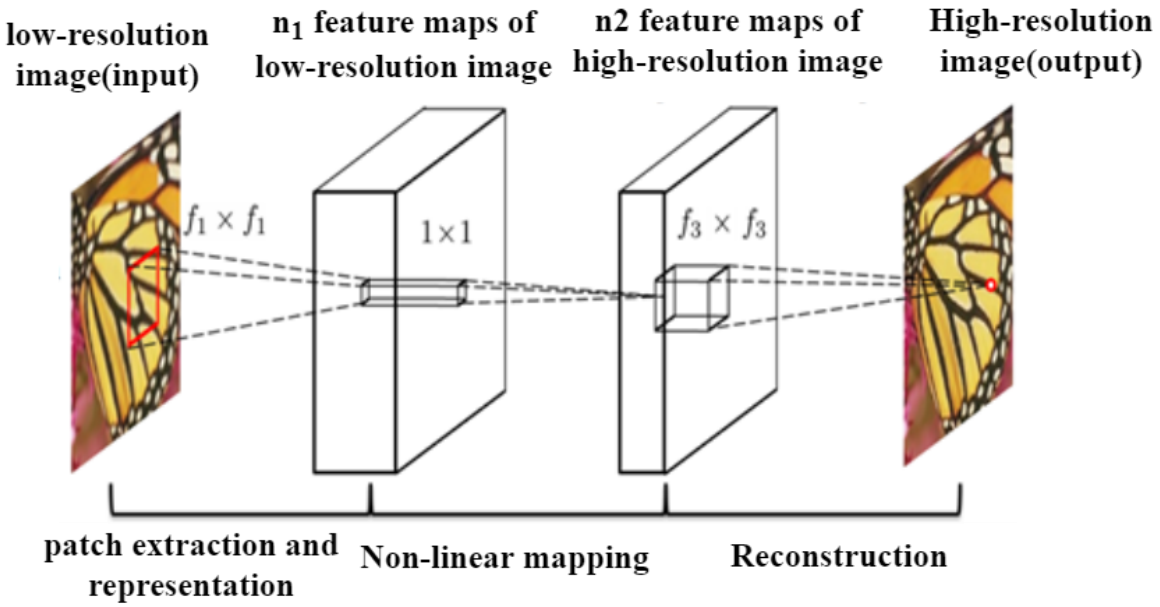


Figure 3.2 SRCNN Architecture

Before being fed into the network, an image needs to be up-sampled via bicubic interpolation. It's then converted to YCbCr color space, while only luminance channel (Y) is used by the network. The network's output is then merged with interpolated CbCr channels to produce a final color image. They chose this procedure because they were not interested in changing colors (this is the information stored in the CbCr channels), but only their brightness (the Y channel), and ultimately because human vision is more sensitive to luminance ("black and white") differences than chromatic differences. The model was implemented using PyTorch library.

The following are parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on 14k 32×32 sub-images from the same dataset on 91 images and tested on Set5 and Set14 and BSD200. The loss function used is mean squared error (MSE). MSE is used to measure the difference between the generated SR images and the ground truth HR images

3.2 Fast Super Resolution Convolution Neural Network

The proposed FSRCNN (Fast Super Resolution Convolution Neural Network) [17]is different from SRCNN mainly in three aspects. First, FSRCNN adopts the original low-resolution image as input without bicubic interpolation. A deconvolution layer is introduced at the end of the network to perform up-sampling. Second, the non-linear mapping step in SRCNN is replaced by three steps in FSRCNN, namely the shrinking, mapping, and expanding step. Third, FSRCNN adopts smaller filter sizes and a deeper network structure. These improvements provide FSRCNN with better performance but lower computational cost than SRCNN. This figure shows the network structures of the SRCNN and FSRCNN.

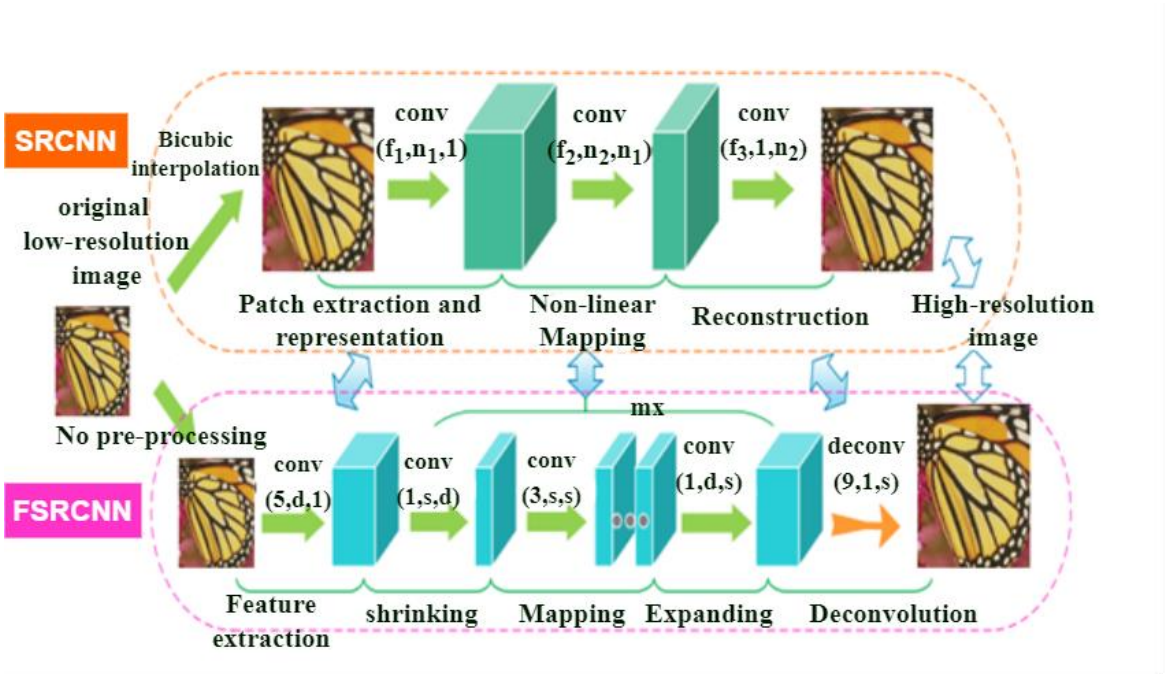


Figure 3.3 FSRCNN and SRCNN

Fig. 3.3 shows the five main steps of FRCNN which are feature extraction, shrinking, mapping, expanding and deconvolution where d is the LR feature dimension and s is the level of shrinking.

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on 14k 32×32 sub-images from the same dataset on 91 images and tested on Set5, Set14, B100 datasets. The loss function used is mean squared error (MSE).

3.3 Enhanced Deep Residual Networks for Single Image Super-Resolution

Enhanced Deep Residual Networks for Single Image Super-Resolution [18] is a super-resolution model proposed after SRResNet. SRResNet successfully solved the problems of processing time and memory consumption, but ResNet used in SRResNet is a model architecture for image classification, which is not optimal for super-resolution. Therefore, EDSR builds a more optimal model for super-resolution by removing unnecessary modules from ResNet. For example, Batch Normalization is removed because it loses range flexibility.

Architecture of EDSR

In EDSR they proposed different architecture of ResBlock which more efficient to train the model.

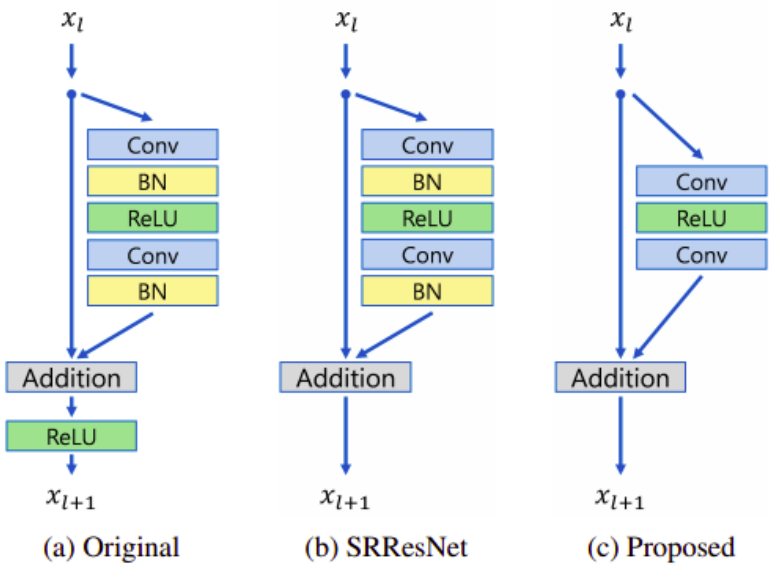


Figure 3.4 Comparison of residual block

They found that it is better to remove batch normalization layer from their model. This baseline model without batch normalization layer saves approximately 40% of memory usage during training, compared to SRResNet as shown in fig. 3.4. Consequently, they built up a larger model that has better performance than conventional ResNet structure under limited computational resources.

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on div2k dataset on 800 images and tested on Set5, Set14, B100, and Urban100 datasets. The loss function used is mean squared error (MSE). MSE is used to measure the difference between the generated SR images and the ground truth HR images.

3.4 Efficient Subpixel Convolution Network

Approaches based on the convolutional neural network like SRCNN, FSRCNN, and VDSR have some drawbacks. Firstly, CNN approaches need to use interpolation methods to up-sample the LR image. Secondly, the CNN approach increases the resolution before or at the first layer of the network which will increase the computational complexity and memory cost. The new approach ESPCN [8] (Efficient Subpixel Convolution Network) has been proposed to add an efficient sub-pixel convolutional layer to the CNN network to solve the previous mentioned problems. ESPCN increases the resolution at the very end of the network and there is no need to use the interpolation method. The network can learn a better LR to HR mapping compared to an interpolation filter upscaling before feeding into the network. Due to the reduced input image size, a smaller filter size can be used to extract features. The computational complexity and memory cost is reduced so that the efficiency can be greatly enhanced.

Network Structure

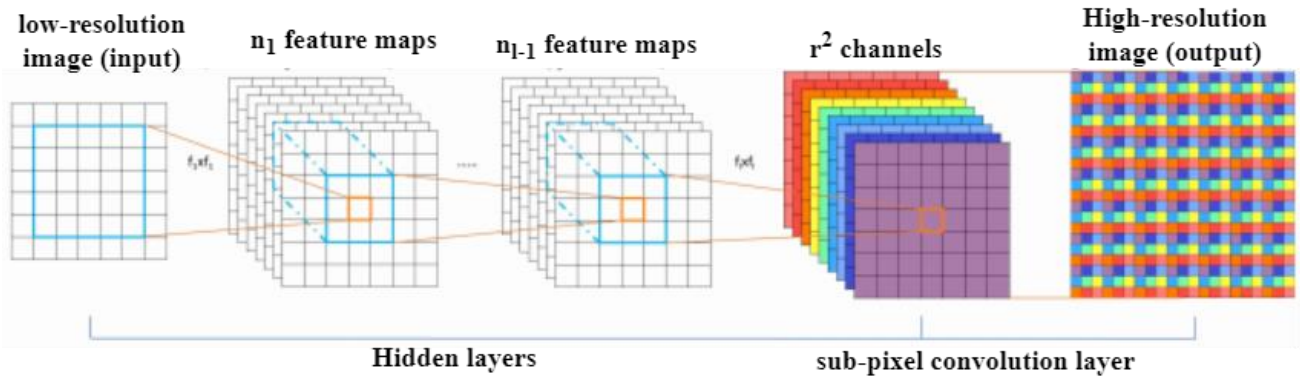


Figure 3.5 ESPCN Network Structure

The network structure of ESPCN was represented in Fig. 3.5. Suppose there are L layers for the network, the first L-1 layers are convolutional layers which obtain feature maps of the input LR images. And the last layer is the efficient sub-pixel convolutional layer to recover the output image size with a specified upscale factor.

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on 91 images and tested on Set5, Set14, BSD500, BSD300 datasets. The loss function used is mean squared error (MSE).

3.5 Residual Dense Network for Super Resolution

Residual Dense Network (RDN) [19]is constructed by combining residual learning and dense connection so that to provide full use of hierarchical feature information. The main architecture of RDN is shown in fig. 3.7. They proposed a very deep residual dense network (RDN) for image SR, where residual dense block (RDB) shown in fig. 3.6 serves as the basic build module. In each RDB, the dense connections between each layers allow full usage of local layers. The local feature fusion (LFF) not only stabilizes the training wider network, but also adaptively controls the preservation of information from current and preceding RDBs. RDB further allows direct connections between the preceding RDB and each layer of current block, leading to a contiguous memory (CM) mechanism. The local residual leaning (LRL) further improves the flow of information and gradient. Moreover, they propose global feature fusion (GFF) to extract hierarchical features in the LR space. By fully using local and global features, RDN leads to a dense feature fusion and deep supervision.

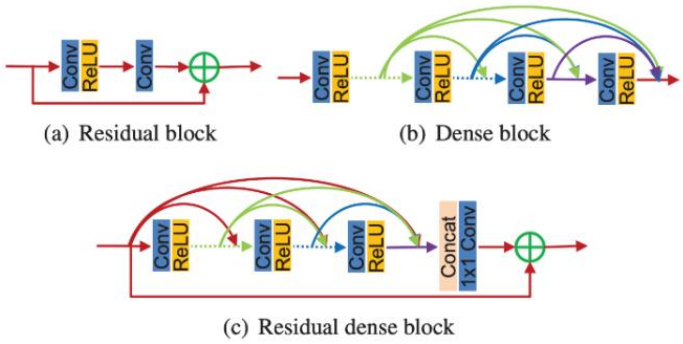


Figure 3.6 Residual block, Dense block, and Residual dense block

RDN mainly consists of four parts which are shallow feature extraction net (SFENet), residual dense blocks (RDBs) , dense feature fusion (DFF) and Up- Sampling net (UPNet).

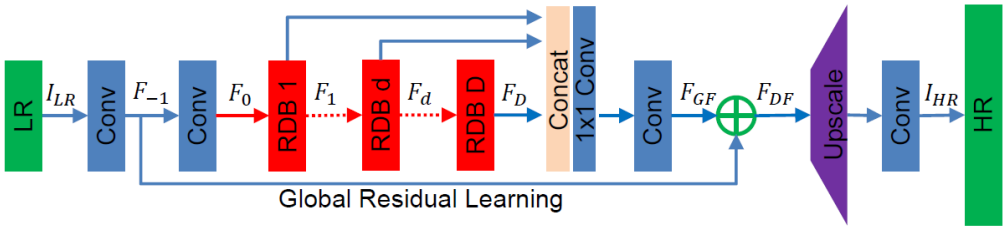


Figure 3.7 RDN Architecture

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, B100, Urban100 and Manga109 datasets. The loss function used is mean squared error (MSE).

3.6 Lightweight SISR with MSAN

They proposed Multi-Scale Spatial Attention Networks (MSAN)[20] that can adaptively give attention to the most appropriate scale of features in a specific region of the image. They designed a Multi-Scale Spatial Attention Block (MSAB) as a basic building block for the MSAN. They introduced a variant model with the recursive scheme, namely MSAN-X (extremely lightweight model).

Complicated deep neural network models have demonstrated better SR performance than conventional methods, but it is difficult to implement them on low complexity, low-power, and low-memory devices Due to the massive network parameters and convolution operations of deeper and denser networks.

MSAN is designed to extract and exploit enriched features with a reduced number of parameters. MSAN-X is designed to further reduce the number of parameters by employing a recursive scheme. Fig. 3.8 shows the architecture of MSAN which consists of MSABs with some convolution layers and feature fusion (FF) layers, dense connection, and a global feature fusion (GFF) layer.

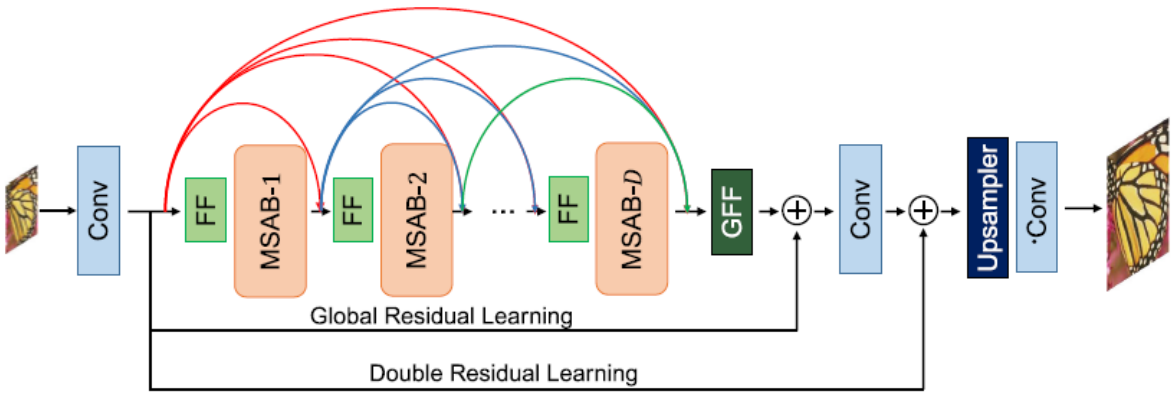


Figure 3.8 Architecture of MSAN

The architecture of MSAB is illustrated in fig. 3.9 where it consists of four parallel 1 x 1 convolution layers followed by ReLU activation function. Then dilated convolution layers with different dilation rates followed by ReLU activation function. After that a spatial attention module and finally a local feature fusion (LFF) layer.

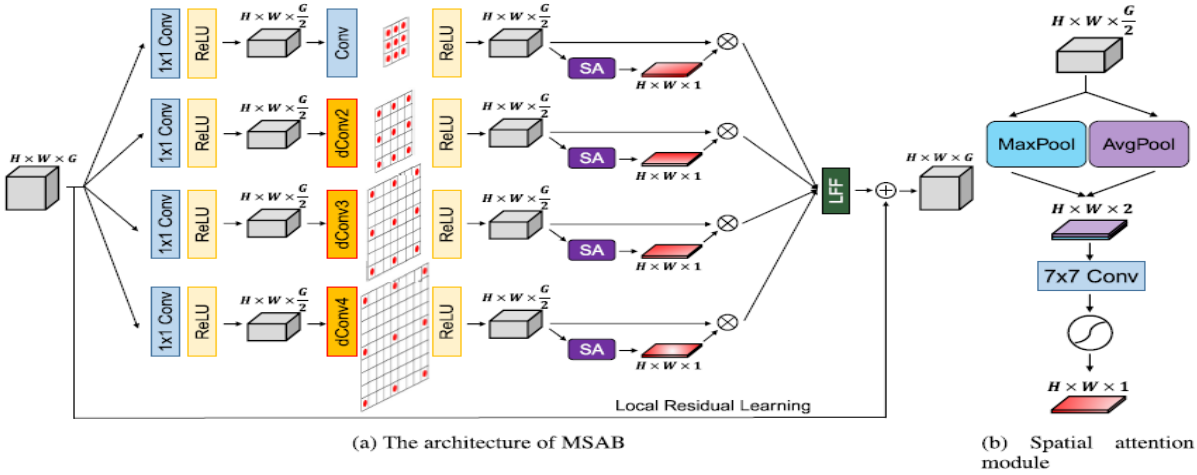


Figure 3.9 Architecture of MSAB

The following parameters of the model implementation where the learning rate of $4*10^{-4}$ is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, B100 and Urban100 datasets. The loss function used is mean squared error (MSE).

3.7 SISR using CNN based Lightweight Neural Networks

They proposed two SR-based lightweight neural networks (LNNs) [21] with hybrid residual inter-layered SR-LNN (SR-ILLNN) and dense networks which are simplified SR-LNN (SR-SLNN)

They proposed methods that were designed to produce similar image quality while reducing the number of networks parameters. Firstly, SR-ILLNN learns the feature maps, which are derived from both low-resolution and interpolated low-resolution images. Secondly, SR-SLNN is designed to use only low-resolution feature maps of the SR-ILLNN for a few more reducing the network complexity.

The architecture of SR-ILLNN is illustrated in Fig. 3.10. It consists of three parts, which are LR feature layers from convolutional layer 1 (Conv1) to Conv8, HR feature layers from Conv9 to Conv12, and shared feature layers from Conv13 to Conv15.

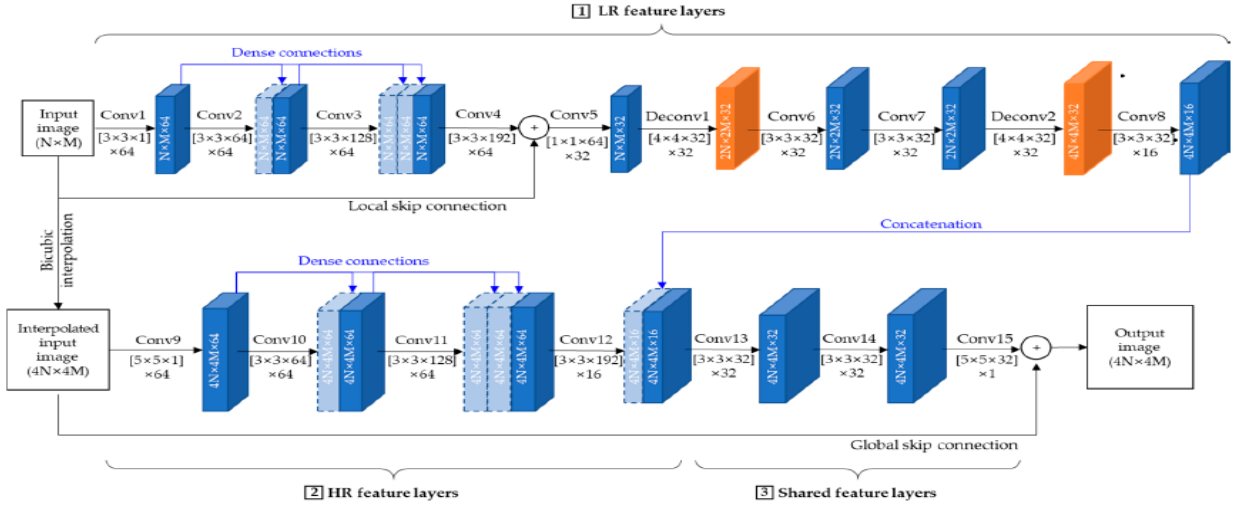


Figure 3.10 Architecture of SR-ILLNN

To reduce the network complexity of SR-ILLNN, they proposed SR-SLNN model. HR feature layers, shared feature layers, and two convolution layers between deconvolution layers of SR-ILLNN are removed. The architecture of SR-SLNN is shown in Fig. 3.11. It has seven convolution layers and two deconvolution layers.

Experimental results show that the SR-ILLNN and SR-SLNN can significantly reduce the number of parameters by 8.1% and 4.8%, respectively, while maintaining similar image quality compared to the previous methods.

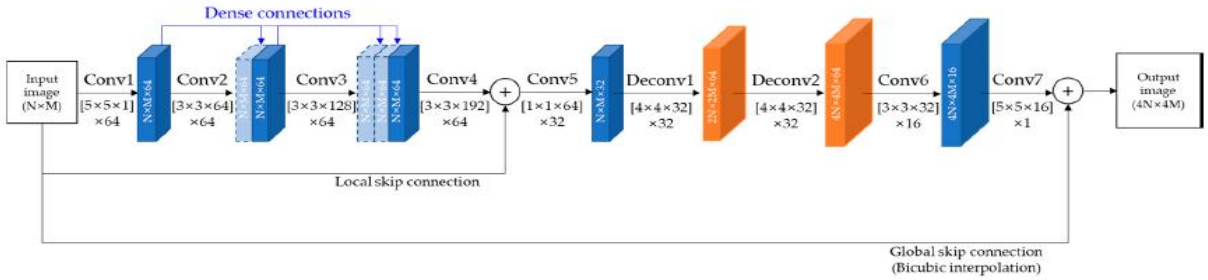


Figure 3.11 Architecture of SR-SLNN

The following parameters of the model implementation where the learning rate of 10^{-3} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, BSD100 and Urban100 datasets. The loss function used is mean squared error (MSE).

3.8 Super Lightweight Super-Resolution Network (SLWSR)

Super Lightweight Super-Resolution Network (SLWSR), [22] they built an SR model with symmetric architecture, possessing an assistant information pool. The skip connection mechanism combined with multi-level information from chosen layers, they built the information pool to transmit features to high-dimensional channels. This new information pool enforces better features transmission between the first and the second half of the model.

The most effective factor of model size is channel numbers, so they modify the model size by different setting of channel numbers. By introducing a novel compression module, the model size can be reduced by partly replacing normal residual blocks. They can control the total number of parameters within the ideal size by properly choosing the channel number and replacing specific layers with the new compression module.

They removed some activation operations to retain object details in their lightweight model. This minor modification improves the performance of their lightweight SR model.

The architecture of SLWSR is shown in Fig. 3.12. It consists of three sub-procedures: original feature extraction, detailed information learning, and SR image restoration. The more channels involved; the better performance achieved. So, they set the channel number of all residual blocks, as the primary factor of model size. A sequence of basic residual blocks consecutively connected, aiming at learning the feature map between Ilr and Isr.

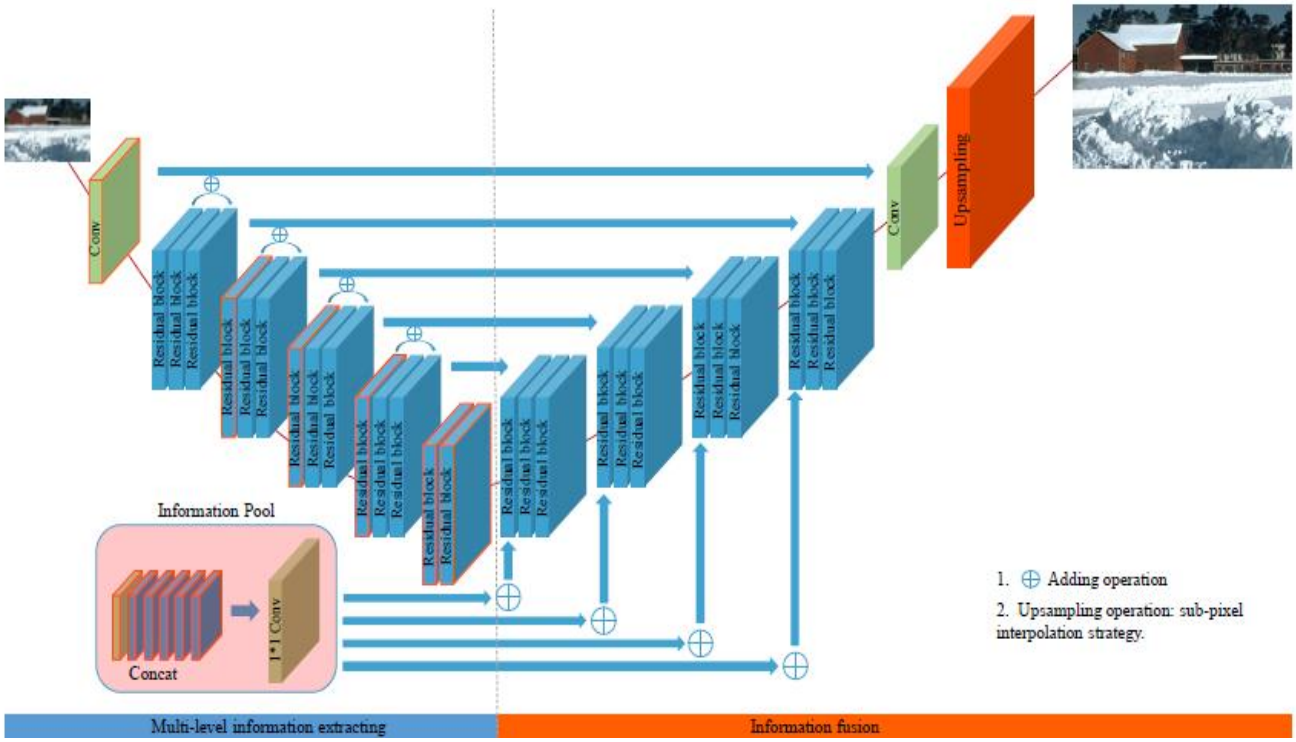


Figure 3.12 Architecture of SLWSR

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, B100 and Urban100 datasets. The loss function used is mean squared error (MSE).

3.9 Residual Feature Distillation Network for Lightweight Image Super-Resolution

Residual Feature Distillation Network for Lightweight Image Super-Resolution (RFDN) [23] , they proposed the feature distillation connection (FDC) that is functionally equivalent to the channel splitting operation while being more lightweight and flexible.

RFDN uses multiple feature distillation connections to learn more discriminative feature representations. They also proposed a shallow residual block (SRB) as the main building block of RFDN so that the network can benefit most from residual learning while still being lightweight enough.

They showed an information distillation network (IDN) that explicitly split the intermediate features into two parts along the channel dimension, one was retained, and the other was further processed by succeeding convolution layers. Later, information multi-distillation network (IMDN) further improved IDN by designing an information multi-distillation block (IMDB) that extracted features at a granular level.

They gave a more comprehensive analysis of the information distillation mechanism (IDM) and proposed the feature distillation connection (FDC) that is more lightweight and flexible than the IDM. They used IMDN as the baseline model since it makes a good trade-off between the reconstruction quality and the inference speed, which is very suitable for mobile devices. They rethinked the architecture of IMDN and proposed the residual feature distillation network (RFDN).

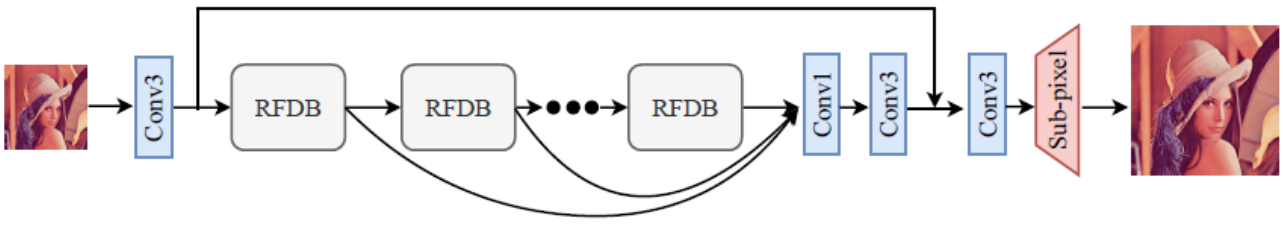


Figure 3.13 RFDN Architecture

The architecture of RFDN is shown in Fig. 3.13. RFDN contains multiple RFDB blocks. RFDB shown in fig. 3.14 contains shallow residual block (SRB) that is used as the building

blocks of RFDN to further improve the SR performance. The SRB consists of one convolutional layer, an identical connection, and an activation unit at the end. It can benefit from the residual learning without introducing extra parameters compared with plain convolutions.

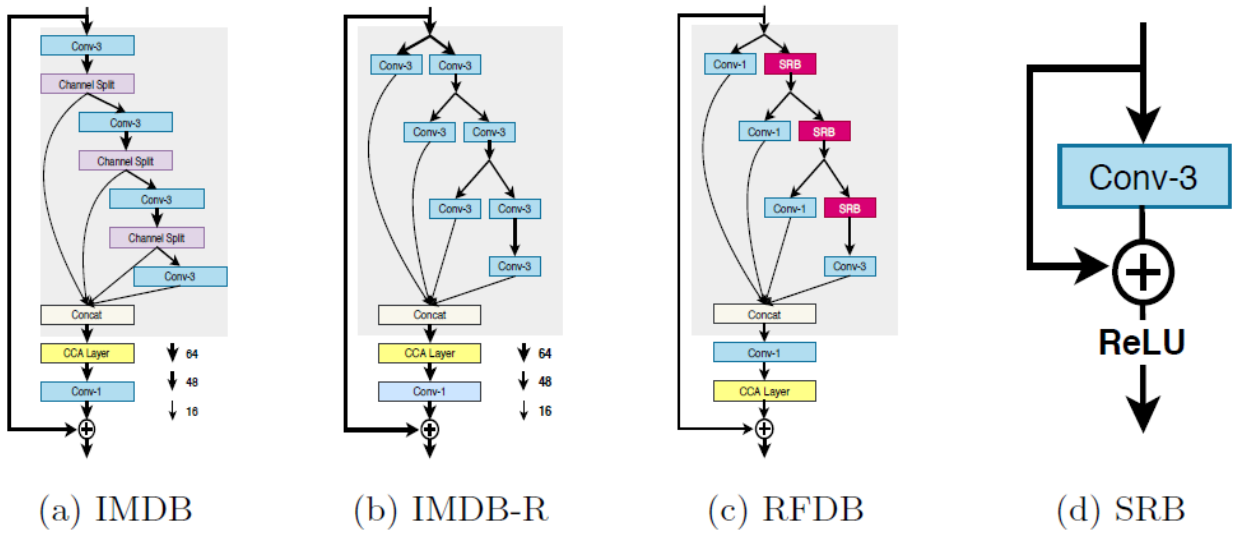


Figure 3.14 IMDB, RFDB, and SRB

The following parameters of the model implementation where the learning rate of $5*10^{-4}$ is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, B100 and Urban100 datasets. The loss function used is mean squared error (MSE).

3.10 Auto encoder for Super Resolution

Autoencoder was developed by adding different layers of the convolutional neural networks. Each layer of the neural network was used to process the input image into different stages and matrices to denoise the image. The model was then trained on a set of low-resolution images and their corresponding high-resolution images to extract the features of those images in a designed encoder. The loaded weights of the encoded images were stored, after the training, as feature vectors. An input test image, which is of low quality, is used to observe the results of the trained model. The model improves the quality of the input image by using the pre-trained autoencoder model. The features used in the model are highlighted below, which gave a detailed understanding of the working and architectural flow of the presented mode. Autoencoder used unsupervised learning techniques to denoise the image. It performed convolution and deconvolution procedures on the input image using the same input and output matrix.[24]

The architecture of autoencoder is shown in fig.3. 15. It consists of encoder and decoder.

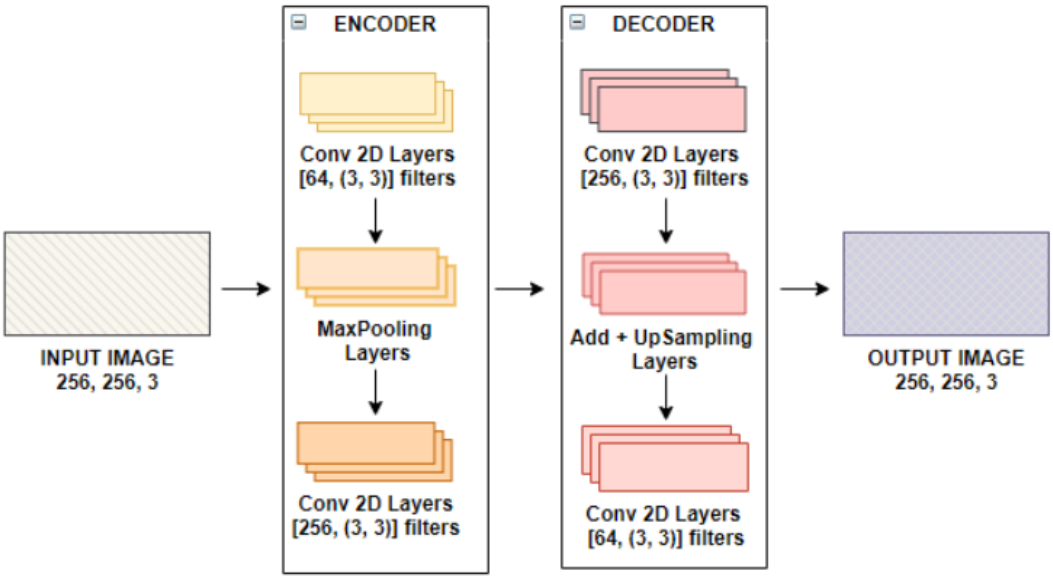


Figure 3.15 Autoencoder Model Structure

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is adadelta optimizer. The final network was trained on a dataset of 9544 images. The loss function used is mean squared error (MSE).

3.11 Super Resolution GAN

SRGAN is a generative adversarial network for single image super-resolution. It uses a perceptual loss function which consists of an adversarial loss and a content loss. The adversarial loss pushes the solution to the natural image manifold using a discriminator network that is trained to differentiate between the super-resolved images and original photo-realistic images. In addition, the authors used a content loss motivated by perceptual similarity instead of similarity in pixel space. Previous super resolution methods are mainly driven by the choice of the optimization function. Most used optimization target for the supervised SR algorithms is the minimization of the mean-squared error (MSE) and maximization of peak signal-to-noise (PSNR) which are defined based on the pixel-wise image differences. The PSNR ratios obtained from these methods is high, but the images are perceptually not satisfying. SRGAN which is a GAN-based network optimized for a new perceptual loss. Here MSE-based content loss is replaced with a loss calculated on feature maps of the VGG network, which are more invariant to changes in pixel space. Evaluation with an extensive mean opinion score (MOS) test on images from three public benchmark datasets to confirm

that SRGAN is the new state of the art, by a large margin, for the estimation of photo-realistic SR images with high upscaling factors (4×).

The architecture of SRGAN is shown in Fig. 3.16.The model consists of 2 main blocks, generator, and discriminator

In training, a low-resolution image (ILR) is obtained by applying a gaussian filter to a high resolution image (IHR) followed by a down-sampling operation with down-sampling factor r. Generator function G estimates for a given input LR image its corresponding HR image which is a super-resolved image SR. Discriminator D is trained to distinguish super resolved images and real images.

The following parameters of the model implementation where the learning rate of 10^{-4} is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, B100 and Urban100 datasets. The loss function used is mean squared error (MSE).

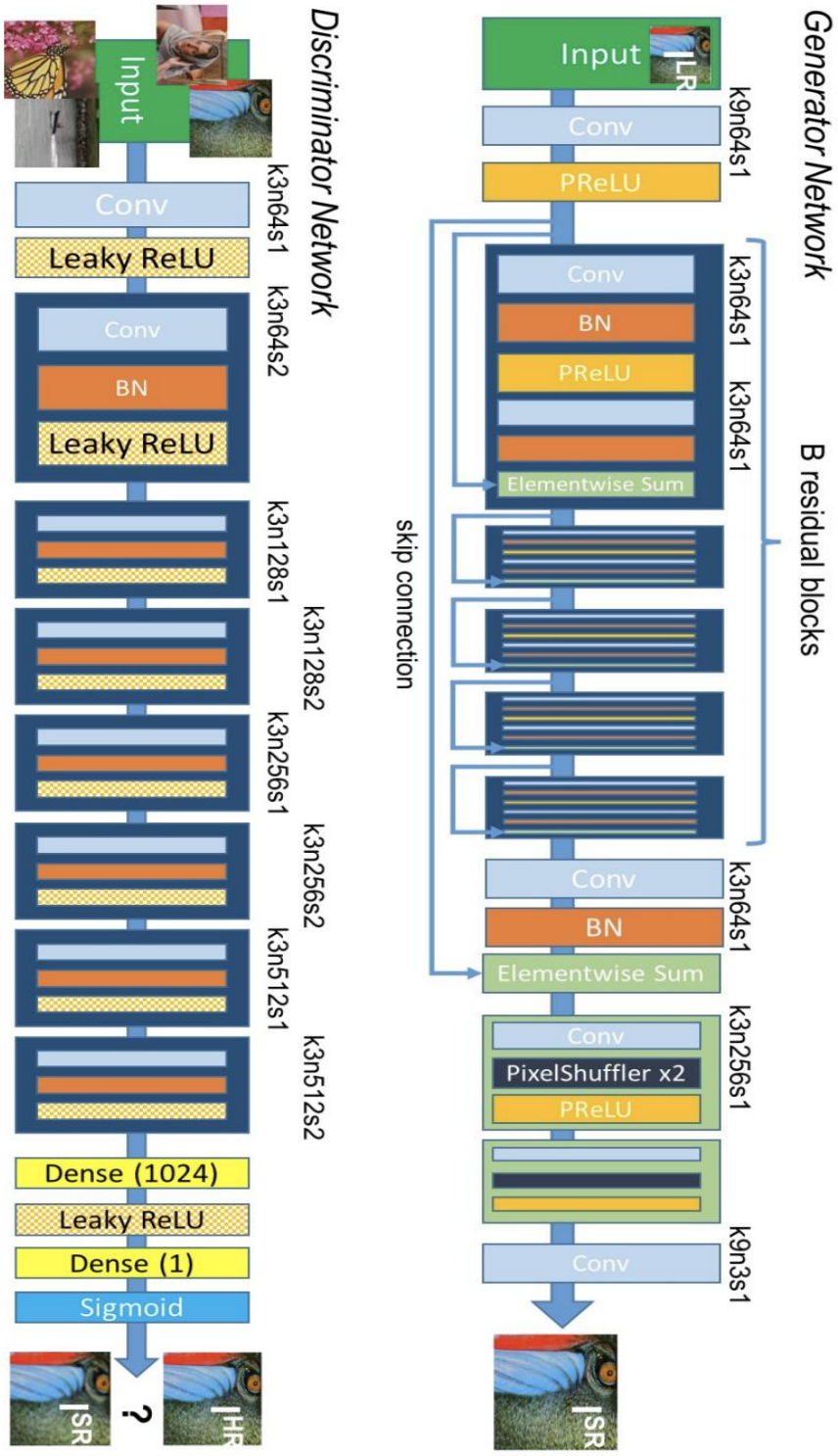


Figure 3.16 Architecture of Generator and Discriminator

3.12 Enhanced Super Resolution GAN

ESRGAN [25] is the enhanced version of the SRGAN, they revisited the key components of SRGAN and improve the model in three aspects. First, they improve the network structure by introducing the Residual-in-Residual Dense Block (RDDB), which is of higher capacity and easier to train. They also removed Batch Normalization (BN) layers as in and use residual scaling and smaller initialization to facilitate training a very deep network. Second, they improved the discriminator using Relativistic average GAN (RaGAN), which learns to judge whether one image is more realistic than the other" rather than whether one image is real or fake". There experiments show that this improvement helps the generator recover more realistic texture details. Third, they proposed an improved perceptual loss by using the VGG features before activation instead of after activation as in SRGAN. They empirically that the adjusted perceptual loss provides sharper edges and more visually pleasing results.

The main architecture of the ESRGAN is shown in fig. 3.17. It is the same as the SRGAN with some modifications. ESRGAN has Residual in Residual Dense Block (RRDB) shown in fig. 3.18 which combines multi-level residual network and dense connection without Batch Normalization. Enhanced Super-Resolution Generative Adversarial Network (ESRGAN) is a perceptual-driven approach for single image super-resolution that can produce photorealistic images.

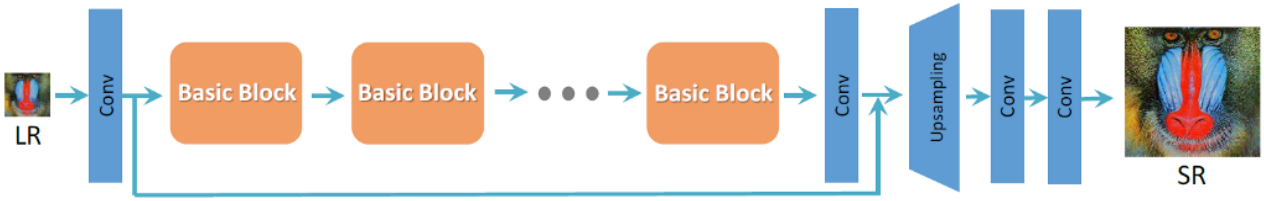


Figure 3.17 ESRGAN Architecture

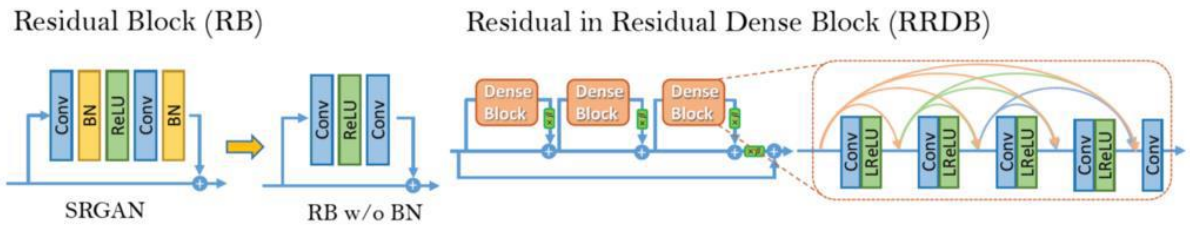


Figure 3.18 RRDB Architecture

Different from the standard discriminator D in SRGAN, which estimates the probability that one input image x is real and natural, a relativistic discriminator tries to predict the probability that a real image x_r is relatively more realistic than a fake one x_f.

Besides using standard discriminator of ESRGAN shown in fig. 3.19 they used the relativistic GAN, which tried to predict the probability that the real image is relatively more realistic than a fake image.

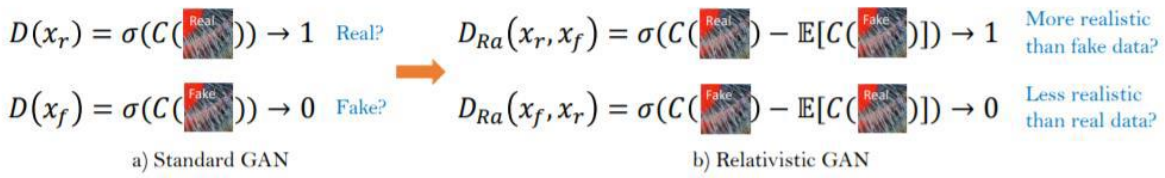


Figure 3.19 Relativistic Discriminator

The following parameters of the model implementation where the learning rate of s 2 $\times 10^{-4}$ is used for all layers. The optimizer used is Adam optimizer. The final network was trained on DIV2K dataset and tested on Set5, Set14, B100 and Urban100 datasets. The loss function used is mean squared error (MSE).

3.13 Similar Applications

This section discusses real world applications like Adobe Photoshop, Topaz Gigapixel AI, and Let's Enhance, that are similar to our system and that shows the importance of the super resolution when a well-known application like adobe added this feature to their Photoshop application in march 2021. This application is for comparative study with our implemented models and the results are viewed in chapter 5.

3.13.1 Adobe Photoshop

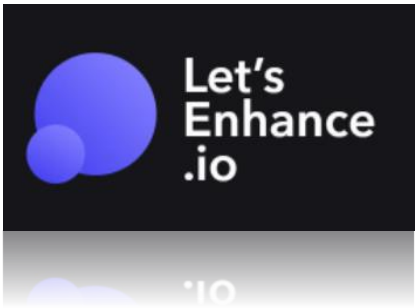
- Uses machine learning
- Adobe has been adding machine-learning and AI-powered features to Photoshop and Photoshop Lightroom. In March 2021.
- It only takes the images in 2 formats (JPG, TIFF)
- It has only x2 scale

3.13.2 Topaz Labs Gigapixel AI

- Use of machine learning/artificial intelligence (AI)
- Topaz Gigapixel AI is an advanced image enlarging software that uses artificial intelligence to upscale photos by up to 600%
- It has x2, x4, x6 and x8 scale

3.13.3 Let's Enhance

- Let's Enhance is a Ukrainian start-up which develops an online service driven by artificial intelligence.
- Which allows improving images and zooming them without losing quality.
- According to the developers, they used the super-resolution technology of machine learning.
- It has only x2 scale



Chapter 4

System Implementation

Chapter 4: System Implementation

This chapter cover system implementation, the system description, platform used, system training which includes the training dataset, 7 implemented models their flowcharts, models structure, algorithms, and parameters. Also the learning curves (training loss and accuracy) of each model. Finally, the system GUI is presented. Fig. 4.1 is the map of this chapter for clarification.

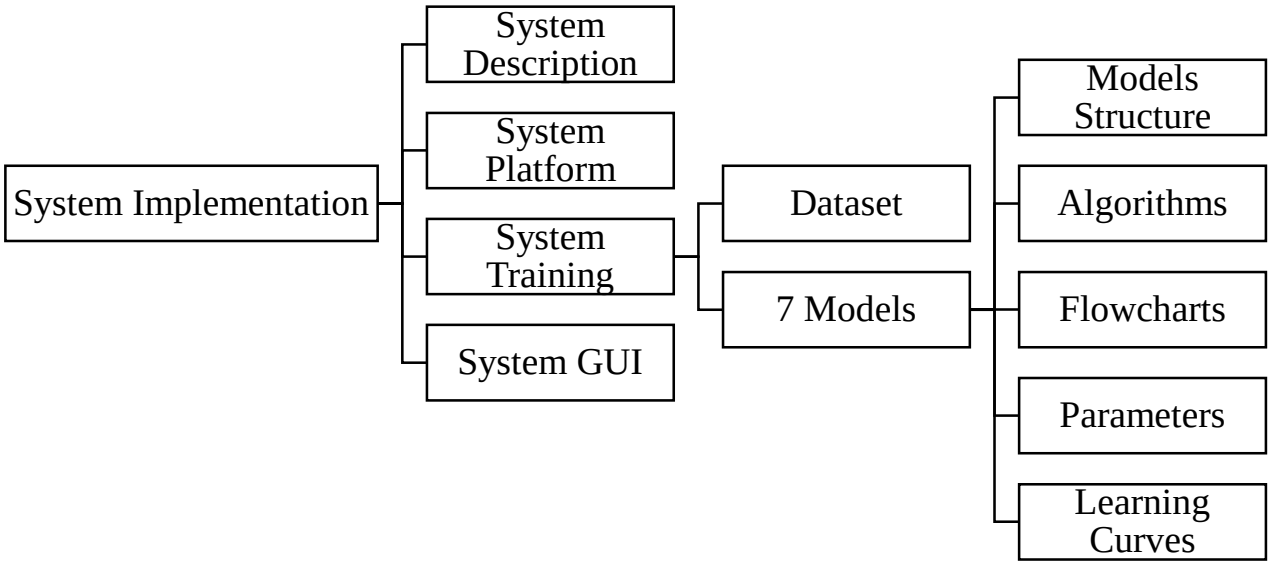


Figure 4.1 CH.4 System Implementation

4.1 System Description

In this project we aim to reach super resolution image, the process is to take an input low resolution image and then producing an output super resolution image that represents the high resolution image of the input.

Firstly, Dataset is inserted to the system. Then, cross validation is done where the model is trained using the subset of the data-set and then evaluate using the complementary subset of the data-set. Then, preprocessing is applied on the input images such as resizing. After that the training process is made where setting the parameters for tuning the model such as number of epochs, batch size, learning rate, etc., and extracting features, mapping, and reconstruction. Then validating the model to make sure that no over fitting or high loss function detected in the training session by the learning curves. Last but not least, saving the weights of the model

after training so it can be used by the model again. Finally, the model is ready for testing. Fig. 4.2 shows the system description flow for clarification.

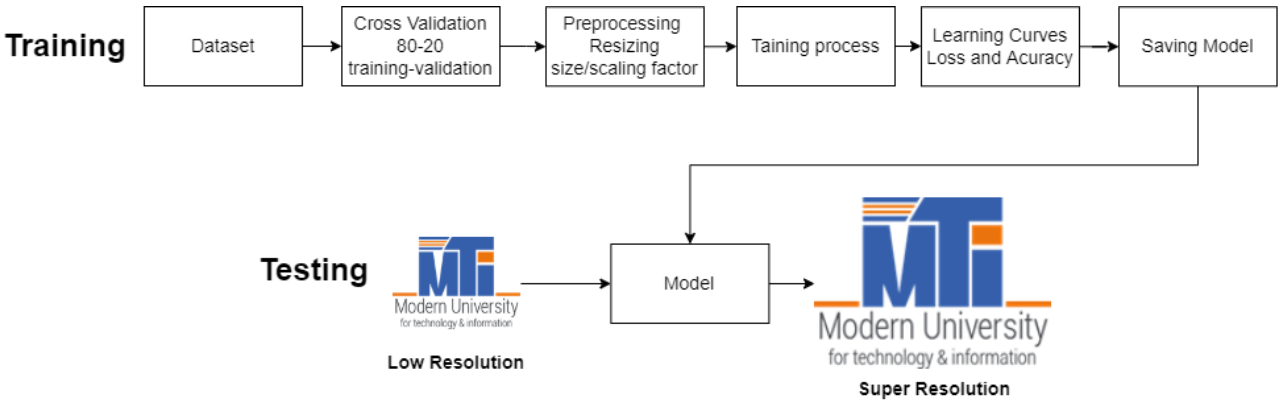


Figure 4.2 System Description

4.2 System Platform

This section includes system platforms, API's, frameworks, and environment used to implement the models. The platforms used for implementation is Python (Google Colab & Saturn cloud), the environment is Anaconda3, APIs and frameworks is TensorFlow and Keras. They are further discussed in the following subsections.

4.2.1 Python

Python is considered as a fast, efficient and reliable language, and it can work across many domains such as web development, mobile applications, Python has plotting libraries which produces error charts, and more charts to trace changes in the training. Version (3.9. 4) is used in models implementation. It has powerful APIs and frame works that helped in our implementations. Common frameworks for the deep learning tasks, and used for model implementation are discussed in the following section.

4.2.2 TensorFlow

TensorFlow is an open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state of art in machine learning. TensorFlow can train and run deep neural networks for image enhancement, natural language processing, and classification. Version (2.9.0) is used in models implementation.

4.2.3 Keras

Keras is a high-level neural networks API, written in python and capable of running on top of TensorFlow. It was developed with a focus on enabling fast experimentation. Keras allows easy and fast prototyping “through user friendliness, modularity and extensibility”. It supports both convolutional neural networks and recurrent networks, and it runs seamlessly on CPU and GPU. So, we used Keras in most of our CNN implementation.

4.2.4 Anaconda

Anaconda is an open-source distribution of the Python and R programming languages for data science that aims to simplify package management and deployment (set environment). It is used for data science, machine learning, deep learning, etc. With the availability of more than 300 libraries for data science, it becomes fairly optimal for any programmer to work on anaconda for data science. Anaconda helps in simplified package management and deployment. Version (2.1. 1) is used in models implementation.

4.2.5 Cloud Servers

Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user. Large clouds often have functions distributed over multiple locations, each location being a data center.
There are different cloud servers like Microsoft Azure, Saturn, Google Colab and Amazon AWS.
Saturn Cloud Server is the used server in models implementation.
Saturn Cloud natively supports Python and R, with support for additional languages available. It has a built-in JupyterLab or RStudio online, or you connect from preferred IDE like VSCode or PyCharm with SSH. It also has a built-in functionality for git repos so you can easily manage your code.

4.3 System Training

This section covers the system training, where the used dataset for training is introduced, and the 7 implemented models training process is covered.

4.3.1 Training Dataset

The dataset used is DIV2K [45] which is a popular single-image super-resolution dataset which contains 900 high quality RGB images with different scenes and sizes with PNG

extension. It was collected for NTIRE2017 and NTIRE2018 Super-Resolution Challenges to encourage research on image super-resolution with more realistic degradation.

This dataset contains low resolution images with different types of degradations. Apart from the standard bicubic down sampling, several types of degradations are considered in synthesizing low resolution images for different tracks of the challenges.

Track 1 of NTIRE 2017 contains low resolution images with unknown x4 downscaling. Track 2 and track 4 of NTIRE 2018 correspond to realistic mild ×4 and realistic wild ×4 adverse conditions, respectively.

Low-resolution images under realistic mild x4 setting suffer from motion blur, Poisson noise and pixel shifting. Degradations under realistic wild x4 setting are further extended to be of different levels from image to image.

4.3.2 Models Description

This section discusses briefly the training process of the 7 implemented models, where flowcharts, model structure, and algorithm is presented for clarification.

a. Training process:

The 7 models are trained on DIV2K dataset described before with 80-20% training and validation. They are trained in different scales (x2, x3, x4, or x8). Firstly, the models were trained on local personal computers, it was slow due to the computer specifications. So as mentioned before in section 4.2.4 cloud servers are much faster, the models were trained on Saturn Cloud Server having the following specifications; Size: 8 cores – 64 GB RAM, 40Gi Disk Space – CPU Hardware.

b. Explanation of the implemented models:

The implemented models were mentioned before in ch.3 (the 7 highlighted models). The models architecture and summary were viewed, but not clearly discussed. Firstly, description of parameters is introduced. Then the explanation of the models and the models details is presented.

Parameters Definitions

- **Training set:** is the data that the algorithm will learn from.
- **Validation set:** to estimate how well your model has been trained (that is dependent upon the size of your data, the value you would like to predict, input etc.)
- **Epoch:** is one complete presentation of the data set to be learned to a learning machine
- **Batch-size:** is the number of training examples utilized in one iteration.

- **Optimizer:** used to minimize or maximize of the loss function.
- **Loss Function:** used to measure the inconsistency between predicted value (^y) and actual label (y).
- **LR:** learning rate used for training. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Models Details

1. **SRCNN**

Super Resolution using Deep Convolution Neural Network (SRCNN) has been implemented using 2 different scales (2x, and 4x). As mentioned before it is trained on DIV2K dataset and 80-20 cross validation. With total number of parameters (sum of all the weight and bias on the neural network) of 85,889, and the following table 4.1 contains the hyper parameters of the model.

Table 4.1 SRCNN Parameters

Parameters	**SRCNN**
Optimizer	ADAM
Learning Rate	$3*10^{-3}$
Activation Function	Relu
Loss Function	MSE
Number of Epochs	60
Batch Size	10

SRCNN Model Structure

SRCNN model structure fig. 4.3 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

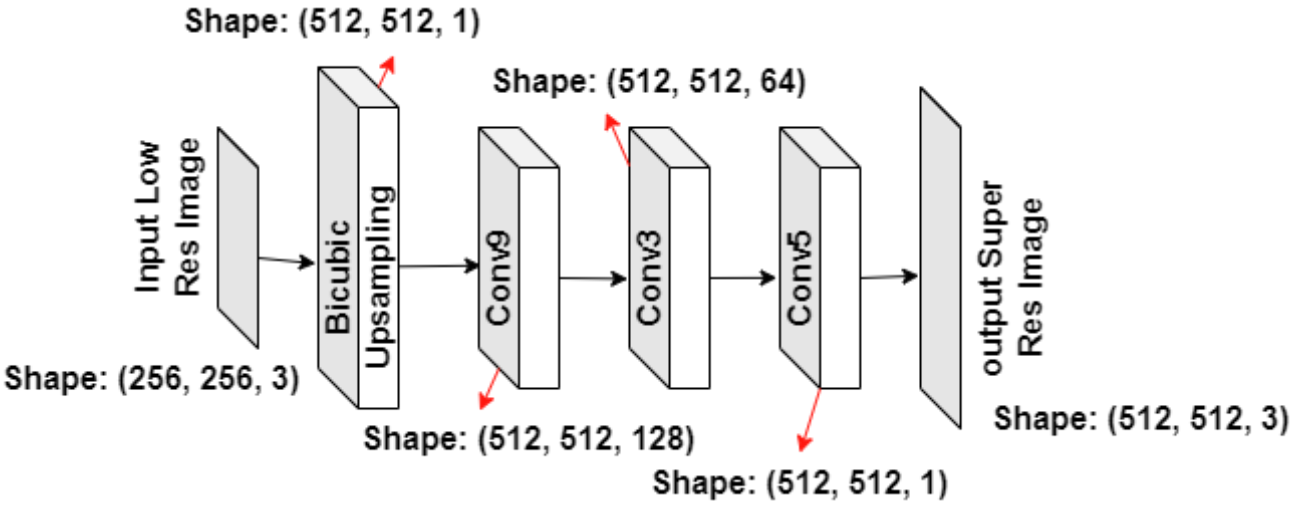


Figure 4.3 SRCNN Model Structure

SRCNN Model Algorithm

1. The input image (low resolution image) with any size but for example (256, 256,3)
2. Bicubic upsampling is applied to the low resolution image with shape (512, 512, 1)
3. Followed by convolution layer of 9x9 kernel with shape (512, 512, 128).
4. The second layer applies a convolution layer of 3x3 kernel with shape (512, 512, 64).
5. The third layer applies a 5x5 kernel to generate the output image with shape (512, 512, 1).
6. The output image (Super Resolution Image) with shape (512, 512, 3).

SRCNN Experiment details

The model takes the low and high resolution data from the dataset, then resize the images with the desired size and convert bgr to YCrCb and get the first channel Y. Then take Y channels float and divide by 255. Last but not least changing images to tensors and start training the model depending on the model structure that was described in fig. 4.3 and the algorithm before.

The following fig. 4.4 is a flowchart diagram for clarification of the whole implementation of the model.

SRCNN Flowchart

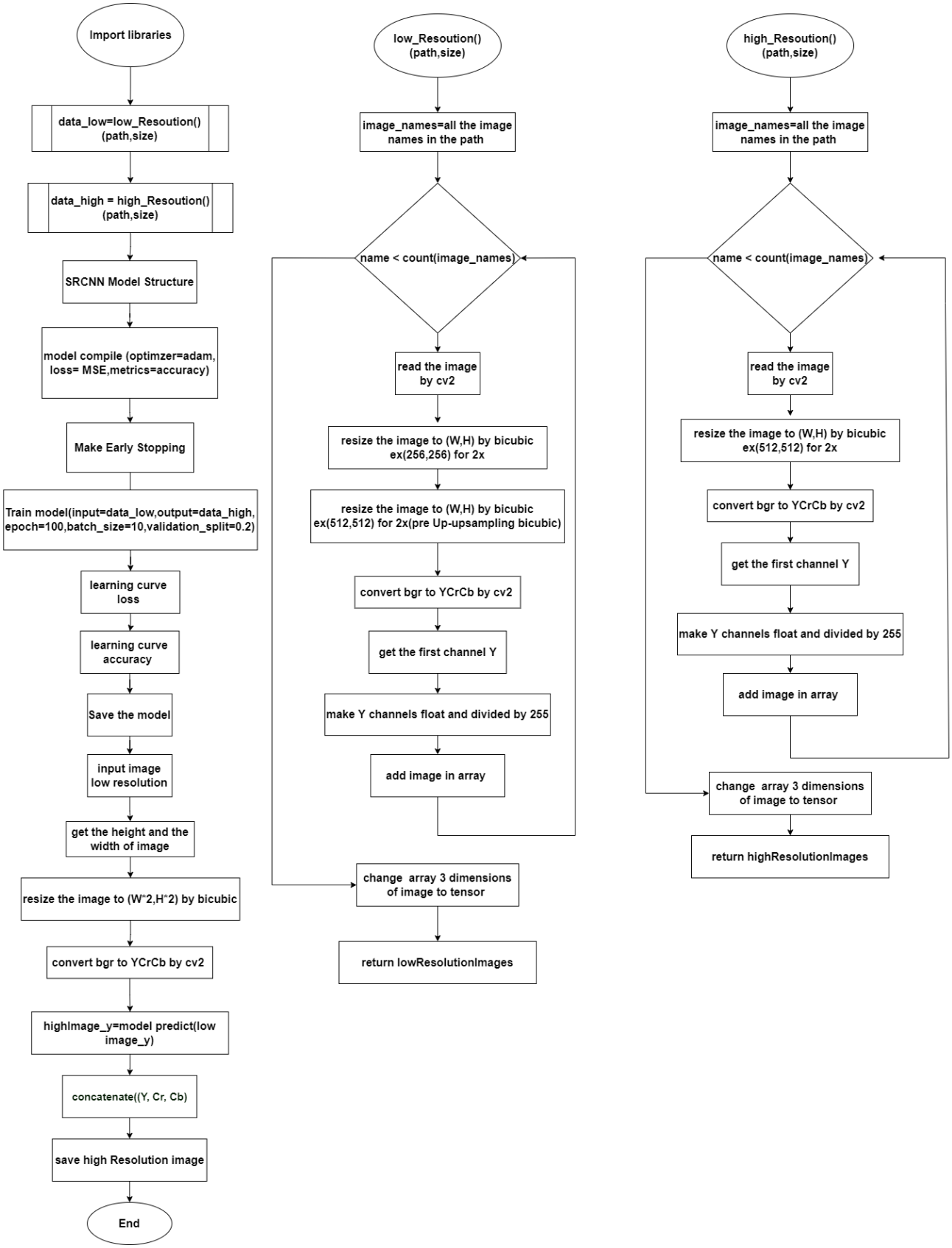


Figure 4.4 SRCNN Flowchart

SRCNN Learning Curves

Learning curves are graphs that compares and show the performance of a model on training and validation data over number of epochs. It shows the training and validation loss in fig. 4.5, and the training and validation accuracy in fig. 4.6 of SRCNN model.

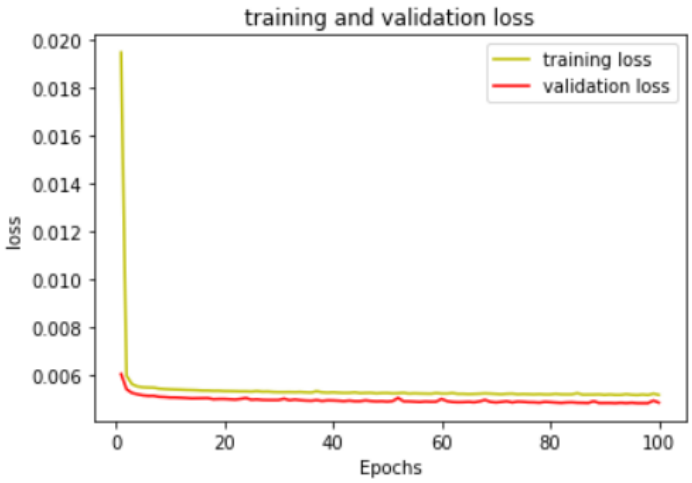


Figure 4.5 SRCNN Training and Validation Loss Learning Curve

As seen in fig. 4.5, and fig. 4.6 the training and validation loss is decreasing and the training and validation accuracy is increasing the training and validation loss is decreasing with the increasing of the number of epochs, that shows that the model is well trained and no overfit. The model was trained on 100 epochs as mentioned before, we could have increased the number of epochs since there is a possibility that the loss may still decrease, but early stopping (mentioned in ch.2) should be put into consideration to prevent the model from overfitting.

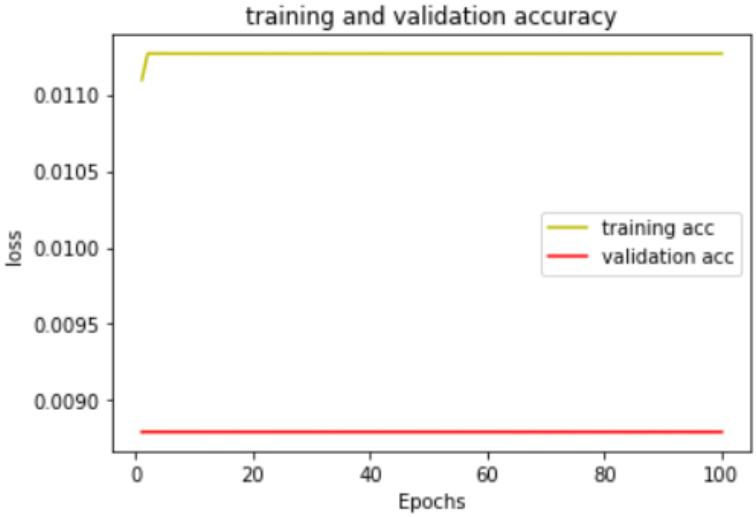


Figure 4.6 SRCNN Training and Validation Accuracy Learning Curve

2. FSRCNN

Fast Super Resolution using Deep Convolution Neural Network (FSRCNN) has been implemented using 3 different scales (2x, 4x, 8x). As mentioned before it is trained on DIV2K dataset and 80-20 cross validation. With total number of parameters for 2x scale is 13,763, and for 4x scale is 21,827. The following table 4.2 contains the hyper parameters of the model.

Table 4.2 FSRCNN Parameters

Parameters	FSRCNN
Optimizer	ADAM
Learning Rate	10^{-3}
Activation Function	PRelu
Loss Function	MSE
Number of Epochs	100
Batch Size	10

FSRCNN Model Structure

FSRCNN model structure fig. 4.7 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

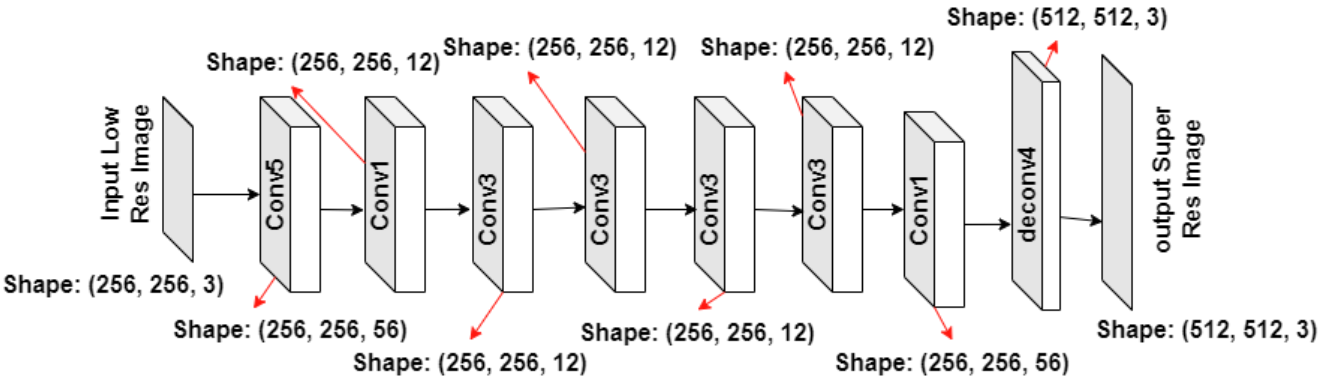


Figure 4.7 FSRCNN Model Structure

FSRCNN Model Algorithm

1. Firstly, we input low resolution image with any desired shape, ex. (256,256,3) pass through a convolution layer 5x5 with shape (256,256,56) for feature extraction.

2. Then passes through convolution layer 1x1 with shape (256,256,12) for shrinking after feature extraction.
3. Adding four layers of convolutional 3x3 with output shape of (256, 256, 12) for mapping.
4. The expanding layer acts like an inverse process of the shrinking layer, then we apply convolution layer 1x1 with shape (256,256,56) for expanding.
5. The last part is a deconvolution layer, which upsamples and aggregates the previous features with a set of deconvolution filters, where The final layer of the model is applied to the convolutional layer 4x4 with the output shape of (512, 512, 3) to obtain the final result of FSRCNN.
6. Output image (Super Resolution Image) with shape (512, 512, 3)

FSRCNN Experiment details

The model takes the low and high resolution data from the dataset, then resize the images with the desired size and convert bgr to rgb and get the first channel Y. Then take Y channels float and divide by 255. Last but not least changing images to tensors and start training the model depending on the model structure that was described in fig. 4.7 and the algorithm before.
The following fig. 4.8 is a flowchart diagram for clarification of the whole implementation of the model.

FSRCNN Flowchart

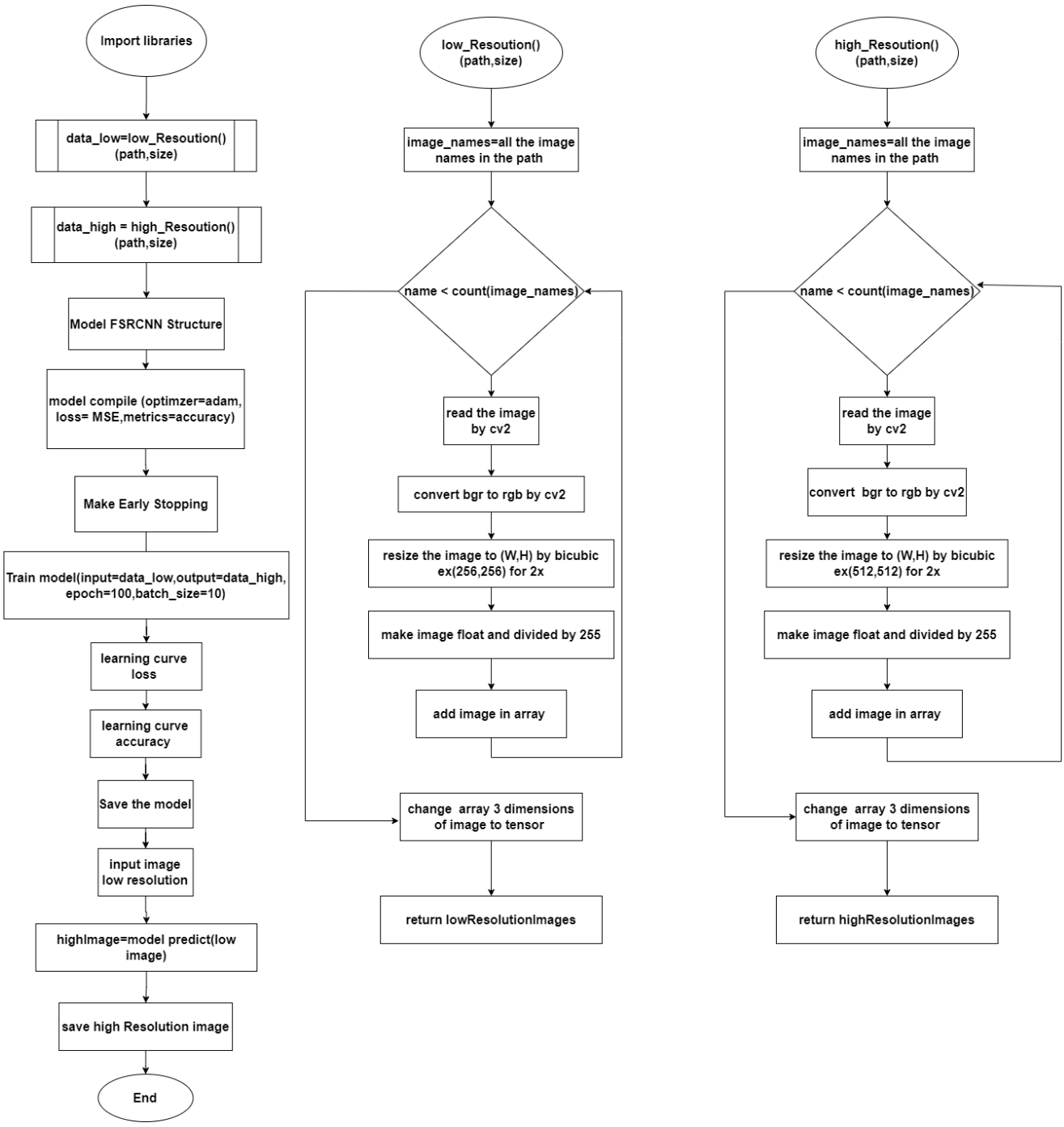


Figure 4.8 FSRCNN Flowchart

FSRCNN Learning Curves

The training and validation loss is shown in fig. 4.9, and the training and validation accuracy in fig. 4.10 of FSRCNN model.

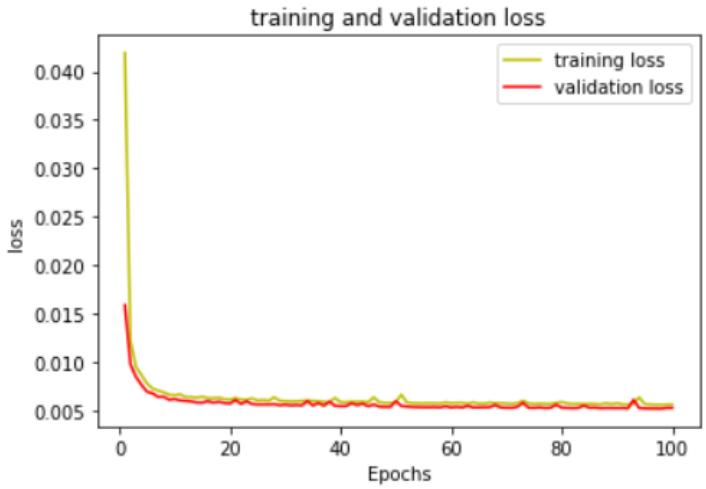


Figure 4.9 FSRCNN Training ad Validation Loss Learning Curve

As seen in fig. 4.9, and fig. 4.10 the training and validation loss is decreasing and the training and validation accuracy is increasing the training and validation loss is decreasing with the increasing of the number of epochs, that shows that the model is well trained and no overfit. The model was trained on 100 epochs as mentioned before, we could have increased the number of epochs since there is a possibility that the loss may still decrease, but early stopping (mentioned in ch.2) should be put into consideration to prevent the model from overfitting.

Figure 4.10 FSRCNN Training and Validation Accuracy Learning Curve

3. ESPCN

Efficient Subpixel Convolution Network (ESPCN) has been implemented using 2 different scales (2x, 4x). As mentioned before it is trained on DIV2K dataset and 80-20 cross validation. With total number of parameters for 2x scale is 75,552, and for 4x scale is 61,680. The following table 4.3 contains the hyper parameters of the model.

Table 4.3 ESPCN Parameters

Parameters	ESPCN
Optimizer	ADAM
Learning Rate	10^{-3}
Activation Function	Relu
Loss Function	MSE
Number of Epochs	100
Batch Size	15

ESPCN Model Structure

ESPCN model structure fig. 4.11 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

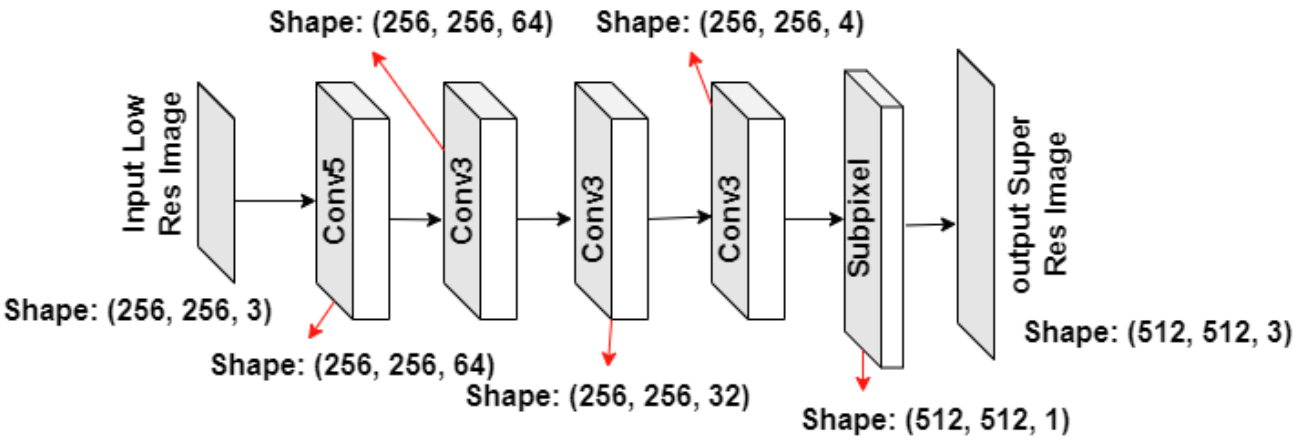


Figure 4.11 ESPCN Model Structure

ESPCN Algorithm

1. The input image with the desired size but for example shape (256, 256,3)
2. First layer: convolutional layer with 64 filters and the kernel size of 5 × 5, with Shape (256, 256, 64).

3. Second layer: convolutional layer with 64 filters and the kernel size of 3 × 3, with shape (256, 256, 64).
4. Third layer: convolutional layer with 32 filters and the kernel size of 3 × 3, with shape (256, 256, 32).
5. Fourth Layer: convolutional layer with 4 filters and the kernel size of 3 × 3, with shape (256, 256, 4).
6. Apply the sub-pixel shuffle function so that the output SR image will have the shape (512, 512, 3).

ESPCN Experiment details

The model takes the images from the dataset, and cross validate 80-20% then resize the images with the desired size and make the training data, then takes the images and convert it to high resolution and low resolution. Low resolution by dividing the image on the scale factor, and high resolution by converting RGB to YUV and taking the first channel Y. Last but not least changing images to tensors and start training the model depending on the model structure that was described in fig. 4.11 and the algorithm before.

The following fig. 4.12 is a flowchart diagram for clarification of the whole implementation of the model.

ESPCN Flowchart

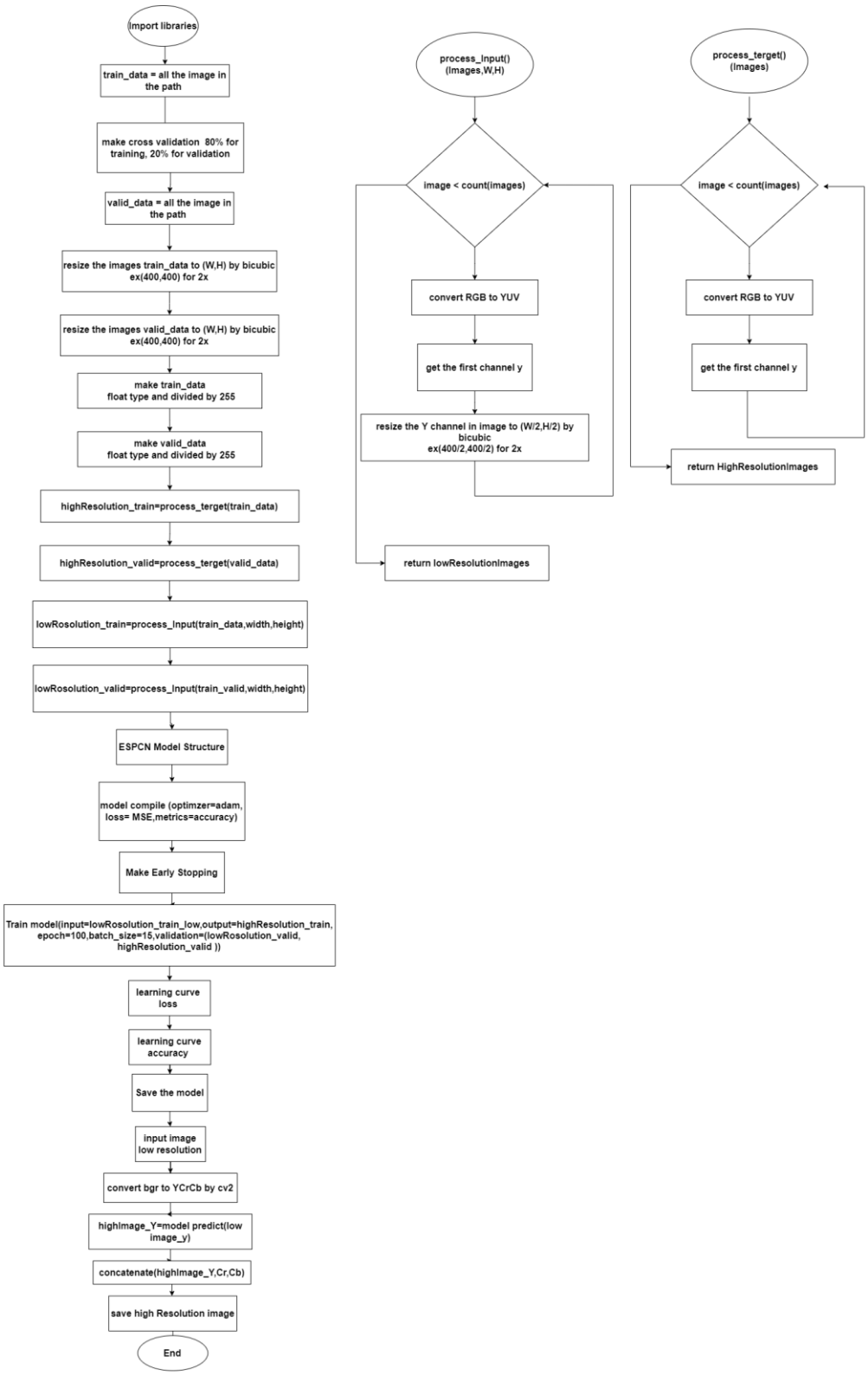


Figure 4.12 ESPCN Flowchart

ESPCN Learning Curves

The training and validation loss is shown in fig. 4.13, and the training and validation accuracy in fig. 4.14 of ESPCN model.

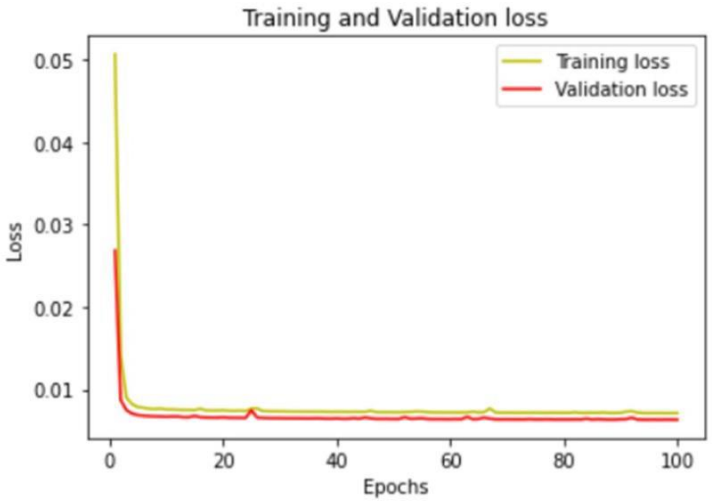


Figure 4.13 ESPCN Training and Validation Loss Learning curve

As seen in fig. 4.13, and fig. 4.14 the training and validation loss is decreasing and the training and validation accuracy is increasing the training and validation loss is decreasing with the increasing of the number of epochs, that shows that the model is well trained and no overfit. The model was trained on 100 epochs as mentioned before, we could have increased the number of epochs since there is a possibility that the loss may still decrease, but early stopping (mentioned in ch.2) should be put into consideration to prevent the model from overfitting.

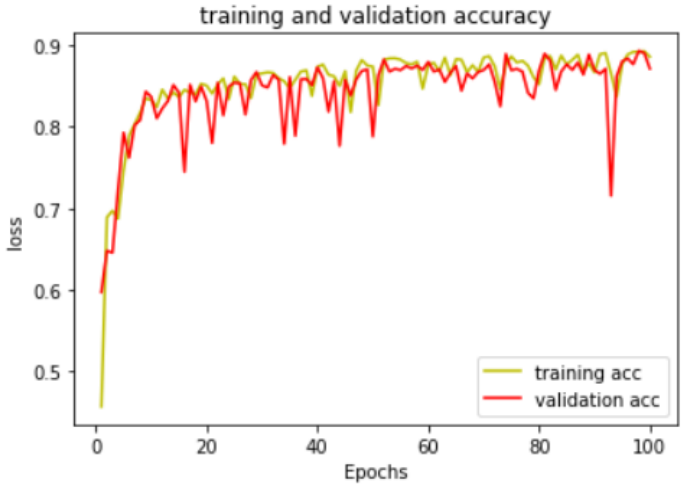


Figure 4.14 ESPCN Training and Validation Accuracy

4. RDN

Residual Dense Network (RDN) has been implemented using 2 different scales (2x, 4x). As mentioned before it is trained on DIV2K dataset and 80-20 cross validation. With total number of parameters for 2x scale is 16,347,392, and for 4x scale is 16,357,796. The following table 4.4 contains the hyper parameters of the model.

Table 4.4 RDN Parameters

Parameters	RDN
Optimizer	ADAM
Learning Rate	0.02
Activation Function	Relu
Loss Function	MSE
Number of Epochs	70
Batch Size	16

RDN Model Structure

RDN model structure fig. 4.16 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

RDN model as described before in chapter 3, have a main building block which is RDB (Residual Dense Block) fig. 4.15

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

RDB Structure

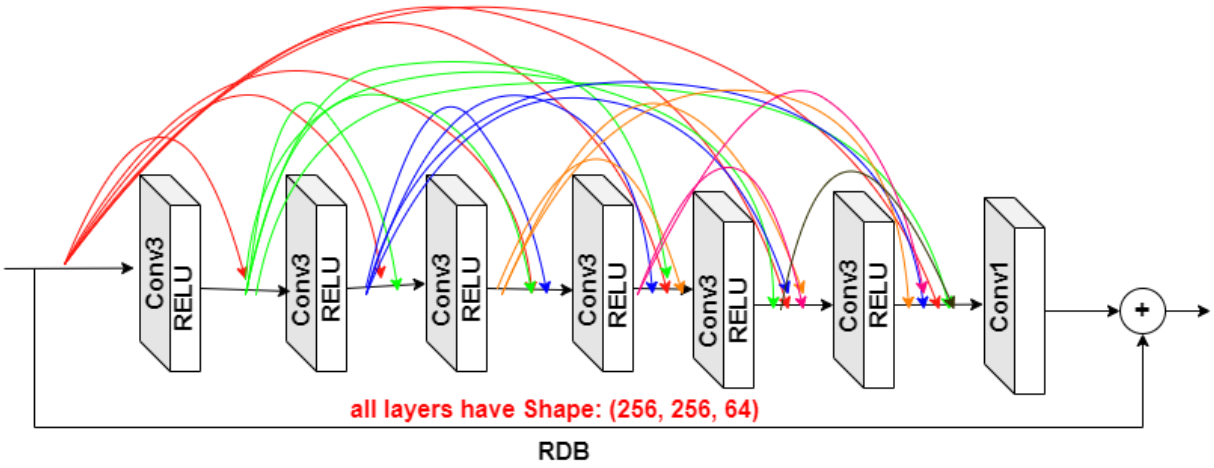


Figure 4.15 RDB Structure

RDN Structure

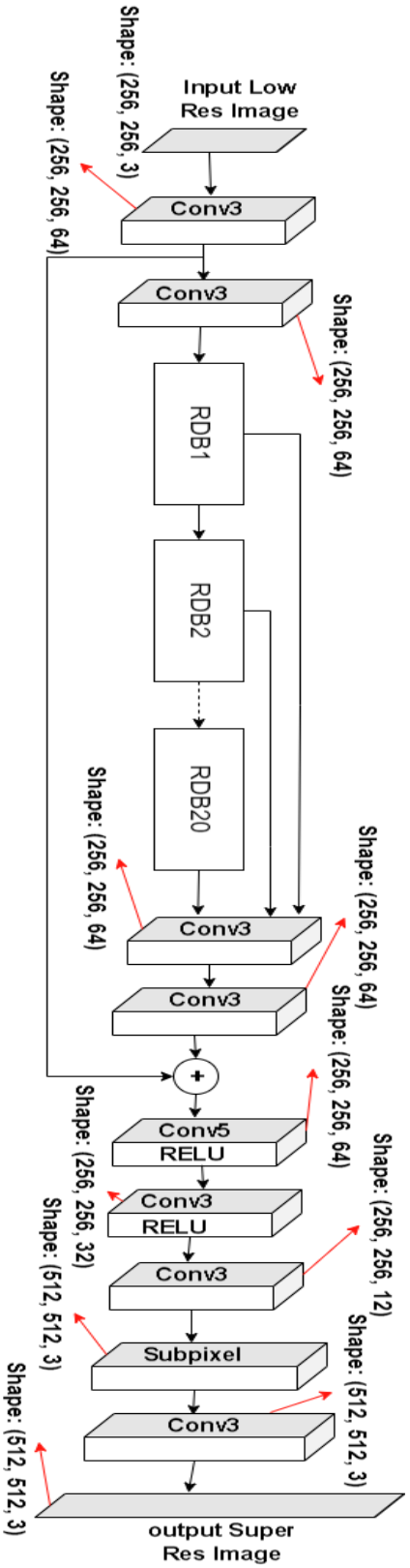


Figure 4.16 RDN Model Structure

RDN Algorithm

1. First we input low resolution image with any desired shape, ex. (256, 256, 3).
2. We use two Convolution layers of kernel 3x3 with shape (256, 256, 64) to extract features.
   - The first Conv layer extracts features from the LR input.
   -Second Conv layer extracts features and it used as input to residual dense blocks (RDB).
3. Then RDBs composite function of operations, convolution of kernel 3x3 and the last convolution layer is 1x1 and rectified linear units (ReLU), with shapes (256, 256, 64) and it's contain dense connected layers. As shown in figure 95.
4. Then passing the state of preceding RDB to each layer of current RDB.
5. After extracting hierarchical features with 20 RDBs, pass them through two Convolution layers of kernel 3x3 with shape (256, 256, 64).
6. Then concatenation of the feature-maps from previous blocks.
7. Pass first convolution layer and concatenation to a convolution layer of kernel 5x5 and RELU with shape (256, 256, 64).
8. Then pass it to a convolution layer of kernel 3x3 and RELU with shape (256, 256, 32).
9. After that pass it to a convolution layer of kernel 3x3 with shape (256, 256, 12) before upsampling.
10. Then subpixel Layer to upsample the image with output (super resolution) shape (512, 512, 3).

RDN Experiment details

The model takes the high resolution and low resolution image for training and validation,then take the tfrecord path, and make a tfrecord for training and validation. Where it encode and read the images in binary and store them in a tfrecord file. That method helps in the very deep models which require a large number of parameters as it takes less space in memory, so kernel dead is avoided. Then read those tfrecords file and start training the model depending on the model structure that was described in fig. 4.16 and the algorithm before.

The following fig. 4.17 is a flowchart diagram for clarification of the whole implementation of the model.

RDN Flowchart

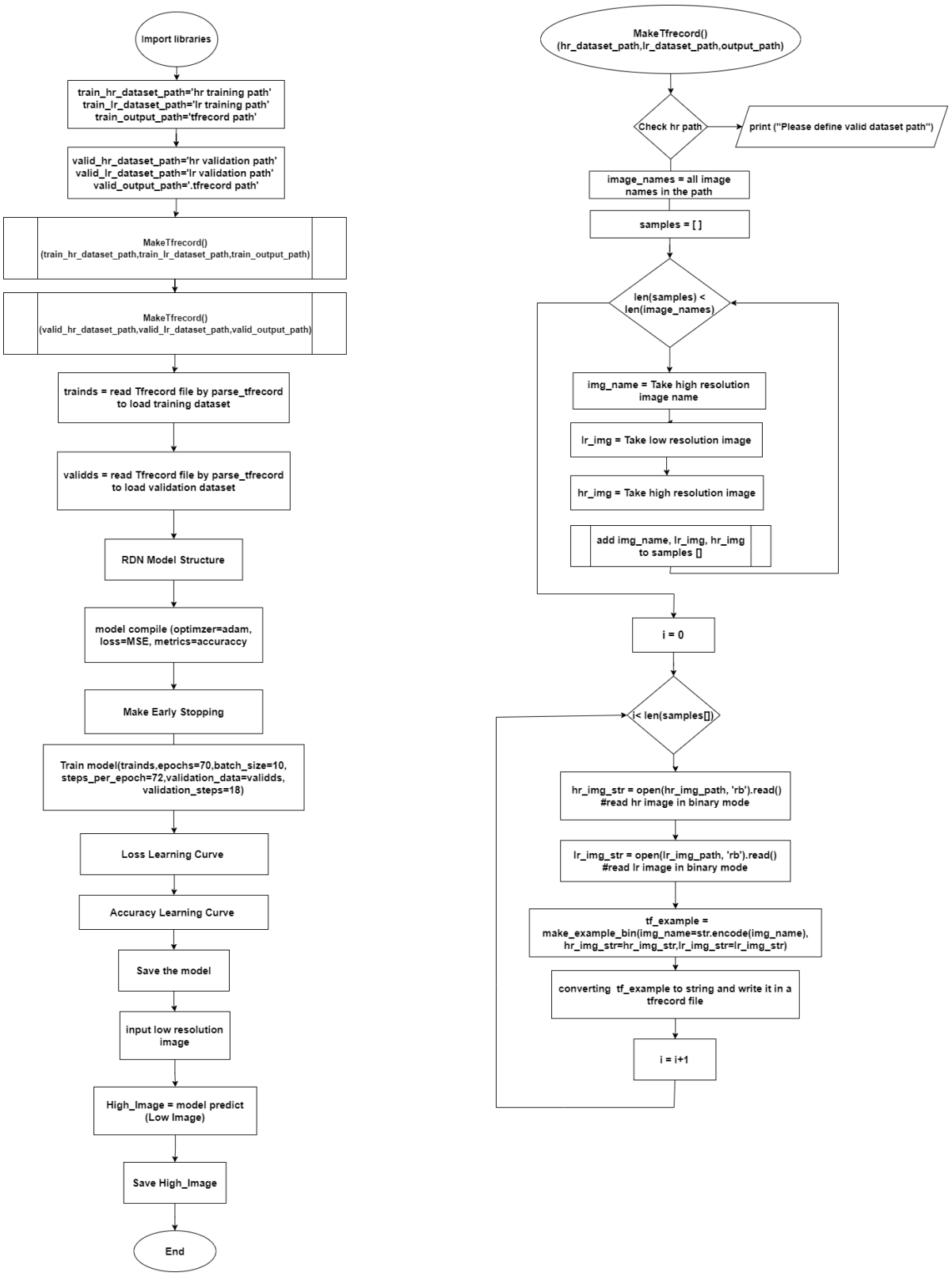


Figure 4.17 RDN Flowchart

RDN Learning Curves

The training and validation loss is shown in fig. 4.18, and the training and validation accuracy in fig. 4.19 of RDN model.

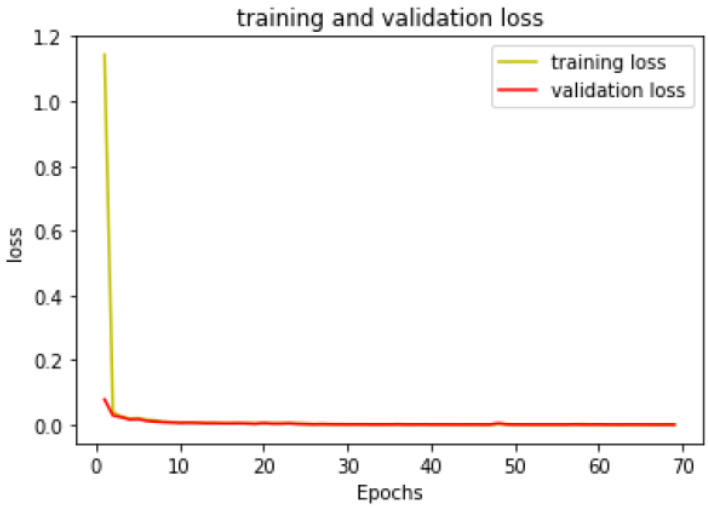


Figure 4.18 RDN Training and Validation Loss Learning Curve

As seen in fig. 4.18, and fig. 4.19 the training and validation loss is decreasing and the training and validation accuracy is increasing with the increasing of the number of epochs, that shows that the model is well trained and no overfit. The model was trained on 100 epochs as mentioned before, but it stopped in epoch 70, because overfitting could've appeared but the early stopping prevented the model from this.

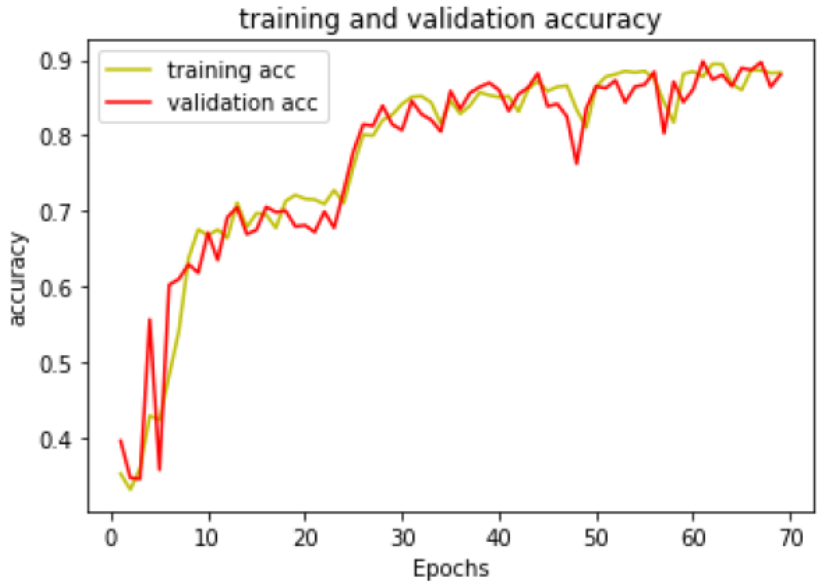


Figure 4.19 RDN Training and Validation Accuracy Learning Curve

5. RFDN

Residual Feature Distillation Network for Lightweight Image Super-Resolution (RFDN) has been implemented using 4 different scales (2x, 3x, 4x, 8x). As mentioned before it is trained on DIV2K dataset and 80-20 cross validation. With total number of parameters for 2x scale is 778,051, for 3x scale is 962,691, for 4x scale is 1,221,187, and for 8x scale is 2,250,365. The following table 4.5 contains the hyper parameters of the model.

Table 4.5 RFDN Parameters

Parameters	RDN
Optimizer	ADAM
Learning Rate	10^{-3}
Activation Function	Relu
Loss Function	MSE
Number of Epochs	100
Batch Size	15

RFDN Model Structure

RDN model structure fig. 4.20 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

RFDN model as described before in chapter 3, have a main building block which is RFDB (Residual Feature Distillation Block) fig. 4.21

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

RFDN Structure

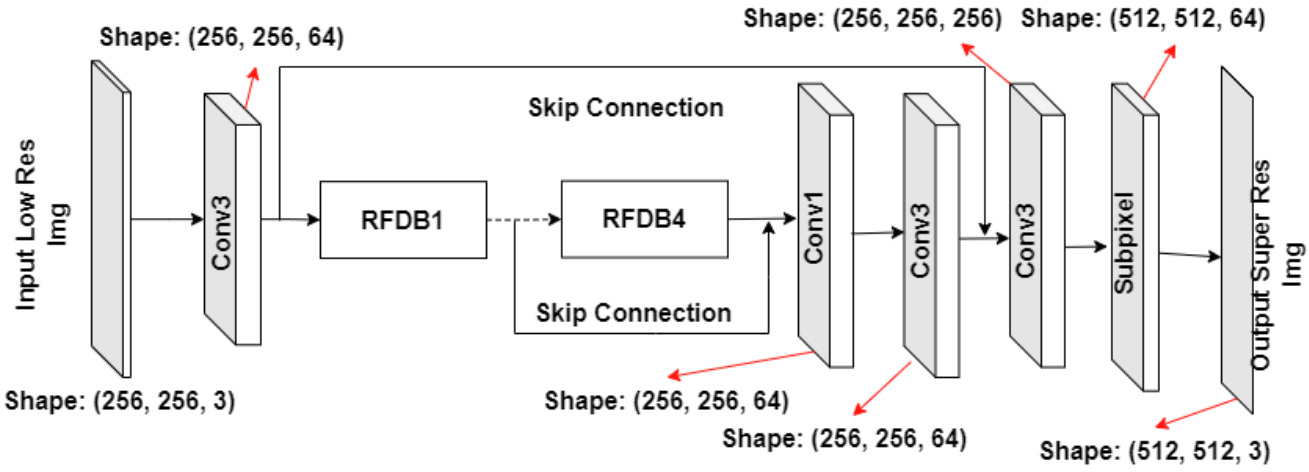


Figure 4.20 RFDN Model Structure

RFDB Structure

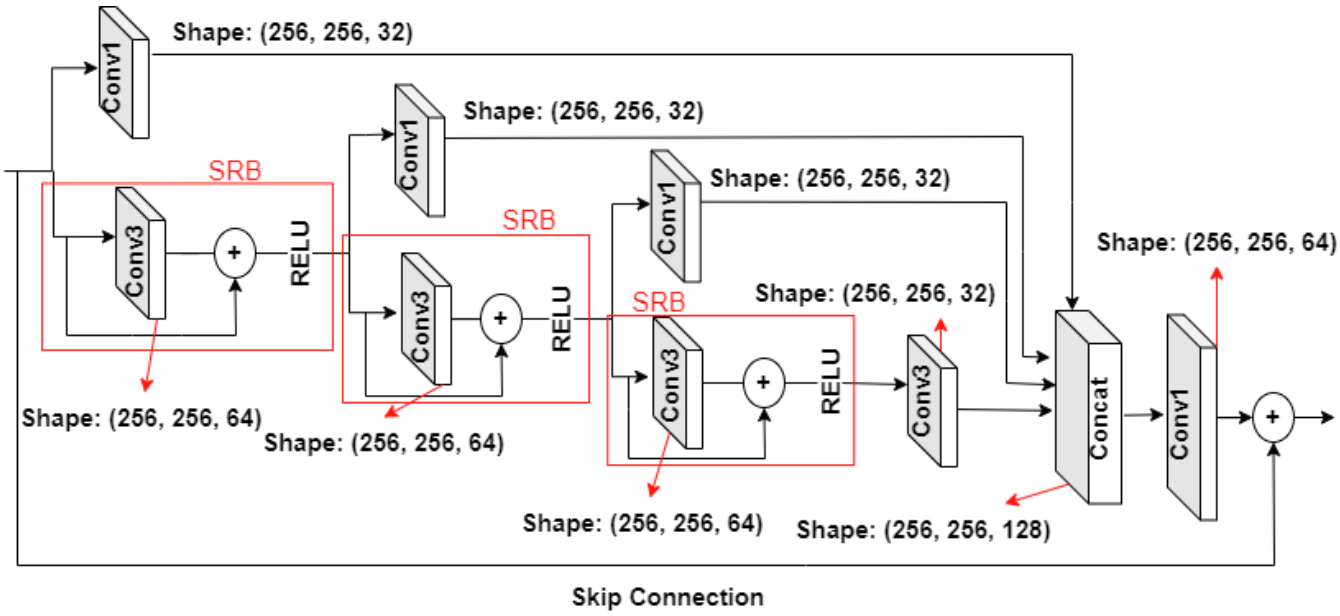


Figure 4.21 RFDB Structure

RFDN Algorithm

1. Input low resolution image with any desired shape, ex. (256,256,3) pass through a convolution layer 3x3 with shape (256,256,64) for feature extraction.
2. It will pass through 4 residual feature distillation block (RFDB), where the block consists of the three convolutions on the top with 1x1 convolutions with shapes (256,256,32), which significantly reduces the amount of parameters.
3. The bottom convolution uses 3x3 kernels with shapes (256,256,64). This is because it locates on the main body of the RFDB and it must take the spatial context into account to better refine the features.
4. We have 3 shallow residual block (SRB), as shown in the model structure, which consists of a 3x3 convolution, an identity connection and the activation unit RELU. The SRB can benefit from residual learning without introducing any extra parameters.
5. Then we pass the RFDB through convolution layer of kernel 1x1 with shape (256,256,64).
6. the previous convolution layer is passed to a convolution layer of kernel 3x3 with shape (256,256,64).
7. Then a convolution layer of kernel 3x3 with shape (256,256,256) takes the output of the first convolution layer and the output of the previous convolution layer.
6. Finally we apply subpixel for upsampling the image.
7. The output is the image super resolution after being upscales by 2. ex. (512,512,3).

RFDN Experiment details

The model takes the images from the dataset, and cross validate 80-20% then resize the images with the desired size for the high resolution image and then divide those images on the desired scale to make low resolution image. Then data is ready to enter the model described in fig. 4.20 and fig. 4.21. The following figures are learning curves of the training and validation loss shown in fig. 4.22, and the training and validation accuracy in fig. 4.23 of RFDN model.

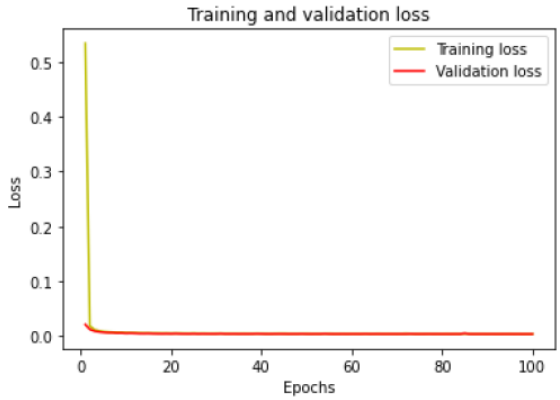


Figure 4.22 RFDN Training and Validation Loss Learning Curve

As seen in fig. 4.22, and fig. 4.23 the training and validation loss is decreasing and the training and validation accuracy is increasing with the increasing of the number of epochs, that shows that the model is well trained and no overfit. The model was trained on 100 epochs as mentioned before, we could have increased the number of epochs since there is a possibility that the loss may still decrease, but early stopping (mentioned in ch.2) should be put into consideration to prevent the model from overfitting.

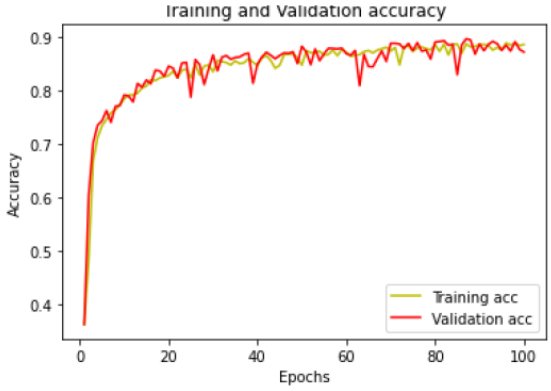


Figure 4.23 RFDN Training and Validation Accuracy Learning Curve

The following fig. 4.24 is a flowchart diagram for clarification of the whole implementation of the model.

RFDN Flowchart

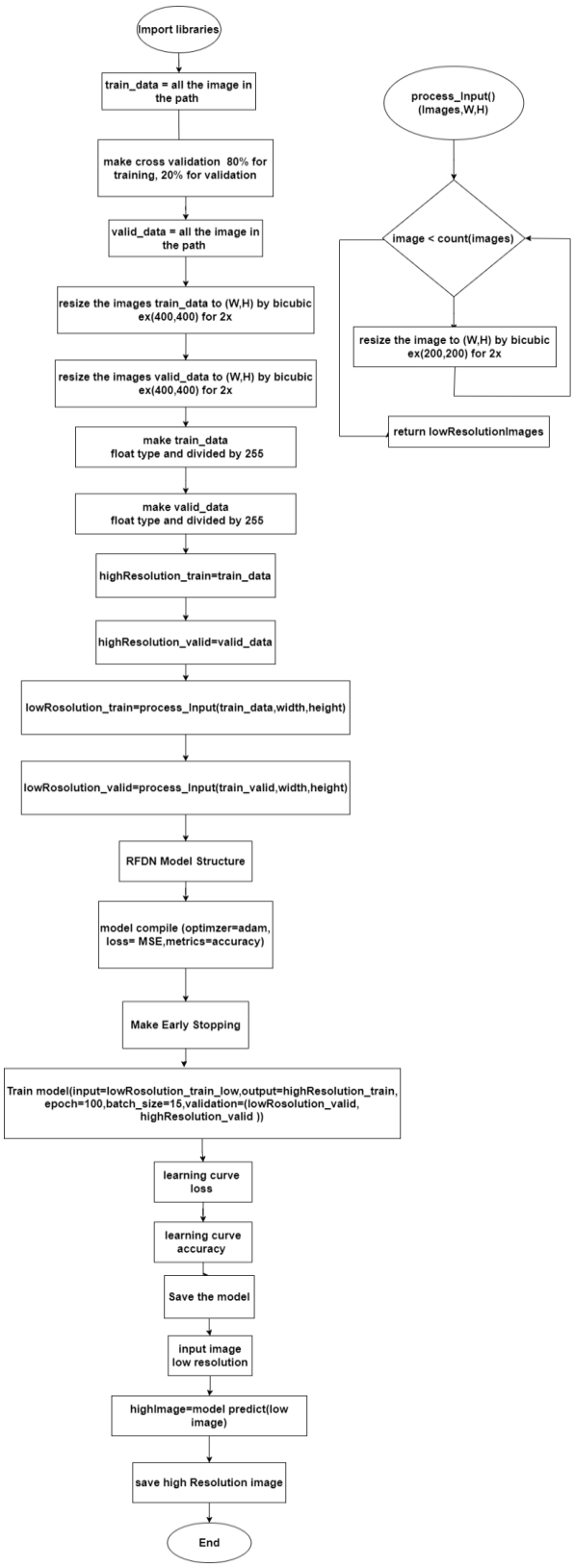


Figure 4.24 RFDN Flowchart

6. Autoencoder

Autoencoder has been implemented by two up-sampling techniques (subpixel and deconvolution), The model was trained on two different scales (2x and 4x), They have the same algorithm and structure. As mentioned, before it is trained on DIV2K dataset and 80-20 cross validation. With total number of parameters for 2x scale is 1,454,915, for 4x scale is 1,504,067 for deconvolution, for 2x scale is 1,586,179, and for 4x scale is 3,801,859 for subpixel. The following table 4.6 contains the hyper parameters of the model.

Table 4.6 Autoencoder deconvolution Parameters

Parameters	Autoencoder
Optimizer	ADAM
Learning Rate	10^{-3}
Activation Function	Relu
Loss Function	MSE
Number of Epochs	107
Batch Size	10

Autoencoder Model Structure

Autoencoder model structure fig. 4.25 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

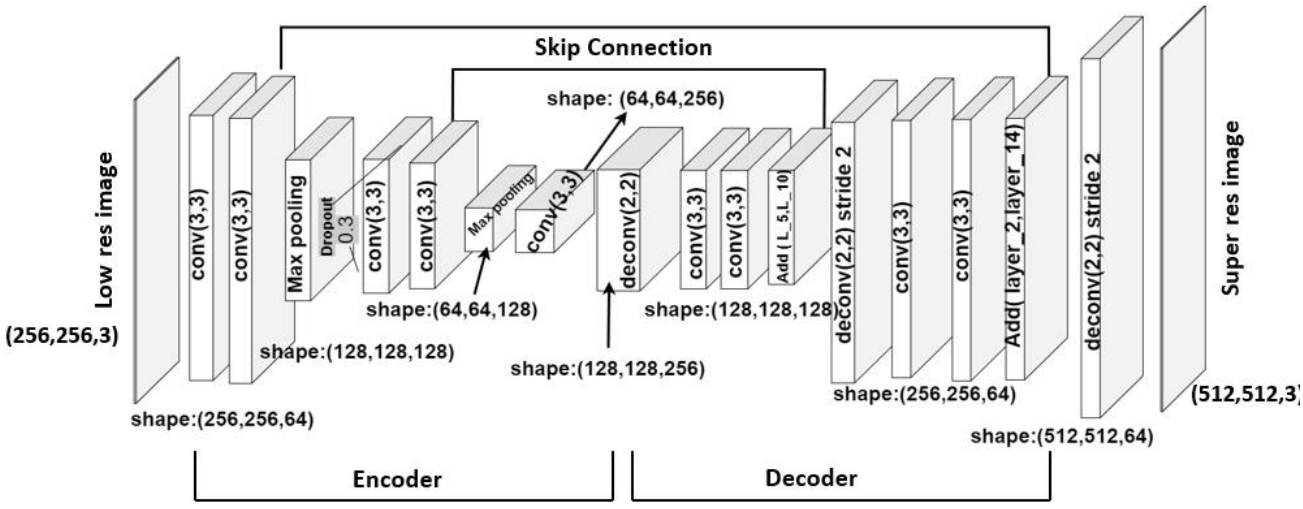


Figure 4.25 Autoencoder Model Structure

Autoencoder Algorithm

1. Adding two layers of convolutional two-dimensional with output shape of (256, 256, 64) and an activation layer of rectified linear units (ReLU) function, y=max(0,x). These two layers are stored separately in the outer layer 1 and extracts important features from the input image before going down to smaller feature space.
2. Max pooling layer, with the output shape of (128, 128, 64), is connected to the previous two-dimensional convolution layer and it reduces the dimensionality into smaller feature space.
3. Dropout layer, with the shape of (128, 128, 64), is used to regularize the complex neural network layers running in parallel.
4. Again, two layers of two-dimensional convolutional layers are applied one after the other with the same output shape of (128, 128, 128) and with the activation function of ReLU calculated from the formula given above. The layers up to this are stored in outer layer 2 for further processing.
5. Max pooling layer, with the output shape of (64, 64, 128), is connected to the previous two-dimensional convolution layer.
6. A two-dimensional convolution layer is applied after the previous max pooling layer with the output shape of (64, 64, 256).
7. Up-sampling layer, where we use transpose technique, with the output shape of (128, 128, 256), is applied to expand the small image matrix into a larger one.
8. Again, two layers of two-dimensional convolutional layers are applied one after the other with the same output shape of (128, 128, 128) and with the activation function of ReLU.
9. The addition operation is performed on the previously convolutional layer with the outer layer 2, which was stored for further processing with the shape of (128, 128, 128).
10. Up-sampling layer is applied again with the output shape of (256, 256, 128).
11. Two layers of two-dimensional convolutional layers are applied one after the other with the same output shape of (256, 256, 64) and with the activation function of ReLU.
12. Another operation is performed on the previously convolutional layer with the outer layer 1, which was stored for further processing with the shape of (256, 256, 64).
13. Up-sampling layer is applied again with the output shape of (512, 512, 64).
14. The final layer of the model is applied to the two-dimensional convolutional layer with the output shape of (512, 512, 3) to obtain the resultant autoencoder output.

The previous algorithm is just for the shape (256, 256, 3) but the model accepts any shape. We also implemented the model in different upscale methods transpose, deconvolution, and subpixel.

Autoencoder Experiment details

The model takes the images from the dataset, and cross validate 80-20% then resize the images with the desired size for the high-resolution image and then divide those images on the desired scale to make low resolution image. Then data is ready to enter the model described in fig. 4.25.

Auto Encoder Learning Curves

The following figures are learning curves of the training and validation loss, and the training and validation accuracy in fig. 4.26 of autoencoder model.

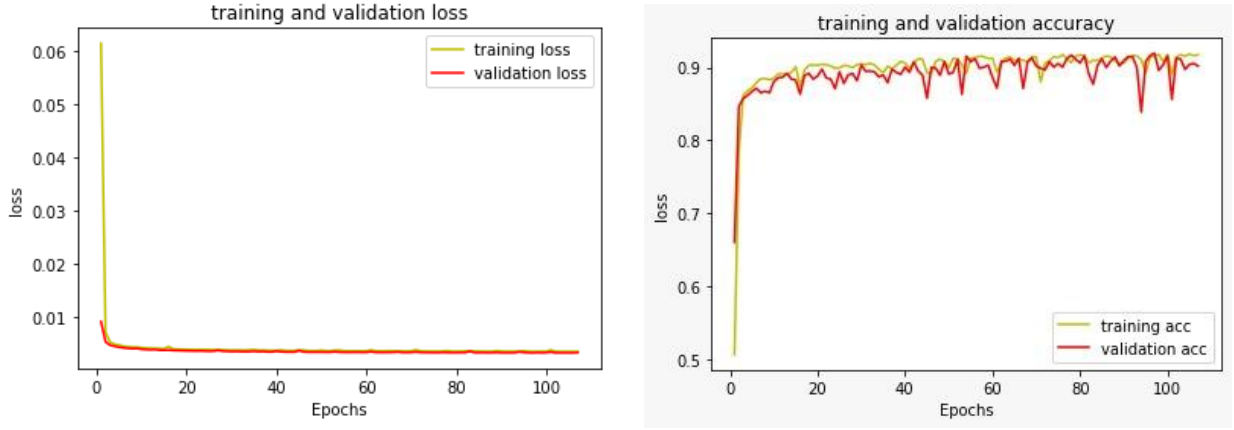


Figure 4.26 Autoencoder Training and Vlaidation Loss and Accuracy Learning Curve

As seen in fig. 4.26, the training and validation loss is decreasing and the training and validation accuracy is increasing with the increasing of the number of epochs, that shows that the model is well trained and no overfit. The model was trained on 120 epochs as mentioned before, but it stopped in epoch 107, because overfitting could've appeared, but the early stopping prevented the model from this.

The following fig. 4.27 is a flowchart diagram for clarification of the whole implementation of the model.

Autoencoder Flowchart

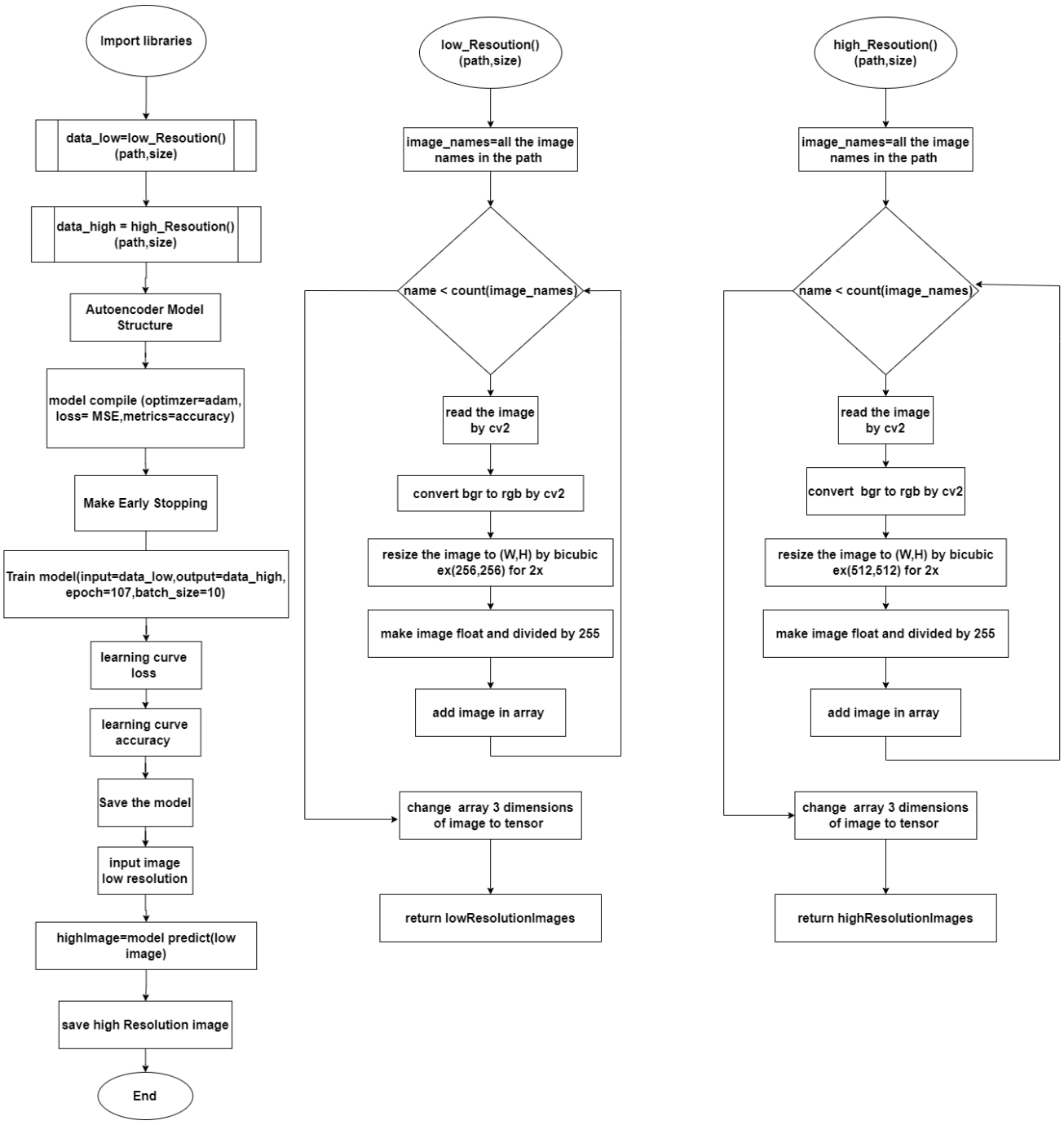


Figure 4.27 Autoencoder Flowchart

7. ESRGAN

Enhanced Super-Resolution Generative Adversarial Networks (ESRGAN) has been implemented using scale 4x. As mentioned, before it is a pretrained model and trained on DIV2K dataset. It divided into generator and discriminator with total number of parameters for generator is 16,697,987 and for discriminator is 14,505,161. The following table 4.7 contains the hyperparameters of the model

Table 4.7 ESRGAN Parameters

Parameters	**ESRGAN**
Optimizer	ADAM
Learning Rate G	$5*10^{-5}$
Learning Rate D	$5*10^{-5}$
Activation Function	Relu
Loss Function	MSE
Number of Epochs	2000
Batch Size	15

ESRGAN Model Structure

ESRGAN model structure fig. 4.28 view the layers of the model, the shape of each layer and the model flow. After it, is the model algorithm where each layer is described for more clarification.

ESRGAN as described in ch.3 has a generator and a discremnator shown in figures fig. 4.28, and fig. 4.29.

The following model structure is on a specific example of input image with size (256,256,3) but our model takes any input size.

Generator Structure

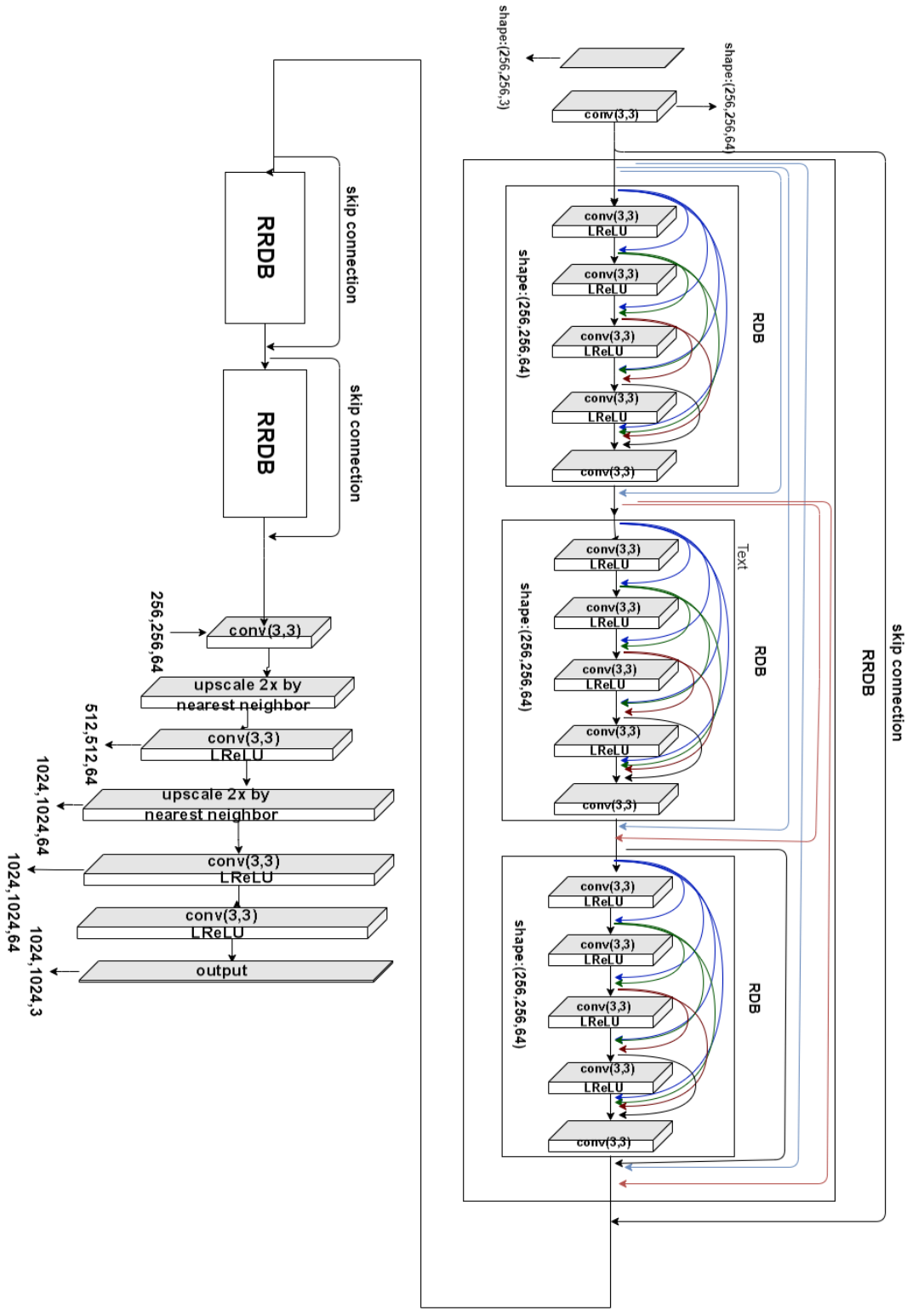


Figure 4.28 Generator Structure

Generator Algorithm

1.The low-resolution input with shape (256, 256, 3), which is passed through an initial convolutional layer of 3×3 kernels and 64 feature maps with shape (256, 256, 64).

2. The next layer of the feed-forward fully convolutional RRDB model. We have 3 RRDB

a. Each contains 4 convolutional layers of 3×3 kernels and 64 feature maps followed by a Parametric LReLU activation function. with shape (256, 256, 64).

b. Another convolutional layer of 3×3 kernels and 64 feature maps with shape (256, 256, 64).

3. We make use of the nearest neighbor in the generator model architecture after the 4x Up-sampling of the convolutional layer to produce the super-resolution images.

4. Then it enters another convolutional layer of 3×3 kernels and 64 feature maps followed by a Parametric LReLU layer

5. Then enter a convolution layer that up-sample the image to be resized into 4X the input size

6. Two layers of two-dimensional convolutional layers are applied one after the other with the activation function of LReLU.

Discriminator Structure

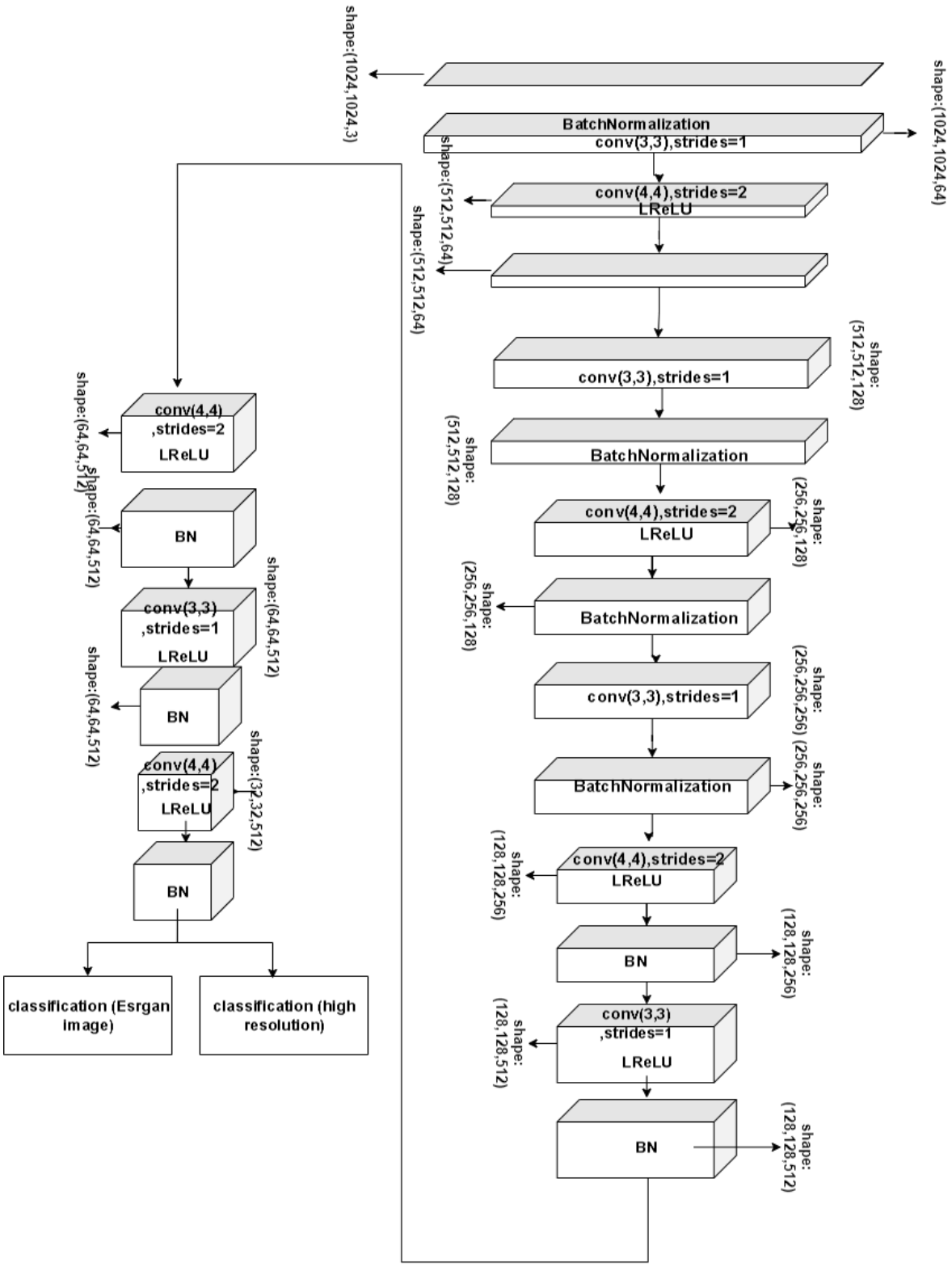


Figure 4.29 Discriminator Model Structure

Discriminator Algorithm

1.The high-resolution image with shape (1024, 1024, 3) and the ground-truth with shape (1024, 1024, 3) are the inputs of Discriminator, which passes through an initial convolutional layer of 3×3 kernels and 64 feature maps and stride 2. with shape (1024, 1024, 64).

2. The second layer is convolutional layer with 64 filters and the kernel size of 4 × 4 and stride 2 by a Parametric LReLU activation function to resize the image to half. with shape (512, 512, 64).

3. The third layer is batch normalization to make the network more stable during training.

4. The fourth layer is convolutional layer with 64 filters and the kernel size of 3 × 3 and stride 1 with shape (512, 512, 128).

5. The fifth layer is batch normalization to makes the network more stable during training.

6. The sixth layer is convolutional layer with 64 filters and the kernel size of 4 × 4 and stride 2 by a Parametric LReLU activation function to resize the image to half. with shape (256, 256, 128).

7. The seventh layer is batch normalization to makes the network more stable during training.

8.The following layers are the same from fourth layer to seventh layer are repeated four times. With the output shape (32,32,512).

9. The last layers measures probability of the High-resolution image is relatively more realistic than a ESRGAN image.

ESRGAN Experiment details

The model takes the high resolution and low resolution image for training and validation,then take the tfrecord path, and make a tfrecord for training and validation. Where it encode and read the images in binary and store them in a tfrecord file. That method helps in the very deep models which require a large number of parameters as it takes less space in memory, so kernel dead is avoided. Then read those tfrecords file and start training the model depending on the model structure that was described in fig. and the algorithm before.

The following fig. 4.30 is a flowchart diagram for clarification of the whole implementation of the model.

ESRGAN Learning Curves

ESRGAN is a pretrained model, so we don't have learning curve for it, but it has good results according to ch.5

ESRGAN Flowchart

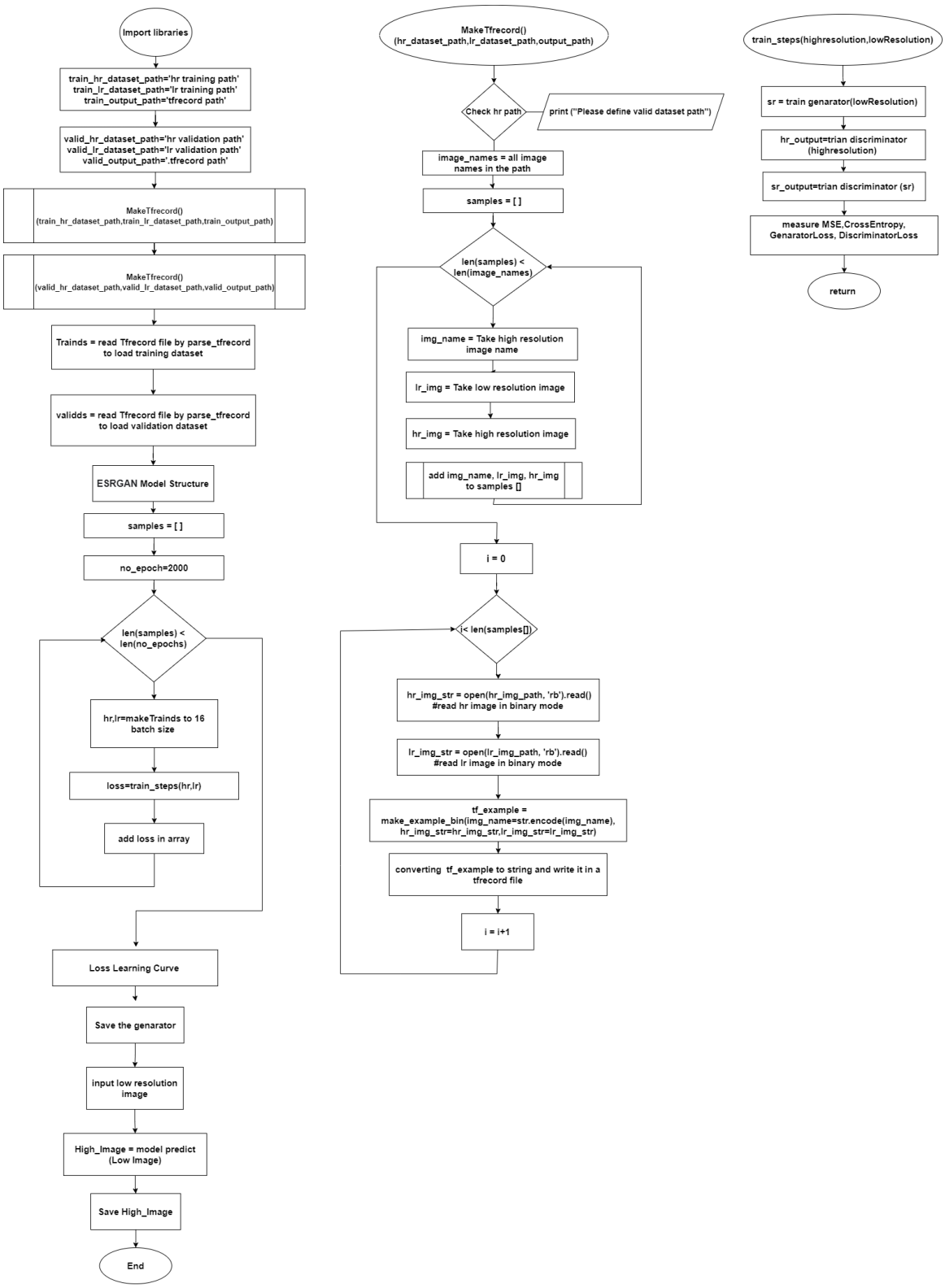


Figure 4.30 ESRGAN Flowchart

4.4 GUI

The user interface design process is an interaction process involving relations between users and system. The 3 core activities in this process are:

1. Understand what the users will do with the system. System prototyping.
2. Develop a series of prototypes for experiment. And Interface evaluation.
3. Experiment these prototypes with users.

4.4.1 System Specification:

Our System can work in any device, but we chose a windows application for simplicity, we used Tkinter library to create our application, Tkinter library is a cross-platform GUI framework that comes built into the Python standard library. Since it is a cross-platform framework, the same code can be used on any operating system, such as Linux, macOS, and Windows. Tkinter provides an object-oriented interface to the Tk GUI toolkit. The library tools include buttons, menus, and various kinds of entry fields and display areas.

4.4.2 Scenario:

1. User choose image from computer (.png and .jpg formats), choose the model from the implemented models we discussed before, and finally choose the scaling factor and click start.

2. Then screen will load and then display the input and output image, also the size of image and the chosen model.

3. Then the user can click download the image in the desired place in computer.

4.4.3 Validation:

If the user clicked start without entering the image, the application will display a message for the user to select the image.

4.4.4 Interface:

Our application consists of three pages:

1. **First Page**

The first page fig. 4.31 has 3 buttons, 2 dropdown menus, display image, loading bar.

a. Buttons

- Select Image → Where the user select the desired image (.png, .jpg formats) from his computer.
- About → To display the third page which is discussed later.
- Start → to start converting the low resolution image into a super resolution image.

b. Dropdown Menus

- Model → Contains the seven models (SRCNN, FSRCNN, ESPCN, ESRGAN, RDN, RFDN, Autoencoder)
- Scaling Factor → contains the available scaling factors of the chosen model (x2, x3, x4, or x8).

c. Display Image

- A box that display the chosen image after selecting it and the name of the image under the box. Fig. 4.32

d. Loading Bar

- After selecting Start, a loading bar is displayed till the conversion is completed and then display the second page. Fig. 4.33

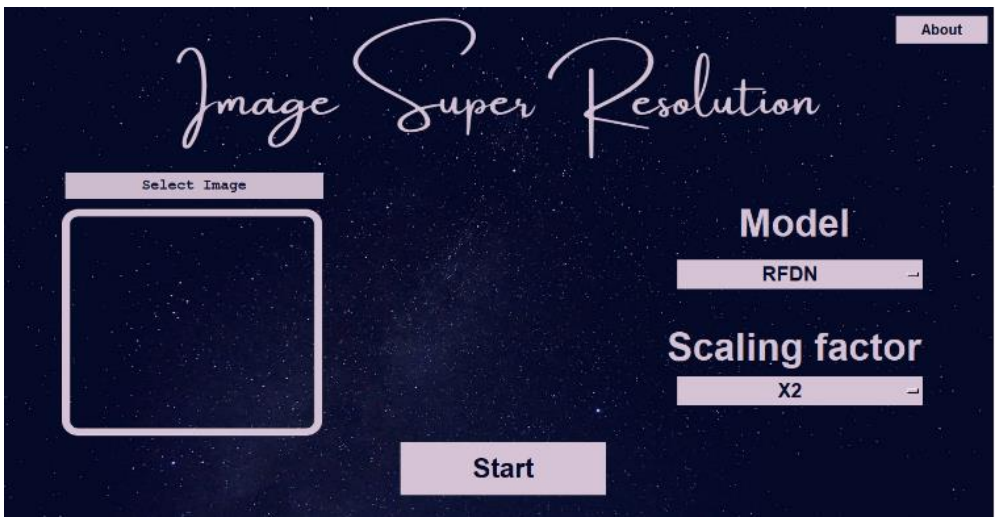


Figure 4.31 First page of GUI

Figure 4.32 Selected Image, Model, Scaling Factor

Figure 4.33 Loading Bar

2. Second Page

The second page fig. 4.34 has displays, 2 buttons.

a. Displays

- Input Image (Low Resolution Image)
- Output Image (Super resolution Image)
- Image size (before and after scaling)
- The chosen model

b. Buttons

- Download → User download (save) the super resolution image in the desired place in his computer.
- Select Other Image → User select new image if wanted.

Figure 4.34 Second Page of GUI

3. Third Page

The third page is the About Page, it displays the team members, supervisors, and a summary of the project.

It also has a back button to go back the home (first) page.

Figure 4.35 Third Page of GUI

Chapter 5

System Evaluation and Testing

Chapter 5: System Evaluation and Testing

This chapter discuss system evaluation and testing, where the test datasets are viewed and the performance of each model with results.

The performance can be assessed in 3 factors objectively, subjectively as it's mentioned before and time of processing (time taken for each model to predict the super resolution image)

5.1 Test Datasets

5.1.1 Set5 Dataset

The Set5 is a benchmark dataset commonly used for testing performance of Image Super-Resolution models and it consist of 5 images with different sizes ranges from (256,256) to (512,512).

5.1.2 Set14 Dataset

The Set14 is a benchmark dataset commonly used for testing performance of Image Super-Resolution models and it consist of 14 images with different sizes ranges from (276, 276) to (656,720).

5.1.3 Medical dataset

Medical dataset is a collected dataset used to test the system on medical application and consist of 5 images (periapical x-ray, x-ray) with different sizes ranges from (796, 824) to (1312, 1080)

5.1.4 Project's dataset

Project dataset is a collected dataset used to test the system on 8 images from our daily life and it consist of 8 with different sizes ranges from (1920, 1080) to (4624, 3468).

5.2 PSNR/ SSIM Results

The first factor of performance assessments is objectively, as mentioned before in chapter 2. PSNR and SSIM are computed for scale 2 and 4 in the 7 implemented models, on the 4 testing datasets described before and compared them with the similar applications mentioned before. The following tables are the average (PSNR/SSIM) of the tested datasets.

5.2.1 Set5 Dataset Results

Table 5.1 Set5 average (PSNR/SSIM) Results

Models		Scales		
		X2	**X4**	**X2X2**
Implemented models	**SRCNN**	32.63916/0.90650	31.1635/0.6996	31.661/0.780
	FSRCNN	32.58261/ 0.8710	31.3444/0.7421	30.318/0.714
	RDN	33.7007/ 0.8983	30.5153/0.7212	31.515/0.783
	ESPCN	31.66237/ 0.8891	31.3510/0.7090	30.366/0.780
	RFDN	34.13620/ 0.9069	31.3595/0.7626	31.857/0.793
	Autoencoder (Deconvolution)	34.45608/ 0.9142	31.4657/0.7842	32.190/0.820
	Autoencoder (Subpixel)	33.30608/ 0.8870	31.7322/0.7734	30.941/0.744
	ESRGAN	-	32.0604/0.8103	-
Similar Systems	**Adobe**	32.63916/ 0.85887	-	-
	Topaz	32.39536/ 0.85871	30.6606/0.6889	30.249/0.622
	Let's enhance	31.43409/0.80774	-	-

From the Previous table 5.1 it's assumed that for scale 2 the best results are for Autoencoder (Deconvolution) with results (34.45608/ 0.9142) and RFDN with results (34.13620/ 0.9069), While the worst results for Let's enhance with results (31.43409/0.80774) and ESPCN with results (31.66237/ 0.8891). However, for scale 4 ESRGAN has the best results with (32.0604/0.8103).

The 4th column in the table is X2X2 where it's an experiment to test the performance of double testing scale 2 and comparing it with scale 4. As seen, the difference between X2X2 and X4 are close. However, scale 4 takes more training time, while double scale 2 takes more testing time.

5.2.2 Set14 Dataset Results

Table 5.2 Set14 average (PSNR/SSIM) Results

Models		Scales	
		X2	X4
Implemented models	SRCNN	32.43738/0.80284	30.8697/0.58027
	FSRCNN	31.53111/0.78463	30.7079/0.61426
	RDN	32.32212/0.83647	30.04425/0.60644
	ESPCN	30.96817/0.80313	30.95899/0.5885
	RFDN	32.78939/0.85742	30.65615179/0.64027
	Autoencoder (Deconvolution)	32.01110/0.78212	30.80071/0.64341
	Autoencoder (Subpixel)	32.09487/0.78657	30.95064/0.63760
	ESRGAN	-	30.9526/0.63648
Similar Systems	Adobe	31.93029/0.81220	-
	Topaz	31.73144/0.79075	30.41046/0.5943
	Let's enhance	30.98211/0.75176	-

From the previous table 5.2, it's assumed that for scale 2 the best results are for RFDN with results (34.13620/ 0.9069) and SRCNN with results (32.43738/0.80284), while worst results are for let's enhance with results (30.98211/0.75176).

However, for scale 4 ESRGAN has the best results with (30.9526/0.63648).

5.2.3 Medical Dataset Results

Table 5.3 Medical average (PSNR/SSIM) results

Model		Scales	
		X2	X4
Implemented models	SRCNN	42.7448/0.9797	35.7031/0.80991
	FSRCNN	38.2166/0.9695	36.8302/0.90391
	RDN	43.6734/0.9849	37.0073/0.91915
	ESPCN	38.8061/0.9867	35.9979/0.80065
	RFDN	43.9371/0.9846	36.6184/0.89959
	Autoencoder (Deconvolution)	43.0859/0.9866	36.0785/0.91405

	Autoencoder (Subpixel)	41.0616/0.9689	37.2252/0.91079
	ESRGAN	-	35.1667/0.84963
Similar Systems	**Adobe**	41.0863/0.9634	-
	Topaz	35.7185/0.8924	33.99009/ 0.83946
	Let's enhance	32.8545/0.7240	-

From the previous table 5.3, it's assumed that for scale 2 the best results are for RFDN with results (43.9371/0.9846), RDN with results (43.6734/0.9849), while the worst results are for let's enhance with results (32.8545/0.7240).

However, for scale 4 Autoencoder (Deconvolution) (37.2252/0.91079) has the best results, and RDN with results (37.0073/0.91915).

5.2.4 Project's Dataset

Table 5.4 Project average (PSNR/SSIM) results

Model		**Scales**	
		X2	**X4**
Implemented models	**SRCNN**	35.84501/0.88099	33.47987/0.77020
	FSRCNN	34.28589/0.86739	33.26920/0.75841
	RDN	35.30866/0.88066	32.37607/0.74327
	ESPCN	34.04948/0.88422	33.25715/ 0.765083
	RFDN	35.43145/0.88052	32.44613/0.718838
	Autoencoder (Deconvolution)	35.3721/0.86799	32.5335/0.7643
	Autoencoder (Subpixel)	35.54974/0.87153	33.3456/0.76206
	ESRGAN	-	32.3277/0.68941
Similar Systems	**Adobe**	35.26006/0.879351	-
	Topaz	33.70534/0.82786	32.31021/0.70849

From the previous table 5.4, it's assumed that for scale 2 the best results are for SRCNN with results (35.84501/0.88099), RFDN with results (35.43145/0.88052) and the rest are close.

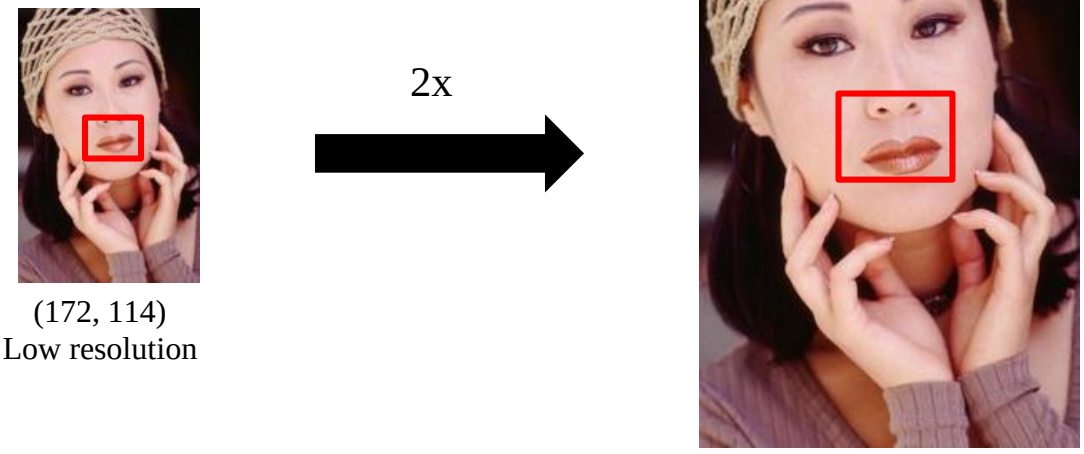
However, for scale 4 SRCNN (33.47987/0.77020), RFDN (32.44613/0.718838).

5.3 Comparisons

The second factor of performance assessment is subjectively, this section shows samples from the tested datasets (zoomed in) to make it easy to visualize and its objective assessment for the implemented models and 3 systems to compare it.

5.3.1 Set5 Dataset

Woman Image:



(172, 114)
Low resolution

(344, 228)
Super resolution

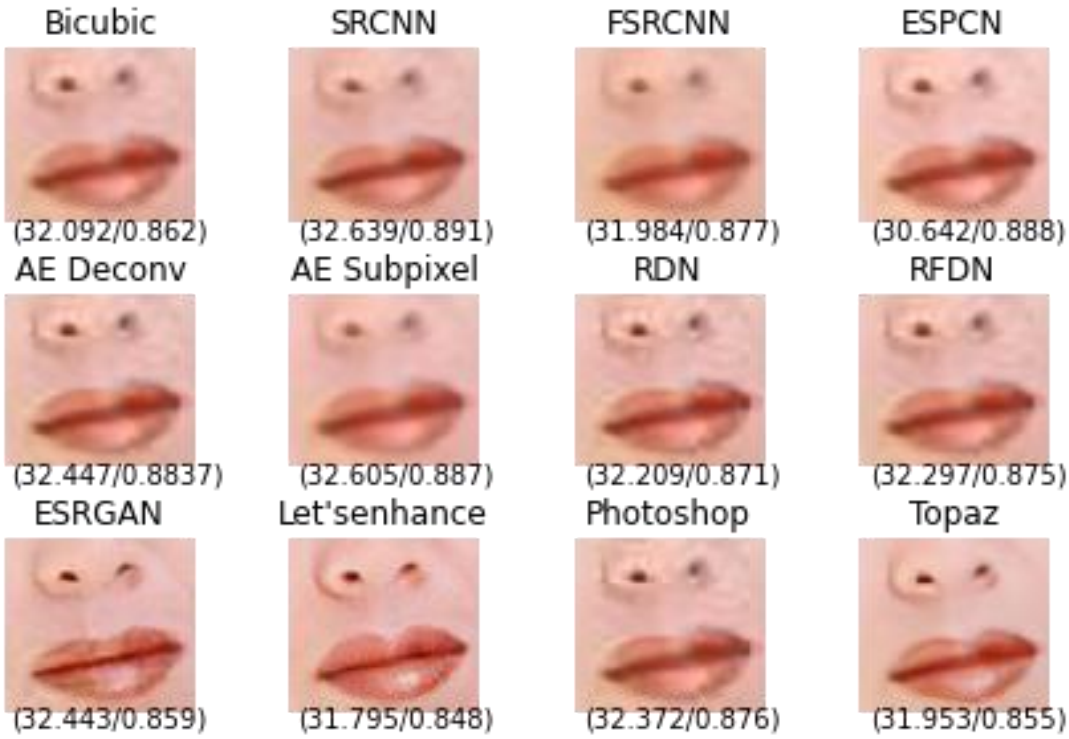
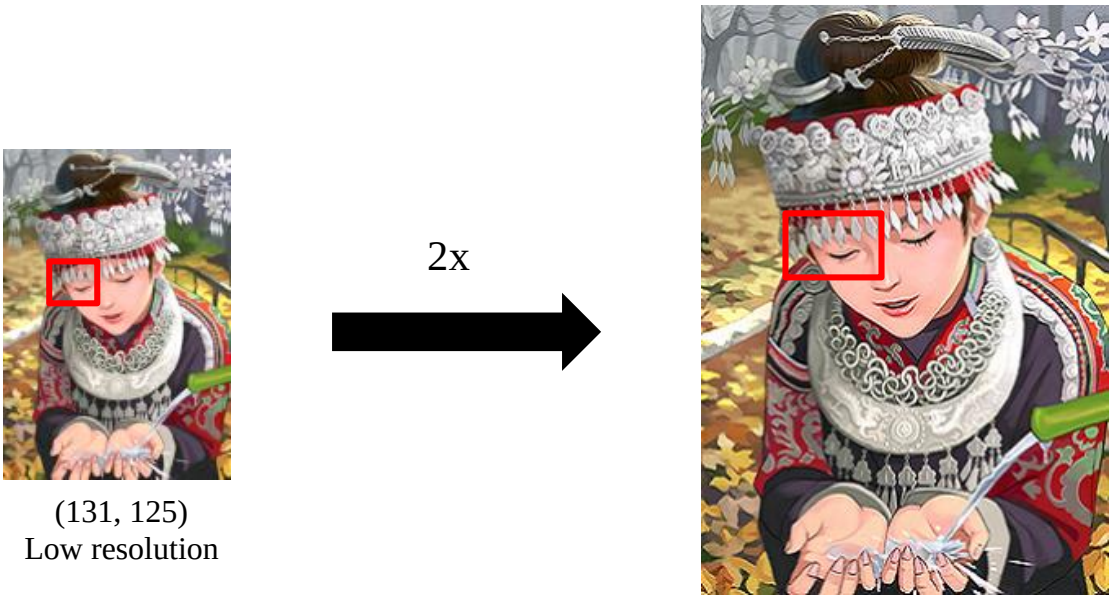


Figure 5.1 Set5 Woman Image Comparison

5.3.2 Set14 Dataset

Comic Image:



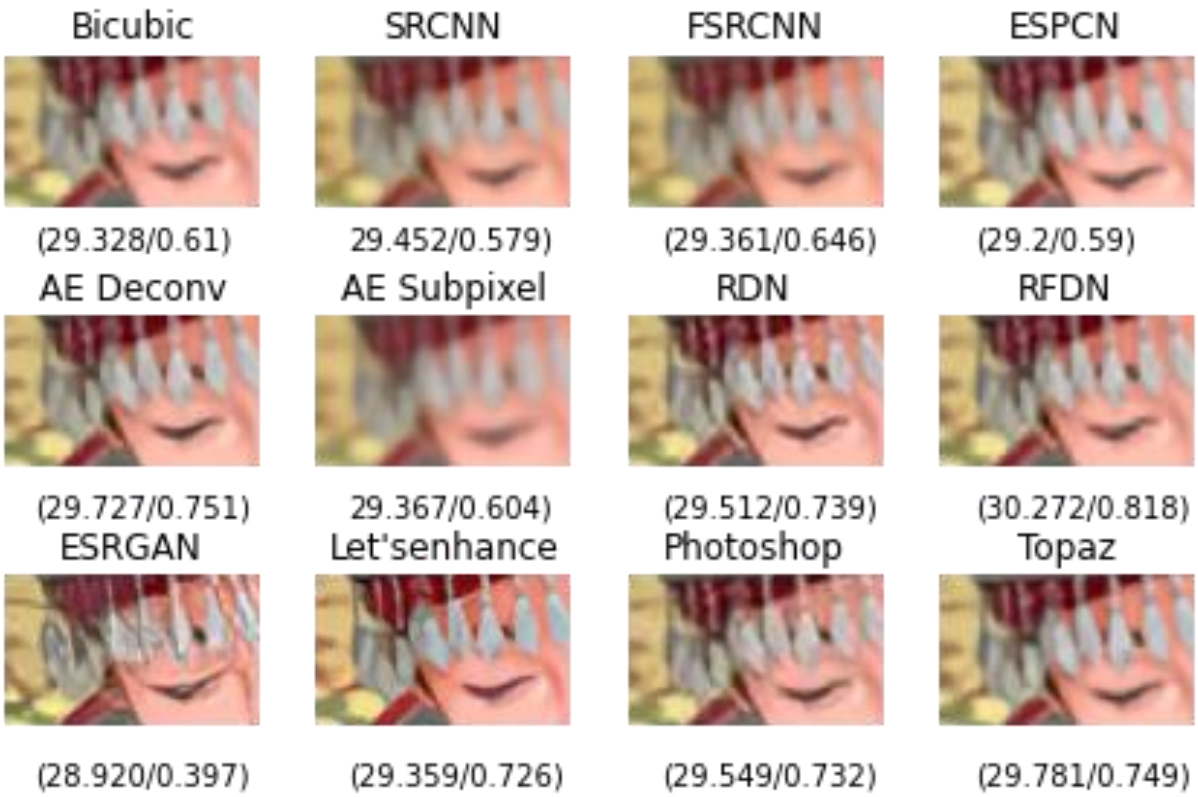
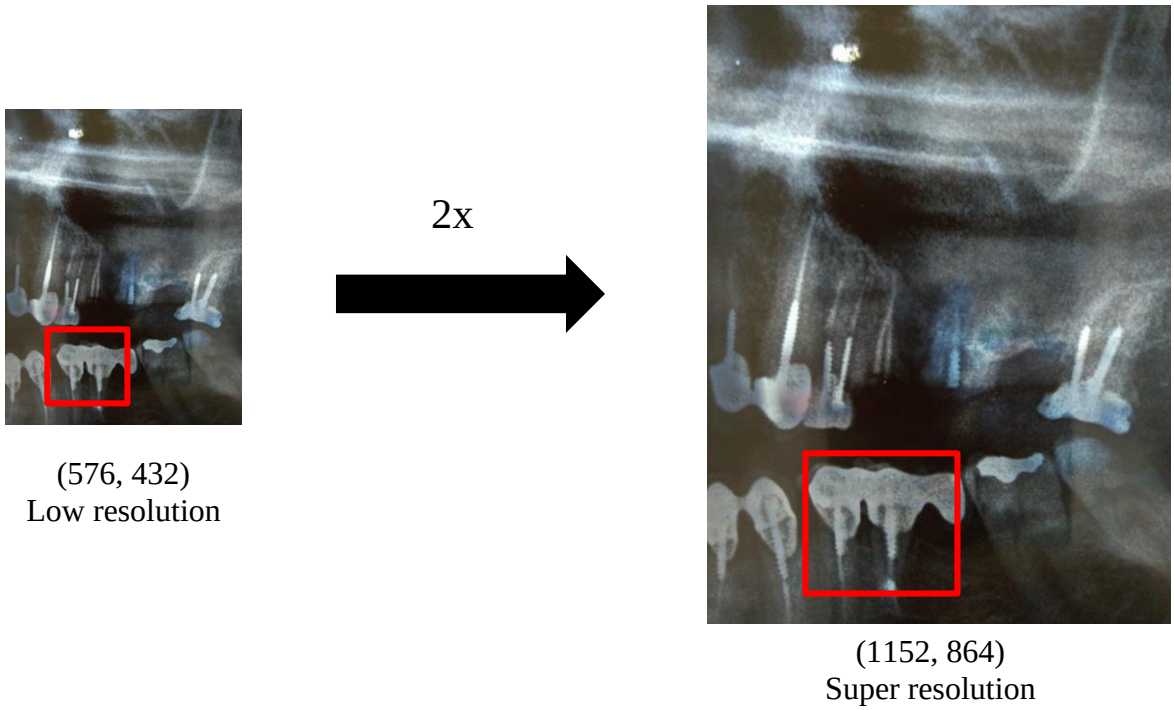


Figure 5.2 Set14 Comic Image Comparison

5.3.3 Medical Dataset

Fifth Image



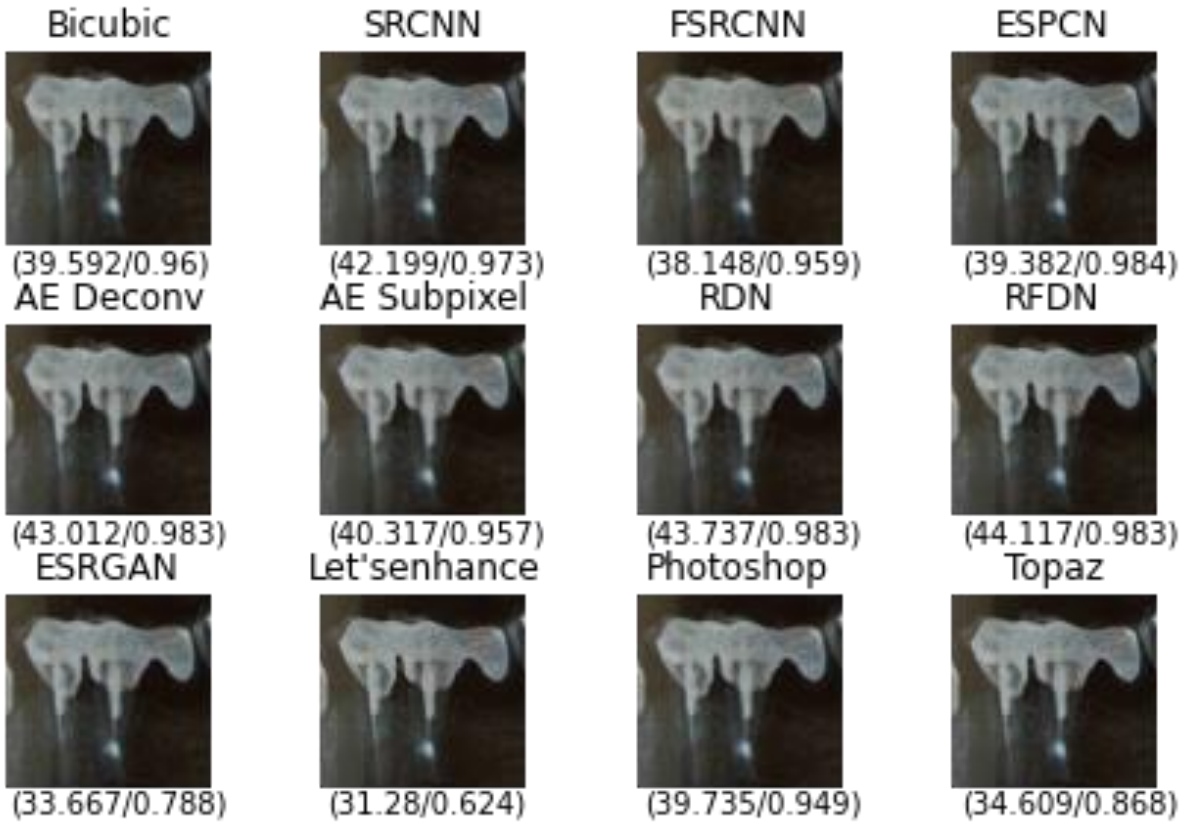


Figure 5.3 Medical Fifth Image Comparison

5.3.4 Project Dataset

Nature2 Image

(960, 540)
Low resolution

(1920, 1080)
Super resolution

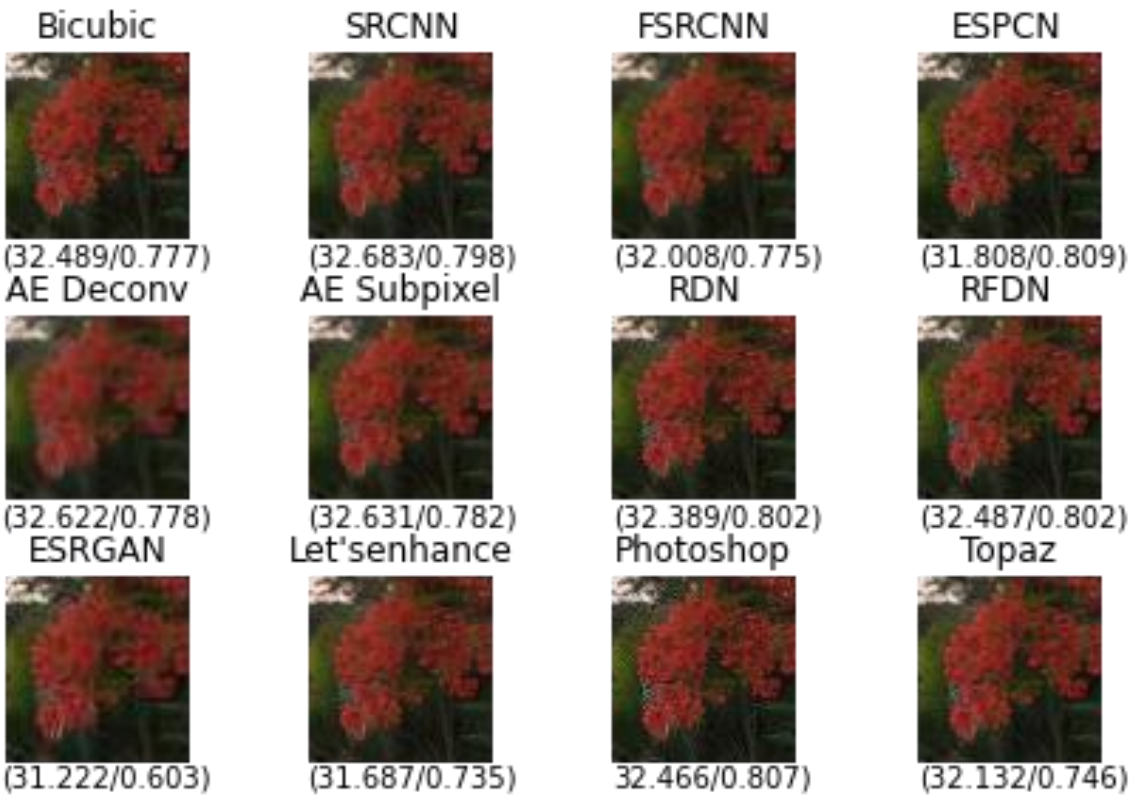


Figure 5.4 Project Dataset Nature2 Image Comparison

5.4 Time of processing

In this section shows the time by seconds taken for some images (from test datasets) to be converted to super resolution by time () function in python, also stating the size of each image.

5.4.1 Set5 Dataset

Table 5.5 Set5 Time Evaluation

Model	Baby (512, 512)		Bird (288, 288)		Woman (228, 344)	
	X2	X4	X2	X4	X2	X4
SRCNN	0.96	3.13	0.31	1.04	0.29	1.00
FSRCNN	0.31	0.34	0.19	0.12	0.08	0.12
RDN	82.01	32.02	22.84	22.98	22.02	9.39
ESPCN	0.20	0.22	0.08	0.09	0.09	0.08
RFDN	1.76	2.91	0.54	1.09	0.49	1.24
Autoencoder (Deconvolution)	0.97	1.79	0.57	0.80	0.32	0.54
Autoencoder (Subpixel)	1.19	2.63	0.58	1.03	0.38	0.80
ESRGAN	-	38.86	-	13.72	-	13.58

5.4.2 Set14 Dataset

Table 5.6 Set14 Time Evaluation

Model	Baboon (500, 480)		Face (276, 276)		Lenna (512, 512)	
	X2	X4	X2	X4	X2	X4
SRCNN	0.74	2.94	0.27	0.96	0.83	3.10
FSRCNN	0.20	0.32	0.07	0.10	0.22	0.36
RDN	84.29	30.57	21.28	20.62	67.68	33.14
ESPCN	0.20	0.20	0.08	0.09	0.19	0.23
RFDN	1.56	2.65	1.15	3.24	1.79	3.11
Autoencoder (Deconvolution)	0.87	1.59	0.32	0.53	0.96	1.78
Autoencoder (Subpixel)	1.06	2.41	0.53	0.79	1.17	2.61
ESRGAN	-	34.57	-	13.29	-	41.46

5.4.3 Medical Dataset

Table 5.7 Medical Time Evaluation

Model	First (1080, 1312)		Second (824, 796)		Third (960, 1280)	
	X2	X4	X2	X4	X2	X4
SRCNN	4.38	17.54	1.98	7.83	3.63	14.56
FSRCNN	1.17	1.85	0.49	0.84	1.01	1.62
RDN	325.81	243.18	202.07	142.08	298.71	150.96
ESPCN	0.82	1.07	0.39	0.52	0.72	0.95
RFDN	9.30	15.69	4.33	7.23	8.07	13.71
Autoencoder (Deconvolution)	5.46	9.58	2.37	4.39	4.32	8.14
Autoencoder (Subpixel)	6.44	14.33	2.95	6.64	5.46	12.25
ESRGAN	-	212.54	-	98.13	-	184.29

5.4.4 Project Dataset

Table 5.8 Project Dataset Time Evaluation

Model	Details (816, 612)		MTI (867, 1156)		Toys (879, 732)	
	X2	X4	X2	X4	X2	X4
SRCNN	1.55	6.15	3.01	11.94	1.91	7.89
FSRCNN	0.38	0.64	0.84	1.27	0.48	0.81
RDN	116.27	142.18	232.88	123.47	148.93	144.72
ESPCN	0.31	0.41	0.77	0.82	0.44	0.56
RFDN	3.97	10.34	7.23	21.00	4.63	21.05
Autoencoder (Deconvolution)	1.82	3.34	3.66	6.76	2.36	4.35
Autoencoder (Subpixel)	2.22	5.04	4.47	8.32	2.88	6.56
ESRGAN	-	77.53	-	152.27	-	116.50

From the previous tables, we can conclude that FSRCNN and ESPCN are the fastest models. While RDN and ESRGAN are the slowest models.

Chapter 6
Conclusion

Chapter 6: Conclusion and Future Works

In this section, the concludes which models are better on the 3 factors that was mentioned before, the experiments done on the implemented models, and the set of experiments having different algorithm and parameters are established to compare the efficiency among different impact of factors. Then the future work will be declared.

6.1 Conclusion

We have given a comprehensive overview of DL-based single image super resolution models (Image Super-Resolution Using Deep Convolutional Networks (SRCNN), Fast Super Resolution Convolutional Neural Network (FSRCNN), Residual Dense Network (RDN), Efficient Sub-Pixel Convolutional neural network (ESPCN), Residual Feature Distillation Network for Lightweight Image Super-Resolution (RFDN), Deep Auto-encoder for Single Image Super-Resolution, and Single Image Super Resolution using Enhanced Generative Adversarial Network (ESRGAN)) according to their architecture and results, including reconstruction efficiency, reconstruction accuracy, perceptual quality, and other technologies that can further improve models performance.

Meanwhile, we provided a detailed introduction to the related works of SISR, and compared the results of the implemented models with the similar applications. In order to view the performance of each model more intuitively, we also provided a detailed comparison of reconstruction results.

According to the 3 factors mentioned in ch.5, from the results we can assume that:

1. **Objectively:** The most consistent model over all the tested datasets is the RFDN.
2. **Subjectively:** The most consistent model over all the tested datasets is the RFDN, and Let's Enhance.
3. **Time of Processing:** The fastest models are ESPCN, and FSRCNN.

Implemented models had a higher accuracy than the 3 testing applications that were used for comparison almost on every dataset.

ESRGAN is the best model in scale 4, however putting the time into consideration so FSRCNN .

RDN have higher results on the medical dataset takes long time but so in medical applications where time isn't needed as the quality it's better in this case

6.2 Future Work

- Improving results of the models by training the models on more time and larger datasets.
- Build a Web app with stable server that can handle the models and predict fast super resolution images.
- Build a Mobile app (IOS, and Android) that can predict fast super resolution images.
- Modification of the architecture used in building the model so that we can achieve higher accuracy and less information loss.
- Working on videos Super Resolution.

References

[1] X. Zhang, E. Y. Lam, E. X. Wu, and K. K. Y. Wong, "Application of Tikhonov regularization to super-resolution reconstruction of brain MRI images," in *International Conference on Medical Imaging and Informatics*, 2007, pp. 51–56.

[2] L. Yue, H. Shen, J. Li, Q. Yuan, H. Zhang, and L. Zhang, "Image super-resolution: The techniques, applications, and future," *Signal Processing*, vol. 128, pp. 389–408, 2016.

[3] H. Shen, M. K. Ng, P. Li, and L. Zhang, "Super-resolution reconstruction algorithm to MODIS remote sensing images," *Comput. J.*, vol. 52, no. 1, pp. 90–100, 2009.

[4] S. M. A. Bashir, Y. Wang, M. Khan, and Y. Niu, "A comprehensive review of deep learning-based single image super-resolution," *PeerJ Comput. Sci.*, vol. 7, p. e621, 2021.

[5] Z. Wang, J. Chen, and S. C. H. Hoi, "Deep learning for image super-resolution: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 10, pp. 3365–3387, 2020.

[6] P. S. Parsania and P. V Virparia, "Computational time complexity of image interpolation algorithms," *Int. J. Comput. Sci. Eng.*, vol. 6, pp. 491–496, 2018.

[7] D. Su and P. Willis, "Image Interpolation by Pixel-Level Data-Dependent Triangulation," in *Computer graphics forum*, 2004, vol. 23, no. 2, pp. 189–201.

[8] W. Shi *et al.*, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1874–1883.

[9] J. Li, Z. Pei, and T. Zeng, "From beginner to master: A survey for deep learning-based single-image super-resolution," *arXiv Prepr. arXiv2109.14335*, 2021.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[11] A. Aggarwal, M. Mittal, and G. Battineni, "Generative adversarial network: An overview of theory and applications," *Int. J. Inf. Manag. Data Insights*, vol. 1, no. 1, p. 100004, 2021.

[12] W. Wang, Y. Hu, Y. Luo, and T. Zhang, "Brief survey of single image super-resolution reconstruction based on deep learning approaches," *Sens. Imaging*, vol. 21, no. 1, pp. 1–20, 2020.

[13] R. Mormont, P. Geurts, and R. Marée, "Comparison of deep transfer learning strategies for digital pathology," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2018, pp. 2262–2271.

[14] F. Zhuang *et al.*, "A comprehensive survey on transfer learning," *Proc. IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[15] P. Mohammadi, A. Ebrahimi-Moghadam, and S. Shirani, "Subjective and objective quality assessment of image: A survey," *arXiv Prepr. arXiv1406.7799*, 2014.

[16] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 2, pp.

295–307, 2015.

[17] W.-S. Lai, J.-B. Huang, N. Ahuja, and M.-H. Yang, “Fast and accurate image super-resolution with deep laplacian pyramid networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 11, pp. 2599–2613, 2018.

[18] B. Lim, S. Son, H. Kim, S. Nah, and K. Mu Lee, “Enhanced deep residual networks for single image super-resolution,” in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 136–144.

[19] Y. Zhang, Y. Tian, Y. Kong, B. Zhong, and Y. Fu, “Residual dense network for image super-resolution,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2472–2481.

[20] J. W. Soh and N. I. Cho, “Lightweight single image super-resolution with multi-scale spatial attention networks,” *IEEE Access*, vol. 8, pp. 35383–35391, 2020.

[21] S. Kim, D. Jun, B.-G. Kim, H. Lee, and E. Rhee, “Single image super-resolution method using CNN-based lightweight neural networks,” *Appl. Sci.*, vol. 11, no. 3, p. 1092, 2021.

[22] B. Li, B. Wang, J. Liu, Z. Qi, and Y. Shi, “s-lwsr: Super lightweight super-resolution network,” *IEEE Trans. Image Process.*, vol. 29, pp. 8368–8380, 2020.

[23] J. Liu, J. Tang, and G. Wu, “Residual feature distillation network for lightweight image super-resolution,” in *European Conference on Computer Vision*, 2020, pp. 41–55.

[24] J. Kim, S. Song, and S.-C. Yu, “Denoising auto-encoder based image enhancement for high resolution sonar image,” in *2017 IEEE underwater technology (UT)*, 2017, pp. 1–5.

[25] X. Wang *et al.*, “Esrgan: Enhanced super-resolution generative adversarial networks,” in *Proceedings of the European conference on computer vision (ECCV) workshops*, 2018, p. 0.

Appendix

Appendix A: Datasets

Here is a sample of the training dataset and the 4 testing datasets.

1. **Div2k**:

Number 675

Shape=(2040,1356)

Number 235

Shape=(2040,1356)

2. **Set5:**

butterfly

Shape=(256, 256)

Baby

Shape=(512, 512)

3. **Set14:**

Pepper

Shape=(512, 512)

comic

Shape=((362, 250)

4. **Medical:**

Second

Shape=(796, 824)

third

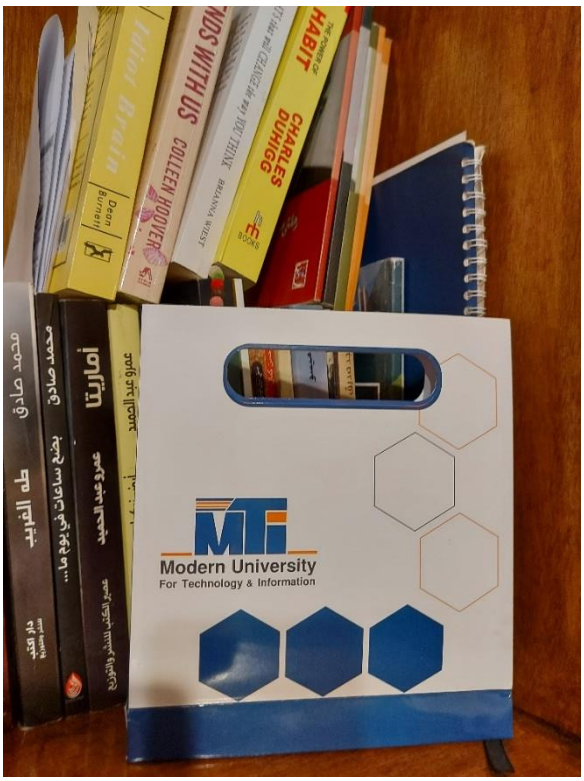
Shape=(1280, 960)

5. Project dataset:

cat

Shape=(4624, 3468)

MTI



Shape=(4624, 3468)

Appendix B: Codes

Training

1. SRCNN

```
pip install opencv-python
pip install tensorflow
pip install os-sys
pip install scikit-image
pip install opencv-python-headless
pip install scikit-image

import sys
import numpy
import matplotlib
import skimage
import tensorflow as tf
import cv2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.optimizers import Adam
from skimage import measure
#import measure.compare_ssim as ssim
from skimage.metrics import structural_similarity as ssim
from matplotlib import pyplot as plt
import numpy as np
import math
import os
import matplotlib.pyplot as plt

"""# Load low resolution dataset """

def train_low(path,size,scale):
    names = sorted(os.listdir(path)) # get  all items  in path
    print(len(names)) # print no. of img
    data = []
    for name in names:
        fpath = path + name  # add name  of img to the path
        img = cv2.imread(fpath, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (size,size),cv2.INTER_CUBIC)  # downsample bicubic
        img = cv2.resize(img, ((size*scale),(size*scale)),cv2.INTER_CUBIC) # make  upsamble bicubic to model
        img = modcrop(img, scale)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2YCrCb)
        img = img[:, :, 0].astype(float) / 255   # y is the luma
```

```
        shape = img.shape
        data.append(img)
    data = np.array(data)
    return data

"""# Load high resolution dataset """

def train_high(path,size,scale):
    names = sorted(os.listdir(path))
    print(len(names))
    data = []
    for name in names:
        fpath = path + name
        img = cv2.imread(fpath, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (size,size))
        img = modcrop(img, scale)
        img=shave(img, 6)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2YCrCb)
        img = img[:, :, 0].astype(float) / 255   # y is the luma.
        shape = img.shape
        data.append(img) # no. of images  #900 ---->(w,h)
    data = np.array(data)   # (no. of image, w, h)
    return data

"""# Model"""

# define the SRCNN model
def model():

    # define model type
    SRCNN = Sequential()

    # add model layers  #    (1392, 2040, 3)
    SRCNN.add(Conv2D(filters=128, kernel_size = (9, 9), kernel_initializer='glorot_uniform',
                     activation='relu', padding='valid', use_bias=True, input_shape=(None,None, 1)))
    SRCNN.add(Conv2D(filters=64, kernel_size = (3, 3), kernel_initializer='glorot_uniform',
                     activation='relu', padding='same', use_bias=True))
    SRCNN.add(Conv2D(filters=1, kernel_size = (5, 5), kernel_initializer='glorot_uniform',
                     activation='linear', padding='valid', use_bias=True))

    # define optimizer
    adam = Adam(lr=0.0003)

    # compile model
```

```
    SRCNN.compile(optimizer=adam, loss='mean_squared_error', metrics='accuracy')
    return SRCNN

srcnn = model()
srcnn.summary()

# define necessary image processing functions
def modcrop(img, scale):
    tmpsz = img.shape
    sz = tmpsz[0:2]       #
    sz = sz - np.mod(sz, scale)
    img = img[0:sz[0], 1:sz[1]]
    return img

def shave(image, border):
    img = image[border: -border, border: -border]
    return img

Xtrain_high =train_high('./data/DIV2K_train_HR/',512,2)

Xtrain_high.shape

y_train_low=train_low('./data/DIV2K_train_HR/',256,2)

y_train_low.shape

low= numpy.zeros((y_train_low.shape[0], y_train_low.shape[1], y_train_low.shape[2], 1), dtype=float)
low[:, :, :, 0]=y_train_low[:, :, :].astype(float)
low.shape

high= numpy.zeros((Xtrain_high.shape[0], Xtrain_high.shape[1], Xtrain_high.shape[2], 1), dtype=float)
high[:, :, :, 0]=Xtrain_high[:, :, :].astype(float)
high.shape

callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=10,restore_best_weights=True)

history=srcnn.fit(low, high, epochs=100, batch_size=10, validation_split=0.2, callbacks=[callback], shuffle=True)

loss=history.history["loss"]
val_loss=history.history["val_loss"]
epochs=range(1,len(loss)+1)
plt.plot(epochs,loss,'y',label="training loss")
plt.plot(epochs,val_loss,'r',label="validation loss")
plt.title("training and validation loss")
```

```
plt.xlabel("Epochs")
plt.ylabel("loss")
plt.legend()
plt.show()

acc=history.history["accuracy"]
val_acc=history.history["val_accuracy"]
epochs=range(1,len(loss)+1)
plt.plot(epochs,acc,'y',label="training acc")
plt.plot(epochs,val_acc,'r',label="validation acc")
plt.title("training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("loss")
plt.legend()
plt.show()

srcnn.save('srcnn_model2x.h5')
```

2. FSRCNN Model

```
pip install tensorflow
pip install opencv-python
pip install opencv-python-headless
pip install scikit-image

from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout
from tensorflow.keras.layers import Conv2DTranspose, UpSampling2D, add,PReLU, Activation
from skimage.transform import resize, rescale
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
import matplotlib.pyplot as plt
from scipy import ndimage, misc
from matplotlib import pyplot
import tensorflow as tf
import numpy as np
np.random.seed(0)
import re
import os
from tensorflow.keras.models import Sequential
import cv2

def train_low(path,size):
    names = sorted(os.listdir(path)) # get  all items  in path
    data = []
    for name in names:
        fpath = path + name  # add name  of img to the path
        img = cv2.imread(fpath, cv2.IMREAD_COLOR)
        img=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (size,size),cv2.INTER_CUBIC)  # downsample bicubic
        img = img[:, :, :].astype(float) / 255
        data.append(img)
    data = np.array(data)
    return data
def train_high(path,size):
    names = sorted(os.listdir(path)) # get  all items  in path
    data = []
    for name in names:
        fpath = path + name  # add name  of img to the path
        img = cv2.imread(fpath, cv2.IMREAD_COLOR)
        img=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (size,size),cv2.INTER_CUBIC)  # downsample bicubic
        img = img[:, :, :].astype(float) / 255
        data.append(img)
```

```
    data = np.array(data)
    return data
d=56
s=12
m=4
from tensorflow.keras import initializers
FRCNN = Sequential()
FRCNN.add(Input(shape=(None, None, 3)))
# feature extraction
FRCNN.add(
    Conv2D(
        kernel_size=5,
        filters=d,
        padding="same",
        kernel_initializer=initializers.HeNormal(),
    )
)
FRCNN.add(PReLU(alpha_initializer="zeros", shared_axes=[1, 2]))
# shrinking
FRCNN.add(
    Conv2D(
        kernel_size=1,
        filters=s,
        padding="same",
        kernel_initializer=initializers.HeNormal(),
    )
)
FRCNN.add(PReLU(alpha_initializer="zeros", shared_axes=[1, 2]))
# mapping
for _ in range(m):
    FRCNN.add(
        Conv2D(
            kernel_size=3,
            filters=s,
            padding="same",
            kernel_initializer=initializers.HeNormal(),
        )
    )
    FRCNN.add(PReLU(alpha_initializer="zeros", shared_axes=[1, 2]))
# expanding
FRCNN.add(Conv2D(kernel_size=1, filters=d, padding="same"))
FRCNN.add(PReLU(alpha_initializer="zeros", shared_axes=[1, 2]))
# deconvolution
FRCNN.add(
```

```
    Conv2DTranspose(
        kernel_size=4,
        filters=3,
        strides=2,
        padding="same",
        kernel_initializer=initializers.RandomNormal(mean=0, stddev=0.001),
    )
)
FRCNN.summary()
FRCNN.compile(optimizer = 'adam', loss = 'mean_squared_error',metrics='accuracy')
low_data = train_low("./images/DIV2K_train_HR/",256)
high_data = train_high("./images/DIV2K_train_HR/",512)
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=10,restore_best_weights=True)

history=FRCNN.fit(low_data,                                                    high_data,epochs=100,batch_size=10,shuffle=True,
callbacks=[callback],validation_split=0.2)
loss=history.history["loss"]
val_loss=history.history["val_loss"]
epochs=range(1,len(loss)+1)
plt.plot(epochs,loss,'y',label="training loss")
plt.plot(epochs,val_loss,'r',label="validation loss")
plt.title("training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("loss")
plt.legend()
plt.show()
acc=history.history["accuracy"]
val_acc=history.history["val_accuracy"]
epochs=range(1,len(loss)+1)
plt.plot(epochs,acc,'y',label="training acc")
plt.plot(epochs,val_acc,'r',label="validation acc")
plt.title("training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("loss")
plt.legend()
plt.show()
FRCNN.save('FRCNN1_X2.h5')
```

3. ESPCN Model

```
import tensorflow as tf
import os
import math
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import array_to_img
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing import image_dataset_from_directory
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
import PIL
from IPython.display import display

root_dir= "DIV2K_train_HR" #path of div2k hr folder
resize = 400
upscale_factor = 2
input_size = resize // upscale_factor
batch_size = 15
train_ds = image_dataset_from_directory(
    root_dir,
    batch_size=batch_size,
    image_size=(resize, resize),
    validation_split=0.2,
    subset="training",
    seed=1337,
    label_mode=None,
    interpolation='bicubic'
)

valid_ds = image_dataset_from_directory(
    root_dir,
    batch_size=batch_size,
    image_size=(resize, resize),
    validation_split=0.2,
    subset="validation",
    seed=1337,
    label_mode=None,
    interpolation='bicubic'
)
```

```
def scaling(input_image):
    input_image = input_image / 255.0
    return input_image
# Scale from (0, 255) to (0, 1)
train_ds = train_ds.map(scaling)
valid_ds = valid_ds.map(scaling)
#to display random images in your training dataset
for batch in train_ds.take(1):
    for img in batch:
        display(array_to_img(img))
def process_input(input, input_size, upscale_factor):
    input = tf.image.rgb_to_yuv(input)
    last_dimension_axis = len(input.shape) - 1
    y, u, v = tf.split(input, 3, axis=last_dimension_axis)
    return tf.image.resize(y, [input_size, input_size], method="area")
def process_target(input):
    input = tf.image.rgb_to_yuv(input)
    last_dimension_axis = len(input.shape) - 1
    y, u, v = tf.split(input, 3, axis=last_dimension_axis)
    return y
train_ds = train_ds.map(
    lambda x: (process_input(x, input_size, upscale_factor), process_target(x))
)
train_ds = train_ds.prefetch(buffer_size=32)
valid_ds = valid_ds.map(
    lambda x: (process_input(x, input_size, upscale_factor), process_target(x))
)
valid_ds = valid_ds.prefetch(buffer_size=32)
#to display the images in training before and after
for batch in train_ds.take(1):
    for img in batch[0]:
        display(array_to_img(img))
    for img in batch[1]:
        display(array_to_img(img))
#building model
#Compared to the paper, they add one more layer and they used the relu activation function instead of tanh.
#It achieves better performance even though we train the model for fewer epochs.
def get_model(upscale_factor=2, channels=1):
    conv_args = {
        "activation": "relu",
        "kernel_initializer": "Orthogonal",
        "padding": "same",
    }
```

```
    inputs = keras.Input(shape=(256, 256, channels))
    x = layers.Conv2D(64, 5, **conv_args)(inputs)
    x = layers.Conv2D(64, 3, **conv_args)(x)
    x = layers.Conv2D(32, 3, **conv_args)(x)
    x = layers.Conv2D(channels * (upscale_factor ** 2), 3, **conv_args)(x)
    outputs = tf.nn.depth_to_space(x, upscale_factor)
    return keras.Model(inputs, outputs)
def plot_results(img, prefix, title):
    """Plot the result with zoom-in area."""
    img_array = img_to_array(img)
    img_array = img_array.astype("float32") / 255.0
    # Create a new figure with a default 111 subplot.
    fig, ax = plt.subplots()
    im = ax.imshow(img_array[::-1], origin="lower")
    plt.title(title)
    # zoom-factor: 2.0, location: upper-left
    axins = zoomed_inset_axes(ax, 2, loc=2)
    axins.imshow(img_array[::-1], origin="lower")
    # Specify the limits.
    x1, x2, y1, y2 = 200, 300, 100, 200
    # Apply the x-limits.
    axins.set_xlim(x1, x2)
    # Apply the y-limits.
    axins.set_ylim(y1, y2)
    plt.yticks(visible=False)
    plt.xticks(visible=False)
    # Make the line.
    mark_inset(ax, axins, loc1=1, loc2=3, fc="none", ec="blue")
    plt.savefig(str(prefix) + "-" + title + ".png")
    plt.show()
def get_lowres_image(img, upscale_factor):
    """Return low-resolution image to use as model input."""
    return img.resize(
        (img.size[0] // upscale_factor, img.size[1] // upscale_factor),
        PIL.Image.BICUBIC,
    )
def upscale_image(model, img):
    """Predict the result based on input image and restore the image as RGB."""
    ycbcr = img.convert("YCbCr")
    y, cb, cr = ycbcr.split()
    y = img_to_array(y)
    y = y.astype("float32") / 255.0
    input = np.expand_dims(y, axis=0)
    out = model.predict(input)
```

```
    out_img_y = out[0]
    out_img_y *= 255.0
    # Restore the image in RGB color space.
    out_img_y = out_img_y.clip(0, 255)
    out_img_y = out_img_y.reshape((np.shape(out_img_y)[0], np.shape(out_img_y)[1]))
    out_img_y = PIL.Image.fromarray(np.uint8(out_img_y), mode="L")
    out_img_cb = cb.resize(out_img_y.size, PIL.Image.BICUBIC)
    out_img_cr = cr.resize(out_img_y.size, PIL.Image.BICUBIC)
    out_img = PIL.Image.merge("YCbCr", (out_img_y, out_img_cb, out_img_cr)).convert(
        "RGB"
    )
    return out_img
early_stopping_callback = keras.callbacks.EarlyStopping(monitor="loss", patience=10)
checkpoint_filepath = "weights"
model_checkpoint_callback = keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath + '/x8.h5',
    save_weights_only=True,
    monitor="loss",
    mode="min",
    save_best_only=True,
)
model = get_model(upscale_factor=upscale_factor, channels=1)
model.summary()
Mcallbacks = [early_stopping_callback, model_checkpoint_callback]
loss_fn = keras.losses.MeanSquaredError()
Moptimizer = keras.optimizers.Adam(learning_rate=0.001)
ep = 100
model.compile(
    optimizer=Moptimizer, loss=loss_fn, metrics = ['accuracy']
)
history = model.fit(
    train_ds, epochs=ep, callbacks=Mcallbacks, validation_data=valid_ds, verbose=1
)
history
model.save('thenewx4.h5')
```

4. RDN Model

```
import tensorflow as tf
from tensorflow.keras.initializers import RandomUniform
from tensorflow.keras.layers import concatenate, Input, Activation, Add, Conv2D, Lambda, UpSampling2D
from tensorflow.keras.models import Model
import os
from absl import app, flags, logging
from absl.flags import FLAGS
import cv2
import pathlib
import numpy as np
import tensorflow as tf
import functools
from tensorflow.keras.layers import Dense, Flatten, Input, Conv2D, LeakyReLU
from tensorflow.keras import Model
from absl import app, flags, logging
from absl.flags import FLAGS
import tqdm
import glob
import random
import matplotlib.pyplot as plt


def _bytes_feature(value):
    """Returns a bytes_list from a string / byte."""
    if isinstance(value, type(tf.constant(0))):
        value = value.numpy()
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))
def _float_feature(value):
    """Returns a float_list from a float / double."""
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))
def _int64_feature(value):
    """Returns an int64_list from a bool / enum / int / uint."""
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
def make_example_bin(img_name, hr_img_str, lr_img_str):
    # Create a dictionary with features that may be relevant (binary).
    feature = {'image/img_name': _bytes_feature(img_name),
               'image/hr_encoded': _bytes_feature(hr_img_str),
               'image/lr_encoded': _bytes_feature(lr_img_str)}
    return tf.train.Example(features=tf.train.Features(feature=feature))
def make_example(img_name, hr_img_path, lr_img_path):
    # Create a dictionary with features that may be relevant.
    feature = {'image/img_name': _bytes_feature(img_name),
               'image/hr_img_path': _bytes_feature(hr_img_path),
               'image/lr_img_path': _bytes_feature(lr_img_path)}
```

```
    return tf.train.Example(features=tf.train.Features(feature=feature))
def MakeTfrecord(hr_dataset_path,lr_dataset_path,output_path,is_binary):
    if not os.path.isdir(hr_dataset_path):
        logging.info('Please define valid dataset path.')
    else:
        logging.info('Loading {}'.format(hr_dataset_path))
    samples = []
    logging.info('Reading data list...')
    for hr_img_path in glob.glob(os.path.join(hr_dataset_path, '*.png')):
        img_name = os.path.basename(hr_img_path).replace('.png', '')
        lr_img_path = os.path.join(lr_dataset_path, img_name + 'x2.png')
        samples.append((img_name, hr_img_path, lr_img_path))
    random.shuffle(samples)
    if os.path.exists(output_path):
        logging.info('{:s} already exists. Exit...'.format(output_path))
        exit(1)
    logging.info('Writing {} sample to tfrecord file...'.format(len(samples)))
    with tf.io.TFRecordWriter(output_path) as writer:
        for img_name, hr_img_path, lr_img_path in tqdm.tqdm(samples):
            if is_binary:
                hr_img_str = open(hr_img_path, 'rb').read()
                lr_img_str = open(lr_img_path, 'rb').read()
                tf_example = make_example_bin(img_name=str.encode(img_name),
hr_img_str=hr_img_str,lr_img_str=lr_img_str)
            else:
                tf_example = make_example(img_name=str.encode(img_name),
                                hr_img_path=str.encode(hr_img_path),lr_img_path=str.encode(lr_img_path))
            writer.write(tf_example.SerializeToString())
# # TFrecord train data
train_hr_dataset_path='./data/DIV2K_train_HR/' # 'path to high resolution dataset'
train_lr_dataset_path='./data/DIV2K_train_LR_bicubic/X2/'#'path to low resolution dataset')
train_output_path='./data/DIV2K720_sub_bin.tfrecord'#'path to ouput tfrecord')
is_binary=True #'whether save images as binary files" or load them on the fly.')
MakeTfrecord(train_hr_dataset_path,train_lr_dataset_path,train_output_path,is_binary)
# # TFrecord validation data
train_hr_dataset_path='./data/DIV2K_valid_HR/' # 'path to high resolution dataset'
train_lr_dataset_path='./data/DIV2K_valid_LR_bicubic/X2/'#'path to low resolution dataset')
train_output_path='./data/DIV2K180_sub_bin.tfrecord'#'path to ouput tfrecord')
is_binary=True #'whether save images as binary files" or load them on the fly.')
MakeTfrecord(train_hr_dataset_path,train_lr_dataset_path,train_output_path,is_binary)
def load_dataset(pathTf,key, shuffle=True, buffer_size=32):
    """load dataset"""
    dataset_cfg = pathTf
    logging.info("load {} from {}".format(key, pathTf))
```

```
    dataset = load_tfrecord_dataset(
        tfrecord_name= pathTf, batch_size= 10 ,gt_size= 128 ,scale= 2, shuffle=shuffle , using_bin= True , using_flip= True,using_rot=True , buffer_size=buffer_size)
    return dataset
def load_tfrecord_dataset(tfrecord_name, batch_size, gt_size,scale, using_bin=False, using_flip=False, using_rot=False, shuffle=True, buffer_size=10240):
    """load dataset from tfrecord"""
    raw_dataset = tf.data.TFRecordDataset(tfrecord_name)
    raw_dataset = raw_dataset.repeat()
    if shuffle:
        raw_dataset = raw_dataset.shuffle(buffer_size=buffer_size)
    dataset = raw_dataset.map(_parse_tfrecord(gt_size, scale, using_bin, using_flip, using_rot),num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.batch(batch_size, drop_remainder=True)
    dataset = dataset.prefetch( buffer_size=tf.data.experimental.AUTOTUNE)
    return dataset
def _parse_tfrecord(gt_size, scale, using_bin, using_flip, using_rot):
    def parse_tfrecord(tfrecord):
        if using_bin:
            features = {
                'image/img_name': tf.io.FixedLenFeature([], tf.string),
                'image/hr_encoded': tf.io.FixedLenFeature([], tf.string),
                'image/lr_encoded': tf.io.FixedLenFeature([], tf.string)}
            x = tf.io.parse_single_example(tfrecord, features)
            lr_img = tf.image.decode_png(x['image/lr_encoded'], channels=3)
            hr_img = tf.image.decode_png(x['image/hr_encoded'], channels=3)
        else:
            features = {
                'image/img_name': tf.io.FixedLenFeature([], tf.string),
                'image/hr_img_path': tf.io.FixedLenFeature([], tf.string),
                'image/lr_img_path': tf.io.FixedLenFeature([], tf.string)}
            x = tf.io.parse_single_example(tfrecord, features)
            hr_image_encoded = tf.io.read_file(x['image/hr_img_path'])
            lr_image_encoded = tf.io.read_file(x['image/lr_img_path'])
            lr_img = tf.image.decode_png(lr_image_encoded, channels=3)
            hr_img = tf.image.decode_png(hr_image_encoded, channels=3)
        lr_img, hr_img = _transform_images(gt_size, scale, using_flip, using_rot)(lr_img, hr_img)
        return lr_img, hr_img
    return parse_tfrecord
def _transform_images(gt_size, scale, using_flip, using_rot):
    def transform_images(lr_img, hr_img):
        lr_img_shape = tf.shape(lr_img)
        hr_img_shape = tf.shape(hr_img)
        gt_shape = (gt_size, gt_size, tf.shape(hr_img)[-1])
```

```
lr_size = int(gt_size / scale)
lr_shape = (lr_size, lr_size, tf.shape(lr_img)[-1])
tf.Assert(
    tf.reduce_all(hr_img_shape >= gt_shape),
    ["Need hr_image.shape >= gt_size, got ", hr_img_shape, gt_shape])
tf.Assert(
    tf.reduce_all(hr_img_shape[:-1] == lr_img_shape[:-1] * scale),
    ["Need hr_image.shape == lr_image.shape * scale, got ",
     hr_img_shape[:-1], lr_img_shape[:-1] * scale])
tf.Assert(
    tf.reduce_all(hr_img_shape[-1] == lr_img_shape[-1]),
    ["Need hr_image.shape[-1] == lr_image.shape[-1]], got ",
     hr_img_shape[-1], lr_img_shape[-1]])
# randomly crop
limit = lr_img_shape - lr_shape + 1
offset = tf.random.uniform(tf.shape(lr_img_shape), dtype=tf.int32,
                           maxval=tf.int32.max) % limit
lr_img = tf.slice(lr_img, offset, lr_shape)
hr_img = tf.slice(hr_img, offset * scale, gt_shape)
# randomly left-right flip
if using_flip:
    flip_case = tf.random.uniform([1], 0, 2, dtype=tf.int32)
    def flip_func(): return (tf.image.flip_left_right(lr_img),
                             tf.image.flip_left_right(hr_img))
    lr_img, hr_img = tf.case(
        [(tf.equal(flip_case, 0), flip_func)],
        default=lambda: (lr_img, hr_img))
# randomly rotation
if using_rot:
    rot_case = tf.random.uniform([1], 0, 4, dtype=tf.int32)
    def rot90_func(): return (tf.image.rot90(lr_img, k=1),
                              tf.image.rot90(hr_img, k=1))
    def rot180_func(): return (tf.image.rot90(lr_img, k=2),
                               tf.image.rot90(hr_img, k=2))
    def rot270_func(): return (tf.image.rot90(lr_img, k=3),
                               tf.image.rot90(hr_img, k=3))
    lr_img, hr_img = tf.case(
        [(tf.equal(rot_case, 0), rot90_func),
         (tf.equal(rot_case, 1), rot180_func),
         (tf.equal(rot_case, 2), rot270_func)],
        default=lambda: (lr_img, hr_img))
# scale to [0, 1]
lr_img = lr_img / 255
hr_img = hr_img / 255
```

```
        return lr_img, hr_img
    return transform_images
train_dataset  =  load_dataset('./data/DIV2K720_sub_bin.tfrecord','train_dataset', shuffle=False)
valid_dataset  =  load_dataset('./data/DIV2K180_sub_bin.tfrecord','valid_dataset', shuffle=False)
def _pixel_shuffle(input_layer):
    """ PixelShuffle implementation of the upscaling layer. """
    initializer = RandomUniform( minval=-0.05, maxval=0.05, seed=None)
    x = Conv2D(
        3 * 2 ** 2,
        kernel_size=3,
        padding='same',
        name='UPN3',
        kernel_initializer=initializer,
    )(input_layer)
    return Lambda(
        lambda x: tf.nn.depth_to_space(x, block_size=2, data_format='NHWC'),
        name='PixelShuffle',
    )(x)
def _upsampling_block( input_layer):
    """ Upsampling block for old weights. """
    initializer = RandomUniform( minval=-0.05, maxval=0.05, seed=None)
    scale=2
    x = Conv2D(
        3 * scale ** 2,
        kernel_size=3,
        padding='same',
        name='UPN3',
        kernel_initializer=initializer,
    )(input_layer)
    return UpSampling2D(size=scale, name='UPsample')(x)
def _UPN(input_layer):
    upscaling='shuffle'
    """ Upscaling layers. With old weights use _upsampling_block instead of _pixel_shuffle. """
    initializer = RandomUniform( minval=-0.05, maxval=0.05, seed=None)
    x = Conv2D(
        64,
        kernel_size=5,
        strides=1,
        padding='same',
        name='UPN1',
        kernel_initializer=initializer,
    )(input_layer)
    x = Activation('relu', name='UPN1_Relu')(x)
```

```
    x = Conv2D(
        32, kernel_size=3, padding='same', name='UPN2', kernel_initializer=initializer
    )(x)
    x = Activation('relu', name='UPN2_Relu')(x)
    if upscaling == 'shuffle':
        return _pixel_shuffle(x)
    elif upscaling == 'ups':
        return _upsampling_block(x)
    else:
        raise ValueError('Invalid choice of upscaling layer.')
def _RDBs(input_layer):
        """RDBs blocks.
        Args:
            input_layer: input layer to the RDB blocks (e.g. the second convolutional layer F_0).
        Returns:
            concatenation of RDBs output feature maps with G0 feature maps.
        """
        initializer = RandomUniform( minval=-0.05, maxval=0.05, seed=None)
        rdb_concat = list()
        rdb_in = input_layer
        for d in range(1, 20+ 1):
            x = rdb_in
            for c in range(1, 6 + 1):
                F_dc = Conv2D(
                    64,
                    kernel_size=3,
                    padding='same',
                    kernel_initializer=initializer,
                    name='F_%d_%d' % (d, c),
                )(x)
                F_dc = Activation('relu', name='F_%d_%d_Relu' % (d, c))(F_dc)
                # concatenate input and output of ConvRelu block
                # x = [input_layer,F_11(input_layer),F_12([input_layer,F_11(input_layer)]), F_13..]
                x = concatenate([x, F_dc], axis=3, name='RDB_Concat_%d_%d' % (d, c))
            # 1x1 convolution (Local Feature Fusion)
            x = Conv2D(
                64, kernel_size=1, kernel_initializer=initializer, name='LFF_%d' % (d)
            )(x)
            # Local Residual Learning F_{i,LF} + F_{i-1}
            rdb_in = Add(name='LRL_%d' % (d))([x, rdb_in])
            rdb_concat.append(rdb_in)
        assert len(rdb_concat) == 20
        return concatenate(rdb_concat, axis=3, name='LRLs_Concat')
initializer = RandomUniform( minval=-0.05, maxval=0.05, seed=None)
```

```
LR_input = Input(shape=(None, None, 3), name='LR')
F_m1 = Conv2D( 64,kernel_size=3,padding='same',kernel_initializer=initializer,name='F_m1',)(LR_input)
F_0 = Conv2D(64,kernel_size=3,padding='same',kernel_initializer=initializer,name='F_0',)(F_m1)
FD = _RDBs(F_0)
# Global Feature Fusion
# 1x1 Conv of concat RDB layers -> G0 feature maps
GFF1 = Conv2D(64,kernel_size=1,padding='same',kernel_initializer=initializer,name='GFF_1',)(FD)
GFF2 = Conv2D(64,kernel_size=3,padding='same',kernel_initializer=initializer,name='GFF_2',)(GFF1)
# Global Residual Learning for Dense Features
FDF = Add(name='FDF')([GFF2, F_m1])
# Upscaling
FU = _UPN(FDF)
# Compose SR image
SR = Conv2D(3,kernel_size=3,padding='same',kernel_initializer=initializer,name='SR',)(FU)
Model_RDN=Model(inputs=LR_input, outputs=SR)
Model_RDN.summary()
Model_RDN.compile(optimizer = 'adam', loss = 'mean_squared_error',metrics='accuracy')
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=10,restore_best_weights=True)
history=Model_RDN.fit(train_dataset,epochs=100,batch_size=10,steps_per_epoch=72,validation_data=valid_dataset,
        validation_steps=18,callbacks=[callback])
Model_RDN.save('RDNt69.h5')
```

5. RFDN Model

```
import os
import math
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from model import *
from tensorflow import keras
from IPython.display import display
from tensorflow.keras import Input, Model
from tensorflow.keras.preprocessing import image_dataset_from_directory
from utils import *

root_dir= "DIV2K/DIV2K_train_HR"
resize = 400
upscale_factor = 2
input_size = resize // upscale_factor
batchsize = 15

train_ds = image_dataset_from_directory(
    root_dir,
    batch_size=batchsize,
    image_size=(resize, resize),
    validation_split=0.2,
    subset="training",
    seed=1337,
    label_mode=None,
    interpolation='bicubic'
)

valid_ds = image_dataset_from_directory(
    root_dir,
    batch_size=batchsize,
    image_size=(resize, resize),
    validation_split=0.2,
    subset="validation",
    seed=1337,
    label_mode=None,
    interpolation='bicubic'
)

# Scale from (0, 255) to (0, 1)
train_ds = train_ds.map(scaling)
valid_ds = valid_ds.map(scaling)
```

```
train_ds = train_ds.map(
        lambda x: (process_input(x, input_size, upscale_factor), process_target(x))
)

train_ds = train_ds.prefetch(buffer_size=32)

valid_ds = valid_ds.map(
        lambda x: (process_input(x, input_size, upscale_factor), process_target(x))
)

valid_ds = valid_ds.prefetch(buffer_size=32)
print(train_ds)

rfanet_x = RFDNNet()
#build keras model
x = Input(shape=(None, None, 3))
out = rfanet_x.main_model(x, upscale_factor)
model = Model(inputs=x, outputs=out)
model.summary()
early_stopping_callback = keras.callbacks.EarlyStopping(monitor="loss", patience=10)
checkpoint_filepath = "weights"
model_checkpoint_callback = keras.callbacks.ModelCheckpoint(
   checkpoint_filepath + '/x2.h5',
   monitor="loss",
   mode="min",
   save_best_only=True,
   period=1
)
#ESPCNCallback(),
MyCallbacks = [ early_stopping_callback, model_checkpoint_callback]
loss_fn = keras.losses.MeanSquaredError()
optimizer = keras.optimizers.Adam(learning_rate=0.001)
ep = 100
model.compile(
   optimizer=optimizer, loss=loss_fn, metrics = ['accuracy']
)
history = model.fit(
   train_ds, epochs=ep, callbacks=MyCallbacks, validation_data=valid_ds, verbose=1
)
```

6. Autoencoder (Subpixel)

```
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout
from tensorflow.keras.layers import Conv2DTranspose, UpSampling2D, add
from skimage.transform import resize, rescale
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
from IPython.display import display
from tensorflow.keras import Input, Model
import matplotlib.pyplot as plt
from scipy import ndimage, misc
from matplotlib import pyplot
import tensorflow as tf
from tensorflow import keras
import numpy as np
np.random.seed(0)
import re
import cv2
import os
import math
import numpy as np

root_dir = os.path.join("data/", "DIV2K_train_HR/")
resize_size = 512
upscale_factor = 2
input_size = resize_size // upscale_factor
batch_size = 10

train_ds = image_dataset_from_directory(
    root_dir,
    batch_size=batch_size,
    image_size=(resize_size, resize_size),
    validation_split=0.2,
    subset="training",
    seed=1337,
    label_mode=None,
    interpolation='bicubic'
)
valid_ds = image_dataset_from_directory(
    root_dir,
    batch_size=batch_size,
    image_size=(resize_size, resize_size),
    validation_split=0.2,
    subset="validation",
```

```
        seed=1337,
        label_mode=None,
        interpolation='bicubic'
)
def scaling(input_image):
    input_image = input_image / 255.0
    return input_image
train_ds = train_ds.map(scaling)
valid_ds = valid_ds.map(scaling)
def process_input(input_x, input_size, upscale_factor):
    return tf.image.resize(input_x, [input_size, input_size], method="area")
def process_target(input_x):
    return input_x
train_ds = train_ds.map(
          lambda x: (process_input(x, input_size, upscale_factor), process_target(x))
)
train_ds = train_ds.prefetch(buffer_size=32)
valid_ds = valid_ds.map(
         lambda x: (process_input(x, input_size, upscale_factor), process_target(x))
)
valid_ds = valid_ds.prefetch(buffer_size=32)
scale_factor=2
input_img = Input(shape=(None, None, 3))
l1 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(input_img)

l2 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l1)

l3 = MaxPooling2D(padding = 'same')(l2)
l3 = Dropout(0.3)(l3)

l4 = Conv2D(128, (3, 3),  padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l3)

l5 = Conv2D(128, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l4)

l6 = MaxPooling2D(padding = 'same')(l5) #2

l7 = Conv2D(256, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l6)
l8 = Conv2DTranspose(256, (2,2), strides=(2,2))(l7)
```

```
l9 = Conv2D(128, (3, 3), padding = 'same', activation = 'relu',
           activity_regularizer = regularizers.l1(10e-10))(l8)

l10 = Conv2D(128, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l9)     # 2  /   2

l11 = add([l5, l10])
l12 = Conv2DTranspose(128, (2,2), strides=(2,2))(l11)

l13 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l12)

l14 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(l13)

l15 = add([l14, l2])
postUpsampling= Conv2DTranspose(64, (2,2), strides=(2,2))(l15)
X_up = Conv2D(64 * (scale_factor ** 2), 3, padding='same')(l15)
postUpsampling = tf.nn.depth_to_space(X_up, scale_factor)
decoded = Conv2D(3, (3, 3), padding = 'same',
                activation = 'relu', activity_regularizer = regularizers.l1(10e-10))(postUpsampling)
autoencoder = Model(input_img, decoded)
autoencoder.summary()
autoencoder.compile(optimizer = 'adam', loss = 'mean_squared_error',metrics='accuracy')
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=10,restore_best_weights=True)
history=autoencoder.fit(train_ds,epochs=80,batch_size=10,shuffle=True, callbacks=[callback],validation_data=valid_ds)
autoencoder.save('Autoencoder_subpixel_ep_10batch_model.h5')
```

7. Autoencoder (Deconvolution)

```
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout
from tensorflow.keras.layers import Conv2DTranspose, UpSampling2D, add
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
import matplotlib.pyplot as plt
from scipy import ndimage, misc
from matplotlib import pyplot
import tensorflow as tf
import numpy as np
np.random.seed(0)
import re
import os
import cv2

def lowResolution(path,sizeHigh,size):
    names = sorted(os.listdir(path))
    allLowImages = []
    for name in names:
        fpath = path + name
        img = cv2.imread(fpath, cv2.IMREAD_COLOR)
        image=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        image = cv2.resize(image,(int(sizeHigh),int(sizeHigh)),cv2.INTER_CUBIC)
        image = cv2.resize(image,(int(size),int(size)),cv2.INTER_CUBIC)
        lowimage=image[:, :, :].astype(float) / 255
        allLowImages.append(lowimage)
    allLowImages = np.array(allLowImages)
    return allLowImages
def highResolution(path,size):
    names = sorted(os.listdir(path))
    allHighimages = []
    for name in names:
        fpath = path + name
        img = cv2.imread(fpath, cv2.IMREAD_COLOR)
        image=cv2.cvtColor(img, cv2.COLOR_BGR2RGB) #change bgr to rgb
        highimage = cv2.resize(image,(size,size),cv2.INTER_CUBIC)
        highimage=highimage[:, :, :].astype(float) / 255
        allHighimages.append(highimage)
    allHighimages = np.array(allHighimages)
    return allHighimages
input_img = Input(shape=(None, None, 3))
l1 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
            activity_regularizer = regularizers.l1(10e-10))(input_img)
```

```
l2 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
        activity_regularizer = regularizers.l1(10e-10))(l1)
l3 = MaxPooling2D(padding = 'same')(l2)
l3 = Dropout(0.3)(l3)
l4 = Conv2D(128, (3, 3),  padding = 'same', activation = 'relu',
        activity_regularizer = regularizers.l1(10e-10))(l3)
l5 = Conv2D(128, (3, 3), padding = 'same', activation = 'relu',
        activity_regularizer = regularizers.l1(10e-10))(l4)
l6 = MaxPooling2D(padding = 'same')(l5) #2
l7 = Conv2D(256, (3, 3), padding = 'same', activation = 'relu',
        activity_regularizer = regularizers.l1(10e-10))(l6)
l8 = Conv2DTranspose(256, (2,2), strides=(2,2))(l7)
l9 = Conv2D(128, (3, 3), padding = 'same', activation = 'relu',
        activity_regularizer = regularizers.l1(10e-10))(l8)
l10 = Conv2D(128, (3, 3), padding = 'same', activation = 'relu',
         activity_regularizer = regularizers.l1(10e-10))(l9)     # 2  /   2
l11 = add([l5, l10])
l12 = Conv2DTranspose(128, (2,2), strides=(2,2))(l11)
l13 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
         activity_regularizer = regularizers.l1(10e-10))(l12)
l14 = Conv2D(64, (3, 3), padding = 'same', activation = 'relu',
         activity_regularizer = regularizers.l1(10e-10))(l13)
l15 = add([l14, l2])
postUpsampling= Conv2DTranspose(64, (2,2), strides=(2,2))(l15)
decoded = Conv2D(3, (3, 3), padding = 'same',
            activation = 'relu', activity_regularizer = regularizers.l1(10e-10))(postUpsampling)
autoencoder = Model(input_img, decoded)
autoencoder.summary()
autoencoder.compile(optimizer = 'adam', loss = 'mean_squared_error',metrics='accuracy')
x_train_low=lowResolution("./data/DIV2K_train_HR/",512,256)
x_train_high=highResolution("./data/DIV2K_train_HR/",512)
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=10,restore_best_weights=True)
history=autoencoder.fit(x_train_low,x_train_high,epochs=120,batch_size=10,shuffle=True,
callbacks=[callback],validation_split=0.2)
autoencoder.save('Autencoder_Deconv_0ep120.h5')
```

Testing

1. Final Measurements

```
pip install xlsxwriter
import xlsxwriter
import math
import cv2
import math
import numpy as np
import os
import cv2
from skimage.metrics import structural_similarity as ssim


def psnr(img1, img2):
   mse = np.mean((img1 - img2) ** 2)
   if mse == 0:
      return 100
   PIXEL_MAX = 255.0
   return 20 * math.log10(PIXEL_MAX / math.sqrt(mse))
# Python program to get average of a list
def Average(lst):
return sum(lst) / len(lst)
Model_Name = input("enter your Model Name:")            #file name
destination_path = '/content/drive/MyDrive/graduation project/psnr ssim/project Dataset/'     #path of the destination don't forget exstar / after the path
workbook = xlsxwriter.Workbook(destination_path+str(Model_Name)+'.xlsx')          #create xlsx sheet
worksheet = workbook.add_worksheet()
worksheet.write('A1', 'Name')                          #add Name to column1
worksheet.write('B1', 'Shape')
worksheet.write('C1', 'ssim')
worksheet.write('D1', 'psnr')
orignal = []
orignalpath = "/content/drive/MyDrive/graduation project/Datasets/our dataset/Dataset/"  #path of the dataset    don't forget exstar / after the path
for filename in os.listdir(orignalpath):
     pathimage = orignalpath+filename
     orignal.append(pathimage)
orignal = np.array(orignal)
orignal.sort()
print(orignal)
subimage = []
subimagepath = "/content/drive/MyDrive/graduation project/All models/our models/ESRGAN(Adham)/project dataset/"  #path of the model   don't forget exstar / after the path
for filename in os.listdir(subimagepath):
```

```
    pathimage = subimagepath+filename
    subimage.append(pathimage)
subimage = np.array(subimage)
subimage.sort()
print(subimage)

i =0
avgpsnr=[]
avgssim=[]
for x in orignal:
 temp=os.path.basename(os.path.normpath(orignal[i]))
 print(temp)
 worksheet.write('A'+str(i+2),str(temp))  #image name first column
 print(orignal[i])
 print(subimage[i])
 image1= cv2. imread(orignal[i], 1)         #read dataset image from array
 image2= cv2. imread(subimage[i])           #read subimage from array

 print(image1.shape)                    #making sure 2 images matches in shape
 print(image2.shape)
 worksheet.write('B'+str(i+2), str(image1.shape)) #image shape second column

 SSIM1 = ssim( image1,image2 , multichannel =True)   #calculate the SSim for the 2 images
 print(f'SSIM for {x}'"  is",SSIM1)                 # print ssim
 avgssim.append(SSIM1)
 worksheet.write('C'+str(i+2), SSIM1)              #puting ssim value in third column
 PSNR1 = psnr(image1, image2)                       #calculate the psnr for the 2 images
 print(f'psnr for {x}'"is",PSNR1)                   # print psnr
 avgpsnr.append(PSNR1)
 worksheet.write('D'+str(i+2), PSNR1)                #puting psnr value in forth column

 i+=1
worksheet.write('C'+str(len(orignal)+2), Average(avgssim))   #avg ssim
worksheet.write('D'+str(len(orignal)+2), Average(avgpsnr))    #avg psnr
worksheet.write('A'+str(len(orignal)+2), "AVG")
workbook.close()
```

2. Comparison Figures

```
import cv2
from matplotlib import pyplot as plt
from PIL import Image
import numpy as np

from google.colab import drive
drive.mount('/content/drive')

fig = plt.figure(figsize=(450, 450))        # create figure
# setting values to rows and column variables
rows = 3
columns = 4
# Setting the points for cropped image
left = 795
top = 765
right = 978
bottom = 951

Orignal = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/bicubic interpolation/project dataset/('downNature2cub', '.jpg').png")
FSRCNN = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/FSRCNN(abdo)/our dataset/x2/fsrcnn_downNature2cub.png")
ESPCN = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/ESPCN(Tarek)/Bicubic(project dataset)/half/downNature2cubESPCN.png")
RDN = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/RDN(saad)/x2_RDN/Ourx2/RDN_downNature2cub (1).png")
RFDN = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/RFDN(Jannah)/Dataset/X2 /cubRFDNNature2.png")
Deconvolution = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/Autoencoder(amr)/version 2/our_dataset/deonv4x/AutoencoderquarterNature2.png")
SRCNN = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/Srcnn(abdo)/our dataset/x2/srcnn_downNature2cub.png")
SUBPIXEL = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/Autoencoder(amr)/last version/Subpixel/dataset bicubic subpixel x2/Nature2AutoencoderSubpixel.png")
Gan = Image.open(r"/content/drive/MyDrive/graduation project/All models/our models/ESRGAN(Adham)/project dataset/ganquarterNature2.png")
adobe = Image.open(r"/content/drive/MyDrive/graduation project/All models/test models/Photoshop/Project_dataset/Nature2.jpg")
web = Image.open(r"/content/drive/MyDrive/graduation project/All models/test models/Let's Enhance/Project_dataset/downNature2cub_auto_x2.jpg")
Topaz = Image.open(r"/content/drive/MyDrive/graduation project/All models/test models/Topaz AI/topaz bicubic/Nature2.jpg")
```

```
Models_names = [Orignal, SRCNN, FSRCNN, ESPCN, Deconvolution, SUBPIXEL, RDN, RFDN , Gan, web, adobe, Topaz]
Models_names2 = ['Bicubic', 'SRCNN', 'FSRCNN', 'ESPCN','AE Deconv','AE Subpixel','RDN','RFDN','ESRGAN","Let'senhance","Photoshop","Topaz"]
ASSESMENT = ['(32.489/0.777)', '(32.683/0.798)', '(32.008/0.775)', '(31.808/0.809)','(32.622/0.778)','(32.631/0.782)','(32.389/0.802)','(32.487/0.802)','(31.222/0.603)',"(31.687/0.735)","32.466/0.807)","(32.132/0.746)"]
# Bicubic SRCNN FSRCNN ESPCN Deconv Subpixel RDN RFDN ESRGAN Let'senhance Photoshop Topaz

x = 0
rows_counter = 1
columns_counter = 1
low = 0.05
high= 0.97
for rows_counter in range(rows):
  high-=0.32
  low = 0.05
  for columns_counter in range(columns):
    read_image = Models_names[x]
    croped_image =read_image.crop((left, top, right, bottom))
    ax = plt.subplot(rows, columns, x+1)
    plt.imshow(croped_image)
    plt.axis("off")
    plt.margins(1)
    plt.title(str(Models_names2[x]),)
    plt.figtext(low,high, str(ASSESMENT[x]), fontsize = 10)#start 0.66 for y,x=0.29 for second+25
    low+=0.25
    #plt.xlabel(str(ASSESMENT[x]))
    plt.subplots_adjust(left=0.1,
                right=0.9,
                top=0.9,
                wspace=0.5,
                hspace=0.5)
    x+=1
    columns_counter+=1
  rows_counter+=1
plt.tight_layout()
plt.savefig('Nature2.png')
```