SUPER RESOLUTION IMAGE DENOISER NETWORK TOWARD REAL WORLD IMAGE NOISE REMOVAL

BY:

P. Mohit Harsh - 21951A04A3
A. Vivek Teja Vardhan - 21951A04Q8
M. Vivek Vishnu Vardhan - 21951A04Q9
J. Manoj Kumar - 21951A04R4

SUPER RESOLUTION IMAGE DENOISER NETWORK TOWARD REAL WORLD IMAGE NOISE REMOVAL

A Project Report
submitted in partial fulfilment of the
requirements for the award of the degree of

Bachelor of Technology
in
Electronics and Communication Engineering

By

P. MOHIT HARSH – 21951A04A3
A. VIVEK TEJA VARDHAN – 21951A04Q8
M. VIVEK VISHNU VARDHAN – 21951A04Q9
J. MANOJ KUMAR – 21951A04R4

Under The Guidance of
Dr V. Padmanabha Reddy,

Professor

Department of Electronics and Communication Engineering

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad – 500 043, Telangana

May, 2025

DECLARATION

We certify that,

a. The work contained in this report is original and has been done by us under the guidance of my supervisor(s).

b. The work has not been submitted to any other Institute for any degree or diploma.

c. We have followed the guidelines provided by the Institute in preparing the report.

d. We have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever we have used materials (data, theoretical analysis, figures, and text) from other sources, we have given due credit to them by citing them in the text of the report and giving their details in the references. Further, we have taken permission from the copyright owners of the sources, whenever necessary.

Place: **Signature of Student(s)**

Date: **Roll No: 21951A04A3**

21951A04Q8

21951A04Q9

21951A04R4

CERTIFICATE

This is to certify that the project report entitled **SUPER RESOLUTION IMAGE DENOISER NETWORK: TOWARD REAL LIFE IMAGE NOISE REMOVAL** submitted by **P. Mohit Harsh, A. Vivek Teja Vardhan, M. Vivek Vishnu Vardhan** and **J. Manoj Kumar** to the Institute of Aeronautical Engineering, Hyderabad in partial fulfillment of the requirements for the award of the Degree Bachelor of Technology in Department of Electronics and Communication Engineering is a bonafide record of work carried out by them under our guidance and supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute for the award of any Degree.

Supervisor	**Head of Department**
Dr V. Padmanabha Reddy	Dr P. Munaswamy
Professor	Professor & Head

Date:

APPROVAL SHEET

This project report entitled **SUPER RESOLUTION IMAGE DENOISER NETWORK: TOWARD REAL LIFE IMAGE NOISE REMOVAL** by **P. Mohit Harsh, A. Vivek Teja Vardhan, M. Vivek Vishnu Vardhan** and **J. Manoj Kumar** is approved for the award of the Degree Bachelor of Technology in the Department of Electronics and Communication Engineering.

Examiner(s)

Supervisor

Dr V. Padmanabha Reddy

Professor, ECE

Principal

Dr. L V Narasimha Prasad

Date:

Place:

ACKNOWLEDGEMENT

We would like to take this opportunity to thank Principal, **Dr. L.V. Narasimha Prasad**, and the Management for providing us with the necessary facilities and opportunities that helped us to complete our project work successfully. We would like to extend our gratitude to the teaching and non-teaching faculty members of the Department of Electronics and Communication Engineering for their support, guidance, and the facilities provided during my project work.

We want to express our heartfelt gratitude to **Dr P. Munaswamy**, Head of the Department, for helping us discover our hidden talents, supporting our professional development, boosting our confidence, nurturing our commitment and organization skills, and guiding me towards success.

We are deeply grateful to our project guide, **Dr. V. Padmanabha Reddy**, from the Department of Electronics and Communication Engineering, for his invaluable guidance, constant support, and inspiration which have sustained us to accomplish our work successfully. Without his valuable inputs and expertise, this project would not have been possible.

We would like to offer our gratitude to our Parents for their unwavering support, encouragement, keen appreciation, and active interest in our academic achievements. Their constant presence and support have been a great source of motivation for us throughout our academic journey. This project has been a significant milestone in our career development, and we intend to use the skills and knowledge we gained in the best possible way. We look forward to continuing my cooperation with all of you in the future.

With Gratitude,

P. Mohit Harsh (21951A04A3)

A. Vivek Teja Vardhan (21951A04Q9)

M. Vivek Vishnu Vardhan (21951A04Q8)

J. Manoj Kumar (21951A04R4)

ABSTRACT

In recent years, deep learning has significantly advanced image denoising techniques, with methods such as transformers and GANs pushing the boundaries of performance. However, these state-of-the- art models are typically resource intensive, requiring vast computational power, large datasets, and extensive training time, which also hampers real-time inference speed. To address these challenges, we propose the Super Resolution Image Denoiser Network (SRIDNet), a lightweight yet highly effective model designed to achieve near state-of-the-art results with a fraction of the data and computational resources. SRIDNet significantly reduces the model size while maintaining competitive performance in image denoising and super-resolution tasks. We evaluated our model on multiple datasets, where it demonstrated high PSNR scores, and the fastest inference times compared to existing models. Our results confirm that SRIDNet offers an efficient solution for real-time image denoising, making it a practical alternative to more cumbersome models without compromising on quality.

Keywords—Convolution Neural Network (CNN), Adversarial Neural Network (GAN), SRIDNet, PSNR, Transformers.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Definition
CNN	Convolutional neural network
CBDNet	Convolutional Blind denoising network
SRIDNet	Super Resolution Image Denoising Network
RIDNet	Residual Image Denoising Network
PSNR	Peak Signal to Noise Ratio
SSIM	Structural Similarity Index Measure

CHAPTER-1

INTRODUCTION

1.1 INTRODUCTION

Image denoising is an essential preprocessing technique in computer vision and image analysis, focused on enhancing corrupted images by minimizing noise while maintaining crucial structural features. Traditional methods like Gaussian filters and wavelet transforms have been widely used for this purpose but often struggle to handle the complex and varied noise patterns found in real-world images.

With the advent of deep learning, image denoising has seen significant progress, offering robust, data-driven approaches capable of learning complex noise characteristics and delivering high-quality restorations. Convolutional Neural Networks (CNNs) have emerged as a leading framework for this task, effectively capturing local spatial information through layered convolutional operations. By training on extensive datasets of noisy and corresponding clean images, CNN-based models have surpassed the performance of classical techniques.

Moreover, newer models, including Generative Adversarial Networks (GANs) and Transformer-based architectures, have further advanced the field by modeling global dependencies and producing more realistic outputs. However, despite their impressive achievements in controlled experimental settings, these cutting-edge models often demand significant computational power, large memory capacities, and substantial training data. Such requirements limit their deployment in real-world scenarios, especially in environments with limited hardware and data resources.

To address these challenges, we propose the Super Resolution Image Denoiser Network (SRIDNet), a novel approach that strikes a balance between denoising performance and computational efficiency. SRIDNet is designed to deliver competitive results with significantly lower hardware demands and reduced model complexity, making it suitable for deployment in practical settings.

1.2 DEEP NEURAL NETWORKS

A "**Deep Neural Network"** [2] **(DNN)** is a type of "Artificial Neural Network" [2] (ANN) characterized by the presence of multiple hidden layers between its input and output layers. Like traditional shallow networks, DNNs are capable of modeling intricate non-linear patterns. The core purpose of a neural network is to process a set of inputs through a sequence of computations and generate outputs that can address real-world challenges, such as classification tasks. In this discussion, we focus exclusively on **feedforward neural networks**, where data moves sequentially from input to output without looping back.

In deep networks, information flows through a series of layers, and these models are widely adopted in supervised learning and reinforcement learning scenarios. A deep learning model typically has many — sometimes even thousands of — hidden layers, most of which apply non-linear transformations. Thanks to this depth, deep learning models often achieve superior performance compared to traditional machine learning models. **Gradient Descent** is the most commonly used optimization technique to train these models by minimizing the loss function.

An example of an application includes using large datasets like **ImageNet**, which contains millions of labeled images, to classify objects (e.g., distinguishing between cats and dogs). Deep learning has also expanded to handling dynamic data like video streams, time series, and even textual information.

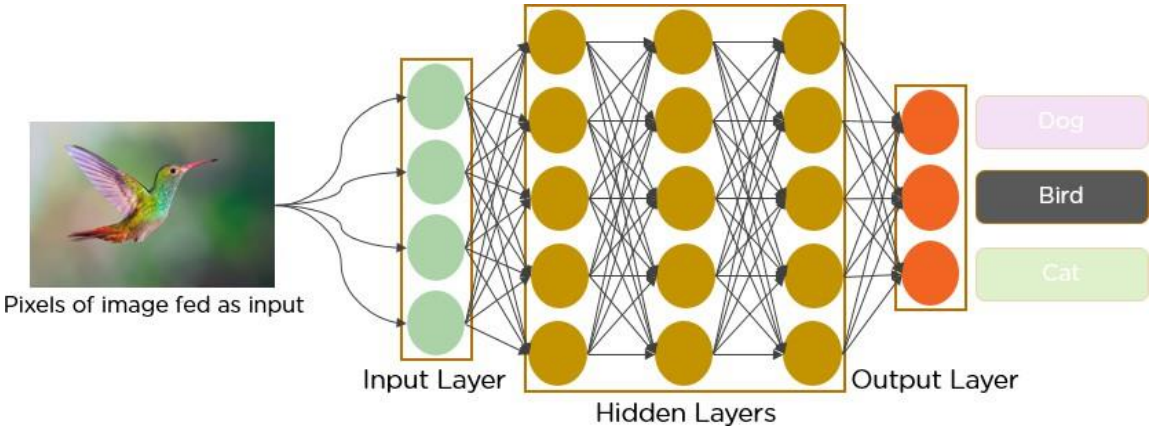

Figure 1. Deep convolutional Neural Network

Training plays a vital role in deep learning. A key algorithm used during training is **Backpropagation**, which adjusts weights and biases to reduce the error between predicted and actual outputs. Deep learning focuses on building large neural networks capable of transforming complex input-output relationships. Examples include mapping images to the names of individuals (as seen on social media) or generating descriptive captions for photos.

At a fundamental level, neural networks act as functions that transform inputs (x1, x2, x3, etc.) into outputs (z1, z2, z3, etc.). In shallow networks, this transformation occurs in two stages, while in deep networks, it happens through many intermediate layers. The parameters such as weights (often denoted by **w** and **v**) and biases evolve as the data progresses through each layer.

Deep learning finds its strongest applications in **supervised learning**, where we have a labeled dataset with known outputs. The **MNIST** dataset of handwritten digits is a classic example. Using frameworks like **Keras**, one can build and train **Convolutional Neural Networks** [19] to classify these digits effectively.

The neural network's activation, or "firing," at each node produces a numerical score that influences decision-making. For instance, in medical diagnosis, features like height, weight, temperature, and blood pressure are inputted, and a high activation score might predict illness, while a low score suggests health.

During **forward propagation**, input data passes through successive hidden layers, with each layer refining the information until the final output is generated. This sequential flow is strictly in the forward direction.

The **Credit Assignment Path (CAP)** refers to the chain of transformations linking inputs to outputs, illustrating how causality flows through the network. In a feedforward neural network, the CAP depth is calculated as the number of hidden layers plus one (for the output layer). In contrast, **Recurrent Neural Networks (RNNs)** can have theoretically infinite CAP depths since information can circulate through the same layers multiple times.

1.3 CONVOLUTIONAL NEURAL NETWORKS

A “Convolutional Neural Network” [19] (CNN) is a specialized architecture in Deep Learning, widely applied in the field of Computer Vision. Computer Vision focuses on enabling machines to interpret and understand visual information from the world around them.

In the realm of Machine Learning, “Artificial Neural Networks” [2] (ANNs) have shown excellent performance across different types of data, including images, audio, and text. Various neural network architectures are tailored for specific tasks — for instance, “Recurrent Neural Networks” [20] (RNNs), particularly “Long Short-Term Memory networks” [20] (LSTMs), are commonly used for tasks like predicting sequences in text. Meanwhile, CNNs are predominantly used for image classification. In this article, we'll explore how to construct a fundamental building block of a CNN.

Typically, a traditional Neural Network consists of three main types of layers:

1. **Input Layer**: This is the first layer where data is introduced into the model. The number of neurons in the input layer corresponds to the number of features in the dataset — in the case of images, this would be the total number of pixels [19].
2. **Hidden Layers**: After the input layer, data flows into one or more hidden layers. These layers can vary in number depending on the complexity of the model and the dataset. Each hidden layer typically contains a larger number of neurons compared to the input layer. In these layers, outputs are generated by multiplying the input with trainable weights, adding biases, and applying an activation function that introduces non-linearity into the model [17].
3. **Output Layer**: The final hidden layer connects to the output layer. Here, functions like softmax or sigmoid are used to transform the outputs into probability scores corresponding to different classes [17].

As the data moves through the layers during what is called the “**feedforward”** [17] process, the model generates outputs. An error or loss function — such as cross-entropy loss or mean squared error — is then used to measure how far the model's predictions are from the actual results. To improve accuracy, the model undergoes “**backpropagation”** [18], a process where derivatives are calculated and used to adjust

the weights and biases, minimizing the loss over time.

Convolution Neural Network

A “Convolutional Neural Network” [20] (CNN) is an advanced form of an artificial neural network (ANN) designed mainly for extracting features from data organized in a grid-like structure. It is especially effective for handling visual data such as images and videos, where recognizing patterns is crucial.

CNN architecture

A “Convolutional Neural Network” [20] (CNN) is composed of several types of layers, namely the input layer, convolutional layer, pooling layer, and fully connected layers. The convolutional layer is responsible for applying filters to an input image to extract important features. The pooling layer then reduces the spatial dimensions of the data to minimize computational load, while the fully connected layers perform the final prediction. Through the process of backpropagation and gradient descent, the network learns the best set of filters during training.

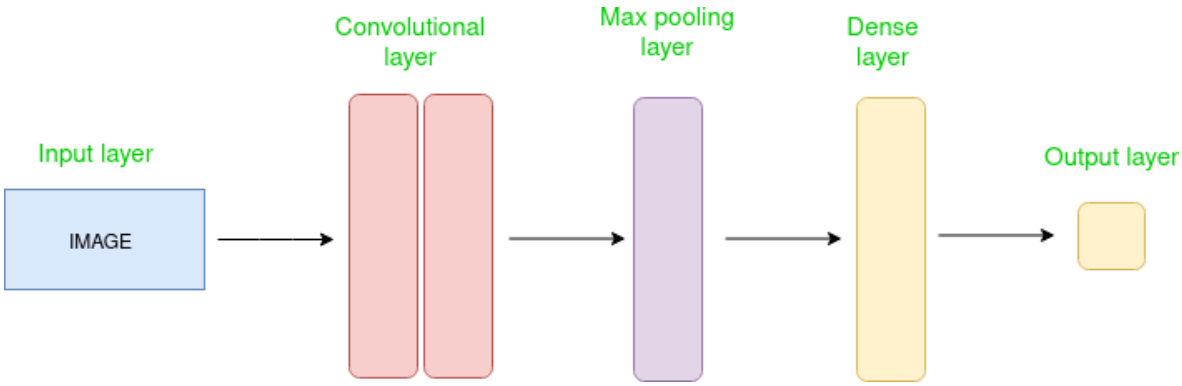

Figure 2. CNN Architecture

The Convolutional layer [19] applies filters to the input image to extract features, the Pooling layer down samples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

How Convolutional Layers works

“Convolutional Neural Networks” [19] (CNNs or ConvNets) are a type of neural network where parameters are shared across different parts of the model. Consider an image represented as a three-dimensional structure: the width and height correspond to

the dimensions of the image, and the depth represents the color channels (typically Red, Green, and Blue).

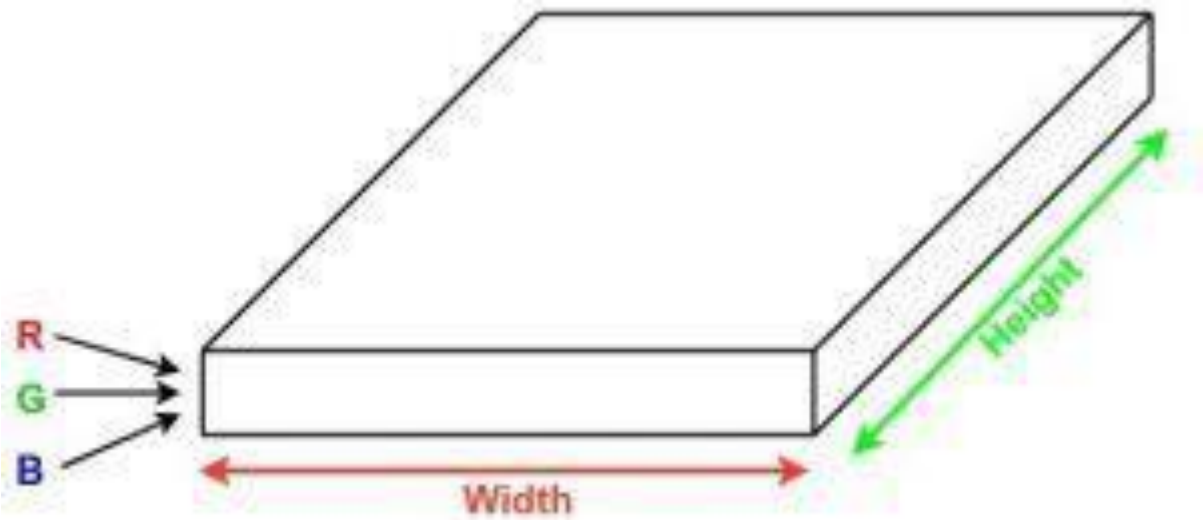

Figure 3. Convolutional Layer

Now, imagine selecting a small section (patch) of the image and passing it through a tiny neural network — often referred to as a filter or kernel — which produces a certain number of outputs, say K, arranged vertically. By sliding this small network across the entire image (a process known as convolution), a new set of outputs is generated, resulting in an output volume with a reduced width and height but an increased depth. Unlike the original three channels (RGB), this new volume will have more channels corresponding to the number of filters applied. If instead the filter covered the entire image, it would function like a traditional fully connected network,

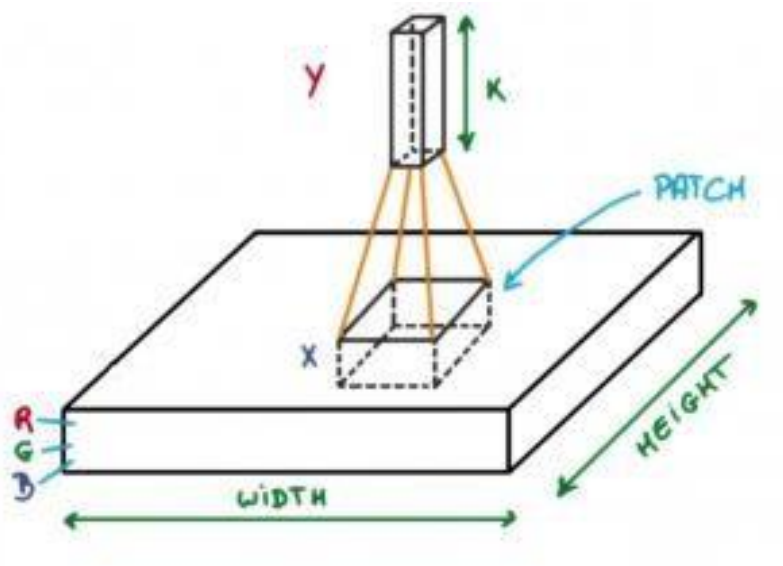

Figure 4. Convolution Patch

requiring significantly more weights. Using smaller patches drastically reduces the number of parameters.

Let’s explore the mathematics behind the convolution operation:

- A convolutional layer contains several learnable filters (kernels), each with small width and height but with the same depth as the input (e.g., 3 channels for RGB images).
- For instance, if the input image has dimensions 34×34×3, the filters might have dimensions like 3×3×3, 5×5×3, or 7×7×3 — small compared to the overall image size.
- During the forward pass, each filter moves across the input volume in small steps, where each step size is called the stride (commonly 1, 2, 3, or more for larger images). At each step, a dot product is computed between the filter weights and the corresponding input patch.
- After sliding across the whole input, each filter produces a 2D activation map. Stacking the outputs from all filters results in an output volume, where the depth equals the number of filters used. The network automatically learns these filters during training.

Layers used to build ConvNets

A complete “Convolutional Neural Network” [19] (CNN) architecture, often referred to as "convnets," is built as a series of layers, where each layer transforms its input volume into an output volume using a differentiable function.

Let’s walk through an example using a CNN on an image of size 32 × 32 × 3:

- **Input Layer**: This is the starting point where data is fed into the network. Typically, in CNNs, the input consists of an image or a set of images. Here, the input holds an image with width 32, height 32, and 3 color channels (depth).
- **Convolutional Layer**: This layer is responsible for extracting features from the input. It uses a collection of learnable filters, often called kernels, typically sized 2×2, 3×3, or 5×5. These filters move across the image, computing a dot product between their weights and the corresponding sections of the input image. The result is a set of feature maps. For instance, if 12 filters are used, the output volume would have dimensions 32 × 32 × 12.

- **Activation Layer**: Activation functions are applied here to introduce non-linearity into the network, helping it learn more complex patterns. Each element of the output from the convolutional layer is passed through an activation function such as ReLU (max(0, x)), Tanh, or Leaky ReLU. The size of the volume remains the same, so the output would still be 32 × 32 × 12.
- **Pooling Layer**: Pooling layers are added at intervals in the CNN to reduce the dimensions of the data, thus lowering computation requirements, saving memory, and mitigating overfitting. Common pooling techniques include max pooling and average pooling. For example, applying a 2×2 max-pooling operation with a stride of 2 would reduce the output volume to 16 × 16 × 12.

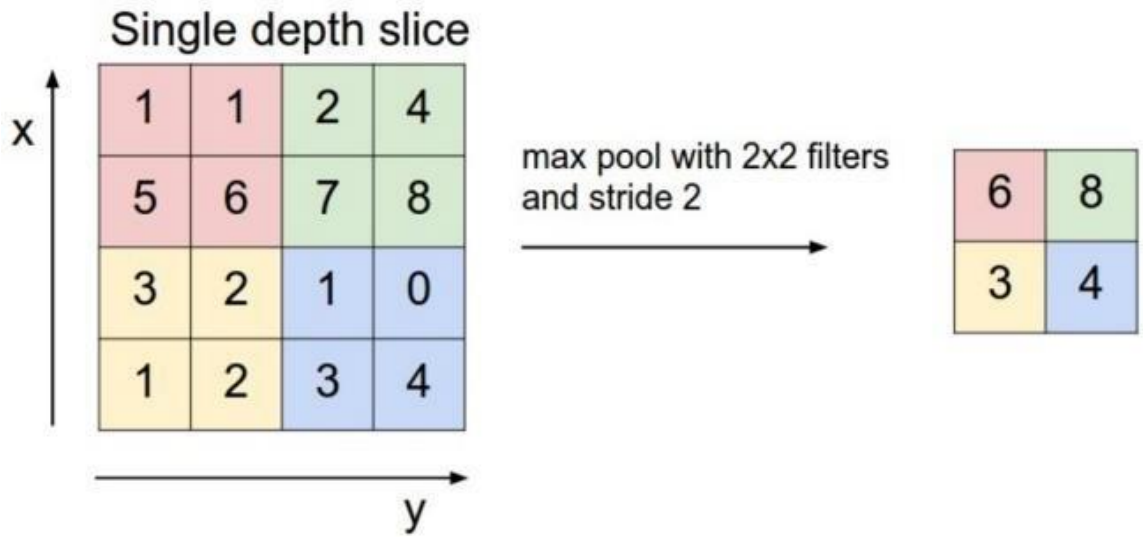

Figure 5. Max Pooling

- **Flattening:** After the convolution and pooling operations, the feature maps are converted into a one-dimensional array. This flattened output is then fed into a fully connected layer to perform tasks like classification or regression.
- **Fully Connected Layers:** These layers receive input from the flattened layer and are responsible for making the final predictions, whether for classification or regression purposes.

- **Output Layer:** After the features have been extracted and processed through multiple convolutional and fully connected layers, the final output layer is responsible for

reconstructing the clean, denoised version of the input image. Unlike classification tasks where activation functions like sigmoid or softmax are used to convert outputs into probability scores for each class, image denoising typically requires a continuous output that matches the pixel intensity values of the original clean image.

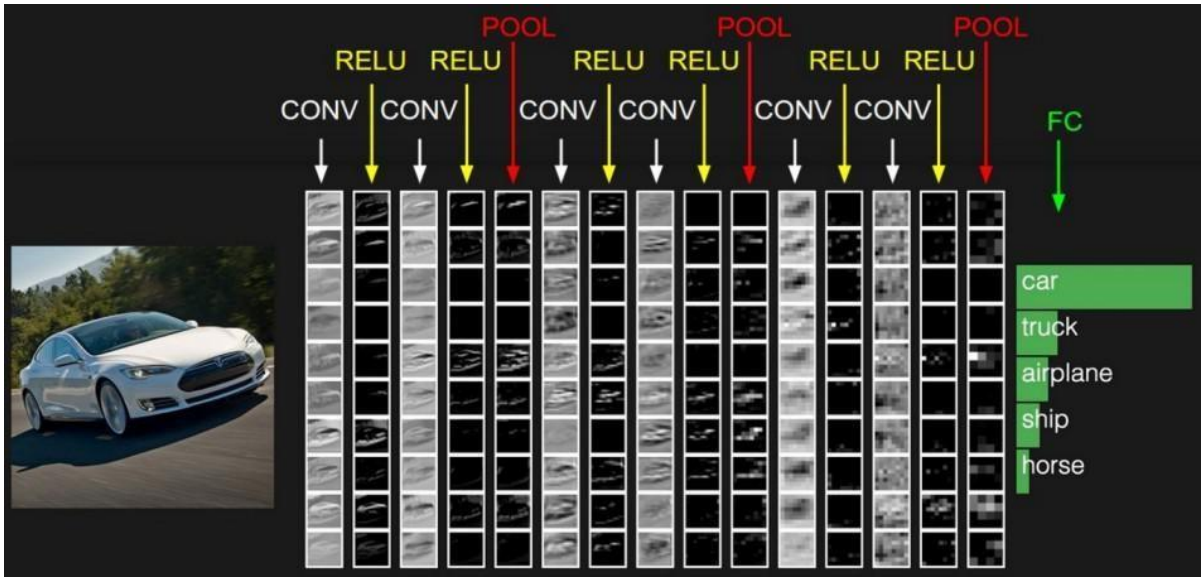

Figure 6. Practical Application of Convolution

Therefore, the output layer usually employs a linear activation function (or no activation function) to allow the network to predict real-valued pixel intensities directly. In some cases, if the pixel values are normalized between 0 and 1, a sigmoid activation may be used to constrain the output within the same range. The output produced is a denoised image of the same dimensions as the input, minimizing the difference between the predicted clean image and the ground truth clean image, typically using loss functions such as "Mean Squared Error" (MSE) during training.

CHAPTER-2

LITERATURE SURVEY

2.1 INTRODUCTION

Over the past decade, “deep convolutional neural networks” [20] (CNNs) have significantly advanced high-level vision tasks like visual recognition, motion analysis, and object segmentation. More recently, CNNs have been applied to low-level vision tasks, such as super-resolution (SR), image denoising [20], and compression artifact reduction, where they are trained to map low-quality images to high-quality outputs, typically aiming to remove noise or minimize artifacts. While “deeper is better” [20] is a widely accepted principle in high-level vision tasks—evidenced by networks like VGG, GoogleNet, and ResNet achieving substantial breakthroughs—this principle has not shown as much impact in low-level vision tasks. Despite the use of networks with 20 to 30 layers, such as DnCNN [1] and RED-Net, the performance gains in lowlevel tasks have been modest compared to earlier methods. This is because low-level vision tasks rely more on pixel-level features, where depth is less crucial. Instead, statistical priors, like non-local similarity or pixel distribution patterns (e.g., Gaussian noise), play a key role in enhancing the accuracy of these tasks, offering a more effective solution to image degradation issues.

2.2 EXISTING WORK

In CBDNet [2], an asymmetric loss function was employed to enhance the model’s ability to generalize to real-world noise scenarios, while also facilitating convenient interactive denoising. For training, they utilized a dataset comprising 400 images from BSD500 [4], 1600 images from Waterloo [11], and 1600 images from the MIT-Adobe FiveK dataset [10]. A batch size of 32 was selected, with each patch being 128 × 128 in size. The model was trained over 40 epochs, starting with a learning rate of 10^{-3} for the first 20 epochs, followed by a learning rate of 5×10^{-4} for fine-tuning during the remaining epochs. The model demonstrated PSNR scores of 30.78 on the SIDD dataset [5] and 38 on the DND dataset. Processing a 512x512 image takes approximately 0.4 seconds.

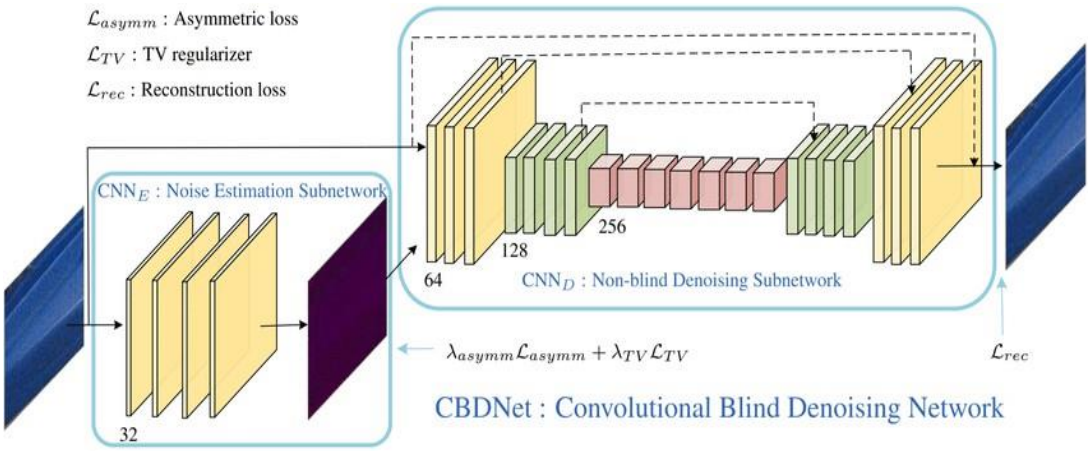

Figure 7. CBNet Architecture

In their work on RIDNet [3], the authors introduce a CNN-based denoising model specifically designed for both synthetic noise and real-world noisy images. This model is a single-blind denoising network for real noisy images, differing from prior methods. To enhance the network's ability to learn and improve feature extraction, they incorporated a restoration module along with feature attention, which adjusts the channel-wise features by considering the interdependence between channels. Additionally, they implemented LSC, SSC, and SC mechanisms, allowing low-frequency information to bypass the network, enabling it to focus on learning residuals. The model's architecture consists of four Enhanced Attention Mechanism (EAM) blocks, where most convolutional layers have a kernel size of 3x3, except for the final layer in the enhanced residual block and feature attention units, which utilize a 1x1 kernel. To ensure feature map dimensions remain consistent, the model applies zero-padding to the 3x3 convolutions. Each layer operates with 64 channels, and a downscaling factor of 16 is applied in the feature attention process, reducing the number of feature maps. The model processes a 512x512 image in approximately 0.2 seconds during evaluation and achieved PSNR scores of 31.38, 39.23, and 38.71 on the BSD68 [4], DnD [6], and SIDD [5] datasets, respectively.

The NBNet [7] architecture is built upon a modified UNet framework. NBNet incorporates four encoder and decoder stages, where feature maps are downsampled using strided convolutions in the encoder and upsampled using deconvolutions in the decoder. Skip connections between encoder and decoder stages transfer low-level features, and the primary innovation lies in the introduction of Subspace Attention (SSA) modules within these skip connections. Unlike conventional UNet architectures, where low- and high-level feature maps are directly fused, NBNet leverages SSA modules to project low-level features into a signal subspace guided by upsampled high-level features before fusion. This projection allows the model to better capture global structure information while preserving local details, improving denoising performance. It achieved PSNR scores of 29.16, 39.62 and 39.75 on BSD68 [4], DnD [6] and SIDD [5] datasets respectively.

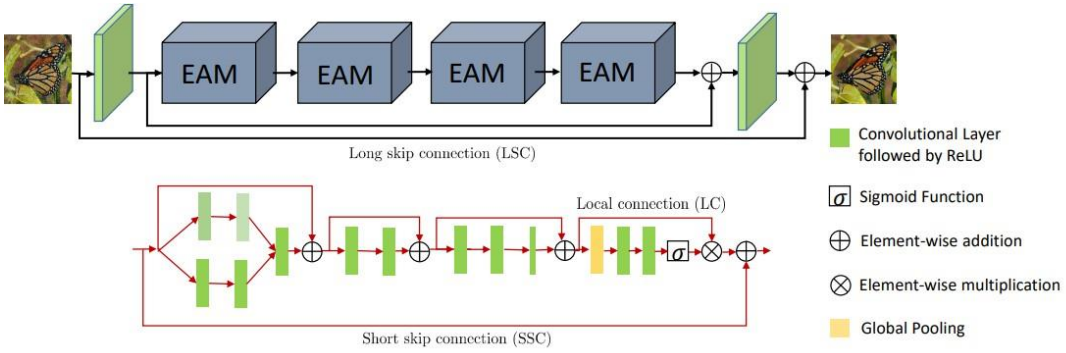

Figure 8. RidNet Architecture

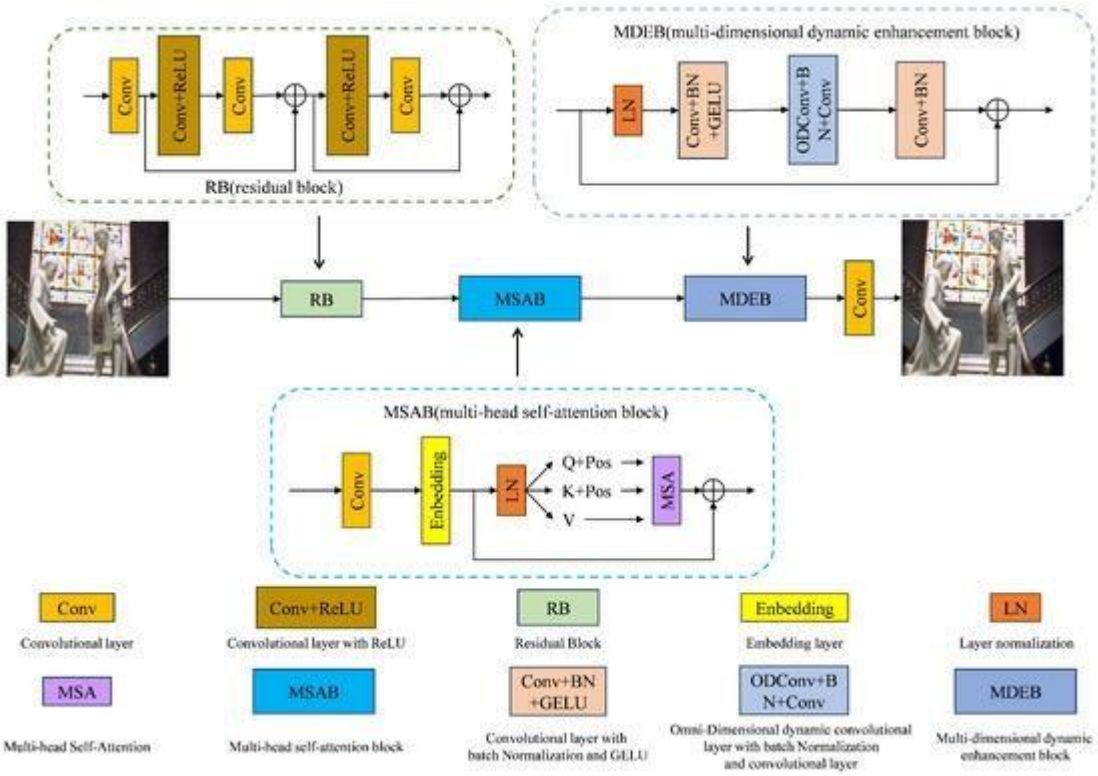

Figure 9. NBNet Architecture

DANet [2] is the latest architecture introduced for realworld image denoising tasks. Its core concept revolves around unfolding in-camera processing pipelines or learning the noise distribution through a generative adversarial network (GAN). In this dual adversarial framework, three components require optimization: the denoiser (R), the generator (G), and the discriminator (D). To stabilize training, they incorporated the gradient penalty technique from WGAN-GP [3], ensuring the discriminator adheres to a 1-Lipschitz constraint by adding an additional gradient penalty term, d. DANet includes two hyperparameters—one primarily affecting the performance of the denoiser R, while the other directly impacts the generator G. This architecture achieved PSNR scores of 39.25 and 39.79 on the SIDD and DND benchmark datasets, respectively.

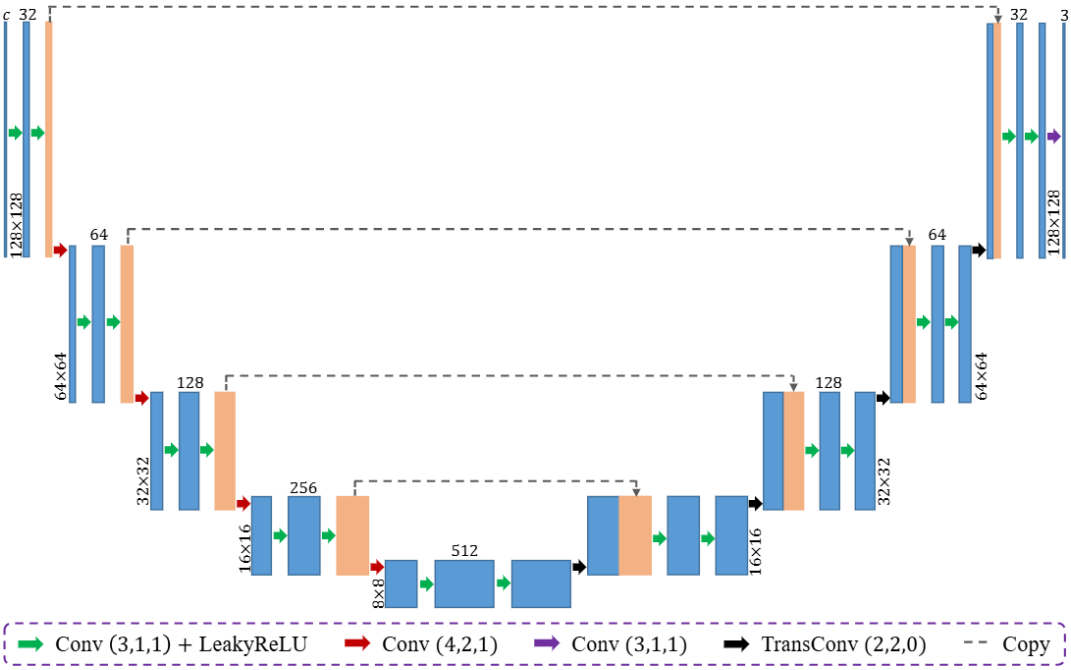

Figure 10. DANet Model Architecture

2.3 EVALUATION METRICS

1) **MSE** [8]: The Mean Squared Error(MSE) is a statistical measure that calculates the average of the squared differences between predicted and actual values. It serves as a risk function and is a key metric in empirical risk minimization, particularly in machine learning. The MSE formula takes the mean of the squared errors, where an error is the difference between the observed and predicted values. It can be calculated both within-sample (on the same data used to fit the model) and out of sample (using test data). In matrix form, it can be written as the squared errors summed and averaged over the number of data points.

$$MSE = \frac{1}{m \times n} \sum_{i=0}^{m} \sum_{j=0}^{n} \big(I(i,j) - K(i,j)\big)^2 \qquad [1.1]$$

2) **PSNR** [8]: The Peak Signal-to-Noise Ratio (PSNR) [8] measures the ratio between the maximum possible signal power and the power of the noise that distorts image quality. Typically expressed in decibels (dB), PSNR [8] uses a logarithmic scale due to the wide dynamic range of signals, which reflects the difference between the largest and smallest possible values. It is a common metric for evaluating the quality of

image reconstruction in

lossy compression codecs, where the signal represents the original data and noise represents distortion or error. PSNR [8] values typically range from 30 to 50 dB for 8-bit data and 60 to 80 dB for 16-bit data in image and video compression. In wireless transmission, a quality loss of 20 to 25 dB is generally acceptable. The PSNR [8] formula is given by:

$$PSNR = 10 \times log_{10}\left(\frac{MAX_i^2}{MSE}\right) \quad [1.2]$$

3) **SSIM** [8] **(Structural Similarity Index Method):** is a perception-based model for evaluating image degradation by analyzing changes in structural information. It measures perceived image and video quality by comparing an original image to its degraded version. SSIM emphasizes the interdependence of spatially close pixels and includes factors like luminance masking (distortions less visible along edges) and contrast masking (distortions less noticeable in textured areas).

Variants of SSIM:

i. **Multi-Scale SSIM** [8] (MS-SSIM): Evaluates images at different scales, considering luminance, contrast, and structural changes across resolutions for improved subjective performance.

ii. **Three-Component SSIM** [8] (3-SSIM): Accounts for human visual system's sensitivity to texture over smooth areas by disaggregating images into edges, texture, and smooth regions with weights of 0.5, 0.25, and 0.25 respectively. A 1/0/0 weighting (focusing on edges) aligns closer to human perception.

Formula for SSIM(x, y):

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \quad [1.3]$$

SSIM index ranges from -1 to 1, where 1 represents perfect similarity between images.

2.4 DATASETS

1) **SIDD** [5]: In recent years, there has been a significant shift from DSLR and point-and-shoot cameras to smartphone imaging, where the smaller apertures and sensor sizes of smartphones result in higher noise levels. The discipline lacked a comprehensive collection of genuine noisy photos with associated high-quality ground truth, despite a great deal of study on smartphone image denoising. The Smartphone Image Denoising Dataset (SIDD), a new dataset that includes roughly 30,000 photos from 10 scenes in various lighting conditions taken with five different smartphone cameras, fills this gap by providing a methodical approach to estimating ground truth for noisy smartphone images. CNN-based techniques trained on this high-quality dataset have been shown to outperform models trained with alternative methodologies, such as using low-ISO images as proxy ground truth. This dataset also makes it possible to benchmark denoising algorithms.

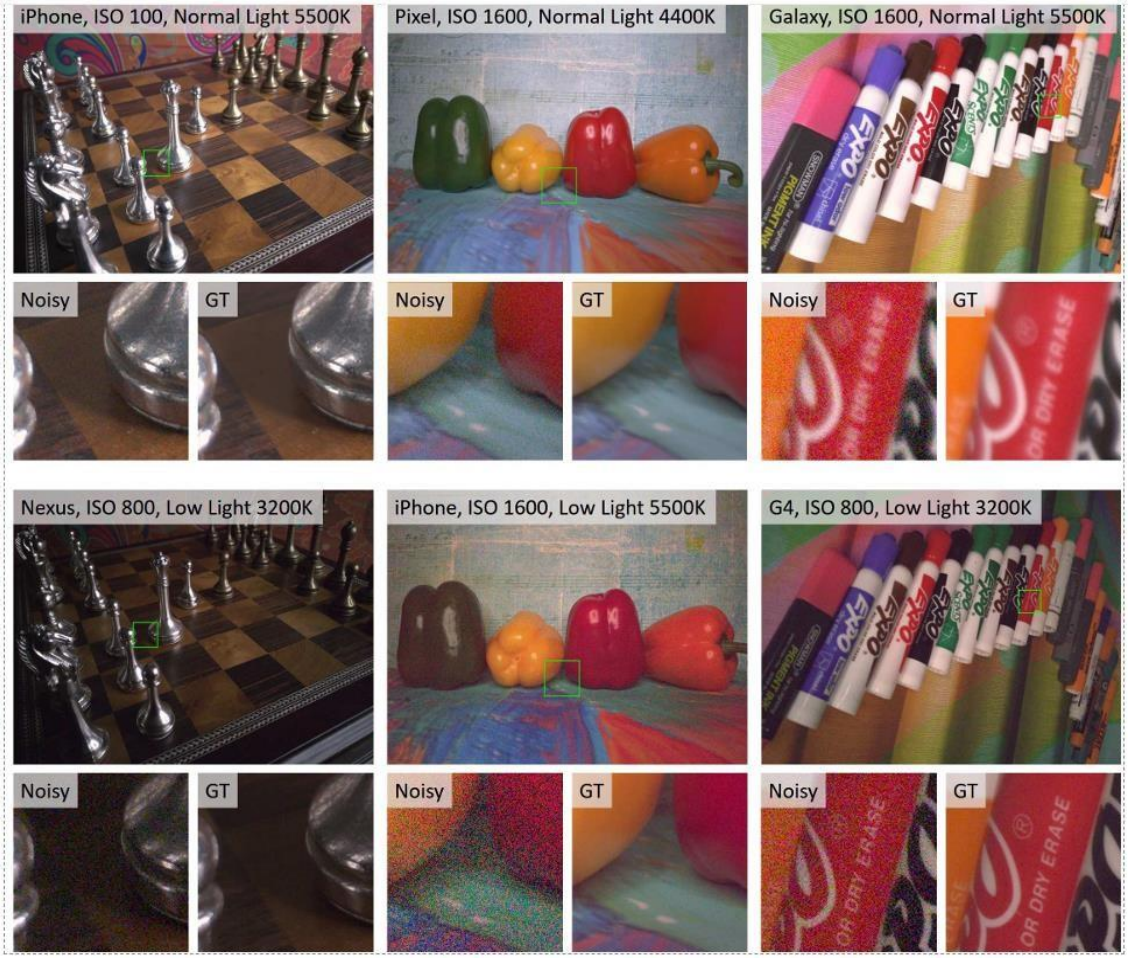

Figure 11. SIDD Dataset

2) **BSD** [4]: The Berkeley Segmentation Dataset (BSD) [4] is a widely used collection of images for evaluating segmentation and image processing algorithms, including image denoising. It contains several subsets with varying numbers of images, each providing a rich source of diverse natural images.

i. **BSD68** [4]: This subset consists of 68 images selected from the larger dataset. It is often used as a benchmark for evaluating image segmentation and denoising algorithms due to its manageable size and representative diversity.

ii. **BSD100** [4]: This version includes 100 images, expanding on the BSD68 [4] set. The increased number of images allows for more comprehensive testing and validation of algorithms, offering a broader range of textures, colors, and structures.

iii. **BSD300** [4]: This dataset features 300 images, providing an even more extensive set of natural images. The larger size enhances the dataset's ability to assess the performance of denoising methods across a wider variety of scenarios, making it useful for researchers looking to develop and test robust algorithms. Overall, these datasets serve as standard benchmarks for evaluating the effectiveness of image denoising techniques and help researchers compare their methods against established results in the field.

Figure 12. BSD Dataset

3) **Urban100** [9]: The dataset consists of 100 high resolution images that capture various urban scenes, including buildings, streets, and other architectural features. These images contain a range of textures, colors, and structures, which make them suitable for testing the robustness of denoising algorithms. The dataset consists of 100 high-resolution images that capture various urban scenes, including buildings, streets, and other architectural features. These images contain a range of textures, colors, and structures, which make them suitable for testing the robustness of denoising algorithms. The dataset includes a wide variety of scenes, ensuring that models trained or tested on Urban100 [9] can generalize well across different types of urban imagery. While the original images are clean, researchers often add synthetic noise (e.g., Gaussian noise) at various levels to create noisy versions for denoising tasks. This setup allows for a controlled evaluation of how well different algorithms can recover the original clean images from their noisy counterparts.

Figure 13. Urban100 Dataset

2.5 DRAWBACKS OF EXISTING METHODS

1) **High Computational Complexity and Resource Demand:** State-of-the-art denoising models typically consist of millions of trainable parameters, leading to significant computational demands. While these models perform exceptionally well in controlled laboratory environments, their need for vast datasets and computational resources makes them impractical for many real-world applications.

2) **Inefficiency in Image Processing Speed:** Current state of the art models require 0.2 to 0.4 seconds to denoise a single 512x512 image. This processing time limits their usability in scenarios where real-time or high-speed image processing is essential.

2.6 OBJECTIVES

1) **Reduction of Computational Complexity and Resource Demand:**

   One of the primary goals is to design a model that significantly reduces the computational burden typically associated with deep learning-based image denoising. High computational complexity often leads to increased training times, higher memory requirements, and greater energy consumption, making it challenging to deploy models on resource-constrained devices. By optimizing the model architecture, reducing the number of parameters, and adopting efficient training strategies, we aim to create a solution that can be trained and inferred using minimal hardware resources without compromising performance.

2) **Improvement of Image Processing Speed:**

   In real-world applications, especially in areas such as medical imaging, security, and mobile photography, the speed at which images are processed is critical. Slow processing times can hamper usability and operational efficiency. Therefore, the model is designed to offer faster inference, enabling real-time or near-real-time denoising capabilities. Techniques such as lightweight convolutional operations and efficient network designs are incorporated to ensure that the model can handle high-resolution images swiftly and effectively.

3) **Maintaining or Enhancing State-of-the-Art Performance While Ensuring Efficiency:**

   While achieving efficiency is crucial, it is equally important to maintain or even surpass the denoising quality achieved by existing state-of-the-art methods. The objective is to strike a balance between model performance and operational efficiency. This involves careful selection of network architectures, loss functions, and training methodologies that not only preserve fine image details and textures but also ensure robustness against various types and levels of noise. The goal is to deliver high-quality denoising results comparable to or better than current leading approaches while keeping the model lightweight and practical for deployment across a range of platforms.

CHAPTER-3

METHODOLOGY

3.1 INTRODUCTION

In this section, we present the methodology adopted for the blind denoising of images using a Convolutional Neural Network [19] (CNN) -based approach. The proposed model architecture is carefully designed to address the task of image denoising, catering to high-performance requirements in resource-limited environments. Our CNN-based architecture comprises two main parts: a Super Resolution network and a Denoiser network, enabling the model to upscale noisy, low-resolution images before removing noise. By leveraging the structural capabilities of CNN layers [19], the model efficiently captures essential features for both high-frequency detail recovery and noise suppression, achieving competitive denoising results. The architecture's streamlined design balances computational efficiency with denoising effectiveness, providing a practical solution for image restoration across a variety of noise levels and types without prior information on noise characteristics.

Input Layer

When deep learning models are built, the foundation step of the model preparation starts with the input layer. "***Keras Input Layer*** is essential for defining the shape and size of the input data the model with receive. The Input Layer in Keras is the starting point of any neural network. It is not a traditional layer that processes or transforms data. Instead, it serves as a specification of the kind of input the model expects, including the dimensions and type of data. Essentially, it defines how the network should receive the input data, setting the stage for the subsequent layers" [17].

There are two ways to specify the input layer:

- Specifying input layer explicitly.
- Implicit specification by passing **'input_shape'** dimension in the first layer of the neural network.

Key Features of Keras Input Layer

1. **Defining Input Dimensions**: The Input layer is used to specify the dimensions of the incoming data, excluding the batch size. For example, if the input is a 28x28 pixel image, the shape would be provided as (28, 28).
2. **Ensuring Compatibility**: It prepares the first layer of a model to properly accept the incoming data, ensuring that the data flows correctly through the network.
3. **Providing Flexibility**: The Input layer can accommodate inputs of varying sizes, which is particularly beneficial for models working with sequences or time-series data.

Parameters of keras.Input

1. **shape**: A tuple that specifies the dimensions of each input sample, not including the batch size. For instance, to input 32x32 RGB images, use shape=(32, 32, 3).
2. **batch_size**: Defines a constant batch size for the input. Setting this is particularly important when working with stateful recurrent neural networks.
3. **dtype**: Indicates the data type of the input, such as 'float32' or 'int32'. If left unspecified, it "defaults to 'float32'" [18].
4. **sparse**: A boolean flag that tells the model whether the input should be treated as a sparse tensor "(tf.sparse.SparseTensor)" [18]. This is useful for processing sparse data, such as tokenized text sequences.
5. **batch_shape**: Similar to shape, but includes the batch size as well. For example, "batch_shape=(10, 32)" [18] means each batch has 10 samples, each with 32 features.
6. **name**: Lets you assign a custom name to the input layer, which can help with identifying layers when saving or loading the model or in more complex model architectures.
7. **tensor**: If you pass an existing tensor, the Input layer will use it instead of creating a new one. This is helpful when integrating external tensors into your model.

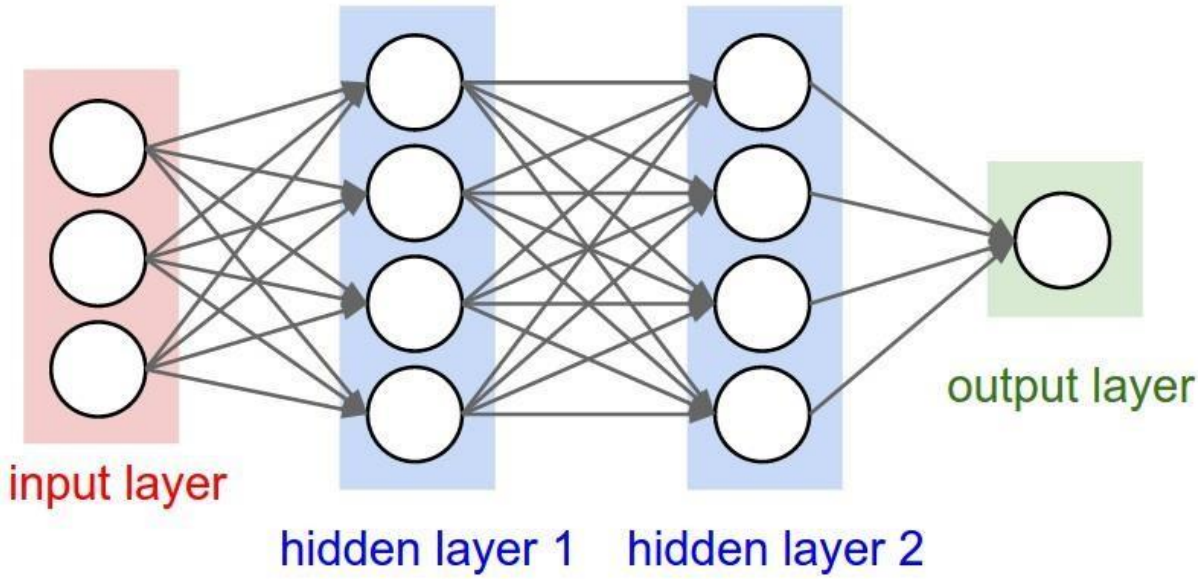

Figure 14. Input Layer

Input Shape: (112, 112, 3)

The model accepts an input image of size 112 × 112 with three color channels (RGB). This compact image resolution is chosen to reduce computational complexity and memory requirements while ensuring efficiency in model training and inference.

3.2 SUPER RESOLUTION BLOCK

The Efficient Sub-Pixel [18] Convolutional Neural Network (ESPCN) [2] is designed to reconstruct high- resolution (HR) images from low-resolution (LR) inputs. Unlike traditional methods that upscale images using bicubic interpolation before applying a CNN, ESPCN performs the super-resolution operation directly in the LR space, which reduces computational complexity and improves performance.

Key Components

1. **Sub-Pixel Convolution Layer [18]**:
    - This layer is the core innovation of ESPCN [18]. It learns an array of upscaling filters that convert LR feature maps into HR images. Instead of upscaling the image first and then applying convolutions, the sub-pixel convolution layer rearranges the LR feature maps into a higher resolution output.

2. **Network Architecture**:

    - The network consists of several convolutional layers followed by the sub-pixel convolution layer [18]. The initial layers extract features from the LR image, and the final sub-pixel convolution layer upscales these features to produce the HR image.

3. **Training**:

    - The network is trained using pairs of LR and HR images. The loss function typically used is the mean squared error (MSE) between the predicted HR image and the ground truth HR image.

Advantages

- **Efficiency**: By performing convolutions in the LR space, ESPCN [18] reduces the number of operations required, making it faster and more efficient.
- **Quality**: The sub-pixel convolution layer allows the network to learn more complex upscaling filters, resulting in better image quality compared to traditional methods.

Applications

- **Real-Time Video Super-Resolution**: ESPCN [18] can be used to enhance the resolution of video frames in real-time, making it suitable for applications like video streaming and surveillance.
- **Image Enhancement**: It can be applied to improve the quality of images in various fields, including medical imaging, satellite imagery, and photography.

Output Shape: (224, 224, 3)
Trainable Parameters: 110,515

The Super-Resolution Block is responsible for upscaling the input image from (112 × 112) to (224 × 224) by leveraging a series of convolutional layers and residual connections. This block effectively doubles the spatial resolution of the image and forms the encoder part of the architecture.

- **Initial Convolution Layer:** A Conv2D layer with 32 filters and a kernel size of 3×3 is applied to the input, followed by a LeakyReLU activation. This step

extracts low-level features from the image.

- **Residual Blocks:** Two Residual Blocks are employed. Each block consists of two convolutional layers with a kernel size of 3 × 3 and 32 filters, followed by Batch Normalization and LeakyReLU activations. These blocks capture intricate features while addressing the vanishing gradient problem. Skip connections are added to improve feature flow and gradient propagation, enhancing model training.
- **Final Convolution and Up sampling:** The final convolutional layer in the Super- Resolution Block has 64 filters, followed by a Depth-to-Space operation (also known as Pixel Shuffling), which rearranges the tensor to increase its spatial resolution to 224×224. This technique provides an efficient way to upscale the image.
- **Output:** A Conv2D layer with 3 filters reconstructs the RGB image, which now has double the spatial dimensions (224 × 224) compared to the input.

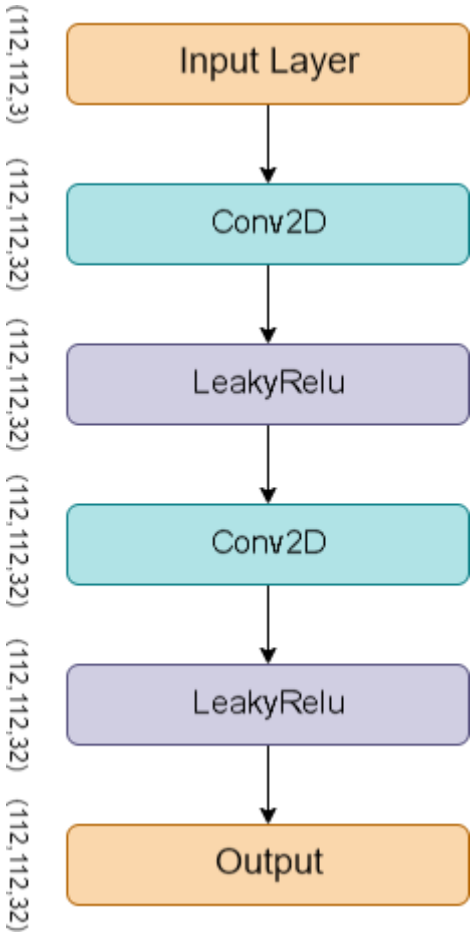

Figure 15. Residual Block

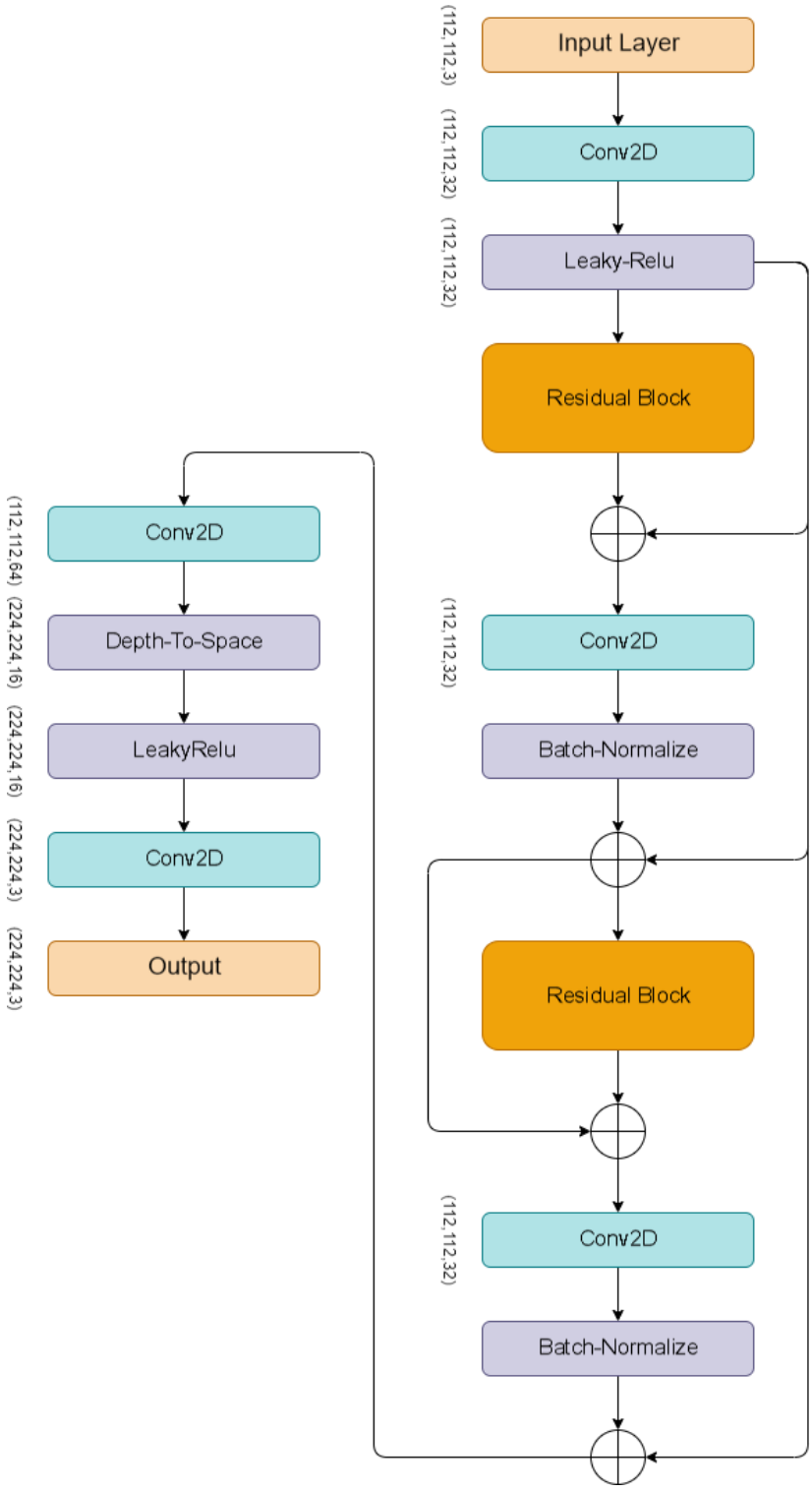

Figure 16. Super Resolution Block

3.2 DENOISER BLOCK

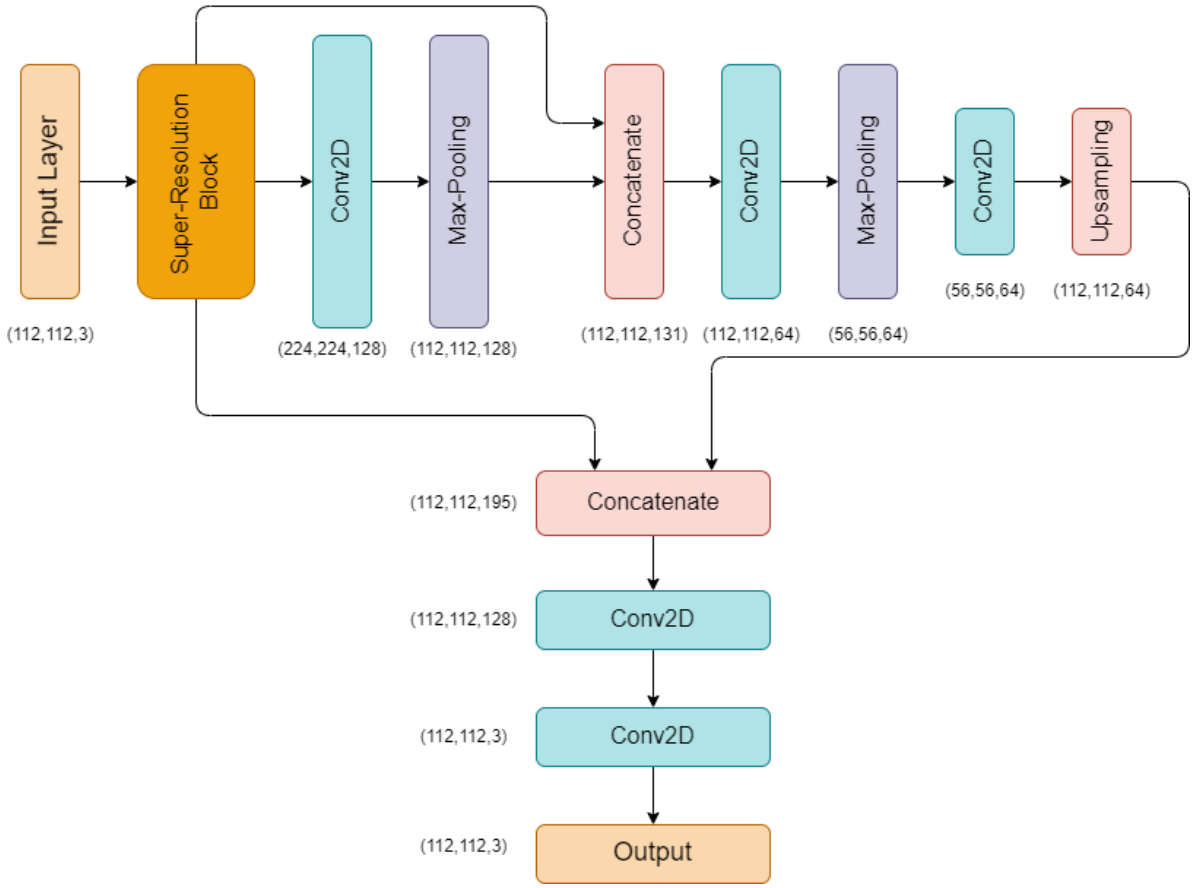

Figure 17. Model Architecture

Output Shape: (112, 112, 3)

Trainable Parameters: 344,259

The Denoiser Block performs the task of noise removal by processing the upscaled image from the Super-Resolution Block. This component can be considered a decoder block, reducing the resolution back to (112 × 112) while preserving important visual features through skip connections and convolution operations.

- Convolution Layers: The first convolution layer in the Denoiser Block has 128 filters, followed by a MaxPooling2D layer that downscales the image to (112 × 112) for processing.
- Concatenation with Input: A skip connection is established where the original input image is concatenated with the down sampled version of the upscaled image, producing an intermediate feature map of size (112×112×131). This

fusion ensures the model retains essential features from the initial input.

- Down sampling and Up sampling: – The image is further down sampled using another convolutional layer with 64 filters, followed by another MaxPooling2D operation that reduces the spatial size to (56 × 56). An UpSampling2D layer increases the resolution back to (112 × 112).
- Concatenation with Intermediate Features: Skip connections are introduced again by concatenating the up sampled feature map with previous intermediate layers, producing a feature map of size (112 × 112 × 195).
- Final Convolutions: The concatenated feature map is passed through a convolutional layer with 128 filters and finally reduced to 3 channels (RGB) using a Conv2D layer with 3 filters. The output is the denoised image, reconstructed to its original resolution of (112 × 112).

3.3 WORKING PRINCIPLE

- The architecture leverages the Super-Resolution Block to enhance the resolution of the input image from (112 × 112) to (224×224). This block employs a combination of convolutional layers, residual connections, and depth-to-space transformation to ensure that the upscaled image retains fine details without introducing significant artifacts.

- Following this, the Denoiser Block takes over, utilizing down sampling and upsampling techniques, combined with skip connections, to effectively remove noise from the upscaled image. The final result is a clean, denoised image at the original resolution of (112 × 112).

- Skip connections play a crucial role in both blocks by preserving critical features from the earlier stages and preventing information loss during down sampling and upsampling.

3.4 SUMMARY

• Total Parameters: 454,902
• Trainable Parameters: 454,774
• Non-trainable Parameters: 128

This architecture is highly optimized for the task of image denoising, leveraging the efficiency of the Super-Resolution Block to work with smaller input patches, significantly reducing the amount of data and time required for training and inference. Furthermore, the Denoiser Block ensures that the model can effectively remove noise while retaining critical image features, providing high-quality denoised images as output.

CHAPTER-4

IMPLEMENTATION

4.1 INTRODUCTION

In this section, we outline the process for implementing the CNN-based architecture used for blind denoising of images. We begin with data preprocessing, where the noisy images are prepared for training by resizing them to a consistent shape and applying normalization to aid in model convergence. Gaussian noise is artificially introduced to simulate realistic image degradation, which helps in enhancing the model's robustness. Next, we detail the model training using TensorFlow and Keras, including architectural choices, hyperparameter tuning, and training strategies. The network is trained with a progressive learning rate schedule and evaluated using PSNR to optimize both denoising accuracy and computational efficiency.

4.2 DATA PREPROCESSING

1) **SIDD** [5]: In our study, we utilized images from the SIDD [5] dataset, resizing them to dimensions of (2576, 1456, 3). From each image, we extracted patches with a resolution of (112, 112, 3). For training, we randomly selected 7,000 patches, while 3,000 patches were allocated for validation. The final 10 images from the dataset were reserved for testing, which, after patching, yielded 2,990 test patches.

2) **Urban100** [9]: In our study, we utilized the Urban100 [9] dataset to extract a total of 80 images for the training set. These images were cropped to a size of 560 × 560 × 3 pixels. Subsequently, we generated patches of size 112×112×3 from each image, resulting in a total of 2,000 patches designated for training with a validation split of 0.1. For the testing phase, we selected 20 remaining images from the dataset, which were similarly cropped to 560×560×3 pixels. Following the patch generation process, we obtained 500 patches of size 112 × 112 × 3 for testing purposes.

4.3 MODEL TRAINING

Forward Pass

During the forward pass of a neural network, the input moves through a series of layers in a specific sequence:

1. **Input Layer:** The process begins when the network receives input data, such as an image or a batch of images.
2. **Convolutional Layers [19]:** The input is then processed by convolutional layers, where filters (kernels) are applied to detect features like edges, textures, and patterns within the data.
3. **Activation Functions [19]:** After convolution, non-linear activation functions (such as ReLU) are used to introduce non-linearity, allowing the model to learn more complex relationships.
4. **Pooling Layers:** Pooling operations (like max pooling) are performed to downsample the feature maps, preserving key information while reducing the spatial dimensions and computational load.
5. **Fully Connected Layers [19]:** The resulting feature maps are flattened into a single vector and passed through fully connected layers, which interpret and combine the learned features to make a final decision.
6. **Output Layer:** Finally, the network generates its predictions. For classification

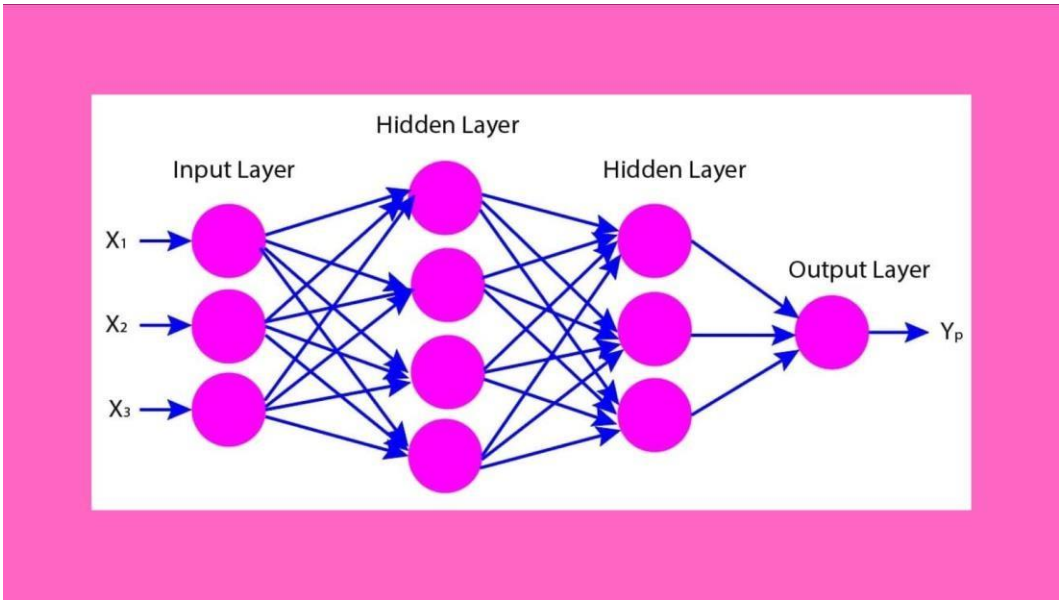

Figure 18. Forward Passing in Neural Networks

problems, a softmax function is often applied to produce probability scores across different classes.

Loss Calculation

The loss calculation step involves computing the error between the network's predictions and the true labels. This is done using a loss function, which quantifies the difference between the predicted and actual values. Common loss functions include:

- **Mean Squared Error (MSE)**: Used for regression tasks, it calculates the average squared difference between predicted and actual values.
- **Categorical Cross-Entropy**: Used for multi-class classification tasks, it measures the difference between the predicted probability distribution and the true distribution (one- hot encoded labels).

The computed loss provides a measure of how well the network's predictions match the true labels, guiding the subsequent weight updates.

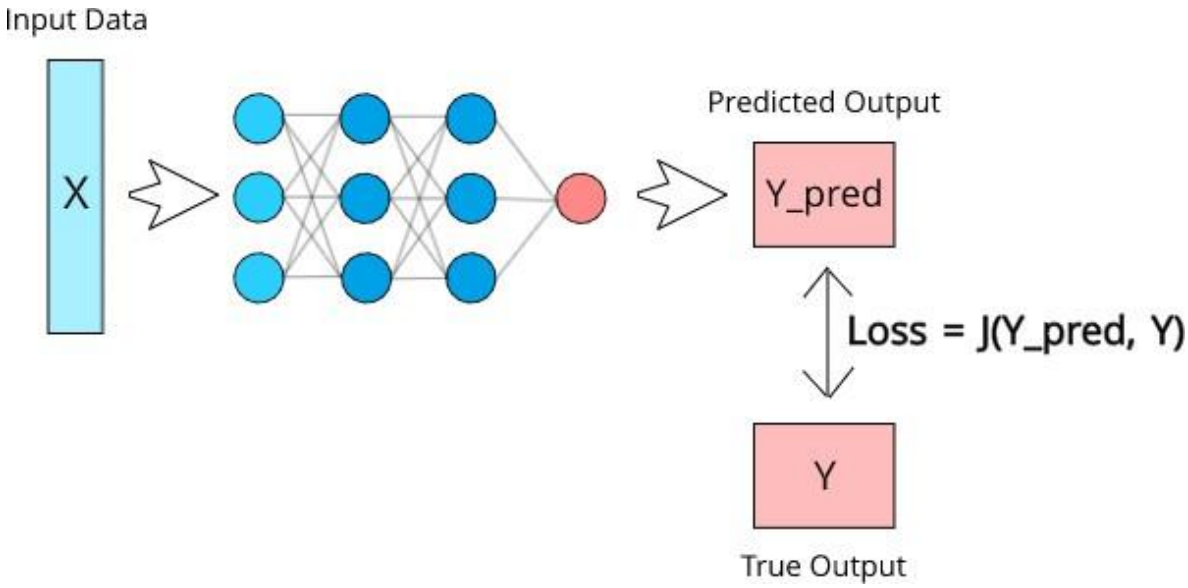

Figure 19. Loss Calculation

Backward Pass

The backward pass, also known as backpropagation, involves calculating the gradients of the loss with respect to the network's weights. This process includes:

1. **Gradient Calculation**: Using the chain rule of calculus, the gradients of the loss function with respect to each weight are computed. These gradients indicate the direction and magnitude of the change needed to minimize the loss.
2. **Error Propagation**: The error is propagated backward through the network, from the output layer to the input layer, updating the gradients at each layer.

Backpropagation ensures that each weight in the network is adjusted in a way that reduces the overall loss.

Weight Updates

During the weight update step, the optimizer uses the calculated gradients to adjust the network's weights. This process involves:

1. **Learning Rate**: A hyperparameter that controls the size of the weight updates. A smaller learning rate results in smaller updates, leading to more gradual convergence, while a larger learning rate results in larger updates, which can speed up convergence but may cause instability.

2. **Optimizer Algorithms**: Common optimizers include:

    - **Stochastic Gradient Descent (SGD) [17]**: Updates weights using the gradients of a single batch of data.

    - **Adam (Adaptive Moment Estimation) [17]**: Combines the advantages of two other extensions of SGD, namely AdaGrad and RMSProp, to adapt the learning rate for each parameter.

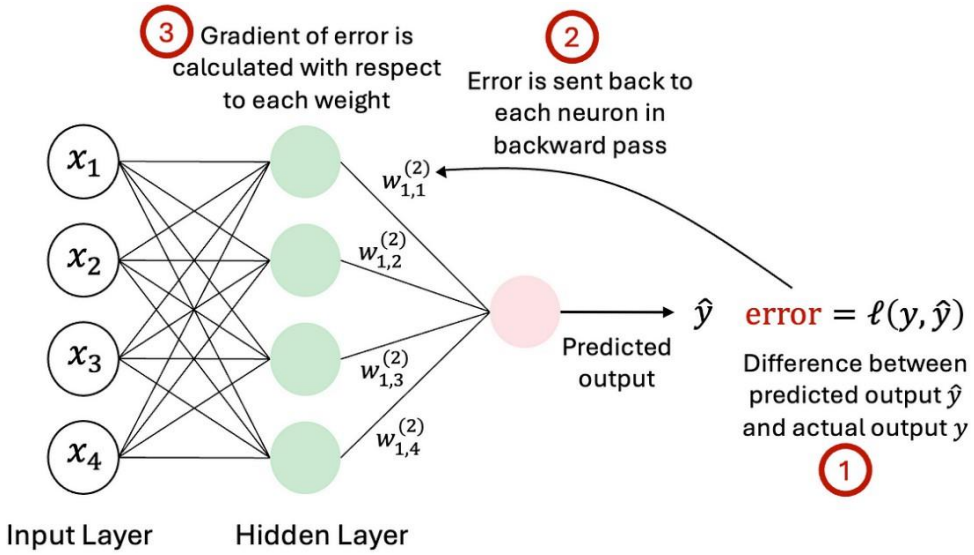

Figure 20. Back Propagation and weight updates

The optimizer iteratively updates the weights to minimize the loss function, improving the model's performance.

Epochs and Batches

Training a neural network involves iterating over the entire training dataset multiple times, known as epochs. Within each epoch, the dataset is divided into smaller subsets called batches. The training process includes:

1. **Epochs**: One complete pass through the entire training dataset. Multiple epochs are used to ensure the model learns effectively from the data.
2. **Batches**: Subsets of the training data processed in parallel. Using batches helps in efficient computation and faster convergence.

During each epoch, the model's weights are updated after processing each batch, allowing the model to learn incrementally from the data. This process continues until the model's performance stabilizes or improves sufficiently.

The Super-Resolution Image Denoiser Network (SRIDNet) was trained using the TensorFlow and Keras frameworks on image patches of size (112, 112, 3). The training process was carried out in three stages, each with progressively reduced learning rates to enhance model performance and stability. Initially, the model was trained for 20 epochs with a learning rate of 0.001.

This was followed by an additional 20 epochs at a reduced learning rate of 0.0001, and a final 10 epochs at a further reduced learning rate of 0.00001. The Adam optimizer was employed to minimize the mean squared error (MSE) loss function, with Peak Signal-to-Noise Ratio (PSNR) used as the evaluation metric. The batch size was set to 16 for all training stages. On the Urban100 dataset, the model required approximately 45 seconds per epoch, while on the SIDD dataset, it required 130 seconds per epoch. These timings reflect the computational complexity and dataset size differences between the two datasets.

Stage 1: Initial Training

- **Epochs**: 20
- **Learning Rate**: α=0.001
- **Objective**: The initial stage aimed to provide a robust starting point for the model by allowing significant weight adjustments. The relatively higher learning rate facilitated rapid convergence during the early epochs.

Stage 2: Intermediate Training

- **Epochs**: 20
- **Learning Rate**: α=0.0001
- **Objective**: In the second stage, the learning rate was reduced by an order of magnitude to 0.00010.0001. This reduction helped in fine-tuning the model parameters, allowing for more precise adjustments and preventing overshooting of the optimal weights.

Stage 3: Final Training

- **Epochs**: 10
- **Learning Rate**: α=0.00001
- **Objective**: The final stage involved a further reduction in the learning rate to 0.000010.00001. This stage focused on stabilizing the model by making very fine adjustments to the weights, ensuring convergence to a local minimum.

Optimization and Loss Function

- **Optimizer**: Adam (Adaptive Moment Estimation) [17]
    - The Adam optimizer was employed due to its adaptive learning rate capabilities and efficient handling of sparse gradients. Adam combines the advantages of two other extensions of stochastic gradient descent, namely AdaGrad and RMSProp.

- **Loss Function**: Mean Squared Error (MSE)
    - The MSE loss function was used to quantify the difference between the predicted high-resolution images and the ground truth images.

Evaluation Metric

- **Peak Signal-to-Noise Ratio (PSNR)**: PSNR was used as the primary evaluation metric to assess the quality of the reconstructed images.

Training Configuration

- **Batch Size**: 16
    - A batch size of 16 was chosen to balance computational efficiency and the stability of gradient updates.

Computational Performance

- **Urban100 Dataset**: The model required approximately 45 seconds per epoch on the Urban100 dataset. This timing reflects the computational complexity associated with processing high-resolution urban images.

- **SIDD Dataset**: On the SIDD dataset, the model required approximately 130 seconds per epoch. The increased time per epoch is attributed to the larger size and higher complexity of the SIDD dataset, which contains diverse indoor scenes with varying lighting conditions.

These timings highlight the differences in computational demands between the two datasets, emphasizing the importance of dataset characteristics in determining training efficiency.

CHAPTER-5

RESULTS

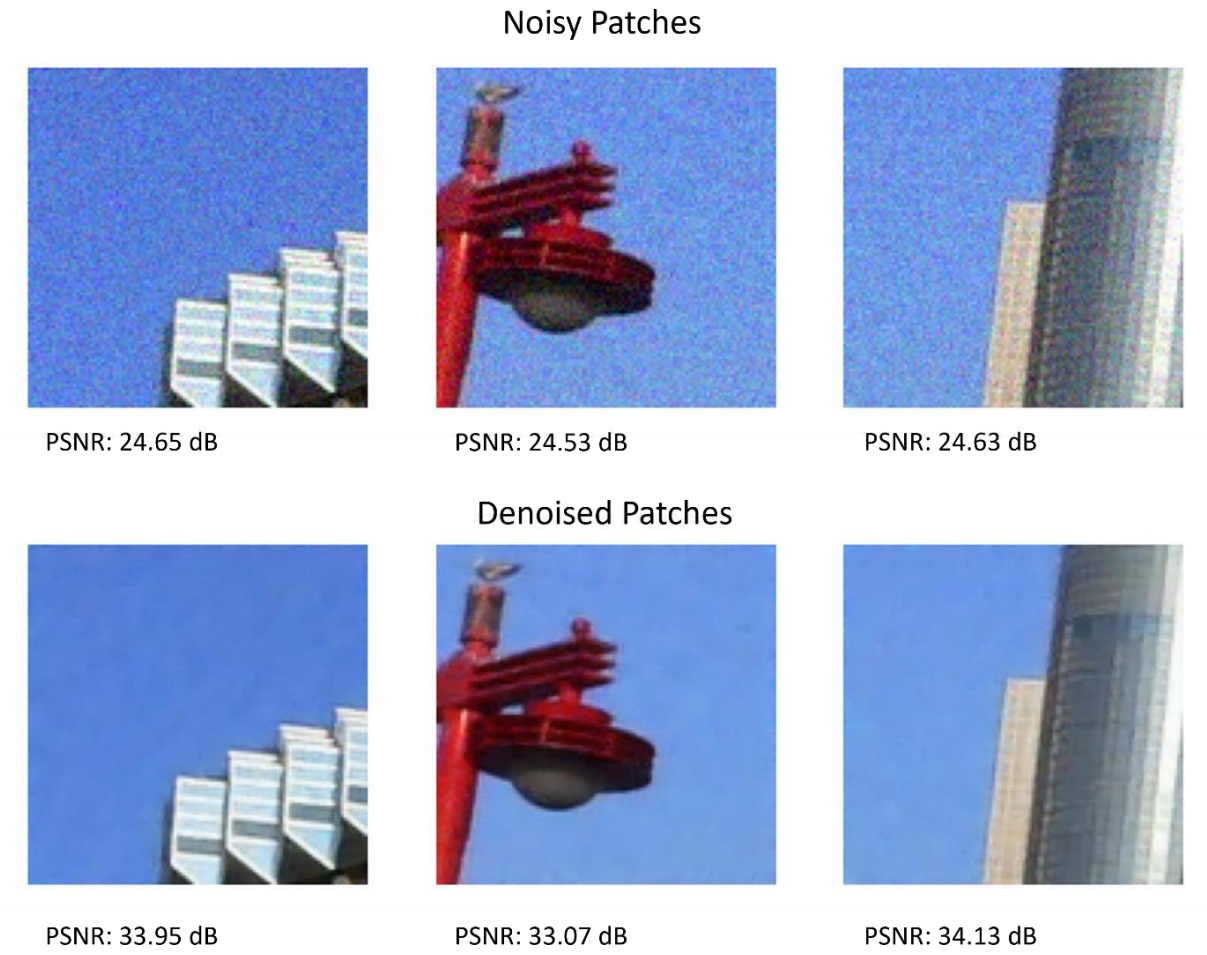

Figure 21. PSNR results on 112 × 112 × 3 image patches (σ = 15) from Urban100 dataset

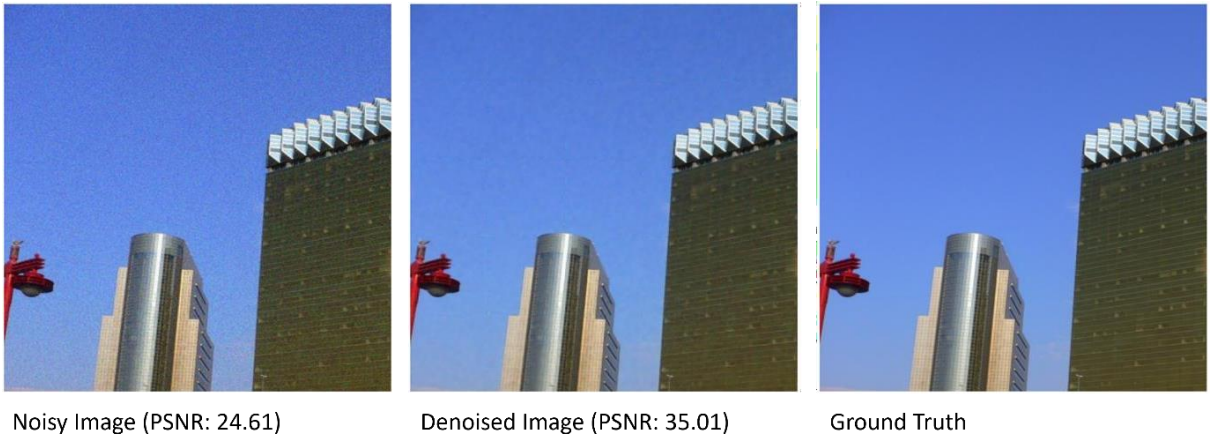

Figure 22. PSNR results on an entire image (σ = 15) of size 560 × 560 × 3 from Urban100 dataset

The proposed Super-Resolution Image Denoiser Network (SRIDNet) was evaluated on the Urban100 [9] dataset, with the last 10 images used as the test set. The evaluation process was conducted with TensorFlow, and inference was completed in an average time of 20-24 ms per image of size 560×560×3 at noise level (σ = 15). The model's performance was assessed using Peak Signal-to-Noise Ratio (PSNR) as the evaluation metric. The results indicate that SRIDNet achieved an average PSNR of 34.23 dB across the test images, with a maximum PSNR of 35.82 dB.

```
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 19ms/step
Avg PSNR:  tf.Tensor(34.230599650363615, shape=(), dtype=float64) Max PSNR:  tf.Tensor(35.81651203385877, shape=(),
dtype=float64)
```

Figure 23. Avg. PSNR and evaluation time per image in TensorFlow and Keras for 560 × 560 × 3 images at noise level (σ = 15) from Urban100 dataset.

Table 1. Comparison of Avg. PSNR results with existing models on Urban100 dataset.

Model	Sigma	PSNR (dB)
DnCNN	15	32.98
	25	30.81
	50	27.59
FFDNet	15	33.83
	25	31.40
	50	28.05
DRUNet	15	34.81
	25	32.60
	50	29.61
SwinIR	15	35.13
	25	32.90
	50	29.82
Ours	15	**34.13**
	25	**32.01**
	50	**28.67**

The proposed Super-Resolution Image Denoiser Network (SRIDNet) was evaluated on the Urban100 dataset and compared against state-of-the-art denoising models, including DnCNN [1], FFDNet [12], DRUNet [13], and SwinIR [14], across different noise levels (σ = 15, 25, 50). The evaluation metric is Peak Signal-to-Noise Ratio (PSNR) as presented in Table 1.

```
10/10 [==============================] - 1s 104ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
10/10 [==============================] - 1s 91ms/step
tf.Tensor(39.50148010907847, shape=(), dtype=float64)
```

Figure 24. Avg. PSNR and evaluation time of SIDD images in TensorFlow and Keras

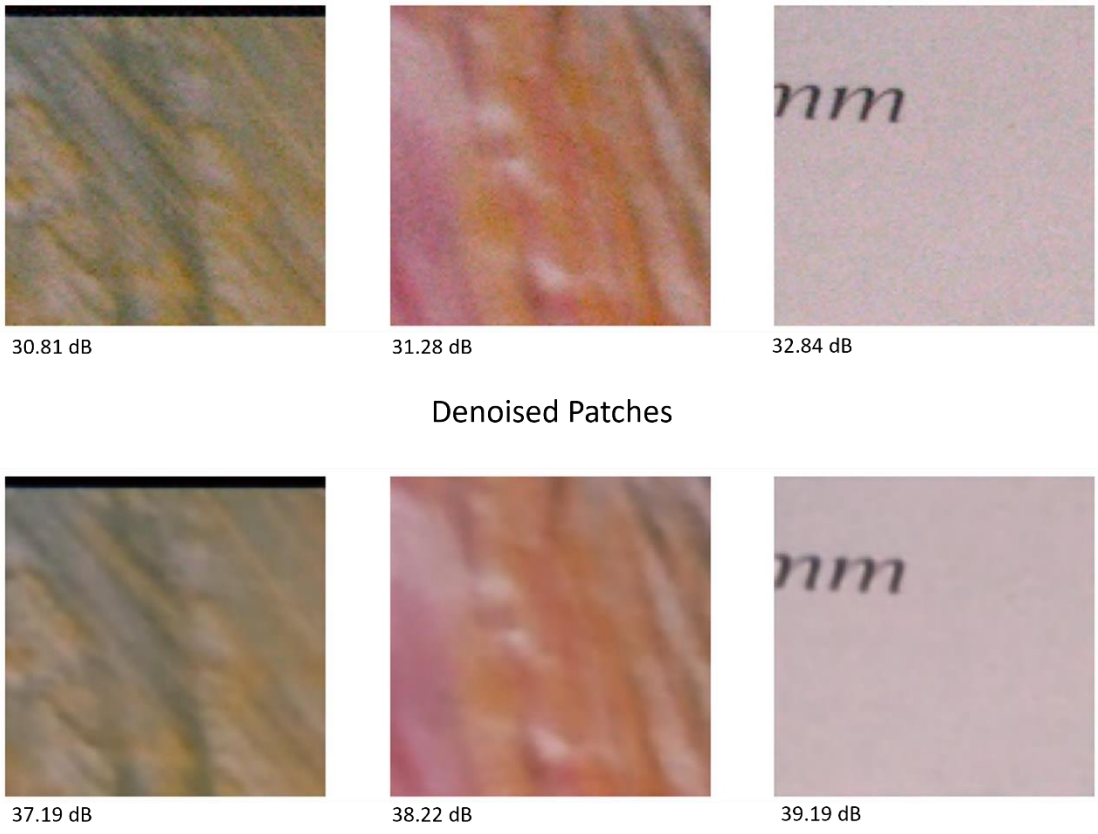

Figure 25. Denoising results on SIDD dataset images

The proposed model was evaluated on the SIDD dataset and benchmarked against several state-of-the-art denoising models, including DANet [16], VDN [15], RIDNet [3], and CBDNet [2]. The model achieved an average PSNR of 39.5 on the SIDD dataset, with an average inference time of 95 ms for images sized 2576 × 1456, which were divided into 299 patches of size 112×112.

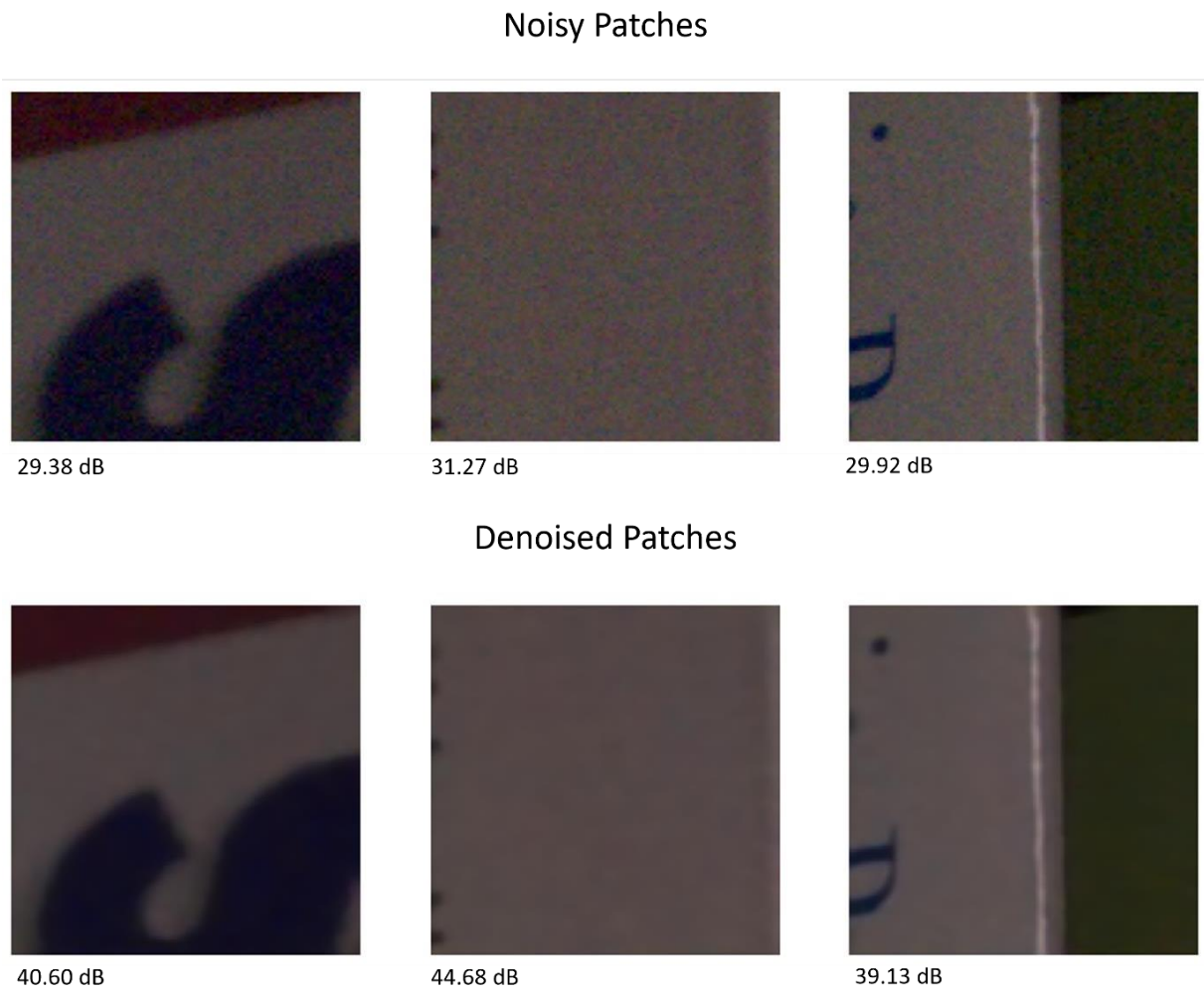

Figure 26. Denoising results on SIDD dataset images

Figures 14. and 15. illustrate the denoising results of lowlight and bright-light images from the SIDD [5] dataset, respectively, using (112 × 112 × 3) image patches. Fig 19. presents a comparison of denoising performance across different models on the SIDD [5] dataset, showing that our model outperformed others by achieving a higher average PSNR. While DnCNN [1] and FFDNet [12]—models of similar size—performed well on images with additive Gaussian white noise (AGWN) from Urban100 [9] dataset, they underperformed on real-world photographic images from the SIDD [5] dataset.

In contrast, our model demonstrated robust performance across both datasets with greater efficiency. Table 3. compares the inference times of the models for denoising an image of size 256 x 256 evaluated in this study. While our model is faster than all models except FFDNet [12], it offers more consistent results on both the Urban100 and SIDD [5] datasets, where FFDNet [12] struggles with real-world images.

Table 2. Comparison of Avg. PSNR results with existing models on SIDD dataset.

Model	**PSNR (dB)**
CBDNet	30.78
RIDNet	38.71
VDN	39.28
DANet	39.47
DnCNN	26.21
FFDNet	29.20
Ours	**39.50**

Table 3. Comparison of inference time with existing models.

Model	**Inference Time (s)**
DnCNN	0.0314
FFDNet	0.0071
CBDNet	0.4
RIDNet	0.2
Ours	**0.0208**

CHAPTER-6

CONCLUSION AND FUTURE SCOPE

6.1 CONCLUSION:

In conclusion, this research presents SRIDNet, a novel deep learning architecture for image denoising that effectively addresses the limitations of current state-of-the-art models, such as high computational demands, extensive training data requirements, and slow inference times. By designing a lightweight model that integrates super-resolution and denoising tasks, SRIDNet achieves competitive results with significantly reduced resource consumption. Extensive testing on both synthetic (AGWN) and real-world noisy images from the Urban100 and SIDD datasets demonstrates the model's robustness and efficiency. With PSNR scores of
34.23 on Urban100 and 39.50 on SIDD, SRIDNet delivers near stateof-the-art performance while maintaining the smallest model size and fastest inference time. This balance of efficiency and effectiveness positions SRIDNet as a promising solution for practical image denoising applications, particularly in environments with limited computational resources.

6.2 FUTURE SCOPE:

There's a growing need for lightweight models that can denoise images in real-time, especially for applications on mobile devices or embedded systems. Techniques like model quantization, pruning, and knowledge distillation could enhance efficiency without compromising performance. Deploying efficient models in environments with limited computational resources, such as medical devices, is a critical area of focus.

APPENDIX

```
from tensorflow import keras
from PIL import Image
import cv2
import tensorflow as tf
import numpy as np
import torch
import matplotlib.pyplot as plt
import os
```

Resize and Save Image array

```
dir = './SIDD_Small_sRGB_Only/Data/'

noisy_image_names = [x[2][1] for x in list(os.walk(dir))[1:]]
clear_image_names = [x[2][0] for x in list(os.walk(dir))[1:]]
paths = list(os.walk(dir))[0][1]

X = []
y = []

for i in range(len(paths)):

    img = np.array(Image.open(f"{dir}{paths[i]}/{noisy_image_names[i]}"))
    img2 = np.array(Image.open(f"{dir}{paths[i]}/{clear_image_names[i]}"))
    X.append(img)
    y.append(img2)

X_resize = [cv2.resize(x,(2576,1456)) for x in X]
y_resize = [cv2.resize(x,(2576,1456)) for x in y]

np.save('./X_resize',X_resize)
np.save('./y_resize',y_resize)
```

Image Patching

```
X_patches = []
y_patches = []
X_resize = np.load('./X_resize.npy')
y_resize = np.load("./y_resize.npy")
```

```
with torch.device("cuda"):
    for image in X_resize:
        extracted_patches =
tf.image.extract_patches(images=tf.expand_dims(image
,0),sizes=[1,112,112,1],strides=[1,112,112,1],padding='SAME',rates=[1,1,1,1])
        X_patches.extend(list(tf.reshape(extracted_patches,[-1,112,112,3])))
with torch.device("cuda"):
    for image in y_resize:
        extracted_patches =
tf.image.extract_patches(images=tf.expand_dims(image
,0),sizes=[1,112,112,1],strides=[1,112,112,1],padding='SAME',rates=[1,1,1,1])
        y_patches.extend(list(tf.reshape(extracted_patches,[-1,112,112,3])))
X_patches = np.array(X_patches)
y_patches = np.array(y_patches)

np.save('./X_patches',X_patches)
np.save('./y_patches',y_patches)
```

Train And Save Model

```
X_patches = np.load('./X_patches.npy')[:10000]/255
y_patches = np.load("./y_patches.npy")[:10000]/255


import tensorflow_hub as hub
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split


X_train, X_val, y_train, y_val = train_test_split(X_patches, y_patches,
test_size=0.3)


def pixel_mse_loss(y_true,y_pred):
  return tf.reduce_mean( (y_true - y_pred) ** 2 )

def PSNR(y_true,y_pred):
  mse=tf.reduce_mean( (y_true - y_pred) ** 2 )
  return 20 * log10(1 / (mse ** 0.5))

def log10(x):
  numerator = tf.math.log(x)
  denominator = tf.math.log(tf.constant(10, dtype=numerator.dtype))
  return numerator / denominator


def residual_block_gen(ch=32,k_s=3,st=1):
    model=tf.keras.Sequential([
        tf.keras.layers.Conv2D(ch,k_s,strides=(st,st),padding='same'),
```

```
        tf.keras.layers.LeakyReLU(),
        tf.keras.layers.Conv2D(ch,k_s,strides=(st,st),padding='same'),
        tf.keras.layers.LeakyReLU(),
    ])
    return model

def Upsample_block(x, ch=64, k_s=3, st=1):
    x = tf.keras.layers.Conv2D(ch,k_s, strides=(st,st),padding='same')(x)
    x = tf.nn.depth_to_space(x, 2) # Subpixel pixelshuffler
    x = tf.keras.layers.LeakyReLU()(x)
    return x

def get_enhancer():

    input_lr=tf.keras.layers.Input(shape=(112,112,3))
    input_conv=tf.keras.layers.Conv2D(32,5,padding='same')(input_lr)
    input_conv=tf.keras.layers.LeakyReLU()(input_conv)
    SRRes=input_conv

    for x in range(2):
        res_output=residual_block_gen()(SRRes)
        SRRes=tf.keras.layers.Add()([SRRes,res_output])
        SRRes=tf.keras.layers.Conv2D(32,5,padding='same')(SRRes)
        SRRes=tf.keras.layers.BatchNormalization()(SRRes)
        SRRes=tf.keras.layers.Add()([SRRes,input_conv])

    SRRes=Upsample_block(SRRes)
    output_sr=tf.keras.layers.Conv2D(3,5,activation='sigmoid',padding='same')(
SRRes)

    return tf.keras.Model(input_lr,output_sr)


def Generator():
    model = keras.Sequential([

        keras.layers.Conv2D(filters=128, kernel_size=3,
padding='same',activation='relu',input_shape=(224,224,3)),
        keras.layers.MaxPooling2D(2),
        keras.layers.Conv2D(filters=64, kernel_size=3,
padding='same',activation='relu'),
        keras.layers.MaxPooling2D(2),
        keras.layers.Conv2D(filters=64, kernel_size=3,
padding='same',activation='relu'),
        keras.layers.UpSampling2D(2),
        keras.layers.Conv2D(filters=128, kernel_size=3,
padding='same',activation='relu'),
        keras.layers.Conv2D(3,3,padding='same',activation='sigmoid')
```

```
    ])

    return model

inputs = layers.Input(shape=(112,112,3))
hidden = get_enhancer()(inputs)
hidden =  keras.layers.Conv2D(filters=128, kernel_size=3,
padding='same',activation='relu',input_shape=(224,224,3))(hidden)
hidden = keras.layers.MaxPooling2D(2)(hidden)
hidden1 = keras.layers.Concatenate()([inputs,hidden])
hidden = keras.layers.Conv2D(filters=64, kernel_size=3,
padding='same',activation='relu')(hidden1)
hidden = keras.layers.MaxPooling2D(2)(hidden)
hidden = keras.layers.Conv2D(filters=64, kernel_size=3,
padding='same',activation='relu')(hidden)
hidden = keras.layers.UpSampling2D(2)(hidden)
hidden = keras.layers.Concatenate()([hidden1,hidden])
hidden = keras.layers.Conv2D(filters=128, kernel_size=3,
padding='same',activation='relu')(hidden)
hidden = keras.layers.Conv2D(3,3,padding='same',activation='sigmoid')(hidden)
model = keras.Model(inputs,hidden)

model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.00001),loss='mse
',metrics=[PSNR])
model.summary()

model.fit(X_train,y_train,validation_data=(X_val,y_val),epochs=10,batch_size=1
6)

keras.models.save_model(model,'denoiserEnhancerSRGB.h5')
```

SIDD Testing

```
X_resize = np.load('./X_resize_test.npy')
y_resize = np.load('./y_resize_test.npy')


model = keras.models.load_model("./denoiserEnhancerSRGB.h5",{"PSNR":PSNR})


extracted_patches = tf.image.extract_patches(images=tf.expand_dims(X
,0),sizes=[1,112,112,1],strides=[1,112,112,1],padding='VALID',rates=[1,1,1,1])
X_patches = list(tf.reshape(extracted_patches,[-1,112,112,3]))


extracted_patches =
tf.image.extract_patches(images=tf.expand_dims(y,0),sizes=[1,112,112,1],stride
s=[1,112,112,1],padding='VALID',rates=[1,1,1,1])
y_patches = list(tf.reshape(extracted_patches,[-1,112,112,3]))
```

```
res = model.predict(np.array(X_patches))


patch_size = 112
n_col,n_row,n_channel = (1456, 2576, 3)
n_patch = n_row*n_col // (patch_size**2) #assume square patch

# patches =
tf.image.extract_patches(np.array([X]),sizes=[1,patch_size,patch_size,1],strid
es=[1,patch_size,patch_size,1],rates=[1, 1, 1, 1],padding='VALID')
patches = tf.reshape(res,[n_patch,patch_size,patch_size,n_channel])

rows = tf.split(patches,n_col//patch_size,axis=0)
rows = [tf.concat(tf.unstack(x),axis=1) for x in rows]

reconstructed_X = np.array(tf.concat(rows,axis=0))


patch_size = 112
n_col,n_row,n_channel = (1456, 2576, 3)
n_patch = n_row*n_col // (patch_size**2) #assume square patch

# patches =
tf.image.extract_patches(np.array([X]),sizes=[1,patch_size,patch_size,1],strid
es=[1,patch_size,patch_size,1],rates=[1, 1, 1, 1],padding='VALID')
patches = tf.reshape(y_patches,[n_patch,patch_size,patch_size,n_channel])

rows = tf.split(patches,n_col//patch_size,axis=0)
rows = [tf.concat(tf.unstack(x),axis=1) for x in rows]

reconstructed_y = np.array(tf.concat(rows,axis=0))


fig,axes = plt.subplots(nrows=2,ncols=3,figsize=(10,10))
axes[0][0].axis('off')
axes[0][0].imshow(X_patches[20])
axes[0][1].axis('off')
axes[0][1].imshow(X_patches[40])
axes[0][2].axis('off')
axes[0][2].imshow(X_patches[80])
axes[1][0].axis('off')
axes[1][0].imshow(res[20])
axes[1][1].axis('off')
axes[1][1].imshow(res[40])
axes[1][2].axis('off')
axes[1][2].imshow(res[80])
```

REFERENCES

[1] Kai Zhang, Wangmeng Zuo, Yunjin Chen, Deyu Meng, and Lei Zhang. Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. IEEE Transactions on Image Processing, 26:3142–3155, 2017.

[2] S. Guo, Z. Yan, K. Zhang, W. Zuo, L. Zhang: Toward Convolutional Blind Denoising of Real Photographs. In: CVPR (April 2019).

[3] S. Anwar, N. Barnes.: Real Image Denoising with Feature Attention. In: ICCV (March 2020).

[4] Stefan Roth and Michael J Black. Fields of experts. IJCV, 2009. 1, 2, 5, 6, 7

[5] Abdelrahman Abdelhamed, Stephen Lin, and Michael S Brown. A highquality denoising dataset for smartphone cameras. In CVPR, 2018. 5, 8

[6] Tobias Plotz and Stefan Roth. Benchmarking denoising algorithms with real photographs. arXiv preprint arXiv:1707.01313, 2017. 5, 7

[7] S. Cheng1, Y. Wang1, H. Huang, D. Liu, H. Fan and S. Liu.: NBNet: Noise Basis Learning for Image Denoising with Subspace Projection. In: CVPR (May 2021).

[8] Sara, U., Akter, M. and Uddin, M. (2019) Image Quality Assessment through FSIM, SSIM, MSE and PSNR—A Comparative Study. Journal of Computer and Communications, 7, 8-18. doi: 10.4236/jcc.2019.73002.

[9] J.-B. Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), June 2015.

[10] V. Bychkovsky, S. Paris, E. Chan, and F. Durand, "Learning photographic global tonal adjustment with a database of input/output image pairs," in Proc. 24th IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), 2011.

[11] K. Ma, Z. Duanmu, Q. Wu, Z. Wang, H. Yong, H. Li, and L. Zhang, "Waterloo Exploration Database: New challenges for image quality assessment models," IEEE Transactions on Image Processing, vol. 26, no. 2, pp. 1004-1016, Feb. 2017.

[12] Kai Zhang, Wangmeng Zuo, and Lei Zhang. Ffdnet: Toward a fast and flexible solution for cnn-based image denoising. TIP, 2018. 1, 2, 5, 6, 7, 8

[13] Kai Zhang, Yawei Li, Wangmeng Zuo, Lei Zhang, Luc Van Gool, and Radu Timofte. Plug-and-play image restoration with deep denoiser prior. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2021. 1, 2, 7, 8

[14] J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool, and R. Timofte, "SwinIR: Image Restoration Using Swin Transformer," arXiv preprint arXiv:2108.10257, 2021.

[15] Z. Yue, H. Yong, Q. Zhao, D. Meng, and L. Zhang, "Variational denoising network: Toward blind noise modeling and removal," in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, 2019.

[16] Z. Yue, Q. Zhao, L. Zhang, and D. Meng, "Dual adversarial network: Toward real-world noise removal and noise generation," in Proc. Eur. Conf. Comput. Vis. (ECCV), Aug. 2020.

[17] GeeksforGeeks, "Keras Input Layer," [Online]. Available: https://www.geeksforgeeks.org/keras-input-layer/..

[18] Keras, "Image Super-Resolution using an Efficient Sub-Pixel CNN," [Online]. Available: https://keras.io/examples/vision/super_resolution_sub_pixel/..

[19] GeeksforGeeks, "Introduction to Convolution Neural Network," [Online]. Available: https://www.geeksforgeeks.org/introduction-convolution-neural-network/..

[20] Analytics Vidhya, "Image Denoising Using Deep Learning," [Online]. Available: https://medium.com/analytics-vidhya/image-denoising-using-deep-learning-c2b19a3fd54..