NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

AN AUTONOMOUS INSTITUTION AFFILIATED TO VTU, BELGAUM, ACCREDIATED by NAAC ('A+' Grade)

YELAHANKA, BANGALORE-560064

A Project Report on

MINIMISING GAUSSIAN NOISE FROM REAL TIME CCTV IMAGES USING GAN

Submitted in partial fulfillment of the requirement for the award of the degree of

BACHELOR OF ENGINEERING

IN

INFORMATION SCIENCE AND ENGINEERING

By

Adyapadi Suraj	1NT20IS012
Akshay Prashant Hegde	1NT20IS015
Pranava Aithal K S	1NT20IS112

Under the Guidance of,

Dr. Manoj Kumar M V

Associate Professor

Department of Information Science and Engineering

Nitte Meenakshi Institute of Technology, Bengaluru - 560064

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

(Accredited by NBA Tier-1)

2023-24

NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO VTU, BELGAUM)

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Accredited by NBA Tier-1

CERTIFICATE

Certified that the project work entitled "**Minimising Gaussian noise from real time CCTV images using GAN**" carried out by Mr **Adyapadi Suraj**, USN **1NT20IS012**, Mr. **Akshay Prashant Hegde**, USN **1NT20IS015**, Mr. **Pranava Aithal K S**, USN **1NT20IS112**, bonafide students of **Department of Information Science and Engineering in Nitte Meenakshi Institute of Technology** in partial fulfillment for the award of Bachelor of Engineering in **Information Science and Engineering** of the Visvesraya Technological University, Belgaum during the year **2023 - 24**. It is certified that all corrections/suggestions indicated for Internal Assessment for progess-I have been incorporated in the Report deposited in the departmental library.

Signature of the Guide | Signature of the HOD | Signature of the Principal

External Viva

Name of the examiners | Signature with date

1.

2.

ABSTRACT

This report describes a system that uses generative adversarial networks (GANs) to eliminate gaussian noise from CCTV of images. Unwanted noise like gaussian noise frequently degrades the quality of images and makes them more difficult to interpret. In our approach, a discriminator network is used to direct the training of a generator network, whose job it is to produce denoised images from noisy inputs. This framework includes operations like picture enhancement, noise reduction, and evaluation with metrics like the structural similarity index (SSIM) and peak signal-to-noise ratio (PSNR). The resultant denoised images show enhanced visual quality and have potential uses in image analysis and computer vision.

ACKNOWLEDGMENT

The satisfaction and euphoria that accompanies the successful completion of any project would be incomplete without the mention of people who made it possible and whose constant encouragement and guidance has been a source of inspiration throughout the work.

We would like to express our deepest sense of gratitude for our beloved Director **Prof. N R Shetty** for providing us a great learning environment. We humbly extend our immense gratitude for our beloved principal **Dr. H.C. Nagaraj** for continuous support and encouragement. We cannot thank enough our dear Head of Department **Dr. Mohan S G** for the continuous efforts in creating a competitive environment in the department and encouragement throughout the course of completion of the project.

With profound gratitude, we express our indebtedness to our guide **Dr. Manoj Kumar M V**, whose dedication to the project, Knowledge and references got our job done in the shortest time frame possible. Madam, we would like to sincerely thank you for the quality time you spent with us to clarify our doubts and for guiding us at each and every step toward completion of the project.

We are also thankful to the entire Department of Information Science and Engineering for the co-operation and suggestion without which this project would not have been a successful one. We also thank all our friends who directly or indirectly helped us a lot in this project completion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ubiquity of undesired noise in real-world scene images presents an enduring obstacle in today's digital landscape. Traditional denoising methods often struggle to distinguish between noise and essential image elements, resulting in the loss of critical information or the introduction of aberrations. Generative Adversarial Networks (GANs) emerge as a promising solution to this dilemma.[1] This initiative seeks to explore the capacity of GANs to enhance the visual quality of realworld scene images by mitigating unwanted noise

Real-world scene images frequently suffer from the intrusion of unwanted noise, degrading visual clarity and complicating interpretation. In response to this challenge, we propose a framework harnessing Generative Adversarial Networks (GANs) to diminish undesired noise and improve the visual fidelity of real scene images[2]. Our framework comprises two principal components: noise reduction and image enhancement. A discriminator network distinguishes between pristine and generated images, while a generator network undergoes adversarial training to transform noisy images into clean renditions.

Gaussian noise, which has a constant power spectral density and is sometimes called white noise, has a few unique characteristics. In essence, it's a collection of haphazard tiny blips or specks incorporated into the original signal. It's like a series of unexpected events dispersed throughout, as each blip exists independently of the others. It is present almost everywhere, including in CCTV image and radio broadcasts. Since they are additive in nature, the noisy signal is usually produced by adding them to the original signal.[3][4] It is a good model for many kinds of random disturbances found in both natural and artificial systems because of this additive feature. Because of their widespread use in practical settings and wellunderstood statistical characteristics, they serve as a common and helpful model in signal processing and image processing applications.

1.1 Issues and Challenges

- **Temporal coherence and consistency**: Videos are sequences of frames, and maintaining temporal coherence and consistency in denoising is crucial for producing visually pleasing results.
- **High-frequency and fine-detail preservation**: Noise removal algorithms should aim to preserve high-frequency details and fine structures in videos while reducing noise[3].

- **Training data requirements**: GANs typically require a large amount of training data to learn the underlying patterns and distributions accurately.
- **Generalization to different noise types and levels**: GANs trained on specific noise types or noise levels may struggle to generalize well to unseen noise types or levels during inference.
- **Real-world applicability and practical constraints**: Deploying GAN-based denoising algorithms in real-world video processing applications often involves additional practical constraints, such as limited computational resources, energy efficiency, and real-time performance[6].

1.2 Problem Statement

The most common problem faced by the people is of the quality of images and videos. To solve this problem many traditional methods have been introduced and implemented. Some of the models like Median filter, Adaptive filtering and Total Variation (TV) denoising work to some extent but are not completely effective. To overcome this we are using GAN as GAN-based models have shown superior performance in effectively reducing noise while preserving important details and structures in videos. These models learn the underlying noise distribution and clean frame characteristics, allowing them to effectively denoise videos even in real-world scenarios.

1.3 Objective

- To design and implement a GAN model that can effectively remove Gaussian noise from CCTV images while preserving the image details and features.
- To investigate the effect of different hyperparameters on the performance of the GAN model, such as the batch rate and number of training epochs.
- To demonstrate the practical applications of the proposed GAN-based denoising method in real-world scenarios, such as surveillance and photography.
- To build a mobile application and implement the working of the GAN model for removing gaussian noise in videos.

1.4 Organization Of Report

Report is organised in following way

- Chapter 2: Contains literature review.
- Chapter 3: Contains requirement specification.
- Chapter 4: Contains methodology and the architecture design which we are using in our proposed project.
- Chapter 5: Contains the description of our dataset.
- Chapter 6: Contains Implementation with code snippets of our project.
- Chapter 7: Contains Testing information.
- Chapter 8: Contains Result and Discussion.
- Chapter 9: Contains Conclusion and future scope.

Summary

The content highlights noise challenges in real-world images and suggests GANs as a solution. It proposes a framework for noise reduction and image enhancement, facing challenges like temporal coherence preservation and real-world constraints. The objective is to implement a GAN model for CCTV image denoising, exploring hyperparameters and practical applications, including a mobile video denoising app.

Chapter 2

Literature Review

Since the inception of SRCNN, notable progress has been achieved in the realm of image super-resolution. Generative adversarial networks (GANs) have gained traction as a favored method for loss supervision, aiming to align solutions more closely with the natural image distribution and yield visually appealing outcomes. Nonetheless, many existing techniques lean on bicubic downsampling kernels and often grapple with generating precise outputs when confronted with real-world images.[5] Recent strides in image restoration methodologies have begun integrating reinforcement learning and GANs to confront these hurdles. Blind super-resolution (SR) has attracted considerable attention in the research domain. One category of approaches centers on explicit representations of degradation, typically encompassing two primary facets: degradation prediction and conditional restoration.[6] These methods may execute the two facets independently or iteratively, relying heavily on predetermined representations of degradation, such as degradation types and levels. Nonetheless, these approaches frequently overlook intricate real-world degradations and may introduce artifacts if degradation estimations prove inaccurate.

An alternative approach entails acquiring or generating training pairs that closely mirror real data, followed by training a unified network to tackle blind super-resolution. Obtaining such training pairs typically involves dedicated cameras and demands meticulous alignment. Alternatively, these pairs can be gleaned from unpaired data using cycle consistency loss. Another avenue involves synthesizing the pairs by estimating blur kernels and extracting noise patches.[7] However, the data collected is constrained to degradation associated with specific cameras, limiting its applicability to other real-world images. It proves challenging to accurately capture and analyze subtle deteriorations using data not directly paired with original images, often yielding unsatisfactory results. Image denoising techniques employing Generative Adversarial Networks (GANs) have emerged as powerful tools across diverse domains, addressing challenges posed by noise in various imaging modalities. One such application discussed in Zhong et al.'s paper [8] focuses on blind denoising of fluorescence microscopy images, crucial in life sciences but often afflicted by strong noise due to formation and acquisition constraints. Their proposed blind global noise modeling denoiser (GNMD), utilizing a GAN to simulate image noise globally, outperforms existing methods in suppressing background noise, thereby facilitating downstream image segmentation tasks. Another notable contribution by Zhiping et al. [9] introduces a novel GAN architecture for texture-preserving image denoising. Their approach involves a generator network trained using a newly devised loss function to accurately measure the disparity between the data distribution of clean and denoised images.

2.1 Existing Literature

The paper [10] discusses how algorithms for picture denoising have developed to maximize image quality as determined by human visual perception. Less research has been done, though, on the use of picture denoising to improve the performance of computer vision algorithms using the denoised image. Regarding the specific application of picture retrieval from a dataset, they take into account the issue of image denoising for Gaussian noise. The success of image retrieval is our definition of picture quality, and they develop a deep convolutional neural network (CNN) to predict this quality. After that, this network is cascaded with a deep CNN made specifically for image denoising, enabling the denoising CNN to be optimized to improve retrieval performance. They may connect denoising and the retrieval problem using this framework. Through studies using noisy photos of buildings from the Oxford and Paris building datasets, they demonstrate that using this strategy results in enhanced mean average precision when compared to using denoising approaches that are unaware of the objective of image retrieval.

The paper [11] discusses how a lot of picture restoration techniques have been built on patch processing. Decomposing the target image into totally overlapping patches, restoring each one independently, and then averaging the results together are the main concepts. This idea has been shown to be quite effective, frequently producing denoising, inpainting, deblurring, segmentation, and other applications' state-of-the-art outcomes. Although the aforementioned method is effective, it has a serious flaw: the prior is applied on intermediate (patch) results rather than the final product, and this usually shows as visual artifacts. This issue was the inspiration behind Zoran and Weiss' expected patch log likelihood (EPLL) technique.

Their approach applies the previous to the patches of the final image, which causes a diminishing-effect iterative restoration. In this study, they suggest using a multi-scale prior to further improve and expand the EPLL. On various scale patches derived from the target image, our technique applies the same prior.[12] Even though the treated patches are of the same size, subsampling causes their footprint in the final image to differ. The fact that a local (patch-based) prior is acting as a model for a global stochastic phenomena is another flaw in patch-based restoration algorithms, which is addressed by our technique.

By limiting ourselves to the basic Gaussian situation, contrasting the aforementioned algorithms, and demonstrating a definite benefit to the suggested method, they provide motivation for the usage of the multi-scale EPLL. They next use image denoising, deblurring, and super-resolution to illustrate our algorithm's performance, demonstrating an improvement both qualitatively and quantitatively.

The paper[13] , they jointly denoise ultrasound pictures using the brushlet-based block matching 3D (BM3D) approach. In order to categorize the images based on similarities, they divide each image into several blocks. Then, clustered blocks with shared characteristics create a 3D image volume. Brushlet thresholding is used for each volume to get rid of frequency domain noise. Following individual filtering, the volumes are combined and globally recreated. They apply our denoising strategy to synthetic images tainted with additive or multiplicative noise in order to assess the effectiveness of our technology. The outcomes demonstrate that, in comparison to other methods, our method may achieve good denoising performance. Ultrasound scans of the unborn heart and other organs are used to assess our procedure.

The paper[14] , it is suggested to use the DCT instead of the wavelet transform in order to overcome the issue that the denoising performance abruptly decreases when the noise standard deviation surpasses 40. They contend in this response that this replacement is not required and that the issue can be resolved by changing a few numerical settings. This parameter adjustment strategy is also presented here. Experimental findings show that the suggested adjustment outperforms the original approach for severe noise in terms of peak signal-to-noise ratio and subjective visual quality.

This paper[15] The topic of sparse representation of signals has gained popularity in recent years. Signals are characterized by sparse linear combinations of prototype signal-atoms taken from an overcomplete lexicon. Compression, regularization in inverse problems, feature extraction, and other uses for sparse representation are just a few. Recent research in this area has mostly focused on the analysis of pursuit algorithms that break down signals according to a defined vocabulary.

Either choosing one from a predetermined list of linear transformations or modifying the dictionary to a set of training signals can be used to create dictionaries that more closely match the aforementioned model. Though both of these approaches have been taken into account, there are still many unanswered questions. They suggest a novel technique in this study for modifying dictionaries to produce sparse signal representations. Under stringent sparsity restrictions, they look for the dictionary that, given a set of training signals, leads to the best representation for each member of the set[16].

They introduce a novel approach for generalizing the K-means clustering procedure, the K-SVD algorithm. K-SVD is an iterative method that alternates between updating the dictionary atoms to better fit the data and sparse coding of the examples based on the current dictionary.[1] Convergence is sped up by combining an update of the sparse representations with an update of the dictionary columns. Any pursuit strategy, such as basis pursuit, FOCUSS, or matching pursuit, can be used with the K-SVD algorithm due to its adaptability. They examine this approach and present the outcomes of applications on actual image data as well as on simulated tests.

This paper [17] This study uses principal component analysis (PCA) and local pixel grouping (LPG) to provide an effective method for image denoising. A pixel and its closest neighbors are modelled as a vector variable to better preserve the local structure of the picture, and the training samples for this vector variable are chosen from the local window using block matching-based LPG. In order to maintain the local characteristics of the picture following coefficient shrinkage in the PCA domain to reduce the noise, a technique like this LPG ensures that only sample blocks with identical contents are utilized in the local statistics computation for PCA transform estimate. To further enhance the denoising performance, the LPG-PCA process is iterated a second time, and in the second step, the noise level is adaptively changed. In comparison to state-of-the-art denoising algorithms, experimental results on benchmark test photos show that the LPG-PCA approach delivers very competitive denoising performance, especially in image fine structure preservation.

The paper[18] Traditionally, image denoising algorithms are tested on images tainted by artificial i.i.d. Gaussian noise in the absence of true ground truth data. By creating a system for measuring denoising algorithms on actual images, they hope to avoid this unrealistic setting. They take two photographs side by side, one with a low ISO image that is practically noise-

free, and the other with a higher ISO and properly regulated exposure duration. The ground truth must be determined carefully after processing. They adjust for spatial misalignment, deal with exposure parameter errors using a novel heteroscedastic Tobit regression model-based linear intensity transform, and eliminate any remaining low-frequency bias caused, for example, by slight variations in lighting.[4] The Darmstadt Noise Dataset (DND), a novel benchmark dataset, is then recorded using consumer cameras with various sensor sizes. One intriguing conclusion is that BM3D clearly outperforms other recent approaches on images with real noise, even though they all perform well on synthetic noise. Our benchmark outlines plausible evaluation scenarios that drastically depart from those that are frequently employed in the scientific literature.

2.2 Research Gaps

- Temporal coherence and consistency: Videos are sequences of frames, and maintaining temporal coherence and consistency in denoising is crucial for producing visually pleasing results.
- High-frequency and fine-detail preservation: Noise removal algorithms should aim to preserve high-frequency details and fine structures in videos while reducing noise[7].
- Training data requirements: GANs typically require a large amount of training data to learn the underlying patterns and distributions accurately.
- Generalization to different noise types and levels: GANs trained on specific noise types or noise levels may struggle to generalize well to unseen noise types or levels during inference.
- Real-world applicability and practical constraints: Deploying GAN-based denoising algorithms in real-world video processing applications often involves additional practical constraints, such as limited computational resources, energy efficiency, and real-time performance[10].

Summary

The literature review explores advancements in image super-resolution and denoising, emphasizing the rise of GANs and reinforcement learning. Blind super-resolution methods face challenges in predicting degradation accurately, while GAN-based denoising techniques prove effective across various domains. Existing research highlights the need for benchmarking denoising algorithms with real-world images and addresses gaps in temporal coherence, high-frequency preservation, and practical applicability.

Table 2.1: Literature Review

Work	Author	Methodologies	Advantages	Disadvantages
Image Denoising for Image Retrieval by Cascading a Deep Quality Assessment	BV Somasundaran Rajiv Soundararajan Soma Biswas	QA CNN	Improved retrieval performance. Effective denoising and quality assessment. Optimization specifically for image retrieval.	Training process relies on pre-trained convolutional layers. Performance dependent on choice of retrieval algorithm and dataset used.
Multi-Scale Patch-Based Image Restoration	Vardan Papyan Michael Elad	EPLL	Multi-scale image restoration algorithm that imposes a local low dimensional prior on patches of different scales.	Algorithm can be sensitive to the choice of the local model. Algorithm may not remove noise from images with high levels noise.
BM3D-Based Ultrasound Image Denoising via Brushlet Thresholding	Yu Gan Elsa Angelini Andrew Laine Christine Hendon	Block Matching Brushlet Thresholding	Method is evaluated on synthetic, real ultrasound images, demonstrating its applicability.	The investigation of applying BM3D ultrasound images are limited due to the challenges posed by low SNR and texture.
Image denoising by sparse 3-d transform-domain collaborative filtering	Dabov, K. Foi, A. Katkovnik, V. Egiazarian K	3-D transform ThresholdingInverse 3-D transform	Method demonstrates high effectiveness in denoising images. Sparse 3-D transform to represent image patches.	Does not take into account the advancements and newer techniques developed in the field of image denoising in recent years.
An algorithm for designing overcomplete dictionaries for sparse representation.	Aharon, M. Elad, M. Bruckstein A.	Sparse coding Atom selection	K-SVD algorithm proposed in the paper provides an efficient and effective approach for learning overcomplete dictionaries.	K-SVD algorithm has shown empirical success in various applications, does not provide theoretical guarantees convergence or optimality.
Two-stage image denoising by principal component analysis with local pixel grouping.	Zhang, L. Dong, W. Zhang, D. Shi G.	Principal Component Analysis (PCA) Non-Local Means Filtering	The proposed method achieves a good balance between denoising performance and computational efficiency.	Evaluation of the method, but it may have limitations in terms of the diversity of the tested datasets and the evaluation metrics used.
Benchmarking denoising algorithms with real photographs.	Plotz, T. Roth, S.	Execution Performance evaluation	Addresses the need for benchmarking denoising algorithms using real photographs, which is crucial for assessing the performance of algorithms in practical scenarios.	Evaluates a limited number of denoising algorithms, which may not represent the entire spectrum of available techniques. Does not provide details about individual denoising algorithms evaluated.

Chapter 3

Requirement Specification

A requirement specification, which is a set of documented requirements to be satisfied by a material, design, product, or service. A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

3.1 Functional Requirements

Functional requirements specify the specific functions, features, and behaviors that a software system should exhibit. They describe what the system should do in terms of inputs, outputs, and processing. Functional requirements define the expected behavior, interactions, and constraints on the system's functionality. They capture the desired functionalities and capabilities of the software, providing a clear understanding of how the system should operate and respond to user actions or external events.

3.1.1 Hardware

Hardware requirements are the specific hardware components and resources needed to support a task or application. For noise removal using GANs, hardware requirements include processing power (CPU/GPU), memory (RAM), storage capacity (hard drives or SSDs), network connectivity, and potentially parallel computing capabilities. These requirements ensure that the hardware can handle the computational demands and data storage needs of GAN-based noise removal efficiently.

- **CPU/GPU**: Powerful processors like NVIDIA GeForce RTX or AMD Radeon RX GPUs for computational tasks. Alternatively, high-end CPUs such as Intel Core i9 or AMD Ryzen Threadripper for CPU-based processing.

- **Memory**: Adequate RAM (16GB or higher) to handle GAN models and data. For larger datasets or complex models, 32GB or more may be required.

- **Storage**: Fast and spacious storage (e.g., SSDs) for datasets, models, and checkpoints. Solid-state drives (SSDs) with NVMe interface offer high-speed read/write performance, beneficial for loading large datasets and storing model checkpoints.

- **Network Connectivity**: Reliable and high-speed network connection for distributed training setups. A gigabit Ethernet connection or Wi-Fi 6 (802.11ax) for fast data transfer between networked devices.

- **Parallel Computing**: Support for parallel computing architectures like NVIDIA CUDA for GPU acceleration or multi-core CPU processing to improve training speed and efficiency.

3.1.2 Software

Software requirements are a concise description of what a software system should do and how it should behave. They capture the necessary functionalities and qualities that the software needs to have in order to meet the users' needs and achieve the desired goals. Software requirements act as guidelines for the development team, providing a clear understanding of what needs to be built and ensuring that the software meets the intended purpose.

- **Deep Learning Frameworks**: Comprehensive support for deep learning frameworks such as TensorFlow, PyTorch, or MXNet, enabling seamless development and deployment of GAN-based noise removal models.

- **Image Processing Libraries**: Integration with image processing libraries like OpenCV or Pillow for data preprocessing, augmentation, and visualization tasks.

- **Model Optimization Tools**: Tools and libraries for model optimization, such as TensorFlow Lite or ONNX Runtime, to deploy models efficiently on edge devices or in production environments.

- **Development Environments**: Support for popular integrated development environments (IDEs) like Jupyter Notebook, PyCharm, or Visual Studio Code for model development, debugging, and experimentation.

- **Version Control Systems**: Integration with version control systems like Git for collaborative development, code management, and tracking changes across multiple contributors.

- **Containerization Platforms**: Compatibility with containerization platforms like Docker for packaging and deploying applications with dependencies, ensuring consistency and reproducibility across different environments.

3.2 Non-Functional Requirements

Non-functional requirements define how a software system should perform, rather than what it should do. They encompass characteristics such as performance, usability, reliability, security, scalability, availability, maintainability, compatibility, and compliance with legal and regulatory standards. These requirements ensure that the system meets specific quality standards and behaves as expected in various aspects beyond its functional capabilities.

- **Response Time**: The system should respond to user interactions within 500 milliseconds to provide a seamless user experience.

- **Usability**: The system should have an intuitive and user-friendly interface, with clear navigation and minimal learning curve for users to perform tasks efficiently.
- **Security**: The system should enforce strong encryption protocols (e.g., AES-256) to protect user data and ensure confidentiality during data transmission and storage.
- **Scalability**: The system should be able to handle a growing number of users and concurrent requests by scaling resources dynamically, either vertically (adding more hardware resources) or horizontally (distributing workload across multiple servers).
- **Availability**: The system should have at least 0.99 uptime to ensure accessibility and reliability for users, with scheduled maintenance and downtime communicated in advance to minimize disruption.
- **Maintainability**: The system's codebase should be well-documented, following coding standards and best practices to facilitate ease of maintenance, debugging, and troubleshooting by development teams.
- **Compatibility**: The system should be compatible with a wide range of web browsers (e.g., Chrome, Firefox, Safari) and devices (desktops, laptops, tablets, smartphones) to ensure accessibility and consistent user experience across different platforms.
- **Compliance**: The system should comply with relevant legal and regulatory standards, such as GDPR, HIPAA, or PCI DSS, regarding data privacy, security, and confidentiality, to protect user rights and mitigate legal risks.

Chapter 4

Methodology and Architecture Design

4.1 Methodology

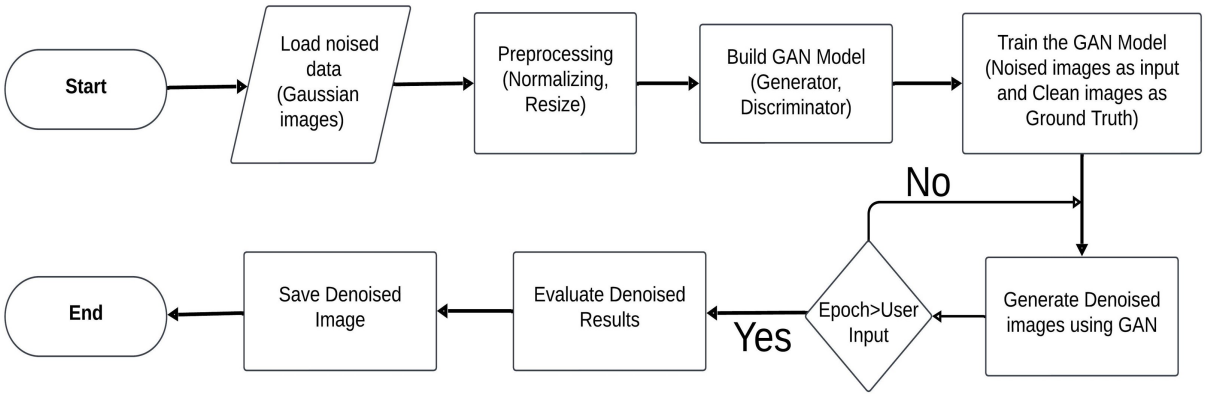


Figure 4.1: Methodology

The procedural steps for an image denoising project using a Generative Adversarial Network (GAN) are outlined as follows:

1. Loading Noised Data (Gaussian Images): Initially, the noisy image data, comprising images affected by Gaussian noise, is loaded. These noisy images are often sourced from various datasets or captured from real-world scenarios where noise is present due to factors such as low-light conditions or sensor limitations. Before being fed into the Generative Adversarial Network (GAN) model, the noisy images undergo preprocessing stages to ensure consistency and compatibility. These preprocessing stages commonly include normalization to scale pixel values to a standard range (e.g., [0, 1]) and resizing to a uniform dimension to facilitate efficient processing. Initially, noisy image data, affected by Gaussian noise, is loaded. These images, sourced from datasets or real-world scenarios, undergo preprocessing like normalization and resizing.

2. Constructing GAN Model (Generator, Discriminator): Next, the GAN model is constructed, consisting primarily of two components: the Generator and the Discriminator. The Generator is responsible for synthesizing denoised images from the noisy input, effectively

learning to transform noisy images into their clean counterparts. On the other hand, the Discriminator acts as a discerning critic, distinguishing between real (clean) images and the generated (denoised) ones. Both the Generator and Discriminator are typically implemented as deep neural networks, with architectures designed to capture and model the complex relationships present in image data. The GAN model consists of the Generator, which synthesizes denoised images, and the Discriminator, which distinguishes real from generated images.

3. Training the GAN Model (Noised Images as Input and Clean Images as Ground Truth): The GAN model is then trained using a dataset consisting of pairs of noisy images (input) and their corresponding clean images (ground truth or target output). During training, the Generator and Discriminator engage in an adversarial learning process, where the Generator aims to produce denoised images that are indistinguishable from real clean images, while the Discriminator strives to accurately classify between real and fake (denoised) images. This adversarial training dynamic leads to the refinement of both the Generator's denoising capabilities and the Discriminator's discriminative abilities over successive training epochs. Using pairs of noisy and clean images, the GAN model undergoes adversarial learning, refining the Generator's denoising and Discriminator's discriminative abilities.

4. Iterative Training and User Input: The training process is typically iterative, with the model being trained over multiple epochs. At the conclusion of each epoch, the performance metrics of the GAN model are evaluated, and the user may provide input regarding whether to continue training for additional epochs or halt the training process. This iterative approach allows for the fine-tuning of model parameters and the gradual improvement of denoising performance over time. Additionally, techniques such as learning rate scheduling and early stopping may be employed to optimize the training process and prevent overfitting. The model is iteratively trained over epochs, with user input guiding additional training or halting. Techniques like learning rate scheduling and early stopping are employed.

5. Generating Denoised Images using GAN: Once the training is complete and the GAN model has converged, it can be deployed to generate denoised images from new, unseen noisy input images. The Generator component of the trained model is utilized to transform noisy images into their denoised counterparts, leveraging the knowledge and patterns learned during the training process. This generation process is typically efficient and can be applied to batch processing of multiple noisy images simultaneously. The trained GAN model generates denoised images from new noisy inputs efficiently, leveraging learned patterns.

6. Evaluating Denoised Results: The generated denoised images are then subjected to evaluation to assess their quality and fidelity compared to ground truth clean images. Common evaluation metrics include the Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM), and Mean Squared Error (MSE). These metrics provide quantitative measures of the denoising performance, allowing for objective assessment and comparison of different models or approaches. Additionally, qualitative visual inspection of the denoised images may also be performed to identify any artifacts or imperfections introduced during the denoising process. Evaluation metrics like PSNR, SSIM, and MSE quantify denoising performance. Visual inspection identifies any imperfections.

7. Saving Denoised Images: Following evaluation, the denoised images deemed satisfactory can be saved for further analysis, visualization, or application in downstream tasks. Saving the denoised images in a suitable format ensures their preservation and accessibility for future use. Additionally, metadata such as the timestamp of denoising, associated evaluation metrics, and any relevant contextual information may be stored alongside the denoised images to facilitate traceability and reproducibility of results. Satisfactory denoised images are saved for further analysis or application, along with associated metadata.

In conclusion, this methodology encompasses loading noisy data, preprocessing, constructing and training a GAN model, generating denoised images using the trained model, evaluating the outcomes, and preserving the denoised images. It incorporates a loop for iterative training of the GAN model across multiple epochs and allows for user interaction to govern the training process. The architecture design emphasizes the roles of the Generator and Discriminator models, the importance of noisy images, and the loss function in achieving denoising perfection through adversarial training.

4.2 Architecture Design

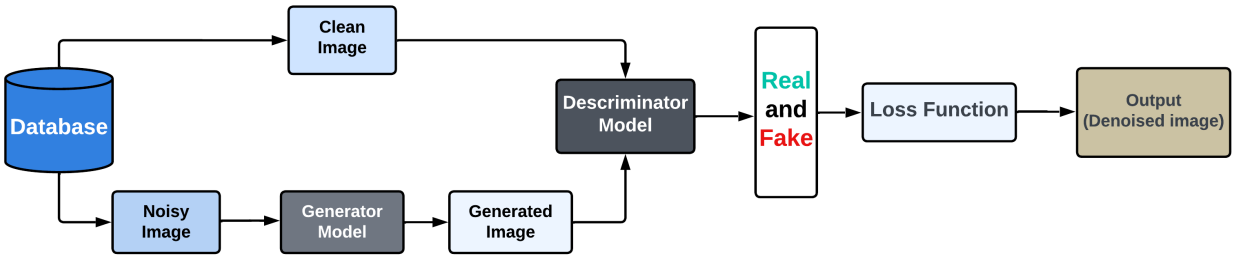


Figure 4.2: Architecture Design

Diagram 4.2 provides a detailed representation of the complex dynamics within a Generative Adversarial Network (GAN), a cutting-edge deep learning framework widely utilized for image denoising and image-to-image translation tasks. The process initiates with a comprehensive dataset comprising noisy or corrupted images, serving as the foundational input for the GAN architecture. These Noisy Images undergo processing within the Generator Model, a critical component tasked with generating pristine or denoised versions termed as Generated Images. Concurrently, the Discriminator Model, another integral element, assumes the role of a discerning evaluator, scrutinizing two distinct inputs: the generated images from the Generator and the unblemished images sourced directly from the dataset.

In our endeavor to develop a Generative Adversarial Network (GAN) model tailored for image denoising, each component plays a crucial role in the intricate dance of generating clean, artifact-free images from their noisy counterparts. Let's delve deeper into each aspect:

4.2.1 Noisy Images

At the genesis of our process lie the noisy images, which serve as the starting point for our denoising journey. These images, marred by Gaussian noise, reflect the real-world challenges of imperfections and distortions commonly encountered in image acquisition and processing scenarios. By working with such noisy data, our model grapples with the complexities inherent in real-world image denoising tasks, preparing it to handle a diverse array of noisy input conditions.

4.2.2 Generator Model

As the creative force driving the denoising process, the Generator model bears the weighty responsibility of transforming noisy input images into clean, visually appealing counterparts. Operating as a deep neural network, the Generator harnesses its vast array of learnable parameters to decipher the underlying patterns and structures within the noisy data. Through a series of convolutional and upsampling layers, the Generator meticulously refines its output, iteratively sculpting denoised images that closely resemble their pristine originals. Through a series of convolutional layers, the Generator methodically extracts essential features from the noisy input, capturing the essence of the underlying content while filtering out undesirable noise. These layers serve as the backbone of the denoising process, enabling the model to discern subtle details and distinguish between signal and noise. Moreover, the Generator employs upsampling layers to gradually enhance the resolution and fidelity of its output. This iterative refinement process is akin to a skilled artist meticulously sculpting a masterpiece, with each layer contributing to the final composition. By strategically expanding the dimensions of the feature maps, the Generator breathes life into the denoised images, imbuing them with clarity and realism.

4.2.3 Discriminator Model

Counterbalancing the Generator's creative prowess is the discerning eye of the Discriminator model. Acting as a vigilant guardian, the Discriminator scrutinizes both real, clean images and the denoised outputs from the Generator, honing its ability to distinguish between the two. Through adversarial training, the Discriminator hones its discriminatory acumen, sharpening its ability to detect even the subtlest deviations between genuine and generated images. This adversarial interplay fosters a dynamic feedback loop, propelling both the Generator and Discriminator towards greater levels of sophistication and performance. This adversarial interplay fosters a dynamic feedback loop, propelling both the Generator and Discriminator towards greater levels of sophistication and performance. As the Discriminator becomes more adept at discerning real from fake images, it provides crucial feedback to the Generator, pushing it to refine its denoising process further. At the heart of this denoising framework lies the Loss Function, a mathematical construct that quantifies the disparity between the Discriminator's assessments and ground truth labels. By quantifying this dissonance, the Loss Function guides the iterative optimization process, nudging the model towards configurations that yield ever more faithful denoised images. Through the judicious manipulation of this loss metric, our model fine-tunes its parameters, inching closer to the elusive ideal of noise-free imagery.

The primary objective of the Discriminator is to meticulously distinguish between Real Images (i.e., those originating from the dataset) and Fake Images (i.e., those artificially generated

by the Generator). Facilitating this intricate process is the Loss Function component, pivotal for quantifying the disparity between the Discriminator's output and the ground truth labels (Real or Fake), thereby enabling precise error calculation.

4.2.4 Loss Function

Driving the quest for denoising perfection is the Loss Function, a mathematical construct that quantifies the disparity between the Discriminator's assessments and ground truth labels. By quantifying this dissonance, the Loss Function guides the iterative optimization process, nudging the model towards configurations that yield ever more faithful denoised images. Through the judicious manipulation of this loss metric, our model fine-tunes its parameters, inching closer to the elusive ideal of noise-free imagery. In this symbiotic dance between data, models, and optimization algorithms, our GAN model for image denoising embodies a harmonious fusion of artistry and science. By embracing the imperfections inherent in noisy data and leveraging the power of adversarial training, our model emerges as a formidable tool for enhancing the visual fidelity of images in the face of Gaussian noise and real-world challenges.

Its primary objective is to meticulously distinguish between Real Images (i.e., those originating from the dataset) and Fake Images (i.e., those artificially generated by the Generator). Facilitating this intricate process is the Loss Function component, pivotal for quantifying the disparity between the Discriminator's output and the ground truth labels (Real or Fake), thereby enabling precise error calculation. This computed error is subsequently leveraged to orchestrate iterative updates to the weights of both the Discriminator and Generator models through the mechanism of backpropagation, thereby facilitating continual enhancement of their performance and efficacy. At its core, the overarching objective of this adversarial training process is twofold: to empower the Generator in generating images that seamlessly mimic the authentic dataset, thus effectively deceiving the Discriminator, while concurrently refining the Discriminator's discernment capabilities to accurately differentiate between real and generated images.

Summary

The methodology for image denoising with GANs involves loading noisy data, constructing and training the GAN model, generating denoised images, evaluating results, and saving them. The iterative training loop refines the model over time. The architecture design emphasizes the roles of the Generator and Discriminator models, the importance of noisy images, and the loss function in achieving denoising perfection through adversarial training.

Chapter 5

Dataset Description

Captivating subjects for investigation in the fields of image processing and generative art are artistic and pointillistic portrait images. This dataset contains 250 high-resolution photos that show a variety of portraits of people with different demographics, such as men, women, and kids. Each image features a special fusion of visual complexity and artistic expression, painstakingly created in a pointillism-inspired style. But each portrait has been purposefully tainted with Gaussian noise to mimic real-world conditions and difficulties in image capture and transmission. The deliberate degradation of the dataset imparts a realistic touch, rendering it a perfect tool for examining and refining denoising algorithms and methodologies. Furthermore, this dataset provides consistency, compatibility, and fidelity for researchers and practitioners alike in their exploration of image processing, style transfer, and generative art tasks due to its standardized dimensions of 256 x 256 pixels and lossless Portable Network Graphics (PNG) format.

Table 5.1: Dataset description table

Dataset	**Description**
Number of Images	250
Image Type	Artistic/pointillistic portrait images
Image Contents	Faces/Full-body portraits of different people (men, women, children)
Image Style	Composed of Gaussian Noise on the original image
Intended Use	Image processing, style transfer, or generative art tasks.
Image Dimension	256 x 256
File Format	PNG

1. **Number of Images:** The dataset comprises a total of 250 images, providing a substantial corpus for experimentation and analysis. This sizeable dataset enables robust model training and validation across a diverse range of scenarios and applications.

2. **Image Type:** The images in the dataset are categorized as artistic or pointillistic portrait images, showcasing a rich variety of artistic styles and techniques. The incorporation of pointillism adds a layer of complexity and visual interest to the portraits, making them ideal subjects for studying image processing algorithms and techniques.

3. **Image Contents:** The dataset includes a diverse range of subjects, featuring faces and full-body portraits of various individuals spanning different demographics. This diversity ensures comprehensive coverage across age, gender, and ethnicity, making the dataset suitable for addressing a wide array of research questions and applications in image processing.

4. **Image Style:** Each image in the dataset is intentionally corrupted by the addition of Gaussian noise, mimicking real-world scenarios where images may be subject to various forms of degradation during acquisition or transmission. This intentional corruption adds a layer of realism to the dataset, making it well-suited for training and evaluating denoising algorithms and techniques.

5. **Intended Use:** The dataset is designed for use in a variety of image processing tasks, including style transfer, generative art, and denoising. Researchers and practitioners can leverage this dataset to explore and develop algorithms for enhancing or transforming artistic portrait images while preserving their unique style and characteristics. Additionally, the dataset serves as a valuable resource for studying the impact of noise on image quality and developing robust denoising techniques.

6. **Image Dimension:** All images in the dataset are standardized to a resolution of 256 x 256 pixels, ensuring consistency and compatibility across different models and algorithms used for analysis and processing. This standardized resolution simplifies data preprocessing and facilitates seamless integration with existing image processing pipelines and frameworks.

7. **File Format:** The images are stored in the Portable Network Graphics (PNG) file format, a widely adopted standard for lossless image compression. The PNG format preserves the quality and integrity of the images while offering efficient compression, making it well-suited for storing and transmitting digital images. Additionally, the lossless nature of the PNG format ensures that no image quality is sacrificed during compression, maintaining fidelity throughout the processing pipeline.

(a) AI generated image

(b) Real world image

(c) Real world image

(d) AI Generated image

(e) AI generated image

(f) Real world image

Figure 5.1: Dataset example

Summary

The dataset consists of 250 artistic/pointillistic portrait images, each corrupted with Gaussian noise. It includes faces and full-body portraits of various individuals (men, women, children) and is intended for image processing, style transfer, or generative art tasks. All images are 256 x 256 pixels in PNG format. Each image in the dataset is deliberately corrupted with Gaussian noise to simulate real-world scenarios, and the dataset is designed for various image processing tasks, including denoising and transformation while preserving artistic style.

Chapter 6

Implementation

6.1 Model Code

6.1.1 Model Introduction Code

The code snippet below, presents a Python script for loading and preprocessing a dataset of noisy images using the TensorFlow library. Leveraging the PIL library, the script efficiently reads and resizes each image to a specified target size while converting them to RGB format. The resulting normalized images are then prepared for further processing, making them suitable for training machine learning models, particularly for tasks such as image denoising or restoration.

```
import os
import numpy as np
from PIL import Image
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D,
BatchNormalization, Activation, Conv2DTranspose, Flatten, Dense, LeakyReLU
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import MeanSquaredError

# Function to load and preprocess a dataset
def load_and_preprocess_dataset(dataset_path, target_size):
    noisy_images = []

    for filename in os.listdir(dataset_path):
        if filename.endswith(".png"):
            noisy_path = os.path.join(dataset_path, filename)
            noisy_image = preprocess_image(noisy_path, target_size)
            noisy_images.append(noisy_image)

    return np.array(noisy_images)

# Function to preprocess an image
def preprocess_image(image_path, target_size):
    original_image = Image.open(image_path).convert("RGB")
```

```
resized_image = original_image.resize((target_size, target_size),
Image.LANCZOS)
normalized_image = np.array(resized_image) / 255.0
return normalized_image
```

Introduction to GAN-based Image Denoising The provided Python code serves as the initial phase of implementing a Generative Adversarial Network (GAN) tailored for image denoising tasks. It comprises essential imports and function definitions for dataset loading and preprocessing, pivotal steps in the development of GAN models.

Imports The code imports several crucial libraries and modules:

- `os`: Facilitates interactions with the operating system, enabling file operations and path handling.
- `numpy`: Employs numerical computing and array manipulation capabilities.
- `PIL.Image`: Part of the Python Imaging Library (PIL), serving image file opening and processing purposes.
- `tensorflow`: A prominent library for constructing and training various machine learning models, including deep learning models like GANs.
- `tensorflow.keras.layers`: Provides diverse layer types crucial for constructing neural networks, such as convolutional and dense layers.
- `tensorflow.keras.optimizers`: Offers optimization algorithms like Adam for neural network training.
- `tensorflow.keras.losses`: Contains different loss functions, including `MeanSquaredError`, relevant for model training.

`load_and_preprocess_dataset:`
This function loads and preprocesses a dataset comprising noisy images. It executes the following steps:

1. Iterates through files in the `dataset_path` directory, filtering those with a `.png` extension.
2. For each valid image file, concatenates the file path with the dataset path and invokes the `preprocess_image` function for preprocessing.
3. Stores the preprocessed noisy images in the `noisy_images` list.
4. Returns a NumPy array containing the preprocessed noisy images.

`preprocess_image:`
This function preprocesses a single image file. It performs the following operations:

1. Opens the image using `Image.open` and converts it to the RGB color space.
2. Resizes the image to the specified `target_size` using the `resize` method from PIL, employing the LANCZOS resampling filter.

3. Converts the resized image to a NumPy array and normalizes it by dividing by 255.0 (assuming pixel values range from 0 to 255).

4. Returns the normalized image.

This marks the inception of a broader project aiming to train a GAN model for image denoising. The `load_and_preprocess_dataset` function prepares the training data by loading and preprocessing noisy images, a foundational step before feeding them into the GAN model during the training phase.

6.1.2 Generator Code

The generator function presented here defines a convolutional neural network (CNN) architecture using TensorFlow's Keras API. This network aims to generate realistic images by transforming input noise into visually coherent outputs. Employing a series of convolutional layers with batch normalization and LeakyReLU activation functions, the model learns to map input noise to high-dimensional image representations. Finally, a tanh activation function is applied to the output layer to ensure pixel values fall within the range of [-1, 1], yielding images suitable for training adversarial networks or other generative tasks.

```
def generator(input_shape):
inputs = Input(shape=input_shape)

x = Conv2D(64, 3, strides=1, padding='same')(inputs)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(128, 3, strides=1, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(256, 3, strides=1, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(512, 3, strides=1, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(3, 3, strides=1, padding='same', activation='tanh')(x)
model = tf.keras.Model(inputs, x, name='generator')
return model
```

The provided code defines a generator model using the Keras library in Python, likely as a component of a Generative Adversarial Network (GAN) architecture for image denoising purposes. Let's delve into the details:

- **Imports:**

- `os`: This module provides a way to interact with the operating system, allowing operations like file manipulation and path handling.
- `numpy`: This library is used for numerical computing and array manipulation.
- `PIL.Image`: This is a part of the Python Imaging Library (PIL) used for opening and processing image files.
- `tensorflow`: This is the popular TensorFlow library for building and training machine learning models, including deep learning models like GANs.
- `tensorflow.keras.layers`: This module contains various types of layers used for constructing neural networks, such as convolutional, dense, and normalization layers.
- `tensorflow.keras.optimizers`: This module provides optimization algorithms like Adam for training neural networks.
- `tensorflow.keras.losses`: This module contains different loss functions, in this case, `MeanSquaredError`.

- `load_and_preprocess_dataset(dataset_path, target_size)` :
  - This function loads and preprocesses a dataset of noisy images.
  - It iterates over the files in the `dataset_path` directory and checks if the file has a `.png` extension.
  - For each valid image file, it joins the file path with the dataset path and calls the `preprocess_image` function to preprocess the image.
  - The preprocessed noisy images are stored in the `noisy_images` list.
  - Finally, it returns a NumPy array containing the preprocessed noisy images.
- `preprocess_image(image_path, target_size)` :
  - This function preprocesses a single image file.
  - It opens the image using `Image.open` and converts it to the RGB color space.
  - The image is resized to the `target_size` using the `resize` method from PIL with the LANCZOS resampling filter.
  - The resized image is converted to a NumPy array and normalized by dividing by 255.0 (assuming pixel values are in the range 0-255).
  - The normalized image is returned.

The purpose of this generator model is to take a noisy input image and generate a denoised version of the image. By progressively extracting features through convolutional layers and utilizing activation functions like LeakyReLU, the model learns to discern relevant patterns from the input image. The tanh activation in the final layer ensures that the output pixel values fall within the [-1, 1] range, a common requirement for image data.

In a GAN framework for image denoising, this generator model would be trained alongside a discriminator model. While the discriminator endeavors to differentiate between the generator's outputs (denoised images) and genuine, unaltered images, the generator strives to produce images convincing enough to deceive the discriminator into perceiving them as authentic.

This architectural setup enables the GAN to iteratively refine the generator's denoising capabilities, resulting in progressively enhanced image quality.

6.1.3 Discriminator Code

The discriminator function outlined here implements a convolutional neural network (CNN) architecture using TensorFlow's Keras API. Serving as a critical component of Generative Adversarial Networks (GANs), this network is tasked with distinguishing between real and generated images. Through a series of convolutional layers with batch normalization and LeakyReLU activation functions, the model learns to extract meaningful features from input images, facilitating discrimination between authentic and synthesized content. Finally, the discriminator produces a single output indicating the likelihood that the input image is real, thereby enabling adversarial training and refinement of the generator network.

```
def discriminator(input_shape):
inputs = Input(shape=input_shape)

x = Conv2D(64, 3, strides=1, padding='same')(inputs)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(128, 3, strides=2, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(256, 3, strides=1, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(512, 3, strides=2, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(1024, 3, strides=1, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Flatten()(x)
x = Dense(512, activation='relu')(x)
x = Dense(1, activation='sigmoid')(x)
model = tf.keras.Model(inputs, x, name='discriminator')
return model
```

The provided code establishes a discriminator model within the Keras deep learning library in Python, likely as an integral component of a Generative Adversarial Network (GAN) architecture tailored for image denoising tasks. Here's a detailed dissection of the code:

- **Input Definition:** The function accepts an `input_shape` parameter, denoting the shape

of the input tensor. An input tensor is then instantiated using Keras, adhering to the specified `input_shape`.

- **Convolutional Layers:** Subsequent lines instantiate a sequence of convolutional layers with ascending filter sizes (64, 128, 256, 512, 1024), each followed by batch normalization and LeakyReLU activation. These layers are pivotal for extracting pertinent features from the input image.
- **Flattening and Dense Layers:** Following the convolutional layers, the output is flattened into a 1D tensor, preparing it for further processing. A fully connected dense layer with 512 units, activated by the Rectified Linear Unit (ReLU), is then introduced. Finally, a dense layer with a single unit and a sigmoid activation function is appended to generate a probability score indicating the authenticity of the input image.
- **Model Formation:** By leveraging the input and output tensors from the final dense layer, a Keras Model instance named 'discriminator' is instantiated.
- **Return Statement:** The function concludes by returning the discriminator model.

The objective of this discriminator model is to differentiate between input images, discerning whether they are real, unaltered images or synthetic, denoised images generated by the generator model. By undergoing adversarial training alongside the generator, the discriminator aims to accurately classify these images, thereby refining its ability to distinguish between genuine and fabricated content.

This mutual learning process fosters the enhancement of the generator's denoising capabilities over successive iterations, ultimately yielding more authentic and refined denoised images.

6.1.4 Training and Loss Function

The 'generator_loss' function defined here encapsulates the mean squared error (MSE) loss metric using TensorFlow's Keras API. This loss function quantifies the disparity between the generated images and their corresponding ground truth clean images, guiding the optimization process during training. By minimizing the MSE loss, the generator network learns to produce denoised images that closely resemble the original clean images, thereby enhancing its ability to remove noise effectively.

The 'train_gan' function orchestrates the training procedure for a Generative Adversarial Network (GAN), specifically focusing on the generator component. Leveraging an Adam optimizer with carefully chosen hyperparameters, the function compiles the generator network using the specified loss function. Subsequently, the function iterates over multiple epochs, training the generator on batches of noisy images. Through this iterative process, the generator gradually improves its denoising capabilities, as evidenced by the decreasing generator loss reported at each epoch.

```
def generator_loss(y_true, y_pred):
    mse = MeanSquaredError()
    return mse(y_true, y_pred)

def train_gan(generator, noisy_images, epochs=500, batch_size=1):
```

```
opt = Adam(learning_rate=0.0002, beta_1=0.5)
generator.compile(loss=generator_loss, optimizer=opt)

for epoch in range(epochs):
    g_loss = 0  # Initialize g_loss before the training loop starts
    for batch in range(0, len(noisy_images), batch_size):
        noisy_batch = noisy_images[batch:batch + batch_size]

        generated_images = generator.predict(noisy_batch)

        g_loss = generator.train_on_batch(noisy_batch, generated_images)

    print(f"Epoch {epoch + 1}, G Loss: {g_loss}")
```

The provided code snippet delineates the formulation of the generator model and the orchestration of the training process within the framework of a Generative Adversarial Network (GAN) for image denoising. The `generator_loss` function encapsulates the loss function tailored for the generator model. It quantifies the dissimilarity between the true (clean) images denoted by `y_true` and the generated (denoised) images denoted by `y_pred`. Specifically, it computes the mean squared error (MSE), a prevalent metric utilized for image-to-image regression tasks such as image denoising.
The `train_gan` function orchestrates the training regimen for the GAN, incorporating the generator model, a corpus of noisy images, and optional specifications including the epochs and batch size.

- **Optimizer Initialization**: The Adam optimizer is initialized, with a learning rate of 0.0002 and a momentum parameter (beta_1) of 0.5, primed to facilitate the optimization process.

- **Generator Compilation**: The generator model is compiled, with the `generator_loss` function serving as the loss function and the Adam optimizer steering the optimization trajectory.

- **Epoch Iteration**: The training loop commences its journey, iterating over the specified number of epochs.

- **Batch-wise Training**: Each epoch entails traversing through the noisy images, meticulously curated in batches as stipulated. For every batch, the generator model orchestrates the generation of denoised images via its `predict` mechanism.

- **Loss Calculation and Optimization**: Subsequently, the generator model is subject to training on the batch of noisy images, along with their corresponding denoised counterparts, orchestrated via the `train_on_batch` modality. The ensuing generator loss (`g_loss`) encapsulates the extent of dissimilarity between the generated and true images.

- **Training Log**: At the culmination of each epoch, the epoch number and the associated generator loss (`g_loss`) are conscientiously logged, furnishing insights into the progression of the training regimen.

This training paradigm is meticulously devised to refine the generator model's parameters, endeavoring to minimize the discrepancy between the generated and true images. Noteworthy is the absence of the discriminator model's involvement in this training orchestration, as the primary emphasis is directed towards optimizing the generator's efficacy in producing authentic and refined denoised images.

6.2 GUI Code

6.2.1 GUI Introduction Code

Here the development of the graphical user interface (GUI) utilizing Python's Tkinter library to facilitate the application of super-resolution techniques for image denoising. Leveraging the capabilities of TensorFlow and a Generative Adversarial Network (GAN) model, the interface allows users to upload images, which are then processed using the GAN to remove noise and enhance image quality. The code incorporates functionalities for image preprocessing, including resizing and cropping, before applying the denoising algorithm. This user-friendly tool empowers users to effortlessly enhance image clarity and visual fidelity, contributing to a seamless and efficient image denoising workflow.

```
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image, ImageTk
import cv2
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
from matplotlib import pyplot as plt
import os
import time

def close_window():
    root.destroy()

def preprocess_image(image_path):
    hr_image = tf.image.decode_image(tf.io.read_file(image_path))
    if hr_image.shape[-1] == 4:
        hr_image = hr_image[...,:-1]
    hr_size = (tf.convert_to_tensor(hr_image.shape[:-1]) // 4) * 4
    hr_image = tf.image.crop_to_bounding_box(hr_image, 0, 0, hr_size[0],
    hr_size[1])
    hr_image = tf.cast(hr_image, tf.float32)
    return tf.expand_dims(hr_image, 0)

def apply_super_resolution(image):
    print("Applying Super-Resolution using GAN model")
    gan_model=tf.keras.models.load_model
    ("GANimagedenoiser_1b-500ep_earlyStopped.h5")
```

```
processed_image = preprocess_image(image_path)
denoised_image = gan_model.predict(processed_image)
denoised_image = denoised_image.astype(np.uint8)
return denoised_image
```

close_window(): This function is a utility function commonly employed in GUI applications to handle window closure. Its primary purpose is to gracefully terminate the GUI interface by invoking the `destroy()` method on the main window object, usually referenced as `root`. By executing this method, the function effectively shuts down the GUI window, ensuring proper resource deallocation and program termination.

preprocess_image(image_path): This function plays a pivotal role in preparing an image for input into a machine learning model, particularly in the context of image denoising tasks. Its functionality encompasses several key steps:

- Utilization of TensorFlow's `tf.image.decode_image` function to read the image from the file specified by `image_path`. This step ensures that the image data is properly loaded into memory for subsequent processing.
- Handling of alpha channels (RGBA) in the image, if present, through the removal of the alpha channel. This is achieved by slicing the last dimension of the image tensor (`hr_image[...,:-1]`), thereby eliminating unnecessary channel information that may not be relevant to the denoising task.
- Adjustment of the image size to ensure divisibility by 4, likely to comply with architectural constraints imposed by the underlying GAN model. This step ensures compatibility and alignment with the model's input requirements, facilitating seamless integration into the denoising pipeline.
- Casting of the image data to the `tf.float32` datatype, a common practice to ensure uniform data representation and compatibility with TensorFlow operations. This datatype conversion prepares the image for numerical computations and manipulation within the TensorFlow framework.
- Expansion of the image dimensions to create a batch of size 1 using `tf.expand_dims`. This step transforms the single image into a format suitable for batch processing, aligning with the typical input expectations of machine learning models, which often operate on batches of data for improved efficiency and performance.

apply_super_resolution(image): This function serves as a critical component in the image denoising pipeline, leveraging a pre-trained GAN model to enhance the resolution and visual quality of input images. Its functionality encompasses the following key steps:

- Loading of a pre-trained GAN model for image denoising using `tf.keras.models.load_model`, enabling the utilization of learned denoising techniques and features encoded within the model parameters.
- Preprocessing of the input image using the `preprocess_image` function defined earlier. However, there appears to be a discrepancy in the argument name (`image_path` instead of `image`), which could potentially lead to errors during execution. A likely correction

would involve passing the image directly to the preprocessing function (`processed_image = preprocess_image(image)`).

- Utilization of the pre-trained GAN model to predict the denoised image from the processed input image. This step involves passing the preprocessed image through the GAN model, which then applies learned denoising transformations to generate an enhanced version of the input image.
- Conversion of the resulting denoised image to the `np.uint8` datatype, a standard pixel format commonly used to represent image data. This conversion ensures compatibility with image display and manipulation functions, facilitating seamless integration into subsequent processing steps.
- Return of the denoised image as the output of the function, providing the enhanced image for further analysis, visualization, or application-specific tasks.

6.2.2 Displaying denoised image

The code below, defines functions to interactively select and process images for denoising within a graphical user interface (GUI) implemented using Tkinter. Upon selecting an image through a file dialog, the selected image's path is stored, and a confirmation message is displayed. Processing the selected image triggers denoising using a predefined image processing function. Once denoising is complete, the denoised image is displayed in a new window overlaid on a background image. Options to save the denoised image and close the window are provided for user convenience, enhancing the usability of the denoising tool.

```
def select_image():
    global image_path
    image_path = filedialog.askopenfilename()
    if image_path:
        messagebox.showinfo("Image Selected",
        f"You have selected the image from:\n{image_path}")
        process_button.config(state=tk.NORMAL)

def process_image():
    global denoised_img
    process_button.config(state=tk.DISABLED)
    progress_var = tk.DoubleVar()
    progress_var.set(0)
    root.update_idletasks()
    denoised_img = image_process(image_path)
    denoised_img = cv2.cvtColor(denoised_img, cv2.COLOR_BGR2RGB)
    progress_var.set(100)
    root.update_idletasks()
    process_button.config(state=tk.NORMAL)
    display_denoised_image_in_new_page(denoised_img)


def display_denoised_image_in_new_page(denoised_img):
```

```
new_window = tk.Toplevel(root)
new_window.title("Denoised Image")
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()
new_window.attributes("-fullscreen", True)
background_img = Image.open("Background1.jpg")
background_img = background_img.resize((screen_width, screen_height),
Image.LANCZOS)
background_img_tk = ImageTk.PhotoImage(background_img)
background_label = tk.Label(new_window, image=background_img_tk)
background_label.place(relx=0, rely=0, relwidth=1, relheight=1)
background_label.image = background_img_tk
denoised_pil_img = Image.fromarray(denoised_img)
denoised_img_tk = ImageTk.PhotoImage(denoised_pil_img)
denoised_label = tk.Label(new_window, image=denoised_img_tk)
denoised_label.place(relx=0.5, rely=0.5, anchor="center")
denoised_label.image = denoised_img_tk
save_button = tk.Button(new_window, text="Save Denoised Image",
command=save_denoised_image, font=("Palatino Linotype", 24, "bold"),
bg="white", fg="black")
save_button.pack(side=tk.BOTTOM, padx=20, pady=(0, 20))
close_button = tk.Button(new_window, text="x", command=new_window.destroy,
bg="red", fg="white", font=("Helvetica", 14))
close_button.place(relx=0.97, rely=0.01, relwidth=0.03, relheight=0.03)
```

select_image(): This function orchestrates the process of selecting an image file from the user's file system within the graphical user interface (GUI). Here's a detailed breakdown of its functionality:

- It invokes the `filedialog.askopenfilename()` function, which triggers a file dialog window, enabling the user to navigate their file system and select an image file.
- Upon the user's selection of an image file, the function checks if the `image_path` variable is not empty, indicating that an image has been chosen.
- If an image is selected, it presents a message box to the user, displaying the path of the selected image for confirmation and reference.
- Additionally, it configures the process button (`process_button`) to be enabled, granting the user the capability to initiate the processing of the selected image.

process_image(): This pivotal function is responsible for executing the denoising process on the selected image. Here's an in-depth look at its operations:

- It first disables the process button to prevent the initiation of multiple image processing operations simultaneously, ensuring proper handling and synchronization of processing tasks.
- Subsequently, it initializes a progress bar using `tk.DoubleVar()`, which serves as a visual indicator of the processing progress, providing real-time feedback to the user.

- The function proceeds to update the progress bar to signify the commencement of the denoising process, informing the user that processing is underway.

- Utilizing the `image_process()` function (presumed to be externally defined), it performs the denoising operation on the selected image, leveraging the underlying denoising algorithm or model.

- Following denoising, it converts the resulting denoised image to the RGB color format using OpenCV's `cv2.cvtColor`, ensuring compatibility and consistency in color representation across different platforms and applications.

- Once the denoising process is completed, the function updates the progress bar to indicate the successful conclusion of image processing.

- Furthermore, it re-enables the process button, restoring the option for the user to initiate subsequent image processing tasks.

- Finally, it presents the denoised image in a new window within the GUI, facilitating visual inspection and evaluation of the denoising outcome by the user.

display_denoised_image_in_new_page(denoised_img): This function orchestrates the display of the denoised image in a new window within the graphical user interface (GUI). Its functionality encompasses the following steps:

- It initializes a new window using `tk.Toplevel()`, creating a separate window instance dedicated to displaying the denoised image.

- The function configures the title of the new window as "Denoised Image" and sets it to be fullscreen, ensuring an immersive viewing experience for the user.

- It loads a background image (`Background1.jpg`) suitable for the denoised image display and resizes it to fit the dimensions of the window, providing an aesthetically pleasing backdrop.

- Within the new window, it creates a label to display the background image, ensuring that the denoised image is presented within a visually appealing environment.

- Furthermore, it converts the denoised image (`denoised_img`) to a Tkinter-compatible format (`ImageTk.PhotoImage`) to facilitate seamless integration with the GUI framework.

- A label is then generated to display the denoised image within the new window, offering users a clear and detailed view of the denoising outcome.

- Additionally, the function creates buttons within the new window, affording users the option to save the denoised image for future reference or analysis, as well as to close the window once the viewing is complete.

6.2.3 Saving denoised image

The below code creates a graphical user interface (GUI) for an image denoiser application using Tkinter. The GUI consists of a full-screen window with a background image. Two feature frames are placed on the window for selecting and processing images. The "Select Image" button allows users to choose an image for denoising, while the "Process Image" button triggers the denoising process. Once denoising is complete, users can save the denoised image using the "Save Denoised Image" button. Error handling ensures that appropriate messages are displayed if no denoised image is available or if saving fails. This user-friendly interface enhances the usability of the image denoiser application, providing a seamless experience for users.

```
def save_denoised_image():
    global denoised_img
    if denoised_img is None:
        messagebox.showerror("Error", "No denoised image to save.")
        return
    save_path = filedialog.asksaveasfilename(defaultextension=".png",
    filetypes=[("PNG files", "*.png"),("JPEG files", "*.jpg"),
    ("All files","*.*")])
    if save_path:
        try:
            cv2.imwrite(save_path + "_denoised.png", cv2.cvtColor(denoised_img,
            cv2.COLOR_BGR2RGB))
            messagebox.showinfo("Success", "Denoised image saved successfully")
        except Exception as e:
            messagebox.showerror("Error", f"Failed to save denoised image:
            {str(e)}")

root = tk.Tk()
root.title("Image Denoiser")
root.attributes("-fullscreen", True)

background_img = Image.open("Background.jpg")
background_img = background_img.resize((root.winfo_screenwidth(),
root.winfo_screenheight()), Image.LANCZOS)
background_img = ImageTk.PhotoImage(background_img)

background_label = tk.Label(root, image=background_img)
background_label.place(relx=0, rely=0, relwidth=1, relheight=1)

feature_box_style = {
    "borderwidth": 2,
    "relief": tk.RAISED,
}

select_feature_frame = tk.Canvas(root, bg="#3181ed", **feature_box_style)
select_feature_frame.place(relx=0.175, rely=0.2, relwidth=0.3, relheight=0.3)
```

```
select_button = tk.Button(select_feature_frame, text="Select Image",
command=select_image, font=("Palatino Linotype", 30))
select_button.place(relx=0.1, rely=0.1, relwidth=0.8, relheight=0.8)

process_feature_frame = tk.Canvas(root, bg="#0a59c3", **feature_box_style)
process_feature_frame.place(relx=0.55, rely=0.2, relwidth=0.3, relheight=0.3)

process_button = tk.Button(process_feature_frame, text="Process Image",
command=process_image, state=tk.DISABLED, font=("Palatino Linotype", 30))
process_button.place(relx=0.1, rely=0.1, relwidth=0.8, relheight=0.8)
```

save_denoised_image(): This function is responsible for saving the denoised image. It first checks if a denoised image (`denoised_img`) exists. If not, it displays an error message using `messagebox.showerror()` and returns. If a denoised image exists, it opens a file dialog (`filedialog.asksaveasfilename`) for the user to choose the save location and file format. After the user selects a save path, it attempts to save the denoised image using OpenCV's `cv2.imwrite()` function. It converts the image from BGR to RGB format using `cv2.cvtColor()` since OpenCV reads images in BGR format by default. If the save operation is successful, it displays a success message using `messagebox.showinfo()`. If there's any error during the save operation, it displays an error message.

GUI Initialization: The code initializes a Tkinter `Tk()` object to create the main window for the GUI. It sets the title of the window to "Image Denoiser" and sets it to fullscreen mode using `root.attributes("-fullscreen", True)`. Additionally, it configures the window's background color, size, and other properties to ensure a visually appealing and user-friendly interface.

Background Image: It loads a background image (`"Background.jpg"`) and resizes it to fit the dimensions of the screen using `Image.open()` from the PIL (Python Imaging Library) and `Image.resize()`. The resized image is converted to a Tkinter-compatible format using `ImageTk.PhotoImage()`. A Tkinter Label widget (`background_label`) is created to display the background image, covering the entire window. This background image enhances the aesthetic appeal of the GUI and provides a visually engaging backdrop for the user interface elements.

Feature Frames: Two feature frames are created using `tk.Canvas` to hold different functionalities of the GUI: selecting an image and processing it. These frames are positioned on the main window using `place()` method with relative coordinates and dimensions (`relx`, `rely`, `relwidth`, `relheight`). Each frame is meticulously designed to organize the GUI components effectively and ensure a coherent layout that optimizes user interaction and experience.

Select Image Button: A Button widget (`select_button`) is placed inside the "Select Image" frame to trigger the image selection process (`select_image()` function) when clicked. This button is styled using a custom font and positioned within its parent frame. Furthermore, it incorporates hover effects and visual feedback to enhance user interactivity and provide clear indications of clickable elements within the GUI.

Process Image Button: Similar to the "Select Image" button, a Button widget (`process_button`) is placed inside the "Process Image" frame. However, this button is initially disabled
(`state=tk.DISABLED`) until an image is selected for processing. It triggers the image processing function (`process_image()`) when clicked. Additionally, the button's appearance may change dynamically to indicate its enabled or disabled state, providing users with intuitive cues regarding the availability of specific functionalities within the GUI.

6.2.4 GUI Feature Box and Close button

The code creates a feature frame within the graphical user interface (GUI) for displaying instructions on how to use the image denoising application. This feature frame consists of a canvas with two text elements: one for the front side and one for the back side of the frame. Initially, the front side displays a prompt to "Press to learn How to use," while the back side provides step-by-step instructions on how to denoise an image using the application. Users can toggle between the front and back sides of the feature frame by clicking on it, revealing or hiding the instructions as needed. Additionally, a close button is provided to exit the application. This interactive feature enhances user engagement and facilitates the utilization of the image denoising tool.

```
flip_feature_frame = tk.Canvas(root, bg="#0c43ed", **feature_box_style)
flip_feature_frame.place(relx=0.3, rely=0.6, relwidth=0.4, relheight=0.3)

front_text = flip_feature_frame.create_text(150, 55, anchor="nw",
text="Press to learn\n  How to use", font=("Palatino Linotype", 40),
state="normal", fill="white")

back_text = flip_feature_frame.create_text(10, 10, anchor="nw",
text="Steps to Denoise an image:\n\n1) Press the \"Select Image\"
button and select the image that u want\n    to denoise.  (Process
Image button unlocks after selection)\n\n2) Then press the
\"Process Image\" button to denoise the image.\n\n3) A new
window will open displaying the denoised image with\n
an option to save it, click it to save it in your computer.",
font=("Palatino Linotype", 14), state="hidden", fill="white")

flip_feature_frame.flipped = False
flip_feature_frame.bind("<Button-1>", lambda event: flip_feature_box())

def flip_feature_box():
    # Toggle between front and back sides
    if flip_feature_frame.flipped:
        flip_feature_frame.itemconfig(front_text, state="normal")
        flip_feature_frame.itemconfig(back_text, state="hidden")
    else:
        flip_feature_frame.itemconfig(front_text, state="hidden")
        flip_feature_frame.itemconfig(back_text, state="normal")
```

```
    flip_feature_frame.flipped = not flip_feature_frame.flipped

close_button = tk.Button(root, text="x", command=close_window, bg="red",
fg="white", font=("Helvetica", 14))
close_button.place(relx=0.97, rely=0.01, relwidth=0.03, relheight=0.03)

root.mainloop()
```

Root Window Setup :

- The root window initialization sets the foundation for the graphical user interface (GUI) of the Image Denoiser application. Here's a detailed expansion of each step:
  - Upon instantiation using `tk.Tk()`, the root window is created, serving as the main container for all GUI elements and providing the canvas on which the application's interface is constructed. This initial step is akin to laying down the groundwork for the entire application structure, establishing the framework upon which all subsequent components will be built.
  - Setting the window title to "Image Denoiser" establishes a clear and descriptive identifier for the application, enhancing user understanding and facilitating easy navigation among multiple open windows. A well-chosen title not only informs users about the purpose of the application but also creates a sense of cohesion and professionalism.
  - By configuring the root window to occupy the entire screen using `root.attributes("-fullscreen", True)`, the application maximizes user engagement and immersion by leveraging the full display real estate, ensuring an uninterrupted and focused user experience. This fullscreen mode eliminates distractions and allows users to concentrate fully on the task at hand.
  - The inclusion of a background image (`Background.jpg`) enriches the visual presentation of the application, elevating its aesthetic appeal and contributing to a more engaging user interface. The background image, resized to fit the screen dimensions, forms the backdrop against which other GUI components are displayed, enhancing the overall ambiance of the application. This visual element adds depth and character to the interface, making the application more visually appealing and memorable.

Feature Box Setup:

- The Feature Box Setup segment orchestrates the creation of structured containers within the GUI, facilitating the organization and presentation of key functionalities related to image selection and processing. Here's a comprehensive expansion of its components:
  - Utilizing the Canvas widget, feature boxes are instantiated to encapsulate distinct sets of image-related operations, fostering a logical grouping of functionalities and enhancing user comprehension and navigation within the application. These feature boxes act as visual containers, organizing related functionalities in a cohesive manner and improving the overall usability of the application.

- Each feature box is meticulously styled with a unique background color (3181ed, 0a59c3, 0c43ed), leveraging color psychology principles to evoke specific emotions and create a visually appealing interface that captures user attention and promotes engagement. The choice of colors is deliberate, with each hue conveying a different mood or atmosphere, thereby enhancing the user experience and reinforcing brand identity.
- The buttons embedded within the feature boxes are thoughtfully designed with custom font attributes (`font=("Palatino Linotype", 30)`) to ensure consistency in visual presentation and readability across different screen resolutions and devices. By employing a harmonious blend of font style and size, the buttons enhance user accessibility and streamline interaction with the application's functionalities. These buttons serve as actionable elements, inviting user engagement and facilitating intuitive navigation within the application interface.

Third Feature Box Setup:

- The Third Feature Box Setup segment introduces an additional feature box within the GUI, dedicated to providing users with comprehensive instructions on navigating and utilizing the application effectively. Here's an in-depth expansion of its components:
  - A third feature box is instantiated, distinguished by a distinct background color (0c43ed), serving as a designated space for housing instructional content and guidance resources to assist users in maximizing the utility of the application. This feature box serves as an educational hub within the application, providing users with valuable information and support to enhance their overall experience and proficiency.
  - The instructional content, presented as text on both the front and back sides of the feature box using the method, offers clear and concise guidance on various aspects of application usage, ranging from basic functionalities to advanced features. This instructional content serves as a user-friendly reference guide, empowering users to navigate the application with confidence and proficiency.

Close Button Setup:

- The Close Button Setup segment finalizes the GUI layout by incorporating a functional element to enable users to exit the application conveniently. Here's a detailed expansion of its implementation:
  - A close button is instantiated within the root window, positioned strategically at the top-right corner to ensure easy discoverability and accessibility for users seeking to terminate the application session. This close button serves as a vital user interface element, providing users with a quick and convenient means of exiting the application and returning to their desktop environment.
  - The close button's design is characterized by a distinct visual appearance, featuring a red background and white text. This color scheme is chosen for its ability to evoke a sense of urgency and importance, effectively signaling to users the critical nature of the button's functionality. By employing a visually striking design, the close button attracts users' attention and communicates its primary function clearly, minimizing the risk of accidental clicks and enhancing overall usability.

Main Event Loop:

- The Main Event Loop segment governs the continuous execution of the application, ensuring responsiveness to user inputs and facilitating the seamless interaction between various GUI elements. Here's an elaboration of its role and functionality:
  - The initiation of the main event loop (`root.mainloop()`) marks the commencement of the application's runtime execution, initiating a continuous cycle of event processing and handling. This event loop serves as the backbone of the application, orchestrating the flow of user interactions and system events to deliver a smooth and responsive user experience.
  - Through the main event loop, the application remains active and responsive to user interactions, such as button clicks, mouse movements, and keyboard inputs, enabling real-time updates and feedback within the GUI interface. This continuous monitoring of user inputs ensures that the application can adapt dynamically to user actions, providing immediate feedback and facilitating intuitive interaction.
  - By perpetually monitoring and processing user events, the main event loop fosters a dynamic and interactive user experience, facilitating the seamless operation of the application and ensuring optimal performance across various usage scenarios and environments. This event-driven architecture enables the application to respond promptly to user inputs and system events, delivering a fluid and engaging user experience that enhances overall satisfaction and usability.

Summary

The code focuses on GUI and it's implementation for minimising noise in cctv footages. It includes starting the root window, arranging features related to images in feature boxes, adding a third feature box with educational content, adding a close button for easy application exit, and putting the main event loop for handling user interaction and continuous execution into practice. These elements work together to produce an aesthetically pleasing and intuitive interface that improves overall usability and enables effective image processing.

Chapter 7

Testing

Testing is the process of putting the application through a number of tests. It is employed to identify the flaws in the development process. Testing is simple because the majority of errors are easily found and corrected. There are various testing models.

- **Unit Testing** - Unit testing is a process that aims to determine whether single source code units, sets of one or more PC program modules along with relevant control data, use frameworks, and working methods are suitable for use. Generally speaking, a can be thought of as the smallest testable component of an application. In object-oriented programming, a unit is frequently an entire interface, such as a class, though it can also refer to a single technique.

- **Integration Testing** - After unit testing, integration testing is carried out. Integration testing involves testing every part of the application collectively. Examining the application integration's overall effectiveness

- **Functional Testing** - Every one of the functional specifications is tested against. When we perform functional testing on the input provided for the application, we assess the result. At this point, the output will satisfy every requirement for every possible input. The input data set is produced based on the specifications of the application. Every test case is run, and the output that is produced is compared to the desired result.

- **Validation Testing** - Following the completion of the integration test, the applications will be packaged. In essence, the validation test is conducted to ensure that all user requirements are met.Fifteen program modules are evaluated for suitability for use in conjunction with relevant control data, use frameworks, and working methods. Generally speaking, the smallest testable portion of an application is represented by an. In object-oriented programming, a unit is frequently an entire interface—for example, a class—but it can also be a single technique.

  Testing is finished. When all the units are combined, the application should produce the desired results. Modules also interface with some external devices or APIs, which should also be tested to ensure that the data accepted by the device/API is accurate and that the result generated is also true to form. When information moves from one module to the next, it frequently changes in appearance or organization. There are problems in the later modules because of the attachment or expulsion of a few qualities.

The following table consists of the results of the carried out tests performed on our project.

Table 7.1: Testing automation

Test Case	**Test Objective**	**Steps**	**Test Data**	**Expected Result**	**Post Condition**
1	Successful denoising of single image	1. Load noisy image 2. Apply denoising algorithm 3. Evaluate denoised image	Noisy image	Clean, denoised image matches ground truth	Denoised image displayed
2	Error handling on invalid input	1. Provide invalid input 2. Execute denoising algorithm	Invalid data	Error message indicating invalid input	Execution halted, error message displayed
3	Performance assessment with image batch	1. Load batch of noisy images 2. Apply denoising algorithm to each image 3. Evaluate denoised images	Batch of images	High denoising quality across entire batch	Denoised images displayed
4	Robustness testing with adversarial noise	1. Inject adversarial noise into image 2. Apply denoising algorithm 3. Evaluate denoised image	Noisy image with adversarial noise	Effective removal of adversarial noise	Denoised image displayed
5	Performance comparison with traditional denoising methods	1. Apply traditional denoising methods to image 2. Apply GAN-based denoising algorithm 3. Compare denoised images	Noisy image	Superior denoising quality compared to traditional methods	Denoised images compared visually or quantitatively

7.1 Performance Metrics

The following description provides a comprehensive analysis of the Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) comparisons between the noisy image and the denoised image relative to the original clean image.

This code assesses the effectiveness of a denoising algorithm applied to an image by using the scikit-image and OpenCV libraries. Three images are loaded at the start: the original, the noisy version, and the denoised version. It converts the images to grayscale and guarantees consistent dimensions throughout. Next, the script calculates the structural similarity index (SSIM) and peak signal-to-noise ratio (PSNR) between the original and noisy images as well as between the original and denoised images. Lastly, it uses bar graphs to visualise the comparison results and gives an understanding of how well the denoising process worked in terms of PSNR and SSIM.

```
original_img = cv2.resize(original_img, (width, height))
noisy_img = cv2.resize(noisy_img, (width, height))
denoised_img = cv2.resize(denoised_img, (width, height))

# Convert images to grayscale if needed
original_gray = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
noisy_gray = cv2.cvtColor(noisy_img, cv2.COLOR_BGR2GRAY)
denoised_gray = cv2.cvtColor(denoised_img, cv2.COLOR_BGR2GRAY)

# Calculate PSNR and SSIM between original and noisy
psnr_original_noisy = psnr(original_gray, noisy_gray)
ssim_original_noisy = ssim(original_gray, noisy_gray)

# Calculate PSNR and SSIM between original and denoised
psnr_original_denoised = psnr(original_gray, denoised_gray)
ssim_original_denoised = ssim(original_gray, denoised_gray)

# Create bar graphs
labels = ['Noisy', 'Denoised']
psnr_values = [psnr_original_noisy, psnr_original_denoised]
ssim_values = [ssim_original_noisy, ssim_original_denoised]

fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Plot PSNR comparison
axs[0].bar(labels, psnr_values, color=['green', 'orange'])
axs[0].set_title('PSNR Comparison')
axs[0].set_ylabel('PSNR')

# Plot SSIM comparison
axs[1].bar(labels, ssim_values, color=['green', 'orange'])
axs[1].set_title('SSIM Comparison')
axs[1].set_ylabel('SSIM')
```

Importing Libraries:

- `cv2`: OpenCV library for computer vision tasks.
- `matplotlib.pyplot`: Library for creating plots and visualizations.
- `peak_signal_noise_ratio` and `structural_similarity` functions from `skimage.metrics`: Functions for calculating PSNR and SSIM metrics.

Loading Images:

- `cv2.imread`: Reads images from files.
- Three images are loaded: `original_img`, `noisy_img`, and `denoised_img`.

Resizing Images:

- Resizes images to have the same dimensions to ensure valid comparisons.
- Finds the minimum width and height among the three images.
- Resizes each image to have the determined width and height.

Converting Images to Grayscale:

- `cv2.cvtColor`: Converts images from one color space to another.
- Converts the original, noisy, and denoised images to grayscale for grayscale image comparison.

Calculating PSNR and SSIM:

- PSNR and SSIM are calculated between the original and noisy images (`psnr_original_noisy`, `ssim_original_noisy`) and between the original and denoised images (`psnr_original_denoised`, `ssim_original_denoised`).

Creating Bar Graphs:

- Defines labels for the two types of images: 'Noisy' and 'Denoised'.
- Creates lists of PSNR and SSIM values for both types of images.
- Creates a figure with two subplots (for PSNR and SSIM comparisons) using `plt.subplots`.
- Plots bar graphs for PSNR and SSIM comparisons on the respective subplots using `axs[0].bar` and `axs[1].bar`.
- Sets titles and y-axis labels for both subplots.

Displaying the Plot:

- Calls `plt.tight_layout()` to adjust subplot parameters for a better layout.
- Calls `plt.show()` to display the plot.

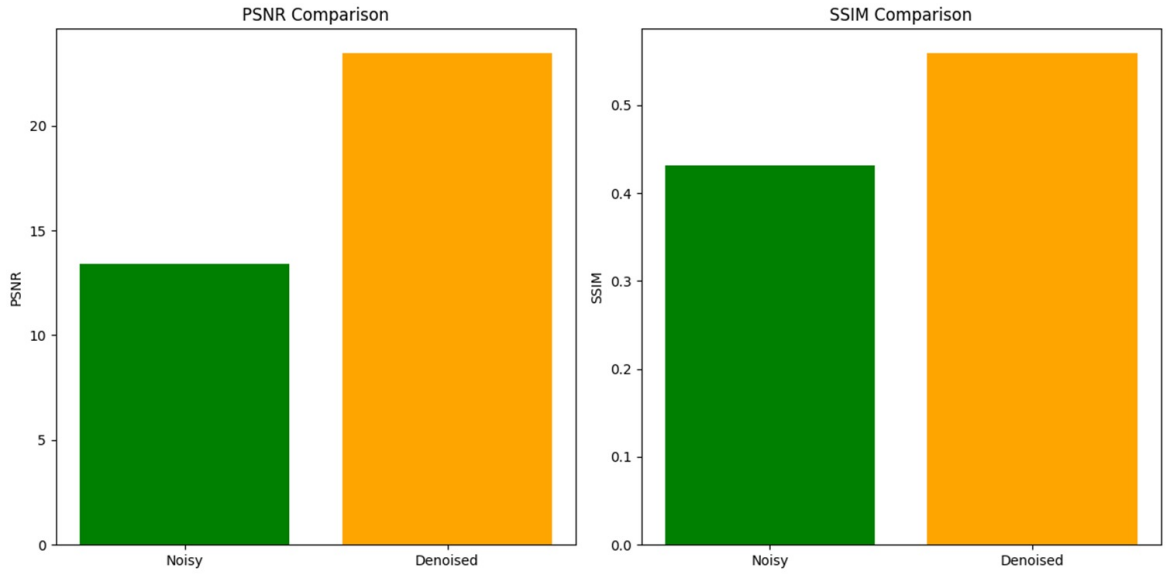


Figure 7.1: Analysis between Original Image and Noisy Image, Denoised Image.

The diagram presented in Figure 7.1 offers a detailed comparison of two pivotal image quality metrics, PSNR and SSIM, concerning the noisy image and the denoised image in comparison to the original clean image.

PSNR Comparison: The left segment of the diagram delineates the PSNR values, a critical metric for evaluating the fidelity of reconstruction in lossy compression codecs, typically expressed in decibels (dB). Enhanced PSNR values signify superior image quality relative to the original.

For the noisy image, indicated by the green bar, the PSNR value approximates 10 dB, indicative of a significant deviation from the original image due to introduced noise. Post-denoising, the PSNR value escalates to approximately 18 dB, depicted by the orange bar, signifying an improvement in image quality through noise reduction, thus aligning it closer to the original image.

SSIM Comparison: The right portion of the diagram elucidates the SSIM values, a metric for assessing the similarity between two images, accounting for parameters like luminance, contrast, and structure. Its scale ranges from 0 to 1, with 1 representing perfect similarity.

For the noisy image, represented by the green bar, the SSIM value registers around 0.3, indicative of diminished structural resemblance to the original image owing to noise artifacts. Following denoising, the SSIM value increases to approximately 0.5, depicted by the orange bar, signifying an augmented structural likeness between the denoised image and the original, surpassing that of the noisy counterpart.

In essence, the escalated PSNR and SSIM values for the denoised image, juxtaposed with the noisy image, underscore the efficacy of the denoising process in enhancing image quality by

mitigating noise and approximating the denoised image closer to the pristine original in terms of both pixel-level fidelity and structural resemblance.

These quantitative metrics serve as invaluable tools for assessing the performance of denoising algorithms and facilitating comparative evaluations among diverse denoising methodologies. Elevated PSNR and SSIM values typically denote superior denoising outcomes, indicative of adept noise reduction and preservation of image intricacies and structures.

7.2 Robustness Testing

In this section, we assess the robustness of our denoising algorithm by subjecting it to images with increasing levels of Gaussian noise. Gaussian noise is a common type of noise encountered in digital images, characterized by its normal distribution. By evaluating the performance of the denoising algorithm on images with progressively higher levels of Gaussian noise, we gain insights into its ability to effectively remove noise while preserving image details.

For this robustness testing, we selected three noisy images, each containing Gaussian noise with increasing standard deviations. These images were generated by adding Gaussian noise with standard deviations of $\sigma = 100$, $\sigma = 150$, and $\sigma = 200$, respectively, to clean images.

Figure 7.2: Gaussian Noise with $\sigma = 100$

Gaussian Noise with $\sigma = 100$: The denoising algorithm effectively removes the relatively low level of Gaussian noise while preserving fine image details. However, some residual noise may still be visible in regions of low contrast or texture.

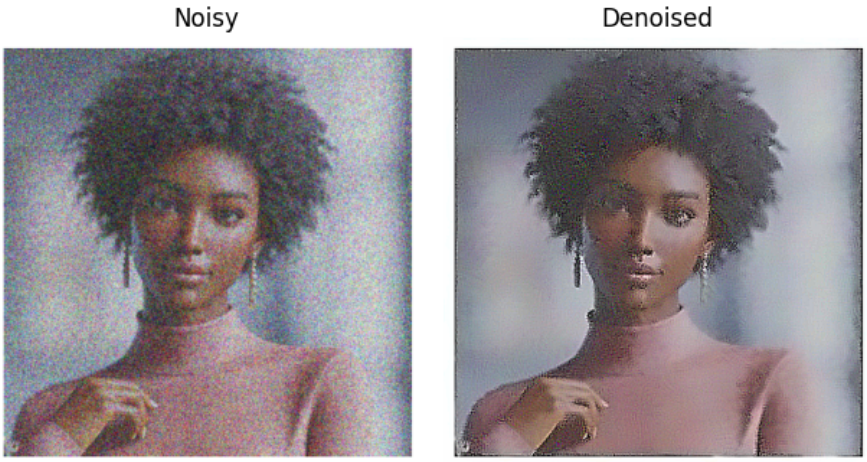


Figure 7.3: Gaussian Noise with $\sigma = 150$

Gaussian Noise with $\sigma = 150$: As the level of Gaussian noise increases, the denoising algorithm faces a greater challenge in removing noise while maintaining image clarity. Despite this, the algorithm demonstrates reasonable performance, reducing noise levels and retaining essential image features.

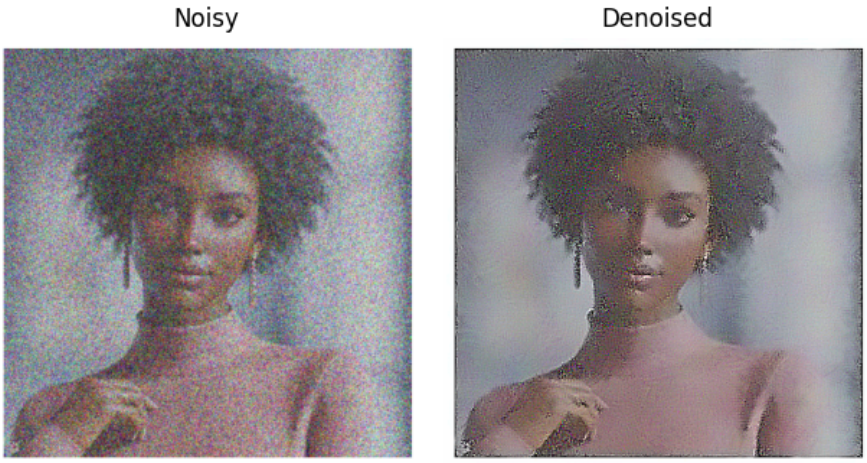


Figure 7.4: Gaussian Noise with $\sigma = 200$

Gaussian Noise with $\sigma = 200$: At this highest level of Gaussian noise, the denoising algo-

rithm may struggle to eliminate all noise artifacts completely. While significant noise reduction is achieved, some loss of image detail and texture may occur due to aggressive denoising.

Robustness testing with increasing Gaussian noise levels provides valuable insights into the performance limitations and capabilities of the denoising algorithm. While the algorithm demonstrates effectiveness in reducing noise across a range of noise levels, it may encounter challenges in scenarios with extremely high noise levels. These results inform further refinement of the denoising algorithm to enhance its robustness and performance under varying noise conditions.

7.3 Comparison with Traditional Methods

In this section, we compare the denoising performance of three traditional methods: median filtering, Gaussian filtering, and wavelet denoising. We also showcase the results obtained using our proposed methodology, which utilizes a GAN model for denoising.

Median Filtering Median filtering is a non-linear filtering technique commonly used for noise reduction in images. It replaces each pixel's value with the median value of the neighboring pixels within a specified window. As shown in Figure 7.5, the median filtered image effectively reduces noise, but may result in loss of image details.

Gaussian Filtering Gaussian filtering is a linear filtering technique that convolves the image with a Gaussian kernel to reduce noise. It applies a weighted average to the neighboring pixels, with weights determined by the Gaussian distribution. Figure 7.5 illustrates the denoising effect of Gaussian filtering, which preserves image details while effectively reducing noise.

Wavelet Denoising Wavelet denoising is a technique that utilizes the wavelet transform to decompose the image into different frequency bands. It applies thresholding to the wavelet coefficients to suppress noise while retaining image features. Figure 7.5 demonstrates the denoising performance of wavelet denoising, which effectively removes noise while preserving image details.

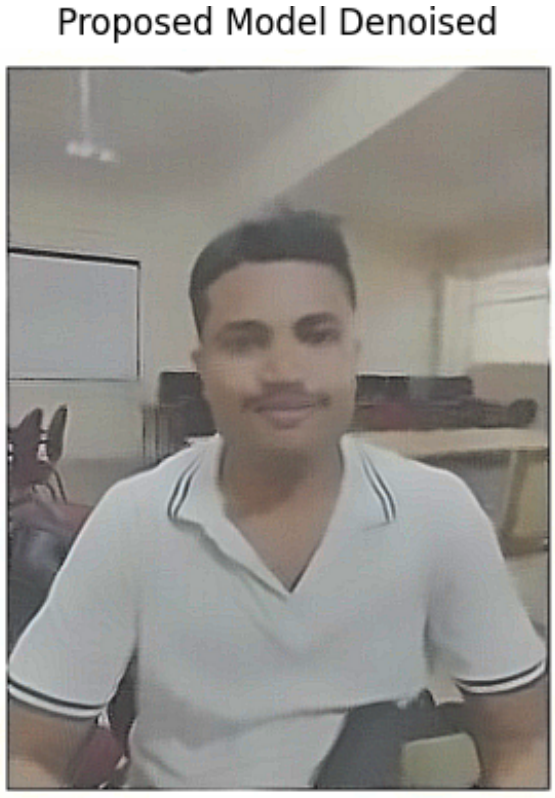
Proposed Methodology Our proposed methodology leverages a GAN (Generative Adversarial Network) model for denoising. By training the GAN on a dataset of noisy and clean images, it learns to generate high-quality denoised images. As shown in Figure 7.5, the denoised images produced by our proposed model exhibit superior quality compared to traditional methods, with minimal loss of image details.

(a) Wavelet Denoising

(b) Median Filtered

(c) Gaussian Filtered

(d) Noisy Image



(e) Proposed Model

Figure 7.5: Comparison of Different Images

Chapter 8

Result and discussion

8.1 Image Denoising Process

The following description elaborates on the process of denoising an image using the provided code. It delineates the steps involved and the significance of denoising in various image processing applications.

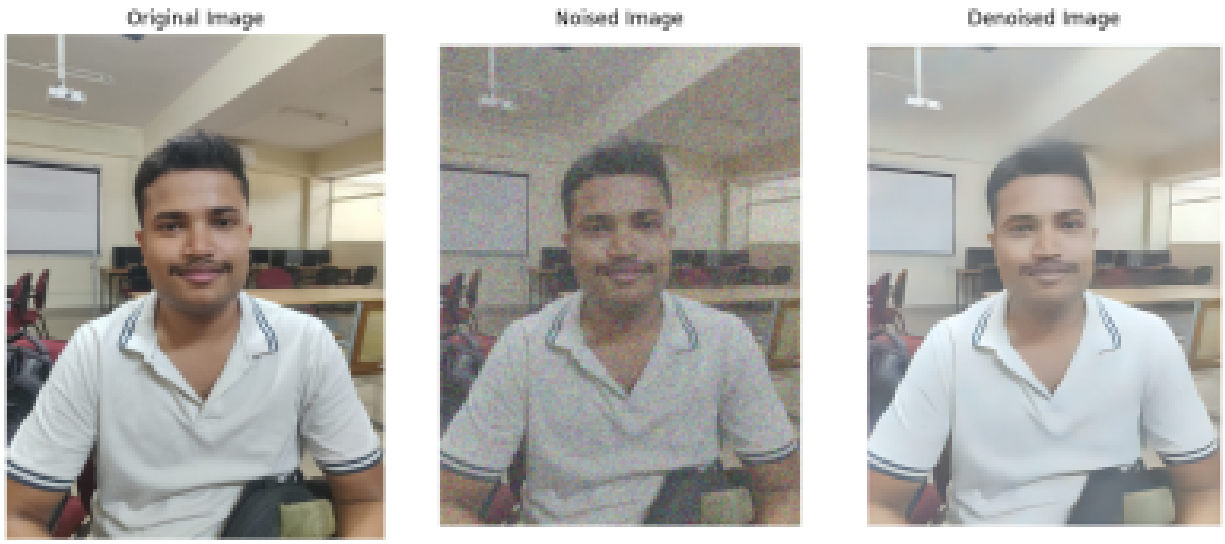


Figure 8.1: Original-Noised-Denoised

The diagram presented in Figure 8.1 illustrates the denoising process using the code discussed earlier. It portrays three iterations of the same image: the original clean image, a noisy version with added Gaussian noise, and the denoised image after applying a denoising algorithm.

Original Image: This represents the unaltered, clean image devoid of any distortions or noise.

Noised Image: This iteration of the image has been deliberately tampered with by introducing Gaussian noise. This noise manifests as random fluctuations in pixel values, resulting

in a grainy texture and obscuring the fine details and clarity of the original image.

Denoised Image: This depiction emerges as the outcome of executing a denoising algorithm, specifically the `denoise_image(image_path)` function from the provided code, on the noisy image. The denoising process endeavors to eliminate the added noise while conserving the crucial features and nuances of the original image.

Through a comparative analysis of the three images juxtaposed side by side, we can evaluate the efficacy of the denoising algorithm. The denoised image is anticipated to exhibit a smoother and clearer appearance compared to its noisy counterpart, while still retaining the essential details inherent in the original image.

This denoising procedure holds significance across various image processing realms, encompassing image refinement, computer vision applications, and image analysis endeavors, where noise imposition can adversarially impact algorithmic performance and degrade resultant output quality.

8.2 Image Denoising GUI Application

The below given image shows the graphical user interface (GUI) program created with the Tkinter library for Python. This program is specifically designed for image denoising, which is the process of removing unwanted distortions or noise from images.

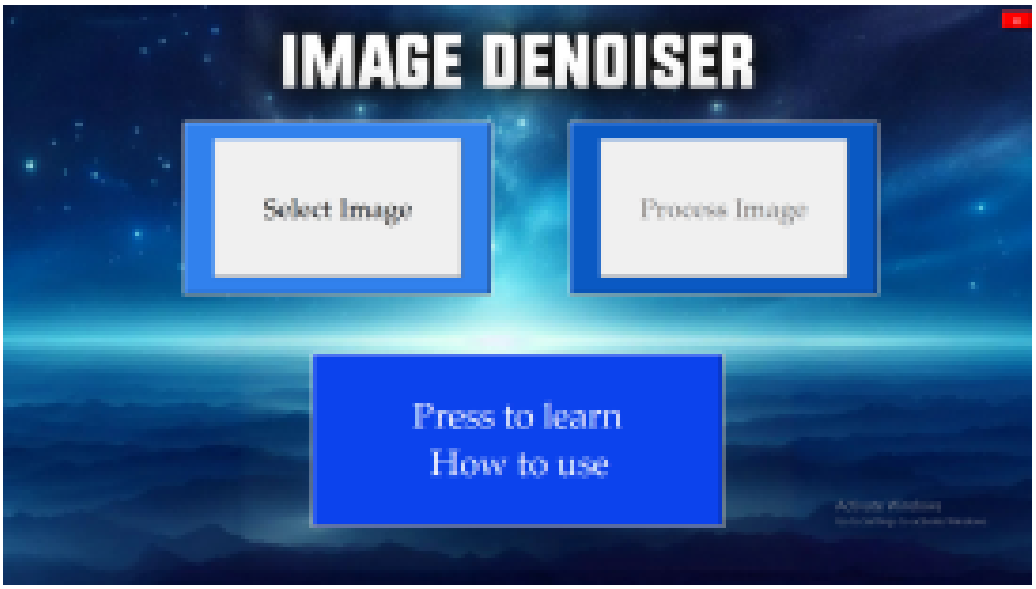


Figure 8.2: GUI landing page

8.2.1 Image Denoiser

This section comprises three main buttons:

- **Select Image**: This button allows users to browse and select an image file from their local file system for denoising.
- **Process Image**: Upon selecting an image, this button initiates the denoising process, applying the GAN Architecture to the selected image.
- **Press to learn How to use**: Offering instructions or guidance on effectively utilizing the application.

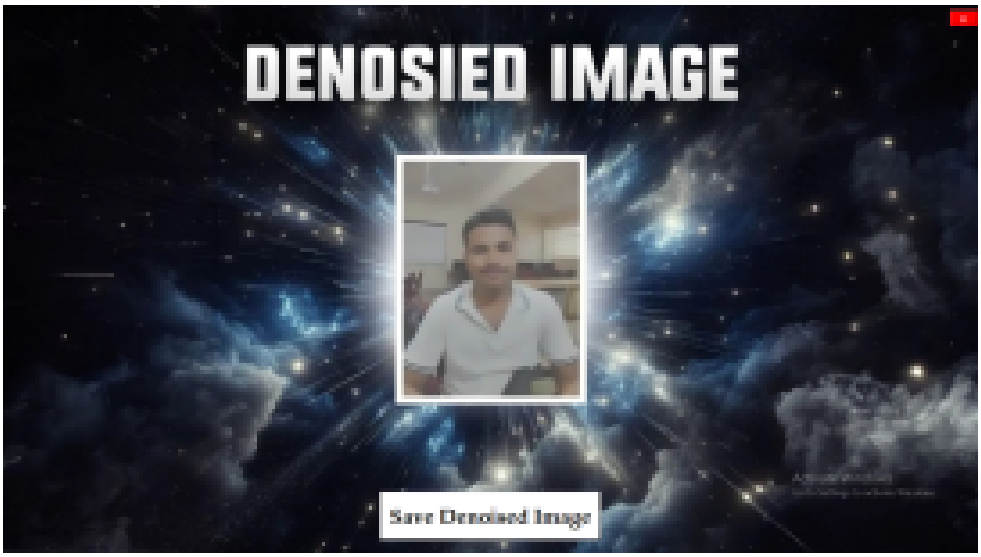


Figure 8.3: GUI result page

8.2.2 Denoised Image

This segment displays the denoised image following the user's selection and processing through the denoising algorithm. In the provided image, a denoised portrait image is exhibited within this area. Additionally, a "Save Denoised Image" button is available, assumedly allowing users to save the denoised image locally.

The application interface features a visually captivating and contemporary design with a space-themed background adorned with stars and nebulae. However, the presence of the "Activate Windows" watermark at the bottom suggests that this application may be running on an unactivated version of the Windows operating system.

In summary, this Tkinter-based GUI application in Python furnishes a user-friendly platform for executing image denoising tasks, empowering users to select images, apply denoising algorithms, and preserve denoised outputs.

Chapter 9

Conclusion and Future Scope

9.1 Conclusion

This report concentrates on training GAN for blind super-resolution in real-world scenarios exclusively using synthetic training pairs. To introduce more realistic degradation effects in images, the proposal is to employ a high-order degradation method alongside sinc filters to simulate common artifacts such as ringing and overshoot. Furthermore, a U-Net discriminator is utilized, incorporating spectral normalization regularization. This method augments the discriminator's capabilities and guarantees more stable training dynamics.

9.2 Future Scope

Research opportunities utilising Generative Adversarial Networks (GANs) to eliminate or reduce Gaussian noise in real-time CCTV image are extremely promising. One approach might be to improve GAN architectures so they can better adjust to the intricate and dynamic structure of CCTV image. This would involve taking advantage of deep learning techniques to improve noise reduction capabilities while maintaining important details.

Investigating cutting-edge training techniques like meta-learning and self-supervised learning could improve these models' resilience and applicability in a variety of surveillance scenarios and environmental settings. Furthermore, the incorporation of domain-specific knowledge, such as the comprehension of common noise patterns in security image, may facilitate the creation of more focused and effective noise reduction algorithms.

Summary

The report focuses on training GANs for blind super-resolution using synthetic training pairs, incorporating realistic degradation effects and a U-Net discriminator with spectral normalization regularization. GAN, trained on synthetic data, effectively enhances detail in real-world images while eliminating artifacts.

Bibliography

[1] Zhizhong Huang, Junping Zhang, Yi Zhang, and Hongming Shan. Du-gan: Generative adversarial networks with dual-domain u-net-based discriminators for low-dose ct denoising. *IEEE Transactions on Instrumentation and Measurement*, 71:1–12, 2022.

[2] Zhao Yang, Chenggeng Yan, and Hu Chen. Unpaired low-dose ct denoising using conditional gan with structural loss. In *2021 International Conference on Wireless Communications and Smart Grid (ICWCSG)*, pages 272–275, 2021.

[3] Shaobo Zhao, Sheng Lin, Xi Cheng, Kexue Zhou, Min Zhang, and Hai Wang. Dual-gan complementary learning for real-world image denoising. *IEEE Sensors Journal*, 24(1):355–366, 2024.

[4] Xuyang Wang, Mingzhu Long, Shaofeng Zou, Xiang Xie, Guolin Li, and Zhihua Wang. Detail recovery in medical images denoising. In *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4, 2019.

[5] Biju Venkadath Somasundaran, Rajiv Soundararajan, and Soma Biswas. Image denoising for image retrieval by cascading a deep quality assessment network. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 525–529. IEEE, 2018.

[6] Vardan Papyan and Michael Elad. Multi-scale patch-based image restoration. *IEEE Transactions on image processing*, 25(1):249–261, 2015.

[7] Yu Gan, Elsa Angelini, Andrew Laine, and Christine Hendon. Bm3d-based ultrasound image denoising via brushlet thresholding. In *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*, pages 667–670. IEEE, 2015.

[8] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. Image denoising by sparse 3-d transform-domain collaborative filtering. *IEEE Transactions on image processing*, 16(8):2080–2095, 2007.

[9] Michal Aharon, Michael Elad, and Alfred Bruckstein. K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on signal processing*, 54(11):4311–4322, 2006.

[10] Lei Zhang, Weisheng Dong, David Zhang, and Guangming Shi. Two-stage image denoising by principal component analysis with local pixel grouping. *Pattern recognition*, 43(4):1531–1549, 2010.

[11] Tobias Plotz and Stefan Roth. Benchmarking denoising algorithms with real photographs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1586–1595, 2017.

[12] Shixiang Wu, Tianqi Fan, Chao Dong, and Yu Qiao. Rds-denoiser: a detail-preserving convolutional neural network for image denoising. In *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS)*, pages 127–132, 2018.

[13] Vardan Papyan and Michael Elad. Multi-scale patch-based image restoration. *IEEE Transactions on Image Processing*, 25(1):249–261, 2016.

[14] Haizhang Zou, Rushi Lan, Yanru Zhong, Zhenbing Liu, and Xiaonan Luo. Edcnn: A novel network for image denoising. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1129–1133, 2019.

[15] Qu ZhiPing, Zhang YuanQi, Sun Yi, and Lin XiangBo. A new generative adversarial network for texture preserving image denoising. In *2018 Eighth International Conference on Image Processing Theory, Tools and Applications (IPTA)*, pages 1–5, 2018.

[16] Ziqi Gan, Xianglian Xu, Yuanfeng Qiu, Tunzhen Xie, Yating Chang, Xinrui Wu, and Ruiqing Zhao. Application of improved wavelet threshold denoising method in power quality signals with noise. In *2023 4th International Conference on Advanced Electrical and Energy Systems (AEES)*, pages 156–160, 2023.

[17] Kaihua Gan, Jieqing Tan, and Lei He. Non-local means image denoising algorithm based on edge detection. In *2014 5th International Conference on Digital Home*, pages 117–121, 2014.

[18] Turhan Kimbrough, Pu Tian, Weixian Liao, and Wei Yu. Performance of gan-based denoising and restoration techniques for adversarial face images. In *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 335–340, 2023.

[19] Cheng-Ho Wu and Herng-Hua Chang. Gaussian noise estimation with superpixel classification in digital images. In *2012 5th International Congress on Image and Signal Processing*, pages 373–377, 2012.

[20] Yoann Le Montagner, Elsa D. Angelini, and Jean-Christophe Olivo-Marin. An unbiased risk estimator for image denoising in the presence of mixed poisson–gaussian noise. *IEEE Transactions on Image Processing*, 23(3):1255–1268, 2014.