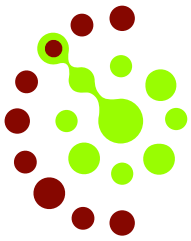

IPBeja

INSTITUTO POLITÉCNICO
DE BEJA

Higher School of Technology and Management

Master's Degree in Internet of Things

Noise Reduction in Radio Astronomy Images using ML

a comparison of methods for detection and treatment

Pedro Miguel Medinas Fresco

INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Mestrado em Internet das Coisas

Noise Reduction in Radio Astronomy Images using ML

a comparison of methods for detection and treatment

Pedro Miguel Medinas Fresco

Advisor :

Dr. João Carlos Martins, IPBeja

Master's thesis

Abstract

Noise Reduction in Radio Astronomy Images using ML

a comparison of methods for detection and treatment

This thesis has two objectives. The first is to present and make an overview of radio astronomy. This understanding is fundamental to pave the way for the second objective.
The second objective is to tackle the intricate and challenging task of noise reduction in astronomical data obtained by radio telescopes.
We do this by using two distinct Machine Learning (ML) models recurring to an HPC using tensorflow GPU. An Autoencoder model capable of reducing the noise by itself, and a CNN, that classifies the noise type and then needs a noise reduction algorithm to reduce it.
The results of both approaches are compared and what option makes the most sense to use is proposed.

Keywords: *Smartglow, Machine Learning, Radio Astronomy, Artificial Intelligence, HPC, CUDA, Tensorflow, Keras, CNN, Python, Phosim, SAOImageDS9, Noise Reduction.*

Resumo

Redução de ruído em imagens de radioastronomia usando ML

Comparação de métodos de deteção e tratamento

Esta dissertação tem dois objetivos, o primeiro é apresentar e descrever uma visão geral da astronomia de rádio de modo a estabelecer uma base para desenvolver trabalho em radio astronomia. O segundo objetivo é ser capaz de reduzir o ruído nos dados astronómicos, usando e treinando dois modelos de ML, usando HPC usando tensorflow GPU, um Autoencoder capaz de reduzir o ruído e um CNN que classifica o ruído e precisa de um algoritmo de redução de ruído para a redução. E no fim comparar os resultados de ambos e decidir qual opção faz mais sentido usar neste caso.

Palavras-chave: *Smartglow, Learning, Radio Astronomy, Machine Learning, Artificial Intelligence, HPC, CUDA, Tensorflow, Keras, CNN, Python, Phosim, SAOImageDS9, Noise Reductiono.*

acknowledgements

I thank my entire family for being the support for the realization of my life goals, specially to my grandpa that passed away during the writing of this thesis. I thank my thesis advisor Doctor João Carlos Martins, ESTIG/IPBeja who accompanied me, for all his collaboration, availability, patience, understanding and professionalism that allowed me to complete this thesis. I thank all the teachers who accompanied me in my training, for enriching my knowledge. I thank all the colleagues, with whom I had the pleasure of working as a team and who provided me with great learning experiences.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Introduction

This thesis is based on the work done in the SmartGlow project. Whose main objective is to specify and develop a flexible solution for the production and self-consumption of renewable energy with applications in the industrial and healthcare sectors, as well as in emergency situations. The final solution is an integrated platform that incorporates high-efficiency conversion solutions, innovative applications for microgrid and flexible load management, and advanced analytics for processing critical data. The project aims to demonstrate the solution in the context of the SKA-Low pilot [1] in Portugal, which has demanding requirements for waveform quality and low EMI noise.
Aside the main goal, within the context of the Smartglow project, the SKALA radio telescope antennas produce images of the space that collects noise. In summary, the SmartGlow project aims to create a flexible solution for renewable energy production and self-consumption, with a focus on industrial and healthcare sectors and emergency situations. The solution will be an integrated platform that includes efficient energy conversion, innovative microgrid and load management, and advanced data analytics.
This thesis will focus on an overview of the relevant information related to noise in astronomical data and how to identify and attenuate it.

1.2 Work Overview

The first part of this thesis is focused on understanding what is radio astronomy and the related relevant information to reduce noise and showcase the technologies used in this work.
The second part involves the creation of a dataset and developing ML models, namely a CNN and various Autoencoder models, and comparing the results of its application to the noise classification and removing. The workflow of the second part can be visualized in the Fig.1.1.

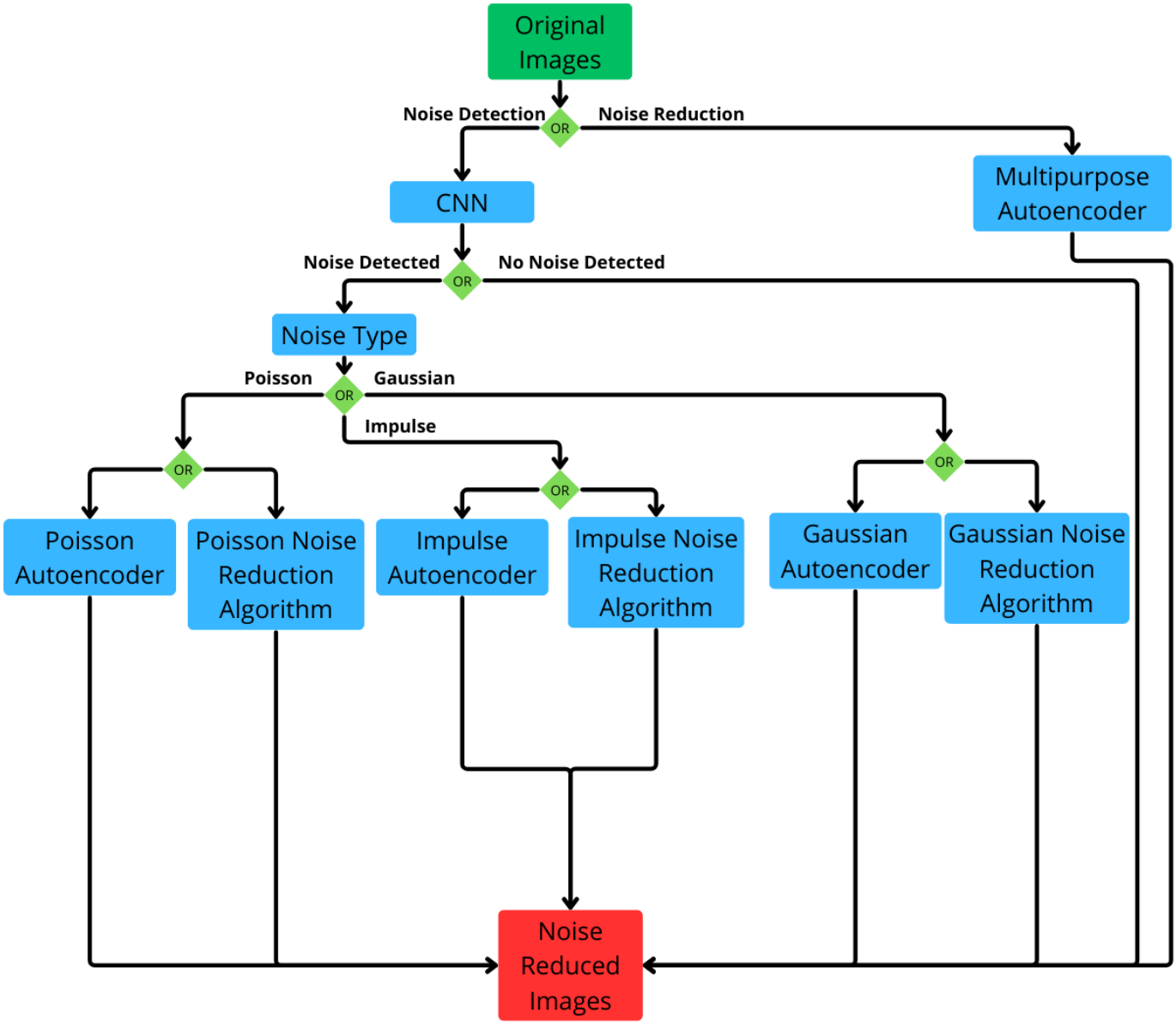

Figure 1.1: Models Workflow

1.3 Document Organization

The thesis is structured in the following way:

- Chapter 1: Introduction to the project and problem this thesis is based on and how the workflow of the implemented solution.
- Chapter 2: A general overview of technologies used in the thesis and a in depth overview of what radio astronomy is, what it is for and relevant terms and information

to the problem and the implemented solution.

- Chapter 3: Step by step implementation of the solution, from how to visualize the astronomical data, how the dataset used in training was simulated and added noise, the code used to train the models and the various steps used to decrease its runtime and the results from the training, validation and predictions.
- Chapter 4: Conclusions taken from the results and possible changes to do in future work.

Chapter 2

Radio Astronomy Background and State-of-the-Art

2.1 Introduction

This chapter, has a in depth explanation of radio astronomy and important themes related to the problem this thesis is solving: how the data in radio astronomy is usually formatted, what are the origin and the types of noise, how to detect and common ways of reducing noise, and others technologies used in the thesis.

2.2 Radio Astronomy

Radio astronomy uses radio telescopes to analyze radio frequency emissions from celestial bodies such as stars, galaxies, nebulae, and black holes, as seen in Fig.2.1. These emissions are part of the electromagnetic spectrum and provide valuable information about the properties of celestial objects, such as their chemical makeup, temperature, magnetic fields, and movements [2].

Radio astronomy uses radio telescopes to observe and study, celestial objects by detecting and analyzing their radio frequency emissions. Various frequency bands are used for radio astronomy observation, and some are recognized as generally accepted spectral regions. While some segments are shared, and not protected explicitly from interference by other authorized users, others are protected by the International Astronomical Union (IAU) to avoid in-band, band-edge, and harmonic emissions. Some examples of protected frequencies used for radio astronomy are [3]:

- Hydrogen (HI): 1420.406 MHz
- Carbon monoxide (CO): 115.271 GHz, 230.538 GHz, and 691.473 GHz

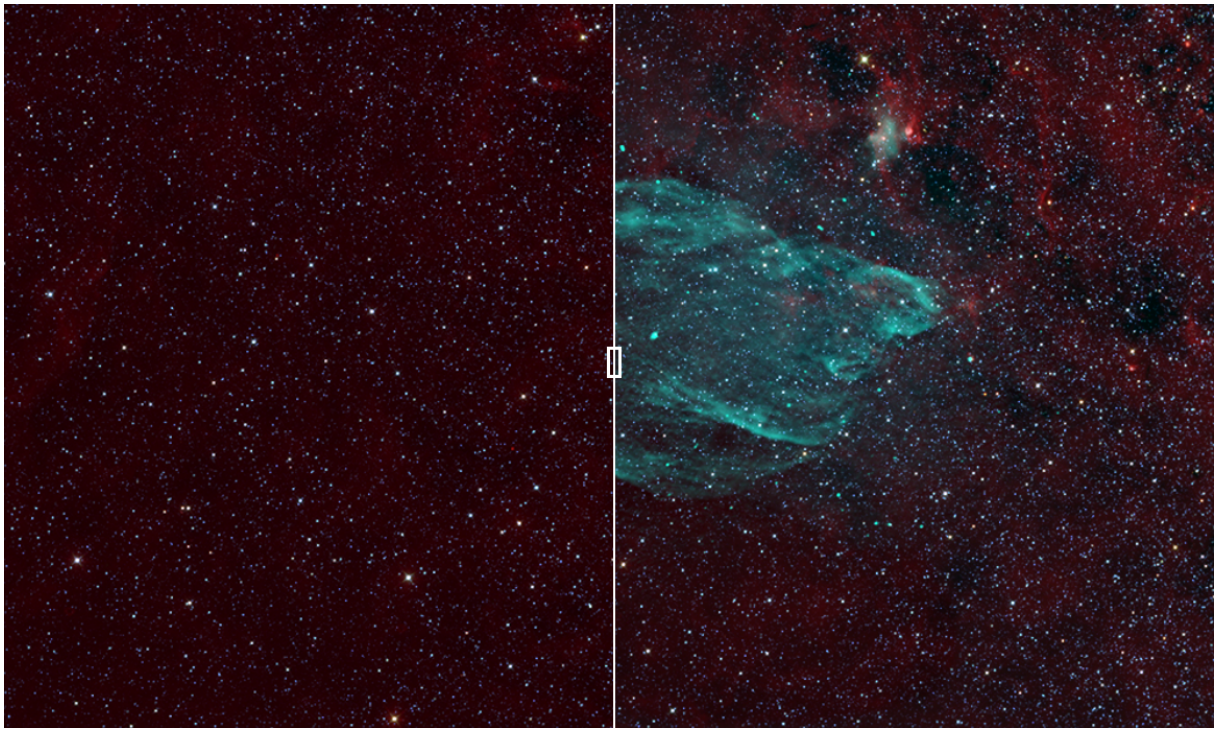

Figure 2.1: The W50 "Manatee" Nebula shown in visible light (Left) and visible + radio light (Right) [2].

- Water vapor (H2O): 22.235 GHz and 380.197 GHz
- Formylium (HCO+): 89.189 GHz and 802.653 GHz
- Hydrogen cyanide (HCN): 88.632 GHz and 797.433 GHz

Radio astronomy aims to understand celestial objects' properties and behavior. For example, radio telescopes can be used to study the radio emissions of hydrogen, a key component of galaxies, and map its distribution in space. They also examine the radio emissions from molecular clouds, where stars are born. Additionally, radio astronomy also aims to study the formation and evolution of stars, galaxies, and black holes. By observing the life cycle of stars, mapping the distribution of stars and galaxies in the universe, and understanding the impact of supermassive black holes on their surroundings, radio astronomers gain insights into the history and structure of the universe.
Another objective of radio astronomy is searching for signals from extraterrestrial civilizations. The Search for Extraterrestrial Intelligence (SETI) is an ongoing effort to detect signals from other civilizations. Radio telescopes are used to look for artificial signals in the radio frequency range, which may suggest the existence of extraterrestrial civilizations [4].

2.2.1 Radio Telescopes

Radio telescopes are the primary tools used in radio astronomy to collect and analyze radio frequency emissions from celestial objects. They come in two main types: ground-based and space-based. Each type has advantages and limitations, impacting the choice of which telescope to use for a specific observation [5].

Ground-based Telescopes

Ground-based radio telescopes are typically installed on the Earth's surface and can be single dishes or arrays. They come in various sizes and shapes and can be configured to observe different parts of the radio spectrum. The performance of ground-based telescopes is influenced by the size of the dish or array, the altitude of the telescope, and the shape of the dish or array. Some examples of ground-based radio telescopes include the Arecibo Observatory in Puerto Rico, the Very Large Array in New Mexico, the Square Kilometer Array (SKA), and the MeerKat, as seen in Fig.2.2, in South Africa. Ground-based telescopes have the advantage of being able to observe large areas of the sky at once. However, they are limited by interference from the Earth's atmosphere and human-made radio signals [6].

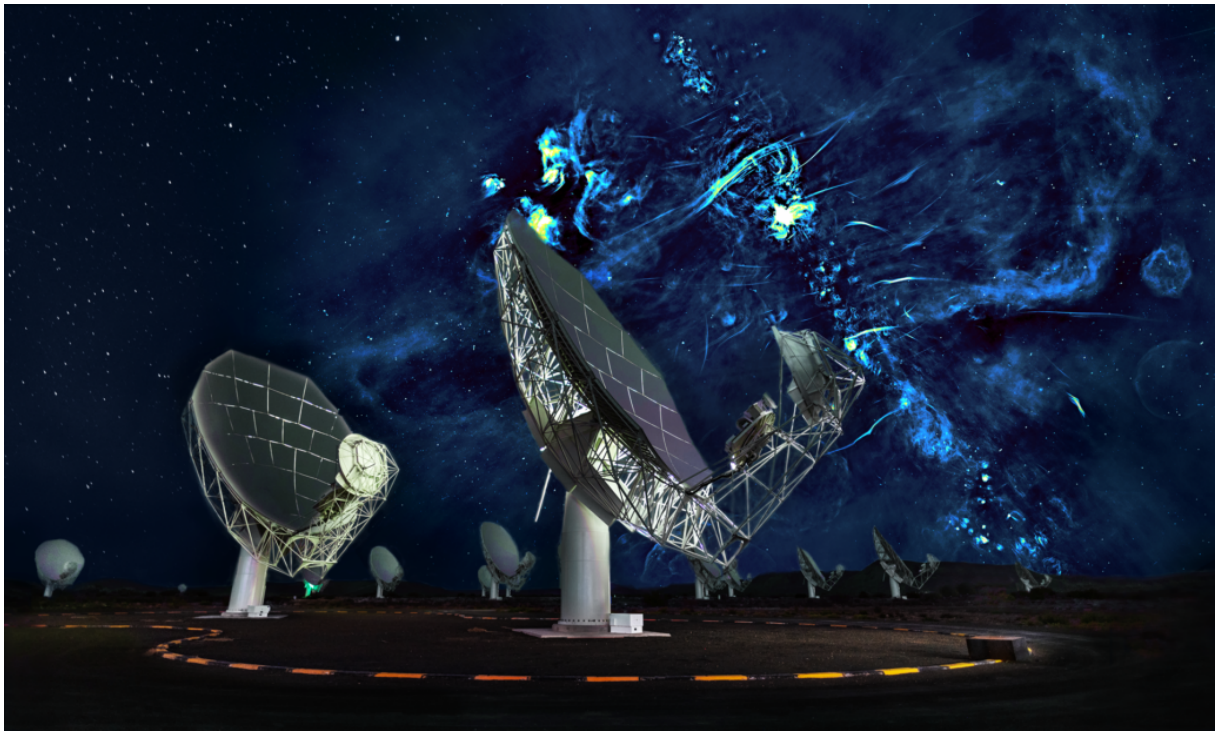

Figure 2.2: MeerKat Telescope [7].

Space-based Telescopes

Space-based radio telescopes are placed in orbit around the Earth and are not influenced by the Earth's atmosphere. They can observe the universe with much greater sensitivity and resolution than ground-based telescopes, and are not limited by interference from the Earth's atmosphere and human-made radio signals. However, they are limited by their short lifespan and installation and operation costs. Some examples of space-based radio telescopes include the Hubble Space Telescope Fig.2.3, the Chandra X-ray Observatory, and the James Webb Space Telescope. The telescope's orbit influences the performance of space-based telescopes, the telescope's design, and the telescope's lifespan [8].

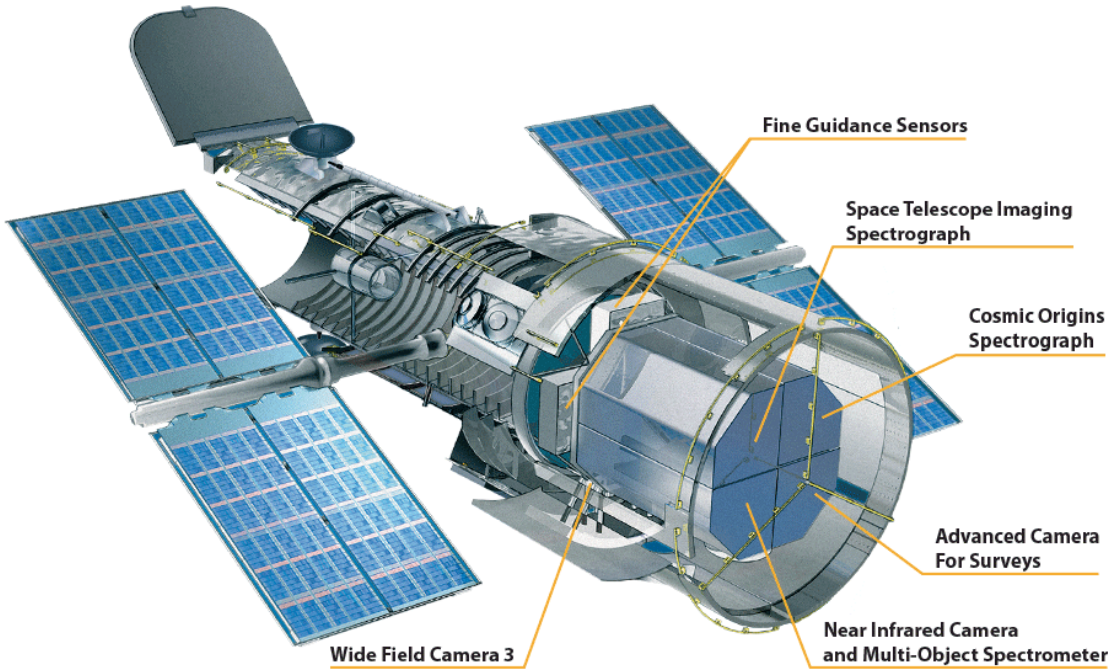

Figure 2.3: The Hubble Space Telescope [9].

2.2.2 Radio Astronomical Data Formats

Radio astronomical data can be represented in various formats. Understanding these different data formats is essential for interpreting and analyzing the data collected by radio telescopes. The following are the most common data formats and their visual representations.

1. **Time-domain signals:** Time-domain signals represent the radio astronomical data as a function of time. This data format can be represented visually as a linear graphic, as seen in Fig.2.4, where the signal's amplitude is plotted against time. Time-domain signals help study the time-variability of astronomical objects and for detecting transient events [10].

2. **Spectra:** Spectra represents the radio astronomical data as a function of wavelength. This type of data format can be represented visually as a spectrum plot, as seen in Fig.2.5, or a spectrogram. A spectrum plot shows the signal's amplitude as a function of wavelength, while a spectrogram shows both the amplitude and wavelength over time. Spectra helps study the spectral properties of astronomical objects, such as their spectral lines and continuum emission [10].

3. **Images:** Images represent the radio astronomical data as an image, with each pixel representing the intensity of the signal at a particular location in the sky, as seen in Fig.2.6. Images help study the spatial distribution of astronomical objects and for creating maps of the sky [10].

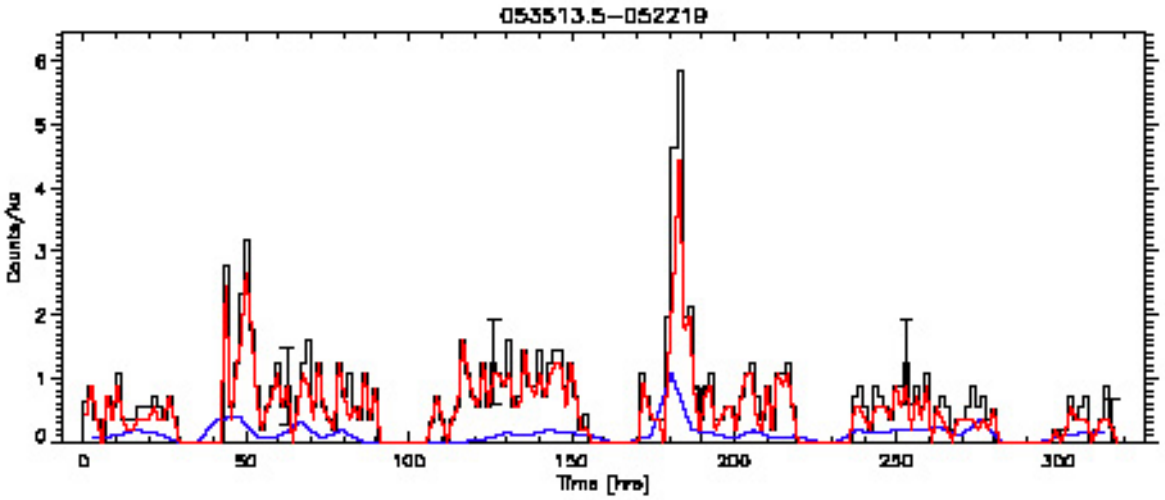

Figure 2.4: A faint flaring Orion star with 678 photons [10].

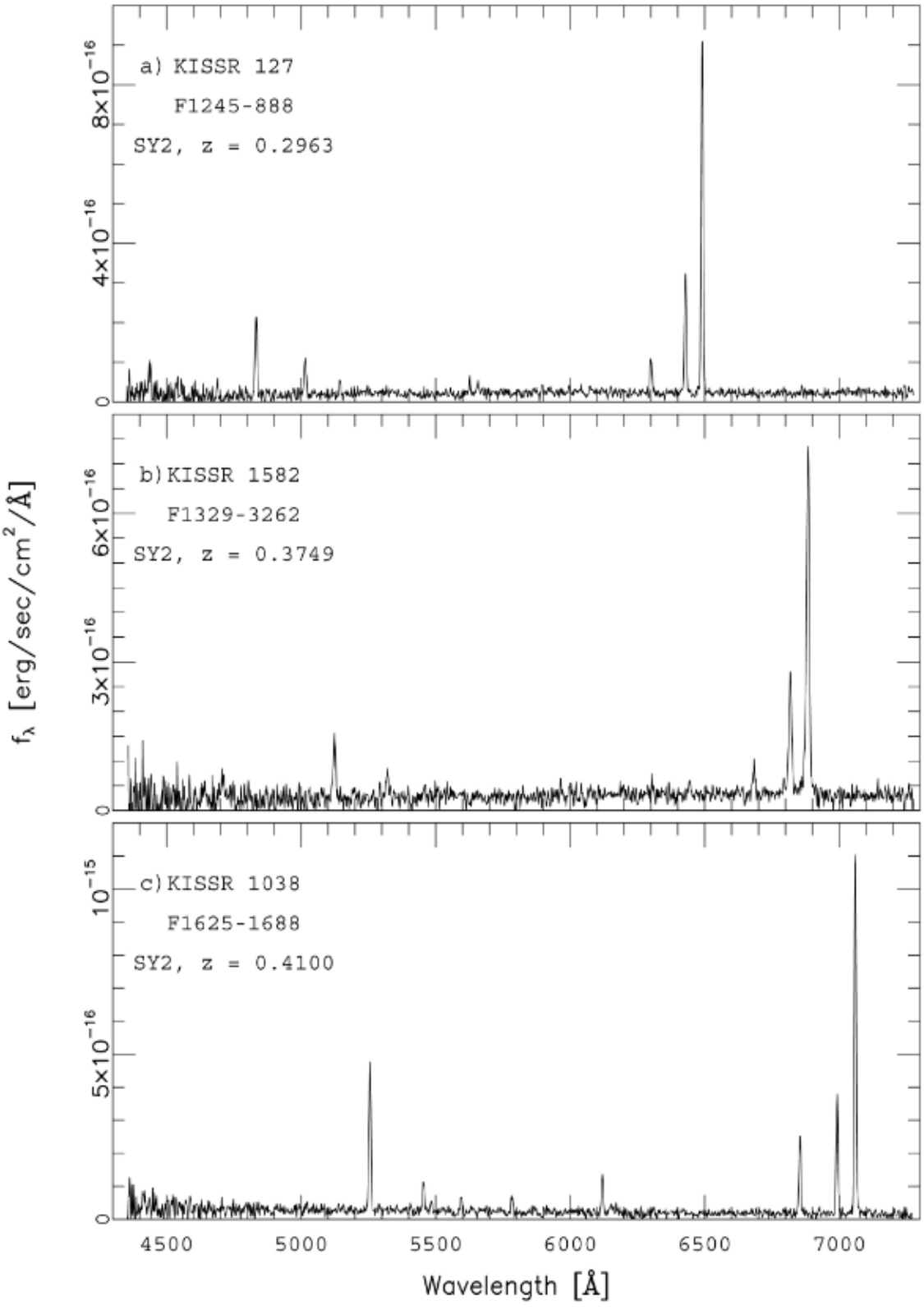

Figure 2.5: Three spectra from Seyfert 2 galaxies [10].

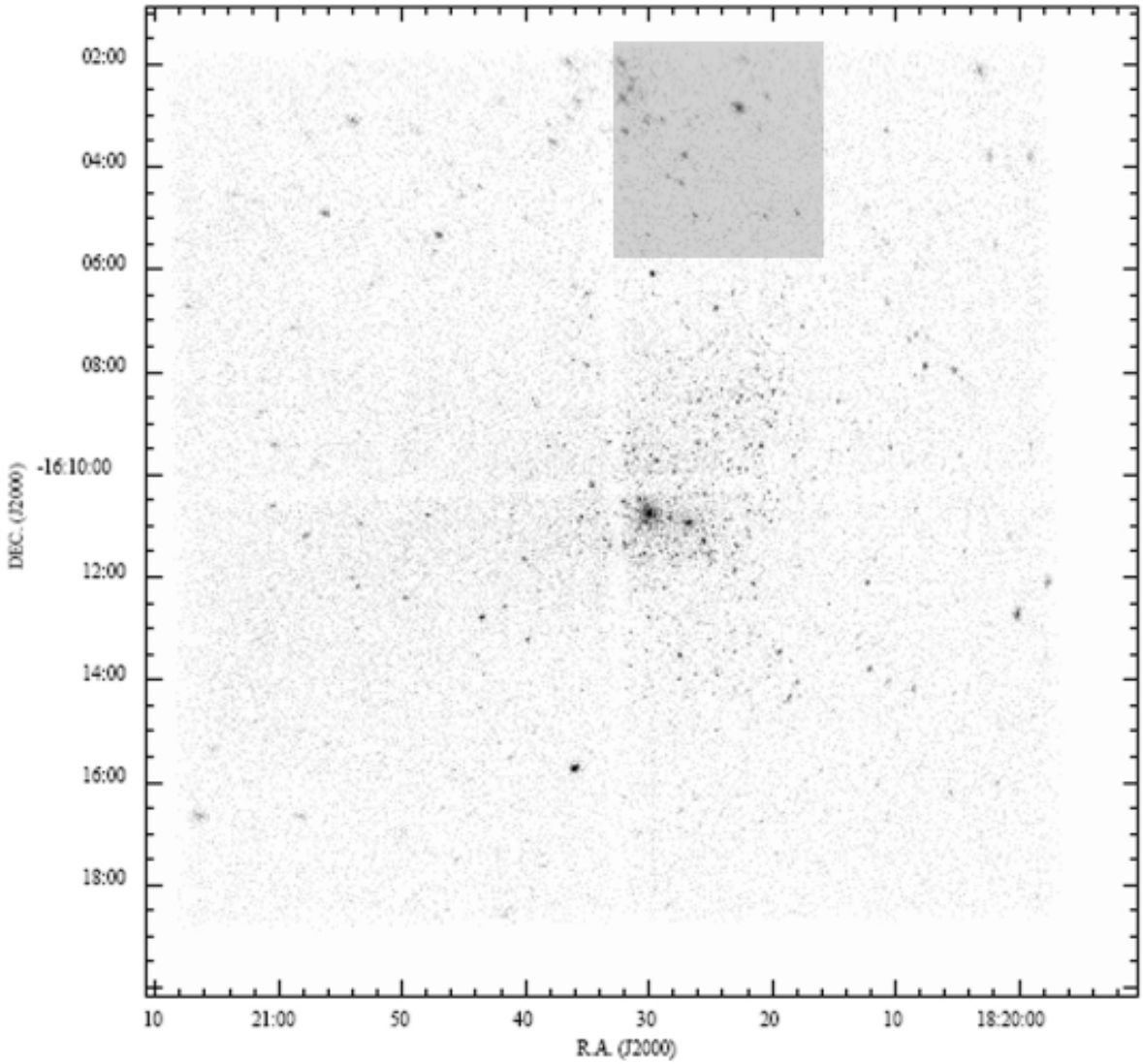

Figure 2.6: M17-North region with a cluster of protostars embedded deep in the host molecular cloud [10].

2.3 Noise In Astronomical Images

Electromagnetic noise is a common challenge in radio astronomy, as it can contaminate the data collected by radio telescopes and impact the accuracy of astronomical measurements. Several types of noise can affect radio astronomical data, each with its unique characteristics and origins. To ensure high-quality data, it is crucial to understand and effectively mitigate the impact of noise.

2.3.1 Origin of Noise

Understanding the origin of the various types of noise that can impact the data collected by radio telescopes is essential. Different sources of radio frequency energy can result in different types of noise, each with its unique characteristics. Understanding the origin of these noises is crucial in developing effective strategies for mitigating their impact on astronomical measurements.

1. **Radio Frequency Interference (RFI):**
   RFI is caused by artificial sources of radio frequency energy, such as cell phone towers, radio and television broadcast stations, and military and aviation communications systems. RFI can interfere with the measurement of astronomical signals, appearing as narrow-band or broad-band interference. The interference from these sources can be modeled as Gaussian noise [11].

2. **Thermal Noise:**
   Thermal noise is caused by the random motion of electrons in electronic components. The noise increases with temperature and bandwidth and is modeled as Gaussian noise. Reducing thermal noise is an important consideration in designing radio telescopes, as it is a fundamental limit to their sensitivity [12].

3. **Atmospheric Emission:**
   Atmospheric emission is caused by the emission of radio frequency energy from the Earth's atmosphere. This type of noise is modeled as a baseline drift that is not perfectly Gaussian but has a wide distribution of amplitudes [13].

4. **Cosmic Noise:**
   Cosmic noise, also known as galactic radio noise, is high-energy particles that can penetrate the front-end electronics of a radio telescope and cause impulse noise [14].

5. **Systematic Errors:**
   Systematic Errors can be caused by imperfections in the hardware or software used to make the observations. These errors can result in biases in the data and affect the accuracy of astronomical measurements. The distribution of systematic errors

can vary depending on the specific type of error. In radio astronomy, instrumentation effects can be a common source of systematic errors, such as gain variations or polarization errors [15].

2.3.2 Types of Noise

Different types of noise can affect the data collected by radio telescopes in different ways. The type of noise impacts the signal-to-noise ratio and the accuracy of astronomical measurements. Understanding the different types of noise is essential to mitigate their impact. The following are the most common types of noise that affects radio astronomical data:

1. **Gaussian Noise:**
   Gaussian noise is a type of noise that can be modeled as a normal distribution. It can appear as white noise, which has equal power across all frequencies. It is often used as a reference in signal processing to describe random signals with the same power in all frequency bands or as narrow-band/broad-band noise, affecting only a specific range of frequencies. Gaussian noise is uncorrelated, meaning that the value of the noise at one point in time does not have any relationship to the value of the noise at any other point in time and is widely used in many fields, including radio astronomy, telecommunications, and signal processing. The standard deviation of the Gaussian noise can be adjusted to control the amount of noise in the signal, and it is often added to signals to test the performance of signal processing algorithms and systems [16].

2. **Impulse Noise:**
   Impulse noise is characterized by short, sharp spikes or glitches in the data. This type of noise is often caused by high-energy particles, such as cosmic rays, that penetrate the front-end electronics of a radio telescope. Impulse noise can be difficult to remove from the data, as it can appear suddenly and unpredictably [17].

3. **Baseline Drift:**
   Baseline Drift is characterized by a background signal that is not constant but changes over time. This type of noise is modeled as a baseline drift that is not perfectly Gaussian, meaning that it does not follow a normal distribution but has a wide distribution of amplitudes. Unlike other types of noise, baseline drift noise cannot be easily modeled or removed from the data, as it is inherently present in the environment [18].

4. **Narrow-Band and Broad-Band Interference:**
   Narrow-band and Broad-Band Interference are types of noise caused by artificial radio frequency energy sources, such as cell phone towers, radio and television broadcast stations, and military and aviation communications systems. This type of noise is

modeled as a signal added to the astronomical signal, and it can appear as a narrow-band signal that affects only a specific range of frequencies or as a broad-band signal that affects all frequencies equally and is modeled as Gaussian noise. Narrow-band and Broad-Band Interference can be difficult to remove from the data, as it is often present in the environment and can appear suddenly and unpredictably [19].

2.3.3 Noise Detection

Detecting noise in radio astronomical data is crucial to ensure accurate results. Various methods are used to detect and remove noise, including flag and mask, statistical analysis, and machine learning [20].

1. **Flag and Mask:**
   This method involves marking data samples as "bad" (i.e., contaminated by noise) and excluding them from further analysis. This is usually done through manual inspection of the data, where an astronomer will visually inspect the data and identify any samples that appear to be contaminated by noise. These samples are then flagged and masked, marking them as insufficient data and excluded from further analysis [21].

2. **Statistical Analysis:**
   This method involves analyzing the statistical properties of the data (e.g., mean, variance, distribution) to identify and remove samples that deviate significantly from the norm. For example, an astronomer may calculate the mean and standard deviation of the data and exclude any samples that fall outside of a certain number of standard deviations from the mean. This method can also be combined with machine learning algorithms to increase accuracy and efficiency [22].

3. **Machine Learning:**
   This method involves training machine learning algorithms to identify and remove noise in the data based on examples of noise and clean data. For example, an astronomer may provide the algorithm with a large dataset of clean data and a smaller dataset of noisy data and train the algorithm to recognize the differences between the two. The algorithm can then be applied to new data to identify and remove any samples contaminated by noise. This method is becoming increasingly popular due to its ability to quickly and accurately identify and remove noise in large datasets [23].

2.3.4 Noise Reductions Algorithms

The detection and reduction of noise in radio astronomy data is a important step in data analysis. To effectively reduce noise, it is vital to have a thorough understanding of the type

and origin of the noise and the characteristics of the data itself. A variety of algorithms can be used to reduce noise in radio astronomy data, some of them include the following:

1. **Principal Component Analysis (PCA):**
   PCA is a method of reducing the dimensionality of a dataset by transforming it into a new set of orthogonal components. These new components, known as principal components, capture the most significant variations in the data. In radio astronomy, PCA can separate the astronomical signal from the noise. The first few components represent the astronomical signal, while the remaining components represent the noise. PCA effectively reduces thermal noise and atmospheric emission noise due to its ability to capture the most significant variations in the data [24].

2. **Kalman Filters:**
   Kalman filters are a class of algorithms that can be used to estimate the state of a dynamic system based on noisy measurements. In radio astronomy, Kalman filters can estimate the astronomical signal by filtering out the noise. These filters are particularly effective at removing narrow-band RFI and cosmic ray noise because they are designed to handle the dynamic nature of the system. Kalman filters use a recursive algorithm to estimate the system's state based on measurements [25].

3. **Wavelet Analysis:**
   Wavelet analysis is a method of analyzing signals in both the time and frequency domains. This technique decomposes the data into different frequency components, allowing for the analysis of time-frequency and time-scale information. In radio astronomy, wavelet analysis can detect and remove impulse noise, such as cosmic ray hits, while preserving the astronomical signal. Wavelet analysis effectively removes impulse noise because it allows for the analysis of the signal at different frequencies, making it easier to identify and remove noise [26].

4. **Singular Value Decomposition (SVD):**
   SVD is a powerful mathematical technique that separates a data set's most significant features from the noise. In radio astronomy, SVD is used to identify the astronomical signal and separate it from the noise. This is achieved by decomposing the data into a set of orthogonal components and selecting the most important components to represent the astronomical signal. SVD is particularly effective at reducing thermal noise and atmospheric emission noise [27].

5. **Sparse Representation:**
   Sparse Representation separates the astronomical signal from the noise by representing the signal as a combination of a small number of basis functions. This approach is practical because it allows the noise and the astronomical signal to be treated as

separate components, making it easier to remove it while preserving the astronomical signal. Sparse Representation is beneficial for removing narrow-band RFI [28].

6. **Deep Learning:**
   Deep learning algorithms use artificial neural networks to model the complex relationships in the data. In radio astronomy, these algorithms can be trained to separate the astronomical signal from noise. This is done by providing the algorithm with examples of clean and noisy data, allowing it to learn how to distinguish between them. Deep learning algorithms have shown promising results in removing various types of noise, including RFI and thermal noise [29].

7. **Wiener Filter:**
   The Wiener filter is a mathematical method that uses a linear time-invariant filter to estimate the desired signal by removing additive noise. It minimizes the mean square error between the actual and desired signals. This method is widely used in signal detection applications, such as speech detection, to improve signal quality by reducing noise [25].

8. **Boll Spectral Subtraction:**
   Boll Spectral Subtraction is a noise reduction algorithm that removes additive noise from a signal. This algorithm works by partitioning the noisy signal into small frames and multiplying each frame by a window function. This helps to separate the signal and noise components and reduce the noise in the signal [25].

9. **Signal Dependent Rank Order Mean (SDROM):**
   SDROM is a non-linear algorithm designed to remove noise from highly corrupted signals while preserving the signal's characteristics. This algorithm replaces corrupted samples by estimating the true value based on the information from neighboring samples. SDROM is a computationally efficient method for reducing noise in signals and is well-suited for highly corrupted signals where other methods may not be effective [25].

10. **Discrete Wavelet Transform Filter for White Gaussian Noise (DWT):**
    DWT is a popular algorithm used to remove white Gaussian noise from signals. White Gaussian noise is a type of noise spread uniformly over all frequencies, and it can significantly impact the signal quality. The DWT filter converts the noisy signal into the wavelet domain, where separating the signal and the noise is more manageable. The result of this process is a denoised signal that is free from white Gaussian noise [25].

11. **Least Mean Square Filter (LMS):**
    LMS is a type of filter that is used to reduce noise in signals. It minimizes the mean

square error between the actual and the desired signals. The LMS filter updates its filter coefficients over time, allowing it to adapt to signal and noise changes. This makes it a helpful tool for reducing noise in signals that are subject to time-varying noise [25].

12. **Median Filter:**
    The median filter is a non-linear algorithm that reduces impulse noise, such as cosmic ray hits, in signals. This type of noise is characterized by sudden spikes in the signal that can have a significant impact on its quality. The median filter replaces each pixel in the signal with the median value of its neighboring pixels. This effectively removes the spikes in the signal while preserving its overall structure and features. The median filter is a simple and effective tool for reducing impulse noise [30].

13. **Total Variation Denoising:**
    Total Variation Denoising minimizes the total variation of the data while preserving its edges and features. It effectively reduces Gaussian noise and other types of smooth noise by considering the total variation of the signal [31].

14. **Independent Component Analysis (ICA):**
    ICA separates the independent signal sources from the data by considering their statistical independence. It is an effective method to reduce noise with a statistical distribution different from astronomical signals. while similar to PCA, the difference is that PCA finds the main patterns in the data, while ICA separates the sources of information in the data [32].

15. **Non-Negative Matrix Factorization (NMF):**
    NMF is a matrix factorization technique that separates the astronomical signal from the noise by identifying the most significant components in the data. It is particularly effective at reducing noise with a statistical distribution different from an astronomical signal [33].

16. **Empirical Mode Decomposition (EMD):**
    EMD involves breaking down the data into a set of intrinsic mode functions (IMFs) and oscillatory signals that capture the most significant variations in the data. EMD can be used to separate the astronomical signal from the noise in radio astronomy and is most effective when the noise has a different frequency distribution than the astronomical signal [34].

17. **Adaptive Filters:**
    Adaptive Filters are algorithms that estimate the desired signal by removing the additive noise. In radio astronomy, these filters can reduce noise by adjusting the filter coefficients based on the current data. Adaptive filters are handy when the noise changes over time [35].

2.3.5 Evaluating Noise Reduction Performance

The assessment of the effectiveness of the noise reduction algorithm is an essential step in the data analysis process. noise reduction algorithms can reduce noise, but they may also affect astronomical signals. Therefore, it is crucial to assess the algorithm's performance and compare it with the original data to ensure that the desired level of noise reduction has been achieved without compromising the quality of the astronomical signal. The following metrics can be used to evaluate the performance of the noise reduction algorithm:

1. **Signal-to-Noise Ratio (SNR):**
   SNR is one of the most commonly used metrics for evaluating the performance of a noise reduction algorithm. It is defined as the ratio of the signal's power to the power of the noise in each signal. A higher SNR value indicates that the noise reduction algorithm has effectively removed noise from the signal [36].

2. **Mean Squared Error (MSE):**
   MSE measures the average of the squared difference between the original signal and the reconstructed signal. Lower MSE values indicate that the noise reduction algorithm has effectively removed noise from the signal [37].

3. **Peak Signal-to-Noise Ratio (PSNR):**
   PSNR is another metric that measures the quality of the reconstructed signal. It is defined as the ratio of the signal's maximum power to the power of the noise in the reconstructed signal. Higher PSNR values indicate a higher quality of the reconstructed signal [38].

4. **Visual Comparison:**
   A visual comparison of the original signal and the reconstructed signal can be used to evaluate the performance of a noise reduction algorithm. This can be done by overlaying the two signals on a graph or image, which allows for the assessment of how much noise has been removed from the signal and how closely the reconstructed signal resembles the original signal [39].

5. **Comparison with Ground Truth:**
   If ground truth data is available, the noise-reduced data can be compared with the ground truth data to determine the accuracy of the noise-reduction algorithm. This method provides a more accurate evaluation of the performance of the noise reduction algorithm as it allows for a direct comparison with actual, verified data [40].

2.4 Machine Learning (ML)

In this thesis a set of ML models are used to detect and reduce noise in radio astronomy images. Machine learning is a subfield of artificial intelligence that focuses on developing al-

gorithms that can automatically learn from data and improve their performance over time. Machine learning algorithms can solve many problems, including classification, regression, clustering, and dimensionality reduction.

Deep learning is a type of machine learning technique that uses artificial neural networks to model complex relationships in the data. Deep learning algorithms are practical in various applications, including image classification, natural language processing, and speech recognition.

In radio astronomy, deep learning algorithms can be used to detect and remove noise from the data generated by radio telescopes. These algorithms can be trained on large datasets to learn the characteristics of the astronomical signal and the noise, and they can then be used to separate the two in real-time. Deep learning algorithms have shown promise in removing RFI and thermal noise and have the potential to be used in a wide range of radio astronomy applications.

In addition to detecting and removing noise, deep learning algorithms can also be used for tasks such as source extraction and classification and for improving the accuracy and speed of data processing pipelines. Their ability to learn from data and improve their performance over time makes them well-suited for use in radio astronomy, where the data generated by radio telescopes can be complex and challenging to process [41].

2.5 Compute Unified Device Architecture (CUDA)

In this thesis CUDA was used with its connection with tensorflow GPU for the sake of decreasing training run time. CUDA is a parallel computing platform and API developed by NVIDIA that leverages the power of GPUs (Graphics Processing Units) to boost computing performance dramatically. CUDA enables the development of high-performance applications for various domains, including scientific computing, data analytics, machine learning, and computer graphics, among others.

Parallel computing involves the simultaneous execution of multiple calculations or processes to tackle complex problems. This approach is instrumental in high-performance computing when large amounts of data need to be processed, such as astronomical data. Parallel computing leverages the processing power of multiple processors, GPUs, or computers to solve problems much faster than a single processor or computer could.

CUDA for parallel computing brings several benefits, such as improved computational performance, more efficient data processing and analysis, shorter processing times, and increased efficiency in solving complex problems. Additionally, CUDA provides a comprehensive development environment and an extensive library of algorithms, making it easier for developers to create high-performance applications without writing complex code.

CUDA is widely used in various industries, including scientific computing, computer vision, natural language processing, financial modeling, and oil and gas exploration, among

others. Specifically, CUDA plays a crucial role in radio astronomy due to the large amounts of data generated by radio telescopes and the need for real-time analysis. The parallel processing capabilities of CUDA enable the efficient processing of these large datasets, making it possible to reduce noise and analyze the data promptly and effectively [42].

2.6 Other Technologies Used

1. Programming Language: Python
   Used as the foundation for the codebase.

2. Python Library: Tensorflow
   Used to create the ML models and as CUDA built-in

3. High-performance computing
   Used to run the scripts to create the ML models

4. Visualization Tool: SAOImageDS9
   Used to visualize astronomical images

5. Astronomical Data simulator: Photon Simulator(PhoSim)
   Used to simulate the dataset used in training the ML models

Chapter 3

Implementation

3.1 Introduction

This chapter, showcases what was done to implement the solution, the steps taken to be able to visualize and simulated both the astronomical data and noise to create the dataset, how the models where implemented and the multiple steps taken to reduce the runtime from around 38 minutes to 4 minutes and half in the case of the multipurpose autoencoder, the results from the training and validation and the visual results of the predictions.

3.2 Astronomical Data

3.2.1 Visualizing FITS image format

Due to the format of the images, a special way of visualizing images is needed.
Although visualizing it as a normal image is possible using matplotlib, much visual information is lost.
The program used for this was SAOImageDS9, which is an astronomical imaging and data visualization application, created by the Harvard–Smithsonian Center for Astrophysics [43]. It has several scaling algorithms, including linear, log, power, sqrt, squared, asinh, sinh, histogram, min-max, and zscaled, with separated options for sixteen possibilities. These algorithms can be used to view multiple details that otherwise would be hidden from normal view. Even the color scales are useful due to the intensity of astronomical objects usually not being homogeneous. However, it is visually hard to see differences simply using a linear color scale like a grey scale.

As an example of how relevant this is, the Fig.3.1 displays the image of how the FITS file looks like by default, the default settings being, linear (the original pixel values are used) and min-max (defines the smallest and highest pixel value of the image to create the edges of the intensity scale), and to note it is similar to how it would be if it were opened

using matplotlib, just in a different angle, which looks almost like a blank image which just some spots.
Nevertheless, if it is scaled using squared (every pixel show is the squared value of its original) and zscaled (increases the contrast of the intensity scale) like in the Fig.3.2, the images suddenly looks like a completely different image due to the sheer amount of astronomical objects that can be seen on it, despite being the same image, that were just hidden from normal view.

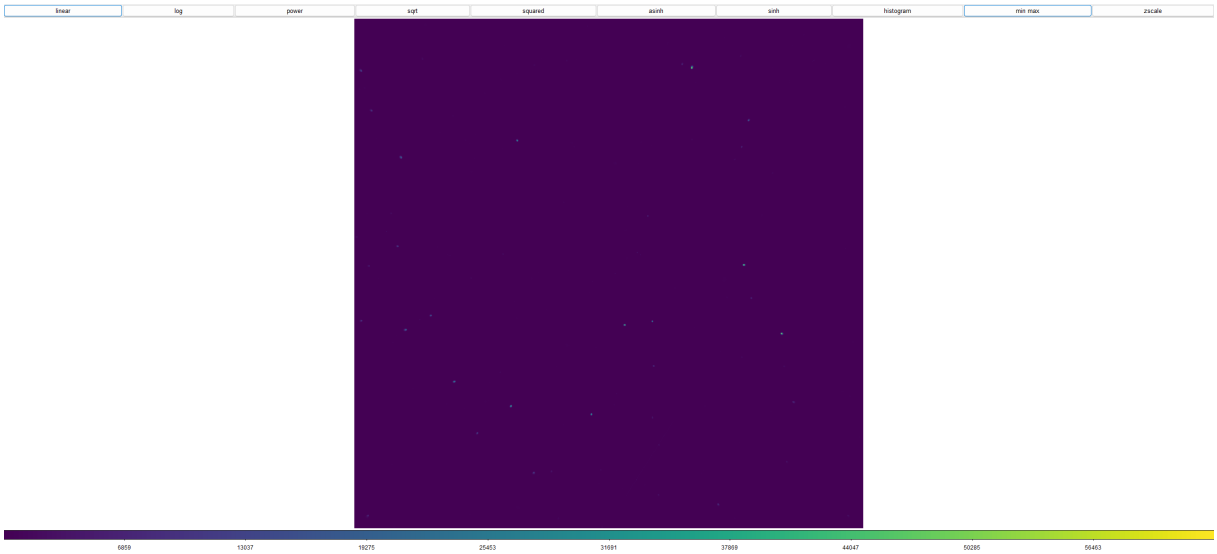

Figure 3.1: Simulation of an extensive catalog with a generic telescope, scale: linear - min max

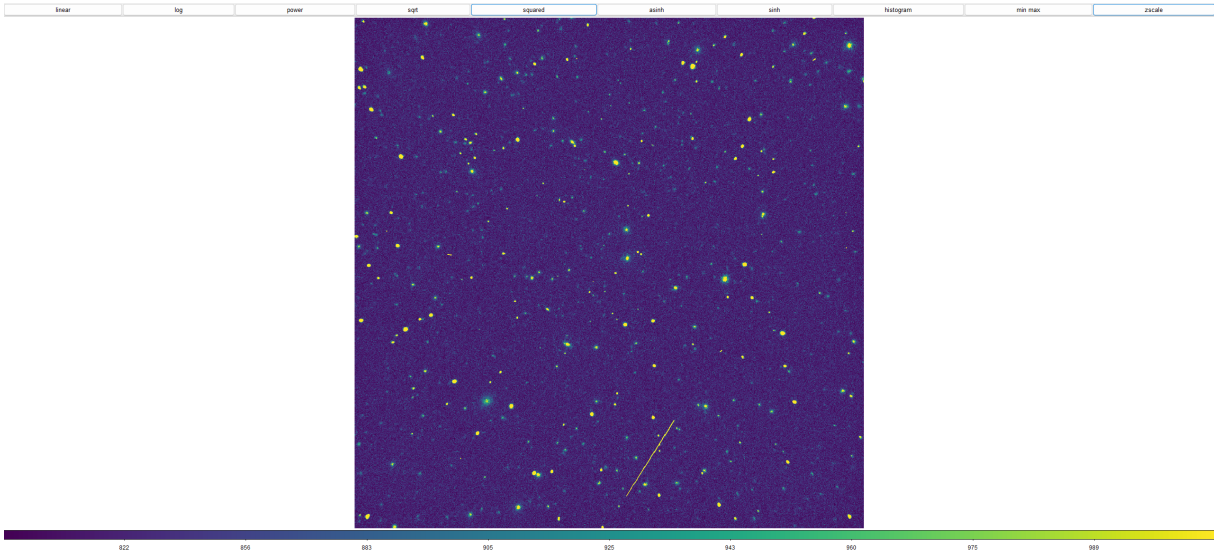

Figure 3.2: Simulation of an extensive catalog with a generic telescope, scale: squared - zscaled

3.2.2 Gather Astronomical Data

The original plan for the thesis was to use the data gathered for the SmartGlow project, but while by the end of the project the antennas were built to gather astronomical data, they by the of writing this thesis are still not fit to gather data. To the best of our knowledge, that is create the dataset for training three existing possibilities are available:

1. Finding an existing dataset.
2. Gathering real-life data.
3. Simulating data.

An existing dataset for training ML models was not found, so we moved on to attempting to gather real data from astronomical data archives.
There were multiple options found. Four of those options are the following:

1. NRAO Archive Interface [44]
2. SARAO Web Archive [45]
3. ALMA Science Archive [46]
4. SkyserverSDSS [47]

For example, on how to retrieve data from SkyserverSDSS, in this case, the Object ID: 1237662301903192106, Right ascension: 229.525575753922 and Declination: 42.7458537608544, as seen in the Fig.3.3.

Which contains a sizeable area around the specifically chosen showcased in multiple formats. The only option to get the actual data seems to be in FITS, other options are seemingly just static images of graphs and others, as seen in the Fig.3.4.
Opening the relevant FITS file, as seen in the Fig.3.5, shows the expected result, with the different orientation simply due to the differences in how SkyserverSDSS and DS9 orient the image.

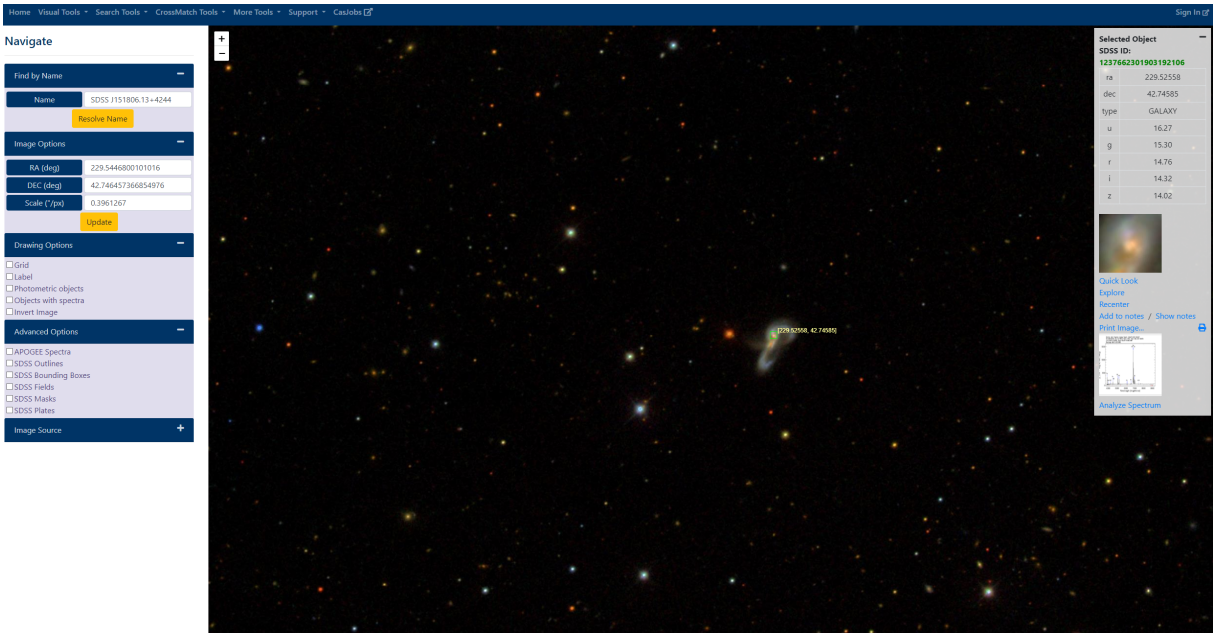

Figure 3.3: SkyserverSDSS selected object

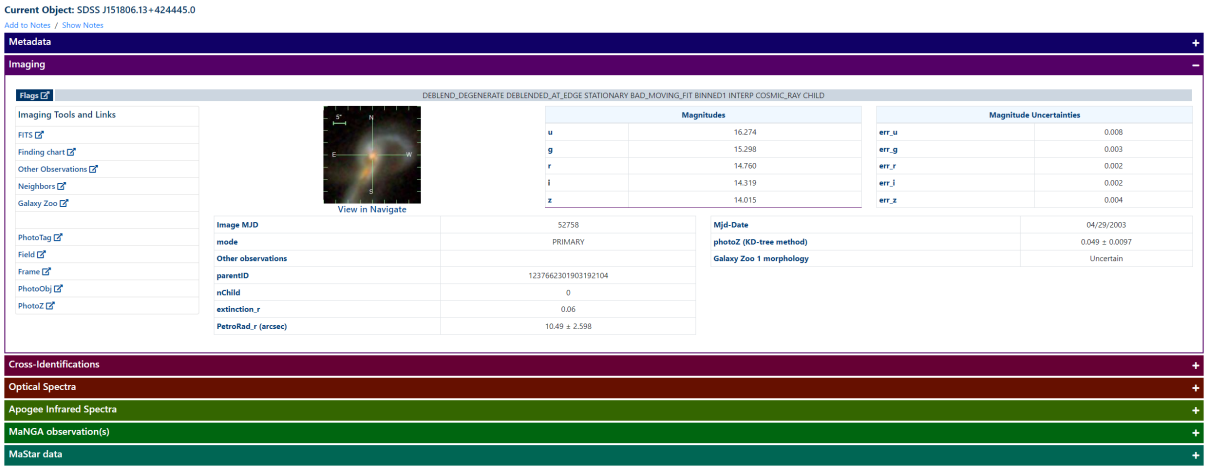

Figure 3.4: SkyserverSDSS selected object's data

Although possible, there are quite a few problems when gathering the dataset this way. The need to manually download each entry due to the seeming lack of an option to batch select, the state of the data not being necessarily equal, needing even more time to check each entry one by one, and also the problem of not knowing if the data is clean or noisy or even readable. For the current case, some states of the data include a lot of irrelevant data, which escalates with the sheer size of the entries, from 1 GB to 30 TB. At the same time, it is either impossible or extremely time-consuming to select only the necessary data on the download page due to most of the referred files being named generally and not giving an idea of what is inside each file.

Due to all these issues, which increase the complexity, the needed hardware, and time re-

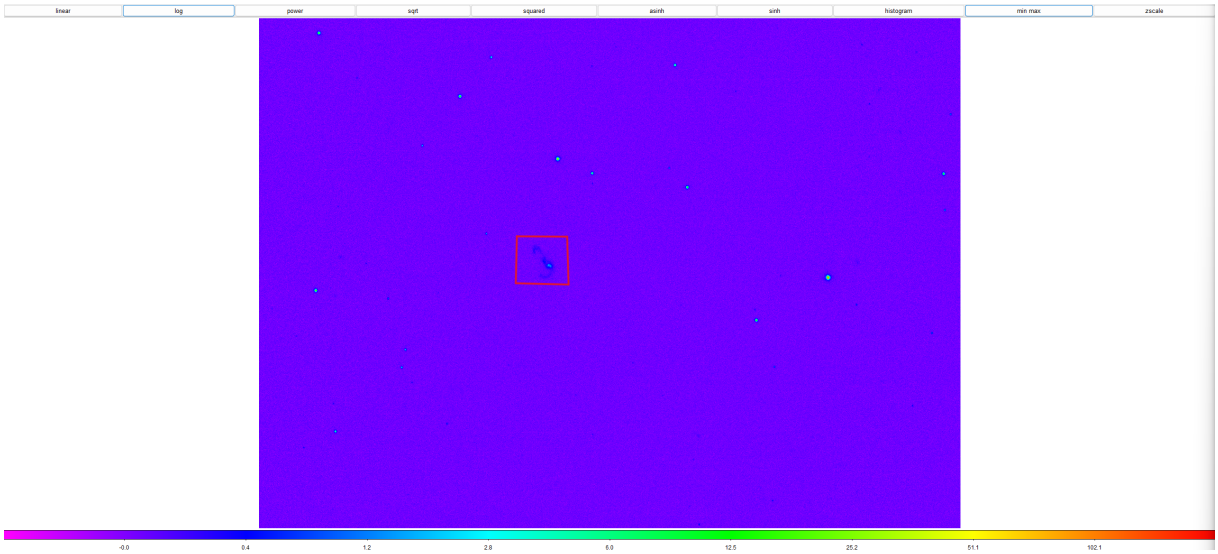

Figure 3.5: SkyserverSDSS selected object's data viewed in ds9 with scale log - min max

quirement for creating a large dataset for training the ML models, it was taken the option to create a realistic simulated dataset.

3.2.3 Simulating Astronomical Data

Common Astronomy Software Applications (CASA)

The Common Astronomy Software Applications (CASA) is a key tool in radio astronomy for processing data from telescopes like ALMA, VLA, VLBI, ATCA, GMRT, and others[48]. Its main function is to create pipelines for calibration of antennas and imaging. Additionally,the CASA software can simulate realistic observations from multiple telescopes and even define custom ones, much focus was put into CASA to create the simulated dataset. Following the CASA tutorial, a working simulation test was created, which can be found in the script in Listing I.1 with the result in the Fig.3.6 but after multiple attempts at trying to create a more realistic result and making it scalable to create a large number of different results. It was realized that there is a certain lack of scalability in the creation of the simulated data, due to the requiring to manually set parameters, for example like: shape, intensity, frequency, position, angle and much more, for each object in the simulation, and other concepts, which demands some expertise in these specific concepts, which at the end it was not very useful in the creation of the dataset to the purpose in mind [49, 50, 51].

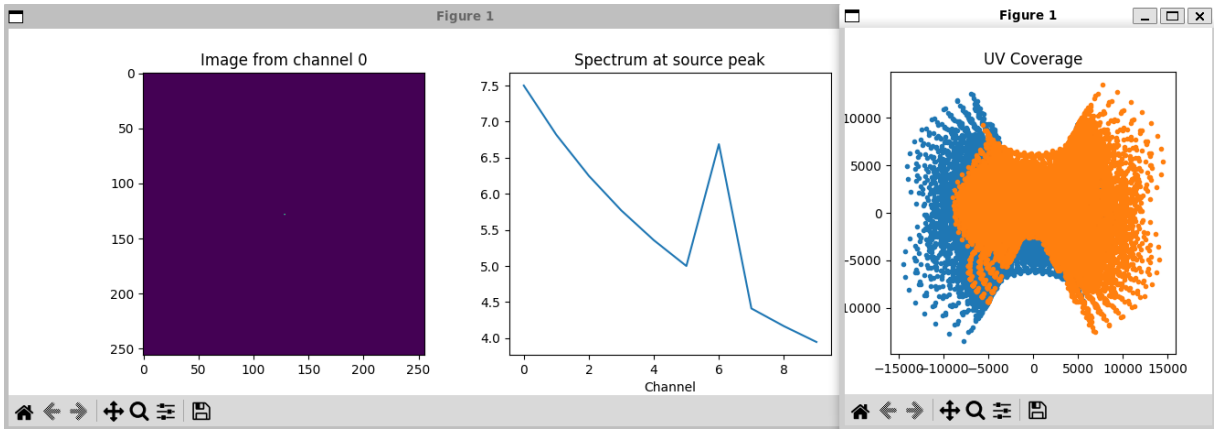

Figure 3.6: Result of CASA Simulation

Photon Simulator (PhoSim)

To create the simulated dataset, Phosim was used, following the logic from another dissertation about machine learning on deep space images which used it to compensate for the low amount of labeled images [52].
Photon Simulator (PhoSim) is a set of photon-efficient multithreadable Monte Carlo codes, which use repeated random sampling to simulate complex physical interactions [53], designed to simulate realistic astronomical images across various wavelengths, capable of simulating individual astrophysical objects in milliseconds and large fields of objects in a matter of seconds to minutes, depending on their size and depth. It achieves this by applying advanced numerical techniques to model the physical interactions of photons and electrons within the atmosphere, telescope, and camera.

It employs advanced numerical techniques, including advanced raytracing, diffraction, and quantum mechanical interactions, to simulate these physical processes. It also incorporates detailed models of the atmosphere, telescope, and camera, including hydrodynamic-based descriptions of the atmosphere, elasticity theory calculations of optics deformations, and electrostatic simulations of sensors for a realistic, randomized, and customizable simulation.

This software is used in the simulation of the results of a telescope during its design, construction, or commissioning phase, planning future observations, and generating realistic training sets for machine learning and advanced image processing algorithms [54, 55, 56].
PhoSim can be run using either shell commands or a GUI, At the moment of usage the GUI is not fully functional. One of the problems was that some functions used to define the simulation settings are either static with the default value or give an error when selected by default, while the bash command version is fully functional.
The Listing 3.1 is an example of a shell command used to create the simulations, where

it is defined the large catalog with long exposure, on which the configuration, complexity, and possible objects of the simulation are defined are specified, in this case it creates a large amount of data with randomized objects and high focus to simulate space, the output folder is localization, the amount of threads used it is important that the p×t is equal or less then the total amount of available CPU threads. In this case it was used 112×56=6272 threads in use during this simulation.

```
./phosim examples/large_catalog_long_exposure -o ../simulation/test/  -p 2 -t 56
```

Listing 3.1: PhoSim call

A total of 3 extensive simulations were done, resulting in ten FITS images of around 4000x4000 pixels each, that took 20 minutes to run. For example, the Fig.3.2 shows an extensive set of celestial objects generated with the Phosim simulator.

3.3 Developing and Testing Hardware

The simulations where run in an HPC owned by the Polytechnic University of Beja (IP-Beja), and the multiple scripts showcased in further sections. Specifically, it was used the SuperMicro "A+ Server 2124GQ-NART" [57], which of note as the following specifications:

- CPU: Dual 28 Core AMD EPYC™ 7453 Gen 3 Processor, 56 Threads, 2.75GHz, 64Mb L3 Cache, TDP 225W
- GPU: NVIDIA® HGX™ A100 4-GPU 80GB (HBM2e), Driver Version: 545.23.08 and CUDA Version: 12.3. Each of the 4 GPUs has 6912 CUDA Cores and 108 Streaming Multiprocessor, which results in 27648 threads that can run simultaneously being while capable of keeping residen(active but not necessarily running) 442368 threads
- RAM: 500GB during the simulation and preparation of the dataset, being upgraded to 2TB during the models training.
- OS: Ubuntu 22.04.3 LTS Server

The connection to the HPC was made via SSH, needing the configuration of a VPN for external access. For a few cases where it was needed to use a GUI, due to the lack of it since the HPC runs a server version of the OS, the following .bat file in Listing 3.2 was

used to redirect the GUI to a VcXsrv window to a local PC.

```
@echo off
for /f "skip=3" %%a in ('tasklist /fi "imagename eq vcxsrv.exe"') do if not
↪  "%%a" =="" goto :skip
start "VcXsrv" "C:\Program Files\VcXsrv\vcxsrv.exe"
:skip
set DISPLAY=localhost:0.0
ssh -Y *hostname*
```

Listing 3.2: Windows Bat to connect to the HPC with a GUI

The Physics Simulator (PhoSim) was installed and python package manager Conda was then used to establish a Virtual Machine (VM) workspace [58].
The workspace was equipped with all the pertinent libraries and dependencies, with TensorFlow being the most notable among them, the entire setup was prepared and optimized for GPU execution. [59]
Due to being unable to write the necessary variables without the admin permissions to run tensorflow in GPU on every new console, there is the need to run the following two commands, shown in Listing 3.3.

```
CUDNN_PATH=$(dirname $(python -c "import
↪  nvidia.cudnn;print(nvidia.cudnn.__file__)"))
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CONDA_PREFIX/lib/:$CUDNN_PATH/lib
```

Listing 3.3: Commands to allow GPU use in tensorflow

For a simple and straightforward workflow, JetBrains Gateway was used to be able to run PyCharm IDE in the HPC, while being able to work on it directly from a personal local PC. [60]

3.4 Preparation of the Dataset

3.4.1 Splitting the simulations

Due to the huge size of the astronomical image (4000x4000), and the need to later deal with the most likely problem of different-sized images, the generated images were split using multiprocessing into multiple 256x256 FITS images, as seen in Fig.3.7.

With the objective of faster training and deciding on a standard size for the model. Due to a later step, some randomization is added while adding noise to each image, allowing the use of the step between the new images be 30 pixels instead of the expected 256 to increase the amount of data from each simulated large image, since while it creates very similar images step by step the noisy images will still have a decent amount of difference between them, which otherwise would create a overfitting problem, while following a naming convention of "clean_data_coordinates.fits". This can be seen in the script in Listing I.2.

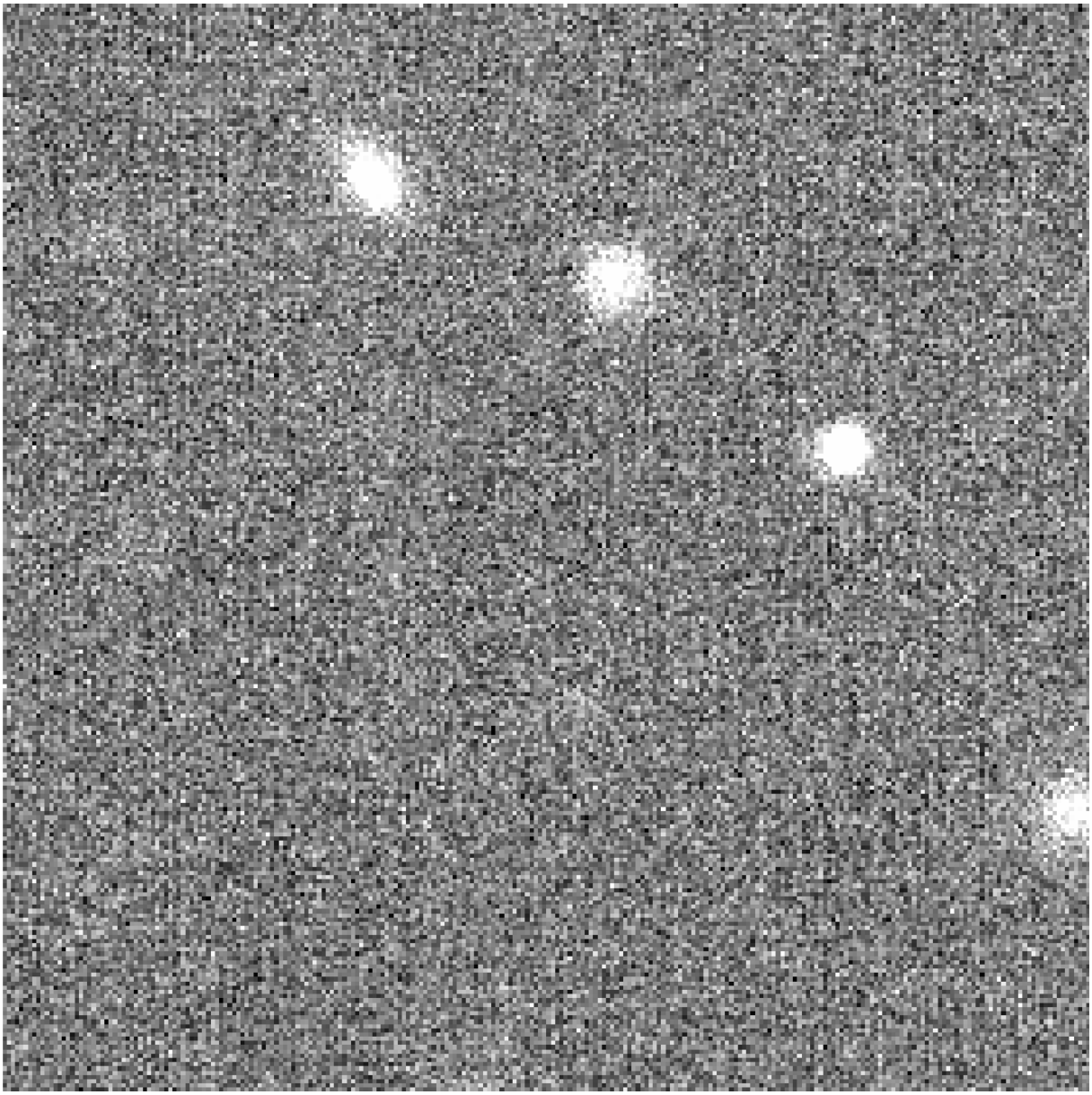

Figure 3.7: Children Image, scale: linear - zscaled

3.4.2 Adding Noise

Since all the data in the current state is clean, each of the smaller images that were created in the step before will be used to create new FITS images with added noise, while keeping the use of multiprocessing to speed up the process.
The decision was to only use the 3 of the most common types of noise with different strengths. Those being Gaussian noise as seen in the script in Listing I.3 and Fig.3.8, Impulse noise as seen in the script in Listing I.4 and Fig.3.9 and Poisson noise as seen in the script in Listing I.5 and Fig.3.10.
In general, skimage random_noise was used to add noise to the images except for Poisson, which was added simply using numpy. A naming convention based on the previous one was also added, in this case, a noisy folder being "noise_type of noise_intensity of noise_data_coordinates.fits" again except for Poisson, which was not added intensity.

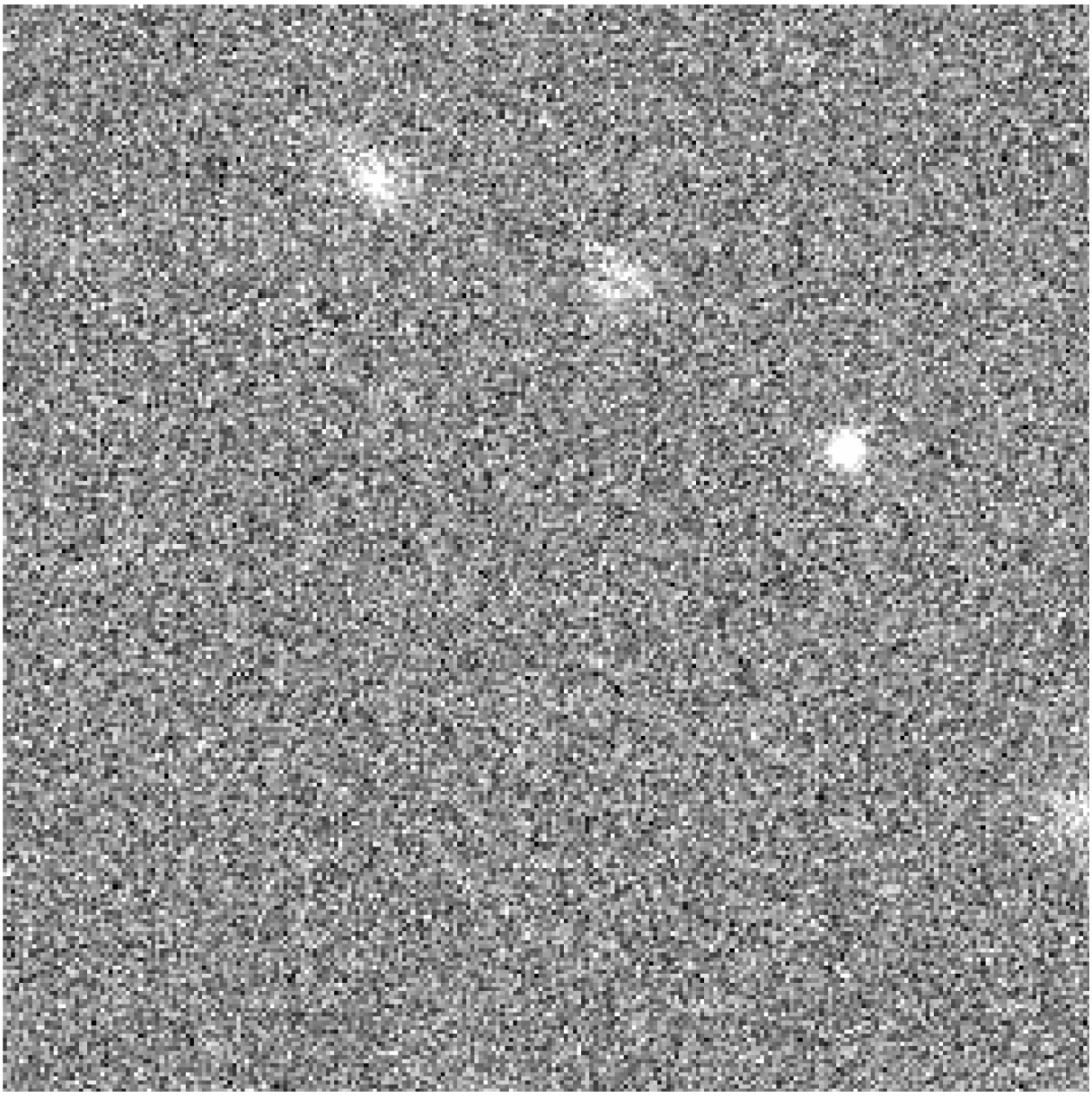

Figure 3.8: Image with Gaussian noise, scale: linear - zscaled

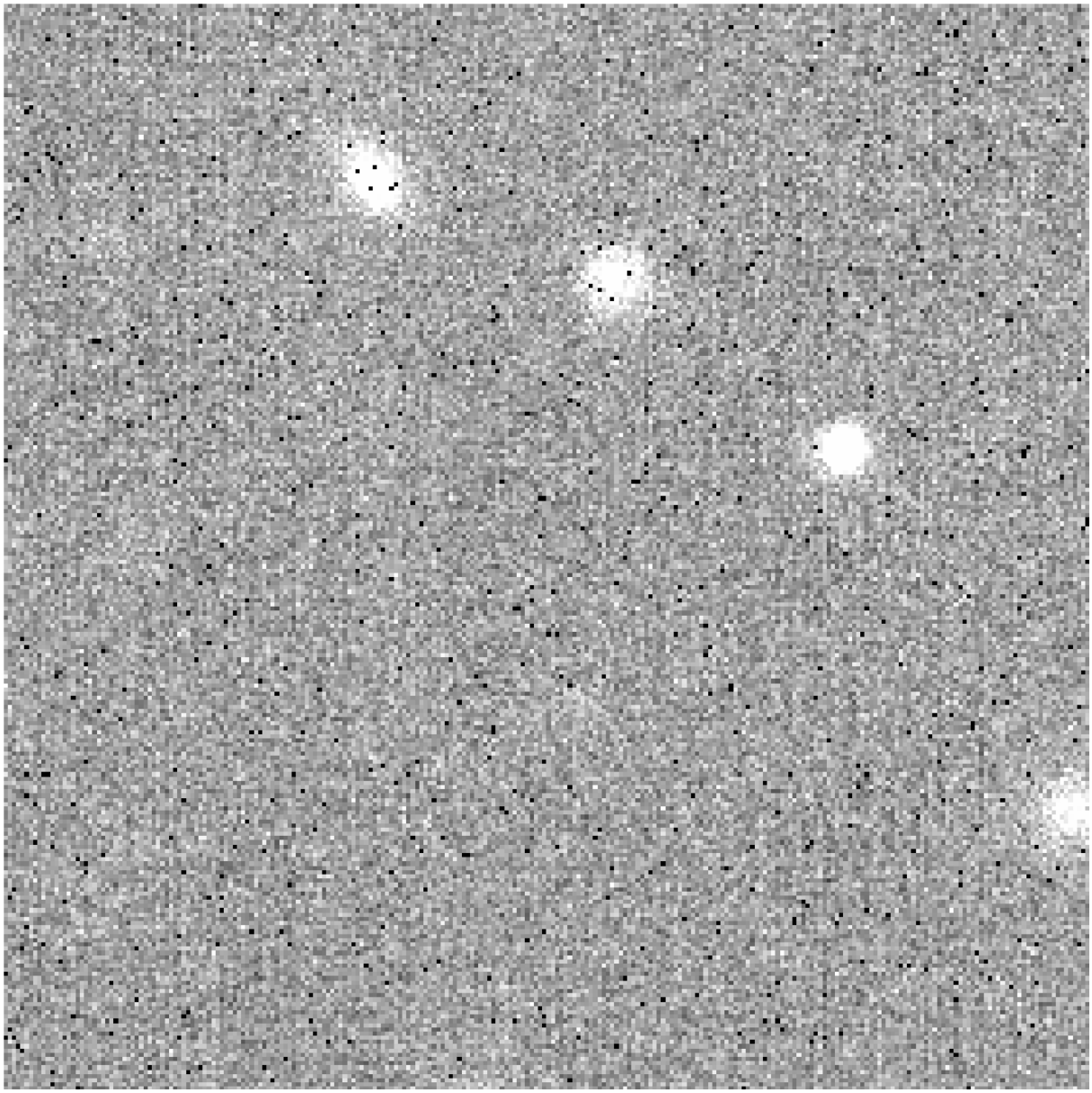

Figure 3.9: Image with Impulse noise, scale: linear - zscaled

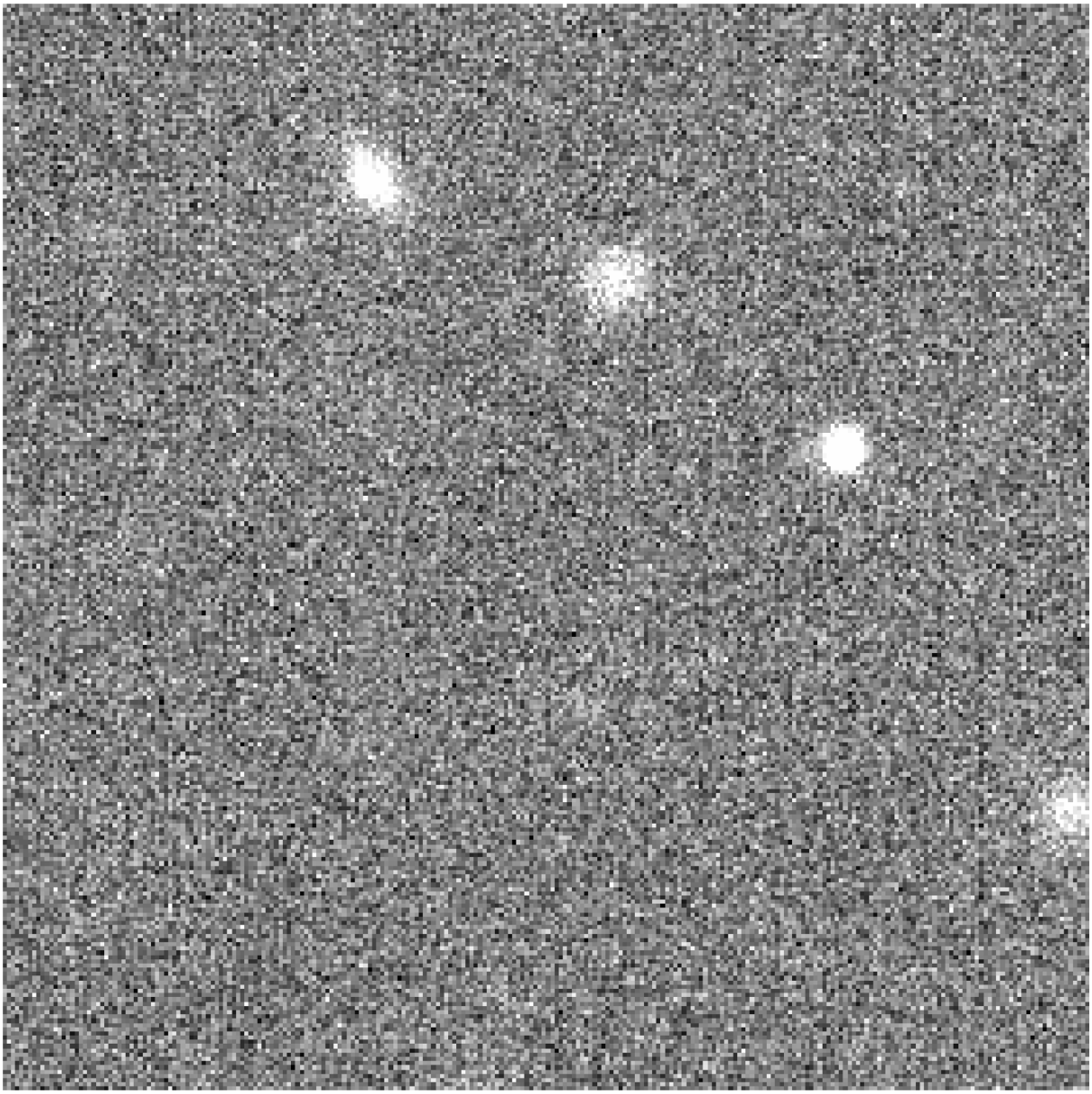

Figure 3.10: Image with Poisson noise, scale: linear - zscaled

3.4.3 Dataset Organization

The dataset is organized in order to facilitate the analysis, which can also be seen in the Fig. 3.11:

1. "root" folder named "dataset".
2. N amount of folders based on the number of total Phosim simulations keeping their respective name.
3. N amount of folders based on how many images are inside the Phosim Simulation, since each Phomsim simulation might have multiple images instead of being just one image per simulation, keeping their respective name.
4. All the clean images are labeled with the file name "clean_data_X_Y.fits", x and y being their start coordinate relative to their parent image. There is also a folder named "noise".
5. Inside the "noise" folder, there will be folders for all types of noise in the dataset, which currently are "Gaussian", "Impulse", and "Poisson".
6. There are two possibilities depending on the type of noise. If multiple versions with different noise intensities are added for the same noise. Then in that case, there will be N amount of folders respective to the number of different intensities named after their respective value, otherwise the noisy images will be located in this folder the name "noise_Type of noise_data_X_Y.fits", x and y being their start coordinate relative to their parent image.
7. Inside each noises intensity folders, there will be all noisy images with it's respective intensity with the following name "noise_Type of noise_Strength of the noise_data_X_Y.fits", x and y being their start coordinate relative to their parent image.

The dataset, which is saved in the HPC server, has 868056 files and a total size of 219 GB. A smaller prototype dataset, which contains data used to test the models, is saved a local PC.

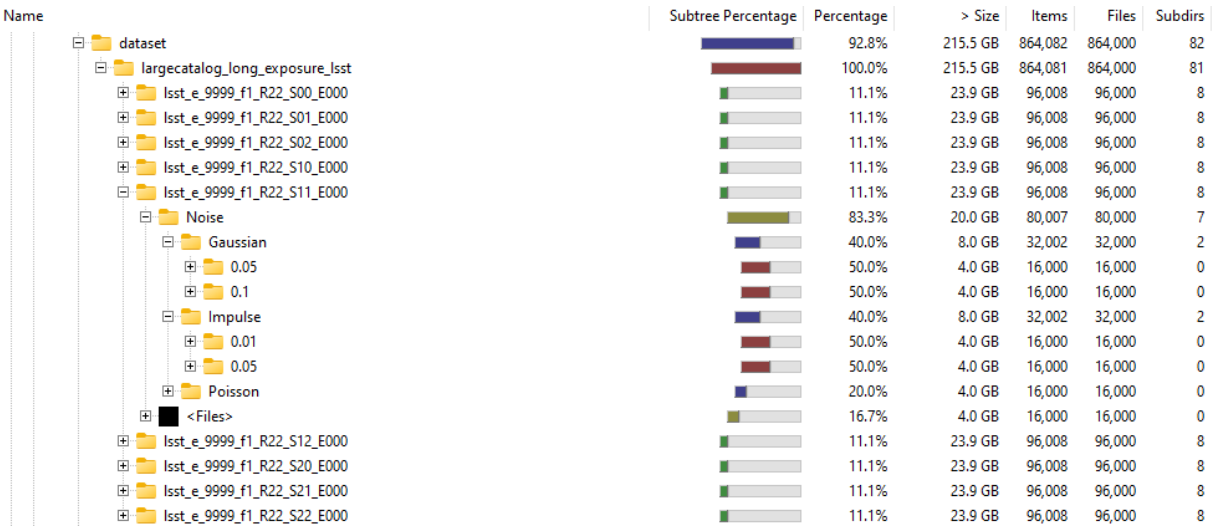

Figure 3.11: Visualization of how part of the dataset looks visually

3.5 Machine Learning Models

To compare two possibilities, we will use ML learning to reduce noise from the images or detect the type of noise and then, using another existing algorithm or a model training to clean a specific noise to remove the relevant noise.
Five models were created with tensorflow, with 4 of them being the same type:

3.5.1 Autoencoder

A autoencoder is a type of neural network where the output layer has the exact dimensions as the input layer. The Autoencoder learns to compress the data while minimizing the reconstruction error.
It is composed of three components:
An encoding portion that compresses the data, a component that handles the compressed data, and a decoder portion that reconstructs the output. Autoencoders are used for unsupervised learning purposes to discover underlying correlations among data and represent data in a smaller dimension. Most types of autoencoders are used for feature extraction task, for example: data compression, image denoising, anomaly detection and facial recognition. In this case, there are 4 autoencoders, one is for multipurpose and the other three are specialized for a certain type of noise, which receive The FITS files will be compressed and reconstructed without noise [61, 62, 63].

3.5.2 Convolutional Neural Network (CNN)

A CNN is an machine learning algorithm commonly used to help computers understand and interpret images [64]. It does this by using a special mathematical operation called convolution, which helps the computer process the pixel data in the image. Being composed

of various layers, which, in this case, will return four possible labels from an input FITS file.

3.6 Code Implementation

The explanation of the scripts and the logic used to create the models will be divided into a few crucial blocks, which will, due to a considerable similarity between them, be identical, with the parts that branch off between both being placed in a subsection of a block with the model name. So unless specified, the code or logic is identical or with negligible differences between both, the complete scripts are located in Listing I.6 and Listing I.7.

3.6.1 Environment Settings and Imports

The Listing 3.4 shows the imports and the environment variables used in to create the models. The models are created and trained using TensorFlow and its Keras API.

TensorFlow is an open-source machine-learning library developed by Google. It provides a comprehensive ecosystem of tools, libraries, and community resources, allowing researchers and developers to build and deploy machine learning models [65].

Keras is a high-level API for TensorFlow that provides a more user-friendly interface for building and training deep learning models. It abstracts many of the lower-level details of TensorFlow, making it easier to create complex neural networks [65].

Multiple Tensorflow related environment variables were defined in the Listing 3.4:

- *TF_XLA_FLAGS*:
  Guarantees that the model does not use the Accelerated Linear Algebra (XLA) library. Usually, its use is defined by a "jit_compile" Boolean input variable in the Tensorflow functions, which is an open-source compiler for machine learning that optimizes the models from various popular ML frameworks like Tensorflow, Pytorch, and JAX for high-performances execution across GPUs, CPUs, and ML accelerators. But either due to some of the Tensorflow functions being used not being fully implemented yet, or the custom functions not being efficient while using XLA, or related to the use of the HPC, it's use increased the time taken to train the model by a decent amount. It was first noted early in the creation of the model that when removing all uses of it decreased the time taken per epoch from 30 min to 24 min.
  At the present state and after all the optimizations using XLA doubles the time taken, it seems to be related to the creation of large overheads while barely reducing the runtime [66].

- TF_FORCE_GPU_ALLOW_GROWTH: The model was trained in an HPC with 4 GPUs, each with 81920 MiB of VRAM, which the models barely uses 15% of the total, while training in the current settings. But Tensorflow by default, holds onto all the VRAM even without it being used is a waste of resources. This variable is set to force Tensorflow to use only the amount of memory it needs, while allowing it to use more memory in the case it is needed.

- TF_GPU_THREAD_MODE and TF_GPU_THREAD_COUNT: These variables are related. Thread mode ensures that the GPU has private CPU threads only working to launch its kernels to reduce kernel launching overheads, which by default allocates two threads to each GPU. Thread count sets the number of threads used by the GPU, which is set to 10, which in total means that 40 of the 112 threads are private. This is around the sweet spot. Even though it still has a decent amount of kernel-related overhead, it is reduced significantly, but increasing the number of threads does indeed reduce the amount of overhead even further and starts instead creating overhead in other CPU-related processes [67].

- The import of *tf_fits.image*, also known as *"tensorflow_fits"*, is a Python script capable of opening FIT files inside a Tensorflow function while the usual option being astropy which is not capable of such a feature without creating a significant overhead in the Tensorflow function, which will be looked more deeply in a future section [68].

```
1  import os
2
3  os.environ['TF_XLA_FLAGS'] = '--tf_xla_enable_xla_devices=false'
4  os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5  os.environ['TF_GPU_THREAD_MODE'] = 'gpu_private'
6  os.environ['TF_GPU_THREAD_COUNT'] = '10'
7
8  import tensorflow as tf
9  import numpy as np
10 from tf_fits.image import image_decode_fits
11 from tensorflow.keras import layers
```

Listing 3.4: Imports and Environment variables

3.6.2 Data Pipeline

The data pipeline creates an array of strings with the paths to the dataset files since there are too many files to load directly in one go. It takes roughly 8 minutes to fully load the

dataset, compared to 8 seconds of simply getting the paths and then loading them when necessary.
To be more specific, a tuple is created with the input and its expected output, as seen in the script in Listing3.5. This is the multipurpose autoencoder version, adding constriction to only get the files with a specific noise is the only change between the multipurpose autoencoder version and its specific noise counterparts, it also differs from the CNN version due to the use of the function clean path, while the CNN version uses a function *get_label* which will get the noise label from the image name and add it to the expected output part of the tuple.

```
1  def get_files(path):
2      fits_files = []
3      for root, dirs, files in os.walk(path):
4          for file in files:
5              if file.endswith(".fits"):
6                  fits_files.append(get_clean_path(os.path.join(root, file)))
7      return fits_files
8
9
10 path = "dataset"
11 file_list = get_files(path)
12 file_list = np.array(file_list)
```

Listing 3.5: Get Paths for Autoenconder

Initially, a custom Keras sequence was used for the sake of dealing with the FITS files as they are not in a format directly compatible with Tensorflow, which would turn them into a numpy array, normalize it, and get its expected results, either being its original image in the autoencoder case, or its label in the CNN case, as seen in the script in Listing 3.6.

```
1  class FITSDataGenerator(tf.keras.utils.Sequence):
2      def __init__(self, file_list, batch_size):
3          self.file_list = file_list
4          self.batch_size = batch_size
5          self.indexes = np.arange(len(self.file_list))
6          self.on_epoch_end()
7
8      def __len__(self):
9          return int(np.ceil(len(self.file_list) / float(self.batch_size)))
10
```

```
11      def __getitem__(self, idx):
12          indexes = self.indexes[idx * self.batch_size:(idx + 1) *
            ↪  self.batch_size]
13          batch_files = [self.file_list[k] for k in indexes]
14          batch_data = []
15          result_batch_data = []
16          for file in batch_files:
17              data = fits.getdata(file, ext=0)
18              batch_data.append(data)
19
20              result_data = fits.getdata(getCleanPath(file), ext=0)
21              result_batch_data.append(result_data)
22
23          batch_data = zero_one_scaler(batch_data)
24          result_batch_data = zero_one_scaler(result_batch_data)
25
26          return batch_data, result_batch_data
27
28      def on_epoch_end(self):
29          np.random.shuffle(self.indexes)
30
31  train_gen = FITSDataGenerator(file_list, batch_size=100)
```

Listing 3.6: Custom Keras Sequences for Autoenconder

Nevertheless, due to apparent incompatibility with these custom sequences and multi GPU, creating a large overhead which increased the runtime from 30 minutes to 38 minutes and even without using multi GPU, as seen the in the Fig. 3.12, 33.9% of each step in training was spent waiting for the input.
Due to these problems, the custom Keras sequence was scrapped to use a tensorflow data pipeline, which is generally faster and compatible with the use of multi-GPU [69].

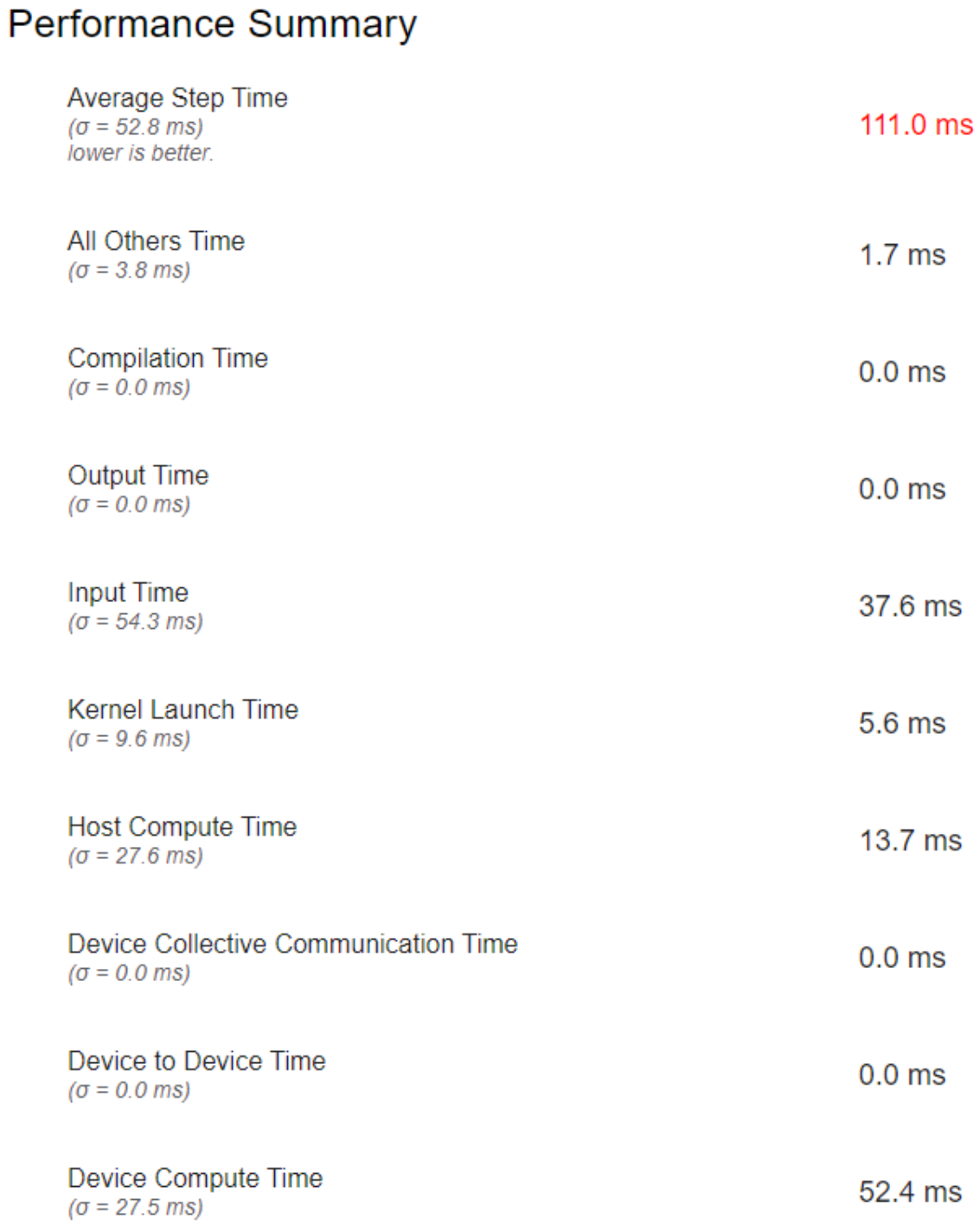

Figure 3.12: Profile of Autoencoder using custom keras sequence

Resulting in the current state of the data pipeline in the Listing 3.7, this being again the autoencoder version. The CNN version is identical, but instead of dealing with two FITS files, it is just one and an integer (the label).

One of the biggest problems during this transition is that the original lib, astropy [70], used to read the FITS files into a readable array is not compatible with Tensorflow function, meaning that it will partially break the conversion to it (in the code, it refers to the *@tf.function*), which created a large amount of overhead due to the code having to jump from Tensorflow function to Python and back to Tensorflow function.
This conversion is crucial because it increases the performance and reduces the overhead by a large amount while training the model using GPU. So a Python library was found that allows to open the FITS files in a compatible way with the *"tensorflow_fits"*, described in more detail in the section 3.6.1.

```
1
2  @tf.function(reduce_retracing=True)
3  def open_files_batch(path_batch):
4      input_data_batch = tf.map_fn(lambda x: tf.io.read_file(x[0]), path_batch,
       ↪  fn_output_signature=tf.string)
5      output_data_batch = tf.map_fn(lambda x: tf.io.read_file(x[1]), path_batch,
       ↪  fn_output_signature=tf.string)
6      return input_data_batch, output_data_batch
7
8
9  @tf.function(reduce_retracing=True)
10 def read_files_batch(input_binary_strings_batch, output_binary_strings_batch):
11     input_data_batch = tf.map_fn(lambda x: image_decode_fits(x, 0),
       ↪  input_binary_strings_batch,
12                                  fn_output_signature=tf.float32)
13     output_data_batch = tf.map_fn(lambda x: image_decode_fits(x, 0),
       ↪  output_binary_strings_batch,
14                                   fn_output_signature=tf.float32)
15     return input_data_batch, output_data_batch
16
17
18 @tf.function(reduce_retracing=True)
19 def processing_map(input_images, output_images):
20     input_data = tf.expand_dims(tf.clip_by_value(tf.divide(input_images,
       ↪  100000.0), 0., 1.), axis=-1)
21     output_data = tf.expand_dims(tf.clip_by_value(tf.divide(output_images,
       ↪  100000.0), 0., 1.), axis=-1)
22     return input_data, output_data
23
24
25 dataset = tf.data.Dataset.from_tensor_slices(file_list) \
26     .shuffle(buffer_size=100000, seed=1213, reshuffle_each_iteration=True) \
27     .batch(128*4) \
```

```
28      .map(open_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
29      .map(read_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
30      .map(processing_map, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
31      .cache() \
32      .prefetch(tf.data.AUTOTUNE)
33
```

Listing 3.7: Tensorflow pipeline for Autoencoder

Another big challenge while dealing with this kind of data was to normalize it because of its lack of an exactly fixed scale. In the end, the decision was take to normalizing by using min-max by scaling images in the interval 0 - 10000 to 0 - 1, and clipping, turns values outside the defined range into its closes, the values to guarantee all values are within the 0 - 1 interval. This value was decided by checking every individual pixel in the current dataset. The value chosen was the range of the rounded clean pixels, which can be seen in the following results:
Min-Max Pixel Values:

- Total: Max value: 147776.65625, Min value: -53455.1328125
- Clean: Max value: 100000.0, Min value: 681.0
- Gaussian: Max value: 147776.65625, Min value: -53455.1328125
- Impulse: Max value: 100000.0, Min value: 0.0
- Poisson: Max value: 102988.0, Min value: 1435.0

The remaining parts of the data pipeline are related to performance:

- *tf.data.AUTOTUNE*: it decides the number of CPU threads that will be used automatically to guarantee that the exact amount of resources needed are being spent. This increases performance due to the data pipeline having enough threads to do its processes while not interfering with other CPU-intensive processes, which could result in overheads due to the need to wait for threads to be free.
- *deterministic=False*: This allows elements of the dataset to be processed out of order, to sacrificing determinism, consistent result order, for performance. Since there is no specific order in this dataset, it is not worth keeping its order intact.

- *cache* and *prefetch*: The mixture of these two functions in that order allows that after the first epoch, the data pipeline will prefetch the entire cached dataset, resulting in the wait for input going to almost 0 seconds from 2 minutes, as shown in the Fig.3.13, which shows the current data pipeline being ran by itself.
- batch size: 128 * 4, due to the use of Multi-GPU, which results in each GPU training in 128 batches, since the HPC being used has four GPU cards.

Figure 3.13: Performance of Data pipeline

Fig.3.14 showcases the performance summary with just the change from custom Keras sequences to the current data pipeline, having an 80.8511% (37.6 ms to 7.2 ms) increase in performance related to input time per step.

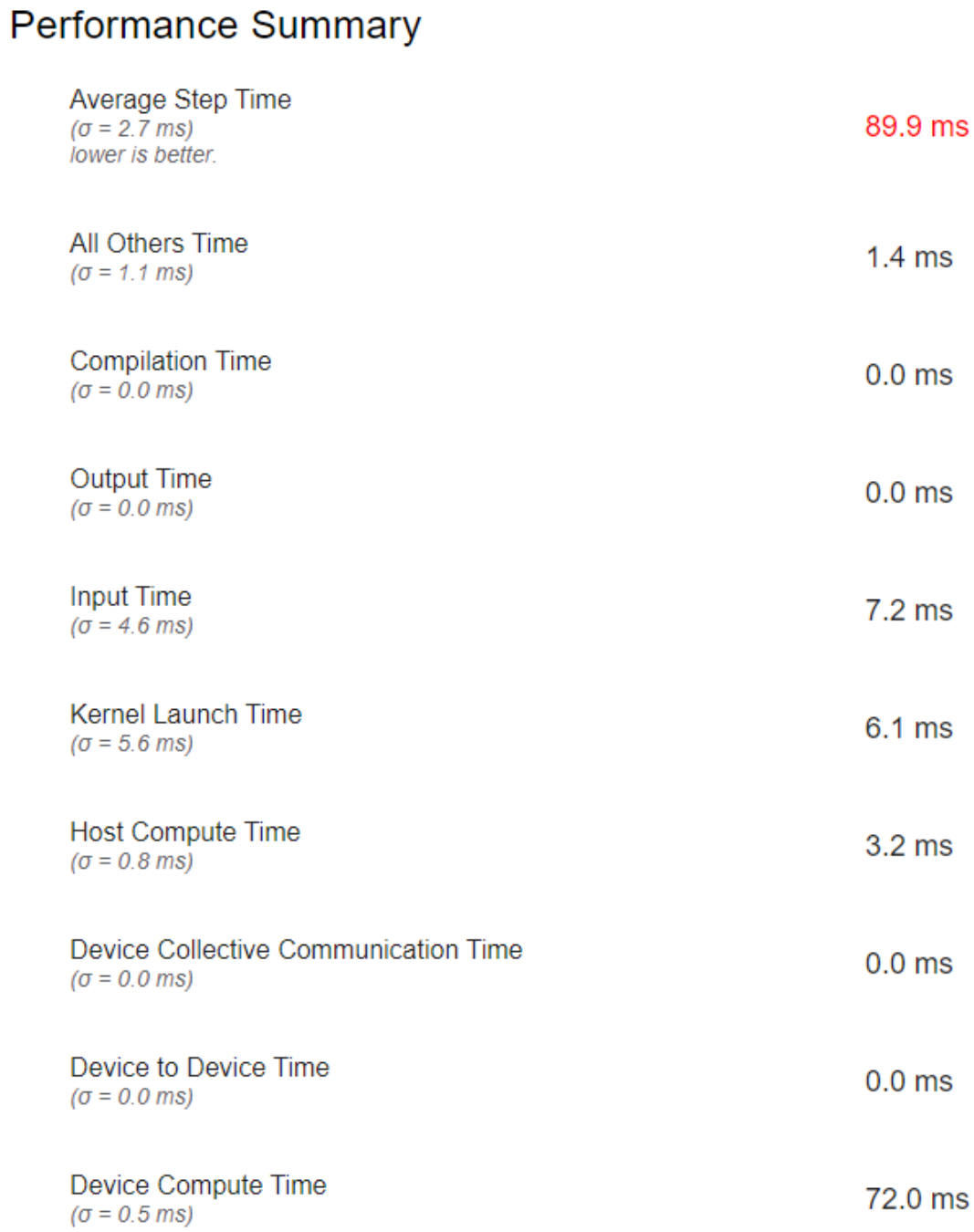

Figure 3.14: Profile of Autoencoder using tensorflow dataset

Autoencoder

The Autoencoder models are expecting to receive a tuple as input: the images (input, expected output). Which in this case are all the FITS file paths, or in the specific noise autoencoders only the FITS file paths of their noise, and the path of their equivalent original clean versions of the images, as seen in function in Listing 3.8.

```python
def get_clean_path(path):
    file_name = os.path.basename(path)
    file_name_split = file_name.split("_")
    if file_name_split[0] == "clean":
        return path, path
    elif file_name_split[1] == "poisson":
        pathSplit = path.split('/')
        cleanPath = ""
        for i in range(len(pathSplit) - 3):
            cleanPath += pathSplit[i] + "/"

        cleanPath += 'clean_data_' + file_name_split[-2] + '_' +
        ↪ file_name_split[-1]
        return path, cleanPath
    else:
        pathSplit = path.split('/')
        # cleanPath = pathSplit[0] + '/' + pathSplit[1] + '/' + pathSplit[2] +
        ↪ '/' + 'clean_data_' + file_name_split[-2] + '_' +
        ↪ file_name_split[-1]
        cleanPath = ""
        for i in range(len(pathSplit) - 4):
            cleanPath += pathSplit[i] + "/"

        cleanPath += 'clean_data_' + file_name_split[-2] + '_' +
        ↪ file_name_split[-1]
        return path, cleanPath

```

Listing 3.8: Get Tuple Autoencoder

CNN

Just like the Autoencoder, the CNN is expecting to receive a tuple for input, being (input, expected output), which in this case are all the FITS file paths and their noise label, which is saved in the file name, as seen in function in Listing 3.9.

```python
def getLabel(path):  # 0 = clean, 1 = Gaussian, 2 = Impulse, 3 = Poisson
    file_name = os.path.basename(path)
    file_name_split = file_name.split("_")
    if file_name_split[0] == "clean":
        return path, 0
```

```
6       elif file_name_split[1] == "gaussian":
7           return path, 1
8       elif file_name_split[1] == "impulse":
9           return path, 2
10      else:
11          return path, 3
12
```

Listing 3.9: Get Tuple CNN

3.6.3 Models Performance

Multiple experiences were done to increase the performance/speed of training the model, and multiple failed attempts were made, some of which attempted to compress the dataset into a single table/Python dictionary for faster readability.

Both resulted in a mix of being much slower than the current way, like loading the entire dataset in a file wastes much memory and is more time consuming. At the same time, the Python dictionaries worked faster when using the custom Keras sequences, but since it was not directly compatible with the Tensorflow dataset, and due to its already high performance/speed, there was no need to try to force it.

Another change that did not result in an increased performance was defining a max queue size in the *fit* function to train the model since it simply chooses how many batches are preloaded into memory. In this case, it loses any difference in runtime due to the use of *prefetch* in the tensorflow dataset, which does the same but only fetches the data that can be used at the time, as can be seen in the results in the Listing 3.10, where the first queue size = 1, the second is max queue size = 100, and the third is max queue size = 1696 which is all the batches queued in one go. There is barely any difference between them, and the differences are more likely being related to resource management in the HPC than due to the change in queue size.

```
1  (tf) pmmf@rolf:~$ python ./MastersCode/Tests/testWeight/test14.py
2  Get Dataset Paths: Took 3.8996307849884033 seconds
3  Autoencoder
4  Epoch 1/1
5  1696/1696 [==============================] - 272s 123ms/step - loss: 0.1737
6  (tf) pmmf@rolf:~$ python ./MastersCode/Tests/testWeight/test14.py
7  Get Dataset Paths: Took 3.3734612464904785 seconds
8  Autoencoder
9  Epoch 1/1
10 1696/1696 [==============================] - 276s 127ms/step - loss: 0.1756
11 (tf) pmmf@rolf:~$ python ./MastersCode/Tests/testWeight/test14.py
```

```
Get Dataset Paths: Took 3.5350682735443115 seconds
Autoencoder
Epoch 1/1
1696/1696 [==============================] - 275s 128ms/step - loss: 0.1775
```

Listing 3.10: Max Queue Size Test

In the current state, six decisions decreased runtime, three for a small amount, while the others had a greater impact on runtime reduction. Most of these were discussed in more depth in the previous sections:

- Removing XLA: While removing it from the single function that used it by default decreased runtime minimally, it is worth noting that using it increased runtime by up to 8 minutes.
- Use of Tensorflow environmental variables: It only saved a few milliseconds per training step, but that accumulates over the iterations to save up a few minutes.
- Use of early stopping callback: It allows to stop the model training once the defined metric is accomplished. In this case, the validation loss has not decreased in the span of three epochs by 0.0001, allowing to guarantee that the model is not being iterated more than necessary, saving a lot of runtime.
- Using multiprocessing in training: Multiprocessing with 20 workers (threads) saved around 3 minutes. In this case, 20 is the sweet spot, as increasing it beyond this value takes more time due to the overhead created by other processes having to wait for the threads to be free [71].
- Use of the Tensorflow data pipeline: The move from a custom Keras sequence to the Tensorflow dataset pipeline, which uses *tf.functions*, was the change that had more impact in the considerable runtime decrease of all of the changes, going from each epoch taking around 1840 seconds (30.6 minutes) to only 564 seconds (9.4 minutes).
- Use of *MirroredStrategy* [72]: this Tensorflow function allows the model to be trained using multi-GPU, which is very important while using an HPC with multiple GPU cards. This increased runtime while using the custom Keras sequence. With the Tensorflow dataset pipeline, it reduced the runtime by half. It went from 564 seconds (9.4 minutes) to 272 seconds (4.5 minutes). This use can be seen in the Listing 3.11.

```

strategy = tf.distribute.MirroredStrategy()
print("Autoencoder")
```

```
with strategy.scope():
    autoencoder = Autoencoder()
    autoencoder.compile(optimizer=tf.keras.optimizers.Adam(jit_compile=False),
    ↪  loss=tf.keras.losses.BinaryCrossentropy())
autoencoder.fit(dataset,
                epochs=2,
                verbose=1, use_multiprocessing=True, workers=20,
                ↪  max_queue_size=1)
```

Listing 3.11: Multi-GPU training

3.6.4 Autoencoder Model

The current autoencoder model has its basis on an autoencoder designed to clean Gaussian noise from 26x26 images of written numbers, created by Francois Chollet, the creator of the Keras lib [73].
Due to the large gap in the main objective, many modifications were made to the refered model. Currently, the autoencoder model comprises the following layers in the encoder in the Fig.3.15 and decoder in the Fig.3.16.

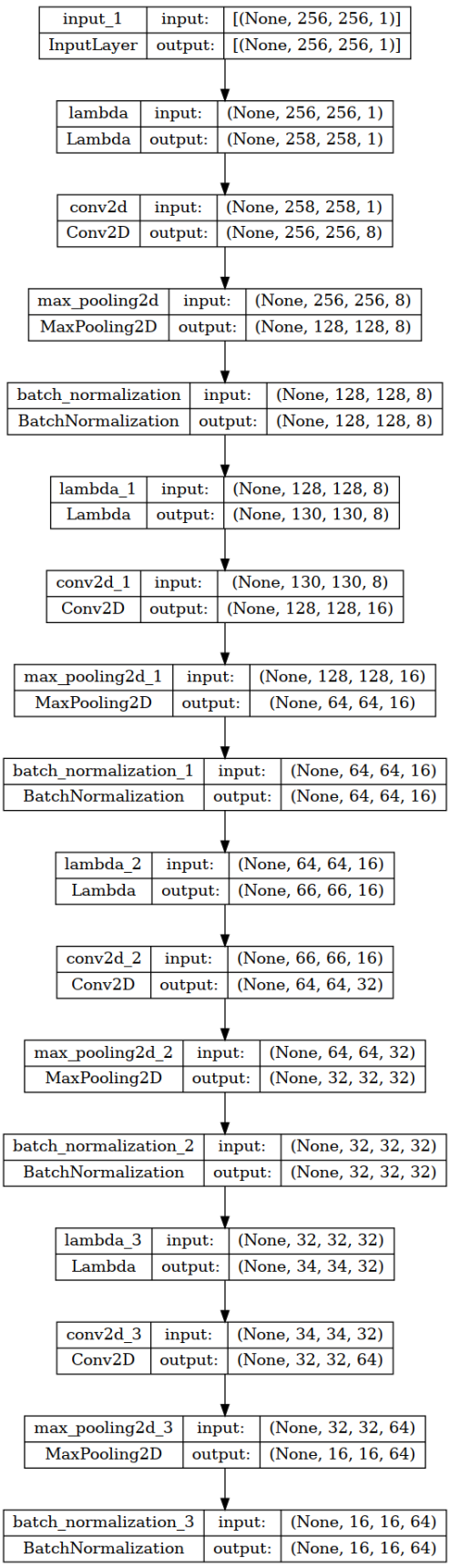

Figure 3.15: Autoencoder Encoder Structure

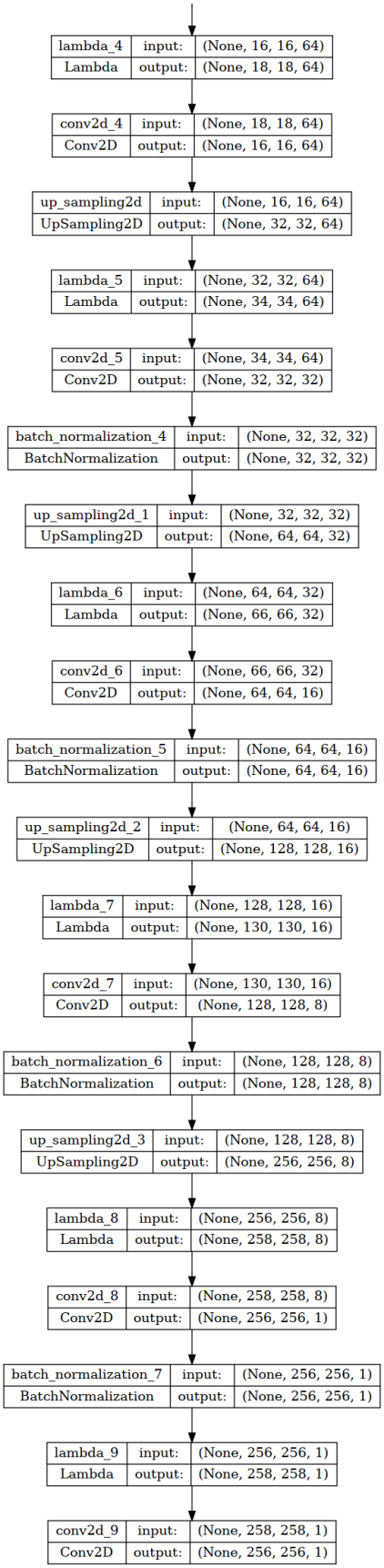

Figure 3.16: Autoencoder Decoder Structure

In the following the most note-worthy modifications in the model are described:

1. Increase in *conv2d* layers due to the increase in image size (26x26 to 256x256) which allowed for more complex patterns. The base model counted on written numbers from 0 to 9, while the current model needs to train on astronomical objects like stars and others. This logic is also behind the increase in filters in the layers, allowing the model to remember more complex patterns [74].

2. The use of *Batch Normalization* after every *conv2d* layer helps stabilize the model and prevent overfitting [75].

3. The *conv2d* layer has a padding option. It only gives two options: either cut every border case, so we are losing data, or add a padding of zeros, which is quite troublesome with the data used due to its exceptionally high dynamic range, meaning that those zeros have a more significant impact than usual, which could result in the training on the images border being corrupted.
   So *lambda* layers in the model were introduced, whose main goal is to create a symmetrical pad around the image, which mirrors the values directly around it. Using it with the padding in the *conv2d* that cuts the border cases will cut off this symmetric padding; this way, it is possible to keep the data on the borders while reducing possible impacts the padding could have.

4. Originally, neither *maxpolling2d* layers nor *upsampling2d* layers were used in the model since *conv2d* and *conv2dtranspose* layers have stride options that do what these layers do: which is to divide the image size by the stride value (*maxpolling2d/conv2d*) or multiply it by the stride value (*upsampling2d/conv2dtranspose*), said stride value is two in the layers used. But using these layers created checkerboard artifacts [76]. Initially it was thought that this checkerboard artifact was related to the kernels being uneven, which was a possibility being them, in this case, was due to the stride in the *conv2d* layers, which during the model training can modify the weights, training it to create the artifacts shown in the Fig.3.17. This occurs during the decoder step, where the image is reconstructed. The solution was the removal from the stride variable from the *conv2d* and *conv2dtranspose* layers and instead the use of the *maxpolling2d upsampling2dseparating* which do not modify the model weight during training, resolving this problem. While the layers in the encoder (*maxpolling2d* and *conv2d*) step theoretically do not affect nor create these artifacts, they were still modified in order to attempt to keep the expected symmetry between both encoder and decoder while this change also helped with the use the custom padding talk in depth previously, since the use of only *conv2d* with stride resulted in the custom padding increase size and needing extra steps to remove it.

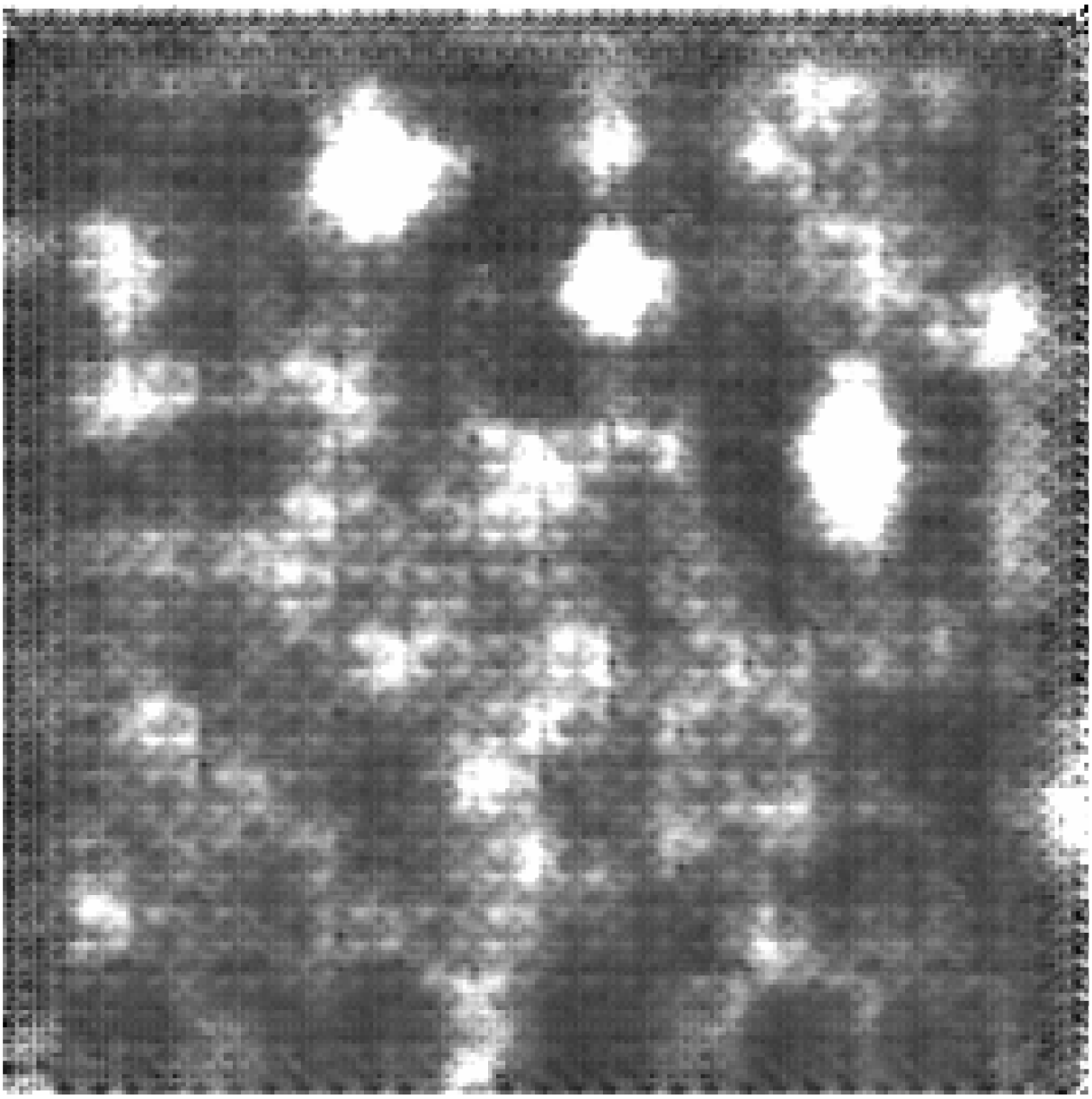

Figure 3.17: Predicted Image showing checkerboard artifacts

It was used the Adam optimizer, and the loss function is binary cross entropy [77]. As stated in the citations [78, 79] this loss function can be used outside of just binary labels (0 and 1) it also works with a range of [0, 1], it can be used in this Autoencoder due to the images being normalized to a range of 0 and 1, allowing for each predicted pixel be compared to its equivalent real pixel in the expected output, and then calculating the loss based on the discrepancies between both, there is also a boon in using this loss function due to ending the layers with a *conv2d* with a *sigmoid* activation function, which helps avoid loss of saturation.

3.6.5 CNN Model

The current CNN model is based on the autoencoder model, reusing the encoder, since both use *conv2d* layers, but instead of reducing the noise by rebuilding the image like tge autoencoder, it will give four possible labels: 0 = clean, 1 = Gaussian, 2 = Impulse, 3 = Poisson, by flattening the images and finishing with a dense layer using the activation function *softmax*, which makes the result of the model an array with the size of the number of labels, where it will predict the probability of it belonging to each label, where the sum of all members will always total 1 (100%). The structure of the model layers can be seen in the Fig.3.18.
Afterwards a relevant noise reduction algorithm or autoencoder will be chosen to reduce the noise.

There are two main changes in the reused encoder: one of the *conv2d* layers is removed due to it creating a lot of overfitting and the batch normalization layers because for some undisclosed reason, their use makes the validation results extremely different, for example, while the loss was around 0.2 in training it would give a validation loss of +100. However, even that is largely inconsistent, sometimes in the same epoch, being just +3 or an acceptable value, only for it to go back to +70 in the next epoch, jumping up and down by large values, and just removing this layer would fix the result back to an acceptable amount of variation.
The optimizer is still the Adam optimizer, but the loss function was changed to sparse categorical cross-entropy since it can deal with multi-label models.

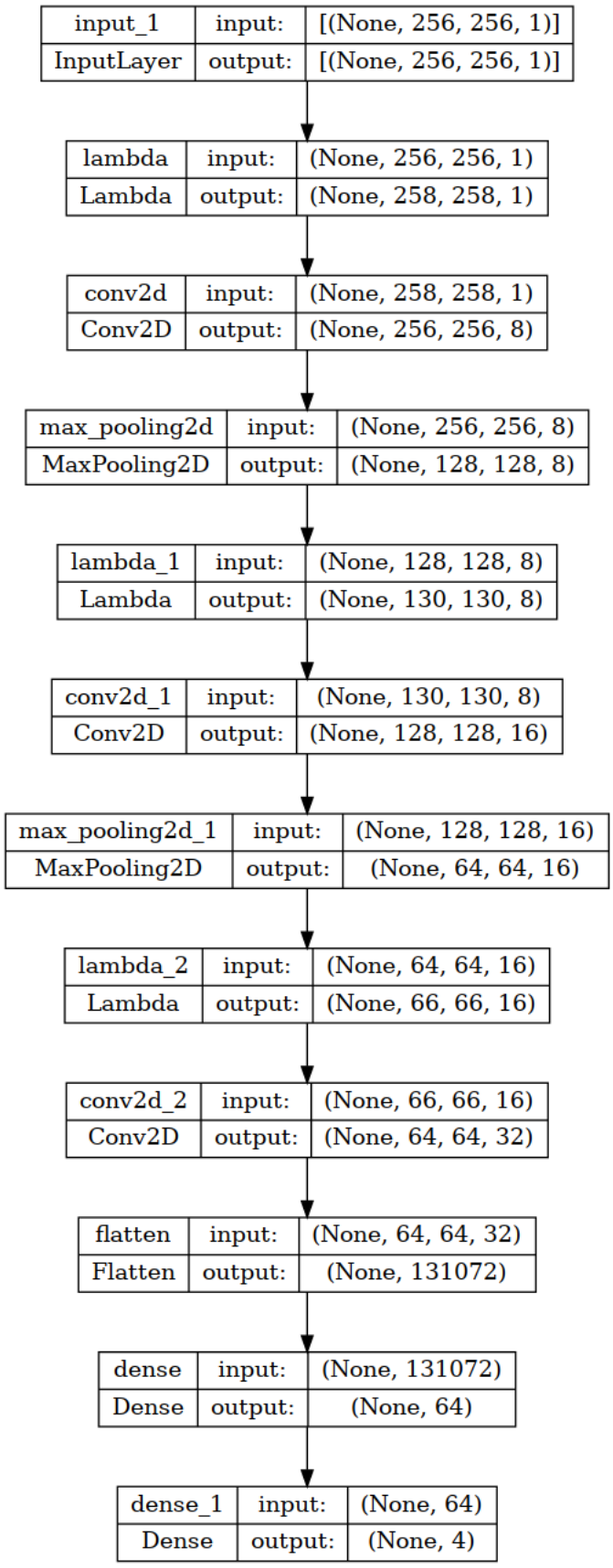

Figure 3.18: CNN Structure

3.7 Results

3.7.1 Training

The tables in this section will showcase the results from the losses and metrics for all the models. The tables epoch column has three possible states: when it showcases a number that line is the results from the training of said epoch number, the line below it showcases the validation of the relevant epoch and the validation used 20% of the total dataset, removing it from the dataset used in training, and the last line of the table shows the results of the secondary validation that was ran using the saved model, that was ran in a local PC using the prototype dataset, created in the local PC for initial tests, which contains 27000 simulated images with the same ratio and organization as the dataset used in the training of the models

While irrelevant to the results themselves, it is interesting to note that the secondary dataset is located in a personal PC, which has a NVIDIA GeForce RTX 3080 with 10 GB VRAM, AMD Ryzen 9 5950X 16-Core Processor, and 32 GB RAM, on which it took 1278 seconds (47.3 milliseconds per image) to run the secondary evaluation of the Autoencoder, compared to the around 240 seconds (0.34 milliseconds per image) in average per epoch during training. Showcasing that even with the extra time that training takes compared to validating, the change from the HPC to a decently powerful local PC had a increase in 99.2812% of runtime per image.

3.7.2 Autoencoder Results

The metrics used in the training and validation of the Autoencoders are:

- Binary Crossentropy as the loss function: It compares and runs its loss equation on each pixel from the prediction to its real equivalent in the expected output image [78, 79].
- Mean Squared Logarithmic Error (MSLE): This metric is the logarithm version of MSE, which is the squared average of the subtraction of the predicted and real, it is capable of dealing with some problems that MSE has while dealing with data or large magnitude in the range of [0,1]. It gives more focus to large differences between both, increasing the metric value a lot more than smaller differences, and it outputs zero if the images are identical, due to the large magnitude involved due to the normalization of the dataset, the results will also have a large magnitude. [80, 81].
- Mean Absolute Percentage Error(MAPE): This metric will create the absolute average of the subtraction of the predicted and expected in percentage, which gives the focus of the errors in a linear way, so while a larger error will still have more weight, an error half its size will have exactly half its weight, and it also wants the output to be as close to zero as possible. [80]

Epoch	Time(s)	Loss	MSLE	MAPE
1	276	0.1917	1.1886e-04	1.1325
Validation		0.0993	1.0909e-05	1.0775
2	243	0.0992	5.6389e-06	0.5065
Validation		0.0992	2.5167e-06	0.1757
3	237	0.0992	4.3835e-06	0.5058
Validation		0.0992	2.5716e-06	0.4332
4	247	0.0992	2.7601e-06	0.4113
Validation		0.0992	1.2202e-06	0.1555
Secondary Validation	1278	0.0994	2.3205e-06	0.1825

Table 3.1: Multipurpose Autoencoder Results

Epoch	Time(s)	Loss	MSLE	MAPE
1	111	0.1017	5.2254e-04	13.4267
Validation		0.0992	7.1197e-06	2.5224
2	131	0.0992	5.8031e-06	4.0872
Validation		0.0992	4.2271e-06	3.1736
3	119	0.0992	3.7641e-06	3.5828
Validation		0.0992	3.8563e-06	3.1520
4	126	0.0992	3.0371e-06	3.0927
Validation		0.0992	4.2026e-06	2.9270
Secondary Validation	511	0.0995	4.2112e-06	2.8531

Table 3.2: Gaussian Autoencoder Results

Epoch	Time(s)	Loss	MSLE	MAPE
1	110	0.1018	5.5026e-04	12.9595
Validation		0.0992	6.2251e-06	2.1864
2	131	0.0992	5.3744e-06	3.4182
Validation		0.0992	3.6032e-06	2.2862
3	124	0.0992	3.6580e-06	3.0509
Validation		0.0992	2.5262e-06	2.0158
4	122	0.0992	2.8797e-06	2.8321
Validation		0.0992	2.1335e-06	2.1263
Secondary Validation	514	0.0994	2.1404e-06	2.1222

Table 3.3: Impulse Autoencoder Results

Epoch	Time(s)	Loss	MSLE	MAPE
1	74	0.1027	7.2065e-04	17.1765
Validation		0.0993	9.9767e-06	2.6702
2	75	0.0992	8.3977e-06	3.0150
Validation		0.0992	5.8689e-06	2.2215
3	68	0.0992	6.4129e-06	3.4289
Validation		0.0992	5.0487e-06	2.1628
4	69	0.0992	5.1286e-06	3.4028
Validation		0.0992	3.6420e-06	2.0563
Secondary Validation	255	0.0994	3.6161e-06	2.0491

Table 3.4: Poisson Autoencoder Results

From the tables to begin with it can be verified that training results and validations are very close proving that the model is not suffering from overfitting nor underfitting. The results appear to be slightly worse on the specific noise autoencoders compared to the multipurpose autoencoder specifically in MAPE, likely due to the larger amount of data used in the training of the multipurpose autoencoder, which means the specific models have a lot more smaller errors. The MAPE is higher then MSLE which means that the model in general has more tiny errors than large ones.

3.7.3 CNN Results

The metrics used in the training and validation of the CNN are:

- Sparse Categorical Crossentropy as the loss function: Follow a similar equation to the binary cross-entropy, but capable of multiple labels, as it will expect a labels-sized array with probability and will check if the position on which the expected label is the index of the most significant percentage.
- Sparse Categorical Accuracy: It will give a percentage of how many times the prediction was correct; the closer to 100%, the better.
- Sparse Top K Categorical Accuracy: The same as the previous metric but will check if the expected label is in the next k, in this case, two, most prominent percentage predictions.

Epoch	Time(s)	Loss	SCA	STopKCA
1	107	0.2637	0.8810	0.9927
Validation		0.1881	0.8897	0.9993
2	66	0.2317	0.9004	0.9949
Validation		0.2016	0.8902	0.9993
3	64	0.2134	0.9125	0.9977
Validation		0.1822	0.9522	0.9986
4	74	0.1845	0.9180	0.9989
Validation		0.1901	0.9083	0.9986
5	68	0.1811	0.9242	0.9970
Validation		0.1101	0.9720	0.9992
6	71	0.1522	0.9304	0.9991
Validation		0.1316	0.9346	0.9993
7	71	0.1842	0.9311	0.9979
Validation		0.0955	0.9987	0.9993
8	63	0.1413	0.9432	0.9988
Validation		0.2091	0.8842	0.9999
9	64	0.1232	0.9455	0.9996
Validation		0.1920	0.8973	1.0000
10	65	0.1196	0.9468	0.9998
Validation		0.1002	0.9428	0.9995
Secondary Validation		0.0985	0.9432	1.0000

Table 3.5: CNN Results

Although the results are not as good as those of the Autoencoder, they are acceptable. There is a large amount of instability in all the metrics between each epoch, which is related to the momentum of the learning curve on the optimizer. Sometimes, the model learns specific patterns, which, while at the time they might decrease loss, will eventually start being a step in the wrong direction. However, due to its momentum, it will continue for a while. It eventually realizes the mistake and stabilizes itself, as seen in the graphics, where they all go into very similar values in the last epoch [82].
Also worth noting that the top k is pretty much always or highly close to 100% due to it checking the top 2 out of 4, which, while somewhat pointless in the current model, will be necessary once there are multiple options or the remaining types of noise are added.

Table 3.6 shows the confusion matrix of the secondary validation, where it showcases that pretty much all errors come from clean images being detected as images with Gaussian noise, being almost one-third of the total amount of clean images being miss labeled. And there is almost no miss classification in between types of noise.

	Predicted Class			
	Clean	Gaussian	Impulse	Poisson
Clean	95396	48596	0	8
Gaussian	457	287543	0	0
Impulse	54	0	287946	0
Poisson	0	0	0	144000

Table 3.6: Confusion Matrix

3.7.4 Visual Results

The current section will present a sequence of figures showcasing the various models' workflow on the same image and its noisy variants, which was taken from the dataset used in secondary validation. The Fig.3.19 shows the same original image in three scales, being those by order linear - min max in grey scale, squared - zscale in grey scale and linear - min max in a non-linear color scale for the sake of seeing in more details differences in the intensity.

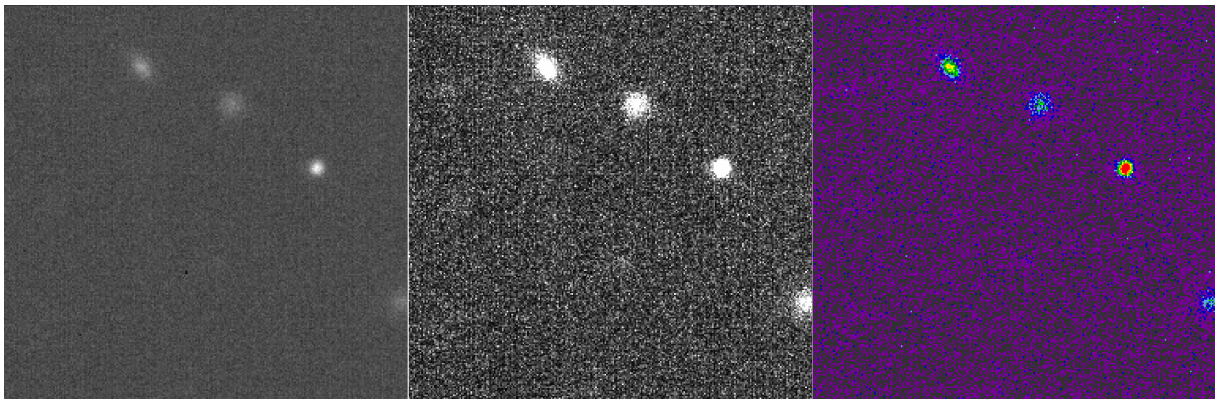

Figure 3.19: Visual Representation of the original image in multiple scales

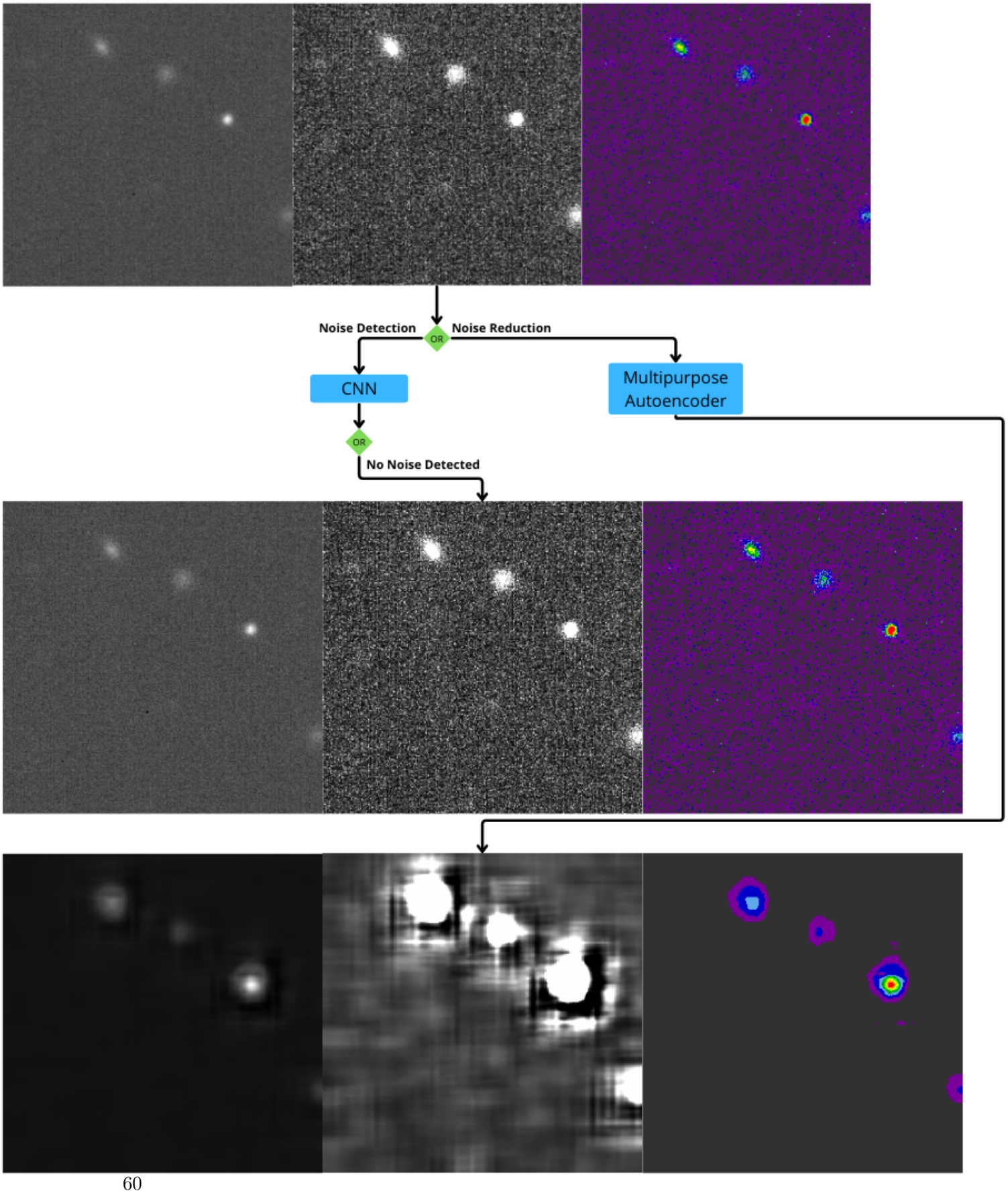

Figure 3.20: Model Workflow in a Clean Image

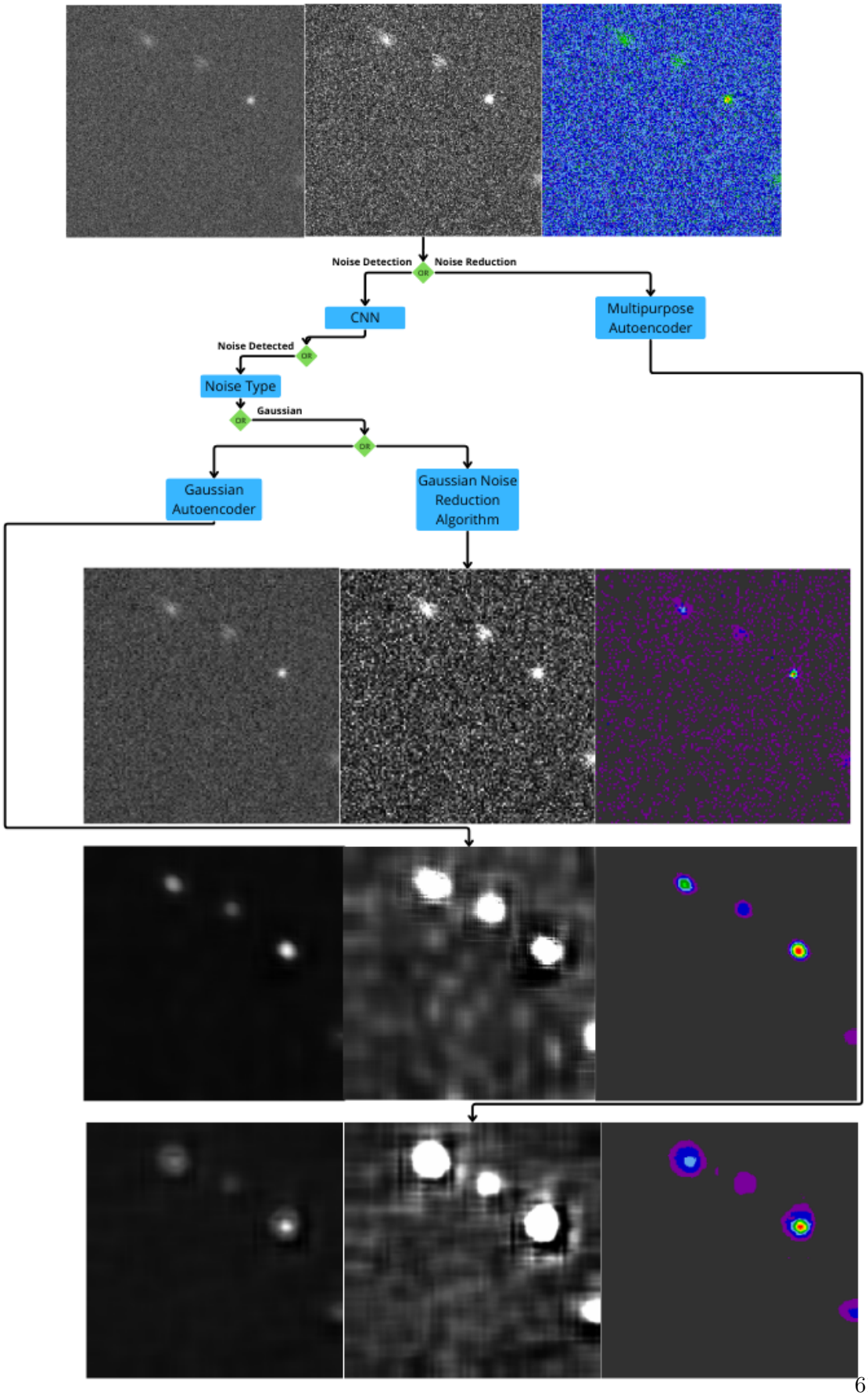

Figure 3.21: Model Workflow in a Gaussian Noise Image

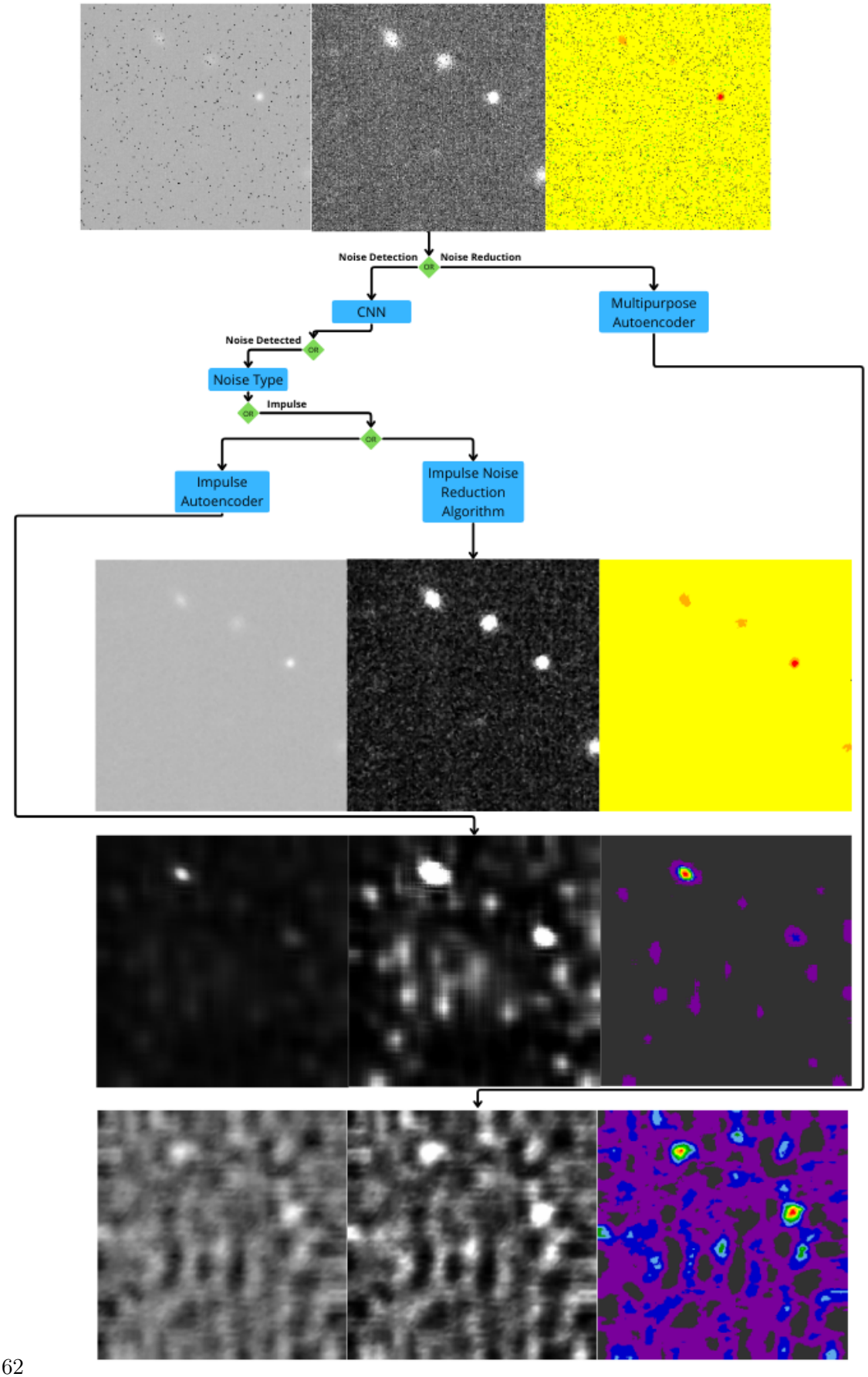

Figure 3.22: Model Workflow in a Impulse Noise Image

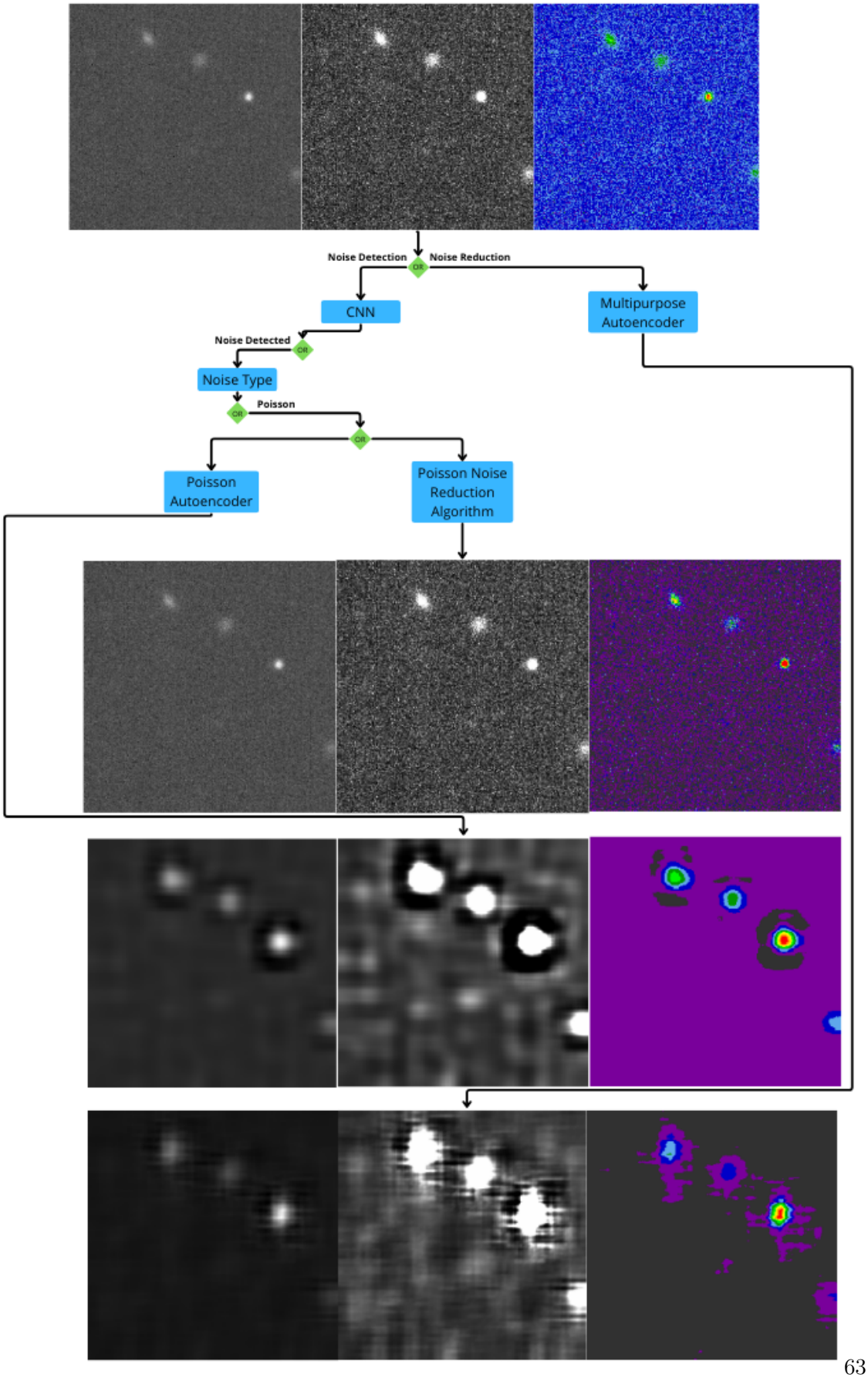

Figure 3.23: Model Workflow in a Poisson Noise Image

A significant detail, which is shown in all the autoencoder predictions, with more intensity in the multipurpose autoencoder, is that the stars look blurry and spread out. It is but a illusion, The background has become fully homogeneous. Being worth pointing out that the grey and black in the background is not zero but a range of values between [2015-2024], but this is not happening around the stars, which has a border of values higher than the background, which creates this visual blur, this effect is the most prominent in the third image of the multipurpose autoencoder of the Fig.3.20, which with its non-linear color scale its possible to view this phenomenon, and that the stars themselves compared to its original is almost the same. The specific autoencoders while still showing having the homogeneous background it has a reduced border reducing the illusion.

There is an parallel between the results in the autoencoders and the CNN. As seen in the CNN confusion matrix in the table 3.6, the most considerable confusion in the model is Clean images being confused as images with Gaussian noise, which ties together with the background becoming homogeneous in the autoencoder results since most likely the models are confusing the background noisy look with Gaussian noise or in the specific autoencoders its own noise type and reducing it. These visual results also justify the increased MAPE in comparison with MSLE, since while there is not large errors, the entire background which was composed of tiny but noticeable differences became an homogeneous value.

In the Fig.3.22, it is shown that in the multipurpose autoencoder the entire image is gone. The theory, based on what can be seen from the colored version is that, it creates new stars that originated from what it theorized is related to the salt noise, pushing the average intensity of the image upwards and hiding the entire image. A smaller scale of this can be seen in the original impulse image on the left, where it is still possible to view the original image but has a lot less intensity and looks almost erased. The specialized autoencoder for impulse noise, while also suffers slightly from this, it is still visible most of the original image.

The Gaussian Noise Reduction Algorithm in the Fig.3.21 uses Discrete Wavelet Transform Filter from the library by wavelets to reduce the noise. It is possible to see the background compared to the autoencoders, but in turn it compressed the stars.
The Impulse Noise Reduction Algorithm in the Fig.3.22 uses a Median filter from the lib scipy, which cleans the noise quite well; it for unknown reasons missed a single pepper, which caused the color scale to keep the base of the image the same, in the middle image, where it had a lot less impact in the color scale of the image, its visible how similar it is to the clean version of the image.
The Poisson Noise Reduction Algorithm in the Fig.3.21 uses Total Variation Denoising from the lib skimage, which managed to keep the data in almost the original clean state without any or at least noticeable modifications.

3.7.5 Conclusions

The results seen in this chapter are promissory, in both runtime and results from the models. There is little that can be done to decrease runtime even more then it was, and what could be done would most likely be just a negligible decrease. Meanwhile the results from the models, while visually it may seem like the autoencoders are doing a better job in most of the cases, specifically the specialized autoencoders, when it comes to actually keeping the image data the closest to the original as possible, which is very important in astronomical data, the CNN is the most promissory since while it does not actually reduce the noise by itself using it together with specialized noise reduction algorithms seems most likely to be the way to keep the most data intact, since it is not recreating the image from scratch in the decoder.

Chapter 4

Conclusion

4.1 Conclusion

In conclusion, all models have their advances and promising future. However, CNN seems to currently the best bet in continuing forward, since it keeps much more of the original data without ruining it, and what is being ruined is easily fixed with a more significant focus on the used algorithms. However, the Multipurpose Autoencoder is easier to use and create for actual use since there is no need to care for the exact type of noise and variation it is. It will clean it regardless, and a more significant and more varied dataset, most likely with real data instead of simulated, could potentially surpass even the algorithms, especially if its a specialized autoencoder instead of multipurpose, but it always keeps the risk that the actual data is no longer real, just similar, which, while in typical images is fine, it is vital that such does not happen in astronomy data. Another important detail is that while the data simulated for the dataset used was enough for the current state of the models, there is a need to have access to actual data and the simulation of more specific noise to be able to proceed into a more proceed into more complex and accurate models that work in more realistic scenarios.

4.2 Future Work

The current results look promising while still allowing for a lot of upgrades, which some of the options are:

1. Autoencoder: Find a way to stop the model from blurring all the background data. Which is most likely being created by detecting the background, like its noise, and reducing it, which might be hard to train the model out of this. Possibilities are either increasing the data in the dataset or modifying the amount of 2d convolution layers, their kernel sizes, and the number of filters in them.

2. CNN: To increase its performance, it would be related to finding better implementations for the noise cleaning algorithms or a more significant focus on said algorithms' parameters, which is unrelated to the model itself.

3. CNN: Find a way to reduce the amount of original clean images being thought to have Gaussian Noise, which is most likely related to the autoencoder problem, where in this case, it is likely detecting the background as Gaussian Noise, even while it is not there.

4. Autoencoder and CNN: Working on each model hyperparameter might help for general performance and efficiency, which should also fix the CNN's instability during training.

5. CNN: increase in the amount of data and data variety, since as of the time of writing, the noises being trained with are generic, missing or ignoring the more specific variation on them, which is essential for the CNN since most algorithms are usually tied/optimized to a version.

6. CNN: As of now, the CNN can be made and prepared to expect only one type of Noise per image and in its entirety. The next step is to accept that images might have multiple types of Noise and focus on it to pass only the noisy sub-image to the relevant noise reduction algorithm.

Bibliography

[1] URL: https://www.skao.int/en/explore/telescopes/ska-low (visited on 04/12/2024) (cit. on p. 1).

[2] *The Science of Radio Astronomy*. en-US. URL: https://public.nrao.edu/radio-astronomy/the-science-of-radio-astronomy/ (visited on 02/15/2023) (cit. on pp. 5, 6).

[3] *Significant Radio Astronomy Frequencies*. URL: http://www.setileague.org/articles/protectd.htm (visited on 02/15/2023) (cit. on p. 5).

[4] URL: https://www.seti.org/about (visited on 02/15/2023) (cit. on p. 6).

[5] *What are Radio Telescopes?* en-US. URL: https://public.nrao.edu/telescopes/radio-telescopes/ (visited on 02/15/2023) (cit. on p. 7).

[6] *Ground Telescopes | National Schools' Observatory*. URL: https://www.schoolsobservatory.org/learn/eng/tels/groundtel (visited on 02/15/2023) (cit. on p. 7).

[7] URL: https://archive.sarao.ac.za (visited on 02/15/2023) (cit. on p. 7).

[8] *Space Telescopes | National Schools' Observatory*. URL: https://www.schoolsobservatory.org/learn/eng/tels/spacetel (visited on 02/15/2023) (cit. on p. 8).

[9] Rob Garner. *Observatory - instruments*. Dec. 2017. URL: https://www.nasa.gov/content/goddard/hubble-space-telescope-science-instruments (cit. on p. 8).

[10] URL: https://astrostatistics.psu.edu/datasets/index.html (cit. on pp. 9–11).

[11] *Radio Frequency Interference - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/earth-and-planetary-sciences/radio-frequency-interference (visited on 02/15/2023) (cit. on p. 12).

[12] *Thermal Noise - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/engineering/thermal-noise (visited on 02/15/2023) (cit. on p. 12).

[13] *Atmospheric Emission - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/earth-and-planetary-sciences/atmospheric-emission (visited on 02/15/2023) (cit. on p. 12).

[14] *Sound in Space | Interstellar Audio - Cosmic Noise | Blog | RHA*. URL: https://www.rha-audio.com/blog/sound-in-space-interstellar-audio (visited on 02/15/2023) (cit. on p. 12).

[15] *Module 4: Methods of Information Collection - Section 2:3*. URL: https://ori.hhs.gov/node/1228/printable/print (visited on 02/15/2023) (cit. on p. 13).

[16] en. URL: https://hasty.ai/docs/mp-wiki/augmentations/gaussian-noise (visited on 02/15/2023) (cit. on p. 13).

[17] *Impulse Noise - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/engineering/impulse-noise (visited on 02/15/2023) (cit. on p. 13).

[18] *Baseline Drift - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/computer-science/baseline-drift (visited on 02/15/2023) (cit. on p. 13).

[19] *Narrowband Interference - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/computer-science/narrowband-interference (visited on 02/15/2023) (cit. on p. 14).

[20] *Introduction to Radio Astronomy Data Analysis I - GROWTH Astronomy School 2018*. en. URL: https://www.youtube.com/watch?v=uCPazA_-rmg (visited on 02/15/2023) (cit. on p. 14).

[21] URL: https://ui.adsabs.harvard.edu/abs/2010ascl.soft10017O/abstract (visited on 04/12/2024) (cit. on p. 14).

[22] *What is statistical analysis?* en. URL: https://www.techtarget.com/whatis/definition/statistical-analysis (visited on 02/15/2023) (cit. on p. 14).

[23] Poonam Pawar et al. "Deep Learning Approach for the detection of noise type in ancient images". In: *Sustainability* 14.18 (2022), p. 11786. DOI: 10.3390/su141811786 (cit. on p. 14).

[24] *Principal Component Analysis (PCA) Explained | Built In*. en. URL: https://builtin.com/data-science/step-step-explanation-principal-component-analysis (visited on 02/15/2023) (cit. on p. 15).

[25] S Kshipra Prasadh, Sai Sriram Natrajan, and S. Kalaivani. *Efficiency analysis of noise reduction algorithms: Analysis of the best algorithm of noise reduction from a set of algorithms*. Nov. 2017. DOI: 10.1109/ICICI.2017.8365318 (cit. on pp. 15–17).

[26] URL: https://www.prl.res.in/~shashi/astro_wavelets/report.pdf (cit. on p. 15).

[27] Oleg R. Nikitin et al. *Using the Singular Value Decomposition of Matrixs for Signal Processing*. New York, NY, USA, Dec. 2018. DOI: 10.1145/3148453.3306242. URL: https://doi.org/10.1145/3148453.3306242 (visited on 02/15/2023) (cit. on p. 15).

[28] Laura Rebollo-Neira and James Bowley. "Sparse representation of astronomical images". EN. In: *JOSA A* 30.4 (Apr. 2013), pp. 758–768. ISSN: 1520-8532. DOI: 10.1364/JOSAA.30.000758. URL: https://opg.optica.org/josaa/abstract.cfm?uri=josaa-30-4-758 (visited on 02/15/2023) (cit. on p. 16).

[29] J. Akeret et al. "Radio frequency interference mitigation using deep convolutional neural networks". en. In: *Astronomy and Computing* 18 (Jan. 2017), pp. 35–39. ISSN: 2213-1337. DOI: 10.1016/j.ascom.2017.01.002. URL: https://www.sciencedirect.com/science/article/pii/S2213133716301056 (visited on 02/15/2023) (cit. on p. 16).

[30] *Median Filter - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/computer-science/median-filter (visited on 02/15/2023) (cit. on p. 17).

[31] *Total Variation Denoising (MM Algorithm)*. URL: https://eeweb.engineering.nyu.edu/iselesni/lecture_notes/TVDmm/ (visited on 02/15/2023) (cit. on p. 17).

[32] *Independent Component Analysis - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/medicine-and-dentistry/independent-component-analysis (visited on 02/15/2023) (cit. on p. 17).

[33] Salman Ibne Eunus. *What is Non-Negative Matrix Factorization (NMF)?* en. Mar. 2022. URL: https://medium.com/codex/what-is-non-negative-matrix-factorization-nmf-32663fb4d65 (visited on 02/15/2023) (cit. on p. 17).

[34] *Empirical Mode Decomposition*. en. URL: https://www.mathworks.com/discovery/empirical-mode-decomposition.html (visited on 02/15/2023) (cit. on p. 17).

[35] *Adaptive Filter - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/computer-science/adaptive-filter (visited on 02/15/2023) (cit. on p. 17).

[36] *What is Signal to Noise Ratio and How to calculate it?* en-US. Mar. 2022. URL: https://resources.pcb.cadence.com/blog/2020-what-is-signal-to-noise-ratio-and-how-to-calculate-it (visited on 02/15/2023) (cit. on p. 18).

[37] Zhou Wang and Alan C. Bovik. "Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures". In: *IEEE Signal Processing Magazine* 26.1 (Jan. 2009), pp. 98–117. ISSN: 1558-0792. DOI: 10.1109/MSP.2008.930649 (cit. on p. 18).

[38] Jari Korhonen and Junyong You. "Peak signal-to-noise ratio revisited: Is simple beautiful?" In: *2012 Fourth International Workshop on Quality of Multimedia Experience*. July 2012, pp. 37–38. DOI: 10.1109/QoMEX.2012.6263880 (cit. on p. 18).

[39] Michael Gleicher et al. "Visual comparison for information visualization". en. In: *Information Visualization* 10.4 (Oct. 2011), pp. 289–309. ISSN: 1473-8716, 1473-8724. DOI: 10.1177/1473871611416549. URL: http://journals.sagepub.com/doi/10.1177/1473871611416549 (visited on 02/15/2023) (cit. on p. 18).

[40] Alberto M. Biancardi, Artit C. Jirapatnakul, and Anthony P. Reeves. "A comparison of ground truth estimation methods". en. In: *International Journal of Computer Assisted Radiology and Surgery* 5.3 (May 2010), pp. 295–305. ISSN: 1861-6429. DOI: 10.1007/s11548-009-0401-3. URL: https://doi.org/10.1007/s11548-009-0401-3 (visited on 02/15/2023) (cit. on p. 18).

[41] URL: http://www.cosmostat.org/wp-content/uploads/2021/01/20210305_lukic.pdf (cit. on p. 19).

[42] *CUDA Zone - Library of Resources*. en. July 2017. URL: https://developer.nvidia.com/cuda-zone (visited on 02/15/2023) (cit. on p. 20).

[43] URL: https://sites.google.com/cfa.harvard.edu/saoimageds9/home (visited on 04/05/2023) (cit. on p. 21).

[44] URL: https://data.nrao.edu/portal/#/ (visited on 04/05/2023) (cit. on p. 23).

[45] URL: https://archive.sarao.ac.za (visited on 04/05/2023) (cit. on p. 23).

[46] URL: https://almascience.eso.org/aq/ (visited on 04/05/2023) (cit. on p. 23).

[47] URL: https://skyserver.sdss.org/ (visited on 04/05/2023) (cit. on p. 23).

[48] URL: https://casa.nrao.edu (visited on 04/05/2023) (cit. on p. 25).

[49] URL: https://casadocs.readthedocs.io/en/stable/notebooks/simulation.html (visited on 04/05/2023) (cit. on p. 25).

[50] URL: https://casaguides.nrao.edu/index.php?title=Main_Page (visited on 04/05/2023) (cit. on p. 25).

[51] URL: https://colab.research.google.com/github/casangi/examples/blob/master/community/simulation_script_demo.ipynb (visited on 05/10/2023) (cit. on p. 25).

[52] URL: https://repositorio-aberto.up.pt/bitstream/10216/143458/2/574262.pdf (cit. on p. 26).

[53] Andrew L. Fielding. *Monte-Carlo techniques for radiotherapy applications I: Introduction and overview of the different Monte-Carlo Codes: Journal of Radiotherapy in practice*. Feb. 2023. URL: https://www.cambridge.org/core/journals/journal-of-radiotherapy-in-practice/article/montecarlo-techniques-for-radiotherapy-applications-i-introduction-and-overview-of-the-different-montecarlo-codes/3D953A43753566765DE3193A461D92DF (cit. on p. 26).

[54] URL: https://www.lsst.org/scientists/simulations/phosim (visited on 05/10/2023) (cit. on p. 26).

[55] URL: https://www.phosim.org (visited on 05/10/2023) (cit. on p. 26).

[56] URL: https://bitbucket.org/phosim/phosim_release/wiki/Home (visited on 05/10/2023) (cit. on p. 26).

[57] URL: https://www.itcreations.com/configurator/model/as-2124gq-nart/475 (visited on 05/10/2023) (cit. on p. 27).

[58] URL: https://docs.conda.io/projects/conda/en/stable/ (visited on 04/12/2024) (cit. on p. 28).

[59] URL: https://www.tensorflow.org/install/pip (visited on 05/10/2023) (cit. on p. 28).

[60] URL: https://www.jetbrains.com/remote-development/gateway/ (visited on 05/10/2023) (cit. on p. 28).

[61] URL: https://www.tensorflow.org/tutorials/generative/autoencoder (visited on 05/10/2023) (cit. on p. 35).

[62] URL: https://academic.oup.com/mnras/article/509/1/990/6408492 (visited on 05/10/2023) (cit. on p. 35).

[63] URL: https://iopscience.iop.org/article/10.1088/1538-3873/ace851/pdf (visited on 05/10/2023) (cit. on p. 35).

[64] URL: https://www.tensorflow.org/tutorials/images/cnn (visited on 05/10/2023) (cit. on p. 35).

[65] URL: https://www.tensorflow.org/guide/keras (visited on 05/10/2023) (cit. on p. 36).

[66] URL: https://openxla.org/xla (visited on 08/10/2023) (cit. on p. 36).

[67] URL: https://www.tensorflow.org/guide/gpu_performance_analysis (visited on 08/10/2023) (cit. on p. 37).

[68] URL: https://github.com/wjpearson/tensorflow_fits (visited on 08/10/2023) (cit. on p. 37).

[69] URL: https://www.tensorflow.org/guide/data (visited on 08/10/2023) (cit. on p. 39).

[70] URL: https://www.astropy.org (visited on 08/10/2023) (cit. on p. 41).

[71] URL: https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit (visited on 08/10/2023) (cit. on p. 47).

[72] URL: https://www.tensorflow.org/guide/distributed_training (visited on 08/10/2023) (cit. on p. 47).

[73] URL: https://blog.keras.io/building-autoencoders-in-keras.html (visited on 08/10/2023) (cit. on p. 48).

[74] URL: https://medium.com/advanced-deep-learning/cnn-operation-with-2-kernels-resulting-in-2-feature-mapsunderstanding-the-convolutional-filter-c4aad26cf32 (visited on 08/10/2023) (cit. on p. 51).

[75] URL: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/ (visited on 08/10/2023) (cit. on p. 51).

[76] URL: https://distill.pub/2016/deconv-checkerboard/ (visited on 08/10/2023) (cit. on p. 51).

[77] URL: https://arxiv.org/pdf/1412.6980v9 (visited on 04/12/2024) (cit. on p. 52).

[78] URL: https://www.deeplearningbook.org (visited on 04/12/2024) (cit. on pp. 52, 55).

[79] URL: https://stackoverflow.com/questions/52441877/how-does-binary-cross-entropy-loss-work-on-autoencoders (visited on 04/12/2024) (cit. on pp. 52, 55).

[80] URL: https://vitalflux.com/mse-vs-rmse-vs-mae-vs-mape-vs-r-squared-when-to-use/ (visited on 04/12/2024) (cit. on p. 55).

[81] URL: https://builtin.com/data-science/msle-vs-mse (visited on 04/12/2024) (cit. on p. 55).

[82] URL: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/ (visited on 04/12/2024) (cit. on p. 58).

Annexes

Annex I

Code

```
# See PyCharm help at https://www.jetbrains.com/help/pycharm/
# Import required tools/tasks
from casatools import simulator, image, table, coordsys, measures,
↪  componentlist, quanta, ctsys
from casatasks import tclean, ft, imhead, listobs, exportfits, flagdata,
↪  bandpass, applycal
from casatasks.private import simutil

import os
import pylab as pl
import numpy as np
from astropy.io import fits
from astropy.wcs import WCS
import matplotlib.pyplot as plt
# Instantiate all the required tools
sm = simulator()
ia = image()
tb = table()
cs = coordsys()
me = measures()
qa = quanta()
cl = componentlist()
mysu = simutil.simutil()

def makeMSFrame(msname='sim_data.ms'):
    """
    Construct an empty Measurement Set that has the desired observation setup.
    """
    #clear previous simulation
    os.system('rm -rf ' + msname)

    # Open the simulator
    sm.open(ms=msname)
```

```
32
33      # Read/create an antenna configuration.
34      # Canned antenna config text files are located here :
35      #
        ↪  \\wsl.localhost\Ubuntu-22.04\home\pmmf\anaconda3\envs\CASA_ENV_V2\lib\python3.8\site-packages\cas
36      antennalist = os.path.join(ctsys.resolve("alma/simmos"), "alma.all.cfg")
37
38      # Fictitious telescopes can be simulated by specifying x, y, z, d, an,
        ↪  telname, antpos.
39      #     x,y,z are locations in meters in ITRF (Earth centered) coordinates.
40      #   d, an are lists of antenna diameter and name.
41      #     telname and obspos are the name and coordinates of the observatory.
42      (x, y, z, d, an, an2, telname, obspos) = mysu.readantenna(antennalist)
43
44      # Set the antenna configuration
45      sm.setconfig(telescopename=telname,
46                   x=x,
47                   y=y,
48                   z=z,
49                   dishdiameter=d,
50                   mount=['alt-az'],
51                   antname=an,
52                   coordsystem='global',
53                   referencelocation=me.observatory(telname))
54
55      # Set the polarization mode (this goes to the FEED subtable)
56      sm.setfeed(mode='perfect R L', pol=[''])
57
58      # Set the spectral window and polarization (one data-description-id).
59      # Call multiple times with different names for multiple SPWs or pol setups.
60      sm.setspwindow(spwname="LBand",
61                     freq='1.0GHz',
62                     deltafreq='0.1GHz',
63                     freqresolution='0.2GHz',
64                     nchannels=10,
65                     stokes='RR LL')
66
67      # Setup source/field information (i.e. where the observation phase center
        ↪  is)
68      # Call multiple times for different pointings or source locations.
69      sm.setfield(sourcename="fake",
70                  sourcedirection=me.direction(rf='J2000', v0='19h59m28.5s',
                    ↪  v1='+40d44m01.5s'))
71
72      # Set shadow/elevation limits (if you care). These set flags.
73      sm.setlimits(shadowlimit=0.01, elevationlimit='1deg')
74
```

```
75      # Leave autocorrelations out of the MS.
76      sm.setauto(autocorrwt=0.0)
77
78      # Set the integration time, and the convention to use for timerange
        ↪  specification
79      # Note : It is convenient to pick the hourangle mode as all times specified
        ↪  in sm.observe()
80      #        will be relative to when the source transits.
81      sm.settimes(integrationtime='2000s',
82                  usehourangle=True,
83                  referencetime=me.epoch('UTC', '2019/10/4/00:00:00'))
84
85      # Construct MS metadata and UVW values for one scan and did
86      # Call multiple times for multiple scans.
87      # Call this with different sourcenames (fields) and spw/pol settings as
        ↪  defined above.
88      # Timesteps will be defined in intervals of 'integrationtime', between
        ↪  starttime and stoptime.
89      sm.observe(sourcename="fake",
90                 spwname='LBand',
91                 starttime='-5.0h',
92                 stoptime='+5.0h');
93
94      # Close the simulator
95      sm.close()
96
97      # Unflag everything (unless you care about elevation/shadow flags)
98      flagdata(vis=msname, mode='unflag')
99
100
101 def plotData(msname='sim_data.ms', myplot='uv'):
102     """
103     Options : myplot='uv'
104               myplot='data_spectrum'
105     """
106     from matplotlib.collections import LineCollection
107     tb.open(msname)
108
109     # UV coverage plot
110     if myplot == 'uv':
111         plt.figure(figsize=(4, 4))
112         plt.clf()
113         uvw = tb.getcol('UVW')
114         plt.plot(uvw[0], uvw[1], '.')
115         plt.plot(-uvw[0], -uvw[1], '.')
116         plt.title('UV Coverage')
117
```

```
118     # Spectrum of chosen column. Make a linecollection out of each row in the
        ↪ MS.
119     if myplot == 'data_spectrum' or myplot == 'corr_spectrum' or myplot ==
        ↪ 'resdata_spectrum' or myplot == 'rescorr_spectrum' or myplot ==
        ↪ 'model_spectrum':
120         dats = None
121         if myplot == 'data_spectrum':
122             dats = tb.getcol('DATA')
123         if myplot == 'corr_spectrum':
124             dats = tb.getcol('CORRECTED_DATA')
125         if myplot == 'resdata_spectrum':
126             dats = tb.getcol('DATA') - tb.getcol('MODEL_DATA')
127         if myplot == 'rescorr_spectrum':
128             dats = tb.getcol('CORRECTED_DATA') - tb.getcol('MODEL_DATA')
129         if myplot == 'model_spectrum':
130             dats = tb.getcol('MODEL_DATA')
131 
132         xs = np.zeros((dats.shape[2], dats.shape[1]), 'int')
133         for chan in range(0, dats.shape[1]):
134             xs[:, chan] = chan
135 
136         npl = dats.shape[0]
137         fig, ax = plt.subplots(1, npl, figsize=(10, 4))
138 
139         for pol in range(0, dats.shape[0]):
140             x = xs
141             y = np.abs(dats[pol, :, :]).T
142             data = np.stack((x, y), axis=2)
143             ax[pol].add_collection(LineCollection(data))
144             ax[pol].set_title(myplot + ' \n pol ' + str(pol))
145             ax[pol].set_xlim(x.min(), x.max())
146             ax[pol].set_ylim(y.min(), y.max())
147     plt.show()
148 
149     tb.close()
150 
151 
152 def makeCompList(clname_true='sim_onepoint.cl'):
153     # Make sure the cl doesn't already exist. The tool will complain otherwise.
154     os.system('rm -rf ' + clname_true)
155     cl.done()
156 
157     # Add sources, one at a time.
158     # Call multiple times to add multiple sources. ( Change the 'dir', obviously
        ↪ )
159     cl.addcomponent(dir='J2000 19h59m28.5s +40d44m01.5s',
160                     flux=5.0,  # For a gaussian, this is the integrated area.
```

```
                    fluxunit='Jy',
                    freq='1.5GHz',
                    shape='point',  ## Point source
                    #                   shape='gaussian',    ## Gaussian
                    #                   majoraxis="5.0arcmin",
                    #                   minoraxis='2.0arcmin',
                    spectrumtype="spectral index",
                    index=-1.0)

    # Print out the contents of the componentlist
    # print('Contents of the component list')
    # print(cl.torecord())

    # Save the file
    cl.rename(filename=clname_true)
    cl.done()


def makeEmptyImage(imname_true='sim_onepoint_true.im'):
    ## Define the center of the image
    radir = '19h59m28.5s'
    decdir = '+40d44m01.5s'

    ## Make the image from a shape
    ia.close()
    ia.fromshape(imname_true, [256, 256, 1, 10], overwrite=True)

    ## Make a coordinate system
    cs = ia.coordsys()
    cs.setunits(['rad', 'rad', '', 'Hz'])
    cell_rad = qa.convert(qa.quantity('8.0arcsec'), "rad")['value']
    cs.setincrement([-cell_rad, cell_rad], 'direction')
    cs.setreferencevalue([qa.convert(radir, 'rad')['value'], qa.convert(decdir,
    ↪  'rad')['value']], type="direction")
    cs.setreferencevalue('1.0GHz', 'spectral')
    cs.setreferencepixel([0], 'spectral')
    cs.setincrement('0.1GHz', 'spectral')

    ## Set the coordinate system in the image
    ia.setcoordsys(cs.torecord())
    ia.setbrightnessunit("Jy/pixel")
    ia.set(0.0)
    ia.close()
```

```
205 ### Note : If there is an error in this step, subsequent steps will give errors
    ↪ of " Invalid Table Operation : SetupNewTable.... imagename is already opened
    ↪ (is in the table cache)"
206 ## The only way out of this is to restart the kernel (equivalent to exit and
    ↪ restart CASA).
207 ## Any other way ?
208
209
210 def evalCompList(clname='sim_onepoint.cl', imname='sim_onepoint_true.im'):
211     ##  Evaluate a component list
212     cl.open(clname)
213     ia.open(imname)
214     ia.modify(cl.torecord(),subtract=False)
215     ia.close()
216     cl.done()
217
218
219 def editPixels(imname='sim_onepoint_true.im'):
220     ## Edit pixel values directly
221     ia.open(imname)
222     pix = ia.getchunk()
223     shp = ia.shape()
224     #pix.fill(0.0)
225     #pix[ int(shp[0]/2), int(shp[1]/2), 0, :] = 4.0    # A flat spectrum
        ↪ unpolarized source of amplitude 1 Jy and located at the center of the
        ↪ image.
226     pix[ int(shp[0]/2), int(shp[1]/2), 0, 6] = pix[ int(shp[0]/2),
        ↪ int(shp[1]/2), 0, 6] + 2.0    # Add a spectral line in channel 1
227     ia.putchunk( pix )
228     ia.close()
229
230
231 # Display an image using AstroPy, with coordinate system rendering.
232 def dispAstropy(imname='sim_onepoint_true.im'):
233         exportfits(imagename=imname, fitsimage=imname+'.fits', overwrite=True)
234         hdu = fits.open(imname+'.fits')[0]
235         wcs = WCS(hdu.header,naxis=2)
236         fig = pl.figure()
237         fig.add_subplot(121, projection=wcs)
238         plt.imshow(hdu.data[0,0,:,:], origin='lower', cmap=pl.cm.viridis)
239         plt.xlabel('RA')
240         plt.ylabel('Dec')
241
242
243 # Display an image cube or a single plane image.
244 # For a Cube, show the image at chan 0 and a spectrum at the location of the
    ↪ peak in chan0.
```

```
# For a Single plane image, show the image.
def dispImage(imname='sim_onepoint_true.im', useAstropy=False):
    ia.open(imname)
    pix = ia.getchunk()
    shp = ia.shape()
    ia.close()
    plt.figure(figsize=(10,4))
    plt.clf()
    if shp[3]>1:
        plt.subplot(121)
    if useAstropy==False:
        plt.imshow(pix[:,:,0,0])
        plt.title('Image from channel 0')
    else:
        dispAstropy(imname)
    if shp[3]>1:
        plt.subplot(122)
        ploc = np.where( pix == pix.max() )
        plt.plot(pix[ploc[0][0], ploc[1][0],0,:])
        plt.title('Spectrum at source peak')
        plt.xlabel('Channel')
    plt.show()



#makeMSFrame()
listobs(vis='sim_data.ms', listfile='obslist.txt', verbose=False,
↪  overwrite=True)
# print(os.popen('obslist.txt').read()) # ?permission denied?
fp = open('obslist.txt')
for aline in fp.readlines():
    print(aline.replace('\n',''))
fp.close()
#plotData(myplot='uv')

## Make the component list
makeCompList()

## Make an empty CASA image
makeEmptyImage()
## Evaluate the component list onto the CASA image
evalCompList()
## Edit the pixels of the CASA image directly (e.g. add a spectral line)
editPixels()

## Display
dispImage()
```

Listing I.1: CASA Simulation Test

```
from astropy.io import fits
import os
from tqdm import tqdm
from multiprocessing import Pool

def process_data(args):
    i, j, image_data, new_file_path = args
    clean_data = image_data[i:i + 256, j:j + 256]
    hdu = fits.PrimaryHDU(data=clean_data)
    hdu.writeto(f'{new_file_path}/clean_data_{i}_{j}.fits')

def main():
    image_pixel_size = 256
    step_between_images = 30
    dataset_path = "dataset"
    #if os.path.exists(dataset_path):
     #   shutil.rmtree('dataset')
    #os.makedirs(dataset_path)

    folder_path = 'OriginalImages'
    directories = [name for name in os.listdir(folder_path) if
    ↪  os.path.isdir(os.path.join(folder_path, name))]

    for directory in tqdm(directories, position=0, desc='Directories'):
        directory_path = os.path.join(folder_path, directory)
        components = directory_path.split('/')
        new_directory_path = dataset_path + "/" + components[1]
        os.makedirs(new_directory_path)
        for file_name in tqdm([f for f in os.listdir(directory_path) if
        ↪  f.endswith('.fits')], position=1, leave=False, desc='Files'):
            file_path = os.path.join(directory_path, file_name)
            head, tail = os.path.split(file_path)
            new_file_path = new_directory_path + "/" + tail[:-5]
            os.makedirs(new_file_path)
            image_data = fits.getdata(file_path, ext=0)
            with Pool(processes=112) as pool:
                args = [(i, j, image_data, new_file_path) for i in
                ↪  range(0,image_data.shape[0] - image_pixel_size - 1,
                ↪  step_between_images) for j in range(0,image_data.shape[1] -
                ↪  image_pixel_size - 1, step_between_images)]
                for _ in tqdm(pool.imap_unordered(process_data, args),
                ↪  total=len(args), desc='Processing', leave=False):
```

```
                    pass

if __name__ == '__main__':
    main()
```

Listing I.2: Code used to split the simulations

```
import numpy as np
from astropy.io import fits
import os
from tqdm import tqdm
from multiprocessing import Pool


def process_data(args):
    file_simulation_directories_path, clean_file_name, stddev_percentage_array, gaussian_folder_path = args
    clean_file_path = os.path.join(file_simulation_directories_path, clean_file_name)
    clean_image_data = fits.getdata(clean_file_path, ext=0)
    mean = 0
    dynamic_range = np.max(clean_image_data) - np.min(clean_image_data)
    for stddev_percentage in stddev_percentage_array:
        stddev = stddev_percentage * dynamic_range
        noise = np.random.normal(mean, stddev, clean_image_data.shape)
        noise_image_data = clean_image_data + noise
        noise_image_name = clean_file_name.replace("clean", "noise_gaussian_" + str(stddev_percentage))
        noise_image_path = os.path.join(os.path.join(gaussian_folder_path, str(stddev_percentage)), noise_image_name)
        hdu = fits.PrimaryHDU(data=noise_image_data.astype(np.float32))
        hdu.writeto(noise_image_path, overwrite=True)


def main():
    stddev_percentage_array = [0.05, 0.1]
    folder_path = 'dataset'
    simulation_directories = [name for name in os.listdir(folder_path) if os.path.isdir(os.path.join(folder_path, name))]
    for simulation_directory in tqdm(simulation_directories, position=0, desc='Directories'):
        simulation_directories_path = os.path.join(folder_path, simulation_directory)
```

```
30          file_simulation_directories = [name for name in
            ↪ os.listdir(simulation_directories_path) if
            ↪ os.path.isdir(os.path.join(simulation_directories_path, name))]
31          for file_simulation_directory in tqdm(file_simulation_directories,
            ↪ position=1, leave=False, desc='Simulations'):
32              file_simulation_directories_path =
                ↪ os.path.join(simulation_directories_path,
                ↪ file_simulation_directory)
33              noise_folder_name = "Noise"
34              noise_folder_path = os.path.join(file_simulation_directories_path,
                ↪ noise_folder_name)
35              if not os.path.exists(noise_folder_path):
36                  os.makedirs(noise_folder_path)
37              gaussian_folder_name = "Gaussian"
38              gaussian_folder_path = os.path.join(noise_folder_path,
                ↪ gaussian_folder_name)
39              if not os.path.exists(gaussian_folder_path):
40                  os.makedirs(gaussian_folder_path)
41              for stddev_percentage in stddev_percentage_array:
42                  tmp_folder_path = os.path.join(gaussian_folder_path,
                    ↪ str(stddev_percentage))
43                  if not os.path.exists(tmp_folder_path):
44                      os.makedirs(tmp_folder_path)
45              clean_files = [name for name in
                ↪ os.listdir(file_simulation_directories_path) if
                ↪ os.path.isfile(os.path.join(file_simulation_directories_path,
                ↪ name)) and name.endswith('.fits')]
46
47              with Pool(processes=112) as pool:
48                  args = [(file_simulation_directories_path, clean_file_name,
                    ↪ stddev_percentage_array, gaussian_folder_path) for
                    ↪ clean_file_name in tqdm(clean_files, position=2,
                    ↪ leave=False, desc='Files')]
49                  for _ in tqdm(pool.imap_unordered(process_data, args),
                    ↪ total=len(args), desc='Processing', leave=False):
50                      pass
51
52
53  if __name__ == '__main__':
54      main()
55
```

Listing I.3: Code used to add Gaussian Noise

```python
import numpy as np
from astropy.io import fits
import os
from tqdm import tqdm
from multiprocessing import Pool
from skimage.util import random_noise

def process_data(args):
    file_simulation_directories_path, clean_file_name, impulse_folder_path, amount_impulse = args
    clean_file_path = os.path.join(file_simulation_directories_path, clean_file_name)
    clean_image_data = fits.getdata(clean_file_path, ext=0)
    for amount in amount_impulse:
        noise_image_data = random_noise(clean_image_data, mode='s&p', clip=False, amount=amount)
        noise_image_name = clean_file_name.replace("clean", "noise_impulse_" + str(amount))
        noise_image_path = os.path.join(os.path.join(impulse_folder_path, str(amount)), noise_image_name)
        hdu = fits.PrimaryHDU(data=noise_image_data.astype(np.float32))
        hdu.writeto(noise_image_path, overwrite=True)


def main():
    amount_impulse = [0.01, 0.05]
    folder_path = 'dataset'
    simulation_directories = [name for name in os.listdir(folder_path) if os.path.isdir(os.path.join(folder_path, name))]
    for simulation_directory in tqdm(simulation_directories, position=0, desc='Directories'):
        simulation_directories_path = os.path.join(folder_path, simulation_directory)
        file_simulation_directories = [name for name in os.listdir(simulation_directories_path) if os.path.isdir(os.path.join(simulation_directories_path, name))]
        for file_simulation_directory in tqdm(file_simulation_directories, position=1, leave=False, desc='Simulations'):
            file_simulation_directories_path = os.path.join(simulation_directories_path, file_simulation_directory)
            noise_folder_name = "Noise"
            noise_folder_path = os.path.join(file_simulation_directories_path, noise_folder_name)
            if not os.path.exists(noise_folder_path):
                os.makedirs(noise_folder_path)
            impulse_folder_name = "Impulse"
```

```
34              impulse_folder_path = os.path.join(noise_folder_path,
                ↪  impulse_folder_name)
35              if not os.path.exists(impulse_folder_path):
36                  os.makedirs(impulse_folder_path)
37              for amount in amount_impulse:
38                  tmp_folder_path = os.path.join(impulse_folder_path, str(amount))
39                  if not os.path.exists(tmp_folder_path):
40                      os.makedirs(tmp_folder_path)
41              clean_files = [name for name in
                ↪  os.listdir(file_simulation_directories_path) if
                ↪  os.path.isfile(os.path.join(file_simulation_directories_path,
                ↪  name)) and name.endswith('.fits')]
42              with Pool(processes=112) as pool:
43                  args = [(file_simulation_directories_path, clean_file_name,
                    ↪  impulse_folder_path, amount_impulse) for clean_file_name in
                    ↪  tqdm(clean_files, position=2, leave=False, desc='Files')]
44                  for _ in tqdm(pool.imap_unordered(process_data, args),
                    ↪  total=len(args), desc='Processing', leave=False):
45                      pass
46
47
48  if __name__ == '__main__':
49      main()
50
```

Listing I.4: Code used to add Impulse Noise

```
1   import numpy as np
2   from astropy.io import fits
3   import os
4   from tqdm import tqdm
5   from multiprocessing import Pool
6
7
8   def process_data(args):
9       file_simulation_directories_path, clean_file_name, poisson_folder_path =
        ↪  args
10      clean_file_path = os.path.join(file_simulation_directories_path,
        ↪  clean_file_name)
11      clean_image_data = fits.getdata(clean_file_path, ext=0)
12      noisemap = np.ones(clean_image_data.shape) * np.mean(clean_image_data)
13      noise_image_data = clean_image_data + np.random.poisson(noisemap)
14      noise_image_name = clean_file_name.replace("clean", "noise_poisson")
15      noise_image_path = os.path.join(poisson_folder_path, noise_image_name)
16      hdu = fits.PrimaryHDU(data=noise_image_data.astype(np.float32))
```

```
    hdu.writeto(noise_image_path, overwrite=True)


def main():
    folder_path = 'dataset'
    simulation_directories = [name for name in os.listdir(folder_path) if os.path.isdir(os.path.join(folder_path, name))]
    for simulation_directory in tqdm(simulation_directories, position=0, desc='Directories'):
        simulation_directories_path = os.path.join(folder_path, simulation_directory)
        file_simulation_directories = [name for name in os.listdir(simulation_directories_path) if os.path.isdir(os.path.join(simulation_directories_path, name))]
        for file_simulation_directory in tqdm(file_simulation_directories, position=1, leave=False, desc='Simulations'):
            file_simulation_directories_path = os.path.join(simulation_directories_path, file_simulation_directory)
            noise_folder_name = "Noise"
            noise_folder_path = os.path.join(file_simulation_directories_path, noise_folder_name)
            if not os.path.exists(noise_folder_path):
                os.makedirs(noise_folder_path)
            poisson_folder_name = "Poisson"
            poisson_folder_path = os.path.join(noise_folder_path, poisson_folder_name)
            if not os.path.exists(poisson_folder_path):
                os.makedirs(poisson_folder_path)
            clean_files = [name for name in os.listdir(file_simulation_directories_path) if os.path.isfile(os.path.join(file_simulation_directories_path, name)) and name.endswith('.fits')]

            with Pool(processes=112) as pool:
                args = [(file_simulation_directories_path, clean_file_name, poisson_folder_path) for clean_file_name in tqdm(clean_files, position=2, leave=False, desc='Files')]
                for _ in tqdm(pool.imap_unordered(process_data, args), total=len(args), desc='Processing', leave=False):
                    pass


if __name__ == '__main__':
    main()

```

Listing I.5: Code used to add Poisson Noise

```
import os

os.environ['TF_XLA_FLAGS'] = '--tf_xla_enable_xla_devices=false'
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
os.environ['TF_GPU_THREAD_MODE'] = 'gpu_private'
os.environ['TF_GPU_THREAD_COUNT'] = '10'

import absl.logging

absl.logging.set_verbosity(absl.logging.ERROR)

import tensorflow as tf
import time
import numpy as np
from tf_fits.image import image_decode_fits
from tensorflow.keras import layers


class Autoencoder(tf.keras.Model):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = tf.keras.Sequential([
            layers.Input(shape=(256, 256, 1)),
            layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
            ↪  mode='SYMMETRIC')),
            layers.Conv2D(8, (3, 3), padding='valid',
            ↪  activation=tf.nn.leaky_relu),
            layers.MaxPooling2D((2, 2)),
            layers.BatchNormalization(),
            layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
            ↪  mode='SYMMETRIC')),
            layers.Conv2D(16, (3, 3), padding='valid',
            ↪  activation=tf.nn.leaky_relu),
            layers.MaxPooling2D((2, 2)),
            layers.BatchNormalization(),
            layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
            ↪  mode='SYMMETRIC')),
            layers.Conv2D(32, (3, 3), padding='valid',
            ↪  activation=tf.nn.leaky_relu),
            layers.MaxPooling2D((2, 2)),
            layers.BatchNormalization(),
```

```
37              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
38              layers.Conv2D(64, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
39              layers.MaxPooling2D((2, 2)),
40              layers.BatchNormalization(),
41
42          ])
43          self.decoder = tf.keras.Sequential([
44              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
45              layers.Conv2D(64, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
46              layers.UpSampling2D(size=(2, 2)),
47              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
48              layers.Conv2D(32, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
49              layers.BatchNormalization(),
50              layers.UpSampling2D(size=(2, 2)),
51              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
52              layers.Conv2D(16, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
53              layers.BatchNormalization(),
54              layers.UpSampling2D(size=(2, 2)),
55              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
56              layers.Conv2D(8, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
57              layers.BatchNormalization(),
58              layers.UpSampling2D(size=(2, 2)),
59              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
60              layers.Conv2D(1, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
61              layers.BatchNormalization(),
62              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
63              layers.Conv2D(1, (3, 3), padding='valid', activation=tf.nn.sigmoid),
64          ])
65
66      def call(self, x):
67          encoded = self.encoder(x)
68          decoded = self.decoder(encoded)
69          return decoded
70
```

```
71
72  def get_clean_path(path):
73      file_name = os.path.basename(path)
74      file_name_split = file_name.split("_")
75      if file_name_split[0] == "clean":
76          return path, path
77      elif file_name_split[1] == "poisson":
78          pathSplit = path.split('/')
79          cleanPath = ""
80          for i in range(len(pathSplit) - 3):
81              cleanPath += pathSplit[i] + "/"
82
83          cleanPath += 'clean_data_' + file_name_split[-2] + '_' +
            ↪  file_name_split[-1]
84          return path, cleanPath
85      else:
86          pathSplit = path.split('/')
87          # cleanPath = pathSplit[0] + '/' + pathSplit[1] + '/' + pathSplit[2] +
            ↪  '/' + 'clean_data_' + file_name_split[-2] + '_' +
            ↪  file_name_split[-1]
88          cleanPath = ""
89          for i in range(len(pathSplit) - 4):
90              cleanPath += pathSplit[i] + "/"
91
92          cleanPath += 'clean_data_' + file_name_split[-2] + '_' +
            ↪  file_name_split[-1]
93          return path, cleanPath
94
95
96  def get_files(path):
97      fits_files = []
98      for root, dirs, files in os.walk(path):
99          for file in files:
100             if file.endswith(".fits"):
101                 fits_files.append(get_clean_path(os.path.join(root, file)))
102     return fits_files
103
104
105 @tf.function(reduce_retracing=True)
106 def processing_map(input_images, output_images):
107     input_data = tf.expand_dims(tf.clip_by_value(tf.divide(input_images,
        ↪  100000.0), 0., 1.), axis=-1)
108     output_data = tf.expand_dims(tf.clip_by_value(tf.divide(output_images,
        ↪  100000.0), 0., 1.), axis=-1)
109     return input_data, output_data
110
111
```

```
@tf.function(reduce_retracing=True)
def open_files_batch(path_batch):
    input_data_batch = tf.map_fn(lambda x: tf.io.read_file(x[0]), path_batch, fn_output_signature=tf.string)
    output_data_batch = tf.map_fn(lambda x: tf.io.read_file(x[1]), path_batch, fn_output_signature=tf.string)
    return input_data_batch, output_data_batch


@tf.function(reduce_retracing=True)
def read_files_batch(input_binary_strings_batch, output_binary_strings_batch):
    input_data_batch = tf.map_fn(lambda x: image_decode_fits(x, 0), input_binary_strings_batch,
                                 fn_output_signature=tf.float32)
    output_data_batch = tf.map_fn(lambda x: image_decode_fits(x, 0), output_binary_strings_batch,
                                  fn_output_signature=tf.float32)
    return input_data_batch, output_data_batch


print("Get Dataset Paths", end='')
start_time = time.time()
path = "dataset"
file_list = get_files(path)
file_list = np.array(file_list)
np.random.shuffle(file_list)
end_time = time.time()
print(f": Took {end_time - start_time} seconds")

train_files = file_list[:int(len(file_list) * 0.8)]
val_files = file_list[int(len(file_list) * 0.8):]

train_dataset = tf.data.Dataset.from_tensor_slices(train_files) \
    .shuffle(buffer_size=100000, seed=1213, reshuffle_each_iteration=True) \
    .batch(128 * 4) \
    .map(open_files_batch, num_parallel_calls=tf.data.AUTOTUNE, deterministic=False) \
    .map(read_files_batch, num_parallel_calls=tf.data.AUTOTUNE, deterministic=False) \
    .map(processing_map, num_parallel_calls=tf.data.AUTOTUNE, deterministic=False) \
    .cache() \
    .prefetch(tf.data.AUTOTUNE)

val_dataset = tf.data.Dataset.from_tensor_slices(val_files) \
    .batch(128 * 4) \
```

```
151     .map(open_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪ deterministic=False) \
152     .map(read_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪ deterministic=False) \
153     .map(processing_map, num_parallel_calls=tf.data.AUTOTUNE,
        ↪ deterministic=False) \
154     .cache() \
155     .prefetch(tf.data.AUTOTUNE)
156
157 strategy = tf.distribute.MirroredStrategy()
158 print("Autoencoder")
159 with strategy.scope():
160     autoencoder = Autoencoder()
161     autoencoder.compile(optimizer=tf.keras.optimizers.Adam(jit_compile=False),
162                         loss=tf.keras.losses.BinaryCrossentropy(),
163                         metrics=[tf.keras.metrics.MeanSquaredError(),
                            ↪ tf.keras.metrics.MeanAbsoluteError()])
164
165 early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
    ↪ patience=3, min_delta=0.0001)
166 autoencoder.fit(train_dataset,
167                 epochs=50,
168                 verbose=1,
169                 validation_data=val_dataset,
170                 use_multiprocessing=True,
171                 workers=20,
172                 max_queue_size=1, callbacks=[early_stopping])
173
174 autoencoder.save(f'FinalAutoencoder')
```

Listing I.6: Code used to Train the Autoencoder

```
1  import os
2
3  os.environ['TF_XLA_FLAGS'] = '--tf_xla_enable_xla_devices=false'
4  os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
5  os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
6  os.environ['TF_GPU_THREAD_MODE'] = 'gpu_private'
7  os.environ['TF_GPU_THREAD_COUNT'] = '10'
8
9  import absl.logging
10
11 absl.logging.set_verbosity(absl.logging.ERROR)
12
13 import tensorflow as tf
```

```
14  import time
15  import numpy as np
16  from tf_fits.image import image_decode_fits
17  from tensorflow.keras import layers
18
19
20  def getLabel(path):  # 0 = clean, 1 = Gaussian, 2 = Impulse, 3 = Poisson
21      file_name = os.path.basename(path)
22      file_name_split = file_name.split("_")
23      if file_name_split[0] == "clean":
24          return path, 0
25      elif file_name_split[1] == "gaussian":
26          return path, 1
27      elif file_name_split[1] == "impulse":
28          return path, 2
29      else:
30          return path, 3
31
32
33  class CNN(tf.keras.Model):
34      def __init__(self):
35          super(CNN, self).__init__()
36          self.model = tf.keras.Sequential([
37              layers.Input(shape=(256, 256, 1)),
38              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
39              layers.Conv2D(8, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
40              layers.MaxPooling2D((2, 2)),
41              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
42              layers.Conv2D(16, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
43              layers.MaxPooling2D((2, 2)),
44              layers.Lambda(lambda x: tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]],
                ↪  mode='SYMMETRIC')),
45              layers.Conv2D(32, (3, 3), padding='valid',
                ↪  activation=tf.nn.leaky_relu),
46              layers.Flatten(),
47              layers.Dense(64, activation=tf.nn.leaky_relu),
48              layers.Dense(4, activation=tf.nn.softmax)
49          ])
50
51      def call(self, x):
52          model = self.model(x)
53          return model
54
```

```
55
56  def get_files(path):
57      fits_files = []
58      for root, dirs, files in os.walk(path):
59          for file in files:
60              if file.endswith(".fits"):
61                  fits_files.append(getLabel(os.path.join(root, file)))
62      return fits_files
63
64
65  @tf.function(reduce_retracing=True)
66  def open_files_batch(path_batch):
67      input_data_batch = tf.map_fn(lambda x: tf.io.read_file(x[0]), path_batch,
        ↪  fn_output_signature=tf.string)
68      output_data_batch = tf.map_fn(lambda x: tf.strings.to_number(x[1],
        ↪  out_type=tf.int32), path_batch,
69                                    fn_output_signature=tf.int32)
70      return input_data_batch, output_data_batch
71
72
73  @tf.function(reduce_retracing=True)
74  def read_files_batch(input_binary_strings_batch, output_labels_batch):
75      input_data_batch = tf.map_fn(lambda x: image_decode_fits(x, 0),
        ↪  input_binary_strings_batch,
76                                   fn_output_signature=tf.float32)
77      return input_data_batch, output_labels_batch
78
79
80  @tf.function(reduce_retracing=True)
81  def processing_map(input_images, output_label):
82      input_data = tf.expand_dims(tf.clip_by_value(tf.divide(input_images,
        ↪  100000.0), 0., 1.), axis=-1)
83      return input_data, output_label
84
85
86  print("Get Dataset Paths", end='')
87  start_time = time.time()
88  path = "dataset"
89  file_list = get_files(path)
90  file_list = np.array(file_list)
91  np.random.shuffle(file_list)
92  end_time = time.time()
93  print(f": Took {end_time - start_time} seconds")
94
95  train_files = file_list[:int(len(file_list)*0.8)]
96  val_files = file_list[int(len(file_list)*0.8):]
97
```

```
98  train_dataset = tf.data.Dataset.from_tensor_slices(train_files) \
99      .shuffle(buffer_size=100000, seed=1213, reshuffle_each_iteration=True) \
100     .batch(128*4) \
101     .map(open_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
102     .map(read_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
103     .map(processing_map, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
104     .cache() \
105     .prefetch(tf.data.AUTOTUNE)
106
107 val_dataset = tf.data.Dataset.from_tensor_slices(val_files) \
108     .batch(128*4) \
109     .map(open_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
110     .map(read_files_batch, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
111     .map(processing_map, num_parallel_calls=tf.data.AUTOTUNE,
        ↪  deterministic=False) \
112     .cache() \
113     .prefetch(tf.data.AUTOTUNE)
114
115 strategy = tf.distribute.MirroredStrategy()
116 with strategy.scope():
117     topK = tf.keras.metrics.SparseTopKCategoricalAccuracy(k=2)
118     cnn = CNN()
119     cnn.compile(optimizer=tf.keras.optimizers.Adam(jit_compile=False),
120                 loss=tf.keras.losses.SparseCategoricalCrossentropy(),
121                 metrics=[tf.keras.metrics.SparseCategoricalAccuracy(),
122                          topK,
123                          ])
124
125 early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
    ↪  patience=3, min_delta=0.0001)
126 cnn.fit(train_dataset,
127         epochs=30,
128         verbose=1,
129         validation_data=val_dataset,
130         use_multiprocessing=True,
131         workers=20,
132         max_queue_size=1,
133         callbacks=[early_stopping]
134         )
135
136
137 cnn.save(f'FinalCNN')
```

138

Listing I.7: Code used to Train the CNN

GitHub Repository: `https://github.com/pmmfsu/Noise-Reduction-in-Radio-Astronomy-Images-using-ML`