Noisier2Noise: Learning to Denoise from Unpaired Noisy Data

Yaniv Hajaj 316411578, Samy Nehmad 341218097

Faculty Advisor: Ethan Fetaya, Faculty of Engineering, Bar-Ilan University, Ramat-Gan, Israel

Abstract

In this project, we explore the Noisier2Noise method, a novel approach for image denoising that operates without the need for clean training data or paired noisy examples. Traditional image denoising techniques often require a dataset containing both noisy and corresponding clean images to effectively train neural networks. However, in real-world scenarios, acquiring such data is often impractical or impossible. Noisier2Noise overcomes this challenge by leveraging only a single noisy realization of each training example and a statistical model of the noise distribution. This allows for the training of neural networks that can perform denoising tasks under a variety of noise conditions, including spatially structured noise, without ever accessing a clean image.

Additionally, we introduce a significant enhancement to the method by incorporating an overlap image technique during the testing phase. This technique involves generating multiple predictions of the denoised image (with parameter k predictions) and then aggregating these predictions using mean or median operations to produce a final result. This approach was found to improve the model's ability to handle complex noise models and enhance the overall quality of the denoised images. The overlap image method successfully reduces the impact of noise during inference, leading to more robust and accurate image restoration.

Through a series of experiments, we validate the robustness of the Noisier2Noise approach across different datasets and noise conditions. Our findings indicate that this method not only simplifies the data collection process but also provides a powerful tool for image restoration in settings where clean data is unavailable. The results of this project highlight the potential of Noisier2Noise to advance the field of image processing, making it more accessible and applicable to a wider range of practical scenarios.

Acknowledgements

We extend our deepest gratitude to our instructor, Ethan Fetaya, whose guidance and constant support played a crucial role in the completion of this project. Throughout our fourth year, despite facing the challenges of over 300 combined days of military service, Ethan's dedication provided us with the consistent help we needed for every issue, big or small. We are profoundly thankful for his mentorship and the substantial impact he has made on our academic journey.

Lastly, we are deeply appreciative of our families and life partners—Debora and Shahar—for their unconditional support, especially during the most challenging and demanding final days of this project. Their encouragement not only sustained us but also helped make our work less "noisy."

Table of Contents

1. Introduction

In recent years, the field of image processing has seen significant advancements, particularly in the development of techniques for image denoising. Image denoising is the process of removing noise from an image, where noise refers to random variations in brightness or color that can obscure or distort the true content of the image. This is a critical task in various applications, including medical imaging, satellite imagery, and photography, where clarity and accuracy of images are paramount.

Traditionally, image denoising methods have relied on access to large datasets containing pairs of noisy and clean images. These paired datasets allow neural networks to learn the mapping from noisy inputs to their clean counterparts, effectively removing noise while preserving important image details. However, in many practical scenarios, obtaining clean images is either difficult or impossible.

This challenge has led to the development of methods that do not require paired noisy and clean images. Among these, the Noise2Noise method gained attention for its ability to train denoising networks using pairs of noisy images, thereby eliminating the need for clean targets. While effective, Noise2Noise still requires two independent noisy realizations of each image, which may not always be available in practice. This limitation sparked interest in developing even more flexible methods, leading to the proposal of the Noisier2Noise approach.

Problem Statement

The Noisier2Noise method was introduced to address the limitations of previous denoising techniques by requiring only a single noisy realization of each training image and a statistical model of the noise distribution. This method is particularly relevant in situations where obtaining multiple noisy images or clean images is impractical. Noisier2Noise offers a versatile solution that can be applied to a wide variety of noise models, including additive Gaussian noise, Poisson noise, and even spatially structured noise. The central idea of Noisier2Noise is to add synthetic noise to an already noisy image and train a neural network to predict the original noisy image from this doubly noisy input. The network learns to distinguish between the original noise and the added synthetic noise, allowing it to reconstruct a cleaner version of the image.

Significance of the Project

The significance of the Noisier2Noise approach lies in its ability to make image denoising accessible in a broader range of practical scenarios. By eliminating the need for clean training data or multiple noisy realizations, Noisier2Noise expands the applicability of denoising networks to situations where traditional methods would be infeasible.

In this project, we implement the Noisier2Noise method and evaluate its performance across different types of noise and datasets. We aim to validate the method’s robustness and explore its potential for further improvements. Our work builds on the foundational concepts introduced in the original Noisier2Noise paper, extending its application and demonstrating its effectiveness in various noise conditions. Through this project, we contribute to the growing body of research in image denoising and provide insights into how machine learning techniques can be adapted to overcome the challenges posed by noisy data.

2. Motivation

Image denoising has long been a fundamental challenge in the field of image processing, driven by the need to enhance image quality in various applications. From medical imaging to astrophotography, the ability to reduce noise and improve image clarity is critical. In space photography, for instance, images captured by telescopes or spacecraft are often degraded by various types of noise due to low light conditions, long exposure times, and the limitations of imaging equipment in harsh environments. Traditional methods of image denoising rely heavily on the availability of paired noisy and clean datasets, which are used to train neural networks to perform the denoising task. However, in many real-world scenarios, including space photography, acquiring such clean images is either impractical or impossible.

Challenges in Traditional Image Denoising

The standard approach to image denoising involves training a neural network using paired datasets, where each noisy image is matched with a corresponding clean image. The network learns to map the noisy input to its clean counterpart, effectively removing the noise.
Basically we need a set like so (with thousands of paired photos) :

While this method has proven effective, it has several significant limitations.

Firstly, the acquisition of clean images can be prohibitively expensive or technically unfeasible. For instance, in medical imaging, capturing a clean image often requires increased exposure to radiation, which poses health risks to patients. In other fields, such as remote sensing or surveillance, environmental conditions may prevent the capture of clean images altogether. As a result, the dependency on clean data restricts the applicability of traditional denoising methods.

Secondly, in situations where only noisy images are available, capturing multiple noisy realizations of the same scene is often necessary to apply methods like Noise2Noise. However, obtaining these multiple realizations can be challenging, especially in dynamic environments where the scene may change between captures. This constraint further limits the practicality of existing denoising techniques.

Motivation for Noisier2Noise

The Noisier2Noise method addresses these challenges by eliminating the need for clean images or multiple noisy realizations. Instead, it leverages a single noisy realization of each image and a statistical model of the noise distribution. This innovation opens up new possibilities for image denoising in environments where traditional methods fall short. For example, in low-light photography, capturing a clean image may require long exposure times, which can introduce motion blur and other artifacts. The ability to denoise images using only noisy data, without relying on clean examples, offers a significant advantage in such situations. In medical imaging, reducing patient exposure to radiation is a critical concern, and the ability to produce high-quality images with minimal exposure is highly desirable. By enabling denoising from unpaired noisy data, Noisier2Noise offers a pathway to achieving this goal, potentially improving patient outcomes by allowing for safer imaging protocols.
Another important motivation for this project is the potential to generalize the Noisier2Noise method across different types of noise models. Traditional denoising methods are often tailored to specific noise types, such as Gaussian or Poisson noise, and may not perform well when applied to other noise distributions. The flexibility of Noisier2Noise to handle a variety of noise models, including spatially structured noise, makes it a versatile tool for a wide range of applications. This adaptability is crucial in fields like astronomy, where noise can vary significantly depending on the source and conditions of observation.

Now we can use only set like so : (where $N \sim A$ and A is a known statistical model of the noise distribution):

Impact and Future Potential

The potential impact of Noisier2Noise extends beyond the immediate benefits of improved image quality. By making image denoising more accessible and less dependent on clean data, this method has the potential to democratize access to high-quality image processing tools. This is particularly important in resource-constrained environments, where the availability of clean data and computational resources may be limited. Additionally, the principles underlying Noisier2Noise could inspire future research in other areas of machine learning and computer vision. The idea of learning from noisy data without clean examples is a powerful concept that could be applied to other tasks, such as image reconstruction, super-resolution, and even generative modeling. By pushing the boundaries of what is possible with limited data, Noisier2Noise paves the way for new innovations in the field.

In conclusion, the motivation for developing and implementing the Noisier2Noise method is driven by the need to overcome the limitations of traditional denoising techniques and to create a more versatile and applicable solution for real-world image processing challenges. This project not only addresses a critical gap in the current methods but also contributes to the ongoing evolution of machine learning models that are robust, flexible, and capable of operating in less-than-ideal conditions.

3. Theoretical Background

Traditional Image Denoising Methods
Before the advent of deep learning, image denoising was primarily addressed using statistical and mathematical techniques. Methods such as Gaussian smoothing, median filtering, and wavelet-based denoising were widely used. These approaches, while effective to some extent, often resulted in the loss of fine details and edges, as they were based on assumptions about the nature of noise and the underlying image.
One of the most notable advancements in traditional denoising techniques was the introduction of the BM3D (Block-Matching and 3D Filtering) algorithm. BM3D leverages non-local self-similarity by grouping similar 2D image patches into 3D blocks and applying collaborative filtering. This method set a new standard for image denoising, particularly in handling additive white Gaussian noise (AWGN). However, BM3D, like other traditional methods, required an explicit noise model and was less effective when the noise deviated from the assumed model.

Machine Learning-Based Denoising
The emergence of deep learning revolutionized image processing, including denoising. Convolutional Neural Networks (CNNs), in particular, became a powerful tool for this task. CNNs learn to map noisy images to their clean counterparts by training on large datasets of paired noisy and clean images. This supervised learning approach allows the network to learn complex representations and effectively remove noise while preserving important image details.
One of the early breakthroughs in machine learning-based denoising was the Denoising Autoencoder (DAE), which trained networks to reconstruct clean images from noisy inputs. However, the requirement for clean target images remained a significant limitation, especially in scenarios where such data is difficult or impossible to obtain.

The Noise2Noise Approach
In response to the limitations of needing clean training data, Lehtinen et al. (2018) introduced the Noise2Noise method [1], which demonstrated that it is possible to train a denoising network using only noisy images. The key insight of Noise2Noise is that under certain conditions, the mean of two independent noisy images of the same scene can serve as a surrogate for the clean image. By training a network on pairs of noisy images, where the target is another noisy version of the same image, Noise2Noise effectively eliminates the need for clean data.
While Noise2Noise represented a significant advancement, it still required multiple noisy realizations of the same scene, which may not always be available. This constraint motivated further research into denoising methods that could operate with even fewer assumptions.

The Noisier2Noise Method
The Noisier2Noise method, introduced by Moran et al. (2019) [2], builds on the principles of Noise2Noise but removes the requirement for multiple noisy realizations. Instead, Noisier2Noise requires only a single noisy image and a statistical model of the noise distribution. The central idea is to create a "noisier" version of the already noisy

image by adding synthetic noise generated according to the known noise model. The network is then trained to predict the original noisy image from this doubly noisy input.

Mathematical Formulation
Gaussian Noise Typically modelled as $A \sim\ N(0, \sigma 2)$, where σ is the standard deviation of the noise. Gaussian noise is added to the clean image to produce noisy outputs, as demonstrate in the Noisier2Noise **[2]** .

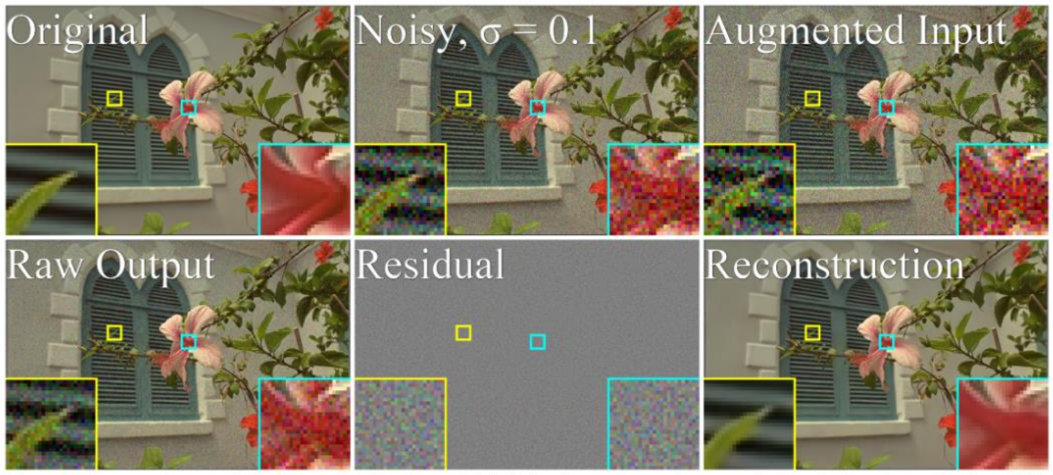


Figure 2. The steps of our method with $\sigma = 0.1$. Top Row: Original clean image (unseen by our method), a singly-noisy realization (which is our training target), a doubly-noisy realization (which is the input to our network). Bottom Row: The raw output of our network, the implicit estimate of the remaining noise, and the final reconstruction after our correction step.

In the figure above, the original image, denoted as X and located at the top left, is inaccessible to us. The image labeled Y at the top middle is the noisy version of X, with added Gaussian noise N, and serves as our training target. The process described in the article involves taking Y and adding additional noise M from the same Gaussian distribution to obtain Z.

$N \sim A$ and A is a known noise distribution, we introduce additional synthetic noise $M \sim A$ to create a noisier image:

$$Z = Y + M = X + N + M.$$

the system describe as follows:

$$X \rightarrow^{+N} Y \rightarrow^{+M} Z$$

The Noisier2Noise algorithm employs a neural network that learns to map a noisier input Z to a less noisy target Y by minimizing the expected mean squared error, represented mathematically as:

$$E_z[\| f(Z;\theta) - Y \|_2]^2$$

Given that the network never sees N or M in isolation, the ideal approach of simply subtracting M from Z isn't feasible.

Instead, the best strategy is to predict $E[Y \mid Z]$ By leveraging the relationship $E[\,Y \mid Z\,] = E[\,X + N \mid Z\,] = E[X \mid Z] + E[N \mid Z]$ and **noting the independence and identical distribution of M and N, we derive:**

$$E[Y \mid Z] = E[\,X + N \mid Z\,] = E[X \mid Z] + E[N \mid Z] \quad \rightarrow^{\text{M,N} \sim \text{A iid}}$$

$$2\text{E}[\text{Y} \mid \text{Z}] = \text{E}[\text{X} \mid \text{Z}] + \text{E}[\text{X} \mid \text{Z}] + \text{E}[\text{N} \mid \text{Z}] + \text{E}[\text{N} \mid \text{Z}] \ \rightarrow^{\text{E}[\text{N} \mid \text{Z}] = \text{E}[\text{M} \mid \text{Z}]}$$

$$2\text{E}[\text{Y} \mid \text{Z}] = \text{E}[\text{X} \mid \text{Z}] + (\text{E}[\text{X} \mid \text{Z}] + \text{E}[\text{N} \mid \text{Z}] + \text{E}[\text{M} \mid \text{Z}]) \quad \rightarrow$$

$$2\text{E}[\text{Y} \mid \text{Z}] = \text{E}[\text{X} \mid \text{Z}] + \text{E}[\text{X} + \text{N} + \text{M} \mid \text{Z}] = \text{E}[\text{X} \mid \text{Z}] + \text{E}[\text{Z} \mid \text{Z}] = \text{E}[\text{X} \mid \text{Z}] + \text{Z}$$

Thus, we can extract an estimate of the clean image $E[X \mid Z]$ by calculating:

$$\boldsymbol{2E[Y \mid Z] - Z}$$

We can therefore recover an estimate of the clean image by doubling our network's output and subtracting the noisier version.

Intuitively, the network above learns to output an image halfway between its doubly-noisy input and its best estimate of the clean image.

Step 1: Generate Synthetic Noise: The method starts by creating a sample of synthetic noise using a statistical model that describes how noise typically behaves (e.g., Gaussian noise).
Step 2: Add Synthetic Noise to Noisy Image: This synthetic noise is then added to an already noisy image. This results in an image that is even noisier—hence, a "doubly-noisy" image.
Step 3: Train the Network to Predict the Original Noisy Image: The neural network is trained to take this doubly-noisy image as input and try to predict the original noisy image (before the synthetic noise was added). The key idea is that the network cannot easily distinguish between the noise that was originally in the image and the synthetic noise that was added.

Halfway Between: Since the network can't distinguish between the two types of noise, it learns to predict an image that is "halfway" between the doubly-noisy image (input) and what would be a clean image. This means it assumes that about half of the observed noise in each pixel comes from the original noise and the other half from the synthetic noise.

Estimate of Clean Image: After the network outputs an image that is halfway between the doubly-noisy image and the clean image, we can use a simple mathematical adjustment (doubling the network's output and subtracting the doubly-noisy input) to estimate what the original clean image should have looked like.

By training the network to output an image that is halfway between the doubly-noisy image and the clean image, it effectively "learns" to separate the noise from the actual content of the image. This allows the network to denoise images even when it has never seen a perfectly clean version of the image during training.

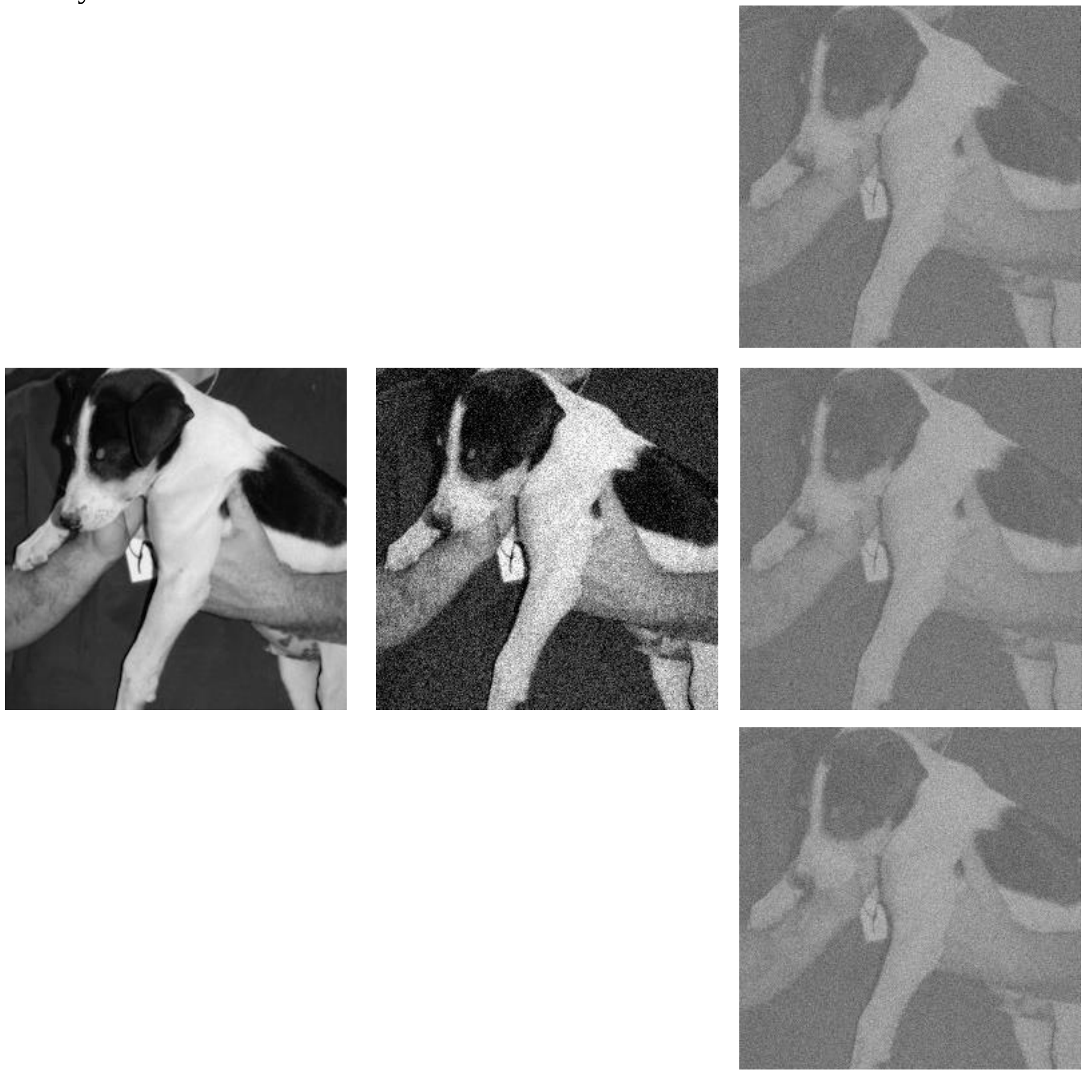
Overlapping Technique

Our goal is to expand upon this implementation even more by applying it to a larger dataset consisting of multiple images. Specifically, we will process a sequence of images $Z1, Z2, \dots, Zk,$ each derived by adding noise in the manner described. After processing these k images, we aim to aggregate the results and apply a mean/median/other operation across them to refine our solution and achieve a more effective denoising outcome.

our system describe as follows:

$$X \rightarrow^{N} Y \rightarrow^{M} \begin{array}{c} Z_1 \\ Z_2 \\ \cdot \\ \cdot \\ \cdot \\ Z_{k-1} \\ Z_k \end{array}$$

And visually looks like so :



In our implementation, we introduced an overlapping technique aimed at refining the denoising process by averaging the predictions from multiple noisy versions of the same image. Specifically, during testing, we generate k=30 different noisy versions of the input image by adding synthetic noise multiple times.

Instead of relying on a single prediction, we aggregate the outputs by calculating the **mean**, **median**, or **trimmed mean** of both across these k predictions. This approach effectively reduces the impact of outlier noise artifacts in the final denoised image. Given a noisy image Y, we generate multiple noisy versions Z_1, Z_2, ..., Z_k by adding synthetic noise. Each of these versions is processed by the model to produce a prediction $f(Z_i)$. The final denoised output $\hat{X}$ is obtained by taking the mean or median of these predictions

For the mean:

$$\hat{X} = \frac{1}{k}\sum_{i=1}^{k} f(Z_i)$$

For the median:

$$\hat{X} = \text{median}\{f(Z_1), f(Z_2), \ldots, f(Z_k)\}$$

In addition to mean and median, we also implemented a trimmed mean technique. This method takes a parameter T and removes the lowest $T\%$ and the highest $T\%$ of the predictions before calculating the mean. Our intuition behind this approach is to discard the extreme values, which may represent outliers, and thus potentially improve the denoising result. If $T = 0.1$, we remove the lowest 10% and the highest 10% of the predictions, leaving us with 80% of the data:

$$\hat{X} = \frac{1}{(1-2T)k} \sum_{i \in \text{Trimmed Set}} f(Z_i)$$

"Trimmed Set" refers to the middle $(1 - 2T)K$ % of predictions.

Potential Improvements

Method 1 (unaugmented noise input):
One complication of our method is that our network is trained to take as input doubly-noisy images, and we therefore need to add this extra noise even during inference, resulting in the network having an artificially poor view of the noisy image. It may be desirable to reduce this effect, and feed the network an input that is closer to the singly-noisy image. We explore two options for accomplishing this. **The first is to simply feed the network unaugmented noisy images at test time, with the hope that it will be somewhat robust to this shift in input distribution.** This is plausible because, during training, the network will see local patches of images which happen to have relatively small noise values, simply due to chance.
It is not unreasonable to imagine that the network will also be capable of operating on an image in which all pixels have a smaller than normal noise magnitude. While this approach is not a principled one, we find that it is able to produce PSNR improvements over the base algorithm in practice. However, the visual quality of results from this method tend to be lower, as they appear overly smooth and lack fine detail. This approach may be well-suited for domains in which mean squared error is truly the important metric, but not in domains where visual quality is paramount.

It is important to note that when feeding the network an unaugmented input, we still must perform the same correction step as in the standard method. The network still tends to produce an output which is approximately halfway between its input and the clean target, even though its input is now less noisy than those seen during training.

Method 2 (distribution change):
The second option is to note that we need not add noise of the same intensity as the natural noise during training. For example, let our noise distribution A have standard deviation σA.
In our standard approach, we sample our synthetic noise $M \sim A$.
Instead, we could sample M ~ B, where $\sigma B < \sigma A$, thus reducing the additional distortion caused by our synthetic noise, and affording the network a clearer view of the unaugmented image. Changing the distribution from which M is sampled also changes the value of $E[Y|Z]$, and thus induces a change in our correction step. The derivation of the proper correction depends on the specific choice of A and B. Here we derive it for zero-mean Gaussians A and B with $\sigma B = \alpha\sigma A$. We will make use of the following lemma:

Lemma 3.1. *Let $N \sim \mathcal{A}$ and $M \sim \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are Gaussians with zero mean, and $\sigma_{\mathcal{B}} = \alpha\sigma_{\mathcal{A}}$. Further let $X \sim \mathcal{X}$, $Y \triangleq X + N$ and $Z \triangleq X + N + M$. Then $\mathbb{E}[M|Z] = \alpha^2\mathbb{E}[N|Z]$.*

To recover $E[X|Z]$ from an estimate of $E[Y|Z]$, we first
note that:

$$(1 + \alpha^2)E[Y|Z] = E[X|Z] + E[N|Z] + \alpha^2 E[X|Z] + \alpha^2 E[N|Z] \quad \rightarrow$$
$$(1 + \alpha^2)E[Y|Z] = \alpha^2 E[X|Z] + (E[X|Z] + E[N|Z] + E[M|Z]) \quad \rightarrow$$
$$(1 + \alpha^2)E[Y|Z] = \alpha^2 E[X|Z] + Z$$

$Therefore$:

$$E[X|Z] = \frac{(1 + \alpha^2)E[Y|Z] - Z}{\alpha^2}$$

When $\alpha = 1$, this reduces to $E[X|Z] = 2E[Y|Z] - Z$, exactly the formula derived in the previous section. We note that the optimal value of α may depend on the dataset and the noise model, and may be difficult or impossible to derive in the absence of clean validation data. Intuitively, a smaller α affords the network a clearer view of the original noisy image. However, during the correction step, the output of the network is multiplied by α^{-2} , so as α decreases our performance becomes more sensitive to small errors in the network's prediction. We find that a network trained for one value of α can be quickly fine-tuned to work with a new value, allowing rapid exploration of candidate values.

Understanding PSNR and SSIM in Image Quality:

PSNR (Peak Signal-to-Noise Ratio) and **SSIM (Structural Similarity Index)** are two critical metrics used to evaluate the quality of images, especially in tasks such as image denoising, compression, and restoration. These metrics provide insights into how closely a processed image resembles the original or intended appearance.

PSNR measures the ratio between the maximum possible power of a signal (in this case, the original image) and the power of corrupting noise that affects the fidelity of its representation (the processed image). It is usually expressed in decibels (dB).

The formula for PSNR is:

$$\mathrm{PSNR} = 10 \times \log_{10}\left(\frac{\mathrm{MAX}^2}{\mathrm{MSE}}\right)$$

MAX is the maximum possible pixel value of the image (255 for 8-bit grayscale image).
MSE is the mean squared error between the original and the processed image.
Intuitive Meaning:

- A higher PSNR value suggests a lower level of error and noise, indicating better image quality. High PSNR values are generally associated with images that retain fine details closely resembling the original.

SSIM assesses the visual impact of three characteristics of an image: luminance, contrast, and structure. It compares local patterns of pixel intensities that have been normalized for luminance and contrast. SSIM values range between -1 and 1, where 1 indicates perfect similarity.
Intuitive Meaning:

- A high SSIM value indicates that the structural integrity, brightness, and contrast of the image are well-preserved compared to the original. This metric correlates more strongly with visual and perceptual quality as experienced by human observers.

Practical Implications

- **PSNR** is straightforward and useful for measuring the absolute errors but may not align perfectly with human visual perception, especially in cases where error distribution is uneven across the image.
- **SSIM** is designed to improve upon the limitations of traditional metrics like PSNR by incorporating perceptual phenomena, making it more aligned with how human vision perceives image quality.
- Utilizing both metrics provides a more rounded evaluation of image quality, combining an error-based approach (PSNR) and a perception-based approach (SSIM), to better understand the effectiveness of image processing techniques.

These metrics are essential in fields like medical imaging, satellite imaging, and any application where the quality of image restoration directly impacts functionality and analytical accuracy.

4. Methodology

The implementation of the Noisier2Noise method requires careful consideration of the tools, software, and techniques used to ensure optimal performance. This section outlines the tools and software employed, provides a step-by-step guide to the training and testing processes, and explains the key arguments and parameters used in the train.py and test.py scripts. Additionally, the section includes diagrams that illustrate the overall workflow of the project.

Tools and Software Used

Our Noisier2Noise project (with help of Git **[3]**) leverages a range of software tools and libraries that are essential for developing and testing the denoising algorithm. Below is a summary of the key tools and software utilized:

- **Python 3.7+**: The primary programming language used for implementing the Noisier2Noise method. Python's extensive libraries and frameworks make it ideal for machine learning and image processing tasks.
- **PyTorch**: A deep learning framework that provides the flexibility and computational power required for training neural networks. PyTorch is particularly well-suited for research and experimentation due to its dynamic computation graph.
- **NumPy**: A fundamental package for numerical computations in Python. NumPy is used for handling arrays, performing mathematical operations, and managing data structures that support the training process.
- **OpenCV**: An open-source computer vision library used for image processing tasks, such as reading, manipulating, and saving images.
- **Scikit-Image**: A collection of algorithms for image processing in Python. It is used for various image transformations and quality metrics in the project.
- **Matplotlib**: A plotting library used to visualize results, such as loss curves and performance metrics, throughout the training and testing processes.

These tools are integrated into the project's development environment, enabling efficient model training, evaluation, and visualization of results.

Training Process

The training process for the Noisier2Noise model involves several key steps that must be carefully followed to ensure the network learns to denoise images effectively. Below is a detailed step-by-step guide to the training process:

1. **Dataset Preparation**:
    - The first step involves preparing the dataset, which consists of noisy images. In this project, we used 1000 gray photos from the net, where each image is artificially corrupted with noise according to a specific noise model (e.g., Gaussian noise).
    - The images are preprocessed by resizing, normalizing, and converting them into grayscale if necessary. The dataset is then split into training, validation, and test sets.

2. **Model Initialization**:

    - The neural network architecture, typically a convolutional neural network (CNN), is initialized with random weights. The architecture includes layers that are specifically designed to handle the input image dimensions and the expected noise characteristics.
3. **Adding Synthetic Noise**:
    - During training, additional synthetic noise is added to the noisy images in the dataset. This synthetic noise is generated according to the same noise distribution used to corrupt the images. The result is a doubly noisy image, which serves as the input to the network. The prediction is the result of $E[X|Z]$.
4. **Training Loop**:
    - The training loop iterates over the dataset for a specified number of epochs. In each iteration, a batch of doubly noisy images is passed through the network, and the network predicts the original noisy images.
    - The loss function, typically the Mean Squared Error (MSE), computes the difference between the predicted and original noisy images. The network's weights are then updated using backpropagation and an optimization algorithm, such as Adam.
5. **Loss Monitoring and Checkpointing**:
    - The loss is monitored throughout the training process to ensure that the network is learning effectively. If the loss plateaus or increases, adjustments to the learning rate or other hyperparameters may be necessary.
    - Checkpoints are saved at regular intervals to capture the state of the model at different stages of training. These checkpoints can be used for later analysis or to resume training if needed.

Testing Process

The testing process involves evaluating the trained model on a separate test set that was not used during training. The goal is to assess the model's ability to denoise images and generalize to new data. The following steps outline the testing process:

1. **Test Data Preparation**:
    - Similar to the training process, the test data is prepared by corrupting clean images with noise. The test images are preprocessed in the same way as the training images to ensure consistency.
2. **Loading the Trained Model**:
    - The trained model is loaded from the saved checkpoints. The specific checkpoint to load can be chosen based on the best performance observed during training (e.g., the checkpoint with the lowest validation loss).
3. **Inference**:
    - The test images are passed through the trained model to generate denoised outputs. During inference, the model add another synthetic noise creating a noisier version before using the trained model on it to find the $E[X|Z]$.
4. **Performance Evaluation**:
    - The performance of the model is evaluated using metrics such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM).

    These metrics provide a quantitative measure of the quality of the denoised images.
    - Visual comparisons between the noisy input, denoised output, and ground truth (if available) are also made to assess the model's effectiveness.
5. **Overlapping Technique**:
    - In our implementation, we introduced an overlapping technique aimed at refining the denoising process by averaging the predictions from multiple noisy versions of the same image. Specifically, during testing, we generate k different noisy versions of the input image by adding synthetic noise multiple times and using a overlapping (as a mean) version of them.

Diagrams of the Workflow
To better understand the flow of data and the sequence of operations in the Noisier2Noise method, the following diagrams illustrate the key components of the training and testing workflows:

1. Training Workflow:[Input Noisy Image] -> [Add Synthetic Noise] -> [Doubly Noisy Image] -> [CNN Model] -> [Predicted Noisy Image] -> [Loss Calculation] -> [Backpropagation & Optimization]

2. **Testing Workflow**: [Input Noisy Image] -> [For K times: [Add Synthetic Noise] -> [Doubly Noisy Image] -> [CNN Model] -> [Predicted Noisy Image]] -> [Use a overlapping of those K images] -> [Metrics Calculation] -> [Save Images for Visual Reference] -> [Average Metrics Calculation].

Key Arguments and Parameters
The train.py and test.py scripts include several key arguments and parameters that control the training and testing processes. Below is an explanation of the most important ones:

For train.py:
- **--exp_detail**: A description of the experiment for logging and tracking purposes. Example: "Train Nr2N public".
- **--gpu_num**: The GPU index to use. Default is 0, which refers to the first GPU.
- **--seed**: Seed for random number generation to ensure reproducibility. Default is 100.
- **--load_model**: Boolean flag indicating whether to load a pre-trained model. Default is False.
- **--load_exp_num**: The experiment number of the pre-trained model to load. Used in conjunction with --load_model. Default is 1.
- **--load_epoch**: The specific epoch of the pre-trained model to load. Used in conjunction with --load_model. Default is 500.
- **--n_epochs**: Number of epochs to train the model. Default is 500.
- **--start_epoch**: The epoch to start training from. Useful when continuing training from a checkpoint. Default is 0.
- **--decay_epoch**: The epoch after which the learning rate starts decaying. Default is 150.

- **--batch_size**: Number of samples per batch. Default is 4.
- **--lr**: Learning rate for the optimizer. Default is 0.001.
- **--noise**: Specifies the type and intensity of noise applied during training. Format: 'noise_type_intensity'. Example: 'gauss_25' or 'poisson_50'. Default is 'gauss_25'.
- **--crop**: Boolean indicating whether to crop the images during training. Default is True.
- **--patch_size**: Size of the image patches to use during training. Default is 256.
- **--normalize**: Boolean indicating whether to normalize the image data. Default is True.
- **--mean**: Mean value of training dataset used for normalization. Default is 0.4050 (if not insert will be calculated during running train.py).
- **--std**: Standard deviation of training dataset used for normalization. Default is 0.2927 (if not insert will be calculated during running train.py)

For test.py:

- **--test_info**: A string providing additional information about the test run. Default is 'None info given'.
- **--gpu_num**: The GPU index to use. Default is 0, which refers to the first GPU.
- **--seed**: Seed for random number generation to ensure reproducibility. Default is 90.
- **--exp_num**: Experiment number to identify the specific configuration used for the test. Default is 10.
- **--n_epochs**: Number of epochs for which the model was trained. Default is 180.
- **--noise**: Specifies the type and intensity of noise applied during the test. Format: 'noise_type_intensity'. Example: 'gauss_25' or 'poisson_50'. Default is 'gauss_25'.
- **--dataset**: Name of the dataset used for testing. Examples include Set12, BSD100, and Kodak. Default is Set12.
- **--exp_rep**: Experiment repetition identifier. This is an optional string that can be used to distinguish different runs of the same experiment. Default is None.
- **--aver_num**: Number of averages to be used during testing to stabilize the performance evaluation. Default is 10.
- **--alpha**: A scaling factor for the synthetic noise intensity added during the test. Default is 1.0.
- **--trim_op**: A float value representing the trimming operation parameter. Used for trimming outliers in the overlap operations. Default is 0.05.
- **--crop**: Boolean indicating whether to crop the images during testing. Default is True.
- **--patch_size**: Size of the image patches to use during testing. Default is 256.
- **--normalize**: Boolean indicating whether to normalize the image data. Default is True.
- **--mean**: Mean value used for normalization. Default is 0.4097 (calculated during train based on the train data set).
- **--std**: Standard deviation used for normalization. Default is 0.2719 (calculated during train based on the train data set).
- **--noisy_input**: Boolean indicating if the input images are already noisy. Default is False.

5. Implementation

Installation Steps

1. **Clone the Repository**
   git@github.com:YanivHajaj/BIU-Final-Project-Learning-to-Denoise-from-Unpaired-Noisy-Data.git

2. **Install the Required Python Packages (use the requirements.txt file)**

3. **Training the Model (execute model_train.py)**

4. **Testing the Model (execute test.py)**

For more information please refer to README.md

Code Structure and Main Scripts (train.py and test.py)
The project is organized to maintain clarity and ease of use, with the main scripts being train.py and test.py. These scripts are the backbone of the project, managing the model's training and evaluation processes.

- **train.py**: This script is central to the training process of the Noisier2Noise model. It handles the loading of training data, model initialization, training loops, and checkpointing. The script is modular, allowing for easy customization of training parameters and the addition of new features.
- **test.py**: The test.py script is designed for evaluating the trained model on a test dataset. It loads the model from a specified checkpoint, runs inference on the test data, and calculates key performance metrics like PSNR and SSIM. This script is essential for assessing how well the model generalizes to unseen data.

Description of Dataset Preparation and Loading
The preparation and loading of the dataset are crucial steps in ensuring the model is trained effectively. Below is an outline of these processes:

1. **Dataset Loading**:
   - The dataset consists of noisy images, which are loaded using a custom Dataset class, leveraging PyTorch's Dataset class for seamless integration.
   - Images are preprocessed by resizing, normalizing, and converting to grayscale if necessary, ensuring consistency and compatibility with the model's input requirements.
2. **Synthetic Noise Addition**:
   - During training, synthetic noise is added to the already noisy images. This process creates a "noisier" image, helping the model learn to differentiate between the original noise and the added noise, ultimately enhancing its denoising capabilities.
3. **DataLoader**:
   - The prepared dataset is loaded into memory in batches using PyTorch's DataLoader. This batching process is essential for efficient training,

allowing the model to process multiple images simultaneously and making the training process more scalable.

Key Functions and Their Purposes

- **load_state_dict()**:
  - This function is responsible for loading a pre-trained model from a specified checkpoint. It is used in both the training and testing scripts, enabling the resumption of training or evaluation of a saved model.
- **train()**:
  - The core function in train.py, responsible for iterating over the dataset, passing images through the model, calculating the loss, and updating the model weights using backpropagation. It handles the entire training loop and ensures that the model learns effectively from the data.
- **test()**:
  - The core function in test.py, responsible for evaluating the trained model on a test dataset. It loads the model from a checkpoint, processes the test images through the model, and calculates key performance metrics such as PSNR and SSIM. This function ensures that the model’s ability to denoise new images is accurately assessed, and it helps determine how well the model generalizes to unseen data.
- **calculate_metrics()**:
  - This function calculates performance metrics, including PSNR and SSIM, for different stages of the image processing pipeline. It is used to quantify the quality of the denoised images and is instrumental in comparing the model's performance across different scenarios. It is critical for assessing how well the model denoises images.

Pseudocode for train.py and test.py

Pseudocode for train.py:

```
function train():
    Load the dataset
    Initialize the model
    Set up the optimizer and loss function

    for each epoch in number_of_epochs:
        for each batch in dataset:
            Load noisy images and create "noisier" images
            Pass images through the model
            Calculate the loss (difference between prediction and original noisy image)
            Backpropagate the loss and update model weights
            Log training progress

            Calculate image metrics each 5 epochs
            Save model checkpoint each 10 epochs
    End for
End function
```

Pseudocode for test.py:

```
function test():
    Load the dataset
    Load the trained model from a checkpoint
    Set the model to evaluation mode

    for each image in test dataset:
        Pass image through the model
        Calculate the PSNR and SSIM metrics
        Save the denoised image
        Log the metrics

    End for
    Print overall test performance
End function
```

6. Design

The design of the Noisier2Noise model is central to its effectiveness in denoising images under various noise conditions. This section outlines the architecture of the neural network used and describes the specific improvements and adaptations made to the model to enhance its performance.

torch.nn

The torch.nn module is a fundamental part of PyTorch, providing the building blocks necessary for creating neural networks. It includes a wide range of predefined layers, utilities, and classes that simplify the process of constructing neural networks. Here are some of the key components found in this module:

- **Layers:** Fundamental components such as Linear (fully connected layers), Conv2d (convolutional layers), and many more that are used to build neural network architectures.
- **Activation Functions:** Non-linearities like ReLU, Sigmoid, and Tanh which are crucial for learning complex patterns in data.
- **Loss Functions:** Includes loss computation utilities such as MSELoss (mean squared error).
  used for training neural networks by comparing the output with true targets.
- **Utilities:** Tools like Module, the base class for all neural network modules which your models should also subclass. It provides features such as parameter tracking, gradients, and moving computations to different devices (CPU/GPU).

The DnCNN model, as discussed in the article "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising," [4] emphasizes a deep convolutional neural network (CNN) approach for image denoising, particularly handling unknown noise levels, such as Gaussian noise. The model integrates advanced CNN architectures, learning algorithms, and regularization techniques to enhance image denoising performance. Here’s a detailed explanation of the model architecture, layers, residual learning, batch normalization, and the mean squared error (MSE) loss function.

In our code, you can find the **DnCNN** class definition, which begins by initializing the network architecture. It sequentially adds layers as specified: convolutional layers, batch normalization, and ReLU activation functions.

Model Architecture and Layers
The DnCNN model employs a deep architecture comprising several convolutional layers that each contribute uniquely to the denoising process:

1. **Input Layer**: The network accepts noisy images, where the primary goal is to separate the clean image component from the noise.
2. **Hidden Layers**: consists of multiple layers, each structured with convolutional operations followed by batch normalization and ReLU activation functions. This setup is consistent throughout the network except for the last layer.
    - **Convolutional Layers**: These layers use 3x3 kernels which are typical in CNNs for maintaining spatial hierarchies between image pixels. Each convolutional layer aims to extract features at various levels of abstraction, progressively focusing more on noise patterns as deeper layers are reached.
    - **Batch Normalization**: Positioned after each convolution operation but before activation, this component normalizes the outputs of the convolution, stabilizing learning and allowing for higher learning rates.
    - **ReLU Activation**: This non-linearity is applied after batch normalization to introduce non-linear capabilities to the model, helping it learn more complex noise patterns.
3. **Output Layer**: The final convolutional layer maps the features learned back to the image space, aiming to output the residual image, which ideally contains the noise that needs to be subtracted from the input to obtain a denoised image.

Residual Learning
Residual learning is a pivotal aspect of the DnCNN architecture. Instead of aiming to output the clean image directly, DnCNN predicts the residual (the noise or degradation). This approach simplifies the learning task since learning the noise (often simpler and sparser than the clean image) is easier than learning the clean image directly. The output from the network is then subtracted from the noisy input to produce the denoised image, enhancing the network's ability to focus on the noise components that are more predictable and less varied than the image content itself.

Batch Normalization
Batch normalization plays a crucial role in the DnCNN framework by addressing the internal covariate shift problem—where the distribution of network activations varies significantly during training. By normalizing layer inputs to have zero mean and unit variance, batch normalization ensures that the scale of activations remains more consistent across different training batches. This normalization significantly speeds up the training process, reduces the sensitivity to network initialization, and helps in achieving faster convergence.

Mean Squared Error (MSE) Loss
The MSE loss function measures the average of the squares of the differences between the predicted residuals and the actual noise (calculated as the difference between the noisy input and the clean target). This loss function is critical as it directly encourages the

network to learn an accurate prediction of the noise, refining the network's ability to identify and subtract the noise from the noisy input image effectively.

Model Improvements and Adaptations

The Noisier2Noise **[2]** method introduces several key improvements and adaptations to the base neural network architecture to enhance its denoising capabilities include:

Adaptive Learning Rate (Adam Optimizer [5]):

- The network training process includes an adaptive learning rate schedule that adjusts the learning rate based on the progress of the training. Early in the training, a higher learning rate is used to quickly converge to a good solution. As training progresses, the learning rate is gradually reduced to fine-tune the network and prevent overshooting. This adaptive learning rate schedule helps the network achieve better generalization to unseen data.
- To further improve the optimization process, the Adam optimizer is used. Adam is an adaptive learning rate optimization algorithm designed to handle sparse gradients on noisy problems. It combines the advantages of two other popular optimizers: AdaGrad, which works well with sparse gradients, and RMSProp, which works well in online and non-stationary settings. The Adam optimizer dynamically adjusts the learning rate for each parameter, leading to more efficient convergence during training, reducing the time to achieve optimal performance while maintaining robust results across various noise distributions.

Integration in our Code

implementation of the DnCNN in Python uses these principles by constructing a sequential model with repeated layers of convolutions, batch normalization, and ReLU activations, ending with a final convolution layer that outputs the noise to be subtracted from the input. This setup aligns with the architectural principles discussed in the article **[4]**, emphasizing the effectiveness of deep learning, particularly CNNs, in addressing complex image denoising challenges through advanced architectures and techniques.

Conclusion

The detailed understanding of the network's layers and parameters, combined with targeted adaptations for specific noise types, ensures that the Noisier2Noise method is both powerful and versatile, making it a valuable tool for image denoising in challenging real-world scenarios.

7. Simulations and Results

Overview

Denoising tests performed using two datasets: Set20 and Set22. These tests involve multiple experiments to evaluate the effectiveness of the Noisier2Noise method, with a focus on parameters like the overlap technique, alpha adjustments, using noisy as a noisier input, and the application of mean, median, and trimmed mean calculations for denoising.

Data Sets Used (can be seen in the code):

1. **Set22**: Contains 100 new images added for testing. (10% of the train 1000 images, the model didn't saw in training).

2. **Set20**: Contains 10 images for testing (smaller number for quicklier and preliminary tests).

Summary of Experiments (used as baseline for the results in this project book):

- **exp9**: Set20 Examines the effectiveness of the overlap median, mean and trimmed_mean for values of K ranging from 0 to 30.

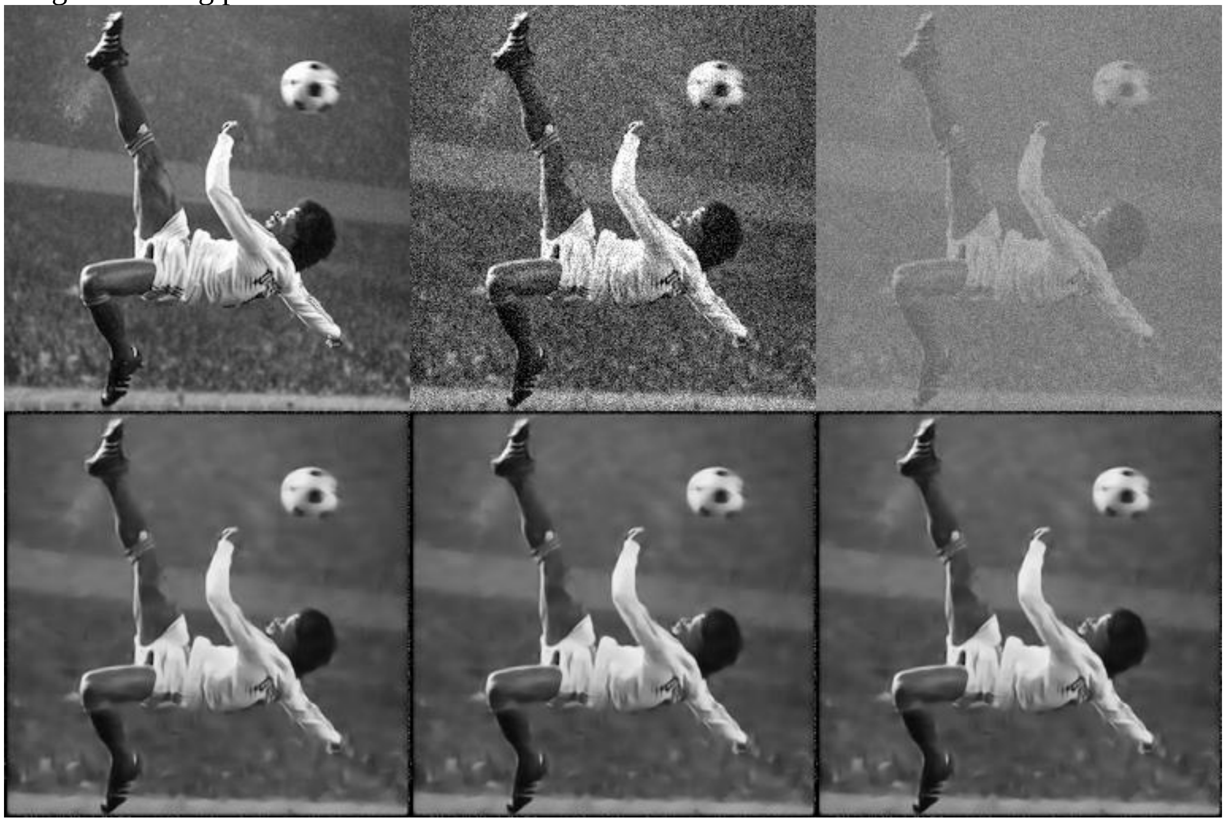
- **exp5**: Examines the effectiveness of the trimmed mean method for values of K ranging from 0 to 30, with a trimming of 5% on each side.

- **exp2-7**: Tested the distribution change improvement technique using the alpha argument and K = 10,
  alpha_values=(1.0 0.5 0.4 0.3 0.2 0.1).
  noise_values=(gauss_25 gauss_12.5 gauss_10 gauss_7.5 gauss_5 gauss_2.5)

- **exp8**: Tested the unaugmented noise input improvement technique (using noisy = noisier).

Image Denoising Results Comparison

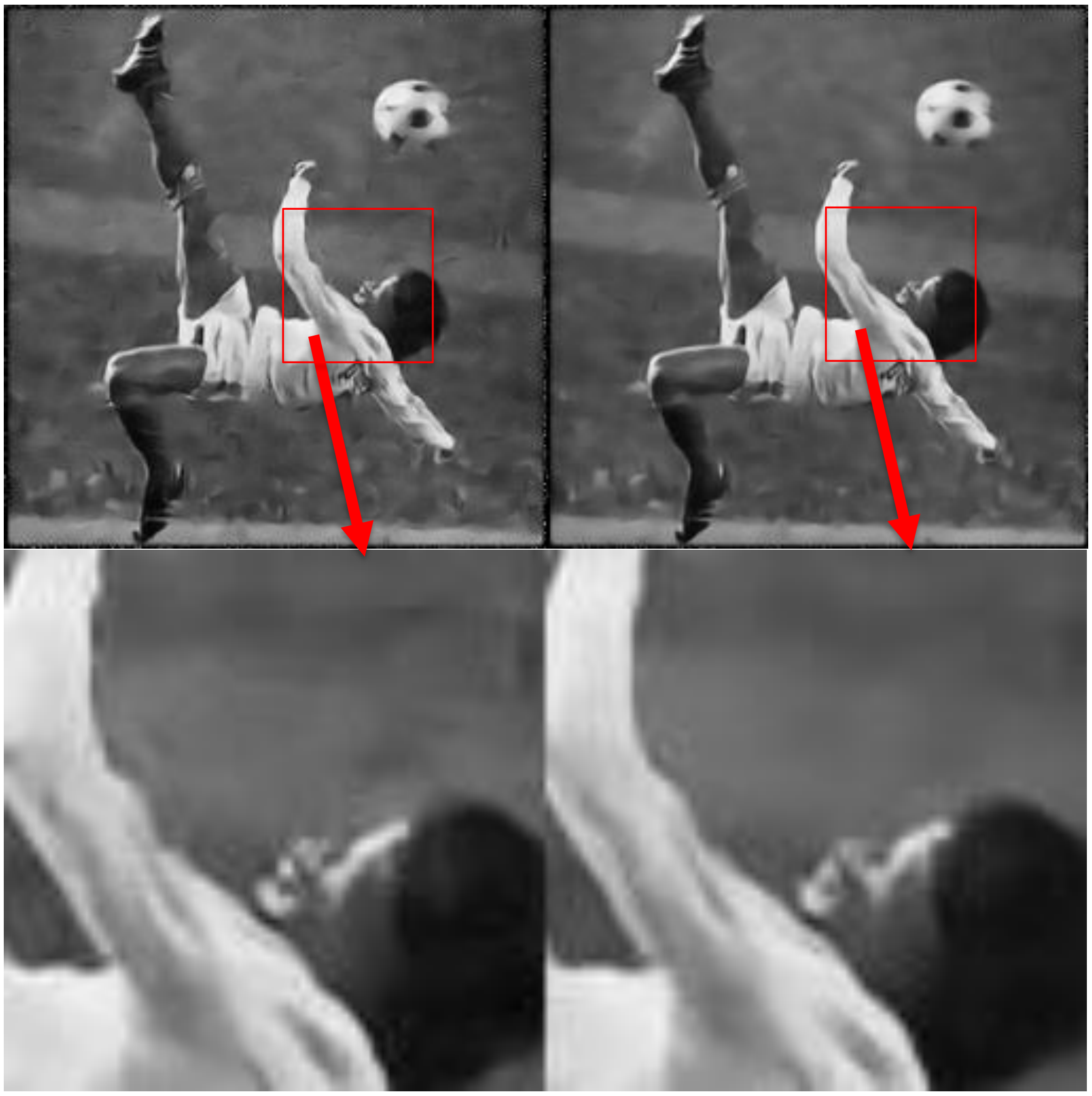
Below are the visual results from the denoising process applied to an image of an athlete performing a bicycle kick. Each image demonstrates a different stage or technique in the image denoising process.

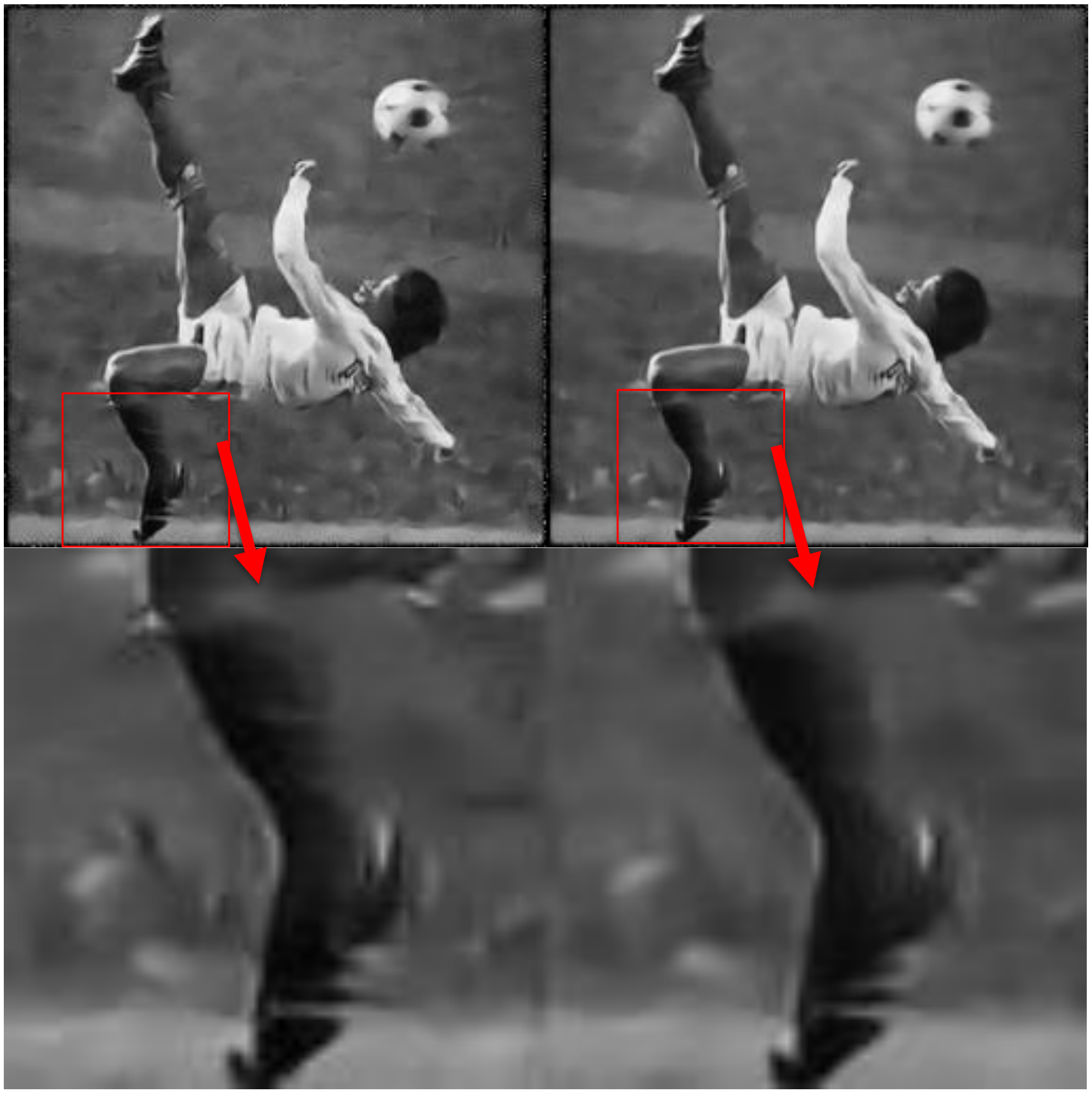


Top row (from left to right): Clean image X , Noisy image Y, Noisier image Z.
Bottom row (from left to right): Mean Overlap, Median Overlap, Trimmed Overlap.
*note that the model cannot see the clean image X.

Better than the regular prediction using k=1

This comparison illustrates how the 75x75 pixel section from the mean k=30 (which is our best overlapping result) looks better than the same section from the prediction. The clarity and detail in the median version are more prominent, demonstrating the advantage of using the median for this part of the image.


Top row (from left to right): prediction k=1, Mean Overlap k=30.
row (from left to right): prediction k=1, Mean Overlap k=30 (75X75 pixels)



- **Top row (from left to right):** prediction k=1, Mean Overlap k=30.
- **Bottom row (from left to right):** prediction k=1, Mean Overlap k=30 (75X75 pixels)

The comparison between the denoising results from the prediction and the improved overlap technique clearly highlights significant enhancements in image quality. In the overlap method, particularly the Mean Overlap k=30, the improvements are visually evident:

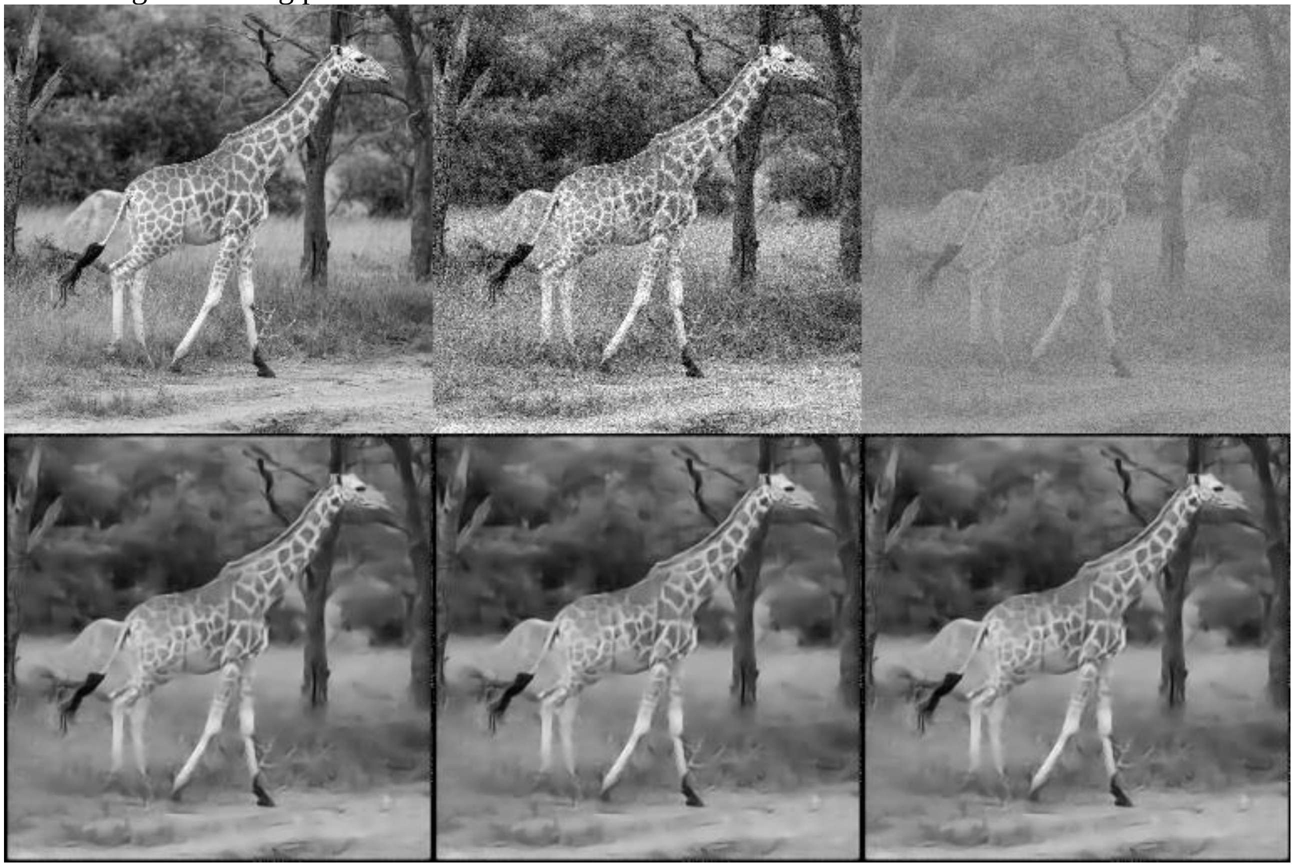
1. **Cleaner Demarcation Line**: The line that divides the upper and lower segments of the gate in the image is significantly better and more defined in the Mean Overlap k=30 compared to the original prediction k=1. This cleaner separation adds to the overall clarity and visual appeal of the image.

2. **Reduction in Artifacts**: The original prediction method displayed noticeable artifacts—distortions or anomalies in the image that detract from its quality. These artifacts are much smoother and less obtrusive in the overlap method. This smoothing effect particularly enhances the areas of the image where sharp contrasts and movements occur, like the kicking motion of the player.

3. **Smoother Details on the Player's Sleeve**: In the athlete's sleeve, the overlap technique provides a more uniform and smooth appearance. The prediction k=1 version shows some irregularities and roughness that are smoothed out in the overlap technique, resulting in a more realistic and visually pleasing depiction of the fabric and its movement.

Overall, the Mean Overlap k=30 technique demonstrates a superior ability to maintain the integrity and aesthetics of the image while reducing noise. This improvement is not only significant in enhancing the details and clarity but also in preserving the natural look of dynamic scenes, such as the athlete's action shot in this instance.

Below are the visual results from the denoising process applied to an image of a giraffe walking through a forested area. Each image demonstrates a different stage or technique in the image denoising process:


Top row (from left to right): Clean image X , Noisy image Y, Noisier image Z.
Bottom row (from left to right): Mean Overlap, Median Overlap, Trimmed Overlap.
*note that the model cannot see the clean image X.

Better than regular prediction k=1

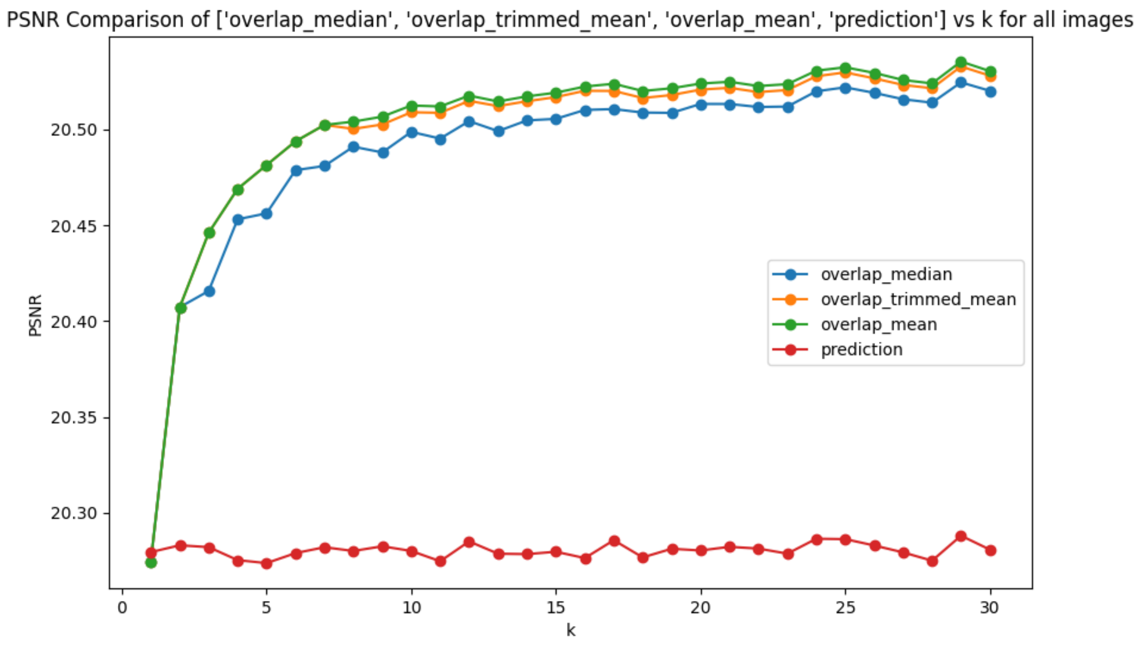
Evaluation of Potential Improvements with Reduced Noise Inputs

We investigated potential enhancements by analyzing the impact of modifying the noise distribution in the input images. This was accomplished by adjusting the parameter alpha to vary the level of added noise relative to the original noise present in the images. Additionally, we evaluated the effect of using unaugmented noisy inputs, where the noisier input was identical to the original noisy image.

As discussed at the end of **Section 3: Theoretical Background**, these techniques may be advantageous when the primary objective is to reduce the Peak Signal-to-Noise Ratio (PSNR), even if this compromises visual similarity to the clean image. However, in our experiments—which prioritize producing outputs that closely resemble the clean image—these techniques yielded suboptimal results compared to the original method presented in this project book. Consequently, the results presented below are based on the original algorithm.

PSNR Comparison of Denoising Techniques:

The graph presents a comparative analysis of the Peak Signal-to-Noise Ratio (PSNR) for various denoising techniques: 'overlap_median', 'overlap_trimmed_mean', 'overlap_mean', and a basic 'prediction' method, across a range of k values. PSNR is an essential metric in image processing that quantifies the quality of a denoised image compared to its original, with higher values indicating superior image fidelity.



1. **Significant Improvement with Overlap Techniques**:
    - All three overlap techniques show a notable improvement in PSNR compared to the 'prediction' method. This enhancement is evident from the clear contrast in PSNR values, where overlap techniques consistently maintain higher PSNR throughout the range of k values. Such improvement underscores the efficacy of overlap techniques in preserving detailed image features while effectively reducing noise.

2. **Performance Saturation Beyond k=10**:
    - The graph indicates a saturation point around k=10 for all three overlap techniques. While there is a steep improvement in PSNR as k increases from 0 to 10, the rate of improvement diminishes significantly beyond this point. This saturation suggests that increasing k further provides marginal gains in image quality, which might not justify the additional computational cost.

3. **Consistency Among Overlap Techniques**:
    - The 'overlap_median', 'overlap_trimmed_mean', and 'overlap_mean' techniques perform similarly across all k values, with minor fluctuations

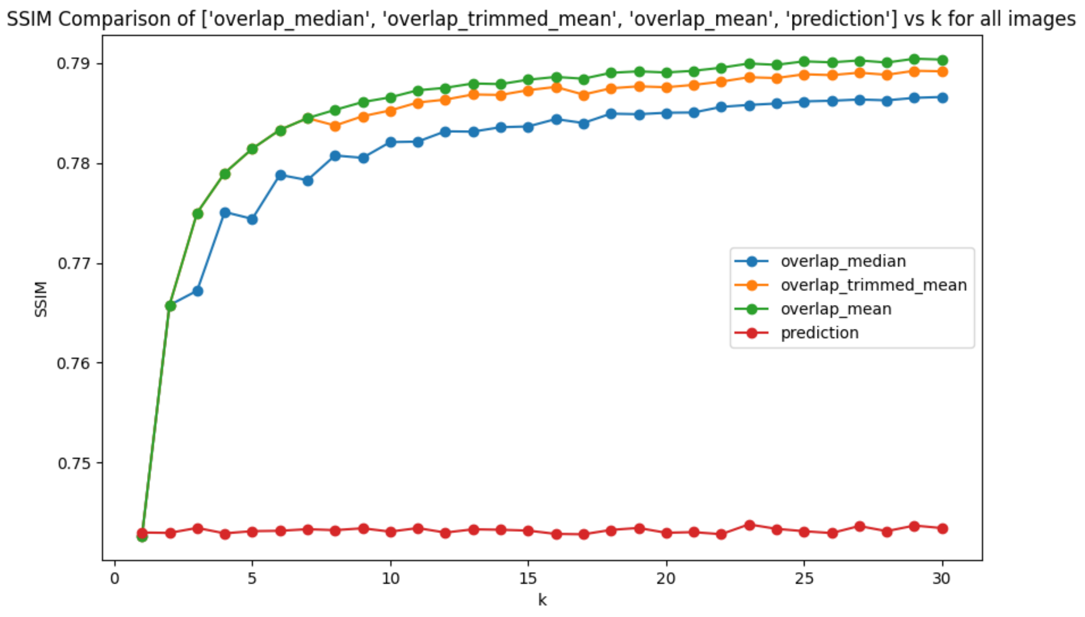
in their PSNR ratings. This consistency points to the robustness of overlap methods in denoising across different scenarios and settings.

Practical Considerations:

- **Computational Complexity**:
    - The overlap techniques involve sorting pixel values per pixel, which can be computationally intensive, especially as the number of processed images (k) increases. Additionally, managing multiple images amplifies the time and space requirements, further impacting the computational overhead.

- **Optimal k Value**:
    - Considering the trade-off between the quality of the denoised image and the computational resources required, an optimal k value would be around 10. At this point, the benefits in image quality begin to plateau, suggesting that higher k values may not be economically or technically prudent given the minimal improvements in PSNR.

SSIM Analysis of Denoising Techniques:



The SSIM graph complements the earlier PSNR results, showing that the overlap techniques ('overlap_median', 'overlap_trimmed_mean', 'overlap_mean') significantly outperform the standard 'prediction' method. Just as with PSNR, the overlap techniques quickly improve in effectiveness up to about k=10, indicating optimal performance without significant additional computational cost.

SSIM values range between -1 and 1, with 1 indicating perfect similarity to the original image. The overlap techniques approaches around 0.79 values, compare to 0.75 of k=1 prediction, signifying that they not only reduce noise effectively but also preserve the structural and visual integrity of the original image. This high similarity score underscores the efficacy of the overlap methods in maintaining the essential details and textures of the original, which is crucial for applications requiring high fidelity in image processing.

Conclusion:

The analysis based on the PSNR and SSIM values across different k values and techniques strongly supports the use of overlap methods for effective denoising. These methods not only enhance the visual quality of images by preserving intricate details but also do so more efficiently up to a certain threshold of k. Choosing the right k value is crucial to balancing image quality improvements with computational feasibility, making k=10 a practical choice for robust denoising performance.

8. Conclusion

The Noisier2Noise method represents a significant advancement in the field of image denoising, offering a practical solution for scenarios where clean or paired noisy data is unavailable. Through the course of this project, we have implemented and tested the Noisier2Noise approach, demonstrating its effectiveness across a range of noise conditions and datasets.

Summary of Findings
The core contribution of the Noisier2Noise method lies in its ability to denoise images using only a single noisy realization of each training example, combined with a statistical model of the noise distribution. This approach eliminates the need for clean training data or multiple noisy realizations, making it applicable to a broader range of real-world scenarios.

Noisier2Noise achieves competitive performance compared to traditional denoising methods, such as BM3D and Noise2Noise [2], which require richer datasets. The method performed particularly well in scenarios involving Gaussian noise, where the denoised images retained a high level of detail and exhibited minimal artifacts.

Our innovation and use of overlapping during inference further enhanced the quality of the denoised images (visually and in terms of PSNR/SSIM) by reducing boundary artifacts and improving the network's ability to capture local context.

Limitations of the Current Approach
Despite its strengths, the Noisier2Noise method has certain limitations that must be acknowledged:

1. **Noise Model Dependency**: The effectiveness of the Noisier2Noise method is heavily dependent on the accuracy of the noise model used during training. If the noise model does not represent the actual noise present in the images, the performance of the denoising algorithm may be compromised.
2. **Computational Complexity**: The overlapping technique, while beneficial for enhancing image prediction, increases the computational complexity. Processing overlapping requires more memory and computational resources, which may limit the scalability of the method to larger datasets or higher-resolution images.

Suggestions for Future Work
To address the limitations of the current approach and further enhance the Noisier2Noise method, the following areas of future work are suggested:

1. **Adaptive Noise Modeling**: Developing adaptive noise modeling techniques that can dynamically adjust to different noise types and levels during training could improve the robustness of the Noisier2Noise method. This could involve the use of more sophisticated noise estimation algorithms or the integration of noise type classification as part of the denoising pipeline.

2. **Efficiency Improvements**: To reduce the computational burden of the overlapping technique, future work could explore more efficient patch processing strategies to reduce the compiutational time of the used overlapping technique.
3. **Generalization to Diverse Noise Types**: Enhancing the network's ability to generalize to diverse noise types, possibly through the use of data augmentation techniques that simulate a wider range of noise conditions during training, could improve its performance on unseen noise types during testing.
4. **Exploring Alternative Loss Functions**: Investigating alternative loss functions that better capture the perceptual quality of denoised images, such as perceptual loss or adversarial loss, could lead to improvements in the visual quality of the denoised outputs.

Conclusion

The Noisier2Noise method represents a significant advancement in the field of image denoising, offering a powerful and versatile solution for scenarios where clean or paired noisy data is unavailable. By leveraging a single noisy image and a statistical model of the noise, Noisier2Noise can achieve denoising performance that rivals traditional methods requiring richer datasets. The use of an overlapping patch technique further enhances the method's effectiveness, ensuring high-quality denoising results with minimal artifacts.

While the method has demonstrated strong performance across various noise conditions, there are still opportunities for improvement, particularly in the areas of noise model dependency, computational efficiency, and generalization to diverse noise types.

By addressing these challenges, future research can further enhance the capabilities of the Noisier2Noise method, making it an even more effective tool for image restoration in practical settings.
As the field of image processing continues to evolve, methods like Noisier2Noise will play a crucial role in making advanced denoising techniques more accessible and widely applicable, especially in real-world applications where clean training data is scarce. The continued development and refinement of such methods will undoubtedly contribute to the broader advancement of image processing technologies.

9. References

[1] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2noise: Learning image restoration without clean data. arXiv preprint arXiv:1803.04189, 2018.

[2] Nick Moran, Dan Schmidt, Yu Zhong, and Patrick Coady. Noisier2Noise: Learning to Denoise from Unpaired Noisy Data. *arXiv preprint* arXiv:1910.11908, 2019.

[3] https://github.com/melobron/Noisier2Noise.git

[4] Kai Zhang, Wangmeng Zuo, Yunjin Chen, Deyu Meng, and Lei Zhang. Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising. arXiv preprint arXiv:1608.03981, 2016.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

10. Addendums

Code snippets, additional data, and any supplementary material that supports the project.

Train.py

```
# usage: test.py [-h] [--test_info TEST_INFO] [--gpu_num GPU_NUM] [--seed SEED] [--exp_num EXP_NUM] [--
n_epochs N_EPOCHS] [--noise NOISE]
#               [--dataset DATASET] [--exp_rep EXP_REP] [--aver_num AVER_NUM] [--alpha ALPHA] [--trim_op
TRIM_OP] [--noisy_input NOISY_INPUT]
#               [--crop CROP] [--patch_size PATCH_SIZE] [--normalize NORMALIZE] [--mean MEAN] [--std STD]

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim

import json
import random
from tqdm import tqdm

from utils import *
from models.DnCNN import DnCNN
from dataset import ImageNetGray


class TrainNr2N:
    def __init__(self, args):
        # Arguments
        self.args = args

        # Device
        self.gpu_num = args.gpu_num
        self.device = torch.device('cuda:{}'.format(self.gpu_num) if torch.cuda.is_available() else 'cpu')

        # Random Seeds
        torch.manual_seed(args.seed)
        random.seed(args.seed)
        np.random.seed(args.seed)

        # Training Parameters
        self.n_epochs = args.n_epochs
        self.start_epoch = args.start_epoch
        self.decay_epoch = args.decay_epoch
        self.lr = args.lr
        self.noise = args.noise
        self.noise_type = self.noise.split('_')[0]
        self.noise_intensity = float(self.noise.split('_')[1]) / 255.

        # Loss
        self.criterion_mse = nn.MSELoss()

        # Transformation Parameters
        self.mean = args.mean
        self.std = args.std

        # Transform
        transform = transforms.Compose(get_transforms(args))
        # args.load_model = False
        # Models
        self.model = DnCNN().to(self.device)
        if args.load_model:
            load_path = './experiments/exp{}/checkpoints/{}epochs.pth'.format(args.load_exp_num, args.load_epoch)
            self.model.load_state_dict(torch.load(load_path))

        # Dataset
        self.train_dataset = ImageNetGray(noise=self.noise, train=True, transform=transform)
        self.test_dataset = ImageNetGray(noise=self.noise, train=False, transform=transform)
        # print(self.train_dataset.clean_dir)

        self.train_dataloader = DataLoader(self.train_dataset, batch_size=args.batch_size, shuffle=True)

        # Optimizer
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr, betas=(0.5, 0.999))

        # Learning Rate Scheduler
        self.scheduler = lr_scheduler.LambdaLR(self.optimizer, lr_lambda=LambdaLR(self.n_epochs, self.start_epoch,
self.decay_epoch).step)

        # Directories
        self.exp_dir = make_exp_dir('./experiments/')['new_dir']
        self.exp_num = make_exp_dir('./experiments/')['new_dir_num']
        self.checkpoint_dir = os.path.join(self.exp_dir, 'checkpoints')
        self.result_path = os.path.join(self.exp_dir, 'results')

        # Tensorboard
        self.summary = SummaryWriter('runs/exp{}'.format(self.exp_num))

    def prepare(self):
        # Save Paths
        if not os.path.exists(self.checkpoint_dir):
            os.makedirs(self.checkpoint_dir)

        if not os.path.exists(self.result_path):
            os.makedirs(self.result_path)

        # Save Argument file
        param_file = os.path.join(self.exp_dir, 'params.json')
        with open(param_file, mode='w') as f:
            json.dump(self.args.__dict__, f, indent=4)

    def train(self):
        print(self.device)
        self.prepare()

        for epoch in range(1, self.n_epochs + 1):
            with tqdm(self.train_dataloader, desc='Epoch {}'.format(epoch)) as tepoch:
                for batch, data in enumerate(tepoch):
                    self.model.train()
                    self.optimizer.zero_grad()

                    clean, noisy, noisier = data['clean'], data['noisy'], data['noisier']
                    clean, noisy, noisier = clean.to(self.device), noisy.to(self.device), noisier.to(self.device)

                    prediction = self.model(noisier)
                    loss = self.criterion_mse(prediction, noisy)
                    loss.backward()
                    self.optimizer.step()
                    tepoch.set_postfix(rec_loss=loss.item())
                    self.summary.add_scalar('loss', loss.item(), epoch)

            self.scheduler.step()

            # Checkpoints
            if epoch % 10 == 0 or epoch == self.n_epochs:
                torch.save(self.model.state_dict(), os.path.join(self.checkpoint_dir, '{}epochs.pth'.format(epoch)))

            if epoch % 5 == 0:
                noisy_psnr, output_psnr, prediction_psnr = 0, 0, 0
                noisy_ssim, output_ssim, prediction_ssim = 0, 0, 0

                with torch.no_grad():
                    self.model.eval()

                    num_data = 10
                    for index in range(num_data):
                        data = self.test_dataset[index]
                        sample_clean, sample_noisy, sample_noisier = data['clean'], data['noisy'], data['noisier']
                        sample_noisy = torch.unsqueeze(sample_noisy, dim=0).to(self.device)
                        sample_noisier = torch.unsqueeze(sample_noisier, dim=0).to(self.device)

                        sample_output = self.model(sample_noisy)
                        sample_prediction = 2*self.model(sample_noisier) - sample_noisier

                        if self.args.normalize:
                            sample_clean = denorm(sample_clean, mean=self.mean, std=self.std)
                            sample_noisy = denorm(sample_noisy, mean=self.mean, std=self.std)
                            sample_output = denorm(sample_output, mean=self.mean, std=self.std)
                            sample_prediction = denorm(sample_prediction, mean=self.mean, std=self.std)

                        sample_clean, sample_noisy = tensor_to_numpy(sample_clean), tensor_to_numpy(sample_noisy)
                        sample_output, sample_prediction = tensor_to_numpy(sample_output),
tensor_to_numpy(sample_prediction)

                        sample_clean, sample_noisy = np.squeeze(sample_clean), np.squeeze(sample_noisy)
                        sample_output, sample_prediction = np.squeeze(sample_output), np.squeeze(sample_prediction)

                        # Calculate PSNR
                        n_psnr = psnr(sample_clean, sample_noisy, data_range=1)
                        o_psnr = psnr(sample_clean, sample_output, data_range=1)
                        p_psnr = psnr(sample_clean, sample_prediction, data_range=1)
                        # print('{}th image PSNR | noisy:{:.3f}, output:{:.3f}, prediction:{:.3f}'.format(index + 1, n_psnr,
o_psnr, p_psnr))

                        noisy_psnr += n_psnr / num_data
                        output_psnr += o_psnr / num_data
                        prediction_psnr += p_psnr / num_data

                        # Calculate SSIM
                        n_ssim = ssim(sample_clean, sample_noisy, data_range=1)
                        o_ssim = ssim(sample_clean, sample_output, data_range=1)
                        p_ssim = ssim(sample_clean, sample_prediction, data_range=1)
                        # print('{}th image SSIM | noisy:{:.3f}, output:{:.3f}, prediction:{:.3f}'.format(index + 1, n_ssim,
o_ssim, p_ssim))

                        noisy_ssim += n_ssim / num_data
                        output_ssim += o_ssim / num_data
                        prediction_ssim += p_ssim / num_data

                        # Save sample image
```

```
                        sample_clean, sample_noisy = 255. * np.clip(sample_clean, 0., 1.), 255. * np.clip(sample_noisy, 0., 1.)
                        sample_output, sample_prediction = 255. * np.clip(sample_output, 0., 1.), 255. * np.clip(sample_prediction, 0., 1.)

                        if index == 0:
                            cv2.imwrite(os.path.join(self.result_path, 'clean_{}epochs.png'.format(epoch)), sample_clean)
                            cv2.imwrite(os.path.join(self.result_path, 'noisy_{}epochs.png'.format(epoch)), sample_noisy)
                            cv2.imwrite(os.path.join(self.result_path, 'output_{}epochs.png'.format(epoch)), sample_output)
                            cv2.imwrite(os.path.join(self.result_path, 'prediction_{}epochs.png'.format(epoch)), sample_prediction)

                    # PSNR, SSIM
                    print('Average PSNR | noisy:{:.3f}, output:{:.3f}, prediction:{:.3f}'.format(noisy_psnr, output_psnr, prediction_psnr))
                    print('Average SSIM | noisy:{:.3f}, output:{:.3f}, prediction:{:.3f}'.format(noisy_ssim, output_ssim, prediction_ssim))
                    self.summary.add_scalar('avg_output_psnr', output_psnr, epoch)
                    self.summary.add_scalar('avg_output_ssim', output_ssim, epoch)
                    self.summary.add_scalar('avg_prediction_psnr', prediction_psnr, epoch)
                    self.summary.add_scalar('avg_prediction_ssim', prediction_ssim, epoch)

        self.summary.close()
```

Test.py

```
# usage: test.py [-h] [--test_info TEST_INFO] [--gpu_num GPU_NUM] [--seed SEED] [--exp_num EXP_NUM] [--n_epochs N_EPOCHS] [--noise NOISE]
#               [--dataset DATASET] [--exp_rep EXP_REP] [--aver_num AVER_NUM] [--alpha ALPHA] [--trim_op TRIM_OP] [--noisy_input NOISY_INPUT]
#               [--crop CROP] [--patch_size PATCH_SIZE] [--normalize NORMALIZE] [--mean MEAN] [--std STD]

import argparse
import random
import math
from glob import glob
import csv
import os
import torch
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim

from models.DnCNN import DnCNN
from utils import *

# Arguments
parser = argparse.ArgumentParser(description='Test Nr2N public')

parser.add_argument('--test_info', default='None info given', type=str)

parser.add_argument('--gpu_num', default=0, type=int)
parser.add_argument('--seed', default=90, type=int)
parser.add_argument('--exp_num', default=10, type=int)

# Model parameters
parser.add_argument('--n_epochs', default=180, type=int)

# Test parameters
parser.add_argument('--noise', default='gauss_25', type=str)  # 'gauss_intensity', 'poisson_intensity'
parser.add_argument('--dataset', default='Set12', type=str)  # BSD100, Kodak, Set12
parser.add_argument('--exp_rep', default=None, type=str)
parser.add_argument('--aver_num', default=10, type=int)
parser.add_argument('--alpha', default=1.0, type=float)
parser.add_argument('--trim_op', default=0.05, type=float)
parser.add_argument('--noisy_input', type=bool, default=False)

# Transformations
parser.add_argument('--crop', type=bool, default=True)
parser.add_argument('--patch_size', type=int, default=256)
parser.add_argument('--normalize', type=bool, default=True)
parser.add_argument('--mean', type=float, default=0.4097)  # ImageNet Gray: 0.4050
parser.add_argument('--std', type=float, default=0.2719)  # ImageNet Gray: 0.2927

opt = parser.parse_args()


def generate(args):
    #device = torch.device('cuda:{}'.format(args.gpu_num))
    device = torch.device('cpu')

    # Random Seeds
    torch.manual_seed(args.seed)
    random.seed(args.seed)
    np.random.seed(args.seed)

    # Model
    model = DnCNN().to(device)
    model.load_state_dict(torch.load('./experiments/exp{}/checkpoints/{}epochs.pth'.format(args.exp_num,
args.n_epochs), map_location=device))
    model.eval()

    # Directory
    img_dir  = os.path.join('../all_datasets/', args.dataset)
    save_dir = create_next_experiment_folder(os.path.join('./results/', args.dataset, 'imgs'), opt.exp_rep)

    # Images
    # Load all PNG and JPG images from the dataset directory - transform to grayscale
    img_paths = glob(os.path.join(img_dir, '*.png')) + glob(os.path.join(img_dir, '*.jpg'))
    imgs      = [cv2.imread(p, cv2.IMREAD_GRAYSCALE) for p in img_paths]

    # Noise
    noise_type = args.noise.split('_')[0]
    noise_intensity = float(args.noise.split('_')[1]) / 255.0

    # print(f'************ noise intensity from {noise_type} is: **************')
    # print(noise_intensity)

    # Transform
    transform = transforms.Compose(get_transforms(args))

    # Denoising params
    psnr_averages, ssim_averages = {}, {}

    # CSV
    csv_header = ['k', 'noisy', 'prediction', 'overlap_mean', 'overlap_median', 'overlap_trimmed_mean']
    csv_folder = create_next_experiment_folder(os.path.join('./results/', args.dataset, 'csvs'), opt.exp_rep)


    for index, clean255 in enumerate(imgs):
        if args.crop:
            clean255 = crop(clean255, patch_size=args.patch_size)

        clean_numpy = clean255/255.
        if noise_type == 'gauss':
            if not opt.noisy_input:
                noisy_numpy      = clean_numpy + np.random.randn(*clean_numpy.shape) * noise_intensity
            else:
                noisy_numpy      = clean_numpy

            noisier_numpy_single = noisy_numpy + np.random.randn(*clean_numpy.shape) * noise_intensity * args.alpha
        elif noise_type == 'poisson':
            if not opt.noisy_input:
                noisy_numpy         = np.random.poisson(clean_numpy * 255. * noise_intensity) / noise_intensity / 255.
            else:
                noisy_numpy      = clean_numpy
            noisier_numpy_single = noisy_numpy + (np.random.poisson(clean_numpy * 255. * noise_intensity) /
noise_intensity / 255. - clean_numpy)
        else:
            raise NotImplementedError('wrong type of noise')

        noisy, noisier = transform(noisy_numpy), transform(noisier_numpy_single)
        noisy, noisier = torch.unsqueeze(noisy, dim=0), torch.unsqueeze(noisier, dim=0)
        noisy, noisier = noisy.type(torch.FloatTensor).to(device), noisier.type(torch.FloatTensor).to(device)

        # Noisier Prediction
        prediction = ((1 + args.alpha ** 2) * model(noisier) - noisier) / (args.alpha ** 2)

        # Overlap Prediction
        noisier = torch.zeros(size=(args.aver_num, 1, *clean_numpy.shape))

        for i in range(args.aver_num):
            if noise_type == 'gauss':
                noisier_numpy = noisy_numpy + np.random.randn(*clean_numpy.shape) * noise_intensity * args.alpha
            elif noise_type == 'poisson':
                noisier_numpy = noisy_numpy + (np.random.poisson(clean_numpy * 255. * noise_intensity) /
noise_intensity / 255. - clean_numpy)
            else:
                raise NotImplementedError('wrong type of noise')

            noisier_tensor      = transform(noisier_numpy)
            noisier_tensor      = torch.unsqueeze(noisier_tensor, dim=0)
            noisier[i, :, :, :] = noisier_tensor

        noisier = noisier.type(torch.FloatTensor).to(device)

        # Calculate overlap using mean and median
        overlap_prediction = ((1 + args.alpha ** 2) * model(noisier) - noisier) / (args.alpha ** 2)

        overlap_mean      = torch.mean(overlap_prediction, dim=0)
        overlap_median, _ = torch.median(overlap_prediction, dim=0)

        # Trimmed mean per pixel calculation - TODO: check if the sort is by pixel (not by image)
        sorted_overlap, _ = torch.sort(overlap_prediction, dim=0)
        trim_percent = args.trim_op  # 10% trimming by default
        num_to_trim = math.floor(trim_percent * sorted_overlap.size(0))

        # Ensure that trimming does not empty the tensor
        if num_to_trim == 0 or 2 * num_to_trim >= sorted_overlap.size(0):
            print(f"Trimming would result in an empty tensor or no trimming possible. Skipping trimming for this image.")
            overlap_trimmed_mean = torch.mean(sorted_overlap, dim=0)  # No trimming applied
        else:
            trimmed_overlap = sorted_overlap[num_to_trim:-num_to_trim, :, :, :]
            overlap_trimmed_mean = torch.mean(trimmed_overlap, dim=0)  # Mean across the batch dimension after
trimming
            print(f"Trimmed tensor shape: {trimmed_overlap.size()}")
            print(f"Calculated overlap_trimmed_mean: {overlap_trimmed_mean}")


        # Change to Numpy
        if args.normalize:
            prediction      = denorm(prediction, mean=args.mean, std=args.std)
            overlap_mean    = denorm(overlap_mean, mean=args.mean, std=args.std)
            overlap_median  = denorm(overlap_median, mean=args.mean, std=args.std)
            overlap_trimmed = denorm(overlap_trimmed_mean, mean=args.mean, std=args.std)
            noisier         = denorm(noisier_numpy_single, mean=args.mean, std=args.std)  # Add denormalization for
noisier (the single noisier of the predict, not overlap)

        prediction, overlap_mean, overlap_median, overlap_trimmed                          = tensor_to_numpy(prediction),
tensor_to_numpy(overlap_mean), tensor_to_numpy(overlap_median), tensor_to_numpy(overlap_trimmed)
        prediction_numpy, overlap_mean_numpy, overlap_median_numpy, overlap_trimmed_numpy  =
np.squeeze(prediction), np.squeeze(overlap_mean), np.squeeze(overlap_median), np.squeeze(overlap_trimmed)
        noisier_numpy                                                                      = np.squeeze(noisier)  # Convert noisier tensor to
numpy and squeeze

        image_metrics = calculate_metrics(clean_numpy, noisy_numpy, prediction_numpy, overlap_mean_numpy,
overlap_median_numpy, noisier_numpy, overlap_trimmed_numpy)

        for metric_type in ['psnr', 'ssim']:
            averages = psnr_averages if metric_type == 'psnr' else ssim_averages
            for key, value in image_metrics[metric_type].items():
                if key in averages:
                    averages[key] += value / len(imgs)
                else:
                    averages[key] = value / len(imgs)

        # Log PSNR and SSIM values
        print('{}th image | PSNR: noisy:{:.3f}, prediction:{:.3f}, overlap_mean:{:.3f}, overlap_median:{:.3f},
overlap_trimmed_mean:{:.3f}, noisier:{:.3f} | SSIM: noisy:{:.3f}, prediction:{:.3f}, overlap_mean:{:.3f},
overlap_median:{:.3f}, overlap_trimmed_mean:{:.3f}, noisier:{:.3f}'.format(
                index + 1, image_metrics['psnr']['noisy'], image_metrics['psnr']['prediction'],
image_metrics['psnr']['overlap_mean'],
                image_metrics['psnr']['overlap_median'], image_metrics['psnr']['overlap_trim'],
image_metrics['psnr']['noisier'], image_metrics['ssim']['noisy'],
                image_metrics['ssim']['prediction'], image_metrics['ssim']['overlap_mean'],
                image_metrics['ssim']['overlap_median'], image_metrics['ssim']['overlap_trim'],
image_metrics['ssim']['noisier']))

        # write on SCV per image SSIM and PSNR
        # Assuming img_paths[index] is the image file path with an extension
        file_name = os.path.splitext(os.path.basename(img_paths[index]))[0]  # Extracts the base file name without
extension

        file_path = f'{csv_folder}PSNR_{index}_{file_name}.csv'
        csv_data  = [args.aver_num ,image_metrics['psnr']['noisy'], image_metrics['psnr']['prediction'],
                    image_metrics['psnr']['overlap_mean'], image_metrics['psnr']['overlap_median'],
image_metrics['psnr']['overlap_trim']]

        write_csv(file_path, csv_data, csv_header)

        file_path = f'{csv_folder}SSIM_{index}_{file_name}.csv'
        csv_data  = [args.aver_num, image_metrics['ssim']['noisy'], image_metrics['ssim']['prediction'],
                    image_metrics['ssim']['overlap_mean'], image_metrics['ssim']['overlap_median'],
image_metrics['ssim']['overlap_trim']]

        write_csv(file_path, csv_data, csv_header)


        # Save sample images (up to 10 images)
        if index <= 10:
            sample_clean, sample_noisy  = 255. * np.clip(clean_numpy, 0., 1.), 255. * np.clip(noisy_numpy, 0., 1.)
            sample_prediction           = 255. * np.clip(prediction_numpy, 0., 1.)
            sample_overlap_mean         = 255. * np.clip(overlap_mean_numpy, 0., 1.)
            sample_overlap_median       = 255. * np.clip(overlap_median_numpy, 0., 1.)
            sample_overlap_trimmed      = 255. * np.clip(overlap_trimmed_numpy, 0., 1.)
            sample_noisier              = 255. * np.clip(noisier_numpy, 0., 1.)  # Prepare noisier image for saving

            cv2.imwrite(os.path.join(save_dir, '{}th_clean.png'.format(index+1)), sample_clean)
            cv2.imwrite(os.path.join(save_dir, '{}th_noisy.png'.format(index+1)), sample_noisy)
            cv2.imwrite(os.path.join(save_dir, '{}th_prediction.png'.format(index+1)), sample_prediction)
            cv2.imwrite(os.path.join(save_dir, '{}th_overlap_mean.png'.format(index+1)), sample_overlap_mean)
            cv2.imwrite(os.path.join(save_dir, '{}th_overlap_median.png'.format(index+1)), sample_overlap_median)
            cv2.imwrite(os.path.join(save_dir, '{}th_overlap_trimmed.png'.format(index+1)), sample_overlap_trimmed)
            cv2.imwrite(os.path.join(save_dir, '{}th_noisier.png'.format(index + 1)),sample_noisier)  # Save noisier image

    # Total PSNR, SSIM
    print('{} Average PSNR | noisy:{:.3f}, prediction:{:.3f}, overlap_mean:{:.3f}, overlap_median:{:.3f},
overlap_trimmed_mean:{:.3f}'.format(
        args.dataset, psnr_averages['noisy'], psnr_averages['prediction'], psnr_averages['overlap_mean'],
psnr_averages['overlap_median'], psnr_averages['noisier'], psnr_averages['overlap_trim']))
    print('{} Average SSIM | noisy:{:.3f},  prediction:{:.3f}, overlap_mean:{:.3f}, overlap_median:{:.3f},
overlap_trimmed_mean:{:.3f}'.format(
        args.dataset, ssim_averages['noisy'], ssim_averages['prediction'], ssim_averages['overlap_mean'],
ssim_averages['overlap_median'], ssim_averages['noisier'], ssim_averages['overlap_trim']))

    # write average PSNR per k
    file_path = f'{csv_folder}PSNR_all_images_average.csv'
    csv_data  = [args.aver_num, psnr_averages['noisy'], psnr_averages['prediction'], psnr_averages['overlap_mean'],
psnr_averages['overlap_median'], psnr_averages['overlap_trim']]
    write_csv(file_path, csv_data, csv_header)

    # write average SSIM per k
    file_path = f'{csv_folder}SSIM_all_images_average.csv'
    csv_data  = [args.aver_num, ssim_averages['noisy'], ssim_averages['prediction'], ssim_averages['overlap_mean'],
ssim_averages['overlap_median'], ssim_averages['overlap_trim']]
    write_csv(file_path, csv_data, csv_header)



    ###### UTILS FUNCTIONS #######
def create_next_experiment_folder(base_folder, exp_repeated = None):
    """
    Creates the next available experiment folder in a base folder.
    The folder is named with an incremental number (e.g., exp1, exp2, etc.).

    Parameters:
    base_folder (str): The base directory where experiment folders are created.

    Returns:
    str: The path to the newly created experiment folder.
    """
    if exp_repeated:
        return os.path.join(base_folder, f'{exp_repeated}/')

    # Ensure the base folder exists
    if not os.path.exists(base_folder):
        os.makedirs(base_folder)

    # Find the next available folder number
    exp_num = 1
    while os.path.exists(f'{base_folder}/exp{exp_num}'):
        exp_num += 1

    # Create the new folder
    new_folder = f'{base_folder}/exp{exp_num}/'
    os.makedirs(new_folder)

    if opt.test_info:
        with open(new_folder + 'info.txt', 'w') as file:
            # Write test_info to the file
            file.write(opt.test_info + "\n")
            file.write(str(vars(opt)))


    return new_folder


def write_csv(file_path, data, header):
    """
    Appends data to a CSV file, creating the file with a header if it doesn't exist.

    Parameters:
    file_path (str): Path to the CSV file.
    data (list): A list of data to write as a new row.
    header (list): A list representing the header of the CSV file.
    """
    file_exists = os.path.isfile(file_path)

    with open(file_path, mode='a', newline='') as file:
        writer = csv.writer(file)
        if not file_exists:
            writer.writerow(header)
        writer.writerow(data)


def calculate_metrics(clean_numpy, noisy_numpy, prediction_numpy, overlap_mean_numpy, overlap_median_numpy,
noisier_numpy, overlap_trimmed_numpy):
    """
    Calculates PSNR and SSIM metrics for various stages of image processing.

    Parameters:
    clean_numpy (numpy.ndarray): The clean image array.
    noisy_numpy (numpy.ndarray): The noisy image array.
    prediction_numpy (numpy.ndarray): The predicted image array.
    overlap_mean_numpy (numpy.ndarray): The image after mean overlap.
    overlap_median_numpy (numpy.ndarray): The image after median overlap.
    noisier_numpy (numpy.ndarray): The noisier version of the image array.

    Returns:
    dict: A dictionary containing PSNR and SSIM metrics for each image stage.
    """
    metrics = {
        'psnr': {
            'noisy'         : psnr(clean_numpy, noisy_numpy, data_range=1),
            'prediction'    : psnr(clean_numpy, prediction_numpy, data_range=1),
            'overlap_mean'  : psnr(clean_numpy, overlap_mean_numpy, data_range=1),
            'overlap_median': psnr(clean_numpy, overlap_median_numpy, data_range=1),
            'overlap_trim'  : psnr(clean_numpy, overlap_trimmed_numpy, data_range=1),
            'noisier'       : psnr(clean_numpy, noisier_numpy, data_range=1)
        },
        'ssim': {
            'noisy'         : ssim(clean_numpy, noisy_numpy, data_range=1),
            'prediction'    : ssim(clean_numpy, prediction_numpy, data_range=1),
            'overlap_mean'  : ssim(clean_numpy, overlap_mean_numpy, data_range=1),
            'overlap_median': ssim(clean_numpy, overlap_median_numpy, data_range=1),
            'overlap_trim'  : ssim(clean_numpy, overlap_trimmed_numpy, data_range=1),
            'noisier'       : ssim(clean_numpy, noisier_numpy, data_range=1)
        }
    }
    return metrics


if __name__ == "__main__":
    generate(opt)
```