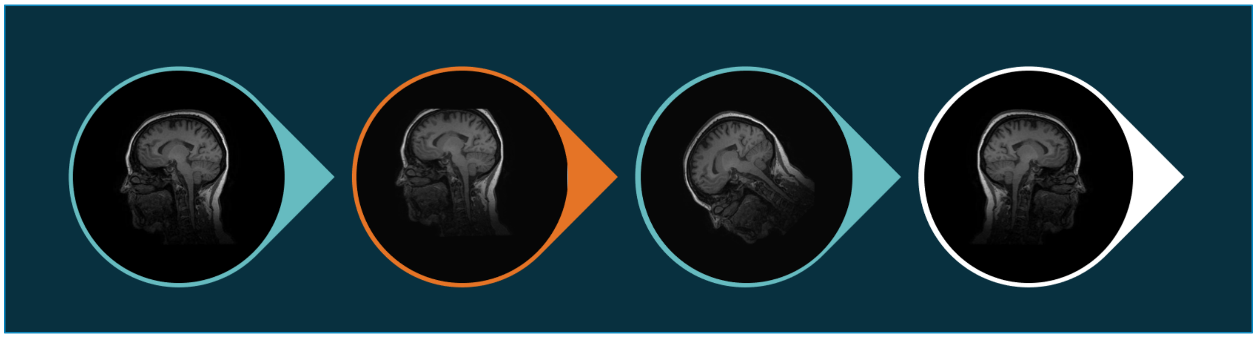
IMAGE AUGMENTATION

Medical Imaging



Name	PSID	Dpmt	Degree	E mail
Likhitha Mandapati	2205694	CS	Graduate	lmandapa@cougarnet.uh.edu
Eswara Poosapati	2183722	CS	Graduate	epoosapa@cougarnet.uh.edu

Introduction

In this project, we introduce a comprehensive system for augmenting medical imaging data, specifically focusing on the diversification of Magnetic Resonance Imaging (MRI) datasets. Utilizing Python and a suite of specialized libraries, we developed a series of algorithms to apply various augmentation techniques, including Gaussian noise, salt-and-pepper noise, and k-space truncation. The project also features an organized approach to processing and visualizing the augmented images. These techniques were rigorously applied to a selected set of MRI images, and the resulting augmented dataset was meticulously analyzed. The findings from this augmentation process are presented, demonstrating the potential of these methods to significantly enhance the quality and variability of medical imaging data for improved machine learning model training and analysis.

Objectives

- **Augmenting High-Quality images**: Utilize a custom augmentation function to increase the number of high-quality MRI samples from the OASIS-1 dataset. The augmentation process involves a series of transformations, applied to each image multiple times to generate diverse variants.

- **Simulate Low-Quality Images**: For each augmented high-quality image, create a corresponding low-quality version.

- **K-Space Truncation**: Apply k-space truncation to the MRI images using two distinct truncation factors. This process will simulate the effect of reduced resolution in MRI scans, enabling a comparative study of how different levels of truncation impact image detail and quality.

- **Noise Addition Techniques**:
    - **Gaussian Noise Addition**: Introduce Gaussian noise to the low quality samples.
    - **Salt and Pepper Noise Addition**: Apply salt-and-pepper noise to the low quality images.

- **Comparative Analysis of Augmentation Impacts**:
    - **Analyze Effects of Different Noise Types**: Evaluate and compare the impact of both Gaussian and salt-and-pepper noise on the augmented high-quality and simulated low-quality MRI images.
    - **Assessment of Image Quality**: Examine the overall effect of the applied augmentation techniques, including the image augmentation process, noise addition, and k-space truncation, on the fidelity and diagnostic utility of the MRI images.

Algorithm

1. **Initialization:**
    a. Import necessary libraries (os, numpy, PIL, matplotlib, imgaug).
    b. Define paths for the image folder and output folder.
2. **Image Loading and Preprocessing:**
    a. Load all images from the specified folder, converting them to grayscale and normalizing pixel values.
    b. Store original images in a list.
3. **Define Image Augmentation Function:**
    a. **augment_image**: Function to augment an image using imgaug library. This includes affine transformations, flipping, and Gaussian blurring, with a parameter to control the number of augmentations.
4. **Define K-Space Truncation Function:**
    a. **k_space_truncation**: Function to perform k-space truncation on an image, with a parameter for truncation factor.
5. **Define Noise Addition Functions:**
    a. **add_gaussian_noise**: Function to add Gaussian noise to an image, with parameters for mean and sigma.
    b. **add_salt_and_pepper_noise**: Function to add salt and pepper noise to an image, with parameters for noise ratio and amount.
6. **Apply Augmentation and Generate Variants:**
    For each original image:
    a. Generate augmented images using the **augment_image** function.

   b. For each augmented image, create low-quality and noisy versions: - Apply k-space truncation. - Add Gaussian noise. - Add salt-and-pepper noise.
   c. Store all variants (augmented, low-quality, Gaussian noisy, salt-and-pepper noisy) in respective lists.

7. **Save and Organize Output Images:**
   a. Create an output directory if it does not exist.
   b. Save each variant of the processed images in the output directory with appropriate naming conventions.
8. **Visualization:**
   a. Display all variants of the images in a structured layout using matplotlib, showing original, augmented, and noisy versions.
   b. Adjust figure layout for clear visualization.

Methodology

1. Image Loading and Preprocessing:

We start the process by importing essential libraries such as NumPy for numerical operations, Pillow (PIL) for image processing, Matplotlib for visualization, OS for filesystem operations, imgaug.augmenters as iaa for advanced image augmentation techniques , and scipy.fft for Fast Fourier Transform (FFT) operations..

Reading Original Images:

We then proceed to read the original images from a specified folder (*images_folder*) using the Pillow library. The images are converted to NumPy arrays and normalized to ensure consistency and compatibility with subsequent processing steps.

2. Image Augmentation:

Image augmentation's primary purpose is to enhance the diversity of a dataset by applying various transformations to the original images. We utilized the `imgaug` library, specifically importing augmenters as `iaa`. This library provides a rich set of tools for augmenting images, enabling the creation of variations that mimic real-world conditions.

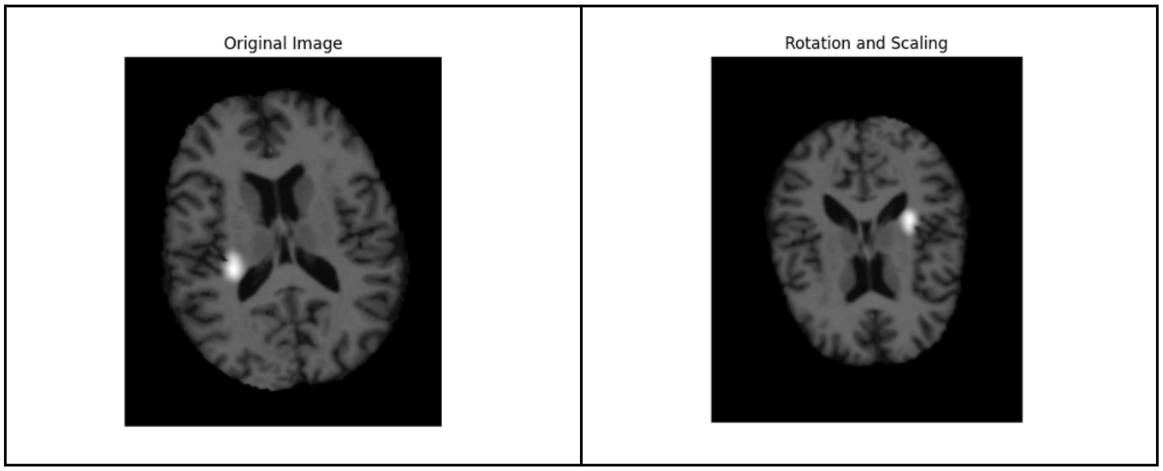
Augmentations include rotation (-10 to 10 degrees), scaling (0.8 to 1.2), horizontal flipping (50% probability), vertical flipping (50% probability), and Gaussian blur (sigma from 0 to 0.5).

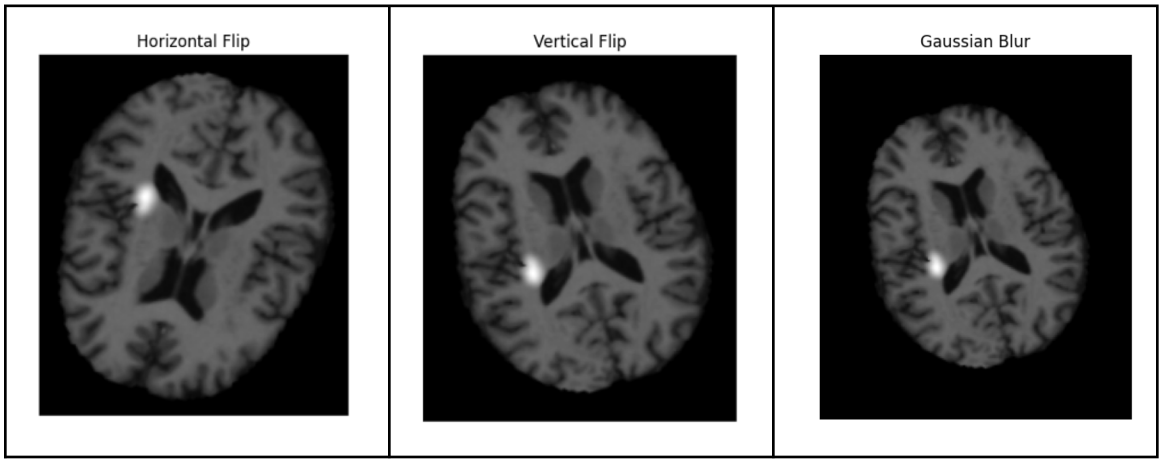
Methods used:

1. Affine Transformations:
   Affine transformations such as rotation and scaling are applied to simulate changes in orientation and size. These transformations help the model recognize objects from different perspectives and scales.

2. Flipping:
   Horizontal and vertical flipping introduces variations in the orientation of objects within the images. This is valuable for training models that need to handle images in different orientations.

3. Gaussian Blur:
   Gaussian blur adds a smoothing effect to the images, simulating situations where the camera or imaging system may introduce a degree of blurriness. This helps the model become more robust to variations in image quality.



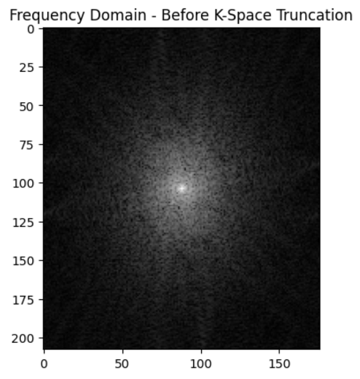


Flexibility for Further Augmentations:
We have designed this to be extensible, allowing users to add more augmentations to the sequence as needed. This flexibility enables customization based on the specific requirements of the dataset and the goals of the machine learning task.
The final result is a collection of augmented images, each with a unique combination of transformations. These augmented images are later used for further processing, including the generation of low-quality and noisy variants.

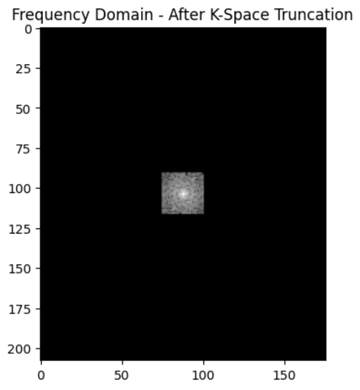
3. K-Space Truncation:
K-space truncation is a technique inspired by Fourier analysis, where the high-frequency components are removed from the image's frequency domain representation. This simulates the effect of acquiring a low-resolution image, as high-frequency details are typically lost in low-quality images.

Implementation:

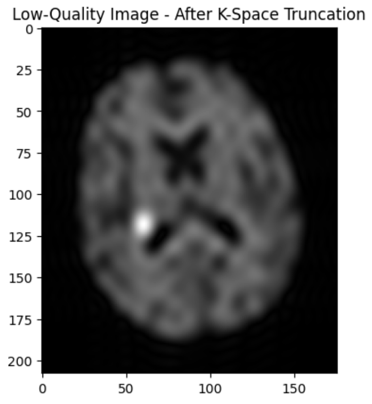
1. The script utilizes the fast Fourier transform (FFT) to convert the image its frequency domain.



2. A mask is created, setting a central region to 1 and the remaining areas to 0, effectively truncating high frequencies.

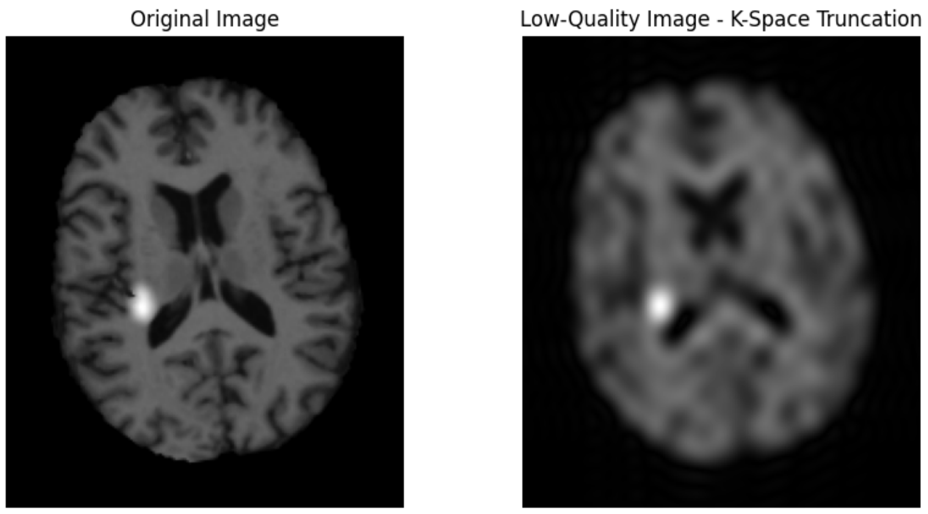


3. The inverse FFT is then applied to ob the low-quality image in the spatial domain.



Outcome:

- The result is a low-quality version of the input image, mimicking the appearance of images obtained with lower-resolution equipment.



Truncation Factor:

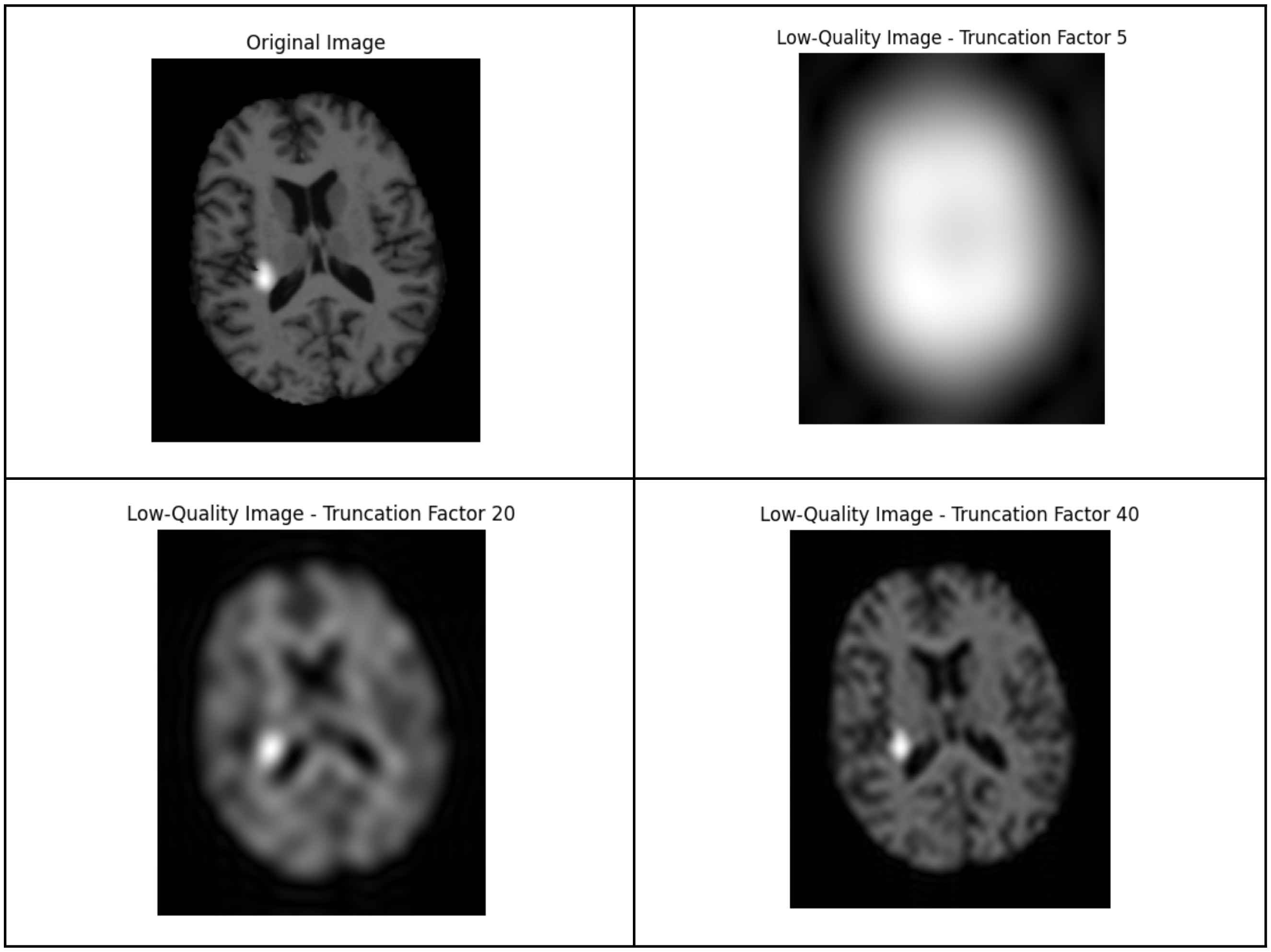
The truncation factor is a crucial parameter when applying K-space truncation in image processing, especially in the context of simulating low-quality images.. Here's an explanation:

Importance of Truncation Factor:

1. Simulating Low-Resolution:
    a. The truncation factor determines the size of the central region of the frequency domain that is retained, effectively removing high-frequency components.
    b. Larger truncation factors result in a broader removal of high-frequency information, simulating the acquisition of lower-resolution images.

2. Control Over Image Quality:
    a. Adjusting the truncation factor provides a way to control the level of degradation introduced into the image.
    b. A smaller truncation factor retains more high-frequency details, resulting in higher image quality.
    c. A larger truncation factor leads to a more pronounced loss of fine details, mimicking the appearance of images obtained with lower-resolution equipment.

3. Effect on Noise:
    a. Higher truncation factors can also affect the visibility of noise in the image.
    b. Removing high-frequency components may reduce the prominence of high-frequency noise, contributing to a smoother appearance.


Original Image
Low-Quality Image - Truncation Factor 5
Low-Quality Image - Truncation Factor 20
Low-Quality Image - Truncation Factor 40

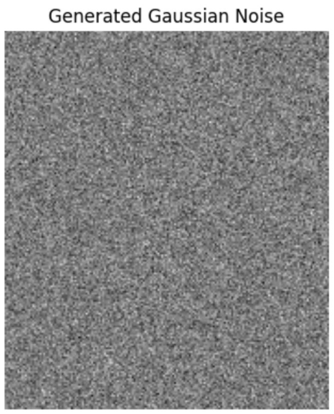
In summary, the truncation factor is a versatile parameter that allows practitioners to control the level of degradation introduced during K-space truncation. Its careful adjustment is essential for tailoring the image processing to specific tasks, simulation scenarios, and dataset requirements.

4. Addition of Gaussian Noise:

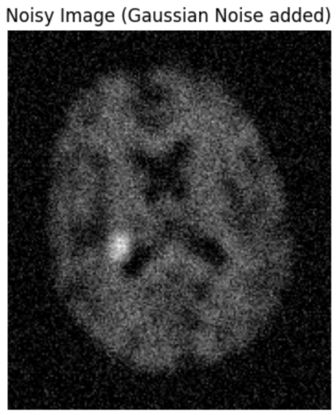
Gaussian noise is a form of random noise that is added to images to simulate the inherent variability present in real-world data. This type of noise often occurs due to imperfections in imaging sensors or environmental factors.

Implementation:

1. Randomness with Normal Distribution:
   a. Gaussian noise is generated using the numpy.random.normal function, which sample random values from a normal distribution.
   b. The normal distribution is characterized by a mean(typically 0) and a standard deviation, determining the spread of the generated value

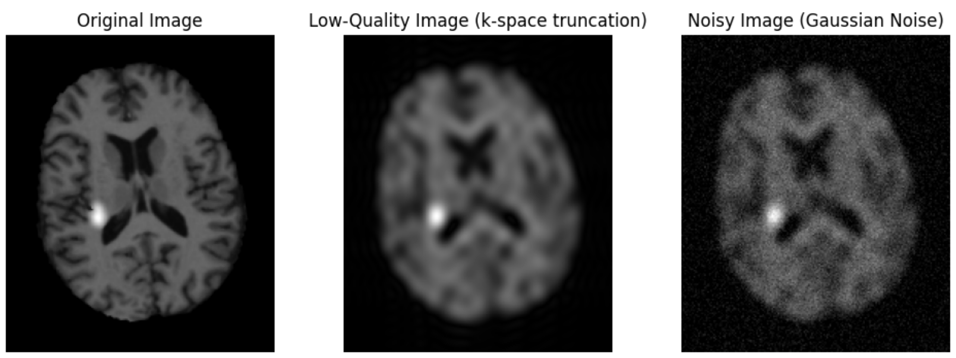


2. Addition to Low-Quality Image:
   c. The generated Gaussian noise is added to the low-quality image, pixel by pixel.
   d. This addition injects a random variation into each pixel value, mimicking unpredictable nature of noise in real-world images.



3. Pixel Value Clipping:
   e. To ensure that the resulting pixel values remain within the valid range of [0, 1], the numpy.clip function is employed.
   f. This prevents pixel values from exceeding the valid range, maintaining the integrity of the image data.

Outcome:
The result is a low-quality image with added Gaussian noise, contributing to the dataset's variability and robustness.


Original Image
Low-Quality Image (k-space truncation)
Noisy Image (Gaussian Noise)

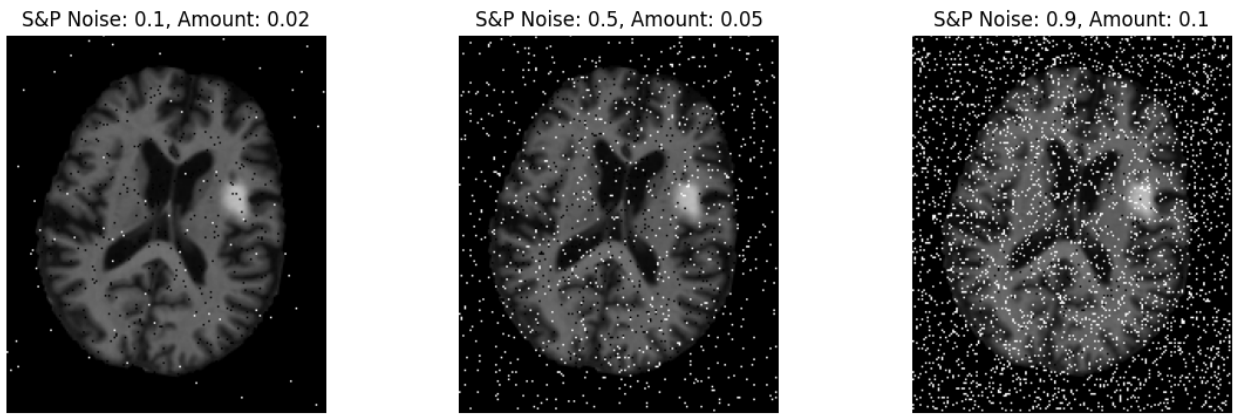
5. Salt-and-Pepper Noise

Salt-and-pepper noise is introduced to simulate anomalies or imperfections that can occur during the image acquisition process. These imperfections manifest as randomly occurring white and black pixels, akin to pixel dropout or spikes in the image.

Implementation:

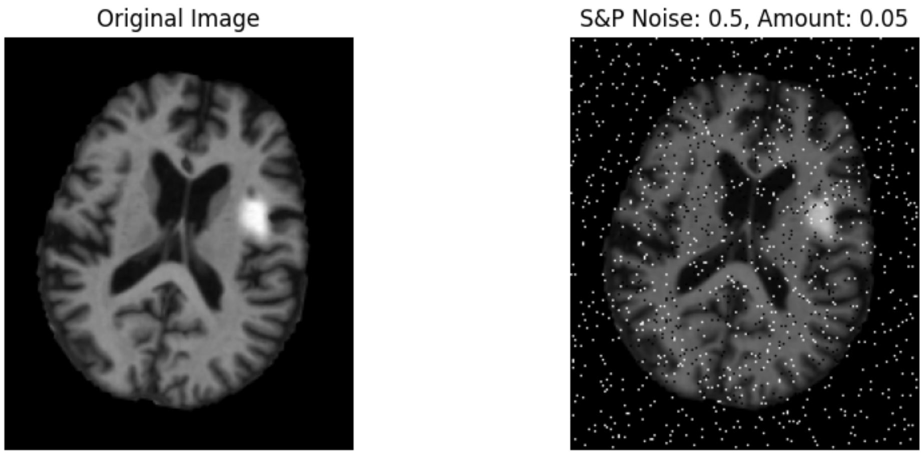
1. Random Pixel Selection:
   a. Salt-and-pepper noise is implemented by randomly selecting pixel coordinates in the image.
   b. For each selected pixel, its value is set to either 0 (black) or 1 (white), representing the addition of salt (white) or pepper (black) to the image.

2. Controlled by Parameter:
   a. The degree of noise introduced is controlled by the `salt_pepper_amount` parameter.
   b. This parameter determines the proportion of pixels in the image that will be affected by salt-and-pepper noise.


S&P Noise: 0.1, Amount: 0.02
S&P Noise: 0.5, Amount: 0.05
S&P Noise: 0.9, Amount: 0.1

Outcome:

The result is a low-quality image with added salt-and-pepper noise. This contributes to the diversity of the dataset by introducing random variations, making the dataset more representative of real-world scenarios.



6. Saving Processed Images:

The processed images are saved in an organized manner within an output folder, allowing for easy retrieval and reference. The output folder is specified as `output_folder` in the script, and it is created if it doesn't exist.

File Naming Convention:

Each processed image is assigned a filename that indicates its type (original, augmented, low-quality, gaussian noisy, salt-and-pepper noisy) and its sequence number.

For example, filenames are structured as `original_image_1.png`, `augmented_image_1_1.png`, `low_quality_image_1_1.png`, and so on.

Image Saving Process:
The `Image.fromarray()` method from the Pillow library is used to convert NumPy arrays back to image format for saving. Images are saved in the PNG format, and the pixel values are appropriately scaled back to the range [0, 255] for compatibility.

Iterating Over Processed Images:
The script iterates through the list of original, augmented, low-quality, gaussian noisy, and salt-and-pepper noisy images.

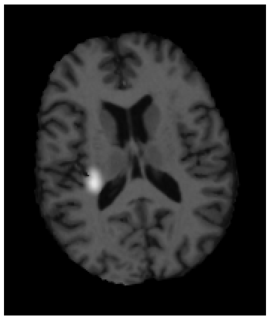
Nearest Neighbour Interpolation:
For each of these processed images, nearest neighbor interpolation is applied during the saving process. During nearest neighbor interpolation, the algorithm determines the color value of each pixel in the destination image by selecting the color value of the nearest pixel from the source image.
This means that when resizing the processed images, the algorithm does not interpolate between pixel values. Each pixel in the resized image takes on the value of the nearest pixel in the original processed image.

Output Folder Content:
After script execution, the output folder contains a comprehensive set of images that represent various aspects of the dataset. Users can easily navigate the output folder to inspect and utilize the processed images for further analysis, training, or evaluation.

This organized storage facilitates subsequent steps such as model training or further analysis, making it easy for users to manage and explore the processed dataset.

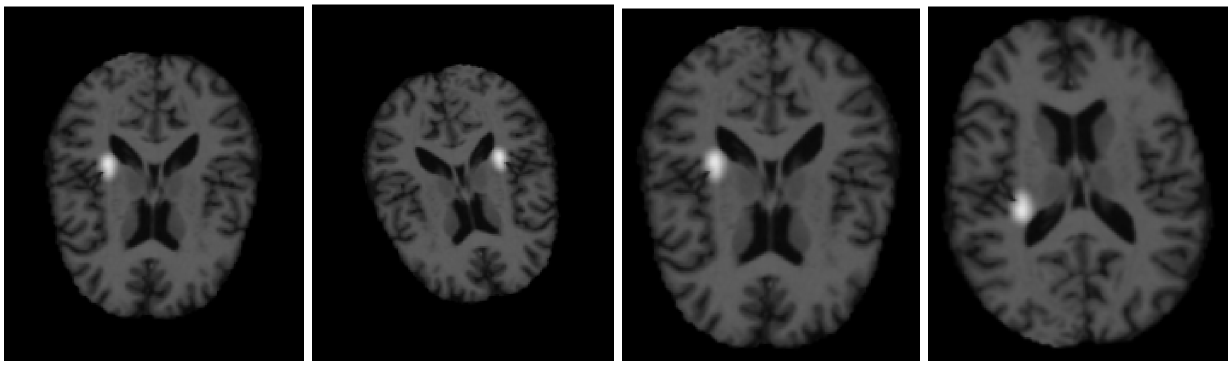
Results

Input 1:

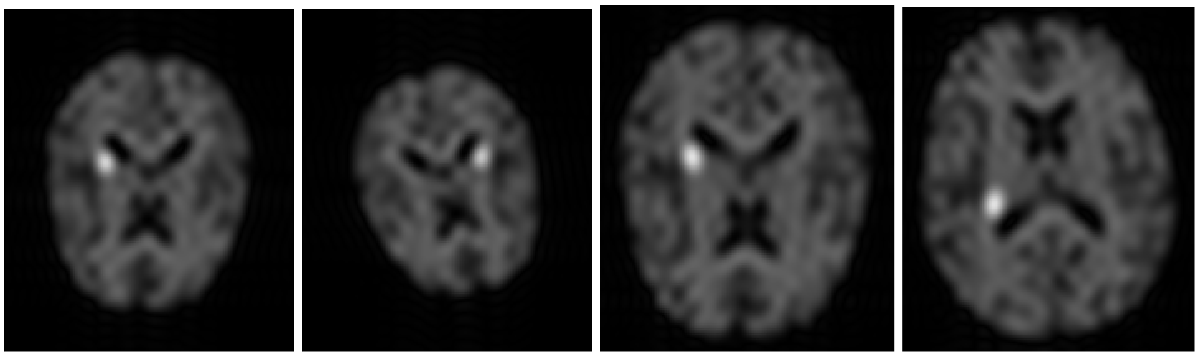


Output:

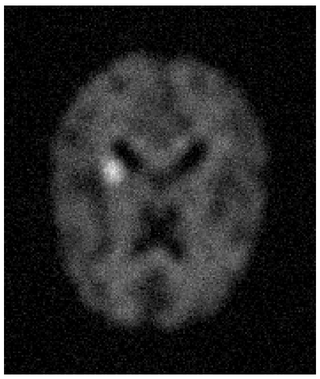
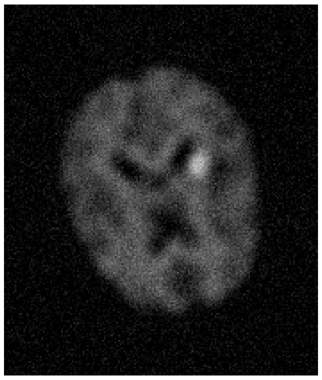
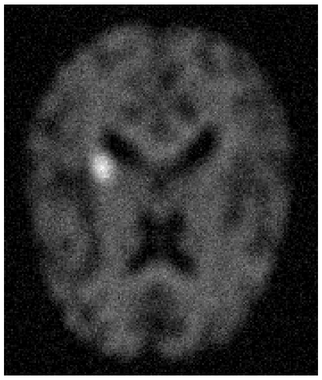
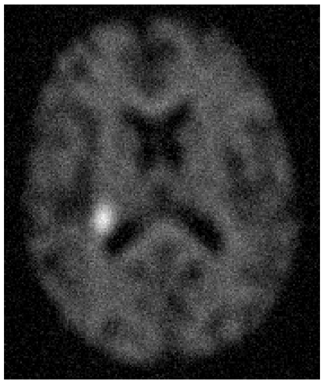
Augmented Images



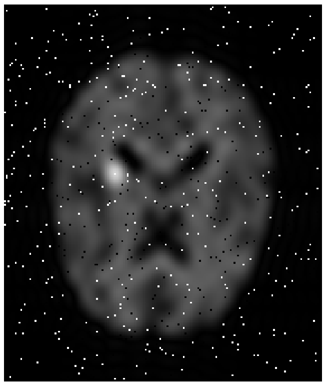
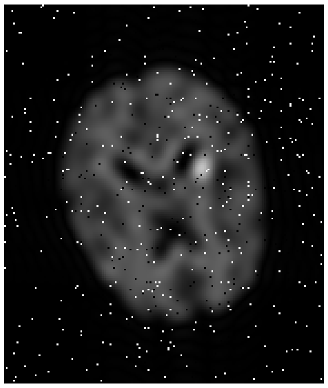
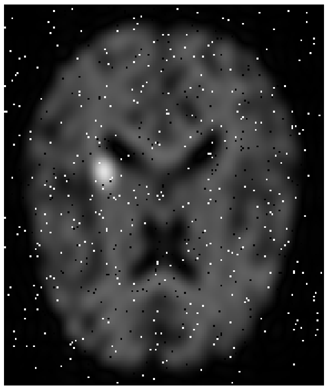
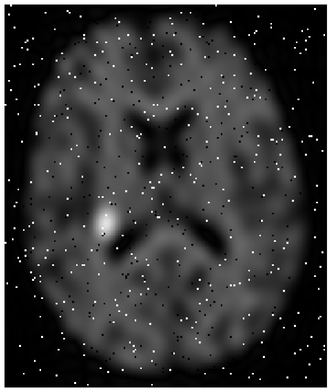
Low quality Images



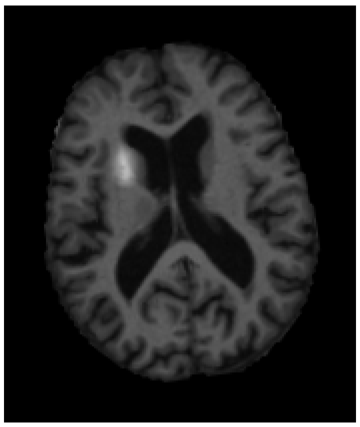
<u>Gaussian Noise</u>

<u>Salt and Pepper Noise</u>

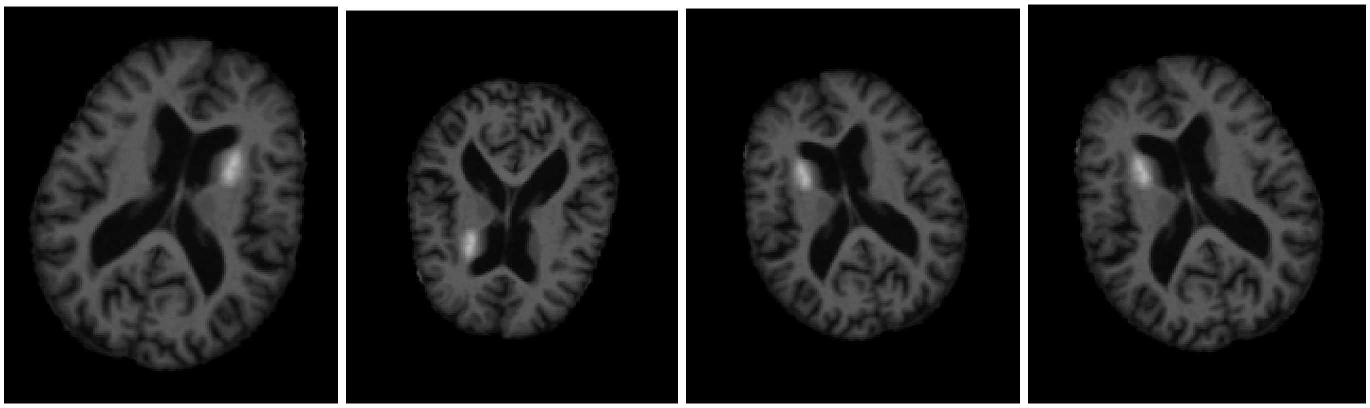





Input 2:

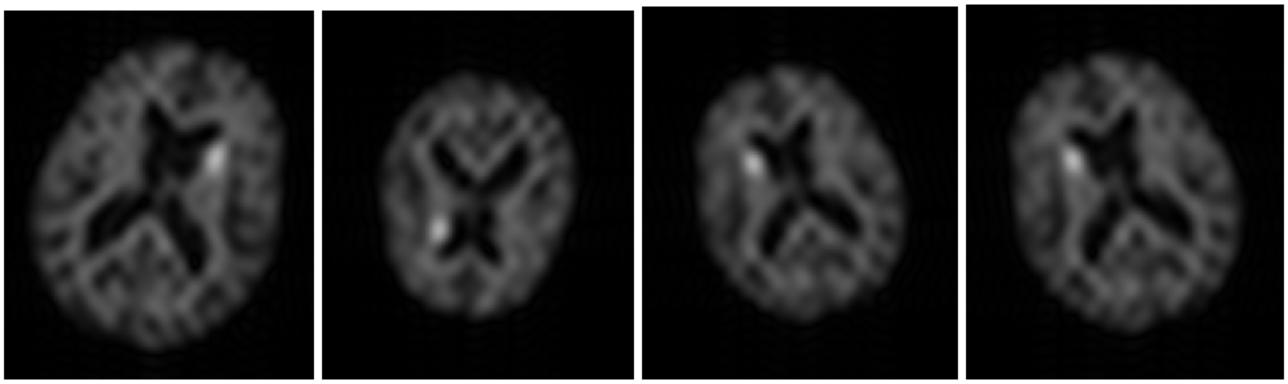


Output:

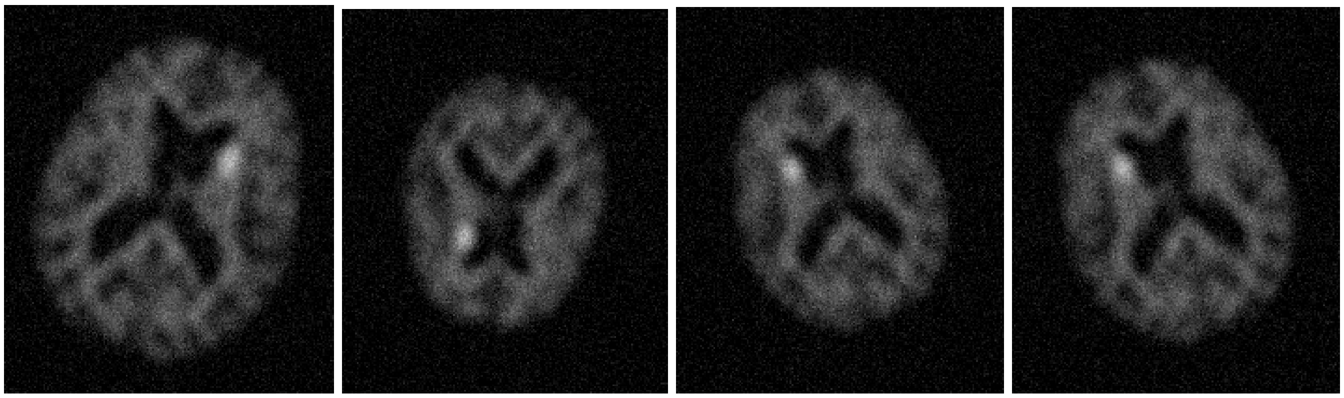
Augmented Images



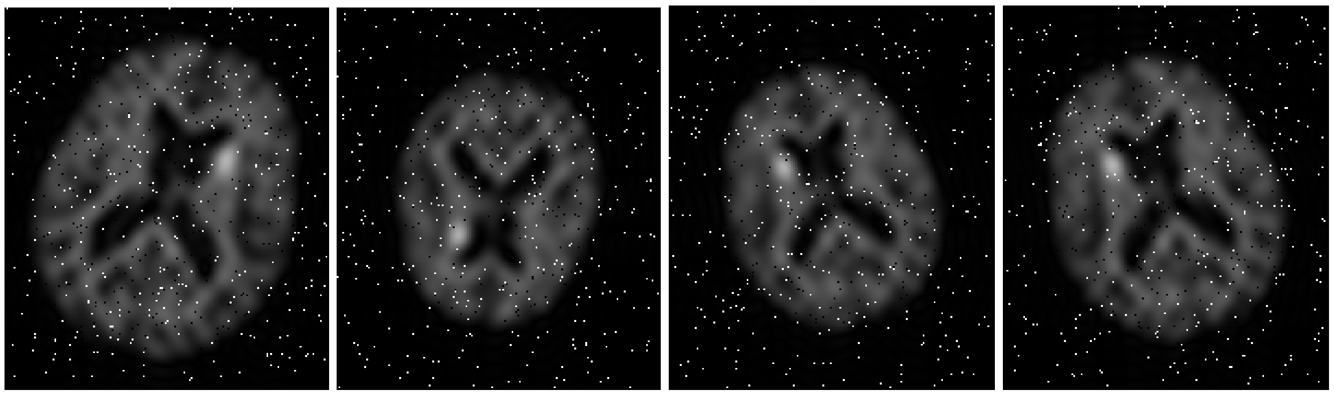
Low quality Images



Gaussian Noise



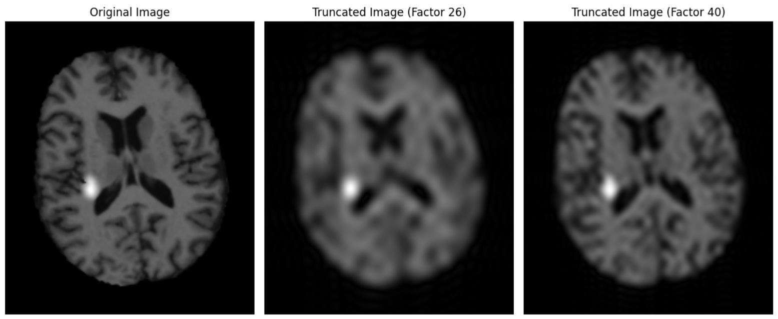
Salt and Pepper Noise

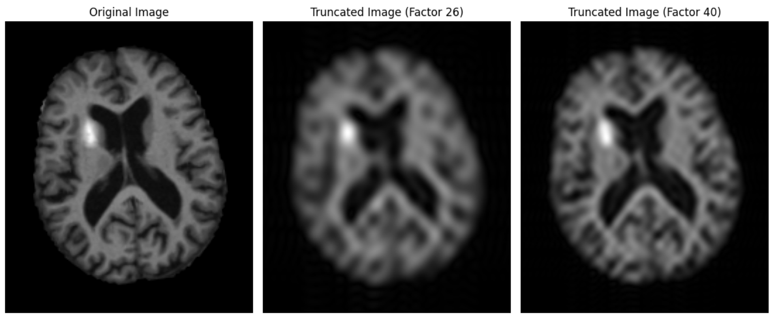


These are the final images for each input. For each original image:

1. Four augmented versions are created due to the specified number of augmentations (num_augmentations).
2. For each augmented image, corresponding versions are saved:
    a. A low-quality version (low_quality_image_i_j.png).
    b. A Gaussian noisy version (gaussian_noisy_image_i_j.png).
    c. A salt-and-pepper noisy version (salt_pepper_noisy_image_i_j.png).

Truncation Factor





Conclusion

This project developed a software tool for augmenting MRI data, focusing on Gaussian and salt-and-pepper noise addition, alongside k-space truncation. The comparative analysis of these augmentation techniques revealed distinct impacts on image quality, demonstrating the effectiveness of each method in simulating realistic medical imaging scenarios. This approach significantly enhances the diversity and realism of datasets, crucial for training advanced machine learning models in medical diagnostics.

Appendix

The python code is attached below. List of instructions to run the code is as follows.

- **Code:** FMI Final code.ipynb
- **Input:** load the images into images_folder.
- **Output:** Output will be presented in the output folder.

Refer **README.md** file for execution instructions and further details.

References

- Simard, Patrice Y., et al. "Best practices for convolutional neural networks applied to visual document analysis." In ICDAR, vol. 3, pp. 958-962. 2003.
- Shorten, Connor, and Taghi M. Khoshgoftaar. "A survey on image data augmentation for deep learning." Journal of Big Data 6.1 (2019): 60.
- Lustig, Michael, et al. "Sparse MRI: The application of compressed sensing for rapid MR imaging." Magnetic Resonance in Medicine: An Official Journal of the

International Society for Magnetic Resonance in Medicine 58.6 (2007): 1182-1195.

- Dabov, Kostadin, et al. "Image denoising by sparse 3-D transform-domain collaborative filtering." IEEE Transactions on Image Processing 16.8 (2007): 2080-2095.
- Jain, Vishal, and Jeffrey F. Cohn. "Adaptive and constrained algorithms for inverse compositional active appearance model fitting." IEEE Transactions on Pattern Analysis and Machine Intelligence 35.2 (2013): 318-331.