Ryan Miller
u1067596
Professor Whitaker
Digital Image Processing

Project 2 - Image denoising with convolution, non linear processes, and neural networks

Introduction

In the second project of CS 6640, we investigate various types of noises and analyze both linear and nonlinear denoising processes to reduce the amount of noise added to images. This report will be split up into two different sections based on the project, the first portion being dedicated to analyzing types of noise and the effectiveness of using non-linear and linear techniques for reducing noise. In the first section, various methods of linear and non linear filtering will be presented along with their effectiveness in denoising while exploring the noise adding and filtering function parameters. The second portion of this project will be dedicated to analyzing the use of different neural network architectures to learn a model to reduce noise on testing and training sets of images. For more information on the python scripts, please read the README documents submitted alongside the report. Just as the report, the python scripts will be organized based on their respective portion to the report, and separate README files are provided for each portion. A variety of images will be presented in the report that can be reproduced by the python scripts. After both sections of the report have been presented and analyzed, credit will be given to other individuals who have made this analysis possible.

Task ½ - Linear and Nonlinear Filtering

In task one, I am asked to build and experiment with several different linear filters with different sizes using correlation/convolution. Using Mean Square Error, I can analyze the effectiveness of these filters when compared against the un-altered "ground truth" images. In this first portion of the project, I have decided to analyze both the linear and nonlinear filters on four different types of noise addition: gaussian, salt and pepper, poisson, and speckle. Gaussian noise is statistical noise that has a probability density function defined by:

$$p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-\bar{z})^2}{2\sigma^2}}$$

where z represents the intensity (from negative infinity to infinity), $\bar{z}$ is the average value of z and σ is its standard deviation. In calling random_noise from skimage utilities, we can specify the type of noise we want to add as Gaussian noise and specify mean and variance parameters:

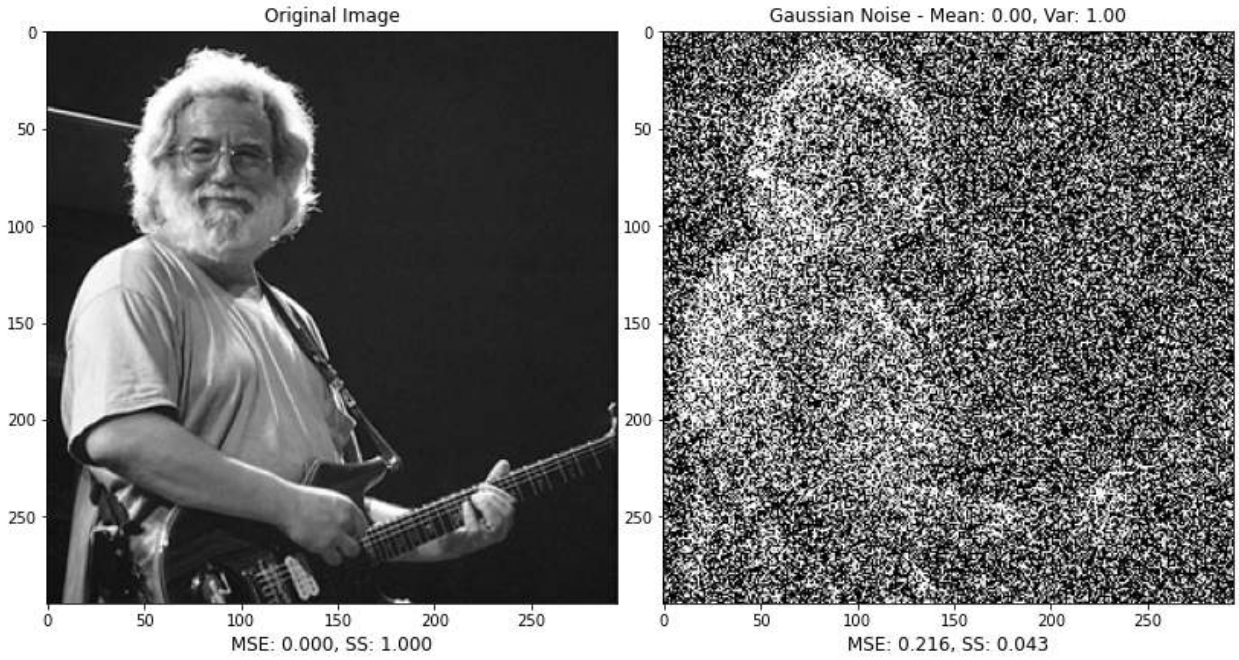

Figure 1: Gaussian Noise Example with Mean: 0.00, Variance: 1.00

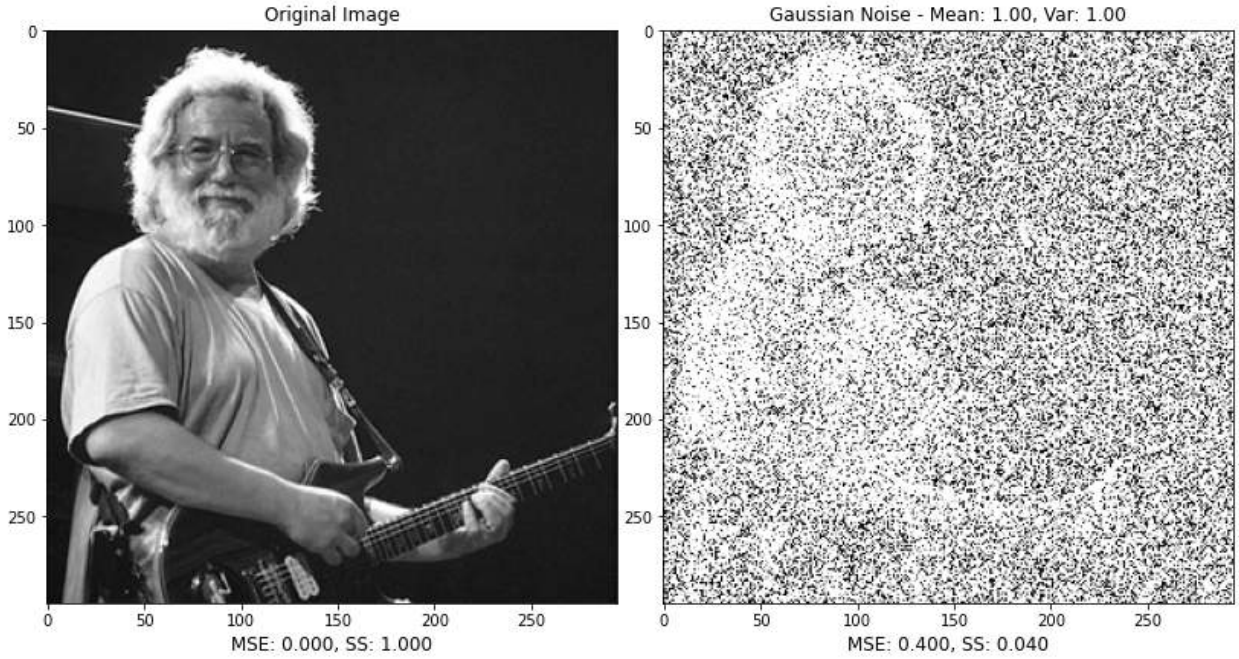

Figure 2: Gaussian Noise Example with Mean: 1.00, Variance: 1.00

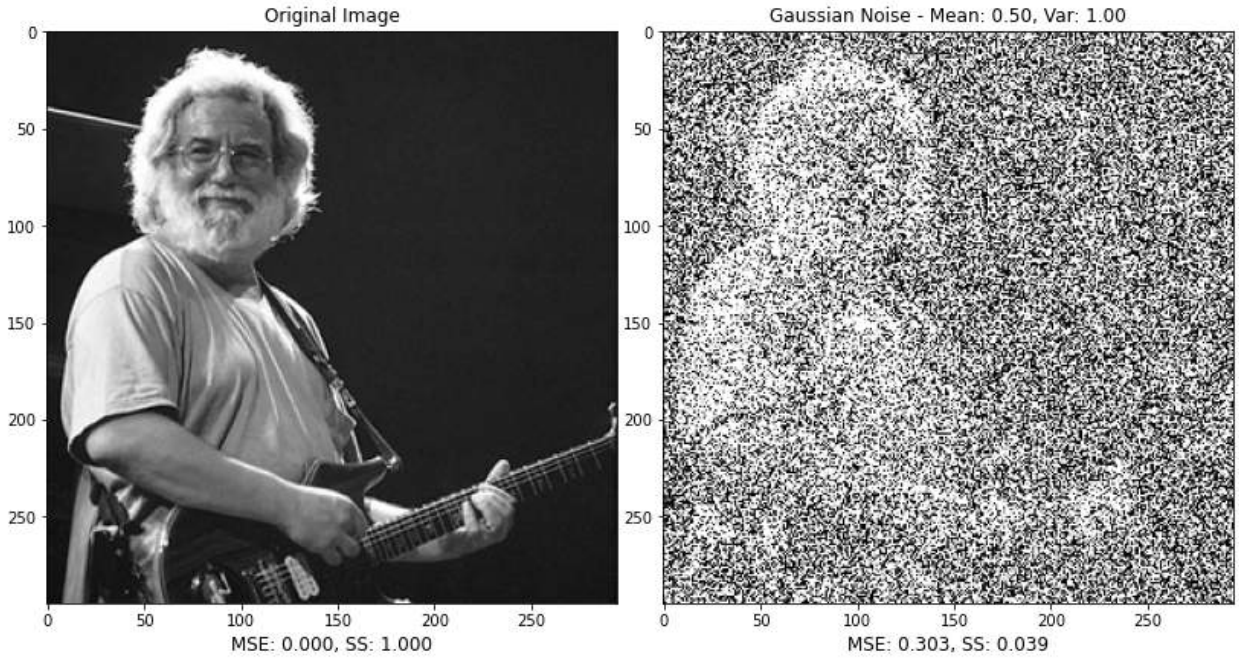

Figure 3: Gaussian Noise Example with Mean: 0.50, Variance: 1.00

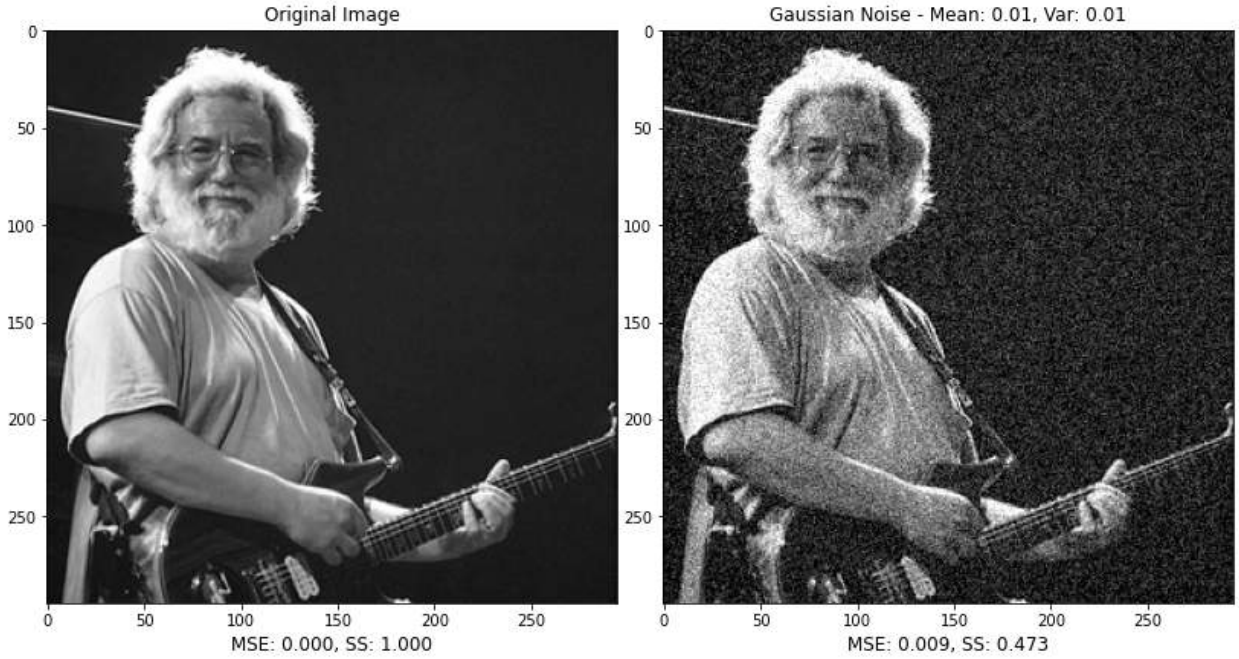

Figure 4: Gaussian Noise Example with Mean: 0.01, Variance: 0.001

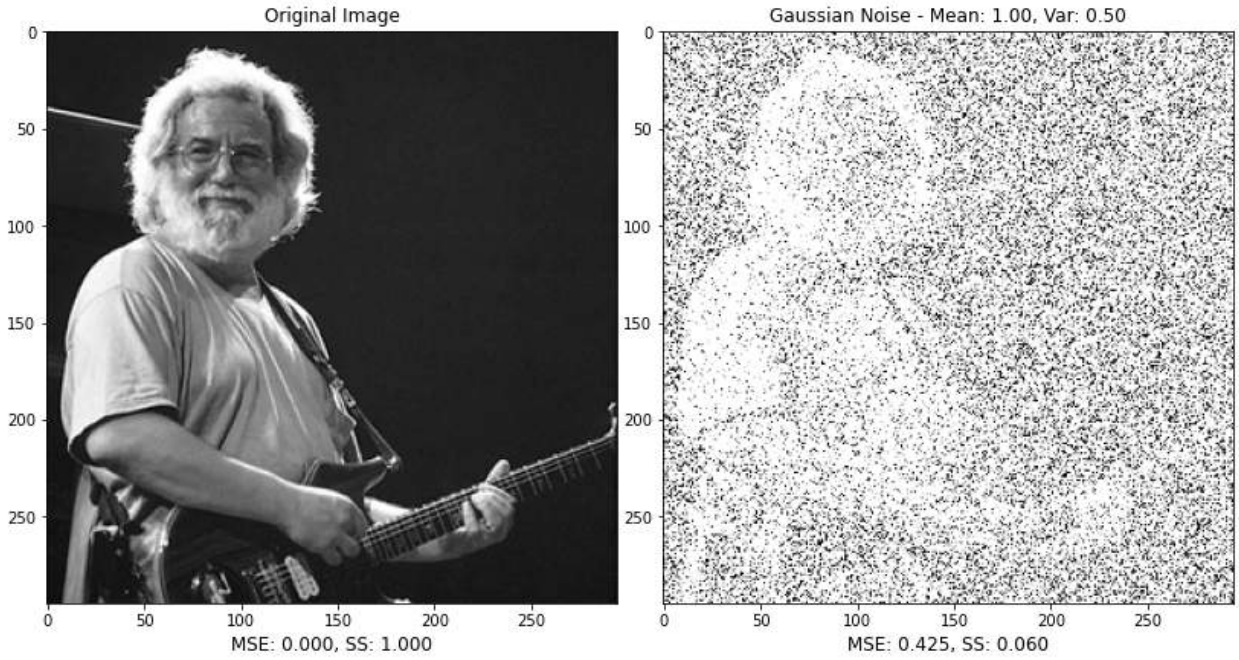

Figure 5: Gaussian Noise Example with Mean: 1.00, Variance: 0.05

From our knowledge of the probability density function of a Gaussian random variable z, we can conclude that high values in either the variance or the mean will produce noisier images than lower values. In comparing Figure 5 to Figure 2, we see that a higher variance produces noise with a greater range (in greyscale terms) of noise added to our image. Figure 5 has more whitenoise added while Figure 2 has both whitenoise and darker noise added to the image. Both images have very similar MSE values, a parameter we are using to quantify the amount of noise in an image. Higher MSE values correspond to a greater overall difference in comparison to our "ground truth" un-noised image. In comparing Figures 2 and 3 we also see that a higher mean produces a greater range of noise compared to that of a lower mean. The higher mean value also produces a greater MSE value by almost 0.1 signaling an overall greater difference between the noisy image and our original unaltered image. Next, we analyze salt and pepper noise, also known as impulse noise, which includes noise that is added as white and black pixels.

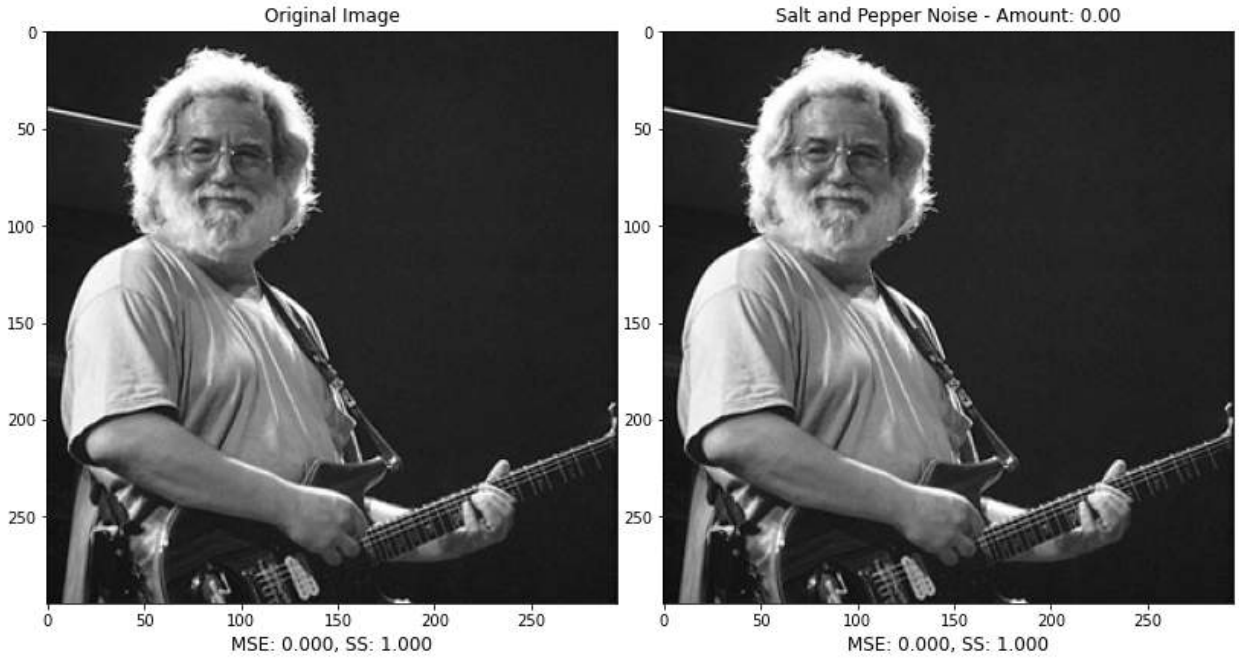

Figure 6: Salt and Pepper Example with Amount 0.00

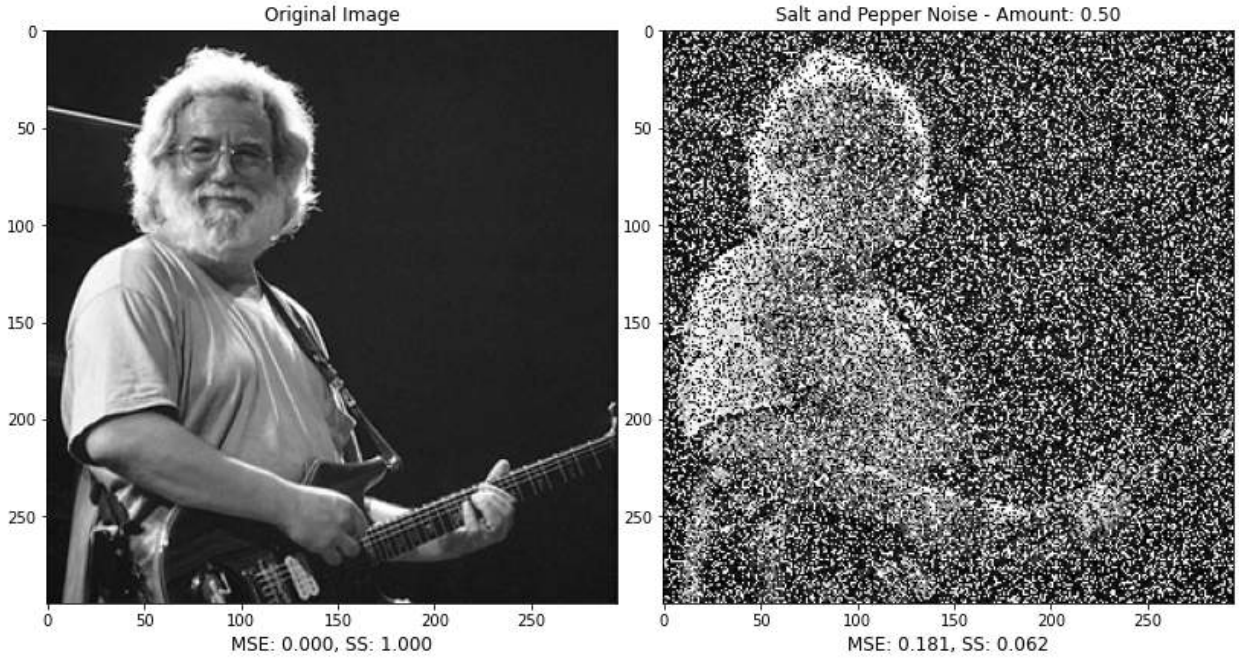

Figure 6: Salt and Pepper Example with Amount 0.50

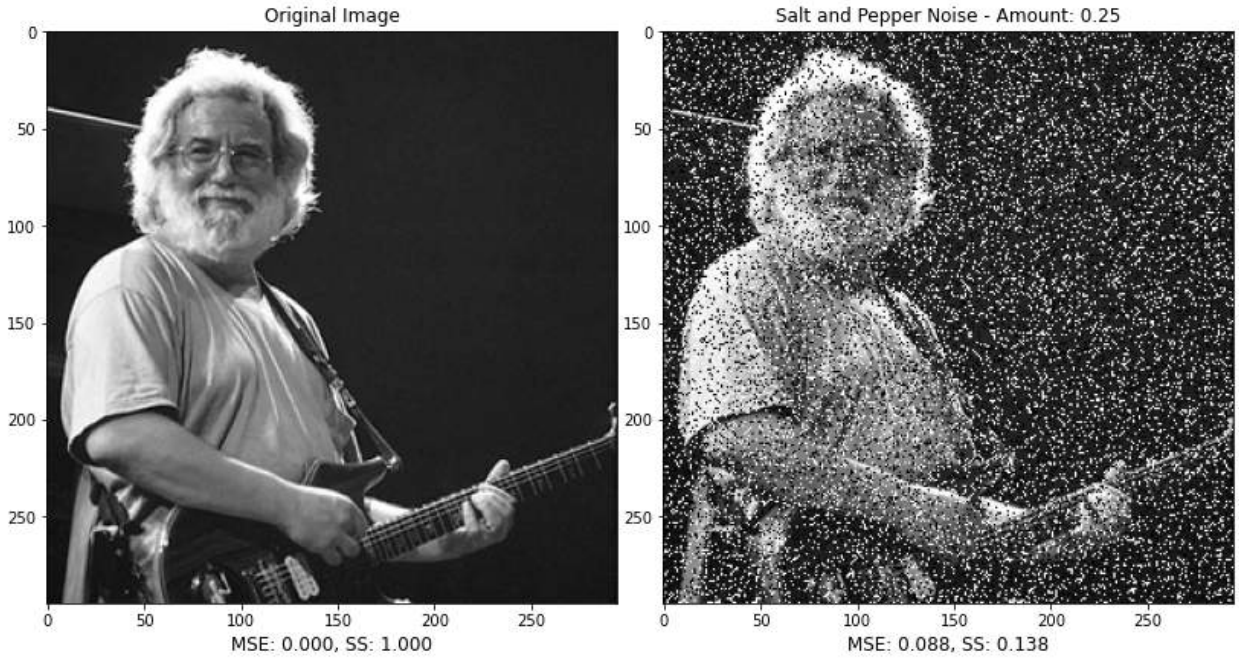

Figure 8: Salt and Pepper Example with Amount 0.25

We see in Figures 6, 7, and 8 that increasing the amount of salt and pepper noise increases the amount of black and white pixels in the image, and produces greater MSE values for the noisy image. The other parameter we analyze is structural similarity (SS) which does a better overall job of analyzing the perceived similarity of the image to the original by taking texture into account. When comparing the ground truth to itself, we see a MSE value of 0 and a similarity of 1, meaning it is completely texturaly similar to itself. Thus, with the highest amount of salt and pepper noise, we see that Figure 7 is the least structurally similar to the ground truth, as expected. The next type of noise we analyze is poisson, which is a statistical noise that follows a poisson distribution. For this type of noise, we do not pass any parameters to the poisson random noise mode included in the random_noise function of skimage to simulate true poisson distribution noise:

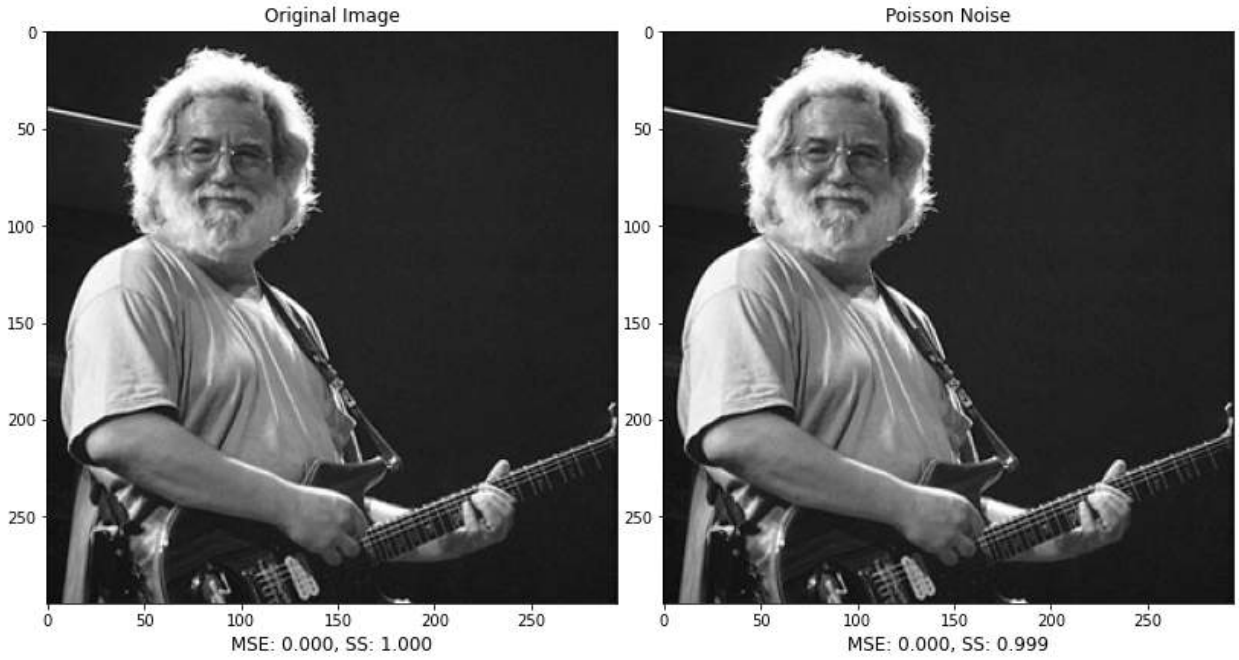

Figure 9: Poisson Example

From Figure 9, we see that poisson noise does not add much noise to the image at all, as the structural similarity is 0.999 indicating it is very similar texturally to the original with an MSE value of 0. The amount of noise added is very minimal. The last type of noise analyzed in portion 1 is speckle. Speckled noise simulates many environmental conditions and simulates granular interference. We can pass in mean and variance parameters to speckle to analyze their effects on the quality of noise added to our original image.

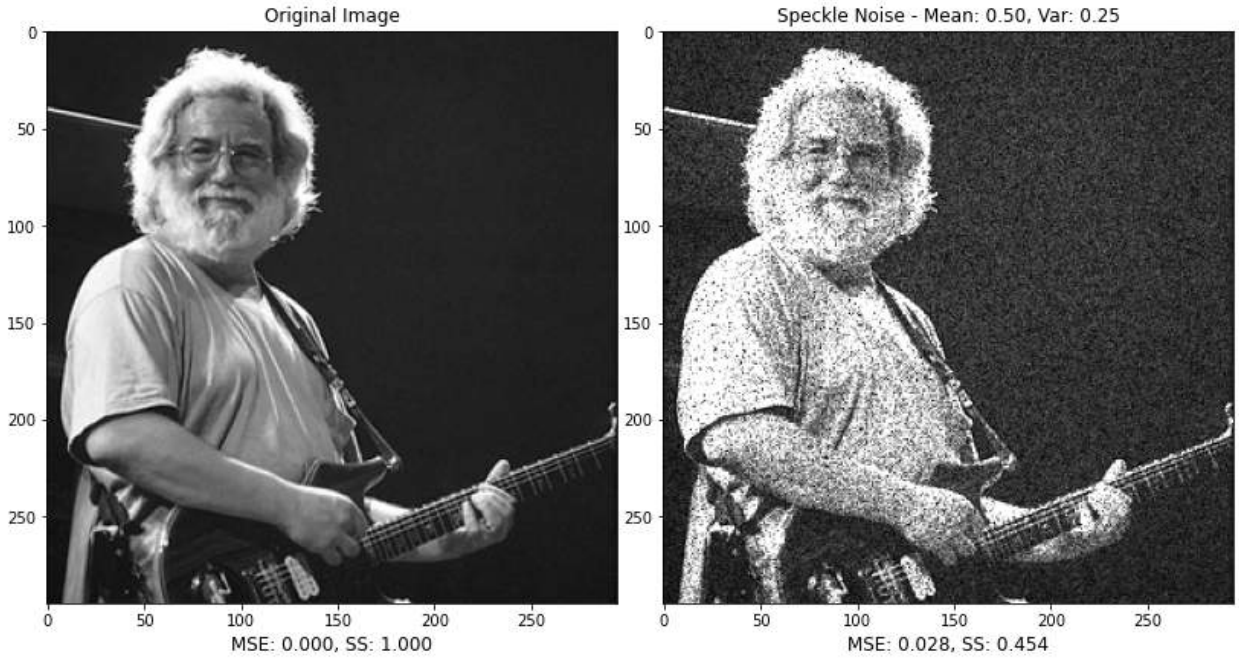

Figure 9: Speckle Example with Mean 0.50 and Variance 0.25

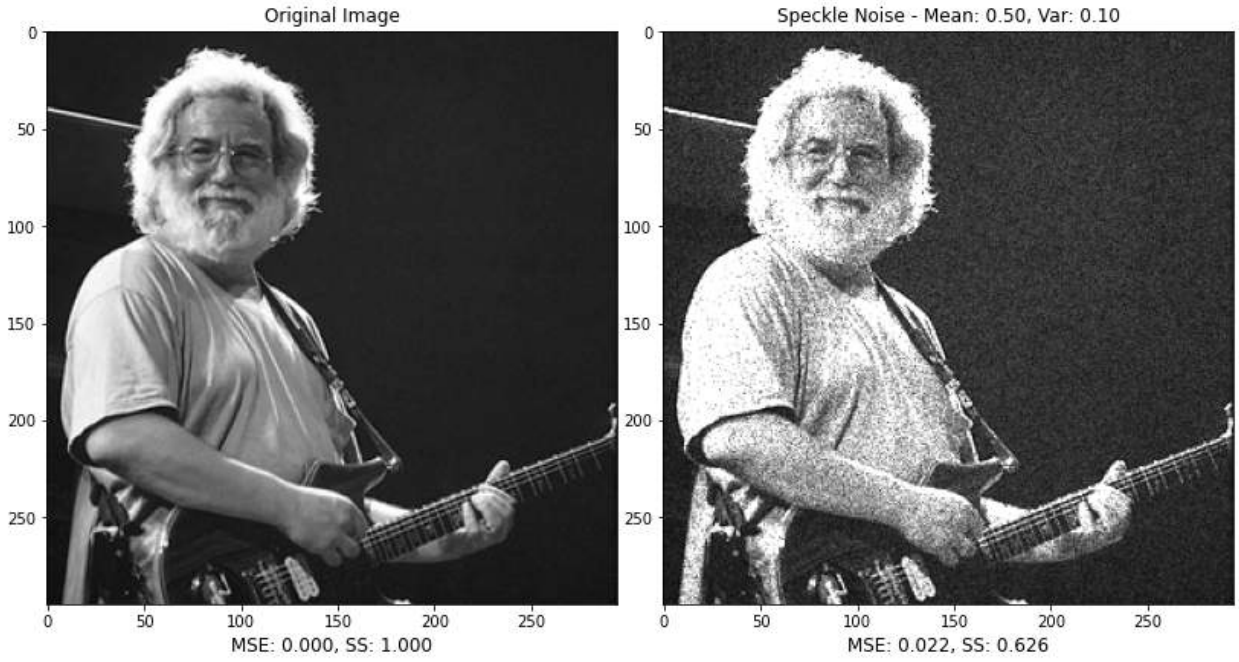

Figure 10: Speckle Example with Mean 0.50 and Variance 0.1

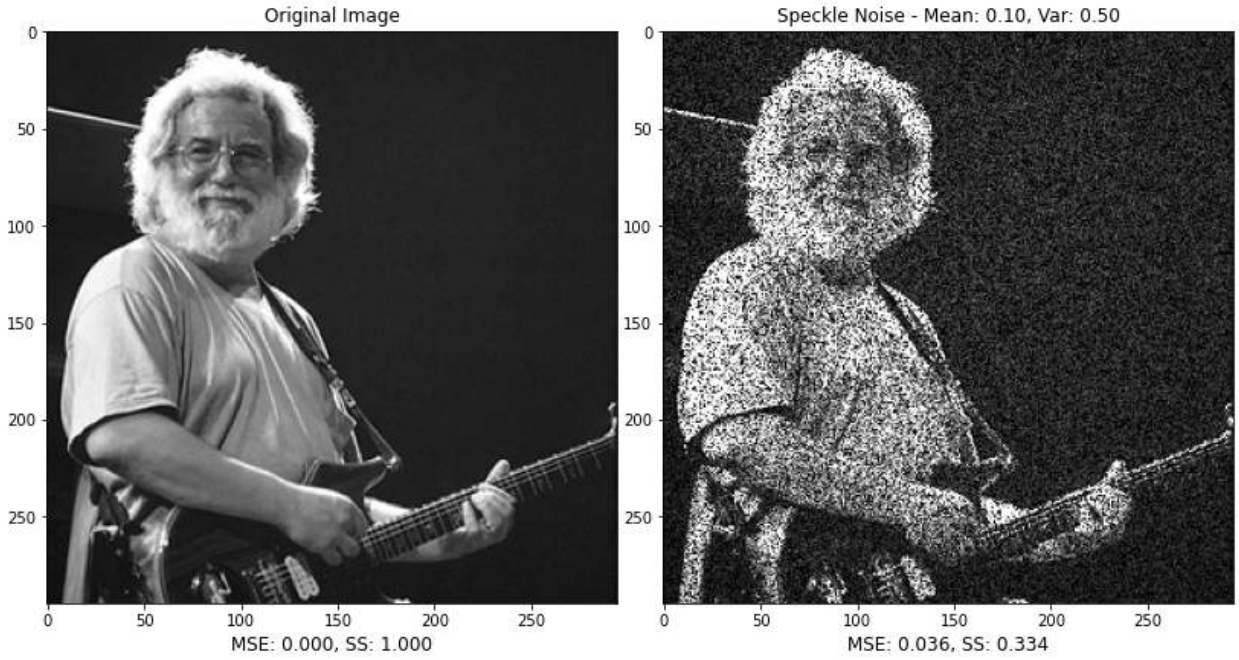

Figure 11: Speckle Example with Mean 0.1 and Variance 0.5

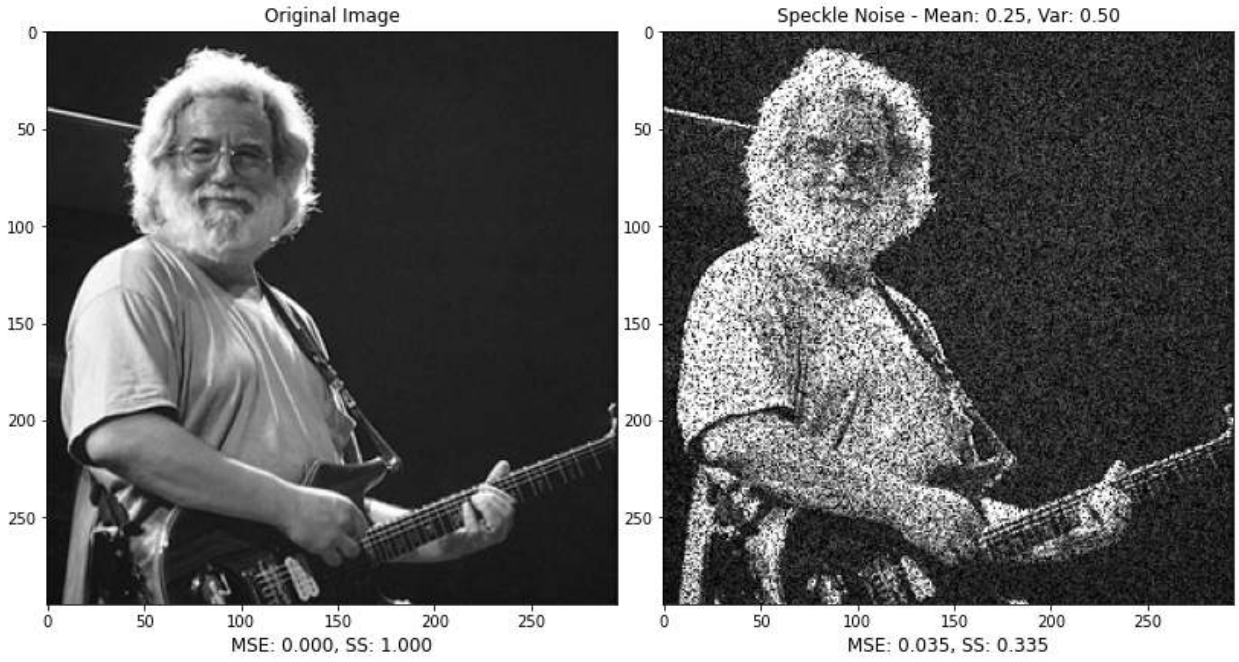

Figure 12: Speckle Example with Mean 0.25 and Variance 0.50

From Figures 9 and 10 we see that decreasing the variance results in noisy images with lower overall MSE and a higher structural similarity, signaling a noisy image that has less noise and resembles the ground truth more so in comparison with an image with a higher variance. From

Figures 11 and 12, we see that mean does not have a great impact on the overall amount of noise added to our ground truth images.

Linear Filters

Now that we have analyzed the four different types of noise we will use our filters on and how the parameters affect the amount of noise, both quantitatively via MSE and qualitatively via SS (structural similarity), we are ready to start analyzing various types of linear filters and their effectiveness in denoising images. In total, we will analyze seven different linear filters: a box filter, derivative filter, wiener filter, 2 gaussian filters, rectangular filter, and a combination of two derivative filters. We will analyze their effectiveness by presenting the results on 5 different images with different amounts of noise added, shown above by changing the parameters passed into the respective noise function. To create a controlled experiment for accurately analyzing the filters' abilities in denoising images, we must compare results of similar types of input noise levels. Thus, for both the linear and nonlinear filters, I will analyze three amounts of noise for each noise type. For noise types that take mean and variance input: Gaussian and speckle, three different trails will be run: trial one consisting of gaussian/speckle noise with mean 0.1 and variance 0.5, trial two with mean 0.25 and variance of 0.25, and trial three with mean 0.5 and variance 0.1. A similar approach will be taken for salt and pepper noise. Three trials will be set up with trail one having a salt and pepper amount of 0.1, trial two with amount 0.25, and trail three with 0.5. The only type of noise that cannot be adjusted is poisson. Both the linear and nonlinear filters will be applied to poisson noise with no arguments passed.

If I were to also adjust the parameters for the filters for all different amounts of noise, the number of images being presented would reach nearly 150. Thus, to keep the analysis compact, I will present on the best filtering parameters, resulting in the best overall denoising and will speak on these parameters in the analysis but will not present the subsequent images.

First, we investigate the effectiveness of linear filters on Gaussian noise via trial one: Gaussian noise with mean 0.1 and variance 0.5:

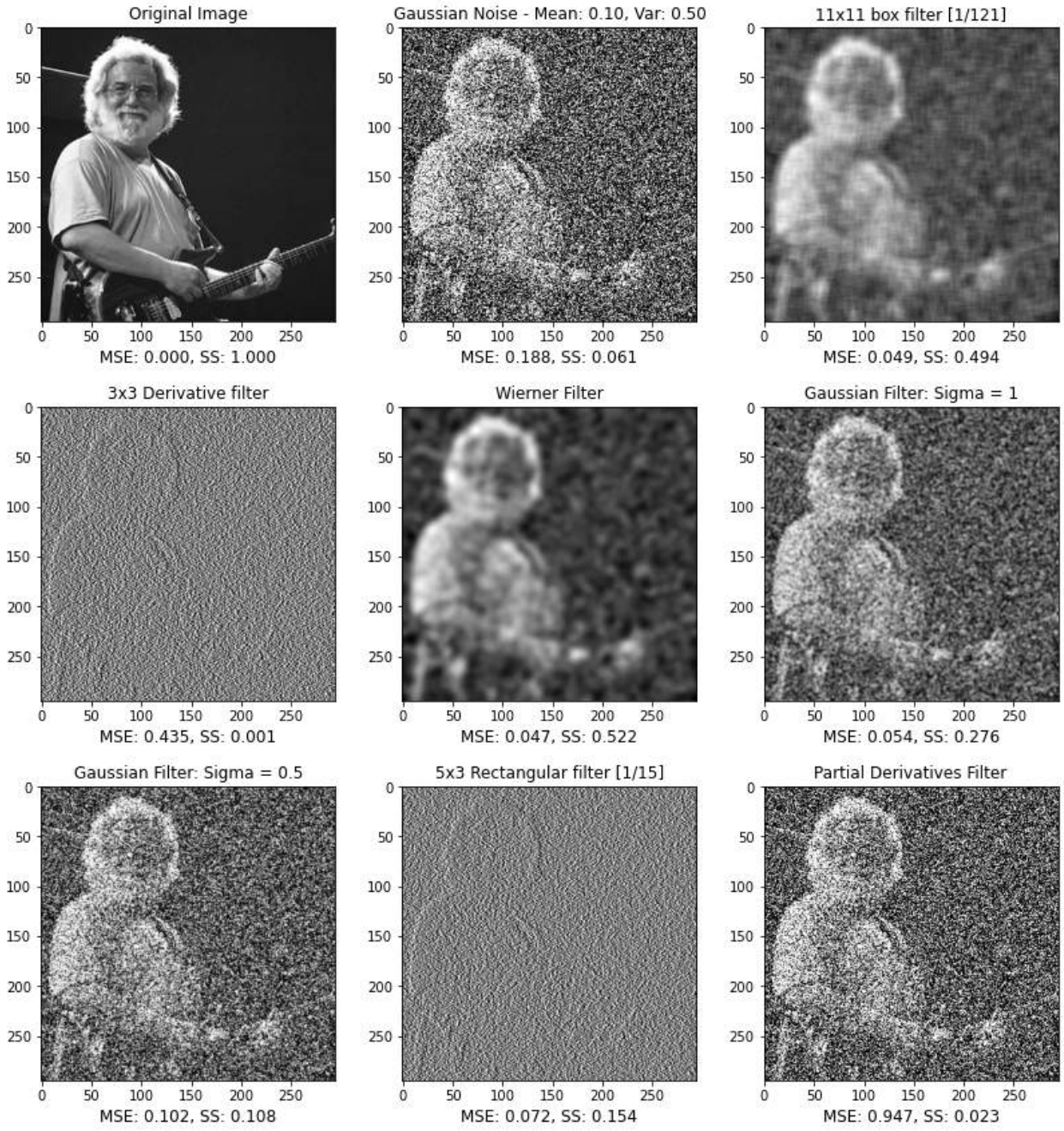

Figure 13: Linear filtering on Gaussian noise with mean 0.1 and variance 0.5

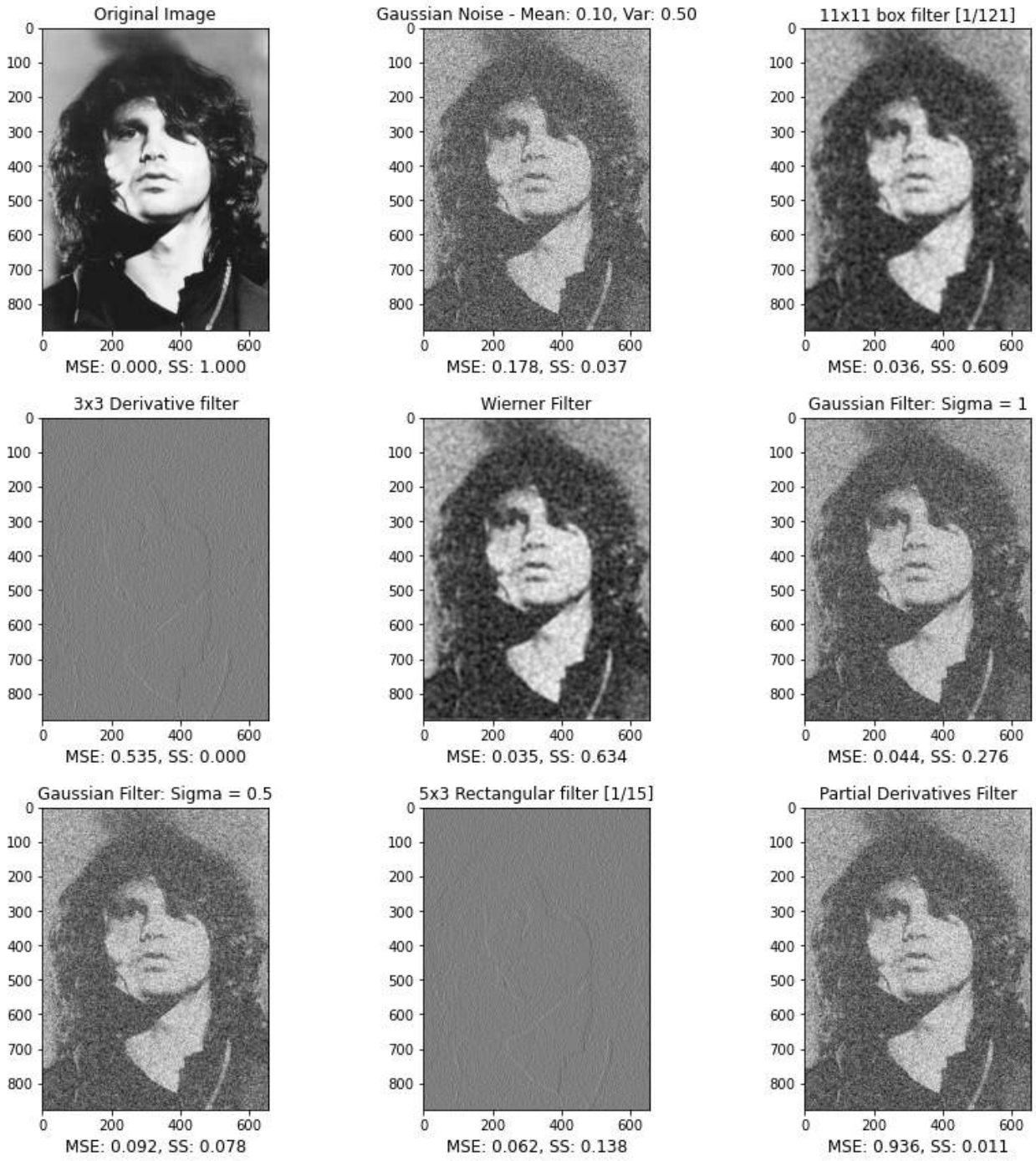

Figure 14: Linear filtering on Gaussian noise with mean 0.1 and variance 0.5

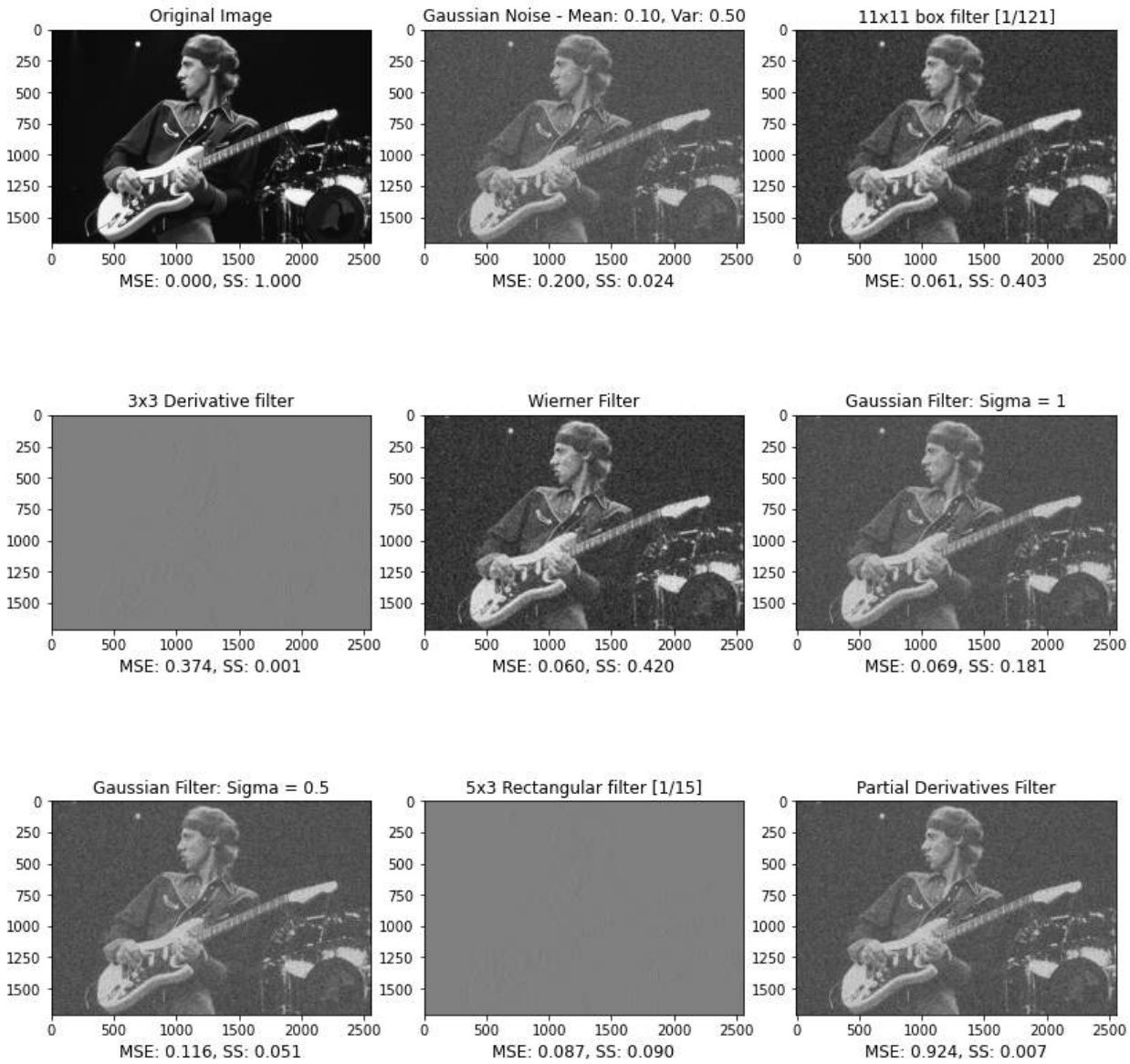

Figure 15: Linear filtering on Gaussian noise with mean 0.1 and variance 0.5

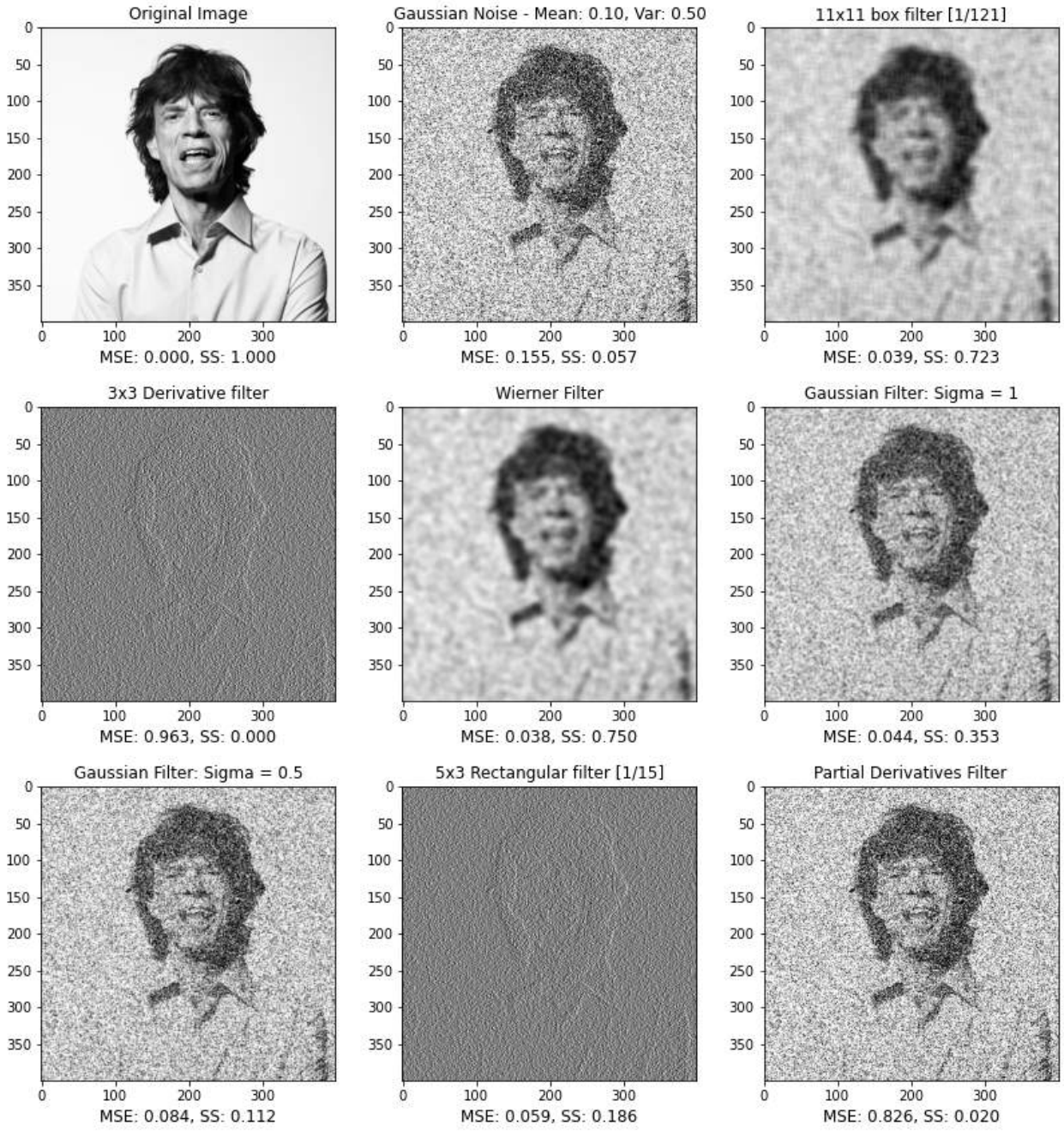

Figure 16: Linear filtering on Gaussian noise with mean 0.1 and variance 0.5

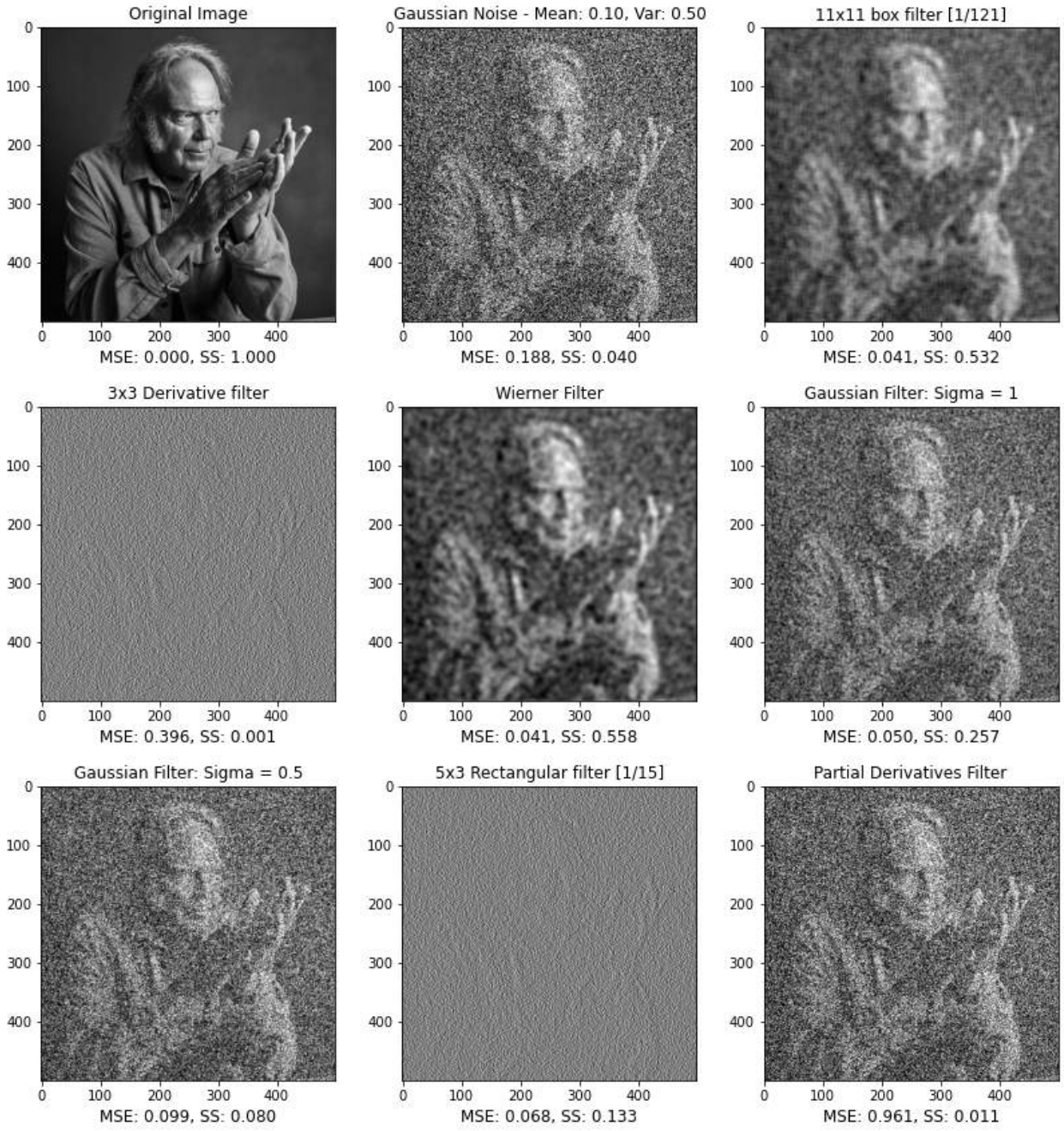

Figure 17: Linear filtering on Gaussian noise with mean 0.1 and variance 0.5

Analyzing the array of images above, it is clear that some linear filters are more effective than others. The box filter does a good job of decreasing the mean squared error of the noisy image and increasing the structural similarity of the noisy image, although it appears to add a good amount of blurring to the image. After trial and error, a box filter of size 11 by 11 produced the best statistical results, a denoised image with the lowest mean squared error and highest overall structural similarity. The overall amount of noise added is substantial, as the noisy image produced has a structural similarity of just 0.057 in Figure 16. The two filters that appear to do the worst to the Gaussian noise are that of the derivative filter and rectangular filter, producing images that are unrecognizable. The wiener filter is very comparable to the large box filter, producing an image that is blurry but has a fairly high structural similarity with a low mean squared error value. The gaussian filter with sigma equal to 1 produces better results than a gaussian filter with smaller sigma. From a visual standpoint, the partial derivatives filter (matrix multiplication of two 3 by 3 derivative filters) makes the image worse, vastly decreasing the structural similarity and increasing the mean squared error when compared to the noisy image. Let us move on to trial two with Gaussian noise with mean 0.25 and variance 0.25:

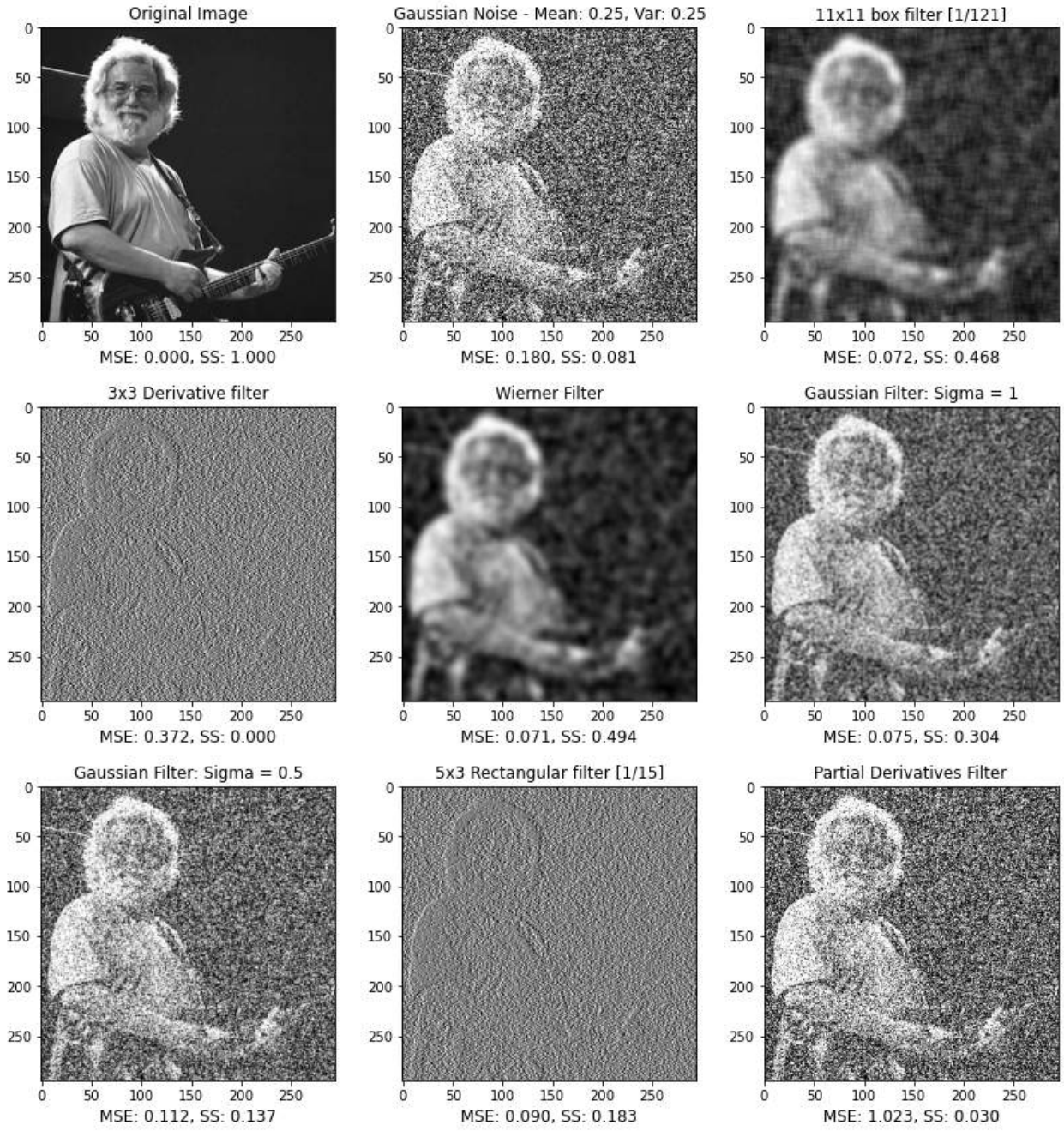

Figure 18: Linear filtering on Gaussian noise with mean 0.25 and variance 0.25

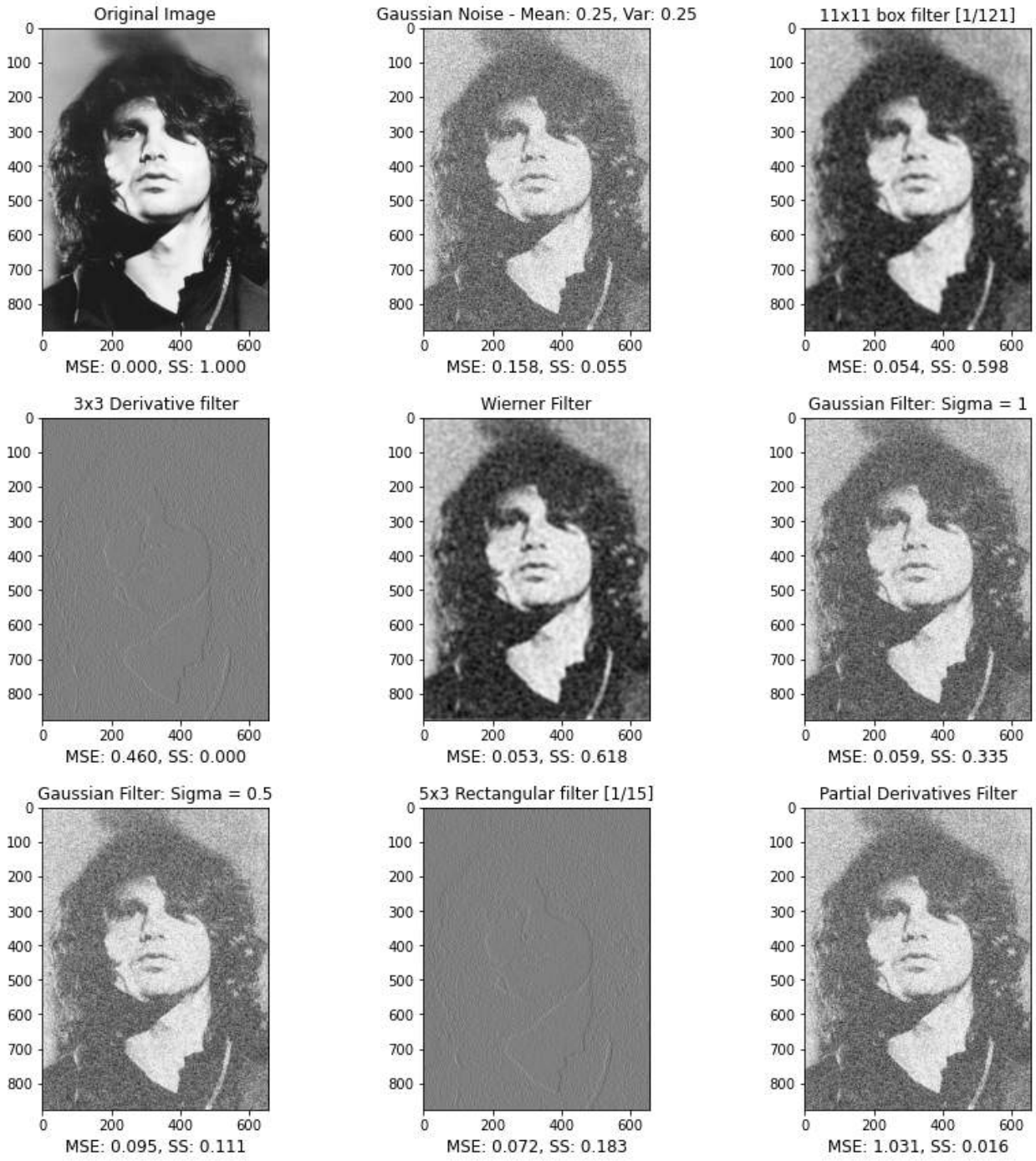

Figure 19: Linear filtering on Gaussian noise with mean 0.25 and variance 0.25

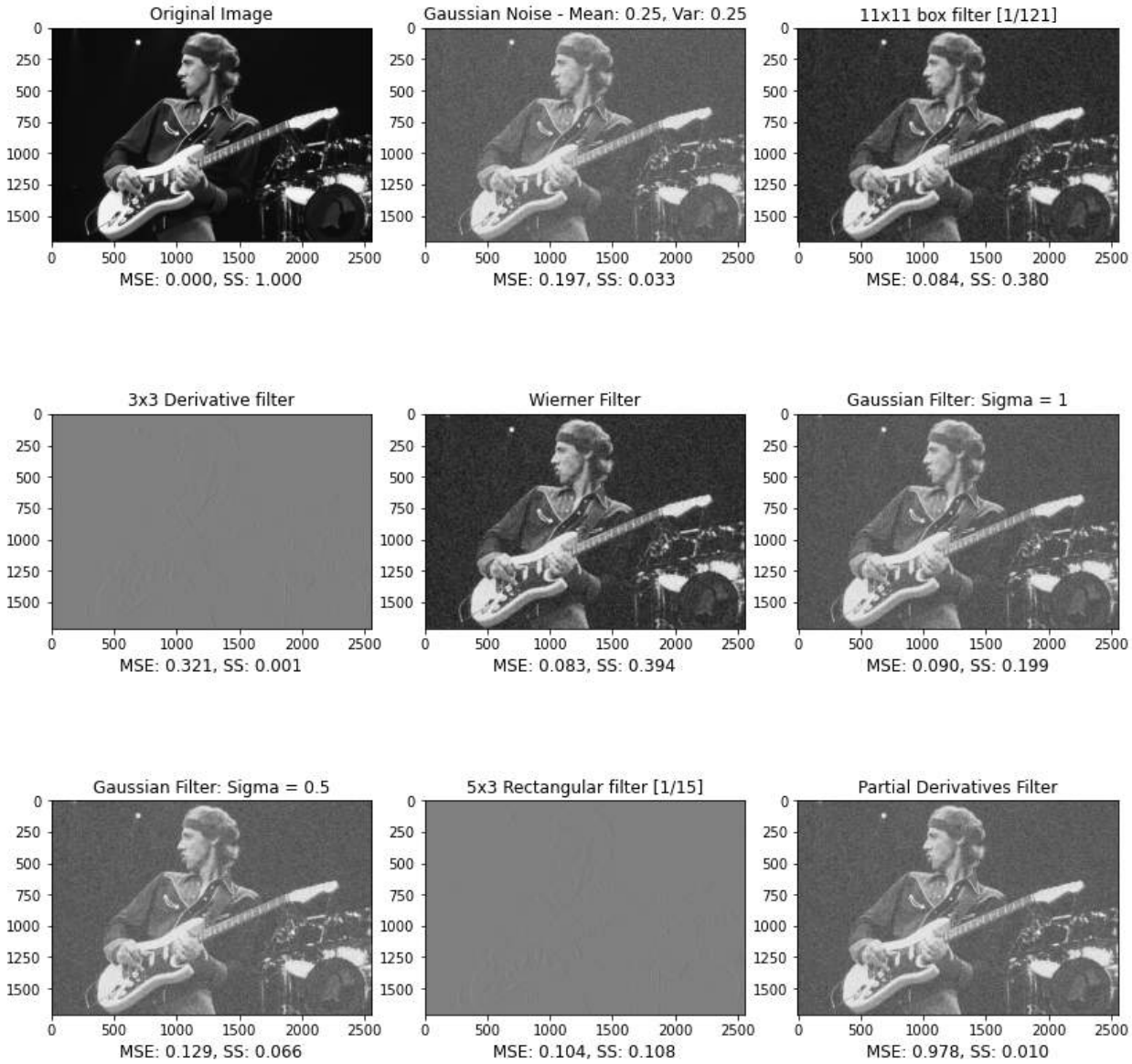

Figure 20: Linear filtering on Gaussian noise with mean 0.25 and variance 0.25

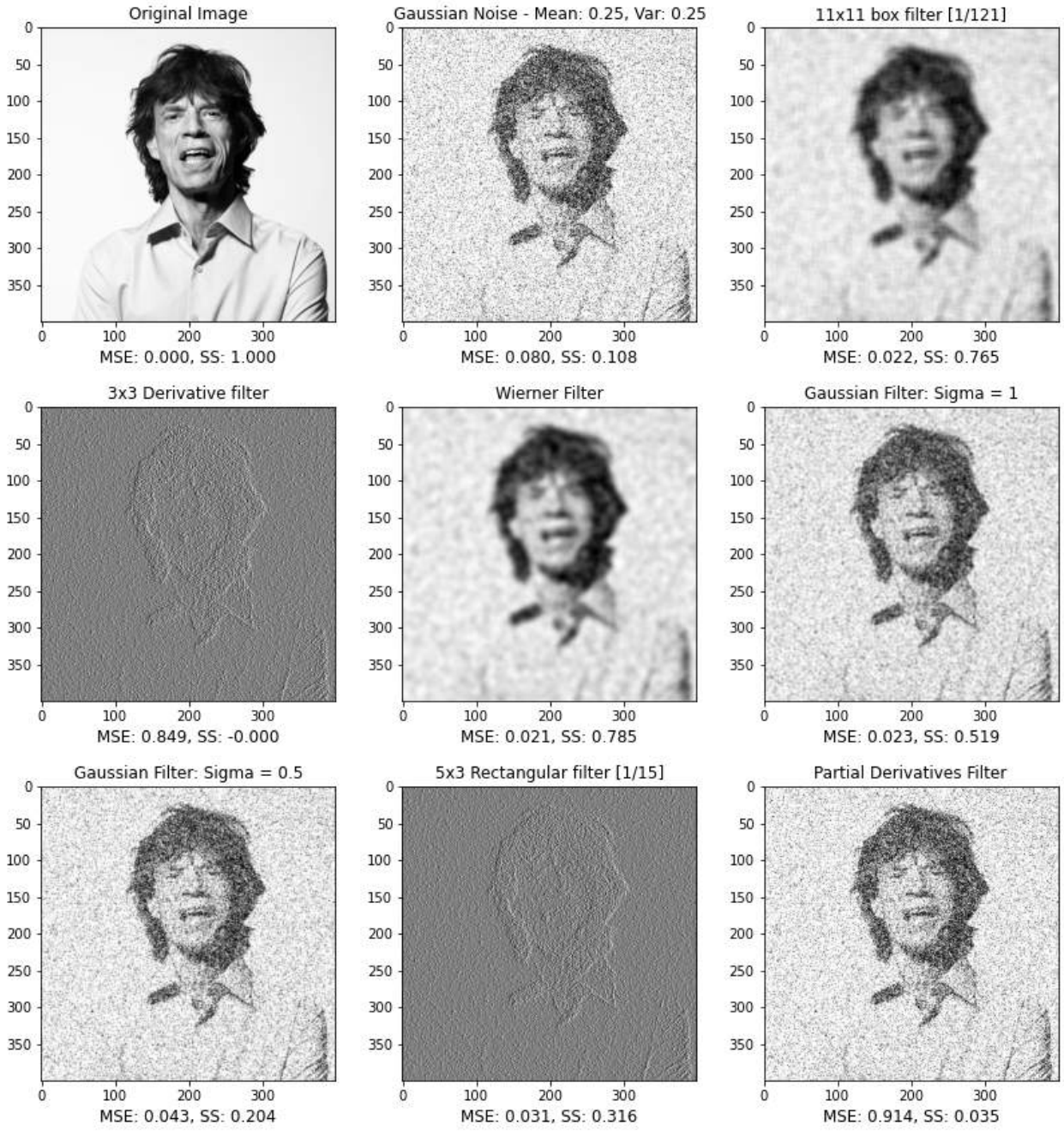

Figure 21: Linear filtering on Gaussian noise with mean 0.25 and variance 0.25

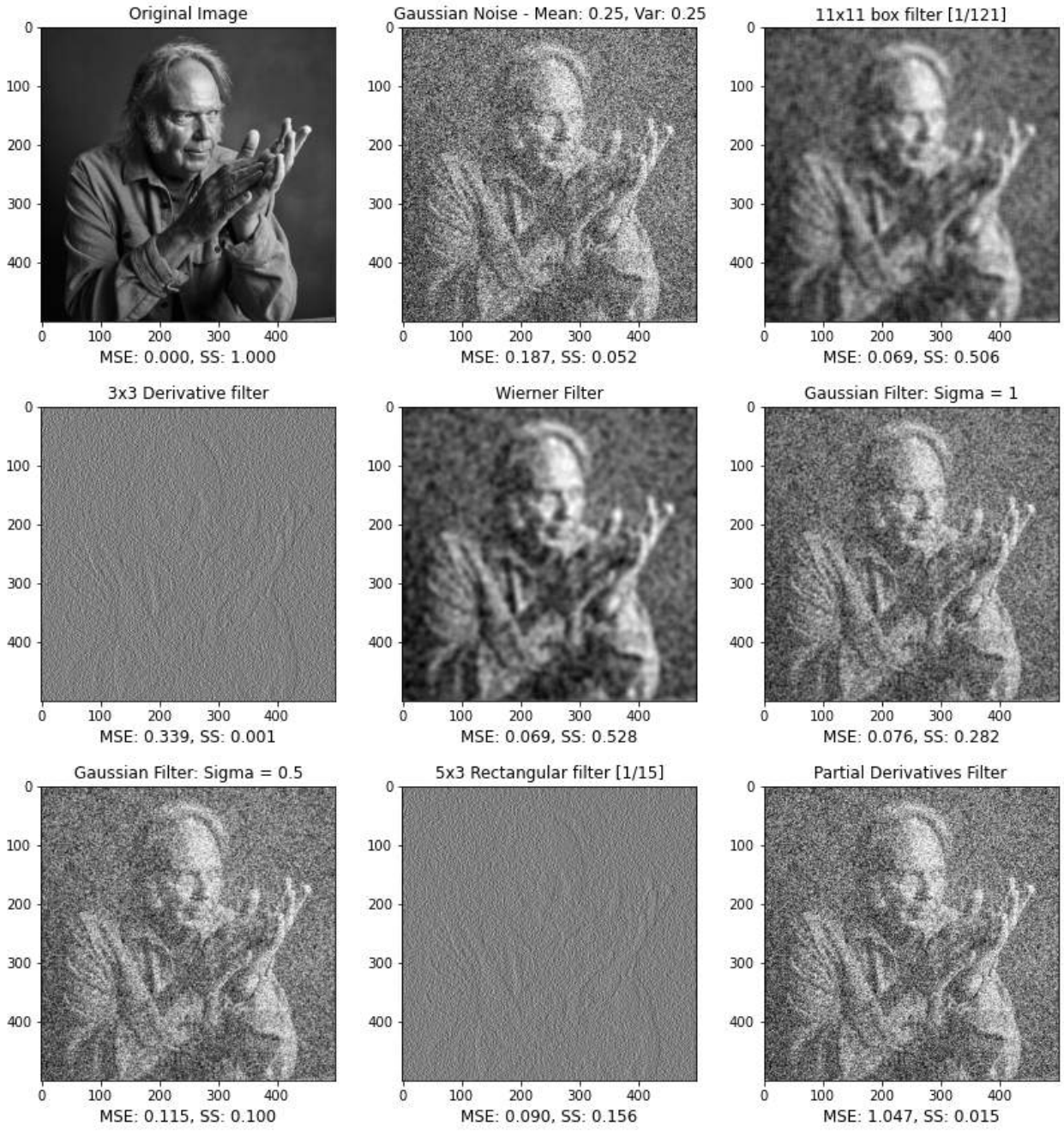

Figure 22: Linear filtering on Gaussian noise with mean 0.25 and variance 0.25

From analyzing the noisy image comparisons of trial 1 and 2, the amount of noise added in trial 2 (Gaussian noise with mean 0.25 and variance 0.25) is less than that of trial one with a lower mean squared error value in the noisy image. Just as seen in trial 1, the 11 by 11 box filter and wiener filter do the best of all the linear filters, producing mean squared error value of about 0.022 and structural similarity values of 0.7. The derivative filter and rectangular filter continue to produce images that are unrecognizable and the Gaussian filter with higher sigma continues to outperform the Gaussian filter with lower sigma value. Trial 3: Gaussian noise with mean of 0.5 and variance of 0.1:

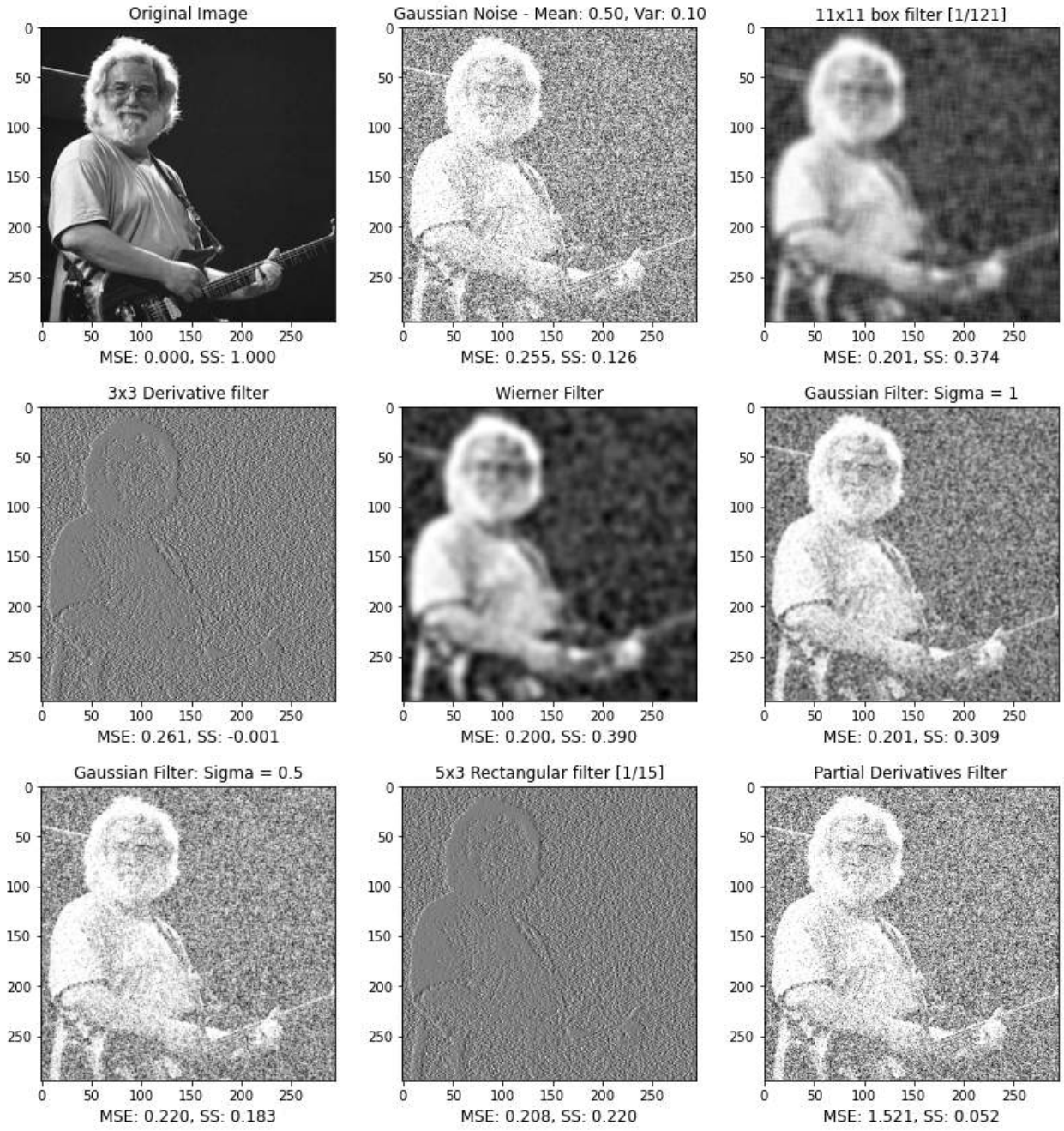

Figure 23: Linear filtering on Gaussian noise with mean 0.5 and variance 0.1

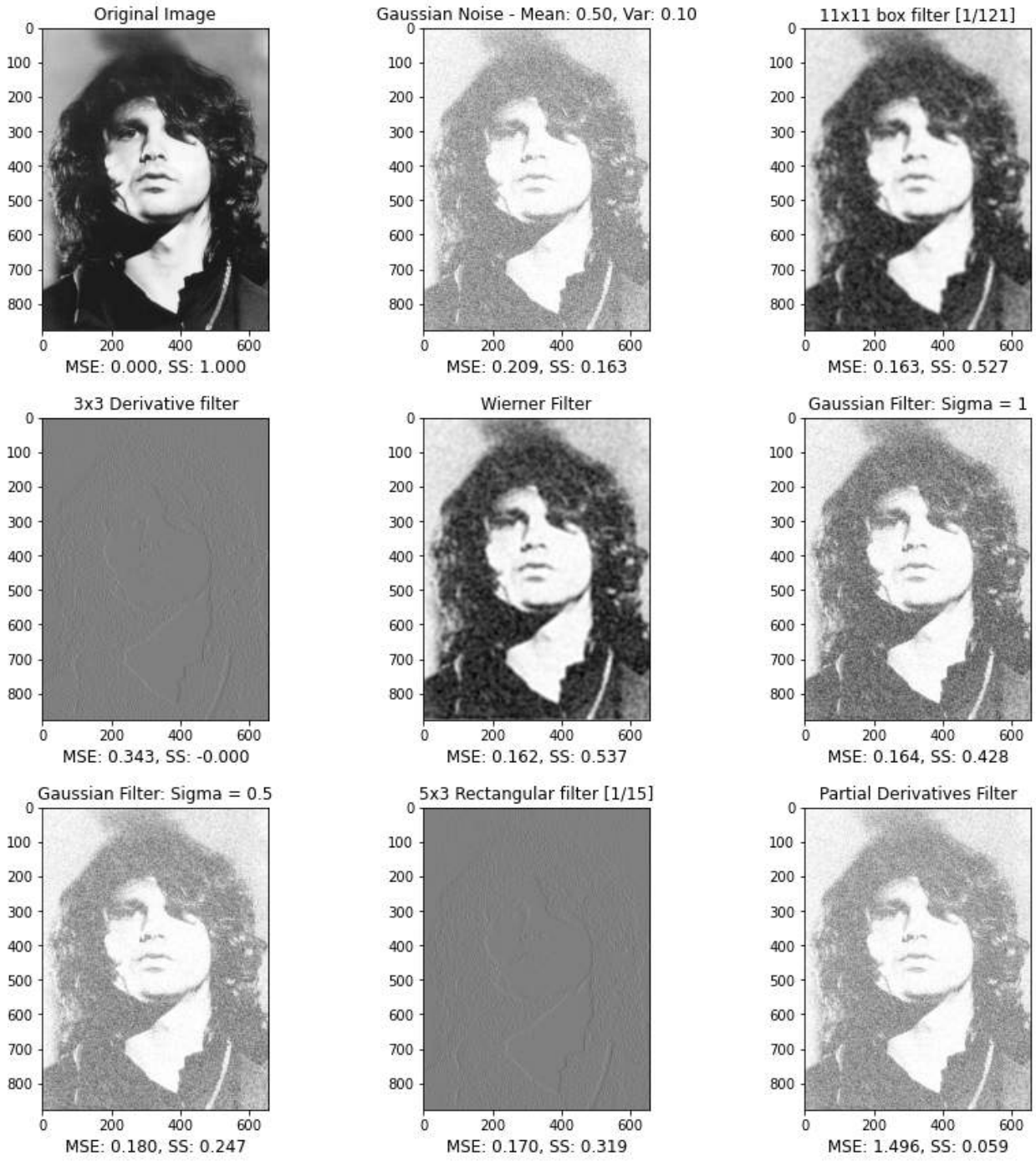

Figure 24: Linear filtering on Gaussian noise with mean 0.5 and variance 0.1

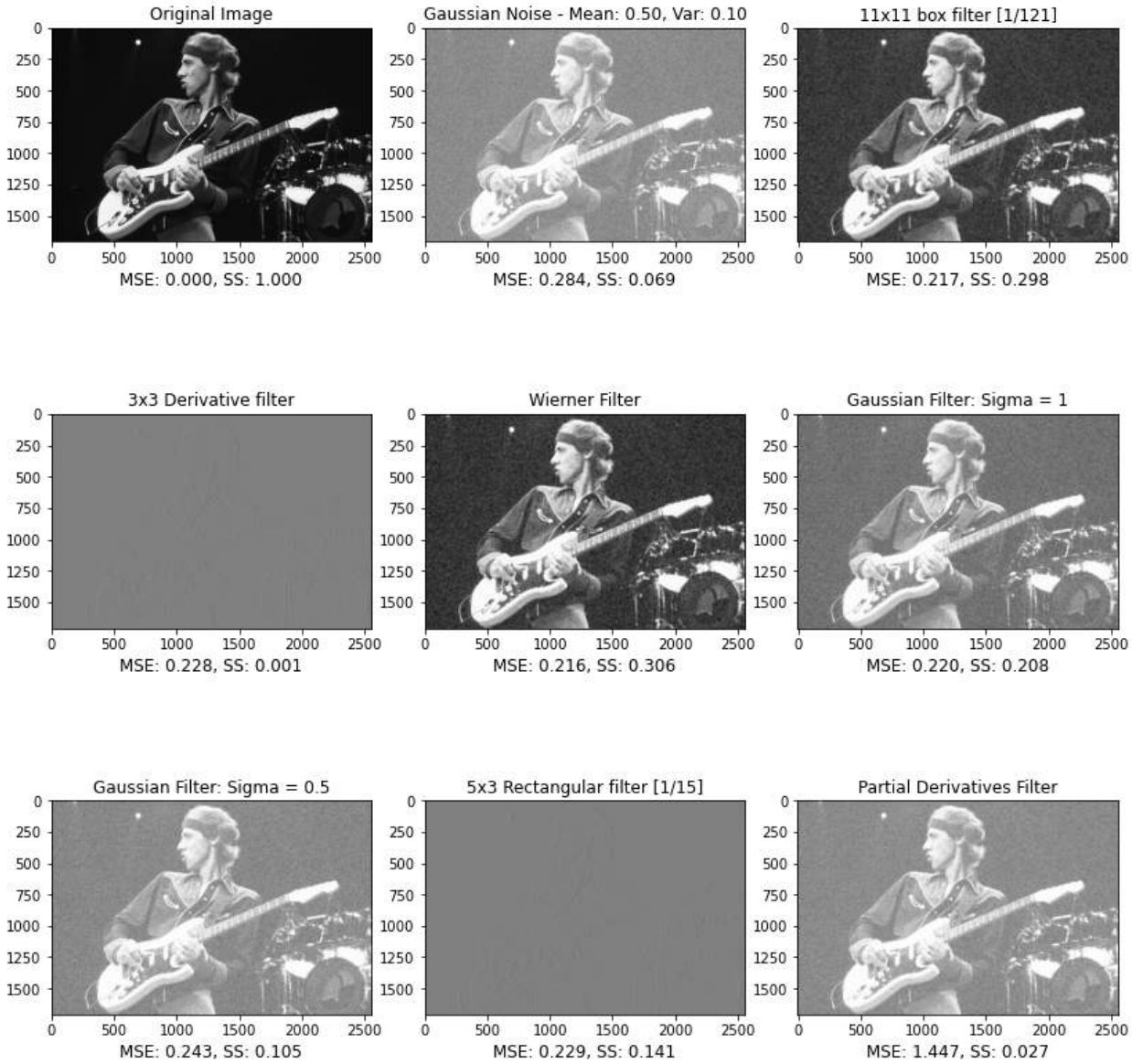

Figure 25: Linear filtering on Gaussian noise with mean 0.5 and variance 0.1

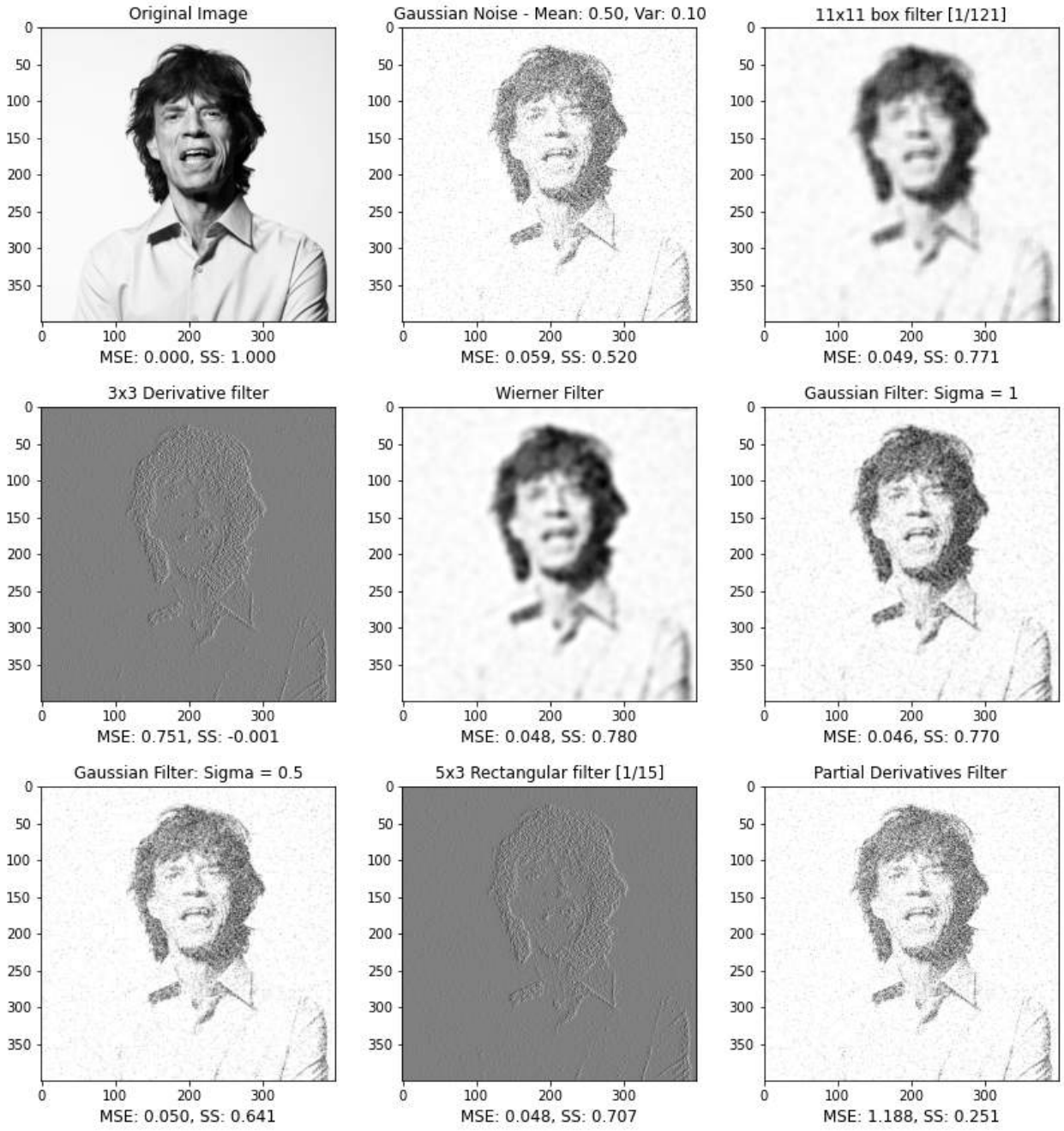

Figure 26: Linear filtering on Gaussian noise with mean 0.5 and variance 0.1

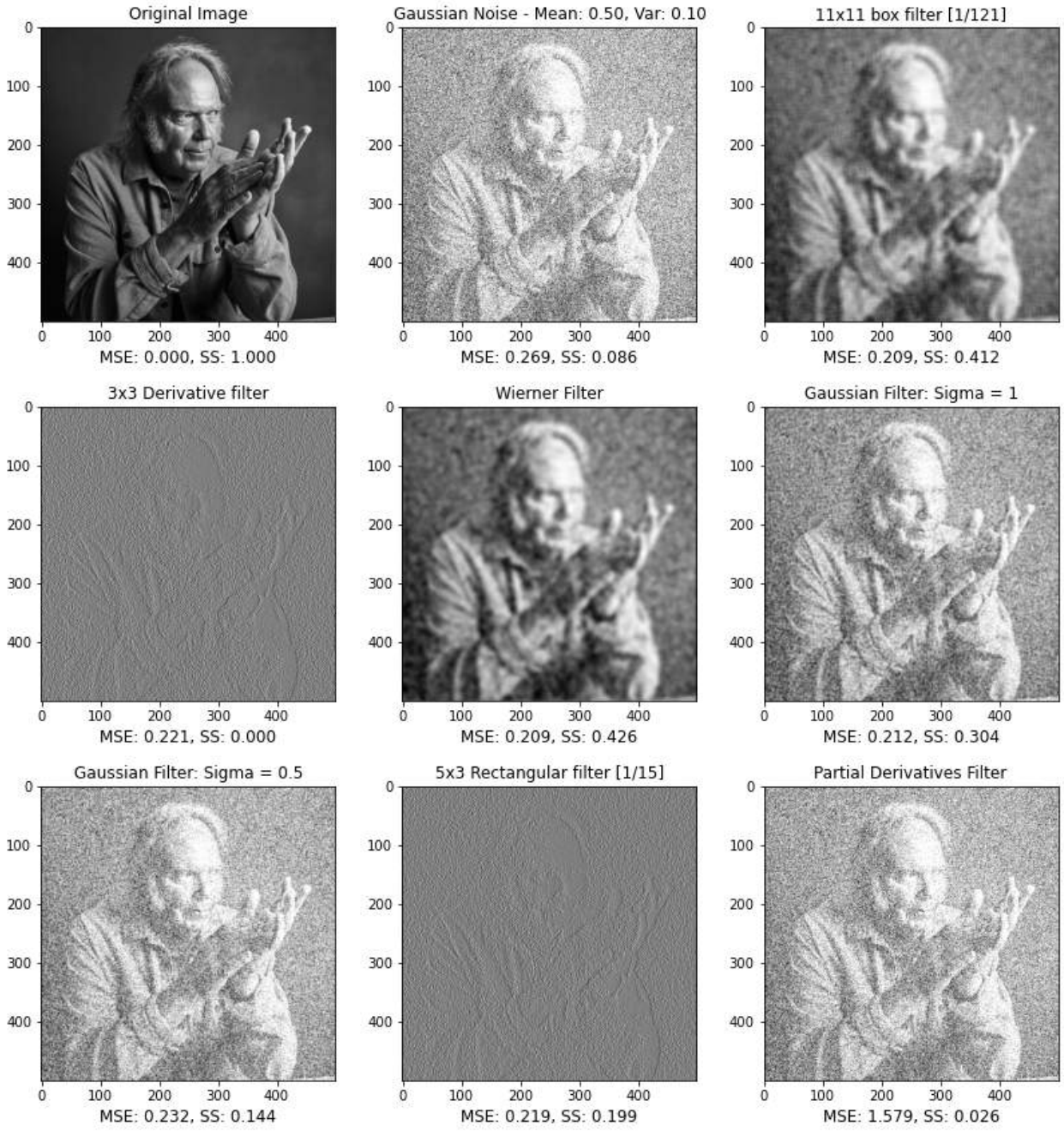

Figure 27: Linear filtering on Gaussian noise with mean 0.5 and variance 0.1

Results are consistent with previous trials. The box filter and wiener filter produce the best results with nearly identical mean squared error and structural similarity values whereas the derivative filter and rectangular filter produce unrecognizable results. We now transition to view the results of the linear filters on salt and pepper noise. The first trial will constraint of amount 0.1, following by trial two at amount 0.25, and trail three at amount 0.5:

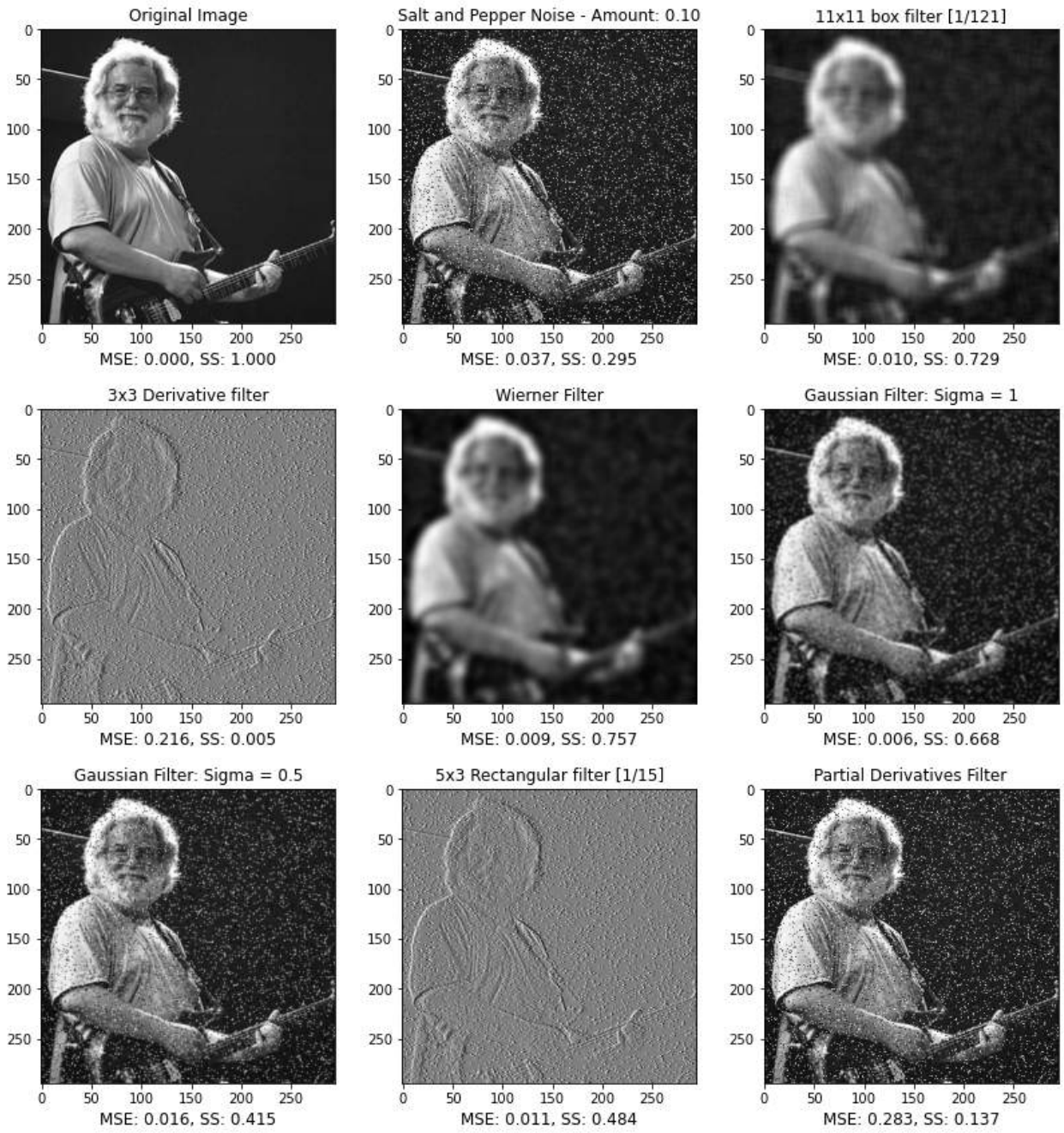

Figure 28: Linear filtering on salt and pepper noise with amount 0.1

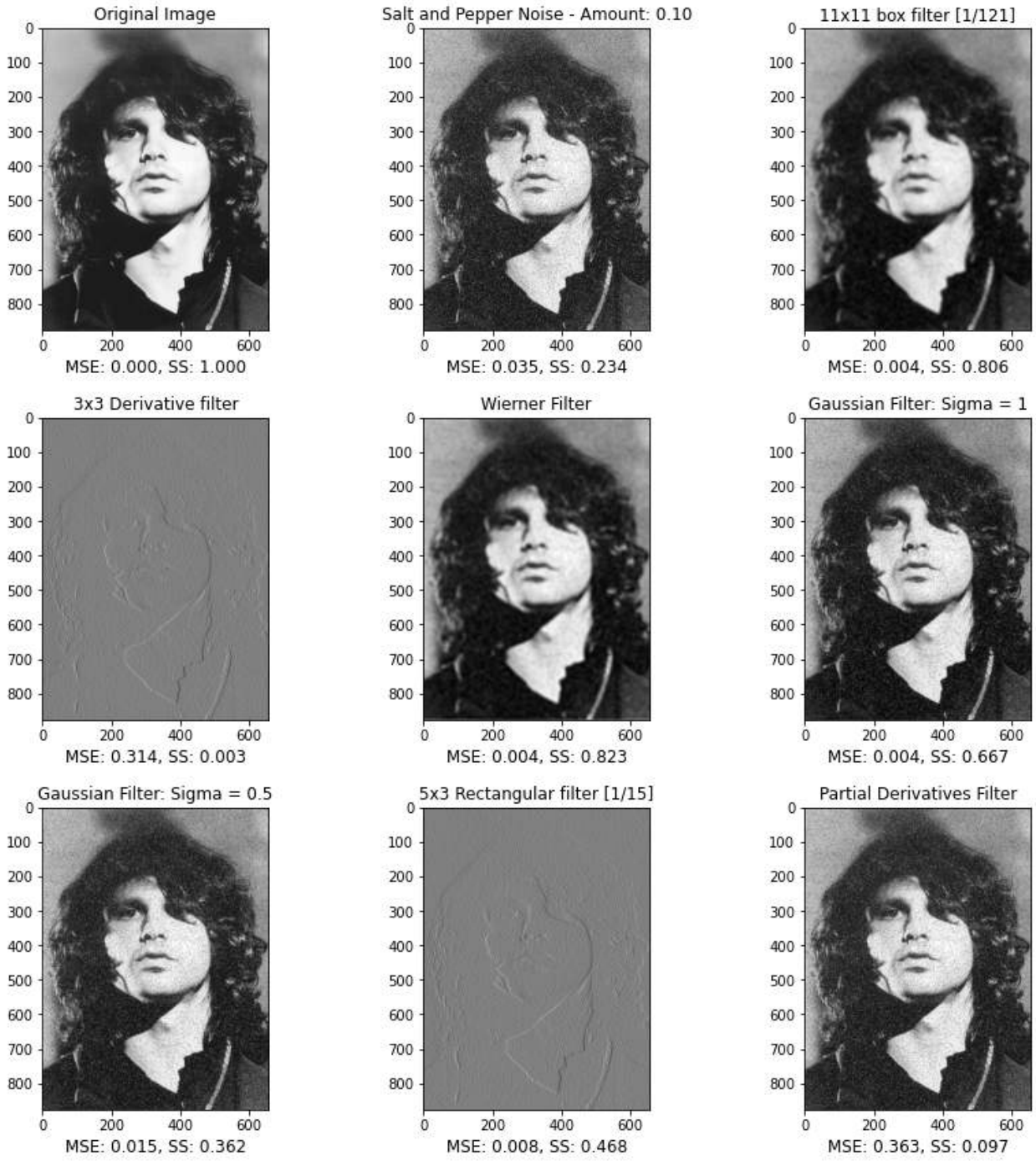

Figure 29: Linear filtering on salt and pepper noise with amount 0.1

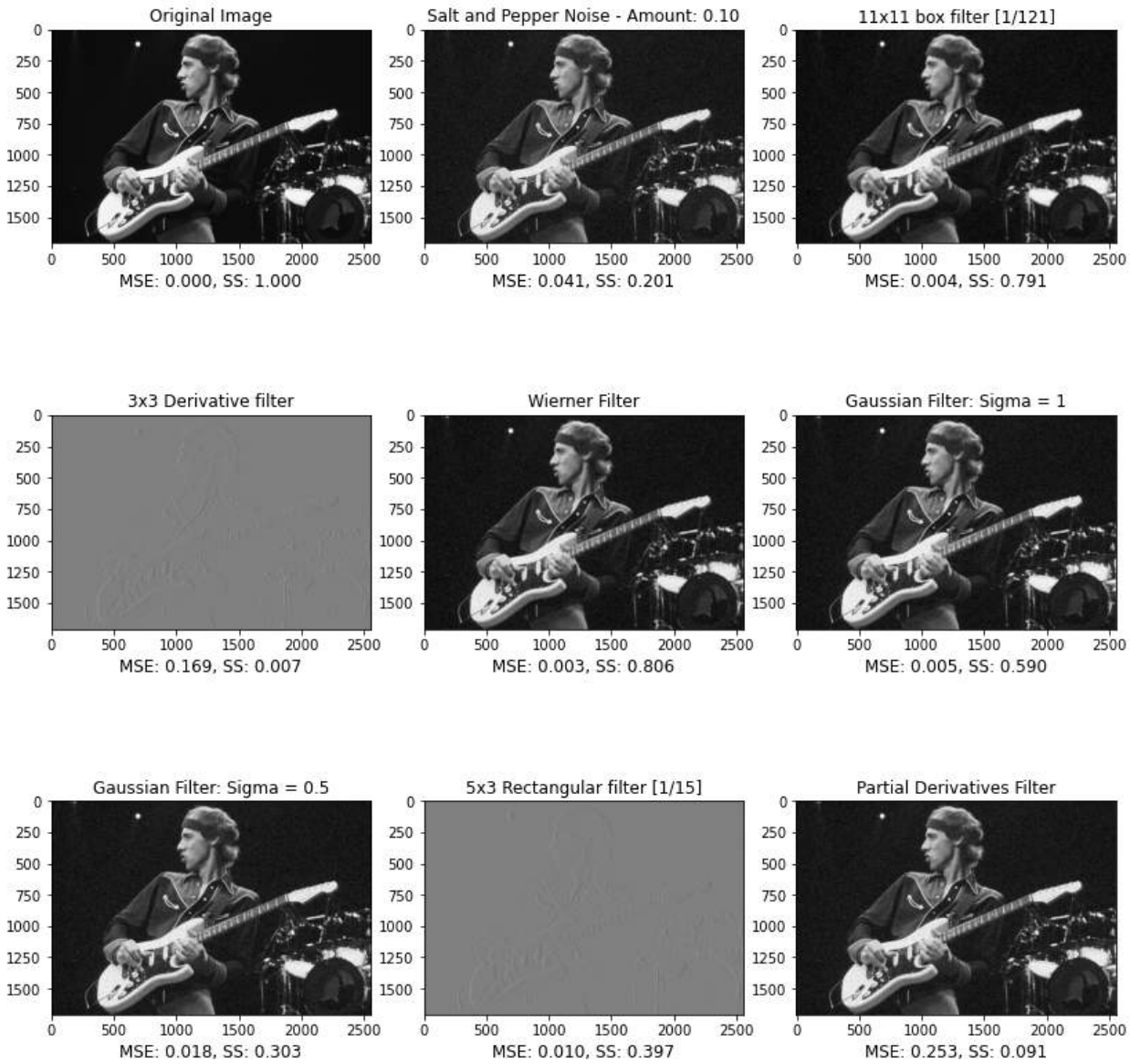

Figure 30: Linear filtering on salt and pepper noise with amount 0.1

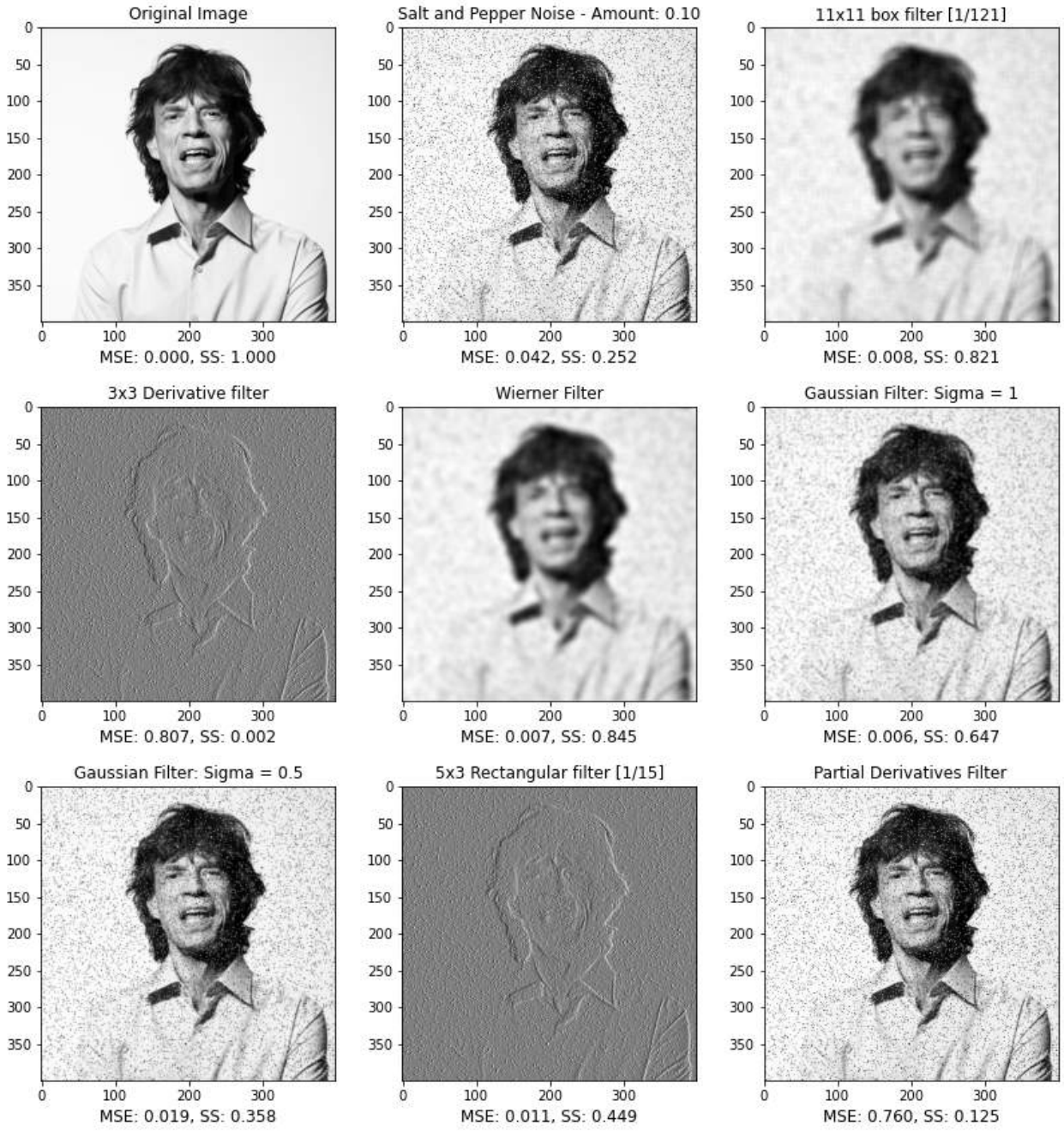

Figure 31: Linear filtering on salt and pepper noise with amount 0.1

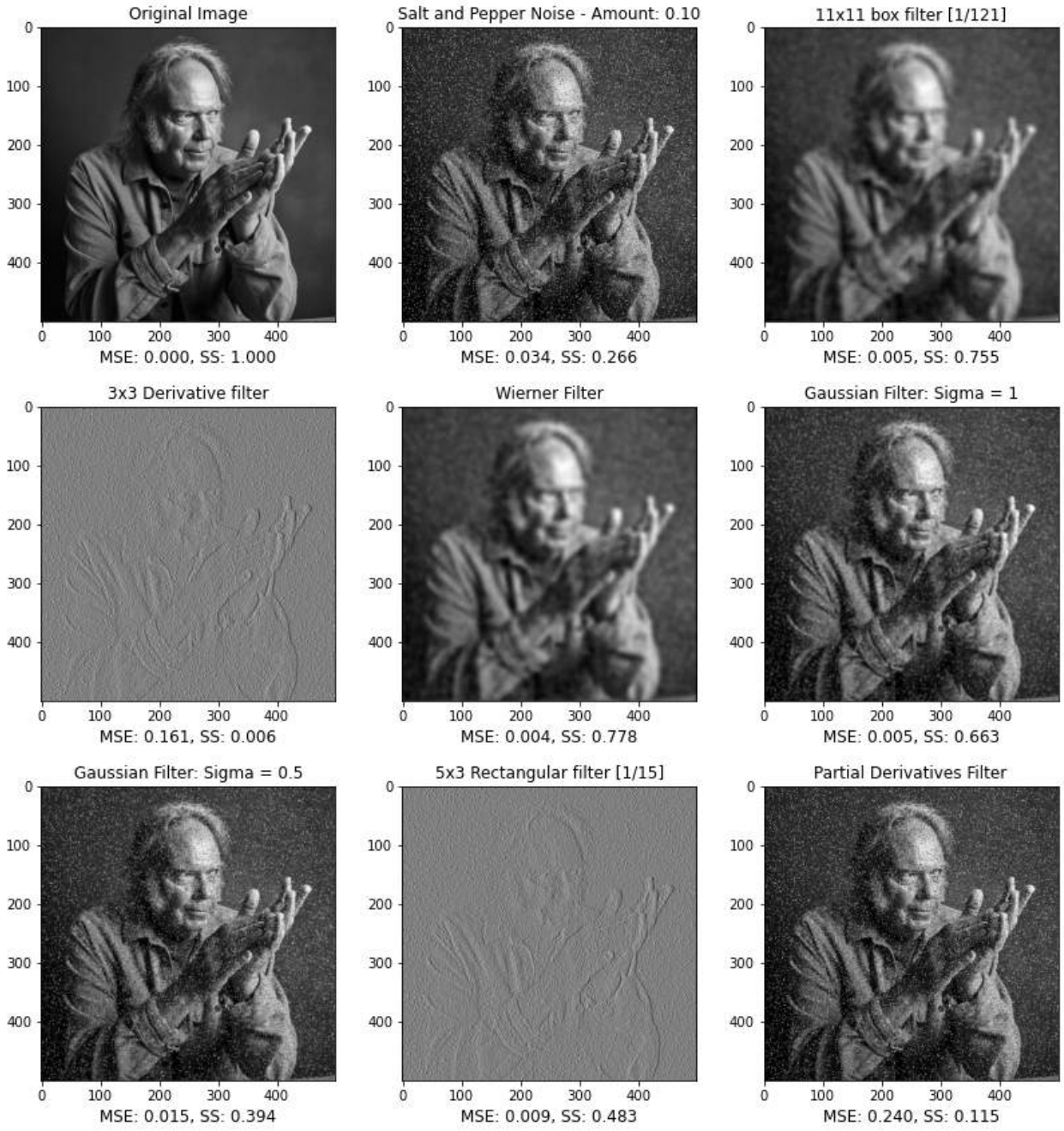

Figure 31: Linear filtering on salt and pepper noise with amount 0.1

With the salt and pepper noise, we see very similar results compared to the gaussian noise. The box filter and wiener filter do the best overall statistical job of denoising our images. The derivative filter and rectangular filter will continue to produce unrecognizable images. The gaussian filter does the best job in decreasing the overall mean squared error but has a lower statistical similarity when comparing to the original image. The important thing to note here is that although it produces a mean squared error value that is less than that of the box and wiener filter, it qualitatively produces a better image that is less blurry. We move on to trial two with a salt and pepper noise amount of 0.25.

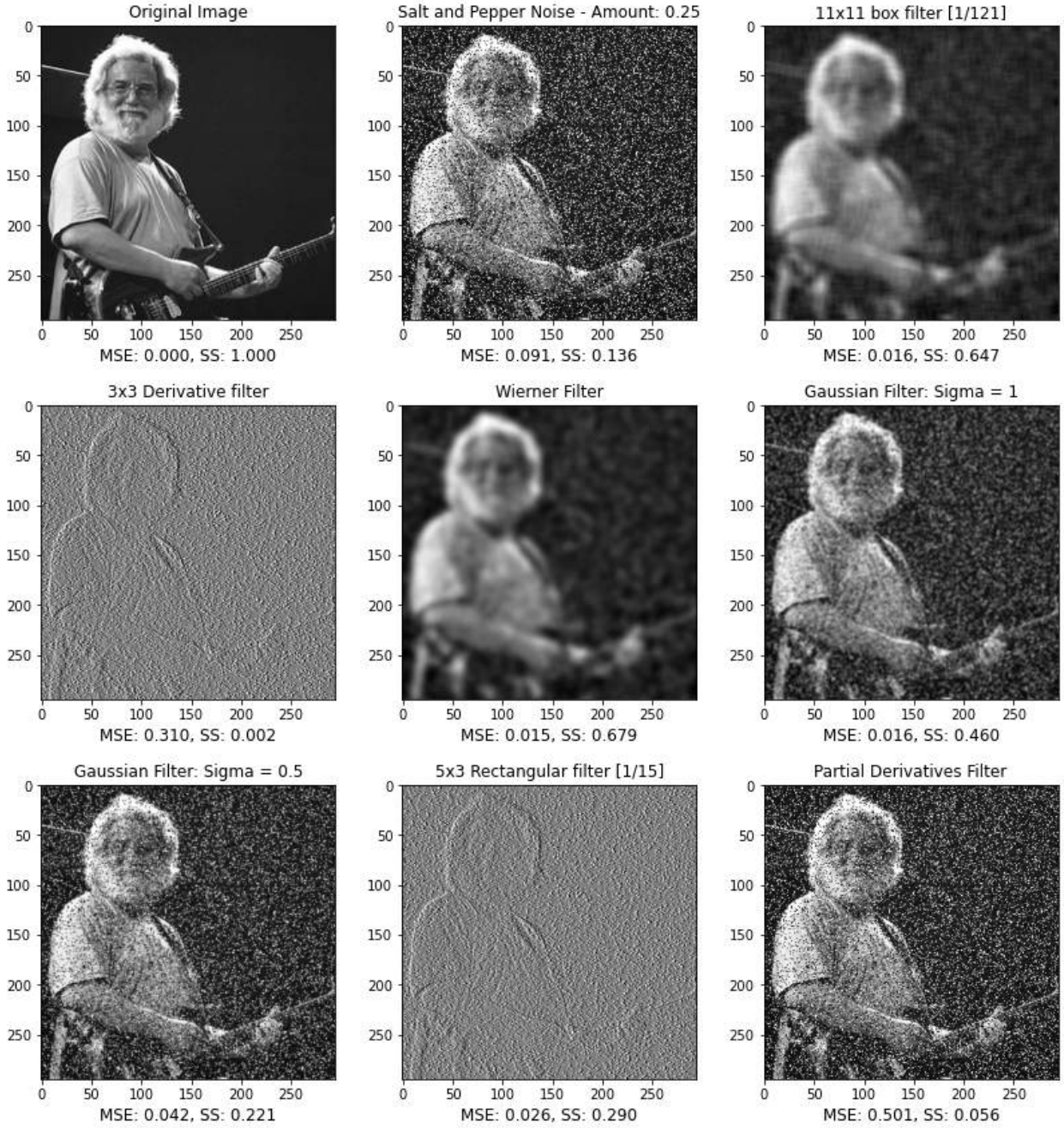

Figure 32: Linear filtering on salt and pepper noise with amount 0.25

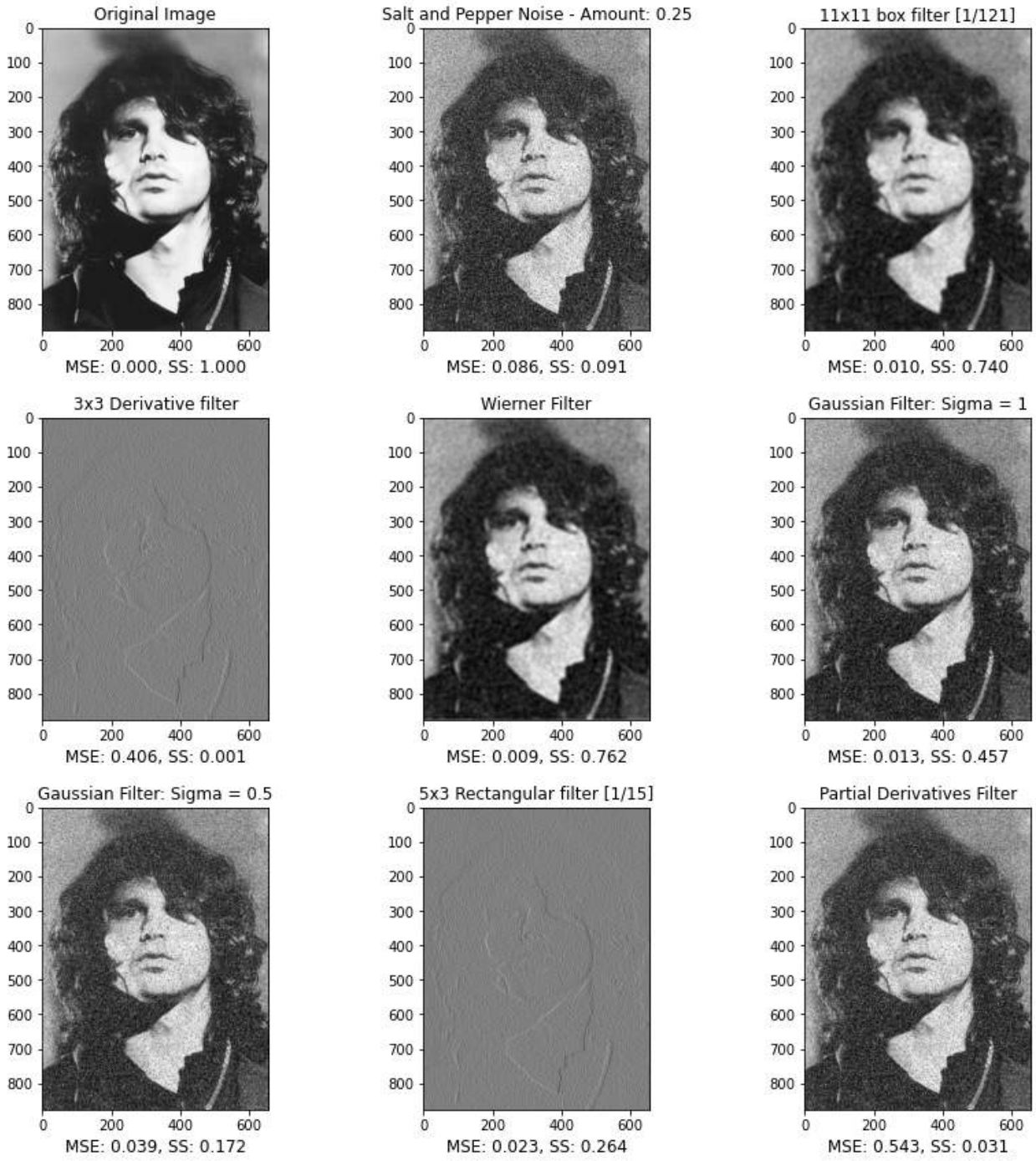

Figure 33: Linear filtering on salt and pepper noise with amount 0.25

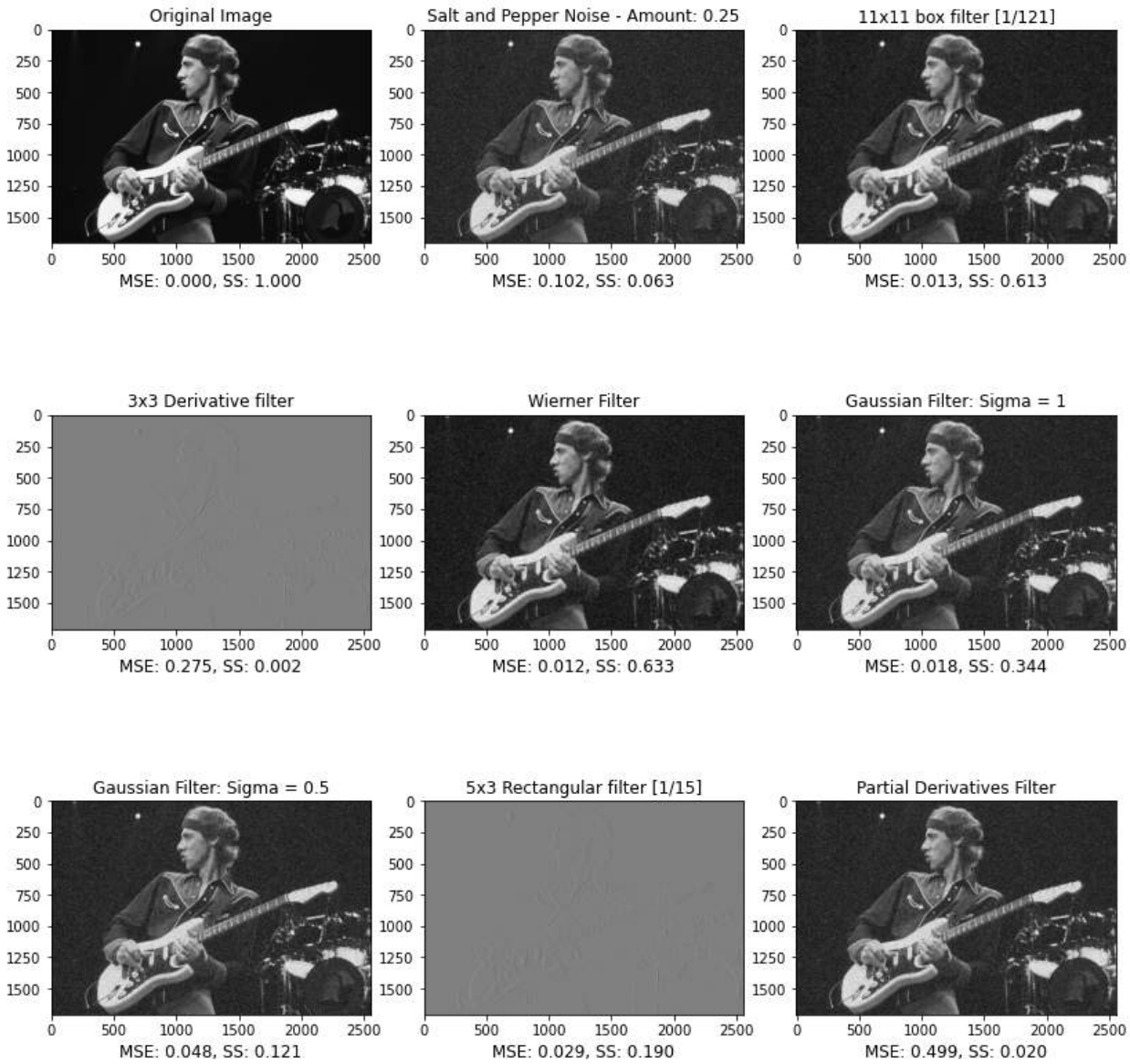

Figure 34: Linear filtering on salt and pepper noise with amount 0.25

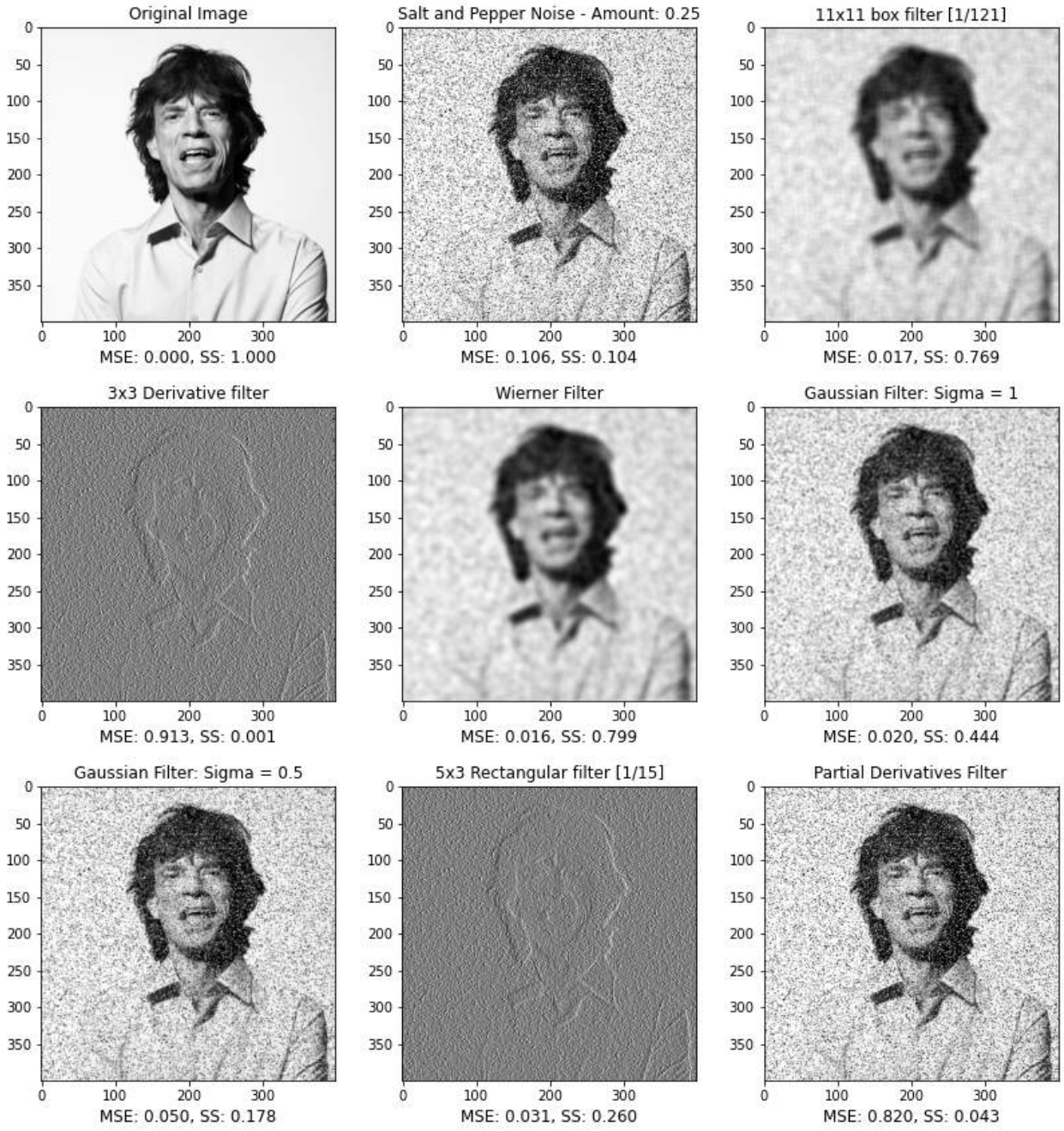

Figure 35: Linear filtering on salt and pepper noise with amount 0.25

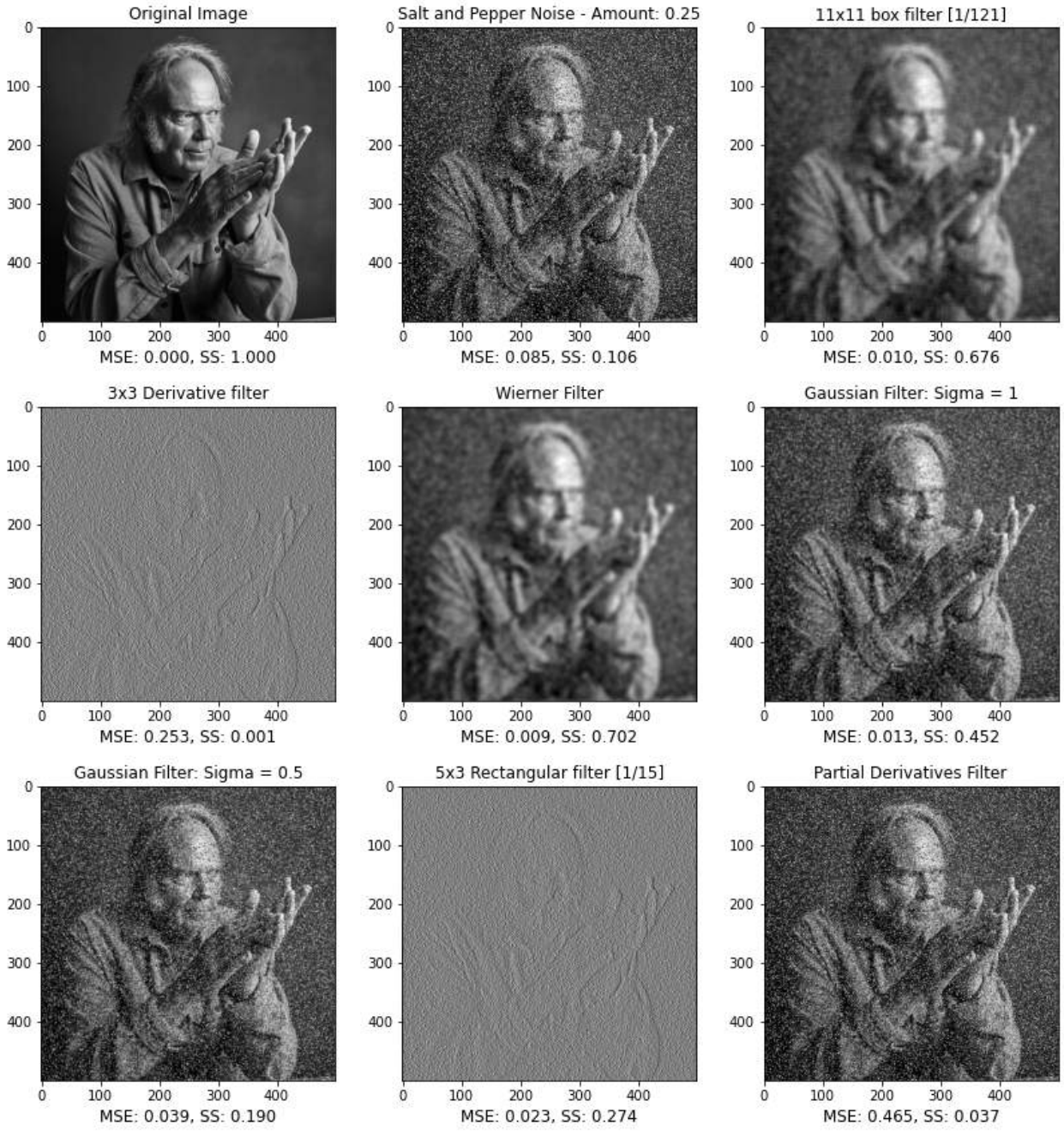

Figure 36: Linear filtering on salt and pepper noise with amount 0.25

With the increase in the amount of salt and pepper noise, we see that the Gaussian filter with sigma equal to one is no longer the best statistical denoiser of the linear filters. The box filter and wiener filter are now, again, the best statistical denoisers and are very similar in their ability to denoise while greatly increasing the blurriness of the noisy image. Trial 3 with salt and pepper amount of 0.5.

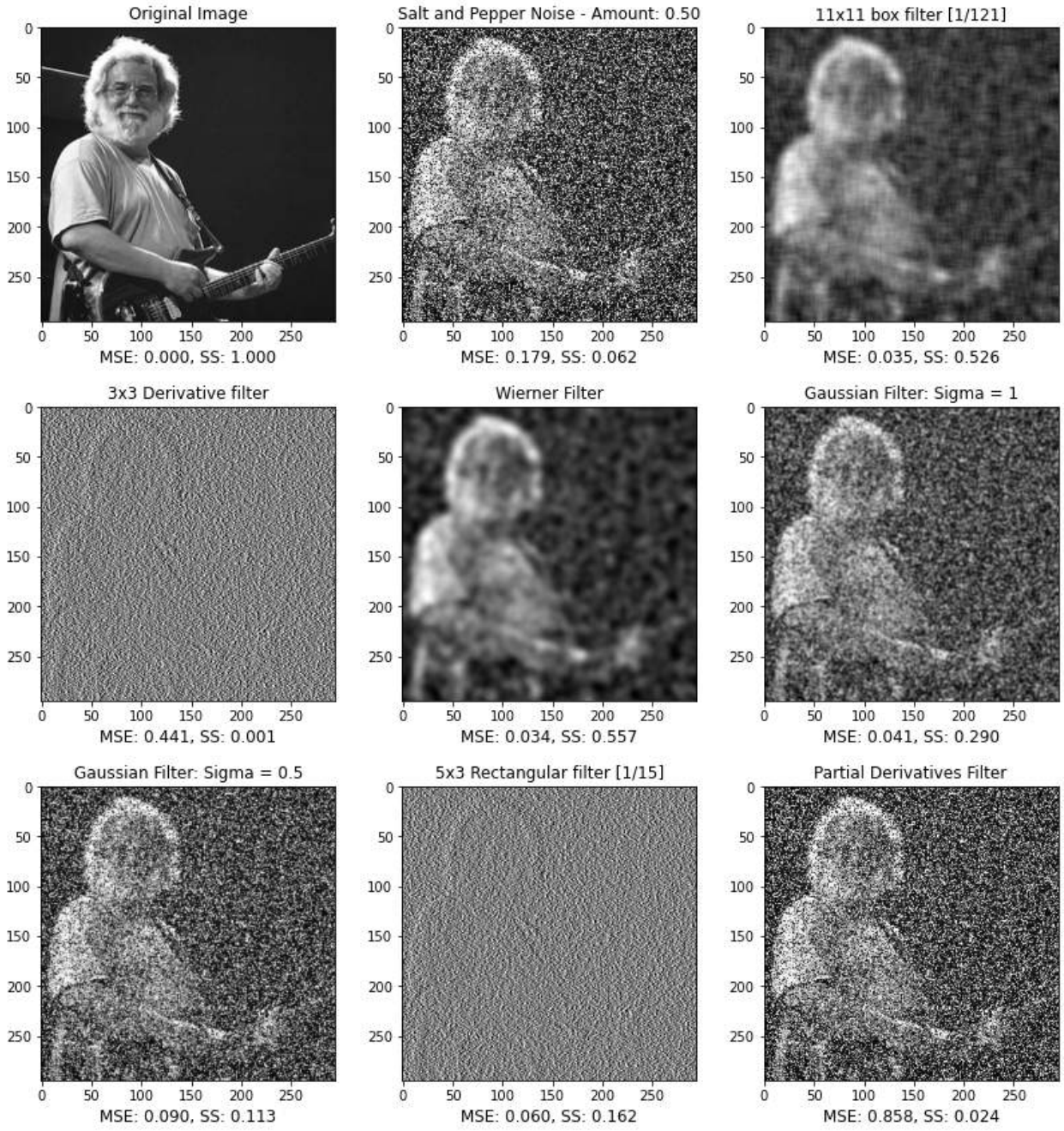

Figure 37: Linear filtering on salt and pepper noise with amount 0.5

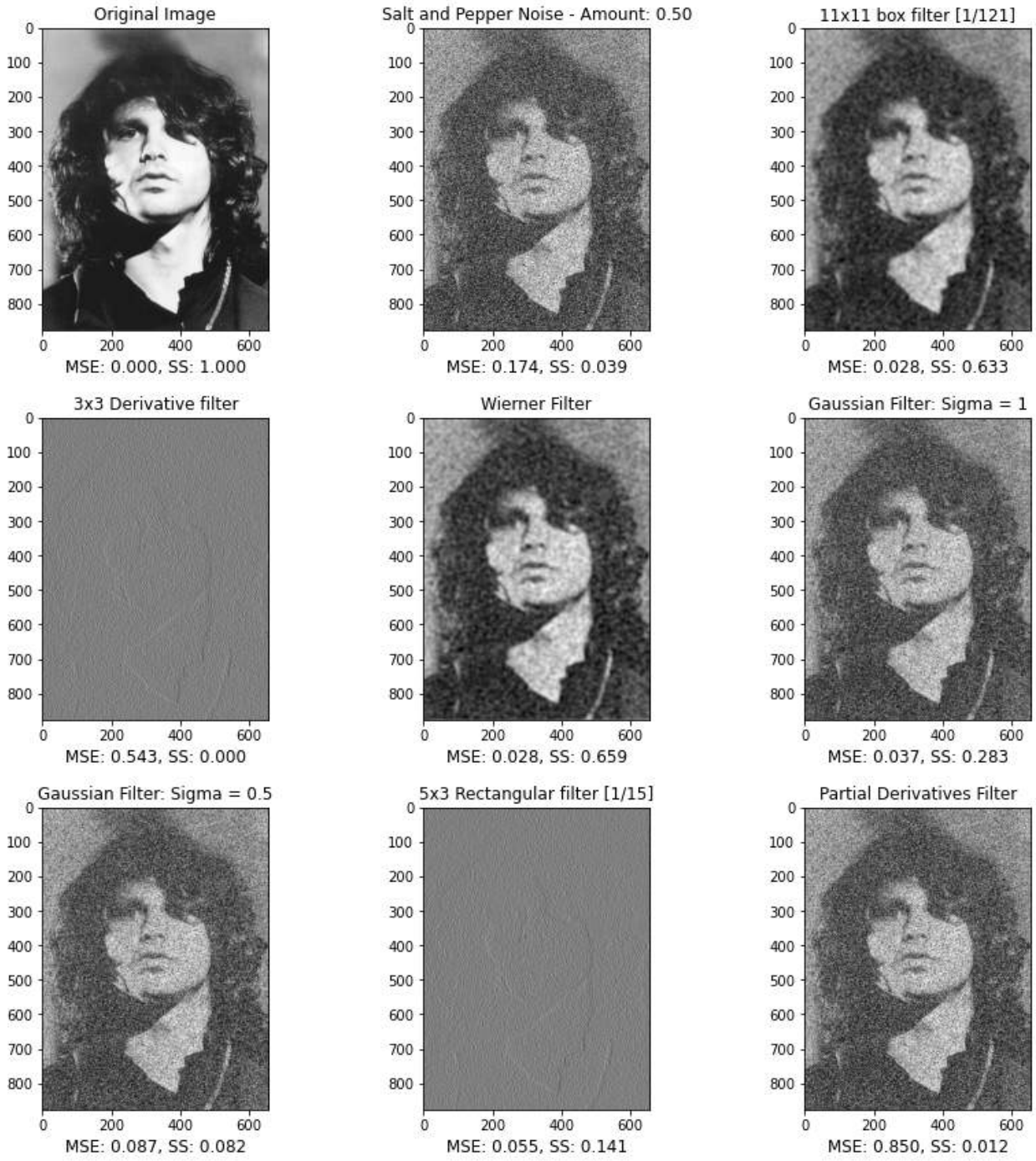

Figure 38: Linear filtering on salt and pepper noise with amount 0.5

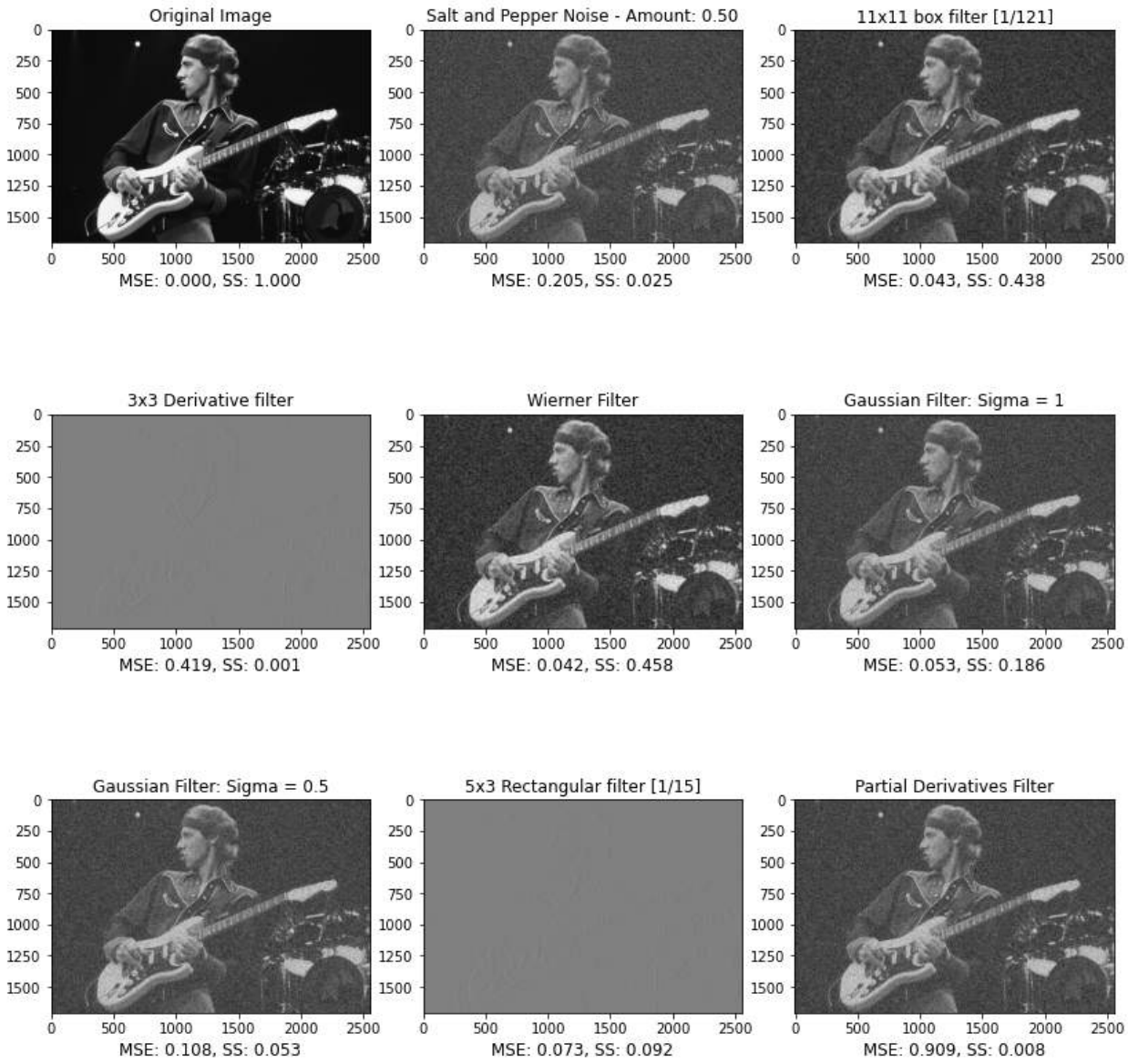

Figure 39: Linear filtering on salt and pepper noise with amount 0.5

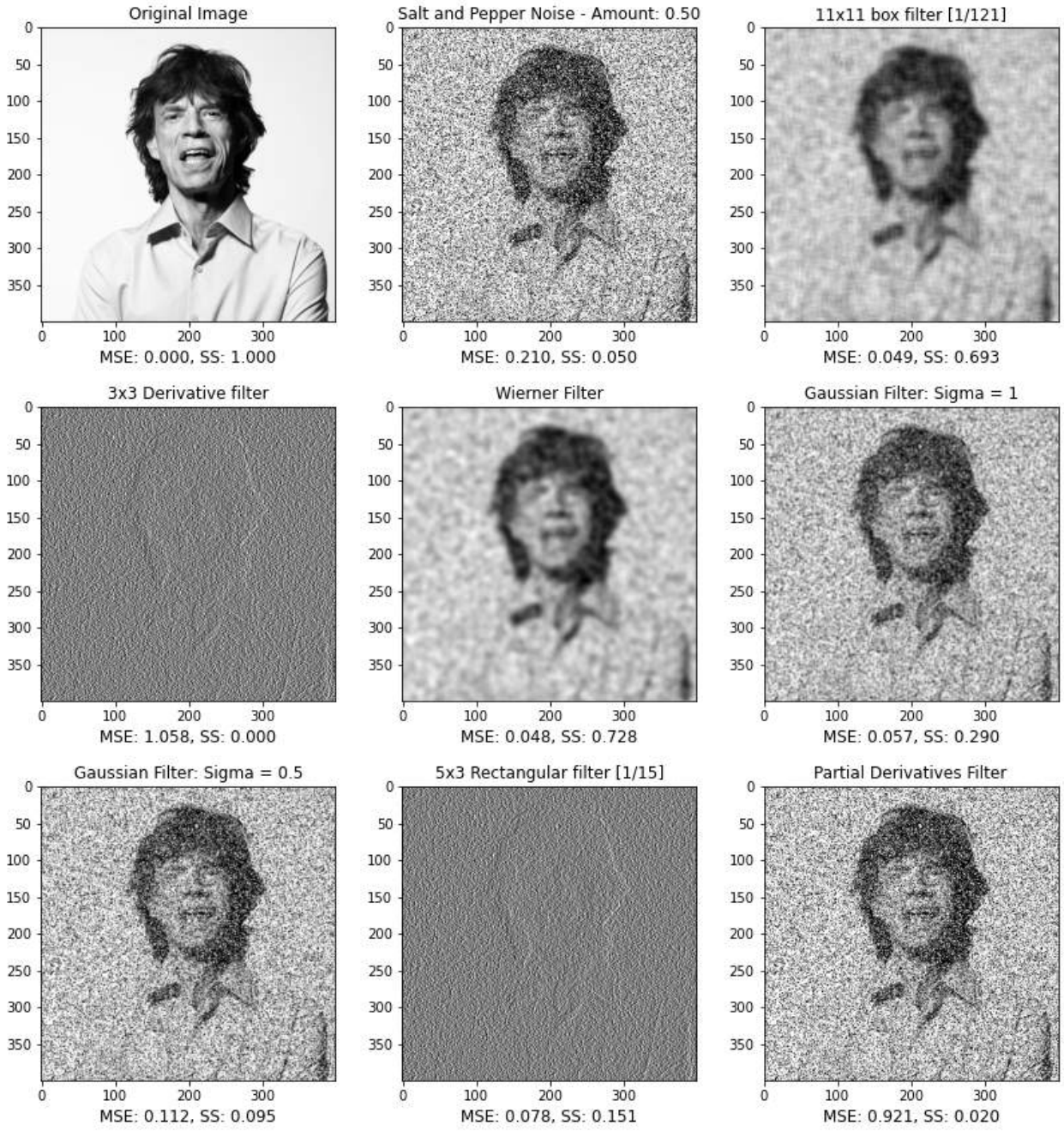

Figure 40: Linear filtering on salt and pepper noise with amount 0.5

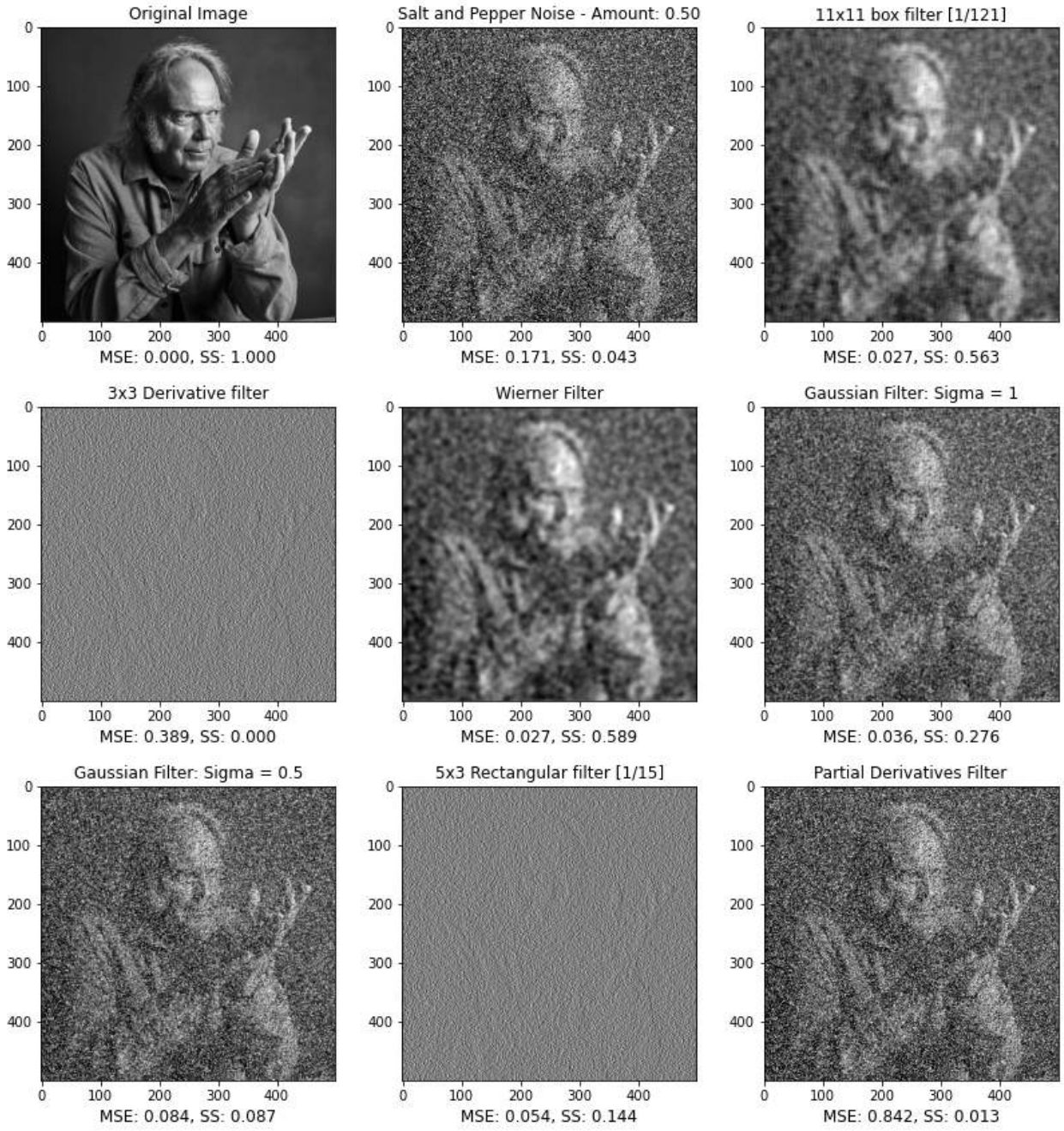

Figure 41: Linear filtering on salt and pepper noise with amount 0.5

In trial three, we see results very similar to that of trial two, with the box filter and wiener filter again producing the least statically noisy images but producing a large amount of blurring. We now move on to poisson noise:

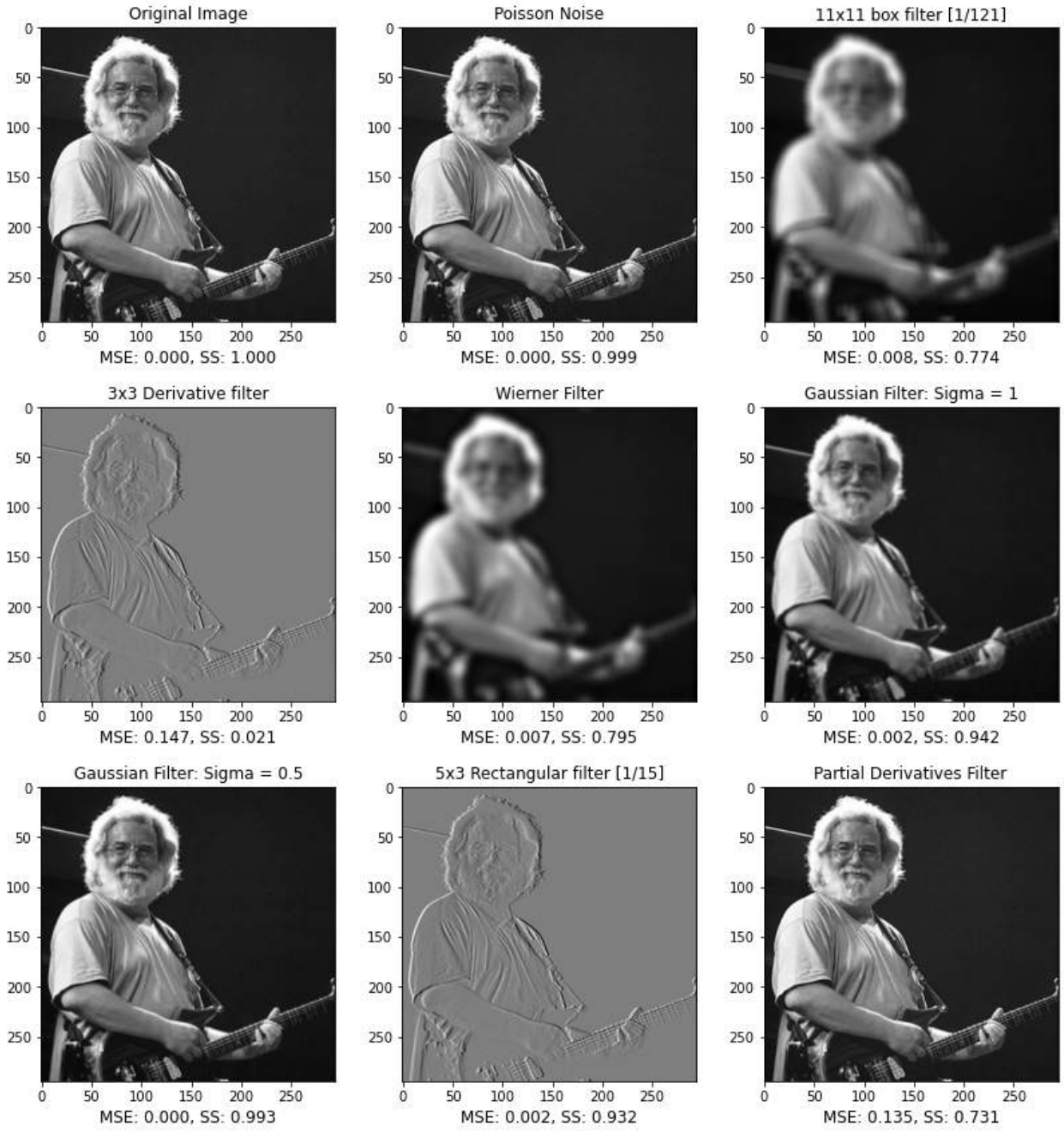

Figure 42: Linear filtering on poisson noise

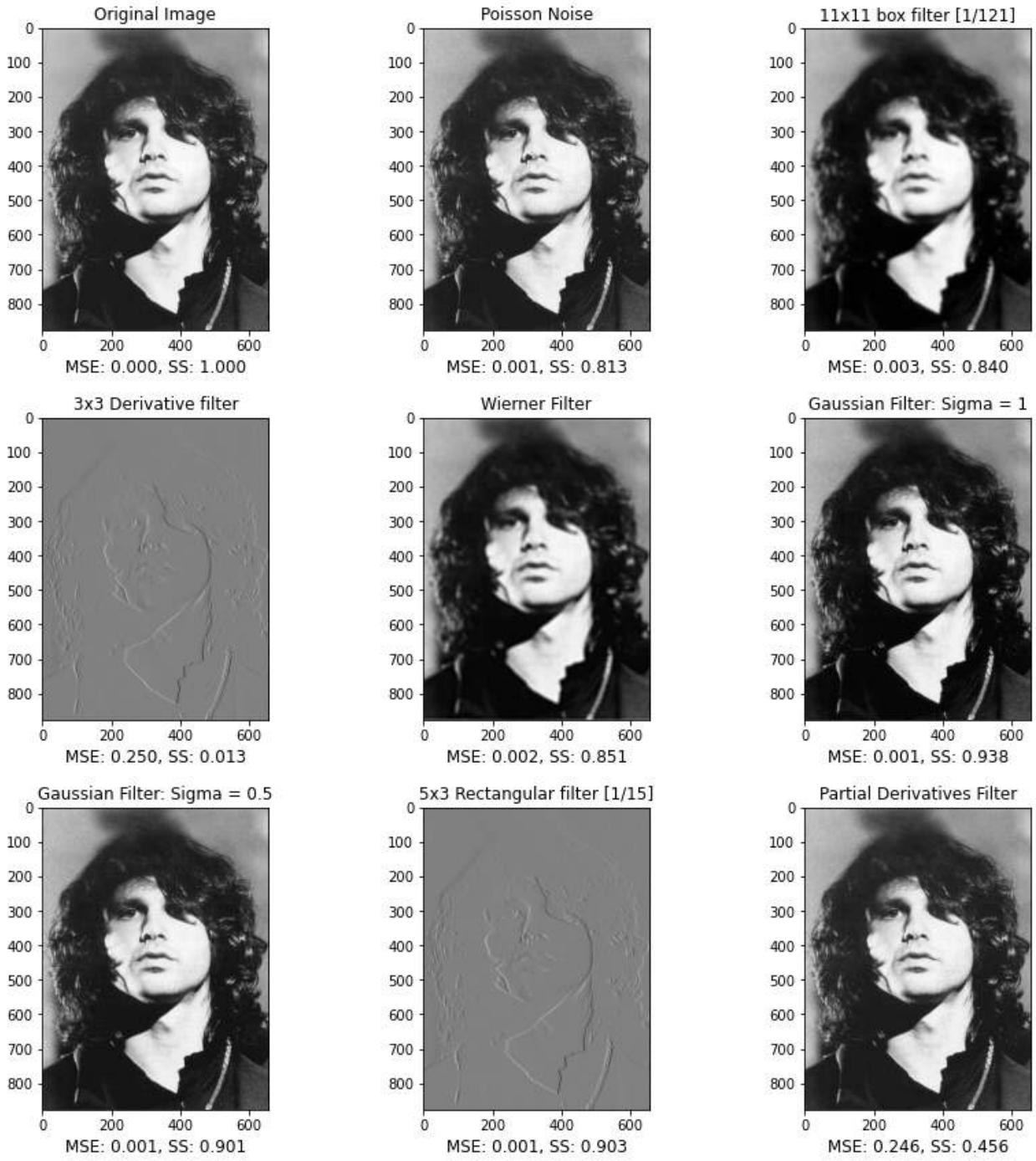

Figure 43: Linear filtering on poisson noise

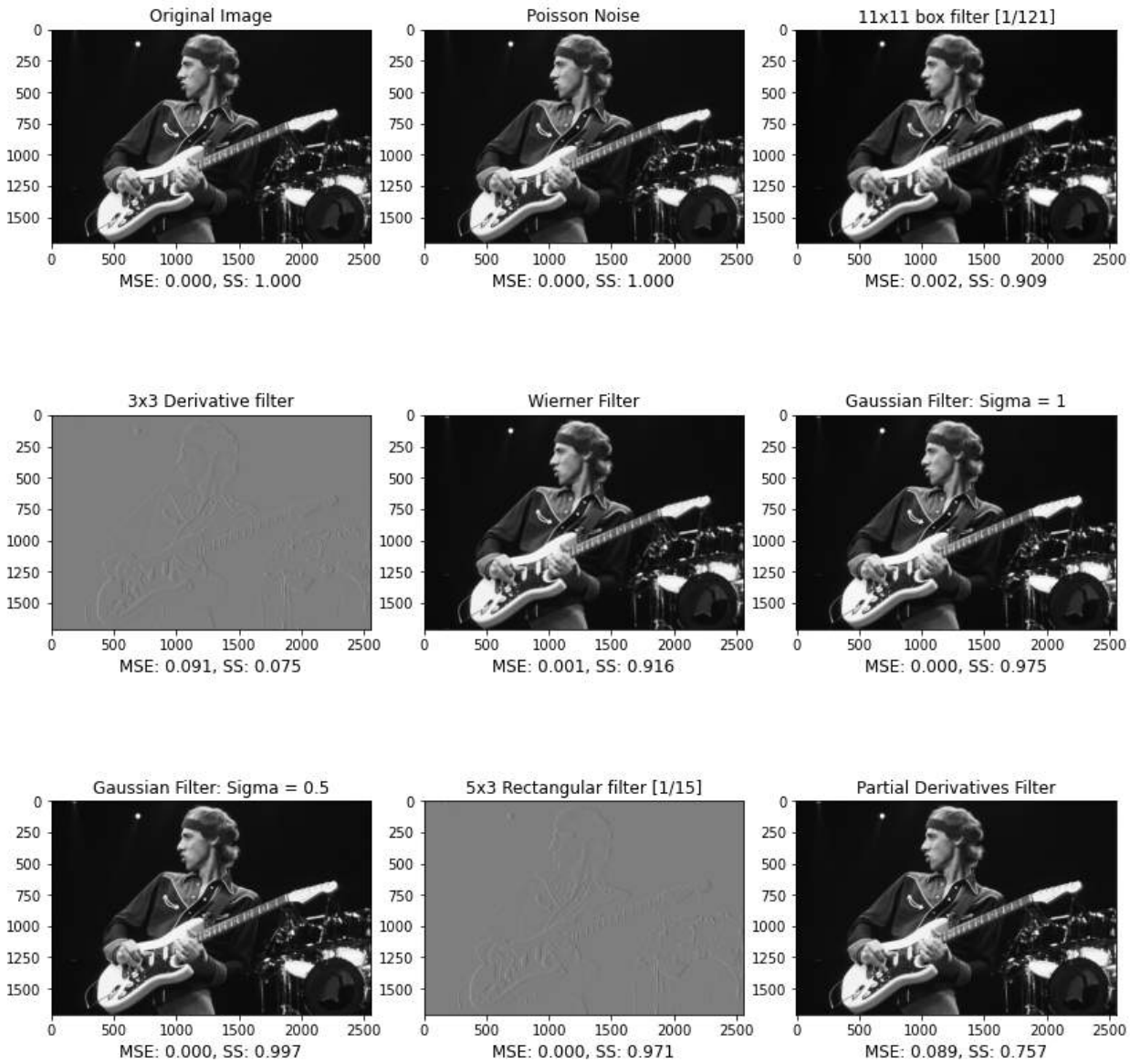

Figure 44: Linear filtering on poisson noise

Figure 45: Linear filtering on poisson noise

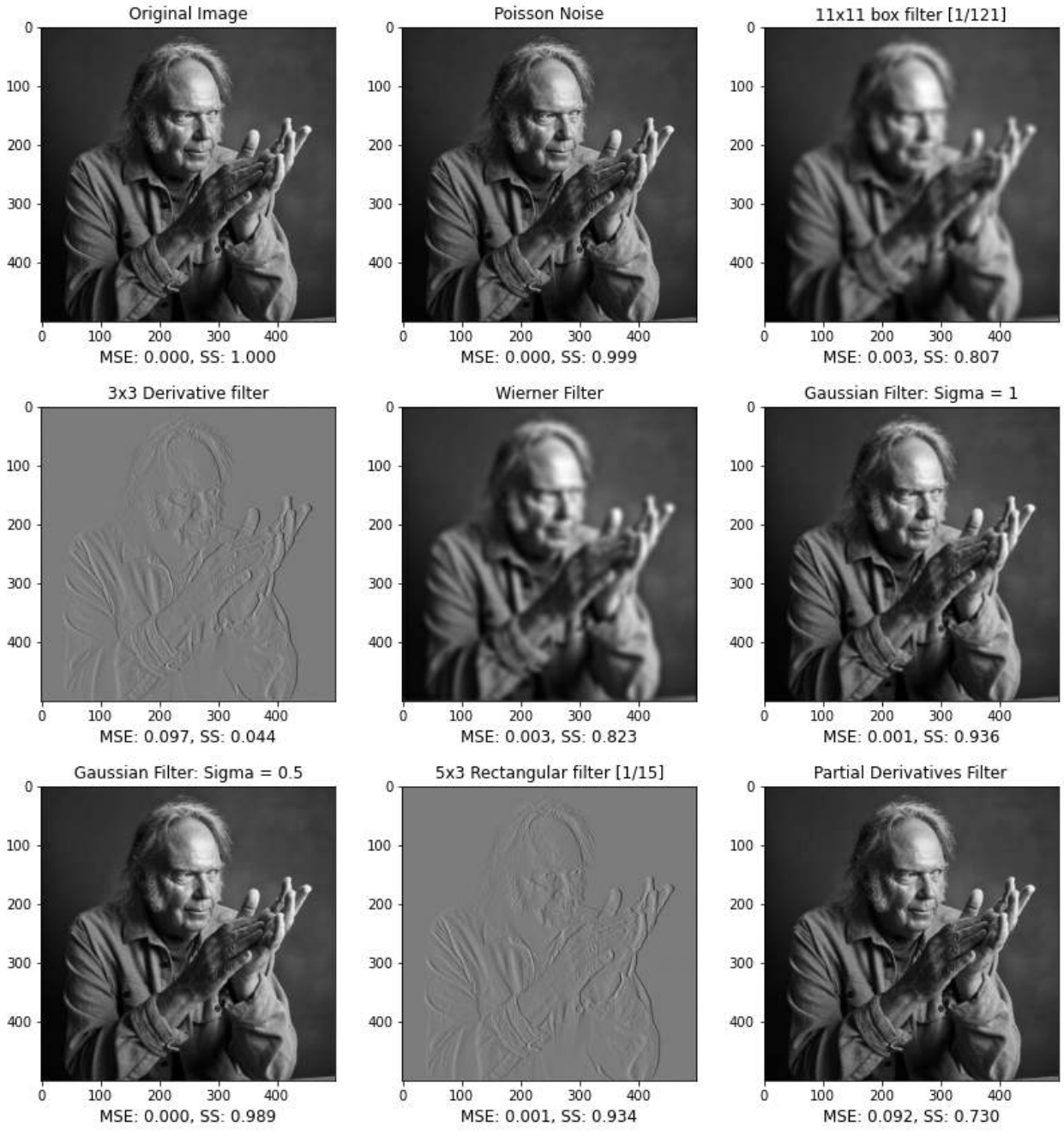

Figure 46: Linear filtering on poisson noise

The amount of noise added to the original images for poisson is very low. In analyzing Figure 46 and 44, we see that the noisy image actually has a very low mean squared error value and a very high structural similarity compared to the original image. All of the filters thus actually add noise to the noisy image and none of the filters produce less noisy images. The rectangular filter does a really good job of finding the edges of the image and thus produces high structural similarity values with interestingly low mean squared error values. In Figure 42 we actually see the

Gaussian filter with lower sigma outperform the Gaussian filter with a higher sgma value. The box and wiener filter continue to produce blurry images. Overall, the analysis of the filters on the poisson noise cannot be taken into consideration in analyzing the filters overall ability to denoise images due to the sheer lack of noise that is being added to the original images. We continue analyzing the linear filters via the addition of our last type of noise: speckled.

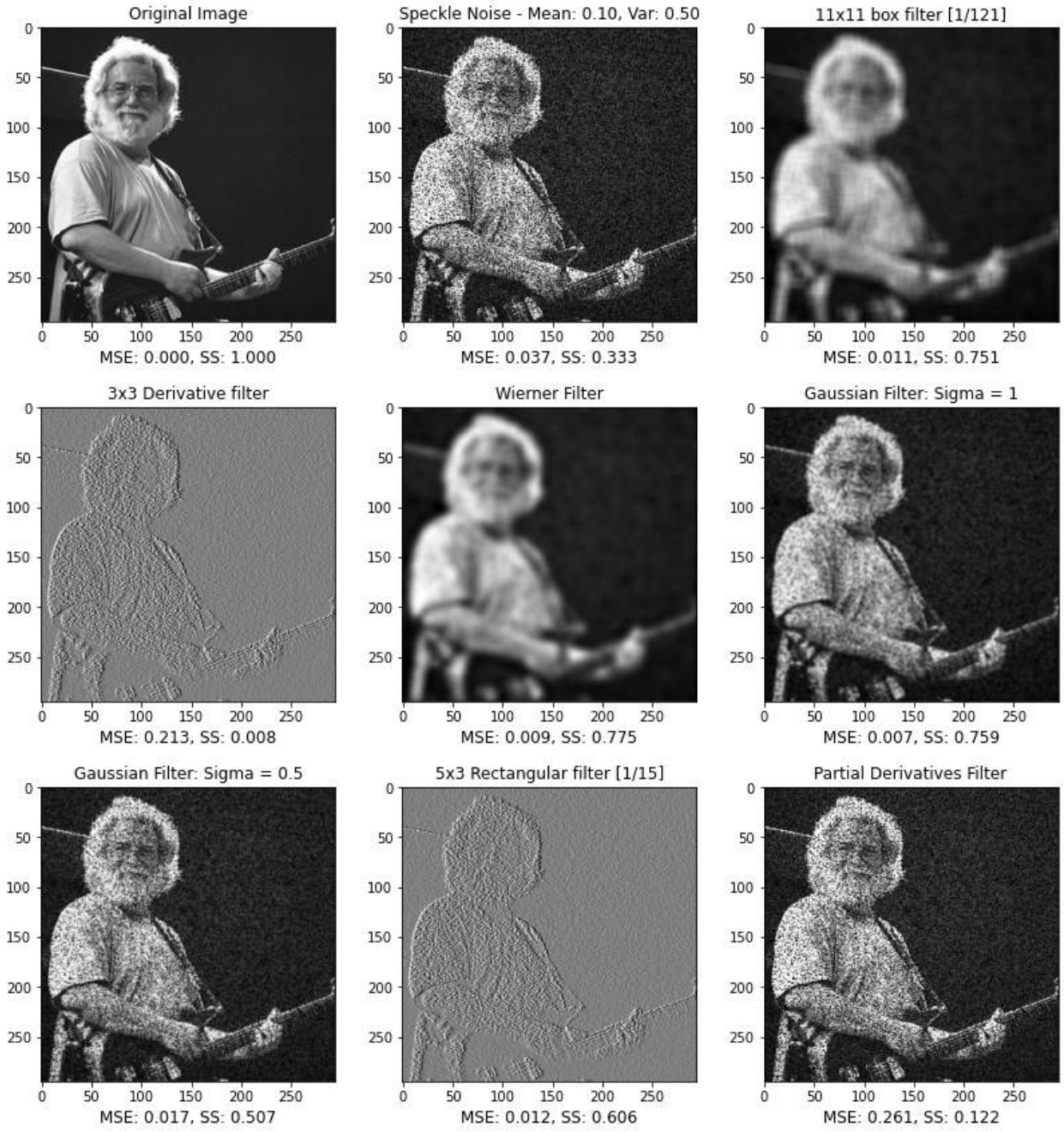

Figure 47: Linear filtering on speckled noise, mean of 0.1 and variance of 0.5

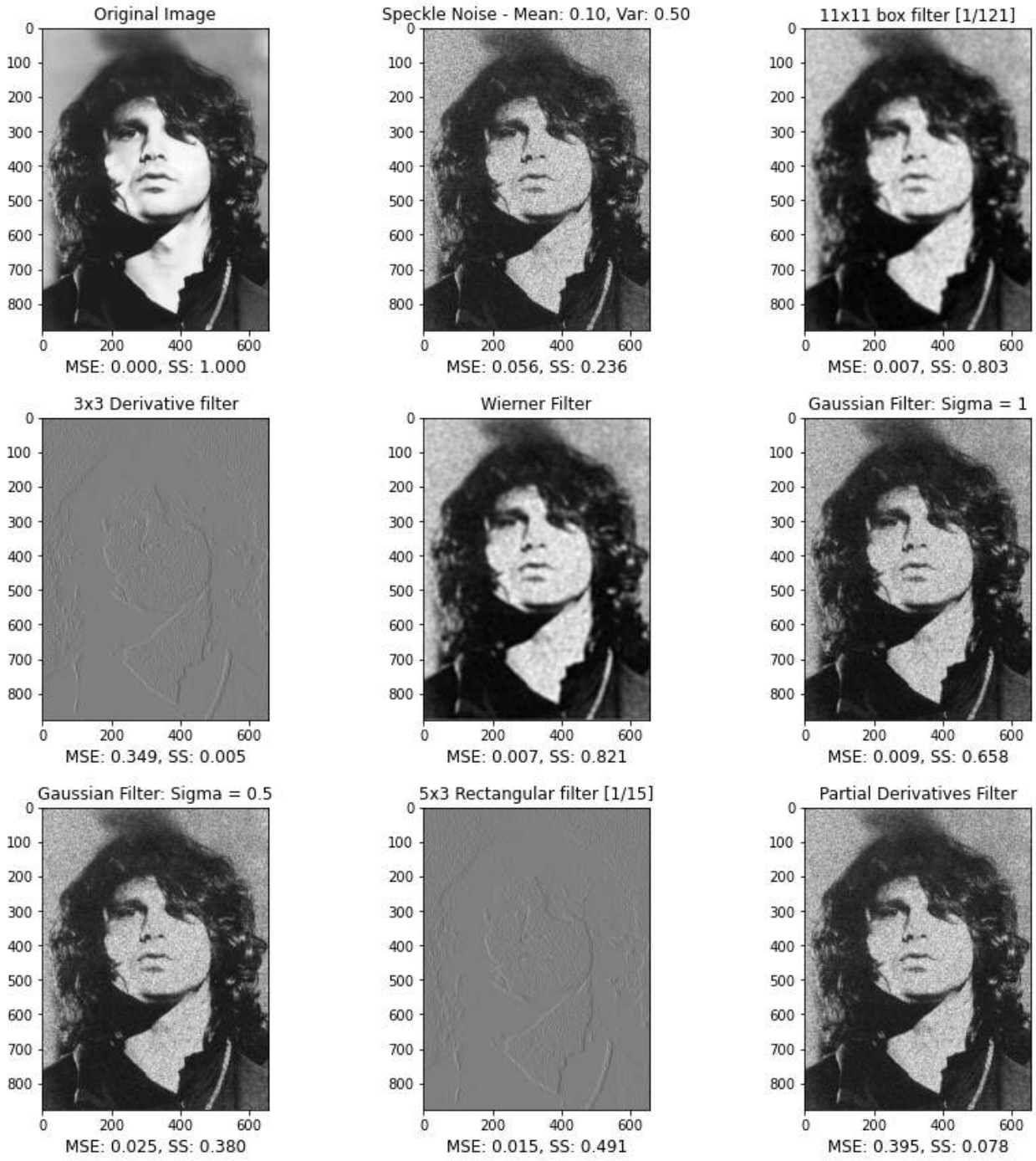

Figure 48: Linear filtering on speckled noise, mean of 0.1 and variance of 0.5

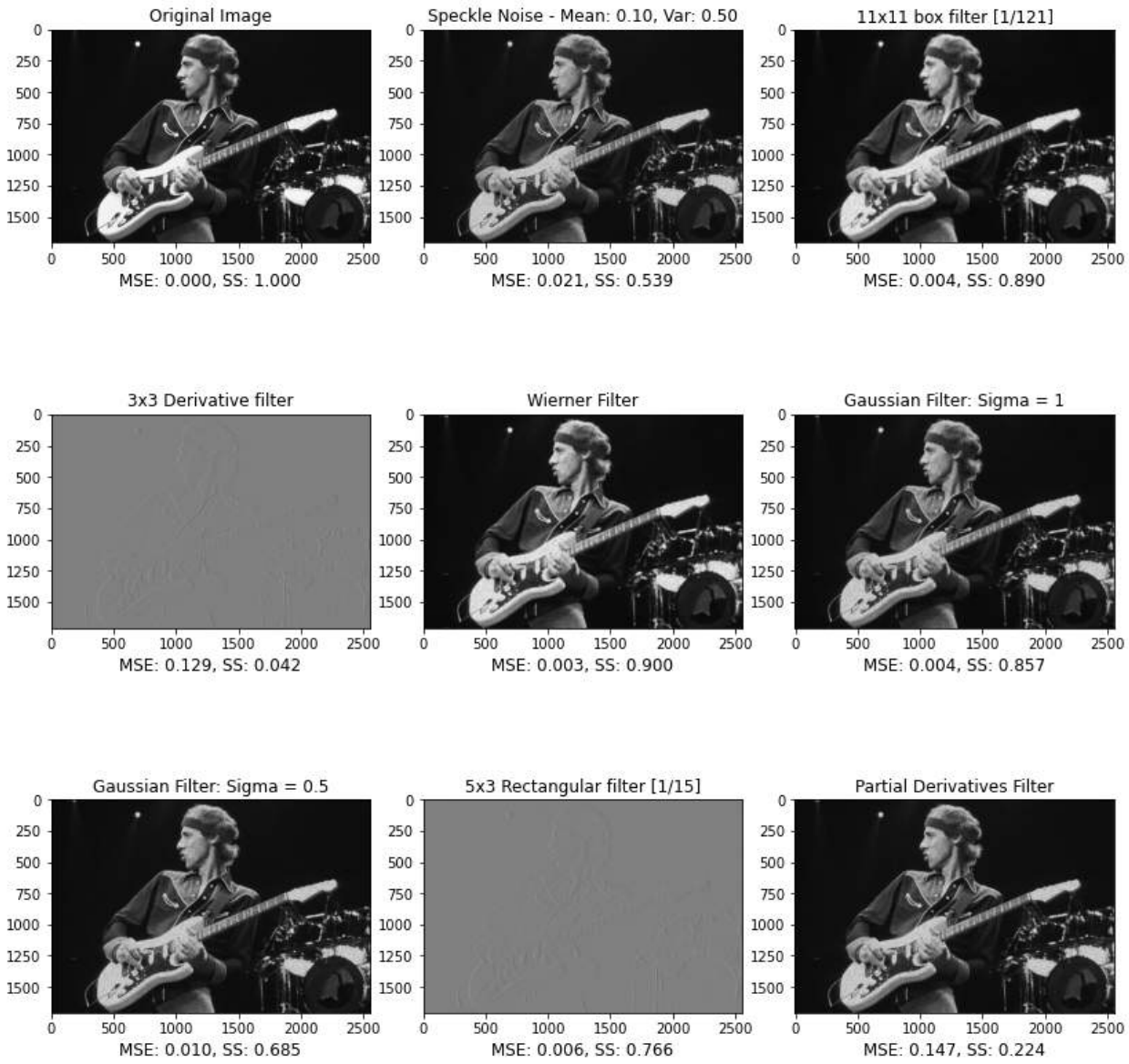

Figure 49: Linear filtering on speckled noise, mean of 0.1 and variance of 0.5

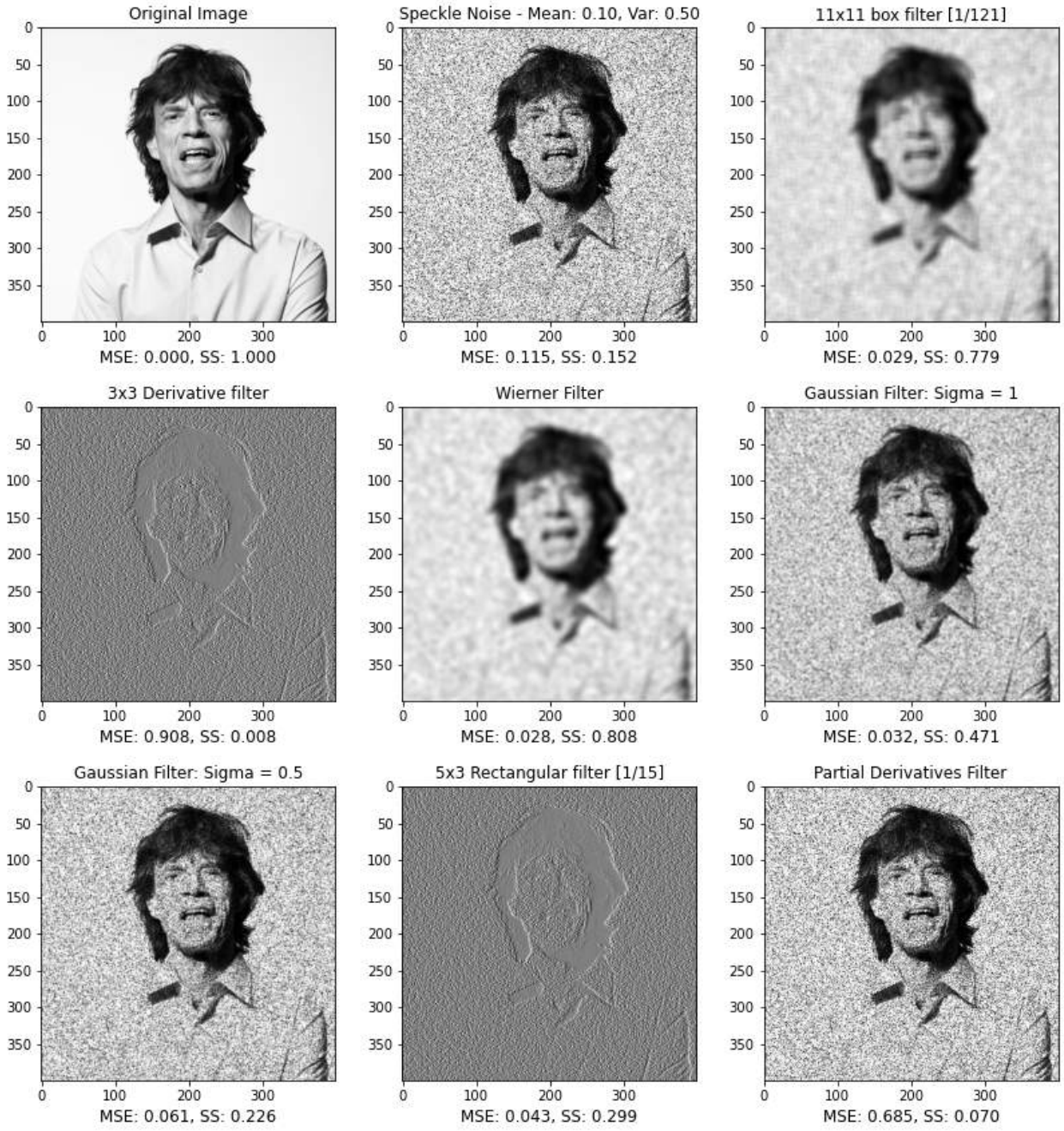

Figure 50: Linear filtering on speckled noise, mean of 0.1 and variance of 0.5

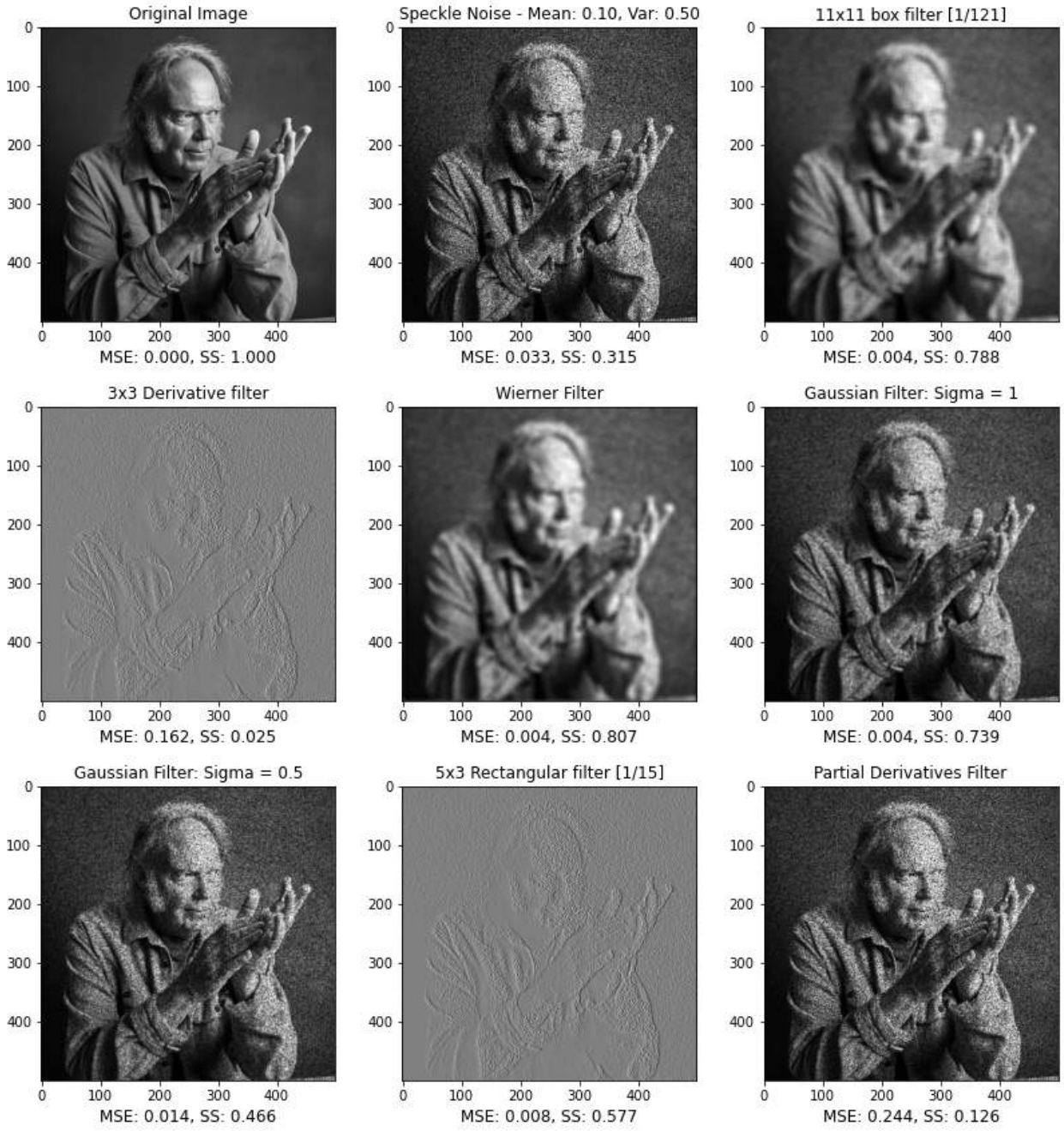

Figure 51: Linear filtering on speckled noise, mean of 0.1 and variance of 0.5

After viewing the results to trial 1 of the speckled noise, we see very similar results to previous noise types. The box filter and wiener filter continue to produce the best results with the gaussian filter with a high sigma being very comparable but with less overall blurring. The rectangular filter, although producing an image that is unrecognizable compared to the original, does a good job of pulling out the edges and thus still produces a structural similarity value that is higher than one would expect after looking at the image. The gaussian filter with higher sigma outperforms the gaussian filter with the lower sigma value. We move on to trial 2:

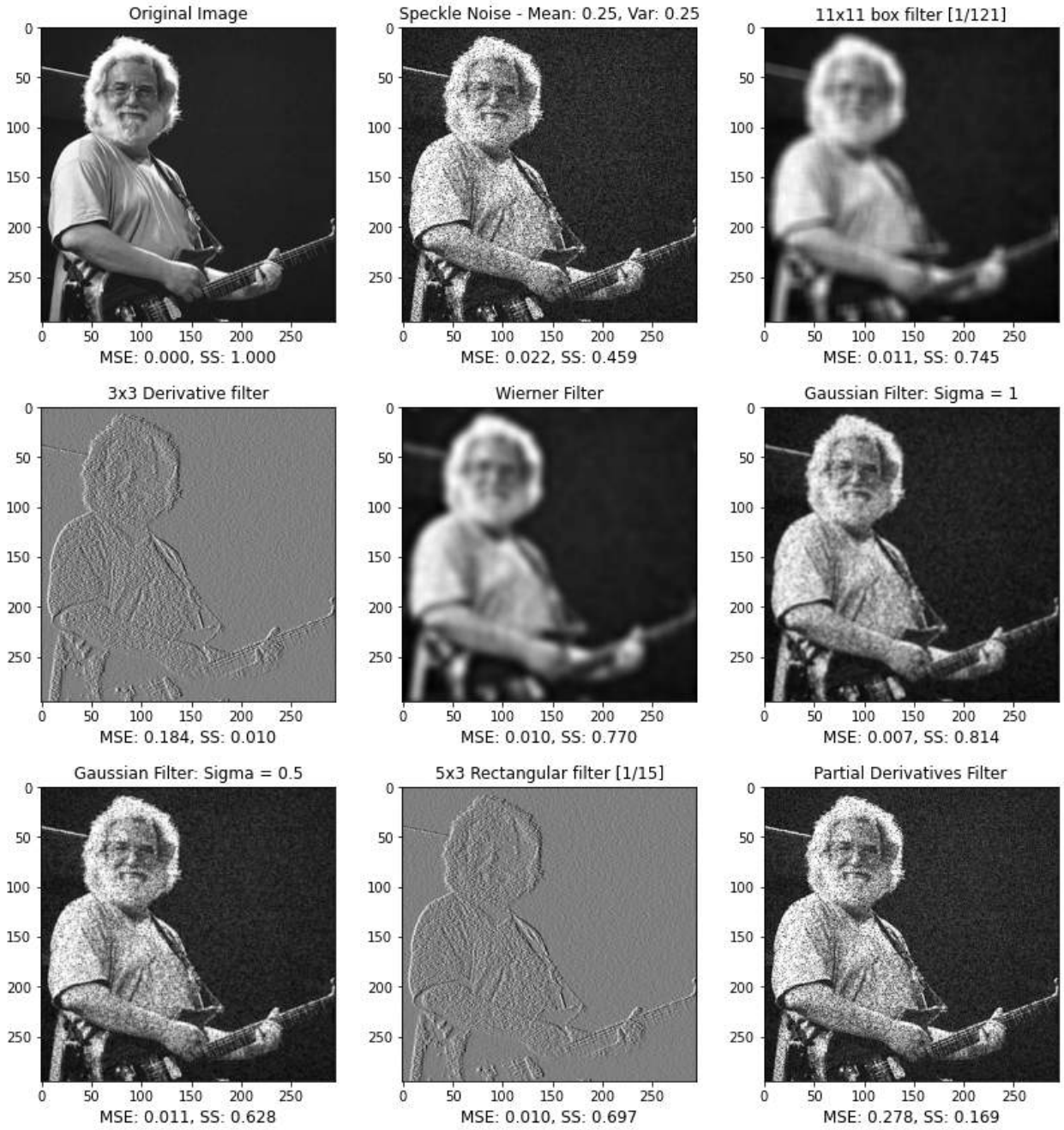

Figure 52: Linear filtering on speckled noise, mean of 0.25 and variance of 0.25

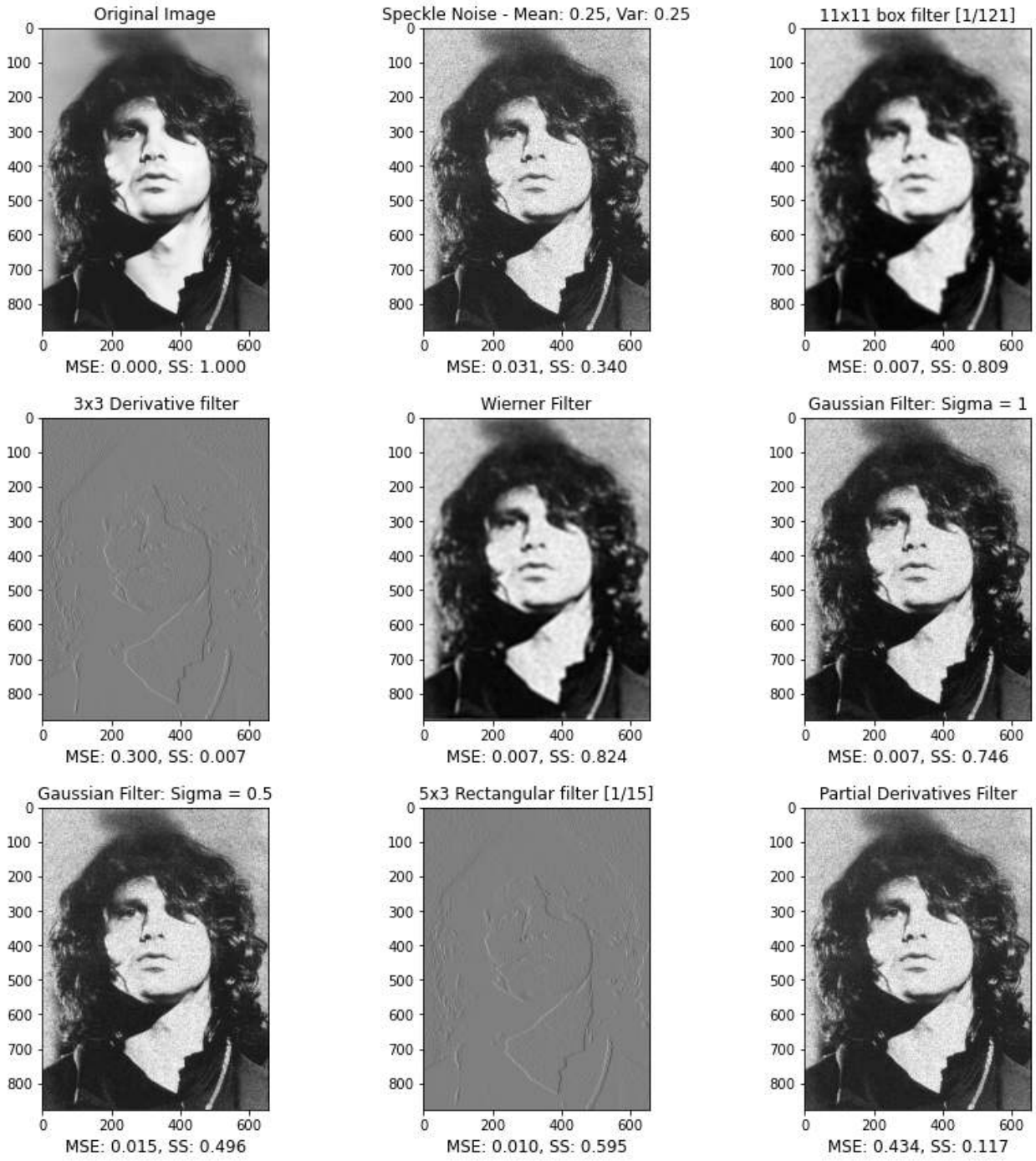

Figure 53: Linear filtering on speckled noise, mean of 0.25 and variance of 0.25

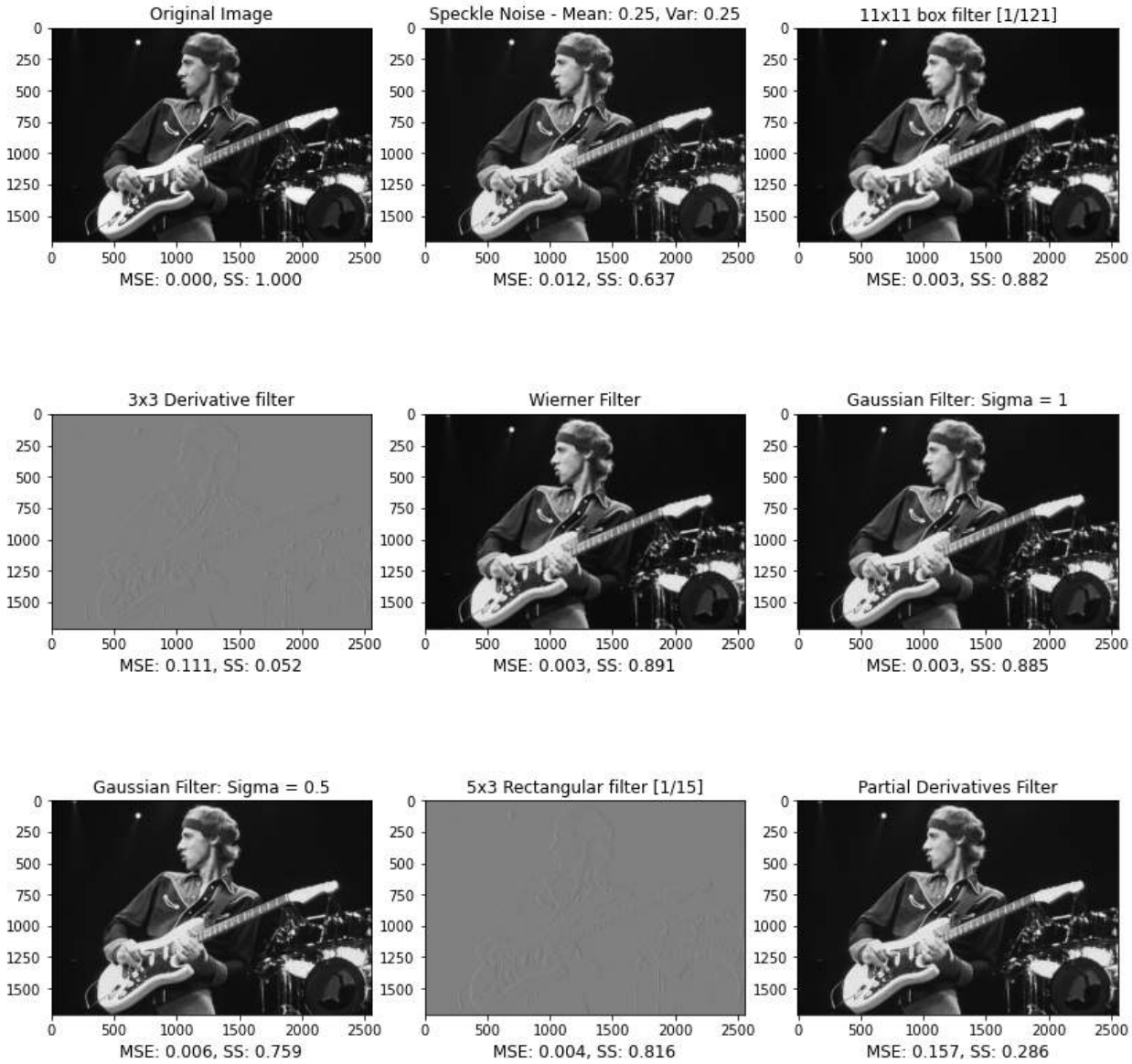

Figure 54: Linear filtering on speckled noise, mean of 0.25 and variance of 0.25

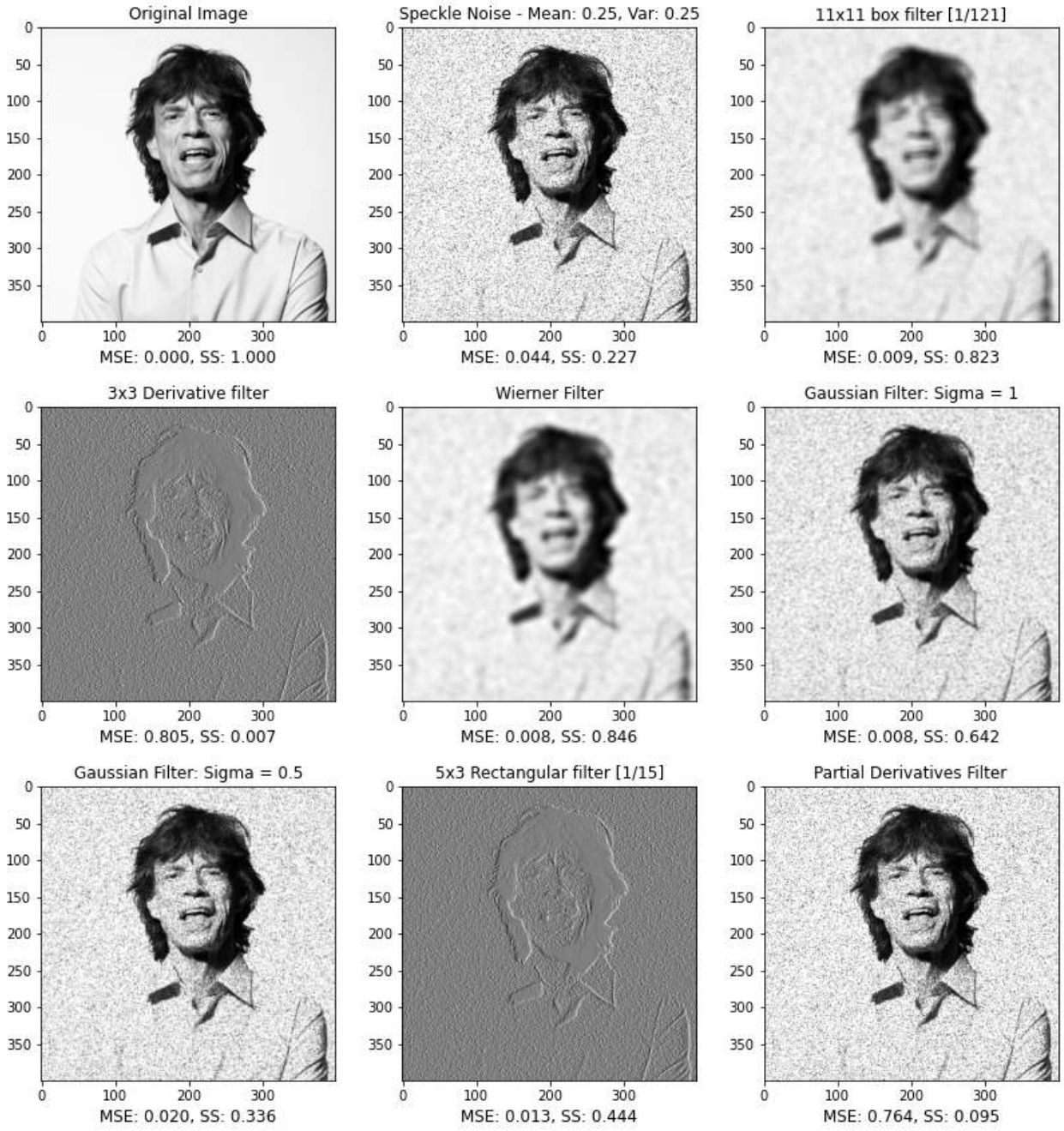

Figure 55: Linear filtering on speckled noise, mean of 0.25 and variance of 0.25

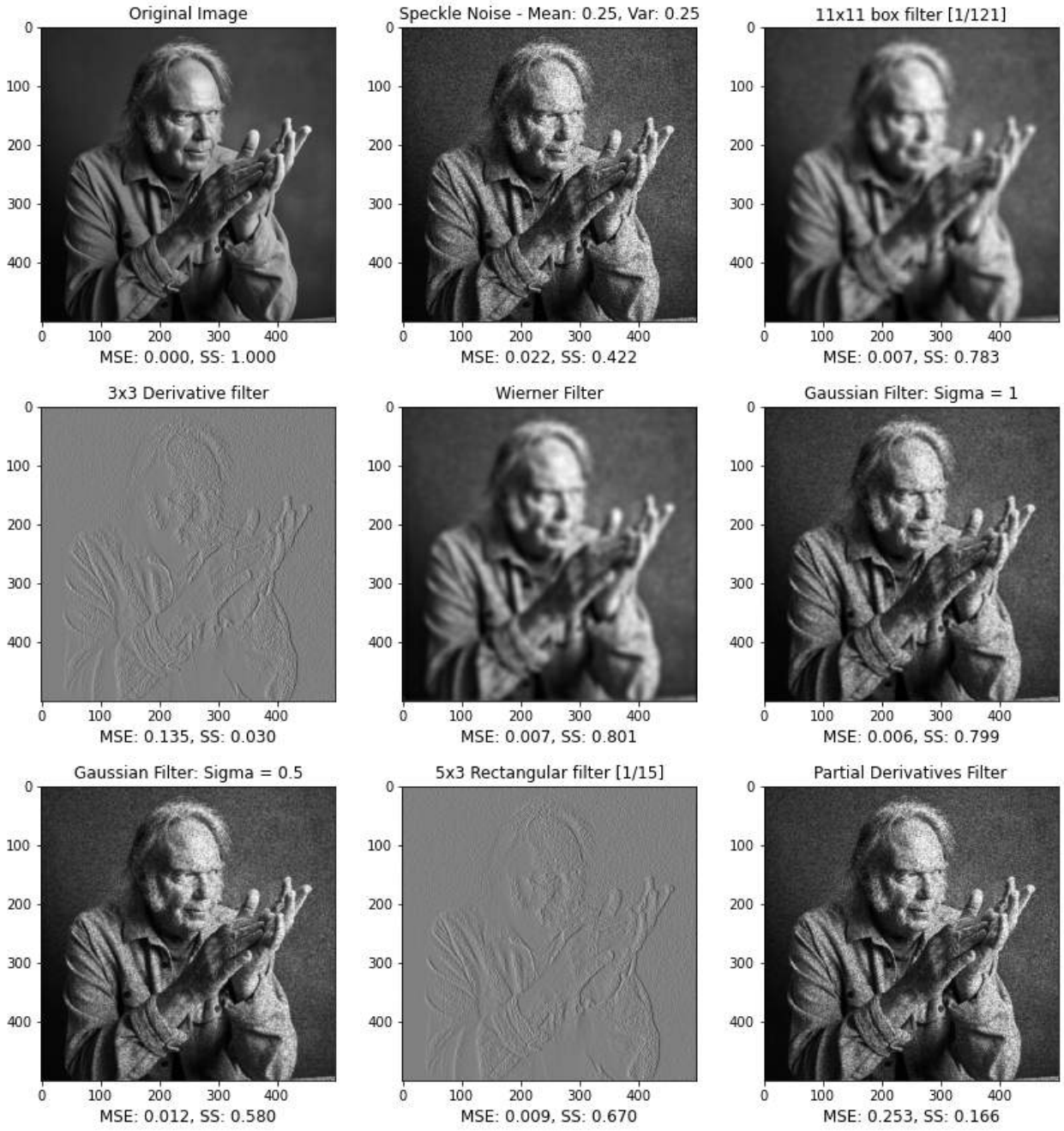

Figure 56: Linear filtering on speckled noise, mean of 0.25 and variance of 0.25

The results of trial 2 are almost identical to that of trial 1. Again the box filter, wiener, and gaussian filter with sigma equal to one produce the best images. Although the wiener filter has a slightly higher structural similarity, it has added blurriness that is not found in the gaussian filter. We conclude our image analysis of linear filters with trial 3 for the speckled noise.

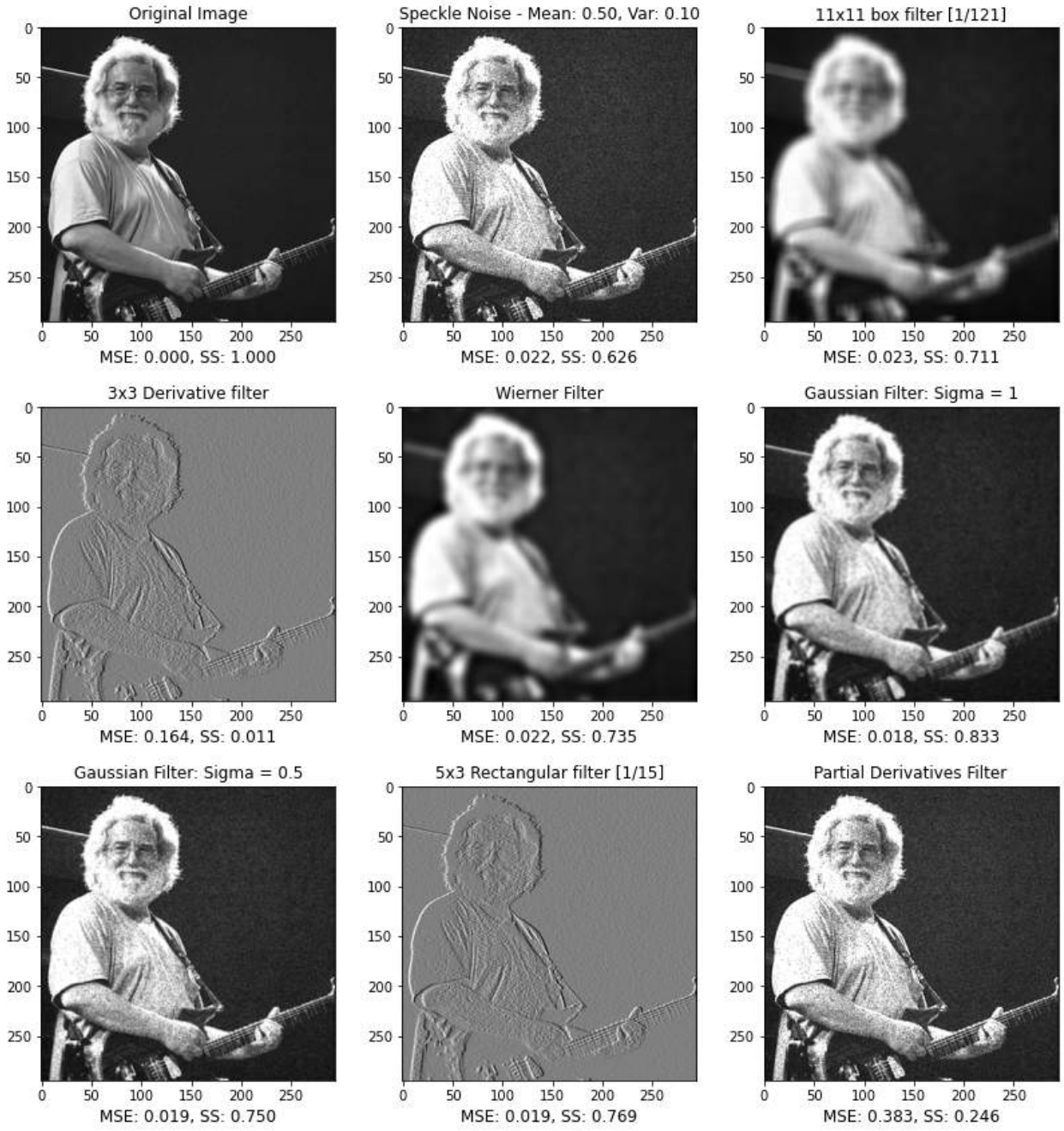

Figure 57: Linear filtering on speckled noise, mean of 0.5 and variance of 0.1

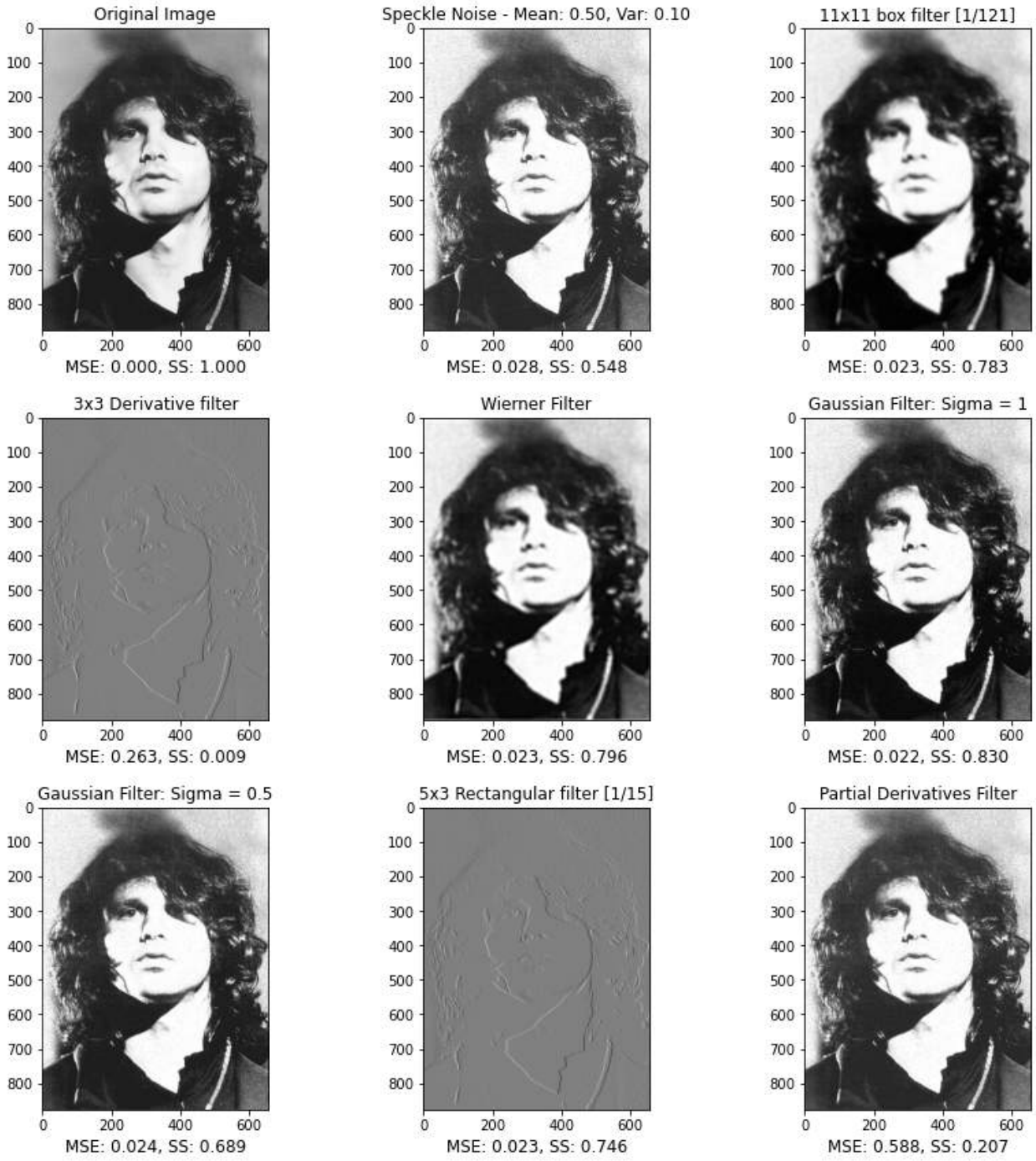

Figure 58: Linear filtering on speckled noise, mean of 0.5 and variance of 0.1

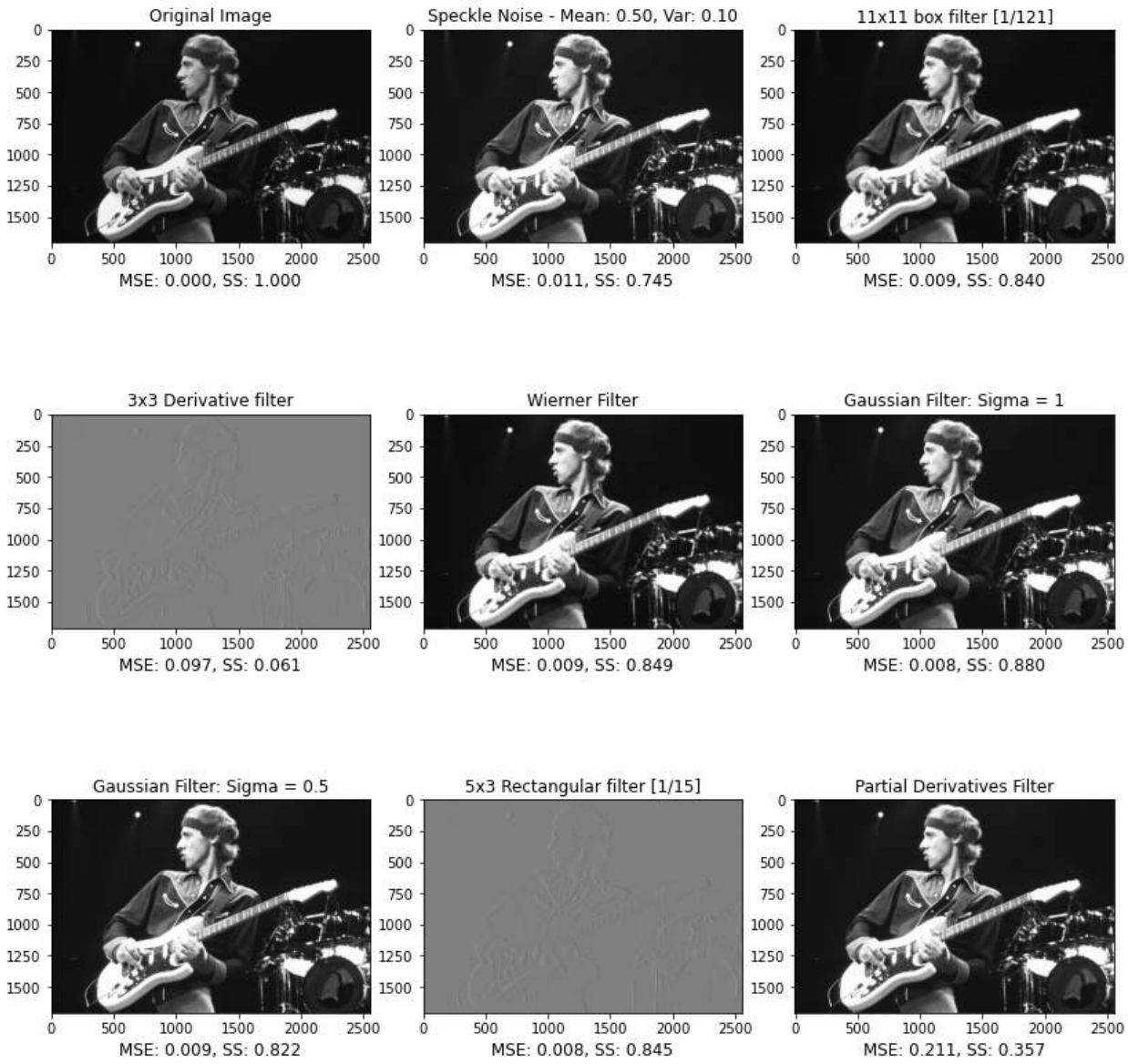

Figure 59: Linear filtering on speckled noise, mean of 0.5 and variance of 0.1

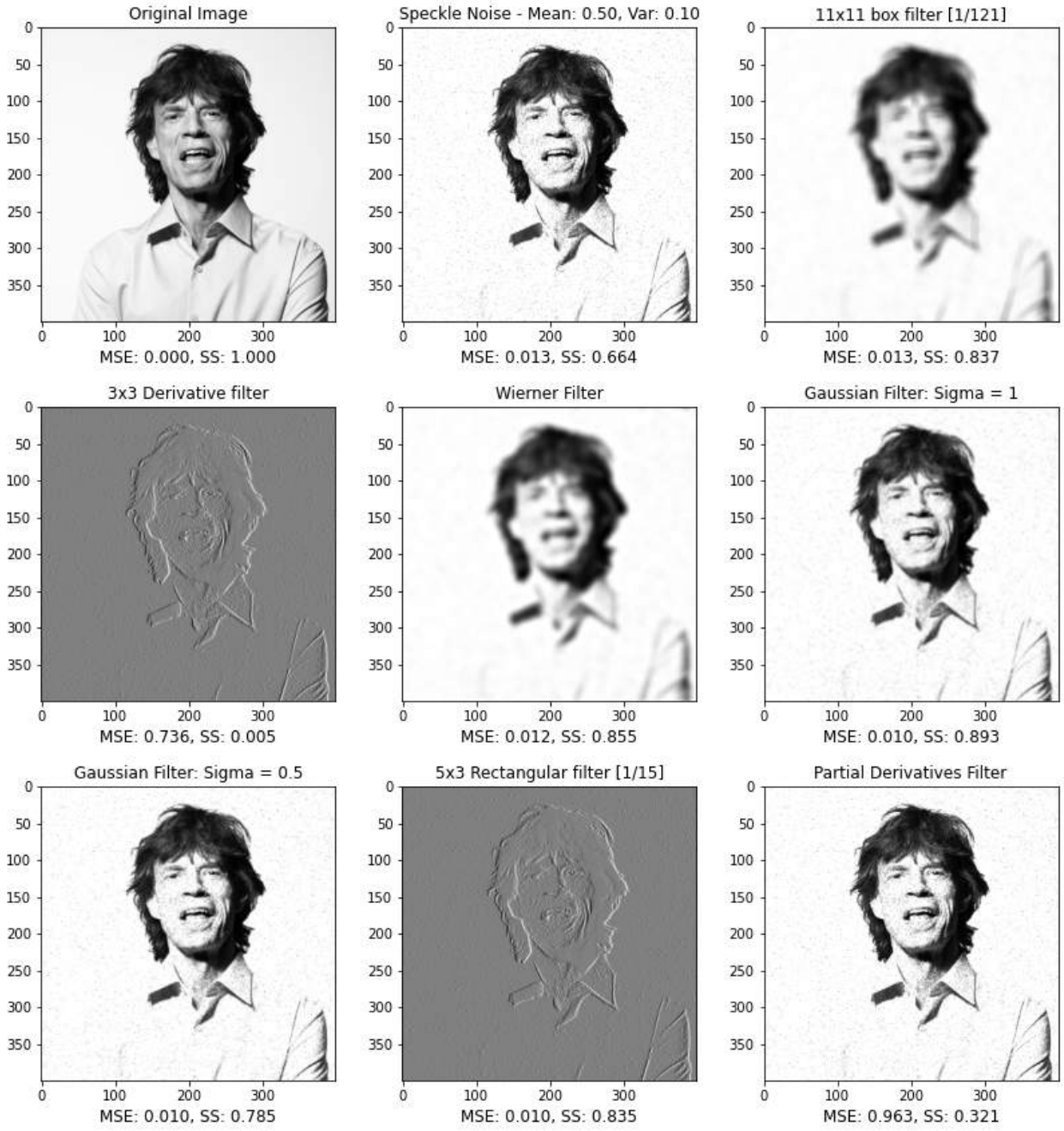

Figure 60: Linear filtering on speckled noise, mean of 0.5 and variance of 0.1

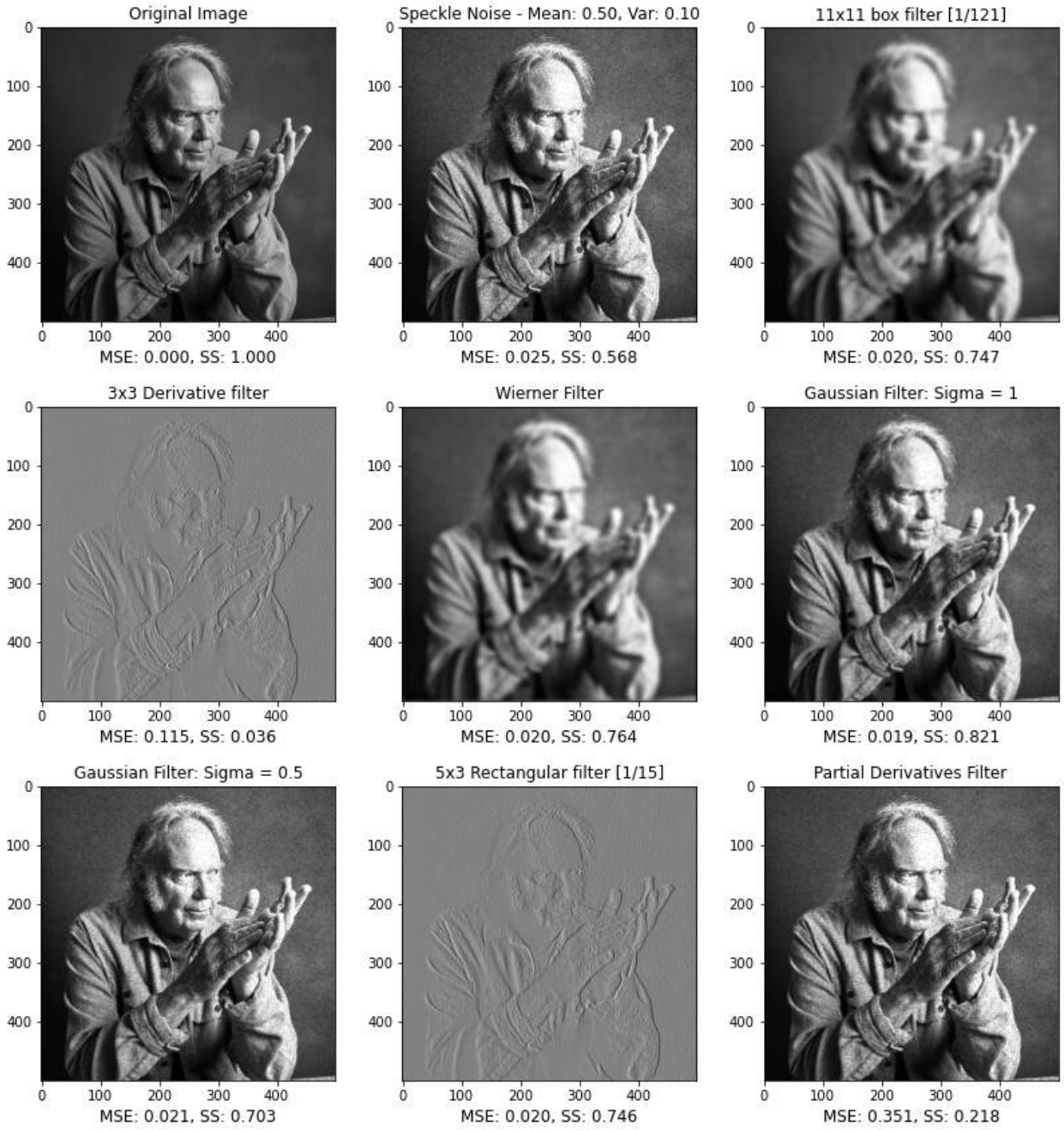

Figure 61: Linear filtering on speckled noise, mean of 0.5 and variance of 0.1

The results of trial 3 are consistent with trials 1 and 2, the box filter, wiener filter, and gaussian filter proving to be the best filters with the gaussian producing less blurriness.

In analyzing the images formed from various linear filters to various types of noise, we saw a similar overall trend. The 11 by 11 box filter and wiener filter did very well statistically, from a mean squared loss and structural similarity standpoint, but added blurriness to the noisy image that decreased its visual appeal. In contrast, the gaussian filter with a higher sigma also did very well statically, sometimes better than that of the box and wiener filter, but did not blur the images as much as the box and wiener filters did. The 3 by 3 derivative filter and rectangular filter did very poorly throughout the whole experiment, as the filters primarily acted in pulling out the edges of the figures in the images analyzed. To this extent, it comes as no surprise that they often did well from a structural similarity standpoint, pulling out the textures of the image. The partial derivative filter, formed from two 3 by 3 derivative filters, did next to nothing for the entirety of the experiment. For some images, it seemed to barely denoise the images from a statistical standpoint, but in a way that was so minute it could not be seen visually.

Overall, the linear filters did a very poor job of denoising the images, regardless of the noise type. From a visual standpoint, none of the filters were able to get the noisy image back to resembling the original image and oftentimes simply averaged out the contrast to create pixels that were blurry but with less overall sharpness in their discrete contrast. This is widely seen in the box filter and wiener filter. Both filters did well statistically, but when analyzing the images that they produced it is clear that all the filter is doing is simply making the black pixelated noise more gray and the white pixelated noise slightly more gray as well. Thus, the noise actually has a denser distribution and the mean squared error of the resulting image decreases. It comes as no surprise that the result of these filters is simply a blurries picture of the noisy image. Next, we move on to nonlinear filters. In a similar manner as above, we will conduct 3 trials for each type of noise except for the poisson noise, keeping the trial parameters the same as the linear filters just analyzed.

Nonlinear Filters

Before diving into the analysis of how the nonlinear filters performed on the noisy images, we must first introduce the nonlinear filters used in the following analysis. In total, seven nonlinear filters were analyzed: median filter, bilateral filter, non local (slow algorithm), non local (fast algorithm), denoise tv chambolle, denoise tv bregman, and denoise wavelet.

The median filter runs through each pixel and replaces the pixel with a median value taken from neighboring pixels. In my analysis, I analyze a median filter using a disk parameter with size 7, thus the algorithm takes the median of neighboring pixels in a circle of size 7. The second filter I analyze is the bilateral filter. The bilateral filter replaces the intensity of each pixel based into the algorithm with a weighted average of the intensity of neighboring pixels. I also analyze the use of non-local filters, using both the fast and slow algorithms. The non-local algorithm averages the value of a given pixel with values of other pixels in the neighborhood. The computing time of the algorithm depends on the patch size passed through the function. In the faster version of the algorithm, the integral of the patches distances for a shift which reduces the overall number of operations carried out in the algorithm. In the slow algorithm, all of the pixels of a patch contribute to the distance to another patch with the same weight, without regard for their distance to the patch, which theoretically will result in worse denoising efforts to a noisy image. In both of these algorithms, an estimated sigma value is passed through the algorithm, taken by the estimated sigma function built into skimage. This function averages the standard deviation of the pixels in the image in an attempt to denoise the image. A patch size of 5 and distance 6 was used in both algorithms and kept constant to analyze the denoising efforts on various types of noise.

The final three denoising nonlinear algorithms analyzed were the denoise tv Chambolle, denoise tv Bregman, and denoise wavelet. The denoising tv chambolle performs a total variation denoising to minimize the overall variation in the noisy image, whereas the denoise tv bregman algorithm uses a split-bregman optimization which tries to find an image with less total variation under the constraint of being similar to the input image, controlled by the regularization parameter. Finally, the denoise wavelet algorithm performs wavelet denoising, which attempts to denoise a signal/image by shrinking the amplitude of the transform of the signals that make up the image. Let's start by analyzing these nonlinear filters on the gaussian noise trials.

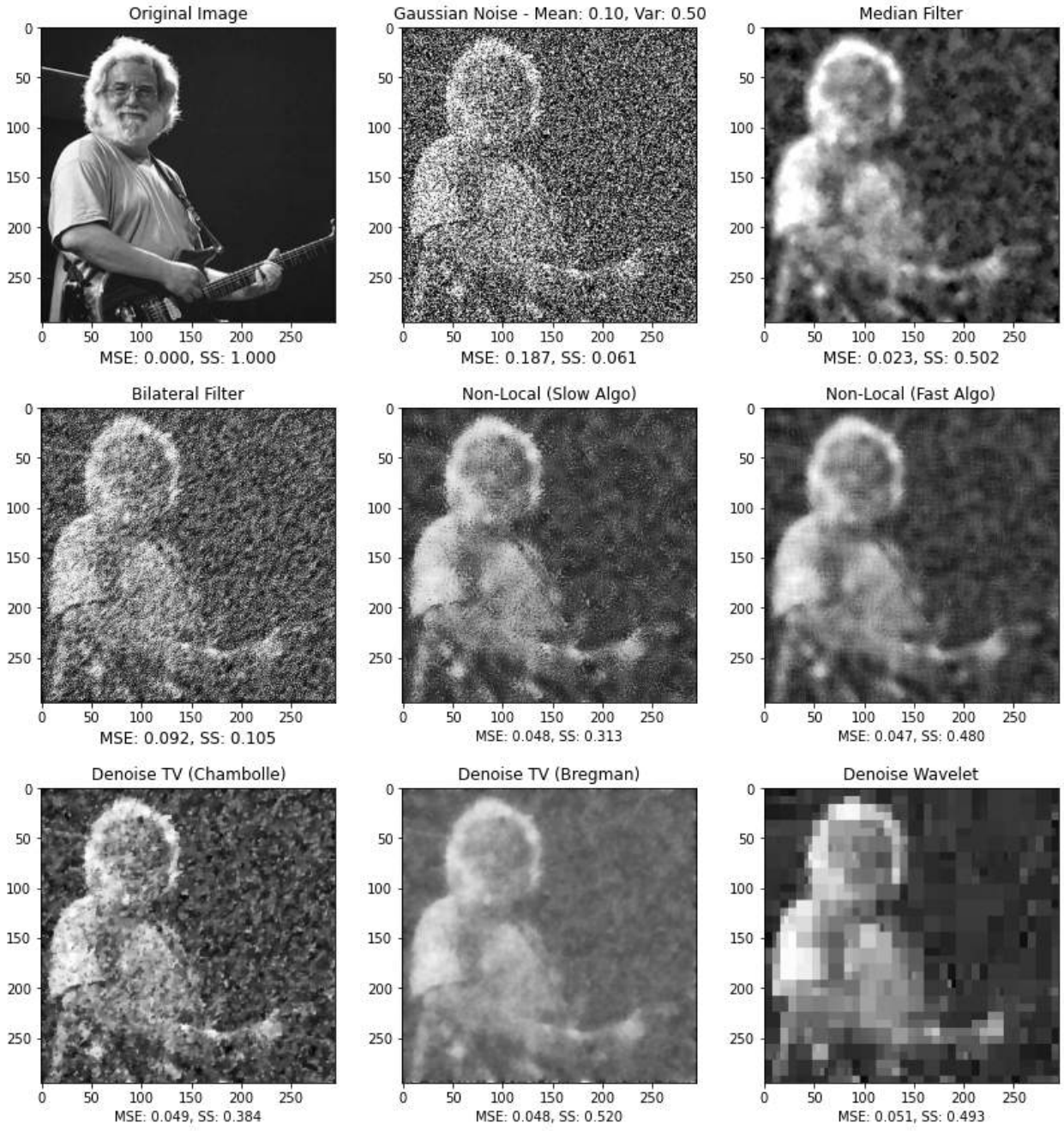

Figure 62: Nonlinear filtering on gaussian noise with mean 0.1 and variance 0.5

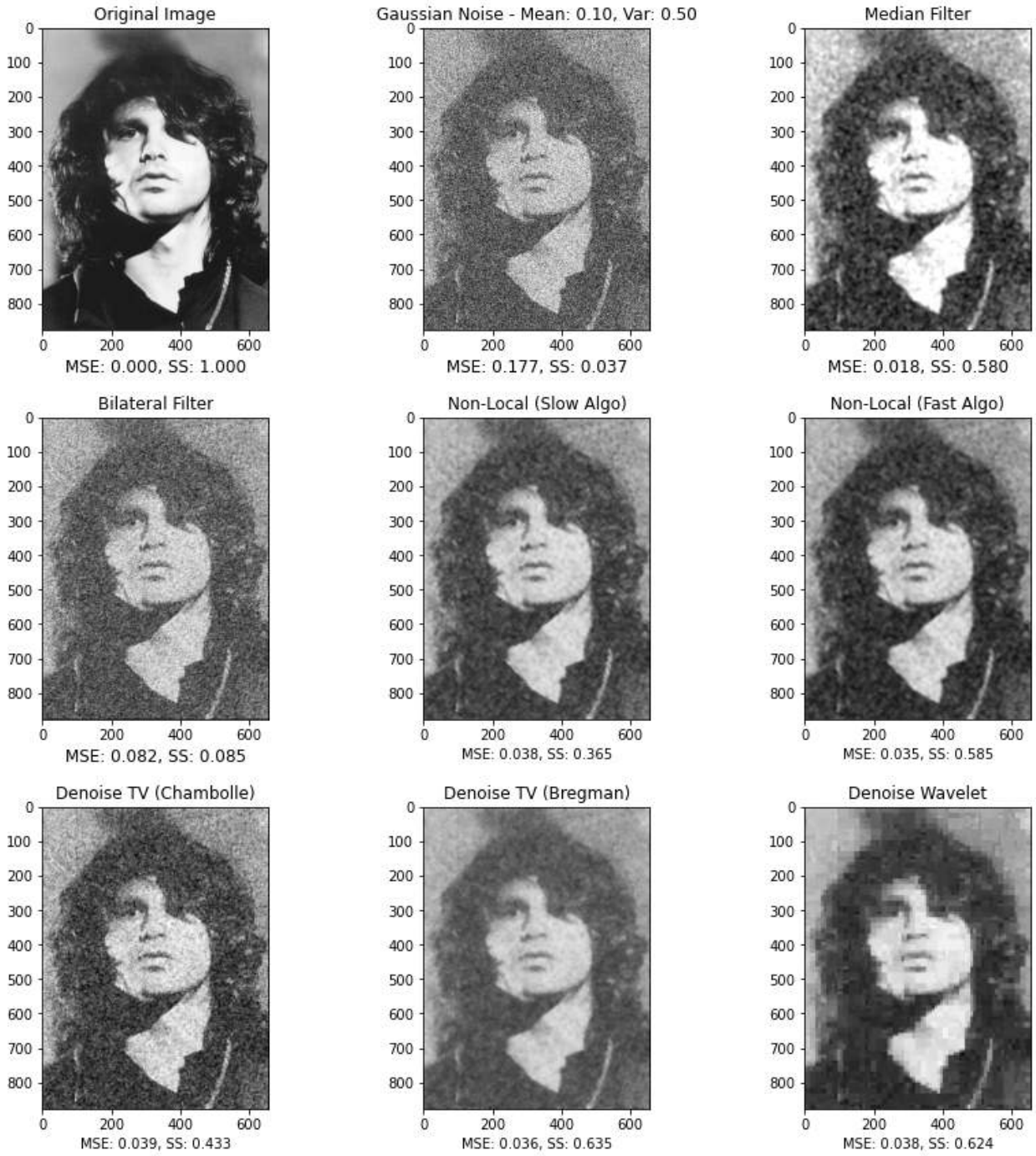

Figure 63: Nonlinear filtering on gaussian noise with mean 0.1 and variance 0.5

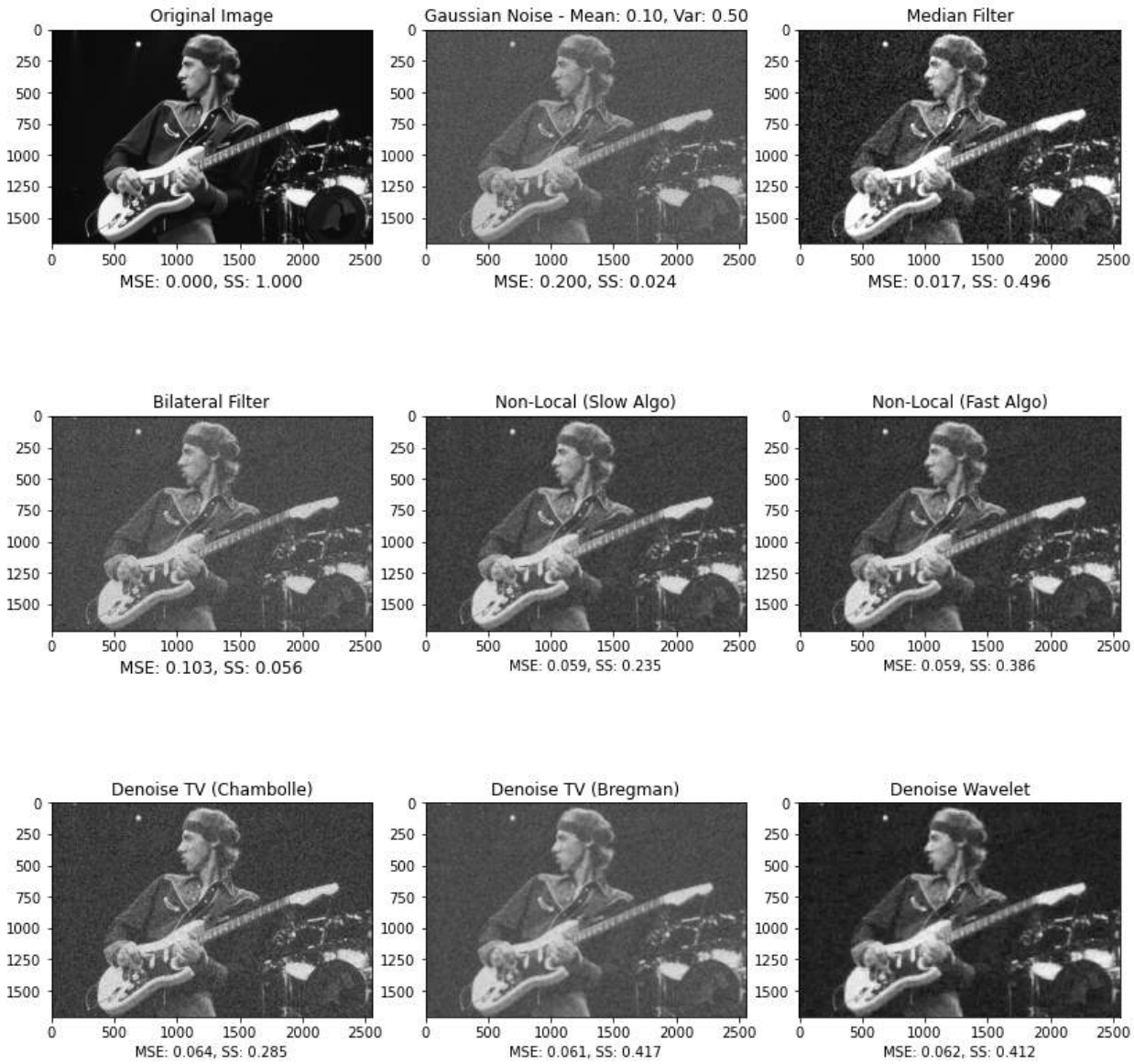

Figure 64: Nonlinear filtering on gaussian noise with mean 0.1 and variance 0.5

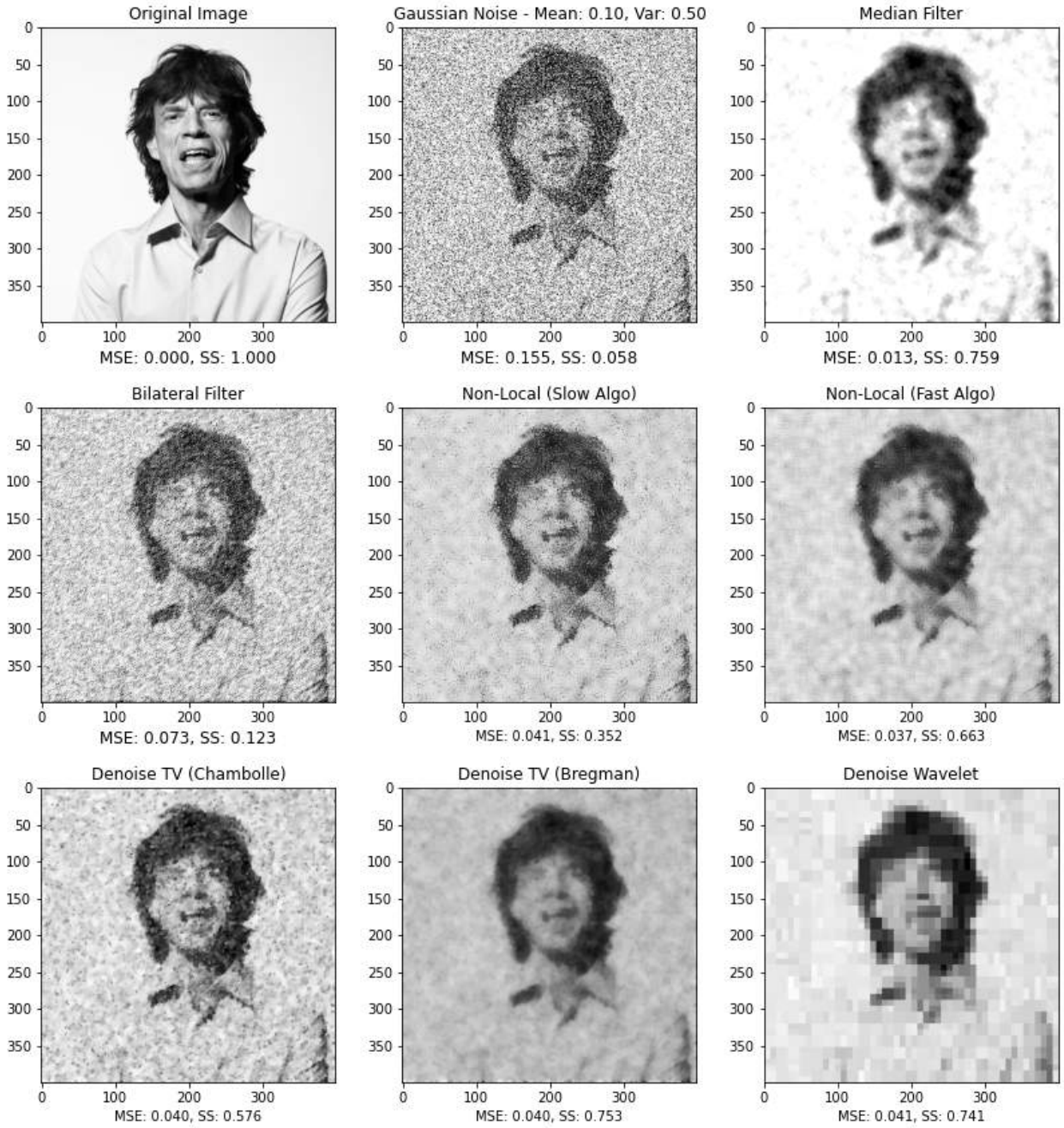

Figure 65: Nonlinear filtering on gaussian noise with mean 0.1 and variance 0.5

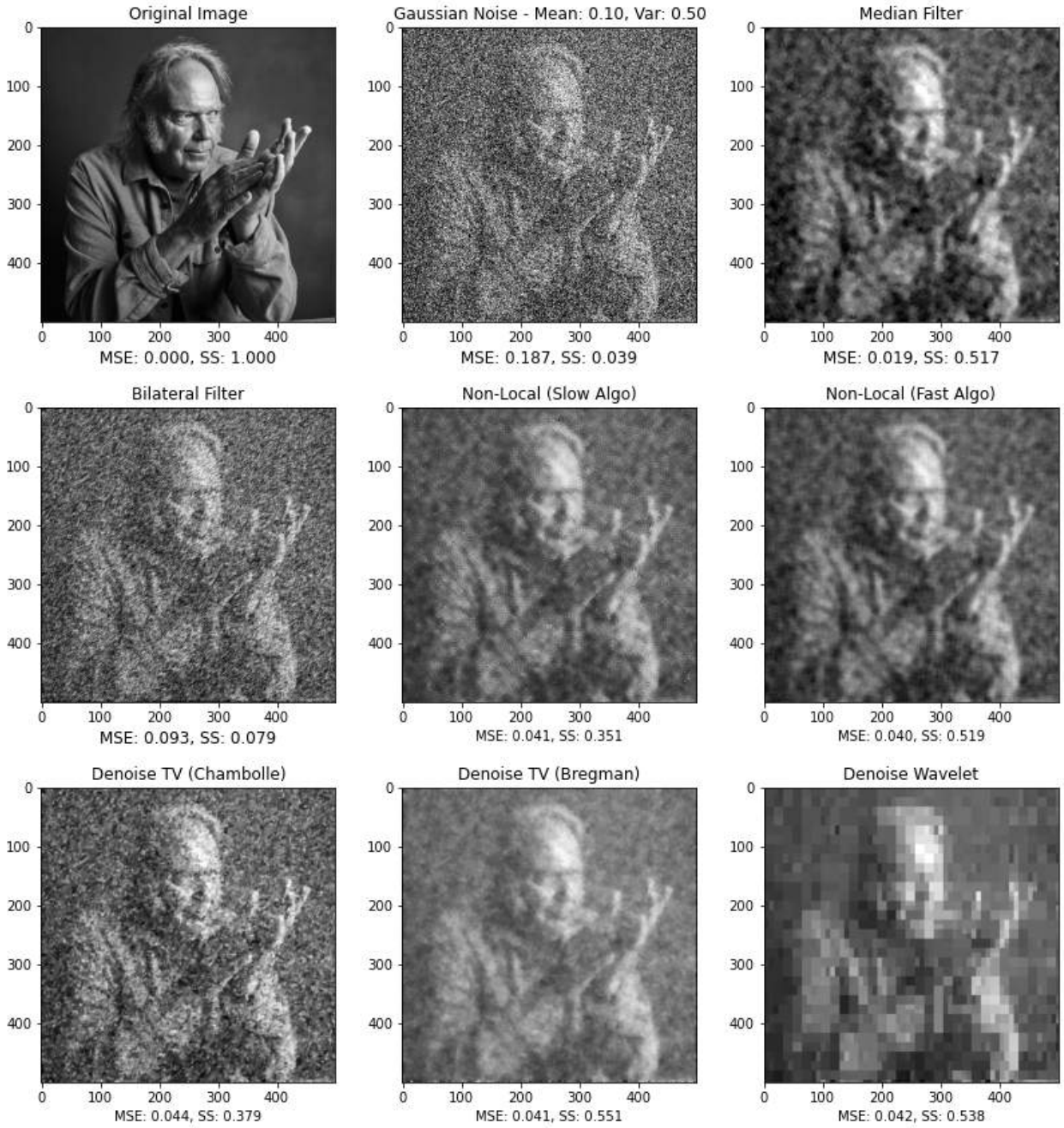

Figure 66: Nonlinear filtering on gaussian noise with mean 0.1 and variance 0.5

From our first look at the non-linear filtering, it is apparent that the nonlinear filtering does a better overall job of denoising in comparison to the linear filtering. The amount of noise introduced by the gaussian noise is quite substantial but all filters decrease the mean squared error and increase the structural similarity of the noisy image. For the median filter, it was found that using a disk of distance 7 produced the best results. Under 7 produced an image that had higher mean squared error but over 7 seemed to blur the image too much and actually resulted in adding more noise to the image. The weights of the denoise tv algorithms were also changed to

find the optimal value. It was found that the tv Chambolle, a weight of 0.25 produced the best results and for the tv Bregman, a weight value of 0.35 produced the best results in denoising the images. Likewise for the non local algorithms, a patch size of 5 with distance 6 produced an algorithm resulting in 5x5 patches with 13x13 search distance which produced the best resulting images for those particular algorithms. After analyzing the effects of all of these parameters, they were kept constant when changing noise types.

Of all nonlinear filters applied to the first trial of the gaussian noise, the median filter produces the best statistical result. All results do a good job of smoothing out the large amount of pixelation but it is evident that the wavelet algorithm produces an image that is substantially more blurry than the rest of the nonlinear algorithms. We move on to trial 2.

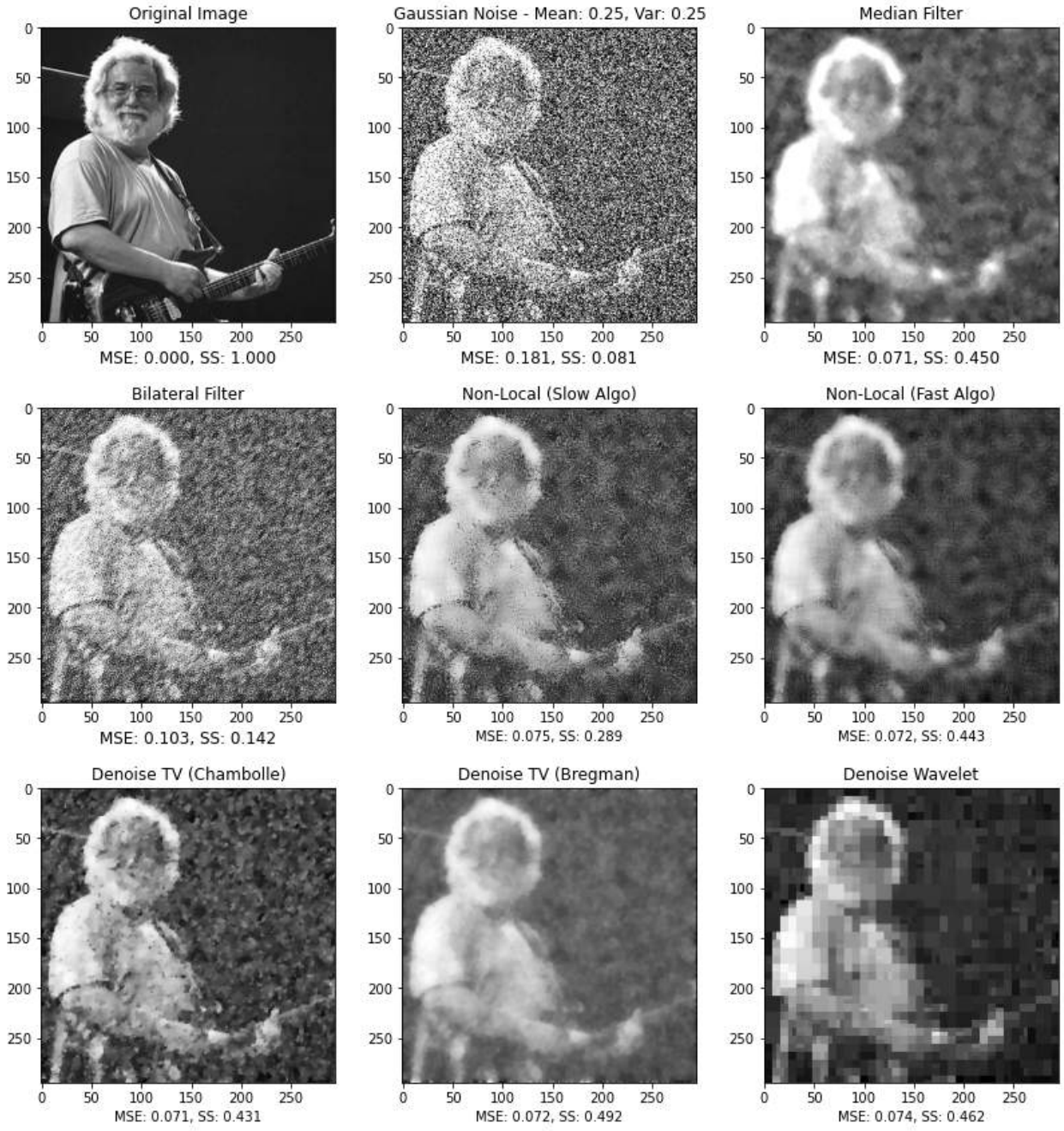

Figure 67: Nonlinear filtering on gaussian noise with mean 0.25 and variance 0.25

Figure 68: Nonlinear filtering on gaussian noise with mean 0.25 and variance 0.25

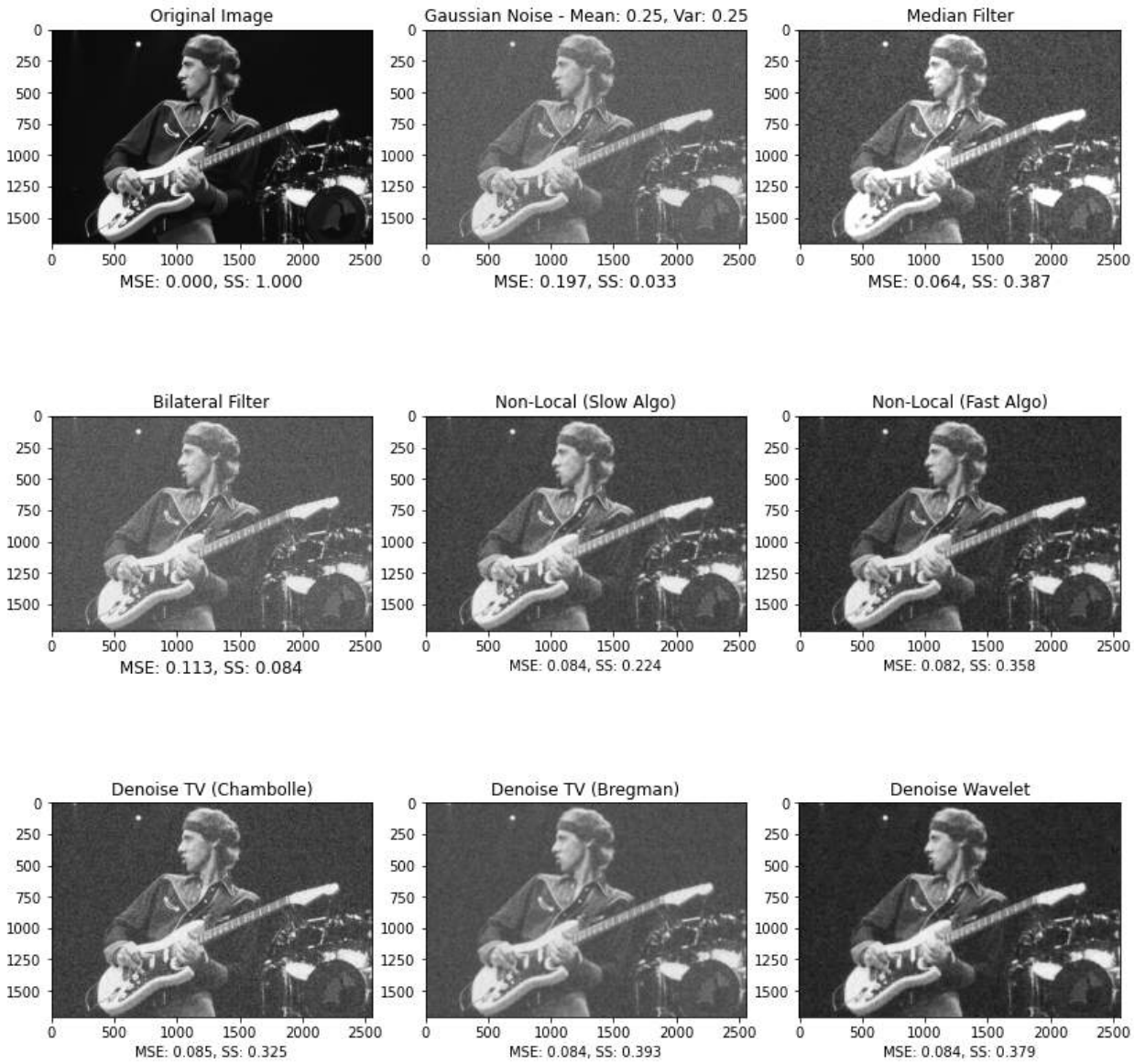

Figure 69: Nonlinear filtering on gaussian noise with mean 0.25 and variance 0.25

Figure 70: Nonlinear filtering on gaussian noise with mean 0.25 and variance 0.25

Figure 71: Nonlinear filtering on gaussian noise with mean 0.25 and variance 0.25

In trial 2 for the Gaussian noise, we see that the bilateral filter does substantially worse in comparison to the other types of filters. The fast algorithm of the non local filter performs better than the slow algorithm, but all of the filters are very comparable. Trial 3:

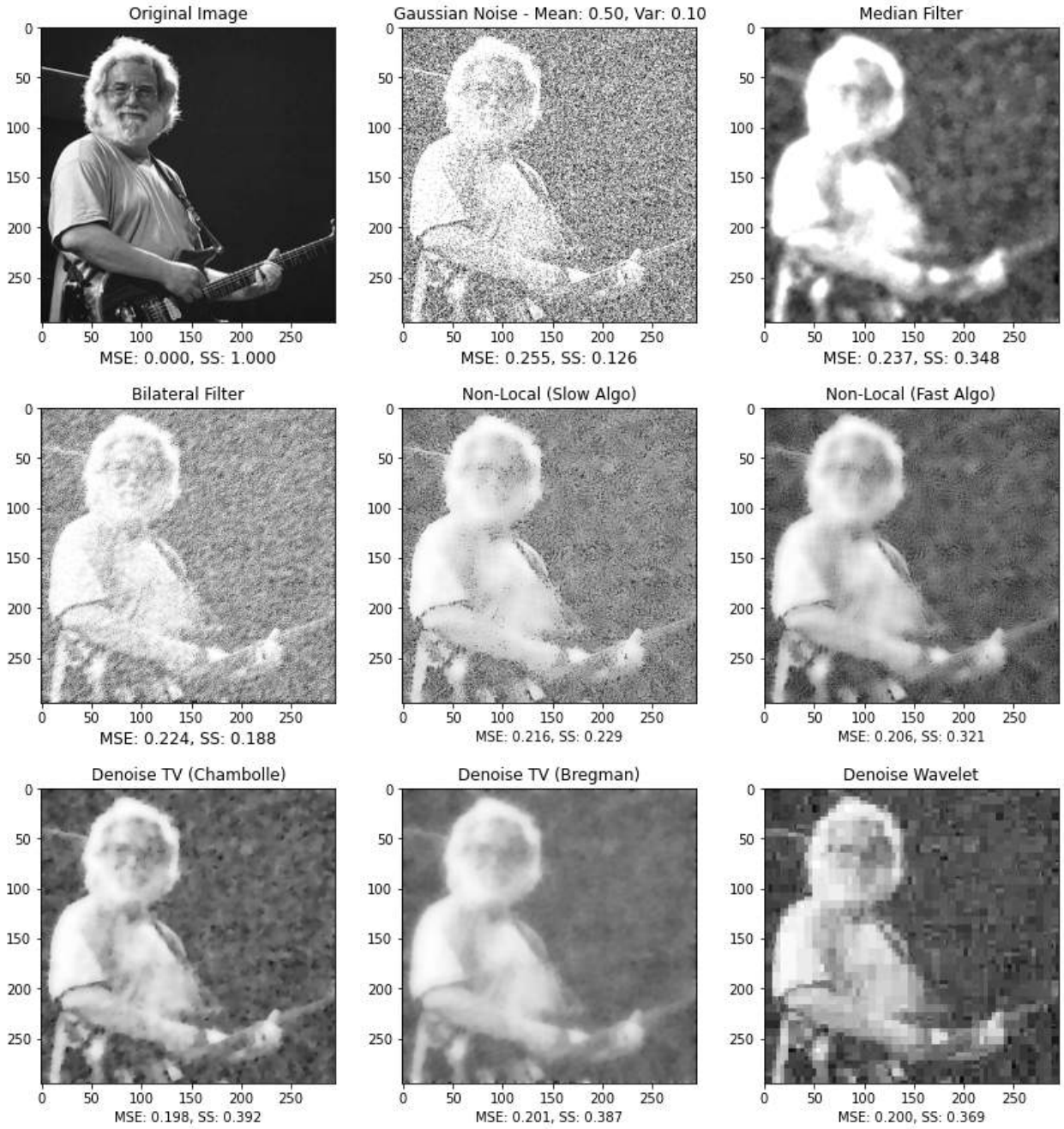

Figure 72: Nonlinear filtering on gaussian noise with mean 0.5 and variance 0.1

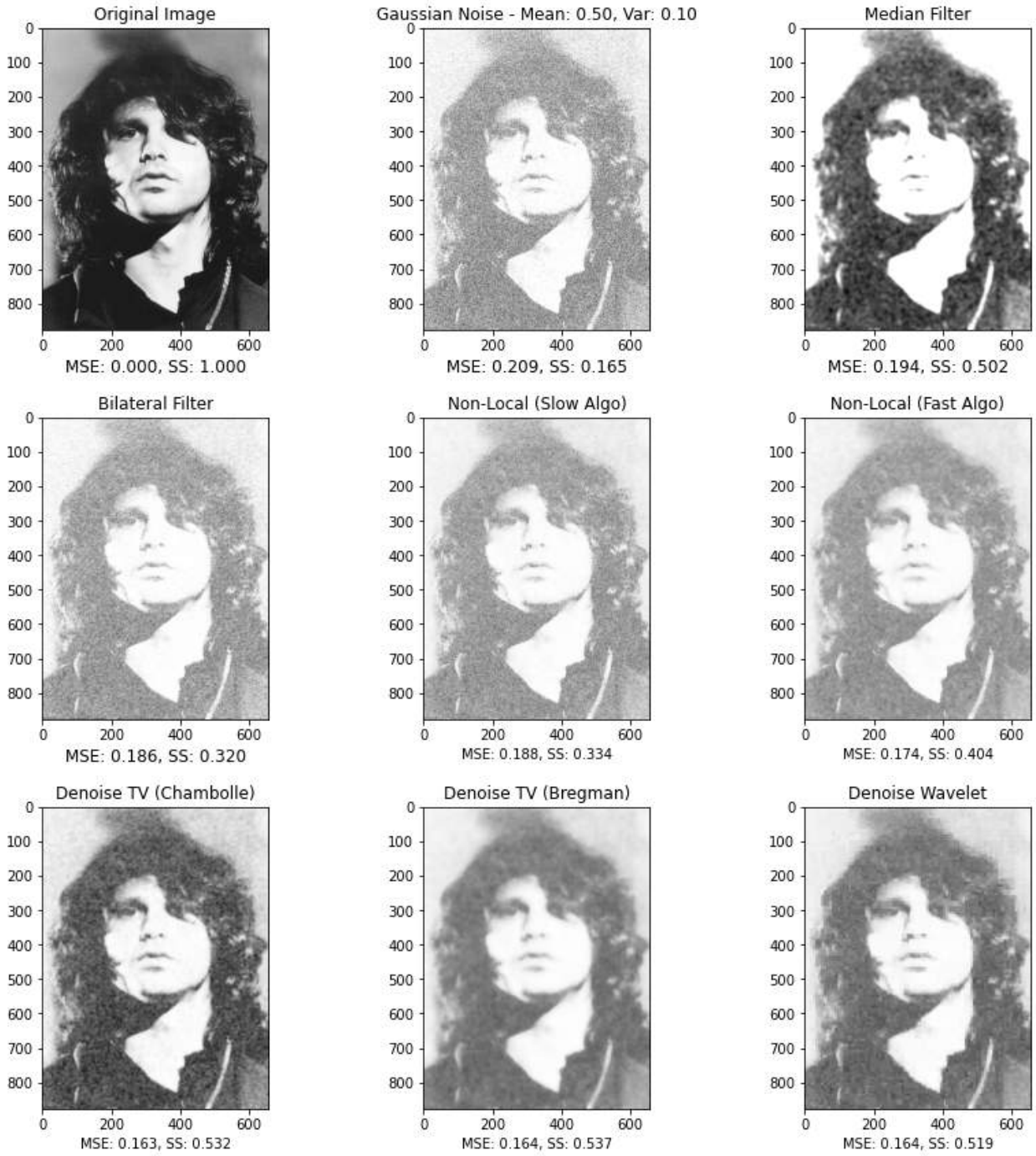

Figure 73: Nonlinear filtering on gaussian noise with mean 0.5 and variance 0.1

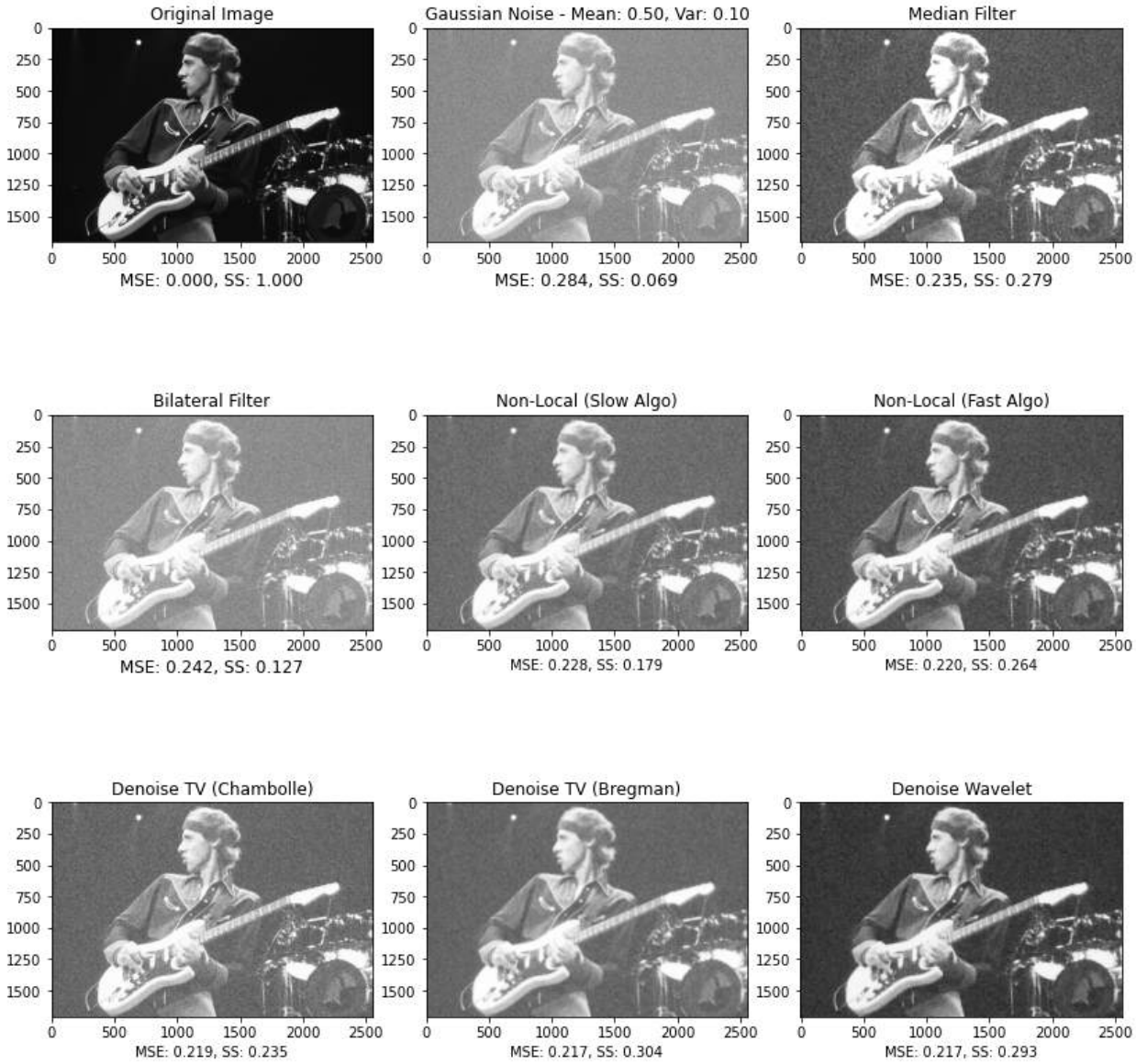

Figure 74: Nonlinear filtering on gaussian noise with mean 0.5 and variance 0.1

Figure 75: Nonlinear filtering on gaussian noise with mean 0.5 and variance 0.1

Figure 76: Nonlinear filtering on gaussian noise with mean 0.5 and variance 0.1

The median filter does a good job of pulling out the consistent areas. As seen in Figure 75, it does a great job of making the background behind Mick constant white, whereas all the other algorithms still include some small bits of pixelated noise, but it goes too far and also applies similar techniques to the facial region. All of the non local algorithms still keep in some pixelation noise in the background and seem to not perform any real denoising at such a high amount of gaussian noise. Now, we move on to salt and pepper noise.

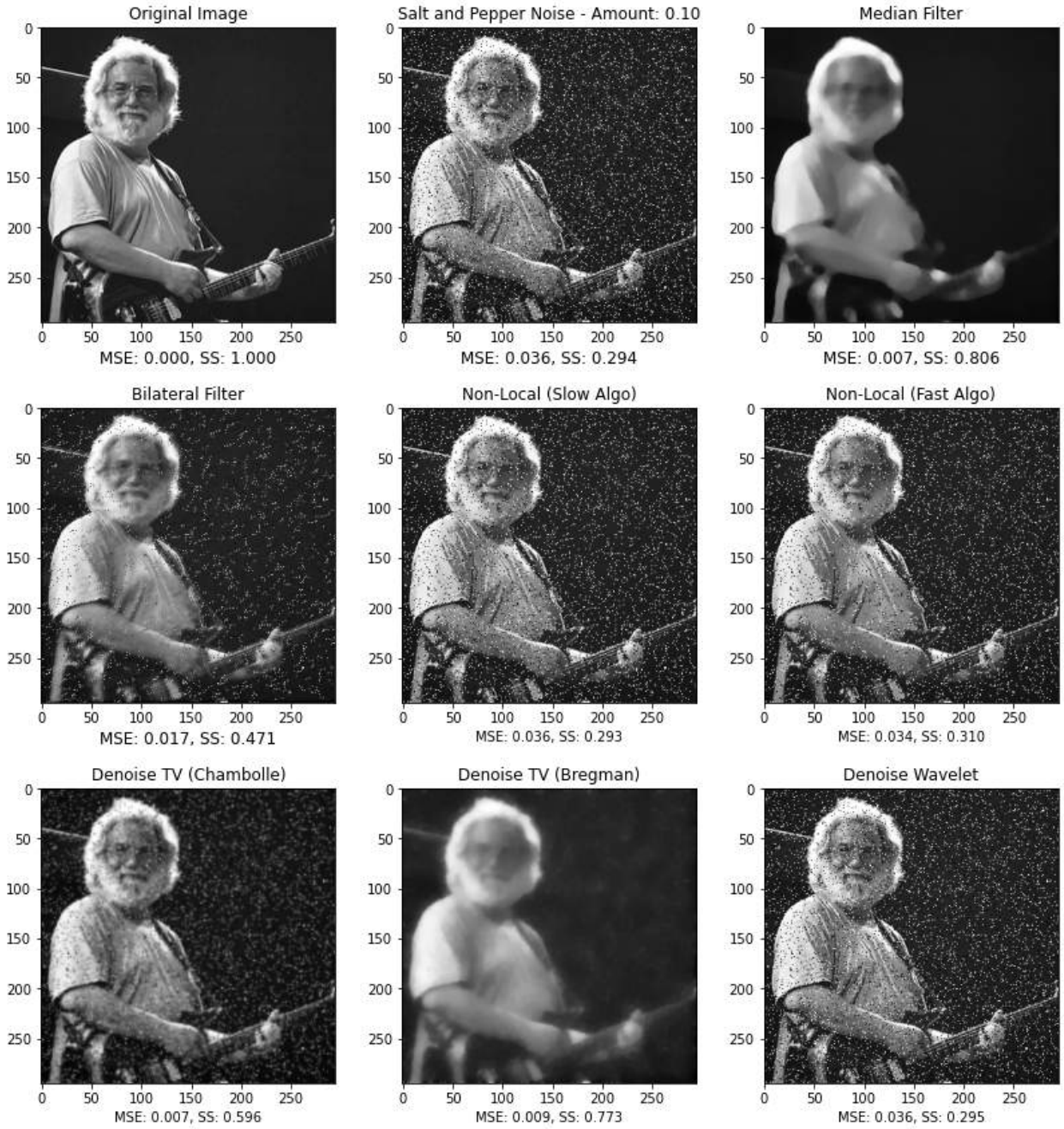

Figure 77: Nonlinear filtering on salt and pepper noise, amount of 0.1

Figure 78: Nonlinear filtering on salt and pepper noise, amount of 0.1

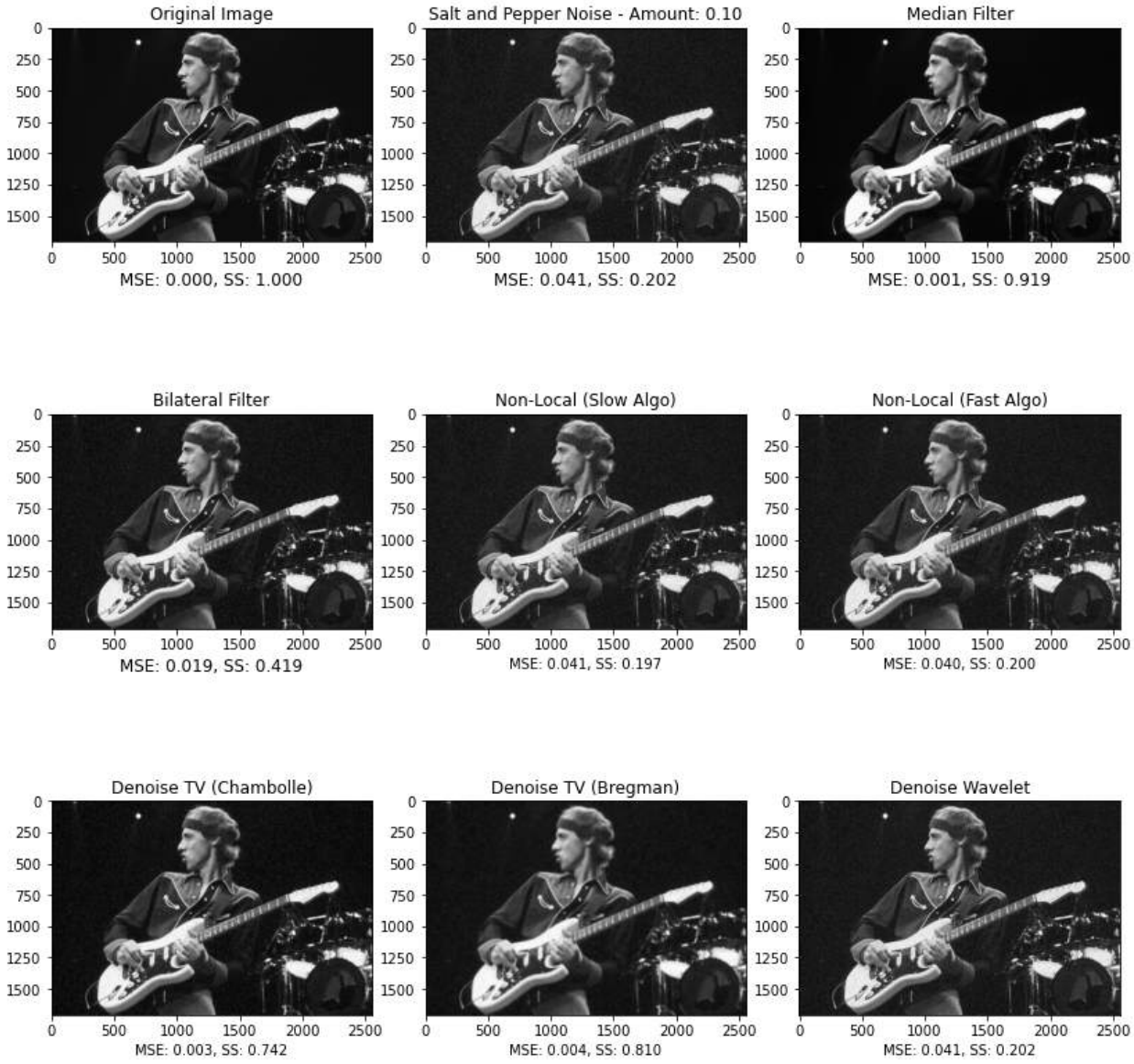

Figure 79: Nonlinear filtering on salt and pepper noise, amount of 0.1

Figure 80: Nonlinear filtering on salt and pepper noise, amount of 0.1

Figure 81: Nonlinear filtering on salt and pepper noise, amount of 0.1

For the salt and pepper noise, we again see very similar filtering characteristics as the gaussian noise. The median filter does a good job of filtering out pixelations in the background but oftentimes over-corrects and the image loses some of its sharpness and contrast. The bilateral filter does a good job of decreasing the mean squared error and increasing the statistical similarity, but the resulting image is not very impressive. The denoise tv bregman continues to add blurring to the image.

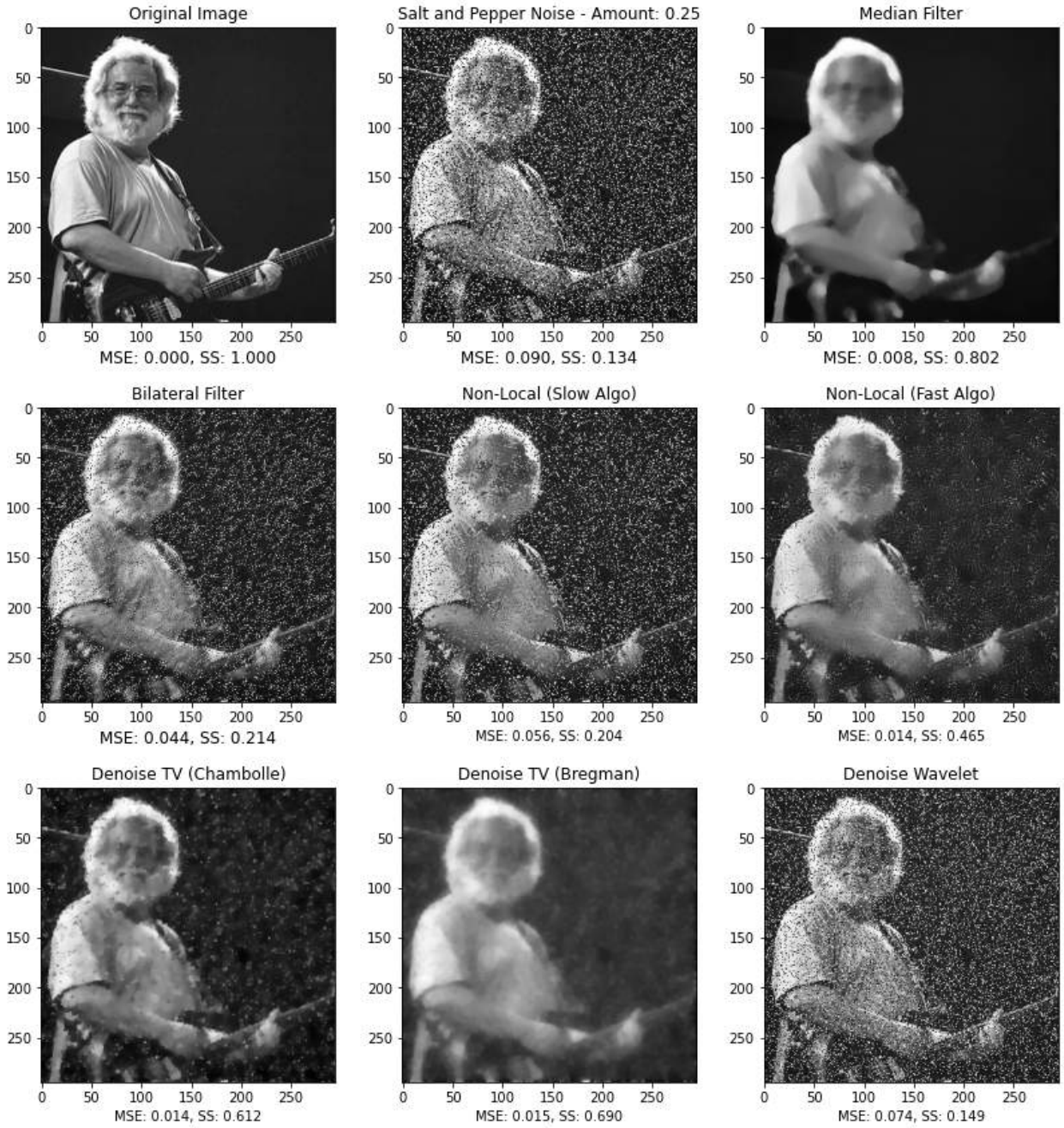

Figure 82: Nonlinear filtering on salt and pepper noise, amount of 0.25

Figure 83: Nonlinear filtering on salt and pepper noise, amount of 0.25

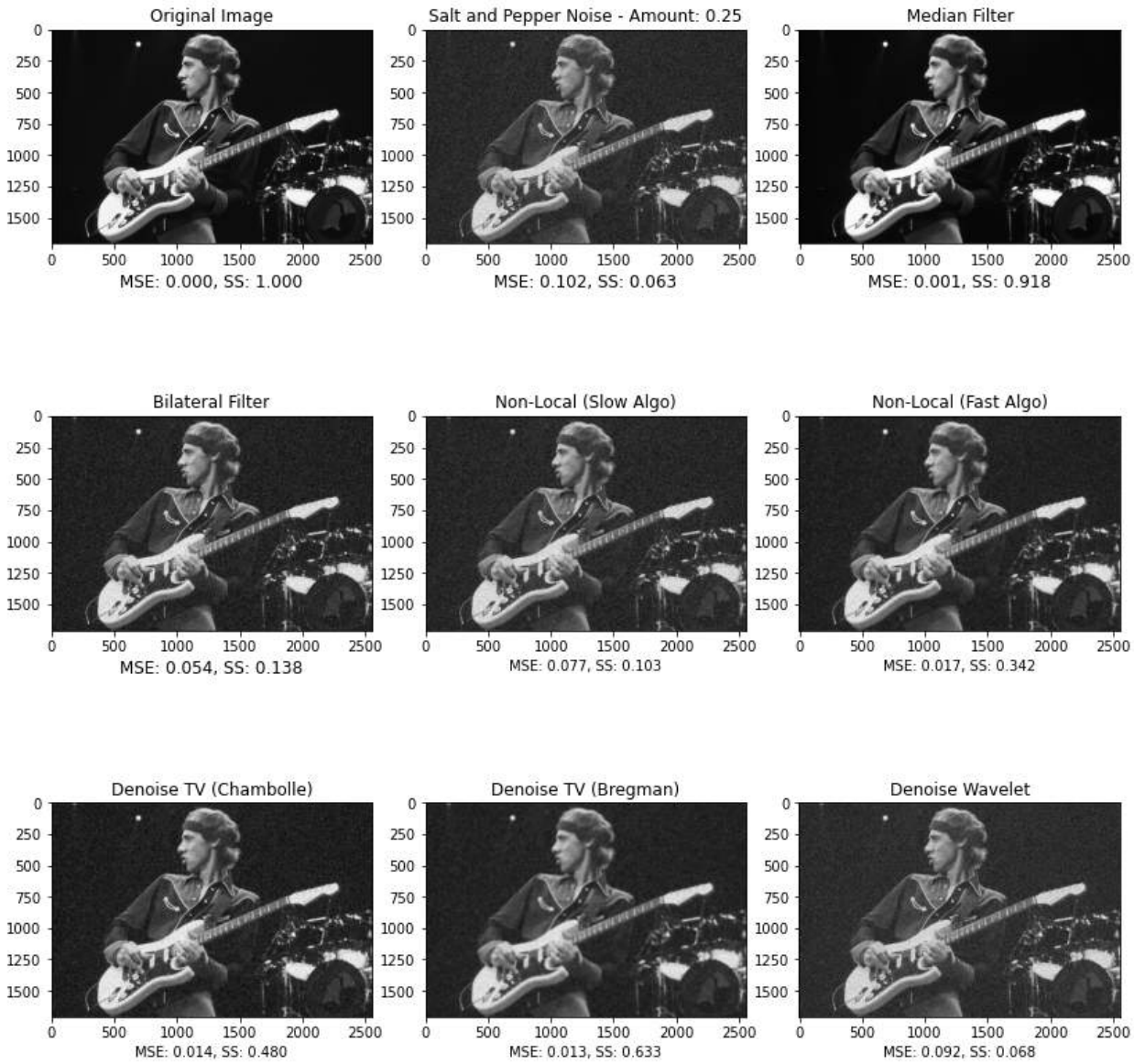

Figure 84: Nonlinear filtering on salt and pepper noise, amount of 0.25

Figure 85: Nonlinear filtering on salt and pepper noise, amount of 0.25

Figure 86: Nonlinear filtering on salt and pepper noise, amount of 0.25

For the salt and pepper noise with amount 0.25, the median filter does a really good job of denoising the image. It must be noted that all the images have constant backgrounds and the median filter does a really good job of pulling out pixels in the constant background area, averaging the neighbors and therefore pulling out the odd man pixels. We move on to trial 3 for salt and pepper.

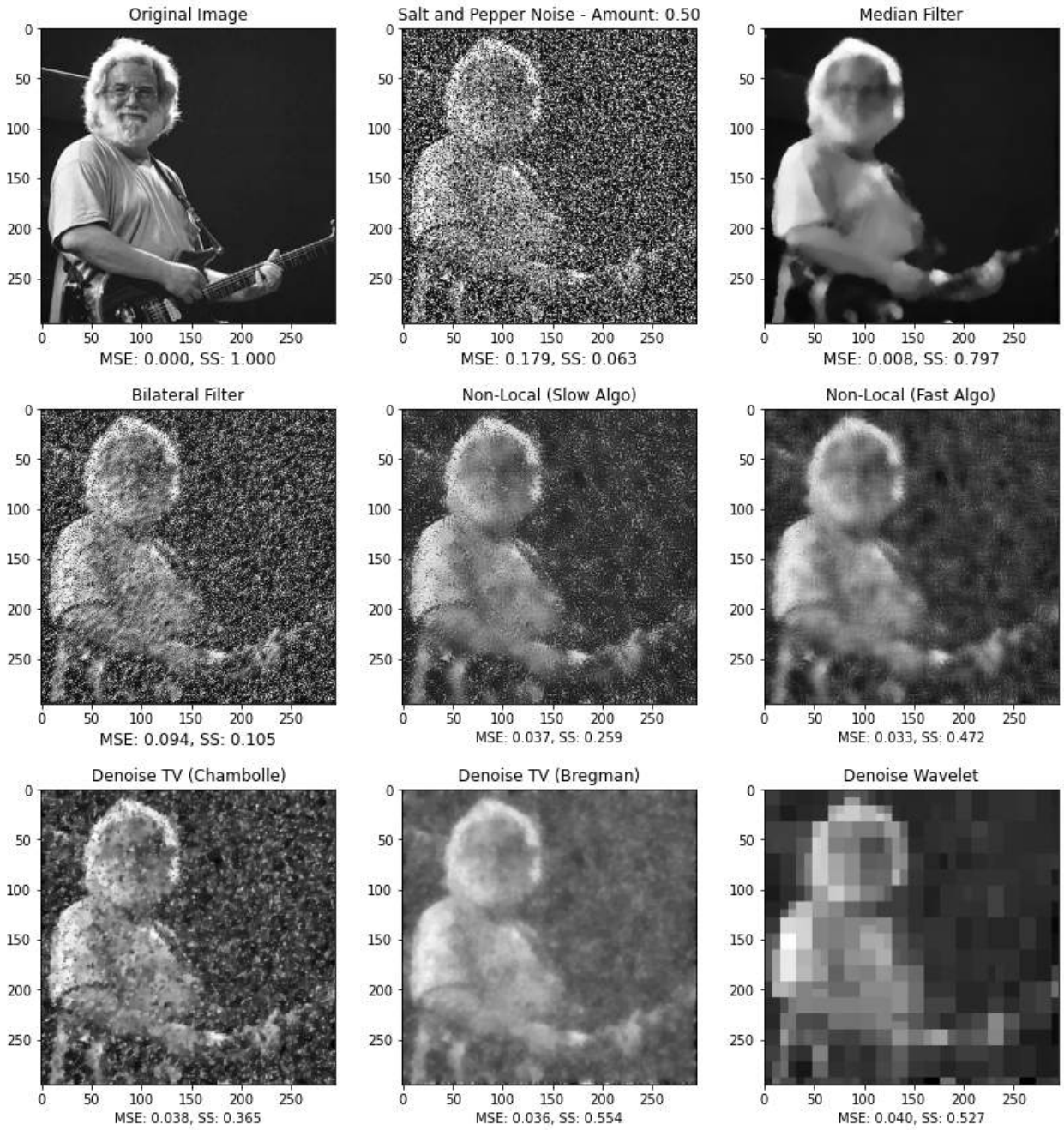

Figure 87: Nonlinear filtering on salt and pepper noise, amount of 0.5

Figure 88: Nonlinear filtering on salt and pepper noise, amount of 0.5

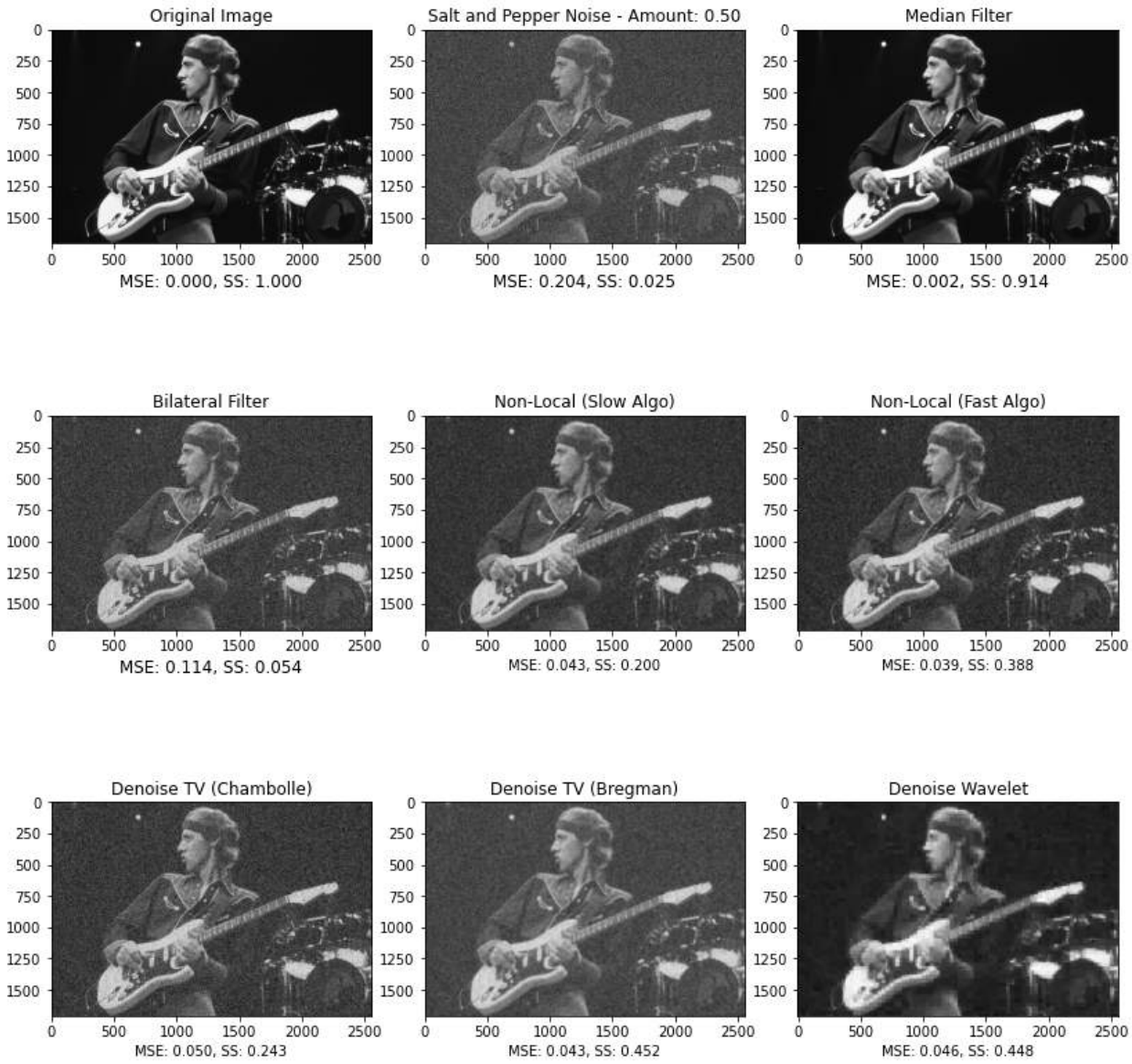

Figure 89: Nonlinear filtering on salt and pepper noise, amount of 0.5

Figure 90: Nonlinear filtering on salt and pepper noise, amount of 0.5

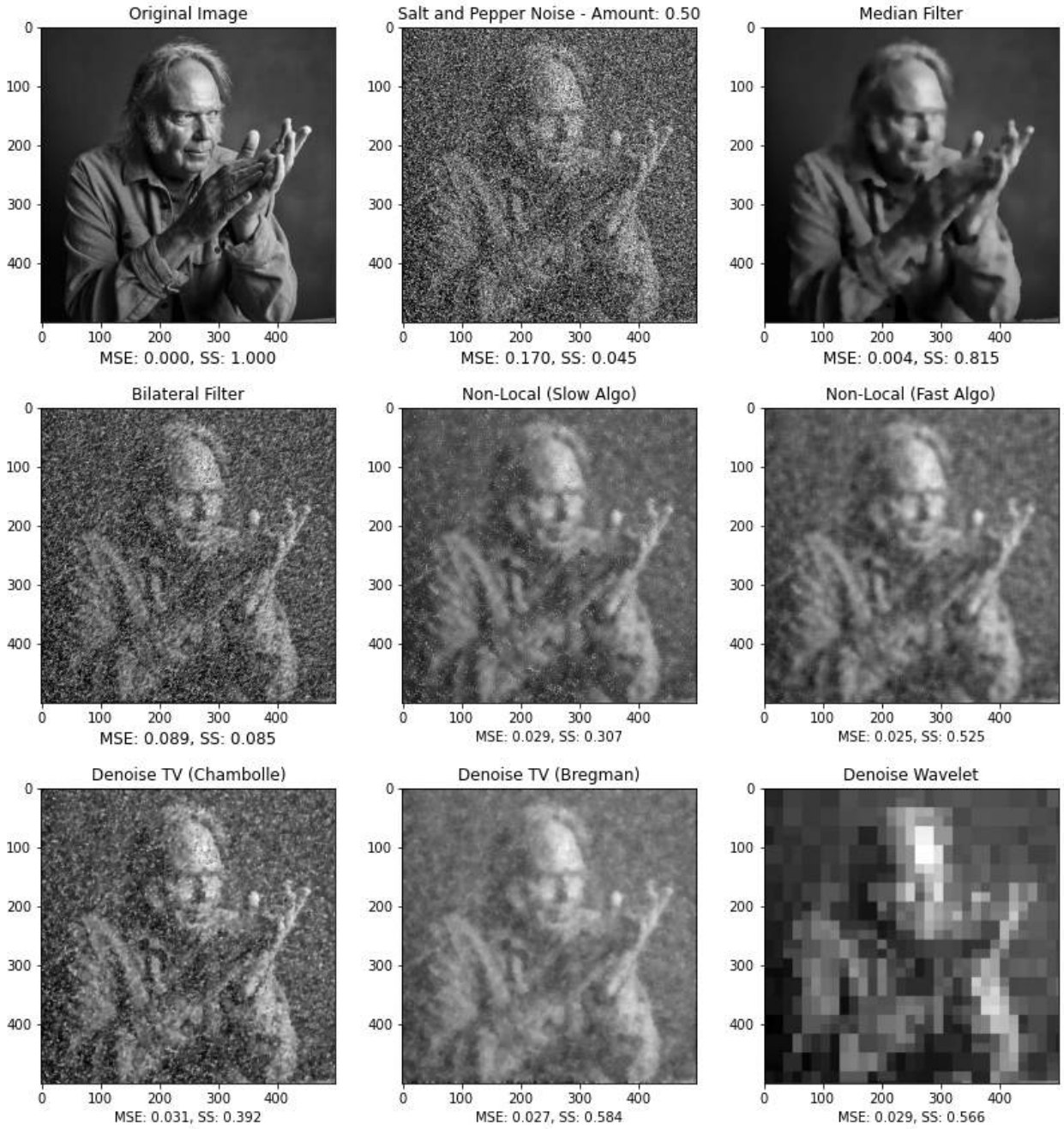

Figure 91: Nonlinear filtering on salt and pepper noise, amount of 0.5

For the large amount of salt and pepper noise, we see that the wavelet denoise does a very bad job of denoising. The median filter again outperforms the rest of the filters and the faster non local algorithm continues to outperform the slower algorithm. Moving on to the poison noise.

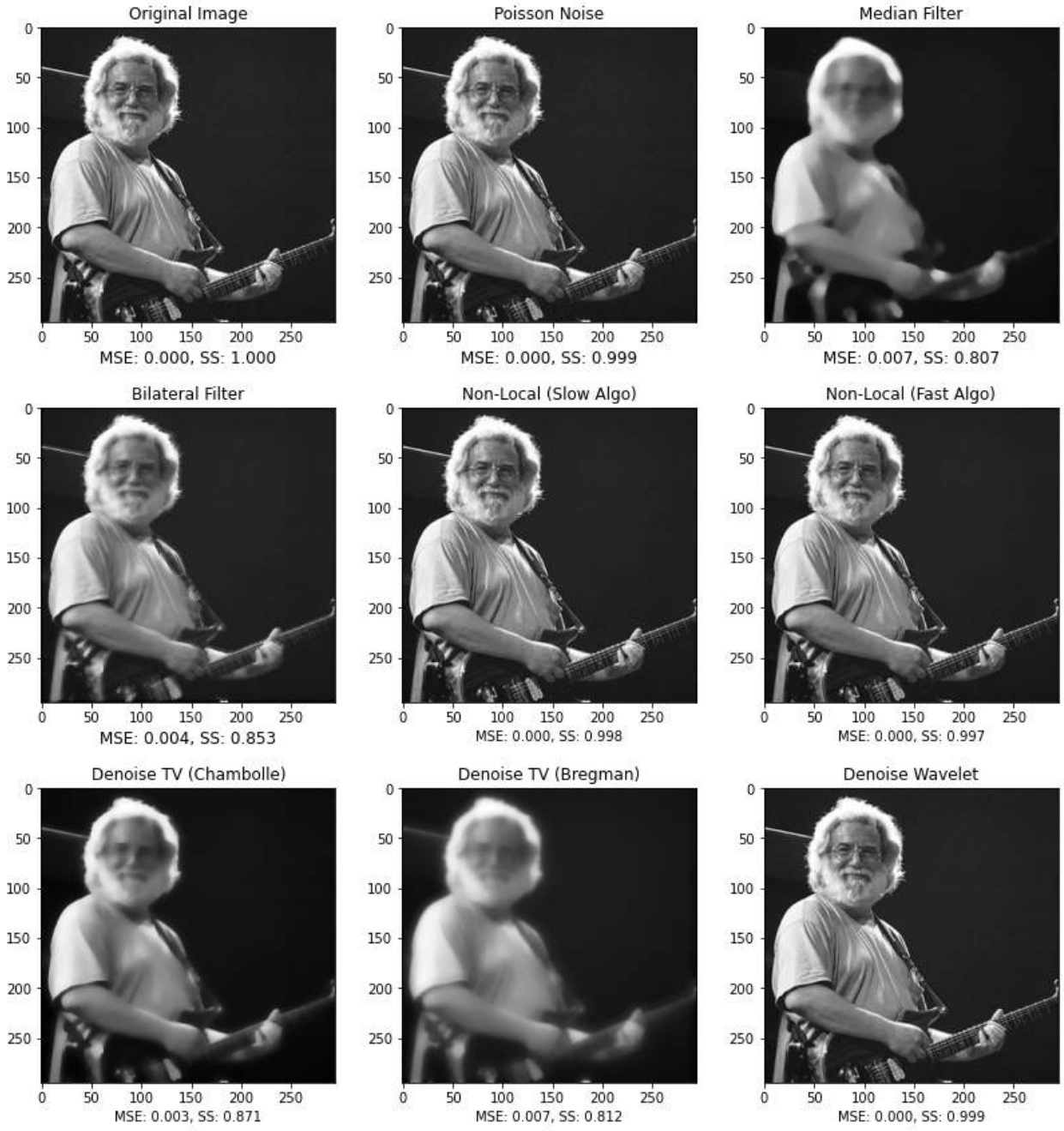

Figure 92: Nonlinear filtering on poisson noise

Figure 93: Nonlinear filtering on poisson noise

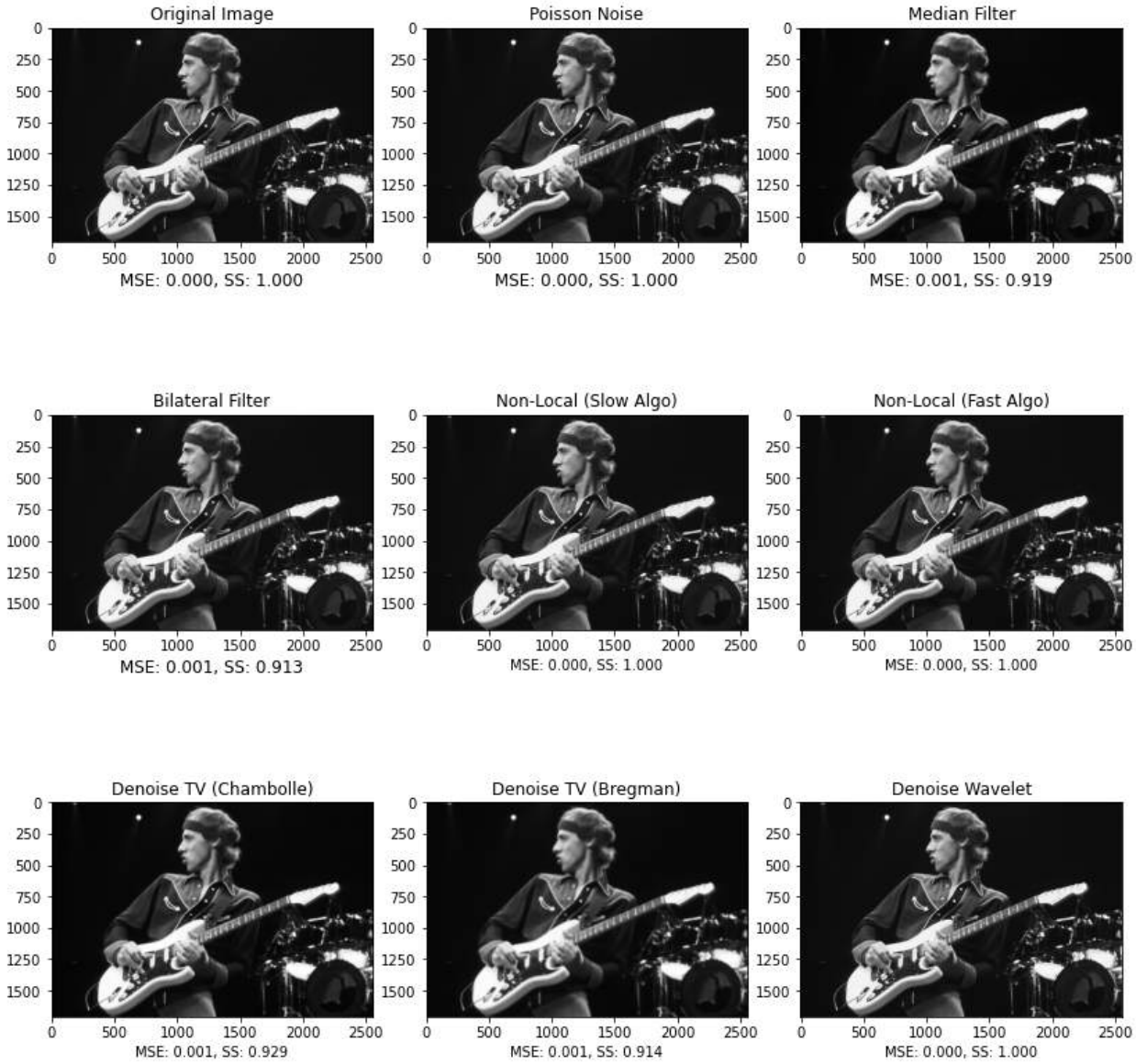

Figure 94: Nonlinear filtering on poisson noise

Figure 95: Nonlinear filtering on poisson noise

Figure 96: Nonlinear filtering on poisson noise

As seen again here, the poisson noise does not add a substantial amount of noise to be able to analyze the nonlinear filters and should be disregarded in analyzing the overall effectiveness of the nonlinear filters in denoising images. We move on to the final type of noise, speckled.

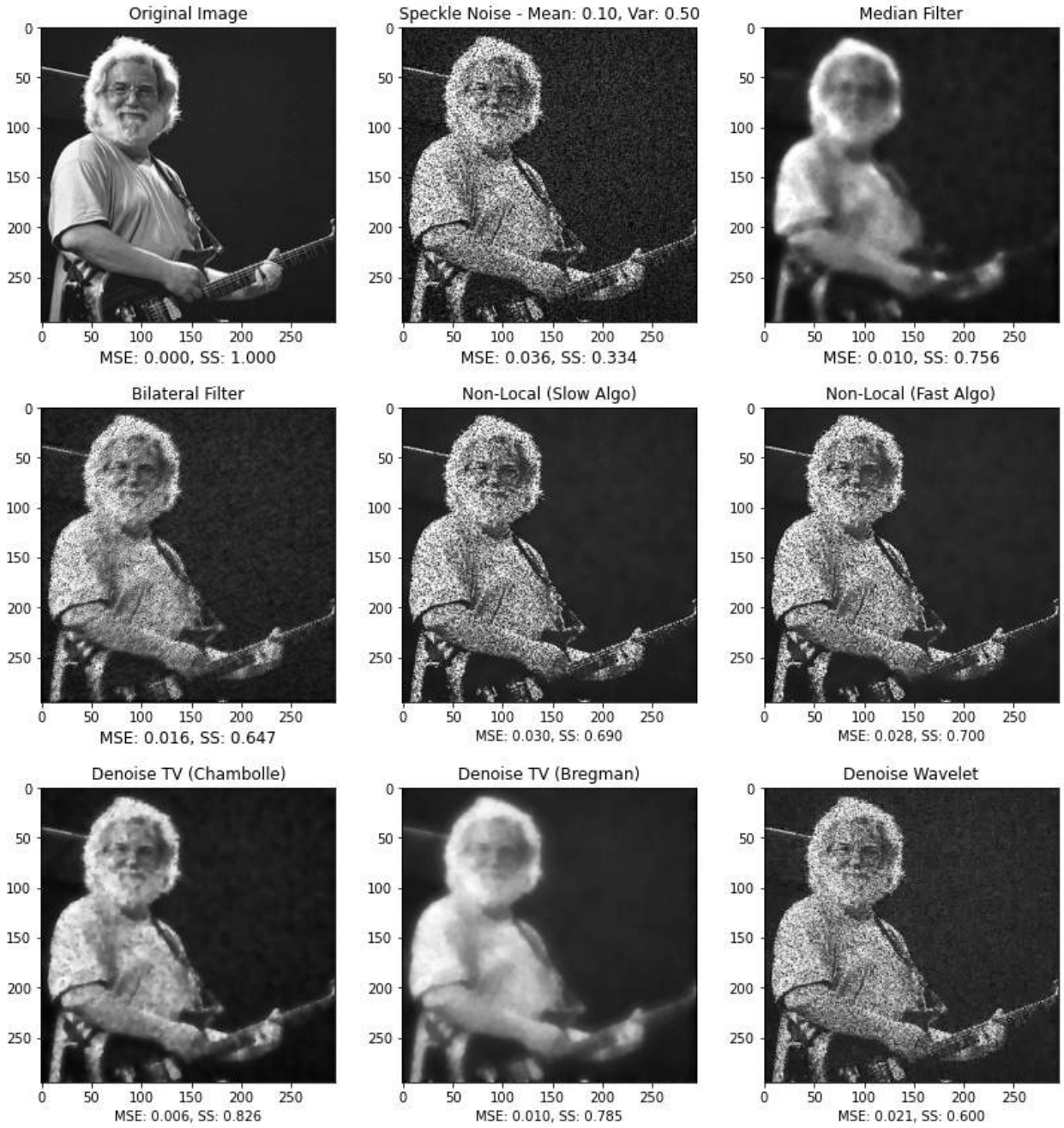

Figure 97: Nonlinear filtering on speckled noise, mean of 0.1 and variance of 0.5

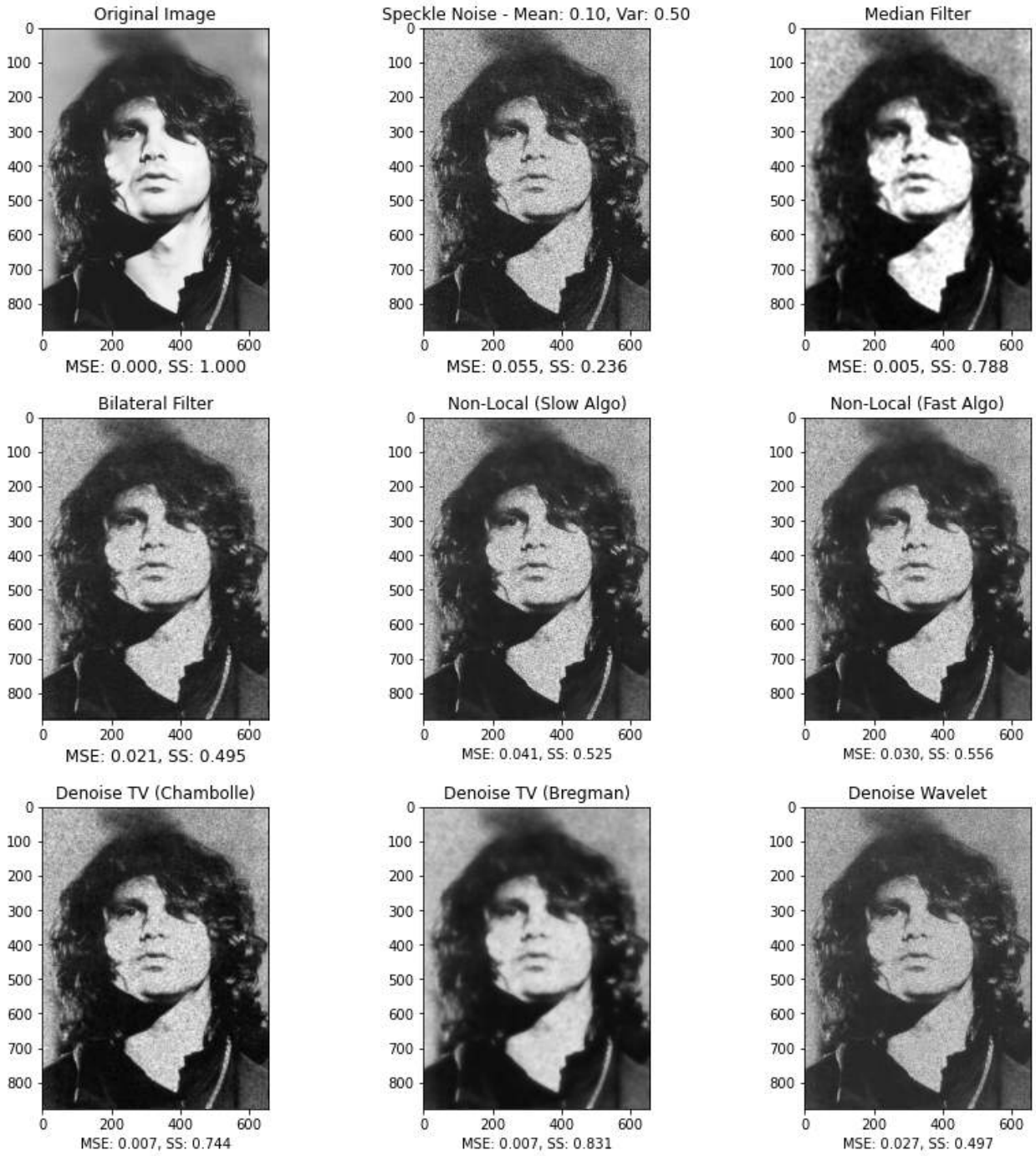

Figure 98: Nonlinear filtering on speckled noise, mean of 0.1 and variance of 0.5

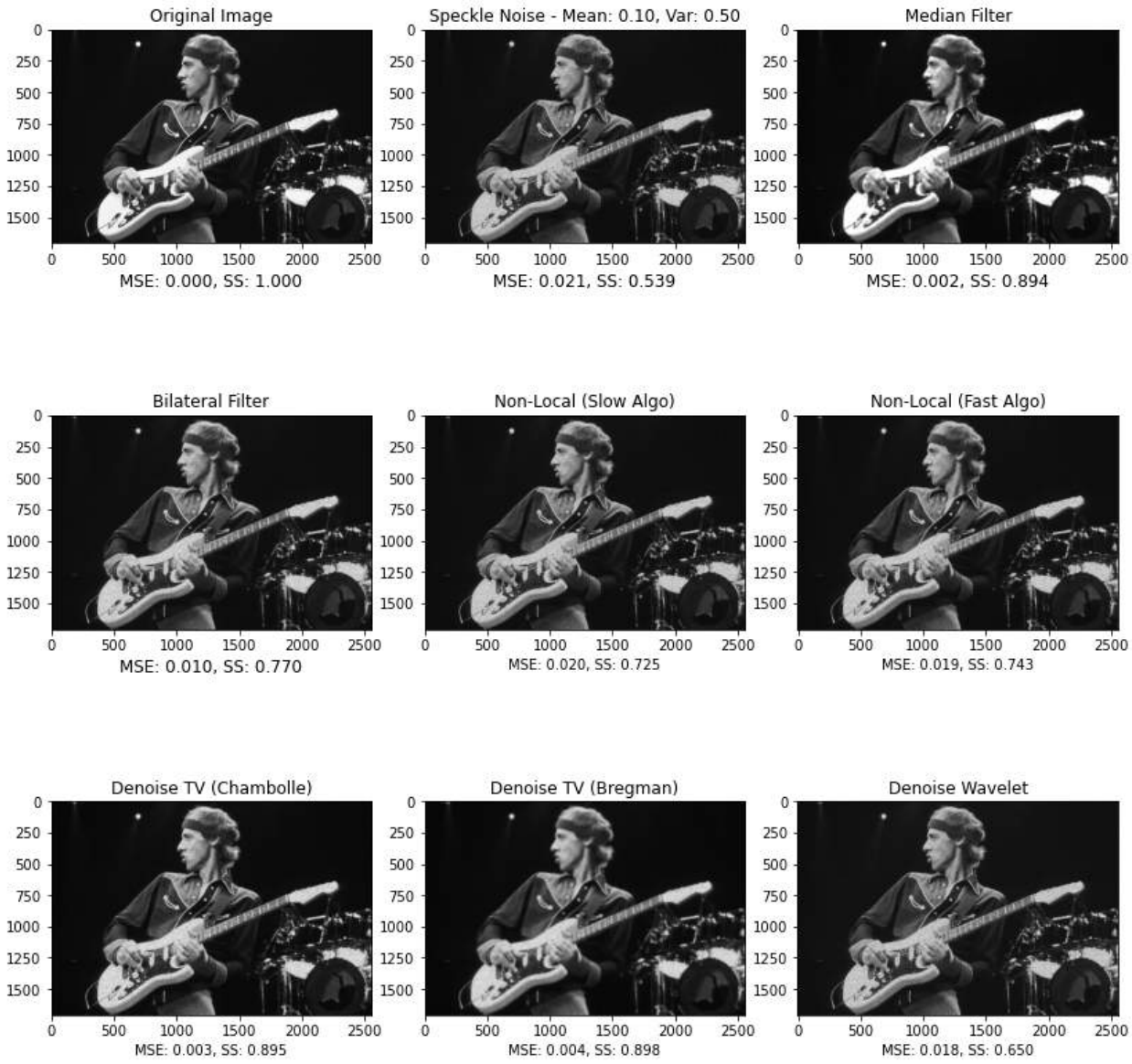

Figure 99: Nonlinear filtering on speckled noise, mean of 0.1 and variance of 0.5

Figure 100: Nonlinear filtering on speckled noise, mean of 0.1 and variance of 0.5

Figure 101: Nonlinear filtering on speckled noise, mean of 0.1 and variance of 0.5

As seen with previous similar amounts of noise in gaussian, the median filter does a great job at denoising. As the variance starts to increase in the speckled noise, we should see its effectiveness start to decrease.

Figure 102: Nonlinear filtering on speckled noise, mean of 0.25 and variance of 0.25

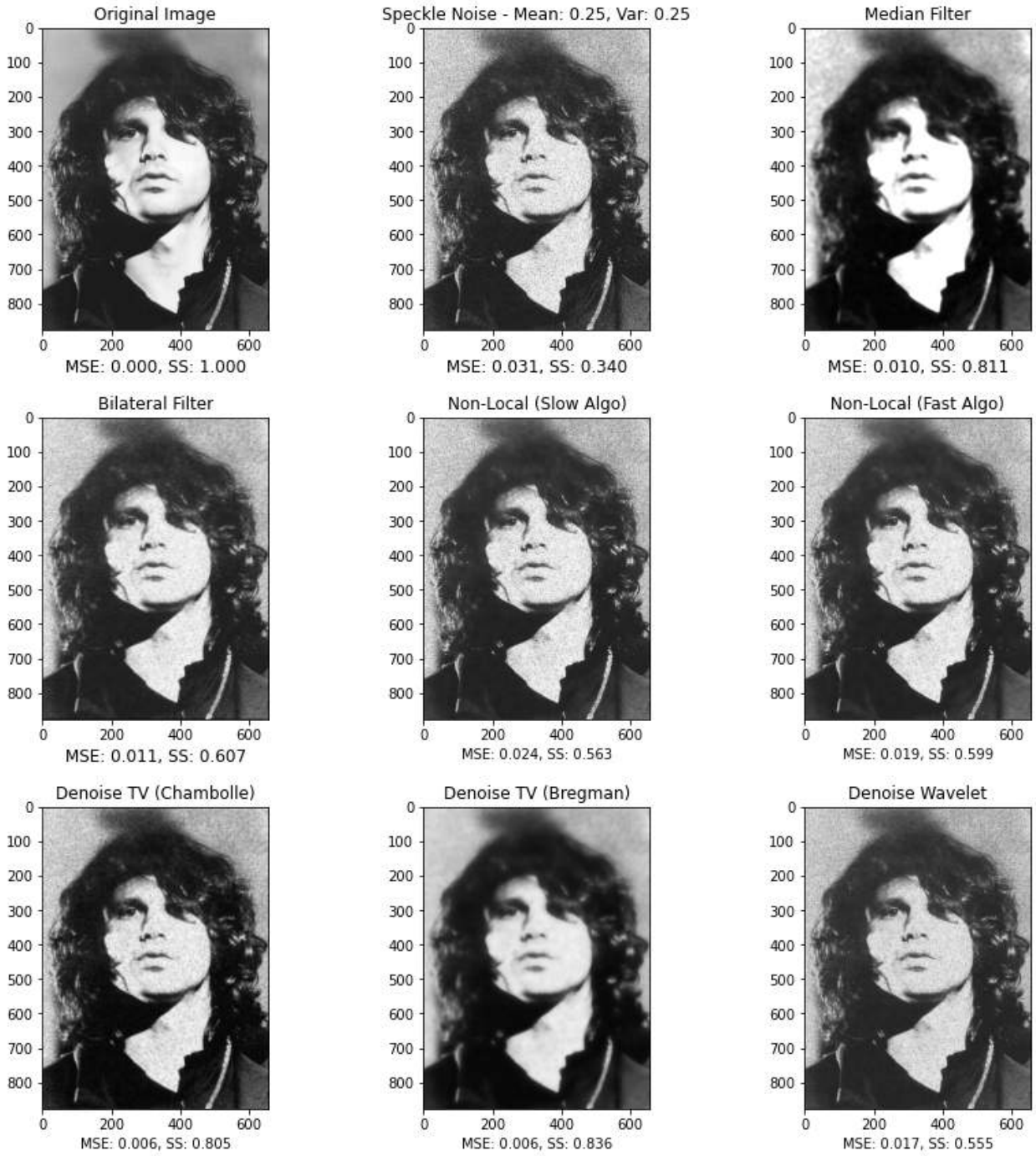

Figure 103: Nonlinear filtering on speckled noise, mean of 0.25 and variance of 0.25

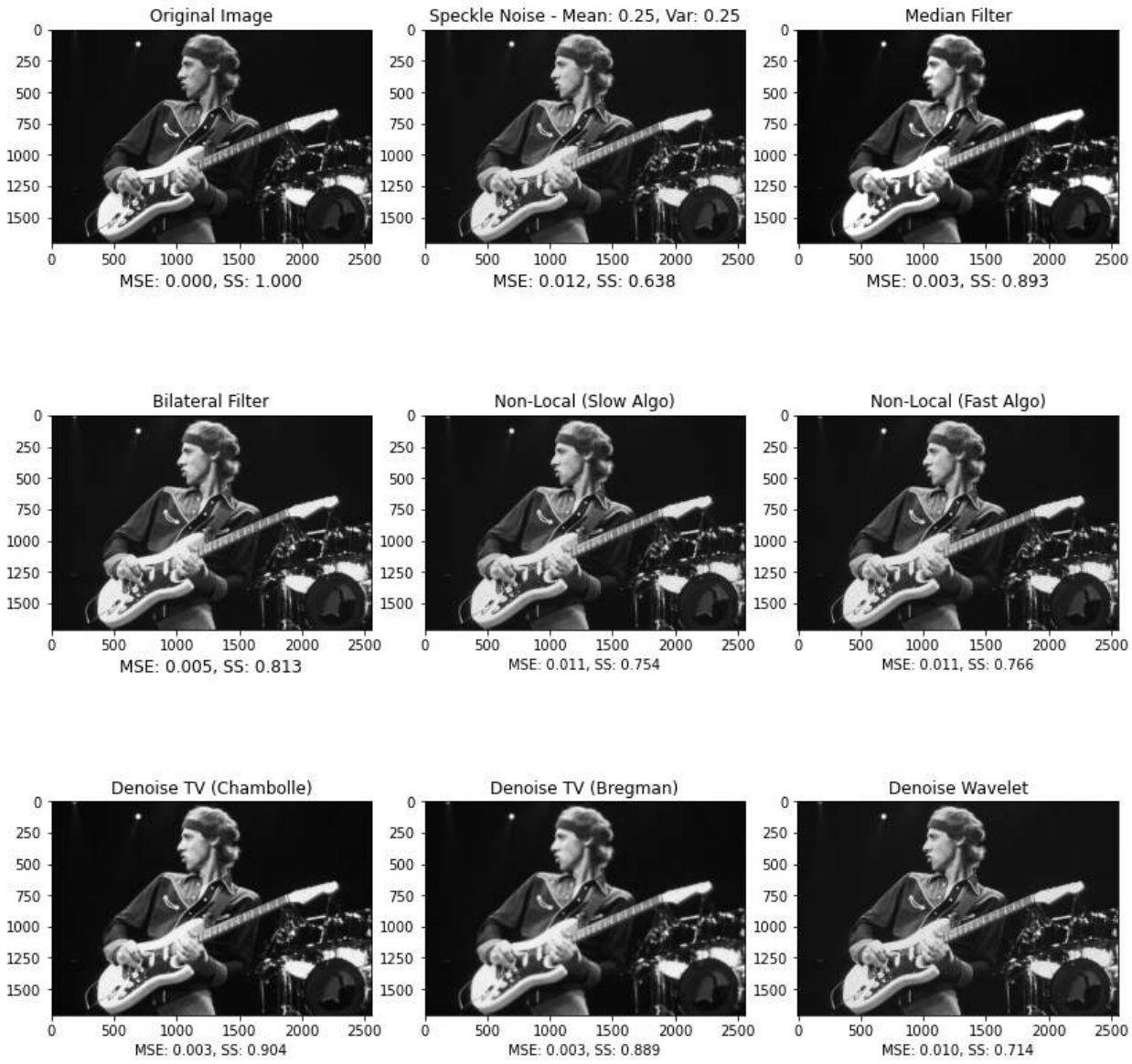

Figure 104: Nonlinear filtering on speckled noise, mean of 0.25 and variance of 0.25

Figure 105: Nonlinear filtering on speckled noise, mean of 0.25 and variance of 0.25

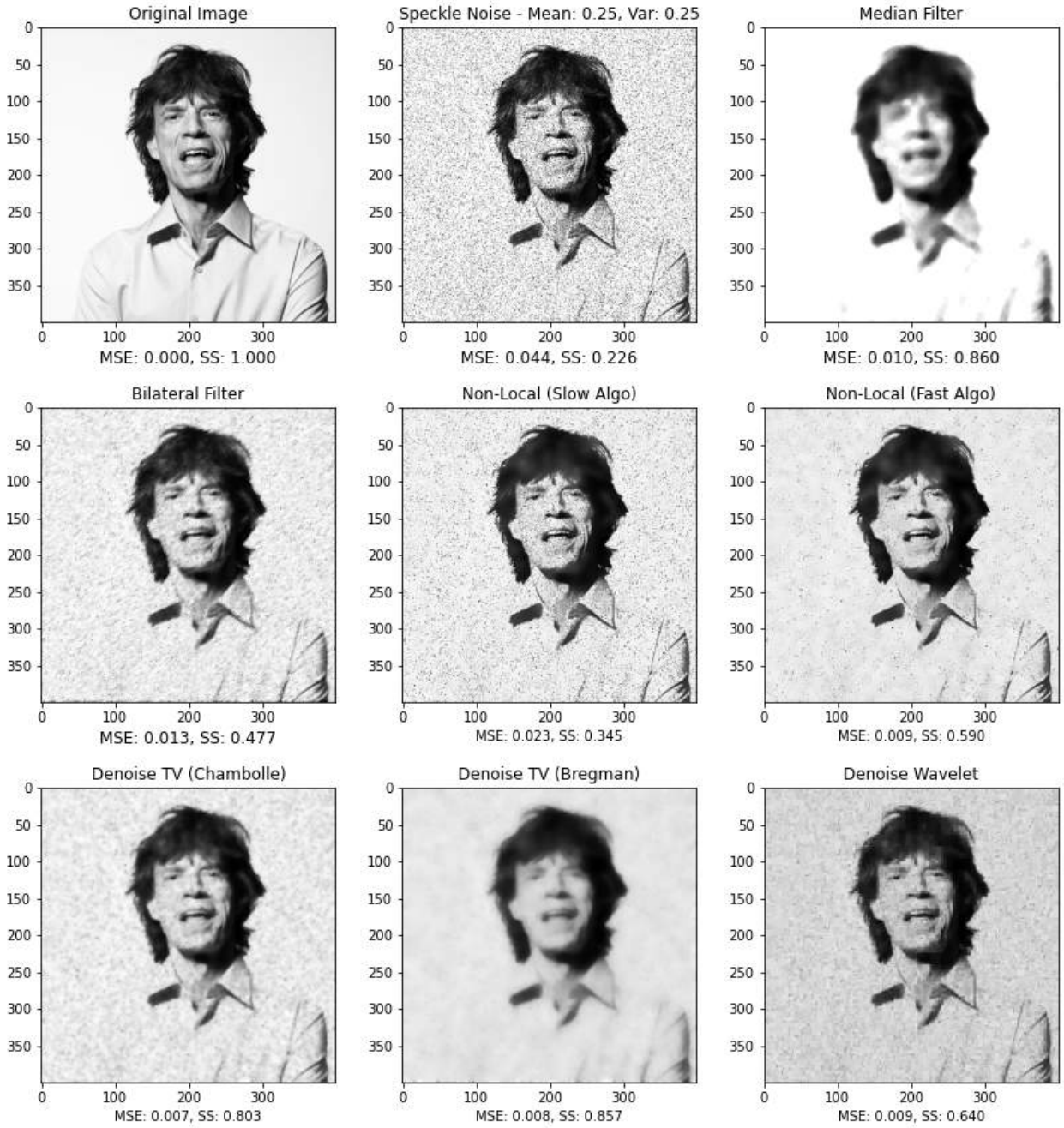

Figure 106: Nonlinear filtering on speckled noise, mean of 0.25 and variance of 0.25

With the overall increase in noise, we see that the median filter has lost some of its effectiveness as the denoise tv bregman and chambolle become the best statistical denoisers of the set. With an increase in noise coming in trial 3, we expect to see this trend continue as the median filter loses its effectiveness and the more rigorous algorithms begin to shine.

Figure 107: Nonlinear filtering on speckled noise, mean of 0.5 and variance of 0.1

Figure 108: Nonlinear filtering on speckled noise, mean of 0.5 and variance of 0.1

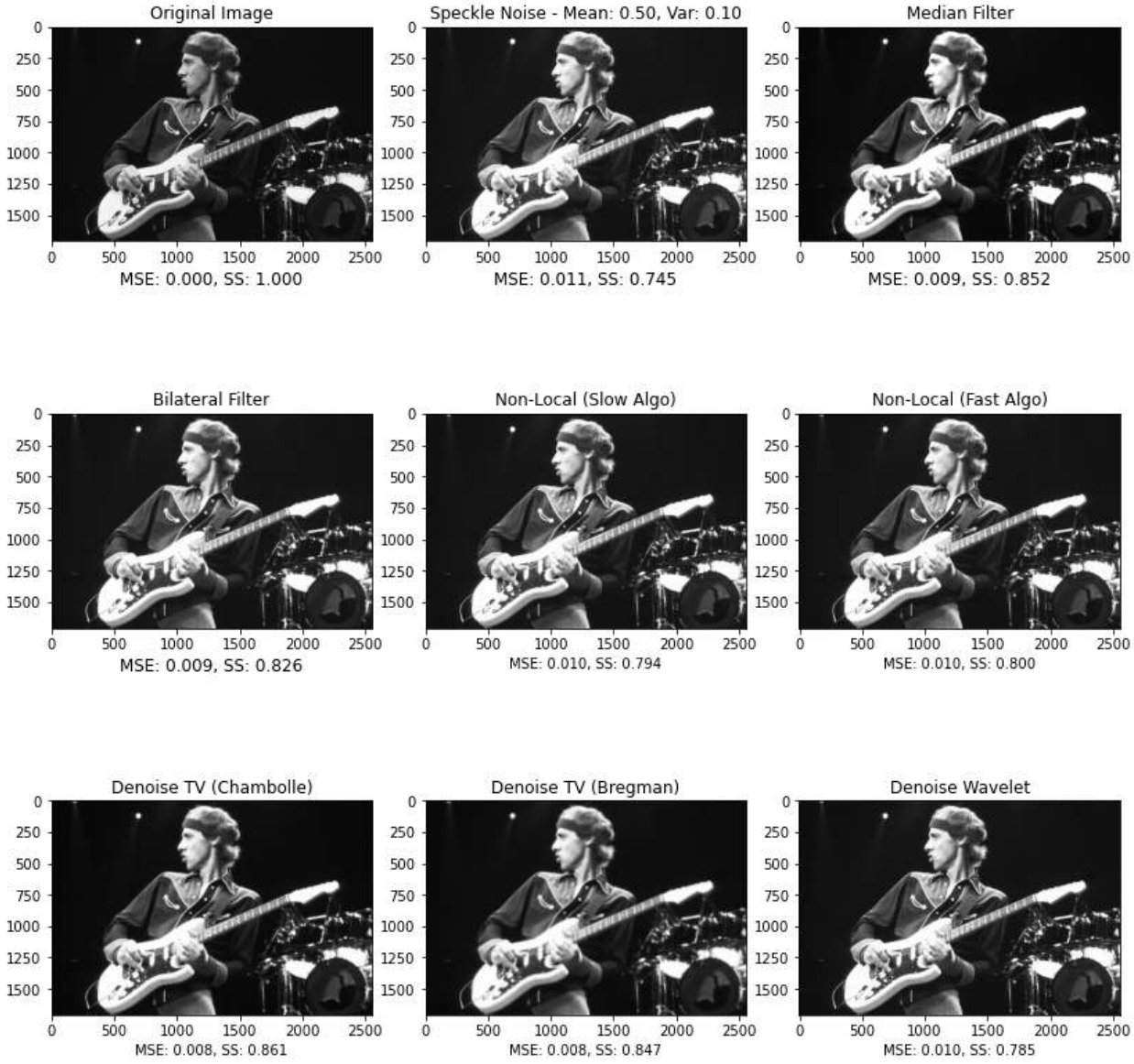

Figure 109: Nonlinear filtering on speckled noise, mean of 0.5 and variance of 0.1

Figure 110: Nonlinear filtering on speckled noise, mean of 0.5 and variance of 0.1

Figure 111: Nonlinear filtering on speckled noise, mean of 0.5 and variance of 0.1

As expected, with the increase in noise amount, we see the effectiveness of the median filter decrease as the denoise tv chambolle and bregman pull out as the visually and statistically better denoisers. Here, we also analyze the effectiveness of the denoise wavelet filter, which is the only filter in Figure 11 that is able to hold on to the background contract effectively, although the pixelated noise in the foreground is still present.

Conclusion

After analyzing both linear and nonlinear filters on the same images with the same amounts of noise, it is clear that the nonlinear filters outperform the linear filters. To no surprise, neither the linear or nonlinear filters are truly able to denoise images that have a lot of noise added in, but the nonlinear filters substantially outperform the linear filters with small amounts of noise. Of the nonlinear filters, the median filter is able to denoise better than the rest. The median filter takes a pixel and averages its neighbors to try and smooth out the image. Since all of the images have constant, consistent colored backgrounds, the median filter is able to shine in its denoising in the foreground of all the images. If more time was dedicated to this project, it would be interesting to see how the median filter is able to respond to a variety of images, with various textures and overall structure. To continue our analysis of denoising, we will next implement the use of neural networks to try and train a model to denoise images through the use of training and testing sets of images, composed of a large quantity of images.

Task ¾ - Neural Networks

We continue our analysis of denoising images, but turn our attention to building and training both linear and nonlinear neural networks to compare their effectiveness in denoising with the filters used in parts 1 and 2. Neural networks are a subset of machine learning techniques that mimic the way biological neurons signal to one another in a human brain. A neural network is composed of layers that contain inputs and outputs as well as associated weights and thresholds. Neural networks use training data to learn and improve their accuracy before being used on testing data. In our analysis of using neural networks to denoise images, we train our models on a set of 500 pokemon cartoon images. We then run our analysis using a variety of different model architectures using Mean Squared Error of the testing data set as a statistical measure of the models effectiveness.

To analyze neural networks' effectiveness in denoising, we vary model parameters and attempt to optimize denoising. Each model is composed at the discretion of the programmer but, for our sake at introducing neural networks for the first time, our networks are made up of various nonlinear activation functions between convolutional layers. Before analyzing different model architectures, I will present the effects that non architecture parameters have on our MSE results. These parameters include number of epochs, batch size, learning rate, and weight decay. I vary these parameters one by one on a linear neural network whose architecture is held constant to allow the varied parameter to act as the dependent variable. After such experiments, I analyze five different neural network models and analyze random testing pictures that have been passed through the model to gauge the model's effectiveness in denoising.

The first model tested was a simple one layer linear network with a convolution network with a kernel size set to 5 by 5 with padding equal to 2:

```
model =  torch.nn.Sequential(torch.nn.Conv2d(1,1,kernel_size = (5,5),padding = 2))
```

Figure 112: Model 1

Being the first model tested, I did not know the optimal learning rate, weight decay, epoch size, and batch size to test the model on. In my first test, I set the number of epochs to 200, batchsize to 4, learning rate to 1e-2 and weight decay to 1e-3. With these parameters, I plotted the loss over epochs, shown in Figure 113.

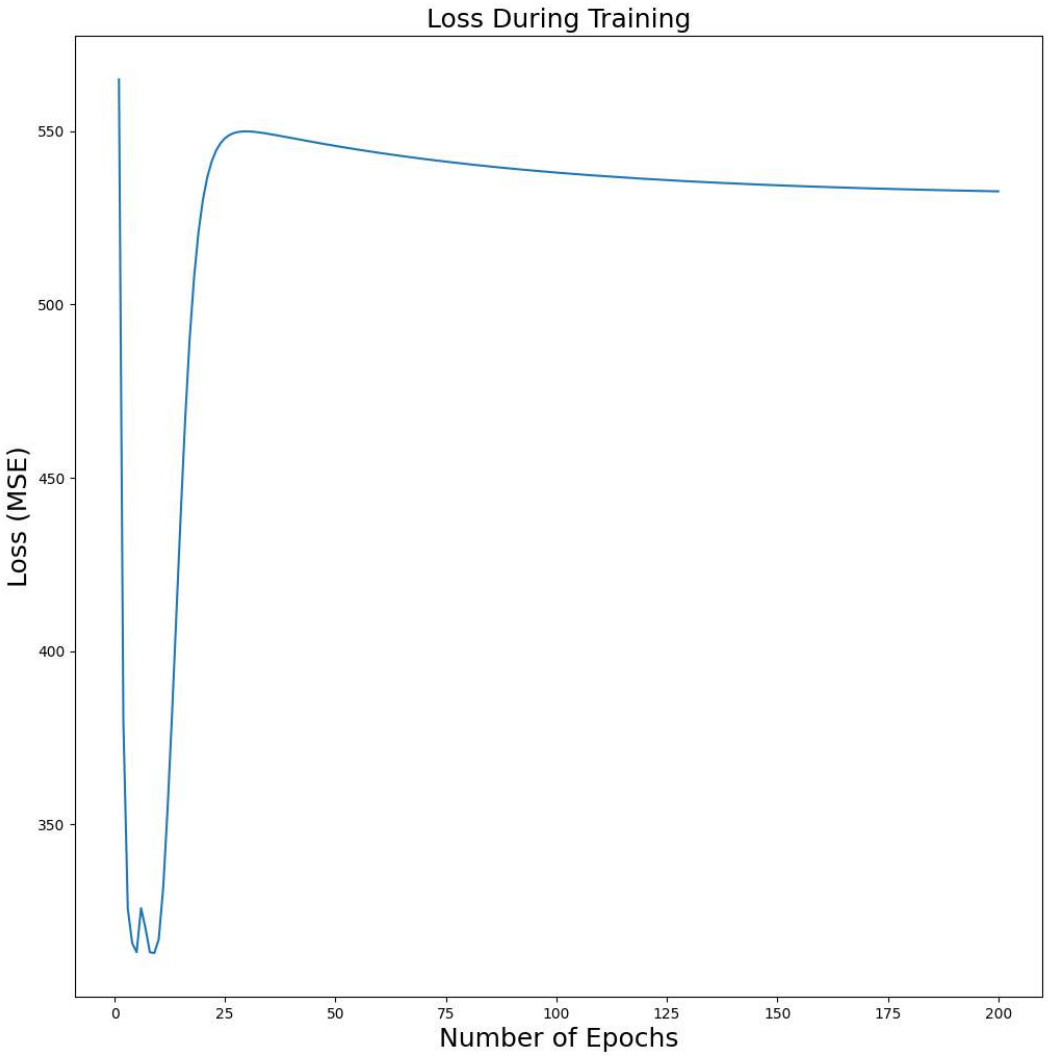

Figure 113: Loss vs Epochs for Model 1: learning rate = 1e-2, batchsize = 4, weight decay = 1e-3

Although this result appears to be learning, as the loss decreases with the number of epochs, it has an unwanted large decrease and rapid increase. Although I could increase the number of epochs to allow the loss to continue to decrease, I know from past experience that 200 epochs is more than enough for the model to converge smoothly. Thus, I took to changing the learning rate and weight decay to analyze their effects on the loss curve over epochs. To further analyze this curves meaning in denoising images, I plotted a few testing images that had been passed through the model:

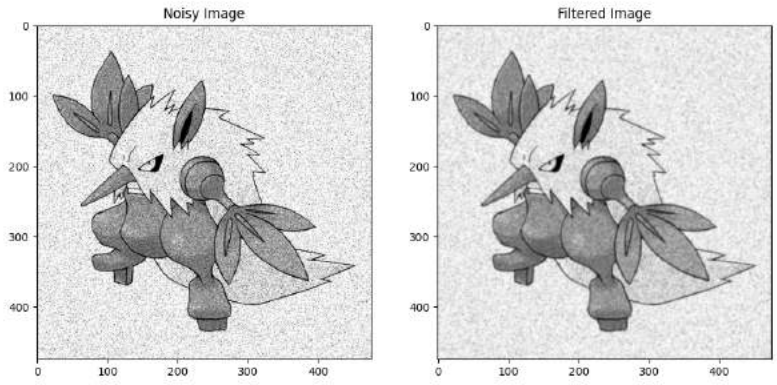

Figure 114: Testing image passed through Model 1, Test 1

From the testing image presented above, the model appears to denoise the image, most evident in the background portion of the cartoon, but only slightly. The MSE for the testing set was found to be 528. In Test 2 (still with model 1 architecture), I set the batch size to 8 and leave the learning rate at 1e-2 and weight decay at 1e-3. The results are presented in the following images:

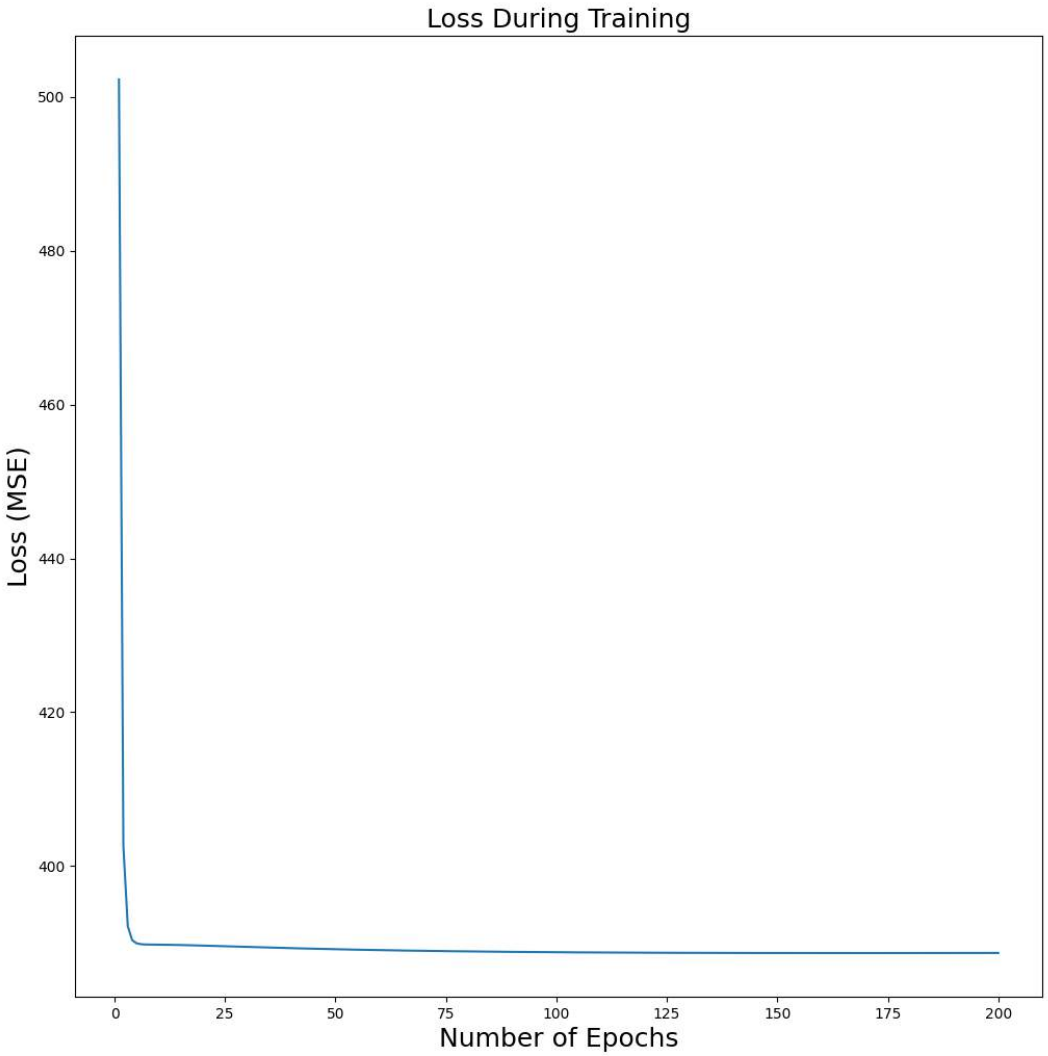

FIgure 115: Loss vs Epochs for Model 1: learning rate = 1e-2, batchsize = 8, weight decay = 1e-3

It appears that increasing the batch size is advantageous for creating a smoother loss graph. The MSE for Test 2 on the testing data was found to be 430. This is a significant decrease from Test 1 which resulted in an MSE of 528. Although the loss graph doesn't have the unwanted rapid decrease followed by increase, it does still have a rapid decrease which we will want to try to further smooth. Let's analyze the resulting images passed through the model:

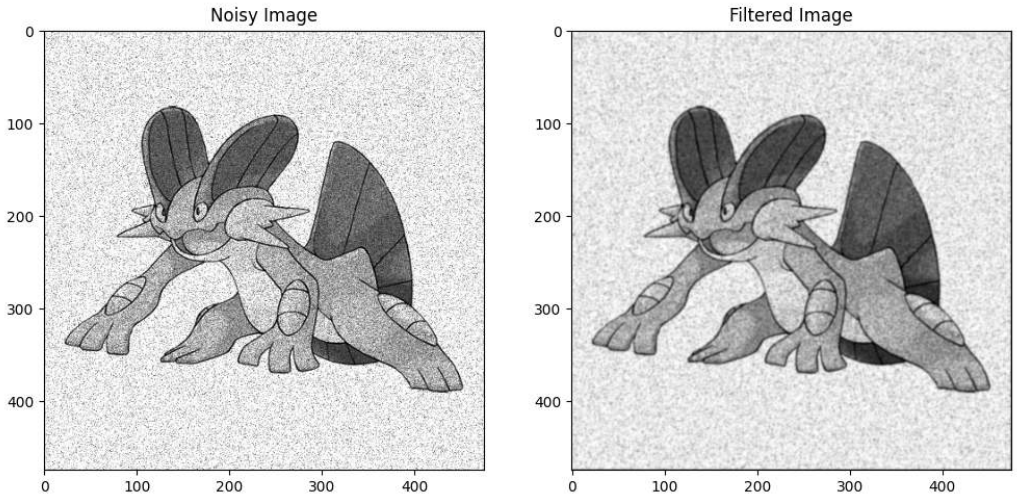

Figure 116: Testing image passed through Model 1, Test 2

In Test 3, I continue to increase the batch size to 16 to further optimize the denoising of the network.

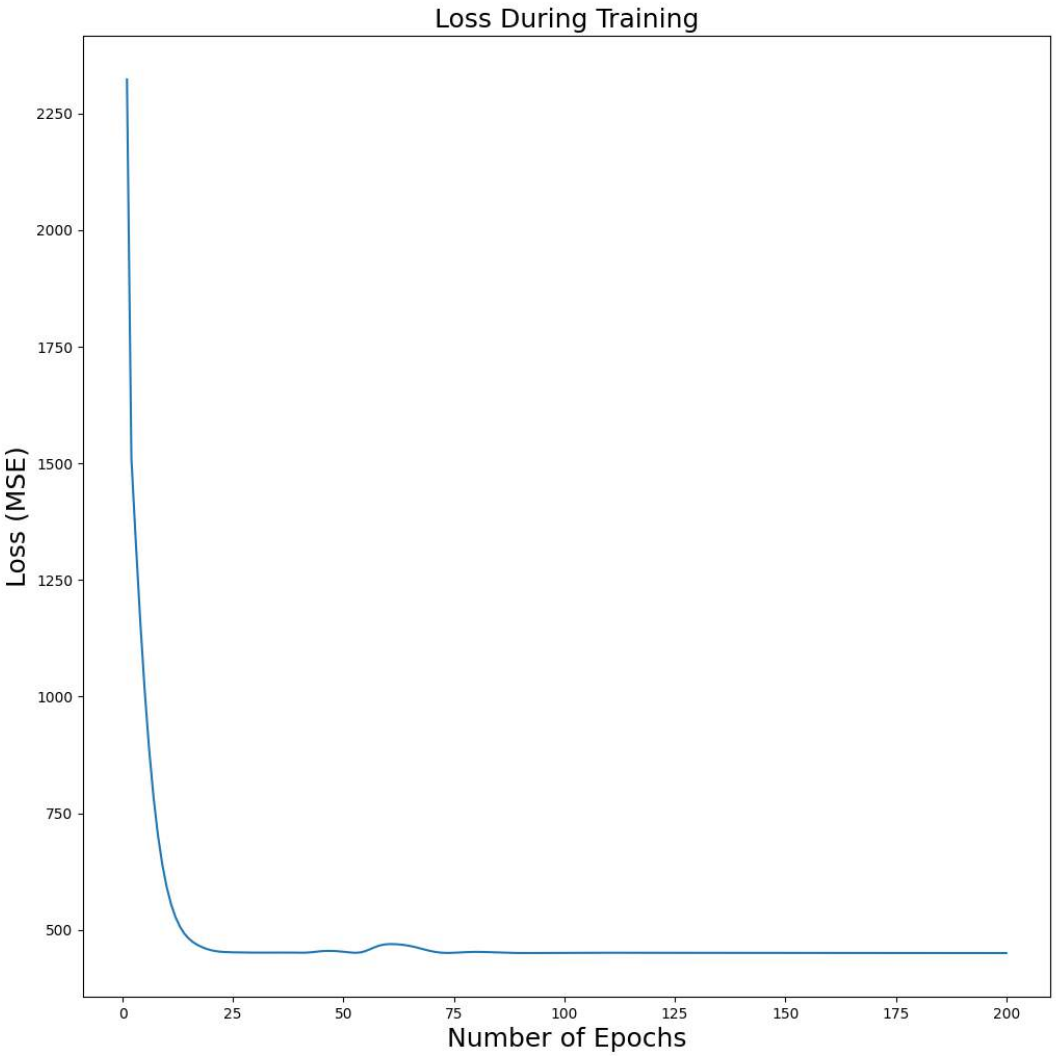

Figure 117: Loss vs Epochs for Model 1: learning rate = 1e-2, batchsize = 16, weight decay = 1e-3

The MSE on the testing data for Test 3 was found to be 427. Although only 3 less than Test 2, the graph does appear to be slightly more smooth as we analyze the gradual decrease over the early epochs rather than the rather sudden and discontinuous decrease seen in Figure 115. The resulting testing image passed through this model is presented below:

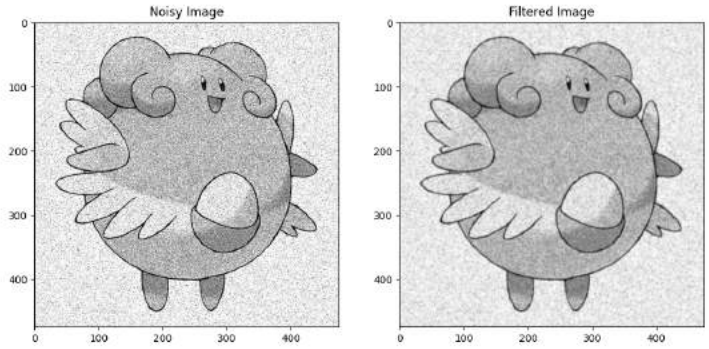

Figure 118: Testing image passed through Model 1, Test 3.

For the final variance on batch size, I increase the batch size from 16 to 32:

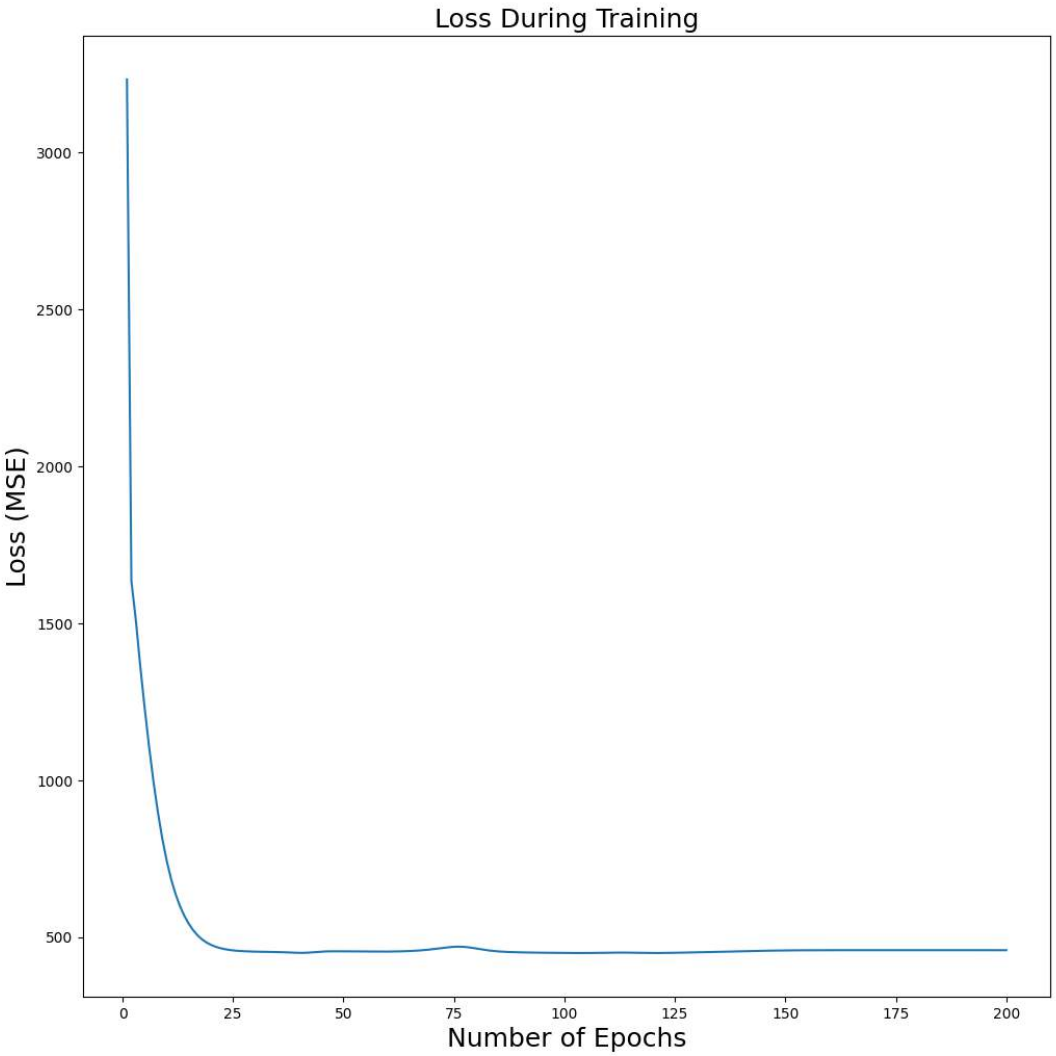

Figure 119: Loss vs Epochs for Model 1: learning rate = 1e-2, batchsize = 32, weight decay = 1e-3

The MSE for Test 4, keeping learning rate at 1e-2, weight decary at 1e-3 but increasing batch size of 32 is found to be 427. This is the same exact MSE value for Test number 3 and thus I conclude that we have optimized the batch size as much as possible.

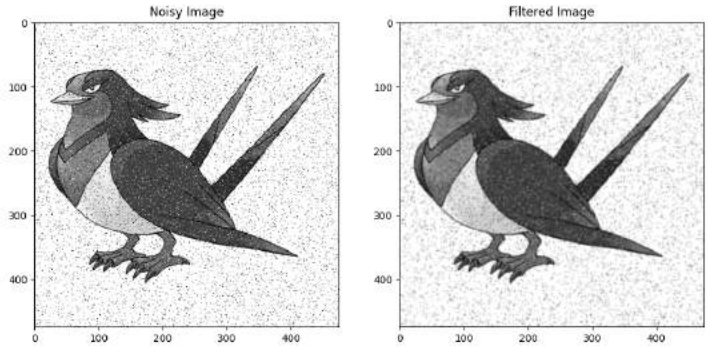

Figure 120: Testing image passed through Model 1, Test 4

It appears that a higher batch size is more advantageous than a lower batch size. In Test 5, I turn my attention to analyzing the learning rate. For subsequent testing, I will keep batch size at 32. Test 5 analyzes results with a weight decay set to 1e-3 but decreasing the learning rate to 1e-3. Results are presented below:

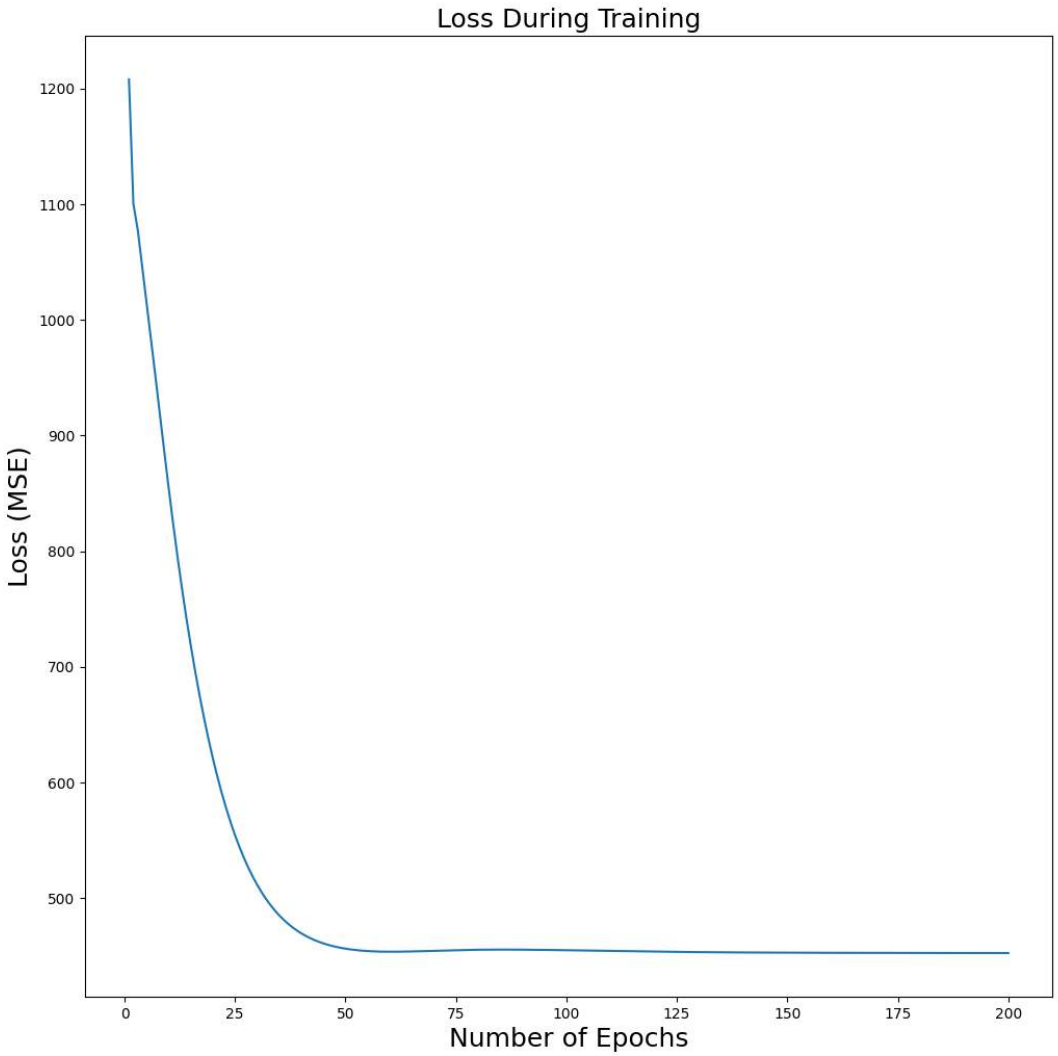

Figure 121: Loss vs Epochs for Model 1: learning rate = 1e-3, batchsize = 32, weight decay = 1e-3

Decreasing the learning rate smoothes out our graph further and results in an MSE on the testing set of 429. From a learning rate of 1e-2 presented earlier, decreasing the learning rate appears to ever so slightly decrease the model's ability to denoise. The biggest takeaway, however, is that decreasing the learning rate appears to take longer for the loss to converge. This is key for our testing on other models that take a significant amount of time to run. Testing images from Test 5 are presented below:

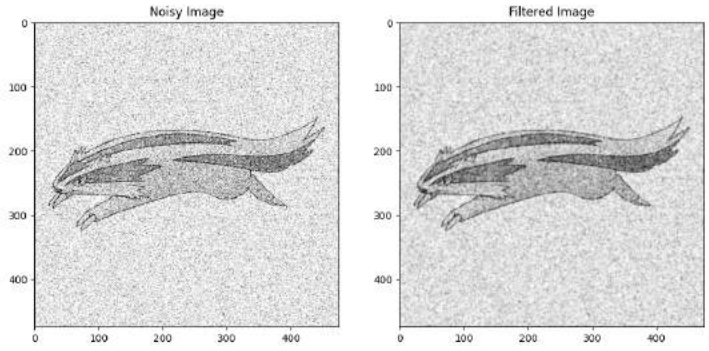

Figure 122: Testing image passed through Model 1, Test 5

In Test 6, I set the learning rate to 1e-1. I analyze an unwanted oscillatory behavior in the loss graph:

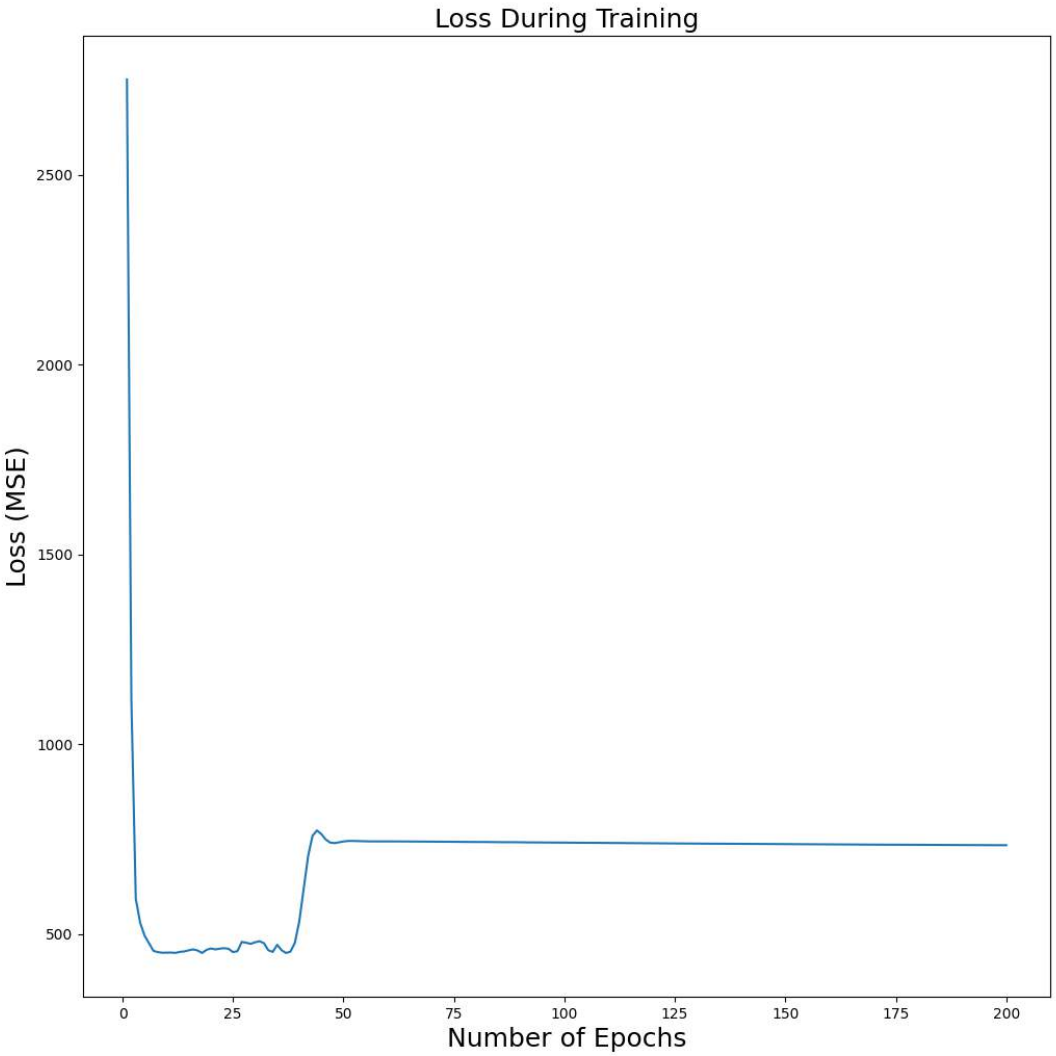

Figure 123: Loss vs Epochs for Model 1: learning rate = 1e-1, batchsize = 32, weight decay = 1e-3

The MSE for this test on the testing data was found to be 432. Although the MSE value is very close in regards to the previous tests, the sudden increase in loss near the 45th epoch is undesirable. The testing image passed through this model is presented in Figure 124.

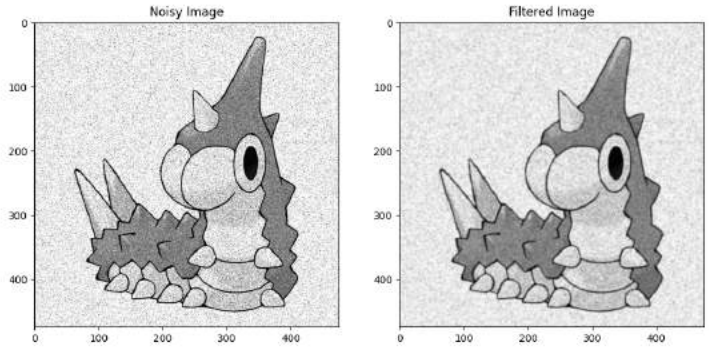

Figure 124: Testing image passed through Model 1, Test 6

In Test 7, we vary the learning rate parameter in the other direction, setting it equal to 1e-4.

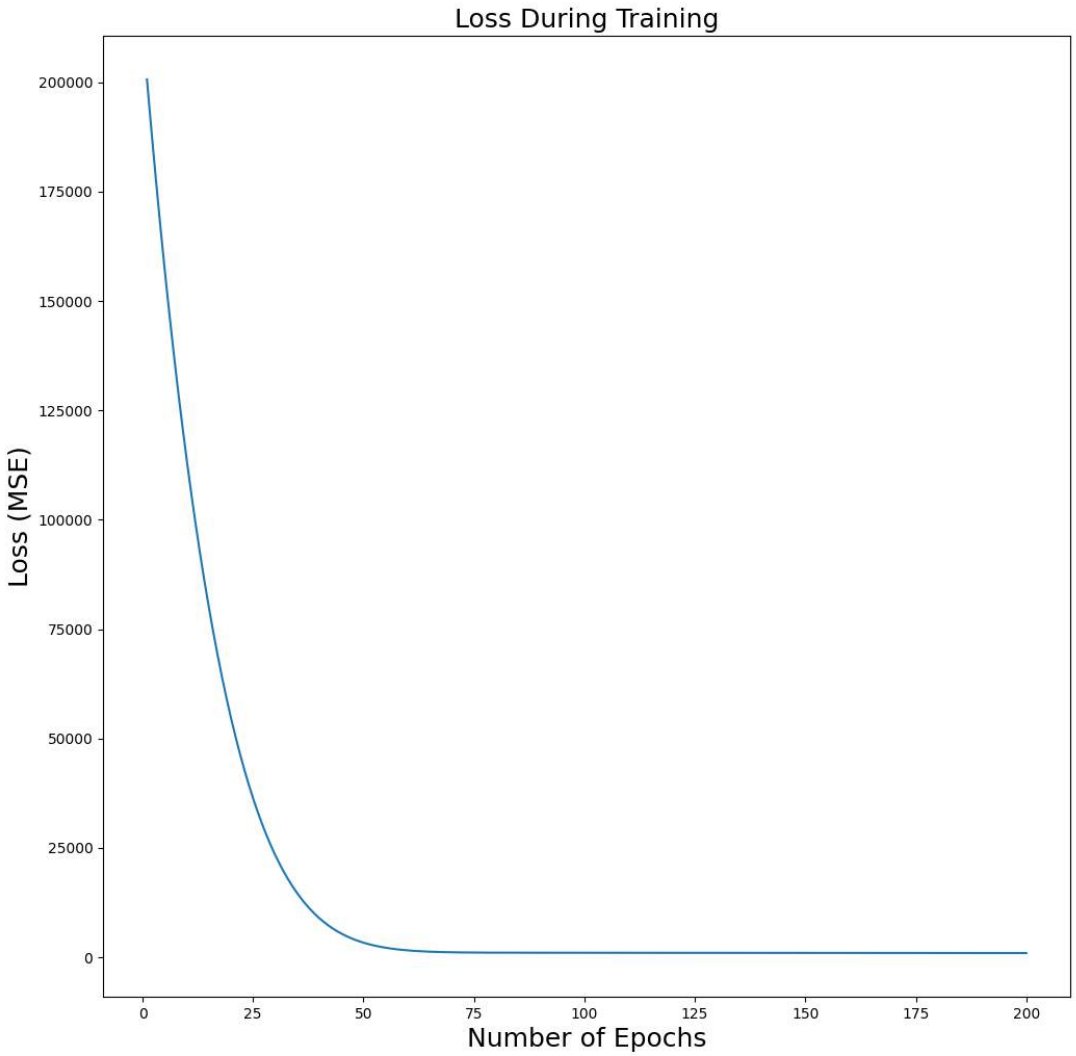

Figure 125: Loss vs Epochs for Model 1: learning rate = 1e-4, batchsize = 32, weight decay = 1e-3

It appears that this learning rate results in a good, smooth loss value over the epoch number. For this particular architecture, this is a good value for the learning rate but it results in a larger MSE value on the testing data, a value of 935. Thus, I conclude that this learning rate is too slow, it takes too long for the loss to converge, something that will be critical in more complex models. The resulting testing image passed through the model is presented below:

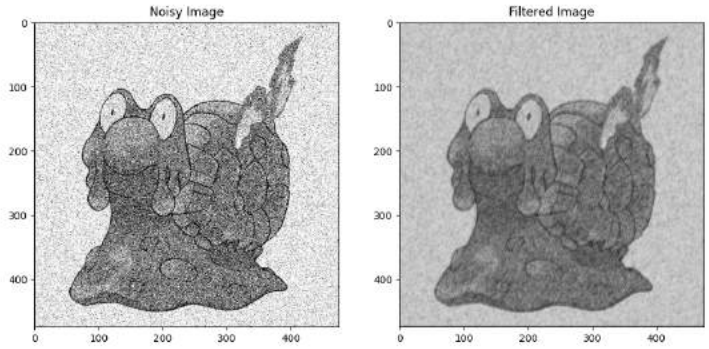

Figure 126: Testing image passed through Model 1, Test 7.

I conclude that for this simple linear model, a learning rate of 1e-3 combines a desirable smooth curve while still making the loss converge at a desirable rate compared to a lower learning rate. I now turn my attention to weight decay. In Test 8 and 9, I hold both the batchsize and learning rate parameters constant at 32 and 1e-3, respectively, and only vary the weight decay. We have already analyzed a weight decay value of 1e-3, so in Test 8, I analyze the outcomes with a weight decay value of 1e-4:

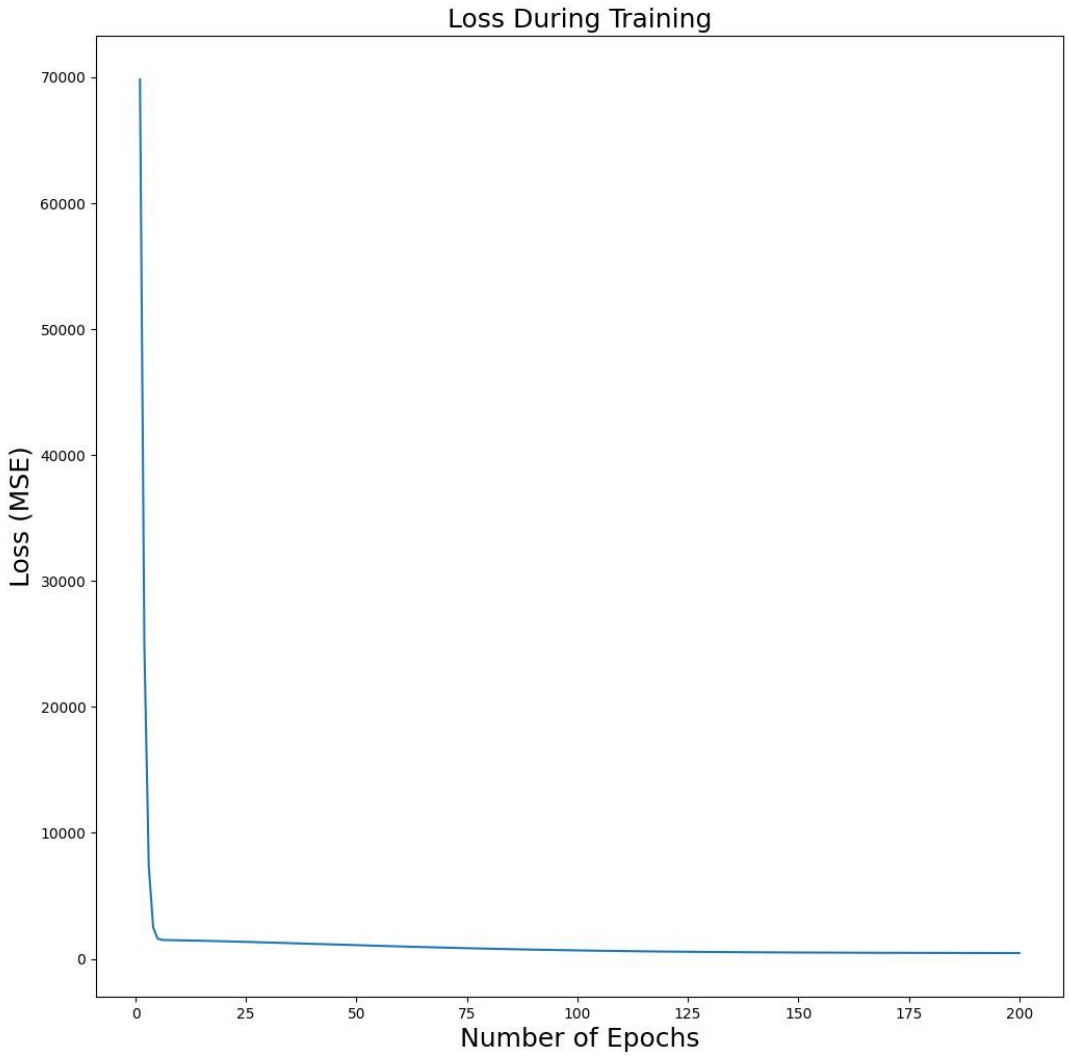

Figure 127: Loss vs Epochs for Model 1: learning rate = 1e-3, batchsize = 32, weight decay = 1e-4

Here we see an undesirable sudden drop before convergence instead of a steady gradual decay that shows us our model is learning at a good rate. The MSE on the training data for these parameters was found to be 431.

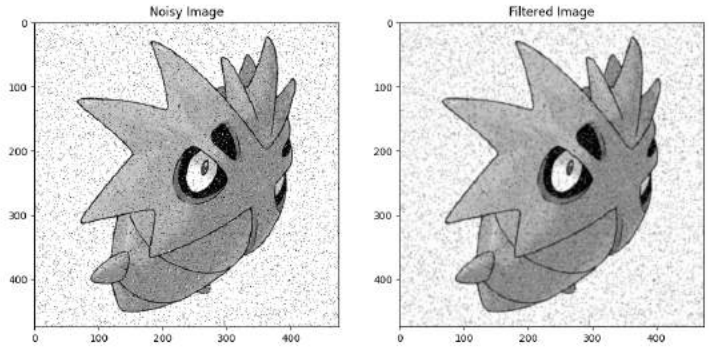

Figure 128: Testing image passed through Model 1, Test 8

In Test 9, I set the weight decay to 1e-2. The MSE on the testing data was found to be 433.

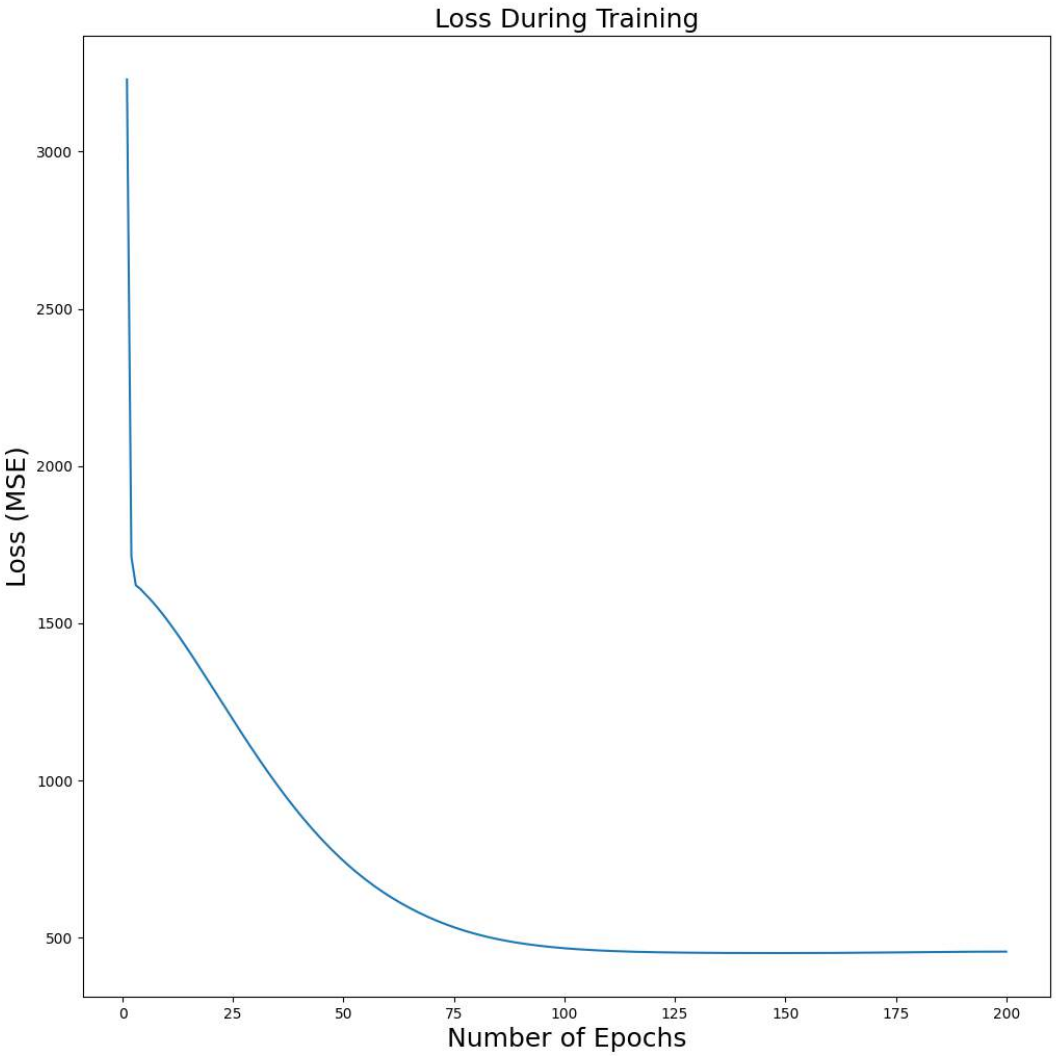

Figure 129: Loss vs Epochs for Model 1: learning rate = 1e-3, batchsize = 32, weight decay = 1e-2

Here, we see an undesirable discontinuity near the 5th epoch. The MSE on the testing set for these parameters was found to be 433.

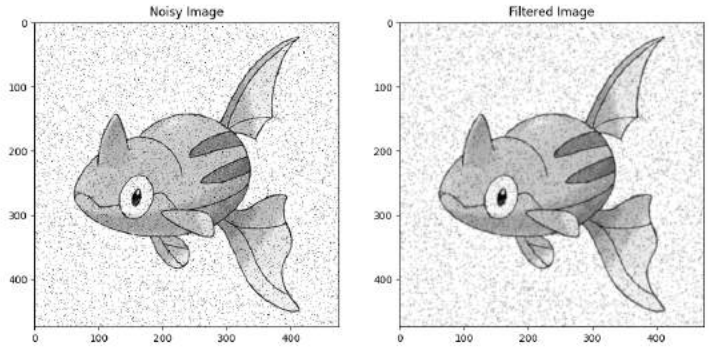

Figure 130: Testing image passed through Model 1, Test 9

After Test 9, I conclude that a weight decay value of 1e-3 yields the most desirable loss graph while keeping the MSE as low as possible. Results of Tests 1-9 are summarized in Table 1:

Test Number	Batch Size	Learning Rate	Weight Decay	MSE on training data	Comments on loss curve
1	4	1e-2	1e-3	528	sudden decrease and increase
2	8	1e-2	1e-3	430	sudden decrease
3	16	1e-2	1e-3	427	good, small hump in middle epochs
4	32	1e-2	1e-3	427	small hump in middle epochs
5	32	1e-3	1e-3	429	good
6	32	1e-1	1e-3	432	large increase and convergence at increased value
7	32	1e-4	1e-3	935	smooth but slow convergence
8	32	1e-3	1e-4	431	quick decrease
9	32	1e-3	1e-1	433	discontinuities observed

Table 1: Results from Tests 1-9 on Model 1

From the results in Table 1, I decided to move forward with parameters: batch size = 32, learning rate = 1e-3, weight decay = 1e-3. As I would see in later models, although these parameters worked best for this exact model, they would not always be the best when the model's architectures are changed. In the subsequent models, I kept the number of epochs large enough to see convergence and left the weight decay at 1e-3 but had to alter my learning rate to 1e-2 to see convergence before the max number of epochs was reached. The final results from Model 1, with parameters discussed above are presented below:

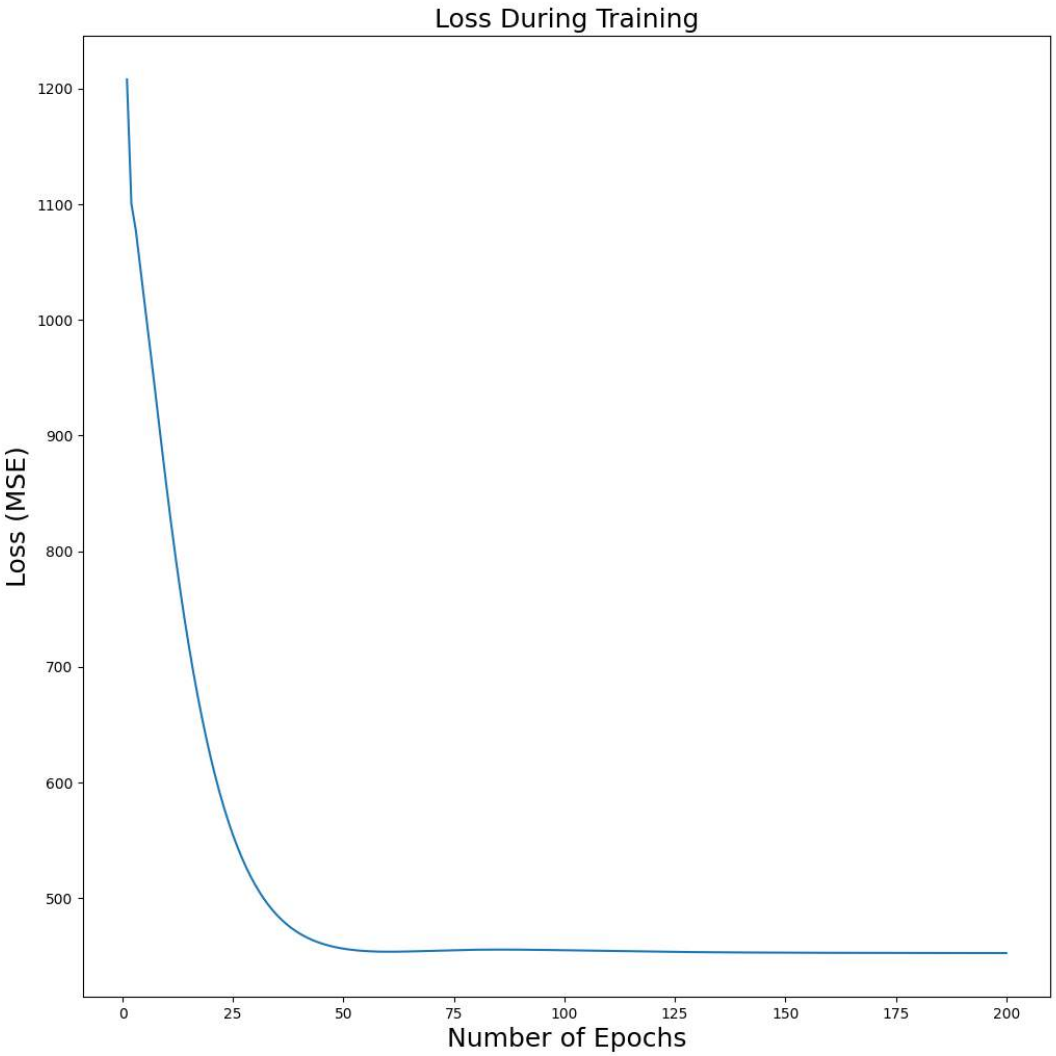

Figure 131: Loss vs Epochs for Model 1: learning rate = 1e-3, batchsize = 32, weight decay = 1e-3 (optimal parameters based on Tests 1-9)

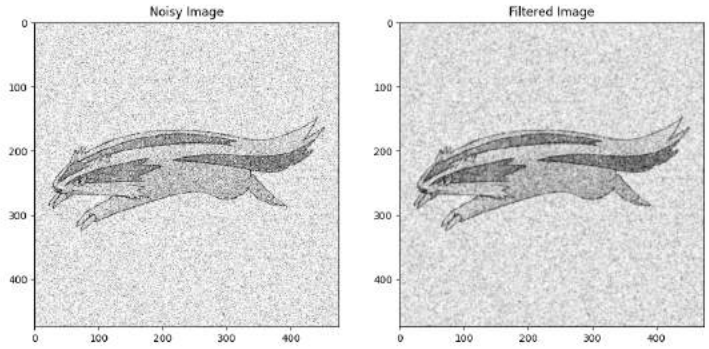

Figure 132: Testing image passed through Model 1 (optimal parameters)

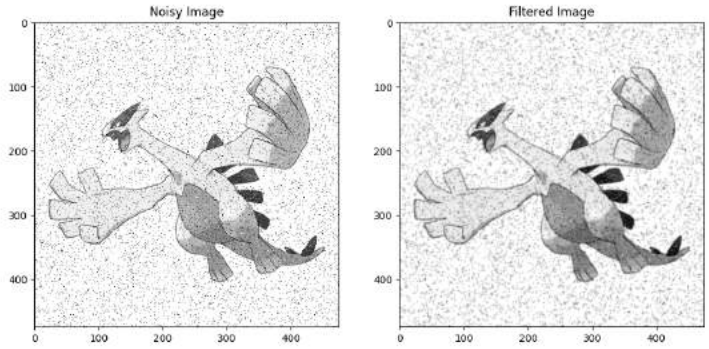

Figure 133: Testing image passed through Model 1 (optimal parameters)

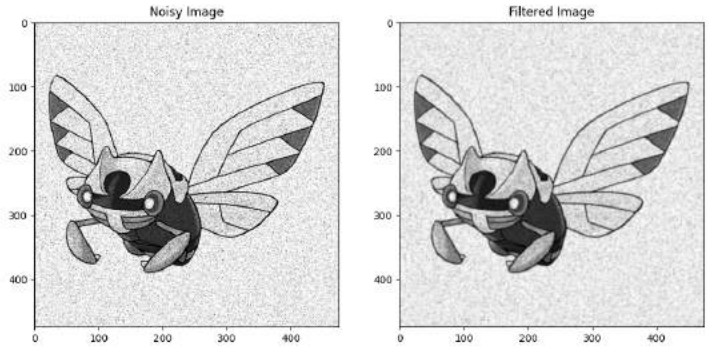

Figure 134: Testing image passed through Model 1 (optimal parameters)

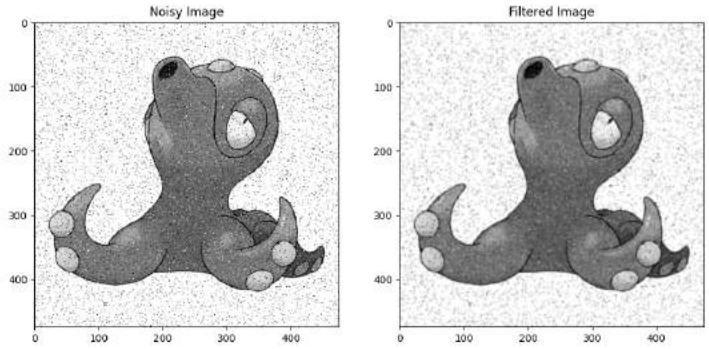

Figure 135: Testing image passed through Model 1 (optimal parameters)

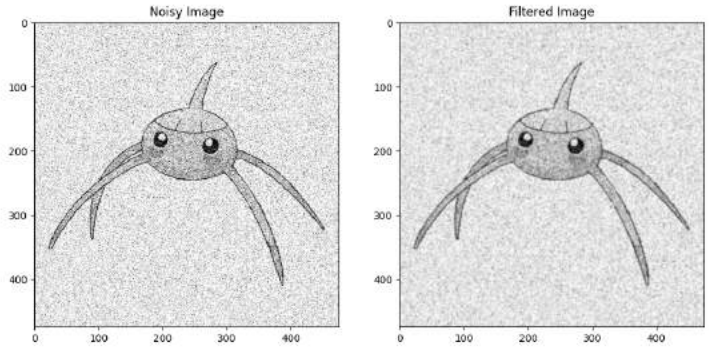

Figure 136: Testing image passed through Model 1 (optimal parameters)

From Figures 132-136, we see slight denoising that is taking place in our testing images. I believe that increasing the complexity of the model and including nonlinear activation functions should result in further denoising. I also tested this model on a separate image that was tested on the nonlinear and linear filters in Parts 1 and 2. This image is presented in Figure 137. As expected, it does not do a great job of denoising this image, which is filled with gaussian noise with mean 0.1 and variance of 0.1. Since we have trained our model on one type of image: pokemon cartoons, we expect to see poor denoising in other types of images.

Figure 137: Non-testing image passed through Model 1

We now turn our attention to Model 2. Model 2 is composed of 5 convolutional layers with batch normalization carried out between layers to aid in convergence. In the middle three layers, I ramp up the number of channels and increase the kennel size to be 7 by 7.

```
model = torch.nn.Sequential(torch.nn.Conv2d(1,2,kernel_size = (5,5),padding = 2),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 2----------
      torch.nn.Conv2d(2,4,kernel_size = (7,7),padding = 3),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 3----------
      torch.nn.Conv2d(4,4,kernel_size = (7,7),padding = 3),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 4----------
      torch.nn.Conv2d(4,2,kernel_size = (7,7),padding = 3),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 5----------
      torch.nn.Conv2d(2,1,kernel_size = (5,5),padding = 2),
      torch.nn.BatchNorm2d(1, affine = True)
)
```

Figure 138: Model 2 architecture

Results of Model 2 are presented below. It must be noted that in first testing Model 2, the optimal parameters were taken from Model 1, but it was observed that the model took too long to converge and was out of bounds of the 500 epoch range. Thus, I decrease the learning rate to a value of 1e-1 to produce the results presented. The MSE on the training dataset was found to be 478.

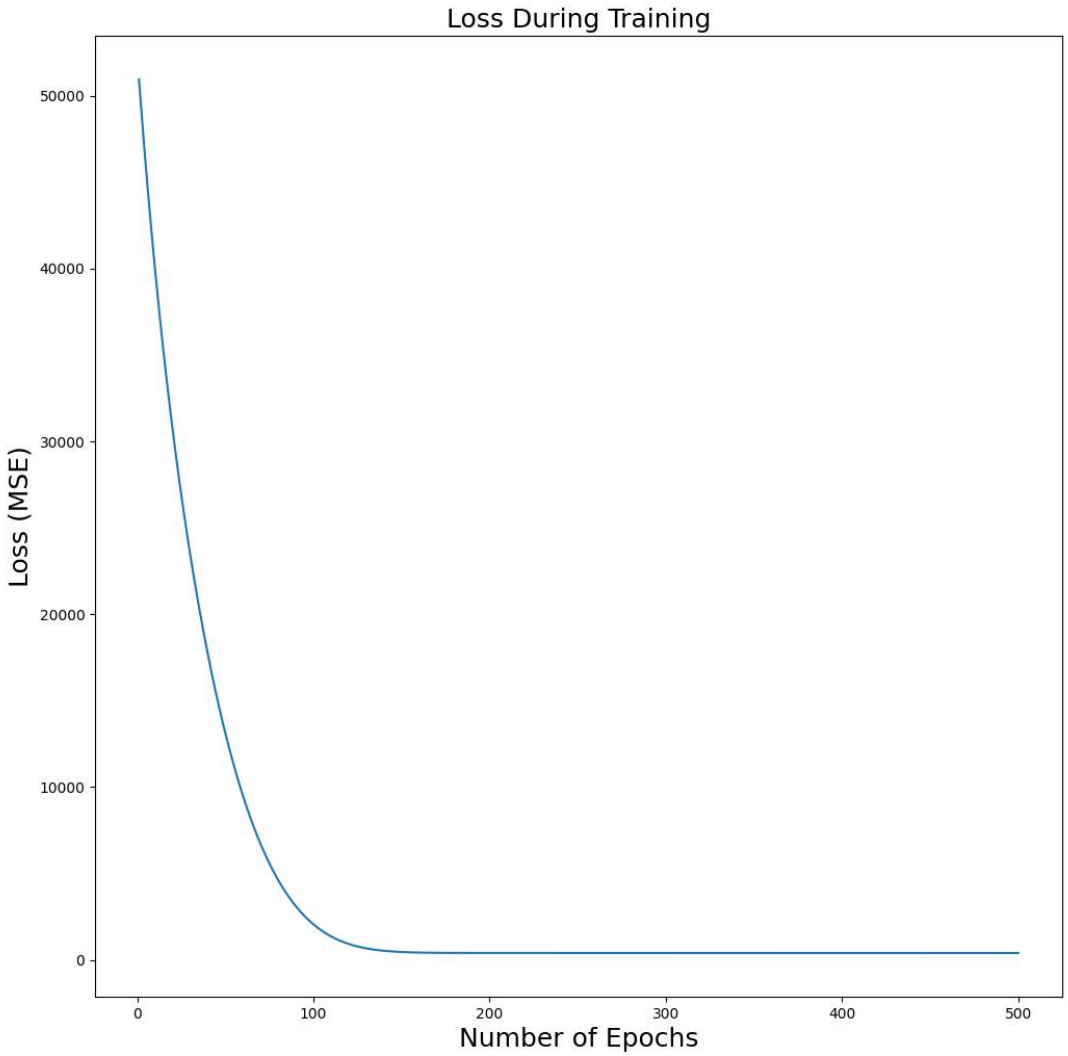

Figure 139: Loss vs Epochs for Model 2: learning rate = 1e-1, batchsize = 32, weight decay = 1e-3

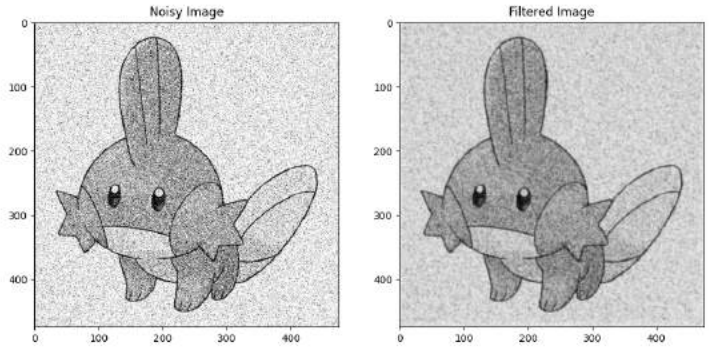

Figure 140: Testing image passed through Model 2

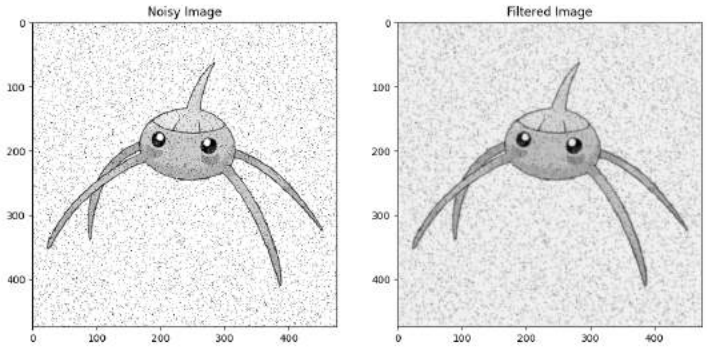

Figure 141: Testing image passed through Model 2

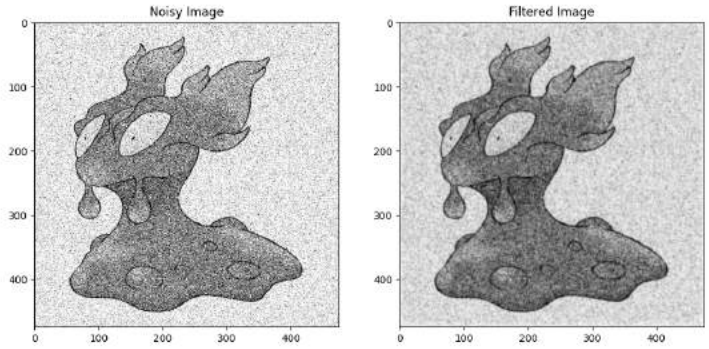

Figure 142: Testing image passed through Model 2

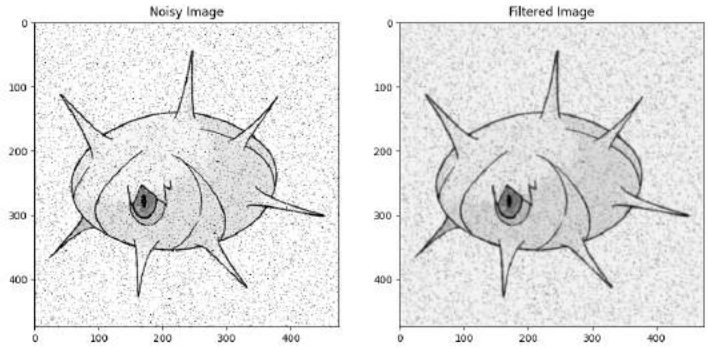

Figure 143: Testing image passed through Model 2

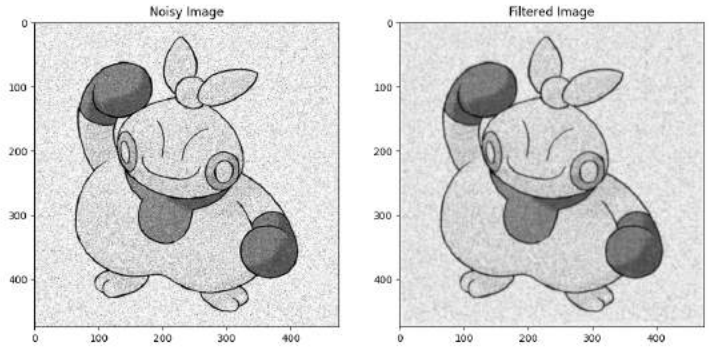

Figure 144: Testing image passed through Model 2

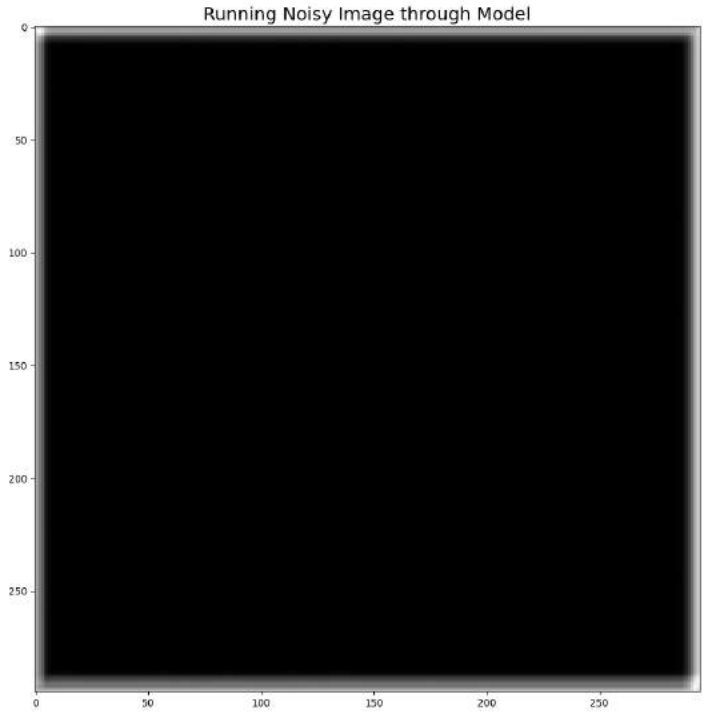

Figure 145: Non-testing image passed through Model 2

From the figures presented above, it is clear that our neural network is learning to denoise images of pokemon cartoons. The MSE on the training data set is higher than observed for Model 1 but only slightly. The main difference we see now is that we are no longer able to denoise a non-testing image when passed through the Model. At first I thought this was a bug in my code. I output the picture that had been run through the model and normalized the pixels to be between 0 and 1 to match up with the testing images. But upon reviewing the resulting pixel values, it was clear that the model actually denoised by creating a completely black picture with very little grey and white pixels that are clustered around the edges of the image, seen in Figure 145. This continued to be a common theme with more complex models, as similar results were observed for the remainder of the models. Thus, this non-testing image will not be presented for further models but it does highlight the fact that our neural network is training on a specific type of image and does not handle images outside of this framework. With a relatively simple model seen in Model 1, we were able to somewhat handle the outside case but with more complex models that aim to create more advantageous denoising with complex layers and activation functions, we are no longer able to handle images outside of the framework whatsoever.

There were no noteworthy results in the testing images with the increased kernel size. Based on results from Parts 1 and 2, I would expect to see various amounts of blurring introduced with varying kernel sizes but since the amount of visual denoising is rather minimal, it is hard to make this comparison. From a theoretical perspective, we know that an increased kernel size will yield a longer run time. In model 4, a kernel size of 11 by 11 was attempted but had to be aborted due

to a substantial increase in overall run time. We move on to view the results of Model 3, which takes the architecture from Model 2 and introduces a non-linear activation function in the layers.

After speaking to a friend involved in PhD research on image processing and speaking with the TA, I chose to introduce the ReLU activation function in Model 3. To analyze solely the effectiveness of introducing this activation function, all other parameters from Model 2 were kept the same, including kernel sizes, batch size, learning rate, and weight decay.

```
model = torch.nn.Sequential(torch.nn.Conv2d(1,2,kernel_size = (5,5),padding = 2),
      torch.nn.ReLU(),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 2----------
      torch.nn.Conv2d(2,4,kernel_size = (7,7),padding = 3),
      torch.nn.ReLU(),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 3----------
      torch.nn.Conv2d(4,4,kernel_size = (7,7),padding = 3),
      torch.nn.ReLU(),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 4----------
      torch.nn.Conv2d(4,2,kernel_size = (7,7),padding = 3),
      torch.nn.ReLU(),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 5----------
      torch.nn.Conv2d(2,1,kernel_size = (5,5),padding = 2),
      torch.nn.ReLU(),
      torch.nn.BatchNorm2d(1, affine = True)
)
```

Figure 146: Model 3

Results from Model 3 were very good. Presented below:

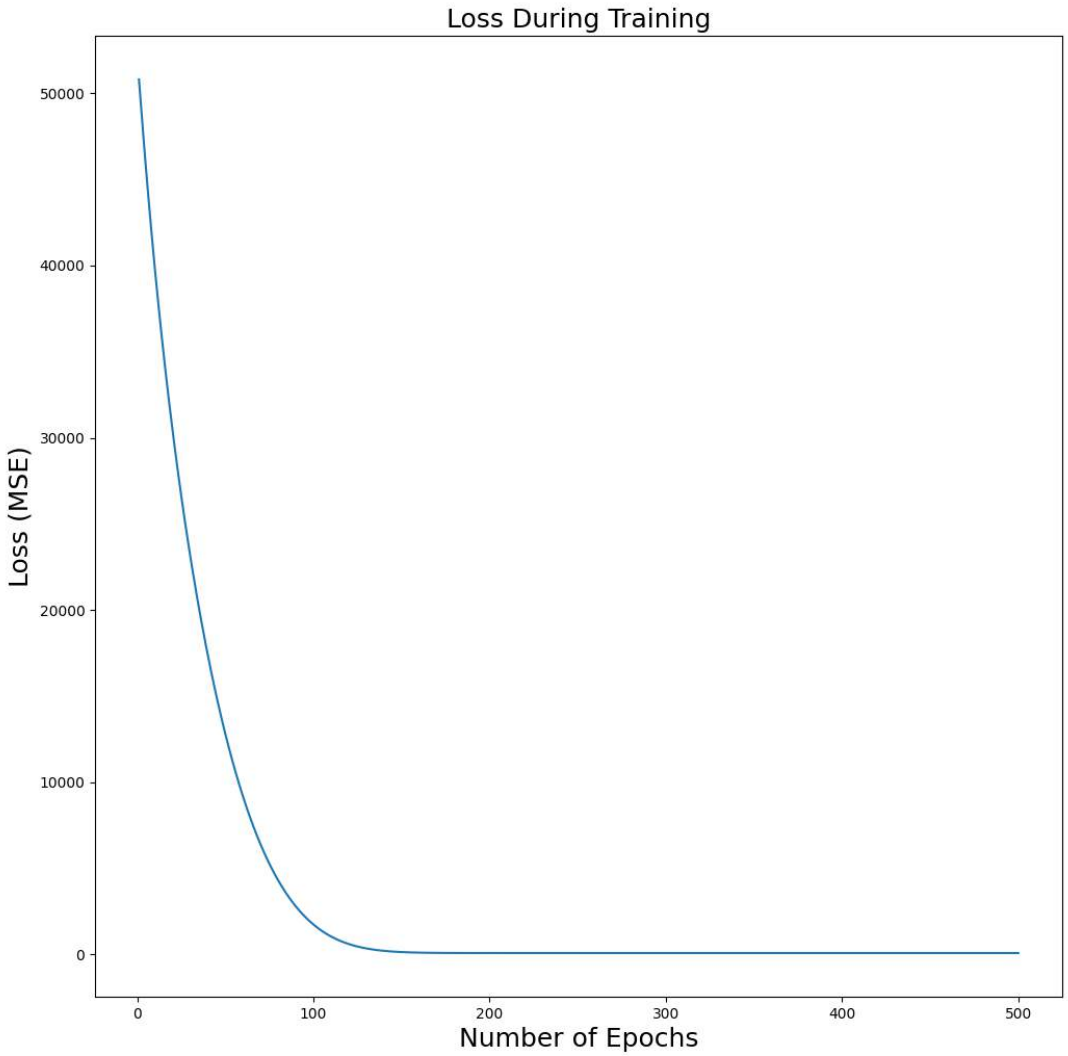

Figure 147: Loss vs Epochs for Model 3: learning rate = 1e-1, batchsize = 32, weight decay = 1e-3

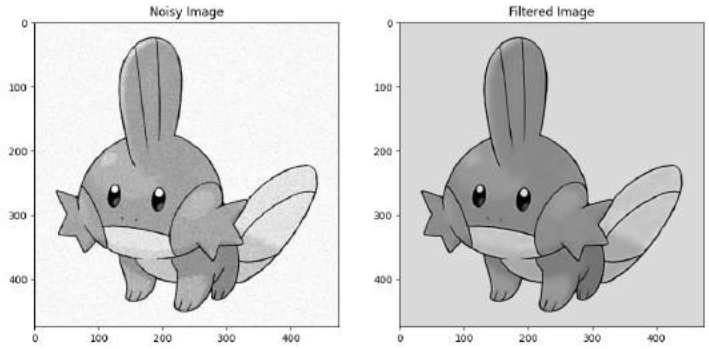

Figure 148: Testing image passed through Model 3

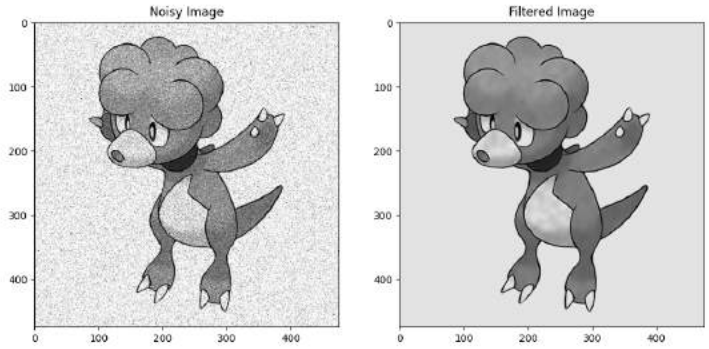

Figure 149: Testing image passed through Model 3

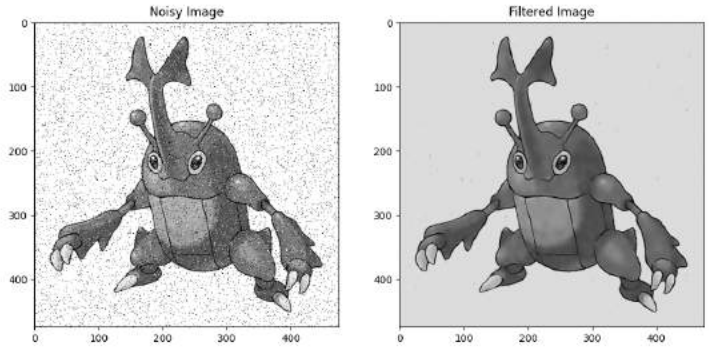

Figure 150: Testing image passed through Model 3

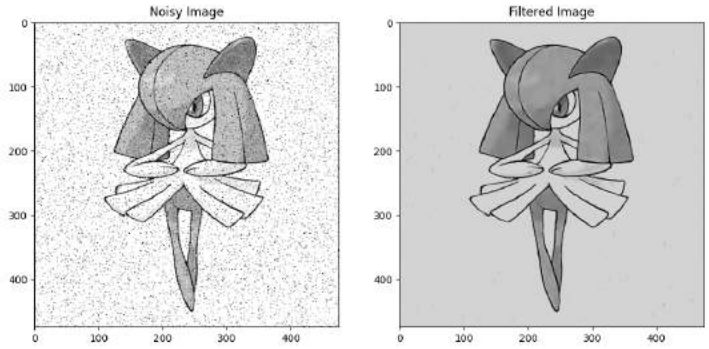

Figure 151: Testing image passed through Model 3

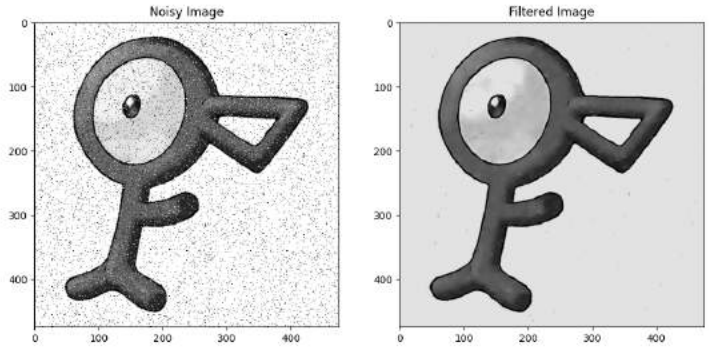

Figure 152: Testing image passed through Model 3

From the images presented above, we see that the denoising efforts of Model 3 are outstanding. The model reported an MSE value of 67 on the testing dataset. Visually, we see that the amount of noise is drastically reduced. The background on the pokemon images is almost perfectly ridden with the speckled noise seen in the noisy image. The ReLU activation function also does a really good job clearing the noise within the pokemon cartoon figure. From Figure 147 we see that the model is able to smoothly bring the loss value down from well over 50,000 to 67 before converging. With knowledge of the results for upcoming models, I can conclude that the results presented here are the best in the report, with this neural network performing orders of magnitudes above the linear and nonlinear filters presented in Parts 1 and 2. To compare the figures presented above with the "ground truth", I present a pokemon cartoon without added noise in Figure 153. From this Figure, we conclude that Model 3 does a great job of denoising but does in fact yield the incorrect background color. To see if other activation functions with similar model architectures as Model 3 can yield better results, I present Model 5 which has the same exact model architecture as Model 3 but instead of using ReLU, the TanH activation function is used in its place.

Figure 153: Ground truth Pokemon Cartoon

```
model = torch.nn.Sequential(torch.nn.Conv2d(1,2,kernel_size = (5,5),padding = 2),
      torch.nn.Tanh(),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 2----------
      torch.nn.Conv2d(2,4,kernel_size = (7,7),padding = 3),
      torch.nn.Tanh(),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 3----------
      torch.nn.Conv2d(4,4,kernel_size = (7,7),padding = 3),
      torch.nn.Tanh(),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 4----------
      torch.nn.Conv2d(4,2,kernel_size = (7,7),padding = 3),
      torch.nn.Tanh(),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 5----------
      torch.nn.Conv2d(2,1,kernel_size = (5,5),padding = 2),
      torch.nn.Tanh(),
      torch.nn.BatchNorm2d(1, affine = True)
)
```

Figure 154: Model 5

Results for Model 5 are presented below:

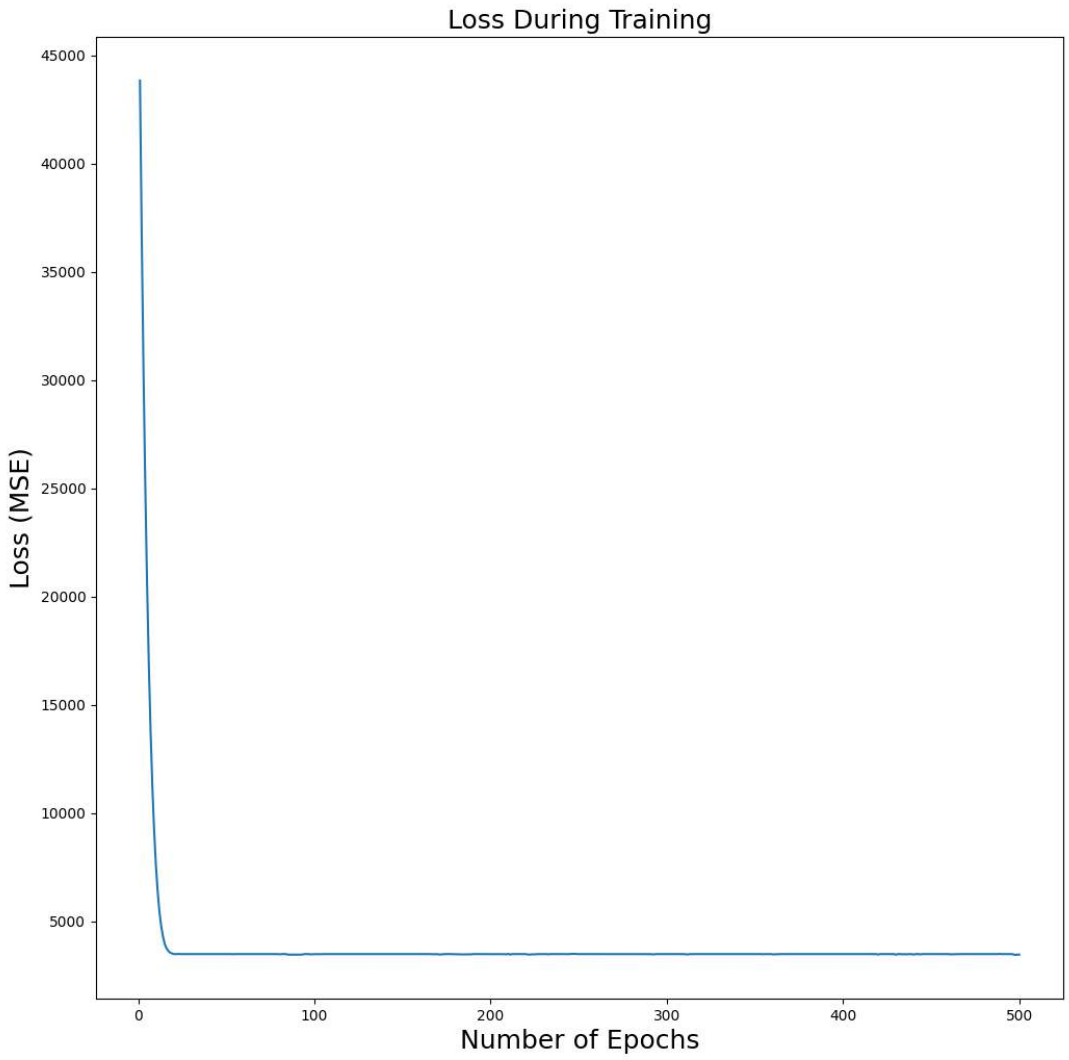

Figure 155: Loss vs Epochs for Model 5

From Figure 155, we see that the TanH activation function does a very poor job in our denoising analysis. It results in an MSE value of 3598 on our testing set and produces testing images that are entirely unrecognizable.

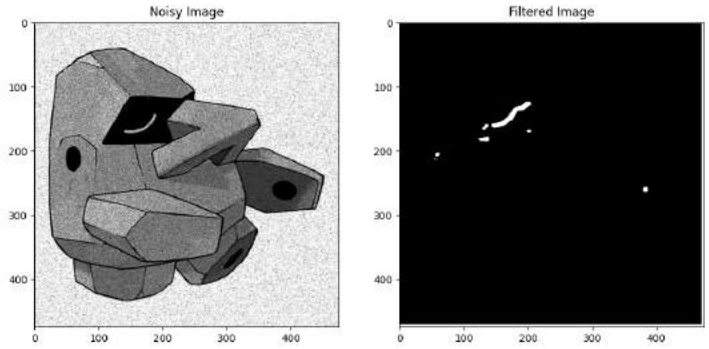

Figure 156: Testing image passed through Model 5

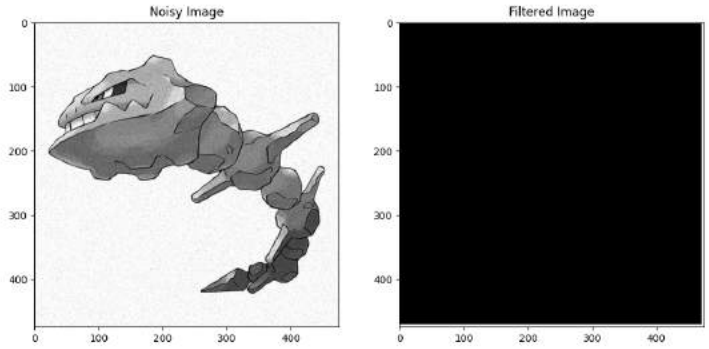

Figure 157: Testing image passed through Model 3

For the sake of saving space, the other 3 testing images passed through model 3 will not be presented, as they yielded completely black results like 157. These results are the same as the non-testing image (picture of Jerry Garcia) that was also passed in to each of the models. As a reminder this image was presented for Model 1 but was not presented afterward because each model yielded completely black results after its passing through the different models. Here we see similar results as Model 5 does a terrible job of denoising. To give us a better understanding of why, I have introduced a picture of the activation function for TanH in Figure 158.

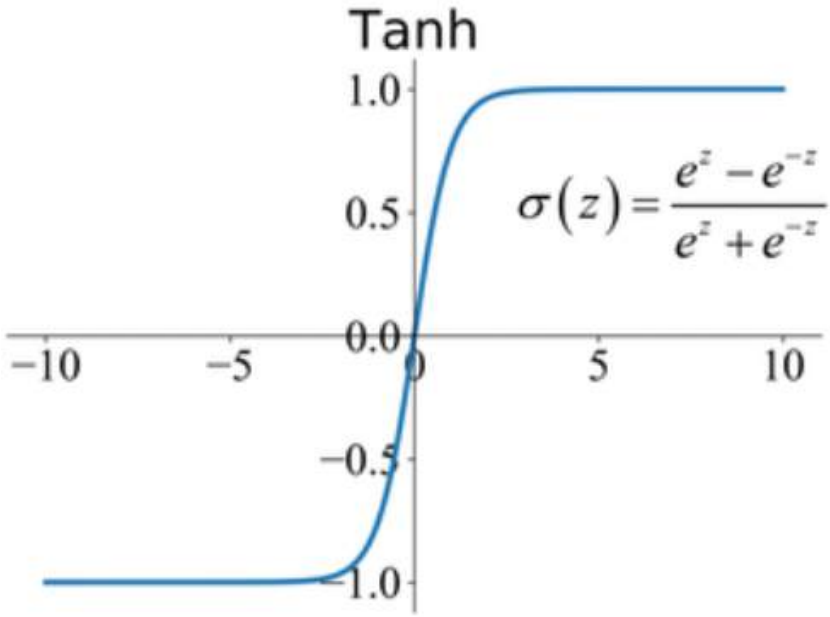

Figure 158: TanH activation function

After analyzing Figure 158, it is completely possible that we have reached saturation levels with tanh and our model has trained to either infinity or negative infinity. If I had more time with this project, it would be interesting to further delve into this idea and run a variety of experiments to see if any of the parameters could be further altered to prevent saturation. Due to the time duration it takes to run these models, this was not possible for Project 2.

One interesting thing that was observed in Figure 156 is the fact that the only portion of the noisy image that is mapped to white is the black colored pixels. This would further back the idea of complete saturation as everything is mapped to black except black mapped to white. We move on to our last model: Model 4.

In Model 4, I wanted to ask the question, is the most complex model the best model? In this model, I introduce 10 layers, with a ramping up of channel sizes, such that layers 5 and 6 have input and output channels of size 16 before ramping back down. I decided to go with a constant activation function between each layer. I chose to utilize the sigmoid activation function for its traditional use in neural networks. Initially, I had also ramped up the kernel size, starting with 5 by 5 at layer 1 and ramping up to 11 by 11, but the model took too long to run with such a large kernel size and thus had to be reduced for the sake of timely results.

```
model = torch.nn.Sequential(torch.nn.Conv2d(1,2,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 2----------
      torch.nn.Conv2d(2,4,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 3----------
      torch.nn.Conv2d(4,8,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(8, affine = True),
#layer 4----------
      torch.nn.Conv2d(8,16,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(16, affine = True),
#layer 5----------
      torch.nn.Conv2d(16,16,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(16, affine = True),
#layer 6----------
      torch.nn.Conv2d(16,16,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(16, affine = True),
#layer 7----------
      torch.nn.Conv2d(16,8,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(8, affine = True),
#layer 8----------
      torch.nn.Conv2d(8,4,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(4, affine = True),
#layer 9----------
      torch.nn.Conv2d(4,2,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(2, affine = True),
#layer 10----------
      torch.nn.Conv2d(2,1,kernel_size = (5,5),padding = 2),
      torch.nn.Sigmoid(),
      torch.nn.BatchNorm2d(1, affine = True)
)
```

Figure 159: Model 4

Results of Model 4 were not what I expected at all. I expected that a model with 10 layers and nonlinear activation functions would train the best, but I was completely misunderstood. The MSE value on the training set was found to be 3616. Resulting images are presented below:

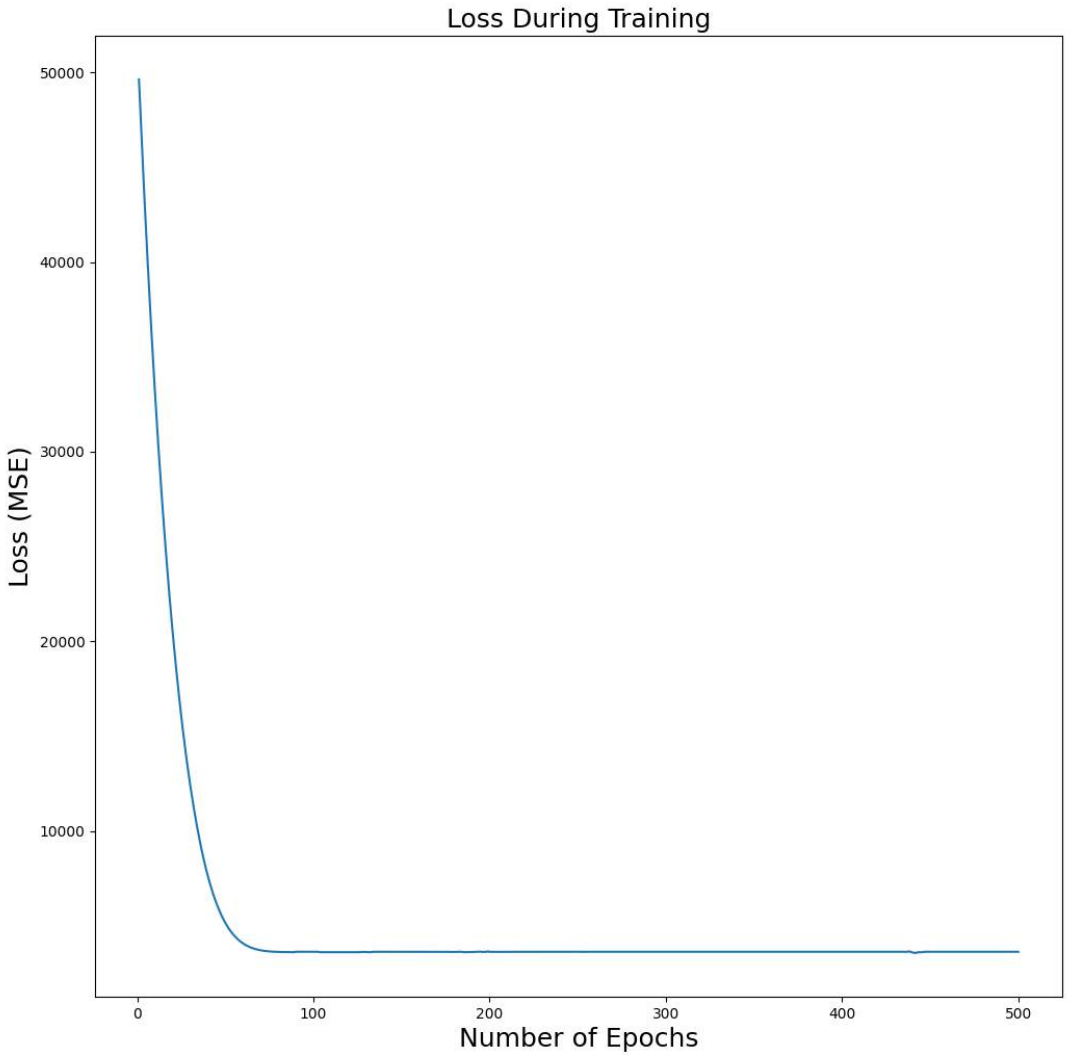

Figure 160: Loss vs Epochs graph for Model 4

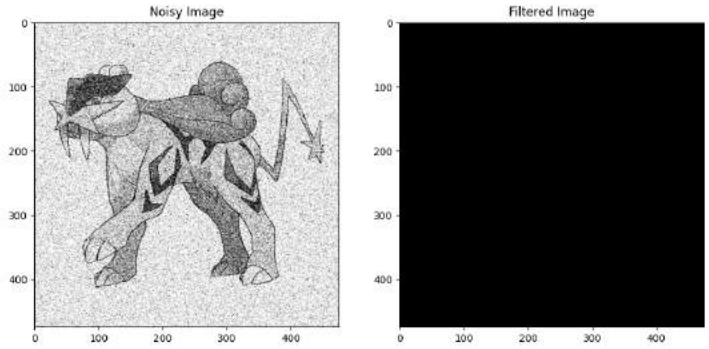

Figure 160: Testing image result from Model 4

Again, we see similar results to that of Model 5, testing images are completely black. This again aligns with passing non-testing/training images through the model, as would be seen passing the Jerry picture through this model. Since the model converges to an MSE value, I conclude that the model is saturated and has produced undesirable results. The ReLU activation function, presented in Figure 161, that produced such great results does not have the same type of saturation limits at infinity and negative infinity seen in both the sigmoid and TanH activation functions. The fact that this model produces very poor results also answers my original question. More complex does not necessarily mean better. This highlights the fact that some neural network architectures are better suited for certain tasks than others. This includes the architecture parameters but also the activation functions. For example, some activation functions might be suited well for classification experiments but performed poorly in denoising. But Model 3 highlights the fact that when the optimal architecture is found and is properly implemented, neural networks can work very efficiently and can be trained in such a way that they can produce very desirable results that classical methods (such as seen in Parts 1 and 2) could not yield.

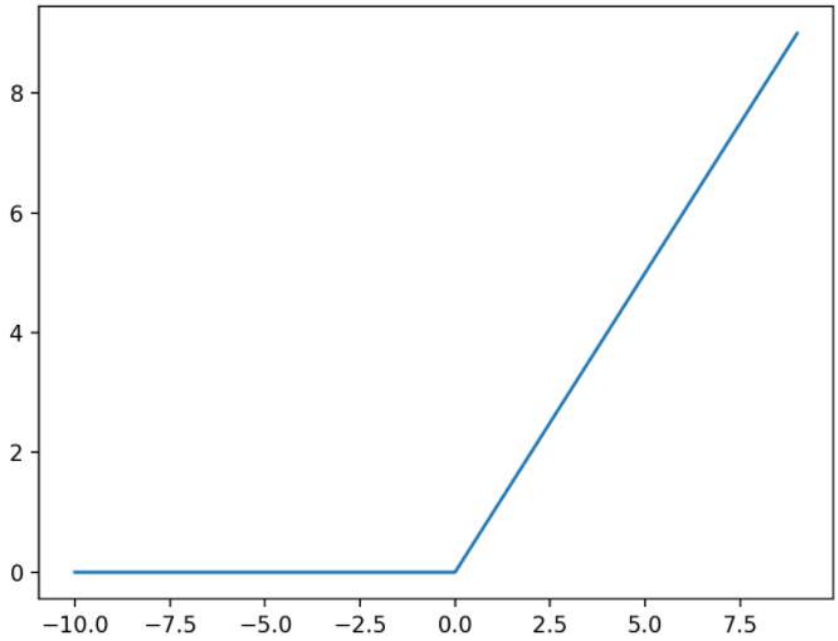

Figure 161: ReLU activation function

Model	Model Architecture	MSE value (testing dataset)	Comments on Resulting Testing Images	Comments on non-testing image tested
Model 1	1 Layer Convolutional	429	Minimal Denoising	Minimal Denoising
Model 2	5 Layer Convolutional	478	Minimal Denoising	Terrible - completely black image produced
Model 3	5 Layers, Convolution and ReLU activation function	67	Outstanding Denoising but model not able to deduce correct background color	Terrible - completely black image produced
Model 4	10 Layers, Convolution and Sigmoid activation function	3616	Terrible denoising, black images produced	Terrible - completely black image produced
Model 5	5 layers, Convolution and TanH activation function	3598	Terrible denoising, black images produced	Terrible - completely black image produced

Table 2: Results from individual neural network models

Other notes:
It must be noted that the number of epochs were changed throughout testing. Initially the number of epochs was set to 200, but it was noticed that some models were not finished converging around 200. Although learning rates were adjusted to ensure faster convergence, the number of epochs was set to 500 to ensure convergence criteria was met before analysis on the resulting models MSE values on the training dataset. It must also be noted that even though the SGD optimizer was recommended, at the hint of the TA, I proceeded to utilize the Adam optimizer from pytorch. The SGD algorithm utilizes stochastic gradient descent whereas the Adam optimizer algorithm utilizes an implementation of the L2 penalty that can be useful when fine tuning a pre-trained network. This is because layers that would otherwise be inutilizable can be added in the training process by the optimizer. It was found that the Adam optimizer helped lower MSE values in comparison to the SGD algorithm.

Conclusion - Parts ¾

A variety of results were presented for a variety of model architectures. In comparing the results with the filters from Parts 1 and 2, I conclude that some models fared better at denoising the images and some, such as Model 5, actually did worse than the filters of parts 1 and 2. This isn't completely unexpected. When training a model to carry out a task for you, you can either train it well and it can be very powerful if trained on enough training data, or it can spiral out of control if the model reaches saturation or if it is trained on small sets of data or with incorrect parameters that don't optimize the activation functions in question. Various architectures work well for some training tasks and some architectures do not perform well for various tasks. In my experiments with Models 1 through 5, it is evident that that ReLU activation function does a very good job at denoising tasks, whereas one would not want to utilize sigmoid or TanH for such a task. These activation functions may fare better in other tasks, such as classification, but for the purpose of training a model for denoising, these activation functions produce undesirable results. In the end, the neural networks have the ability to perform exponentially better than the filters presented in Parts 1 and 2, but only if correct architectures and parameters are tuned for the task at hand.

Credit:

I could not have been able to do the following project without the work of a few people and many online forums where countless people have contributed their own work. I want to thank Dr. Ross Whitaker and Jadie Adams for helping guide me on a few problems. I also want to thank Tushar Kataria for helping me navigate the python virtual environment on the CADE machines.

Parts ½:

- Dr. Ross Whitaker
- Julie Adams
- Tushar Kataria
- https://www.codegrepper.com/code-examples/python/add+gaussian+noise+python
- https://stackoverflow.com/questions/22937589/how-to-add-noise-gaussian-salt-and-pepper-etc-to-image-in-python-with-opencv
- https://theailearner.com/2019/05/07/add-different-noise-to-an-image/
- https://scikit-image.org/docs/dev/api/skimage.util.html#skimage.util.random_noise
- https://scikit-image.org/docs/dev/auto_examples/transform/plot_ssim.html
- https://scikit-image.org/docs/dev/api/skimage.metrics.html#skimage.metrics.structural_similarity
- http://vision.psych.umn.edu/users/kersten/kersten-lab/courses/Psy5036W2017/Lectures/17_PythonForVision/Demos/html/2a.Convolution.html
- https://scikit-image.org/docs/dev/auto_examples/filters/plot_denoise.html
- https://scikit-image.org/docs/0.12.x/auto_examples/filters/plot_denoise.html
- https://scikit-image.org/docs/dev/api/skimage.restoration.html

- https://numpy.org/doc/stable/reference/generated/numpy.dot.html
- https://scikit-image.org/docs/stable/api/skimage.filters.html#skimage.filters.gaussian
- https://stackoverflow.com/questions/17153779/how-can-i-print-variable-and-string-on-same-line-in-python
- https://realpython.com/python-square-root-function/
- https://scikit-image.org/docs/dev/auto_examples/filters/plot_nonlocal_means.html#sphx-glr-auto-examples-filters-plot-nonlocal-means-py

Parts ¾:

- Dr. Ross Whitaker
- Julie Adams
- Tushar Kataria
- https://pytorch.org/docs/stable/nn.html#linear-layers
- https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity
- https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html
- Adam — PyTorch 1.9.1 documentation
- A Gentle Introduction to the Rectified Linear Unit (ReLU) (machinelearningmastery.com)
- torch.nn — PyTorch 1.9.1 documentation
- What are Neural Networks? | IBM