ECE 751 PROJECT (Code Files Attached)

Problem 1

As is well known, the redundancy of the Continuous Wavelet Transform (CWT) may be reduced by way of a wavelet frame representation of a function/signal. This is further improved by way of orthonormal wavelet bases. The vanishing moment property of a wavelet yields a condensed/compressed representation.

A non-linear filtering procedure (such as denoising) is typically constructed by means of a thresholding strategy on the wavelet representation coefficients aiming to eliminate undesired additive white Gaussian noise.

To conduct a comparative analysis of two non-linear filtering- denoising approaches, one based on multiscale data analysis and the other using a Neural-network-based auto-encoder. You will be asked to implement a standard CNN auto-encoder of varying complexity (number of channels and layers) trained with images signals and obtain a Discrete Wavelet Transform of data.

A systematic learning/representation of data by a Convolutional auto-encoder yields the so-called "bottleneck" compressed representation.

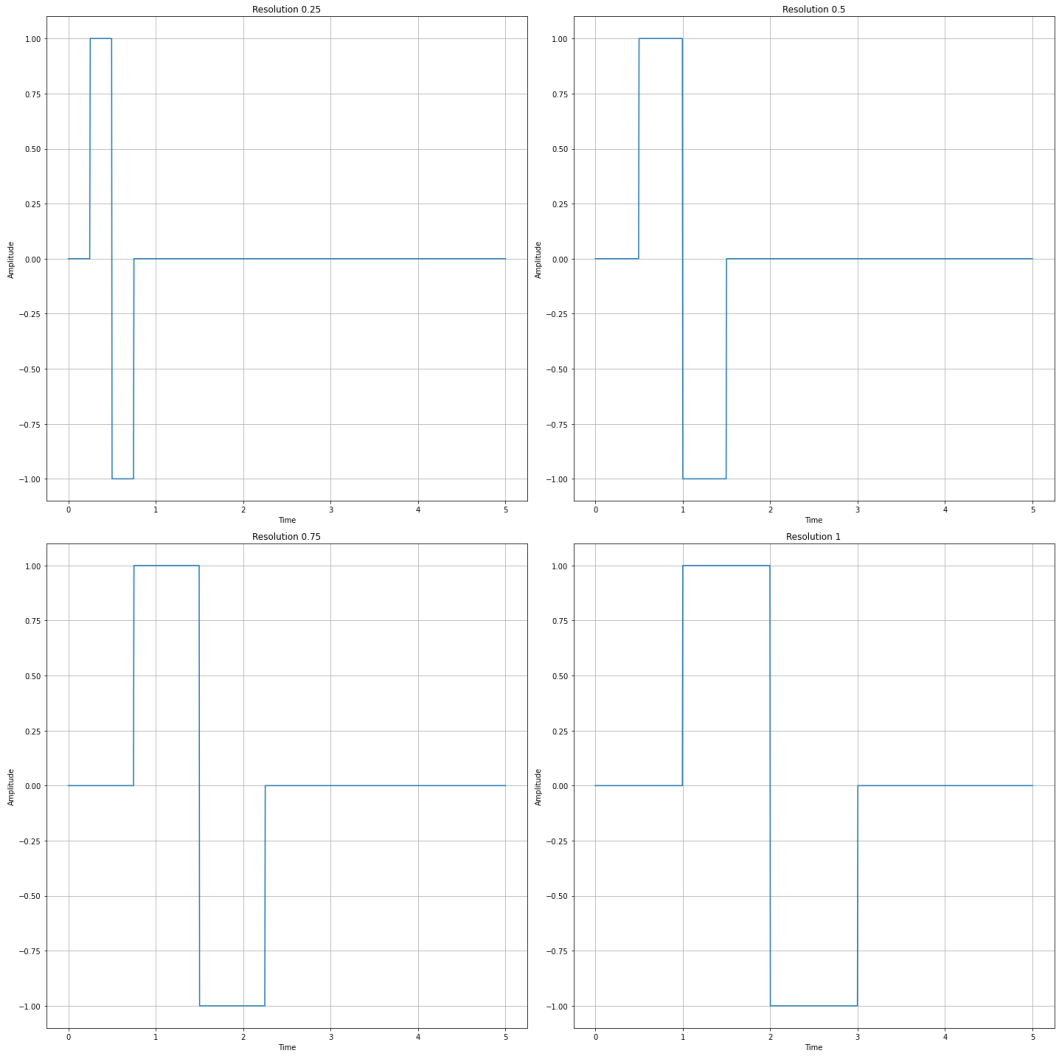
The first part of the project consists of using MATLAB and an attached Python code for auto-encoder. You will subsequently be experimenting with denoising data (time-series and images) and be comparing the two approaches fundamentally rooted in noise coefficients contribution much less compressed) and in signal coefficients much more compressed when representing signal information.

As a result, a thresholding technique, as used in the orthonormal representation of data, eliminates the "Only Noise" contributions and preserves the "Signal + Noise.". Our goal is to compare both non-linear filtering techniques.

Generate/simulate a Haar Wavelet function and plot it along with its Fourier Transform, and do so at 4 consecutive resolutions.

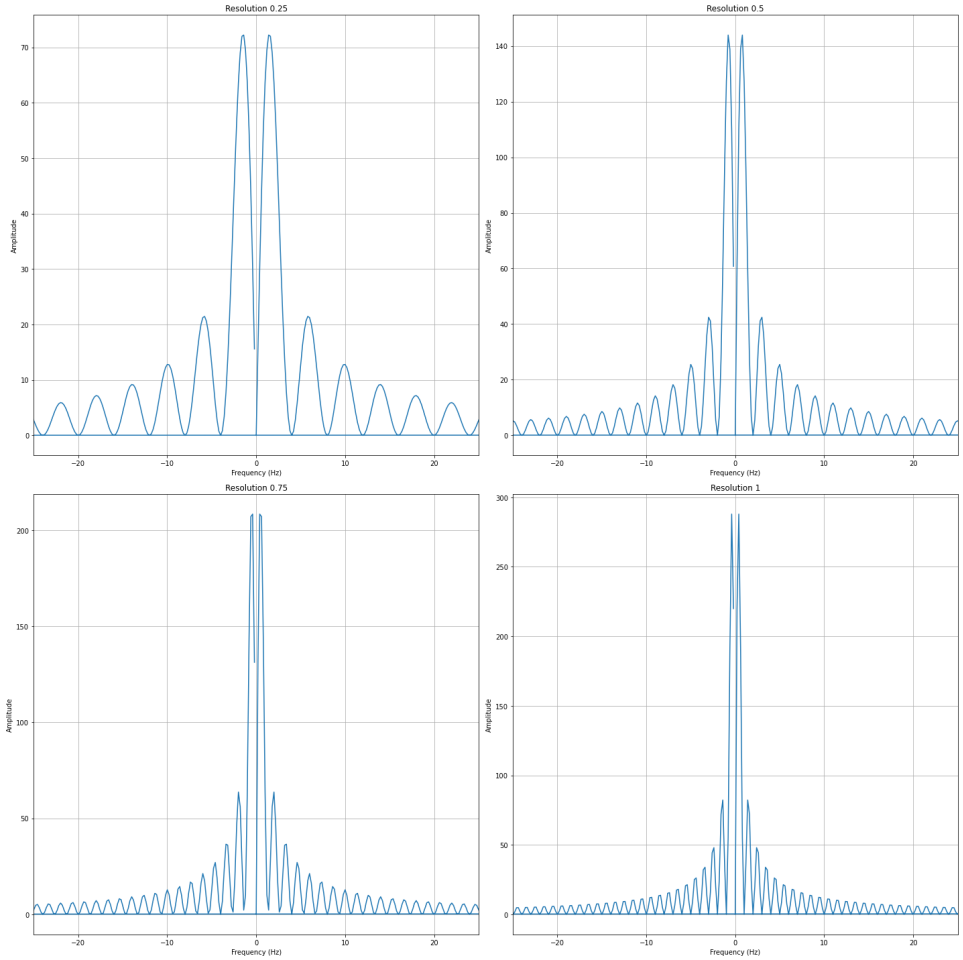
To Generate and simulate this function I will use Python.

Here is the Haar Wavelet Function and its code.


Resolution 0.25
Resolution 0.5
Resolution 0.75
Resolution 1
Amplitude
Time
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5

```
def haar(time, resolution):
    if resolution < time < 2 * resolution:
        return 1
    elif 2 * resolution < time < 3 * resolution:
        return -1
    else:
        return 0

t = np.linspace(0, 5, 1000)

resolutions = [0.25, 0.5, 0.75, 1]


wavelets = []

for resolution in resolutions:
    coefficients = []
    for time in t:
        coefficient = haar(time, resolution)
        coefficients.append(coefficient)
    wavelets.append(np.array(coefficients))


plt.figure(figsize = (20, 20))

for i in range(len(wavelets)):
    wavelet = wavelets[i]
    plt.subplot(2, 2, i + 1)
    plt.plot(t, wavelet)
    plt.title("Resolution " + str(resolutions[i]))
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    plt.grid(True)



plt.tight_layout()
plt.show()
```

And here is the Fourier Transform of the Haar Wavelet Function and its code.


Resolution 0.25
Amplitude
Frequency (Hz)
Resolution 0.5
Amplitude
Frequency (Hz)
Resolution 0.75
Amplitude
Frequency (Hz)
Resolution 1
Amplitude
Frequency (Hz)

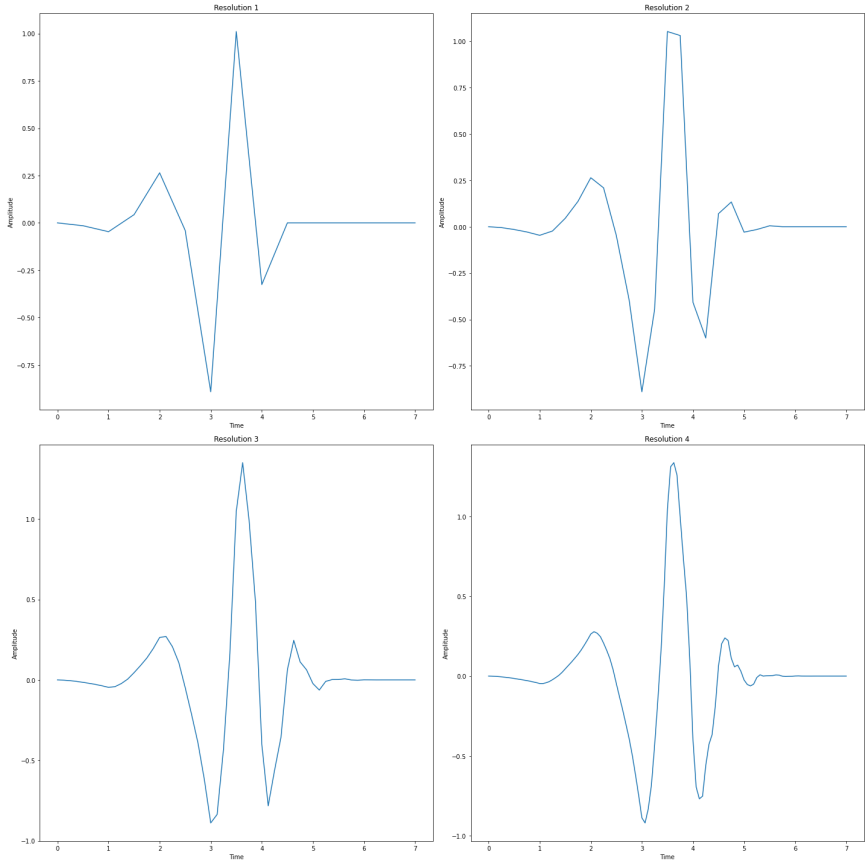
```
freqdomain = fftfreq(len(t), d=(t[1] - t[0]))

fft_wavelets = []
for wavelet in wavelets:
    fft_result = np.fft.fft(wavelet)
    abs_fft_result = np.abs(fft_result)
    fft_wavelets.append(abs_fft_result)

plt.figure(figsize = (20,20))

for i, fft_wavelet in enumerate(fft_wavelets):
    plt.subplot(2, 2, i + 1)
    plt.plot(freqdomain, fft_wavelet)
    plt.title("Resolution " + str(resolutions[i]))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel("Amplitude")
    plt.xlim(-25, 25)
    plt.grid(True)

plt.tight_layout()
plt.show()
```

I plotted the Haar Wavelet Function and its Fourier Transform at 4 different resolutions/wave periods here and the results are above.

Generate/simulate again 4 resolutions (as above) of a Daubechies-4 wavelet and plot them along with their Fourier Transforms.

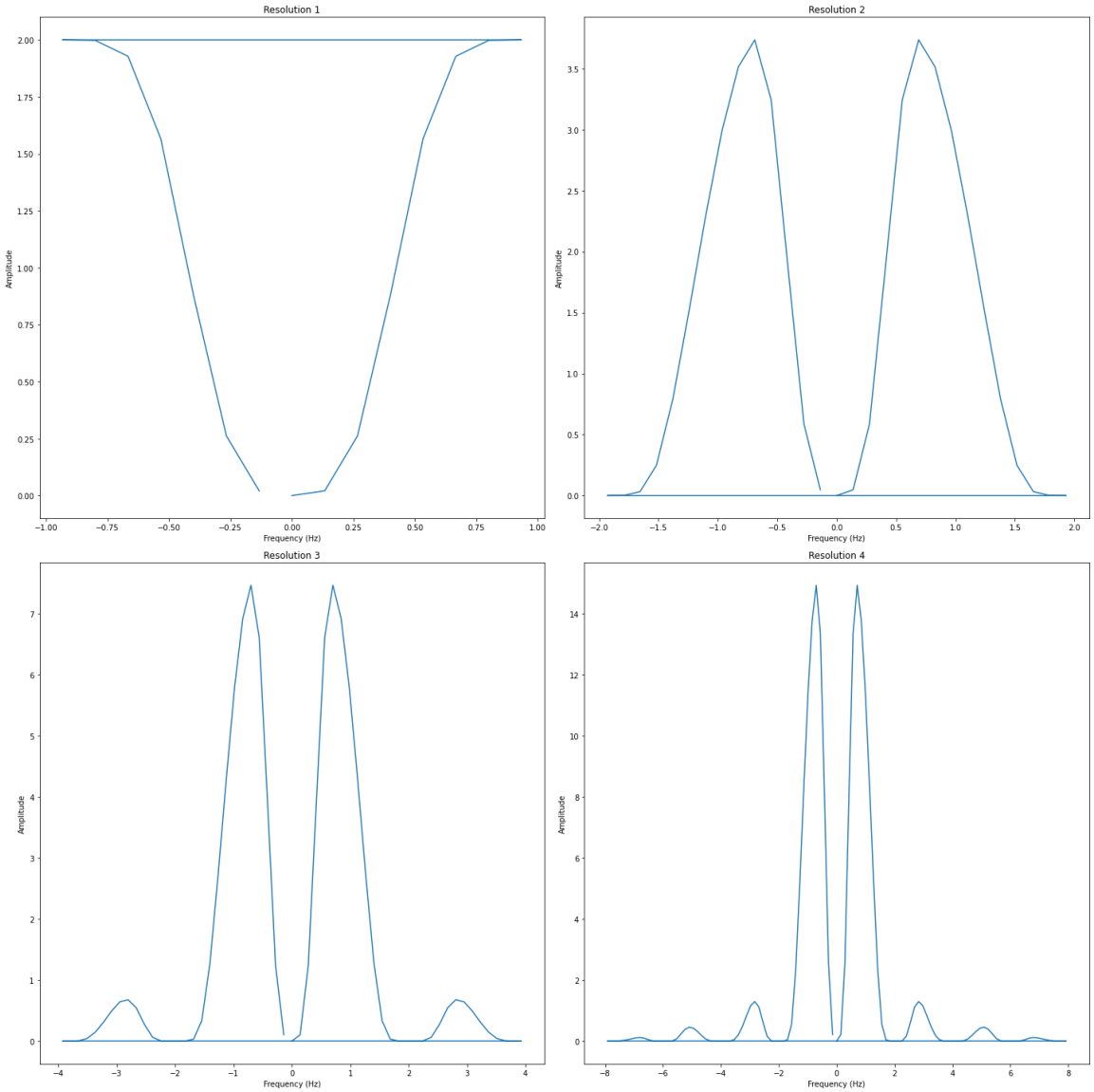
Here is the Daubechies-4 wavelet and its code.

```
import pywt

daubechies4w = pywt.Wavelet('db4')
resolutions = [1, 2, 3, 4]

plt.figure(figsize = (20, 20))

for i, resolution in enumerate(resolutions):
    #\phi(t) = \sqrt{2} \sum_{k=0}^{N-1} h_k \phi(2t-k) (Scaling Function)
    #\psi(t) = \sqrt{2} \sum_{k=0}^{N-1} g_k \psi(2t-k) (Wavelet Function)
    plt.subplot(2, 2, i + 1)
    phi, psi, x = daubechies4w.wavefun(level=resolution)
    plt.plot(x, psi)
    plt.title("Resolution " + str(resolutions[i]))
    plt.xlabel("Time")
    plt.ylabel("Amplitude")

plt.tight_layout()
plt.show()
```



And here is the Fourier Transform of the Daubechies-4 wavelet and its code.

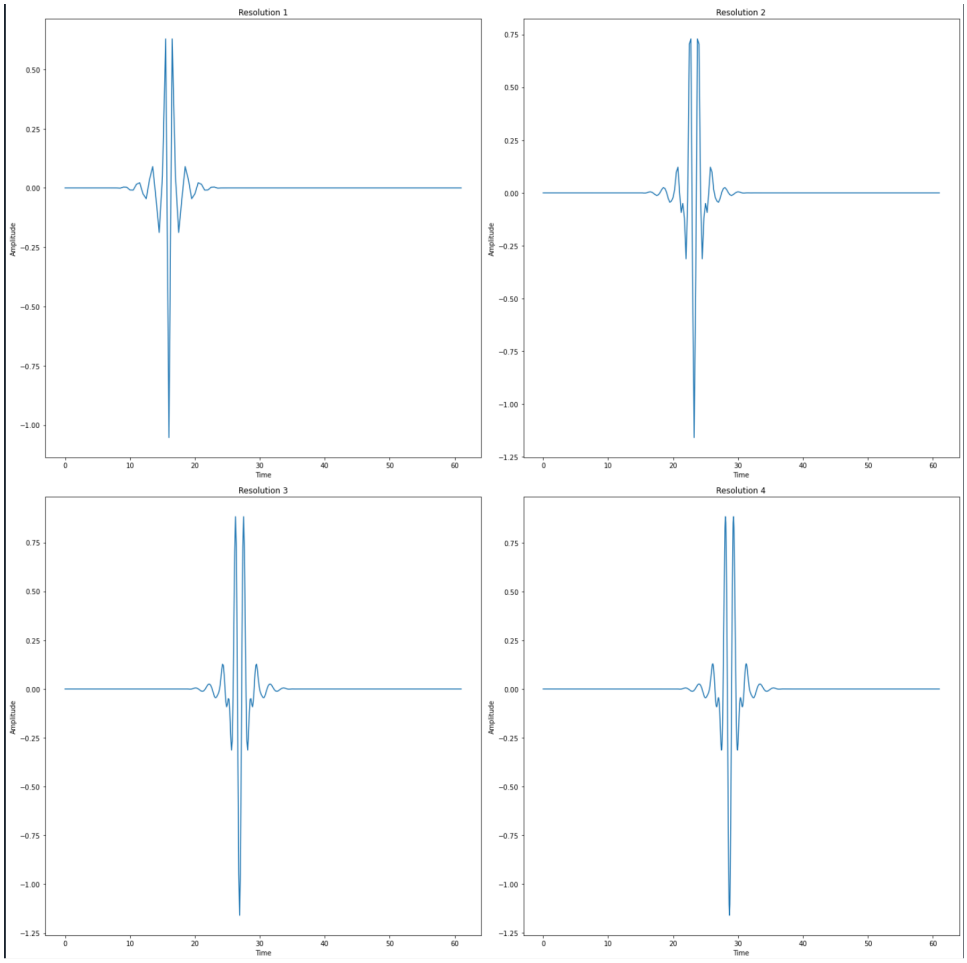
```
plt.figure(figsize = (20, 20))

for i, resolution in enumerate(resolutions):
    plt.subplot(2, 2, i + 1)
    phi, psi, x = daubechies4w.wavefun(level=resolution)
    plt.title("Resolution " + str(resolutions[i]))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel("Amplitude")
    freqdomain = fftfreq(psi.size, d=(x[1] - x[0]))
    fft_psi = np.fft.fft(psi)
    plt.plot(freqdomain, np.abs(fft_psi))


plt.tight_layout()
plt.show()
```



Generate/simulate in the same way 4 resolutions of the Meyer wavelet with their plot as well as their Fourier Transforms.

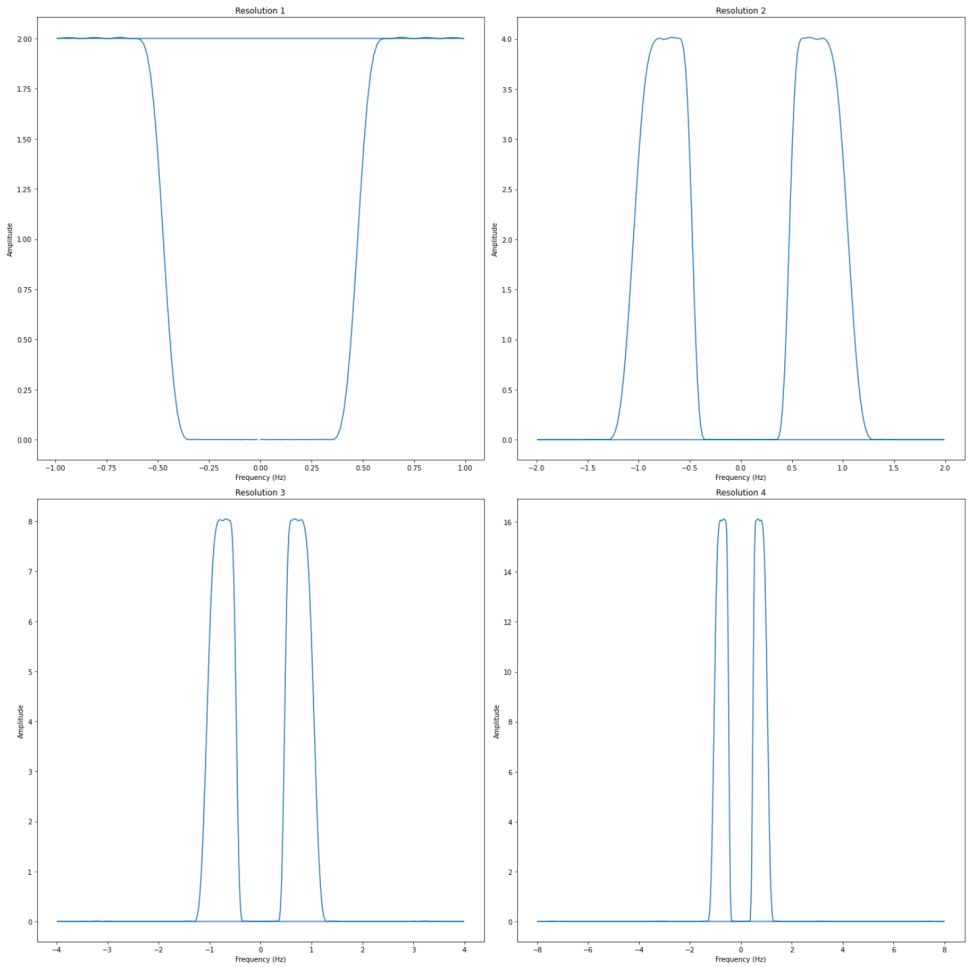
Here is the Meyer wavelet and its code.

```
meyer = pywt.Wavelet('dmey')
resolutions = [1, 2, 3, 4]

plt.figure(figsize = (20, 20))

for i, resolution in enumerate(resolutions):
    plt.subplot(2, 2, i + 1)
    phi, psi, x = meyer.wavefun(level=resolution)
    plt.plot(x, psi)
    plt.title("Resolution " + str(resolutions[i]))
    plt.xlabel("Time")
    plt.ylabel("Amplitude")

plt.tight_layout()
plt.show()
```



And here is the Fourier Transform of the Meyer wavelet and its code.

```
plt.figure(figsize = (20, 20))

for i, resolution in enumerate(resolutions):
    plt.subplot(2, 2, i + 1)
    phi, psi, x = meyer.wavefun(level=resolution)
    plt.title("Resolution " + str(resolutions[i]))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel("Amplitude")
    freqdomain = fftfreq(psi.size, d=(x[1] - x[0]))
    fft_psi = np.fft.fft(psi)
    plt.plot(freqdomain, np.abs(fft_psi))


plt.tight_layout()
plt.show()
```



What conclusion can you draw from the three wavelet spectra? Can you relate that to the function's temporal behavior/structure?

The Haar Wave wavelet has very abrupt transitions, which indicates its sharp nature in the time domain. This makes it suitable for applications such as edge detections in images. Its temporal behavior is akin to that of a step function, being a binary structure with sudden changes. In relation to the spectra, this causes the Haar Wavelet to have a wide and frequency spectrum since in Fourier transforms, sharp edges indicate components of higher frequency. It is good for detecting sudden changes and discontinuities.

The Daubechies-4 limit has more complexity and smoother transitions in contrast to the Haar Wavelet. It has a more confined frequency spectrum than Haar, and it is less abrupt. It allows for smoother waveforms, leading to a narrower frequency spectrum allowing to display more detailed information in a signal which make it more suitable for precise tasks, and signal denoising. It is essentially in the middle between Haar and Meyer, allowing for a good balance between time and frequency localization.

The Meyer Wavelet has a lot of complexity and has a very focused frequency spectrum, nearly smooth in the time domain. It has extremely confined frequency spectrum and it is most effective on processing problems where there is virtually no room for error. Meyer wavelets are continuously differentiable changing gradually over time, resulting in a very concentrated frequency spectrum, and suitable for very precise tasks like financial data analysis and ECG Biomedical Signal Analysis.

Take a CNN architecture for an auto-encoder (i.e. both encoder and decoder) of varying complexity. (see towardsdatascience.com/autoencoders-and-the-denoising-feature-from-theory-to-practice). Use

attached python code for auto-encoder. You will need to have NumPy, Matplotlib and TensorFlow packages installed to run this code. Run the code to train a convolutional autoencoder over clean MNIST dataset (You don't need to download the dataset, the code will do this automatically), The encoder part of your autoencoder will take clean images as input and output a compressed encoded vector. The decoder will take this encoding as input and again aim to output the original clean image. Visualize the vector output (which is the compressed encoding) of the bottleneck layer for 5 samples (sampled from the test set) each corresponding to two digit classes of your choice (say 7 and 8).

This part of the project implements a convolutional autoencoder utilizing the MNIST dataset with both the encoder and decoder component. The encoder compresses the MNIST images while the decoder tries to reconstruct the images from the encoder. We want to visualize the values '7' '8' on the dataset after being passed through the convolutional autoencoder.

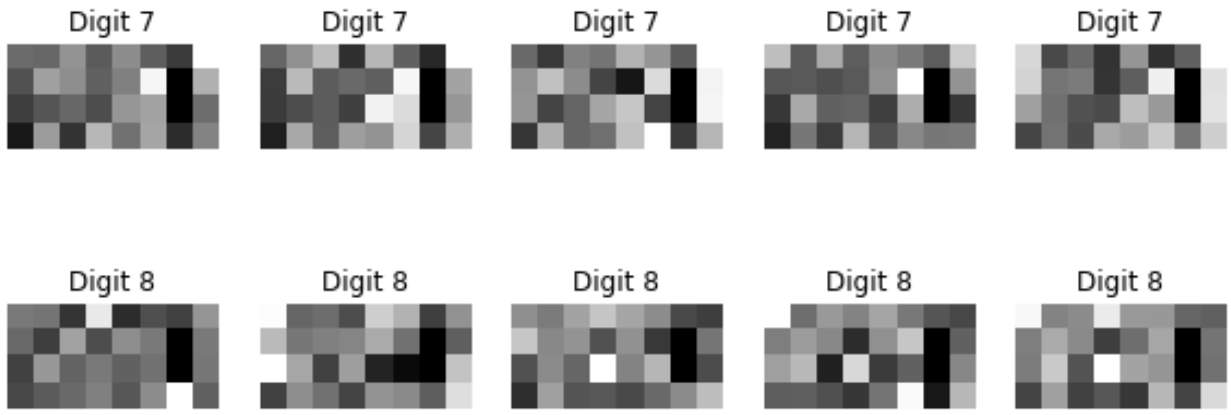
The code resizes the images to 32×32 and the images are normalized. The encoder section has 2 convolutional layer as well as a max pooling layer. The decoder has transpose convolutional layers. The model is trained on this network using the Adam optimizer and binary cross entropy as its loss function. Over several epochs, we want to minimize the loss function teaching the encoder and decoder how to accurately compress and reconstruct the image, respectively.

Then the encoded representation of images of '7' and '8' are visualized and compared, converting the test set to the encoded vectors and the images are visualized as well to evaluate the effectiveness of the network by seeing if the images in the MNIST dataset are easily identifiable.

Here is the code to visualize the encoded representation/plot the encoded digits:

```python
def visualize_encoding(encodings, labels, digits, num_samples):
    sample_indices = {}
    for digit in digits:
        indices = np.flatnonzero(labels == digit)
        sampled_indices = indices[:num_samples]
        sample_indices[digit] = sampled_indices

    fig, axes = plt.subplots(nrows=len(digits), ncols=num_samples, figsize=(num_samples * 2, len(digits) * 2))
    for row, digit in enumerate(digits):
        for col in range(num_samples):
            if len(digits) > 1:
                ax = axes[row][col]
            else:
                ax = axes[col]
            encoding = encodings[sample_indices[digit][col]]
            ax.imshow(encoding.reshape(4, 8), cmap='gray')
            ax.set_title(f"Digit {digit}")
            ax.axis('off')
    plt.show()


autoencoder = load_model('autoencoder_model.h5')

bottleneck_layer = autoencoder.get_layer('bottleneck_layer')
bottleneck_encodings = bottleneck_layer.predict(test_data)

visualize_encoding(bottleneck_encodings, test_label, digits=[7, 8], num_samples=5)
```

And here are the results that were obtained:



(These are the encoded representations of the digits that I chose, digits 7 and 8. These patterns are used to reconstruct the images of the MNIST dataset from the decoder. This represents the feature space of the autoencoder and represents the most important aspects and data necessary for reconstruction.)

Here were our observations:

(The top row is the original test inputs and the bottom row is the outputs after compression and decompression in the autoencoder. It looks like the numbers are generated and the important features of these numbers were able to be extracted, with loss of some data and increase of abstraction. It's a trade off between saving space and preserving detail).

We get a Mean Square error of 0.003925444092601538 which indicates effective performance of the model, suggesting very high image reconstruction quality.

There are a few parameters that we can fine tune such as the number of layers, the kernel size, the padding, the activation function, or the size of the bottleneck layer (which allows the testing of the decoder to reconstruct the most important part of the image. We can improve the quality of the image by increasing the bottleneck layer or save time by decreasing the layer size with tradeoff of image quality) or even the number of epochs (which might be able to extract the important features of each number in MNIST with the risk of overfitting). Changing these parameters can help explore model behaviors and fine-tune the model for optimal efficiency and performance.

Now add I.I.D. samples of gaussian noise with 0 mean and variance 0.75 to each of the pixels of clean digit images from the test set and again generate bottleneck layer outputs for the same digits as used in

previous part (again 5 samples each for the same 2 digits). Compare these outputs with previous results. Explain your observations.

To add the sample of the Gaussian Noise, I uncommented the lines regarding the noisy data and added a noisy bottleneck encoding to test. Here is the code:

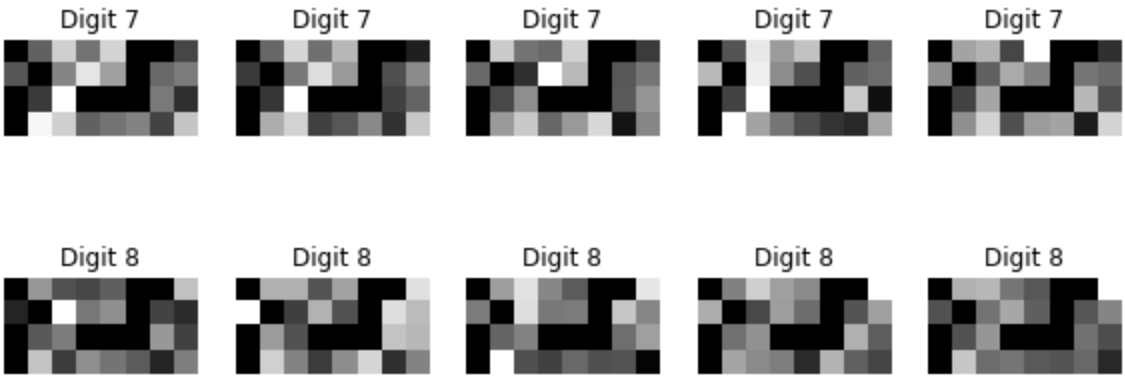
```
# Create a copy of the data with added noise
noisy_train_data = noise(train_data)
noisy_test_data = noise(test_data)

# Display the train data and a version of it with added noise
display(train_data, noisy_train_data,'noisy_vs_orig')
display(train_data, train_data,'noisy_vs_orig')
##################################################################
```

```
'''
Bottleneck layer output
'''
bottleneck = Model(input,bottleneck_layer_output)
bottlenecck_encoding = bottleneck.predict(test_data)
noisy_bottleneck_encoding = bottleneck.predict(noisy_test_data)
```

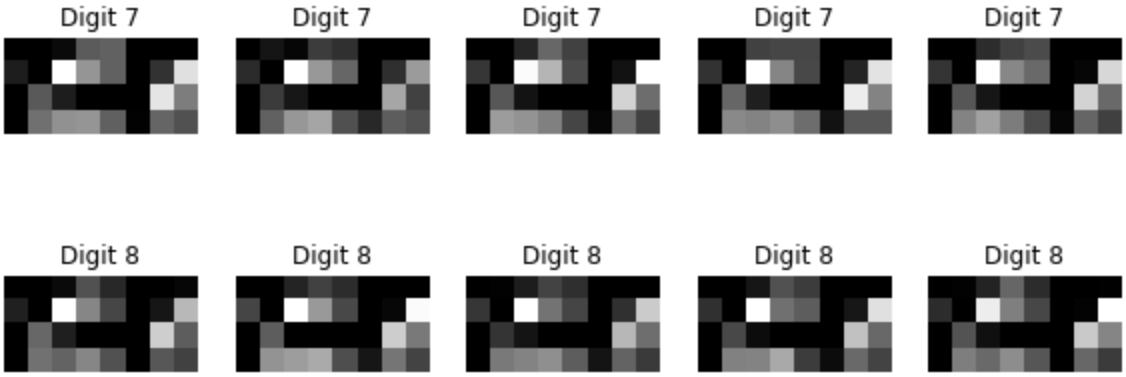
```
print("Visualizing Clean Encodings:")
visualize_encoding(bottlenecck_encoding, test_label, [7, 8], 5)
print("Visualizing Noisy Encodings:")
visualize_encoding(noisy_bottleneck_encoding, test_label, [7, 8], 5)
```

And here are the results:

Without Noise:



With Noise:



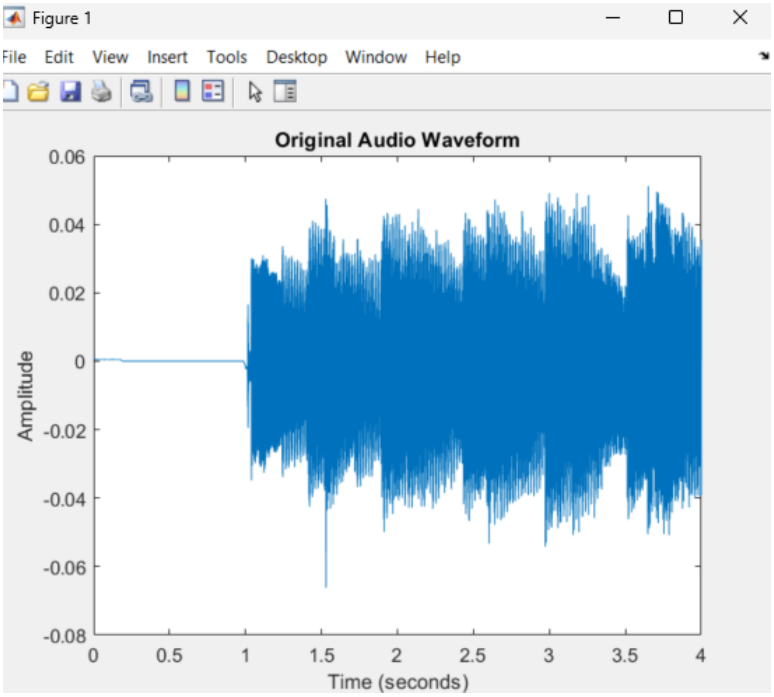
From this we can extrapolate that the noisy encodings on picture 1 and 2 look pretty similar but the ones on picture 1 show more clarity than picture 2. Without the noise, the encodings for digit 7 and digit 8 seem to be very consistent, with a greater variety at play with noise. The noisy encodings are not able to preserve as many essential encoding data as the noiseless one, which would probably not be able to regenerate the numbers as effectively. It seems that the autoencoder is relatively robust to noise due to the similarity of the encodings. If we make the bottleneck smaller we might force the autoencoder neural network to learn more robust features to represent the data successfully.

Problem 2

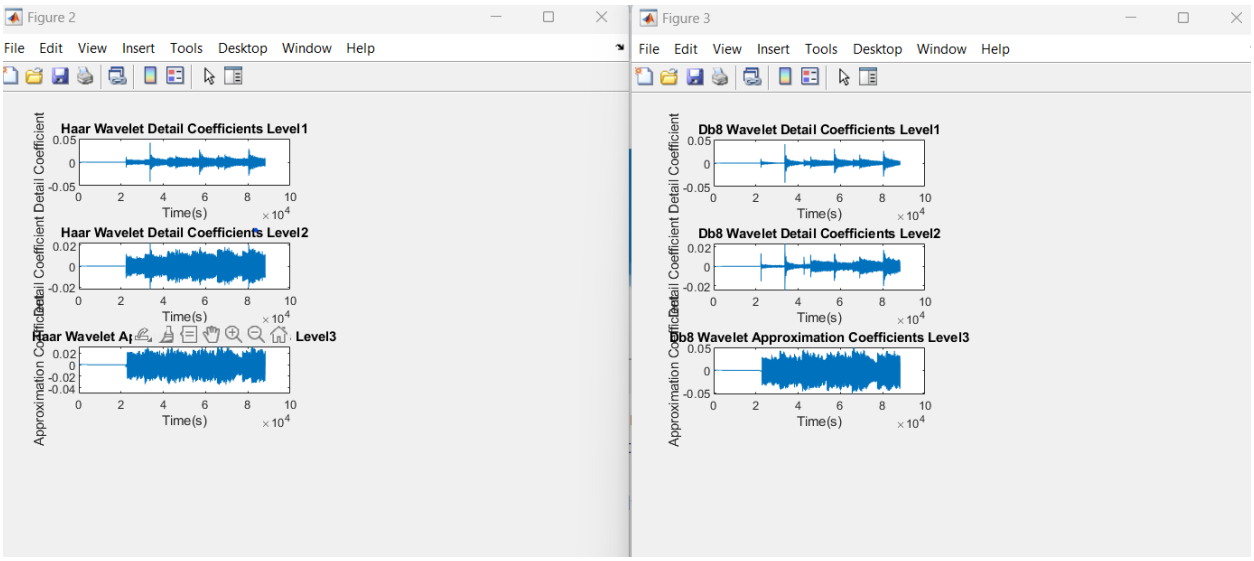
Consider the waveform extracted from the attached 'sample.wav' file. You can use 'audioread' command in Matlab to read the signal from the file.

Perform a wavelet decomposition with a Haar wavelet and then with a Daubechies-8 wavelet.

In this code I posted below, I first read and listened to the "sample.wav" audio data (bell mixed with classic old movie music) performed the wavelet decomposition for the wavelet coefficients, then plotted the original waveform using the 'plot' function. Here are the results.



After which, I performed wavelet decomposition for both Haar and db8 wavelets, and set the decomposition level to 3, plotting out the detail and approximation coefficients. The detail coefficients capture the high-frequency areas of the signal while the approximation coefficients capture the low-frequency areas of the signal. Here are the results for the Haar wavelet and Db8 wavelet:



and here's the MATLAB code:

```
%Consider the waveform extracted from the attached 'sample.wav' file. You can use 'audioread' command
%in Matlab to read the signal from the file.


[data, sampleRate] = audioread('sample.wav');


%plotting the waveform
time = linspace(0, length(data)/sampleRate,numel(data));
plot(time, data);


%a. Perform a wavelet decomposition with a Haar wavelet and then with a Daubechies-8 wavelet.
wavelet = 'haar';
level = 3;

[coefficients, levellength] = wavedec(data, level, wavelet);


wavelet2 = 'db8';

[coefficients2, levellength2] = wavedec(data, level, wavelet2);


figure;
for i = 1:level
    subplot(level + 1, 2, 2*i - 1)
    plot(wrcoef('d', coefficients, levellength, wavelet, i))
    title(['Haar Wavelet Detail Coefficients Level', num2str(i)]);
    xlabel('Time(s)')
    ylabel('Detail Coefficient')

    if i == level
        subplot(level + 1, 2, 2*i - 1)
        plot(wrcoef('a', coefficients, levellength, wavelet, i))
        title(['Haar Wavelet Approximation Coefficients Level', num2str(i)]);
        xlabel('Time(s)')
        ylabel('Approximation Coefficient')
    end
end


figure;
for i = 1:level
    subplot(level + 1, 2, 2*i - 1)
    plot(wrcoef('d', coefficients2, levellength2, wavelet2, i))
    title(['Db8 Wavelet Detail Coefficients Level', num2str(i)]);
    xlabel('Time(s)')
    ylabel('Detail Coefficient')

    if i == level
        subplot(level + 1, 2, 2*i - 1)
        plot(wrcoef('a', coefficients2, levellength2, wavelet2, i))
        title(['Db8 Wavelet Approximation Coefficients Level', num2str(i)]);
        xlabel('Time(s)')
        ylabel('Approximation Coefficient')
    end
end
```

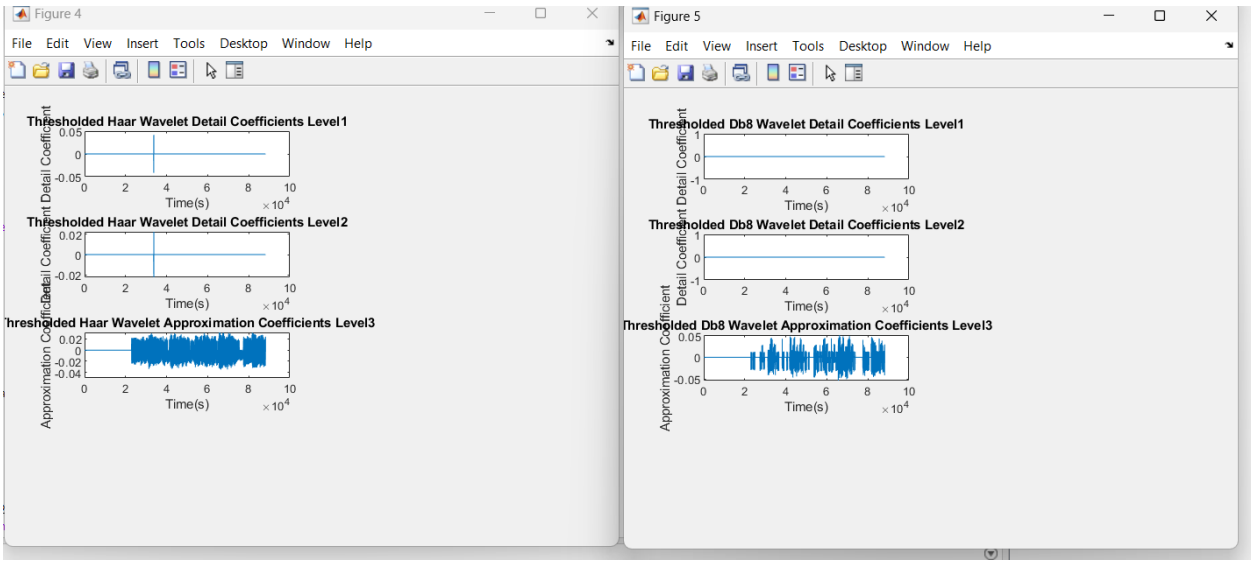
The explanation of each level are as follows:

The coefficients at level 1 represent the highest frequency components at the first level of decomposition. The second level of the wavelets captures slower changes compared to the first level, and the coefficients at level 3 represent the low-frequency components of the signal showing much smoother signal.

Haar wavelets will represent sudden changes very well, while Db8 wavelets can represent smoother gradual changes better. We can reconstruct the signal from the approximation coefficients. For Haar the reconstruction of the wavelet might be more abrupt and less smooth than the Db8 wavelet while the Db8 wavelet will have higher overall reconstruction quality since the approximation waveform for the Db8 wavelet is a little bit smoother. .

Threshold all coefficients which are about three 3 standard deviations smaller than the largest coefficient (i.e. set them equal to 0). Reconstruct the signals. Listen to them and evaluate them both visually and acoustically (compare the two).

Here are the plots for the different Haar and Db8 threshold coefficients:



And here is the code:

```
%b. Threshold all coefficients which are about three 3 standard deviations smaller
% than the largest coefficient (i.e. set them equal to 0). Reconstruct the signals.
% Listen to them and evaluate them both visually and acoustically (compare the two).

haarThreshold = mean(coefficients) + 3 * std(coefficients);
coefficients(abs(coefficients) < haarThreshold) = 0;

haarThreshold = mean(coefficients) + 3 * std(coefficients2);
coefficients2(abs(coefficients2) < db8Threshold) = 0;



figure;
for i = 1:level
    subplot(level + 1, 2, 2*i - 1)
    plot(wrcoef('d', coefficients, levellength, wavelet, i))
    title(['Thresholded Haar Wavelet Detail Coefficients Level', num2str(i)]);
    xlabel('Time(s)')
    ylabel('Detail Coefficient')

    if i == level
        subplot(level + 1, 2, 2*i - 1)
        plot(wrcoef('a', coefficients, levellength, wavelet, i))
        title(['Thresholded Haar Wavelet Approximation Coefficients Level', num2str(i)]);
        xlabel('Time(s)')
        ylabel('Approximation Coefficient')
    end
end


figure;
for i = 1:level
    subplot(level + 1, 2, 2*i - 1)
    plot(wrcoef('d', coefficients2, levellength2, wavelet2, i))
    title(['Thresholded Db8 Wavelet Detail Coefficients Level', num2str(i)]);
    xlabel('Time(s)')
    ylabel('Detail Coefficient')

    if i == level
        subplot(level + 1, 2, 2*i - 1)
        plot(wrcoef('a', coefficients2, levellength2, wavelet2, i))
        title(['Thresholded Db8 Wavelet Approximation Coefficients Level', num2str(i)]);
        xlabel('Time(s)')
        ylabel('Approximation Coefficient')
    end
end


haarReconstruct = waverec(coefficients, levellength, wavelet);
db8Reconstruct = waverec(coefficients2, levellength2, wavelet2);

sound(data, sampleRate);
pause(5);
sound(haarReconstruct, sampleRate);
pause(5);
sound(db8Reconstruct, sampleRate);
```

From this again we see the Haar wavelet captures the abrupt changes well but doesn't capture nuances of the signal well. The db8 wavelet has a less grainy reconstruction than the Haar wavelet since it captures the nuances of the wavelet better, so it would be able to have more of the signal's essential characteristics, indicating better compression quality.

When playing the threshold signal, we realize that there is more graininess of the signal. There is graininess because we get rid of the signals below a certain threshold value and set them to zero. If we set wavelet coefficients to 0, then we introduce artifacts to the system. We might have gotten rid of too many coefficients since 3 standard deviations is quite aggressive. We might have the most prominent signals but ignoring the finer details could make the signal sound grainy. From what is discovered both signals are grainy, but the D-8 wavelet signal is a lot softer than the Haar wavelet. The smoothness of the D-8 wavelet can make it softer than that of the Haar wavelet.

What can you say about the compression abilities of a Haar and of a D-8 wavelet?

The Haar wavelet is more computationally simple than the D-8 wavelet. The Haar wavelet is much better in capturing sharp changes due to its step shape. However, it loses a lot more detail than D-8 wavelet so it is better at compressing blocky signals but not as effective at compressing smoother signals losing more detail the smoother the signal gets. Because of the blockiness, it is better at compressing signal with more abrupt changes, such as image compression.

In contrast, the D-8 wavelet is smoother with a lot more vanishing moments in contrast to Haar. The D-8 wavelet can help compress images more effectively if the image has smoother transitions without too much loss. We are able to better approximate the signal better with more information retained during the compression especially for smoother images with gently flows from one color to another (e.g. gradient images).

Problem 3

Consider a sample, x from the MNIST data-set. Add gaussian noise sampled from $N(0, \sigma^2)$ to each pixel of

the image as:

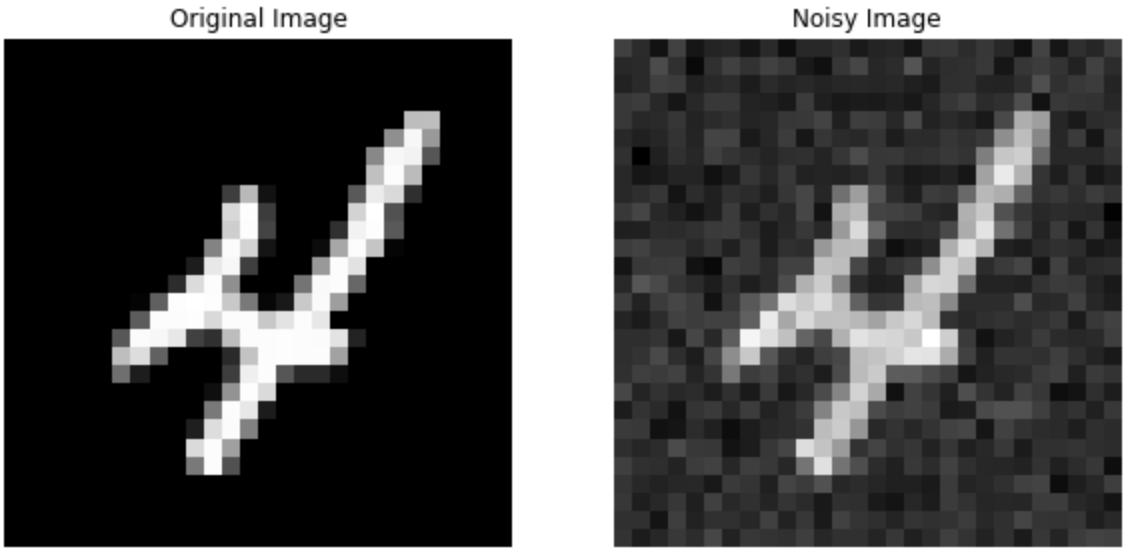
$y(i,j) = x(i,j) + n(i,j), \quad i,j = 1,...,32$ **where** i,j **are row and column indices of the image respectively and** $n(i,j) \sim N(0,\sigma_z^2).\ \sigma_z^2 = 0.75$

Carry out a 2-D Haar wavelet decomposition.

The first thing that we can do here is to Load the MNIST image can add noise to the image. This additional noise for the image will be used for testing in the future. We will use the following code to do this:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

image = train_images[9]

sigma_z = np.sqrt(500)

noise = np.random.normal(0, sigma_z, image.shape)

noisy_image = image + noise

noisy_image_clip = np.clip(noisy_image, 0, 255)

plt.figure(figsize=(10, 5))

plt.subplot(1,2,1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1,2,2)
plt.title("Noisy Image")
plt.imshow(noisy_image, cmap='gray')
plt.axis('off')
```

and here’s the image.



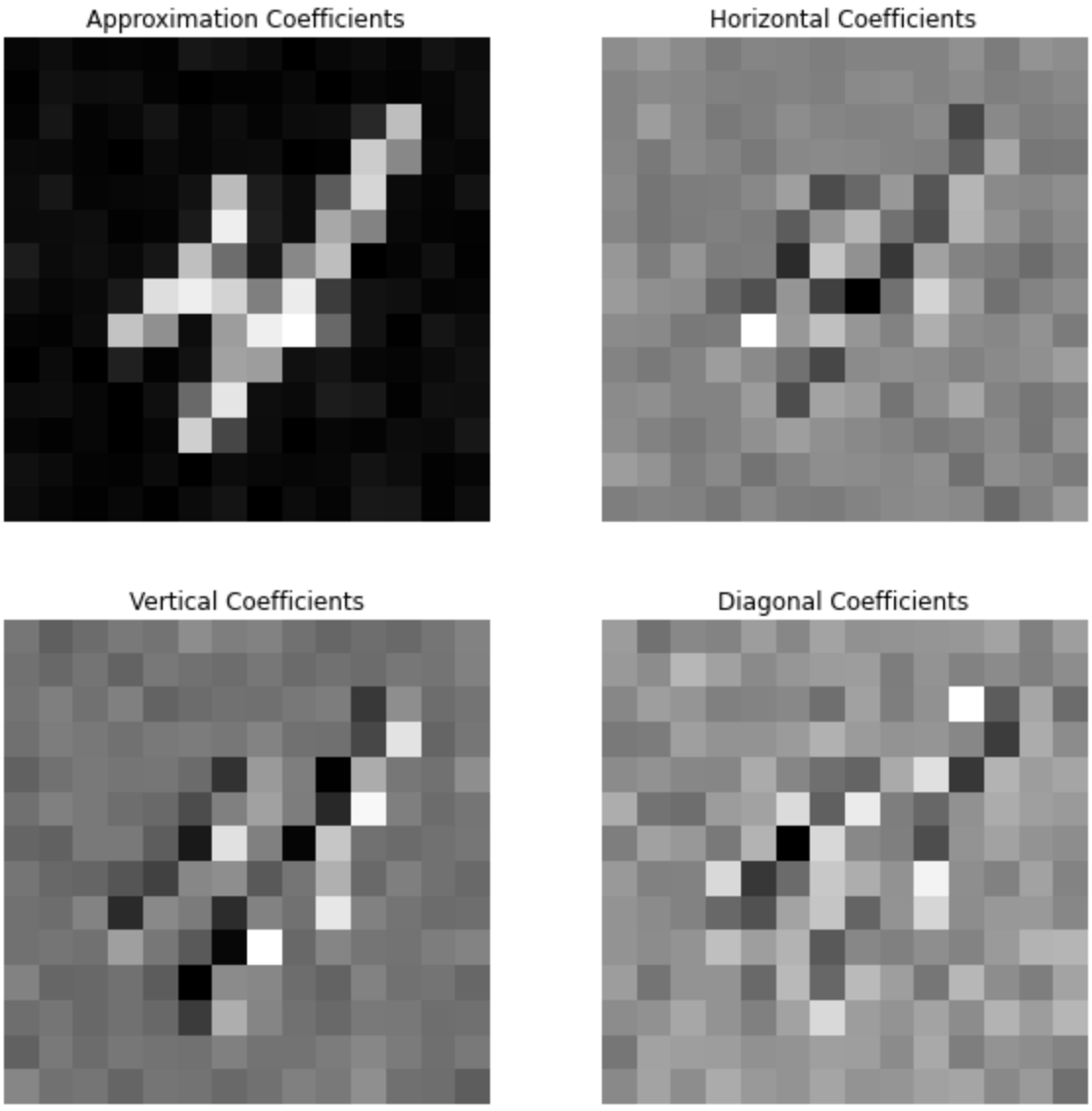
As you can see we introduced some standard deviations of noise to the image.

To perform the 2-D Haar wavelet decomposition of the image, we will perform 4 sub bands of decomposition, the Low Low (approximation coefficients), Low High (horizontal details) High Low (Vertical Details) and High High (diagonal Details). After this, we will visualize the coefficients.

Here’s the code:

```
import pywt

imagedecomp = noisy_image_clip.astype(np.float32)


LL, (LH, HL, HH) = pywt.dwt2(imagedecomp, 'haar')
figure, axis = plt.subplots(2, 2, figsize=(10, 10))

axis[0,0].imshow(LL, cmap = 'gray')
axis[0,0].set_title('Approximation Coefficients')
axis[0,0].axis('off')


axis[0,1].imshow(LH, cmap = 'gray')
axis[0,1].set_title('Horizontal Coefficients')
axis[0,1].axis('off')

axis[1,0].imshow(HL, cmap = 'gray')
axis[1,0].set_title('Vertical Coefficients')
axis[1,0].axis('off')


axis[1,1].imshow(HH, cmap = 'gray')
axis[1,1].set_title('Diagonal Coefficients')
axis[1,1].axis('off')
```

and the visualization(s):



What can you theoretically say about the wavelet coefficient structure in relation to that of the observed signal which is a superposition of an unknown signal and noise (i.e., do the statistical properties change)?

1. Gaussian noise affects both the approximation and detail coefficients. However, it looks like it affects the detail coefficients more because detail coefficients tend to be more sensitive to noise.

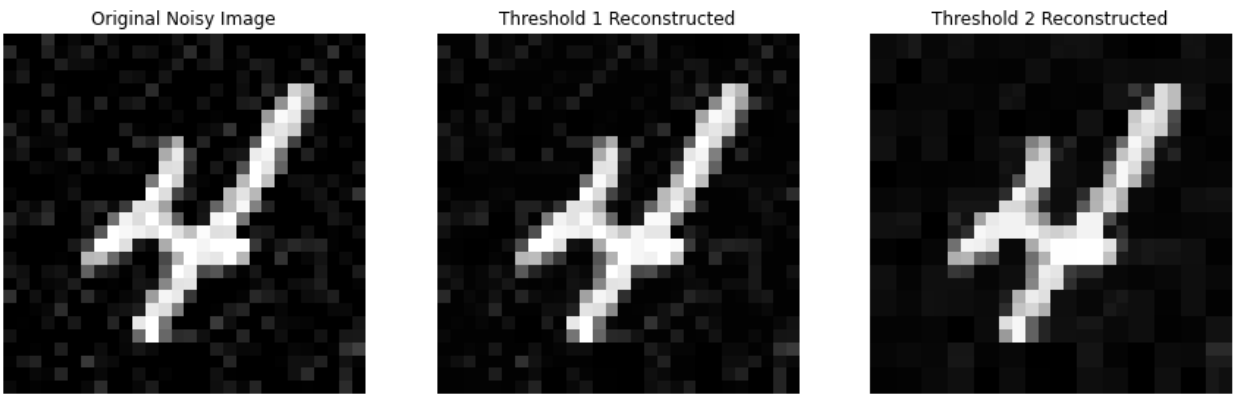
2. Usually the natural images have sparse representation, meaning that the most of the image can be capture by only a few wavelet coefficients. However the noise spreads out across more coefficients, so the sparsity of the representation is decreased. Now that the noise is across more coefficients, we experience more difficulties in signal recovery.
3. The addition of noise to the signal might alter the Gaussian distribution of the wavelet coefficients, an effect that most likely would have to be mitigated by techniques such as thresholding, and it might introduce heavier tails or skewness to the distribution as well. We can try to reduce the number of coefficients affected by noise.
4. We will need robust denoising techniques to mitigate the impact of the noise, removing the noise while simultaneously preserve different features of the original signal. We can use techniques such as Bayesian analysis to do this, using prior knowledge of the coefficients to improve the performance.
5. It's harder to determine the patterns in the wavelet coefficient structure when noise is introduced, and the noise makes it harder to analyze and process the data. This can negatively affect the performance of the techniques that will work with wavelet coefficients. We can mitigate this by careful wavelet selection and algorithm selection.

Next threshold the wavelet coefficients for each of the following levels, $T_1 = \frac{1}{10}\sigma_z\sqrt{\log(1024)}$, $T_2 = \frac{1}{2}\sigma_z\sqrt{\log(1024)}$, and reconstruct the signal.

Here we can reconstruct the image by performing the Inverse Discrete Wavelet Transform. We can apply the thresholding to the Horizontal, Vertical, and Diagonal Coefficients (we maintain the lower frequency part to preserve the "big picture" part of the image) and based on these coefficients, we can reconstruct the image using the Inverse Discrete Wavelet Transform. Here is the script to do this.

```
### c. Next threshold the wavelet coefficients for each of the following
#levels, $T_1 = \frac{1}{10} \sigma_z \sqrt{\log(1024)}$, $T_2 = \frac{1}{2} \sigma_z \sqrt{\log(1024)}$,
#and reconstruct the signal.

def threshold(coefficients, threshold):
    return pywt.threshold(coefficients, threshold, 'soft')

Threshold_1 = (1/10) * sigma_z * np.sqrt(np.log(1024))
Threshold_2 = (1/2) * sigma_z * np.sqrt(np.log(1024))


LH_Threshold1 = threshold(LH, Threshold_1)
HL_Threshold1 = threshold(HL, Threshold_1)
HH_Threshold1 = threshold(HH, Threshold_1)


LH_Threshold2 = threshold(LH, Threshold_2)
HL_Threshold2 = threshold(HL, Threshold_2)
HH_Threshold2 = threshold(HH, Threshold_2)

Threshold_1_Reconstructed = pywt.idwt2((LL, (LH_Threshold1, HL_Threshold1, HH_Threshold1)), 'haar')
Threshold_2_Reconstructed = pywt.idwt2((LL, (LH_Threshold2, HL_Threshold2, HH_Threshold2)), 'haar')


plt.figure(figsize=(15, 5))

plt.subplot(1,3,1)
plt.title("Original Noisy Image")
plt.imshow(noisy_image_clip, cmap='gray')
plt.axis('off')

plt.subplot(1,3,2)
plt.title("Threshold 1 Reconstructed")
plt.imshow(Threshold_1_Reconstructed, cmap='gray')
plt.axis('off')

plt.subplot(1,3,3)
plt.title("Threshold 2 Reconstructed")
plt.imshow(Threshold_2_Reconstructed , cmap='gray')
plt.axis('off')

plt.show()
```

and here are the results:



By comparing the reconstructions for different thresholds, which threshold would you say works best in denoising the signal?

From the image, we can see that threshold 1 is more conservative than threshold 2. The reconstruction with threshold 1 was able to retain more of the signal with the tradeoff being keeping also some of the noise. The reconstruction with threshold 2 was able to remove more noise and have a cleaner look to the signal, with the tradeoff being losing more of the original signal details. Threshold 1 is preferred for more detail, while threshold 2 is preferred for reducing noise the most. Overall, it's a tradeoff. Since we are asking about denoising, I would say the threshold 2 works best in this scenario.

What is the algorithm would you then propose in general?

The algorithm that I will propose would be as follows:

First we will preprocess the image by converting it to grayscale and perform normalization.

Then we will choose an appropriate wavelet based on the application, with Daubechies for smoother scale inputs and Haar to evaluate more abrupt changes. We will then obtain the approximation and detail coefficients by performing 2-D wavelet decomposition .

Afterwards, then we will define a thresholding function such as the one in part (c) and apply hard or soft thresholding to the detail coefficients.

From these coefficients that were thresholded, we can perform a inverse wavelet transform which will reconstruct the denoised image. Finally we will rescale the image and perform post-processing.

We evaluate a bunch of metrics such as the Signal to Noise Ratio(SNR) or the Mean Square Error (MSE). We then can use other techniques such as Bayesian Analysis or machine learning to fine-tune the threshold value to optimize the tradeoff between signal preservation and signal cleanliness.

Now train a convolutional autoencoder model based on the architecture in Part (1), The model should take the noisy images y as input (instead of x as was in Part 1) and output an approximation $\hat{x}$ of the clean image x (So, your target output would remain same, i.e. x as was in Part (1).

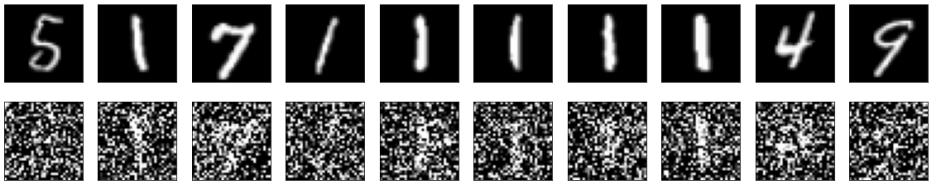
To train my autoencoder, I keep the code the same as the autoencoder code that was attached to this project. There were some small differences. The noise function for this problem was $\sqrt{0.75}$ compared to 0.75 in question 1. This code snippet will display the noisy and original images and the clean and denoised images, while the code in question 1 will only display the input/output images as trained by the autoencoder. We train the autoencoder on the noisy data in the problem in contrast to training the autoencoder on original data from MNIST in question 1. We validate using the noisy test data in this question as well.

Here's the changed code:

```
#################################################################################################

'''
Display noisy and denoised data
'''
predictions = autoencoder.predict(noisy_test_data)
display(train_data, predictions, 'clean_vs_denoised')

'''
Bottleneck layer output
'''
bottleneck_model = Model(inputs=input, outputs=bottleneck_layer_output)
bottleneck_encoding = bottleneck_model.predict(test_data)
noisy_bottleneck_encoding = bottleneck_model.predict(noisy_test_data)
prediction_bottleneck_encoding = bottleneck_model.predict(predictions)

'''
Code to Visualize Encoding (Not posted in the project problem.)
'''


def visualize_encoding(encodings, labels, digits, num_samples):
    sample_indices = {}
    for digit in digits:
        indices = np.flatnonzero(labels == digit)
        sampled_indices = indices[:num_samples]
        sample_indices[digit] = sampled_indices

    fig, axes = plt.subplots(nrows=len(digits), ncols=num_samples, figsize=(num_samples * 2, len(digits) * 2))
    for row, digit in enumerate(digits):
        for col in range(num_samples):
            if len(digits) > 1:
                ax = axes[row][col]
            else:
                ax = axes[col]
            encoding = encodings[sample_indices[digit][col]]
            ax.imshow(encoding.reshape(4, 8), cmap='gray')
            ax.set_title(f"Digit {digit}")
            ax.axis('off')
    plt.show()


print("Visualizing Clean Encodings:")
visualize_encoding(bottleneck_encoding, test_label, [7, 8], 5)
print("Visualizing Noisy Encodings:")
visualize_encoding(noisy_bottleneck_encoding, test_label, [7, 8], 5)
print("Visualizing Denoised/Predicted Encodings:")
visualize_encoding(prediction_bottleneck_encoding, test_label, [7, 8], 5)
```

The results that we obtain are as follows:

Original vs Noisy Image:

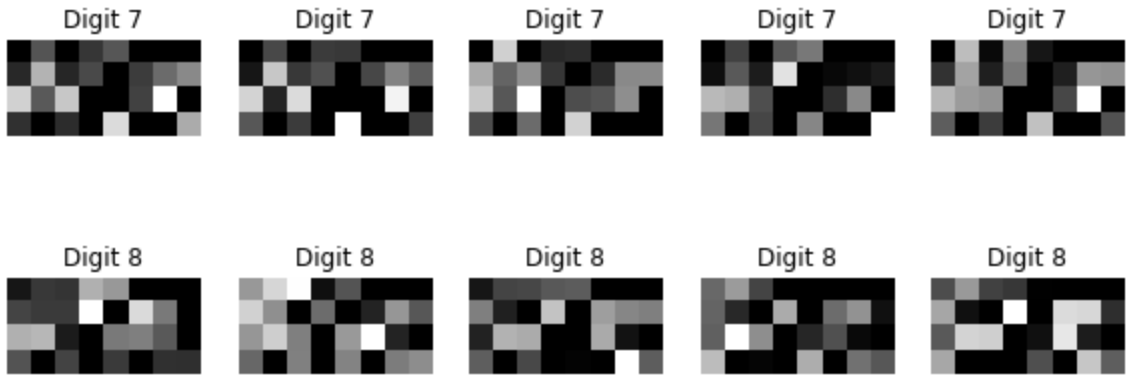


Original vs Denoised Image:

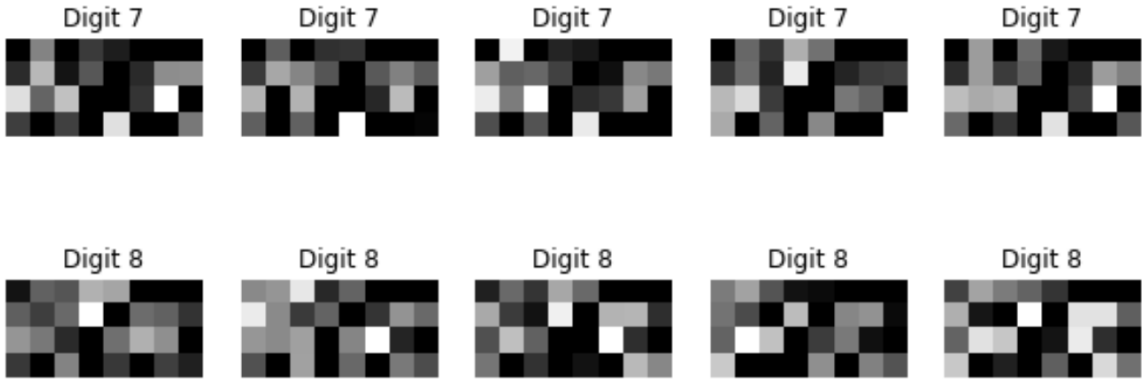
From this we can conclude that this autoencoder was able to decode the original images fairly successfully with the loss of a bit of detail through the data. This indicates that the training was successful and indicates that the wavelet is successful in both directions, whether placing the noisy or clean data as the input.

g. Compare the bottleneck layer output of this new model with the earlier bottleneck layer outputs from Part (1). Explain your observation.

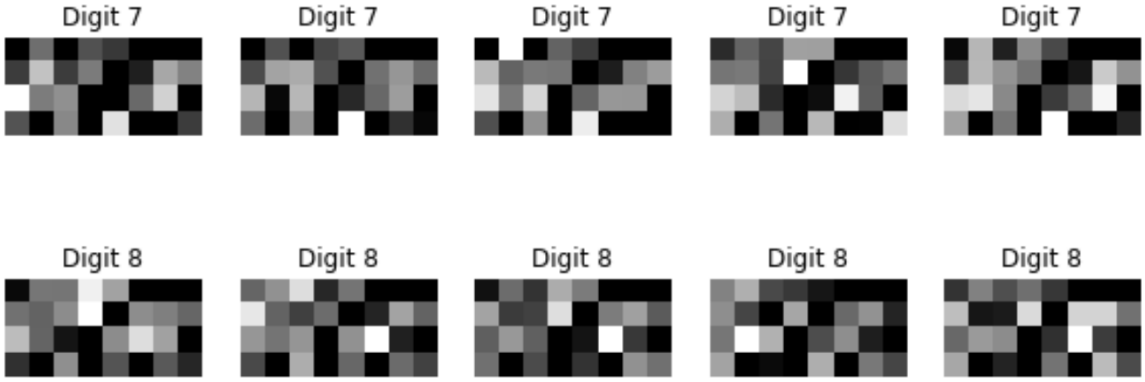
Here is the original signal bottleneck layer output:



And here is the Denoised signal bottleneck layer output:



as well as the noisy signal bottleneck layer output:



As we can see, the denoised signal bottleneck layer output has cleaner data/visualization than the original signal bottleneck layer output. This is expected and indicative of effective denoising and cleaning of any unnecessary data

needed to identify the digits correctly. Another evidence that the autoencoder is performing well is that the denoised and original signal bottleneck layer output have very similar overall shapes, thus identifying very similar “most” important features to preserve. We also see less variation in values in the denoised signal bottleneck layer output in contrast to the original signal layer output. We see less randomness and more structure in the denoised representation. Finally, since the original signal and denoised signal bottleneck layer output still have noticeable differences, we can safely infer that overfitting has not occurred in this MNIST dataset.

Can you improve the performance of the autoencoder by changing the number of layers or the number of convolutional filters in the layers? Demonstrate with empirical results.

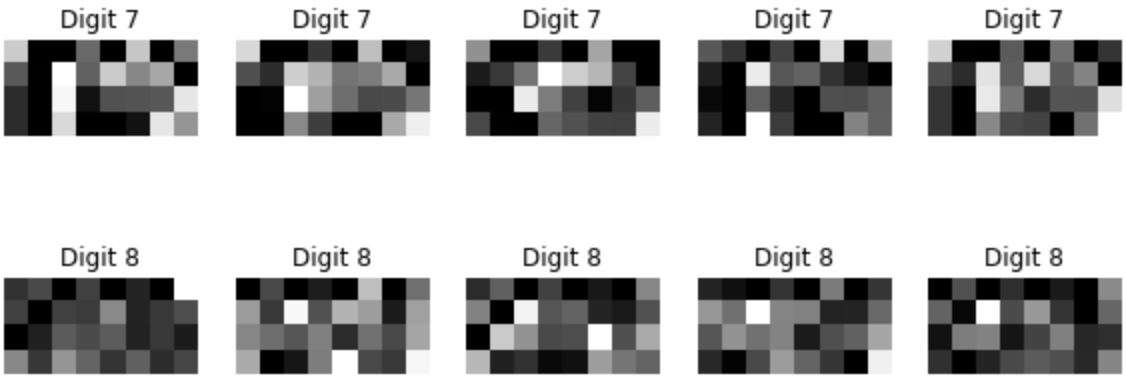
For efficiency purposes, I reduced the number of epochs to 25 and compare the validation loss and the loss in order as a metric to determine the performance of the autoencoder. I will also include any image if necessary.

The 3 tests I performed were as follows:

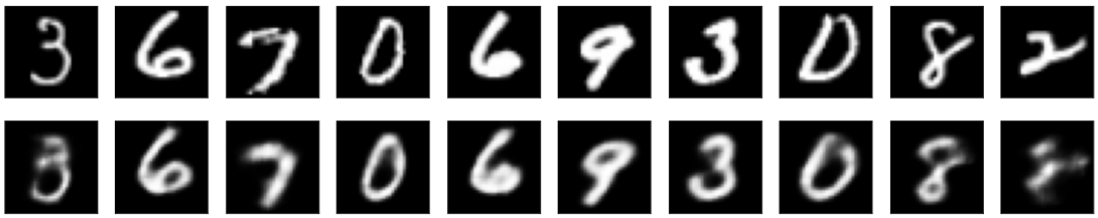
8 Convolutional Filters, 2 Convolutional Layers:

The code is the same as the original code.

And here’s the denoised encoding.



And the clean vs denoised digits from MNIST after 25 epochs.



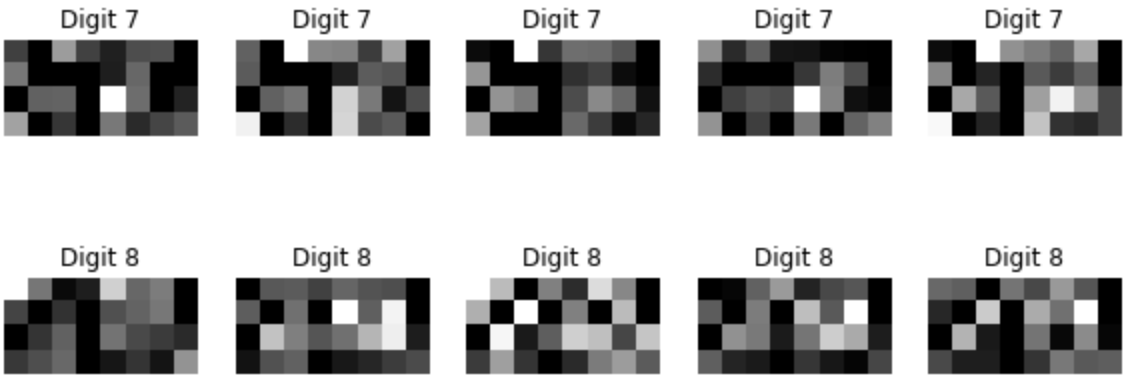
The result is a validation loss of 0.1539 and a overall loss of 0.1539.

16 Convolutional Filters, 2 Convolutional Layers

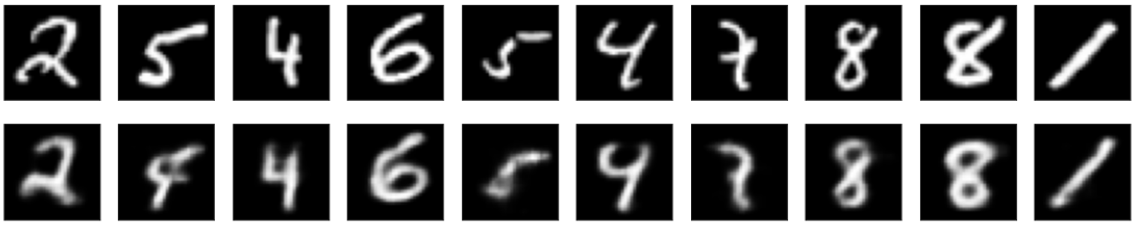
The code is the same as the original code, except I changed the encoder and decoder layer sizes to 16 instead of 8.

```
def sixteenFilterLayerTest():
    #SECOND TEST 16 FILTER LAYERS
    ##################################
    encoder_Layer1_size = 16
    encoder_Layer2_size = 16
    decoder_Layer1_size = 16
    decoder_Layer2_size = 16
```

And here's the denoised encoding.



And the clean vs denoised digits from MNIST after 25 epochs.



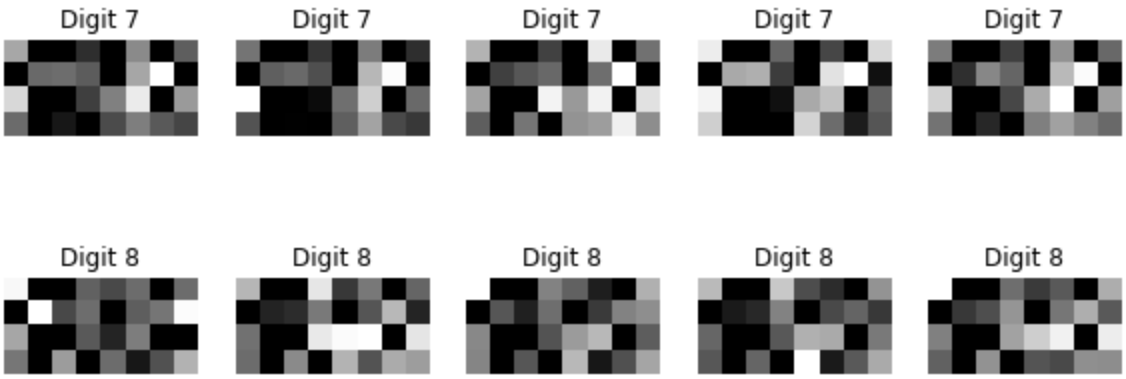
The result is a validation loss of 0.1501 and a overall loss of 0.1488.

8 Convolutional Filters, 3 Convolutional Layers

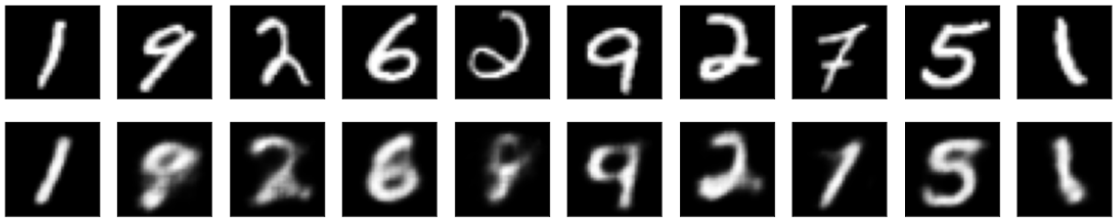
The code is the same as the original code, except I added another upsampling and downsampling layer to the code.

```
input = layers.Input(shape=(32, 32, 1))

# Encoder


x1c = layers.Conv2D(encoder_Layer1_size, (encoder_Layer1_kernel, encoder_Layer1_kernel), activation="relu", padding="same")(input)
x1d = layers.MaxPooling2D((2, 2), padding="same")(x1c)
x2c = layers.Conv2D(encoder_Layer2_size, (encoder_Layer2_kernel,encoder_Layer2_kernel), activation="relu", padding="same")(x1d)
x2d = layers.MaxPooling2D((2, 2), padding="same")(x2c)
x3c = layers.Conv2D(encoder_Layer2_size, (encoder_Layer2_kernel, encoder_Layer2_kernel), activation="relu", padding="same")(x2d)
x3d = layers.MaxPooling2D((2, 2), padding="same")(x3c)

'''
Fully connected layers
'''
xf = layers.Flatten()(x3d)

xld1 = layers.Dense(128, activation='relu')(xf)

bottleneck_layer_size = 32

bottleneck_layer_output = layers.Dense(bottleneck_layer_size, activation='relu')(xld1)


# Decoder
'''
Fully connected layers
'''
xlu1 = layers.Dense(128, activation='relu')(bottleneck_layer_output)

xlu2 = layers.Dense(encoder_Layer2_size*(32//8)*(32//8), activation='relu')(bottleneck_layer_output)

xu = layers.Reshape((32//8, 32//8, encoder_Layer2_size))(xlu2)

'''
First upsampling layer (upsampling by a factor of 2 done using transposed convoltion)
'''
x = layers.Conv2DTranspose(decoder_Layer1_size, (decoder_Layer1_kernel, decoder_Layer1_kernel), strides=2, activation="relu", padding="same")(xu)
'''
Second/Third upsampling layer (upsampling by a factor of 2 done using transposed convoltion)
'''
x = layers.Conv2DTranspose(decoder_Layer2_size, (decoder_Layer2_kernel, decoder_Layer2_kernel), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2DTranspose(decoder_Layer2_size, (decoder_Layer2_kernel, decoder_Layer2_kernel), strides=2, activation="relu", padding="same")(x)
```

And here's the denoised encoding.



And the clean vs denoised digits from MNIST after 25 epochs.



The result is a validation loss of 0.166 and a overall loss of 0.1646.

From the empirical rudimentary results that were obtained through testing number of layers and filter layers, we can conclude that increasing the number of convolutional layers did improve performance but increasing the number of Convolutional Filters did not improve performance. This is because beyond a certain point, increasing the number of filters leads to diminishing returns. However, increasing the number of layers can more or less hierarchically represent the data in many more ways and be able to detect more edges and features associated with a particular datapoint in this case, written numbers, thus increasing the performance of the autoencoder. The clean vs. denoised digits shows a noticeable improvement in performance with the increasing number of layers. Due to shortage of time we cannot perform a comprehensive verification but this is the result after a short test.