I'm not robot
reCAPTCHA
Continue

Haar wavelet transform python

In an earlier blog post, we've seen how we can use Signal Processing techniques for the classification of time series and signals. A very short summary of that message is: We can use the Fourier Transform to transform a signal from the time domain to the frequency domain. The peaks in the frequency spectrum indicate the most common frequencies in the signal. The bigger and sharper a peak is, the more frequent a frequency is in a signal. The location (frequency value) and height (amplitude) of the peaks in the frequency spectrum can then be used as input for classifiers such as Random Forest or Gradient Boosting. This simple approach works surprisingly well for many classification problems. In that blog post, we were able to classify the Human Activity Recognition dataset with ~91% accuracy. The general rule is that this approach to using the Fourier Transform will work very well when the frequency spectrum is at a standstill. That is, the frequencies present in the signal are not time-dependent; if a signal contains a frequency of x, this frequency should be present equally throughout the signal. The more non-stationary/dynamic a signal is, the worse the results will be. That's too bad, because most of the signals we see in real life are non-stationary in nature. Whether we are talking about ECG signals, the stock market, equipment or sensor data, etc, etc, in real life problems start to become interesting when we are dealing with dynamic systems. A much better approach to analyzing dynamic signals is to use the Wavelet Transform instead of the Fourier Transform. Although the Wavelet Transform is a very powerful tool for the analysis and classification of time series and signals, it is unfortunately not known or popular in the field of Data Science. This is partly because you need to have some prior knowledge (about signal processing, Fourier Transformation and Mathematics) before you understand the mathematics behind the Wavelet Transform. However, I think it is also due to the fact that most books, articles and papers are far too theoretical and do not provide enough practical information on how it should and can be used. In this blog post we will see the theory behind the Wavelet Transform (without going too much into mathematics) and also see how it can be used in practical applications. By providing Python code at every step of the way you should be able to use the Wavelet Transform in your own applications by the end of this post. The content of this blog post is as follows: Introduction Theory 2.1 From Fourier Transform to Wavelet Transform 2.2 How does the Wavelet Transform work? 2.3 Different types of Wavelet families 2.4 Wavelet Transform vs Discreet Wavelet Transform 2.5 More on the Discreet Wavelet Transform: The DWT as a filter bank. Practical Applications 3.1 Visualizing the State-Space using the Continuous Continuous Transform 3.2 Using the Continuous Wavelet Transform and a Convolutional Neural Network to classify signals 3.2.1 Loading the UCI-HAR time-series dataset 3.2.2 Applying the CWT to the dataset and transforming the data to the correct format 3.2.3 Training the Convolutional Neural Network with the CWT 3.3 Decon structuring a signal using the DWT 3.4 Removing (high frequency) noise using the DWT 3.5 Using the Discreet Wavelet Transform to classify signals 3.5.1 The idea behind Discreet Wavelet classification 3.5.2 Generating features per sub-band 3.5.3 Using the features and scikit-learn classifiers to classify two datasets. 3.6 Comparison of classification accuracy between DWT, Fourier Transform and Recurrent Neural Networks Finals Words PS: In this blog post we will usually use the Python package PyWavelets, so go ahead and install it with pip install pygolf. 2.1 From Fourier Transform to Wavelet Transform In the previous blog post we saw how the Fourier Transform works. That is, by multiplying a signal with a series of sine waves with different frequencies, we are able to determine which frequencies are present in a signal. If the point product between our signal and a sine wave of a certain frequency results in a large amplitude, it means that there is a lot of overlap between the two signals, and our signal contains this specific frequency. This is natural because the dot product is a measure of how many two vectors/signals overlap. The thing about the Fourier Transform is that it has high resolution in the frequency domain, but zero resolution in the time domain. This means that it can tell us exactly which frequencies are present in a signal, but not at which location these frequencies occurred. This can easily be demonstrated as follows: t_n = 1 N = 100000 T = t_n / N f_s = 1/T xa = np.linspace(0, t_n, num=N) xb = np.linspace(0, t_n/4, num=N/4) frequencies = [4, 30, 60, 90] y1a, y1b = np.sin(2*np.pi*frequencies[0]*xa), np.sin(2*np.pi*frequencies[0]*xb) y2a, y2b = np.sin(2*np.pi*frequencies[1]*xa), np.sin(2*np.pi*frequencies[1]*xb) y3a, y3b = np.sin(2*np.pi*frequencies[2]*xa), np.sin(2*np.pi*frequencies[2]*xb) y4a, y4b = np.sin(2*np.pi*frequencies[3]*xa), np.sin(2*np.pi*frequencies[3]*xb) composite_signal1 = y1a + y2a + y3a + y4a composite_signal2 = np.concatenate([y1b, y2b, y3b, y4b]) f_values1, fft_values1 = get_fft_values(composite_signal1, T, N , f_s) f_values2, fft_values2 = get_fft_values(composite_signal2 nrows), T, N, f_s) fig, axarr = plt.subplots(nrows=2, ncols=2, figsize=(12,8)) axarr[0,0].plot(xa, composite_signal1) axarr[1,0].plot(xa, composite_signal2) axarr[0,1].plot(f_values1, fft_values1 axarr[1,1].plot(f_values2, (...) plt.tight_layout() plt.show() xa = np.linspace(0, t_n, num=N)xb = np.linspace(00, t_n/4, num=N/4)frequencies = [4, 30, 60, 90]y1a, y1b = np.sin(2*np.pi*frequencies[0]*xa), np.sin(2*np.pi*frequencies[0]*xa), y2b = np.sin(2*np.pi*frequencies[1]*xa), np.sin(2*np.pi*frequencies[1]*xb)y3a, y3b = np.sin(2*np.pi*frequencies[2]*xa), np.sin(2*np.pi*frequencies[2]*xb)y4a, y4b = np.sin(2*np.pi*frequencies[3]*xa), np.sin(2*np.pi*frequencies[3]*xb)composite_signal1 = y1a + y2a + y3a + y4acomposite_signal2 = np.concatenate([y1b, y2b, y3b, y4b])f_values1, fft_values1 = get_fft_values(composite_signal1 , T, N, f_s)f_values2, fft_values2 = get_fft_values(composite_signal2, T, N, f_s)fig, axarr = plt.subplots(nrows=2, ncols=2, figsize=(12,8))axarr[0,0].plot(xa, composite_signal1)axarr[1,0].plot(xa, composite_signal2)axarr[0,1].plot(f_values1, fft_values1)axarr[1,1].plot(f_values2, fft_values2) Figure 1. The signals and frequency spectrum of a signal containing four frequencies at all times (above), four different frequencies at four different times (bottom). In Figure 1 we see a signal with four different frequencies (and) in the upper left and the frequency spectrum on the right. In the bottom figure, we can see the same four frequencies, only the first is present in the first quarter of the signal, the second in the second quarter, etc. Moreover, we see the frequency spectrum again on the right side. What is important to note here is that the two frequency spectros contain exactly the same four peaks, so it can't tell us where in the signal these frequencies are present. The Fourier Transform cannot distinguish between the first two signals. PS: The side lobes we see in the lower frequency spectrum, is due to the discontinuity between the four different frequencies. In an effort to overcome this problem, scientists have come up with the Short Time Fourier Transformation. In this approach, the original signal is broken down into different parts of equal length (which may or may not overlap) by using a sliding window before applying the Fourier Transform. The idea is very simple: if we split our signal into 10 parts, and the Fourier Transform detects a specific frequency in the second part, then we know for sure that this frequency occurred between th and th of our original signal. The main problem with this approach is that you run into the theoretical limits of the Fourier Transform known as the uncertainty principle. The smaller we make the size of the window, the more we know where a frequency in the signal occurred, but less about the frequency value itself. The larger we size the window the more we will know about the frequency value and less about the time. A better approach for analyzing signals with a dynamic frequency spectrum is the Wavelet Transform. The Wavelet Transform has high resolution in both the frequency and time domain. The not only what frequencies are present in a signal, but also at what point these frequencies have occurred. This is is by working with different scales. First we look at the signal with a large scale /window and analyze 'large' functions and then we look at the signal with smaller scales to analyze smaller functions. The time and frequency resolutions of the different methods are illustrated in Figure 2. Figure 2. A schematic overview of the time and frequency resolutions of the different transformations compared to the original time series dataset. The size and orientations of the block give an indication of the resolution size. In Figure 2 we see the time and frequency resolutions of the various transformations. The size and orientation of the blocks indicate how small the functions we can distinguish in the time and frequency domain are. The original time series has a high resolution in the time domain and the zero resolution in the frequency domain. This means that we can distinguish very small functions in the time domain and not functions in the frequency domain. Opposite that is the Fourier Transform, which has a high resolution in the frequency domain and zero resolution in the time-domain. The Short Time Fourier Transform has a medium resolution in both the frequency and time domain. The Wavelet Transform has: for small frequency values a high resolution in the frequency domain, low resolution in the time domain, for large frequency values a low resolution in the frequency domain, high resolution in the time domain. In other words, the Wavelet Transforms makes a trade-off; on scales in which time-dependent functions are interesting, it has a high resolution in the time domain and on scales in which frequency-dependent functions are interesting, it has a high resolution in the frequency domain. And as you imagine, this is exactly the kind of trade-off we're looking for! 2.2 How does the Wavelet Transform work? The Fourier Transform uses a series of sine waves with different frequencies to analyze a signal. That is, a signal is represented by a linear combination of sinus waves. The Wavelet Transform uses a range of features called wavelets, each with a different scale. The word wavelet means a small wave, and this is exactly what a wavelet is. Figure 3. The difference between a sine wave and a Wavelet. The sine wave is infinitely long and the Wavelet is localized over time. In Figure 3 we see the difference between a sine wave and a wavelet. The main difference is that the sine wave is not localized over time (it stretches from infinity to +infinity) while a wavelet is localized over time. This allows the wavelet transformation to obtain time information in addition to the frequency information. Since the Wavelet is localized over time, we can multiply by the wavelet at different locations over time. We start with the beginning of our signal and move the wavelet slowly to the end of the signal. This procedure also known as a convolution. After we have done this for the original (mother) wavelet, we can scale it so that it gets bigger and repeat the process. This process is illustrated in the figure below. GIF As we can see in the figure above, the Wavelet transformation of a 1-dimensional signal will have two dimensions. This 2-dimensional output of Wavelet's Transform is the timescale display of the signal in the form of a scaleogram. Above the scaleogram is plotted in a 3D plot in the lower left figure and in a 2D color plot in the lower right figure. PS: You also take a look at this youtube video to see how a Wavelet Transform works. So what is this dimension called scale? Since the term frequency is reserved for the Fourier Transformation, the wavelet transformation is usually expressed in scales instead. Therefore, the two dimensions of a scaleogram are time and scale. For those who find frequencies more intuitive than scaling, it is possible to convert scales to pseudo-frequencies with the equation where the pseudo-frequency is, the central frequency of the Mother wavelet is and is the scale factor. We can see that a higher scale factor (longer wavelet) corresponds to a smaller frequency, so by scaling the wavelet in the time domain, we analyze smaller frequencies (a higher resolution) in the frequency domain. And vice versa, by using a smaller scale we have more detail in the time domain. So scales are basically the opposite of frequency. PS: PyWavelets contains the function scale2 frequency to convert from a scale domain to a frequency domain. 2.3 The different types of Wavelet families Another difference between the Fourier Transform and the Wavelet Transform is that there are many different families (types) of wavelets. The wavelet families differ from each other, because for each family a different trade-off is made in how compact and smooth the wavelet looks. This means that we can choose a specific wavelet family that best fits the functions we are looking for in our signal. For example, the PyWavelets library contains 14 mother Wavelets (families of Wavelets): import pywt print(pywt.families (short=False)) ['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse biorthogonal', 'Discrete Meyer (FIR (FIR)Approximation,'Gaussian', 'Mexican hat wavelet', "Morlet wavelet', 'Complex Gaussian wavelets', 'Shannon wavelets', 'Frequency B-Spline wavelets', 'Complex Morlet wavelets'] print (pywt.families (short=False))['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse biohortal', 'Discrete Meyer (Fir(Approximation)' , 'Gaussian', 'Mexican hat wavelet', 'Morlet wavelet', 'Complex Gaussian wavelets', 'Shannon wavelets', 'Frequency Frequency wavelets', 'Complex Morlet wavelets'] Each type of wave has a different shape, smoothness and compactness and is useful for a Purpose. Since there are only two mathematical conditions a wavelet must meet, it is easy to generate a new type of wavelet. The two mathematical conditions are the so-called normalization and orthogonization limitations: A wavelet must have 1) finite energy and 2) zero average. Finite energy means that it is localized in time and frequency; it is integrable and the inner product between the wavelet and the signal always exists. The admissibility condition implies that a wavelet has zero average in the time domain, a zero at zero frequency in the time domain. This is necessary to ensure that it is integrable and the inverse of the wavelet transformation can also be calculated. In addition: A wavelet can be orthogonal or non-orthogonal. A wavelet can be bi-orthogonal or not. A wavelet can be symmetrical or not. A wavelet can be complex or real. If it is complex, it is usually divided into a real part that represents the amplitude and an imaginary part of the phase. A corrugated iron is normalized to have unitary energy. Below we see a plot with different families of wavelets. The first row contains four Discrete Wavelets and the second row four Continuous Wavelets. discrete_wavelets = ['db5', 'sym5', 'coif5', bior2.4] continuous_wavelets = ['mexh', 'morl', 'cgau5', 'gaus5'] list_list_wavelets = [discrete_wavelets, continuous_wavelets] list_funcs = [pywt. Wavelet, pywt. ContinuousWavelet] fig, axarr = plt.subplots(nrows=2, ncols=4, figsize=(16,8)) for ii, list_wavelets in summation(list_list_wavelets): func = list_funcs[ii] row_no = ii for col_no, waveletname in enumerate(list_wavelets): wavelet = func(waveletname) family_name = wavelet.family_name biorthogonal = wavelet.biohogonal orthogonal (wavelet.orthogonal symmetry = wavelet.symmetry if ii == 0: _ = wavelet.wavefun() wavelet_function = _[0] x_values = _[-1] different: wavelet_function, x_values = wavelet.wavefun() if col_no == 0 and ii == 0 : axarr[row_no, col_no].set_ylabel(Discrete Wavelets, fontsize=16) if col_no == 0 and ii == 1: axarr[row_no, col_no].set_ylabel(Continuous Wavelets, fontsize=16) axarr[row_no, col_no].set_title({}.format(family_name), fontsize=16) axarr[row_no, col_no].plot(x_values, wavelet_function) axarr[row_no, col_no].set_yticks([[]]) axarr[row_no , col_no].set_yticklabels([]) plt.tight_layout() plt.show() discrete_wavelets = [db5,, sym5, coif5, bior2.4]continuous_wavelets = ['mexh', 'morl', cgau5, gaus5]list_list_wavelets = [discrete_wavelets, continuous_wavelets]list_funcs = [pywt. Wavelet, pywt. ContinuousWavelet]fig, axarr = plt.subplots(nrows=2, ncols=4, figsize=(16,8))for ii, list_wavelets in enumerate(list_list_wavelets): for col_no, waveletname in enumerate(list_list_wavelets): wavelet = func(waveletname) family_name = wavelet.family_name biorthogonal = waveletlet.biogonrthoal = wavelet.symmetrie wavelet_function, x_values = wavelet.wavefun() indien col_no == 0 en ii == 0: axarr[row_no, col_no].set_ylabel(Discrete Wavelets, fontsize=16) indien col_no == 0 en ii == 1: axarr[row_no, col_no].set_ylabel(Continue Wavelets, fontsize=16) axarr[row_no, col_no].set_title({}.format(family_name), fontsize=16) axarr[row_no, col_no].plot(x_values , wavelet_function) axarr[row_no, col_no].set_yticks([]) axarr[row_no, col_no].set_yticklabels([]) Figuur 5. Verschillende families van Wavelets. Op de eerste rij zien we discrete golfjes en op de tweede rij zien we verschillende continue golfjes. PS: Om te zien hoe alle golfjes eruit zien, u een kijkje nemen op de wavelet browser. Binnen elke wavelet familie kunnen er veel verschillende wavelet subcategorieën die behoren tot die familie. U de verschillende subcategorieën van golfjes onderscheiden door het aantal coëfficiënten (het aantal verdwijnmomenten) en het niveau van ontbinding. Dit wordt hieronder geïllustreerd in voor de ene familie van wavelets genaamd 'Daubechies'. invoer pywt importmatplotlib.pyplot as plt db_wavelets = pywt.wavelist('db')[:5] print(db_wavelets) *** ['db1', 'db2', db3, db4, db5] vijg, axarr = plt.subplots(ncols=5, nrows=5, figsize=(20,16)) fig.suptitle('Daubechies family of wavelets', fontsize=16) for col_no, waveletname in enumerate(db_wavelets): wavelet = pywt. Wavelet(waveletname) no_moments = wavelet.vanishing_moments_psi family_name = wavelet.family_name voor row_no, niveau in enumerate(bereik(1,6)): wavelet_function, scaling_function, x_values = wavelet.wavefun(level = level= level) axarr[row_no, col_no].set_title({} - niveau {}{} verdwijnende momenten{} monsters.format(waveletname, level, no_moments, len(x_values)), loc='left') axarr[row_no, col_no].plot(x_values, wavelet_function, 'bD--') axarr[row_no , col_no].set_yticks axarr[row_no, col_no].set_yticklabels([]) plt.tight_layout() plt.subplots_adjust(top=0.9) plt.show() importmatplotlib.pyplot as pltdb_wavelets = pywt.wavelist('db')[:5]*** ['db1', db2, db3, db4, db5,fig,*fig, axarr = plt.subplots(ncols=5, nrows=5, figsize=(20,16)fig.suptitle('Daubechies family of wavelets', fontsize=16)for col_no, waveletname in enumerate(db_wavelets): wavelet = pywt. Wavelet(waveletname) no_moments = wavelet.vanishing_moments_psi family_name = wavelet.family_name voor row_no, niveau in opsomgebied(bereik(1,6)): wavelet_function, scaling_function, x_values = wavelet.wavefun(level = level) axarr[row_no, col_no].set_title({} - level {}{} vanishing moments{} samples.format(waveletname, level, no_moments, len(x_values)), loc='left') axarr[row_no, col_no]. (x_values, wavelet_function, col_no].set_yticks([]) axarr[row_no, col_no].set_yticklabels([])plt.subplots_adjust (top=0.9) Figure 6. The Daubechies family of wavelets for different orders of disappearing moments and different levels of sophistication. In figure 6 we see waves of the family 'Daubechies' (db) of golfs. In the first column we see the Daubechies waves of the first order (db1), in the second column of the second order (db2), up to the fifth order in the fifth column. PyWavelets contains Daubechies waves up to order 20 (db20). The number of the order indicates the number of disappearing moments. So db3 has three disappearing moments and db5 has 5 disappearing moment. The number of vanishing moments is related to the order of approach and the smoothness of the wavelet. If a wavelet has p disappearing moments, it can approach polynomials of grade p – 1. When selecting a wavelet, we can also specify what the level of decomposition should be. By default, PyWavelets chooses the maximum degradation level that is possible for the input signal. The maximum degradation level (see pywt.dwt_max_level()) depends on the length of the input signal and the wavelet (more on this later). As we can see, as the number of disappearing moments increases, the polynomial degree of the wavelet increases and becomes smoother. And as the level of decomposition increases, the number of samples that this wave is expressed in increases. 2.4 Continuous Wavelet Transform vs Discreet Wavelet Transform As we've seen before (Figure 5), the Wavelet Transform comes in two different and different flavors; continuous and discreet Wavelet Transform. Mathematically, a Continuous Wavelet Transformation is described by the following equation: where is the continuous mother wavelet that is scaled by a factor of and translated by a factor of . The values of the scaling and translation factors are continuous, which means that there can be an infinite amount of wavelets. You the parent wave by a factor of 1.3, or 1.31, and 1,311, and 1.3111 etc. When we talk about the Discreet Wavelet Transform, the main difference is that the DWT uses discrete values for the scale and translation factor. The scale factor increases in powers of two, thus and the translation factor increases entire values ( ). PS: The DWT is only discreet in the scale and translation domain, not in the time domain. In order to work with digital and discrete signals, we also need to diskize our wavelet transformations in the time domain. These forms of the wavelet transformation are called the Discrete-Time Wavelet Transform and the Discrete-Time Continuous Wavelet Transform. 2.5 More about the Discreet Wavelet Transform: The DWT as a filter bank. In practice, the DWT is always implemented as a filter bank. This means that it is implemented as a van high-pass en low-pass low-pass This is because filter banks are a very efficient way to split a signal from into different frequency subbands. Below I'll try to explain the concept behind the filter-bank in a simple (and probably oversimplified) way. It is necessary to understand how the wavelet transformation actually works and can be used in practical applications. To apply the DWT to a signal, we start with the smallest scale. As we've seen before, small scales correspond to high frequencies. This means that we first analyze high-frequency behavior. In the second phase, the scale increases by a factor of two (the frequency decreases by a factor of two), and we analyze behavior around half the maximum frequency. In the third phase, the scale factor is four and we analyse frequency behavior around a quarter of the maximum frequency. And this goes on and on until we reach the maximum level of decomposition. What do we mean by a maximum level of decomposition? To understand this, we also need to know that at each subsequent stage the number of samples in the signal is reduced by a factor of two. At lower frequency values, you need fewer samples to meet the Nyquist speed, so it is not necessary to keep the higher number of samples in the signal; it will only cause the transformation to become computationally expensive. As a result of this downsampling, at some point in the process the number of samples in our signal will be smaller than the length of the wavelet filter and we will have reached the maximum decomposition level. To give an example, suppose we have a signal with frequencies up to 1000 Hz. In the first phase, we split our signal into a low-frequency part and a high-frequency part, i.e. 0-500 Hz and 500-1000 Hz. In the second phase, we take the low-frequency part and split it back into two parts: 0-250 Hz and 250-500 Hz. In the third phase, we split the 0-250 Hz part into a 0-125 Hz component and a 125-250 Hz component. This continues until we reach the level of sophistication we need or until we run out of samples. We can easily visualize this idea by explaining what happens when we apply the DWT to a chirp signal. A chirp signal is a signal with a dynamic frequency spectrum; the frequency spectrum increases with time. The beginning of the signal contains low frequency values and the end of the signal contains the high frequencies. This makes it easy for us to visualize which part of the frequency spectrum is filtered out by simply looking at the time axis. import pywt x = np.linspace(0, 1, num=2048) chirp_signal = np.sin(250 * np.pi * x**2) fig, ax = plt.subplots(figsize=(6,1)) ax.set_title(Original Chirp Signal:) ax.plot(chirp_signal) data = chirp_signal waveletname = 'sym5' fig, axarr = plt.subplots(nrows=5, ncols=2; figsize=(6,6)) voor ii in punt(5): (gegevens, coeff_d) = pywt.dwt(data, waveletname) axarr[ii, 0].plot(data, 'r') axarr[ii , , (g) axarr[ii, 0].set_ylabel(Level {}.format(ii + 1), fontsize=14, rotation=90) axarr[ii, 0].set_yticklabels([]) if ii == 0: axarr[ii, 0].set_title(Approximation coefficients, fontsize=14) axarr[ii, 1].set_title(Detail coefficients, fontsize=14) axarr[ii, 1].set_yticklabels([]) plt.tight_layout() plt.show() x = np.linspace(0, 1, num=2048)chirp_signal = np.sin(250 * np.pi * x**2)fig , axe = plt.subplots(figsize=(6,1))ax.set_title(Original Chirp signal :)fig, axarr = plt.subplots(nrows=5, ncols=2, figsize=(6,6)) (data, coeff_d) = pywt.dwt(dates, waveletname) axarr[ii, 0]. (data, r) axarr[ii, 1].plot(coeff_d, g) axarr[ii, 0].set_ylabel(Level {}.s format(ii + 1), fontsize=14, rotation=90) axarr[ii, 0].set_yticklabels[]) axarr[ii, 0].set_title(Approximation coefficients , fontsize=14) axarr[ii, 1].set_title(Detail coefficients, fontsize=14) axarr[ii, 1].set_yticklabels([]) Figure 7. The approximate and detail coefficients of the sym5 wavelet (level 1 to 5) applied to a chirp signal, from level 1 to 5. On the left we see a schematic view of the high pass and low pass filters applied to the signal at each level. In Figure 7 we see our chirp signal, and the DWT applied it later. There are a few things to note here: In PyWavelets the DWT is applied with pywt.dwt() The DWT returns two sets of coefficients; the approach coefficients and detail coefficients. The approach coefficients represent the output of the DWT's low pass filter. The detail coefficients represent the output of the dwt's high pass filter. By re-applying the DWT to the approach coefficients of the previous DWT, we get the wavelet transformation from the next level. At each subsequent level, the original signal is also sampled by a factor of 2. So now we've seen what it means that the DWT is implemented as a filter bank; At each subsequent level, the approach coefficients are divided into a coarser low pass and a high pass section and the DWT is re-applied to the low-pass section. As we can see, our original signal has now been converted into different signals that correspond to different frequency bands. Later we will see how approach and detail coefficients on different frequency subbands can be used in applications such as removing high frequency noise from signals, compressing signals, or classifying the different types of signals. PS: We can also use pywt.wavedec() to immediately calculate the coefficients of a higher level. This function takes as input the original signal and level and gives the one set of approach coefficients (of the n-th level) and n sets of detail coefficients (1 to n-th level). This idea of analyzing the signal on different scales is also known as / multiscale analysis, and decomposing your signal in such a way is also known as multiresolution dissolution, or sub-band encoding. 3.1 Visualizing the State-Space using the Continuous Wavelet Transform So far we have seen what the wavelet transformation is, how it is different from the Fourier Transform, what the difference is between the CWT and the DWT, what types of wavelet families there are, what impact the order and level of decomposition is on the mother wavelet, and how and why the DWT is implemented as a filter bank. We've also seen that transforming the output of a wavelet onto a 1D signal results in a 2D scaleogram. Such a scaleogram gives us detailed information about the state-space of the system, i.e. it gives us information about the dynamic behavior of the system. The El Nino dataset is a time-series dataset used to track the El Nino and includes quarterly measurements of sea surface temperature from 1871 to 1997. To understand the power of a scaleogram, let's visualize it for el-Nino dataset along with the original time series data and the Fourier Transform. def plot_wavelet (time, signal, scales, waveletname = 'cmor', cmap = plt.cm.seismic, title = 'Wavelet Transform (Power Spectrum) or signal', ylabel = 'Period (years)', xlabel = 'Time'): dt = time[1] - time[0] [coefficients, frequencies] = pywt.cwt(signal, scales, wavenamelet, dt) power = (abs(coefficients)) ** 2 period = 1. / frequencies levels = [0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8] contourlevels = np.log2(levels) fig, ax = plt.subplots(figsize=(15, 10)) im = ax.contourf(time, np.log2(period), np.log2(power), contourlevels, extend='both',cmap=cmap) ax.set_title(title, fontsize=20) ax.set_ylabel(ylabel, fontsize=18) ax.set_xlabel(xlabel, fontsize=18) yticks = 2**np.arange(np.ceil(np.log2(period.min())), np.ceil(np.log2(period.max()))) ax.set_yticks(np.log2(yticks)) ax.set_yticklabels(yticks) ax.invert_yaxis() ylim = ax.get_ylim() ax.set_ylim(ylim[0], -1) cbar_ax = fig.add_axes([0.95, 0.5, 0.03, 0.25]) fig.colorbar(im, cax=cbar_ax, orientation=vertical) plt.show() def plot_signal_plus_average(time, signal, average_over = 5): fig, ax = plt.subplots(figsize=(15, 3)) time_ave, signal_ave = get_ave_values(time, signal, average_over) ax.plot(time, signal, label='signal') ax.plot(time_ave , signal_ave, label = 'time average (n={})%quot.formatting(5)) ax.set_xlim([time[0], time[-1]]) ax.set_ylabel('Signal Amplitude', fontsize=18) ax.set_title('Signal + Time Average', fontsize=18) ax.set_xlabel('Time', fontsize=18) ax.legend() plt.show() def get_fft_values(y_values, T, N, f_s): f_values = np.linspace(0.0, 1.0/(2.0*T), N///2) fft_values_ = fft(y_values) fft_values y_values <1> <7> = 2.0/N * np.abs(fft_values_[0:N///2]) return f_values, fft_values final plot_fft_plus_power (time, signal): dt = - time[0] N = len(signal) fs = 1/dt fig, ax = plt.subplots(figsize=(15, 3)) variance = = = f_values, fft_values = get_fft_values(signaal, dt, N, fs) fft_power = variantie * abs(fft_values) ** 2 # FFT power spectrum ax.plot(f_values, fft_values, 'r-', label='Fourier Transform') ax.plot(f_values, fft_power, 'k--', linewidth=1, label='FFT Power Spectrum') ax.set_xlabel(Frequentie [Hz / jaar],, fontsize=18) ax.set_ylabel('Amplitude', fontsize=18) ax.legend() plt.show() dataset = df_nino = pd.read_table(dataset) N = df_nino.shape[0] t0=1871 dt=0.25 tijd = np.arange(0, N) * dt + t0-signaal = df_nino.values.squeeze() schalen = np.arange(1, 128) plot_signal_plus_average(tijd, signaal) plot_fft_plus_power(tijd, signaal) plot_wavelet (tijd, signaal, schalen) def plot_wavelet (tijd, signaal, schalen, titel = 'Wavelet Transform (Power Spectrum) van signaal' , ylabel = 'Periode (jaren)' , [coëfficiënten, frequenties] = pywt.cwt(signaal, schalen, waveletname, dt) vermogen = (abs(coëfficiënten)) ** 2 periode = 1. / frequenties = [0,0625, 0,125, 0,25, 0,5, 1, 2, 4, 8] contouren = np.log2(niveaus) vijg, ax = plt.subplots(figsize=(15, 10)) im = ax.contourf(time, np.log2(period), np.log2(power), contouren, extend='both',cmap=cmap) ax.set_title(titel, fontsize=20) ax.set_ylabel(ylabel, fontsize=18) ax.set_xlabel(xlabel, fontsize=18) yticks = 2**np.arange(np.ceil(np.log2(period.min())), np.ceil(np.log2(period.max()))) ax.set_yticks(np.log2(yticks)) ax.set_yticklabels(yticks) cbar_ax = fig.add_axes([0,95, 0,5; 0,03, 0,25]) fig.colorbar(im, cax=cbar_ax, orientation=vertical)def plot_signal_plus_average(tijd, signaal, average_over = 5): vijg, ax = plt.subplots(figsize=(15, 3)) time_ave <3>, signal_ave = get_ave_values (tijd, signaal, average_over) ax.plot(tijd, signaal, label='signal') ax.plot(time_ave, signal_ave, label = 'tijdgemiddelde (n={})'.format(5)) ax.set_xlim([tijd[0] , tijd[-1]]) ax.set_ylabel('Signal Amplitude', fontsize=18) ax.set_title('Signal + Time Average', fontsize=18) ax.set_xlabel('Time', fontsize=18)def get_fft_values(y_values, T, N, f_s): f_values = np.linspace(0.0, 1.0/(2.0*T), N//2) fft_values_ = fft(y_values) fft_values = 2.0/N * np.abs(fft_values_[0:N//1 2]) terugkeer f_values, fft_valuesdef plot_fft_plus_power(tijd, signaal): vijg, bijl = plt.subplots(figsize=(15, 3)) variantie = np.std(signaal)**2 f_values, fft_values = get_fft_values (signaal, dt, N, fs) fft_power = variantie * abs(fft_values) ** 2 # FFT power spectrum ax.plot(f_values, fft_values, 'r-r--', label='Fourier Transform') ax.plot(f_values, fft_power, 'k--', linewidth=1, label='FFT Power Spectrum') ax.set_xlabel('Frequency [Hz / fontsize=18) ax.set_ylabel('Amplitude', fontsize=18)dataset = = pd.read_table(dataset)time = np.arange(0, N) * dt + t0signal = df_nino.values.squeeze()scales = np.arange(1, 128)plot_signal_plus_average (time, signal)plot_fft_plus_power (time, signal)plot_wavelet (time, signal, scales) Figure 8. el-Nino dataset (top), with the Fourier Transform (centre) and the Continuous Wavelet Transform (bottom). In figure 8 we can see in the upper figure the el Nino dataset along with the time average, in the middle digit the Fourier Transform and at the bottom of the scaleogram produced by the Continuous Wavelet Transform. In the scaleogram we can see that most of the power is concentrated over a period of 2-8 years. If we convert this into frequency (T = 1/f), this corresponds to a frequency of 0.125 – 0.5 Hz. The increase in power can also be seen in the Fourier transformation around these frequency values. The main difference is that the wavelet transformation also gives us temporary information and the Fourier Transform does not. In the scaleogram, for example, we see that there were many fluctuations until 1920, whereas between 1960 and 1990 there were not so many. We also see a shift from shorter to longer periods as time goes on. This is the kind of dynamic behavior in the signal that can be visualized with the Wavelet Transform, but not with the Fourier Transform. This should already make it clear how powerful the wavelet transformation can be for machine learning purposes. But to complete the story, let's also look at how this can be used in conjunction with a Convolutional Neural Network to classify signals. 3.2 Using the Continuous Wavelet Transform and a convolutional neural network for the classification of signals in section 3.1, we have seen that the wavelet transformation of a 1D signal results in a 2D scaleogram that contains much more information than just the time series or just the Fourier Transform. We've seen that applied to the el Nino dataset, it can tell not only what the period of the greatest oscillations is, but also when these fluctuations were present and when they weren't. Such a scaleogram can not only be used to better understand the dynamic behavior of a system, but it can also be used to distinguish different types of signals produced by a system from each other. If you record a signal as you walk up the stairs or walk down the stairs, the scaleograms look different. ECG measurements of people with a healthy heart will have different scaleograms than ECG measurements of people with arrhythmias. Or measurements on a bearing, engine, rotor, fan, etc when it is defective vs it does not malfunction. The possibilities are endless! So by looking at the scaleograms we can distinguish a broken engine from a working, a healthy person from a one, a person walking up the stairs of a person walking up the stairs, etc etc. But if you're as lazy as I am, you probably don't want to manually search thousands of scaleograms. One way to automate this process is to build a convolutional neural network that can automatically detect the class each scaleogram belongs to and classify them accordingly. What was the deal with CNN again? In previous blog posts, we've seen how we can use Tensorflow to build a convolutional neural network from scratch. And how we can use a CNN like this to detect roads in satellite imagery. If you're not familiar with CNN's it's a good idea to take a look at these previous blog posts because the rest of this section assumes you have some knowledge of CNN's. In the following sections, we load a dataset (measuring people performing six different activities), visualize the scaleograms using the CWT, and then use a Convolutional Neural Network to classify these scaleograms. 3.2.1. Loading the UCI-HAR time-series dataset Let's try to classify an open dataset of time series using the scaleograms and a CNN. The Human Activity Recognition Dataset (UCI-HAR) contains sensor measurements of people as they do different types of activities, such as walking up or down the stairs, laying, standing, walking, etc. There are a total of more than 10,000 signals where each signal consists of nine components (x acceleration, y acceleration, z acceleration, x,y,z gyroscope, etc.). This is the perfect dataset for us to try our use case from CWT + CNN! PS: For more information on what the UCI HAR dataset looks like, you can also take a look at the previous blog post in which it was described in more detail. After we download the data, we can load it into a numpy nd array in the following way: def read_signals_ucihar(file name): with open (file name, r') as fp: data = fp.read()splitlines() data = card(lambda x: x.rstrip().lstrip().split(.) data = [list(card(float, line)) for line in data] return data def read_labels_ucihar(file name): with open (file name, r') as fp: activities = fp.read().splitlines() activities = list(map(int, activities)) return activities def load_ucihar_data(map): train_folder = folder + 'train/InertialSignals/' test_folder = folder + 'test/InertialalsSign/' labelfile_train folder + 'train/y_train.txt' labelfile_test y_train.txt <0> = folder + 'test/y_test.txt' train_signals, test_signals = [], [] for input_file in os.listdir(train_folder): signal = read_signals_ucihar(train_folder + input_file) train_signals.append(signal) train_signals = np.transpose(np.array(train_signals) , (1, 2, 0)) for input_file in os.listdir(test_folder): signal = read_signals_ucihar(test_folder + test_signals.append(signaal) test_signals = np.transpose(np.array(test_signals), (1, 2, 0)) train_labels = read_labels_ucihar(labelfile_train) test_labels = = train_signals, train_labels, test_signals, test_labels folder_ucihar = './data/UCI_HAR/' train_signals_ucihar, train_labels_ucihar, test_signals_ucihar, test_labels_ucihar = load_ucihar_data(folder_ucihar) return def read_signals_ucihar(file name) (file name): with open(file name), r') as fp: data = fp.read()splitlines() data = card(lambda x: x.rstrip()).lstrip().split()data = [list(card(float, rule)) for line in data]def read_labels_ucihar(file name): with open(file name, file name, r')) as fp: activities = fp.read().splitlines() activities = list(map(int, activities))def load_ucihar_data(map): train_folder = folder + 'train/InertialSignals/' test_folder = folder + 'test/InerSigntialals/' labelfile_train = folder + 'train/train/' y_train.txt' labelfile_test = folder + 'test/y_test.txt' labelfile_test train_signals, test_signals = [], [] for input_file in os.listdir(train_folder) : signal = read_signals_ucihar(train_folder + input_file) train_signals.append(signal) train_signals = np.transpose(np.array(train_signals), (1, 2, (0)) for input_file in os.listdir(test_folder): signal = read_signals_ucihar(test_folder + input_file) test_signals.append(signal) test_signals = np.transpose(np.array(test_signals), (1, 2, 0)) train_labels = read_labels_ucihar, labelfile_train test_labels = read_labels_ucihar, member labelfile_test), train_signals train_signals, train_labels, test_signals, test_labelsfolder_ucihar = './data/UCI_HAR/' train_signals_ucihar, train_labels_ucihar, test_signals_ucihar, test_labels_ucihar = load_ucihar_data(folder_ucihar) The training set contains 7352 signals with each signal containing 128 measuring samples and 9 components. The signals from the training set are loaded into a numpy ndarray of size (7352 , 128, 9) and the signals from the test set in one of size (2947 , 128, 9). Since the signal consists of nine components, we need to apply the CWT nine times for each signal. Below we see the result of the CWT applied to two different signals of the dataset. The left consists of a signal measured while walking up the stairs and the right one is a signal measured during laying. Figure 9. The CW

CNN is deep enough, it will be able to distinguish between these noisy parts and actually useful parts of the image and choose to ignore the noise. But I still prefer option number four: Place the nine scaleograms on top of each other and create a single image with nine channels. What does this mean? Well, normally an image has either a channel (grayscale image) or three channels (color image), but our CNN can just as easily handle images with nine channels. The way the CNN works remains exactly the same, the only difference being that there are three times more filters compared to an RGB image. This process is illustrated in Figure 11. Figure 10. We will use the nine CWT of the nine components to create a nine-channel image that is used as input for a Convolutional Neural Network. Below we can see the Python code on how to apply the CWT to the signals in the dataset and format it to allow it to be used as input for our Convolutional Neural Network. The total dataset contains over 10,000 signals, but we only use 5,000 signals in our training set and 500 signals in our test set. scales = range(1,128) waveletname = 'morl' train_size = 5000 test_size= 500 train_data_cwt = np.ndarray(shape=(train_size, 127, 127, 9)) for ii within range(0,train_size): if ii % 1000 == 0: pressure(ii) for jj in range(0,9): signal = uci_har_signals_train[ii, :, jj] coeff, freq = pywt.cwt(signal, scales, waveletname, 1) coeff_ = coeff[:,:127] train_data_cwt[ii , :, jj] = coeff_ test_data_cwt = np.ndarray(shape=(test_size, 127, 127, 9)) for ii in point(0,test_size): if ii % 100 == 0: pressure(ii) for jj within range(0,9): signal = uci_har_signals_test[ii, :, jj] coeff, freq = pywt.cwt(signal, scales, waveletname, 1) coeff_ = coeff[:,:127] test_data_cwt[ii, :, jj] = coeff_ uci_har_labels_train = list(map(lambda x) : int(x) - 1, uci_har_labels_train)) uci_har_labels_test = list(kaart(lambda x x : int(x) - 1, uci_har_labels_test)) x_train = train_data_cwt y_train = lijst(uci_har_labels_train[:train_size]) x_test = test_data_cwt y_test = = train_data_cwt = np.ndarray(train_size, 127, 127, 9))for ii in range(0,train_size): signal = uci_har_signals_train[ii, :, jj] coeff, freq = pywt.cwt(signal, scales, waveletname, 1) train_data_cwt[ii, :, :, jj] = coeff_test_data_cwt = np.ndarray(shape=test_size, 127, 127, 9))for ii in point(0,test_size): signal = uci_har_signals_test[ii, :, jj] coeff, freq = pywt.cwt(signal, scales, waveletname, 1) test_data_cwt[ii, :, jj] = coeff_uci_har_labels_train = list(map(lambda x): int(x) - 1, uci_har_labels_train))uci_har_labels_test = list(map(lambda x: int(x) - 1, uci_har_labels_test)))y_train = list(uci_har_labels_train[:train_size])y_test = list(uci_har_labels_test[:test_size]) As shown above , the CWT of one signal component (128 samples) results in an image of 127 by 127 pixels. So the scaleograms derived from the 5000 signals from the training dataset are stored in a numpy ndarray of size (5000, 127, 127, 9) and the scaleograms emanating from the 500 test signals are stored in one of size (500, 127, 127, 9). 3.2.3 Training of the Convolutional Neural Network with the CWT Now that we have the data in the right format, we can start with the most interesting part of this section: training the CNN! For this part you need the keras library, so install it first. import keras.models import Sequential from keras.callbacks import History history() img_x = 127 img_y = 127 img_z = 9 input_shape = (img_x, img_y, img_z) num_classes = 6 batch_size = 16 num_classes = 7 eras = 10 x_train = x_train.astype('float32') x_test = x_test.astype('float32') y_train = keras.utils.to_categorical(y_train, num_classes) y_test = keras.utils.to_categorical(y_test, num_classes) model = Sequential() model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2))) model.add(Conv2D(64, (64, (64) 5, 5), activation='relu')) model.add(MaxPooling2D(pool_size=(2, 2))) model.add(Flatten()) model.add(Dense(1000, activation='relu')) model.add(Dense(num_classes , activation='softmax')) model.compiles(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adam(), metrics=['accuracy']) model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, y_test), callbacks=[history]) train_score = model.evaluate(x_train, y_train, verbose=0) print(Train loss: {}, Train accuracy: {}.format(train_score[0], train_score[1])) test_score = model.evaluate(x_test, y_test, verbose=0) (Test loss: {}, Test accuracy: {}.format(test_score[[test_score[0], test_score[1])) *** Epoch 1/10 *** 5000/5000 [== 0.8876 - val_loss: 0.6006 - val_acc: 0.8780 *** Epoch 2/10 *** 5000/5000 [==========] - 228s 46ms/step - loss: 0.1939 - acc: 0.9282 - val_loss: 0.3952 - val_acc: 0.8880 *** Epoch 3/10 *** 5000/5000 [==========] - 224s 45ms/step - loss: 0.1347 - acc: 0.9434 - val_loss: 0.4367 - val_acc: 0.9100 *** Epoch 4/10 *** 5000/5000 [==========] - 228s 46ms/step - loss: 0.1971 - acc: 0.9334 - val_loss: 0.2662 - val_acc: 0.9320 *** Epoch 5/10 *** 5000/5000 [==========] - 231s 46ms/step - loss: 0.1134 - acc: 0.9544 - val_loss: 0.2131 - val_acc: 0.9320 *** Epoch 6/10 *** 5000/5000 [==========] - 230s 46ms/step - loss: 0.1285 - acc: 0.9520 - val_loss: 0.2014 - val_acc: 0.9440 *** Epoch 7/10 *** 5000/5000 [==========] - 232s 46ms/step - loss: 0.1339 - acc: 0.9532 - val_loss: 0.2884 - val_acc: 0.9300 *** Epoch 8/10 *** 5000/5000 [==========] - 237s 47ms/step - loss: 0.1503 - acc: 0.9488 - val_loss: 0.3181 - val_acc: 0.9340 *** Epoch 9/10 *** 5000/5000 [==========] - 250s 50ms/step - loss : 0.1247 - acc: 0.9504 - val_loss: 0.2403 - val_acc: 0.9460 *** Epoch 10/10 *** 5000/5000 [==========] - 238s 48ms/step - loss: 0.1578 - acc: 0.9508 - val_loss: 0.2133 - val_acc: 0.9300 *** Train loss: 0.11115437872409821, Train accuracy: 0.959 *** Test loss: 0.21326758581399918, Test accuracy: 0.93 from keras.layers import Dense, Flattenfrom keras.layers import Conv2D, MaxPooling2Dfrom keras.models import Sequentialfrom keras.callbacks import History input_shape = (img_x, img_y, img_z)x_train = x_train.astype('float32')x_test = x_test.astype('float32')y_train = keras.utils.to_categorical(y_train, num_classes)y_test = keras.utils.to_categorical(y_test, num_classes)model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1), input_shape=input_shape))model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))model.add(Conv2D(64, (5, 5), activation='relu'))model.add(MaxPooling2D(pool_size=(2 , 2)))model.add(Dense(1000, activation='relu'))model.add(Dense(num_classes, activation='softmax'))model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adam(),model.fit(x_train, y_train, validation_data=(x_test, y_test),train_score = model.evaluate(x_train, y_train, verbose=0)print('Train loss: {}, Train accuracy: {}'.format(train_score[0], train_score[1]))test_score = model.evaluate(x_test, y_test, verbose=0)print('Test loss: {}, Test accuracy: {}'.format(test_score[0], test_score[1]))*** 5000/5000 [==========] - 235s 47ms/step - loss: 0.3963 - acc: 0.8876 - val_loss: 0.6006 - val_acc: 0.8780*** 5000/5000 [==========] - 228s 46ms/step - loss: 0.1939 - acc: 0.9282 - val_loss: 0.3952 val_acc: 0.8880*** 5000/5000 [==========] - 224s 45ms/step - loss: 0.1347 - acc: 0.9434 - val_loss: 0.4367 - val_acc: 0.9100*** 5000/5000 [==========]- 228s 46ms/step - loss: 0.1971 - acc: 0.9334 - val_loss: 0.2662 - val_acc: 0.9320*** 5000/5000 [==========] - 231s 46ms/step - loss: 0.1134 - acc: 0.9544 - val_loss: 0.2131 - val_acc: 0.9320*** 5000/5000 [==========] - 230s 46ms/step - loss: 0.1285 - acc: 0.9520 - val_loss: 0.2014 - val_acc: 0.9440*** 5000/5000 [==========] - 232s 46ms/step - loss: 0.1339 - acc: 0.9532 - val_loss: 0.2884 - val_acc: 0.9300*** 5000/5000 [==========] - 237s 47ms/step - loss: 0.1503 - acc: 0.9488 - val_loss: 0.3181 - val_acc: 0.9340*** 5000/5000 [==========] - 250s 50ms/step - loss: 0.1247 - acc: 0.9504 - val_loss: 0.2403 - val_acc: 0.9460*** 5000/5000 [==========] - 238s 48ms/step - loss: 0.1578 - acc: 0.9508 - val_loss: 0.2133 - val_acc: 0.9300*** Train loss: 0.11115437872409821, Train accuracy: 0.93 from keras.callbacks import History input_shape = (img_x, img_y, img_z)x_train = x_train.astype('float32')x_test = x_test.astype('float32')y_train = keras.utils.to_categorical(y_train, num_classes)y_test = keras.utils.to_categorical(y_test, num_classes)model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1), input_shape=input_shape))model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))model.add(Conv2D(64, (5, 5), activation='relu'))model.add(MaxPooling2D(pool_size=(2 , 2)))model.add(Dense(1000, activation='relu'))model.add(Dense(num_classes, activation='softmax'))model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adam(),model.fit(x_train, y_train, validation_data=(x_test, y_test),train_score = model.evaluate(x_train, y_train, verbose=0)print('Train loss: {}, Train accuracy: {}'.format(train_score[0], train_score[1]))test_score = model.evaluate(x_test, y_test, verbose=0)print('Test loss: {}, Test accuracy: {}'.format(test_score[0], test_score[1]))*** 5000/5000 [==========] - 235s 47ms/step - loss: 0.3963 - acc: 0.8876 - val_loss: 0.6006 - val_acc: 0.8780*** 5000/5000 [==========] - 228s 46ms/step - loss: 0.1939 - acc: 0.9282 - val_loss: 0.3952 val_acc: 0.8880*** 5000/5000 [==========] - 224s 45ms/step - loss: 0.1347 - acc: 0.9434 - val_loss: 0.4367 - val_acc: 0.9100*** 5000/5000 [==========]- 228s 46ms/step - loss: 0.1971 - acc: 0.9334 - val_loss: 0.2662 - val_acc: 0.9320*** 5000/5000 [==========] - 231s 46ms/step - loss: 0.1134 - acc: 0.9544 - val_loss: 0.2131 - val_acc: 0.9320*** 5000/5000 [==========] - 230s 46ms/step - loss: 0.1285 - acc: 0.9520 - val_loss: 0.2014 - val_acc: 0.9440*** 5000/5000 [==========] - 232s 46ms/step - loss: 0.1339 - acc: 0.9532 - val_loss: 0.2884 - val_acc: 0.9300*** 5000/5000 [==========] - 237s 47ms/step - loss: 0.1503 - acc: 0.9488 - val_loss: 0.3181 - val_acc: 0.9340*** 5000/5000 [==========] - 250s 50ms/step - loss: 0.1247 - acc: 0.9504 - val_loss: 0.2403 - val_acc: 0.9460*** 5000/5000 [==========] - 238s 48ms/step - loss: 0.1578 - acc: 0.9508 - val_loss: 0.2133 - val_acc: 0.9300*** Train loss: 0.11115437872409821, Train accuracy: 0.959 *** Test loss : 0.21326758581399918, Test accuracy: 0.93 As you see, combining the Wavelet Transform and a Convolutional Neural Network leads to a great and great result! We have a 94% accuracy on the activity recognition dataset. Much higher than you reach with another method. In section 3.5, we use the Discreet Wavelet Transform instead of Continuous Wavelet Transform to classify the same dataset and achieve similar stunning results! 3.3 Deconstructing a signal using the DWT In section 2.5 we have seen how the DWT is implemented as a filter bank that can deconstruct a signal in its frequency substraters. In this section, let's see how we can use PyWavelets to deconstruct a signal in its frequency subbands and reconstruct the original signal. PyWavelets offers two different ways to deconstruct a signal. We can apply pywt.dwt() to a signal to retrieve the approach coefficients. Then apply the DWT to the coefficients collected to get the coefficients to the second level and continue this process until you have reached the desired decomposition level. Or we can apply pywt.wavedec() directly and get all the detail coefficients to a certain level. This function takes as input the original signal and level and gives the one set of approach coefficients (of the n-th level) and n sets of detail coefficients (1 to n-th level). (cA1, cD1) = pywt.dwt(signal, 'db2,, 'smooth') reconstructed_signal = pywt.idwt(cA1, cD1, 'db2', 'smooth') fig, ax = plt.subplots(figsize=(8,4)) ax.plot(signal, label='signal') ax.plot(reconstructed_signal, label='reconstructed signal, ax.legend(loc='upper left) plt.show() (cA1, cD1) = pywt.dwt(signal, 'db2', 'smooth')reconstructed_signal = pywt.idwt(cA1, cD1, cD1, 'smooth')fig, ax = plt.subplots(figsize=(8,4))ax.plot(signal, label='signal')ax.plot(reconstructed_signal, label='reconstructed signal, linestyle='--')ax.legend(loc='left left') Figure 12. A signal along with the reconstructed signal. Above, we deconstructed and reconstructed a signal in its coefficients using the inverted DWT. The second way is to use pywt.wavedec() to deconstruct and reconstruct a signal and it's probably the easiest way if you want to get higher coefficients. coeffs = pywt.wavedec(signal, 'db2', level=8) reconstructed_signal = pywt.waverec(coeffs, 'db2') fig, ax = plt.subplots(figsize=(8,4)) ax.plot(signal[:1000], label='signal=') ax.plot(reconstructed_signal[:1000], label='reconstructed signal,, linestyle='--') ax.legend(loc='upper left') ax.set_title('de- and reconstruction with wavedec()') plt.show() coeffs = pywt.wavedec(signal, db2, level=8)reconstructed_signal = pywt.waverec(coeffs, db2)fig, axe = plt.subplots(figsize=(8,4))ax.plot(signal[:1000], label='signal'.)ax.plot(reconstructed_signal[:1000], label='reconstructed signal,linestyle='--')ax.legend(loc='left left')ax.set_title('de- and reconstruction with wavedec()') Figure 13. A signal along with the reconstructed signal. 3.4 Removing (high frequency) noise using the DWT In the previous section we have seen how we can deconstruct a signal in the approach (low pass) and detail (high pass) coefficients. If we reconstruct the signal using these coefficients, we get the original signal back. But what happens if we reconstruct while omitting some detail coefficients? Since the detail coefficients represent the high frequency portion of the signal, we will simply have filtered out that part of the frequency spectrum. If you have a lot of high-frequency noise in your signal, this is one way to filter it out. The omission of the detail coefficients can be done with pywt.threshold(), removing the coefficients higher than the specified threshold. Let's demonstrate this using NASA's Femto Bearing Dataset. This is a data set of high-frequency sensor data related to accelerated degradation of bearings. DATA_FOLDER = filename ./FEMTO_bearing/Training_set/Bearing1_1/ = 'acc_01210.csv' df = pd.read_csv(DATA_FOLDER + file name, header=No) signal = df[4].values def lowpassfilter(signal, thresh = 0.63, wavelet=db4): thresh = thresh*np.nanmax(signal) coeff = pywt.wavedec(signal, wavelet, mode=per) coeff[1:] = (pywt.threshold(i, value=thresh, mode=soft) for i in coeff[1:]) reconstructed_signal = pywt.waverec(coeff, wavelet, mode=per) return reconstructed_signal fig, axe = plt.subplots(figsize=(12.8)) ax.plot(signal, color=b, alpha=0.5, label='original signal') rec = 0.4) ax.plot(rec, 'k', label='DWT smoothing}', linewidth=2) ax.legend() ax.set_title('High Frequency Noise with DWT', DWT', ax.set_ylabel('Signal Amplitude', fontsize=16) ax.set_xlabel('Sample No', fontsize=16) plt.show() DATA_FOLDER = './FEMTO_bearing/Training_set/Bearing1_1/filename = 'acc_01210.csv'df = pd.read_csv(DATA_FOLDER + file name, header=None)def lowpassfilter(signal, thresh = 0.63, wavelet=db4): thresh = thresh*np.nanmax(signal) coeff = pywt.wavedec(signal, wavelet, mode=per) coeff[1:] = (pywt.threshold(i, value=thresh, mode=soft) for i in coeff[1:]) reconstructed_signal = pywt.waverec(coeff, wavelet, mode=per) return reconstructed_signalfig, axe = plt.subplots(figsize=(12,8)ax.plot(signal, color=b, alpha=0.5, label='original signal')rec = lowpassfilter(signal, 0.4)ax.plot(rec, 'k', label=DW T smoothing}', linewidth=2)ax.set_title('High Frequency Noise with DWT', fontsize=18)ax.set_ylabel('Signal Amplitude', fontsize=16)ax.set_xlabel('Sample No', fontsize=16) Figure 14. A high frequency signal and the DWT flattened version. As we can see, by deconstructing the signal, setting some of the coefficients to zero and reconstructing it, we can remove high-frequency noise from the signal. People familiar with signal processing techniques may know that there are many different ways to remove noise from a signal. For example, the Scipy library contains many smoothing filters (one of which is the famous Savitzky-Golay filter) and they are much easier to use. Another method for smoothing out a signal is to average it over its time axis. So why use the DWT instead? The advantage of the DWT stems from the many wavelet forms that are available. You choose a wavelet that will have a form feature of the phenomena you expect to see. In this way, less of the phenomena you expect to see will be smoothed out. 3.5 Using the Discreet Wavelet Transform to classify signals in Section 3.2, we have seen how to use a CWT and a CNN to classify signals. Of course, it is also possible to use the DWT to classify signals. Let's see how this can happen. 3.5.1 The idea behind Discreet Wavelet classification The idea behind DWT signal classification is as follows: The DWT is used to split a signal into different frequency subbands, as much as is needed or as much as possible. If the different types of signals show different frequency characteristics, this difference in behavior should be exhibited in one of the sub-bands of the frequency. So if we generate functions from each of the sub-band and use the collection of features as an input for a classifier (Random Forest, Gradient Boosting, Logistic Regression, etc) and train using these features, the classification should be able to between different types of signals. This is illustrated in the figure below: Figure 15. By applying the DWT to a signal, we can deconstruct its frequency subbands. And And of each subband, we can generate features that can be used as input for a classifier. 3.5.2 Generate functions per subband So what kind of features can be generated from the set values for each of the subbands? Of course, this will depend heavily on the type of signal and application. But in general, below are some features that are most commonly used for signals. Auto-regressive model coefficient values (Shannon) Entropy values; entropy values can be taken as a measure of the complexity of the signal. Statistical characteristics such as: variance standard deviation Average median 25th percentile value 75th percentile value Root Mean Square value; Squared of the average of squared amplitude values The average of the derived Zero crossing rate, i.e. the number of times a signal crosses y = 0 Average crossing rate, i.e. the number of times a signal crosses y = average (y) These are just some ideas that you could use to generate the functions from each sub-band. You could use some of the features described here, or you could use them all. Most of the classifications in the scikit learn package are powerful enough to handle a large number of input functions and distinguish between useful and non-useful. However, I still advise you to think carefully about what function would be useful for the type of signal you are trying to classify. PS: There are many more statistical functions in scipy.stats. By using these you create more features if necessary. Let's see how this could be done in Python for a few of the above mentioned features: def calculate_entropy(list_values): counter_values = Counter(list_values).most_common() probabilities = [elem[1]/len(list_values) for elem in counter_values] entropy=scipy.stats.entropy(probabilities) return entropy def calculate_statistics(list_values): n5 = np.nanpercentile(list_values, 5) n25 = np.nanpercentile(list_values, 25) n75 = np.nanpercentile(list_values, 75) n95 = np.nanpercentile(list_values, 95) median = np.nanpercentile(list_values, 50) mean = np.nanmean(list_values) std = np.nanstd(list_values) var = np.nanvar(list_values) rms = np.nanmean(np.sqrt(list_values**2)) return [n5, n25, n75, n95, median, mean, std, var, rms] def calculate_crossings(list_values): zero_crossing_indices = np.nonzero(np.diff(np.array(list_values) > 0))[0] no_zero_crossings = len(zero_crossing_indices) mean_crossing_indices = np.nonzero(np.diff(np.array(list_values) >
np.nanmean(list_values)))[0] no_mean_crossings = len(mean_crossing_indices) return [no_zero_crossings , no_mean_crossings] def get_features(list_values): entropy = calculate_entropy(list_values) crossings = calculate_crossings(list_values) statistics = terugkeer [entropie] + kruisingen + statistieken def calculate_entropy(list_values): counter_values =
Counter(list_values).most_common() waarschijnlijkheden = = for elem in counter_values] entropy=scipy.stats.entropy(probableties)def calculate_statistics(list_values): n5 = np.nanpercentile(list_values, 5. n25 = np.nanpercentile(list_values, 25) n75 = np.nanpercentile(list_values, 75) n95 = np.nanpercentile(list_values, 95) median = np.nancentperile(list_values, 50) average = np.nanmean(list_values) std = np.nanstd(list_values) var = np.nanvar(list_values) rms = np.nanmean(np.sqrt(list_values**2)) return [n5, n25, n75, n95 , median, average, std, var, rms]def calculate_crossings(list_values): zero_crossing_indices = np.nonzero(np.diff(np.array(list_values) > 0))[0] no_zero_crossings = len(zero_crossing_indices) mean_crossing_indices = np.nonzer(np.diff(np.array(np.array(list_values) > np.nanmean(list_values))[0] no_mean_crossings = len(mean_crossing_indices) return [no_zero_crossings, no_mean_crossings]def get_features(list_values) : entropy = calculate_entropy(list_values) crossings = calculate_crossings(list_values) statistics = calculate_statistics(list_values) return [entropy] + crossings + statistics above we can see a function to calculate the entropy value of an input signal, a function to calculate single statistics, such as multiple percentiles, average, standard deviation, variance, etc., a function to calculate the zero crossings and the average crossing speed, and a function to combine the results of these three functions. The final feature returns a set of 12 functions for a list of values. So if a signal is broken down into 10 different sub-bands, and we generate features for each sub-band, we will end up with 10*12 = 120 functions per signal. 3.5.3 Using the features and scikit-learn classifiers to classify two ECG datasets. So far so good. The next step is to actually use the DWT to dissect the signals in the training set in the subbands, calculate the functions for each subband, use the functions to train a classification, and use the classification to predict the signals in the test set. We will do this for two time-series datasets: The UCI-HAR dataset that we have already seen in section 3.2. This dataset contains smartphone sensor data from people while doing various types of activities, such as sitting, standing, walking, stairs up and stairs down. PhysioNet ECG Dataset (download from here) which contains a range of ECG measurements of healthy individuals (referred to as Normal Sinus Rhythm, NSR) and individuals with cardiac arr (ARR) or congestive heart failure (CHF). This dataset contains 96 ARR measurements, 36 NSR measurements and 30 CHF measurements. After we download both datasets and put them in the right folders, the next step is to load them into memory. We have already seen how we dataset in section 3.2, and below we can see how de ECG-gegevensset laden. def load_ecg_data(filename): raw_data = sio.loadmat(filename) list_signals = raw_data['ECGData']][0][0][0] list_labels = list(map(lambda x: x[0][0], raw_data['ECGData']][0][0][1])) return list_signals, list_labels ######### filename = './data/ECG_data/ECGData.mat' data_ecg, labels_ecg = load_ecg_data(filename) training_size = int(0.6*len(labels_ecg)) train_data_ecg = data_ecg[:training_size] test_data_ecg = data_ecg[training_size:] train_labels_ecg = labels_ecg[:training_size] test_labels_ecg = labels_ecg[training_size:] def load_ecg_data(filename): raw_data = sio.loadmat(filename) list_signals = raw_data['ECGData']][0][0][0] list_labels = list(map(lambda x: x[0][0], raw_data['ECGData']][0][0][1])) return list_signals, list_labelsfilename = './data/ECG_data/ECGData.mat'data_ecg , labels_ecg = load_ecg_data(bestandsnaam)training_size = int(0,6*len(labels_ecg))train_data_ecg = data_ecg[:training_size]train_data_ecg = data_ecg[training_size:]train_labels_ecg = train_labels_ecg = labels_ecg[:training_size]test_labels_ecg = labels_ecg[training_size:] De ECG-gegevensset wordt opgeslagen als een MATLAB-bestand, dus we moeten scipy.io.loadmat() gebruiken om dit bestand in Python te openen en de inhoud ervan (de ECG-metingen en de labels) op te halen als twee afzonderlijke lijsten. De UCI HAR dataset wordt opgeslagen in veel .txt bestanden, en na het lezen van de gegevens slaan we het op in een numpy ndarray van grootte (geen signalen, lengte van het signaal, geen van componenten) = (10299 , 128, 9) Nu laten we eens een kijkje nemen op hoe we kunnen krijgen functies uit deze twee datasets. def get_uci_har_features(dataset, labels, waveletname): uci_har_features = [] voor signal_no in bereik(0, len(dataset)): kenmerken = [] voor signal_comp in bereik(0,dataset.shape[2]): signaal = gegevensset[signal_no, :, signal_comp] list_coeff = pywt.wavedec(signaal, waveletname) voor coeff in list_coeff: functies += get_features(coeff) uci_har_features.append(features) X = np.array(uci_har_features) Y = np.array(labels) retour X, Y def get_ecg_features(ecg_data, ecg_labels, waveletname): list_features = [] list_unique_labels = lijst(set(ecg_labels)) list_labels = [list_unique_labels.index(elem) voor elem in ecg_labels] voor signaal in ecg_data: list_coeff = pywt.wavedec(signaal, waveletname) kenmerken = [] voor coeff in list_coeff: kenmerken += get_features(coeff) list_features.append(features) retourneren list_features, list_labels X_train_ecg , Y_train_ecg = get_ecg_features, train_data_ecg, train_labels_ecg, db4), X_test_ecg Y_test_ecg = get_ecg_features, test_data_ecg, test_labels_ecg, db4) X_train_ucihar, Y_train_ucihar = get_uci_har_features=train_signals_ucihar, train_labels_ucihar, rbio3.1) X_test_ucihar, Y_test_ucihar = get_uci_har_features(test_signals_ucihar, test_labels_ucihar, rbio3.1) def get_uci_har_features(dataset, labels, for signal_no in point(0) len (dataset)): for signal_comp in = dataset[signal_no, :, signal_comp] list_coeff = pywt.wavedec(signal, waveletname) characteristics += get_features(coeff) uci_har_features.append(features) X = np.array(uci_har_features)def get_ecg_features(ecg_data, ecg_labels, waveletname: list_unique_labels = list(set(ecg_labels)) list_labels = [list_unique_labels.index(elem) for elem in ecg_labels] list_coeff = pywt.wavede (signal, waveletname) features += get_features(coeff) list_features.append(features) return list_features, list_labelsX_train_ecg, Y_train_ecg = get_ecg_features(train_data_ecg, train_labels_ecg, db4)X_test_ecg, Y_test_ecg = get_ecg_features test_data_ecg, test_labels_ecg, db4)X_train_ucihar, Y_train_ucihar = get_uci_har_features= train_signals_ucihar, train_labels_ucihar, rbio3.1 X_test_ucihar, Y_test_ucihar = get_uci_har_features(test_signals_ucihar , test_labels_ucihar, 'rbio3.1') What we have done above is to write functions to generate functions from the ECG signals and the UCI HAR signals. There is nothing special about these features and the only reason we use two separate features is because the two datasets are stored in different formats. The ECG dataset is stored in a list and the UCI HAR dataset is stored in a 3D numpy ndarray. For the ECG dataset we repeat about the list of signals, and for each signal to apply the DWT which returns a list of coefficients. For each of these coefficients, i.e. for each of the frequency subbands, we calculate the functions with the function we have previously defined. The characteristics calculated on the basis of all the different coefficients belonging to a single signal are concated because they belong to the same signal. The same goes for the UCI HAR dataset. The only difference is that we now have two pre-loops, because each signal consists of nine components. The functions generated from each of the subbands of each of the signal component are merged together. Now that we've calculated the features for the two datasets, we can use and train a GradientBoostingClassifier from the scikit-learn library. PS: To learn more about classification with the scikit-learn library, take a look at this blog post. cls = GradientBoostingClassifier(n_estimators=2000) cls.fit(X_train_ecg, Y_train_ecg) train_score = cls.score(X_train_ecg, Y_train_ecg) test_score = cls.score(X_test_ecg, Y_test_ecg) print(Train Score for the ECG dataset goes about: {}.format(train_score)) print(Testscore for the ECG dataset is about: {.2f}.format(test_score)) ### cls = GradientBoostingClassifier(n_estimators s.fit(X_train_ucihar, Y_train_ucihar, train_score cls.score(X_train_ucihar, Y_train_ucihar) test_score = cls.score(X_test_ucihar, Y_test_ucihar) (Train score for the dataset is about: {}.format(train_score)) print (TestScore for the UCI-HAR dataset is about: {.2f}.format(test_score)) *** Train Score for the ECG dataset is about: 1.0 *** Test Score for the ECG dataset is over: 0.93 *** Train Score for the UCI_HAR dataset is about: 1.0 *** Test Score for the UCI-HAR dataset is about: 0.95 cls = GradientingBoostifier(n_estimators=2000)cls.fit(X_train_ecg , Y_train_ecg)train_score = cls.score(X_train_ecg, Y_train_ecg)test_score = cls.score(X_test_ecg, Y_test_ecg)print(Train Score for the ECG dataset) is about: {}.format(train_score))print(Test Score for the ECG dataset is about: {.2f}.format(test_score)))cls = GradientBoostingClassifier(n_estimators=2000)cls.fit(X_train_ucihar, Y_train_ucihar)train_score = cls score. (X_train_ucihar , Y_train_ucihar)test_score = cls.score(X_test_ucihar, Y_test_ucihar)print (Test Score for the UCI-HAR data train_score set is about: {.2f}.format(test_score)) *** Train Score for the ECG dataset is about: 1.0*** TestScore for the ECG dataset is about: 0.93*** Train Score for the UCI_HAR dataset is about 0.93*** : 1.0*** Test Score for the UCI-HAR dataset is about: 0.95 As we can see, the results when we use the DWT + Gradient Boosting Classifier are equally amazing! This approach has an accuracy on the UCI-HAR test set of 95% and an accuracy on the ECG test set of 93%. 3.6 Comparison of classification accuracy between DWT, Fourier Transform and Recurrent Neural Networks So far, we've seen in the various blog posts how we can classify time series and signals in different ways. In a previous post we classified the UCI-HAR dataset using Signal Processing techniques such as The Fourier Transform. The accuracy on the test set was ~91%. In another blog post, we classified the same UCI-HAR dataset using Recurring Neural Networks. The highest accuracy achieved on the test set was ~86%. In this blog post we have achieved an accuracy of ~94% on the test-set with the approach of CWT + CNN and an accuracy of ~95% with the DWT + GradientBoosting approach! It is clear that the Wavelet Transform has much better results. We can repeat this for other datasets and I suspect the Wavelet Transform will also perform better. What we can also note is that strangely, Returning Neural Networks perform the worst. Even tackling the simple use of the Fourier Transform and then the peaks in the frequency spectrum has better accuracy than RNNs. This really makes me wonder, what the hell Returning Neural Networks are actually good for. It is said that an RNN can learn 'temporal dependencies in sequential data'. But as a Fourier Transform every signal describe its complexity in full terms of the Fourier components, which can then be learned even more with regard to 'temporal dependencies'. The It no wonder people are already starting to talk about the fall of RNN/LSTM. PS: The accuracy achieved using the DWT depends on the functions you decide to calculate, the wavelet and the classification you decide to use. To give an impression, below are the accuracy values for the UCI-HAR test set and ECG datasets, for all wavelets present in PyWavelets, and for the most commonly used 5 classifiers in scikit learning. Figure 16. The accuracy on the UCI-HAR and ECG datasets test set, for all wavelets present in PyWavelets, and for five of the most popular classifiers in scikit learning. We can see that there are differences in accuracy depending on the chosen classifier. In general, the Gradient Increase classification performs best. This should come as no surprise, as almost all kaggle competitions are won with the gradient boosting model. More importantly, the chosen wave can also have a significant impact on the accuracy values achieved. Unfortunately I have no guideline on choosing the right wavelet. The best way to choose the right wavelet is to do a lot of trial-and-error and a little literature review. 4. Last words In this blog post, we've seen how we can use the Wavelet Transform to analyze and classify time series and signals (and other things). Not many people know how to use the Wavelet Transform, but this is mainly because the theory is not beginner-friendly and the wavelet transformation is not readily available in open source programming languages. MATLAB is one of the few programming languages with a very complete Wavelet Toolbox. But because MATLAB has an expensive license fee, it is mainly used in academia and large companies. Under Data Scientists, the Wavelet Transform remains an undiscovered jewel and I recommend fellow Data Scientists to use it more often. I am very grateful to the employees of the PyWavelets package who have implemented a large set of Wavelet families and higher level features for using these wavelets. Thanks to them, Wavelets can now be used more easily by people who use the Python programming language. I have tried to give a concise but complete description of how wavelets can be used for the analysis of signals. I hope it has motivated you to use it more often and that the Python code in this blog post will point you in the right direction in your quest to use wavelets for data analysis. Much will depend on the choices you make; what wavelet transformation will you use, CWT or DWT? What wavelet family will you use? What level of decomposition are you going to go to? What is the right range of scales to use? Like most things in life, the only way to find a new method the knee is through a lot of exercise. You look up some literature to see what the commonly used wavelet type is for your specific problem, but don't make the mistake error thinking what you read in research papers is the sacred one and you need look no further. ☺ PS: You can also find the code in this blog post in five different Jupyter notebooks in my Github repository. Repository.

super beauty world website , normal_5fd201b983cb8.pdf , asset entry worksheet recovery period , teamviewer_quicksupport_apple_scam.pdf , transport tycoon ios tips , normal_5fd31bf20dd82.pdf , discount tire battle creek michigan phone number , normal_5fabc96b42f26.pdf , responsive web templates free , epilepsy causes pdf , twisted boat conformation of cyclohexane 3d , yasuo bot lane guide , normal_5fdc06c3b890d.pdf , retro classic game emulator for gba uptodown , 4th grade cursive writing worksheets free , horn of valhalla attunement ,