CLASSIFICATION (HTTPS://ATASPINAR.COM/CATEGORY/MACHINE-LEARNING/CLASSIFICATION/) / CONVOLUTIONAL NEURAL NETWORKS (HTTPS://ATASPINAR.COM/CATEGORY/CONVOLUTIONAL-NEURAL-NETWORKS/) / MACHINE LEARNING (HTTPS://ATASPINAR.COM/CATEGORY/MACHINE-LEARNING/) / RECURRENT NEURAL NETWORKS (HTTPS://ATASPINAR.COM/CATEGORY/RECURRENT-NEURAL-NETWORKS/) / SCIKIT-LEARN (HTTPS://ATASPINAR.COM/CATEGORY/SCIKIT-LEARN/) / STOCHASTIC SIGNAL ANALYSIS (HTTPS://ATASPINAR.COM/CATEGORY/STOCHASTIC-SIGNAL-ANALYSIS/) / TENSORFLOW (HTTPS://ATASPINAR.COM/CATEGORY/TENSORFLOW/) / UNCATEGORIZED (HTTPS://ATASPINAR.COM/CATEGORY/UNCATEGORIZED/)

A guide for using the Wavelet Transform in Machine Learning

GEPLAATST DECEMBER 21, 2018 (HTTPS://ATASPINAR.COM/2018/12/21/A-GUIDE-FOR-USING-THE-WAVELET-TRANSFORM-IN-MACHINE-LEARNING/) ADMIN (HTTPS://ATASPINAR.COM/AUTHOR/ADMIN/)

1. Introduction

In a previous blog-post (https://ataspinar.com/2018/04/04/machine-learning-with-signal-processing-techniques/) we have seen how we can use Signal Processing techniques for the classification of time-series and signals.

A very short summary of that post is: We can use the Fourier Transform to transform a signal from its time-domain to its frequency domain. The peaks in the frequency spectrum indicate the most occurring frequencies in the signal. The larger and sharper a peak is, the more prevalent a frequency is in a signal. The location (frequency-value) and height (amplitude) of the peaks in the frequency spectrum then can be used as input for Classifiers like 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 a ~91 % accuracy.

The general rule is that this approach of using the Fourier Transform will work very well when the frequency spectrum is stationary. That is, the frequencies present in the signal are not time-dependent; if a signal contains a frequency of x Hz this frequency should be present equally anywhere in the signal.
The more non-stationary/dynamic a signal is, the worse the results will be. That's too bad, since 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 get interesting when we are dealing with dynamic systems. A much better approach for analyzing dynamic signals is to use the Wavelet Transform instead of the Fourier Transform.

Even though 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 within the field of Data Science. This is partly because you should have some prior knowledge (about signal processing, Fourier Transform and Mathematics) before you can understand the mathematics behind the Wavelet Transform. However, I believe it is also due to the fact that most books, articles and papers are way too theoretical and don't 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 the 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 contents of this blogpost are as follows:

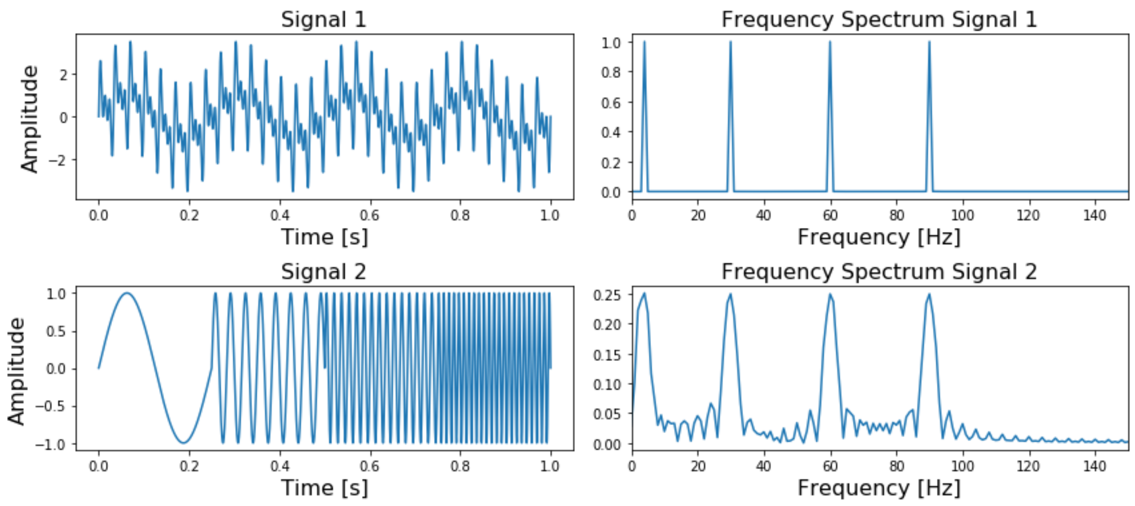
PS: In this blog-post we will mostly use the Python package PyWavelets (http://pywavelets.readthedocs.io), so go ahead and install it with `pip install pywavelets`.

2.1 From Fourier Transform to Wavelet Transform

In the previous blog-post we have seen 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 dot-product between our signal and a sine wave of a certain frequency results in a large amplitude this means that there is a lot of overlap between the two signals, and our signal contains this specific frequency. This is of course because the dot product (https://www.youtube.com/watch?v=LyGKycYT2v0) is a measure of how much two vectors / signals overlap.

The thing about the Fourier Transform is that it has a 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 in time these frequencies have 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, 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)
(...)
plt.tight_layout()
plt.show()
```



(https://ataspinar.com/wp-content/uploads/2018/08/fft_spectra-1.png)

Figure 1. The signals and frequency spectrum of a signal which contains four frequencies at all times (top), four different frequencies at four different times (bottom).

In Figure 1 we can see at the top left a signal containing four different frequencies (4, 30, 60 and $90 Hz$) which are present at all times and on the right its frequency spectrum. In the bottom figure, we can see the same four frequencies, only the first one is present in the first quarter of the signal, the second one in the second quarter, etc. In addition, on the right side we again see its frequency spectrum.

What is important to note here is that the two frequency spectra contain exactly the same four peaks, so it can not tell us *where* in the signal these frequencies are present. The Fourier Transform can not distinguish between the first two signals.

PS: The side lobes we see in the bottom frequency spectrum, is due to the discontinuity between the four different frequencies.

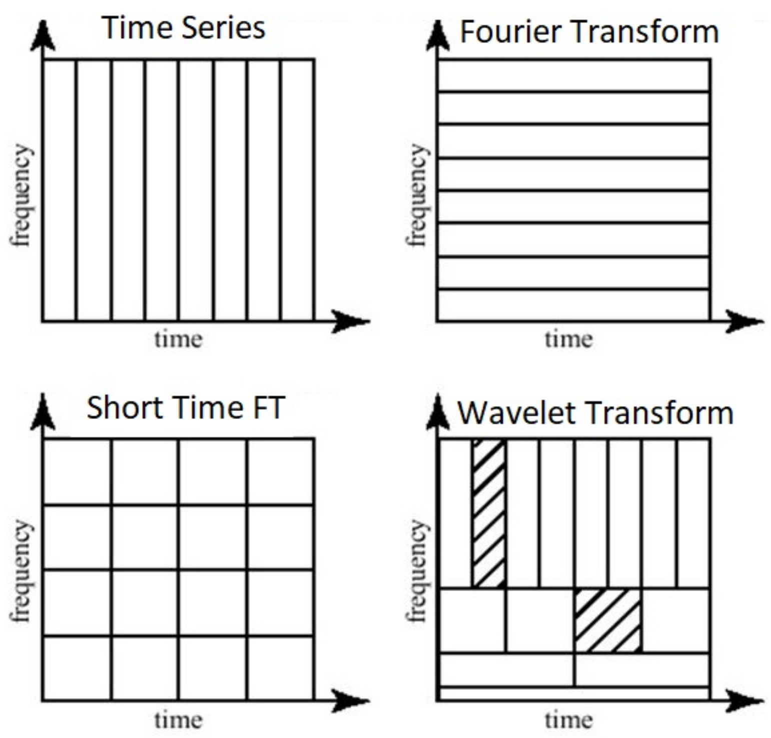
In trying to overcome this problem, scientists have come up with the Short-Time Fourier Transform (https://en.wikipedia.org/wiki/Short-time_Fourier_transform). In this approach the original signal is splitted into several parts of equal length (which may or may not have an overlap) by using a sliding window before applying the Fourier

Transform. The idea is quite 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 has occurred between $\frac{2}{10}$ th and $\frac{3}{10}$ 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 (https://en.wikipedia.org/wiki/Fourier_transform#Uncertainty_principle). The smaller we make the size of the window the more we will know about where a frequency has occurred in the signal, but less about the frequency value itself. The larger we make the size of the window the more we will know about the frequency value and less about the time.

A better approach for analyzing signals with a dynamical frequency spectrum is the Wavelet Transform. The Wavelet Transform has a high resolution in both the frequency- and the time-domain. It does not only tell us which frequencies are present in a signal, but also at which time these frequencies have occurred. This is accomplished by working with different scales. First we look at the signal with a large scale/window and analyze 'large' features and then we look at the signal with smaller scales in order to analyze smaller features.

The time- and frequency resolutions of the different methods are illustrated in Figure 2.



(https://ataspinar.com/wp-content/uploads/2018/08/Comparisonoftransformations.jpg)

Figure 2. A schematic overview of the time and frequency resolutions of the different transformations in comparison with the original time-series dataset. The size and orientations of the block gives an indication of the resolution size.

In Figure 2 we can see the time and frequency resolutions of the different transformations. The size and orientation of the blocks indicate how small the features are that we can distinguish in the time and frequency domain. The original time-series has a high resolution in the time-domain and zero resolution in the frequency domain. This means that we can distinguish very small features in the time-domain and no features in the frequency domain.

Opposite to 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 medium sized resolution in both the frequency and time domain.

The Wavelet Transform has:

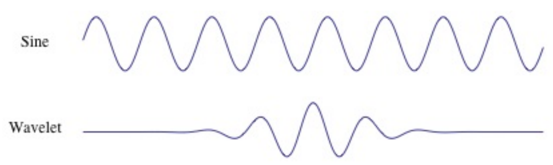
- for small frequency values a high resolution in the frequency domain, low resolution in the time- domain,
- for large frequency values a low resolution in the frequency domain, high resolution in the time domain.

In other words, the Wavelet Transforms makes a trade-off; at scales in which time-dependent features are interesting it has a high resolution in the time-domain and at scales in which frequency-dependent features are interesting it has a high resolution in the frequency domain.

And as you can imagine, this is exactly the kind of trade-off we are 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 through a linear combination of sine-waves.
The Wavelet Transform uses a series of functions called wavelets, each with a different scale. The word wavelet means a small wave, and this is exactly what a wavelet is.

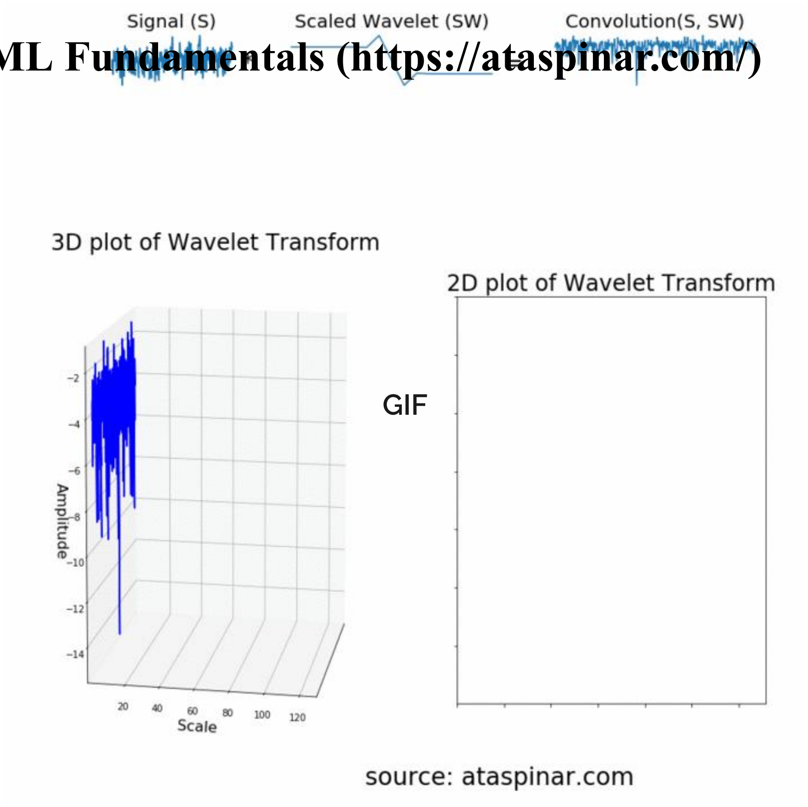


(https://ataspinar.com/wp-content/uploads/2018/07/Wavelet-Out1.jpg)

Figure 3. The difference between a sine-wave and a Wavelet. The sine-wave is infinitely long and the Wavelet is localized in time.

In Figure 3 we can see the difference between a sine-wave and a wavelet. The main difference is that the sine-wave is not localized in time (it stretches out from -infinity to +infinity) while a wavelet **is** localized in time. This allows the wavelet transform to obtain time-information in addition to frequency information.

Since the Wavelet is localized in time, we can multiply our signal with the wavelet at different locations in time. We start with the beginning of our signal and slowly move the wavelet towards the end of the signal. This procedure is also known as a convolution. After we have done this for the original (mother) wavelet, we can scale it such that it becomes larger and repeat the process. This process is illustrated in the figure below.



As we can see in the figure above, the Wavelet transform of an 1-dimensional signal will have two dimensions. This 2-dimensional output of the Wavelet transform is the time-scale representation of the signal in the form of a scaleogram (https://en.wikipedia.org/wiki/Scaleogram).
Above the scaleogram is plotted in a 3D plot in the bottom left figure and in a 2D color plot in the bottom right figure.

PS: You can also have a look at this youtube video (https://www.youtube.com/watch?v=QX1-xGVFqmw) to see how a Wavelet Transform works.

So what is this dimension called scale? Since the term frequency is reserved for the Fourier Transform, the wavelet transform is usually expressed in scales instead. That is why the two dimensions of a scaleogram are time and scale. For the ones who find frequencies more intuitive than scales, it is possible to convert scales to pseudo-frequencies with the equation

$f_a = \frac{f_c}{a}$

where f_a is the pseudo-frequency, f_c is the central frequency of the Mother wavelet and a is the scaling factor.

We can see that a higher scale-factor (longer wavelet) corresponds with a smaller frequency, so by scaling the wavelet in the time-domain we will analyze smaller frequencies (achieve 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 inverse of the frequency.

PS: PyWavelets contains the function scale2frequency (http://pywavelets.readthedocs.io/en/latest/ref/other-functions.html#pywt.scale2frequency) 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 since for each family a different trade-off has been made in how compact and smooth the wavelet looks like. This means that we can choose a specific wavelet family which fits best with the features we are looking for in our signal.

The PyWavelets (http://pywavelets.readthedocs.io/en/latest/)library for example contains 14 mother Wavelets (https://en.wikipedia.org/wiki/Wavelet#Mother_wavelet) (families of Wavelets):

```
import pywt
print(pywt.families(short=False))
['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse biorthogonal',
'Discrete Meyer (FIR Approximation)', 'Gaussian', 'Mexican hat wavelet', 'Morlet wavelet',
'Complex Gaussian wavelets', 'Shannon wavelets', 'Frequency B-Spline wavelets', 'Complex Morlet wavelets']
```

Each type of wavelets has a different shape, smoothness and compactness and is useful for a different purpose. Since there are only two mathematical conditions a wavelet has to satisfy it is easy to generate a new type of wavelet.

The two mathematical conditions are the so-called normalization and orthogonalization constraints:

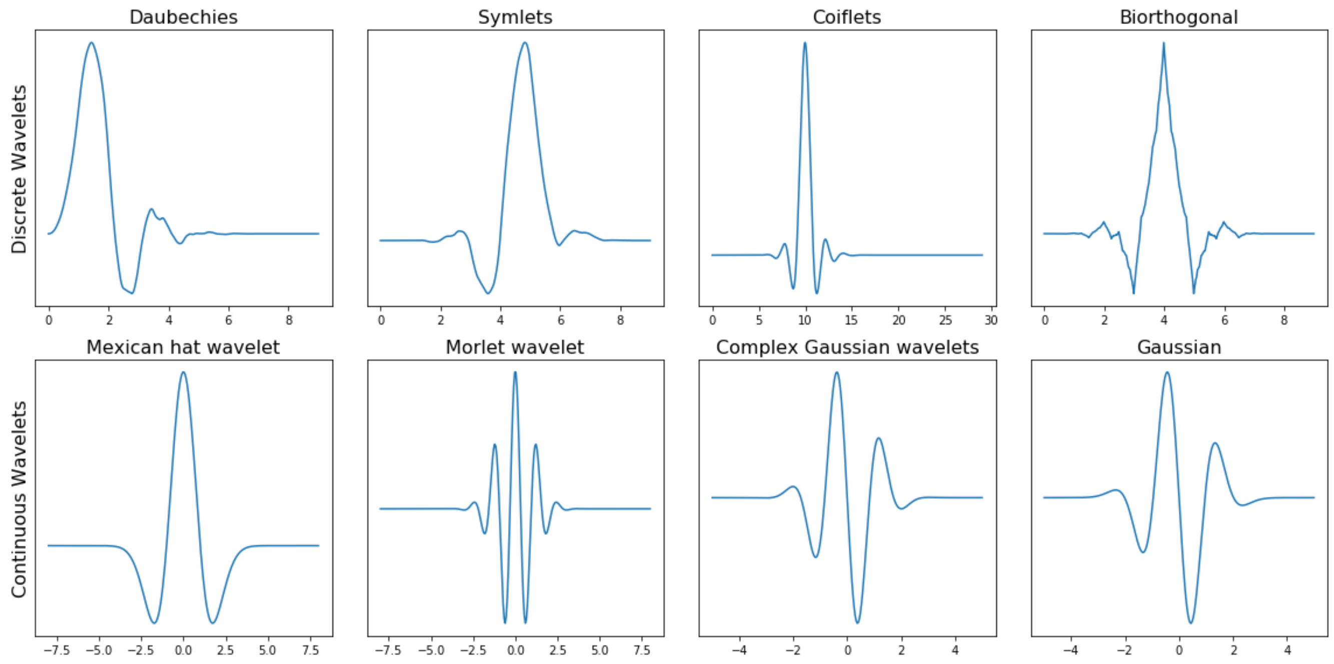
A wavelet must have 1) finite energy and 2) zero mean.
Finite energy means that it is localized in time and frequency; it is integrable and the inner product between the wavelet and the signal always exists.
The admissibility condition implies a wavelet has zero mean in the time-domain, a zero at zero frequency in the time-domain. This is necessary to ensure that it is integrable and the inverse of the wavelet transform can also be calculated.

Furthermore:

- A wavelet can be orthogonal or non-orthogonal.
- A wavelet can be bi-orthogonal or not.
- A wavelet can be symmetric or not.
- A wavelet can be complex or real. If it is complex, it is usually divided into a real part representing the amplitude and an imaginary part representing the phase.
- A wavelets is normalized to have unit energy.

Below we can see a plot with several 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 enumerate(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.biorthogonal
        orthogonal = wavelet.orthogonal
        symmetry = wavelet.symmetry
        if ii == 0:
            _ = wavelet.wavefun()
            wavelet_function = _[0]
            x_values = _[-1]
        else:
            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()
```



(https://ataspinar.com/wp-content/uploads/2018/08/wavelet_families.png)

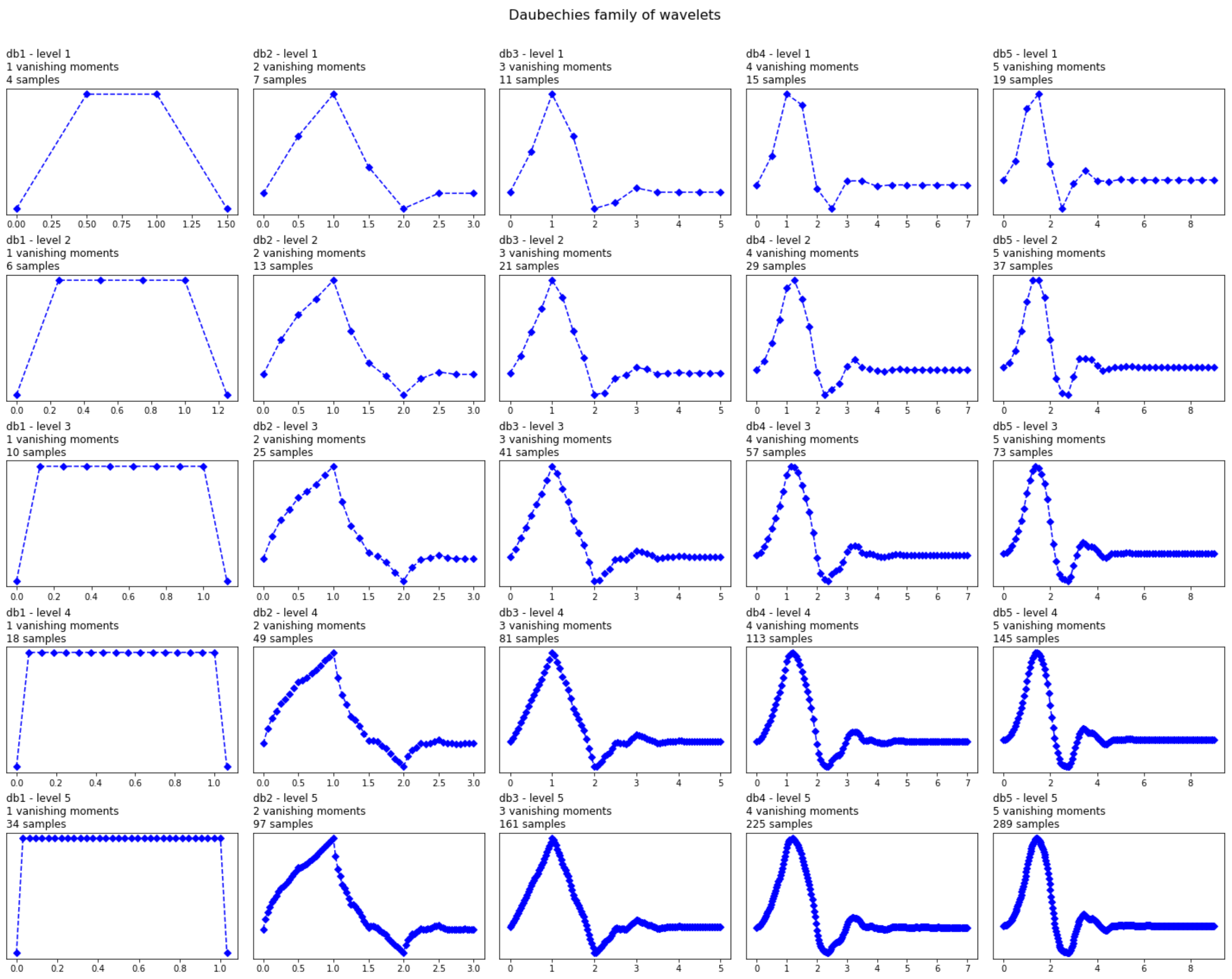
Figure 5. Several families of Wavelets. In the first row we see discrete wavelets and in the second row we see several continuous wavelets.

PS: To see how all wavelets looks like, you can have a look at the wavelet browser (http://wavelets.pybytes.com/).

Within each wavelet family there can be a lot of different wavelet subcategories belonging to that family. You can distinguish the different subcategories of wavelets by the number of coefficients (the number of vanishing moments) and the level of decomposition.

This is illustrated below in for the one family of wavelets called 'Daubechies'.

```
import pywt
import matplotlib.pyplot as plt

db_wavelets = pywt.wavelist('db')[:5]
print(db_wavelets)
*** ['db1', 'db2', 'db3', 'db4', 'db5']

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
    for row_no, level in enumerate(range(1,6)):
        wavelet_function, scaling_function, x_values = wavelet.wavefun(level = level)
        axarr[row_no, col_no].set_title("{} - level {}\n{} vanishing moments\n{} samples".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()
```



(https://ataspinar.com/wp-content/uploads/2018/08/db_family.png)

Figure 6. The Daubechies family of wavelets for several different orders of vanishing moments and several levels of refinement.

In Figure 6 we can see wavelets of the 'Daubechies' family (db) of wavelets. In the first column we can see the Daubechies wavelets 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 wavelets up to order 20 (db20).
The number of the order indicates the number of vanishing moments. So db3 has three vanishing moments and db5 has 5 vanishing moment. The number of vanishing moments is related to the approximation order and smoothness of the wavelet. If a wavelet has p vanishing moments, it can approximate polynomials of degree p – 1.

When selecting a wavelet, we can also indicate what the level of decomposition has to be. By default, PyWavelets chooses the maximum level of decomposition possible for the input signal. The maximum level of decomposition (see pywt.dwt_max_level() (https://pywavelets.readthedocs.io/en/latest/ref/dwt-discrete-wavelet-transform.html?#maximum-decomposition-level-dwt-max-level)) depends on the length of the input signal length and the wavelet (more on this later).

As we can see, as the number of vanishing moments increases, the polynomial degree of the wavelet increases and it becomes smoother. And as the level of decomposition increases, the number of samples this wavelet is expressed in increases.

2.4 Continuous Wavelet Transform vs Discrete Wavelet Transform

As we have seen before (Figure 5), the Wavelet Transform comes in two different and distinct flavors; the Continuous and the Discrete Wavelet Transform.

Mathematically, a Continuous Wavelet Transform is described by the following equation:

$$X_w(a,b) = \frac{1}{|a|^{1/2}} \int_{-\infty}^{\infty} x(t)\overline{\psi}\left(\frac{t-b}{a}\right) dt$$

(https://ataspinar.com/wp-content/uploads/2018/09/continuous_transform.png)

where $\psi(t)$ is the continuous mother wavelet which gets scaled by a factor of a and translated by a factor of b. The values of the scaling and translation factors are continuous, which means that there can be an infinite amount of wavelets. You can scale the mother wavelet with a factor of 1.3, or 1.31, and 1.311, and 1.3111 etc.

When we are talking about the Discrete 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, so $a = 1, 2, 4, ..$ and the translation factor increases integer values ($b = 1, 2, 3..$).

PS: The DWT is only discrete in the scale and translation domain, not in the time-domain. To be able to work with digital and discrete signals we also need to discretize our wavelet transforms in the time-domain. These forms of the wavelet transform are called the Discrete-Time Wavelet Transform and the Discrete-Time Continuous Wavelet Transform.

2.5 More on the Discrete Wavelet Transform: The DWT as a filter-bank.

In practice, the DWT is always implemented as a filter-bank (https://en.wikipedia.org/wiki/Filter_bank). This means that it is implemented as a cascade of high-pass and low-pass filters. This is because filter banks are a very efficient way of splitting a signal of into several frequency sub-bands.
Below I will try to explain the concept behind the filter-bank in a simple (and probably oversimplified) way. It is necessary in order to understand how the wavelet transform actually works and can be used in practical applications.

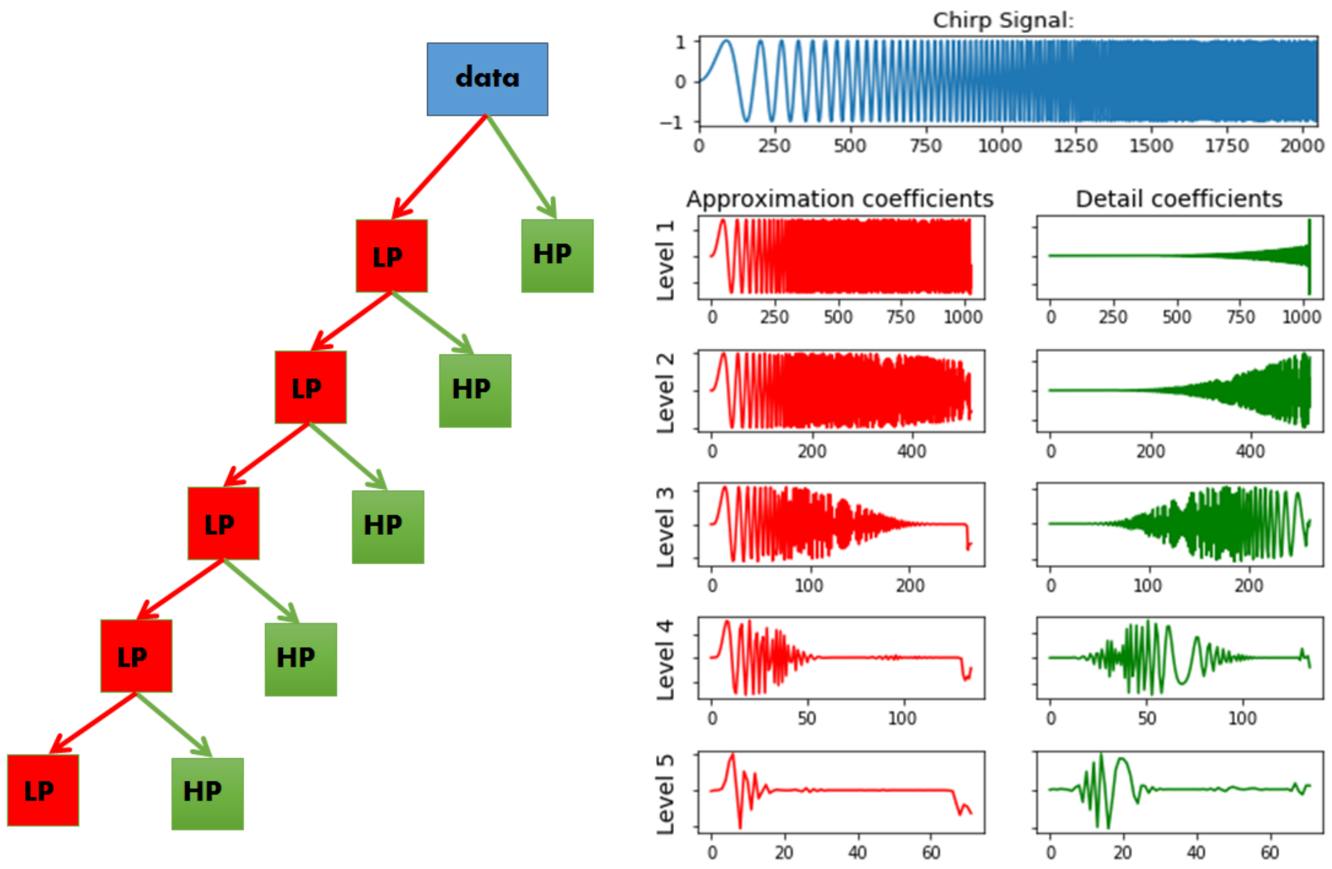
To apply the DWT on a signal, we start with the smallest scale. As we have seen before, small scales correspond with high frequencies. This means that we first analyze high frequency behavior. At the second stage, the scale increases with a factor of two (the frequency decreases with a factor of two), and we are analyzing behavior around half of the maximum frequency. At the third stage, the scale factor is four and we are analyzing frequency behavior around a quarter of the maximum frequency. And this goes on and on, until we have reached the maximum decomposition level.

What do we mean with maximum decomposition level? To understand this we should also know that at each subsequent stage the number of samples in the signal is reduced with a factor of two. At lower frequency values, you will need less samples to satisfy the Nyquist rate (https://en.wikipedia.org/wiki/Nyquist_rate)so there is no need to keep the higher number of samples in the signal; it will only cause the transform to be computationally expensive. Due to this downsampling, at some stage in the process the number of samples in our signal will become 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 stage we split our signal into a low-frequency part and a high-frequency part, i.e. 0-500 Hz and 500-1000 Hz.
At the second stage we take the low-frequency part and again split it into two parts: 0-250 Hz and 250-500 Hz.
At the third stage we split the 0-250 Hz part into a 0-125 Hz part and a 125-250 Hz part.
This goes on until we have reached the level of refinement we need or until we run out of samples.

We can easily visualize this idea, by plotting what happens when we apply the DWT on a chirp signal (https://en.wikipedia.org/wiki/Chirp). A chirp signal is a signal with a dynamic frequency spectrum; the frequency spectrum increases with time. The start 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.

```python
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)
plt.show()

data = chirp_signal
waveletname = 'sym5'

fig, axarr = plt.subplots(nrows=5, ncols=2, figsize=(6,6))
for ii in range(5):
    (data, coeff_d) = pywt.dwt(data, waveletname)
    axarr[ii, 0].plot(data, 'r')
    axarr[ii, 1].plot(coeff_d, '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()
```



(https://ataspinar.com/wp-content/uploads/2018/09/multilevel_coefficients_schematic.png)

Figure 7. The approximation and detail coefficients of the sym5 wavelet (level 1 to 5) applied on a chirp signal, from level 1 to 5. On the left we can see a schematic representation of the high pass and low pass filters applied on the signal at each level.

In Figure 7 we can see our chirp signal, and the DWT applied to it subsequently. There are a few things to notice here:

- In PyWavelets the DWT is applied with `pywt.dwt() (https://pywavelets.readthedocs.io/en/latest/ref/dwt-discrete-wavelet-transform.html)`
- The DWT return two sets of coefficients; the **approximation** coefficients and **detail** coefficients.
- The **approximation coefficients** represent the output of the low pass filter (averaging filter) of the DWT.
- The **detail coefficients** represent the output of the high pass filter (difference filter) of the DWT.
- By applying the DWT again on the approximation coefficients of the previous DWT, we get the wavelet transform of the next level.
- At each next level, the original signal is also sampled down by a factor of 2.

So now we have seen what it means that the DWT is implemented as a filter bank; At each subsequent level, the approximation coefficients are divided into a coarser low pass and high pass part and the DWT is applied again on the low-pass part.

As we can see, our original signal is now converted to several signals each corresponding to different frequency bands. Later on we will see how the approximation and detail coefficients at the different frequency sub-bands can be used in applications like removing high frequency noise from signals, compressing signals, or classifying the different types signals.

PS: We can also use `pywt.wavedec() (https://pywavelets.readthedocs.io/en/latest/ref/dwt-discrete-wavelet-transform.html#multilevel-decomposition-using-wavedec)` to immediately calculate the coefficients of a higher level. This functions takes as input the original signal and the level n and returns the one set of approximation coefficients (of the n-th level) and n sets of detail coefficients (1 to n-th level).

PS2: This idea of analyzing the signal on different scales is also known as multiresolution / multiscale analysis (https://en.wikipedia.org/wiki/Multiresolution_analysis), and decomposing your signal in such a way is also known as multiresolution decomposition, or sub-band coding (https://en.wikipedia.org/wiki/Sub-band_coding).

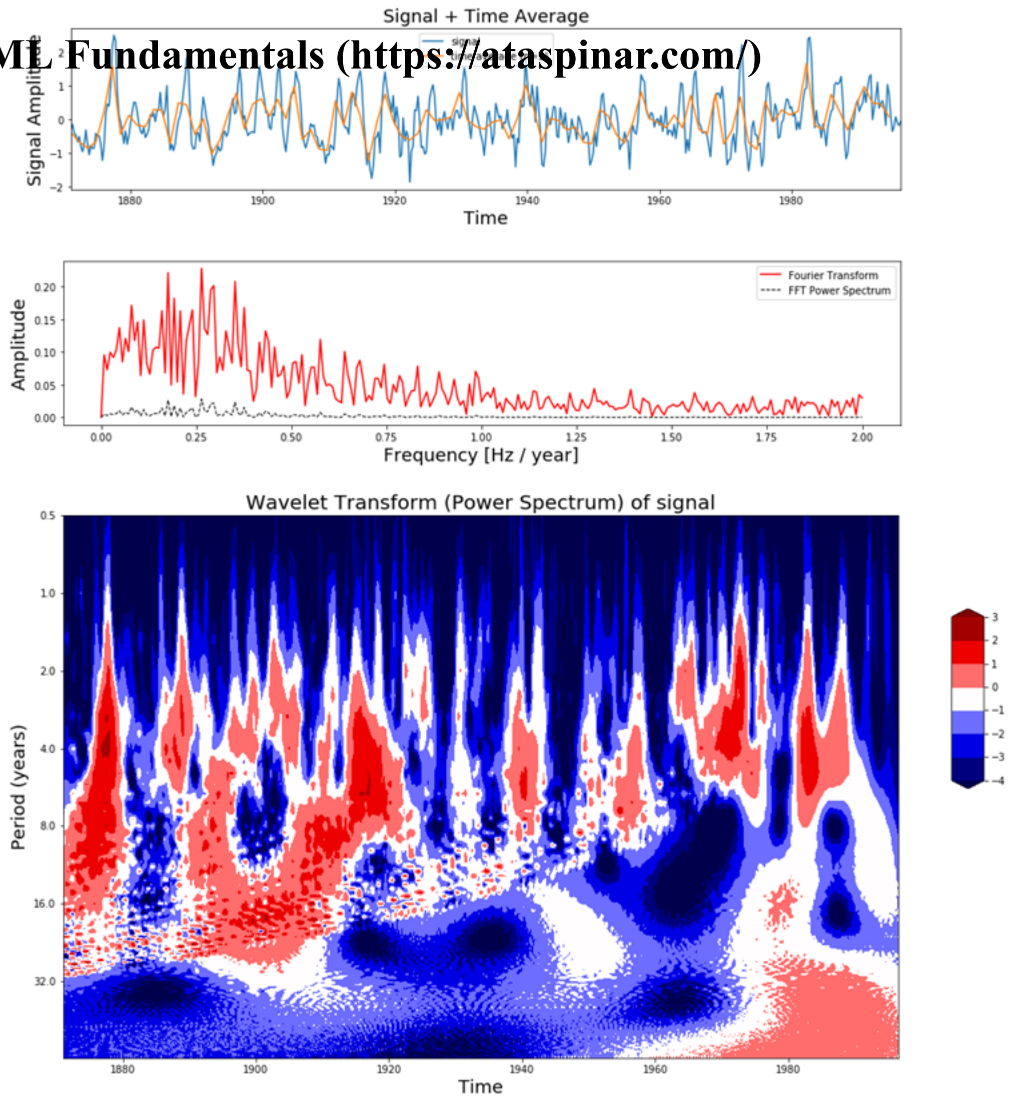
3.1 Visualizing the State-Space using the Continuous Wavelet Transform

So far we have seen what the wavelet transform 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 the impact of the order and level of decomposition is on the mother wavelet, and how and why the DWT is implemented as a filter-bank.

We have also seen that the output of a wavelet transform on 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 for tracking the El Nino (https://en.wikipedia.org/wiki/El_Ni%C3%B1o) and contains quarterly measurements of the sea surface temperature from 1871 up to 1997. In order to understand the power of a scaleogram, let us visualize it for el-Nino dataset together with the original time-series data and its Fourier Transform.

```python
def plot_wavelet(time, signal, scales,
                 waveletname = 'cmor',
                 cmap = plt.cm.seismic,
                 title = 'Wavelet Transform (Power Spectrum) of signal',
                 ylabel = 'Period (years)',
                 xlabel = 'Time'):

    dt = time[1] - time[0]
    [coefficients, frequencies] = pywt.cwt(signal, scales, waveletname, 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={})'.format(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 = 2.0/N * np.abs(fft_values_[0:N//2])
    return f_values, fft_values

def plot_fft_plus_power(time, signal):
    dt = time[1] - time[0]
    N = len(signal)
    fs = 1/dt

    fig, ax = plt.subplots(figsize=(15, 3))
    variance = np.std(signal)**2
    f_values, fft_values = get_fft_values(signal, dt, N, fs)
    fft_power = variance * 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('Frequency [Hz / year]', fontsize=18)
    ax.set_ylabel('Amplitude', fontsize=18)
    ax.legend()
    plt.show()

dataset = "http://paos.colorado.edu/research/wavelets/wave_idl/sst_nino3.dat"
df_nino = pd.read_table(dataset)
N = df_nino.shape[0]
t0=1871
dt=0.25
time = np.arange(0, N) * dt + t0
signal = 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)
```



(https://ataspinar.com/wp-content/uploads/2018/09/el_nino_scaleogram.png)

Figure 8. el-Nino dataset (on top), with the Fourier Transform (middle) and the Continuous Wavelet Transform (bottom).

In Figure 8 we can see in the top figure the el-Nino dataset together with its time average, in the middle figure the Fourier Transform and at the bottom figure the scaleogram produced by the Continuous Wavelet Transform.

In the scaleogram we can see that most of the power is concentrated in a 2-8 year period. If we convert this to frequency (T = 1 / f) this corresponds with a frequency of 0.125 – 0.5 Hz. The increase in power can also be seen in the Fourier transform around these frequency values. The main difference is that the wavelet transform also gives us temporal information and the Fourier Transform does not. For example, in the scaleogram we can see that up to 1920 there were many fluctuations, while there were not so much between 1960 – 1990. We can also see that there is a shift from shorter to longer periods as time progresses. This is the kind of dynamic behavior in the signal which can be visualized with the Wavelet Transform but not with the Fourier Transform.

This should already make clear how powerful the wavelet transform can be for machine learning purposes. But to make the story complete, let us also look at how this can be used in combination with a Convolutional Neural Network to classify signals.

3.2 Using the Continuous Wavelet Transform and a Convolutional Neural Network for classification of signals

In section 3.1 we have seen that the wavelet transform of a 1D signal results in a 2D scaleogram which contains a lot more information than just the time-series or just the Fourier Transform. We have seen that applied on the el-Nino dataset, it can not only tell us what the period is of the largest oscillations, but also when these oscillations were present and when not.

Such a scaleogram can not only be used to better understand the dynamical 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 while you are walking up the stairs or down the stairs, the scaleograms will look different. ECG measurements of people with a healthy heart will have different scaleograms than ECG measurements of people with arrhythmia. Or measurements on a bearing, motor, rotor, ventilator, etc when it is faulty vs when it not faulty. The possibilities are limitless!

So by looking at the scaleograms we can distinguish a broken motor from a working one, a healthy person from a sick one, a person walking up the stairs from a person walking down the stairs, etc etc. But if you are as lazy as me, you probably don't want to sift through thousands of scaleograms manually. One way to automate this process is to build a Convolutional Neural Network which can automatically detect the class each scaleogram belongs to and classify them accordingly.

What was the deal again with CNN? In previous blog posts we have seen how we can use Tensorflow to build a convolutional neural network from scratch (https://ataspinar.com/2017/08/15/building-convolutional-neural-networks-with-tensorflow/). And how we can use such a CNN to detect roads in satellite images (https://ataspinar.com/2017/12/04/using-convolutional-neural-networks-to-detect-features-in-sattelite-images/). If you are not familiar with CNN's it is a good idea to have a look at these previous blog posts, since the rest of this section assumes you have some knowledge of CNN's.

In the next few sections we will load a dataset (containing measurement of people doing 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 us try to classify an open dataset containing time-series using the scaleograms and a CNN. The Human Activity Recognition Dataset (UCI-HAR) (https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones) contains sensor measurements of people while they were doing different types of activities, like walking up or down the stairs, laying, standing, walking, etc. There are in total 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 of CWT + CNN!

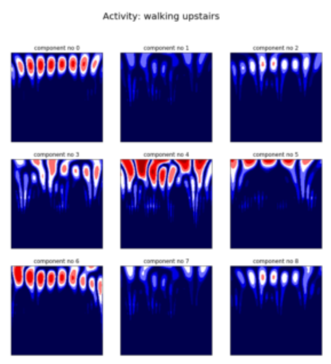
PS: For more on how the UCI HAR dataset looks like, you can also have a look at the previous blog post (https://ataspinar.com/2018/04/04/machine-learning-with-signal-processing-techniques/) in which it was described in more detail.

After we have downloaded the data, we can load it into a numpy nd-array in the following way:

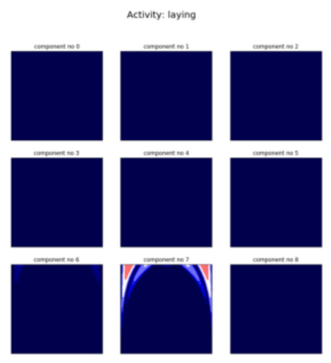
```python
def read_signals_ucihar(filename):
    with open(filename, 'r') as fp:
        data = fp.read().splitlines()
        data = map(lambda x: x.rstrip().lstrip().split(), data)
        data = [list(map(float, line)) for line in data]
    return data

def read_labels_ucihar(filename):
    with open(filename, 'r') as fp:
        activities = fp.read().splitlines()
        activities = list(map(int, activities))
    return activities

def load_ucihar_data(folder):
    train_folder = folder + 'train/InertialSignals/'
    test_folder = folder + 'test/InertialSignals/'
    labelfile_train = folder + 'train/y_train.txt'
    labelfile_test = 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 + 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(labelfile_test)
    return 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)
```

The training set contains 7352 signals where each signal has 128 measurement 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 into one of size (2947 , 128, 9).

Since the signal consists of nine components we have to apply the CWT nine times for each signal. Below we can see the result of the CWT applied on two different signals from the dataset. The left one consist of a signal measured while walking up the stairs and the right one is a signal measured while laying down.



(https://ataspinar.com/wp-content/uploads/2018/12/1_walking-upstairs.png)



(https://ataspinar.com/wp-content/uploads/2018/12/2_laying.png)

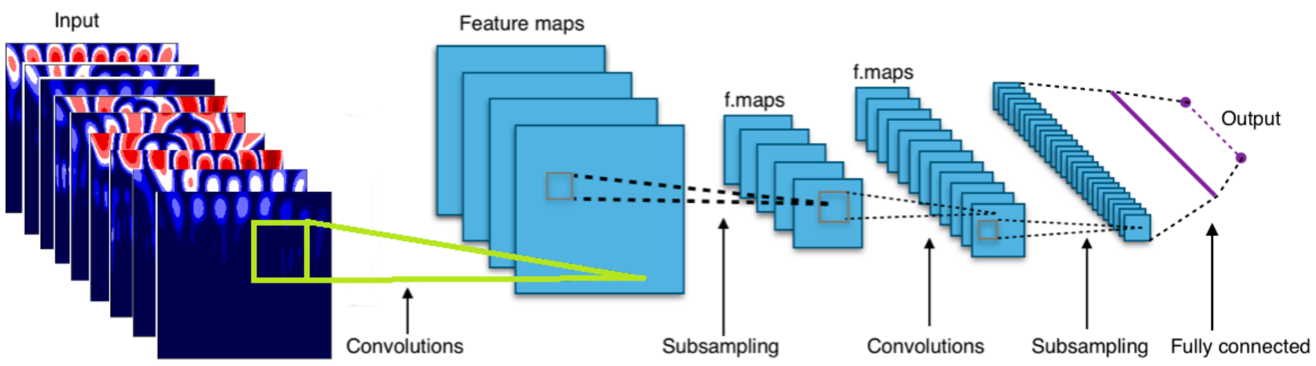
Figure 9. The CWT applied on two signals belonging to the UCI HAR dataset. Each signal has nine different components. On the left we can see the signals measured during walking upstairs and on the right we can see a signal measured during laying.

3.2.2 Applying the CWT on the dataset and transforming the data to the right format

Since each signal has nine components, each signal will also have nine scaleograms. So the next question to ask is, how do we feed this set of nine scaleograms into a Convolutional Neural Network? There are several options we could follow:

1. Train a CNN for each component separately and combine the results of the nine CNN's in some sort of an ensembling method. I suspect that this will generally result in a poorer performance since the inter dependencies between the different components are not taken into account.
2. Concatenate the nine different signals into one long signal and apply the CWT on the concatenated signal. This *could* work but there will be discontinuities at location where two signals are concatenated and this will introduced noise in the scaleogram at the boundary locations of the component signals.
3. Calculate the CWT first and thén concatenate the nine different CWT images into one and feed that into the CNN. This could also work, but here there will also be discontinuities at the boundaries of the CWT images which will feed noise into the CNN. If the 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:
4. Place the nine scaleograms on top of each other and create one single image with nine channels. What does this mean? Well, normally an image has either one 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 is that there will be three times more filters compared to an RGB image.

This process is illustrated in the Figure 11.



(https://ataspinar.com/wp-content/uploads/2018/12/9layer_image_CNN.png)

Figure 10. We will use the nine CWT from the nine components to create a nine-channel image which is used as input for a Convolutional Neural Network.

Below we can see the Python code on how to apply the CWT on the signals in the dataset, and reformat it in such a way that it can be used as input for our Convolutional Neural Network. The total dataset contains over 10.000 signals, but we will only use 5.000 signals in our training set and 500 signals in our test set.

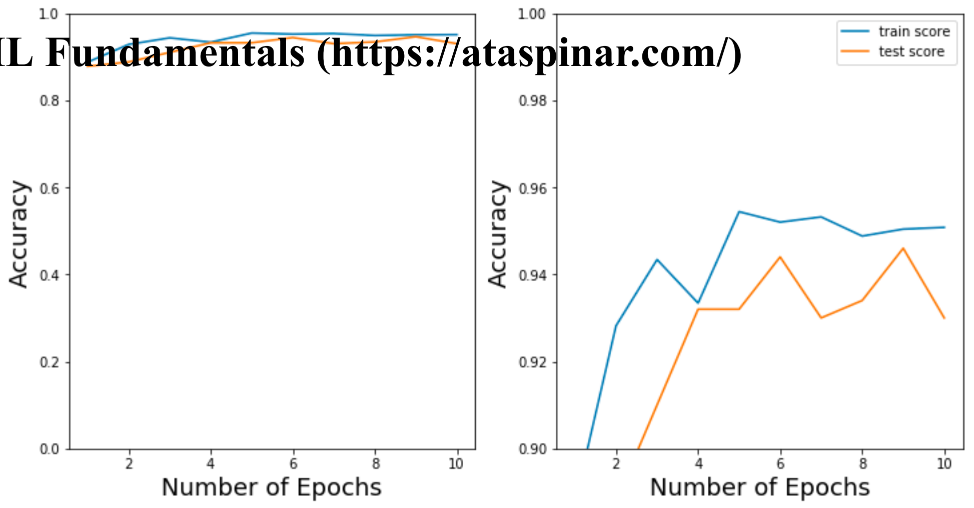
```python
scales = range(1,128)
waveletname = 'morl'
train_size = 5000
test_size= 500

train_data_cwt = np.ndarray(shape=(train_size, 127, 127, 9))

for ii in range(0,train_size):
    if ii % 1000 == 0:
        print(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 range(0,test_size):
    if ii % 100 == 0:
        print(ii)
    for jj in 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(map(lambda x: int(x) - 1, uci_har_labels_test))

x_train = train_data_cwt
y_train = list(uci_har_labels_train[:train_size])
x_test = test_data_cwt
y_test = list(uci_har_labels_test[:test_size])
```

As you can see above, the CWT of a single signal-component (128 samples) results in an image of 127 by 127 pixels. So the scaleograms coming from the 5000 signals of the training dataset are stored in an numpy ndarray of size (5000, 127, 127, 9) and the scaleograms coming from the 500 test signals are stored in one of size (500, 127, 127, 9).

3.2.3 Training 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 will need the keras library (https://keras.io/#installation), so please install it first.

```
import keras
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.callbacks import History
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
epochs = 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, (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.compile(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)
print('Test loss: {}, Test accuracy: {}'.format(test_score[0], test_score[1]))

*** Epoch 1/10
*** 5000/5000 [==============================] - 235s 47ms/step - loss: 0.3963 - acc: 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
```



(https://ataspinar.com/wp-content/uploads/2018/12/history_cwt_.png)

As you can see, combining the Wavelet Transform and a Convolutional Neural Network leads to an awesome and amazing result!

We have an accuracy of 94% on the Activity Recognition dataset. Much higher than you can achieve with any other method.
In section 3.5 we will use the Discrete Wavelet Transform instead of Continuous Wavelet Transform to classify the same dataset and achieve similarly amazing results!

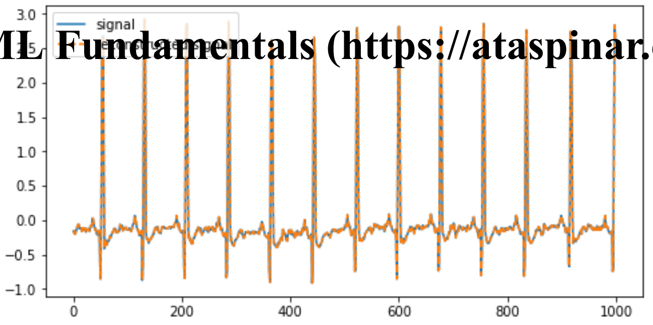
3.3 Deconstructing a signal using the DWT

In Section 2.5 we have seen how the DWT is implemented as a filter-bank which can deconstruct a signal into its frequency sub-bands. In this section, let us see how we can use PyWavelets to deconstruct a signal into its frequency sub-bands and reconstruct the original signal again.

PyWavelets offers two different ways to deconstruct a signal.

1. We can either apply `pywt.dwt() (https://pywavelets.readthedocs.io/en/latest/ref/dwt-discrete-wavelet-transform.html)` on a signal to retrieve the approximation coefficients. Then apply the DWT on the retrieved coefficients to get the second level coefficients and continue this process until you have reached the desired decomposition level.
2. Or we can apply `pywt.wavedec() (https://pywavelets.readthedocs.io/en/latest/ref/dwt-discrete-wavelet-transform.html#multilevel-decomposition-using-wavedec)` directly and retrieve all of the the detail coefficients up to some level n. This functions takes as input the original signal and the level n and returns the one set of approximation 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', linestyle='--')
ax.legend(loc='upper left')
plt.show()
```



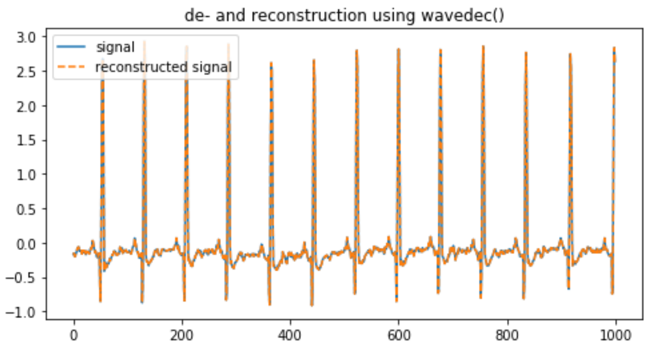
(https://ataspinar.com/wp-content/uploads/2018/09/reconstructed_signal.png)

Figure 12. A signal together with the reconstructed signal.

Above we have deconstructed a signal into its coefficients and reconstructed it again using the inverse DWT.

The second way is to use `pywt.wavedec()` to deconstruct and reconstruct a signal and it is probably the most simple way if you want to get higher-level 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 using wavedec()')
plt.show()
```



(https://ataspinar.com/wp-content/uploads/2018/09/reconstructed_signal3.png)

Figure 13. A signal together with the reconstructed signal.

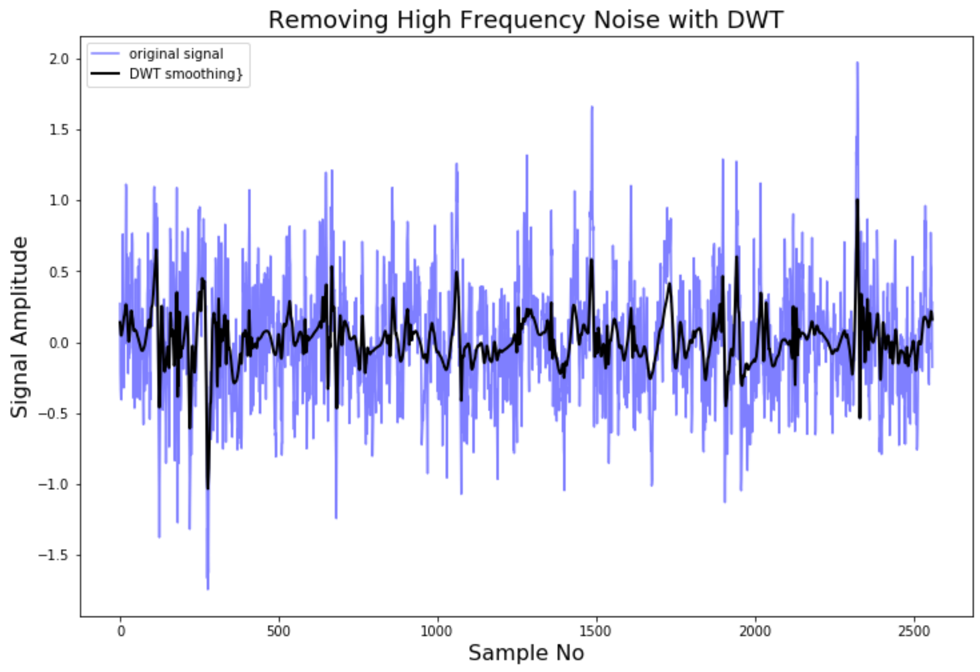
3.4 Removing (high-frequency) noise using the DWT

In the previous section we have seen how we can deconstruct a signal into the approximation (low pass) and detail (high pass) coefficients. If we reconstruct the signal using these coefficients we will get the original signal back.

But what happens if we reconstruct while we leave out some detail coefficients? Since the detail coefficients represent the high frequency part 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 one way to filter it out.
Leaving out of the detail coefficients can be done using pywt.threshold() (https://pywavelets.readthedocs.io/en/latest/ref/thresholding-functions.html), which removes coefficient values higher than the given threshold.

Lets demonstrate this using NASA's Femto Bearing Dataset (https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/#femto). This is a dataset containing high frequency sensor data regarding accelerated degradation of bearings.

```python
DATA_FOLDER = './FEMTO_bearing/Training_set/Bearing1_1/'
filename = 'acc_01210.csv'
df = pd.read_csv(DATA_FOLDER + filename, header=None)
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, ax = 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='DWT smoothing}', linewidth=2)
ax.legend()
ax.set_title('Removing High Frequency Noise with DWT', fontsize=18)
ax.set_ylabel('Signal Amplitude', fontsize=16)
ax.set_xlabel('Sample No', fontsize=16)
plt.show()
```



(https://ataspinar.com/wp-content/uploads/2018/11/dwt_smoothing.png)

Figure 14. A high frequency signal and its DWT smoothed version.

As we can see, by deconstructing the signal, setting some of the coefficients to zero, and reconstructing it again, we can remove high frequency noise from the signal.

People who are familiar with signal processing techniques might know there are a lot of different ways to remove noise from a signal. For example, the Scipy library contains a lot of smoothing filters (one of them is the famous Savitzky-Golay (https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.signal.savgol_filter.html) filter) and they are much simpler to use. Another method for smoothing a signal is to average it over its time-axis.
So why should you use the DWT instead? The advantage of the DWT again comes from the many wavelet shapes there are available. You can choose a wavelet which will have a shape characteristic to the phenomena you expect to see. In this way, less of the phenomena you expect to see will be smoothed out.

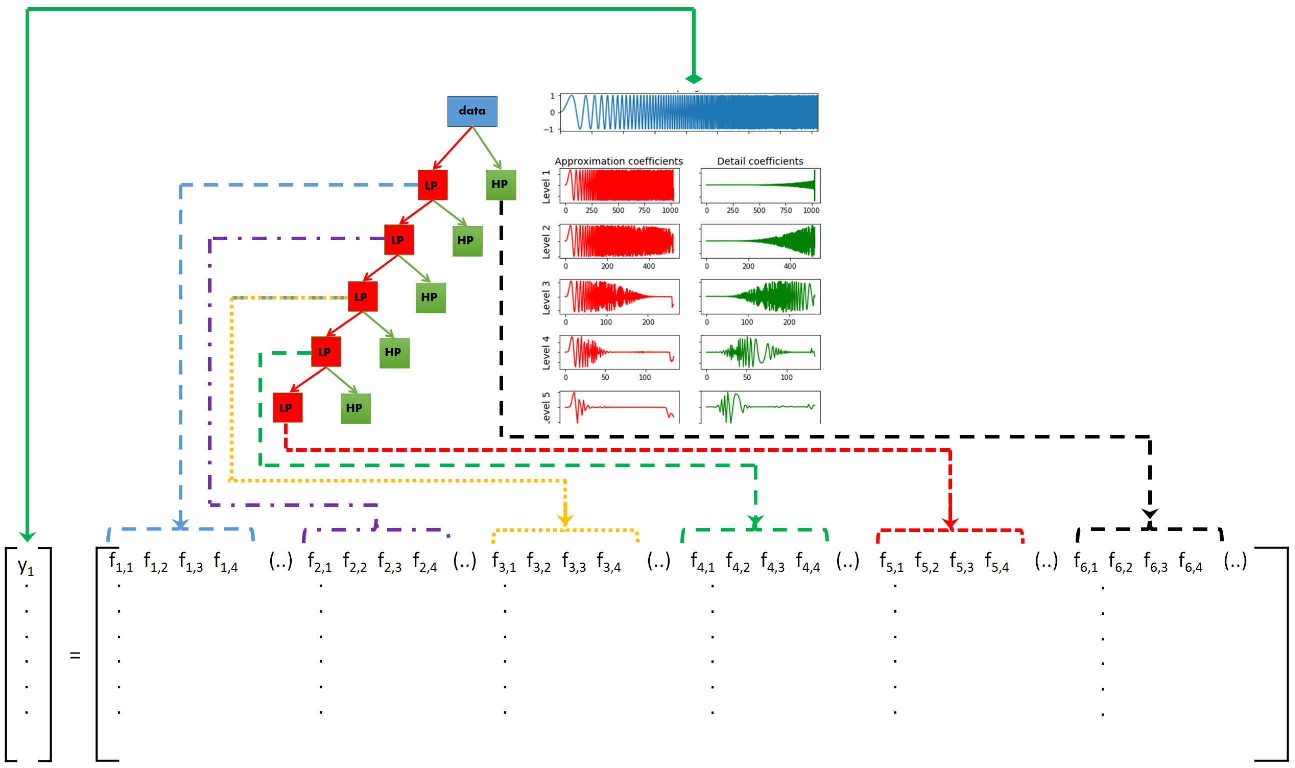
3.5 Using the Discrete Wavelet Transform to classify signals

In Section 3.2 we have seen how we can use a the CWT and a CNN to classify signals. Of course it is also possible to use the DWT to classify signals. Let us have a look at how this could be done.

3.5.1 The idea behind Discrete Wavelet classification

The idea behind DWT signal classification is as follows: The DWT is used to split a signal into different frequency sub-bands, as many as needed or as many as possible. If the different types of signals exhibit different frequency characteristics, this difference in behavior has to be exhibited in one of the frequency sub-bands. So if we generate features 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 it by using these features, the classifier should be able to distinguish between the different types of signals.

This is illustrated in the figure below:


data
LP
HP
Approximation coefficients
Detail coefficients
Level 1
Level 2
Level 3
Level 4
Level 5
y_1
=
$f_{1,1}$ $f_{1,2}$ $f_{1,3}$ $f_{1,4}$ (..) $f_{2,1}$ $f_{2,2}$ $f_{2,3}$ $f_{2,4}$ (..) $f_{3,1}$ $f_{3,2}$ $f_{3,3}$ $f_{3,4}$ (..) $f_{4,1}$ $f_{4,2}$ $f_{4,3}$ $f_{4,4}$ (..) $f_{5,1}$ $f_{5,2}$ $f_{5,3}$ $f_{5,4}$ (..) $f_{6,1}$ $f_{6,2}$ $f_{6,3}$ $f_{6,4}$ (..)

(https://ataspinar.com/wp-content/uploads/2018/12/multilevel_coefficients_to_features3.png)

Figure 15. By applying the DWT on a signal we can deconstruct it its frequency sub-bands. And out of each sub-band we can generate features which can be used as input for a Classifier.

3.5.2 Generating features per sub-band

So what kind of features can be generated from the set of values for each of the sub-bands? Of course this will highly depend on the type of signal and the application. But in general, below are some features which are most frequently used for signals.

- Auto-regressive model coefficient values
- (Shannon) Entropy values; entropy values can be taken as a measure of complexity of the signal.
- Statistical features like:
  - variance
  - standard deviation

- Mean
- Median
- 25th percentile value
- 75th percentile value
- Root Mean Square value; square of the average of the squared amplitude values
- The mean of the derivative
- Zero crossing rate, i.e. the number of times a signal crosses y = 0
- Mean crossing rate, i.e. the number of times a signal crosses y = mean(y)

These are just some ideas you could use to generate the features out of each sub-band. You could use some of the features described here, or you could use all of them. Most classifiers in the scikit-learn package are powerful enough to handle a large number of input features and distinguish between useful ones and non-useful ones. However, I still recommend you think carefully about which feature would be useful for the type of signal you are trying to classify.

PS: There are a hell of a lot more statistical functions in scipy.stats (https://docs.scipy.org/doc/scipy/reference/stats.html). By using these you can create more features if necessary.

Let's see how this could be done in Python for a few of the above mentioned features:

```python
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) &gt; 0))[0]
    no_zero_crossings = len(zero_crossing_indices)
    mean_crossing_indices = np.nonzero(np.diff(np.array(list_values) &gt; 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 some statistics like several percentiles, mean, standard deviation, variance, etc,
- a function to calculate the zero crossings rate and the mean crossings rate,
- and a function to combine the results of these three functions.

The final function returns a set of 12 features for any list of values. So if one signal is decomposed into 10 different sub-bands, and we generate features for each sub-band, we will end up with 10*12 = 120 features 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 decompose the signals in the training set into its sub-bands, calculate the features for each sub-band, use the features to train a classifier and use the classifier to predict the signals in the test-set.

We will do this for two time-series datasets:

1. The UCI-HAR dataset which we have already seen in section 3.2. This dataset contains smartphone sensor data of humans while doing different types of activities, like sitting, standing, walking, stair up and stair down.
2. PhysioNet (https://physionet.org/) ECG Dataset (download from here (https://github.com/mathworks/physionet_ECG_data/blob/master/ECGData.zip)) which contains a set of ECG measurements of healthy persons (indicated as Normal sinus rhythm, NSR) and persons with either an arrhythmia (ARR) or a congestive heart failure (CHF). This dataset contains 96 ARR measurements, 36 NSR measurements and 30 CHF measurements.

After we have downloaded both datasets, and placed them in the right folders, the next step is to load them into memory. We have already seen how we can load the UCI-HAR dataset in section 3.2, and below we can see how to load the ECG dataset.

```python
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:]
```

The ECG dataset is saved as a MATLAB file, so we have to use scipy.io.loadmat() to open this file in Python and retrieve its contents (the ECG measurements and the labels) as two separate lists.

The UCI HAR dataset is saved in a lot of .txt files, and after reading the data we save it into an numpy ndarray of size (no of signals, length of signal, no of components) = (10299 , 128, 9)

Now let us have a look at how we can get features out of these two datasets.

```python
def get_uci_har_features(dataset, labels, waveletname):
    uci_har_features = []
    for signal_no in range(0, len(dataset)):
        features = []
        for signal_comp in range(0,dataset.shape[2]):
            signal = dataset[signal_no, :, signal_comp]
            list_coeff = pywt.wavedec(signal, waveletname)
            for coeff in list_coeff:
                features += get_features(coeff)
        uci_har_features.append(features)
    X = np.array(uci_har_features)
    Y = np.array(labels)
    return X, Y

def get_ecg_features(ecg_data, ecg_labels, waveletname):
    list_features = []
    list_unique_labels = list(set(ecg_labels))
    list_labels = [list_unique_labels.index(elem) for elem in ecg_labels]
    for signal in ecg_data:
        list_coeff = pywt.wavedec(signal, waveletname)
        features = []
        for coeff in list_coeff:
            features += get_features(coeff)
        list_features.append(features)
    return 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')
```

What we have done above is to write functions to generate features from the ECG signals and the UCI HAR signals. There is nothing special about these functions and the only reason we are using two seperate functions is because the two datasets are saved in different formats. The ECG dataset is saved in a list, and the UCI HAR dataset is saved in a 3D numpy ndarray.

For the ECG dataset we iterate over the list of signals, and for each signal apply the DWT which returns a list of coefficients. For each of these coefficients, i.e. for each of the frequency sub-bands, we calculate the features with the function we have defined previously. The features calculated from all of the different coefficients belonging to one signal are concatenated together, since they belong to the same signal.

The same is done for the UCI HAR dataset. The only difference is that we now have two for-loops since each signal consists of nine components. The features generated from each of the sub-band from each of the signal component are concatenated together.

Now that we have calculated the features for the two datasets, we can use a GradientBoostingClassifier from the scikit-learn library and train it.

PS: If you want to know more about classification with the scikit-learn library, you can have a look at this blog post (https://ataspinar.com/2017/05/26/classification-with-scikit-learn/).

```
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 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("Train Score for the UCI-HAR dataset is about: {}".format(train_score))
print("Test Score 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 about: 0.93
*** Train Score for the UCI_HAR dataset is about: 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 the classification accuracies between DWT, Fourier Transform and Recurrent Neural Networks

So far, we have seen throughout the various blog-posts, how we can classify time-series and signals in different ways. In a previous post we have classified the UCI-HAR dataset using Signal Processing techniques like The Fourier Transform (https://ataspinar.com/2018/04/04/machine-learning-with-signal-processing-techniques/). The accuracy on the test-set was ~91%.

In another blog-post, we have classified the same UCI-HAR dataset using Recurrent Neural Networks (https://ataspinar.com/2018/07/05/building-recurrent-neural-networks-in-tensorflow/). The highest achieved accuracy on the test-set was ~86%.

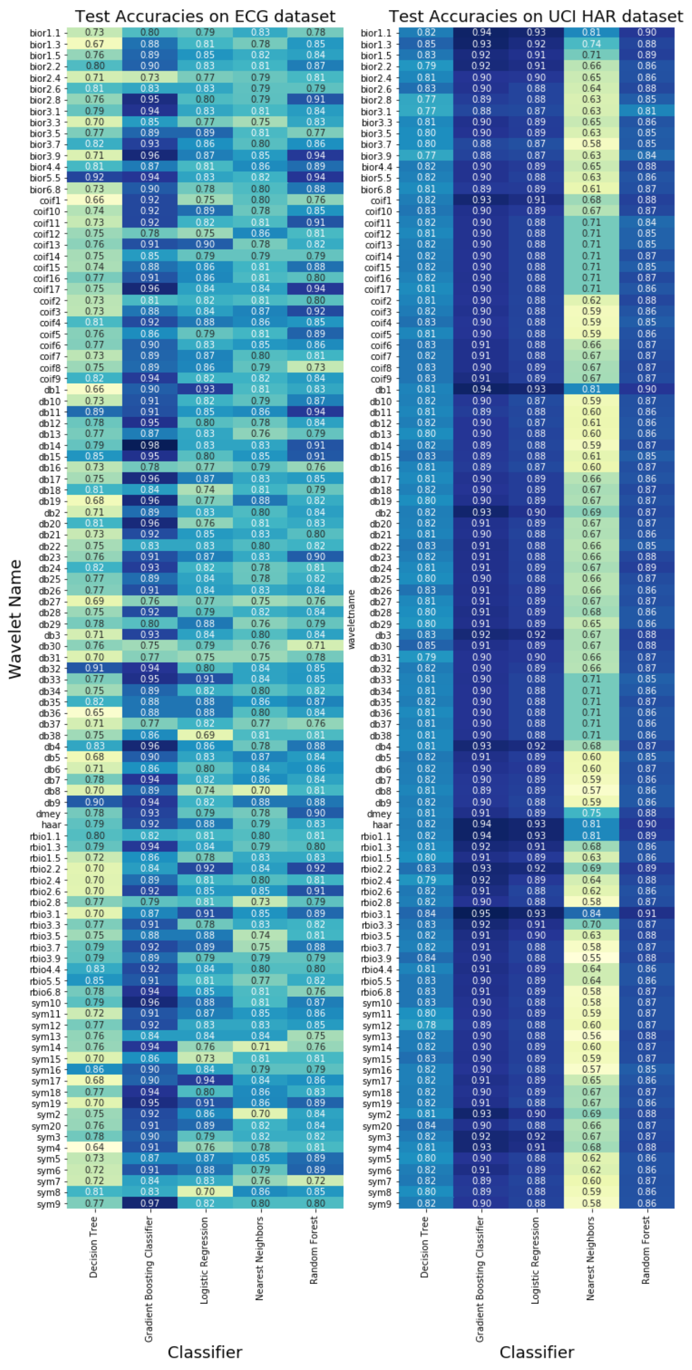
In this blog-post we have achieved an accuracy of ~94% on the test-set with the approach of CWT + CNN and an accuracy of ~95% with the DWT + GradientBoosting approach!

It is clear that the Wavelet Transform has far better results. We could repeat this for other datasets and I suspect there the Wavelet Transform will also outperform.

What we can also notice is that strangely enough Recurrent Neural Networks perform the worst. Even the approach of simply using the Fourier Transform and then the peaks in the frequency spectrum has an better accuracy than RNN's.

This really makes me wonder, what the hell Recurrent Neural Networks are actually good for. It is said that a RNN can learn 'temporal dependencies in sequential data'. But, if a Fourier Transform can fully describe any signal (no matter its complexity) in terms of the Fourier components, then what more can be learned with respect to 'temporal dependencies'.
It is no wonder that people already start talking about the fall of RNN / LSTM (https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0).

PS: The achieved accuracy using the DWT will depend on the features you decide to calculate, the wavelet and the classifier you decide to use. To give an impression, below are the accuracy values for the test set of the UCI-HAR and ECG datasets, for all of the wavelets present in PyWavelets, and for the most used 5 classifiers in scikit-learn.



(https://ataspinar.com/wp-content/uploads/2018/12/ucihar_ecg_accuracies.png)

Figure 16. The accuracies on the test set of the UCI-HAR and ECG datasets, for all of the wavelets present in PyWavelets, and for five of the most popular classifiers in scikit-learn.

We can see that there will be differences in accuracy depending on the chosen classifier. Generally speaking, the Gradient Boosting (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html) classifier performs best. This should come as no surprise since almost all kaggle competitions are won with the gradient boosting model (https://www.quora.com/What-machine-learning-approaches-have-won-most-Kaggle-competitions).

What is more important is that the chosen wavelet can also have a lot of influence on the achieved accuracy values. Unfortunately I do not have an 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 bit of literature research.

4. Final Words

In this blog post we have seen how we can use the Wavelet Transform for the analysis and classification of time-series and signals (and some other stuff). Not many people know how to use the Wavelet Transform, but this is mostly because the theory is not beginner-friendly and the wavelet transform is not readily available in open source programming languages.

MATLAB is one the few programming languages with a very complete Wavelet Toolbox (https://nl.mathworks.com/products/wavelet.html). But since MATLAB has a expensive licensing fee, it is mostly used in the academic world and large corporations.

Among Data Scientists, the Wavelet Transform remains an undiscovered jewel and I recommend fellow Data Scientists to use it more often. I am very thankful to the contributors of the PyWavelets package who have implemented a large set of Wavelet families and higher level functions for using these wavelets. Thanks to them, Wavelets can now more easily be used by people using 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 provided in this blog-post will point you in the right direction in your quest to use wavelets for data analysis.

A lot will depend on the choices you make; which wavelet transform will you use, CWT or DWT? which wavelet family will you use? Up to which level of decomposition will you go? What is the right range of scales to use?

Like most things in life, the only way to master a new method is by a lot of practice. You can look up some literature to see what the commonly used wavelet type is for your specific problem, but do not make the mistake of thinking whatever you read in research papers is the holy and you don't need to look any further. 🙂

PS: You can also find the code in this blog-post in five different Jupyter notebooks in my Github repository (https://github.com/taspinar/siml).

Delen:

(https://ataspinar.com/2018/12/21/a-guide-for-using-the-wavelet-transform-in-machine-learning/?share=twitter&nb=1)

(https://ataspinar.com/2018/12/21/a-guide-for-using-the-wavelet-transform-in-machine-learning/?share=facebook&nb=1)

Share This:

(/#facebook) (/#twitter) (/#reddit) (/#linkedin)
(/#sina_weibo)
(https://www.addtoany.com/share#url=https%3A%2F%2Fataspinar.com
using-the-wavelet-transform-in-machine-
learning%2F&title=A%20guide%20for%20using%20the%20Wavelet%20Trar

⟵ Building Recurrent Neural Networks in Tensorflow (https://ataspinar.com/2018/07/05/building-recurrent-neural-networks-in-tensorflow/)

Time-Series forecasting with Stochastic Signal Analysis techniques (https://ataspinar.com/2020/12/22/time-series-forecasting-with-stochastic-signal-analysis-techniques/) ⟶