Machine Learning for IoT
Lab 2 – Data Pre-processing

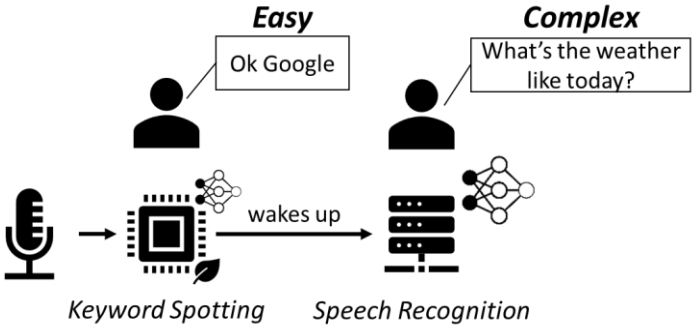
Before Starting:

You need to install on your RPI board an additional dependency:

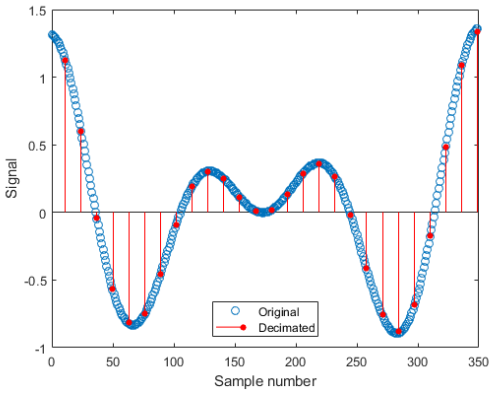
```
sudo apt install -y libatlas-base-dev
```

Background 1: Pre-Processing of Audio Signals – Resampling

Speech is a popular way to interact with consumer electronic devices, such as smartphones or smart speakers. Always-on speech recognition is not energy efficient as it requires to transmit a continuous audio stream to the cloud, where data get processed. To mitigate this concern, devices first detect short keywords such as "Hey Siri" or "Ok Google" that wake up the device and trigger the full-scale speech recognition. This task, called Keyword Spotting, is much simpler, and therefore can be performed on board of the sensing nodes with lightweight Convolutional Neural Networks. In this way, it is possible to cut transmission energy and minimize network congestion.

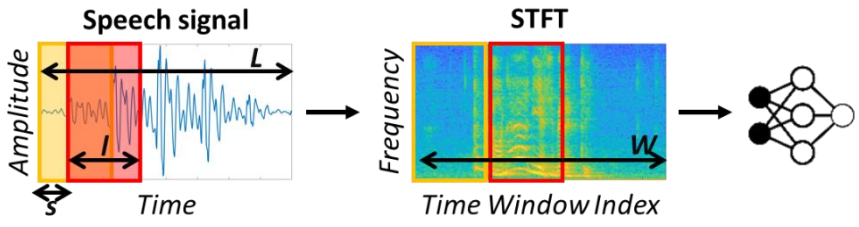


The first step is to resample the recorded signal to a lower frequency in order to reduce the memory requirements and accelerate the subsequent processing steps.
When dealing with digital audio signals, the resampling procedure involves the application of lowpass filters to avoid aliasing. Among the available techniques, resampling with poly-phase filtering is one of the most efficient method for sample rate conversion.
(Ref.: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.resample_poly.html)



Background 2: Pre-Processing of Audio Signals – STFT

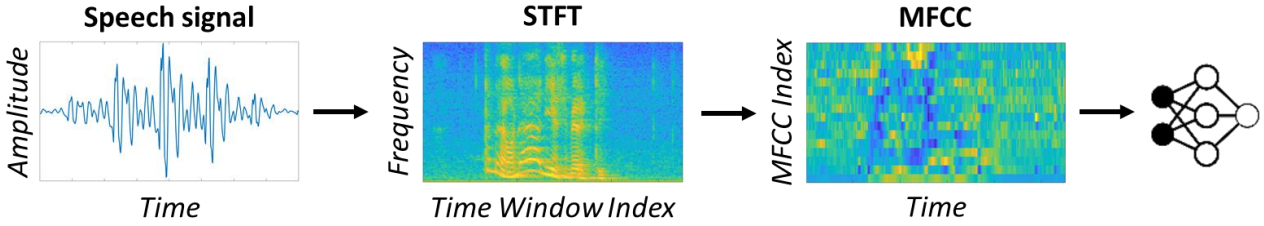
Before feeding the data to a Convolutional Neural Networks, it is required to perform a set of pre-processing steps. The most common strategy is to move from the time domain to the frequency domain using Short Time Fourier Transform (STFT). The input signal of length L is split in a set of frames of length l, partially overlapping among each other with a stride s. The STFT is computed stacking the DFTs processed on each frame. This transformation converts a one-dimensional time-series signal into a two-dimensional image, enabling to solve keyword spotting as a simple image classification problem.



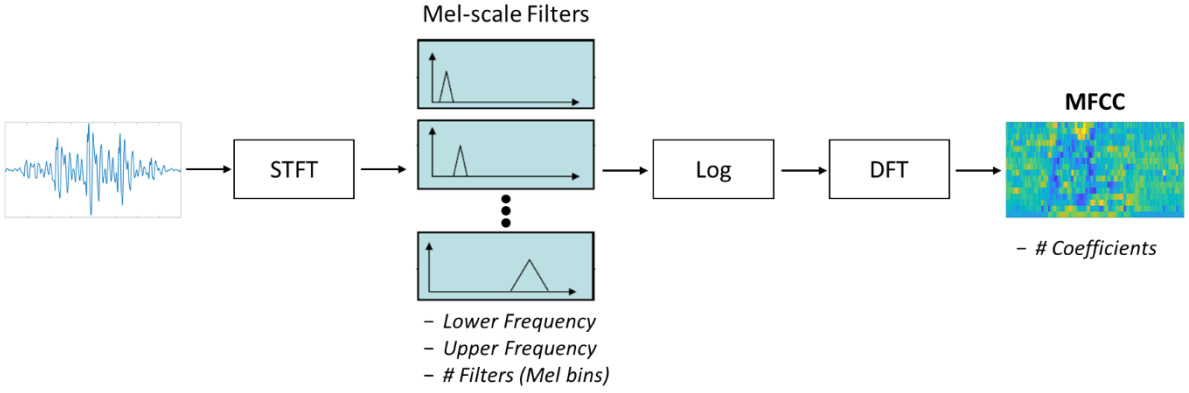
$$W = \frac{L-l}{s} + 1$$

Background 3: Feature Extraction from Audio with MFCCs

Feature extraction relies on the hypothesis that representing sounds as they are perceived by the human ear improves the classification accuracy. A good approximation of the human perception can be achieved extracting the **Mel-frequency cepstral coefficients** (**MFCCs**) from the input signal.

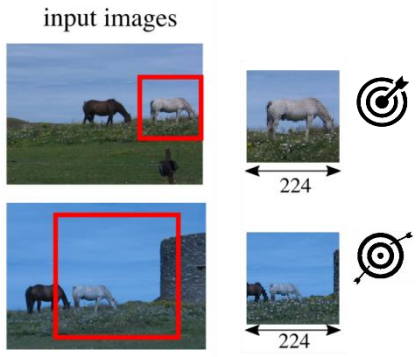


The Mel-frequency cepstrum is a representation of the STFT of a sound, that tries to mimic how the membrane in our ear senses the vibrations of sounds. The MFCCs are coefficients that composes the Mel-frequency cepstrum. They represent phonemes (distinct units of sound) as the shape of the vocal tract (which is responsible for sound generation) is manifest in them.



Background 4: Image pre-processing

Before feeding an image to a ConvNets for classification, it is paramount to detect the region of interest (ROI). As the input size of a ConvNets is generally fixed and defined at design-time, the ROI must be scaled in order to obtain a compliant size.



Exercise 5: Pre-process audio signals with resampling

4.1. Record an audio signal with *L*=1s at a 48kHz sampling rate and 16-bit resolution (Lab 1)
4.2. Read the audio signal with the *wavfile.read* method from *scipy.io*. E.g.:

```
from scipy.io import wavfile
rate, audio = wavfile.read('/home/pi/WORK_DIR/audio.wav')
```

4.3. Resample the signal with poly-phase filtering at a frequency *f*=16kHz:

```
from scipy import signal
audio = signal.resample_poly(audio, 1, sampling_ratio)
```

4.4. Measure the execution time, store the output on disk and measure the file size:

- Cast the signal to the original datatype (int16):

```
audio = audio.astype(np.int16)
```

- Store the signal on disk with the *wavfile.write* method.
- Compare the sizes of the input and output wav with the *os.path.getsize* method.
- Play the original and resampled wav and comment the results.

Exercise 6: Pre-process audio signals with STFT

4.1. Read the audio signal from Ex. 5 with the *tf.io.read_file* method. E.g.:

```
audio = tf.io.read_file('/home/pi/WORK_DIR/audio_16.wav')
```

4.2. Convert the signal in a TensorFlow tensor using *tf.audio.decode_wav* method:

```
tf_audio, rate = tf.audio.decode_wav(audio)
tf_audio = tf.squeeze(tf_audio, 1)
```

4.3. Convert the waveform in a spectrogram applying the STFT:

- Use the *tf.signal.stft* method:

```
stft = tf.signal.stft(tf_audio,
                      frame_length=frame_length,
                      frame_step=frame_step,
                      fft_length=frame_length)
spectrogram = tf.abs(stft)
```

- Set the *frame_length* parameter such that l=40ms.

  Suggestion: $frame\_length = f \times l$

- Set the *frame_step* parameter such that s=20ms.

4.4. Measure the execution time needed to compute the spectrogram with the *time* method.

4.5. Store the spectrogram and measure the file size:

- Convert the tensor in a byte string with *tf.io.serialize_tensor*.
- Store the string on disk with *tf.io.write_file* method.
- Measure the size of the output file with the *os.path.getsize* method.
- Compare the size of the spectrogram with that of the original audio signal and comment the results.

4.6. Record multiple times the words “yes” and “no” and downsample the collected samples. Compute and visualize the STFTs as a grayscale PNG image for the different samples.

- Transpose the spectrogram to represent time on x-axis:

```
image = tf.transpose(spectrogram)
```

- Add the “channel” dimension:

```
image = tf.expand_dims(image, -1)
```

- Take the logarithm of the spectrogram for better visualization:

```
image = tf.math.log(image + 1.e-6)
```

- Apply min/max normalization and multiply by 255 (images are unsigned bytes):

```
min_ = tf.reduce_min(image)
max_ = tf.reduce_max(image)
image = (image - min_) / (max_ - min_)
image = image * 255.
```

- Cast the tensor to uint8:

```
image = tf.cast(image, tf.uint8)
```

- Convert the tensor to a PNG with the *tf.io.encode_png* method.
- Store the PNG on disk with the *tf.io.write_file* method.
- Comment the results.

Exercise 7: Extract the MFCC from audio signals

5.1. Load the spectrogram generated in Exercise 4.1:
- Read the byte string stored on disk with the *tf.io.read_file* method.
- Convert the byte string in a TF tensor with the *tf.io.parse_tensor* method (set *out_type=tf.float32*).

5.2. Compute the log-scaled Mel spectrogram with 40 Mel bins, 20Hz as lower frequency, and 4kHz as upper frequency:

```
num_spectrogram_bins = spectrogram.shape[-1].value
linear_to_mel_weight_matrix = tf.signal.linear_to_mel_weight_matrix(
            num_mel_bins,
            num_spectrogram_bins,
            sampling_rate, # 16000
            lower_frequency,
            upper_frequency)
mel_spectrogram = tf.tensordot(
      spectrogram,
      linear_to_mel_weight_matrix,
      1)
mel_spectrogram.set_shape(spectrogram.shape[:-1].concatenate(
            linear_to_mel_weight_matrix.shape[-1:]))
log_mel_spectrogram = tf.math.log(mel_spectrogram + 1e-6)
```

5.3. Compute the MFCCs from the log-scaled Mel spectrogram and take the first 10 coefficients:

```
mfccs = tf.signal.mfccs_from_log_mel_spectrograms(
            log_mel_spectrogram)[…, :10]
```

5.4. Store the MFCCs and measure the file size.
5.5. Compute the MFCCs of the "yes" and "no" samples. Visualize the MFCCs and comment the results.

Exercise 8: Pre-process Images

6.1. Capture a 640x480 JPG image with the PiCamera and store it on the disk (see Lab 1) or download a JPG image from the web.
6.2. Read the image with the *tf.io.read_file* method.
6.3. Convert the image in a TF tensor with the *tf.io.decode_jpeg* method.
6.4. Take a center crop of size 168x168 from the image with the *tf.image.crop_to_bounding_box* method.
6.5. Resize the image to 224x224 with the *tf.image.resize* method:
- Try different resize methods (bilinear, bicubic, nearest, area).
- Measure the execution time for each method.

6.6. Store the resized images on disk:
- Cast the tensor to uint8.
- Convert the tensors in JPG images *tf.io.encode_jpeg*.
- Store the JPG on disk with *tf.io.write_file* method.

6.7. Visualize the images and comment the results.