cpp_conversion

November 10, 2020

1 C++ Feature Engineering Implementation

In order to deploy our TensorflowLite converted model onto our Arduino Nano 33 BLE (pictured below) we must convert the code generating features from `python` to `C/C++` given that Arduino only supports using `C/C++` binaries. The conversion for the model is taken care of by TensorflowLite but the features we will have to generate ourselves.

Here, we will walk through how to take the raw audio sampled by our MEMS microphone and convert them into the input filter banks the model uses to make predictions.

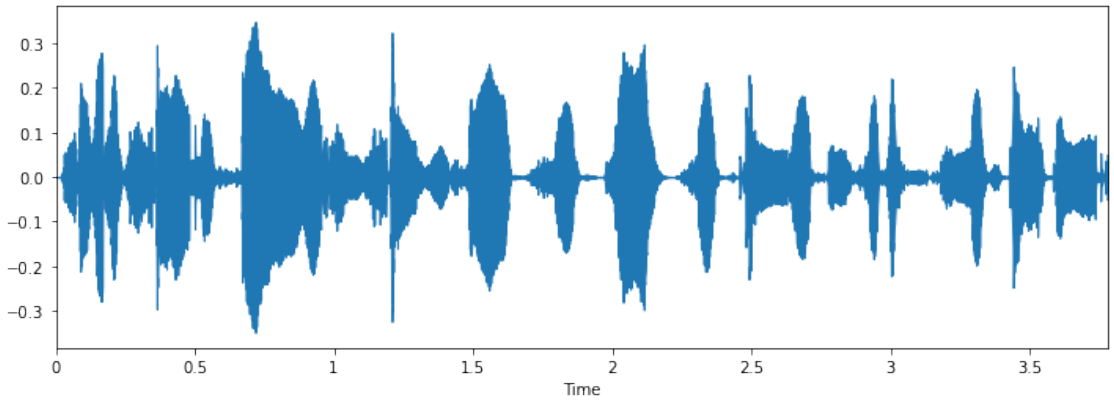
Note, that Arduino does not have native support for the extensive `C++` Standard Library so we will need to implement the feature engineering steps *without* the Standard Library in raw `C++`.

1.1 Setup

[7]:
```
import os

import librosa
import librosa.display
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal
```

[8]:
```
ROOT = "/home/thomas/Dir/ccny/ccny-masters-thesis"
```

[9]:
```
# need a file with a window of > 1.2 seconds from our extractor
AUDIO_FILE = f"{ROOT}/raw-data/LibriSpeech/dev-clean/6345/93302/6345-93302-0012.
 ↪flac"

y, sr = librosa.load(AUDIO_FILE, sr=None)
y_trim, _ = librosa.effects.trim(y, 20) # trim silence

plt.figure(figsize=(12,4))
_ = librosa.display.waveplot(y_trim)
```



[10]:
```
NFFT = 512
WIN_LENGTH = int(sr * 0.025)
HOP_LENGTH = int(sr * 0.01)
N_MELS = 40
```

[11]:
```
# write out for C++ implementations: our model expects 1.2 second long audio

signal_length = int(sr * 1.2)
y_trunc = y_trim[:signal_length]

out_path = f"{ROOT}/cpp/sample_wave.out"

with open(out_path, 'w') as stream:
    for sample in y_trunc:
        stream.write(f'{sample}\n')
    stream.close()
```

[12]:
```
def convert_cpp_file(path):
    frames = []
    with open(path, 'r') as f:
        for i, line in enumerate(f):
            window = line.rstrip().split(',')[:-1]
            frames.append([ float(val) for val in window ])
    return np.array(frames)
```

2 Framing

We begin with an audio sample that is 1.2 seconds long. With a sampling rate of 16000 (per second) that leaves us with $16000 * 1.2 = 19200$ samples. Over the length of that sample the frequencies will change. In order to build features, we'll make the assumption that over *short* frames in time the signal is **roughly** stationary. So, here we will chop the signal into frames of length `WIN_LENGTH`,

which is 25ms. We will then walk down the signal at `HOP_LENGTH`, or 10ms hops.

Below is our implementation of this procedure in `C++`. It is similar to the implementation of `librosa.util.frame`. Note that `librosa` performs reflective padding of `NTTT // 2` on either side of the signal, which we have also done.

```
float ** frame(float waveform[], int waveform_length, int win_length, const int hop_length, co

  // pad the waveform on either side with nfft//2 with reflection
  int wave_pad = nfft / 2;
  float padded_waveform[waveform_length + nfft];
  for (int l = 0; l < wave_pad; l++) {
    padded_waveform[wave_pad - l] = waveform[l];
  }
  for (int m = 0; m < waveform_length; m++) {
    padded_waveform[wave_pad + m] = waveform[m];
  }
  for (int r = 0; r < wave_pad; r++) {
    padded_waveform[wave_pad + waveform_length + r] = waveform[waveform_length - r - 1];
  }

  float** frames = new float*[NUM_FRAMES];

  bool pad_frame = nfft > win_length;
  int frame_length = pad_frame ? nfft : win_length;
  int offset = pad_frame ? (nfft - win_length) / 2 : 0;
  int start = 0;

  for (int i = 0; i < NUM_FRAMES; i++) {

    float* frame = new float[frame_length];

    for (int j = 0; j < offset; j++) frame[j] = 0.0;
    for (int k = 0; k < win_length; k++) {
      frame[offset+k] = padded_waveform[start+k];
    }
    for (int l = offset + win_length; l < frame_length; l++) frame[l] = 0.0;

    hamming(frame, nfft);
    frames[i] = frame;
    start += hop_length;
  }

  return frames;
}
```

3 Windowing

In order to deal with spectral leakage and potential numerical stability issues, we make use of different windowing functions once we have built our frames.

3.1 Hamming

One popular window function is the `hamming` function which is defined below,

$$w[n] = 0.54 - 0.46\cos(\frac{2\pi n}{N-1})$$

And our `C++` implementation takes the form:

```
void hamming(float window[], int window_size) {
    for(int i = 0; i < window_size; i++) {
        window[i] *= 0.54 - (0.46 * cos( (2 * PI * i) / (window_size - 1) ));
    }
};
```

3.1.1 C++

[67]:
```
# load the cpp file
cpp_ard_frames_file = f"{ROOT}/cpp/out/arduino/orig_frames.txt"

frames_cpp_ard = convert_cpp_file(cpp_ard_frames_file)
frames_cpp_ard.shape # [n_windows x window_length]
```

[67]:
```
(121, 512)
```

3.1.2 librosa

[68]:
```
y_padded = np.pad(y_trunc, int(NFFT/2), mode='reflect')
frames_lib = librosa.util.frame(y_padded, frame_length=WIN_LENGTH,␣
 ↪hop_length=HOP_LENGTH).transpose()
frames_lib.shape # [n_windows x window_length]
```

[68]:
```
(121, 400)
```

For a direct comparison to our raw C++ implementation, we have to `pad` and perform the `hamming` window transformation given that we placed those implementations in `frame`.

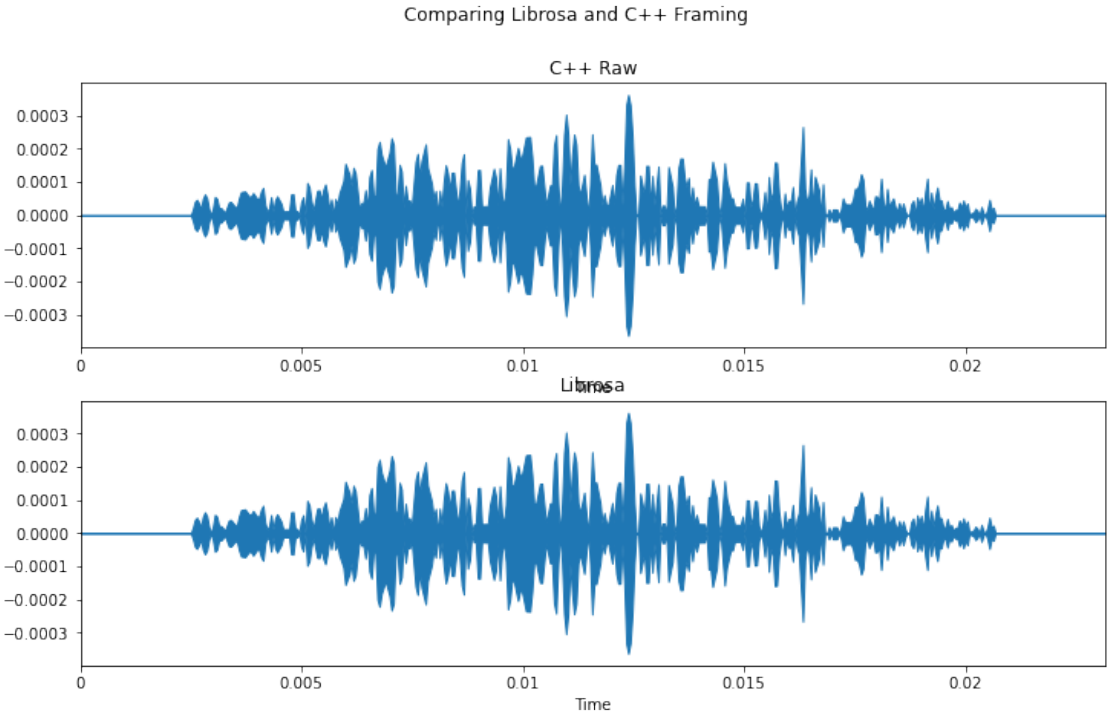
[69]:
```
# pad
padded_lib = librosa.util.pad_center(frames_lib, NFFT, axis=1)

# hamming
```

```
hamming = np.hamming(NFFT)
padded_hamming_lib = hamming * padded_lib
padded_hamming_lib.shape
```

[69]: (121, 512)

3.1.3 Plot

[70]:
```
idx = 1

fig, ax = plt.subplots(2, figsize=(12,7))
fig.suptitle('Comparing Librosa and C++ Framing')

librosa.display.waveplot(frames_cpp_ard[idx], ax=ax[0])
ax[0].set_title('C++ Raw')

librosa.display.waveplot(padded_hamming_lib[idx], ax=ax[1])
_ = ax[1].set_title('Librosa')
```



4 Short term Fourier Transform

So far, we have been working in the time domain. To generate meaningful, speaker specific, representations of our signal it wil lserve us to operate in the frequency domain. We can do this by applying a N-point Fast Fourier transform on the windows we generated above.

```
Complex ** stft(float ** windows, int num_frames = NUM_FRAMES, int frame_length = N_FFT) {

  Complex** stft_frames = new Complex*[num_frames];

  for (int i = 0; i < num_frames; i++) {

    Complex stft_frame[frame_length];
    for (int j = 0; j < frame_length; j++) {
      stft_frame[j] = Complex (windows[i][j], 0.0f);
    }
    fft(stft_frame, frame_length);

    // take only the LHS; b/c real-valued signal means this is reflection symmetric
    Complex* left_frame = new Complex[frame_length / 2 + 1];
    for (int k = 0; k < frame_length / 2 + 1; k++) {
      left_frame[k] = stft_frame[k];
    }

    stft_frames[i] = left_frame;
  }

  return stft_frames;
};
```

Note, here, that because real-valued signals are reflection symmetric we only need to save the left (negative) side. This behavior closely follows that of `librosa.core.stft`.

4.0.1 Spectrogram

After we've translated into the frequency domain via the FFT, we'll need to convert them into the power spectrum to generate spectrograms. We do this for power, P, as defined:

$$P = \frac{|FFT(x_i)|^2}{N}$$

where N is the `n_fft` or the number of windowed samples (potentially zero-padded) in our waveform. Again, this is exposed in `librosa` by means of `librosa.feature.melspectrogram`.

4.0.2 C++

[1]:
```
cpp_ard_spec_file = f"{ROOT}/cpp/out/arduino/spec_frames.txt"
spec_frames_cpp = convert_cpp_file(cpp_ard_spec_file)

spec_frames_cpp.shape # [n_windows x 1 + (nfft/2)]
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-1-5856154f6d9a> in <module>
----> 1 cpp_ard_spec_file = f"{ROOT}/cpp/out/arduino/spec_frames.txt"
      2 spec_frames_cpp = convert_cpp_file(cpp_ard_spec_file)
      3
      4 spec_frames_cpp.shape # [n_windows x 1 + (nfft/2)]


NameError: name 'ROOT' is not defined
```

4.0.3 librosa

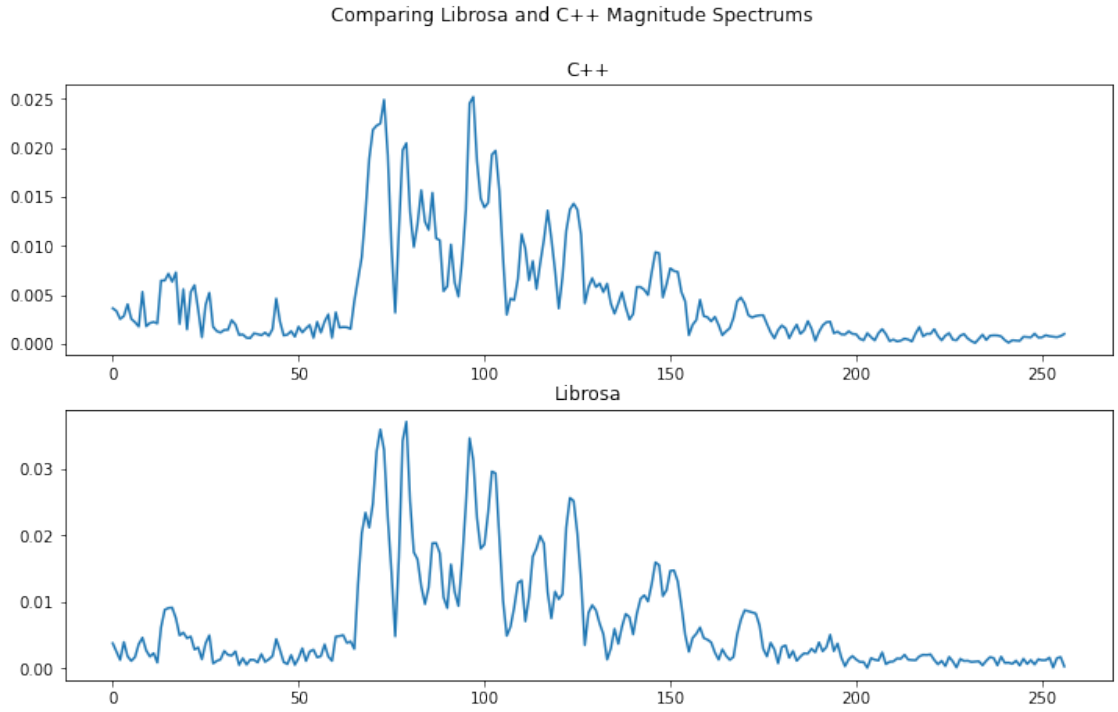
[81]:
```
stft_lib = librosa.core.stft(
    y_trunc,
    n_fft=NFFT,
    hop_length=HOP_LENGTH,
    win_length=WIN_LENGTH,
    window='hamming'
).T
spec_frames_lib = np.abs(stft_lib)
spec_frames_lib.shape # [n_windows x 1 + (nfft/2)]
```

[81]:
```
(121, 257)
```

4.0.4 Plot

[83]:
```
idx = 2

fig, ax = plt.subplots(2, figsize=(12,7))
fig.suptitle('Comparing Librosa and C++ Magnitude Spectrums')
```

```
ax[0].plot(spec_frames_cpp[idx])
ax[0].set_title('C++')

ax[1].plot(spec_frames_lib[idx])
_ = ax[1].set_title('Librosa')
```



5 Filter Banks

Finally, we'll apply triangular filters (here 40) on a Mel-scale to the above derived power spectrum to extract frequency bands. The mel-scale mimics the non-linear way in which humans hear, focusing more on the lower ends of the frequency spectrum.

We can generate the filters with the following

```
float * mel_filters(int nfilter, int sr = SIGNAL_RATE) {
  float low_freq = 0.0;
  float high_freq = (2595 * std::log10(1 + (sr/2)/ 700.0f));
  float step = (high_freq - low_freq) / (nfilter+1);

  float * filters = new float[nfilter+2];
  filters[0] = 0.0f;
  for (int i = 1; i < nfilter+2; i++) {
      filters[i] = filters[i-1] + step;
```

```
  }
  return filters;
}
```

and then build the filter bank matrix like so,

```
float ** filter_bank(int n_mels, int sr = 16000, int n_fft = 512) {

  // mel scale
  float * filts = mel_filters(n_mels, sr);
  mel_to_hz(filts, n_mels+2);

  // difference between mel steps
  float fdiff[n_mels+1];
  for (int i = 0; i < n_mels + 1; i++) {
    fdiff[i] = filts[i+1] - filts[i];
  }

  // FFT frequencies
  float fft_freqs[1 + n_fft / 2];
  float fft_freq_step = (sr * 1.0 / 2) / (n_fft / 2);
  float fft_freq = 0.0;
  for (int i = 0; i < 1 + n_fft / 2; i++) {
    fft_freqs[i] = fft_freq;
    fft_freq += fft_freq_step;
  }

  // outer subtraction: filts - fft_freqs
  float ramps[n_mels+2][1 + n_fft/2];
  for (int i = 0; i < n_mels+2; i++) {
    for (int j = 0; j < 1 + n_fft/2; j++) {
      ramps[i][j] = filts[i] - fft_freqs[j];
    }
  }

  // now build our filter bank matrix
  float ** weights = new float*[n_mels];

  for (int i = 0; i < n_mels; i++) {

    float * w = new float[1+n_fft/2];
    for (int j = 0; j < 1 + n_fft/2; j++) {
      float lower = -1.0 * ramps[i][j] / fdiff[i];
      float upper = ramps[i+2][j] / fdiff[i+1];
      float bound = lower < upper ? lower : upper;
      w[j] = 0.0 > bound ? 0.0 : bound;
    }

    weights[i] = w;
```

```
  }
  // Slaney normalize
  float enorm;
  for (int i = 0; i < n_mels; i++) {
    enorm = 2.0 / (filts[i+2] - filts[i]);
    for (int j = 0; j < 1 + n_fft/2; j++) {
      weights[i][j] *= enorm;
    }
  }

  return weights;
}
```

Then, given that we've generated a spectrogram, the final feature is just the dot product of our filter bank matrix and our power spectrogram.

5.0.1 C++

[13]:
```
cpp_ard_filter_banks_file = f"{ROOT}/cpp/out/arduino/filter_banks.txt"
S_cpp_ard = convert_cpp_file(cpp_ard_filter_banks_file)
```

5.0.2 librosa

[26]:
```
S = librosa.feature.melspectrogram(
    y=y_trunc,
    n_fft=512,
    win_length=int(sr * .025),
    hop_length=int(sr * .01),
    n_mels=40,
    window='hamming',
    sr=sr
)
S = np.log10(S + 1e-6)
```

5.0.3 Plot

[27]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16,6))
fig.suptitle('Comparing Librosa and C++ Spectrograms')

librosa.display.specshow(S, ax=ax[0])
_ = librosa.display.specshow(S_cpp_ard, ax=ax[1])
```

Comparing Librosa and C++ Spectrograms