Audio processing in TensorFlow

Dario Cazzani
Jun 30, 2017 · 6 min read

An implementation of the Short Time Fourier Transform

I found audio processing in TensorFlow hard, here is my fix

There are countless ways to perform audio processing. The usual flow for running experiments with Artificial Neural Networks in TensorFlow with audio inputs is to first preprocess the audio, then feed it to the Neural Net.

What happens though when one wants to perform audio processing somewhere in the middle of the computation graph?
TensorFlow comes with an implementation of the Fast Fourier Transform, but it is not enough.

The code

All the code is available on my **GitHub**: Audio Processing in Tensorflow.
Feel free to add your contribution there.

Audio preprocessing: the usual approach

When developing a Speech Recognition engine using Deep Neural Networks we need to feed the audio to our Neural Network, but… what is the right way to preprocess this input?

There are 2 common ways to represent sound:

- *Time domain:* each sample represents the variation in air pressure.
- *Frequency domain*: at each time stamp we indicate the amplitude for each frequency.

Despite the fact that Deep Neural Networks are extremely good at learning features automagically, it is always a good idea to rely on known features that carry the information needed for the task that we are trying to solve.

For most application, a Speech Recognition Engine included, the features we are interested in are encoded in the frequency domain representation of the sound.

The spectrogram and the Short Time Fourier Transform

A spectrogram shows how the frequency content of a signal changes over time and can be calculated from the time domain signal.
The operation, or transformation, used to do that is known as the Short Time Fourier Transform.

I could let the Neural Network figure out how to learn this operation, but it turns out to be quite complex to learn with 1 hidden layer. (refer to the Universal approximation theorem)

I could add more layers, but I want to keep the complexity of the Neural Networks as small as possible and learn features only where it is most needed.

tasks involving non-speech audio like noise reduction, music genre classification, whale call detection, etc.

A particular project that I want to mention is Magenta, from the Google Brain team, who's aim is to advance the state of the art in machine intelligence for music and art generation.

Why TensorFlow?

I mainly use TensorFlow when implementing Artificial Neural Networks and, because I haven't found an implementation of the Short Time Fourier Transform in TF, I decided to implement our own.
[edit: June 4th 2018] — since TensorFlow 1.3 they added some useful DSP functionalities.

There can also be multiple reasons why a deep learning practitioner might want to include the Short Time Fourier Transform (STFT for my friends) in the computation graph, and not just as a separate preprocessing step.

Keep in mind that I haven't focused on making this efficient. It should (and will) be improved before being used in production.

What you need to know

In order to understand how the STFT is calculated, you need to understand how to compute the Discrete Fourier Transform.

Discrete Fourier Transform — DFT

This part can appear quite technical for those who are not familiar with these concepts, but I think it is important to go through some maths in order give a complete understanding of the code.

Theory

Fourier analysis is fundamentally a method for expressing a function as a sum of periodic components, and for recovering the function from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT).

Given a vector $\boldsymbol{x}$ of n input amplitudes such as:

the Discrete Fourier Transform yields a set of *n* frequency magnitudes.

The DFT is defined by this equation:

DFT equation

- k is used to denote the frequency domain ordinal
- n is used to represent the time-domain ordinal
- N is the length of the sequence to be transformed.

Fast Fourier Transform
The Fast Fourier Transform is an efficient implementation of the DFT equation. The signal must be restricted to be of size of a power of 2.

This explains why N (the size of the signal in input to the DFT function) has to be power of 2 and why it must be zero-padded otherwise.

One can detect whether x is a power of 2 very simply in python:

We only need half of it
Real sine waves can be expressed as the sum of complex sine waves using Euler's identity

Because the DFT is a linear function, the DFT of a sum of sine waves is the sum of the DFT of each sine wave. So for the spectral case you get 2 DFTs, one for the positive frequencies and one for the negative frequencies, which are symmetric.
This symmetry occurs for real signals that can be viewed as an infinite (or finite in our case) sum of sine waves.

Windowing
Truncating a signal in the time domain will lead to ripples appearing in the frequency domain.
This can be understood if you think of truncating the signal as if you applied a rectangular window. Applying a window in the time domain results in a convolution in the frequency domain.
The ripples are caused when we convolve the 2 frequency domain representations together.

Find out more about spectral_leakage if you're interested.

Here is an example of an implementation of windowing in Python:

Zero-phase padding

In order to use the FFT, the input signal has to have a power of 2 length. If the input signal does not have the right length, zeros can be appended to the signal itself both at the beginning and at the end.
Because the zero sample is originally at the center of the input signal, I split the padded signal through the middle and swap the order of these 2 parts.

The next code snippet shows how to do this in TensorFlow for a batch of inputs:

FFT, Magnitude and Phase
You now have everything you need to calculate the magnitude of the spectrogram in decibels and the phase of the signal:

Short Time Fourier Transform

content varies with time.
You can do this by:

1. Taking segments of the signal.
2. Windowing those out from the rest of the signal, and applying a DFT to each segment.
3. Sliding this window along each segment.

You get DFT coefficients as a function of both time and frequency.

The complete code is divided in 2 parts: *helpers.py* and *stft.py*.

Conclusion

The possibility of doing the STFT in TensorFlow allows Machine Learning practitioners to perform the transformation of a signal, from time-domain to frequency domain, anywhere in the computation graph.
New tools always bring new ideas and we hope this post will be the source of new ideas for developing new Deep Learning solutions.

Thanks to Matt Norris, Joshua Frattarola, and Andy Payne.