EE113 Digital Signal Processing

Spring 2020

Final Project

Due: Monday, June 8 at 4pm

Instructor: Prof. Flavio Lorenzelli

TA: Alexander Johnson and Mandar Deshpande

May 11, 2020

Total: 100 points

Note:

- Students should work in groups of up to 4 students
- Students should not share work with students outside of their group.
- Students taking the course for Letter Grade need to solve all 4 problems.
- Students taking the course for Pass/No Pass grading need to solve only Problem 1 and 2
- Questions 1 and 2 can be done in MATLAB or Python. Questions 3 and 4 must be done in Python

Problem 1. ***(20 points)***
In this section we will implement an audio signal transformation pipeline for feature extraction. Usually when we discuss speech processing tasks discrete Fourier transform (DFT) as the features instead of using just the audio data vector.

For the purposes of this project, we will be using the chords data set containing 2000 chords split up in 10 classes, giving up to 200 chords per chord type. The files are stored in raw WAV 16 bits mono 44100Hz format. You can access the dataset - here
You have to perform the follow tasks:

(a) *Use the audio clip data to extract DFT features. Plot the DFT for one chord from each chord type (10 in total). What do you observe?*

(b) *Create a function to take the cross correlation any unknown chord type with existing data you have for each chord to detect which chord it belongs to. Remember that the cross correlation is equivalent to the convolution with one of the signals reversed in time:*

$$crosscorr(x[n], h[n]) = convolve(x[n], h[-n])$$

Walkthrough: *If we have A is an A chord, C is a C chord, E is an E chord, and U is a chord of unknown type then we would expect*

$$U = argmax_s \frac{1}{N} ||crosscorr(U, s)||^2$$

where

$$s \in [A, C, E]$$

and N is the length of the signal vector crosscorr(U,s).

In other words, a signal should resonate with itself, so the cross correlation of a signal with itself should have higher energy than the cross correlation of itself with any other signal of the same energy. You could pick one recording of each type to be the wavelet and perform this over the remaining signals. You should also normalize the energy of the wavelets. Take the cross correlation of every signal with every wavelet, and classify the signal as the same chord type of the wavelet that produces the highest energy cross correlation.

(c) *Compare the test audio files provided with the given 10 chords dataset to determine the chord type using the convolution operation. Create a matrix in which each column and each row corresponds to a chord of a given type. For each test case, if a test case is actually of chord type j and is predicted by your implementation as chord type i, add 1 element (i,j) in the matrix. The more the confusion matrix looks like a diagonal matrix, the better the prediction algorithm is. This is called a confusion matrix. Use imshow(), imagesc(), or another analogous command to plot the confusion matrix as an image. Be sure to label the axes.*

From this section, you must turn in:

- *All code*
- *A plot of the DFT of a single chord from each types and your observations on each*
- *The results of your code given as a confusion matrix. Along with some analysis of your results*
- *A suggestion for improving your accuracy.*

Problem 2. ***(30 points)***
In this section, we will implement a simplified version of the Griffin-Lim Algorithm (Griffin, Lim 1984). The goal of this algorithm is to attempt to recreate a time domain signal from only the magnitude of its frequency response. Ordinarily, it is impossible to solve for a signal exactly given only its magnitude response. For example, if you are told that a vector x has a magnitude of 10 and contains 2 elements, you do not have enough information to give those elements exactly. However, the Griffin-Lim algorithm makes assumptions about phase to come up with a plausible phase for the signal. Several signal processing algorithms throw away the phase information of the input signal, taking only the spectral magnitude information. We may wish to reconstruct the original signal even if the phase information is not available. Other times, we may want to synthesize a new signal and know its desired magnitude spectrum. The Griffin-Lim algorithm can then create a phase portion of the frequency response with which to construct the signal. We will implement a simplified version of the algorithm as follows:

(a). *Read in a signal and take its DFT*

(b). *Take the magnitude of its DFT*

(c). *Initialize a phase for the signal as random samples from a normal distribution on the range*

$$[0, 2\pi]$$

(d). *Combine the original magnitude information with the phase information*

(e). *Take the IDFT of the newly created Fourier Transform*

(f). *Take only the real part of the newly created signal*

(g). *Take the phase of the new signal*

(h). *Recombine the new phase with the original magnitude*

(i). *Repeat steps (e)-(h) for a given number of iterations*

The idea of the Griffin-Lim Algorithm is that we make a guess about the phase of the signal and then continually update that guess with the information known about the signal: the signal has the magnitude we extracted from it, and the signal is real. We can think about these givens as constraints that define a feasible region for the original signal. We then seek to make a guess for the phase and *update it by projecting it onto the feasible regions. Here, we realize the projection more simply by assigning the known magnitude to the signal and then taking the real part of the reconstructed signal. The original Griffin-Lim Algorithm guarantees convergence towards the to original phase with enough iterations. In processing audio here, we take advantage of the fact that humans are said to be "phase deaf." That is, we can only detect relative phase differences and inconsistencies, but cannot tell if the phase of a signal is globally shifted. This means that we need not converge on the original phase but rather just a phase that is plausible enough to sound passable to a human listener. Psuedocode for the algorithm can be written as follows:*

```
$orig\_mag \leftarrow abs(dft(signal))$
$phase \leftarrow 2\pi * randn()$
$for\ iter\ in\ iterations$
    $new\_ft \leftarrow orig\_mag * exp(1j * phase)$
    $new\_signal \leftarrow idft(new\_ft)$
    $new\_signal \leftarrow real(new\_signal)$
    $new\_ft \leftarrow dft(new\_signal)$
    $phase \leftarrow angle(new\_ft)$
$end\ for$
```

Task 1: Implement this code with the inception.wav audio file provided. You can discard one channel of the audio and perform the algorithm on only one channel for 100 iterations. Save the output to a .wav file While efficiency is not a requirement, your code will take significantly longer to run if you choose not to optimize. Doing this, we run into a problem. Describe how this output audio file differs from the original. We can think of the magnitude as containing information on what frequency components are present in the signal and the phase as containing information on when the components appear in the signal. While this is a gross oversimplification, it does illustrate the point that we cannot accurately reconstruct the signal from the magnitude taken across its entire duration. To remedy this, we turn to the Short Time Fourier Transform (STFT). This process divides the signal into smaller segments, or windows, and takes the DFT of each window individually. If we take the magnitude of small enough segments over which there is little change in the sound quality, then we will be able to pinpoint how each individual segment should sound. Use the built in STFT function to take the STFT of the input signal. Modify the code as such:

```
$orig\_mag \leftarrow abs(stft(signal))$
$phase \leftarrow 2\pi * randn()$
$for\ iter\ in\ iterations$
    $new\_ft \leftarrow orig\_mag * exp(1j * phase)$
    $new\_signal \leftarrow istft(new\_ft)$
    $new\_signal \leftarrow real(new\_signal)$
    $new\_ft \leftarrow stft(new\_signal)$
    $phase \leftarrow angle(new\_ft)$
$end\ for$
```

The output of the stft is a N_windows by window_size matrix where N_windows is the number of windows whose Fourier Transform were taken and window_size is the size of the window used. We can also define the additional parameters overlap (or hopsize) and window type. When we take windows of the signal, we can sometimes get errors at the boundaries of the windows since we abruptly cut the signal there. That is why it can be advantageous to include some of the points at the edge of the window in both the given window and the next window. This means that we slide over the window by an amount less than the window size in order to ensure redundant encoding of the boundary points as shown in the figure.

The amount that we slide over the window from one from to the next is called the hop size. The number of points which are shared between two adjacent windows is called the overlap. Either hopsize or overlap can be defined in terms of the other given the window size. Including the same point twice due to overlap gives that point more weight in the Fourier Transform. That is (one of the many reasons) why we typically attenuate the outer points of each window by multiplying it by a function that is higher towards the center values and lower towards the outer values. This can be thought of forming the Fourier Transform as a weighted average of different frames. Task 2: implement the algorithm updated with the stft. Experiment with different window sizes, window types, and overlap sizes in your implementation and report the best one. Then run your code for 500 iterations and save the output to a wav file at 1, 50, 100, 250, and 500 iterations. Take the phase of the stft of the original signal and the phase at 1, 50, 100, 250, and 500 iterations and find the magnitude of the difference between the matrices. We calculate the magnitude of a matrix here by the square root of the sum of the square of all elements in the matrix. For a matrix A, the magnitude is

$$A = \sqrt{\sum_i sum_j |a_{i,j}|^2}$$

where

$$a_{i,j}$$

is the element in row i and column j of A. This term is commonly called the Frobenius norm of a matrix. In other words, we are calculating the Frobenius norm of the difference between the phase of the STFT matrix of the original signal and the phase from the reconstructed STFT matrix Then plot the magnitude of the phase difference vs the number of iterations. What do you notice? Can you explain this?

From this section, you must turn in:

(a). Your code from Task 1

(b). The output of the code from Task 1 as a .wav file

(c). A short explanation of how the output of task 1 differs from the original audio and why

(d). Your code from Task 2

(e). The output of the code from Task 2 as a .wav file for 500 iterations

(f). A plot of the norm of the difference between the original phase of the signal and the reconstructed phase across the iterations. Also include your observations on how the graph looks and explanation for why it appears that way and whether or not the phase difference reflects the perceived audio quality.

Problem 3. ***(20 points)*** *We now are able to determine the class of any unknown chord using traditional methods in signal processing described in Problem 1.*

In this section we are trying to solve the chord classification task using deep learning. Use the features generated earlier (eg DFT) to train a fully connected neural network.

Use Keras to create a dense neural network and make the output of the network to be the one hot encoded labels. We have already setup the network to be used for this Problem in the shared Jupyter Notebook.

The basic pipeline for data loading has been setup as follows:

- *Read in all audio files*
- *Find the length of the longest audio clip, if you find they are not of the same length, and call it N.*
- *Create a padding function, which appends zeros to all audio files such that all have an equal length N*
- *Create labels for each audio file according to the respective chord name and use encode as one-hot vectors.*

The neural Network will accept the input as the audio file, or features from the audio file and then map them to the chord labels represented as one-hot encoded vectors.

Your task is to use the extracted features from Problem 1 and find out the binary chord classification for the set of 2 chords provided.

Focus is on experimentation and not on accuracy of results. By following this pipeline you would be able to compare and contrast the traditional method with a machine learning based approach.

(a). Determine the class of test audio files in Problem2_test.zip using the fully connected network for binary classification.

(b). Add white Gaussian noise to all of the test data at signal to noise ratio of 5dB, 0dB, -5dB, etc and evaluate the performance. Implement an algorithm to improve this performance. You may choose to perform enhancement on the noisy signal or come up with a noise robust feature extraction process. The snr of a noisy signal is given by

$$SNR\ db = 10log_{10}(\frac{signal\ power}{noise\ power})$$

This means that we need to choose the noise to have power:

$$noise\ power = \frac{signal\ power}{10^{SNR\ db/10}}$$

To achieve noise of the desired SNR, we can first create random Gaussian noise using the np.random.normal() command and then give the noise the desired noise power given the power of the signal. We can take the power of the signal as the mean of the square of its elements. We can then divide the noise by its mean amplitude and multiply it by

$$\sqrt{\frac{signal\ power}{10^{SNR\ db/10}}}$$

to give it the desired power level. Finish the function awgn() in the Python notebook to add white Gaussian noise to a signal and perform the classification on the noisy signals.

(c). Here, we add zeros to the end of each audio file so that they are all the same size. Suggest and implement a better method of length normalization. Repeat part 3(a) for the new method.

(d). Highlight any specific advantages you see in the deep learning based approach when compared with the traditional methods.

Problem 4. ***(30 points)*** *We now perform a substitute for the Griffin-Lim Algorithm using a Convolutional Neural Network (CNN). Simply put, given an input signal x and an output signal y, a CNN attempts to find the kernel h such that*

$$x * h = y$$

. We give the network a large number of signals as training data, and it solves for the single kernel h that best minimizes the difference between the actual y given and the approximation of y calculated as

$$x * h$$

. As a dataset, we use a portion of the Carnegie Mellon University Arctic Speech synthesis database included. To carry out the mapping approximation, we may need to split up this process into several smaller processes in order to reasonably be able to solve. For example, we may need to individually find h_1*,*h_2*,...,*h_n *such that* $(((h_3 * (h_2 * (h_1 * x))) = y$*. However, properties of convolution tell us that* $(((h_3 * (h_2 * (h_1 * x))) = (h_3 * h_2 * h_1) * x = h_{eq}xx$*. That is, any number of successive convolutions can be simplified to a single convolution. Our network will not gain any ability to better approximate a complicated input output relationship if convolutions are stacked in this way. That is why we introduce nonlinearity into the approximation:* $((f_3(h_3 * f_2(h_2 * f_1(h_1 * x))) = y$ *where* $f_1(), f_2(), \ldots, f_n()$ *are non LTI functions such as upsampling, downsampling, and taking the absolute value, that are applied to the result of a convolution. Doing this allows our network to learn around the functions and approximate a mapping that is more complex than a convolution alone can achieve. Here, we will use the magnitude of our stft as the input x and the phase of our stft as our output y. We can then attempt to create a mapping from magnitude to phase as done in the deterministic Griffin-Lim Algorithm. A CNN is set up for you in the part2b.ipynb file. Although we have mainly covered 1D signals in this class, a convolution is in fact defined for higher dimensional signals (like images and videos). Therefore, we need not restrict ourselves to making x or h one dimensional. We feed in the 2D stft magnitude as x and can even choose any h to be 3D or higher if desired.*

Although you are free to do so, your job is not to change the network here. A neural network has a limited complexity, meaning that there are some functions too difficult or complicated for it to learn. Then we can change the network architecture to allow it to learn a more complicated function. We might also attempt to make our input-output relationship simpler. To attempt this, we will use the method of filterbanks. Filterbanks divide the signal into regions of different frequencies. For example, from one signal bandlimited at 4000Hz, we may divide it into 4 signals: one with all components above 1kHz lowpass filtered out, one with all components below 1kHz and above 2kHz bandpass filtered out, one with all components below 2kHz and above 3kHz bandpass filtered out, and one with all components above 3000Hz highpass filtered out. The diagram below shows an example of a filterbank. The signal x[n] is passed through various filters that eliminate all but a select section of the frequency response of the signal. Then an operation is performed on each of the individual frequency bands. After that, the bands are added back together to form a new output signal.

Your job is to break the input signals into at least three frequency bands. Then train a different network for each of the bands as shown. The hope is that the relationship between magnitude and phase is easier to model over just one frequency band as opposed to over the whole spectrum. The parameters of implementation (like cutoff frequency, etc.) of this are up to you. You should

(a). Break up the input signal into bands

(b). Take the stft of each band using the best parameters found in section 1b

(c). Feed the stft of each band into a new network

(d). Predict the phase component of a test signal based on the training for each band

(e). Recombine the magnitude and phase for each band

(f). Recombine the bands

(g). Reconstruct the signal

(h). Play the output

We've actually already learned enough to implement a bandpass filter manually. In homework, we looked at the equation for a lowpass filter with cutoff frequency f1. If the BP(f1, f2) is a bandpass filter that cuts out frequency outside of the range f1 to f2 then we can write BP(f1,f2)=LP(f2)-LP(f1) where LP(f) is a lowpass filter with cutoff frequency f. You can choose to implement these bandpass filters yourself using the time domain formulas given with convolution or use python built in functions. The goal here is to show that different frequency bands may need to be processed differently. We have learned enough in this class to do this. Note that you are graded on your implementation and design decisions, not your performance. We would require more knowledge about machine learning to be able to optimize for better performance. However, you have the signal processing tools to be able to construct an optimal input for the neural network.

From this section, you must turn in:

(a). Your completed Python notebook with your implementation of the filter bank decomposition

(b). An explanation of how you chose the cutoff frequencies for your filter banks.

(c). An explanation of any other design choices implemented if any

(d). The .wav file containing your best result