TensorFlow Signal Processing

Joel Shapiro

problem overview

The objective of this project is to evaluate the effectiveness of doing audio effect emulation using deep learning. For audio, there are two main classifications for tools: generators and effects. A generator is something which takes non-audio input, either physical or midi, and creates audio out of it. This would include instruments, synthesizers, drums - basically anything that really stands out as being traditionally musical. The other category, effects, are elements which take audio as input and transform it into some other audio to output. This can range from a simple filter to more complex effects such as distortion or reverb; even the echo of a room or quality drop over a phone call is an effect. The idea behind this project is to see if we can't train a network to emulate one of these effects using deep learning.

Audio is an interesting medium to work in for machine learning, as like image data the output can be judged both quantitatively and qualitatively. On top of this, audio itself is a complex structure: the additive property of waves can cause some unforeseen outcomes. On top of that, digital audio data is inherently convoluted: it is stored as a time series of points which are sampled from the audio signal itself. These points are fast Fourier transformed back into the signal whenever the audio is ready to be output. Because of this, a lot of the information which is affected by effects is hidden behind this signal processing problem.

In the past, doing signal processing in machine learning involved doing some manual decomposition of the input in order to abstract away the signal processing [1]. Often audio would be rendered into images of the spectrogram, which show the frequency distribution of the audio. While this works well for classification problems, it lacks accuracy for end to end regression problems like ours. For that, we need to do actual signal processing in order to detect the features that matter.

The current progress on this project is available at github.com/jshap70/TensorFlow-Signal-Processing

sample types and why we care

Previously I mentioned how audio is a conceptually complex structure. This is because audio data is time series data of the amplitude of the audio, however almost all of the information that we think of as being "stored" in the sound is stored in the frequency space of the sound. The relationship between the two is extracted by using a Fourier transform. An example can be seen below, where the time series data on the left would produce the frequency chart on the right.
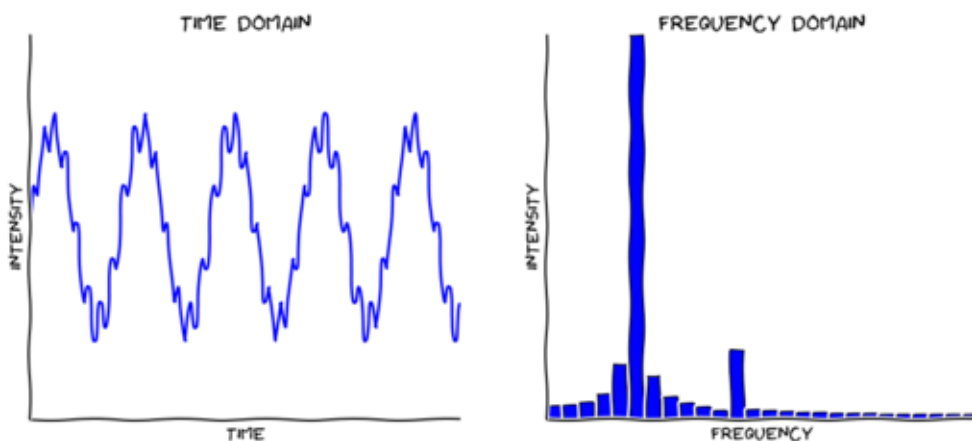

[2]

However, this is an oversimplification. In reality, the frequency chart is adding a dimension to the data, so the frequency chart above is losing the time data of the sound. That frequency chart is only true for a small time cross section of the audio. A real frequency distribution of the sound would look as such.

Frequency-domain plot of a sustained note (220 Hz) bowed on a cello
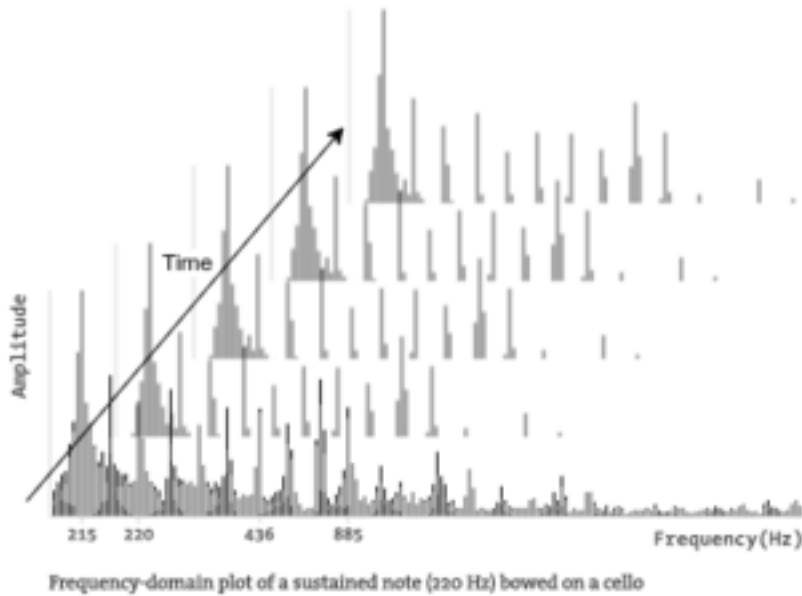
[3]

And in fact this is what most machine learning uses to train audio on, except instead of having a height in the amplitude dimension they use image channels and color intensity to represent it. This type of representation is called a Spectrogram. Spectrograms actually store 3 dimensional data, with frequency shown in the vertical direction, amplitude shown as color intensity, and time shown along the horizontal axis. You can see an example below.
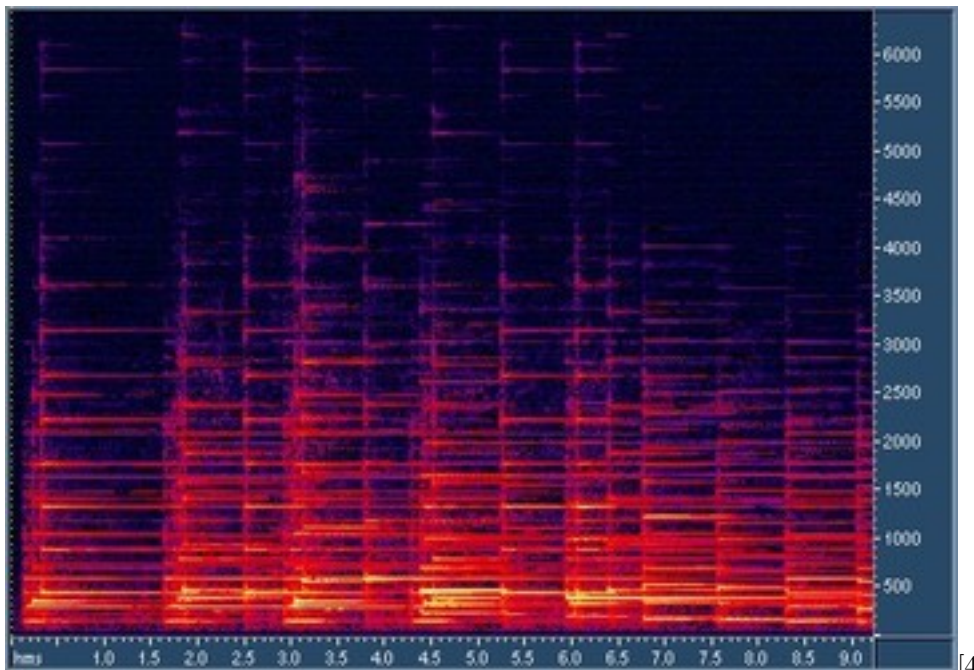
[4]

That is why the goal of this project is to attempt to have the network learn the frequency-amplitude relationship on it's own, so that we can skip the step which manually extracts the important features.

Digital audio data is stored as sampled points from the amplitude vs time graph, which is to be expected given that it's the direct form -albeit with a Fourier transform- that the output needs to be. A basic example can be seen below.

2
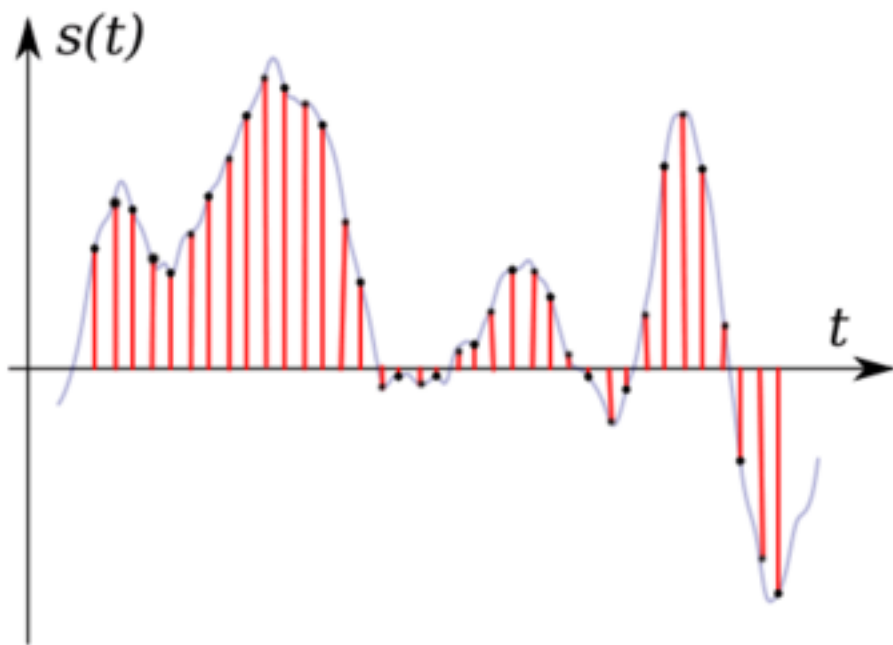
The audio used to train this project is mostly composed of some simple, generated audio samples which covers a varying types of sound. On the more simple side, we have simple sine, triangle, and saw waves that move through a frequency range. More difficult samples include piano recordings and voice data. The scope of this project was only on simple effects because of the time and resources available; however, it would be interesting to see the impact filter complexity has on training difficulty.

The audio used in this project has a uniform sample rate, which allows us to batch it easier.

the network

The plan to teach the network how to interpret the audio data needed to address 2 main concerns: first, it needed to be able to look at the audio and extract frequency data from it, and second it needed to be able to "undo" this operation so that the data could be converted back into audio.
As far as the first problem is concerned, it's possible for us to add time as a dimension to the audio data similar to the frequency spectrogram concept above. In that model, time is represented as part of the image by being one of it's axis. In this way, the 2 dimensional instantaneous frequency plot becomes a 3 dimensional image. For our data, we have a 1 dimension of data: amplitude. By adding time as a dimension to this data, by batching it in contiguous time chunks, we can attempt to expose the network to patterns in the data. Or at least that's the idea.
The second major issue deals with making the system end-to-end. We are looking to be able to take the output of the network, write it to a file, and play it back without having to take any extra steps. For a linear or nonlinear system, this is about as difficult as it is accurate - which is to say not very. However, for a convolutional network which is introducing extra depth in the network, it's necessary to have a convolutional transpose layer. This type of layer is sometimes referred to as a 'deconvolutional' layer, however it's important to note that this is actually a misnomer, as deconvolution is a completely different process related to computer vision. Regardless of the terminology, a convolutional transpose layer allows you to take layers which have been convolved and attempt to separate out the data back into more meaningful data - in our case, back into the amplitude graph.
With this in mind, we'll move on to the main design.

layer design

Intuitively, it would make sense that a standard linear network would most likely not be able to properly model this problem. The data is probably too complex for it to interpret it. However, I still wanted to form a baseline to see just what kind of benefit we would achieve by moving to a more advanced network.

So to start, I used a standard fully connected regression neural network, varying the depth of the hidden layer to find something that seemed reasonable to train. The goal of this network was to try to overfit the training data to show that it can at least be brute forced. And boy did I have to brute force it. With the standard training set I was using, these networks were taking upwards of 4,000 epochs to train.

Moving past the basic networks, it seems somewhat intuitive that this problem would be decently well represented by a convolutional network because of it's ability to attempt to train filters on sections of the data. If these filters are large enough to detect full oscillations, it may be able to extract some relevant frequency data. As mentioned previously, any time we use a convolutional layer we will have to use a convolutional transpose layer on the output. So currently we've built up a system which should be able to look at the data in a fashion which is potentially more true to the frequency space of the data. Now, all we need to do to finish off this basic setup is to place a fully connected layer in between the convolution and convolutional transpose layer

// maybe we could do multiple filter sizes together?

data sampling and batching

Looking at the data itself, the wav files are stereo 16bit PCM (integer)) files. To begin with, I converted the data to a 32bit float wav file and normalized the audio to fit within that standard. I split apart each file into mono tracks because it allows for us to experiment with different network designs a lot faster. However, there are filters which have different effects across stereo channels, so we will lose the ability to train on those for now.

The audio we are training on is a set of sine, square, and saw waves which travel through a range of requencies. Although these waves are very basic, the idea is that the simpler wave shapes might help to train the network to understand frequency analyis easier. The validation data is extracted from the same dataset as the training, but the testing data is entirely different. It uses the same filter, but it is a recording of a piano being played through it. The idea is that this is a much more complex wave shape, so it will be a better test of the network.

Because it is time series data, the batching process is a bit trickier. Although the data needs to be kept in contiguous chunks, we can still extract the smaller sections of it to train independently on to ensure the network is trained uniformly. To do this, I implemented a batching system that does a scrolling window selection of the audio for discrete time periods, and then I shuffle those batches for every epoch. If we set the offset of each window to the one next to it is smaller than the length of each window, then we can get some overlap in the windows to further increase the number of available batches.

*Side note: It might seem at first that we would want to take cuts of the data at small enough intervals to only allow for a handful of oscillations in the data. This might ensure that the net would get as close as possible of an idea of the instantaneous frequency data. But in reality this wont work. The issue is that the length of an oscillation is directly the result of pitch, so if the pitch changes the window might then cut off parts which are needed to extract the data. This is another reason why we must rely on the convolutional filters to slice the data for us.

results

Before we look at the netwroks themselves, lets look at the expected input and output.

input:

expected output:

First up, the results of the linear network.

```
x, y, P, MSE, sess = run_lin(1000, 4000)
run_test(x, y, P, MSE, sess, run_name='best_linear')

  starting from epoch: 4000
```

```
                  mse                          rmse                               std
 training  validation      training  validation      training  validation      reference
   0.00342      0.00327       0.05847     0.05722       0.05885     0.05723        0.10412


   test mse:  0.00118764
   test rmse: 0.0344621
   test std:  0.0344703680322
```

. . . They're bad. Really, really bad.

Here's the

Firstly, I found the network was only accurate with around 1000 nodes, which in and of itself poses a number of issues. It took 4000 epochs to train the data, which even then was just barely able to overfit on the training data. An rmse value that is only half of the standard deviation of the input set is not terrible, but for training we really expect that to be lower. On top of that, it took almost 5 hours to train this network, and for that time I wasa hogging as much of the math department's server as I could.
Interestingly

There are a couple of possible reasons why the validation error could be equally as low as

future plans

// maybe we could do multiple filter sizes together?

// once we've trained the convolutional layers responsible for converting to and from raw audio data, maybe re-use them to train on another filter so we can focus more on training the middle layers

// RNN's and how they might help with time series data

Notes / Sources

[1] At least this is true for most practical applications. An example can be seen here: github.com/markostam/audio-deepdream-tf

[2] Image showing the relationship between time series and frequency data. Source: learn.adafruit.com/fft-fun-with-fourier-transforms/background

[3] This image is heavily modified from the source, but still it originally came from: processing.org/tutorials/sound/

[4] Spectrogram image from: dwutygodnik.com/artykul/673-uwaznosc-fraktale-spektra-modele.html

[5] Image showing how digital audio data is stored. Source: progulator.com/digital-audio/sampling-and-bit-depth/ However, note that there are some very large errors in this article. Most importantly, it incorrectly does not cover how Fourier transforms are used to go from the digital point sampling back to the analog signal and makes the common fault of believing the data is just directly interpreted as an averaging operation.

[misc]
some audio midi from - http://www.piano-midi.de/brahms.htm
wavio - https://github.com/mgeier/python-audio/ - by Warren Weckesser