236781 – Final project

(TODO) Experimenting with DDSP, by Tal Skverer and Amit Zukier

# Abstract

Summarize your work. Briefly introduce the problem, the methods and state the key results.

TODO

# Intro

Review the papers relevant to your project. Explain the problem domain, existing approaches and the specific contribution of the relevant paper(s). Also detail the drawbacks which you plan to address. If it’s a custom project, explain your specific motivation and goals. Cite any other work as needed.

Differential Digital Signal Processing (or DDSP for short), is a library which enables the addition of signal processing elements into TensorFlow[[1]](#footnote-1), a well-known library with modern automatic differentiation.

This work was done to fill in the gap in neural networks training for audio generation. While most generative models (such as WaveGAN[[2]](#footnote-2), SING[[3]](#footnote-3), MCNN[[4]](#footnote-4), WaveNet[[5]](#footnote-5), GANSynth[[6]](#footnote-6) and others) directly generate samples in either the time or frequency domains (and sometimes both of them), DDSP integrates classic signal processing elements (synthesizers and effects), to improve neural networks’ approximation by using the strong structural priors of these tools, which promotes generalization. Specifically, unlike the works mentioned before, using said DSP elements might successfully incorporate their ability to convey audio, as they align with the data domain.

This is, as explained in the paper, because these elements exploit the periodic structure of resonating, similarly to how the human ear has evolved, unlike other audio synthesis models.

For example, generative models such as WaveGAN generate waveforms directly. Since audio usually includes many frequencies, the model must generate aligned waveforms, included with every filter applied to them, which is generally very challenging. Using a harmonic oscillator (called an additive synthesizer) in a neural network eliminates this issue by automatically outputting a signal with several sinusoidal components at harmonic frequencies (integer multiples of a fundamental frequency).

More issues covered in the original paper are of Fourier-based models such as GANSynth which suffer from spectral leakage problem, and autoregressive waveform models such as WaveNet which bypass all aforementioned issues by generating the waveform a single sample at a time. However, this causes them to require larger networks to learn this complex model and exposes them to bias during generation.

The DDSP library[[7]](#footnote-7) currently features 6 interpretable DSP elements implemented as TensorFlow layers: 3 synthesizers and 3 effects:

* Additive synthesizer (a harmonic oscillator)
* Filtered noise (“subtractive” synthesizer)
* Wavetable (interpolative lookup from small chunks of waveforms)
* Reverberation (3 different methods)
* FIR Filter (linear time-varying finite impulse response)
* ModDelay (variable length delay lines)

The implementation and the demo presented in the paper arises some issues and drawbacks[[8]](#footnote-8) of the methods introduced:

1. While the experiments presented in the paper showed success with synthesizing monophonic audio, it was unclear how and if synthesizing polyphonic audio is possible. As synthesizing several notes, from different instruments will without doubt require different network architecture, such as more encoders, altering the latent space to be able to represent all necessary information, and including additional DDSP elements for synthesis.  
   Additionally, the demo used Spectral Modeling Synthesis[[9]](#footnote-9) for synthesis, due to its large parametric size, which leads to high expressiveness. For larger networks, that might be required for polyphonic synthesis, large parametric size might become a constraint to successful learning process, and a less expressive model will be required, which in turn might hinder quality polyphonic audio generation.
2. Since the output of the decoder goes simultaneously into different DDSP elements, the latent representation is highly entangled and therefore networks with many elements might fail to give any meaningful interpretability to the latent code.
3. We feel that there have been insufficient experiments to show the claims of the strength of DDSP. There are only three variants of DDSP elements that were used (additive synth, white noise filter and reverb), and for relatively simple task – synthesizing a monophonic, single instrument. There hasn’t been extensive experiment that requires large DDSP element count, on a global music datasets such as NSynth?..  
   Just one comparison using NSyth dataset against WaveRNN.

TODO

**Controls**

**Signal**

Figure 1. A visualization of the processor model.

**Inputs**

get\_controls()

get\_signal()

Processor object

# Methods

If implementing an existing paper, explain the original approach as well as your ideas for modifications, additions or improvements to the algorithm/task/domain etc., as relevant. Otherwise, provide a detailed explanation of your approach. In both cases, explain the empirical and/or theoretical motivation for what you are doing. Finally, describe the data you will be using for evaluation.

Firstly, we will cover the implementation of the elements described in the paper.  
This step was upmost important to us, as understanding the process helped us discover how the authors integrated these elements into a neural network by making them differentiable.

At the heart of the code’s design is a *processor* object, the main type of object in the library, and the base class for both the synthesizers and effects detailed below.

A processor receives an input, which is generally an output of a neural network, and outputs some signal relevant to its function. Internally, such object converts the input vector(s) to a valid-formatted vector(s) for this specific processor, called *controls*, and then processes these controls to create a *signal*. This process is visualized in figure 1.

## Synthesizers

In this subsection, we cover the implementation of the synthesizers in the DDSP library.

### Additive synthesizer

#### Hyperparameters

An additive synthesizer processor has 4 available hyperparameters:

1. Number of required samples from the resulted signal (defaults to 64000).
2. Audio sample rate (defaults to 16000, combined with the default samples results in 4 seconds of audio).
3. Scaling function for the amplitudes and the harmonic distribution (defaults to an "exponentiated sigmoid" function).
4. Whether to eliminates frequencies above the Nyquist frequency[[10]](#footnote-10) (default as enabled).

#### Controls

The synthesizer accepts 3 inputs (Denoting all time frames as ):

* Amplitudes, a 3-D tensor of shape , that for each batch, contains a 2-D tensor that has an amplitude value for each time frame.
* Harmonic distribution, a 3-D tensor of shape , that for each batch, contains a   
  2-D tensor that has harmonic distributions for each time frame.
* Fundamental Frequencies, a 3-D tensor of shape , that for each batch, contains a 2-D tensor that has a fundamental frequency value for each time frame. Treats the values as if they were in hertz.

Three issues with the input values may arise, and the processor fixes them to be valid controls:

1. Amplitudes must be non-negative values.
2. The sum of all distributions must be 1.
3. Any frequencies above the Nyquist frequency might lead to aliasing.

To handle these problems, the processor executes the following steps:

* Scales and using the scale function.
* If enabled, changes the amplitude of any frequency above the Nyquist frequency to 0.
  + Creates the harmonic frequencies from by multiplying every integer frequency up to new frequencies.
  + Zeroes the probability of every harmonic frequency that is above half the sampling rate. (This is done to allow the next step to work properly, while also voids the need to zero out the actual amplitudes).
* Normalizes by dividing every value by the sum of all values, effectively making the sum of all distributions to 1.

#### Signal

It is important to note that the generated audio is monophonic by design. Therefore, the processor receives fundamental frequencies, amplitudes, and harmonic distributions and creates the audio waveform by:

* Creating two 3-D tensors of shape , that for each batch, contains a 2-D tensor that has harmonic frequencies/amplitudes for each time frame.
* Resamples both tensors by interpolating to total number of time steps (according to the hyperparameter) by up-sampling using hamming window method[[11]](#footnote-11) for amplitude envelopes, and linearly[[12]](#footnote-12) for frequency envelopes.  
  This creates two new tensors of envelopes of shape .
* Generates audio. First, creates phases , where -th element is:

Summed over all samples, and the final audio , where its -th element is:  
Summed over the last dimension (all sinusoids).

## Effects

In this subsection, we cover the implementation of the effects in the DDSP library.

### Reverb

#### Hyperparameters

A reverb effect processor has 4 available hyperparameters:

1. Whether the processor should learn the impulse response (defaults to false).
   1. Length of the impulse response (defaults to 48000).
2. Whether to add the dry audio to reverberated signal (defaults to true).

#### Controls

The effect accepts 1 or 2 inputs – audio to add reverb to, and impulse response, if it is not learnable.

Creating the controls doesn’t do much besides pad the impulse response as needed to match the batch size (since the learnable data is global for the whole dataset).

### Signal

The signal process expects two input – the audio and the impulse response.

It then proceeds to mask the first impulse response to mask the dry signal, and then convolve the signal with new impulse response using FFT.

In conclusion, every operation done on the input tensors detailed on controls and signal generation process of every DSP element are differentiable. Therefore, it is clear that each whole processor is differentiable, and can be simply added as a layer in a neural network, letting the auto-differentiation method of TensorFlow – the tape do all the work by going backwards through the operations when required.

TODO

TODO

TODO

# Implementation and experiments

Describe the experiments performed and their configurations, what was compared to what and the evaluation metrics used and why. Explain all implementation details such as model architectures used, data preprocessing/augmentation approaches, loss formulations, training methods and hyperparameter values.

Note: You can use existing code, e.g. in your implementation but specify what you used and which parts you implemented yourself.

Firstly

# Results

Present all results in an orderly table and include graphs or figures as you see fit. Discuss, analyze and explain your results. Compare to previous works and other approaches for your task.

Firstly

1. [TensorFlow](https://www.tensorflow.org/overview), a public tool for machine learning, which supports [auto-differentiation using computation graph](https://deepnotes.io/tensorflow) [↑](#footnote-ref-1)
2. Chris Donahue, Julian McAuley, and Miller Puckette. [*Adversarial Audio Synthesis*](https://openreview.net/pdf?id=ByMVTsR5KQ) [↑](#footnote-ref-2)
3. Alexandre Defossez, Neil Zeghidour, Nicolas Usunier, Leon Bottou, and Francis Bach. [*SING: Symbol-to-Instrument Neural Generator*](http://papers.nips.cc/paper/8118-sing-symbol-to-instrument-neural-generator.pdf) [↑](#footnote-ref-3)
4. S. O. Arik, H. Jun, and G. Diamos. [*Single-Image Crowd Counting via Multi-Column Convolutional Neural Network*](https://zpascal.net/cvpr2016/Zhang_Single-Image_Crowd_Counting_CVPR_2016_paper.pdf) [↑](#footnote-ref-4)
5. Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu. [*WaveNet: A generative model for raw audio*](https://deepmind.com/blog/article/wavenet-generative-model-raw-audio) [↑](#footnote-ref-5)
6. [Jesse Engel](https://openreview.net/profile?email=jesseengel%40google.com), [Kumar Krishna Agrawal](https://openreview.net/profile?email=kumarkagrawal%40gmail.com), [Shuo Chen](https://openreview.net/profile?email=chenshuo%40google.com), [Ishaan Gulrajani](https://openreview.net/profile?email=igul222%40gmail.com), [Chris Donahue](https://openreview.net/profile?email=christopherdonahue%40gmail.com), [Adam Roberts](https://openreview.net/profile?email=adarob%40google.com). [*GANSynth: Adversarial Neural Audio Synthesis*](https://openreview.net/pdf?id=H1xQVn09FX) [↑](#footnote-ref-6)
7. Code available in [GitHub](https://github.com/magenta/ddsp), also a direct URL to the elements’ implementation: [synthesizers](https://github.com/magenta/ddsp/blob/master/ddsp/synths.py) and [effects](https://github.com/magenta/ddsp/blob/master/ddsp/effects.py). [↑](#footnote-ref-7)
8. Some were also discussed by reviewers on the [paper’s OpenReview site](https://openreview.net/forum?id=B1x1ma4tDr). [↑](#footnote-ref-8)
9. Xavier Serra, [A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition](https://repositori.upf.edu/bitstream/handle/10230/34072/Serra_PhDthesis.pdf?sequence=1&isAllowed=y) [↑](#footnote-ref-9)
10. [Nyquist frequency](https://en.wikipedia.org/wiki/Nyquist_frequency) is half of the sampling rate of a discrete signal. Sampling frequencies over it can cause aliasing. [↑](#footnote-ref-10)
11. [Han and Hamming windows](https://ccrma.stanford.edu/~jos/sasp/Generalized_Hamming_Window_Family.html), specific type of a cosine-sum window function. [↑](#footnote-ref-11)
12. [Bilinear interpolation](https://en.wikipedia.org/wiki/Bilinear_interpolation), one of the basic techniques of resampling. [↑](#footnote-ref-12)