Creating Music Interpolations with NSynth

Daniel Gareev
University of Luxembourg
Email: daniel.gareev.001@student.uni.lu

Amro Najjar
University of Luxembourg
Email: amro.najjar@uni.lu

Abstract

Researchers are developing new deep learning algorithms for generating art such as music and images. Audio synthesis is the task of generating raw audio such as music and speech. The goal of this project is to generate interpolations between short audio samples by using a neural network that generates raw audio. By interpolating the sounds that were synthesized by the machine learning models, new hybrid musical sounds could be discovered. As an outcome of the project, the application with the user interface should be created where it could be possible to generate palettes of different sounds (e.g., real instruments, synthesized sounds, drums) to create evolving interpolated sounds.

1. Introduction

The current technological advancements have transformed the way we not only produce, but listen and work with music. Traditionally, music was generated manually. However, with the extensive development of deep learning, it has now become possible to generate music without human intervention. This offers artists more creative inspiration and the ability to explore different domains in music.

There are two classes of music generation approaches, audio music generation and symbolic music generation [1]. Some of the popular examples include the GANSynth model [2] for audio-domain generation (i.e, generating WAVs) and MelodyRNN [3] for symbolic-domain generation (i.e., generating MIDIs). In this project, we focus on the audio generation domain, which requires learning from audio [1]. The audio has different semantic aspects such as timbre, harmonics and dynamics, making the audio generation a difficult task.

In this project, we first introduce the basic elements of audio synthesis. Then, we review the state-of-the-art related work in the area of audio generation with neural networks. Then, we use NSynth, a model from Google's open-sourced research project Magenta to generate audio interpolations between various sound snippets [4].

Finally, we introduce an interactive web application, which is built upon the pre-trained NSynth model. The application shows how the neural network interpolates two sounds and creates a new unique sound. To evaluate the application usability, we carry out a small number of user tests. With this application, we hope to enable people to learn more about machine learning.

2. Project Description

2.1. Domains

2.1.1. Scientific. The scientific domain of the project mainly focuses on two following areas. First, we focus on audio generation with neural networks and their application in the area of audio generation. Second, we motivate the reason for modeling audio by presenting an introduction to the audio synthesis and reviewing state-of-the-art related work in the field of audio generation.

2.1.2. Technical. The primary technical domain of this project focuses on generating audio interpolations with neural networks. We use Google's open-source research project Magenta model called NSynth (Neural Synthesizer) and use one of the pre-trained models to generate audio interpolations [5]. The project also extends to areas of web development technologies and human-computer interaction as we aim to develop a web application that demonstrates a process of music interpolation with neural networks.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The main goal of the scientific deliverable is to provide an overview of the applications of neural networks in generating audio data by reviewing the state-of-the-art work in the area. Furthermore, the basic fundamentals of the audio synthesis should be presented. Specifically, we focus on waveforms as they are widely used for the audio generation with neural networks. We will also motivate why raw audio modeling is both an interesting and complex problem. The overview of the generative models and their architectures will be discussed (e.g., WaveNet, GanSynth) [6] [7]. We will also take a look at some of the variations of the models that exist and some important ways in which they differ from each other.

The project also involves human-computer interaction as a method to create a usable and intuitive application for both musical and non-musical audiences. Another relevant area of the project is UX and UI, as the project deals with developing the user interface for the application.

2.2.2. Technical deliverables. The main goal of the technical deliverable is to implement a model to interpolate audio sounds. One of the models in the area of audio

generation, NSynth by Google's Magenta, is able to produce unusual new sounds with machine learning by mixing the latent representations of the sounds that result in a new hybrid sound [4]. Another goal of the project is to collect audio samples to interpolate between. Various sounds (e.g., real instruments, audio synthesized sounds, drums) will be collected and compiled as a dataset and will be used as an input to the model. We will experiment with them to find the most interesting and meaningful interpolations.

As an outcome of the project, we will develop a web application that enables generation of interpolations between the sounds. However, as the computation of the interpolations is expensive, rather than generating sounds on demand, we will create a set of original sounds ahead of time. We will then synthesize all of their interpolated representations and use them in the web application. Another technical deliverable of the project is to learn about user interface (UI) and user experience (UX), as building an application that is usable and responsive involves understanding of both of these aspects.

3. Prerequisites

3.1. Scientific prerequisites

One of the scientific prerequisites is a basic understanding of neural networks fundamentals. In addition, basic knowledge in audio generation and familiarity with the concepts such as Musical Instrument Digital Interface (MIDI), a digital audio workstation (DAW), is required. Moreover, familiarity with some of the basic audio concepts is needed (e.g., harmonics, timbre and dynamics).

3.2. Technical prerequisites

The required technical prerequisites to start working on this project are prior programming experience in Python programming language, as well as experience with machine learning. Furthermore, a background in graphic design and web technologies (such as HTML, CSS, Javascript) is required to implement the web application. Furthermore, a basic understanding of prototyping, sketching and usability testing is necessary.

4. A Scientific Deliverable 1: Generating Audio with Deep Neural Networks

4.1. Requirements

In this section, we will attempt to motivate the reason for modeling music in the waveform domain. We will also motivate why raw audio modeling is both an interesting and complex problem. Then, we will review the related work in the field of audio generation. Specifically, we will cover state of the art in both likelihood-based and adversarial models of raw music audio. We will also take a look at some of the variations of the models that exist and some important ways in which they differ from each other.

4.2. Design

We produce the section by reviewing a set of academic articles and related work in the area of audio generation with neural networks. This helps us to present an overview of existing generative models for audio generation. Next, we focus on a set of articles that provide a basic understanding of audio synthesis.

4.3. Production

4.3.1. Audio Generation. Music generation using neural networks has attracted much attention recently. Furthermore, a large number of neural network models have been proposed for generating audio. This includes models for generating a melody sequence or audio waveforms. There are two classes of music generation approaches, symbolic music generation and audio music generation [1]. Music generation has been widely studied in the symbolic domain, where the output of the generative process could be a musical score or a sequence of MIDI events [8]. However, the process of producing the sound has often been abstracted away. Most of the work on music generation using machine learning has made use of musical scores such as MIDI or some other higher-level representation because it allows to capture performance-specific aspects of the music without having to model the sound [9]. It also reduces the dimensionality of the data and allows for lower-capacity models to be used effectively [9]. In this project, we focus on the area of audio generation, which is modeling music in the waveform domain.

The process of audio generation has seen a rise with the advent of synthesizers that allow to create audio signals through various synthesis methods such as additive synthesis, subtractive synthesis and frequency modulation synthesis. These synthesizers are now largely available to musicians as standalone or VST, VST3 and AU plugins and are now present in most modern music. They are implemented through some combination of hard-coded rules and modeling of the physical processes.

More recently, the use of machine learning for audio generation has seen a rise; instead of having to model the process of the audio synthesis, machine learning algorithms allow us to learn the statistical features that make up the sound and use that to generate new original sounds. The production of electronic sounds with this novel method proposes an alternative to the use of oscillators, synthesizers, analog hardware, and other audio components that have been the standard resources of the producer of electronic music.

However, digital audio representations require a high sample rate (typically 16,000 samples per second or more) to achieve high fidelity, and modeling comes with higher computational requirements [9]. Modeling realistic-sounding raw audio is a complex task as it requires predicting at a high sample rate, with structure at many time-scales.

4.3.2. Waveforms. There are many types of audio digital representations. For reproduction, sound is stored by encoding the shape of the waveform as it changes over time [9]. A waveform can be thought of as an abstract representation of sound waves that shows the air molecule's displacement over time. The waveform has an amplitude (how much a molecule is displaced from its initial position) and the frequency (how many times the waveform repeats in a given amount of time) [10]. Amplitude can be thought of as loudness (although loudness is our perception of the sound pressure); the more the air molecules are displaced, the louder the sound is perceived. Frequency is similar to the "pitch" of a sound; the faster a wave repeats itself, the higher the pitch of the note. Amplitude is measured in decibels; however, it is measured in various ways through the signal flow. The measure of the amplitude in the air is represented by the Decibels of Sound Pressure Level, abbreviated as "dB SPL". For frequency, the common unit of measurement is the Hertz, abbreviated as "Hz", which represents the number of repetitions per second.

The waveform in Figure 1 shows a single oscillation of a sound wave; it starts by displacing the air molecule in the positive direction until the maximum amount is displaced, and then in the negative direction until it becomes silent [10].

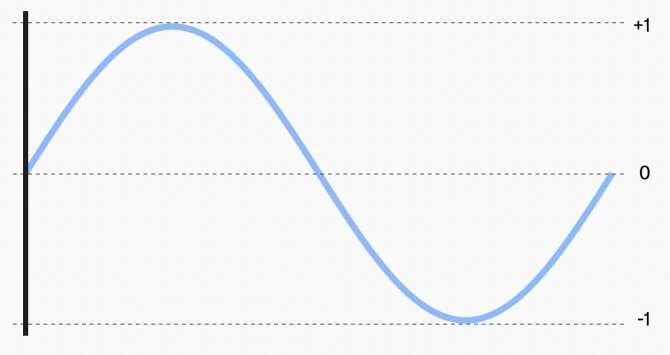


Fig. 1. Amplitude in a Waveform. Figure Source: [10]

The waveform in Figure 2 repeats twice as fast; it represents an example of the periodic waveform that can be repeated to produce a constant tone [10]. The graph shows a 1-second interval and a wave is oscillating at 2Hz.

The sound waves come in many shapes and forms. A sine waveform which was demonstrated earlier represents a *fundamental* waveform [10]. When we have a sine wave, we hear energy at a single frequency. For example, if we play a 1HZ sine wave, the only frequency we will hear is 1HZ. However, that rarely happens as most of the sounds have energy at multiple frequencies. When a waveform has some "side effect" frequencies, we call them *harmonics* [10]. These describe the spectrum of how the sound is distributed at multiple frequencies. To help us understand how harmonics work, we need a way to represent the additional frequencies. The bar graph in Figure 3 shows that a sine wave doesn't have any harmonics, as it is the *fundamental* waveform [10].

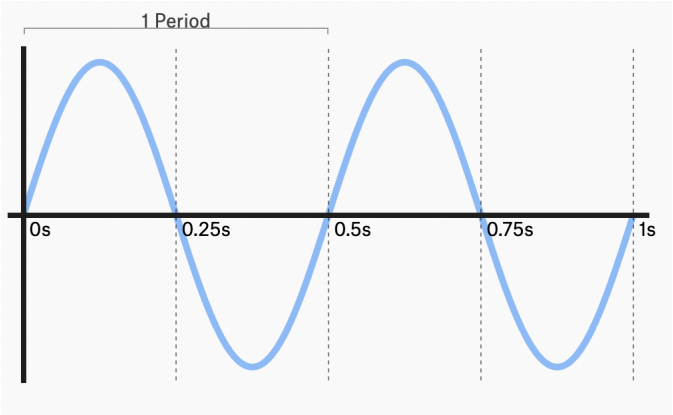


Fig. 2. Frequency in a Waveform. Figure Source: [10]

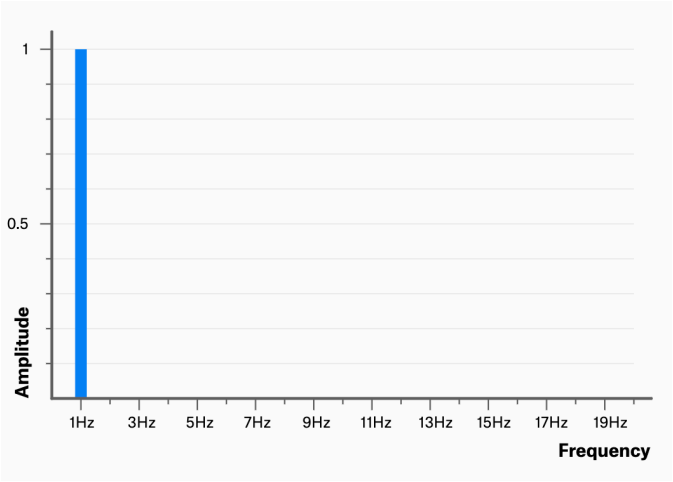


Fig. 3. Harmonics in a Sine Wave. Figure Source: [10]

Harmonics are additional frequencies that are created by specific waveforms. Let's look at the triangle waveform, where the straight lines connect in triangle-like shapes (Figure 4). This sound wave typically sounds a little "brighter" because of the added harmonics. The graph in Figure 5 shows additional harmonics of the triangle waveform. They are visualized as additional lines in the frequency spectrum. Different waveforms have different selections, but they always follow the pattern of the fundamental frequency, followed by second harmonic, third harmonic, fourth harmonic, etc. [10].

The waveforms can be layered with each other to create more complex sounds with a wider frequency spectrum. This is called *additive synthesis* [10]. The waveform graph in Figure 6 shows us two waves: 1Hz at 1 amplitude and 3Hz at 0.33 amplitude. The addition of the waveforms works by adding individual displacement values at each point; the new set of points is a new single waveform [10].

However, the addition of the waveform does not always make the sound louder. The *phase* property of the waveform demonstrates that clearly, which is the amount of offset applied to a wave, measured in degrees [10]. For example,

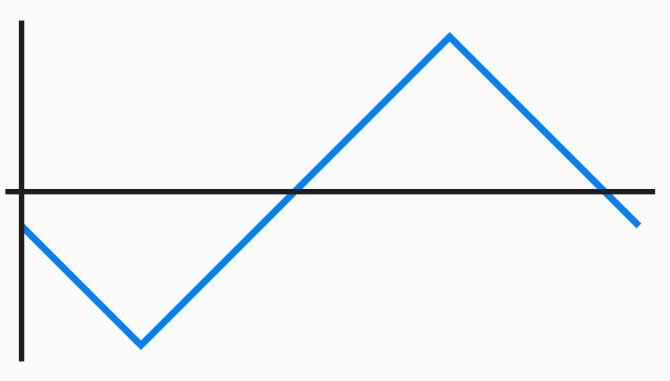


Fig. 4. Triangle Waveform. Figure Source: [10]

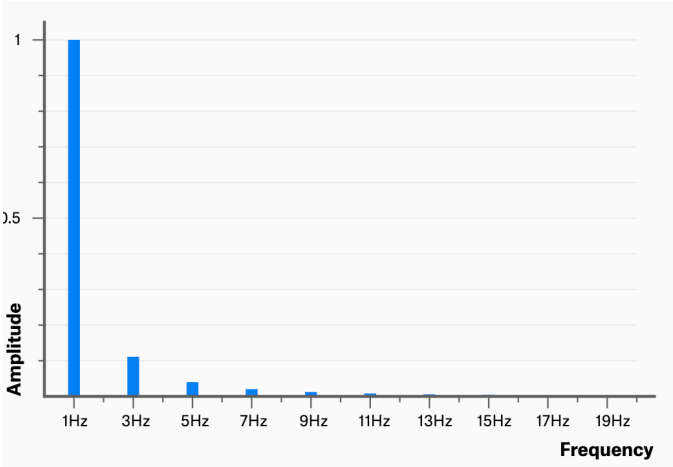


Fig. 5. Harmonics in a Triangle Wave. Figure Source: [10]

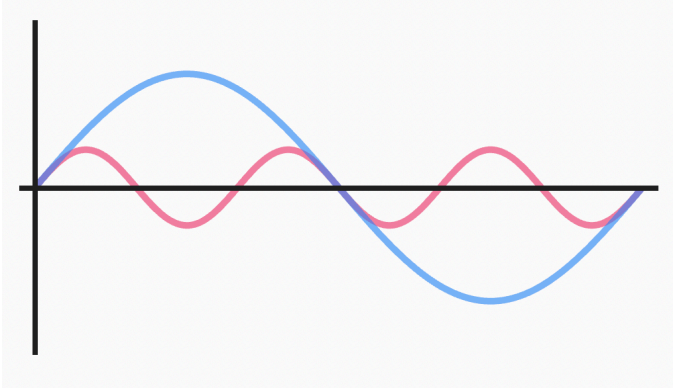


Fig. 6. Waveform Addition. Figure Source: [10]

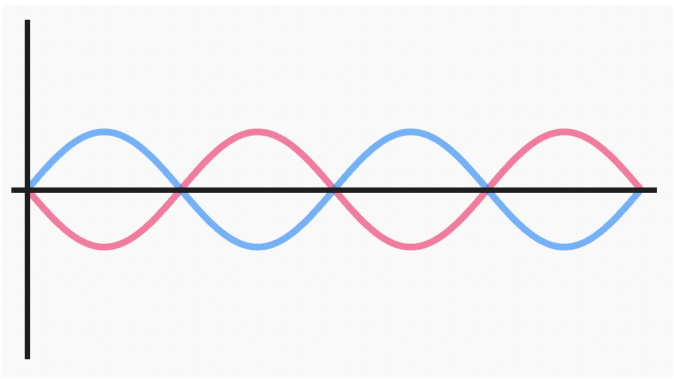


Fig. 7. Phased Waveforms. Figure Source: [10]

if a wave is 180 degrees out of phase, that means it's delayed by 50% of its period. The graph in Figure 7 reflects this and shows two identical sine waves in terms of frequency and amplitude; however, the delayed sine wave cancels out the original wave as its phase is offset by 180 degrees [10]. When generating sound, phase is an essential term as it affects our perception of a sound.

For analysis, we often make use of *spectrograms*, both for computational methods and for visual examination [9]. The spectrogram represents the frequency and the amplitude of a signal of a particular waveform at each point in time. The graph in Figure 8 shows the snapshot of a sound over 30 seconds (the horizontal line), the frequency (the vertical lines) and the amplitude (the color change).

When representing a waveform digitally, we need to *discretize* it in both time and amplitude [9]. Note that perfect human hearing ranges from 20Hz to 20,000Hz, with 20Hz being the lowest we can hear. As waveforms are band-limited, we can discretize the waveform in time without any loss of information, as long as the sample rate is high [9].

4.3.3. Generative Models. If we think of a dataset of examples $x_1, \ldots, x_n$ as samples from a true data distribution $p(x)$, a generative model learns to approximate this data distribution $p(x)$ and generate new samples. To train a generative model, we first collect data in some domain (e.g., sounds) and then train a model to generate data like it. The waveform modeling is currently the most common approach in the generative setting.

There are *implicit* and *explicit* generative models [9]. An implicit model can produce new samples but cannot be used to infer the likelihood of an example. If we have an explicit (also called *likelihood-based*) model, we can model the distribution of the data $p(x)$ directly with a likelihood function. Some of the examples of the likelihood-based models include autoregressive models, flow-based models and variational autoencoders [9].

Autoregressive models train a network that models the conditional distribution of every element x_i in a sequence given the previous element. They can provide quick inference (i.e., computing $p_X(x)$ for some x); however, they are slow to sample from, as samples have to be drawn sequentially for each position in the sequence [9]. Flow-based models need to be quite large to be effective, but they are quick both for inference and sampling [9]. The variational autoencoders are one of the most popular likelihood-based models; however, in the context of waveform modeling, they are used much less than other models as they limit the extent to which they can capture dependencies in the data [9].

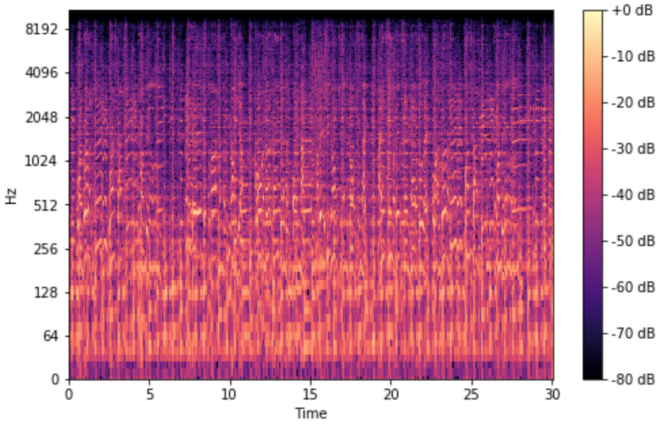


Fig. 8. Spectrogram

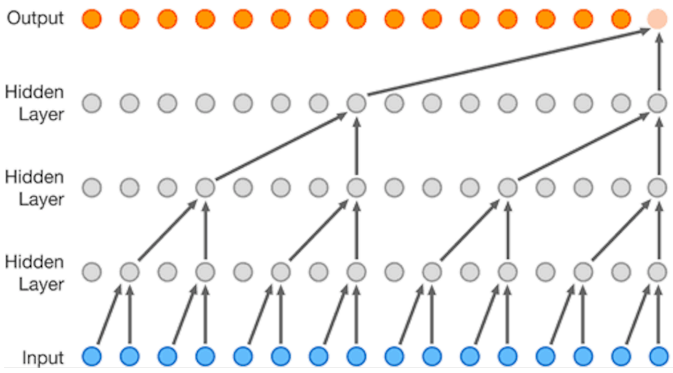


Fig. 9. WaveNet Structure. Figure source: [11]

Generative Adversarial Networks (GANs) are also often used to model waveforms, but they capture the data distribution in a different way. They pose a training process as a game between two separate networks; a generator attempts to produce examples according to the data distribution $p_X(x)$, given latent vectors z, and a discriminator tries to classify samples as either coming from the true distribution or generated model distribution [9]. While adversarial and likelihood-based models are both trying to model data distribution $p_X(x)$, GANs tend to be better at producing realistic examples, but they often fall short to capture the variations of the data distribution [9].

4.3.4. Likelihood-based Models. A large number of likelihood-based models have been proposed for modeling raw audio data. One of the most popular likelihood-based models is called WaveNet, which is a deep generative model for raw audio waveforms. [6]. WaveNet is a convolutional neural network, where the convolutional layers use filters with gaps between the inputs, that are called dilation factors. These filters help the network to create the connectivity pattern across the layers and allow its receptive field to grow exponentially with depth [11]. The illustration in Figure 9 shows how the WaveNet is structured. At training time, the real waveforms are fed into the network; then, we can sample the network to generate synthetic samples. At each step when sampling, the value is drawn from the probability distribution computed by the network and then fed back into the input to create a new prediction for the next step [11]. The network is essentially computing the samples one step at a time. This is computationally expensive, but this step is essential for generating complex, realistic-sounding audio [11]. SampleRNN is another popular likelihood-based audio generation model, which generates one audio sample at a time [12]. Unlike WaveNet, SampleRNN is a recurrent model which uses a stack of recurrent neural networks (RNNs). Each of these RNN layers is stacked and they all run at a distinct frequency. While some of the layers are updated frequently, the higher-level RNNs are learning the long-term structure of the sound over time by updating less frequently.

As autoregressive models generate one audio sample at a time, they are slow and hard to deploy in real-time production [13]. Parallel WaveNet tries to solve that issue by introducing Probability Density Distillation to train a parallel feed-forward network where samples can be drawn in a single feed-forward pass. The network can quickly generate high-fidelity samples of speech and is deployed by Google Assistant [13]. However, Parallel WaveNet is not often used for audio generation as it uses loss functions to make the waveform resemble speech, which makes it hard to create diverse sounds. Another model, ClariNet, introduces a new solution for parallel wave generation by using Gaussian inverse autoregressive flow [14].

There is another set of networks that have been applied to audio data that are called flow-based models. These models are trained with a maximum likelihood (maximizing a likelihood function). WaveGlow and FloWaveNet are flow-based network that can generate high quality speech from mel-spectrograms [15] [16]. These models need to be quite large to generate high-fidelity sounds, but they tend to be fast at sampling as all timesteps can be generated in parallel. WaveFlow is another new model that tries to merge flow-based modeling with autoregressive modeling [17]. This allows the model to be fast to sample from while the size of the model can still be relatively small.

The WaveNet still has limits when generating audio samples as it cannot capture the longer-term structure of audio. For example, it can capture the local structure of a sound (e.g., the timbre of an instrument), but it falls short to model the long-term patterns (e.g., chord progressions) [9]. If we want to train a model that can learn long-term structure across ten seconds, this would require a lot of computational resources. For example, to model one second of audio, we need to have a sequence of 16000 timesteps at 16 kHz, so even at the scale of a second, we need to model long sequences. The Hierarchical WaveNets model tries to solve this issue by using autoregressive discrete autoencoders (ADAs) to enable autoregressive models to capture long-range correlations in waveforms [18]. They use separate blocks to model local structures at different timescales and to capture a high-level

representation of long-term patterns. This helps the model to produce samples that sound more musical and realistic. Another approach is to use high-level representations such as MIDI to construct a hierarchical model instead of learning high-level representations of music from audio data [9]. Wave2Midi2Wave model uses a conditional WaveNet model that generates audio and also MIDI to model long-term correlations in music over time [19]. This approach allows this model to produce high-quality samples in terms of musical structure, but it is limited to the music samples that MIDI can accurately represent [9].

OpenAI introduced another model called Sparse Transformers that can generate long-sequence raw audio [20]. They show impressive results by modeling images and music audio using the model, with sparse attention that can model waveforms of up to 65k timesteps (about 5 seconds at 12 kHz).

4.3.5. Adversarial Models. Recent works have also shown success in generating music using Generative Adversarial Networks (GANs), which is a state-of-the-art method for generating high-quality images. The first attempt at applying GANs to unsupervised synthesis of raw-waveform audio was demonstrated in the WaveGan model [21]. Although the model did not reach high-fidelity in terms of musical samples it produced, it was a first step towards exploring the role of GANs in audio generation.

GANSynth from Google's Magenta is a newer model for generating high-fidelity audio with GANs [7]. Rather than generating audio sequentially, GANSynth generates an entire sequence in parallel, synthesizing audio faster. GANSynth uses a Progressive GAN architecture to incrementally upsample with convolution from a single latent vector [7]. However, it is difficult to align phases for highly periodic signals. In Figure 10, the red-yellow curve is a periodic signal and a black dot shows the beginning of each cycle. If we try to model this signal by splitting it into separate frames (dotted line), the distance between the beginning of the frame (dotted line) and the beginning of the wave (dot) changes over time (black solid line) [7].

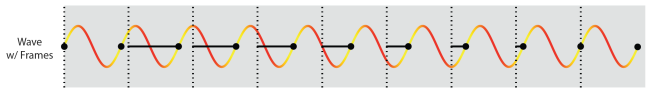


Fig. 10. Waveform with Spectrogram. Figure source: [7]

This difference (black line) is called the phase and it changes over time because the wave and frames have different periodicities [7]. However, we can convert the phase into a quantity that is easier to model. As we can see in Figure 11, phase is a circular quantity (yellow bars, mod 2π), but if we unwrap the phase by shifting it by a multiple of two for each frame (orange bars), it decreases by a constant amount each frame (red bars). This is called the instantaneous frequency, which is capturing a change in phase in time. For sounds that have harmonics, this quantity is expected to be the same over time, as the phase rotates at the same rate.

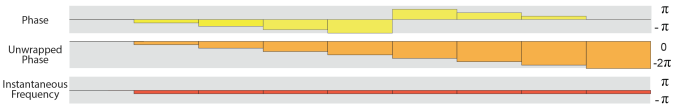


Fig. 11. Magnitude, Phase, Unwrapped Phase and Instantaneous Frequency. Figure source: [7]

This representation can be used to model musical sounds with a lot of harmonics. When we extract the instantaneous frequencies, we see consistent lines reflecting the coherent periodicity of the underlying sound as can be seen in Figure 12 (because phase is an angle, all values that are mod 2π are equivalent) [7].

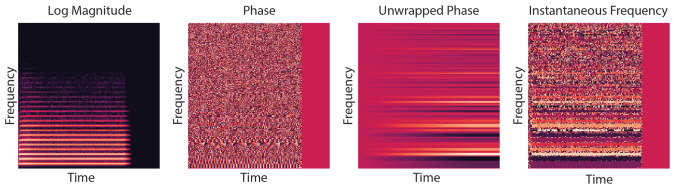


Fig. 12. Phase to Instantaneous Frequency. Figure source: [7]

4.3.6. Discussion. As we can see, the likelihood-based models are currently the most popular models for generating high-fidelity audio samples, with WaveNet and SampleRNN giving a head start for this particular type of audio modeling. However, instead of modeling audio in the waveform domain, there are other approaches that can be used. We can also model other representations of a sound, such as spectrograms. Finally, we can generate music with a long-term structure in the symbolic music generation domain. Models like Magenta's Music Transformer and OpenAI's MuseNet show impressive results in this domain [22] [23].

4.4. Assessment

In this section, we have introduced the basics of audio generation and explored the basics of audio synthesis. We also motivated why raw audio modeling is both an interesting and complex problem. We gave an overview of the two main classes of generative models for modeling audio, likelihood-based and adversarial models. We have presented an overview of existing models for music generation that produce realistic, complex and melodic raw audio. Furthermore, we also discussed the pros and cons of each of the models. Finally, we talked about some of the discussion points.

5. A Technical Deliverable 1: Neural Audio Synthesis with NSynth

5.1. Requirements

The goal of the deliverable is to use Recurrent Neural Networks (RNNs) to generate audio interpolations across instruments. We use Magenta's NSynth (Neural Synthesizer) to interpolate between pairs of samples to create new sounds. The aim of this deliverable is also to collect various samples that will be interpolated.

5.2. Design

We follow NSynth docs on Github to learn more about the model to get started with creating our own sounds [24]. We also use the NSynth Google Colab notebook [25] to upload our own sounds and use NSynth models to reconstruct and interpolate between them.

5.3. Production

5.3.1. Introduction to NSynth. NSynth (Neural Synthesizer) comes from Google's open-source project Magenta that explores the role of machine learning as a tool in the creative process [5]. *"The library includes utilities for manipulating source data (primarily music and images), using this data to train machine learning models, and generating new content from these models"* [5]. The Magenta project has a number of models, each with different features and use cases.

NSynth is a WaveNet-based autoencoder for synthesizing audio. It represents a novel approach to music synthesis designed to inspire the creative process. Unlike a traditional synthesizer which generates audio from manually crafted components like oscillators and wavetables, NSynth uses deep neural networks to create sounds at the level of individual samples [26].

WaveNet is an expressive autoregressive model with dilated convolutions for modeling sequences such as music. Since the network essentially predicts a sample one step at a time, it "draws" the waveform. The mistakes it makes result in the added harmonics that are not presented in the original sound. However, since the context of the dilation block structure is limited to several thousand samples (about half a second), it is difficult to capture a long-term structure and the model needs a guiding signal to improve the long-term structure [26]. To address this, NSynth uses a vanilla WaveNet-style autoencoder that learns its own temporal *embeddings* (mathematical representations of each sound) to synthesize audio; the need for conditioning on external features is removed [4]. The temporal encoder in NSynth replicates the WaveNet and has the same dilation block structure. However, NSynth's convolutions are not casual, so it sees the entire context of the input (Figure 13) [26].

The NSynth network reconstructs the audio by finding its representation by forcing the sound to go through several

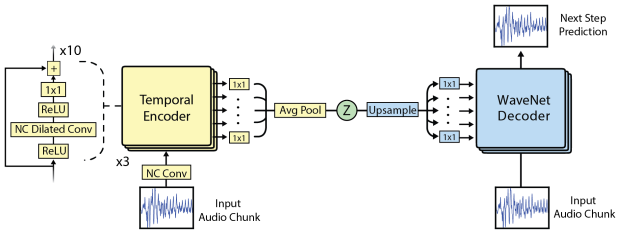


Fig. 13. NSynth Model. Figure source: [26]

layers of transformations. After thirty layers of computation, NSynth uses an average pooling layer to create temporal embeddings of 16 dimensions (16-dimensional vectors) for every 512 samples at 30Hz (32 ms per step) [26]. NSynth conditions the WaveNet decoder with its embedding by upsampling it to the original time resolution, applying a convolution and adding it to each of the decoder's layers [26]. This conditioning is learned by the model and it acts as an external driving force to excite higher harmonics [26]. As the latent code is essentially added to the layers of the WaveNet decoder, it works like an excitation or activation function. The WaveNet is, in essence, a nonlinear infinite impulse response (IIR) filter that is being driven by the embeddings [27]. The spectrograms in Figure 14 show the examples of reconstruction for three different instruments with magnitude represented by the intensity and instantaneous frequency by color. The frequency is shown on the vertical axis and time is shown on the horizontal axis. For the embeddings, the different colors represent the 16 different dimensions at the level of 125 timesteps (32 ms per step). For computational reasons, the NSynth works with mu-law encoded 8-bit 16kHz sound, which adds a slight distortion (grainy texture) to the sound [27].

As the embeddings are time-depending, NSynth allows to shift the snippets in time to listen to how the sound changes dynamically. Also, different computations can be applied to the embeddings, such as synthesizing a temporal mean and standard deviation of a representation.

5.3.2. Interpolations with NSynth. The NSynth network works by finding a representation of sound. Figure 15 shows how the audio signal passes through the encoder network and transforms to the latent representation of a sound. Then, a decoder network converts it back to the sound. We can use the NSynth to linearly interpolate audio in the embedding space.

Normally, mixing the audio waveforms of the two instruments sounds like two instruments are being played at the same time (simultaneously) as the addition of the signals is linear (also called *additive synthesis*). In contrast, mixing the representations and then decoding results in what sounds more like a single hybrid instrument. Therefore, the resulting sound combines different semantic aspects (such as dynamics and harmonics) of two instruments to create a new sound that has a different timbre. This would be impossible with a

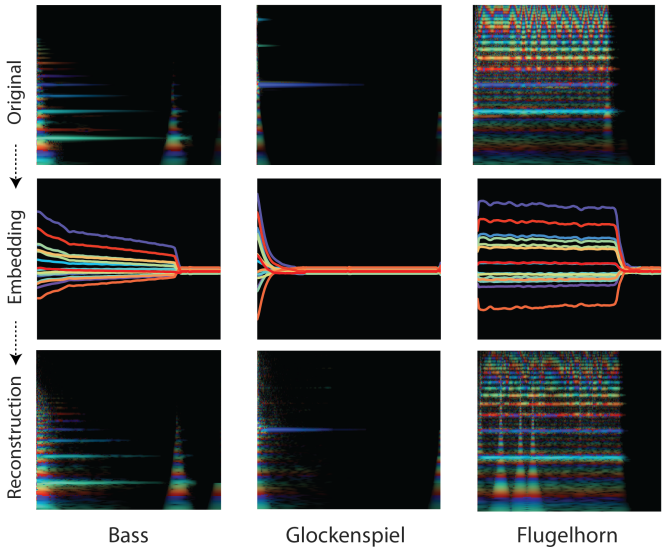


Fig. 14. Spectrograms of audio and reconstructions for three different instruments. Figure source: [26]

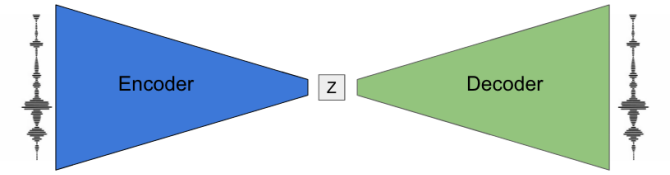


Fig. 15. Encoding and decoding with NSynth. Figure source: [27]

hand-tuned synthesizer or any other musical algorithm.

In Figure 16, there are two columns of audio clips matching the two rows of spectrograms. The top row of spectrograms shows the result of evenly interpolating across instruments by adding the signals. The bottom row shows the result of using NSynth to linearly interpolate in the embedding space. The NSynth adds more harmonics to the original sound across the whole range of the frequency spectrum.

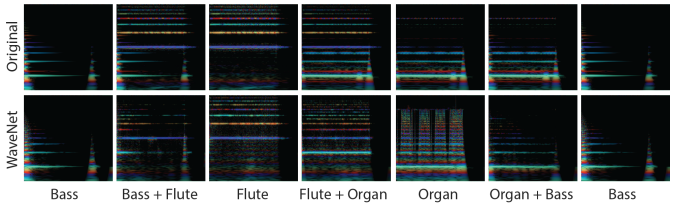


Fig. 16. Interpolations of audio clips. Figure source: [27]

5.3.3. NSynth Dataset. The acoustic qualities and the level of fidelity of the learned instrument in NSynth depend on both the model used and the available training data [28]. Nevertheless, audio signals often contain multi-scale dependencies that make them particularly difficult to model. To address that, Google's Magenta project collected a large-scale and high-quality audio dataset containing 305,979 annotated musical notes, each with a unique pitch, timbre, and envelope [28]. They used commercial sample libraries and generated four seconds 16kHz audio snippets that range over a MIDI range of a piano (21-108) and five different velocities (25, 50, 75, 100, 127) [28]. In terms of sound envelope, the note was held for the first three seconds (known as attack and sustain stages of a sound) and decayed (known as release and decay stages of a sound) [28].

The NSynth Google Colab provides two pre-trained models to choose from [25]. The "Instruments" model is trained on the individual instrument notes of the NSynth Dataset. The second model, "Voices", is trained on a variety of voices audio sounds for an art project "Voices". The audio dataset contains a mix of singing and speaking. In this project, we choose the "Instruments" model as it was trained on a larger quantity of data and it tends to generalize better. While the "Instruments" model was trained on instrument sounds, it can be applied to any sound of any length as it uses temporal embeddings that scale with the length of the sound [27]. Neither of the models reconstructs audio perfectly, but both add their own unique character to sounds.

In this project, we did not attempt to collect our own training dataset, as training for these models is computationally very expensive [24].

5.3.4. Generating Interpolations with NSynth. Our first step in generating interpolations with the model is to upload our own sound files for processing. We can use `.wav`, `.aiff` or `.mp3` files. All files will be converted to `.wav` and downsampled to 16kHz and cropped or silence padded to `Length` seconds as the input and output to the algorithm. We put all sound files into a single folder to reconstruct and interpolate between.

```
Length = 2.0 #@param {type:"number"}
SR = 16000
SAMPLE_LENGTH = int(SR * Length)
```

Listing 1. Set sound length

Next, we encode the audio and also create interpolations between latent vectors (the midpoints between each encoding) from which to re-synthesize audio. We calculate the mean interpolation of two sounds and create one final embedding.

Next, we can observe the encoding of our audio files. They are compressed representations of the audio transformed into a series of 16-dimensional vectors at 30Hz (32ms per frame). Figure 18 shows an example of one of the latent representations of sound. The original audio signal is shown in 17.

Then, we synthesize interpolations. This step is computationally expensive. For example, on a single GPU, it takes about 4 minutes per 1 second of audio per batch.

```
audio = np.array(audio_list)
z = fastgen.encode(audio, ckpt_path, SAMPLE_LENGTH)
print('Encoded %d files' % z.shape[0])

# Start with reconstructions
z_list = [z_ for z_ in z]
name_list = ['recon_' + name_ for name_ in names]

# Add all the mean interpolations
n = len(names)
for i in range(n - 1):
  for j in range(i + 1, n):
    new_z = (z[i] + z[j]) / 2.0
    new_name = 'interp_' + names[i] + '_X_'+ names[j
        ]
    z_list.append(new_z)
    name_list.append(new_name)

print("%d total: %d reconstructions and %d
    interpolations" % (len(name_list), n, len(
    name_list) - n))
```

Listing 2. Generate encodings

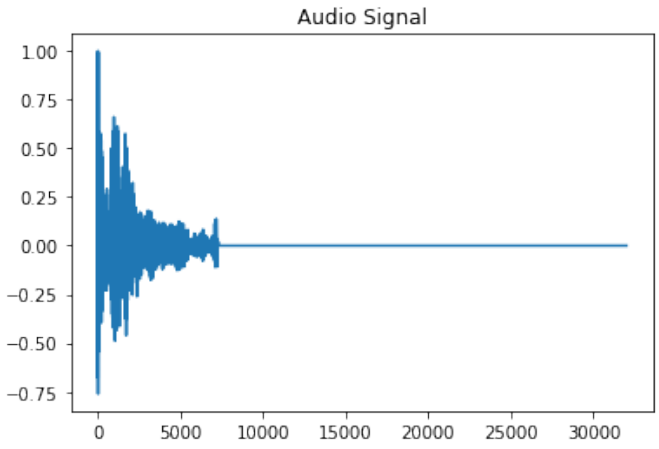


Fig. 17. Audio signal

```
SoundFile = 0
file_number = SoundFile

try:
  print(names[file_number])
  play(audio_list[file_number], sample_rate=SR)
  # fig, axs = plt.subplots(2, 1, figsize=(12, 10))
  plt.figure()
  plt.plot(audio_list[file_number])
  plt.title('Audio Signal')

  plt.figure()
  plt.plot(z_list[file_number])
  plt.title('NSynth Encoding')
except Exception as e:
  print(e)
```

Listing 3. Visualize audio and encoding

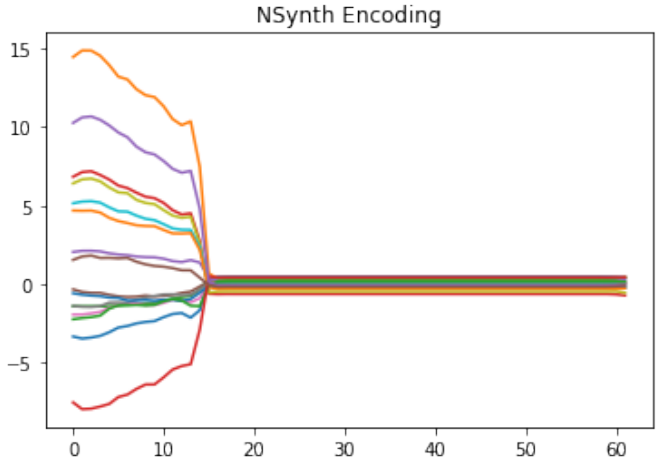


Fig. 18. NSynth encoding

Next, we can download interpolations. The code snippet below downloads the generated interpolations as a single `.zip` file.

We can also generate the outputs and listen to them directly.

For this project, we used a variety of royalty-free sound samples that we have collected online from University of Iowa's sample library called Electronic Music Studios [29]. We have generated in total 232 interpolations between different sound samples (e.g., synthesizer sounds, drums, percussion, real instruments etc.). We have experimented to combine various sounds together (e.g., synthesizer and voice) and listen to the variety of the sound outputs.

5.3.5. Disussion. One of the challenges of audio generation is the evaluation of audio quality, as it is largely subjective and can vary widely for a specific piece of audio. To evaluate the generated audio samples, we have analyzed the outputs qualitatively in terms of the following three metrics: how pleasing, how real, and how interesting it sounds.

In terms of sound, the algorithm does not exactly reproduce the sounds, but regardless of whether it morphs the sounds between strings, synthesizers or drums, there is a distinct character to the sound and it is specific to this model. We have also observed that the sounds with simpler timbre and dynamics (e.g., marimba, piano, vocals) sound much more natural and real-sounding after being reconstructed. In contrast, using highly processed audio samples (e.g., synthesized chords, processed drums, FX) creates overloaded-sounding snippets that sound significantly different from their original sound. We think that it can be the case that the "Instruments" model has been trained on the instruments sample libraries, and the model tries to reconstructs the sounds so that they fit the training data. As the network tries to reproduce the sound so that it sounds like the real one, it maps similar sounds to similar representation. However, it is also interesting to listen to the ways in which the algorithm fails to fit the data.

Surprisingly, while reconstructing multiple notes playing at a single time, we have seen that the model tries to jump between the harmonics of individual notes of a chord progression. It is most likely that the model has only seen a single harmonic series at a time of an individual note, so it tries to capture polyphony. We have also seen that a model

```
print('Total Iterations to Complete: %d\n' %
    SAMPLE_LENGTH)

encodings = np.array(z_list)
save_paths = ['/content/' + name + '.wav' for name
    in name_list]
fastgen.synthesize(encodings,
                   save_paths=save_paths,
                   checkpoint_path=ckpt_path,
                   samples_per_save=int(
                       SAMPLE_LENGTH / 10))
```

Listing 4. Synthesize interpolations

```
import zipfile

def zipdir(path, ziph):
    # ziph is zipfile handle
    for fname in save_paths:
            ziph.write(fname)

zipf = zipfile.ZipFile('Sounds.zip', 'w')
zipdir('/content/', zipf)
zipf.close()

download('/content/Sounds.zip')
```

Listing 5. Download interpolations

has impressive results on reproducing percussive sounds. It reproduces the electronic and acoustic drums closely to the original with a slight tonal bias.

Interestingly, some of the reconstructed snippets seem to fluctuate in fundamental frequency. This can be explained by how the model works. The fundamental frequency is based on the rate of waveform repeats and the harmonic structure is based on the waveform shape. Maintaining the consistent repetitions in the sound is an expensive process as it requires learning long time dependencies that can be over thousands of timesteps. Therefore, when the frequency changes quickly, the model reacts slowly and a large shift in frequency is unlikely. The snippets also follow a similar dynamic envelope of the original sounds. Even some of the expressive variations in the sound like vibrato is captured by the model.

5.4. Assessment

In this section, we have introduced the basic principles of NSynth, as well as how NSynth generates audio interpolations in a latent space. Then, we have demonstrated the process of using NSynth model to interpolate between various audio sounds (e.g., drums, strings, vocals). Furthermore, we have described the NSynth pre-trained model that we used. Finally, we have discussed some of the interesting insights and outcomes of the process.

As future work, we can collect a dataset containing raw audio to train our own NSynth model. Training the NSynth is very expensive and is difficult for many practical setups. Nevertheless, it will be interesting to hear the results of a custom model.

```
print("Originals:\n")
for fname in file_list:
  synth_audio = utils.load_audio(fname,
                                 sample_length=
                                     SAMPLE_LENGTH,
                                 sr=SR)
  print(get_name(fname))
  play(synth_audio, sample_rate=SR)

for i, fname in enumerate(save_paths):
  if i == 0:
    print("Reconstructions:\n")
  if i == len(file_list):
    print("Interpolations:\n")
  synth_audio = utils.load_audio(fname,
                                 sample_length=
                                     SAMPLE_LENGTH,
                                 sr=SR)
  print(get_name(fname))
  play(synth_audio, sample_rate=SR)
```

Listing 6. Listen to the outputs

6. A Technical Deliverable 2: Web Application - Make Unusual New Sounds with NSynth

6.1. Requirements

The goal of this deliverable is to design, prototype and implement the application that is able to use the pre-trained NSynth model to interpolate between audio sounds. The application should be able to generate morphing waveform interpolations and let users play them with their computer or MIDI keyboard. Furthermore, usability tests should be carried out to rate the user interface of the application and collect user feedback. We will also cover some of the relevant human-computer interaction subfields such as user interfaces (UI) and user experience (UX), as these are important aspects to develop an application that is usable and responsive.

6.2. Design

The deliverable is created by researching the existing applications previously created with Magenta library [30]. Specifically, we have explored the source code on Github and tried to replicate the application NSynth: Sound Maker [31]. The application front-end was prototyped in a web-based design tool called Figma [32].

6.3. Production

In this section, we introduce an interactive web application, which is built upon the pre-trained NSynth model. The application shows how the neural network reconstructs the audio of two sounds and morphs the sound to create a new unique hybrid sound. The aim of the application is to demonstrate that machine learning can be used to enable and enhance the creative process. Another goal is to make an application that acts as an educative use case and helps users to learn more about machine learning.

6.3.1. Introduction to UI and UX. The user interface (UI) is the space where human-computer interactions take place. It is a way through which a user can interact with an application. Note that this is not only limited to what a user sees on the screen but also includes keyboard, mouse and any other input interface. The design of the user interfaces intersects between various fields, such as psychology and cognitive sciences. The aim is usually to produce a user interface that is responsive and user-friendly to provide maximum usability to a user. Additionally, UI layers can interact with one or more user senses, such as visual UI (sight), tactile UI (touch) and auditory UI (sound).

The user experience (UX) is the process of how a user interacts with an application. It includes a user's subjective perception of ease of use, usability and efficiency. The user experience while interacting with a product is usually rated through the dimensions such as usability, usefulness, and desirability. For example, one of the frameworks, User Experience Honeycomb was created by Google to guide user interface design [33]. It attempts to address various dimensions of the user experience [33]:

- *Usability*. An application needs to be simple and easy to use.
- *Usefulness*. An application has to fulfill a need.
- *Desirability*. The design of an application should be sleek and to the point.
- *Findability*. Users should be able to quickly find the information they are looking for.
- *Valuableness*. An application should be valuable for an end-user.
- *Accessibility*. An application should be accessible to those with disabilities.
- *Credibility*. An application should be trustworthy and credible.

6.3.2. Prototyping Web Application. To develop a prototype of the application, we have used Figma as a prototyping tool. The process of creating the prototypes in Figma is shown in the Appendix section 8. Currently, the prototype is fully functional, except for the sound when the play button is triggered. This is a technical limitation of Figma as it does not provide the functionality to play audio in a prototype. We have tried to overcome this limitation by using Anima plugin, which allows running code in the prototypes. However, Anima renders the prototype and only runs it in the web browser.

We have created three screens of the application:

- *Splash Screen*. The Splash screen quickly explains the purpose of the application. On this screen, a user can either press "Play" to start interpolating the sounds or check out the "About" section that describes how NSynth works under the hood. The screen is shown in Figure 19.
- *About Screen*. The "About" screen explains how NSynth is functioning and how the interpolations are synthesized. The screen is shown in Figure 20.
- *Interpolations Screen*. This is the main screen of the application. Here, a user can scroll the sound in two carousels to select the sounds between which to interpolate. A user can press "Play" to hear the resulting morphing sound. Additionally, a user can hit any of the MIDI keys on the screen to pitch the sound up or down. It is also possible to play the MIDI keys with the computer's keyboard. The screen is shown in Figure 21.

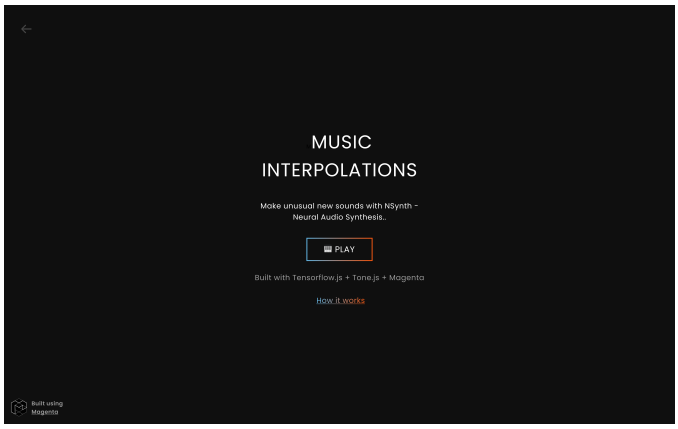


Fig. 19. Splash Screen

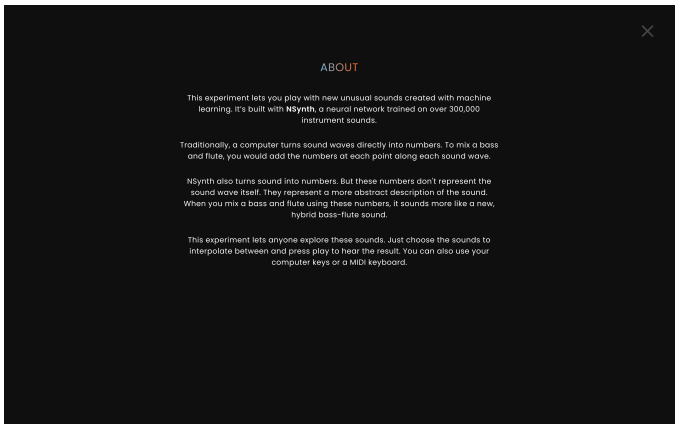


Fig. 20. About Screen

6.3.3. Developing Web Application. The application is composed of several Javascript components that are related to the interface, sound and splash screen. For example, a component `Keyboard.js` is responsible for the MIDI keys that appear on the main screen, and the `Carousel.js` is used to render two carousels. The application also uses Tone.js library to process the audio. As it is computationally expensive to generate interpolations on demand, the sounds of the interpolations were generated in Section 5.3.4 beforehand. Then, we created a specific folder structure so that the interpolations are rendered on the front-end efficiently. Due to the space constraints, we will not go into an in-depth implementation here.

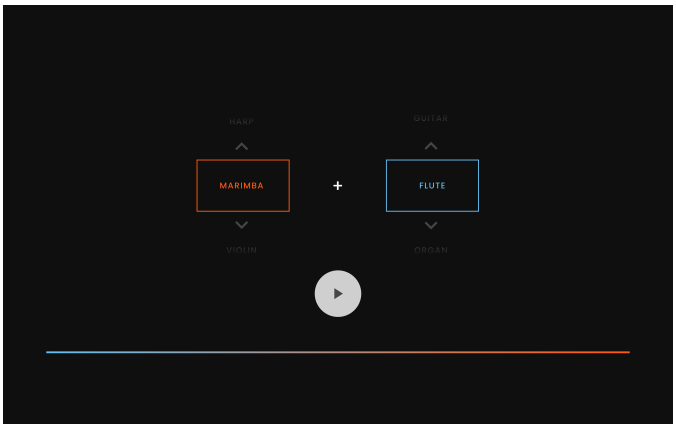


Fig. 21. Interpolations Screen

6.3.4. Usability Testing. The goal of usability testing was two-fold. First, we wanted to see how users perceive the usability of the application. Second, we wanted to see to which extent users understand how the machine learning algorithms work under the hood to produce interpolations. To help answer these questions, we have conducted user tests with 5 participants in total. The participants were mostly students in their Bachelor's or Master's degree and had varying technical knowledge.

We used the User Experience Honeycomb framework [33] to develop a user test and used it as a guide through the user research process. We focused on a few parts of the framework, which were usability, usefulness, desirability, findability, and valuableness, as these were the most relevant dimensions for our application.

The users were guided through a user test in which they were asked to complete the following tasks:

- *Learning More about NSynth.* Find out more details on how the application works. For this task, a user has to press "How it works" button on the Splash screen.
- *Interpolating Sounds.* Try to mix marimba and flute instruments to get the interpolated sound between the two. For this task, a user has to press "Play" button on the Splash screen. Then, a user has to scroll through both of the instruments and then play the sound be either pressing "Play" button, clicking on the MIDI keys on the screen, or using computer's keyboard.

Then, we asked a user follow-up questions. Here, we show some of the responses that we feel are the most insightful. First, we ask several questions related to the sound produced by the model: *"How would you characterize this sound (i.e., real, synthesized)?"*.

> *"It sounds more like a flute. But it doesn't sound like a real instrument. It still has this touch of transformed sound to it. But it sounds more like a hybrid distorted sound."*

Next, we ask participants what do they think machine learning is responsible for in the process *"What do you think the role of machine learning is in this process?"*.

> *"It can create new sounds and you can't create them naturally. You have a lot of ways to create new sounds only with two original sounds. And every time you can have unique sounds."*

Then, to get a better sense of a user's understanding of how the machine learning produces the interpolations, we ask *"How do you think the machine learning works behind the scenes to create new sounds?"*.

> *"It puts together two sounds and finds the average of these two to create another new sound, which is in between of them."*

Finally, we ask a user if the application can be used for educational purposes *"Do you think the app provided user-friendly explanations to help people understand how machine learning works?"*

As a final step, we followed up a user test with a questionnaire that mostly included the Likert-scale questions:

- How satisfied are you with the application (on a scale 0 to 10)?
- How do you rate the usability of an application (on a scale 0 to 10)?
- How do you rate your understanding of machine learning (on a scale 0 to 10)?
- How do you rate your understanding of how the application works (on a scale 0 to 10)?
- Do you think using an app once was enough to understand it (on a scale 0 to 10)?
- Do you have any thoughts or feedback on the application (open-ended question)?

Overall, the users were satisfied with the application and gave it an average score of 8 out of 10. When it comes to the general understanding of machine learning, the responses diverged. However, most of the participants confirmed that they understand how the application works, with an average score of 6 out of 10. The majority of the participants also think that using an app once was enough to understand it, with an average score of 7 out of 10.

Most importantly, the user tests showed that users seem to understand how the application functions and how it interpolates the sound. Moreover, all of the participants said that the new sounds generated after interpolating two instruments sound like a hybrid instrument, with rich harmonics added to the original timbre.

We have also received some useful feedback that can help us to further improve the application. For example, a few of the testers said that they would like to hear the original sounds before they were reconstructed by the model. Also, some of them said that they would like to have a possibility to choose their own sounds to be used as original sounds.

6.3.5. Discussion. In our user interface design, some of the aspects from the model of design criteria were not

addressed. Specifically, we didn't focus on the accessibility and credibility dimensions due to time constraints.

Overall, our application has received positive feedback and most of the users found it useful. However, there are still improvements that can be made. For example, users should be able to hear the original sound before it is reconstructed. Furthermore, it would be useful if users can upload their own sounds to morph. However, this will likely take more time as the computation of interpolations is an expensive process. Furthermore, to smooth out the transitions between the interpolations, we can attempt to additionally mix the audio in real-time from the nearest data point.

Furthermore, there are still improvements that should be made to use an application as an educational case to explain the machine learning algorithms. For example, more visual illustrations can be used along the way to teach the concepts of machine learning.

Finally, we only carried out a few user tests. As future work, more tests can be carried out to discover new insights and provide more statistical significance.

6.4. Assessment

In this section, we have designed, prototyped and implemented the application that is able to use the pre-trained NSynth model to interpolate between audio sounds. Furthermore, we have introduced basic principles of UX and UI. Finally, we have presented the results of the usability testing and discussed some of the future work.

7. Conclusion

In this project, we introduced the basic elements of audio synthesis and motivated why audio generation is a challenging but interesting task. Next, we reviewed the state-of-the-art related work in the area of audio generation with neural networks. Then, we used NSynth to generate audio interpolations between various sounds that we have collected.

Finally, we introduced an interactive web application that shows how the neural network interpolates two sounds and synthesizes a new unique sound. To evaluate the application usability, we carried out a small number of user tests.

References

[1] J. Wu, C. Hu, Y. Wang, X. Hu, and J. Zhu, "A hierarchical recurrent neural network for symbolic melody generation," *IEEE transactions on cybernetics*, vol. 50, no. 6, pp. 2749–2757, 2019.

[2] E. Waite, "Generating long-term structure in songs and stories." https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn.

[3] "Gansynth: Making music with gans." https://magenta.tensorflow.org/gansynth.

[4] J. Engel, C. Resnick, A. Roberts, S. Dieleman, M. Norouzi, D. Eck, and K. Simonyan, "Neural audio synthesis of musical notes with wavenet autoencoders," in *International Conference on Machine Learning*, pp. 1068–1077, PMLR, 2017.

[5] "Magenta." https://magenta.tensorflow.org/.

[6] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[7] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "Gansynth: Adversarial neural audio synthesis," *arXiv preprint arXiv:1902.08710*, 2019.

[8] B. L. Sturm, J. F. Santos, O. Ben-Tal, and I. Korshunova, "Music transcription modelling and composition using deep learning," *arXiv preprint arXiv:1604.08723*, 2016.

[9] "Generating music in the waveform domain." https://benanne.github.io/2020/03/24/audio-generation.html.

[10] "Let's learn about waveforms." https://pudding.cool/2018/02/waveforms/.

[11] "Wavenet: A generative model for raw audio — deepmind." https://deepmind.com/blog/article/wavenet-generative-model-raw-audio.

[12] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "Samplernn: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.

[13] A. Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. Driessche, E. Lockhart, L. Cobo, F. Stimberg, *et al.*, "Parallel wavenet: Fast high-fidelity speech synthesis," in *International conference on machine learning*, pp. 3918–3926, PMLR, 2018.

[14] W. Ping, K. Peng, and J. Chen, "Clarinet: Parallel wave generation in end-to-end text-to-speech," *arXiv preprint arXiv:1807.07281*, 2018.

[15] R. Prenger, R. Valle, and B. Catanzaro, "Waveglow: A flow-based generative network for speech synthesis," 2018.

[16] S. Kim, S. gil Lee, J. Song, J. Kim, and S. Yoon, "Flowavenet : A generative flow for raw audio," 2019.

[17] W. Ping, K. Peng, K. Zhao, and Z. Song, "Waveflow: A compact flow-based model for raw audio," in *International Conference on Machine Learning*, pp. 7706–7716, PMLR, 2020.

[18] S. Dieleman, A. v. d. Oord, and K. Simonyan, "The challenge of realistic music generation: modelling raw audio at scale," *arXiv preprint arXiv:1806.10474*, 2018.

[19] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck, "Enabling factorized piano music modeling and generation with the maestro dataset," *arXiv preprint arXiv:1810.12247*, 2018.

[20] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[21] C. Donahue, J. McAuley, and M. Puckette, "Adversarial audio synthesis," *arXiv preprint arXiv:1802.04208*, 2018.

[22] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, C. Hawthorne, A. M. Dai, M. D. Hoffman, and D. Eck, "Music transformer: Generating music with long-term structure," *arXiv preprint arXiv:1809.04281*, 2018.

[23] "Musenet." https://openai.com/blog/musenet/.

[24] "Nsynth — github." https://github.com/magenta/magenta/tree/master/magenta/models/melody_rnn.

[25] "Nsynth — google colab." https://colab.research.google.com/notebooks/magenta/nsynth/nsynth.ipynb.

[26] "Nsynth: Neural audio synthesis." https://magenta.tensorflow.org/nsynth.

[27] "Making a neural synthesizer instrument." https://magenta.tensorflow.org/nsynth-instrument.

[28] "The nsynth dataset." https://magenta.tensorflow.org/datasets/nsynth.

[29] "University of iowa electronic music studios." http://theremin.music.uiowa.edu/.

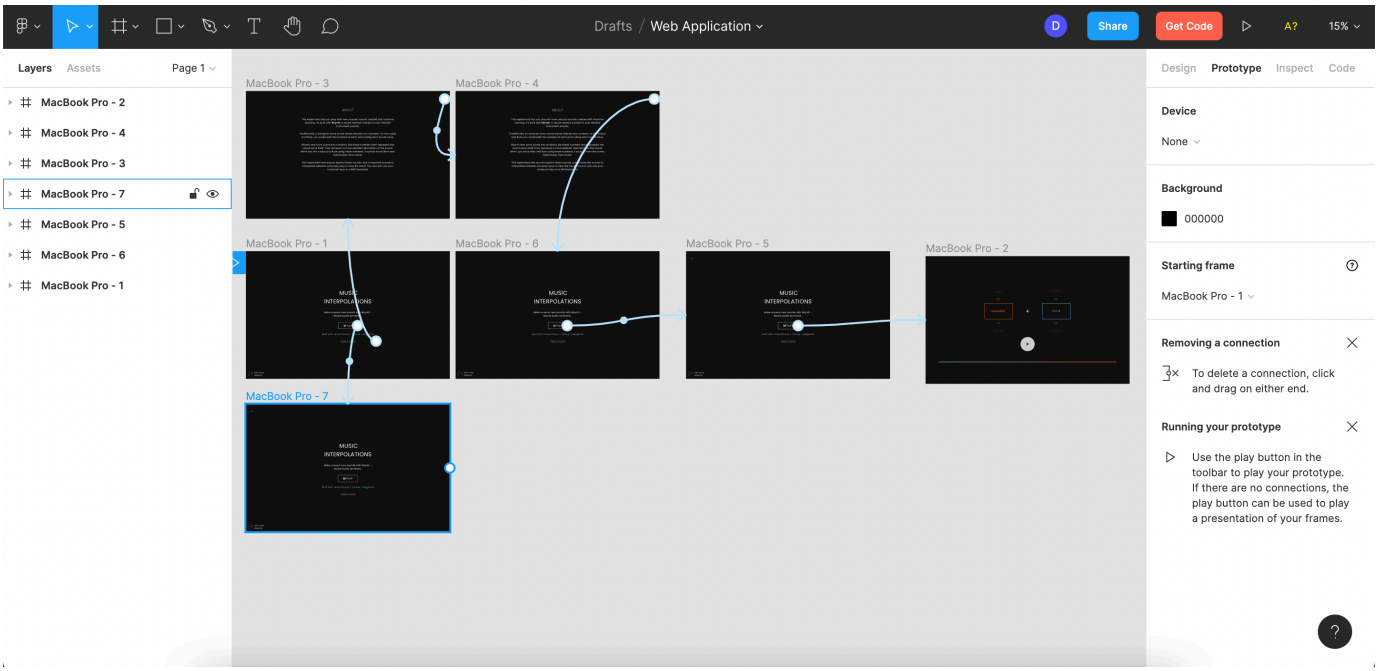
[30] "Magenta demos." https://magenta.tensorflow.org/demos/web/.

[31] "aiexperiments-sound-maker — github." https://github.com/googlecreativelab/aiexperiments-sound-maker.

[32] "Figma." https://www.figma.com/.

[33] D. Wesolko, "Peter morvillefis user experience honeycomb." https://medium.com/@danewesolko/peter-morvilles-user-experience-honeycomb-904c383b6886.

8. Appendix


Drafts / Web Application
Share
Get Code
15%
Layers
Assets
Page 1
MacBook Pro - 2
MacBook Pro - 4
MacBook Pro - 3
MacBook Pro - 7
MacBook Pro - 5
MacBook Pro - 6
MacBook Pro - 1
MacBook Pro - 3
MacBook Pro - 4
MacBook Pro - 1
MacBook Pro - 6
MacBook Pro - 5
MacBook Pro - 2
MacBook Pro - 7
MUSIC INTERPOLATIONS
Design
Prototype
Inspect
Code
Device
None
Background
000000
Starting frame
MacBook Pro - 1
Removing a connection
To delete a connection, click and drag on either end.
Running your prototype
Use the play button in the toolbar to play your prototype. If there are no connections, the play button can be used to play a presentation of your frames.