Tone Transfer: In-Browser Interactive Neural Audio Synthesis

Michelle Carney[a], Chong Li[a], Edwin Toh[a], Nida Zada[a], Ping Yu[a] and Jesse Engel[a]

[a]*Google Inc., 1600 Amphitheatre Pkwy, Mountain View, CA, 94043, USA*

Abstract
Here, we demonstrate Tone Transfer, an interactive web experience that enables users to use neural networks to transform any audio input into an array of several different musical instruments. By implementing fast and efficient neural synthesis models in TensorFlow.js (TF.js), including special kernels for numerical stability, we are able to overcome the size and latency of typical neural audio synthesis models to create a real-time and interactive web experience. Finally, Tone Transfer was designed from extensive usability studies with both musicians and novices, focusing on enhancing creativity of users across a variety of skill levels.

Keywords
interactive machine learning, dsp, audio, music, vocoder, synthesizer, signal processing, tensorflow, autoencoder,

1. Introduction

Neural audio synthesis, generating audio with neural networks, can extend human creativity by creating new synthesis tools that are expressive and intuitive [1, 2, 3, 4]. However, most neural networks are too computationally expensive for interactive audio generation, especially on the web and mobile devices [5, 6, 7]. Differentiable Digital Signal Processing (DDSP) models are a new class of algorithms that overcome these challenges by leveraging prior signal processing knowledge to make synthesis networks small, fast, and efficient [8, 9].

Tone Transfer is a musical experience powered by Magenta's open source DDSP library[1] to model and map between the characteristics of different musical instruments with machine learning. The process can lead to creative, quirky results. For example replacing a capella singing with a saxophone solo, or a dog barking with a trumpet performance.

Tone Transfer was created as an invitation to novices and musicians to take part in the future of machine learning and creativity. Our focus was on cultural inclusion, increased awareness of machine learning for artists and the general public, and inspiring excitement of the future of creative work among musicians. We did this through an interactive, in-browser creative experience.

2. User Interface Design

We created the Tone Transfer website (https://sites.research.google/tonetransfer) to allow anyone to experiment with DDSP, regardless of their musical experience, on both desktop and mobile. Through multiple rounds of usability studies with musicians, we have been able to distill the following three main features in Tone Transfer:

- *Play with curated music samples.* To understand what DDSP can do, the user could click to listen to a wide range of pre-recorded samples and their machine learning transformations in other instruments.

- *Record and transform new music.* We also provided options for users to record or upload new sounds and transform them into four instruments in browser.

- *Adjust the music.* We know that control is important for the user so we allow the user to adjust the octave, loudness, and mixing of the machine learning transformations to get desired music output.

There is also the need to help the user understand how to use Tone Transfer as well as the machine learning technology behind it. Therefore, we designed tips that guide the user through the experience and educate them on the best ways to interact with it. The user

Joint Proceedings of the ACM IUI 2021 Workshops, April 13-17, 2021, College Station USA
EMAIL: michellecarney@google.com (M. Carney); chongli@google.com (C. Li); edwintoh@google.com (E. Toh); nzada@google.com (N. Zada); piyu@google.com (P. Yu); jesseengel@google.com (J. Engel)
ORCID:

[1] g.co/magenta/ddsp

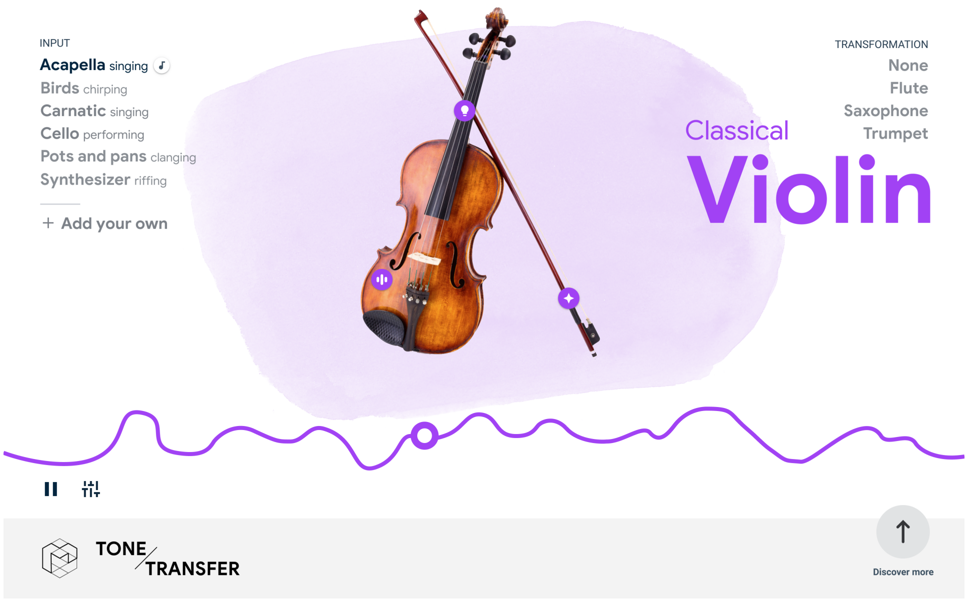

Figure 1: The web user interface of Tone Transfer

can also learn the training process of machine learning models by clicking the "Discover more" button.

3. Models

At a technical level, the goal of our system is to be able to create a monophonic synthesizer that can take coarse user inputs of *Pitch* and *Loudness* and convert them into detailed synthesizer coefficients that produce realistic sounding outputs.

We find this is possible with a carefully designed variant of the standard Autoencoder architecture, where we train the model to:

- *Encode:* Extract pitch and loudness signals from audio.
- *Decode:* Use a network to convert pitch and loundess into synthesizer controls.
- *Synthesize:* Use DDSP modules to convert synthesizer controls to audio.

We then compare the synthesized audio to the original audio with a multi-scale spectrogram loss [10, 8, 11] to train the parameters of the decoder network.

3.1. Encoding Features

To extract pitch during training (fundamental frequency, f_0), we use a pretrained CREPE network [12]. During inference we use the SPICE model, which is faster and has an implementation available in TF.js (https://tfhub.dev/google/tfjs-model/spice/2/default/1).

While the original DDSP paper used perceptually weighted spectrograms for loudness, we find that the root-mean-squared (RMS) power of the waveform works well as a proxy and is less expensive to compute. We train on 16kHz audio, with a hop size of 64 samples (4ms) and a forward-facing (non-centered) frame size of 1024 samples (64ms). We convert power to decibels, and scale pitch and power to the range [0, 1] before passing the features to the decoder.

3.2. Decoder Network

The decoder converts the encoded features (f_0, power) into synthesizer controls for each frame of audio (250Hz, 4ms). As we discuss in Section 3.3, for the DDSP models in this work, the synthesizer controls are the harmonic amplitude (A), harmonic distribution (c_k), and filtered noise magnitudes.

The DDSP modules are agnostic to the model architecture used and convert model outputs to desired control ranges using custom nonlinearities as described in [8].

We use two stacks of non-causal dilated convolution layers as the decoder. Each stack begins with a non-dilated input convolution layer, followed by 8 layers, with a dilation factor increasing in powers of 2 from 1 to 128. Each layer has 128 channels and a kernel size

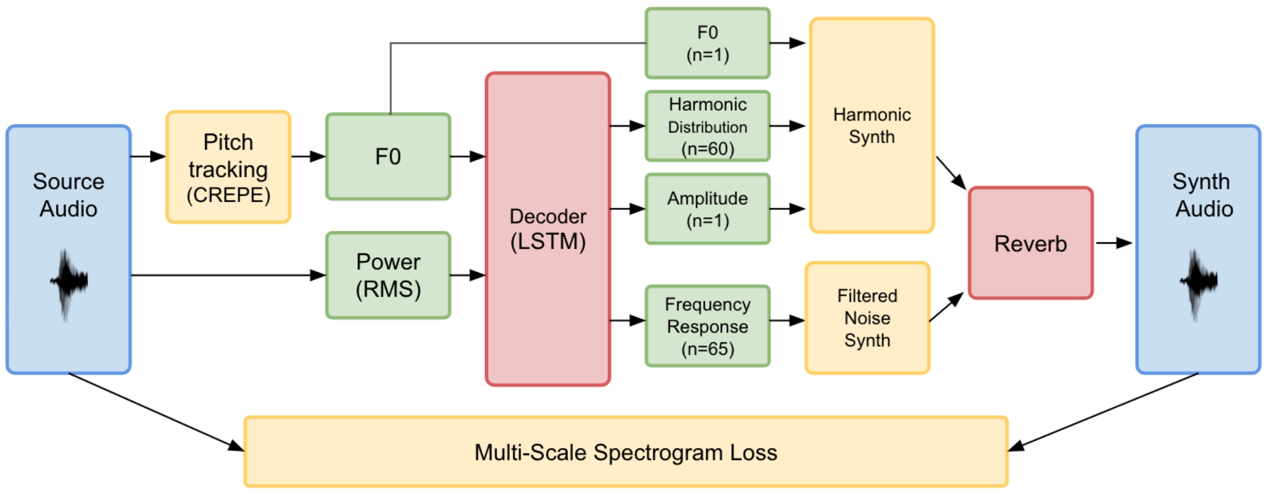

Figure 2: A diagram of the DDSP autoencoder training. Source audio is encoded to a 2-dimensional input feature (pitch and power), that the decoder converts to a 126-dimensional synthesizer controls (amplitude, harmonic distribution, and noise frequency response). We use the CREPE model for pitch detection during training and the SPICE model for pitch detection during inference. These controls are synthesized by a filtered noise synthesizer and harmonic synthesizer, mixed together, and run through a trainable reverb module. The resulting audio is compared against the original audio with a multi-scale spectrogram loss. Blue components represent the source audio and resynthesized audio. Yellow components are fixed components (pitch tracking, DDSP synthesizers, and loss function), green components are intermediate features (decoder inputs and synthesizer controls), and red components have trainable parameters (decoder layers and reverb impulse response).

of 3, and is followed by layer normalization [13], and a ReLU nonlinearity [14]. The scale and shift of the layer normalization are controlled by the pitch and power conditioning after it is run through a 1x1 convolution with 128 channels. The complete model has ~ $830k$ trainable parameters.

3.3. Differentiable Synthesizers

To generate audio, we use a combination of additive (Harmonic) and subtractive (Filtered Noise) synthesis techniques. Inspired by the work of [15], we model sound as a flexible combination of time-dependent sinusoidal oscillators and filtered noise. DDSP makes these operations differentiable for end-to-end training by implementing them in TensorFlow [16]. Full details can be found in the original papers [8, 9], but for clarity, we review the main modules here.

3.3.1. Sinusoidal Oscillators

A sinusoidal oscillator bank is an additive synthesizer that consists of K sinusoids with individually varying amplitudes A_k and frequencies f_k. These are flexibly specified by the output of a neural network over n discrete time steps (250Hz, 4ms per frame):

$$x(n) = \sum_{k=0}^{K-1} A_k(n)\sin(\phi_k(n)), \tag{1}$$

where $\phi_k(n)$ is its instantaneous phase obtained by cumulative summation of the instantaneous frequency $f_k(n)$:

$$\phi_k(n) = 2\pi \sum_{m=0}^{n} f_k(m), \tag{2}$$

The network outputs amplitudes A_k and frequencies f_k every 4ms, which are upsampled to audio rate (16kHz) using overlapping Hann windows and linear interpolation respectively.

3.3.2. Harmonic Synthesizer

Since we train on individual instruments with strong harmonic relationships of their partials, we can reparameterize the sinusoidal oscillator bank as a harmonic oscillator, with a single fundamental frequency f_0, amplitude A, and harmonic distribution c_k. All the output frequencies are constrained to be harmonic (integer) multiples of a fundamental frequency (pitch),

$$f_k(n) = kf_0(n) \tag{3}$$

Individual amplitudes are deterministically retrieved by multiplying the total amplitude, $A(n)$, with the normalized distribution over harmonic amplitudes, $c_k(n)$:

$$A_k(n) = A(n)c_k(n). \tag{4}$$

where ,

$$\sum_{k=0}^{K-1} c_k(n) = 1, c_k(n) \geq 0 \tag{5}$$

3.3.3. Filtered Noise Synthesizer

We can model the non-periodic audio components as a subtractive synthesizer, with a linear time-varying filtered noise source. White noise is generated from a uniform distribution, which we then filter with an Finite Impulse Response (FIR) filter. Since the network outputs different coefficients of the frequency response in each frame, it creates an expressive time-varying filter.

3.3.4. Reverb

To first approximation, room responses with fixed source and listener locations can be approximated by a single impulse response that can be applied as a FIR filter. In terms of neural networks, this is equivalent to a 1-D convolution with a very large receptive field (~40k). We treat the impulse response as a learned variable, and train a new response (jointly with the rest of the model) for each dataset with a unique recording environment.

To better disentangle the signal from the room response, we generate the impulse response with a filtered noise synthesizer as described in Section 3.3.3, and learn the transfer function coefficients to generate a desired impulse response. This prevents coherent impulse responses at short time scales that can entangle the frequency response of the synthesizer with the room response. At inference, we discard the expensive convolutional reverb component to synthesize the "dry" signal, and apply a more efficient stock reverb effect.

3.4. Training

Given that the DDSP model described above is for monophonic instruments, we collect data of individual instruments, and train a separate model for each dataset.

3.4.1. Data

We train models on four instruments: Violin, Flute, Trumpet, and Saxophone. Following [17] and [8], we use home recordings of Trumpet and Saxophone for training, and collected performances of Flute and Violin from the MusOpen royalty free music library [2].

Since DDSP models are efficient to train, for each instrument we only need to collect between 10 and 15 minutes of performance, and we ensure a that all recordings are from the same room environment to allow training a single reverb impulse response.

3.4.2. Optimization

We train models with the Adam optimizer [18], examples 4 seconds in length, batch size of 128, and learning rate of 3e-4. As we would like to use models to generalize to new types of pitch and loudness inputs, we reduce overfitting through early stopping, typically between 20k and 40k iterations.

4. Interactive Models

4.1. On-device Inference with Magenta.js

Musical interaction has strong requirements for close to real-time feedback and low latency. However, machine learning models are typically slow and computationally expensive, requiring GPU or TPU servers to run at all. Further, large model sizes lead to long load times before execution can even begin. Running models on-device, if possible, eliminates serving costs, decreases interactive latency, and increases accessibility. To create an interactive and scalable musical experience, we optimized and converted models to be compatible with Tensorflow.js so that they can run on-device in the browser on both desktop and mobile devices.

Even after optimization, the models are still relatively large (4mb each), so each model is only loaded on demand. This ensured the user downloads only the things they need, and nothing more, which resulted in a fast and responsive website.

The methods to extract pitches, and the four models that are on the website are then open sourced and

[2]Violin: Five pieces by John Garner (II. Double, III. Corrente, IV. Double Presto, VI. Double, VIII. Double, Flute: Four pieces by Paolo Damoro (24 Etudes for Flute, Op. 15 - III. Allegro con brio in G major, 24 Etudes for Flute, Op. 15 - VI. Moderato in B minor, 3 Fantaisies for Solo Flute, Op. 38 - Fantaisie no. 1, Sonata Appassionata, Op. 140)) from https://musopen.org/music/13574-violin-partita-no-1-bwv-1002/

made easier for anyone to download and run their own experiences. Each model comes with a set of custom values that are manually tweaked to create a more accurate output.

These methods are added to the Magenta.js library.[3]

4.2. Custom TF.JS Kernels to Preserve Precision

TensorFlow.js is a web ML platform that provides hardware acceleration through web APIs like WebGL and WebAssembly. DDSP relies on TensorFlow.js to speed up the model execution. To maintain accuracy of DDSP model on a variety of devices, we implemented a couple of special kernels that eliminated overflow ($abs(n) > 65504$) and underflow ($abs(n) < 2^{-10}$) of float16 texture when running on the TensorFlow.js WebGL backend.

For example, the DDSP model uses TensorFlow Cumsum op to calculate the cumulative summation of the instantaneous frequency, then obtain the phase from those values. TensorFlow.js implements a parallel algorithm[4] for cumulative sum, which requires log(n) writes of intermediate tensors to the GPU textures. The cumulative precision loss would cause a large shift on the final phase values. The solution is to register a custom Cumsum op that uses a serialized algorithm that avoids all intermediate texture writes and is incorporated with the phase computation.

5. Conclusion and Future Work

Tone Transfer is an example of an interdisciplinary design, engineering, and AI research teams working together to create a User Interface Design for the next wave of AI. We leverage state-of-the-art machine learning models that are both expressive and efficient, and optimize them for client-side use to enable interactive neural audio synthesis on the web. This work demonstrates that on-device machine learning can enable interactive and creative music making experiences for novices and musicians alike. The technologies that power Tone Transfer have also been open sourced as a part of Magenta.js and provide a solid foundation for further interactive studies. Future work will hopefully allow users to train their own models based on their own instruments, and explore using new types of inputs to create multi-sensory experiences.

[3]https://github.com/magenta/magenta-js/tree/master/music#ddsp

[4]https://en.wikipedia.org/wiki/Prefix_sum#Parallel_algorithms

Acknowledgments

We would like to acknowledge the contributions of everyone who made Tone Transfer possible, including Lamtharn (Hanoi) Hantrakul, Doug Eck, Nida Zada, Mark Bowers, Katie Toothman, Edwin Toh, Justin Secor, Michelle Carney, and Chong Li, and many others at Google. Thank you.

References

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

[2] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, M. Norouzi, Neural audio synthesis of musical notes with WaveNet autoencoders, in: ICML, 2017.

[3] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, A. Roberts, GANSynth: Adversarial neural audio synthesis, in: International Conference on Learning Representations, 2019.

[4] N. Mor, L. Wolf, A. Polyak, Y. Taigman, A universal music translation network, arXiv preprint arXiv:1805.07848 (2018).

[5] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, K. Kavukcuoglu, Efficient neural audio synthesis, arXiv preprint arXiv:1802.08435 (2018).

[6] L. H. Hantrakul, J. Engel, A. Roberts, C. Gu, Fast and flexible neural audio synthesis., in: ISMIR, 2019.

[7] 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, PMLR, 2018, pp. 3918–3926.

[8] J. Engel, L. H. Hantrakul, C. Gu, A. Roberts, Ddsp: Differentiable digital signal processing, in: International Conference on Learning Representations, 2020.

[9] J. Engel, R. Swavely, L. H. Hantrakul, A. Roberts, C. Hawthorne, Self-supervised pitch detection by inverse audio synthesis (2020).

[10] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, I. Sutskever, Jukebox: A generative model for music, arXiv preprint arXiv:2005.00341 (2020).

[11] X. Wang, S. Takaki, J. Yamagishi, Neural source-filter waveform models for statistical parametric speech synthesis, arXiv preprint arXiv:1904.12088 (2019).

[12] J. W. Kim, J. Salamon, P. Li, J. P. Bello, Crepe: A convolutional representation for pitch estimation, in: 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2018, pp. 161–165.

[13] J. L. Ba, J. R. Kiros, G. E. Hinton, Layer normalization, arXiv preprint arXiv:1607.06450 (2016).

[14] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: ICML, 2010.

[15] X. Serra, J. Smith, Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition, Computer Music Journal 14 (1990) 12–24.

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/, software available from tensorflow.org.

[17] G. AIUX Scouts, G. Magenta, Tonetransfer, https://sites.research.google/tonetransfer, 2020. Accessed: 2020-12-10.

[18] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).