Contemporary Machine Learning for Audio and Music Generation on the Web: Current Challenges and Potential Solutions

Mick Grierson
UAL Creative Computing Institute
m.grierson@gold.ac.uk

Matthew Yee-King
Goldsmiths, University of London
m.yee-king@gold.ac.uk

Louis McCallum
Goldsmiths, University of London
l.mccallum@gold.ac.uk

Chris Kiefer
University of Sussex
c.kiefer@sussex.ac.uk

Michael Zbyszynski
Goldsmiths, University of London
m.zbyszynski@gold.ac.uk

ABSTRACT

We evaluate specific Web-based technologies that can be used to implement complex contemporary Machine Learning systems for Computer Music research, in particular for the problem of audio signal generation. As a result of greater investment from large corporations including Google and Facebook in areas such as the development of Web-based, accelerated, cross-platform Machine Learning libraries, alongside greater interest and engagement from the academic community in exploring such approaches, Machine Learning is becoming much more prevalent on the Web. This could have great potential impact for Computer Music research, acting to democratise access to complex, accelerated Machine Learning technologies through increased usability and flexibility, in tandem with clear documentation and examples. However, some problems remain in relation to the creation of more complete Machine Learning pipelines for Music and Sound generation. We discuss some key potential challenges in this area, and attempt to evaluate some relevant solutions for developing more accessible Computer Music Machine Learning systems.

1. INTRODUCTION

The Web Audio API is potentially an excellent platform for Computer Music research. It can synthesize audio in real-time, is widely supported by all major browsers including mobile browsers, supports a variety of known audio methods, and is generally very accessible. There exists a high number of JavaScript and JavaScript compatible libraries for creating interactive sound and music systems, most of which rely on the Web Audio API (Tone.js [5], Gibber.js [10], MaxiLib.js [12]). However, it can be more challenging to create machine learning systems that run well in the browser and that integrate well with Web Audio. This is particularly true for real-time and interactive machine learning systems where processing should ideally run on the client machine, rather than some form of client / server architecture (although some good solutions exist, such as in the RAPID-MIX API [1]).

In 2018, a number of platforms have emerged having been developed with the intention of making contemporary machine learning more accessible for the web. These include Lobe.ai [4], ml5js [6], NeuroJS [3], & ml-js [6]. However, none of these projects are necessarily strong candidates for solving issues relating to Machine Learning for Computer Music research and practice. We have been attempting to understand, explore and evaluate the current state of the art in web-based Machine Learning for Computer Music in order to better describe the relevant challenges, and develop potential solutions. Below we present what we consider to be core problems and relevant solutions for creating more accessible Web Audio Machine Learning systems, including a method for training and generating 16 bit 44,100 Hz samples using Long Short Term Memory Networks (LSTMs) in web browsers. We also consider system design decisions relevant for disseminating such solutions to large numbers of online users.

2. STATE OF THE ART

As is well known, the Web Audio API[1] creates the potential for developers to build sound and music systems with a reasonably high degree of fidelity, providing low latency, a range of audio output generators, and basic analysis. Web Audio also contains a Script Processor Node that provides access to an audio buffer directly, allowing developers to create bespoke signal processing systems in the browser.

Furthermore, tools such as Emscripten[2] and WASM (WebAssembly) make it possible to implement complex Digital Signal Processing functions in C++ (or other languages such as Rust) and then convert them into browser-compatible JavaScript through transpilation.

[1] https://github.com/WebAudio/web-audio-api

[2] https://kripken.github.io/emscripten-site/

Developers can therefore use a wide range of available C++ libraries, for example, to build signal processing functions that can be run inside a Script Processor Node.

With respect to existing Machine Learning for Music in the browser, as mentioned there already exists a range of libraries and services to allow for powerful intelligent systems to be designed and deployed. Some of these, such as the RAPID-MIX API [12], provide robust and accessible tools for the creation of Musical Machine Learning systems that work across a range of platforms (Web, Embedded, Mobile, Desktop). Further, Google has offered small amounts of funding to some Universities to encourage them to create platforms based on Google's own TensorFlow[3] Machine Learning framework (for example, https://ml5js.org). Using TensorFlow.js, a JavaScript implementation of Google's TensorFlow API, more complex forms of Machine Learning (e.g. 'Deep Learning') can be run in the browser. Such systems are able to use WebGL to accelerate the training and serving of Machine Learning models in browsers with performance not vastly dissimilar (1.5 - 2 times slower in some cases according to the TensorFlow.js FAQ [11]) to equivalent models operating natively on the Desktop.

There are a number of issues with the current state of the art. Web Audio's synth graph model provides a very limited set of node types, with inconsistent implementations across browsers. In addition, it has a timing model that lacks sample-precision, and no native plug-in API. Therefore, most serious signal processing research would need to run in a Script Processor Node at some stage for the purposes of real-time interaction. This is far from ideal as the Script Processor Node runs in the main UI thread. This means that anyone wishing to implement or use custom audio processing must necessarily suffer poorer performance than that found in the natively implemented Web Audio nodes. The precise reasoning behind this design choice may not be well defined but it is nevertheless recognised as a severely limiting factor of the Web Audio API. Audio Worklets are a potential fix for this issue but as yet are not widely implemented in modern browsers. Further to this, due to security concerns in relation to exploits such as SPECTRE, Audio Worklets cannot offer multithreading capabilities due to the SharedArrayBuffer object being currently disabled in most major browsers, despite WASM supporting multithreaded compilation.

C++ and Python are significantly more mature as platforms for Signal Processing and Machine Learning than JavaScript. As mentioned, this issue can be partly mitigated through the use of transpilation tools, but performance tests have raised serious questions regarding the capability of such systems following transpilation [13]. Thirdly, 'Deep' Machine Learning systems for more complex audio operations, such as sound analysis and resynthesis, are not currently very advanced with respect to quality or flexibility [8][2], and represent challenges for web deployment - for example, support for Linear Algebra (LA) in JavaScript is not mature and often needs to be hand-coded. Core libraries for LA are written in Fortran (LAPACK, BLAS). Python's Numpy library is only minimally supported via the Numscrypt library[4]. Some JavaScript ports of Numpy are underway, and as an alternative, Eigen[5] is transpilable. However, such efforts can lead to potentially very large JavaScript libraries (over 30MB in size).

These issues all impact on the main purposes of using the Web as a platform - to make such technologies more accessible. We describe efforts to understand, explore and test solutions for a small number of these problems. Specifically, we present further data on the performance of transpiled audio analysis systems running in the Script Processor Node, and describe how we have used these to implement and train bespoke Machine Learning systems on raw audio signals in browsers. We also compare these systems with systems running natively.

3. METHOD

In order to present and explore current challenges in the design of contemporary Audio and Music Machine Learning systems for the Web, we take an existing, in-house Deep Learning system for CD quality audio generation written in Python with TensorFlow, and explore the potential of porting this system to the Web using a combination of Digital Signal Processing tools designed to interoperate with Web Audio, and TensorFlow.js. We first attempt to understand potential performance issues that may impact on the audio analysis and resynthesis approach that we require in order for our model to function. We do this by testing a proposed emscripten implementation against prior benchmarks published here [13]. Following this process, we construct a prototype analysis, training, generation and resynthesis system in JavaScript, and compare its output with that of the original implementation in Python with TensorFlow.

3.1 Emscripten Benchmarking

Transpiling is a complex process, especially when one considers questions of memory management and array processing which are pertinent in digital signal processing applications. We are transpiling from C++ to JavaScript. This means the resulting code will be running in the JavaScript interpreter in the web browser, adding another layer of complexity. Transpiled libraries often given variable results in performance tests when compared with their native counterparts. We used a performance metric also used here [13] to benchmark the maxiLib.js library[6] which is a version of the Maximilian C++ library[7] transpiled using emscripten.

The method works by calling a function many times and timing how long it takes. For example, how long does it take to call the Math.sin() function 10,000

[3] https://www.TensorFlow.org

[4] http://www.transcrypt.org/numscrypt/numscrypt.html

[5] http://eigen.tuxfamily.org/index.php?title=Main_Page

[6] https://gitlab.doc.gold.ac.uk/mick/maxi-js-emscripten

[7] https://github.com/micknoise/Maximilian

times? With that value, we extrapolate to how many times that function can run if we want to run in real-time, i.e. how many times could we run it in one sample frame. This of course assumes nothing else is happening, but it gives a simple performance base-line. Very poor performance was previously reported for emscripten-transpiled signal processing functions in this specific benchmark when compared to roughly equivalent JavaScript performing the same or similar function.

We attempt to verify this result, as its implications for the use of emscripten for the purposes of transpiling signal processing code are significant. From looking at the gists provided in the original paper [13], it appears that the time period of the original test was relatively short. We hypothesised that the test might be too short to account for the initial set-up cost of loading the emscripten-transpiled function into memory - a process that need only happen once at the start when running a programme. We therefore constructed a newer version of the test to see if performance improved when the test was conducted over a longer time period.

3.2 Python Model Structure: MAGNet

We have previously constructed an LSTM model in Python using TensorFlow for CD quality audio generation. The model, which is part of a project which we refer to as MAGNet, is designed to train only on chunks of Real-FFT magnitudes. Therefore, the LSTM can be asked to generate blocks of Real-FFT magnitudes as a sequence. We can then reconstruct the phase using any number of potential phase reconstruction approaches such as Griffin Lim, Locally Weighted Sums [9], Phase Vocoding or similar, and then use an iFFT to resynthesise the audio output. In this case we choose to use LWS, as this has a functioning library that works in Python with C++ bindings and appears to be reasonably robust to potential artefacts caused by the LSTM's artificially generated magnitude spectra. A more precise description of this model and its properties is currently in preparation for submission separately. An example python implementation of MAGNet can be found here:

https://gitlab.doc.gold.ac.uk/mick/rnn_audio.

3.3 In the Browser

Encouraged by the success of the existing Python implementation, we attempted to replicate the approach of modelling magnitudes with an LSTM as a method of creative audio generation in the browser. Given the similarities of the APIs, the structure of the model was easy to replicate in TensorFlow.js. Although an implementation of the LWS approach to reconstructing phases from magnitudes [9] is available as a Python package with C++ bindings, there is no equivalent library available in JavaScript. Therefore, LWS was ported to JavaScript using a combination of TensorFlow.js and math.js. The latter was necessary to handle matrix arithmetic with complex numbers as at the time of writing this had not yet been implemented in TensorFlow.js (0.13.1), although their use as a datatype had been introduced. The JavaScript port of the MAGNet Python example can be found here: https://gitlab.doc.gold.ac.uk/lmcca002/rnn-audio-js. Testing was carried out on an Ubuntu 16.04 LTS system with an Intel i7 processor clocked at 3GHz, 32GB of RAM, and an 8GB NVIDIA GTX1080 GPU.

4. RESULTS

4.1 Emscripten Benchmarking

Our results demonstrate that as per our hypothesis, the emscripten-transpiled function did appear to perform badly only in the first few moments of the test. We found that under a test of 250,000 iterations, the number of possible function calls increased from 11 per sample frame, to over 150 per sample frame when testing in the Chrome browser. We also found that on our Ubuntu 16.04 test system, Firefox was approximately twice as performant as Chrome, achieving up to 300 calls per sample frame. This is much closer to the performance of the original C++ library, and more accurately represents the real-world experience of users who are operating emscripten-transpiled C++ functions in JavaScript. This result may have further implications for those testing the performance of emscripten and WASM transpiled routines. Further to this, it should be mentioned that our test did not factor out the initial period wherein the function was not performing optimally, and in the future we would recommend doing so.

```
var fft = new maximJs.maxiFFT();
var testosc = new maximJs.maxiOsc();
fft.setup(512, 256, 128);
var currentFrame = 0;
var start = Date.now();
var x=0;
var testruns=100000000;
for (var i = 0; i < testruns; i++) {
     testosc.sinewave(100);


}

var finish = Date.now();
var elapsed = finish - start;
console.log(elapsed);
var millisPerOsc = elapsed / testruns;
var bufferTime = 1024/44100.0;
console.log(bufferTime / millisPerOsc);
```

Figure 1: Example benchmark code for testing the performance of transpiled libraries.[8]

[8] Example code available at https://gitlab.doc.gold.ac.uk/mick/maxi-js-emscripten/tree/master/testing

4.2 MAGNet JavaScript performance

When using identical model structures, analysis parameters, hyperparameters and audio training data, training took approximately twice as long in JavaScript in comparison to our existing Python implementation. On our test system, training our model on a 9 second piece of Audio for 1000 epochs took approximately 12 hours. Python code on the same machine was able to train an identical model for 1000 epochs in just over 5 ½ hours. Training time for both models can be reduced greatly by simplifying the model and the size of the audio segment.

Audio generation from the same system was not dissimilar. Generation took approximately twice as long as the Python version of the algorithm. This meant that short samples of up to 2-3 seconds long could be generated in around 5 seconds. This is not good enough for real-time audio generation from deep learning systems in web browsers. However, it is within reasonable and acceptable limits for an online audio generator.

This supports claims discussed earlier in the TensorFlow.js documentation regarding performance. Reconstruction using the new JavaScript LWS method works well enough for testing, although further work is being carried out in order to reduce small artefacts occurring in the reconstruction.

5. DISCUSSION

5.1 Performance of Transpiled Libraries

Our tests demonstrate that some performance issues associated with transpilation approaches have more to do with the specific ways in which transpiled code is being deployed and tested, rather than the explicit limitations of the code itself. For example, in our testing, systems which have been reported as only being able to produce < 10 function calls per sample frame have been demonstrated to be capable of many hundreds of function calls per sample frame. This is because the methods described in some related research do not account for the cost of setting up the function, which will vary from case to case depending on how the native code has been transpiled into asm.js by emscripten. Furthermore, we suspect that performance factors for asm.js may depend greatly on whether browsers are using ahead-of-time or just-in-time compilers. When testing transpiled systems it may be common to fail to account for compile times, which represent another large, fixed cost when any asm.js function is first called. It may be possible that this cost is far less great with WASM code, and this could be interesting to explore.

5.2 Issues Porting Python Machine Learning Systems to JavaScript

The need to port LWS to JavaScript revealed limitations in the TensorFlow.js library in relation to handling arithmetic with complex numbers. This highlights issues that can arise when developing for early stage, pre-release libraries (as mentioned, these data types are now fully supported). Further challenges came from porting often dense NumPy code into JavaScript. As mentioned above, there is yet to be a complete porting of the NumPy API and all its functionality. Considering the prevalence of its use in ML, an automated approach to this would greatly open up porting of SOTA ML solutions for use in the browser.

In general, it seems clear that it may be more sensible at present to train models in Python, and then to load them via TensorFlow.js so that they can be served over the web. In our testing there were inconsistencies with the effectiveness and performance of the JavaScript-trained models when compared to Python models of almost identical structure. However, those same models trained in Python performed identically in both JavaScript and Python. We are still investigating potential causes of these observations.

6. CONCLUSION

We present a potential structure and approach for creating contemporary web-based Machine Learning systems for music based on existing C++ and specifically rewritten Python libraries. We demonstrate that investigating and evaluating transpilation approaches can sometimes lead to misleading results that potentially mis-characterise the performance of such systems. Further to this, we compare methods and approaches for the development of LSTM-based audio generators that are capable of CD quality audio reconstruction using current web technologies. It is fair to say that building such systems is still challenging, and represents some considerable engineering effort. Nevertheless, they are definitely possible to construct, and offer considerable potential for Computer Music research. Finally, we have released our online LSTM audio generator software under an open source license to coincide with ICMC 2019.

Acknowledgments

This work is funded by the Arts and Humanities Research Council as part of the MIMIC project, UK grant reference AH/R002657/1.

7. REFERENCES

[1] Bernardo, F., Grierson, M., and Fiebrink, R. User-Centred Design Actions for Lightweight Evaluation of an Interactive Machine Learning Toolkit. *Journal of Science and Technology of the Arts*, vol. 10, no. 2, pp.2-25., 2018

[2] Donahue, C., McAuley, J.,Puckette, M., Synthesising Audio with Generative Adversarial Networks, arXiv:1802.04208, 2018

[3] Huernermann, J., NeuroJS, [Online], Available www.github.com/janhuenermann/neurojs, [Accessed 15.01.2019]

[4] Lobe Artificial Intelligence, Inc., Lobe - Deep Learning Made Simple. [Online], Available www.lobe.ai, [Accessed 15.01.2019]

[5] Mann, Y. Interactive music with Tone. js. In Proc. WAC'15, 2015.

[6] ml-js, Machine learning and numerical analysis tools in JavaScript for Node.js and the Browser, [Online], Available www.github.com/mljs, [Accessed 15.01.2019]
[7] NYU ITP, ML5 - Friendly Machine Learning for the Web, [Online], Available www.ml5js.org, [Accessed 15.01.2019]
[8] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A generative model for raw audio. *arXiv:1609.03499*, 2016.
[9] Le Roux, J., Kameoka, H., Ono, N., Sagayama, S., Fast Signal Reconstruction from Magnitude STFT Spectrogram Based on Spectrogram Consistency, in Proc. DAFX'10, pp. 397--403, Sep. 2010.
[10] Roberts, C, and JoAnn K. Gibber: Live coding audio in the browser. In Proc, ICMC'12, pp. ,2012.
[11] TensorFlow, TensorFlow.js FAQ, [Online], Available www.js.TensorFlow.org/faq, [Accessed 15.01.2019]
[12] Zbyszyński, M., Grierson, M., & Yee-King, M.. Rapid Prototyping of New Instruments with CodeCircle. In Proc NIME'17, pp. ,2017
[13] Zbyszyński, M., Grierson, M., Fedden, L., and Yee-King, M., Write once run anywhere revisited: machine learning and audio tools in the browser with C++ and emscripten, In Proc. WAC'17, pp. , 2017