Improving Real Time Neural Network Inference Through Approximation

Tom Schroeder*
David Silverman*
jts13@illinois.edu
davids17@illinois.edu
University of Illinois Champaign-Urbana
USA

ABSTRACT

The application of neural networks to realtime applications, such as audio, requires fast methods for inference on trained models. A common approach is to use a systems programming language such as C++ or Rust. While faster than Python, these approaches also struggle as network layer size increases. We consider the use of approximation techniques to speed up inference while minimizing error both numerically and in terms of audio perception.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Real-time system architecture**; • **Applied computing** → **Sound and music computing**; • **Theory of computation** → **Numeric approximation algorithms**.

KEYWORDS

Real-Time Neural Network Inference, Neural Network Approximation Techniques, Activation Function Optimization, Matrix Multiplication Approximation, Audio Processing Optimization, Performance Enhancement in Neural Networks, Loop Perforation, Efficient Inference for Audio Applications, Systems Programming for Machine Learning, C++ Neural Network Engines

ACM Reference Format:
Tom Schroeder and David Silverman. 2024. Improving Real Time Neural Network Inference Through Approximation. In *Proceedings of Approximate and Probabilistic Programming (CS 521).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Inference is the process of applying input data to a trained neural network. For streaming applications, such as audio and video processing, inference through interpreted or garbage collected programming languages, such as Python, may cause unexpected and unwanted lag or audio side-effects such as popping or dropout. This is despite the underlying network inference happening in libraries such as PyTorch or Tensorflow.

In *RTNeural: Fast Neural Inferencing For Real-Time Systems*, Jatin Chowdhury introduces a C++ engine for neural networks that can operate on a "safe" subset of ML model functions and serve as a drop-in replacement for inference on a pretrained PyTorch model.[3] "Safe" here means guaranteed to not have divide/0 or other NaN errors.

RTNeural can be up to 1 billion times faster than PyTorch or TensorFlow, and, as a result, has been used in more than a dozen open source audio projects.[4] However, its performance gain drops dramatically when the number of nodes per layer is greater than 2 dozen depending on the activation function and network architecture.

Chowdhury states, "This performance trade-off between larger and smaller layer sizes is to be expected, since RTNeural is intentionally optimised to perform better for smaller layer sizes." In this paper, we examine if this limitation can be overcome through approximation techniques including, approximated activation functions, loop perforation, approximated matrix multiply, and pruning.

In this paper we present the use of approximation techniques for activation functions, loop perforation, and matrix multiply in RTNeural via source code modification. We describe the approaches attempted and evaluate the resulting performance changes in terms of speed, numerical accuracy, and aural perception on guitar amplifier modeling neural networks. All code available in our GitHub repositories, and selected code excerpts are in the appendices.[1]

2 BACKGROUND: REAL-TIME AUDIO PROGRAMMING AND RTNEURAL

Real-time audio processing has many considerations beyond just interpreted vs. compiled and automated garbage collection triggering a delay in processing. It's fundamentally about worst-case guarantees there will be no unexpected impact on the audio thread. A popular web article in the audio programming community states the following "common mistakes".[21]

(1) Don't hold locks on the audio thread.
(2) Don't use Objective-C/Swift [ed. or Python] on the audio thread.
(3) Don't allocate memory on the audio thread.
(4) Don't do file or network IO on the audio thread.

All of these apply to Python, where locks, memory, and potentially even file IO can happen without user direct control. Given

*Both authors contributed equally to this research.

CS 521, May 1, 2024, Chicago, IL

[1] RTNeural modifications, benchmarking, and associated C++ and Rust in https://github.com/jts13/cs521-project. NN Models and Python code for algorithm development in https://github.com/dsagman/Project521.

that Python is the primary language for machine learning development via PyTorch or TensorFlow libraries, we need a systems programming language such as C++ or Rust, where the developer has explicit control over machine resources and related control flow.

2.1 RTNeural

RTNeural was developed with these concerns in mind to support real-time audio applications such as guitar amp or pedal modeling. To speed up inference, and guarantee performance RTNeural replaces the forward logic for a NN model with custom C++ code. RTNeural provides no training support and relies on loading a trained model and weights from a TensorFlow or PyTorch model.

This custom code replicates the activation and layer logic that would typically be implemented with Tensorflow or PyTorch. As of our writing, RTNeural supports *tanh*, *ReLU*, *Sigmoid*, *SoftMax*, *ELu*, and *PReLU* activation functions, and *Dense*, *GRU*, *LSTM*, *Conv1D*, *Conv2D*, *BatchNorm1D*, and *BatchNorm2D* layers.

RTNeural provides several options for the backend math library: Eigen, xsimd, or STL, the C++ standard template library. The RTNeural developer notes that Eigen generally produces the best results, and it is also the library used by other audio developers (GuitarML, NAM). This is likely because Eigen is explicitly for vectorized linear algebra functions. However, for fundamental functions, such as *tanh*, Eigen is simply a wrapper for STL. XSimd also performs vectorization as a wrapper. We therefore focus our efforts on STL because results obtained there for both fundamental functions and matrix multiplication are the baseline for all the math libraries.

2.2 Selection of RTNeural

Because RTNeural is exact and does not incorporate any approximation methods, it is a good candidate for approximations. Further, the repository author has identified this as an area to pursue.[2]

While there are other approaches such as TensorFlow Lite and ONNX, we chose to focus on RTNeural as a starting point for our research. Our approximation methods should be generalizable to any inference engine.

3 HYPOTHESIS

Our hypothesis is that approximation will allow for larger layer sizes to be inferred in real time. We base this on the benchmarking results from the RTNeural project itself. All of the activation functions and layers have significant dropoff in speed as layer size goes beyond the range 2^4. In figure 1 we can see the benchmark for *tanh* as an example. Given similar results for *sigmoid* and for *dense*, *convolutional*, and *LSTM* layers, we believe approximation should be able to push the boundary where RTNeural is more effective than other techniques.

4 METHOD

Our method was to examine the RTNeural code to locate "knobs" that we could adjust with approximation. We identified the following:

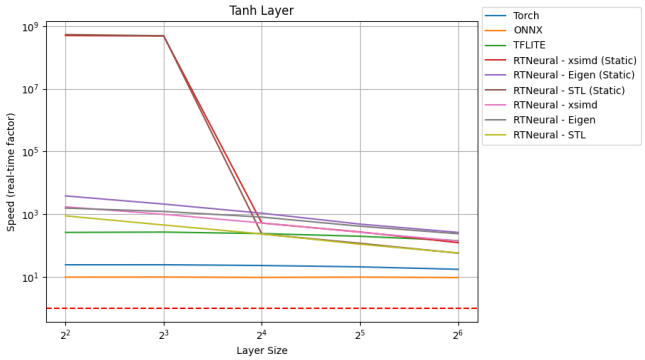


Figure 1: RTNeural Benchmark results from original paper. The red dashed line indicates minimum for real-time. Torch, ONNX, and TFLite are other libraries for inference with varying capability for low-spec hardware/real time. The lines for RTNeural are alternative C++ math libraries.[4]

(1) Exact, non-linear functions: We identified calls to *tanh*, *sigmoid*, and *exp*. Although *sigmoid* is defined via *exp*, we examined calculating its approximation with a low-order polynomial.
(2) Matrix multiply: Given that our application is audio inference, the shape of our input tensor at each inference is a scalar, and hence 0 dimensional. This limits the complexity matrix multiply for most layers except for connections to large *dense* layers. We therefore focused on *dense* layers.
(3) Loops: RTNeural uses loops for matrix multiply, but we already identified that usage above. The other use of loops is in *batchnorm*, *conv1D*, and *conv2D*. We focused on *conv1d* as a potential layer that use loops that we can perforate. We did not consider *conv2d* given our data shape is 1D. We also did not attempt *batchnorm* at this point.

4.1 Approximate Activation

Given that we are focused on speed of inference, we approximated *tanh*, *exp*, and *sigmoid* with approaches that limited computation. We chose four methods to compare: Taylor, spline, Schraudolph, and K-TanH. We explain these further and their applicability to a given function table 1.

4.1.1 Taylor Series. For Taylor series expansion, we took the first 3 terms for *tanh* and *sigmoid* and bounded x such that $|x| < b$. Outside of the range $[-b, b]$ we used the associated limit [-1,1] and [0,1], respectively. For *exp* we used the first 4 terms (including 1).

For example, for *tanh*, $b = 1.365$ the calculation is as follows:

$$tanh(x) = \begin{cases} 1 & \text{if } x \geq 1.365 \\ x - \frac{1}{3}x^3 + \frac{2}{15}x^5 & \text{if} |x| < 1.365 \\ -1 & \text{if } x \leq -1.365 \end{cases} \quad (1)$$

To further reduce multiplications, we use Horner's method and pre-calculated coefficients:

$$x - (x \cdot x)(0.\overline{3}x + (x \cdot x)0.1\overline{3}x$$

[2] https://www.youtube.com/watch?v=UPaphyB4nmY

function	Taylor	Padé	spline	Schraudolph	K-TanH
tanh	x	x	x	via exp	x
exp	x		x	x	
sigmoid	x	via tanh	x	via exp	via tanh

Table 1: Approximating Activation Functions. x indicates an approach used.

4.1.2 Padé. Padé approximations are rational approximations of a function $f(x)$ using polynomials, where the ratio of polynomials $P_m(x)$ and $Q_n(x)$ of degrees m and n, respectively, approximate $f(x)$. These have been considered for activation approximation by Timmons and Rice.[20]

$$f(x) \approx \frac{P_m(x)}{Q_n(x)} = \frac{a_0 + a_1x + a_2x^2 + \ldots + a_mx^m}{1 + b_1x + b_2x^2 + \ldots + b_nx^n}$$

The n/m Padé approximation will agree with the first $m + n$ terms of the Taylor series. According to Wolfram Alpha: "Padé approximations are usually superior to Taylor series when functions contain poles, because the use of rational functions allows them to be well-represented."[3] This, in general, prevents Taylor series tendency towards +/-∞.

For our implementation, we used code from the JUCE audio processing library, which uses coefficients [7/6] and works on the range $|x| < 5$. [4]

4.1.3 spline. A spline is a curve fitted to control points. With multiple splines, a function can be approximated in a piece-wise fashion. This is the approach used in *Efficiently inaccurate approximation of hyperbolic tangent used as transfer function in artificial neural networks*, where the authors use cubic splines to interpolate *tanh*.[16]

Like the Taylor series, above and below bounds, the function is set to 1, -1, respectively. The referenced paper uses $|x| < 18$ as the bound (versus 3.75 for our Taylor). They use the symmetry around 0 to only calculate x>0, and divide the reduced range into three intervals: $n_0 = 0, n_1 = 0.371025186672900, n_2 = 2.572153900248530, n_3 = 18$.

Then, for the intervals $[n_0, n_1], [n_1, n_2], [n_2, n_3]$, they provide corresponding cubic polynomial approximations s_0, s_1, s_2, s_3:

$$\begin{aligned}
s_0 &= -0.3695076086125492x^3 + 0.01987219343897867x^2 + 1 \\
s_1 &= 5.928356367224758 \cdot 10^{-2}(x - n_1)^3 \\
&\quad - 3.914176949486042 \cdot 10^{-1}(x - n_1)^2 \\
&\quad + 8.621472609449146 \cdot 10^{-1}(x - n_1) \\
&\quad + 3.548881072496229 \cdot 10^{-1} \\
s_2 &= -3.347599023061577 \cdot 10^{-6}(x - n_2)^3 \\
&\quad + 5.456777761558641 \cdot 10^{-5}(x - n_2)^2 \\
&\quad + 7.066442941005233 \cdot 10^{-4}(x - n_2) \\
&\quad + 9.884026213740197 \cdot 10^{-1}
\end{aligned}$$

4.1.4 Schraudolph and Schraudolph-NG. In 1999, Schraudolph developed a fast *exp* function that manipulates a standard floating point number as individual bytes to approximately compute, for any base, B, B^x.[15] This is similar in approach to the "bit-twiddling" fast inverse square root method made famous from its use in the video game *Quake III Arena*.[5]

Schraudolph splits the 64-bit double into two 4-byte integers i and j as depicted in figure 2. The algorithm uses only i and is as follows:

$$i = ay + (b - c)$$

Where $a = 2^{20}/\ln(2)$, $b = 1023 \cdot 2^{20}$, and c is an adjustment parameter that controls accuracy. The cited paper recommends choosing c to be 60801.

In our experiments, including with RTNeural, we used 32-bit floating point values instead of doubles, so the formula above must be adjusted accordingly. To translate the algorithm above, we update $a = 2^7/\ln(2)$ and $b = 127 \cdot 2^7$, with an adjustment parameter $c = 8$. To illuminate the source of these constants, the code included in the appendix provides a (compile-time) derivation of these values.

We also considered an alternate version, which we call *Schaudolph-NG*, proposed by Schraudolph on Stack Overflow in 2018 in response to a question about the fastest possible exp function.[6]

> A good increase in accuracy in my algorithm [...] can be obtained at the cost of an integer subtraction and floating-point division by using FastExpSse(x/2)/FastExpSse(-x/2) instead of FastExpSse(x). The trick here is to set the shift parameter (298765 above) to zero so that the piecewise linear approximations in the numerator and denominator line up to give you substantial error cancellation.

What we find interesting about this quote is twofold:

(1) There seems to be endless capacity to eke out additional, if small, gains in performance of these approximation techniques.
(2) This technique, discovered in a comment thread, was the most effective.

We provide a deep dive on Schraudolph in C++, Rust and assembly in the Appendix C.

4.1.5 K-TanH. K-TanH is a fast *tanh* approximation intended for use in neural networks. Similar to Schraudolph, it directly manipulates the bit representation of a number, but rather than casting the number into integers it uses a lookup table based on digits from the exponent and mantissa. [12]

[3] https://mathworld.wolfram.com/PadeApproximant.html
[4] https://github.com/juce-framework/JUCE/blob/master/modules/juce_dsp/maths/juce_FastMathApproximations.h#L92
[5] https://en.wikipedia.org/wiki/Fast_inverse_square_root
[6] https://stackoverflow.com/questions/47025373/fastest-implementation-of-the-natural-exponential-function-using-sse

sxxxxxxx	xxxxmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm
1	2	3	4	5	6	7	8
iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii	jjjjjjjj	jjjjjjjj	jjjjjjjj	jjjjjjjj

Figure 2: Schraudolph's representation of an 8 byte IEEE-754 double (top row) with sign s, exponent x, and mantissa m, or as two 4-byte integers i and j (bottom). Only i is used by the algorithm.[15]

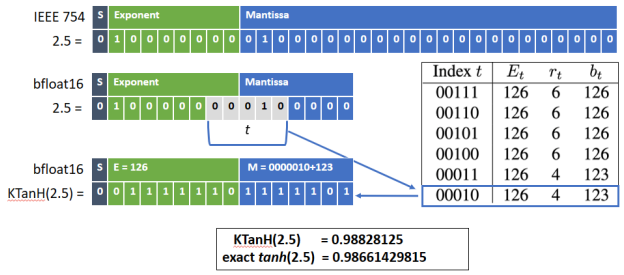


Figure 3: K-TanH algorithm.

This method also supports *sigmoid*, as it can be represented as a rescaled *tanh*:

$$sigmoid(x) = \frac{1}{1+e^{-x}} = \frac{1+tanh(\frac{x}{2})}{2}$$

The algorithm works as follows (see figure 3):

(1) If $x < -3.75$, K-TanH = -1, if $x > 3.75$, K-TanH = 1. (Same as equation 1)
(2) If $|x| < 2.5$, K-TanH = x. Otherwise:
(3) Convert float to bfloat16 by dropping the least significant 16 bits of the mantissa.
(4) Use the least 2 significant bits of the exponent and 3 most significant bits of the mantissa to form lookup table index t. look up t in a pre-computed table to obtain E_t (exponent), r_t (right shift), and b_t (bias term).
(5) Use the found values in the following function on the input i, where s_i is the sign of the input, M_i is the mantissa of the input and $\gg$ indicates right shift (divide by 2).

$$\text{output sign} = s_i$$
$$\text{output exponent} = E_t$$
$$\text{output mantissa} = (M_i \gg r_t) + b_t$$

Using the example from figure 3, $i = 2.5$, $s_i = 0$, $E_i = 128$, $M_i = 0100000$. The lookup value $t = 00010$, and $E_t = 126$, $r_t = 4$, $b_t = 123$. Applying the formula, the sign stays 0, the output exponent is 126, and the output mantissa is the input mantissa right shifted 4 places to 0000010 plus the bias of 123 to give 1111101. The resulting combined floating point number has an error of less than -0.002.

4.2 Loop Perforation

We identified loops in RTNeural in *conv1d*, and we considered the method used in *PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions*[7] in which a mask is used to reduce convolution to a more efficient matrix multiply operation.

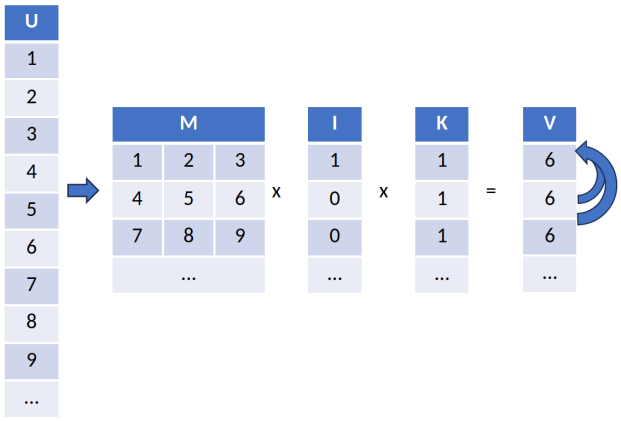


Figure 4: Perforated *conv1d*. U is input, M is mask, I are indices in M to skip or use (0 or 1) K is kernel, and V is output.

Figure 4 contains an example applied to a 1D matrix, which is the shape of our audio data. The input U is chunked into a array of arrays, M, with chunks of the size of the kernel, K. I is a binary matrix that indicates if we are going to skip (0) or use (1) the associated row of M.

I can be generated in a variety of ways. We coded two versions from the paper: random choice and pseudo-random. Random selects 0 or 1 with a probability p. The pseudo-random algorithm comes from Graham in *Fractional Max Pooling*.[8] A sequence of 1's and 2's are generated with the formula $\alpha \cdot u$ where α is a real number $\in (1, 2)$ and u is a uniform distribution $(0, 1)$. α functions similarly to p in the pure random approach.

Finally, the output matrix V is formed by matrix multiplication. All zero entries are replaced with the value of the nearest non-zero neighbor, with ties broken randomly. This keeps the structure and provides low-cost interpolation between exactly calculated values. Note: Our code chooses the nearest neighbor deterministically, as it takes less compute time to not have to compute a random value for tie-break.

We provide a Python version of this algorithm in Appendix C.

4.3 Matrix Multiply

For matrix multiply approximation, we used the randomized matrix reduction as described by Drineas.[6] The steps are as follows for two matrices $A_{m\times n}$ and $B_{n\times p}$.

(1) For each row of A and column of B calculate the Euclidean L2 norm. Assign the values to a variable $prob_k$ (equation 2).
(2) Select $c \in [1..n]$, where c is reduced size of the matrices n dimension, e.g. $c = n/2$ reduces the dimension by half.
(3) Select c rows from A and c columns from B choosing those with the largest $prob_k$ values. Call these A_*i, B_i* and $prob_i$ to align the $prob$ values with the reduced rows and columns.
(4) Matrix multiply each rows and columns of the reduced matrices and divide by the associated $prob_i$. Finally divide the entire result by c. These division rescale the output to be closer to the unreduced original (equation 3).

We provide a Python version of this algorithm in Appendix C.

4.3.1 Johnson Lindenstrauss Lemma. We also considered and coded an approximate matrix multiply based on the Johnson Lindenstrauss (JL) Lemma.[7] JL reduces the dimensionality of a matrix through random projection onto a lower dimension hyperplane. While the idea was compelling, we found that the compute time for projection of our relative small matrices overwhelmed any benefit of reduced matrix sizes.

$$prob_k = \frac{\|A_{*k}\|_2 \cdot \|B_{k*}\|_2}{\sum_{k'=1}^{n} \|A_{*k'}\|_2 \cdot \|B_{k'*}\|_2} \tag{2}$$

$$CR = \sum_{t=1}^{c} \frac{1}{c \cdot prob_{i_t}} A_{*i_t} B_{i_t*} = \frac{1}{c} \sum_{t=1}^{c} \frac{1}{prob_{i_t}} A_{*i_t} B_{i_t*} \tag{3}$$

5 RESULTS

5.1 Approximation Techniques

First we look at the results of the our approximation techniques run by themselves on random data, also known as Monte Carlo simulation.

5.1.1 tanh. In figure 5 we show that Padé and Schraudolph dominate the Pareto frontier. Padé has a nearly 2.5x speedup with no loss in accuracy. Schraudolph-NG has over 4x speedup with accuracy loss around 1e-6 versus base Schaudolph with an order of magnitude more error. Lastly, Taylor is slightly the fastest, but with terrible accuracy.

We were able to get Schraudolph-NG to be faster by using SIMD instructions to perform the required two exp instructions simultaneously.

Note: In our initial testing, K-TanH was fastest when we ran on a sweep of the interval rather than Monte Carlo. We suspect that the processor branch prediction and cache caused this speedup due to the lookup table nature of K-TanH.

5.1.2 Random Matrix Multiply. Figure 6 shows results from the random matrix multiply algorithm. Note that for small matrices the extra setup time of computing the necessary column and row norms resulted in worse performance than exact matrix multiply (dotted line).

The cutoff for significant speedup is a factor of 0.5, meaning half of the matrix is dropped, and a matrix size of 2^8. But, in a trial run of a model with two dense layers of size 2^8, the resulting audio was too noisy to be useful.

Given that the maximum layer size for our real-time audio is 2^8, we did not explore random matrix multiplication further.

5.1.3 Perforated Conv1d. Figure 7 shows the performance of the 1-dimensional perforated convolution versus kernel size (k) and input layer size. We note that the setup penalty makes this approach only useful for larger layer and kernel size. This is evident in table 2.

Perforation has the most significant improvement at 256 layer size of 62 seconds for eigen approx versus 96 seconds for eigen exact. However, this was the time to process 1 second of audio on our test environment of a MacBook Pro M1 laptop.

We suspect that there is more possibility for loop perforation to be useful than random matrix multiply, but given the layer size required for performance gain versus the need for real-time, we do not believe it is practical given the likely compute power available at inference time.

5.2 RTNeural Benchmarks: Relative Performance and Numerical Accuracy

The RTNeural developer provides benchmarking code that contain networks with random weights for testing purposes.[8] We used these benchmarks with our approximation function swapped in to both the C++ standard library (STL) and Eigen library.

Our benchmark environment was a MacBook Pro with an M1 Max and 64 GB of RAM using macOS Sonoma. Full results using Schraudolph-NG and loop perforation (only for *conv1d*) showing wall clock time are in table 2. (Graphical version of this data is in the Appendix A.

Figure 8 shows the use of Schraudolph-NG in a layer with *tanh* activation. Our approximation versus baseline STL is consistently an order of magnitude or more faster. Our approximation applied with Eigen is also faster than exact Eigen for all layer sizes greater than 2^3, although not to the same extent.[9]

We believe that our increased performance for *tanh* over Eigen and XSIMD are due to the lack of vectorized operations when running this benchmark. It is purely about the effectiveness on the activation only.

Full results on all activation functions and layers are provided in the Appendix in figure 10. In the majority of cases there was more opportunity for vectorization, such as matrix multiply, and as a result our approximated Eigen is usually the fastest.

5.3 Guitar Audio Effect Model

While numerical benchmarks are useful, ultimately RTNeural is used for modeling guitar audio effects such as an overdrive pedal or a high-end amplifier. We need to evaluate to what degree the approximations in our method degrade the audio versus the speed up in processing time.

5.3.1 Model Design. Since RTNeural provides no training, to have an audio output that can be evaluated by listening, requires using an existing neural network model or building one.

There is a lot of material online about guitar modeling networks. The following are popular approaches: WaveNet and LSTM[23], WaveNet [22], Neural Amp Modeler (NAM),[10] and temporal convolution networks (TCN)[18]. There is even a code repositories available that can be run in Google Colab.[11]

These approaches all are based on recurrent neural network approaches such as RNN, LSTM, and GRU, in which some amount of data from the past is retained in the forward pass, see figure 9.

[7] https://web.stanford.edu/class/ee270/Lecture6.pdf

[8] https://github.com/jatinchowdhury18/RTNeural-compare

[9] Note: For *tanh* we did not see the same sharp decline in performance at 2^4 as in the RTNeural benchmarks in figure 1. This is likely due to variations in our experimental setup (ARM) versus the GitHub original (Intel).

[10] https://github.com/sdatkinson/neural-amp-modeler

[11] https://github.com/GuitarML/GuitarLSTM/blob/main/guitar_lstm_colab.ipynb

[12] https://gotensor.com/2019/02/28/recurrent-neural-networks-remembering-whats-important

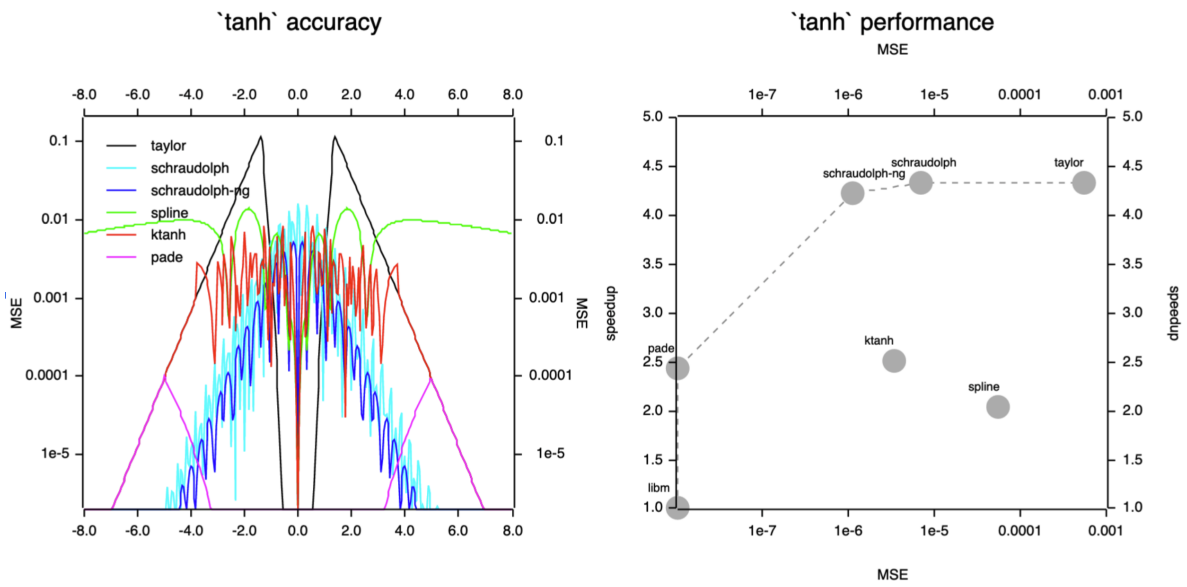


Figure 5: Accuracy versus performance of tanh approximation methods. Left is numerical accuracy over the interval $|x| < 8$. Right is Pareto frontier. Up and to the left is better.

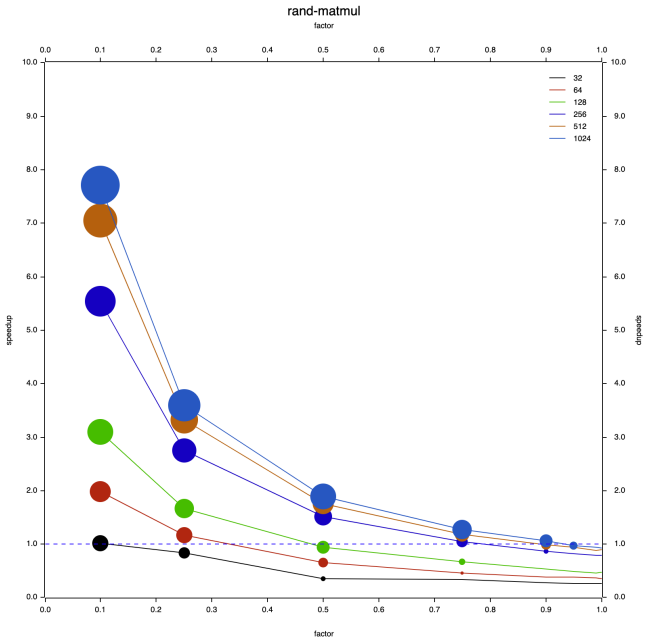


Figure 6: Random matrix multiplication. Horizontal axis is percentage of original matrix retained (factor). Size of plot point is relative error. Dashed line is performance of exact matrix multiply. Each line represents a size of the matrix input.

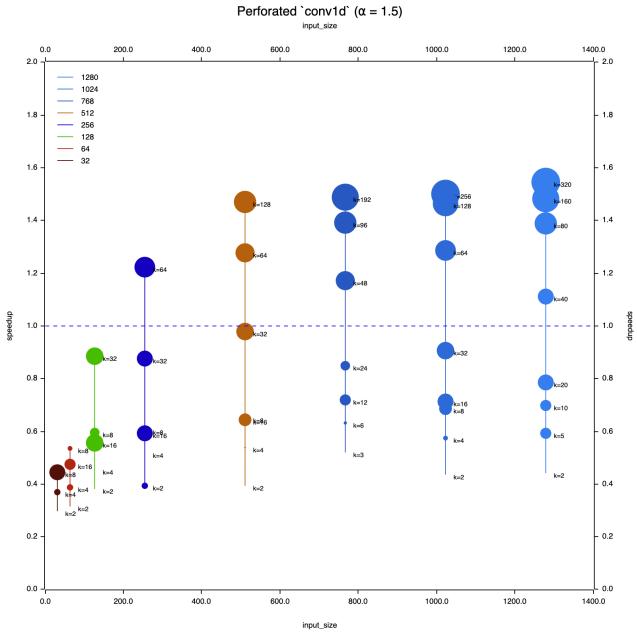


Figure 7: Perforated Conv1d using pseudorandom sequence. Dashed line is baseline performance without perforation. k is the kernel size. The size of circle represents relative error. Horizontal axis is input layer size, and each vertical line is a different size.

Layer Type	Layer Size / Compute Method	4	8	16	32	64	128	256
conv1d	eigen	**0.004805**	**0.0121261**	0.064102	0.148811	1.407740	11.940000	96.349900
	eigen approx	0.011008	0.017569	**0.0493559**	**0.101118**	**0.908134**	**7.70235**	**62.1946**
	stl	0.007266	0.054377	0.653435	1.759890	15.142400	123.585000	998.607000
	stl approx	0.012041	0.042484	0.435642	1.109390	9.020560	78.410500	620.229000
	xsimd	0.010408	0.062329	0.402620	0.556938	2.716340	21.329400	233.906000
dense	eigen	0.008347	0.005731	0.004573	0.008962	0.032131	0.117727	0.763305
	eigen approx	0.006982	**0.00543071**	**0.00435225**	**0.00886629**	**0.0316397**	**0.116607**	**0.750825**
	stl	0.005169	0.008346	0.019095	0.089807	0.499090	2.694920	13.509600
	stl approx	**0.00515608**	0.008075	0.018986	0.090697	0.509343	2.715540	13.510800
	xsimd	0.008295	0.010002	0.018357	0.049436	0.158758	0.630264	3.669640
gru	eigen	0.014499	0.016987	0.026294	0.062539	0.211060	1.265960	4.609600
	eigen approx	**0.012815**	**0.0147148**	**0.0202834**	**0.0521115**	**0.191262**	**1.22934**	**4.4859**
	stl	0.033454	0.089232	0.266512	0.894249	3.920630	16.430500	76.203300
	stl approx	0.014454	0.036857	0.125083	0.602819	3.436030	14.647700	70.024500
	xsimd	0.032143	0.057760	0.128522	0.339466	1.028440	3.911710	21.752500
lstm	eigen	0.012299	0.015120	0.029608	0.083030	0.354849	1.592930	6.128650
	eigen approx	**0.0119018**	**0.0150629**	**0.0217912**	**0.0703061**	**0.324553**	**1.51931**	**5.92699**
	stl	0.040875	0.136037	0.418509	1.376620	5.645630	22.749700	104.408000
	stl approx	0.019541	0.062217	0.207566	0.879157	4.721530	19.794000	93.801000
	xsimd	0.038011	0.072747	0.168950	0.444768	1.428400	5.420920	29.799600
relu	eigen	0.008118	0.009631	0.010882	0.017403	0.029641	0.045957	0.085621
	eigen approx	0.008442	0.010165	0.011556	0.018900	0.032514	0.045713	0.085234
	stl	0.010895	0.015798	0.029106	0.053231	0.109398	0.201204	0.383840
	stl approx	0.010161	0.014874	0.027287	0.050930	0.105940	0.196743	0.380681
	xsimd	**0.00477213**	**0.00550079**	**0.00653871**	**0.0117883**	**0.0255323**	**0.0255204**	**0.0437478**
sigmoid	eigen	0.013573	0.018722	0.025987	0.044098	0.084852	0.168307	0.343467
	eigen approx	0.011123	0.017783	**0.0118903**	**0.0210717**	**0.0418673**	**0.0717785**	**0.141943**
	stl	0.048567	0.099441	0.193933	0.383510	0.748444	1.464700	2.930460
	stl approx	0.011335	0.020355	0.036412	0.072870	0.147187	0.280243	0.562121
	xsimd	**0.0101243**	**0.0140866**	0.022705	0.043485	0.080985	0.156999	0.311811
tanh	eigen	0.011935	0.015326	0.019076	0.029089	0.053222	0.108238	0.201956
	eigen approx	0.012179	0.018851	0.014096	0.022622	0.040776	0.076923	0.153417
	stl	0.036809	0.071125	0.135642	0.263596	0.513995	1.016670	2.031620
	stl approx	**0.00738788**	**0.0136338**	**0.00729296**	**0.0142533**	**0.0268825**	**0.052107**	**0.103485**
	xsimd	0.017125	0.025050	0.040837	0.095256	0.142567	0.281613	0.536932

Table 2: Wall time results in seconds for approximate functions versus exact. The rows with "approx" use Schraudolph-NG for activation. The conv1d results additionally use loop perforation. Fastest times are in bold.

5.3.2 Model Build. We attempted to build our own model for testing purposes, and our results were mixed. We were able to train a model using two dense layers of 2^8 nodes each followed by a dense layer with a single output. We trained this model with a target/gold transform of $tanh$, which, when applied to an audio signal produces a simple distortion effect through amplification and cutoff.

However, our many attempts to build an LSTM-based model from scratch or use available code (as noted above) failed. Our results were either a silent output of zeroes or near zeroes, or a signal where the guitar sounds was evident but overwhelmed by noise. Detail about our model training challenges is in the Appendix B.

5.3.3 Model Selection and Audio Results. We tested audio inference on our dense model and two models we located online.[13] [14] The models and basic error statistics are listed in table 3. The audio input and output is available in our GitHub repository.[15]

We ran a sample guitar audio file through three models with both the exact RTNeural library and our approximated library using Schraudolph-NG. One author listened to both versions of the audio inference and did not detect any difference or degradation due to approximation.

We also did quantitative measurement of the signal difference by looking at the maximum error and error-to-signal ratio (ESR). ESR

[13] https://github.com/IHorvalds/MLTerror15
[14] https://github.com/AidaDSP/aidadsp-lv2/tree/main/models
[15] https://github.com/dsagman/Project521/tree/main/results-approx

Name	Params	Layers	Max Error	ESR
ds/tanh_3_256_tf	66561	dense-256, dense-256, dense-1	0.032593	0.001031
MLTerror15/0.5-0.5-0.5-model-gru-6	12929	gru-64, dense-1	0.361786	0.000925
MLTerror15/0.85-0.5-0.85-model-gru-6	12929	gru-64, dense-1	0.667633	0.003402
MLTerror15/0.85-0.85-0.85-model-gru-6	12929	gru-64, dense-1	0.565399	0.001777
MLTerror15/0.5-0.5-0.5-model-lstm-1	4385	lstm-32, dense-1	0.819916	0.019989
MLTerror15/0.5-0.5-0.5-model-gru-4	3489	gru-32, dense-1	0.359421	0.001815
MLTerror15/0.5-0.85-0.85-model-gru-5	3393	gru-32, dense-1	0.501678	0.001901
aidadsp/tw40_blues_deluxe_deerinkstudios	685	lstm-12, dense-1	0.224091	0.015601
aidadsp/tw40_blues_solo_deerinkstudios	685	lstm-12, dense-1	0.274261	0.022832
aidadsp/tw40_british_lead_deerinkstudios	685	lstm-12, dense-1	0.699188	0.024405
aidadsp/tw40_british_rhythm_deerinkstudios	685	lstm-12, dense-1	0.381012	0.017746
aidadsp/tw40_california_clean_deerinkstudios	685	lstm-12, dense-1	0.159729	0.008632
aidadsp/tw40_california_crunch_deerinkstudios	685	lstm-12, dense-1	0.300690	0.055393

Table 3: Audio models used for evaluation of approximation. ds is our model. MLTerror and aidadsp were located online and URLs are cited in the text. Params are the number of weights and layers includes the layer type and the number of nodes. Max error is the maximum signal error, which can be at most 2. ESR is error-to-signal ratio, see text. ESR best case is 0 and has no upper bound.

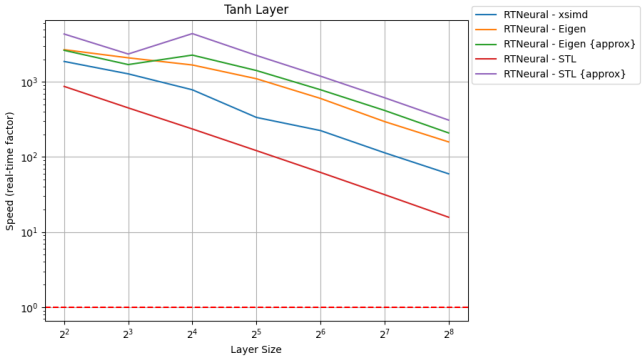


Figure 8: *tanh* results. Purple and green lines are with approximated *tanh* using Schraudolph-NG. Red dashed line is real-time cutoff. Higher is better.

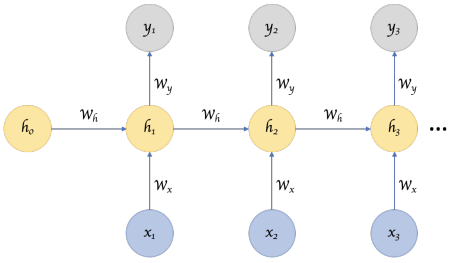


Figure 9: Recurrent Neural Network (RNN) showing hidden state, h_i being passed forward with each input value, x_i.[12]

measures our prediction versus the target, "gold" output rather than just the difference between the two signals as in mean-squared error (MSE). See equations 4 and 5. ESR is consistently below 0.05 and often in the 0.001 range. While not a perfect predictor of perceptual similarity, this low ESR is a good indicator.

$$\text{MSE} = \frac{\sum^{n}(y - \hat{y})^2}{n} \tag{4}$$

$$\text{ESR} = \frac{\sum^{n} |y - \hat{y}|^2}{\sum^{n} y^2} \tag{5}$$

Our testing was limited, and we do propose deeper testing for future work in section 7.7. However, given that the dense model has 2^8 nodes and 2^9 *tanh* function calls, we are confident that this approximation approach results in sufficient quality audio for the following reasons:

(1) Error should scale with the number of approximated function calls.
(2) Schrauldolph-NG has a maximum error of 1e-6.
(3) 2^8 nodes is at, or above, the upper end of what would be possible in real-time.

6 RELATED WORK

There has been much work on speeding up NN inference using a variety of techniques, including approximations such as quantization and pruning such as in [9] and [10]. While a full survey is beyond the scope of this paper, none of these papers focus on the specific task of real time audio processing.

7 FUTURE WORK

Beyond the approaches we implemented, we considered several others that time did not permit us to fully examine. Following are some examples of potential areas for future consideration.

7.1 Activation Functions

Chiluveru et al. provides another method for developing piece-wise linear approximations for activation functions.[2]

7.2 Matrix Multiply

All of the below alternate methods for matrix multiple are ones to evaluate.

- We selected column/row pairs based on the joint Euclidian norm of the two matricies. An alternative, with reduced probability of correctness, is to use just the norm of A or B.[6] Even more methods of probability calculation are in another paper by Drineas, et. al.[5]
- As noted, we considered but did not use the JL lemma approach. For much larger matrices, it could be worthwhile to revisit this.
- The MADDNESS algorithm uses no multiply-add instructions and instead has a non-linear pre-processing step of matrix A to turn matrix B into a lookup table.[1]
- Manne and Pal have a method for square matrices that reduces to a series of linear equations solved with gradient descent.[13]

7.3 Quantization

7.3.1 bfloat. As noted earlier, K-TanH relies on bfloat, aka "brain float," which reduces 32-bit floats to 16 bits. It shows good performance versus full precision and other floating point variants such as posits, and has steadily been adopted by more and more hardware platforms.[14] Originally developed by Google for their Tensor Processing Units (TPUs) in 2017.[16] It is currently available on latest generation Intel, Apple M2, and AMD CPUs, and NVIDIA GPUs. There is also compiler support in LLVM as of as of C++23.[17]

However, we did not pursue quantizing with bfloat at this point as our test bench machine does not have bfloat support, and the only potential device was one author's son's machine with a 4090 RTX. Despite the author's purchasing of the machine, it was determined "off-limits" due to "homework." A full recompilation of RTNeural with bfloat or other limited precision hardware support is considered future work.

7.3.2 Schraudolph-NG. The *i* component in Schraudolph can be viewed as bfloat16 when the algorithm is translated to operate on 32-bit floats. Additional performance improvements can likely be found by storing and evaluating inputs as bfloat16. Using newer hardware with bfloat16 vector instructions, our approach with Schraudolph-NG to improve accuracy while maintaining performance with SIMD instructions can be further optimized.

7.3.3 Other Quantization. Further quantization to 8-bit or possibly lower could be considered. We are doubtful this would be useful given the bit depth needed for audio being at least 16-bit.[18] However, it may be possible to quantize the portions of the network that are not directly connected to the output.

[16] https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/
[17] https://en.cppreference.com/w/cpp/types/floating-point
[18] https://soundgearlab.com/learn/audio-sample-rate-bit-depth-bit-rate/

7.4 Pruning

7.4.1 Unstructured. For our purposes, matrix multiply is essentially random unstructured pruning and allowed us to test out the general concept. Based on our results, especially on how slow large dense layers are, we think larger networks may benefit from pruning, but leave that to future work.

7.4.2 Structured. Pruning has been shown to be effective in reducing the size of NNs and their processing time, especially for dense layers.[9] And there is research that iterative magnitude pruning can be effective when modeling guitar distortion effects.[19]. However, care must be taken depending on the layer type that sparse matrices do not increase computation time.[25]

7.5 Loop Perforation

In addition to *conv1d* there are loops in *batchnorm*. An approach to consider is skipping loops iterations here. We suspect this will not be very useful for the same problem domain reasons as other methods that rely on large amounts of data to be effective such as matrix multiply.

7.6 Other Models

While not directly related to approximation, we would consider investigation additional audio modeling approaches that are not supported currently by RTNeural. This could be as simple as an *add* layer for skip-layer incorporation of the input before the final output step, or as complex as developing a *transformer* architecture.

Interestingly, we did not find transformers used in audio modeling literature. This is likely due to challenges doing signal processing inference with a transformer, and the limited research done in this space to date. A recent survey shows only 4% of transformer applications in signal processing.[11]

7.7 Audio Accuracy

We did limited, subjective testing of the audio output of our approximated models. A more fulsome audio analysis would include:

- Models trained on different effects and amplifiers, including those with more subtle effects rather than gain/distortion.
- Subjective analysis by multiple, expert listeners.
- Detailed frequency domain and spectral analysis.

8 CONCLUSION

We have shown that approximation of activation functions can speed up real-time inference on an audio neural network with acceptable audio noise.

Our *tanh* and *sigmoid* activation functions based on Schraudolph-NG are much faster, with *tanh* performing over 4x faster than the *libm* baseline. For networks with many of these, it would be appropriate to use this approach. If exact accuracy is required, our Padé approximation is 2x faster and nearly lossless.

We have also shown that other common approximation approaches are less useful for this application. Specifically, approximating matrix multiply and convolutional loop perforation are applicable for image processing where there is a high density of data in two or three dimensions. For audio, only a small amount of

data is taken at each forward step: the maximum is the size of an initial layer e.g., convolution or LSTM.

Given the real-time constraint that limits the size of this layer to $< 2^8$, approximating matrix multiply or convolution both introduce either too much setup time or too much error, leading to audio noise.

ACKNOWLEDGMENTS

To our spouses for accepting the statement, "I have to play loud guitar. It's for our project."

REFERENCES

[1] Davis Blalock and John Guttag. 2021. Multiplying Matrices Without Multiplying. arXiv:2106.10860 [cs.LG]

[2] Samba Raju Chiluveru, Manoj Tripathy, and Bibhudutta. 2021. Non-linear activation function approximation using a REMEZ algorithm. *IET Circuits, Devices & Systems* 15, 7 (2021), 630–640. https://doi.org/10.1049/cds2.12058 arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/cds2.12058

[3] Jatin Chowdhury. 2021. RTNeural: Fast Neural Inferencing for Real-Time Systems. arXiv:2106.03037 [eess.AS]

[4] Jatin Chowdhury. 2024. GitHub - jatinchowdhury18/RTNeural: Real-time neural network inferencing — github.com. https://github.com/jatinchowdhury18/RTNeural. [Accessed 15-03-2024].

[5] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. 2006. Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication. *SIAM J. Comput.* 36, 1 (2006), 150. https://doi.org/10.1137/S0097539704442684 arXiv:https://doi.org/10.1137/S0097539704442684

[6] Petros Drineas and Michael W. Mahoney. 2017. Lectures on Randomized Numerical Linear Algebra. , 16-18 pages. arXiv:1712.08880 [cs.DS]

[7] Michael Figurnov, Aijan Ibraimova, Dmitry Vetrov, and Pushmeet Kohli. 2016. PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions. arXiv:1504.08362 [cs.CV]

[8] Benjamin Graham. 2014. Fractional Max-Pooling. *CoRR* abs/1412.6071 (2014). arXiv:1412.6071 http://arxiv.org/abs/1412.6071

[9] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:1510.00149 [cs.CV]

[10] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *CoRR* abs/2102.00554 (2021). arXiv:2102.00554 https://arxiv.org/abs/2102.00554

[11] Saidul Islam, Hanae Elmekki, Ahmed Elsebai, Jamal Bentahar, Nagat Drawel, Gaith Rjoub, and Witold Pedrycz. 2024. A comprehensive survey on applications of transformers for deep learning tasks. *Expert Systems with Applications* 241 (2024), 122666. https://doi.org/10.1016/j.eswa.2023.122666

[12] Abhisek Kundu, Alex Heinecke, Dhiraj Kalamkar, Sudarshan Srinivasan, Eric C. Qin, Naveen K. Mellempudi, Dipankar Das, Kunal Banerjee, Bharat Kaul, and Pradeep Dubey. 2020. K-TanH: Efficient TanH For Deep Learning. arXiv:1909.07729 [cs.LG]

[13] Shiva Manne and Manjish Pal. 2014. Fast Approximate Matrix Multiplication by Solving Linear Systems. arXiv:1408.4230 [cs.DS]

[14] Aleksandr Yu. Romanov, Alexander L. Stempkovsky, Ilia V. Lariushkin, Georgy E. Novoselov, Roman A. Solovyev, Vladimir A. Starykh, Irina I. Romanova, Dmitry V. Telpukhov, and Ilya A. Mkrtchan. 2021. Analysis of Posit and Bfloat Arithmetic of Real Numbers for Machine Learning. *IEEE Access* 9 (2021), 82318–82324. https://api.semanticscholar.org/CorpusID:235455363

[15] Nicol N. Schraudolph. 1999. A fast, compact approximation of the exponential function. *Neural Comput.* 11, 4 (may 1999), 853–862. https://doi.org/10.1162/089976699300016467

[16] T. E. Simos and Ch. Tsitouras. 2021. Efficiently inaccurate approximation of hyperbolic tangent used as transfer function in artificial neural networks. *Neural Comput. Appl.* 33, 16 (aug 2021), 10227–10233. https://doi.org/10.1007/s00521-021-05787-0

[17] Christian J. Steinmetz and Joshua D. Reiss. 2020. auraloss: Audio focused loss functions in PyTorch. In *Digital Music Research Network One-day Workshop (DMRN+15).*

[18] Christian J. Steinmetz and Joshua D. Reiss. 2022. Efficient neural networks for real-time modeling of analog dynamic range compression. arXiv:2102.06200 [eess.AS]

[19] David Südholt, Alec Wright, Cumhur Erkut, and Vesa Välimäki. 2023. Pruning Deep Neural Network Models of Guitar Distortion Effects. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 31 (2023), 256–264. https://doi.org/10.1109/TASLP.2022.3223257

[20] Nicholas Gerard Timmons and Andrew Rice. 2020. Approximating Activation Functions. arXiv:2001.06370 [cs.LG]

[21] Michael Tyson. 2021. Four common mistakes in audio development. https://atastypixel.com/four-common-mistakes-in-audio-development/

[22] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. 2016. WaveNet: A Generative Model for Raw Audio. *CoRR* abs/1609.03499 (2016). arXiv:1609.03499 http://arxiv.org/abs/1609.03499

[23] Alec Wright, Eero-Pekka Damskägg, Lauri Juvela, and Vesa Välimäki. 2020. Real-Time Guitar Amplifier Emulation with Deep Learning. *Applied Sciences* 10, 3 (2020). https://doi.org/10.3390/app10030766

[24] Alec Wright and Vesa Välimäki. 2019. Perceptual Loss Function for Neural Modelling of Audio Systems. arXiv:1911.08922 [eess.AS]

[25] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 548–560. https://doi.org/10.1145/3079856.3080215

A ALL RTNEURAL BENCHMARKS

We include all layers and activation functions tested with RTNeural random benchmarks in figure 10. This is the same data from table 2 in graphical form.

B MODEL TRAINING CHALLENGES

Despite the publicly available code, we ran into several challenges in creating our own model and using it with RTNeural. We spent upwards of 100 hours running and re-running the training with different parameters and debugging attempts (often overnight), but were confronted with confounding outcomes where the model weights tended towards zero.[19]

A brief summary of major issues, by category, follows.

B.1 Data and Data Preparation Challenges

(1) Different input data had surprisingly different training results. Sample data from internet repositories for distortion-based effects such as a Tube Screamer or high-gain Blackstar amplifier worked well. However, sample data created by a professional guitarist local to the authors on a Vox 60 amplifier did not process well. This data was less distorted and more "sparkly."

(2) The Vox 60 data in an LSTM model consistently drove weights towards zero. This occurred in internet sourced training models and also in author created models. We were unsure if this was a data or model problem.

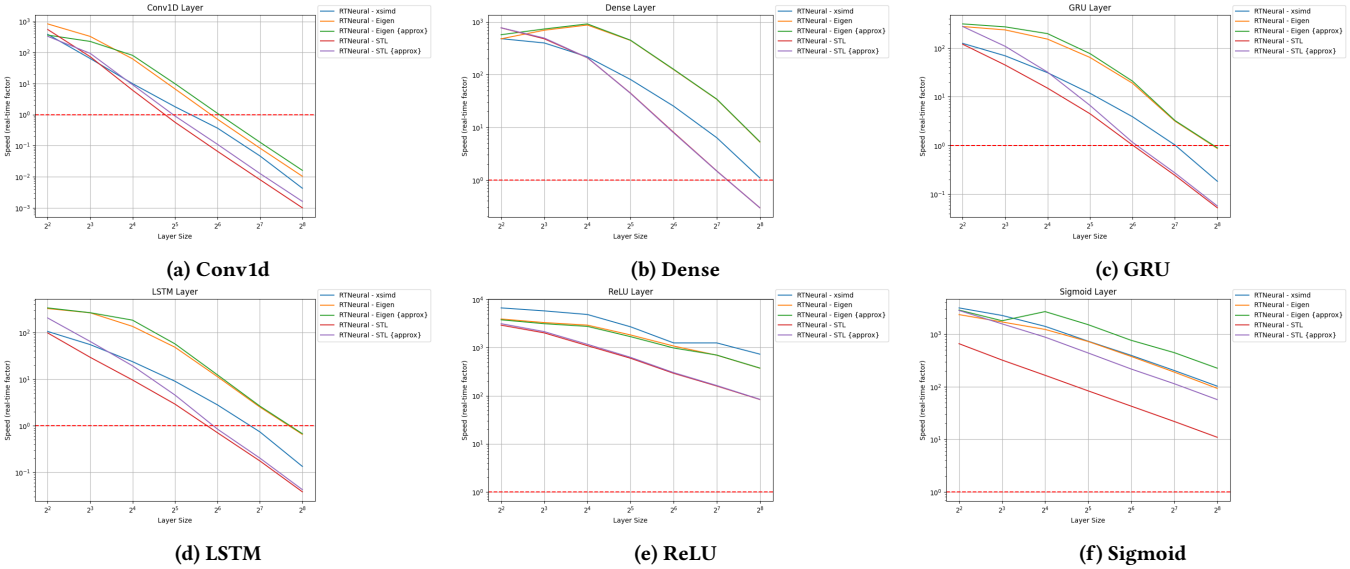
(3) Deep examination of the Vox 60 data revealed differences in wav file format encoding, bit-depth, and sample rate. However, none of these variations applied to the distortion-based effects seemed to cause the same challenges.

B.2 Model and Training Challenges

(1) We tried to use the straightforward LSTM-Dense model as presented in the references.[23] This worked for simple transformations such as applying $tanh(x)$ as a simple distortion effect.[20] But failed for the Vox 60.

[19] All model training code, the good, the bad, and the ugly is at https://github.com/dsagman/Project521.

[20] https://csoundjournal.com/ezine/winter1999/processing/index.html



(a) Conv1d (b) Dense (c) GRU

(d) LSTM (e) ReLU (f) Sigmoid

Figure 10: All RTNeural Benchmarks. Green and purple lines are using approximated activation functions.

(2) Using more complex models with *conv1d* and *dense* layers, in addition to LSTM dramatically increased training time, and made experimentation therefore difficult to iterate.
(3) We wanted to try the recurrent feed-foward approach of adding the input tensor back in just before the output to help drive the correct values.[23] But RTNeural does not have an *add* layer. This could be a possibility for future work.
(4) For the loss function, we used the standard mean squared error (MSE). However, for audio, other measures are potentially better. A common one is error-to-signal-ratio (ESR) between the target, y, and the prediction $\hat{y}$.[17][24] (See equations 4 and 5.) We were able to log the value of ESR during training, but were unsuccessful using it as the primary loss measure.
(5) We were unable to get get the `auraloss` Python library to work for these other aural measures such as spectral based ones.
(6) We tried but did not find value in using a DC offset as proposed by Wright given the nature of our signals coming from a single source (un-amplified guitar) through a single analog amplification.
(7) We tried but did not find value in normalizing or filtering the input or target signal.[21]
(8) We attempted but did not find value in trying to initialize the network to initial values close to the target with a custom initialization function.

B.3 Hyperparameter and Data Preparation Challenges

(1) The models were highly sensitive to learning rate. High learning rates (.01) would over-correct and increase errors. Attempts were made to use reduction of learning rate via keras `ReduceLROnPlateau`. However, lowering the learning rate too far greatly increased training time. For example, in one attempt learning rate moving automatically from .001 to .0001 increased training time per epoch from 30 minutes to 9 hours. This forced a restart after 6 hours of incomplete training.
(2) Selecting other hyperparameters for batch size, epochs, training data size and so on were subject to a great deal of trial and error.
(3) Correctly loading data into a dataset offered many additional options, such as window size, shuffle size, whether to shuffle or not, and so on.
(4) The data loading presented further challenges in determining how the available training code prepared the data versus how RTNeural processes inference with a "ring buffer". We tried multiple variations of sliding window training and inference, but did not locate an approach that worked as desired.

B.4 Software and Hardware Challenges

(1) RTNeural uses a custom JSON format. In trying to use RTNeural's `save_model` function, we discovered a bug due to internal changes in Tensorflow. We submitted an accepted PR to fix the issue.[22]
(2) Initial models were built with PyTorch, but instrumenting those models in RTNeural was less straightforward than using TensorFlow. This lead to rewriting the models with TensorFlow and discovering some unexpected differences.
(3) PyTorch was able to make use of the NVIDIA 4090 GPU in Windows on one author's son's computer, greatly speeding up training. However, TensorFlow's current version requires Windows Subsystem for Linux (WSL) and compilation of

[21] https://github.com/arvidfalch/blackboxRNNmodeling/blob/main/main_training.ipynb

[22] https://github.com/jatinchowdhury18/RTNeural/issues/133

required libraries. However, all attempts to get the GPU working in WSL failed.
(4) The mentioned 4090 GPU had limited availability to the author's son's Spring break week.
(5) Running models overnight on either Windows or Mac would often stall or reboot, despite setting energy savings and auto-update off.
(6) Understanding the output from TensorBoard logging was challenging and did not provide much value.
(7) TensorBoard logging overloaded the available disk storage on one overnight run.

C SELECTED ALGORITHM CODE

Selected examples of algorithms. This appendix contains:

(1) Schraudolph deep dive in Rust, C++ and Assembly
(2) Random Matrix Multiply in Python
(3) Loop Perforated Conv1d in Python

Additions to RTNeural were either written in C++ or Rust. Python code is provided for reference algorithm explanation. Implementation for these in RTNeural required adapting to the interfaces used in that library. RTNeural uses inner product, pointers, and does not have the equivalent of numpy. The Python is more straighforward for exposition puposes.

C.1 Schraudolph Deep Dive

The original algorithm from Schraudolph's paper is as follows:[15]

```
#define EXP_A (1048576 / M_LN2)
#define EXP_C 60801
inline double exponential(double y) {
    union {
        double d;
#ifdef LITTLE_ENDIAN
        struct { int j, i; } n;
#elseif
        struct { int i, j; } n;
#endif
    } _eco;
    _eco.n.i = (int)(EXP_A * (y)) + (1072693248 - EXP_C);
    _eco.n.j = 0;
    return _eco.d;
}
```

We first implemented in Rust with constants derived explicitly in code and we converted to f32 from f64 (i.e. double).

```
pub fn expf(y: f32) -> f32 {
    const BIAS: i16 = f32::MAX_EXP as i16 - 1;
    const MANT_BITS: i16 = f32::MANTISSA_DIGITS as i16 - 1;
    const OFFSET_BITS: i16 = i16::BITS as i16;

    const X: i16 = 1 << (MANT_BITS - OFFSET_BITS);
    const A: f32 = X as f32 / LN_2;
    const B: i16 = X * BIAS;
    const C: i16 = 8; // tuning parameter
    const D: i16 = B - C;

    unsafe {
        let i = (A * y).to_int_unchecked::<i16>() + D;

        core::mem::transmute(
            #[cfg(target_endian = "little")] [0, i],
            #[cfg(target_endian = "big")]    [i, 0],
        )
    }
}
```

The corresponding ARM assembly code is below.[23] The section in red corresponds to a safety check that Rust inserts by default when converting from `f32` to `i16`, which noticeably slows down the implementation. Using `to_int_unchecked` circumvents the check, thus removing the red code section.

```
_expf:
    mov     w8, #43579
    movk    w8, #17208, lsl #16
    fmov    s1, w8
    fmul    s0, s0, s1
    fcvtzs  w8, s0
    mov     w9, #32767
    cmp     w8, w9
    csel    w8, w8, w9, lt
    cmn     w8, #8, lsl #12
    mov     w9, #-32768
    csel    w8, w8, w9, gt
    mov     w9, #1064828928
    add     w8, w9, w8, lsl #16
    fmov    s0, w8
    ret
```

We used this information and the improvements proposed by Schraudolph on Stack Exchange to write C++ and Rust implementations of Schraudolph-NG.[24]

Notes:

- Little endianness assumed
- SIMD assumed and used to take advantage of symmetry in numerator and denominator

C.1.1 *C++ Schraudolph-NG.*

```
float expf(float x) {
  const auto u = (uint32_t)(6051101.5f * x + 1065353216.f);
  const auto v = (uint32_t)(-6051101.5f * x + 1065353216.f);
  return std::bit_cast<float>(u) / std::bit_cast<float>(v);
}
```

C.1.2 *Rust Schraudolph-NG.*

```
#[cfg(target_endian = "little")]
pub fn expf(x: f32) -> f32 {
    const BIAS: u32 = f32::MAX_EXP as u32 - 1;
    const MANTISSA_BITS: u32 = f32::MANTISSA_DIGITS - 1;

    const A: f32 = (1 << MANTISSA_BITS) as f32 / LN_2;
    const B: f32 = (BIAS << MANTISSA_BITS) as f32;

    #[cfg(not(target_feature = "neon"))] {
        f32::from_bits((A / 2. * x + B) as u32)
          / f32::from_bits((-A / 2. * x + B) as u32)
```

[23] Assembly examined using Compiler Explorer at godbolt.org.
[24] https://stackoverflow.com/questions/47025373/fastest-implementation-of-the-natural-exponential-function-using-sse

```
    }

    #[cfg(target_feature = "neon")] unsafe {
        let a = vcreate_f32(transmute([A / 2., -A / 2.]));
        let x = vdup_n_f32(x);
        let b = vdup_n_f32(B);

        let y = vfma_f32(b, a, x);
        let y = vcvt_u32_f32(y);
        let y = vreinterpret_f32_u32(y);

        vget_lane_f32::<0>(y) / vget_lane_f32::<1>(y)
    }
}
```

C.1.3 Assembly Schraudolph-NG.

```
lCPI0_0:
        .long   0x4ab8aa3b //=  6051101.5f
        .long   0xcab8aa3b //= -6051101.5f
_expf:
        adrp        x8, lCPI0_0@PAGE
        ldr         d1, [x8, lCPI0_0@PAGEOFF]
        mov         w8, #1316880384 //= 1065353216.f
        dup.2s      v2, w8
        fmla.2s     v2, v1, v0[0]
        fcvtzu.2s   v0, v2
        dup.2s      v1, v0[1]
        fdiv.2s     v0, v0, v1
        ret
```

C.2 Random Matrix Multiply

Python implementation of random matrix multiply. This code is not optimized.

```
import numpy as np

def approx_matmul(A, B, factor=1):
  m, n = A.shape
  _, p = B.shape
  A_col_norm = np.sqrt(np.sum(A ** 2, axis=0)
  B_row_norm = np.sqrt(np.sum(B ** 2, axis=1)
  sum_norm = np.sum(A_col_norm * B_row_norm)
  prob = np.array([A_col_norm[k] * B_row_norm[k] \
    / sum_norm for k in range(n)])
  # np.testing.assert_almost_equal(np.sum(prob), 1, 5)
  # probability should sum to approx. 1
  c = int(n * factor)
  # arg sort is ascending, so first reverse then slice
  i = np.flip(np.argsort(prob))[:c]

  prob_i = prob[i]
  A_i = A[:,i]
  B_i = B[i,:]

  result = np.zeros((m, p))
    for t in range(c):
      for i in range(m):
        for j in range(p):
        result[i, j] += (1 / prob_i[t]) * A_i[i, t] * B_i[t, j]
     return (1 / c) * result
```

C.3 Loop Perforated Conv1d

Python implementation of loop perforated conv1d. Code is not optimized.

```
import numpy as np

def pseudo_mask(length, alpha=1.5, u_range=(0, 1)):
    """
        Generate a pseudo-random mask based on
        pseudo-random integer sequence
        a_i = ceiling(α(i + u)), α ∈ (1, 2), with some u ∈ (0, 1)
        length: length of the sequence
        alpha: alpha value
        u_range: range of u
        returns mask of items to keep as 0 or 1 at each index
        from paper: https://arxiv.org/abs/1412.6071
    """
    u = np.random.uniform(u_range[0],u_range[1])  # U value
    sequence = np.zeros(length, dtype=int)
    for i in range(length):
        a = np.ceil(alpha * (i + u)).astype(int)
        if a >= length:
            break
        sequence[a] = 1
    return np.array(sequence)

def perf_conv1d(U, K, prob=1.0, padding=0, stride=1,
                dilation=1, type='uniform', alpha=1.5,
                u_range=(0, 1)):
    """
        Convolution with perforation from paper
        https://arxiv.org/abs/1504.08362
        U: input signal
        K: kernel
        prob: probability of keeping the element in the mask
        padding: padding size
        stride: stride size
        dilation: dilation size
        type: uniform or pseudo
        alpha: alpha value for pseudo mask
        u_range: range of u for pseudo mask
    """
    U = np.array(U)
    K = np.array(K)
    if dilation > 1:
        K = np.insert(K[:-1],
             np.repeat(np.arange(len(K)-1), dilation-1), 0)
    U = np.pad(U, (padding,padding), 'constant')
    d = len(K)
    n = len(U)
    # Mask M
    M = np.array([U[i:i+d] for i in range(0, n-d+1, stride)])
    if type=='uniform':
        # Indicies in M to keep
        I = np.random.choice([0, 1],
            size=M.shape[0], p=[1-prob, prob])
    else:
        I = pseudo_mask(M.shape[0], alpha, u_range)
    # perforate M
    M = M*I[:, np.newaxis]
    # Convolution with perforation
    V_hat = np.array([np.sum(m*K) for m in M])
```

```
    # replace 0s with nearest non-zero element
    non_zero_i = np.where(V_hat != 0)[0]
    for i in range(len(V_hat)):
      nearest_i = non_zero_i[np.argmin(np.abs(non_zero_i - i))]
        V_hat[i] = V_hat[nearest_i]
    return np.array(V_hat)

U = np.arange(1,21)
K = np.array([1, 1, 1])
print("input:", U)
print("kernel:", K)
V_hat = perf_conv1d(U, K, prob=0.2,
    padding=padding, stride=stride, dilation=dilation,
    type='pseudo', alpha=1.5, u_range=(0, 1))
print('Perforated Conv1d output:', V_hat)
```