INFERENCE-TIME STRUCTURED PRUNING FOR REAL-TIME NEURAL NETWORK AUDIO EFFECTS

Christopher Johann Clarke

The University of Electro-Communications
chris.clarke@uec.ac.jp

Jatin Chowdhury

Chowdhury DSP
jatin@chowdsp.com

ABSTRACT

Structured pruning is a technique for reducing the computational load and memory footprint of neural networks by removing structured subsets of parameters according to a predefined schedule or ranking criterion. This paper investigates the application of structured pruning to real-time neural network audio effects, focusing on both feedforward networks and recurrent architectures. We evaluate multiple pruning strategies at inference time, without retraining, and analyze their effects on model performance. To quantify the trade-off between parameter count and audio fidelity, we construct a theoretical model of the approximation error as a function of network architecture and pruning level. The resulting bounds establish a principled relationship between pruning-induced sparsity and functional error, enabling informed deployment of neural audio effects in constrained real-time environments.

1. INTRODUCTION

In the development of digital audio effects, developer's will often make trade-offs for the sake of computational efficiency. A common example is the use of a lookup table or other approximation for computing a mathematical expression which would be computationally expensive to perform in real-time (see e.g. [1]).

Machine learning, and particularly black-box neural networks are becoming increasingly common-place in the development of digital audio effects [2, 3, 4]. When performing neural network inference for an effect, the developer may make trade-offs related to the sample rate at which inference is performed [5], as well as mathematical approximations as described above. However, the most obvious trade-off between computational performance and accuracy in the context of a neural network is the number of "parameters" or "weights" contained in the network. Indeed, many prior publications describing neural architectures for digital audio effects demonstrate that increasing the number of parameters in a neural network increases the networks accuracy, while also increasing its computational cost, e.g. [4]. Minimizing the network parameter count is the most significant performance/accuracy trade-off when developing a neural network-based system, since the efficacy of neural networks is often due to their over-parameterizing the problem they are trained to solve [6].

However, a neural network's parameter count is typically chosen at training time, making it challenging for a developer to choose the right trade-off between accuracy and performance, especially considering that training a large neural network can take hours or days. One method for reducing a neural network's parameter count post-training is "pruning" [7]. Pruning algorithms allow parameters to be removed from a neural network with (ideally) minimal change to the neural network's accuracy, thereby reducing the amount of computation required to perform inference.

The remainder of this paper is organized as follows: §2 reviews relevant research on neural network pruning and related techniques; §3 outlines the proposed structured pruning methodology; §4 presents the results of pruning experiments; and §5 concludes the paper. To aid clarity, the experimental results, theoretical error models, and corresponding analyses are presented per network architecture, allowing each model class to be discussed independently and comprehensively.

2. PRIOR WORK

Pruning neural networks was conceived as a way to strategically reduce a network's parameter count by removing "unimportant" weights from the network [8]. While this paper will primarily focus on the use of pruning to improve speed of inference, prior work also discusses the effectiveness of pruning in reducing over-fitting [9], allowing networks to be trained on smaller datasets, and reducing a network's memory footprint, along with improving the network's performance at both training- and inference-time. Pruning is sometimes used as part of an iterative training process, in which the neural network is recursively trained, pruned, and re-trained until some accuracy or performance goal has been met [10].

2.1. Pruning for Sparsity

Most early pruning literature discusses the pruning of individual weights from the network. In practice, this usually results in "zeroing" one of the values in one of the network's weights matrices, so as weights are pruned from networks the weights matrices become more sparse [11]. Achieving matrix sparsity allows the network's inference computations to be optimized using sparse matrix-matrix and matrix-vector multiplication algorithms [12]. However, sparse matrix multiplies often require numerous indirect memory accesses, thereby limiting the potential performance gains, unless inference is being performed on hardware that has dedicated hardware acceleration for sparse matrix operations [13]. This limitation is especially significant for neural networks used in real-time audio systems, where crucial inferencing optimizations rely on hardware-level performance primitives that require sequential memory access (e.g. SIMD operations) [14]. Still, pruning individual weights is a popular technique, and machine learning training libraries often provide tutorials for pruning neural networks with the goal of making the network more sparse [15, 16].

2.2. Structured Pruning in Convolutional Networks

Rather than pruning individual weights, recent research has focused on pruning groups of weights together, referred to as "structured pruning" [17]. This approach has found success in convolutional neural networks used for image classification (e.g. VGG-16 [18]), where entire convolution "filters" can be pruned from the network architecture. Prior research shows significant performance improvements with minimal changes in accuracy when using a pruning-training loop to improve image classification networks [17, 19, 10]. It is worth noting that prior research primarily discusses the application of structured pruning to dense and convolutional networks, and there has been relatively little discussion of structured pruning for recurrent networks.

2.3. Pruning Alternatives

Pruning is not the only method for reducing a neural network's computational cost. Other popular methods include distillation [20] and model quantization [21]. While an in-depth investigation of these methods is outside the scope of this work, a brief discussion is provided in the following sections.

2.3.1. Distillation

Distillation involves training a smaller neural network using the output of a larger network, and has been a successful approach for improving model performance in several domains including speech recognition, speech synthesis, and large language models [22, 23, 24]. Distillation can be especially effective for training smaller "specialist" models from a larger "generalist" model. Note that the distillation process must occur at training-time.

2.3.2. Quantization

"Quantized" neural network inference using reduced-precision (i.e. "quantized") weights. Networks may be trained with the intention of being used for quantized inference (known as "quantization-aware training"), however, any neural network may have its weights quantized regardless of how it was trained. Quantization is a common practice for deploying neural networks on embedded devices that are limited in both memory availability and compute.

As an example, a network trained with 32-bit floating-point weights may have its weights quantized to 8-bit integers. This type of quantization reduces the size of the neural network's weights and allows the model inference to make better use of hardware-level parallelization.[1] More aggressive quantization schemes include "binarized neural networks", which quantize all of the network's weights to either $+1$ or -1 [25].

While quantization can be used in real-time audio processing, its use is somewhat limited for neural networks that are designed to process time-domain audio data, since the quantization will result in a corresponding loss of quality.

[1]Modern Intel CPUs provide 128-bit XMM registers, which can be used to perform operations on 4 single-precision floating point values at a time. Quantizing the network weights to 8-bit integers allows the same 128-bit register to be used to perform operations on 16 quantized values, allowing for a potential 4x speed-up.

3. METHODOLOGY

We begin by formally introducing the structured pruning techniques and ranking strategies that form the basis of the experimental analysis in this paper.

3.1. Notation

Let $\mathcal{D} = \{(x, y)\}$ denote a dataset consisting of observed real-valued input-output pairs, where the inputs x and outputs y are recorded from an audio processing function. An approximator $\mathcal{A}$ is a trainable function that, upon training, produces an approximation f^* of the true underlying function f. Specifically, $\forall x \in \mathcal{D}$, the function f^* produces an output $\hat{y} = f^*(x)$ such that the discrepancy between $\hat{y}$ and the true output $y = f(x)$ is bounded by a given tolerance $\varepsilon > 0$, as quantified by a loss function $\mathcal{L}$; that is,

$$\mathcal{L}\big(f^*(x),\, f(x)\big) \le \varepsilon \quad , \quad \forall (x, f(x)) \in \mathcal{D}. \tag{1}$$

3.2. Structured Pruning Techniques

Pruning typically works by constructing a "ranking" of elements based on how significantly each element affects the accuracy of the neural network. These elements could be individual weights (in traditional pruning) or groups of weights such as a filter or matrix row/column (in structured pruning). Then the N lowest-ranking elements may be pruned from the network. In practice, elements are ranked based on some criterion, which is used as a proxy for that element's relative impact on the network's accuracy. The following sub-sections present several potential ranking criteria.

Formally, let a neural network contain a weight matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ within a layer defined by the mapping

$$y = \sigma(\mathbf{M}x + b), \tag{2}$$

where $x \in \mathbb{R}^n$ is the input to the layer, $b \in \mathbb{R}^m$ is a bias vector, $\sigma : \mathbb{R}^m \to \mathbb{R}^m$ is an elementwise nonlinear activation function, and $y \in \mathbb{R}^m$ is the output vector. In structured pruning, we define a collection of *prunable elements* $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$, where each s_i denotes a structured unit—typically a row or column of $\mathbf{M}$, corresponding to an output or input neuron, respectively.

Each element $s_i \in \mathcal{S}$ is assigned a scalar importance score via a ranking function $\rho : \mathcal{S} \to \mathbb{R}$, which quantifies the relative contribution of that element to the network's behavior. This ranking function is typically defined via a qualified objective (weight magnitude, activation statistics, loss sensitivity), rather than a noticeable impact on loss metrics, due to computational intractability.

Given a pruning budget $N \in \mathbb{N}$, the N lowest-ranked elements according to ρ are selected for removal. Let $\mathcal{S}_{\text{prune}} \subset \mathcal{S}$ be this subset, and $\mathcal{T}$ be one possible pruning candidate subset of size N:

$$\mathcal{S}_{\text{prune}} = \operatorname{argmin}_{\mathcal{T} \subset \mathcal{S},\, |\mathcal{T}|=N} \sum_{s \in \mathcal{T}} \rho(s). \tag{3}$$

Structured pruning modifies $\mathbf{M}$ by eliminating the corresponding rows and/or columns associated with $\mathcal{S}_{\text{prune}}$.

This has two immediate implications —Downstream effects: Pruning an output unit (row of $\mathbf{M}$) removes a corresponding entry in y, which may eliminate dependent units in subsequent layers. Upstream effects: Pruning an input unit (column of $\mathbf{M}$) implies that the corresponding feature in x is unused, potentially enabling upstream pruning of earlier layers.

In practice, this procedure is applied layer-by-layer, and the choice of ρ significantly affects the accuracy-compression trade-off. The next sections define several commonly used ranking functions and analyze their behavior across network architectures.

3.2.1. Minimum Weights Ranking

Minimum weights ranking assigns importance to each structured element (e.g., row or column) of the weight matrix $\mathbf{M}$ based solely on the magnitude of its constituent weights. A common approach is to compute the ℓ_1-norm for each row or column:

$$\rho_{\text{L1}}(s_i) = \sum_j |M_{ij}| \quad \text{or} \quad \sum_j |M_{ji}|, \tag{4}$$

depending on whether rows or columns are being pruned. This method, introduced in [19], is computationally efficient and does not require data or inference, making it suitable for rapid pruning during or after training. However, it assumes a direct correspondence between weight magnitude and contribution to function, which may not hold in neural networks with heavy over-parameterization or strong activation nonlinearity.

3.2.2. Mean Activations Ranking

Mean activations ranking evaluates the empirical contribution of each structured element by measuring its effect on the layer's output activations. Following [26], let $y = \sigma(\mathbf{M}x + b)$ be the layer output, and consider a validation set $\mathcal{D}_{\text{val}}$. For each row s_i of $\mathbf{M}$, define a modified network $\mathcal{A}_{\cap s_i}$ in which the corresponding weights are zeroed. Then compute the output statistics over $\mathcal{D}_{\text{val}}$:

$$\text{argmin } \rho_{\text{act}}(s_i) = \forall s_i \in (\sigma_{x \sim \mathcal{D}_{\text{val}}}[y^{\cap s_i}]) \tag{5}$$

Elements with minimal effect on the output distribution are considered redundant. This method is particularly effective in pruning units whose activations saturate under nonlinearities such as $\tanh$ or ReLU, thereby contributing little dynamic variation to downstream layers. While more accurate than magnitude-based ranking, it incurs additional cost due to inference over a validation set.

3.2.3. Minimization Ranking

Minimization-based ranking seeks to directly minimize a loss function $\mathcal{L}$ while pruning. Let $\mathcal{L}(\mathcal{A})$ denote the network's loss (e.g., MSE) on a dataset. The importance of element s_i is measured by the increase in loss upon its removal:

$$\rho_{\text{loss}}(s_i) = \mathcal{L}(\mathcal{A}_{\cap s_i}) - \mathcal{L}(\mathcal{A}). \tag{6}$$

This approach captures the true functional impact of a structural unit and is thus more principled, but requires an evaluation pass per candidate element. In practice, surrogate approximations of the loss change, such as second-order Taylor expansions or saliency-based heuristics, may be employed to reduce computational cost [10]. Importantly, the optimal cost function may not be the same as the training objective; it should instead reflect the sensitivity of the model to pruning.

3.3. Experimental Setup

An experiment was designed in order to examine the efficacy of different pruning methods for real-time audio processing neural networks. A dataset was constructed consisting of 10 seconds of guitar audio processed through a custom-built "fuzz" effect pedal, and recorded at 96 kHz sample rate. Three networks were trained on the dataset: a memory-less network made up of fully-connected "dense" layers with ReLU activations (**Dense**), a feed-forward network made up of convolutional layers with $\tanh$ activations (**Conv.**), and a recurrent neural network made up of a Long Short-Term Memory layer (**LSTM**). While other recurrent network architectures including Gated Recurrent Units (GRUs) and state-space models are also common in audio processing, the LSTM was chosen since it has more weights per hidden unit, thereby creating more potential pruning candidates. Each network was designed to have approximately the same number of parameters; the number of parameters was chosen arbitrarily for a realtime capable network. Each network was trained for 100 epochs, using the Adam optimizer and the mean-squared error (MSE) as the training loss function. Table 1 shows the specific design and training results for each network. 100 epochs was chosen since each network's training converged to a reasonable degree of accuracy, which allowed the effects of pruning to be measured.

Model	Dense	Conv.	LSTM
# Layers	8	4	1
Hidden Size	64	32	84
Parameters	29,313	30,081	28,981
Training Loss	0.0114	0.0109	0.0036

Table 1: *Training design and results for the neural networks used in the experiment. The number of layers were chosen to approximately match the number of parameters in each network type.*

Next each network went through several iterations of pruning and testing using validation data. In each iteration a certain number of elements were pruned from the network, until the total number of parameters remaining in the network was fewer than 13,000. At each iteration the validation error (MSE) was measured along with the "real-time factor" $F_{\text{RT}} = T_{\text{audio}}/T_{\text{proc}}$, where T_{audio} is the duration of the validation audio data in seconds, and T_{proc} is the time required for the neural network to process the audio data. This process was repeated with each neural network, once with each of the proposed ranking methods: Minimum Weights (MW), Mean Activations (MA), and Minimization (MIN). The Mean Activations and Minimization rankings used the same training data that was used to train the networks. The cost function used by the Minimization ranking was mean-squared error.

4. RESULTS

4.1. Preliminaries

Definition 4.1.1 (Parameter-Complexity Measure)**.** Let $f : [a, b] \to \mathbb{R}$ be a function and fix an approximation tolerance $\varepsilon > 0$, we introduce a parameter-complexity measure $\kappa(f; \varepsilon) \in \mathbb{R}$, which quantifies a property of f that governs how "hard" it is to approximate f within an error ε. $\kappa(f; \varepsilon)$ is not itself the number of parameters of the approximator $\mathcal{A}$ but rather a real number capturing the function's inherent complexity. Furthermore, assume that there

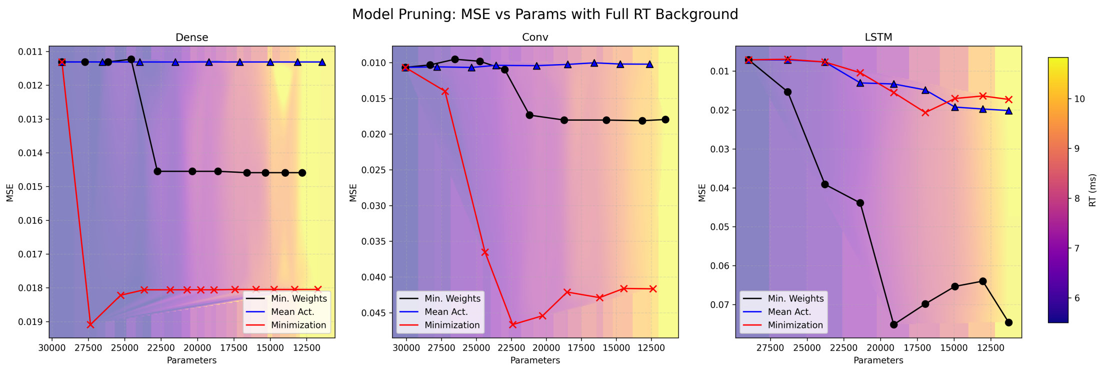

Figure 1: *Pruning performance of Dense, Conv, and LSTM models. X-axis shows parameter count (descending), Y-axis shows MSE (ascending), and RT (ms) is indicated inside each marker.*

exists a function $\Phi : \mathbb{R} \rightarrow \mathbb{N}$, which maps this complexity measure to the number of parameters required in a neural network architecture in order to achieve an approximation error less than ε. That is, the parameter count is given by:

$$\texttt{Param}(\mathcal{A}) = \Phi\left[\kappa(f;\varepsilon), \mathcal{A}, \mathcal{D}\{x \times y\}\right] \tag{7}$$

Note that the parameter count changes depending on the size of the pairs $x \times y \in \mathcal{D}$. Unless explicitly stated, herein we assume the cases where $size(\mathcal{D}) \rightarrow \infty$.

Under the Universal Approximation Theorem [27], consider an approximator $\mathcal{A}$ that seeks to reconstruct the underlying mapping between input and output data. As a base case, we introduce a set of candidate parameter-complexity measuring functions κ that can be directly related to the parameter count $\texttt{Param}(\mathcal{A})$.

4.2. Pruning Feedforward Dense Networks

4.2.1. Results of Pruning

Table 2 shows the results of the dense network pruning experiment using Minimum Weights, Mean Activations, and Minimization rankings respectively. Figure 1 shows a visualization of experiment results for all three ranking methods. Notably, the Mean Activations ranking maintained the smallest validation error as the parameter count decreased, while the largest validation error occurred when using Minimization ranking. As the parameter count decreased, the real-time performance improved by nearly a factor of 2. A model of the expected error is made to explain the behaviour of the error as the parameter count is reduced.

4.2.2. Error Modelling w.r.t Pruning in ReLU Networks

As shown, the error changes as more pruning is performed on the network. This error can be estimated using the work shown in Definition §4.1.1.

Proposition 4.2.1. Let $f : [a, b] \rightarrow \mathbb{R}$ be a continuous function, and let error $\varepsilon > 0$. The $\kappa(f, \varepsilon)$ is defined as the $\lfloor N \in \mathbb{Z}$ s.t $\exists N$ partition functions s, $s_i : [x_{i-l}.x_i] \rightarrow \mathbb{R}$ that is bounded within $[a, b]$ where $S = s_{i=1}^{N}$ partitions $f(x)$ satisfying:

- $a = x_0 < x_1 < \cdots < x_N = b$
- $\forall x \in S = \{s(x_{i-l}.x_i)\}_{i=1}^{N} \implies |f(x) - s_i(x)| \leq \varepsilon$

Lemma 4.2.1. For an error (between actual function f and a piecewise linear approximation), $E \leftarrow f(x) - s_i(x)$, under the maximum bounded f'' using Taylor's theorem [28], with $f''(\xi)$ is the f'' at some point chosen $(x - x_0)$.

$$E = \left|\frac{f''(\xi)}{2}(x - x_0)^2\right| \leq \frac{M}{2}h^2$$

$\varepsilon \leq \frac{M}{2}h^2 \Rightarrow$ a range $h \leq \sqrt{\frac{2\varepsilon*}{M}}$ is true *iff* f'' is bounded by the max(M) from Definition §8.0.2, and the error is bound to $\leq \frac{M}{2}h^2$

Lemma 4.2.2. Restating Theorem 1 of [29], a model with n_0 inputs and k hidden layers of widths $n_1, n_2, ..., n_k$ can divide the input space in $(\Pi_{i=1}^{k-1}\lfloor\frac{n_i}{n_0}\rfloor)\Sigma_{i=0}^{n_0}\binom{n_k}{i}$, and Corollary 5 of [30]: A rectifier neural network with n_0 input units and L hidden layers of width $n \geq n_0$ can compute functions that have $\Omega((\frac{n}{n_0})^{(L-1)n_0}n^{n_0}$ linear regions. It follows that such a network must be able to represent as many linear regions as the number of segments in our piecewise approximation: This yields that $\Phi[\max_{x\in[a,b]}|f''(x)|, \cdots] =$

$$\Omega\left(\frac{n}{n_0}\right)^{(L-1)n_0} n^{n_0} \geq (b - a)\sqrt{\frac{M}{2\varepsilon}} \tag{8}$$

This inequality connects the approximation error ε and the curvature measure M (via the bound on f'') to the complexity (i.e. the number of linear regions) that the ReLU network must be capable of expressing. The parameters of the network must scale so that it can produce a sufficient number of linear regions to approximate f with the desired error ε.

Then, viewing the left-hand side of the inequality as the network's capacity in terms of the number of distinct linear regions, the function Φ maps $\kappa(f; \varepsilon)$ to the required parameter count. For brevity, this inequality will not be restated in the following sections.

4.3. Pruning Feedforward Conv.

4.3.1. Results of Pruning

Table 3 shows the results of the convolutional network pruning experiment. Figure 1 shows a visualization of experiment results for

Iteration	Minimum Weights			Mean Activations			Minimization		
	Params	Error	F_{RT}	Params	Error	F_{RT}	Params	Error	F_{RT}
0	29313	0.01131	5.505	29313	0.01131	5.501	29313	0.01131	5.509
1	27729	0.01131	5.591	26540	0.01131	5.626	27351	0.01909	5.522
2	26145	0.01131	5.942	23967	0.01131	5.716	25277	0.01822	5.545
3	24561	0.01123	6.112	21525	0.01131	6.262	23668	0.01806	5.567
4	22757	0.01455	6.360	19214	0.01131	7.468	21867	0.01806	6.105
5	20358	0.01455	7.405	17092	0.01131	8.522	20710	0.01806	6.747
6	18600	0.01455	7.626	15007	0.01131	9.969	19775	0.01806	7.215
7	16608	0.01459	8.200	13129	0.01131	10.145	18869	0.01806	7.339
8	15332	0.01459	8.726	11444	0.01131	10.711	17430	0.01805	7.916
9	13971	0.01459	9.550	—	—	—	15982	0.01805	8.703
10	12812	0.01459	10.827	—	—	—	14748	0.01805	9.130
11	—	—	—	—	—	—	13322	0.01805	9.762
12	—	—	—	—	—	—	11728	0.01805	10.790

Table 2: *Comparison of structured pruning results for the* ***Dense*** *network across three ranking methods. Each group of columns shows the parameter count, error, and real-time factor F_{RT} at successive pruning iterations. Dashes indicate early termination of the pruning process for that method.*

Iteration	Minimum Weights			Mean Activations			Minimization		
	Params	Error	F_{RT}	Params	Error	F_{RT}	Params	Error	F_{RT}
0	30081	0.01067	3.737	30081	0.01067	3.704	30081	0.01067	3.505
1	28305	0.01032	3.941	27815	0.01059	4.136	27244	0.01400	3.926
2	26529	0.00953	4.515	25348	0.01068	4.549	24385	0.03651	4.424
3	24753	0.00981	4.731	23588	0.01038	4.918	22403	0.04666	4.967
4	22977	0.01097	5.189	20701	0.01043	5.624	20271	0.04543	5.560
5	21201	0.01735	5.431	18467	0.01025	6.402	18516	0.04211	5.917
6	18724	0.01804	6.036	16575	0.01003	6.846	16195	0.04289	6.524
7	15717	0.01804	7.147	14721	0.01020	7.509	14472	0.04161	7.497
8	13149	0.01813	8.239	12622	0.01022	8.206	12384	0.04166	8.505
9	11496	0.01796	9.100	—	—	—	—	—	—

Table 3: *Comparison of structured pruning results for the* ***Convolutional*** *network across three ranking strategies. Each column group reports the number of parameters, error, and real-time factor F_{RT} at successive pruning iterations.*

all three ranking methods. The Mean Activations ranking maintained the smallest validation error as the parameter count decreased. Minimum Weights ranking maintained a similarly small validation error until the parameter count dropped below 22,000, at which point the validation error increased by nearly an order of magnitude. As with the dense network experiment, pruning with Minimization ranking led to the largest validation error. As the parameter count decreased, the real-time performance improved by more than a factor of 2. A model of the expected error is made to explain the behaviour of the error as the parameter count is reduced.

4.3.2. *Error Modelling w.r.t Conv.*

Consider a single-layer 1D CNN with input dimension n_0, kernel size k, stride 1, and d_1 convolutional filters, each followed by a piecewise-linear activation function σ. From [31], the number of linear regions R_N generated by this CNN scales polynomially as $R_N = \Theta(d_1^k)$. The weight sharing and locality imposes structural dependencies. More specifically, the convolutional layer's activation boundaries repeat across the input domain due to translation-invariant filters, and therefore the CNN will not independently position d_1 partitions everywhere on $\mathbb{R}$ —as in §4.2.2.

As layers are stacked, the partitioning of the input becomes progressively finer, and the number of regions can multiply through compositions. Approximately, if layer 1 yields r_1 regions and layer 2 yields r_2 regions on each region of layer 1, the composition would yield on the order of $r_1 \cdot r_2$ total regions [32].

In comparison, weight sharing means fewer independent parameters, so a single convolution layer cannot cover the input space as freely as a dense layer. This explains the polynomial (rather than exponential) scaling of region count. Importantly, this improves parametric efficiency when the target function has local “features" (in function space or the corresponding higher dimensional latent space).

Lemma 4.3.1 (CNN Approximation Error for Smooth Functions)**.** Let $\mathcal{A}$ be a CNN that realizes a continuous function $f : [a, b] \to \mathbb{R}$ that is twice continuously differentiable with a bounded second derivative ($\kappa(f; \varepsilon) \to$ §8.0.2). Then the uniform approximation error $\|\mathcal{A}(x) - f(x)\|_\infty$:

$$\max_{x\in[a,b]} |\mathcal{A}(x) - f(x)| \leq \frac{M}{8} \cdot \max_{1\leq i\leq R} (x_i - x_{i-1})^2 \tag{9}$$

where $x_0 = a < x_1 < \cdots < x_R = b$ are the breakpoints defining the linear segments of $\mathcal{A}$. If these segments are uniformly spaced with $h = (b-a)/R$, the bound simplifies to: $\|\mathcal{A}(x) - f(x)\|_\infty \leq \frac{M(b-a)^2}{8R^2}$. Equivalently, to guarantee an approximation error $\mathcal{L} =$

$\|\mathcal{A}(x) - f(x)\|_\infty \leq \varepsilon$, it suffices that

$$R \geq \frac{(b-a)\sqrt{M}}{\sqrt{8\varepsilon}}. \tag{10}$$

Proof. On each subinterval $[x_{i-1}, x_i]$, Taylor's theorem (with the Lagrange remainder) gives $f(x) = \tilde{f}(x) + \frac{f''(\xi_i)}{2}(x - x_{i-1})(x - x_i)$ for some $\xi_i \in [x_{i-1}, x_i]$, where $\tilde{f}$ is the linear interpolant. The maximum of the error term $|f(x) - \tilde{f}(x)|$ over the interval occurs at the midpoint, yielding the bound $\frac{M}{8}(x_i - x_{i-1})^2$. □

Remark 4.3.1. Since a CNN realizes a piecewise function, its expressivity in terms of the number of distinct linear regions must at least match the number required by classical approximation theory.

In ideal cases, where all partitions are uniform (where $h = \frac{b-a}{R}$ is the length of each subinterval), it suffices to choose $\Phi := R \geq \left\lceil \frac{(b-a)\sqrt{M}}{\sqrt{8\varepsilon}} \right\rceil$ to guarantee $\mathcal{L} \leq \varepsilon$.

4.4. Pruning LSTM

4.4.1. Results of Pruning

Table 4 shows the results of the recurrent network pruning experiment. Figure 1 shows a visualization of experiment results for all three ranking methods. Both the Mean Activations and Minimization rankings maintained similarly low validation errors as the parameter count decreased. Notably, pruning with Minimum Weights ranking led to a significantly larger validation error. It makes sense that Minimum Weights would be a poor ranking strategy for pruning recurrent neural networks given that recurrent systems can be extremely sensitive to small changes to coefficients in the recursive path. As the parameter count decreased, the real-time performance improved by more than a factor of 2. A model of the expected error is made to explain the behaviour of the error as the parameter count is reduced.

4.4.2. Error Modelling w.r.t LSTM

Following Definition 8.0.1, for a given target state-transition function of the underlying dynamical system, let $f : \mathbb{R}^n \times \mathbb{R}^{d_x} \to \mathbb{R}^n$ denote the true state evolution mapping, and $g : \mathbb{R}^n \to \mathbb{R}$ denote the output (readout) function, such that the system evolves as $s_t = f(s_{t-1}, x_t)$ and $y_t = g(s_t)$. The recurrent approximator $\mathcal{A}$ is composed of a learned state transition function $F_\Theta : \mathbb{R}^n \times \mathbb{R}^{d_x} \to \mathbb{R}^n$ and a learned readout function $G_\Theta : \mathbb{R}^n \to \mathbb{R}$, such that the predicted state is $h_t = F_\Theta(h_{t-1}, x_t)$ and the output is $\hat{y}_t = G_\Theta(h_t)$. The subscript Θ denotes the set of learned parameters of the network.

The parameter-complexity measure $\kappa(f; \varepsilon)$ of an approximator $\mathcal{A}$ approximating the true dynamics f depends on the Lipschitz constant L_f of the transition dynamics and the achievable one-step error ε^* for a given tolerance $\varepsilon > 0$.

Proposition 4.4.1. To derive the bound $|\mathcal{A}(x_t) - f(x_t)|$, we begin by defining the hidden state error at time t as $\varepsilon_t = \|s_t - h_t\|$, where s_t is the true hidden state and h_t is the approximated hidden state generated by $\mathcal{A}$. Assuming the target transition function f is Lipschitz continuous with constant L_f yields that $L_f \varepsilon_{t-1} =$

$$L_f \|s_{t-1} - h_{t-1}\| \geq \|f^t(s_{t-1}, x_t) - f^t(h_{t-1}, x_t)\| \tag{11}$$

If the $\mathcal{A}$ realizes a one-step error bounded by $\|f^t(h_{t-1}, x_t) - F_\Theta(h_{t-1}, x_t)\| \leq \varepsilon^*$, then by the triangle inequality [33] the total hidden state error satisfies $\varepsilon_t \leq L_f \varepsilon_{t-1} + \varepsilon^*$. Unrolling this recurrence yields $\varepsilon_t \leq L_f^t \varepsilon_0 + \varepsilon^* \sum_{j=0}^{t-1} L_f^j = L_f^t \varepsilon_0 + \varepsilon^* \frac{1-L_f^t}{1-L_f}$ for $L_f \neq 1$.

If the readout function[2] g is Lipschitz continuous with constant L_g, then the output error is bounded as $|\mathcal{A}(x_t) - f(x_t)| = |G_\Theta(h_t) - g(s_t)| \leq L_g \varepsilon_t$. Substituting the expression for ε_t gives the final bound:

$$|\mathcal{A}(x_t) - f(x_t)| \leq L_g \left(L_f^t \varepsilon_0 + \frac{\varepsilon^*}{1 - L_f}(1 - L_f^t) \right). \tag{12}$$

Proposition 4.4.2 (Parameter Lower Bound via One-Step Error Decay)**.** Suppose the one-step approximation error ϵ of the recurrent approximator $\mathcal{A}$ admits an inverse power-law scaling with parameter count. That is, for some constants $C > 0$ and $\alpha > 0$,

$$\epsilon \leq C\,\texttt{Param}(\mathcal{A})^{-\alpha} \tag{13}$$

Following from §12, the asymptotic regime where $t \to \infty$ or $\varepsilon_0 \to 0$, this reduces to the uniform bound to:

$$|\mathcal{A}(x_t) - f(x_t)| \leq \frac{L_g\,\epsilon}{1 - L_f} \tag{14}$$

To guarantee that the approximation error satisfies $|\mathcal{A}(x_t) - f(x_t)| \leq \varepsilon$, it is necessary that $\epsilon \leq \frac{(1-L_f)\,\varepsilon}{L_g}$. Substituting the assumed scaling of ϵ in terms of $\texttt{Param}(\mathcal{A})$, we obtain

$$C\,\texttt{Param}(\mathcal{A})^{-\alpha} \leq \frac{(1 - L_f)\,\varepsilon}{L_g} \tag{15}$$

which implies a lower bound on the number of parameters:

$$\texttt{Param}(\mathcal{A}) \geq \left(\frac{C\,L_g}{(1 - L_f)\,\varepsilon} \right)^{\frac{1}{\alpha}} \tag{16}$$

4.5. Analysis

When viewing all the experiment results together, several notable results are evident. First, all three neural networks were able to be successfully pruned, to point of achieving a significant improvement in real-time performance with minimal change in validation error. Second, pruning with a Mean Activations ranking achieved good results for all three network architectures. Additionally, pruning with Minimum Weights and Minimization rankings achieved poor results for most of the networks, although Minimization pruning was successful for the recurrent network. Finally, an error model for each network architecture provides insight into the required parameter count needed accurately model a system with some given complexity.

The source code used to train the networks and perform the experiments is available on GitHub.[3] An audio plugin was developed

[2] f^t and g decompose the overall behavior of the system. $f^t(s_{t-1}, x_t)$ describes system transitions from state s_{t-1} to state s_t given x_t; $g(s_t)$ maps the current state to the observable output y_t. Thus, full system behavior can be written as a composition: $f(x_{1:t}) = (g \circ f^{(t)})(s_0, x_{1:t})$, where $f^{(t)}$ is recursively applied over the input sequence to produce s_t, and g maps that state to the final output. $\mathcal{A} = G_\Theta \circ F_\Theta^{(t)}$ mimics this composition.

[3] `https://github.com/jatinchowdhury18/neural-pruning`

Iteration	Minimum Weights			Mean Activations			Minimization		
	Params	Error	F_{RT}	Params	Error	F_{RT}	Params	Error	F_{RT}
0	28981	0.00711	7.157	28981	0.00711	7.228	28981	0.00711	7.233
1	26321	0.01533	8.126	26321	0.00712	8.110	26321	0.00697	8.147
2	23789	0.03911	8.593	23789	0.00767	8.547	23789	0.00768	8.642
3	21385	0.04389	9.737	21385	0.01305	9.699	21385	0.01043	9.799
4	19109	0.07513	10.462	19109	0.01332	10.409	19109	0.01553	10.499
5	16961	0.06982	11.738	16961	0.01485	11.836	16961	0.02066	11.935
6	14941	0.06532	12.675	14941	0.01926	12.944	14941	0.01705	12.952
7	13049	0.06397	14.281	13049	0.01975	14.859	13049	0.01640	14.892
8	11285	0.07459	16.443	11285	0.02015	16.583	11285	0.01729	16.604

Table 4: *Comparison of structured pruning results for the* ***LSTM*** *network across three ranking strategies. Each column group shows parameter count, error, and real-time factor F_{RT} per pruning iteration.*

(CLAP/VST3/AU) based on the LSTM network, to demonstrate the effects of pruning the network with different ranking methods on real-time audio input. Pre-built binaries of the plugin for Windows and MacOS are available in the GitHub repository.

5. CONCLUSIONS

This paper investigated the effectiveness of structured pruning strategies for real-time neural audio effects, evaluating three representative architectures —dense, convolutional, and recurrent networks —under various pruning schedules and ranking criteria. We demonstrated that substantial reductions in parameter count can be achieved with minimal degradation in numerical fidelity, yielding meaningful improvements in real-time performance across all architectures.

We introduced a novel theoretical framework for error modeling as parameter count scales (downwards), providing network architecture specific bounds on approximation error in terms of network complexity and structural properties of the target function. These bounds offer a principled way to relate pruning-induced sparsity to model fidelity and highlight the differences in parameter efficiency and functional flexibility between dense, convolutional, and recurrent models.

Overall, our results suggest that structured pruning, guided by activation-aware ranking strategies, offers a viable and theoretically grounded pathway for optimizing neural audio effects in real-time systems. Future research topics may include pruning audio effect neural networks for sparsity, more sophisticated parameter-complexity analysis, and the development of novel ranking methods, as well as other methods for optimizing neural networks including distillation and quantization.

6. ACKNOWLEDGMENTS

Many thanks to the anonymous reviewers!

7. REFERENCES

[1] S. D'Angelo, L. Gabrielli, and L. Turchet, "Fast Approximation of the Lambert W Function for Virtual Analog Modelling," in *Proc. of the 22nd Int. Conference on Digital Audio Effects (DAFx-19)*, Sept. 2019.

[2] M. A. Martínez Ramirez and J. D. Reiss, "Modeling of Nonlinear Audio Effects with End-to-End Deep Neural Networks," *arXiv preprint arXiv:1810.06603*, Oct. 2018.

[3] E.-P. Damskägg, L. Juvela, and V. Välimäki, "Real-Time Modeling of Audio Distortion Circuits with Deep Learning," in *Proc. of the 16th Sound and Music Computing Conference (SMC-2019)*, May 2019.

[4] A. Wright, E.-P. Damskägg, and V. Välimäki, "Real-Time Black-Box Modelling with Recurrent Neural Networks," in *Proc. of the 22nd Int. Conference on Digital Audio Effects (DAFx-19)*, Sept. 2019.

[5] A. Carson, A. Wright, J. Chowdhury, V. Välimäki, and S. Bilbao, "Sample Rate Independent Recurrent Neural Networks for Audio Effects Processing," in *27th International Conference on Digital Audio Effects*, Surrey, UK, 2024, p. 17.

[6] Neil Gershenfeld, *The Nature of Mathematical Modeling*, Cambridge University Press, USA, 1999.

[7] D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag, "What is the State of Neural Network Pruning?," *arXiv preprint arXiv:2003.03033*, 2020.

[8] Y. LeCun, J. Denker, and S. Solla, "Optimal Brain Damage," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed. 1989, vol. 2, Morgan-Kaufmann.

[9] C.W. Omlin and C.L. Giles, "Pruning Recurrent Neural Networks for Improved Generalization Performance," in *Proceedings of IEEE Workshop on Neural Networks for Signal Processing*, 1994, pp. 690–699.

[10] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning Convolutional Neural Networks for Resource Efficient Inference," *arXiv: Learning*, 2016.

[11] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks," *J. Mach. Learn. Res.*, vol. 22, no. 1, Jan. 2021.

[12] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication using Compressed Sparse Blocks," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2009, SPAA '09, p. 233–244, Association for Computing Machinery.

[13] V. Isaac–Chassande, A. Evans, Y. Durand, and F. Rousseau, "Dedicated Hardware Accelerators for Processing of Sparse Matrices and Vectors: A Survey," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, Feb. 2024.

[14] J. Chowdhury, "RTNeural: Fast Neural Inferencing for Real-Time Systems," *arXiv preprint arXiv:2106.03037*, 2021.

[15] TensorFlow Developers, "Trim Insignificant Weights," [Online; accessed Feb. 16, 2025].

[16] M. Paganini, "Pruning Tutorial," July 2019, [Online; accessed Feb. 16, 2025].

[17] S. Anwar, K. Hwang, and W. Sung, "Structured Pruning of Deep Convolutional Neural Networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, Feb. 2017.

[18] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2015.

[19] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," *arXiv preprint arXiv:1608.08710*, 2017.

[20] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv preprint arXiv:1503.02531*, 2015.

[21] L. Wei, Z. Ma, C. Yang, and Q. Yao, "Advances in the Neural Network Quantization: A Comprehensive Review," *Applied Sciences*, vol. 14, no. 17, 2024.

[22] T. Asami, R. Masumura, Y. Yamaguchi, H. Masataki, and Y. Aono, "Domain Adaptation of DNN Acoustic Models using Knowledge Distillation," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 5185–5189.

[23] R. Yamamoto, E. Song, and J.-M. Kim, "Probability Density Distillation with Generative Adversarial Networks for High-Quality Parallel Waveform Generation," in *Interspeech*, 2019.

[24] S.T. Sreenivas, S. Muralidharan, R. Joshi, M. Chochowski, A.S. Mahabaleshwarkar, G. Shen, J. Zeng, Z. Chen, Y. Suhara, S. Diao, C. Yu, W.C. Chen, H. Ross, O. Olabiyi, A. Aithal, O. Kuchaiev, D. Korzekwa, P. Molchanov, M. Patwary, M. Shoeybi, J. Kautz, and B. Catanzaro, "LLM Pruning and Distillation in Practice: The Minitron Approach," *arXiv preprint arXiv:2408.11796*, 2024.

[25] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[26] T. Ganguli and E. K. P. Chong, "Activation-Based Pruning of Neural Networks," *Algorithms*, vol. 17, no. 1, 2024.

[27] Y. Lu and J. Lu, "A Universal Approximation Theorem of Deep Neural Networks for Expressing Probability Distributions," *Advances in neural information processing systems*, vol. 33, pp. 3094–3105, 2020.

[28] Richard L. Burden and Douglas J. Faires, *Numerical Analysis*, Cengage Learning, 2010.

[29] R. Pascanu, G. Montufar, and Y. Bengio, "On the Number of Response Regions of Deep Feed Forward Networks with Piece-Wise Linear Activations," *arXiv preprint arXiv:1312.6098*, 2014.

[30] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the Number of Linear Regions of Deep Neural Networks," *arXiv preprint arXiv:1402.1869*, 2014.

[31] H. Xiong, L. Huang, M. Yu, L. Liu, F. Zhu, and L. Shao, "On the number of linear regions of convolutional neural networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 10514–10523.

[32] Chinmay Hegde, "Foundations of Deep Learning," 2022, Chapter 3: The Role of Depth.

[33] Harold R. Jacobs, *Geometry: Seeing, Doing, Understanding*, Macmillan, 2003.

8. APPENDIX

Definition 8.0.1 (Lipschitz Continuity)**.** A function f is said to be *Lipschitz continuous* on the interval $[a, b]$ if and only if there exists a non-negative constant L such that:

$$|f(x) - f(y)| \leq L|x - y|, \quad \forall x, y \in [a, b]$$

This definition establishes that the rate of change of a Lipschitz continuous function is bounded by the constant L, often called the Lipschitz constant. Geometrically, this means that the graph of f cannot contain any 'sharp' features like jumps or vertical tangent lines, and the slope between any two points on the graph is at most L.

Definition 8.0.2 (Maximum Bounded Second Derivative)**.** For a function f on $[a, b]$ where the second derivative $f'' = \frac{d^2 f(x)}{dx^2}$ exists and is bounded in absolute value, we define the maximum magnitude of the second derivative as:

$$M = \max_{x \in [a,b]} |f''(x)|$$

The constant M provides a measure of the maximum curvature or "bending" of the function f across its domain. A smaller value of M indicates a function with more gradual changes in its slope, while a larger value of M corresponds to more pronounced changes in the function's rate of change.