ANIRA: An Architecture for Neural Network Inference in Real-Time Audio Applications

Valentin Ackva * 
Audio Communication Group
Technische Universität Berlin
Berlin, Germany
valentin.ackva@gmail.com

Fares Schulz *
Audio Communication Group
Technische Universität Berlin
Berlin, Germany
fares.schulz@tu-berlin.com

***Abstract*—Numerous tools for neural network inference are currently available, yet many do not meet the requirements of real-time audio applications. In response, we introduce *anira*, an efficient cross-platform library. To ensure compatibility with a broad range of neural network architectures and frameworks, *anira* supports ONNX Runtime, LibTorch, and TensorFlow Lite as backends. Each inference engine exhibits real-time violations, which *anira* mitigates by decoupling the inference from the audio callback to a static thread pool. The library incorporates built-in latency management and extensive benchmarking capabilities, both crucial to ensure a continuous signal flow. Three different neural network architectures for audio effect emulation are then subjected to benchmarking across various configurations. Statistical modeling is employed to identify the influence of various factors on performance. The findings indicate that for stateless models, ONNX Runtime exhibits the lowest runtimes. For stateful models, LibTorch demonstrates the fastest performance. Our results also indicate that for certain model-engine combinations, the initial inferences take longer, particularly when these inferences exhibit a higher incidence of real-time violations.**

***Index Terms*—neural network, real-time audio, inference engine, audio effects, deep learning, digital signal processing**

I. Introduction

In recent years, neural networks have become an integral part of modern audio digital signal processing. Their applications include audio classification [1], audio transcription [2], audio source separation [3], audio synthesis [4], [5], [6] and audio effects [7]. While offline processing is inherently supported, translating these architectures to real-time implementations remains challenging. To overcome this hurdle, the special requirements of real-time audio systems must be reconciled with the high computational complexity of neural network inference.

Neural networks are commonly trained in high-level frameworks such as TensorFlow [8] or PyTorch [9] in the Python programming language. While these frameworks provide a wide range of tools for training and evaluating neural networks, they are not optimized for inference and are particularly unsuitable for real-time audio applications. To circumvent this issue, one approach is to implement the network operations in high-performance programming languages such as C or C++, and then export and load the trained parameters from the high-level framework. While this approach provides the ability to optimize the inference code for the neural network being used, it is often time-consuming and lacks the flexibility to integrate different neural networks.

To facilitate this process, a variety of inference engines have been developed in high-performance languages. These engines optimize the execution of neural networks on different hardware platforms and provide implementations of the most common neural network layers. In this paper, we focus on the three most common inference engines: ONNX Runtime, LibTorch, and TensorFlow Lite. To optimize inference runtimes, these engines prioritize average processing times over real-time safety during execution. However, to ensure a continuous and uninterrupted signal flow in real-time audio applications, it is also necessary to minimize the maximum inference time or worst-case execution time.

This need for minimum worst-case execution times, and hence deterministic runtimes in audio callbacks, has led to the adoption of certain real-time principles for conventional audio DSP algorithms. These principles include the avoidance of system calls such as dynamic memory allocation, unconditional thread locks, interaction with the thread scheduler, or code awaiting hardware on operating systems with non-real-time-safe architectures, as they may exhibit undeterministic runtimes [10]. For the aforementioned inference engines, tests have been conducted to verify their adherence to these real-time principles [11], [12]. However, different conclusions have been derived. Our work validates the assumptions made in [11] and quantifies the real-time violations observed in inference executions.

To ensure that these inference engines can be executed safely in real-time audio applications, we propose a library called *anira*. The library provides an architecture to outsource the processing of the inference engines to a thread pool, allowing the audio callback to remain free of blocking operations. Furthermore, the library handles processing of the inference engines in discrete chunks, a critical feature for inference in neural networks that are designed to process inputs of a predetermined size. Ultimately, the library aims to provide a framework for integrating neural networks into real-time audio applications. It is currently compatible with the three

*The authors have contributed equally to this work.

inference engines ONNX Runtime, LibTorch, and TensorFlow Lite, the operating systems Linux, MacOS, and Windows, and supports various types of neural networks. The source code of the library is publicly available [13].

Anira's Linux compatibility allows it to support not only desktop applications but also multicore embedded systems and servers. While there exist pipelines that provide insight into the deployment of neural networks in specific embedded systems [14], [15], these pipelines are limited in their generalizability. In contrast, *anira* employs a library design that aims at an abstraction on a higher level by offering an API compatible with multiple configurations. Moreover, the library's ability to manage inference tasks across multiple threads allows for the optimal utilization of the underlying hardware's processing capabilities. The library's integration with the CMake build system [16] enables its incorporation into a wide range of build pipelines and environments.

In addition to its real-time implementation, the computational load of a neural network is another important factor for real-time executability. While several neural networks have been evaluated for real-time performance, the benchmarks often lack comparability due to different inference implementations and the use of different inference engines or programming languages [6], [7], [17]. Further, the missing comprehensive documentation of the benchmarking procedure make it difficult to understand the circumstances under which the benchmarks were conducted.

We address this issue by including built-in benchmarking capabilities for evaluating the real-time performance of neural networks in our library. The benchmarks may be executed with a variety of configurations, including different buffer sizes, which is a standard feature in most real-time audio environments. The library's compatibility with a range of inference engines, neural networks, and operating systems allows for the execution of benchmarks under identical conditions, thereby ensuring comparability and reproducibility.

To date, the neural network runtimes for the three inference engines in the audio domain have only been compared for audio classifier models consisting of dense and batch normalization layers [12]. In this work, the execution times of three neural network models for audio effect emulation were measured under various circumstances. Each model incorporates a distinct set of layers, including convolutional, recurrent, and stateful layers. Statistical models of the resulting datasets reveal that certain inference engines are more suited to certain neural network layers than others, and that in some cases, the initial inferences performed by the engines are slower than subsequent inferences. In a follow-up study, we examine the performance of the inference engines on a range of differently sized convolutional neural networks. Our findings demonstrate that the engines exhibit varying levels of efficiency across different network sizes.

The remainder of this paper is organized as follows: Section II describes the implementation of the library, including its built-in benchmarking tools. Section III details the methods used to verify the real-time safety of the inference engines and the library as well as evaluate their performances. The results are then presented in Section IV and discussed in Section V.

II. Implementation

A. Inference Engines

The *anira* library enables the use of various inference engines as backends. At present, TensorFlow Lite 2.16.1 [18], ONNX Runtime 1.17.1 [19], and LibTorch 2.2.2 [20] are supported. The choice of these engines was guided by the following criteria: compatibility with C/C++, cross-platform operation, support for common processor architectures, open-source licensing favorable for commercial use, and support of common neural layers. All inference engines are configured with the default settings, with the exception of the number of threads that the engines can utilize, which is limited to one, because *anira* manages the parallelization of the inference tasks.

B. Interface

The interface is designed to streamline real-time inference processes across a wide range of neural network types. Internally, *anira* then manages all critical operations, including thread allocation, memory handling, and execution control. Figure 1 provides an architectural schematic of the library, illustrating the external audio application (*extern*), the interface of the library (*public anira API*), and the internal components (*private anira*).

For interaction with the library, the `InferenceHandler` class is provided. This class serves as the interface for configuring and executing neural network models within the *anira* framework. Its constructor requires an `InferenceConfig` struct to pass the inference settings as outlined in Table I. In addition, an optional `PrePostProcessor` can be configured to perform model-dependent adjustments on the input and output data, such as extending the input sequence with past audio data to meet the specific requirements of the model. It is noteworthy that resampling, which is often necessary due to the sample rate dependency of many neural networks [21], is currently not supported within *anira* and must, therefore, be handled externally.

TABLE I: Parameters for the `InferenceConfig`, used to configure the *anira* library. All options, with the exception of those with a default value, are mandatory. Options denoted with † are dependent on the enabled inference engines within *anira*. Each engine requires its own set of these parameters.

Option	Description
model†	binary / path of model (∗.pt, ∗.onnx, ∗.tflite)
model_input_shape†	input shape, e.g. {batch_size, input_size, 1}
model_output_shape†	output shape, e.g. {batch_size, output_size, 1}
max_inference_time	maximum measured inference time (ms)
model_latency	internal latency of the model (samples)
warm_up	number of warm up inferences (default = 0)
wait_in_process_block	proportional wait time to receive data (value between 0.0 - 0.95, default = 0.0)
bind_session_to_thread	bind instance to one thread (default = off)
num_threads	number of threads created by thread pool (default = number of concurrent threads - 1)

The initialization of the `InferenceHandler` instance requires the specification of the buffer size, the number of

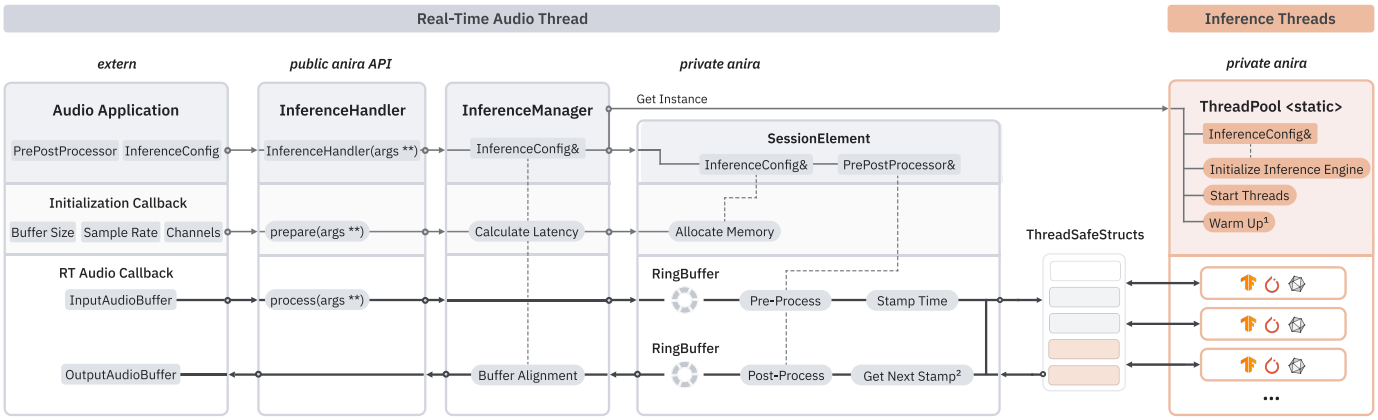

Fig. 1: Architectural overview of the *anira* library. The diagram illustrates the design and the principle components involved in the inference process. The overview is divided into three main sections: the Constructor, the Initialization Callback, and the Real-Time Audio Callback. The "*Warm Up*" operation, marked with [1], is optional and allows for a set number of inferences before the start of the audio callback. The "*Get Next Stamp*" operation, marked with [2], can also be configured to await a specified period of time for the processed data to become available.

channels, and the sample rate of the host. Once initialized, the necessary memory allocations are completed, ensuring that *anira* operates safely within the audio callbacks. During runtime, the inference engine can be selected. The integration of the library is demonstrated through a JUCE audio plugin example included in the repository [13].

C. Internal Architecture

Besides the *public anira API*, the library consists of three main internal components: the `InferenceManager`, the `SessionElement`, and the `ThreadPool`. The interplay of these three classes enables the separation of the inference process from the audio callback onto independent high-priority threads to ensure real-time safety. This also permits the processing of audio data in discrete chunks independent of the host buffer size, which is crucial for models with a fixed input shape. In this process, the `InferenceManager` is responsible for buffer alignment and latency calculation, the `SessionElement` stores the audio data and inference results, and the static `ThreadPool` orchestrates the inference threads.

As data sharing between threads requires precautions to prevent race conditions and deadlocks, the `SessionElement` provides specific `ThreadSafeStucts`. *Anira* offers two different implementations of these structures to regulate access to the audio data and to submit inference tasks to the `ThreadPool`. The first option follows strict real-time safety rules with raw atomic values, while the second option uses semaphores to allow a controlled blocking operation that can further reduce latency (cf. Section II-D). `std::atomic<bool>` / `std::binary_semaphore` ensure that the audio data can only be accessed by one thread, while `std::atomic<int>` / `std::counting_semaphore` are used to submit inference tasks to the queue. Timestamps, realized through inference buffer counting, ensure that the processed audio output is in the correct sequence.

The static design choice of the `ThreadPool` is intentional to overcome a problem called oversubscription that may arise when multiple instances are created, each with its own high-priority inference thread. This becomes problematic when the demand for real-time threads exceeds the number of available hardware threads, leading to a degradation in overall performance [22]. The maximum number of threads in the pool is configurable. For stateless models, all inference threads are created upon initialization. This approach enables accelerated inference through parallel inferencing.

D. Latency

To maintain a stable and uninterrupted signal flow, the `InferenceManager` determines the minimum latency that must be applied. Besides the inherent latency of the host, defined by its host buffer size, there are three different key factors that contribute to the overall latency: potential buffer size mismatches between the host buffer size and the model input size, the worst-case inference time, and the internal latency of the model. Whereas the host buffer size, model input size, and internal model latency are distinct values, the worst-case inference time can only be estimated by extensive benchmarks (cf. Section II-E). This is due to the non-deterministic runtimes of the inference engines, which are a consequence of the real-time violations that the engines exhibit (cf. Section IV-A).

To account for the potential buffer size mismatch, we rely on the algorithm provided by Letz [23]. In the paper, Letz identified the minimum latency required to match buffers of differing sizes A and B. This was achieved by analyzing the remainder of the division of integer multiples n of A by B, $\mathrm{mod}\ (nA, B)$, until the pattern repeats.

Furthermore, it is noteworthy that the buffer size mismatch may also incorporate a varying number of inferences that can be performed within one audio callback. We leverage the pattern approach proposed by Letz to determine the maximum number of possible inferences that can be expected within an audio callback and derive the combined worst-case inference time. However, since the request for inferenced data to the thread pool is made only once per audio callback, the worst-case inference time must be rounded up to the next integer

multiple of the host buffer size.

Finally the model's internal latency is added to the total latency calculation. The resulting formula is as follows:

$$L_{\text{total}} = H_{\text{adapt}} + \left\lceil \frac{I_{\max}}{H_{\text{host}}} \right\rceil \cdot H_{\text{host}} + M_{\text{int}} \quad (1)$$

The term H_{adapt} represents the latency required to adapt the buffer sizes, $I_{\max}$ denotes the combined worst-case inference time in samples, H_{host} is the host buffer size, and M_{int} refers to the internal latency of the model.

Given that the value of the second term in (1) is allways greater than or equal to H_{host}, L_{total} will never reach zero. For some applications, this may be an unacceptable outcome. Therefore, *anira* includes an option within the `InferenceConfig` to specify a wait time and thus introduce a controlled blocking operation to the audio callbacks to receive processed data (cf. Table I). The blocking operation is implemented as a `std::counting_semaphore::try_acquire_until` method, and requires the usage of the `ThreadSafeStucts` implementation with semaphores (cf. Section II-C). This enables *anira* to achieve real-time processing with no additional latency introduced by the library, for certain configurations. Subsequently, (1) is modified to:

$$L_{\text{total}} = H_{\text{adapt}} + \left\lceil \frac{I_{\max} - H_{\text{host}} \cdot W}{H_{\text{host}}} \right\rceil \cdot H_{\text{host}} + M_{\text{int}} \quad (2)$$

In this context, W denotes the proportional wait time, as defined in the `InferenceConfig` .

E. Benchmarks

The benchmarks are designed to evaluate the buffer execution runtimes of the *anira* library across a range of different configuration combinations. These configuration options include the selected inference engine, the neural network model, and the buffer size. Technically, the benchmarking functionality is implemented within the Google Benchmark [24] and Google Test [25] framework. When building the library, the benchmarking functionality can be enabled optionally. Subsequently, the benchmarks can be executed as unit tests.

In order to ensure that the runtime results are representative of the processing times that occur in real-time audio applications, it is essential that the test environment closely simulates the target environment. This is achieved by implementing a series of steps typically found in audio applications. Firstly, the priority of the benchmark process is set to the highest possible level. Subsequently, a static instance of the `InferenceHandler` class is constructed and initialized. In the next step, the time required for an input buffer, filled with random generated samples, to be processed by the `InferenceHandler` instance is measured. This process is repeated a defined number of times, with each iteration being measured individually in order to simulate the continuous processing of audio buffers in a real-time audio environment. Once the iterations have been completed, the `InferenceHandler` instance is destroyed, and the benchmark process sleeps for as long as it took to process all the iterations. The aforementioned steps are then repeated for a specified number of repetitions.

This methodology allows for the simulation of multiple repetitions of the initial inference runs, and therefore, the warm-up phases of the inference engines. Optinally this procedure can be repeated for different configurations.

III. Methods

A. Selected Neural Networks

To evaluate the inference engines and the *anira* library, three neural network models are selected. These models are tailored for end-to-end audio processing tasks, with a special focus on real-time audio effects emulation. Each of these models represents one of the prominent neural network architectures that have been identified as particularly suitable in recent research: convolutional neural networks (CNN), recurrent neural networks (RNN), and hybrid configurations [26]. The hyperparameters of the models are selected to ensure efficient real-time performance. All models have been trained with the same dataset, a three minute dry and wet recording of an Ibanez TS9 Tube Screamer guitar pedal from [27].

1) Convolutional Neural Network: The chosen CNN is a Temporal Convolutional Network (TCN) inspired by the work of Steinmetz and Reiss [17]. TCN architectures use dilated causal convolution layers to capture long-range dependencies in sequential data, achieving a wide receptive field without the need for a deep layered structure. The dilation grows with each TCN block, so that the dilation of the convolution layer in the i-th TCN block is equal to $d_i = d^{i-1}$, where d denotes the dilation factor. In contrast to the model architecture presented by Steinmetz and Reiss, the FiLM and batch normalization layers have been removed for simplicity and to focus on streamlined, efficient computation. As a result, the model consists of TCN blocks containing a causal convolution layer, a PReLU activation layer, and a residual connection (cf. Fig. 2). Three different TCN models are defined with different hyperparameter combinations in Table II: a large (CNN-29k), medium (CNN-15k), and small model (CNN-1k).

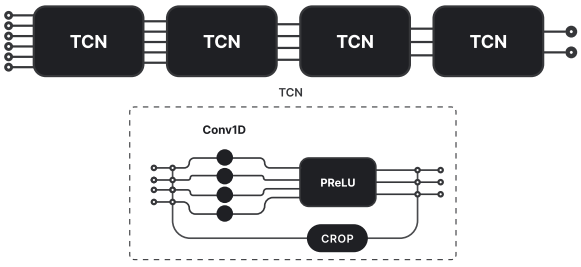

Fig. 2: Schematical representation of the selected CNN model architecture. The top figure shows the overall model structure, while the bottom figure provides a detailed view of the TCN block.

2) Recurrent Neural Network: The chosen RNN is adapted from the work of Wright, Damskägg, Juvela, *et al.* [7]. The model uses a stateful LSTM layer, a dense layer, and a residual connection to predict one audio sample based on the previous

samples (cf. Fig. 3). It is designed to process audio samples sequentially, with the LSTM layer retaining information across inputs. This means that multithreading is not supported, as the internal state of the LSTM layer must be updated after each buffer. Also, this network is only compatible with LibTorch and TensorFlow Lite, as ONNX Runtime inherently supports only stateless operations [28]. However, it should be noted that the state of the model could be handled externally, but, as this would alter the model graph, we have chosen to exclude ONNX Runtime from the RNN comparison. At the end of the model architecture, the dense layer processes the output of the LSTM sequence and outputs the resulting sequence of ouput samples. The selection of the hyperparameters for the defined model, RNN-2k, is detailed in Table II.

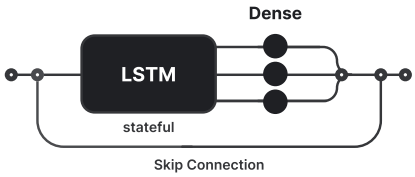

Fig. 3: Schematical representation of the selected RNN model architecture.

3) Hybrid Neural Network: The hybrid configuration model is a stateless LSTM layer combined with two convolutional layers and a dense layer [27], [29]. Stateless LSTM layers are a variant of LSTM layers that does not retain information across batches. For capturing information from past signals but limiting the size of the sequence that gets processed by the LSTM layer, the network incorporates two 1-D convolutional layers with a stride greater than one before the LSTM layer. This leads to a down-sampling of the input sequence, allowing the network to abstract features from a sequence of audio samples. At the end of the model architecture the dense layer processes the output of the last LSTM cell and outputs a single audio sample. The hyperparameters selected for the defined model, HNN-11k, are detailed in Table II.

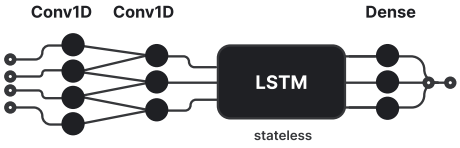

Fig. 4: Schematical representation of the selected hybrid model architecture.

All neural network models in this work were implemented and trained using both major deep learning frameworks, PyTorch and TensorFlow. For each model, an existing implementation in one framework was altered to better fit the use cases of this work. Subsequently, an equivalent model was implemented for the alternative framework. These altered models, along with their equivalent implementations in the alternate framework, are available online [13]. Following this, the TensorFlow models were exported to the TensorFlow Lite format, while the PyTorch models were exported to the LibTorch and ONNX Runtime formats. The TensorFlow models were not exported to the ONNX Runtime format due to the absence of official support for this operation.

TABLE II: Hyperparameters for CNN, RNN, and hybrid models. The variables b, k, c, and d represent the number of TCN blocks, kernel size, number of convolution channels, and dilation factor, respectively, for the CNN models. For the RNN model, h represents the number of hidden units in the LSTM layer. In the hybrid model, s and c denote the stride and number of channels in the convolutional layers, while h represents the number of hidden units in the LSTM layer. The parameter count for each model is also provided. Due to differences in the implementation of the LSTM layer, the parameter number differs for TensorFlow and LibTorch, denoted by * and **, respectively.

Model	b	k	c	d	s	h	Parameters
CNN-29k	4	13	32	10	-	-	29669
CNN-15k	3	13	32	10	-	-	15300
CNN-1k	2	13	32	10	-	-	931
RNN-2k	-	-	-	-	-	20	1781* / 1861**
HNN-11k	-	-	16	-	12	36	10965* / 11109**

B. Real-Time Evaluation

To assess whether the inference engines and *anira* conform to real-time principles, we utilize a code sanitizer to identify potential violations. Code sanitizers function by embedding instrumentation into the program's binary, enabling, for example, the monitoring of memory operations [30]. In this work we are leveraging the Real-time Sanitizer (RadSan) [31], which is tailored to detect common real-time principle violations like memory allocation, deallocation, and thread synchronization.

The test was conducted on a Linux x64 system to determine the quantity and types of real-time violations. For the three inference engines, the inference execution of the models described in Section III-A was monitored over 50 consecutive inferences. To validate the real-time safe operation of *anira*, the library's process method was monitored across different buffer sizes, models and selected inference engines.

C. Operationalization and Datasets

The datasets utilized for the statistical analysis were compiled from a set of benchmarks (cf. Section II-E) that were executed on a variety of operating systems and hardware platforms (*systems*). The first *system* is a MacBook Pro with an Intel Core i9-9980HK processor and 32 GB of RAM, running Arch Linux, kernel version 6.8.4. The second *system* is a MacBook Pro with a M1 processor and 16 GB of RAM, running MacOS 14.4.1. The third *system* is a HP ZBook Fury 16 G9, equipped with an Intel Core i7-12800HX processor and 32 GB of RAM. It runs the Windows 11 operating system. All datasets consist of runtime measurements for processing 50 consecutive buffers, indexed as *iterations*, for specific configurations, repeated identically 10 times (*repetitions*). The measured runtimes are divided by the buffer size in samples to obtain the runtimes per sample (*RpS*).

Configurations differ in the *system* and the inference engine (*engine*), which is either one of the *engines* from Section II-A or a Bypass-Engine. The latter implements the same pre- and post-processing stages as the other *engines*, but writes the input samples directly to the output samples in the inference stage. Other configuration options entail the neural network models as defined in Section III-A. Although the exported

neural network models for LibTorch, ONNX Runtime, and TensorFlow Lite are distinct files with disparate file formats (∗.pt, ∗.onnx and ∗.tflite), the underlying architectures are identical across all three *engines*. For the purposes of statistical analysis, the neural network models have been grouped into categories of similar *models*. The categories are defined in the Table II. Finally, all configurations were benchmarked with *buffer sizes* ranging from 64 to 8192 samples. Despite the *anira* library's capacity to process host buffers, which may have varying dimensions from those required by the inferred *model*, the benchmarks utilized *models* that precisely aligned with the selected *buffer sizes* in the benchmark. This approach was employed to permit an evaluation of the influence of the number of samples inferred simultaneously on the relative time taken to infer one sample.

From these observations, two datasets were created for the main study to ensure complete combinations. Since ONNX Runtime does not support stateful operations [28], the RNN-2k is excluded in the DS-I. Furthermore, only one of the three CNN *models* is included in the DS-I, namely the CNN-29k. The DS-I contains a total of 96,000 *RpS* observations ($N = 96,000$), representing the bottom layer of a seven-layer data structure – $3 \times 4 \times 2 \times 8 \times 10 \times 50$. The initial four layers represent the variables *system*, *engine*, *model*, and *buffer size*, while the fifth and sixth layers correspond to the *repetition* and *iteration* respectively (*system*×*engine*×*model*× *buffer size* × *repetition* × *iteration*). The DS-II is analogous to the DS-I, but it incorporates the RNN-2k and excludes ONNX Runtime. This results in $N = 108,000$ and the form $3 \times 3 \times 3 \times 8 \times 10 \times 50$.

For the follow-up study, which compares the *engine* differences over different hyperparameter combinations of the CNN architecture, the DS-III was created. The DS-III contains only the CNN *models* (CNN-1k, CNN-15k, CNN-29k) and excludes the Bypass-Engine. It has the form $3\times3\times3\times8\times10\times50$ with $N = 108,000$. All measurements were conducted on the *anira* version 0.1.0 and all datasets are publicly available [32].

D. Library Performance

For evaluating the runtime performance of the *anira* library, the DS-II is employed, as it incorporates observations of the Bypass-Engine for the CNN-29k, HNN-11k and RNN-2k. For each of these *models*, a series of descriptive statistics were calculated using the pastecs package [33] in the programming language R [34]. This process involved filtering the dataset for the Bypass-Engine category in the *engine* variable.

E. Statistical Modeling

In order to compare the performance of the *engines*, and in particular to ascertain the effect of the *iteration* and the *buffer size*, we use statistical data analysis to find the best predictors for *RpS* variations obtained in our datasets. Due to the multilevel structure of the datasets, linear mixed effects models (LMMs) are employed for data analysis.

In these models, the *RpS* measure is the dependent variable and the *system*, *engine*, *model*, *buffer size*, and *iteration* are fixed effects. All independent variables are considered factors, including the *buffer size* and *iteration*, since their influence on the *RpS* measures is not necessarily deemed to be linear. Besides the fixed effects, all possible interactions between fixed effects are modeled. To account for *RpS* fluctuations that may arise due to the process scheduling of the operating systems, each *repetition* is given a unique identifier independent of the chosen configuration. The *repetition index*, therefore, represents a specific time window and is included in the LMMs as a random intercept. Since the benchmark process sleeps after every *repetition* (cf. Section II-E) the correlation between the time windows is assumed to be negligible.

For each dataset, a distinct LMM was fitted using the lme4 package [35] in R with maximum likelihood estimation. This results in the statistical models LMM-I for DS-I, LMM-II for DS-II, and LMM-III for DS-III, all of which underwent an analysis of variance (ANOVA) to determine the significance of the fixed effects and their interactions.

Subsequently, the estimated marginal means for selected predictor combinations, including their standard errors (SE) and confidence intervals (CI), were determined with the emmeans package [36], as well as various post-hoc tests (cf. Table III). These include pairwise comparisons to identify significant differences between factor combinations, while accounting for all other variables in the statistical model. To identify significant outliers in the *iteration* predictor, effect contrasts were employed. All p-values are adjusted for multiple comparisons using the Bonferroni-Holm method, as outlined in [37]. Given the size of the datasets, with $N \approx 100,000$, only highly significant p-values are considered in all tests, with a significance level of $p < 0.0001$.

TABLE III: Overview of conducted post-hoc tests on the three different LMMs.

Statistical model	Predictor Combinations	Type
LMM-I, II	*engine*	pairwise
LMM-I, II & III	*engine*, *model*	pairwise
LMM-I & II	*iteration*, *engine*, *model*	effect
LMM-I & II	*buffer size*, *engine*	pairwise

IV. Results

A. Real-Time Safety Tests

The real-time safety tests demonstrate a persistent pattern of real-time violations across all inference engines, neural network models, and inference counts. However, the frequency of these violations varies, as illustrated in Fig. 5.

LibTorch exhibites the highest number of real-time violations, particularly during the initial inferences. Subsequent inferences also show a consistently higher number of violations compared to the other inference engines. In each inference cycle for each model, intensive non-real-time-safe memory operations, such as `malloc`, `free`, and `calloc`, as well as thread synchronization operations using mechanisms like `pthread_mutex_lock` and `pthread_rwlock_rdlock`, are involved. Additionally, the CNN-29k model performs the operation `sleep` and utilises file access functions, such as `fopen`,

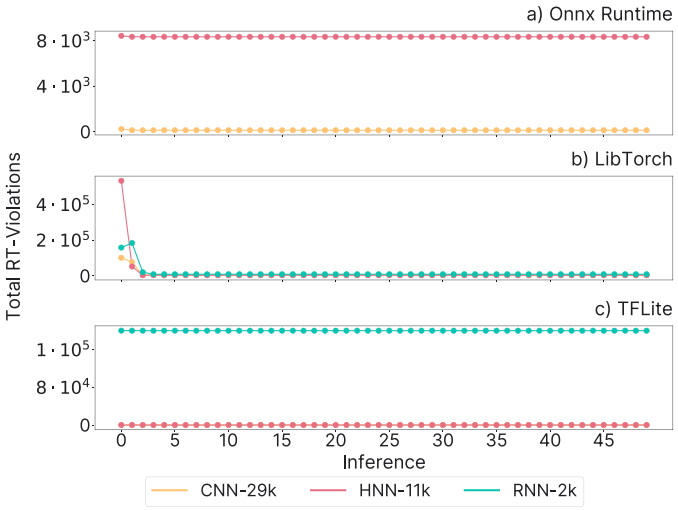

Fig. 5: Total number of real-time violations detected by RadSan for each inference engine, neural network model, and inference count. It is noteworthy that none of the presented graphs reach zero.

during the initial inference. In contrast, TensorFlow Lite's real-time violations are exclusively related to memory operations, specifically `malloc`, `free` and `aligned_alloc`. While the HNN-11k and CNN-29k models demonstrate minimal memory activity, the RNN-2k model exhibites a consistently high number of violations across all inference stages. For ONNX Runtime, the HNN-11k model consistently shows intensive memory usage, while the CNN-29k model exhibits notably fewer violations. The real-time violations observed in ONNX Runtime include `malloc`, `free` and `posix_memalign` operations. Extensive testing of the *anira* library has revealed no violations.

B. Library Performance

In order to assess the runtime performance of the *anira* library across a range of *models*, and consequently different pre- and post-processing steps, descriptive statistics are depicted in Table IV.

TABLE IV: Descriptive statistics of *RpS* observations for the Bypass-Engine across different *models*. The CI employed is 95%. All values are expressed in milliseconds per sample.

model	Mean	SE	CI lower / upper
CNN-29k	$2.03 \cdot 10^{-4}$	$3.46 \cdot 10^{-6}$	$1.96 \cdot 10^{-4}$ / $2.09 \cdot 10^{-4}$
HNN-11k	$3.78 \cdot 10^{-4}$	$3.87 \cdot 10^{-6}$	$3.70 \cdot 10^{-4}$ / $3.85 \cdot 10^{-4}$
RNN-2k	$5.26 \cdot 10^{-5}$	$1.50 \cdot 10^{-6}$	$4.97 \cdot 10^{-5}$ / $5.55 \cdot 10^{-5}$

The results indicate that the RNN-2k exhibits the lowest mean *RpS* value across all *models*, with a mean of $5.26 \cdot 10^{-5}$ milliseconds per sample. This is to be expected, as the RNN-2k requires the least pre- and postprocessing. The values can be compared with the time per sample that an audio application has to process a buffer at a given sample rate. This real-time threshold per sample (*RTT*) is equal to the inverse of the sample rate ($RTT = 1/\text{sample rate}$).

For a sample rate of 48 kilohertz, this results in an *RTT* of 0.0208 milliseconds. Consequently, the processing time of the *anira* library without an actual inference stage is substantially lower than the *RTT* for all *models*.

C. Statistical Models

As a preliminary step in evaluating the significance of the statistical models, the coefficients of determination are examined. The R^2-values presented in Table V indicate that in all three LMMs, the fixed effects and their interactions explain a substantial portion of the variance in the *RpS* observations. Moreover, it is notable that the R^2-*conditional* values are nearly identical to the R^2-*marginal* values in all three statistical models. This indicates that the *repetition index* has only a minimal influence on the *RpS* value, thereby supporting the reproducibility of the benchmarking procedure.

TABLE V: Coefficients of determination for the three LMMs.

Statistical model	R^2-*conditional*	R^2-*marginal*
LMM-I	96.0%	95.9%
LMM-II	95.5%	95.4%
LMM-III	94.9%	94.9%

The influence of fixed effects and their interactions on the *RpS* value is estimated by examining the results of separate ANOVAs for the three LMMs. The analysis reveals that all fixed effects and their interactions significantly predict the *RpS* observations for all three statistical models, always with p-values below 0.0001. Subsequently, the post-hoc tests are employed to delineate precisely which levels of the variables exhibit significant differences and, thus, exert a particular influence.

1) Inference Engine Comparison: The first predictor that is subjected to a closer examination is the *engine*. In both the LMM-I and LMM-II, all contrasts between the *engine* variables are found to be highly significant ($p < 0.0001$). The order from the fastest to the slowest *engine* in the LMM-I is: Bypass-Engine, ONNX Runtime, LibTorch, TensorFlow Lite. In the LMM-II, the contrasts show the same pattern, with TensorFlow Lite, performing worse than LibTorch and the Bypass-Engine being the fastest.

Model-wise comparisons of the *engine* predictor for the CNN-29k and HNN-11k demonstrate that ONNX Runtime consistently outperforms LibTorch and TensorFlow Lite. These comparisons are all highly significant ($p < 0.0001$), with the exception of the difference between ONNX Runtime and TensorFlow Lite for the HNN-11k ($p = 0.045$). The outcome for the TensorFlow Lite versus LibTorch comparison exhibited less consistency. TensorFlow Lite exhibits inferior performance compared to LibTorch for the CNN-29k and RNN-2k, whereas the opposite is true for the HNN-11k. The estimated marginal means and 95% CI for the *engine* and *model* combinations are depicted in Fig. 6.

2) Influence of the Iteration: In light of the varying numbers of real-time violations observed at different inference counts (cf. Section IV-A), it becomes pertinent to investigate whether the *iteration* exerts an influence on the performance of the *engine* for a specified *model*. The effect contrasts of the *iteration* predictor in the LMM-I and LMM-II reveal that

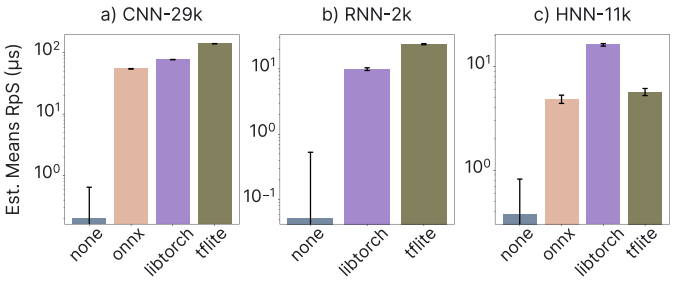

Fig. 6: Estimated marginal means and 95% CI of the *RpS* for the *engine* predictor across different *models*. The values for Subfigure a and c have been computed from the LMM-I, while those for Subfigure b have been derived from LMM-II. The none legend denotes the Bypass-Engine.

this can be answered in the affirmative for certain *engine* and *model* combinations.

In particular, LibTorch exhibits significantly higher *RpS* values for the first *iterations* compared to the average *RpS* across all *models* ($p < 0.0001$). This effect is most pronounced for the CNN-29k, where the first nine *iterations* exhibit significantly higher *RpS* values compared to the average. For TensorFlow Lite, significantly higher *RpS* values are observed only for the CNN-29k, but a tendency towards decreasing *RpS* values after the first iterations can also be observed for the two other *models*. A similar tendency is observed for ONNX Runtime inferencing the HNN-11k. In contrast, the CNN-29k with ONNX Runtime does not exhibit a tendency towards decreasing *RpS* values, but rather a tendency towards altering *RpS* values for consecutive *iterations*.

The Bypass-Engine exhibits extremely high CI values, even crossing the zero line, indicating that the Bypass-Engine has been modeled in a way that makes its *RpS* impossible to discern from zero. Therefore, it is excluded from Fig. 7, which depicts the results of the effect contrasts by displaying the estimated marginal means, averages and CI for the *iteration* predictor *engine*- and *model*-wise.

3) Influence of the Buffer Size: The influence of the *buffer size* on the *RpS* values is analyzed using the LMM-I. The results of the pairwise comparisons for the *buffer size* predictor reveal that almost all *buffer size* pair differences exert a highly significant influence ($p < 0.0001$) on the *RpS* values *engine*-wise. Fig. 8 depicts the estimated marginal means and CI for the *buffer size* predictor. Due to its considerable large CI values, the Bypass-Engine is excluded from the analysis. The results indicate that the *RpS* values decrease with increasing *buffer size* in a remarkably consistent manner across the three *engines*. The impact of the *buffer size* on the *RpS* values is particularly evident at low *buffer sizes* and becomes less pronounced with increasing *buffer size*.

4) CNN Hyperparameter Comparison: In this follow-up study, the performance of the *engines* across various hyperparameter combinations that define the size of the CNN architecture is evaluated. The LMM-III is employed to conduct the required pairwise contrasts for the *engine* predictor *model*-wise. The contrasts indicate that while TensorFlow Lite exhibits significantly inferior performance compared to LibTorch for the CNN-29k ($p < 0.0001$), a different picture emerges when the comparison is conducted at smaller sized *models*. As the size decreases, TensorFlow Lite becomes increasingly faster than LibTorch. This phenomenon reaches a point where TensorFlow Lite no longer exhibits significantly different *RpS* measures from ONNX Runtime at the smallest *model* (CNN-1k). In all other cases, ONNX Runtime outperforms LibTorch and TensorFlow Lite with significant differences. The estimated marginal means and 95% CI for all *engine* and *model* combinations are depicted in Fig. 9.

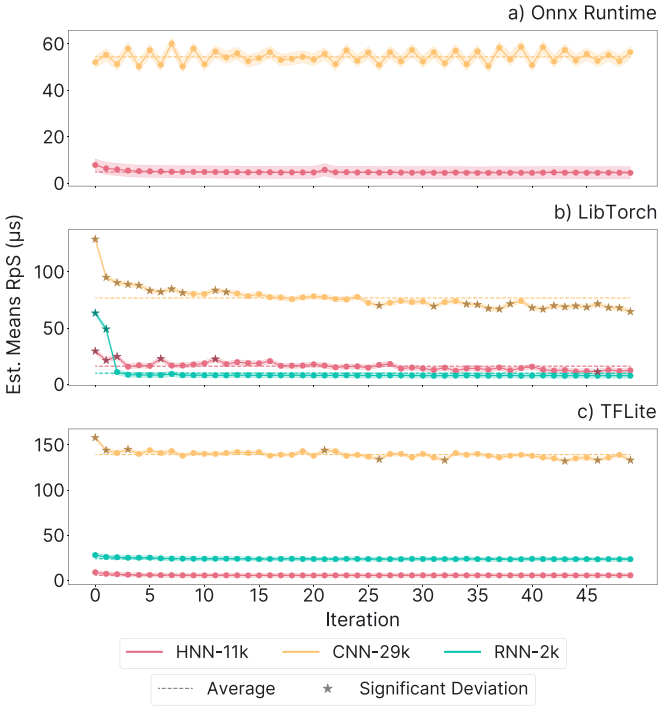

Fig. 7: Estimated marginal means and 95% CI of the *RpS* for different *iterations*, across different *engines* and *models*. The dashed lines represent the average *RpS*, while the stars indicate highly significant differences compared to the average ($p < 0.0001$). For Subfigure a, the values are derived from LMM-I, while for Subfigure b and c, they are derived from LMM-II.

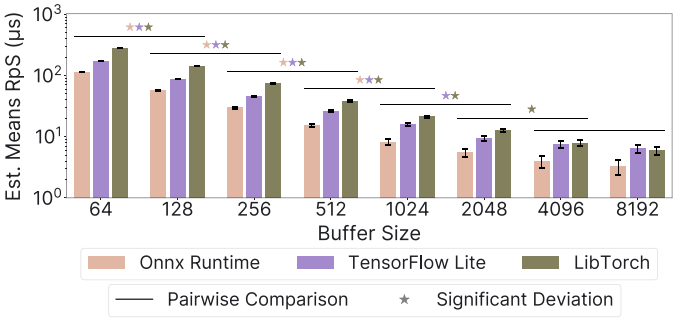

Fig. 8: Estimated marginal means and 95% CI of the *RpS*, calculated from the LMM-I for different *buffer sizes*, across various *engines*. Each *buffer size* is compared pairwise to the next larger *buffer size* of the same *engine*, indicated by the horizontal line above, with stars denoting highly significant differences ($p < 0.0001$).

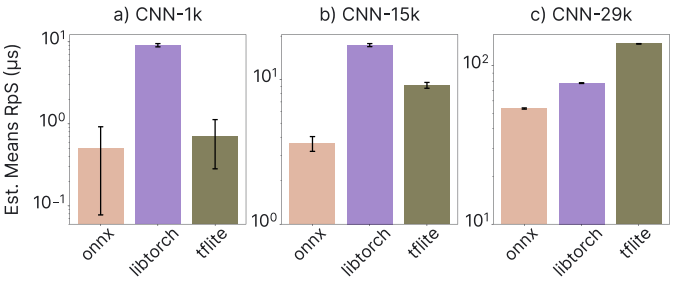

Fig. 9: Estimated marginal means and 95% CI of the *RpS* for the *engine* predictor across different CNN *models*. The values have been computed from the LMM-III.

V. Discussion

This work presents *anira*, a novel cross-platform inference library designed to address the unique challenges of integrating neural network inference into real-time audio applications. The library supports TensorFlow Lite, ONNX Runtime, and LibTorch, which are amongst the most commonly used inference engines. *Anira* and the inference engines were evaluated in terms of real-time safety and performance with regard to inferring three distinct neural network architectures: CNN, RNN, and hybrid networks. In order to assess performance, the integrated benchmarking functionality of *anira* was employed in conjunction with statistical modeling.

As previously observed, the aforementioned inference engines are not entirely real-time safe [11], [12]. By quantifying the real-time violations we discovered their persistence across all engines and models. The LibTorch inference engine exhibited the highest number of violations, likely due to its flexible design. In terms of model architecture, the stateful RNN, possibly due to its internal state, required high amounts of memory allocation. Furthermore, the majority of real-time violation tests demonstrated an increase in occurrences during the initial inferences, although not exclusively. This finding is at contrast to that presented in [12], which indicated that all engines consistently ensure real-time safety after the initial inference. This discrepancy may be attributed to the more complex network architectures that were analyzed in this work and also to the different real-time violation detection method that was employed. While we employed the code sanitizer RadSan [30], which detects real-time violations by intercepting system library calls such as `malloc` and `pthread_mutex_lock` , the authors of [12] relied on monitoring the status of a hard real-time kernel, Xenomai Cobalt.

To circumvent real-time violations, the *anira* library employs a static thread pool to separate the inference from the audio callback. The static thread pool ensures that no potential oversubscription occurs, while its multiple threads allow for parallel inference of incoming data. The evaluation of the *anira* library confirmed its robustness for real-time applications, as it did not exhibit any real-time violations within the audio callback and introduced only minimal runtime overhead per sample. Two different implementations for data sharing and synchronization between the audio callback and the inference threads are provided in *anira*. The choice is left to the user, who may select either an atomic-based or a semaphore-based approach. The atomic-based approach ensures that no system calls are made, whereas the semaphore-based approach accepts minor system calls, contingent on the operating system implementation, in exchange for a further reduction in latency.

The integrated benchmarking capability for evaluating the runtimes of neural networks may be of particular interest for the deployment of audio applications, as it allows for the estimation of the maximum inference time. This measure is an important indicator of the real-time suitability of the network architecture and is a prerequisite for the built-in calculation of latency in the *anira* library. Moreover, the standardized and user-friendly benchmarks have the objective of establishing comparability among neural network inference runtime measurements in real-time audio scenarios. We demonstrated that benchmarks with *anira* can be run for various configurations and created three distinct datasets of runtime measurements.

The statistical modeling of the datasets enabled the explanation of a high proportion of the variations in the runtime observations. The influence of the repetition index as random intercept, which accounts for fluctuations across different time windows in the benchmarks, was found to be minimal. Fluctuations may be attributed to various factors, including process scheduling, since our operating systems were not primarily designed for real-time purposes, allowing other processes to run in the background. Therefore, we conclude that the thread prioritization within the *anira* library is functioning effectively.

Post-hoc tests were employed to identify which levels of the variables exhibited significant differences and their respective impacts. Consequently, it was determined that ONNX Runtime is the fastest engine among all stateless models. When including stateful models in the comparison, which excluded ONNX Runtime, it was found that LibTorch outperforms TensorFlow Lite on average. Furthermore, our investigation revealed significantly longer runtimes in early inferences, particularly for the model and inference engine combinations, which exhibited more frequent real-time violations during the initial inferences. Therefore, we recommend warm up phases before the audio callback to ensure more reliable runtimes. Additionally, when investigating the impact of different model input sizes on per-sample performance, we found that larger model input sizes lead to substantial performance gains. Consequently, for applications with less stringent latency requirements, the use of a larger model input size is advisable.

While ONNX Runtime consistently outperforms the other two inference engines across all model architectures, the results per model were not consistent between TensorFlow Lite and LibTorch. To identify the reasons for this inconsistency, a follow-up study was conducted, in which CNN models with different sizes were compared. The evaluation revealed that TensorFlow Lite is faster for smaller architectures, but as the model parameters increase, LibTorch becomes faster. ONNX Runtime consistently remains the fastest inference engine overall in this comparison.

Although the benchmarks provided valuable insights, there are limitations to consider. The architecture's performance was not evaluated in the context of model input size and host buffer size mismatches. While this approach allows for a more detailed analysis of the influence of the model input size, it does not reflect the circumstances of most audio applications, which typically have varying host buffer sizes. Furthermore, parallel inferences leveraging the thread pool were not benchmarked, which may have limited the assessment of performance.

It is our hope that *anira* will facilitate the integration of neural networks into real-time audio applications. Its flexibility and efficiency make it suitable for a diverse array of real-world applications, including audio plugins, speech processing, embedded audio systems, and smart musical instruments.

Acknowledgment

For their invaluable support and guidance throughout the project, we would like to thank Prof. Dr. Stefan Weinzierl, Prof. Dr. Henrik von Coler, and Dr. Steffen Lepa.

Furthermore, we acknowledge the use of the AI tools ChatGPT, DeepL Write and Github Copilot that have been employed for orthography, punctuation and grammar correction.

References

[1] S. Hershey, S. Chaudhuri, D. P. W. Ellis, *et al.*, "CNN architectures for large-scale audio classification," in *Proc. IEEE Int. Conf. Acoust., Speech, and Signal Process.*, New Orleans, LA, USA, Mar. 5–9, 2017, pp. 131–135.

[2] R. M. Bittner, J. J. Bosch, D. Rubinstein, G. Meseguer-Brocal, and S. Ewert, "A lightweight instrument-agnostic model for polyphonic note transcription and multipitch estimation," in *Proc. IEEE Int. Conf. Acoust., Speech, and Signal Process.*, Singapore, Singapore, Mar. 22–27, 2022, pp. 781–785.

[3] D. Stoller, S. Ewert, and S. Dixon, "Wave-u-net: A multi-scale neural network for end-to-end audio source separation," in *Proc. Int. Soc. Music Inf. Retrieval*, Paris, France, Sep. 23–27, 2018, pp. 334–340.

[4] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "DDSP: Differentiable digital signal processing," 2020. arXiv: 2001.04643.

[5] A. v. d. Oord, S. Dieleman, H. Zen, *et al.*, "WaveNet: A generative model for raw audio," 2016. arXiv: 1609.03499.

[6] A. Caillon and P. Esling, "RAVE: A variational autoencoder for fast and high-quality neural audio synthesis," 2021. arXiv: 2111.05011.

[7] A. Wright, E.-P. Damskägg, L. Juvela, and V. Välimäki, "Real-time guitar amplifier emulation with deep learning," *Applied Sciences*, vol. 10, no. 3, p. 766, Jan. 2020. DOI: 10.3390/app10030766.

[8] M. Abadi, P. Barham, J. Chen, *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX Symp. Operating Syst. Des. and Implementation*, Savannah, GA, USA, Nov. 2–4, 2016, pp. 265–283.

[9] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Conf. Neural Inf. Process. Syst.*, Vancouver, Canada, Dec. 8–14, 2019, pp. 8024–8035.

[10] R. Bencina, "Interfacing real-time audio and file i/o," in *Proc. Australas. Comput. Music Conf.*, Melbourne, Australia, Jul. 9–11, 2014, pp. 21–28.

[11] J. Chowdhury, "RTNeural: Fast neural inferencing for real-time systems," 2021. arXiv: 2106.03037.

[12] D. Stefani, S. Peroni, and L. Turchet, "A comparison of deep learning inference engines for embedded real-time audio classification," in *Proc. Int. Conf. Digit. Audio Effects*, Vienna, Austria, Sep. 6–10, 2022, pp. 256–263.

[13] F. Schulz and V. Ackva, *Anira - an architecture for neural network inference in real-time audio applications*, version 0.1.2, Accessed: Sep. 14, 2024. [Online]. Available: https://github.com/anira-project/anira.

[14] D. Stefani and L. Turchet, "Real-time embedded deep learning on elk audio OS," in *Proc. IEEE Int. Symp. Internet of Sounds*, Pisa, Italy, Oct. 26–27, 2023, pp. 1–10.

[15] T. Pelinski, R. Diaz, A. L. B. Temprano, and A. McPherson, "Pipeline for recording datasets and running neural networks on the bela embedded hardware platform," 2023. arXiv: 2306.11389.

[16] Kitware, Inc, *CMake, the cross-platform, open-source build system*, Accessed: Feb. 14, 2024, 2015. [Online]. Available: https://gitlab.kitware.com/cmake/cmake.

[17] C. J. Steinmetz and J. D. Reiss, "Efficient neural networks for real-time modeling of analog dynamic range compression," in *Proc. 152nd AES Convention*, The Hague, The Netherlands, May 16–19, 2022, p. 10 596.

[18] Tensorflow, *TensorFlow lite*, version 2.16.1, Accessed: Mar. 24, 2024. [Online]. Available: https://github.com/tensorflow/tensorflow/releases/v2.16.1.

[19] Microsoft, *ONNX runtime*, version 1.17.1, Accessed: Mar. 24, 2024. [Online]. Available: https://github.com/microsoft/onnxruntime/releases/tag/v1.17.1 (visited on 03/24/2024).

[20] PyTorch, *LibTorch*, version 2.2.2, Accessed: Mar. 28, 2024. [Online]. Available: https://github.com/pytorch/pytorch/releases/tag/v2.2.2.

[21] A. Carson, A. Wright, J. Chowdhury, V. Välimäki, and S. Bilbao, *Sample rate independent recurrent neural networks for audio effects processing*, 2024. arXiv: 2406.06293.

[22] A. Williams, "Managing threads," in *C++ concurrency in action: practical multithreading*, Shelter Island, NY, USA: Manning, 2012, p. 30.

[23] S. Letz, "Callback adaptation techniques," GRAME - Computer Music Research Lab, hal-02158912, Nov. 1, 2001.

[24] Google LLC, *Google test*, version v1.14.0, Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/google/googletest/releases/tag/v1.14.0.

[25] Google LLC, *Google benchmark*, version v1.8.3, Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/google/benchmark/releases/tag/v1.8.3.

[26] T. Vanhatalo, P. Legrand, M. Desainte-Catherine, *et al.*, "A review of neural network-based emulation of guitar amplifiers," *Applied Sciences*, vol. 12, no. 12, p. 5894, Jun. 2022. DOI: 10.3390/app12125894.

[27] Keith Bloemer, *GuitarLSTM*, Accessed: Feb. 2, 2024. [Online]. Available: https://github.com/GuitarML/GuitarLSTM.

[28] "ONNX runtime architecture," ONNX Runtime. Accessed: Apr. 10, 2024. (Jan. 24, 2023), [Online]. Available: https://onnxruntime.ai/docs/reference/high-level-design.html.

[29] Keith Bloemer. "Neural networks for real-time audio: Stateless LSTM," Medium. Accessed: Jan. 27, 2024. (May 5, 2021), [Online]. Available: https://towardsdatascience.com/neural-networks-for-real-time-audio-stateless-lstm-97ecd1e590b8.

[30] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Ann. Tech. Conf.*, Boston, MA, USA, Jun. 13–15, 2012, pp. 309–318.

[31] D. Trevelyan, A. Barker, and C. Apple, *Realtime sanitizer*, Accessed: Jul. 12, 2024. [Online]. Available: https://github.com/realtime-sanitizer/radsan.

[32] F. Schulz and V. Ackva, *Anira-benchmark-evaluation: Statistical evaluation of benchmarks made with the anira library*, version 0.0.1, Accessed: Jul. 26, 2024. [Online]. Available: https://github.com/anira-project/anira-benchmark-evaluation.

[33] P. Grosjean and F. Ibanez, *Pastecs: Package for analysis of space-time ecological series*, in collab. with M. Etienne, version 1.4.2, Accessed: May. 10, 2024. [Online]. Available: https://github.com/SciViews/pastecs.

[34] R Core Team, *R: A language and environment for statistical computing*, version 4.3.2, Accessed: May. 10, 2024, Vienna, Austria. [Online]. Available: https://www.R-project.org/.

[35] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting linear mixed-effects models using lme4," *J. Statistical Softw.*, vol. 67, no. 1, 2015. DOI: 10.18637/jss.v067.i01.

[36] Russell V. Lenth, *Emmeans: Estimated marginal means, aka least-squares means*, version 1.10.1, Accessed: May. 10, 2024. [Online]. Available: https://CRAN.R-project.org/package=emmeans.

[37] Sture Holm, "A simple sequentially rejective multiple test procedure," *Scand. J. Statist.*, vol. 6, no. 2, pp. 65–70, 1979.