A COMPARISON OF DEEP LEARNING INFERENCE ENGINES FOR EMBEDDED REAL-TIME AUDIO CLASSIFICATION

Domenico Stefani, Simone Peroni and Luca Turchet

Department of Information Engineering and Computer Science
University of Trento
Trento, Italy
domenico.stefani@unitn.it | simone.peroni@studenti.unitn.it | luca.turchet@unitn.it

ABSTRACT

Recent advancements in deep learning have shown great potential for audio applications, improving the accuracy of previous solutions for tasks such as music transcription, beat detection, and real-time audio processing. In addition, the availability of increasingly powerful embedded computers has led many deep learning framework developers to devise software optimized to run pre-trained models in resource-constrained contexts. As a result, the use of deep learning on embedded devices and audio plugins has become more widespread. However, confusion has been rising around deep learning inference engines, regarding which of these can run in real-time and which are less resource-hungry. In this paper, we present a comparison of four available deep learning inference engines for real-time audio classification on the CPU of an embedded single-board computer: TensorFlow Lite, TorchScript, ONNX Runtime, and RTNeural. Results show that all inference engines can execute neural network models in real-time with appropriate code practices, but execution time varies between engines and models. Most importantly, we found that most of the less-specialized engines offer great flexibility and can be used effectively for real-time audio classification, with slightly better results than a real-time-specific approach. In contrast, more specialized solutions can offer a lightweight and minimalist alternative where less flexibility is needed.

1. INTRODUCTION

In recent years, deep learning has become increasingly ubiquitous in various areas of data processing and analysis, including audio and music processing. Some of the sound-related tasks where deep learning has been successfully applied include music tagging [1], beat-detection [2], onset detection [3], instrument classification [4], and, more recently, real-time audio processing [5]. Most of the research in this area has focused on offline learning, where deep learning almost completely replaced traditional machine learning by achieving better accuracy and lower error rates. In contrast, real-time use of deep learning has been explored less. This is due to the added complexity of tight real-time execution-time deadlines, and the fact that neural networks typically take more time to execute than most conventional machine learning counterparts.

Moreover, recent years have seen a growing interest towards deploying deep learning algorithms to real-world applications through embedded computers. This holds for the field of real-time audio, which fostered the development of many embedded platforms for audio, such as the Elk Audio Operating System (Elk Audio OS) [6], Bela [7], or Prynth [8].

However, the requirements of deep learning and conventional frameworks can particularly stress the limited computational resources of even the most recent embedded devices, which is the reason why we have seen an increase in offer in deep learning inference engines (IEs) for embedded devices and single-board computers. In deep learning, the terms “inference engine”, “inferencing library” or “runtime” are used to refer to tools (i.e., code libraries) that can execute pre-trained neural networks.

These IEs include solutions from popular deep learning frameworks, such as TensorFlow, PyTorch, and ONNX. Despite the availability of these tools, it is unclear whether they can safely execute neural network models in real-time audio contexts, where it is crucial to avoid any operation that might slow down or even block the processing of the audio signal. The confusion around these tools led developers to code their specialized approaches to deep learning inference for real-time audio (e.g., RTNeural [9], and [5]). However, specialized approaches tend to be very limited in flexibility, while popular deep learning IEs can, during execution, load a very wide range of neural network models. In addition, it is not clear whether the same exact model can be executed more quickly with an IE than another.

In this paper we present a comparison of four deep learning IEs for real-time audio classification on an embedded CPU, i.e., TensorFlow Lite (From TensorFlow, Google), TorchScript (from Torch/PyTorch, Facebook’s AI Research lab), the ONNX Runtime (from ONNX, Microsoft) and RTNeural [9]. We compare the aforementioned tools in terms of their adherence to real-time-safe programming rules[1] [10]), execution time with multiple neural networks, consumption of computation and memory resources, ease of use, and quality of documentation. We focus on the execution of models on the CPU, which is often the only possibility on off-the-shelf embedded devices because of the general lack of GPUs. The comparison of even more specialized approaches, such as the use of TPUs, DPUs, and FPGAs, is outside of the scope of this study. Every IEs was executed and compared on a Raspberry PI 4 single-board computer paired with Elk Audio OS, which is an open-source and state-of-the-art real-time OS for low-latency embedded audio processing. The Rasberry PI has been widely used in deep learning and it is a platform supported by many inference engine developers, while Elk Audio OS enables high quality and low-latency audio processing thanks to its real-time capabilities. The models used for this comparison were designed to classify eight expressive guitar techniques, where the model output repre-

Copyright: © 2022 Domenico Stefani et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, adaptation, and reproduction in any medium, provided the original author and source are credited.

[1] http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing

sents a prediction of the technique used for each note, in the form of the distribution of probability over all the classes. The code relative to this project was made available in an online repository[2].

The remainder of this paper is organized as follows. Section 2 presents studies that are related to the use of deep learning for audio tasks and embedded computing platforms for audio. The deep learning IEs, metrics, and models involved in the comparison are described in Section 3. Section 4 discusses the results of the comparison. Finally, we draw our conclusions in Section 5.

2. RELATED WORK

Recent years have seen an increase in the interest for deep learning for audio classification and processing tasks. Neural networks have been applied successfully in contexts like onset detection, such as in the work of Eyben et al. [11], where the authors presented a bidirectional Long short-term memory (LSTM) network that was able to surpass previous state-of-the-art scores. A similar neural network was applied to the problems of beat detection and tracking [2], achieving state-of-the-art results. Similarly, Gómez et al. presented a successful approach to instrument classification that uses a convolutional neural network [4]. The authors managed to improve the accuracy of their method through the use of a pre-processing step based on source separation, and a transfer learning approach. However, most of the deep neural networks proposed in research for audio tasks focus on offline inference and are generally unfit for real-time usage, as they can either result in computationally expensive operations (e.g., in [4]) or require non-causal information (e.g., the bidirectional network of [11]).

An interesting approach is described by Sigtia et al. in [12], where a neural network was used for polyphonic transcription of piano performances: while the main solution presented was rather computationally complex, the authors proposed the use of an optimized search algorithm for real-time contexts. Moreover, Bock et al. [3] presented a new version of their offline onset detection model, designed to operate in real-time contexts. More recently, Wright et al. [5] presented an end-to-end neural approach to audio processing, with which authors managed to emulate several distortion pedals and guitar amplifiers. The real-time implementation presented uses the Eigen library and was executed on a desktop computer. This is a very specialized approach that can be help produce very optimized code, but it completely lacks the flexibility of popular deep learning IEs.

These IEs, which include TensorFlow Lite, TorchScript, and ONNX Runtime, can easily load almost any neural network model during execution, without recompiling any code. However, as mentioned in Section 1, there is a general confusion around the compatibility of these IEs with real-time audio applications, which require that the code does not contain any non-real-time-safe operation that can slow down the processing of the audio signal.

For this reason, Chowdhury [9] developed RTNeural, which is a “neural inferencing library” (i.e., deep learning IE) designed to be used for real-time audio applications and similar deep learning tasks that must meet hard time deadlines. The library supports a limited but meaningful range of deep learning operations, and the author stated the intention of implementing more neural network layers. The author compared the performance of the library against the PyTorch C++ API (i.e., TorchScript), but today there is a wider range of available IEs. Moreover, while computation time is crucial for real-time applications, it is fair to compare different IEs also in terms of other metrics, such as how many neural operators are supported, the use of computing resources, memory, general ease of use, and quality of documentation.

Rtneural was successfully used for the implementation of an embedded guitar effect[3] which was successfully deployed on a Raspberry PI running Elk Audio OS [6].

Along with the increase in real-time deep learning approaches for audio classification and processing, there has been a growing number of embedded devices for audio processing. Meneses et al. [13] presented a clear comparison of three open-source embedded audio platforms: *Prynth* [8], the *Bela* framework [7], and a custom processing unit. According to the authors, all the solutions presented different characteristics with no clear winner. More recently, the Elk Audio OS [6] was presented as an open-source real-time operating system for embedded hardware. Similarly to Bela, Elk Audio OS uses the Xenomai Cobalt real-time kernel to handle low latency audio processing, but it is not limited to a single hardware platform and it offers high definition audio inputs and outputs. Vignati et al. [14] compared the performance of the Xenomai Cobalt kernel with that of the more common Preempt RT kernel patch, showing a better overall performance of the former on heavy processing applications.

More recently, Vandendriessche et al. [15] explored the possibilities for hardware acceleration of deep learning inference for audio. This can be achieved with Tensor Processing Units (TPUs), Field Programmable Gate Arrays (FPGAs), and similar hardware. However, these are highly specialized solutions that might not be commonly available across different platforms or infeasible for more practical reasons, such as cost or hard real-time requirements. While we will investigate on these technologies in the future, this paper focuses on CPU inference, which is always possible on both embedded implementations and desktop audio plugins.

In a similar fashion, weight quantization and other model-specific optimizations can help reducing the burden on limited computing devices. A good overview on the limitation of some mobile and embedded devices is presented by Lane et al. [16], who also propose a sparse coding approach that can reduce the use of computation resources. The authors reported on the results obtained on the tasks of speaker recognition acoustic environment classification, which show a significant reduction in mode size. However, these are approximations that reduce the accuracy of deep learning models to an extent that depends on many parameters, including the structure of the target neural network. For this reason, model-specific optimizations are outside the scope of this comparison.

3. METHODOLOGY

This section describes the details of our comparison, which include the deep learning IEs chosen, the benchmark task, the neural networks tested, and the metrics of interest.

This study is meant to compare different deep learning IEs for inference on embedded CPUs. Therefore, the acceleration capabilities of some IEs (e.g., with GPUs and Tensor Processing Units) will not be considered. Moreover, while it is possible to trade model accuracy for quicker execution times with weight quantization, model-specific optimizations are outside of the scope of this comparison

3.1. Inference Engines

The comparison will comprise the following deep learning IEs:

[2] https://github.com/domenicostefani/deep-classf-runtime-wrappers

[3] github.com/GuitarML/NeuralPi/releases/tag/v1.3.0

1. **TFlite** 2.4.1[4]: TensorFlow Lite is the solution offered alongside with TensorFlow (Google) to execute inference of neural network models on embedded devices. The system comprises a converter to produce ".tflite" files starting from models created and trained in TensorFlow, and an Interpreter, which can load TFlite models and run inference.
2. **TorchScript** 1.10.0[5]: TorchScript is the way offered by PyTorch developers to convert models that were trained on a Python environment, to code that can be executed on an environment with no Python dependency.
3. **ONNX Runtime** 1.7[6]: ONNX Runtime is the inference engine provided by Microsoft for ONNX Neural network models. It promises to enable great speedups for both training and inference of neural networks, thanks to its optimization and acceleration features.
4. **RTNeural**[7]: RTNeural [9] is a custom IE, which was developed specifically for audio processing in hard real-time contexts. Contrary to the aforementioned IEs, RTNeural supports only a limited range of neural layers (e.g., MaxPooling and Batch Normalization layers are not supported). However, RTNeural seems a compact and easy-to-use library that could be a strong competitor to the more popular alternatives in some audio processing contexts. Ultimately, we includes RTNeural because the main reason for its development was the confusion around light inference IEs that we are trying to address. RTNeural offers both a dynamic model loading mode (as other IEs) and a compile-time mode. The latter is supposed to reduce the execution time for small networks, according to the documentation of the library.

3.2. Task

Here we compare the performance of four different IEs (see Sec. 3.1) with a series of neural network models for real-time classification of expressive guitar playing techniques, where the model output is the distribution of probability over eight techniques. For this task, the neural network is the last block of an execution pipeline that includes an onset detector and a set of feature extractors (see Figure 1).

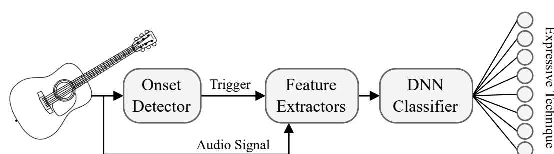

Figure 1: *Expressive guitar technique Classification pipeline.*

When a note onset is detected, the extractors compute a series of timbral features (e.g., MFCC, BFCC) that are fed to the classifier. As a consequence, each model takes as input a one-dimensional vector of features extracted from the first few milliseconds of each note in the audio signal. We focused on 8 categories of expressive guitar techniques, therefore, each classification model has 8 output neurons, where the one with the highest activation value represents the predicted technique. As a consequence, the neural network models compared are all examples of Feed-Forward Neural Networks (FFNNs) with no recursions. The applications of such a classifier impose a computation deadline of 20 ms [17] from the moment that an onset is produced. This is due to the fact that the classification result has to be used to produce new sounds that feel simultaneous with the input sound to the human hearing system, which generally can hardly distinguish complex tones that are separated by less than 30 ms [18]. The more strict deadline of 20ms allows for complex synthesis algorithms that can use the classification result. When compared to tasks with hard real-time constraints such as audio processing, this classification task has a soft real-time deadline which allows us to execute it on a high-priority thread that is separate to the hard real-time processing of the input/output audio signal.

3.3. Models

The models chosen for our comparison are the following:

- **Model A**: the first model is a FFNN composed of four dense hidden layers with 800 neurons each, an input layer with 180 neurons, and a final layer with 8 outputs. Batch normalization was used between each hidden layer with a positive impact on model accuracy. The model comprises a total of 2,083,208 parameters. On the target task, Model A scored an average accuracy of 95.3% across 8 expressive techniques and 5-fold cross-validation.
- **Model B**: Model B is a smaller version of Model A, with six hidden layers of 350 neurons each, an input layer with 173 neurons, and 8 model outputs, resulting in a total of 677,958 model parameters. The lack of Batch Normalization results in a lower accuracy (92.0%). Model B was included to test the performance of the RTNeural framework, which does not support the Batch Normalization layers used in Model A.
- **Model C**: The last model is a drastically smaller version of Model A, with one dense hidden layer with 350 neurons, 173 inputs and 8 outputs, resulting in a total of 63,708 parameters. The accuracy of Model C is lower than Model A and B at just 91.2% for this specific task. Model C was chosen because Model A and B were found to be too large to be executed in the real-time execution thread[8], so only soft real-time constraints can be guaranteed with those models. While this is allowed by our specific task, we offer a comparison that includes the capabilities of each IE to execute models in the real-time thread, which would be required by any model that performs signal processing. The small size of Model C ensures that execution will take less than the time budget between audio interrupts. Model C is used to verify whether the IEs compared here can run without breaking real-time processing constraints (e.g., not allocating dynamic memory or waiting for lower priority tasks and mutexes).

Every model was defined and trained in TensorFlow with the Keras Sequential API and subsequently converted to the formats needed by each IE. The conversion process is described in Section 4.5.3.

[4] https://github.com/tensorflow/tensorflow/releases/v2.4.1
[5] https://github.com/pytorch/pytorch/releases/tag/v1.10.0
[6] github.com/microsoft/onnxruntime/releases/v1.7.0
[7] https://github.com/jatinchowdhury18/RTNeural

[8] The audio plugin created for this task was executed at the rather fast pace of 64 samples at a sample rate of 48 kHz (i.e., 1.33 ms in between audio interrupts) for low-latency audio input and onset detection.

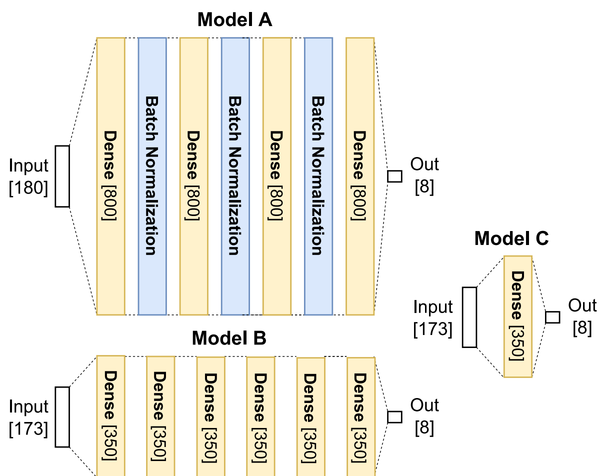

Figure 2: *Neural network models used for the comparison.*

Table 1: *Compatible combinations of models and IEs compared in our analysis. The dynamic-model-load mode (**R.load**) and the compile-time-definition mode (**C.def**) of RTNeural are tested separately. Only Model C is tested by running the IEs in the real-time thread (for the reasons described in Sec. 3.2 and Sec. 3.3), while the remaining are loaded on a separate high priority thread.*

	TFlite	TorchScript	ONNX Runtime	RTNeural	
				[R.load]	[C.def]
MA	✓	✓	✓	✗	✗
MB	✓	✓	✓	✓	✓
MC	✓(RT thread)	✓(RT thread)	✓(RT thread)	✓(RT thread)	✓(RT thread)

3.4. Metrics

Each IE is compared in terms of the following metrics:

1. Real-time safety;
2. Model Execution time;
3. Usage of computation resources (CPU and RAM);
4. Model footprint (file size);
5. Library footprint (library object size);
6. Supported operations;
7. Ease of use;
8. Quality of documentation.

First, we are interested in whether each IE can or cannot execute inference safely on a real-time thread, which refers to the absence of code operations that take an "unbounded" amount of time to complete. Because of the Xenomai hard real-time kernel used by Elk Audio OS, any forbidden operation and system call generates a mode switch, giving back control to the Linux kernel. Mode switches are logged by the system, helping to identify non-safe operations in the real-time thread. For this reason, with Model C each IE is executed in the real-time thread.

Furthermore, we are interested in which IE is quicker at executing the same models, which is of great importance in a real-time context as it limits the minimum latency achievable by the system. Execution time was first measured in an isolated context, by running each IE from the Linux shell, outside of the classification pipeline. This serves as a reliable measure that is independent from many parameters of our classification pipeline, but it lacks to include delays that might be present when deploying the complete pipeline. For this reason, execution time was also measured in the deployed audio classification plugin. In the case of Model A and B, which were executed on a high priority thread separate from the real-time execution, the measurement includes the time needed to schedule the inference and delays due to the real-time audio processing running the foreground.

Additionally, average CPU and RAM usage was measured during the execution of the audio classification plugin thanks to the process status command (`ps`) of Linux and the Xenomai kernel. Moreover, since every Model needed to be converted to a specific format for each IE, we measured the size of the different model-file sizes, which can be relevant with limited storage on some embedded computers.

Metrics 1, 2, and 3 were measured on a Raspberry PI 4 single-board computer (4 GB RAM model). The Raspberry PI board runs the Elk Audio OS (v0.9.0), based on the the Xenomai Cobalt Kernel[9]. Standalone computation times were averaged across 17,604 executions for each combination of model and IE. "Deployment" execution times were averaged across 768 executions, triggered on the classification plugin that was deployed on the target embedded system. The 768 executions were triggered from as many guitar notes in a 26-minutes audio signal that was streamed to the embedded board in real-time for each combination of model and compatible IE. This process was a lengthy operation, hence the reduced number of executions. Additionally, we considered the following four measures that depend only on the different deep learning IEs and not on the models.

Library footprint (library object size): since memory storage can be a constraint on embedded devices, we measured the total size in MiB of the shared or static library objects that are required by each IE. Each library was compiled for the Linux AArch64 architecture (ARM64).

Supported operations: We compiled a list of the most common and more widely used neural layer types and assigned to each IE a score that reflects the fraction of operations supported. In Deep Learning, it is important that as many of the most used operations are supported by IEs, so that a wider range of operations can be used during training. The list of main layers is composed by *Dense*, *Gated Recurrent Unit*, *LSTM*, *1D* and *2D Convolution*, *1D* and *2D MaxPooling*, and *Batch Normalization* layers, while the list of activation types includes: *TanH*, *Sigmoid*, *Softmax*, *ReLU*, *Leaky ReLU*, and *PReLU* activations.

Ease of use: Albeit difficult to quantify, we wanted to include a metric that describes the ease of use of each IE, which includes how easy it was to convert a pre-trained model for each target format, and use the APIs to load a model, read its properties, and execute inference. Two of the authors that worked on the implementation assigned a score from one to ten for each category and the measures were averaged to obtain a final score.

Quality of documentation: the quantity and quality of documentation regarding each deep learning IE. Similarly to ease of use, two of the authors assigned a score from one to ten to each IE, and the two were averaged.

4. RESULTS AND DISCUSSION

This section presents the results of the comparison between the deep learning IEs mentioned in Section 3.1, according to the metrics described in Section 3.4.

[9]https://xenomai.org/

4.1. Real-time safety

The real-time safety capabilities of deep learning runtimes were tested by executing the inference of Model C in the real-time thread while monitoring the status of the Xenomai Cobalt real-time kernel. Interestingly, all the IEs compared here were able to execute multiple inference operations without generating a number of mode-switches that increases at run-time. However, both TorchScript and ONNX Runtime did generate a single mode switch each, on the very first execution of the model.

In the case of TorchScript, the non-safe operation is the allocation of a `std::vector<c10::IValue>` item which happens consistently at the first call of the *forward* function. The solution was to execute a single inference operation on the first execution of the real-time audio processing method. This “priming” operation causes an early allocation of memory in the classifier, where delays that are due to non-safe operations can be acceptable, so that non-real-time-safe operations will not be executed when the first actual classification needs to be made. The same solution applied to ONNX Runtime, where the call to `Ort::Run()` caused memory allocation on its very first call.

4.2. Execution time

Model execution time proved to be different between the various deep learning IEs used, despite the models used being equal. As mentioned in Section 3.4, model execution time was measured both in an isolated context and in an audio plugin running on the real-time Elk Audio OS. Both groups of measures are shown in Figure 3.

The results show that, while most of the IEs offer rather comparable performance in terms of execution time (especially with smaller models), TorchScript is consistently slower than the alternatives. Aside from TorchScript, the two alternatives among the popular IEs (i.e., TFlite and ONNX Runtime) averaged comparable times with Model B, while TFlite slightly prevailed on the smaller Model B and ONNX Runtime worked better with the bigger Model A. The difference for Model A was reduced when running the deployment tests, which indicates that ONNX Runtime could be performing better optimizations on larger numbers of operations.

Finally, the less popular RTNeural showed average execution times that are slightly longer than TFlite and ONNX Runtime, but still very comparable as opposed to the performance of TorchScript. Moreover, RTNeural was tested with both the model loading modalities that it offers: run-time dynamic model parsing and compile-time model definition. Interestingly, the two modalities showed a virtually identical performance in all the tests, while we expected quicker inference with the compile-time model definition, based on the documentation of RTNeural. This can be attributed to the size of the models that we used, which is generally greater than that of the models RTNeural was designed for. It has also to be noted that RTNeural was used with the Eigen backend, which was suggested by the developer for larger networks, but it also supports the use of either xsimd or the C++ STL, which could produce different results.

As expected, the standard deviation of the execution time is negligible for all the standalone execution tests and the execution of Model C in the real-time thread of the deployment application. On the contrary, the worst-case scenario for the execution of Model A and B on the deployment applications is represented by a few cases where the results of the classification reach the real-time thread one or two audio interrupts late. These cases represent a maximum of 6.59% classifications where the results arrive one interrupt later than the average (i.e., 1.33 ms) and 0.37% that arrive two interrupts late (i.e., 2.67 ms). The consistency of these delays shows that they are caused by synchronization issues between the real-time execution and the classification thread, and they have little to do with the deep learning IEs themselves. In the near future we will investigate how to enforce a higher priority for the classification thread, and reduce signaling and synchronization issues through a new mechanism introduced in a recent version of Elk Audio OS (i.e., RTConditionVariable[10]).

4.3. Computational resources

The usage of CPU and RAM was monitored during the execution of the audio classification plugin thanks to the process status command (`ps`) of Linux and the utilities of the Xenomai kernel. All the usage metrics were averaged across 25 minutes and 50 seconds tests, where 768 inference operations were executed. This results in an average of an inference operation every 2 seconds. The results for each test are presented in Table 2.

Table 2: *Usage of CPU and RAM for each combination of model and compatible IE. “Avg.CPU” indicates the CPU usage of the main Linux system (only non-real-time tasks), while “Avg. CpuX” reports the usage of the CPU by real-time tasks in the Xenomai kernel. Average CPU and memory are measured with the* `ps` *command. The percentage measures are relative to the embedded system described in Section 3.4.*

Model	Inference Engine	Avg. Cpu	Avg. CpuX	Avg. RAM
MA	TFlite	8.3 %	6.1 %	5.1 %
	TorchScript	9.0 %	5.8 %	8.7 %
	ONNX Runtime	10.7 %	6.1 %	6.4 %
MB	TFlite	8.0 %	5.9 %	5.0 %
	TorchScript	8.6 %	5.9 %	8.6 %
	ONNX Runtime	9.6 %	5.9 %	5.8 %
	RTNeural(R.time)	8.4 %	5.8 %	5.7 %
	RTNeural(C.time)	8.6 %	5.8 %	5.6 %
MC	TFlite	6.1 %	6.2 %	4.7 %
	TorchScript	5.3 %	5.8 %	7.6 %
	ONNX Runtime	5.1 %	5.8 %	5.2 %
	RTNeural(R.time)	5.2 %	5.8 %	4.7 %
	RTNeural(C.time)	5.1 %	5.8 %	4.7 %

The results indicate that such a low frequency of classification operation generates a low *average* use of computational resources. However, the difference in resource usage between different IEs and models could scale to higher frequency tasks, such as audio processing. In particular, TorchScript consistently scores the highest memory consumption percentage, which is coherent with the inference time results (see Sec. 4.2). Moreover, for both Model A and B, ONNX Runtime shows higher CPU usage, on the standard Linux kernel, than the alternatives. All the IEs present a higher usage of CPU from operations running in the non-hard-real-time domain for both Model A and B, which is to be expected since inference for these two models is executed on a non-real-time thread. On the contrary, CPU usage for all the IEs on Model C is higher on the Xenomai Kernel operations, since Model B is executed in the audio processing thread.

Overall, the average use of CPU is demonstrated to not be an interesting metric for low-rate audio classification, while the execution time measured in the previous section presents a better overall image of processing performances. Finally, the increase of

[10] https://github.com/elk-audio/twine

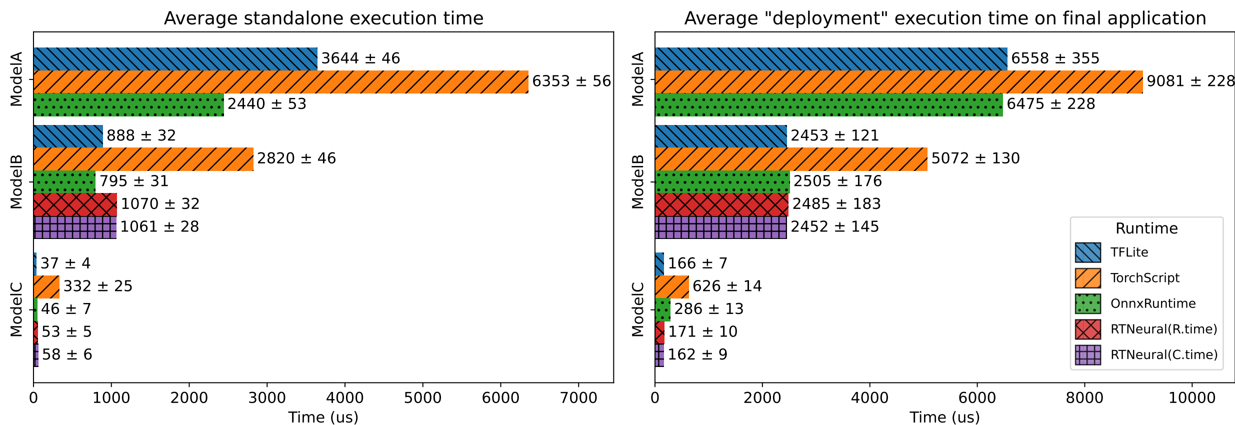

Figure 3: *Mean and standard deviation of model execution time in microseconds, for each combination of model and compatible IE. The image on the left shows the measures performed in an isolated context, while that on the right includes the "deployment" execution time (see Sec. 3.4).*

RAM consumption from Model C to B and A shows to be minimal, despite the drastically different model sizes. This is mainly due to the rather large amount of ram available with the last iteration of the Raspberry PI single-board computer (i.e., 4 GB), which is a testament to the technological advancement of modern embedded computers.

4.4. Model footprint

When converting the original Keras models to the formats accepted by the various IEs, we noticed some differences in the final file sizes, which might be critical on target devices with little memory, such as less recent Raspberry PI boards. TFlite, TorchScript, and ONNX Runtime use compressed model formats that result in very similar file sizes. On the contrary, RTNeural uses the JSON format, which is a human-readable storage format for neural networks, which results in a significantly large file for each model. The results are shown in Figure 4.

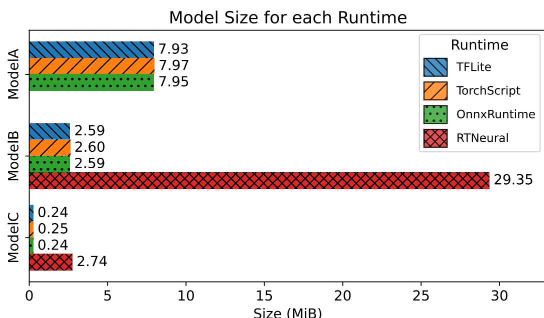

Figure 4: *Size of Model A, B and C when converted for each IE.*

Additionally, a test conversion of only the supported operations of Model A (i.e., no batch normalization) results in a final model file with a size of 94.9 MiB. Nevertheless, since RTNeural is an open-source project, any developer could integrate serialization and deserialization primitives to obtain lightweight models.

4.5. Additional model-independent metrics

The last four measures are *Library size*, *Supported operations*, *Ease of use*, and *Quality of documentation* (see Sec. 3.4). These are independent of the use of either Model A, B, or C. The resulting scores are presented in Table 3 and Figure 5 and will be discussed in the next sections.

Table 3: *Model-independent metric results. All the scores are expressed on a scale from one to ten, with ten being the most desirable score. The Library size score is inversely proportional to the actual size, with ten being the smallest library (RTNeural) and zero being the largest (TorchScript).*

Inference Engine	Library size (score)	Supported Operations	Ease of use	Quality of doc.
TFlite	9.25	**10**	**9.75**	**9.5**
TorchScript	0	**10**	7.75	4.5
ONNX Runtime	8.98	**10**	7	8
RTNeural	**10**	5.7	6.5	3.5

4.5.1. Library Size

Library size refers to the total size of the C++ libraries of each IE. The resulting sizes are presented in Table 4, which shows that TorchScript has a considerably large code library, TFlite and ONNX Runtime are similar in size, and RTNeural is several orders of magnitude smaller than both. However, these measures must be considered along with the size of the models presented previously.

Table 4: *Size of the C++ library objects for each IE. The version of each IE is specified in Section 3.1.*

Inference Engine	Library	Size (MiB)	Total (MiB)
TFlite	libtensorflow-lite.a	8.8	8.8
TorchScript	libtorch_cpu.so libc10.so	116.8 0.5	117.3
ONNX Runtime	libonnxruntime.so	12.0	12.0
RTNeural	libRTNeural.a	0.026	**0.026**

From Figure 6 we can see that the rather compact size of the TorchScript models is undercut by its large code library. Furthermore, even if the size of the RTNeural library is so low to even be practically invisible in the plot, its large uncompressed JSON models are a significant disadvantage. Additionally, a virtual projection of the size of Model A (which is currently unsupported) would be

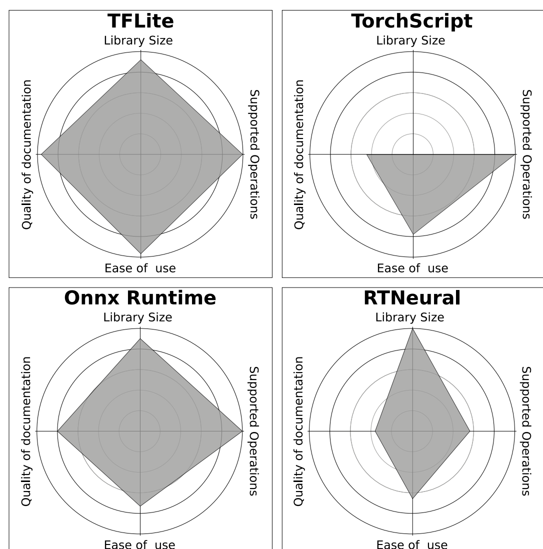

Figure 5: *Graphical representation of the model independent scores for each IE. The four spokes on each graph graph represent respectively the score assigned to the size of the IE library, the number of supported operations, the perceived ease of use and the quality of documentation.*

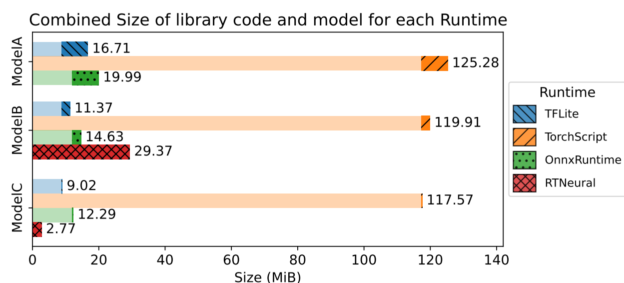

Figure 6: *Combined sizes of models and C++ code libraries for each compatible combination. The lighter-color part of each bar shows the size of each code library, while the remaining part indicates the size of the model.*

north of 94 MiB, making the total size closer to the rather large TorchScript than the other alternatives. However, implementing new serialization and deserialization functions for RTNeural (see Sec. 4.4) would greatly reduce the total footprint because at the current state it depends almost entirely on the model size.

4.5.2. Supported Operations

The amount of **supported operations** is presented as the percentage of operations available in each IE from a list of the most common types of neural layers and activations (see Sec. 3.4). According to their documentation, the more advanced TFlite, TorchScript, and ONNX Runtime cover 100% of the operations listed. RTNeural is instead more limited, as it only supports network layers that are *Dense*, *Gated Recurrent Units*, *LSTM* cells, *1D Convolutions*, and the following activations: *TanH*, *Sigmoid*, *Softmax*, and *ReLU*. RTNeural totals a coverage of 57% of the most common neural network operations. Among the operations missing in RTNeural at the moment, the most crucial are *2D convolutions*, *MaxPooling* and *Batch Normalization*.

However, despite supporting over 50% of the most common operations, the code library of RTNeural is over 300 times smaller than that of TFlite and over 450 times smaller than ONNX Runtime. Along with the execution-time results, which are comparable with the most popular alternatives, this shows a great deal of care towards simplicity and code bloat avoidance.

4.5.3. Ease of Use

Ease of use was rated by two of the authors, according to the perceived complexity of converting a neural model and using the APIs of each IE to load the model, obtain its proprieties (e.g., input and output sizes) and execute the inference. It is to be noted that the scores assigned to the ease of model-conversion are dependent on the fact that the starting point was a regular Keras/TensorFlow model.

The conversion to a TensorFlow Lite model is straightforward since the developers provide a Python tool (i.e., TFLiteConverter) that allows the user to convert a SavedModel, a Keras model, and concrete functions. On the other hand, generating a TorchScript model from a TensorFlow model requires a custom implementation of a TensorFlow-PyTorch converter. Fortunately, the two frameworks represent the most basic layers in a relatively similar way, allowing for a simple conversion of the data types of one library to the other. Once a PyTorch model is obtained, the JIT API provides two ways to generate a TorchScript model: tracing and scripting. Tracing is performed when the computational graph is inferred by recording the operations executed on a sample input, while scripting creates the TorchScript model by analyzing the source code, therefore being a better choice for more complex models (e.g., including conditional statements). In a similar fashion, PyTorch allows users to perform tracing to generate an ONNX model, which is a very simple process. Finally, RTNeural provides a Python script to export the weights of a TensorFlow Model to a JSON file, but it required refining to discard layers that are not needed for inference (i.e., dropout layers).

The utility developed to convert TensorFlow models for each IE is available in the project's repository[11], along with wrappers for each IE, which expose the same API to allow for easy IE interchangeability.

4.5.4. Quality of documentation

The quality of documentation comprises the clarity and quantity of guides, tutorials, and formal API documentation. TFlite showed the best documentation, composed of very user-friendly guides and thorough API documentation. ONNX Runtime was the second-best in the category: the API documentation is complete and detailed, and there is a good number of examples, but it lacks the number of tutorials offered with TensorFlow Lite. TorchScript was instead very different: the technical documentation is very scarce for a project of such entity, and we had to rely on a few incomplete guides. Finally, RTNeural has a very compact amount of information on how to use the IE, which is to be expected by a project of its size. Generally, the use of RTNeural was intuitive, which reflects on the scores assigned in the previous section.

4.6. Key Takeaways

Each IE proved to be safe for real-time inference, with the appropriate code practices. Moreover, In terms of model execution speed, TFlite, ONNX Runtime and RTNeural proved to be the

[11] https://github.com/domenicostefani/deep-classf-runtime-wrappers

quickest, with mostly comparable results, while TorchScript was considerably slower. Interestingly, we found that the compile-time definition mode of RTNeural does not offer any significant speedup with the models tested.

The average CPU and memory Usage was a less insightful metrics for a rather low-frequency task, but the results helped to confirm the difference in resource consumption between neural network models. Moreover, except for RTNeural, all the IEs use a compressed format for neural networks, which results in model files that are considerably smaller than the human-readable representation used by RTNeural. However TorchScript has a significantly larger code library than all the alternatives.

All the popular IEs analyzed support a wide range of neural layers and activation functions. In contrast, RTNeural lacks crucial types of neural layers like *Batch Normalization*, *MaxPooling*, and *2D Convolutions*. However, despite supporting 57% of the most common neural operators, the code library of RTNeural is several orders of magnitude smaller than the competition. Finally, TFlite and ONNX Runtime were deemed to be the easiest to use and have the most detailed documentation.

5. CONCLUSIONS

In this paper, we presented a comparison of four inference engines for real-time audio classification on embedded CPU. Our aim was to shed some light on optimized inference engines for deep learning inference and their properties in relation to real-time audio classification. In our study, we employed models for real-time classification of expressive guitar techniques. We found that many popular deep learning inference engines can be used effectively for real-time audio classification, without needing to resort to more limited and specialized solutions, such as RTNeural. In contrast, more specialized solutions can be lightweight and minimalist alternatives where less flexibility is needed. While we focused on embedded computers and audio classification, most results are likely to translate or scale to audio plugins for desktop computers, and audio processing. The limitations of this study are in the choice of restricting the comparison to Feed-Forward Neural Networks and only four deep learning inference engines. Besides exploring more inference engines, future work should also investigate performance differences with a wider range of deep learning models, such as recurrent and convolutional neural networks. Other possibilities would be to extend this comparison to slower CPUs and to test with quantized neural network models.

6. REFERENCES

[1] K. Choi, G. Fazekas, and M. Sandler, "Automatic Tagging Using Deep Convolutional Neural Networks," in *Proc. 17th Int. Society for Music Information Retrieval Conference (ISMIR)*, August 2016, pp. 805–811.

[2] S. Böck and M. Schedl, "Enhanced beat tracking with context-aware neural networks," in *Proc. 14th Int. Conf. on Digital Audio Effects (DAFx-11)*, 2011, pp. 135–139.

[3] S. Böck, A. Arzt, F. Krebs, and M. Schedl, "Online real-time onset detection with recurrent neural networks," in *Proc. 15th Int. Conf. on Digital Audio Effects (DAFx-12), York, UK*, 2012.

[4] J. S. Gómez, J. Abeßer, and E. Cano, "Jazz Solo Instrument Classification with Convolutional Neural Networks, Source Separation, and Transfer Learning," in *Proc. 19th Int. Society for Music Information Retrieval Conference, (ISMIR)*, 2018, pp. 577–584.

[5] A. Wright, E.-P. Damskägg, L. Juvela, and V. Välimäki, "Real-time guitar amplifier emulation with deep learning," *Applied Sciences*, vol. 10, no. 3, 2020.

[6] L. Turchet and C. Fischione, "Elk Audio OS: an open source operating system for the Internet of Musical Things," *ACM Transactions on the Internet of Things*, vol. 2, no. 2, pp. 1–18, 2021.

[7] A. McPherson and V. Zappi, "An environment for submillisecond-latency audio and sensor processing on beaglebone black," in *Proc. AES 138th Convention, Warsaw, Poland*, 2015.

[8] I. Franco and M. M. Wanderley, "Prynth: A framework for self-contained digital music instruments," in *In Proc. 12th Int. Symposium on Computer Music Multidisciplinary Research (CMMR)*, 2016, p. 357–370.

[9] J. Chowdhury, "Rtneural: Fast neural inferencing for real-time systems," *arXiv preprint arXiv:2106.03037*, 2021.

[10] R. Bencina, "Interfacing real-time audio and file i/o," in *Proc. of the Australasian Computer Music Conference (ACMC)*, 2014, pp. 21–28.

[11] F. Eyben, S. Böck, B. Schuller, and A. Graves, "Universal onset detection with bidirectional long-short term memory neural networks," in *Proc. 11th Intern. Soc. for Music Information Retrieval Conference, ISMIR, Utrecht, The Netherlands*, 2010, pp. 589–594.

[12] S. Sigtia, E. Benetos, and S. Dixon, "An end-to-end neural network for polyphonic piano music transcription," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, pp. 927–939, 2016.

[13] E. Meneses, J. Wang, S. Freire, and M. M. Wanderley, "A comparison of open-source linux frameworks for an augmented musical instrument implementation," in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Porto Alegre, Brazil, June 2019, pp. 222–227, UFRGS.

[14] L. Vignati, S. Zambon, and L. Turchet, "A comparison of real-time linux-based architectures for embedded musical applications," *Journal of the Audio Engineering Society*, vol. 70, no. 1/2, pp. 83–93, 2022.

[15] J. Vandendriessche, N. Wouters, B. da Silva, M. Lamrini, M. Y. Chkouri, and A. Touhafi, "Environmental sound recognition on embedded systems: From fpgas to tpus," *Electronics*, vol. 10, no. 21, 2021.

[16] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, "Squeezing deep learning into mobile and embedded devices," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017.

[17] D. Stefani and L. Turchet, "Bio-Inspired Optimization of Parametric Onset Detectors," in *Proc. 24th Int. Conf. on Digital Audio Effects (DAFx20in21)*, Sept. 2021, vol. 2, pp. 268–275.

[18] B. C. J. Moore, *An introduction to the psychology of hearing*, Brill, 2012.