Model-Structured Neural Networks on Edge Hardware: A Preliminary Study

Giovanni Maria Francesco La Scala, Sebastiano Taddei, Francesco Baroni,
Gioele Defrancesco, Filippo Faccini, Gastone Pietro Rosati Papini

***Abstract*— Deploying neural networks on edge hardware is essential for real-time robotics but remains challenging under tight resource constraints. Model-structured neural networks (MSNNs), while compact and effective, have been less explored in terms of performance on embedded platforms. We evaluate different deployment strategies for a MSNN for vehicle dynamics on a Raspberry Pi 5, comparing C++, ONNX, and PyTorch implementations. Results show that standard approaches designed for large-scale networks are outperformed by non-parallel, hand-optimized C++ implementations.**

I. Introduction

Autonomous robotic systems increasingly rely on deploying machine learning models on edge hardware to meet strict latency, power, and reliability constraints. In particular, microcontrollers (MCUs) play a key role in enabling low-latency control loops, removing dependency on network connectivity, and allowing tight integration with onboard sensing and actuation. Despite being ubiquitous, low-cost, and energy-efficient, MCUs impose severe resource limitations: they typically provide less than 1 MB of RAM and only a few megabytes of Flash memory, often without floating-point units or hardware accelerators [1], [2].

While significant progress has been made in optimizing standard deep learning architectures—such as convolutional and transformer-based networks—for embedded platforms, less attention has been given to model-structured neural networks (MSNNs), a class of models that explicitly incorporate domain knowledge into their architecture.

MSNNs are characterized by heterogeneous, layer-specific computational blocks that often reflect underlying physical models or algorithmic priors. This structure can improve interpretability, data efficiency, and generalization. Notably, MSNNs have already demonstrated strong performance in several robotics domains, particularly in vehicle dynamics modeling and control, where they have achieved high accuracy and improved generalization [3], [4], [5]. Furthermore, this class of models is often inherently compact due to its structured design, which already minimizes redundant parameters, reducing the need for conventional compression techniques such as quantization, pruning, and model distillation. However, this introduces computation patterns that differ from the highly parallel workloads typically targeted by modern accelerators and neural network libraries, and are not yet fully characterized. As a result, a clear understanding of the most effective deployment strategies for MSNNs on embedded platforms is still lacking.

In this preliminary study, we investigate the performance implications of deploying MSNNs on edge hardware. We compare different implementation and deployment strategies on a representative platform, a Raspberry Pi 5, using a MSNN for vehicle lateral dynamics control. In particular, we evaluate multiple deployment approaches, including native C++ implementations, ONNX-based execution, and PT inference via LibTorch. Key metrics include inference latency, RAM usage, and Flash footprint. Our goal is to provide initial insights into the deployment challenges of MSNNs on edge platforms and to motivate further research at the intersection of model-based machine learning and hardware-aware optimization.

II. Model Definition

The proposed MSNN implements a lateral vehicle controller in which the desired steering, δ_k, is obtained as the superimposition of two contributions (see Fig. 1). The first, $K_i(\cdot)$, is a Finite Impulse Response (FIR) filter operating in closed-loop on a window of past steer angles to reconstruct the internal state. The second, $K_t(\cdot)$, is a local model that computes the steering required to match the desired curvature, combining understeer correction and dynamic compensation through an additional FIR filter. Understeer effects are modeled via a fuzzy weighting function of a_x, which computes a weighted combination of understeer correction terms $\boldsymbol{A}$ defined over different acceleration levels, while dynamic behavior is captured by a second fuzzy weighting that blends the outputs of the FIR filters as a function of vehicle speed. The model is an extension of the one proposed in [4]. The overall network comprises 514 trainable parameters and has been validated in a verified simulation environment. The model is implemented using the *nnodely* framework, an open-source library available on GitHub [6]. It is trained on a workstation and subsequently deployed on the embedded system for evaluation.

III. Experiment

The experimental evaluation aims to compare the computational performance of different implementations of the same lateral control MSNN: ONNX Runtime in C++, optimized ONNX in C++, TorchScript executed through LibTorch (PT

The authors are with the Department of Industrial Engineering, University of Trento, Via Sommarive 9, 38123 Povo (TN), Italy (email: gastone.rosatipapini@unitn.it).

This work was supported under the FIS 2 Call, Grant Assignment Decree No. 1236 adopted on 01/08/2023 by the Italian Ministry of University and Research (MUR), for the project FIS-2023-03684 'Structured neural network framework for modeling and control of autonomous systems – Neu4mes', CUP E53C24003800001.

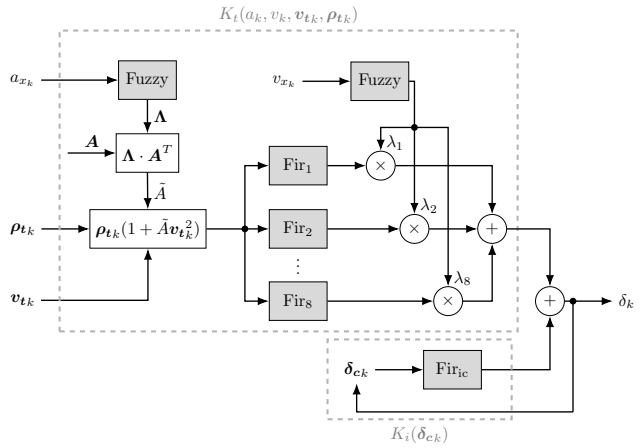

Fig. 1. Block diagram of the vehicle steering controller NN. Inputs $\boldsymbol{v_{t_k}} = [v_{t_k}, \ldots, v_{t_{k+n-1}}]$ and $\boldsymbol{\rho_{t_k}} = [\rho_{t_k}, \ldots, \rho_{t_{k+n-1}}]$ are future windows of target speed and curvature; a_x and v_k are the current acceleration and speed; $\boldsymbol{\delta_{ck}} = [\delta_{k-n}, \ldots, \delta_{k-1}]$ is a window of past steering angles. $\boldsymbol{A}$ is a vector of understeer correction terms. The output is the steering angle δ_k.

C++), and a native C++ implementation. The comparison focuses on inference latency, under identical input conditions, RAM usage, and Flash footprint.

All implementations rely on the same dataset processing pipeline to ensure comparability. The dataset consists of multiple CSV files that contain recorded vehicle signals. In the C++ micro-benchmarks, each timed iteration is one full forward pass of the controller, with batch sizes ($N \in 1, 100, 1000$) (each iteration process N samples). For each valid time index t, the generated inputs include vehicle speed v_x, road curvature history, longitudinal acceleration a_x, and steering history.

Performance measurements are obtained using *Google Benchmark* [7], a C++ micro-benchmarking library that provides precise measurements of execution time. For each iteration, the controller processes a batch of sizes $N = 1$, $N = 100$, $N = 1000$; Figure 2 shows the results for the executed tests reporting the execution times in logarithmic scale (measured in [ms]) with their standard deviation.

IV. Results

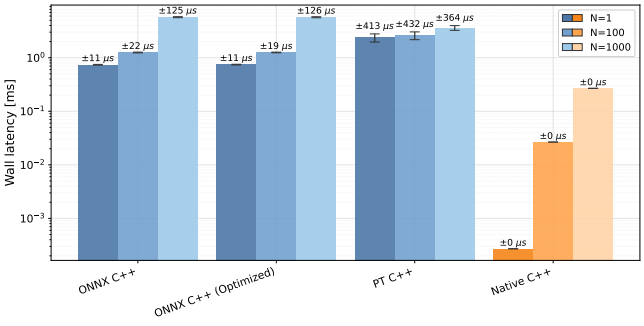

Fig. 2. Comparison of end-to-end inference latency across C++ implementations (ONNX Runtime, ONNX Runtime optimized model, PyTorch C++, and native C++), reported as mean wall-clock latency in milliseconds.

The four C++ inference pipelines considered in this study were evaluated using Google Benchmark with 1000 repetitions with different batch sizes. Table I shows RAM usage and Flash footprint for the proposed approaches.

Implementation	RAM [MB]	Flash	
		Binary [MB]	Dyn. libs [MB]
ONNX C++	49.496	0.500	2.711
ONNX C++ Optimized	49.758	0.500	2.711
PT C++	168.903	0.575	310.190
Native C++	23.216	0.498	2.508

TABLE I

Memory footprint comparison across implementations

The native C++ implementation achieves the lowest latency, confirming that a carefully optimized low-level implementation is substantially more suitable for latency-critical deployment than framework-based alternatives. Motivated by this result, a preliminary analysis was conducted to further investigate the extent to which compiler-level optimizations can improve native backend performance. The results of this analysis are portrayed in Figure 3.

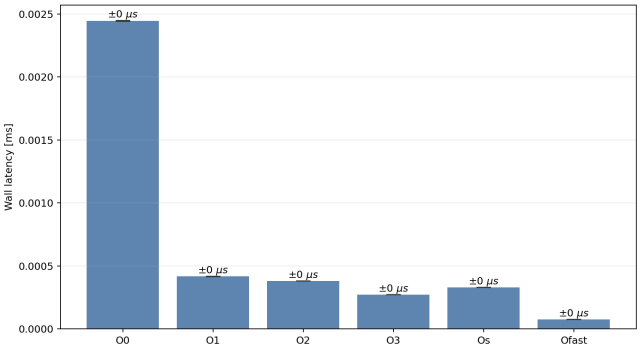

Fig. 3. Wall-clock latency comparison across native C++ compiler optimization levels reported in microseconds. The plot shows a clear performance improvement as the compiler optimization level increases: `-Ofast` provides the best configuration, with a mean latency of approximately 0.0736 μs, compared to 0.2709 μs (`-O3`) and 0.3808 μs (`-O2`). In relative terms, `-O3` and `-O2` are approximately 3.68× and 5.17× slower than `-Ofast`, confirming that the compiler optimization level has a substantial impact on the performance of the native backend.

V. Conclusions and Future Work

The findings suggest that conventional deployment frameworks for neural networks, such as ONNX and PyTorch, do not achieve optimal performance in the case of MSNNs. This is mainly due to the limited data parallelism inherent in these models, which reduces the effectiveness of standard optimization strategies. In contrast, direct native implementations in C++ show superior performance.

Future work will extend this analysis to different hardware platforms (such as FPGA, microntrollers) and MSNNs of increasing complexity. In addition, we will investigate the trade-offs between onboard computation and cloud-based inference by comparing energy consumption and latency for different MSNNs. The goal is to identify configurations that provide the best balance between real-time performance and energy efficiency in latency-sensitive control applications.

REFERENCES

[1] J. Lin, W.-M. Chen, Y. Lin, j. cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 11 711–11 722. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/86c51678350f656dcc7f490a43946ee5-Paper.pdf

[2] M. Bechtel, Q. Weng, and H. Yun, "Deeppicarmicro: Applying tinyml to autonomous cyber physical systems," 2022. [Online]. Available: https://arxiv.org/abs/2208.11212

[3] M. Da Lio, M. Piccinini, and F. Biral, "Robust and sample-efficient estimation of vehicle lateral velocity using neural networks with explainable structure informed by kinematic principles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 12, pp. 13 670–13 684, 2023.

[4] M. Da Lio, R. Donà, G. P. Rosati Papini, F. Biral, and H. Svensson, "A mental simulation approach for learning neural-network predictive control (in self-driving cars)," *IEEE Access*, vol. 8, pp. 192 041–192 064, 2020.

[5] M. Da Lio, D. Bortoluzzi, and G. P. Rosati Papini, "Modelling longitudinal vehicle dynamics with neural networks," *Vehicle System Dynamics*, vol. 58, no. 11, pp. 1675–1693, 2020. [Online]. Available: https://doi.org/10.1080/00423114.2019.1638947

[6] G. P. Rosati Papini, "nnodely," https://github.com/tonegas/nnodely, 2026.

[7] Google, "Google benchmark," 2025, version 1.9.4. [Online]. Available: https://github.com/google/benchmark