Results in Engineering 30 (2026) 110194

Contents lists available at ScienceDirect

Results in Engineering

journal homepage: www.sciencedirect.com/journal/results-in-engineering

Research paper

Leveraging programmable logic controllers for machine learning applications in industrial setups

David Zamora-Arranz [a], Pablo Garcia-Bringas [a], Juan J. Gude [a,*], Javier Del Ser [b,c]

[a] University of Deusto, Bilbao, 48007, Bizkaia, Spain
[b] TECNALIA, Basque Research and Technology Alliance (BRTA), Derio, 48160, Bizkaia, Spain
[c] Department of Mathematics, University of the Basque Country (UPV/EHU), Leioa, 48940, Bizkaia, Spain

ARTICLE INFO

Keywords:
PLC
Machine learning
Artificial intelligence

ABSTRACT

Machine Learning (ML) has become a powerful tool for addressing complex classification and regression tasks in industrial settings. Despite the widespread use of programmable logic controllers (PLCs) in these environments, ML models are typically executed on external computing devices due to inherent PLC limitations: restricted memory capacity, long cycle times, and limited computational power. These constraints hinder the direct deployment of ML algorithms on PLCs and often require additional hardware, increasing system complexity and deployment costs. This paper addresses this challenge by demonstrating the direct implementation of four ML algorithms on a Siemens S7-1516 PLC: Linear Regression, Logistic Regression, k-Nearest Neighbors (kNN), and a Neural Network. In addition, a real-world laboratory prototype modeling the thermal behavior of an industrial 3D printer is presented to illustrate the practical applicability of our research findings. To overcome PLC resource constraints, two optimization strategies are proposed: (1) algorithmic adaptations to reduce execution cycle time, and (2) a custom, high-performance matrix multiplication library designed to replace the naive implementation common in standard tools. We evaluate the performance of the proposed implementations across datasets of varying sizes, comparing the standard TIA Portal functions to our optimized library. The results demonstrate that that the optimized implementations achieve acceptable cycle times and consistently outperform the baseline, confirming the feasibility of efficient, native ML execution on PLCs. These findings open new avenues for embedding ML capabilities directly into PLC-based automation systems, enabling smarter and more autonomous industrial control.

1. Introduction

Historically, industrial control systems relied on electrical circuits composed of relays, switches, and other hardware elements to execute logic operations. This changed in 1969 with the introduction of the first programmable logic controller (PLC), the MODICON 084, developed by Dick Morley in response to General Motors' need for a more flexible, software-driven solution. PLCs eliminated the need for extensive rewiring when modifying control logic, significantly reducing physical complexity and facilitating easier reconfiguration. Since then, PLCs have become foundational to modern manufacturing, offering programmable, reliable, and scalable automation solutions.

Over the decades, PLCs have evolved into the primary control devices in industrial environments. They enable precise control of both continuous and discrete processes by managing signals to and from actuators, sensors, motors, indicator beacons, electro-valves, electro-pneumatic and electro-hydraulic systems, conveyor belts and other elements. Their support for proportional-integral-derivative (PID) control loops further enhances their ability to regulate complex processes. By replacing hard-wired logic with software-configurable logic, PLCs offer considerable advantages in terms of programming, maintenance, and scalability, making them indispensable in automation systems.

Despite the pace of technological advancement, legacy PLCs remain in widespread use due to the challenges associated with replacing them. Many industrial processes require near-continuous uptime, and updating control hardware often involves halting production and undergoing rigorous testing. As a result, numerous factories still rely on older PLC models (some dating back several decades), such as the Siemens S5 or early S7 series (such as the S7-200, S7-300, and S7-400 models). These legacy devices are difficult to replace yet still capable of being reprogrammed. Integrating modern computational capabilities, such as Machine Learning (ML), into these existing PLC infrastructures presents

* Corresponding author.
E-mail addresses: david.zamora@opendeusto.es (D. Zamora-Arranz), pablo.garcia.bringas@deusto.es (P. Garcia-Bringas), jgude@deusto.es (J.J. Gude), javier.delser@tecnalia.com (J. Del Ser).

https://doi.org/10.1016/j.rineng.2026.110194
Received 10 January 2026; Received in revised form 26 February 2026; Accepted 19 March 2026
Available online 22 March 2026

a promising opportunity to enhance their functionality and extend their lifespan without complete system overhauls.

In parallel, Artificial Intelligence (AI) (which includes ML algorithms) has emerged as a transformative force in industrial domains, driving the shift toward smarter, data-driven operations. This is evidenced by a substantial body of research, and the current research landscape indicates that AI plays a crucial role in modern automation industries [1,2]. The adoption of AI for industrial applications from defect detection to product classification, predictive maintenance and logistics optimization enable industries to reduce operational costs, enhance efficiency, and make informed decisions with minimal human intervention. Within the Industry 4.0 paradigm, AI plays a critical role in leveraging data from interconnected systems to support autonomous, adaptive, and intelligent control strategies across multiple industrial domains [3,4].

ML in industrial contexts is typically deployed on external computing devices with processing capabilities and resources suited to deal with the demands of the training and interference stages of ML pipelines. However, this integration imposes additional hardware costs, increases system complexity, and introduces latency between data collection and decision-making. Directly implementing ML logic on PLCs could mitigate these issues, enabling in-situ learning and inference, fewer potential points of failure, reduced energy consumption, and allowing for real-time adaptation to changing process conditions.

High-performance PLCs, such as Siemens' S7-1500 series, offer a promising platform for natively embedding ML capabilities. PLC programming follows the IEC 61131-3:2003 standard [5], which defines the basic software architecture and programming languages for control programs within PLCs. This standard specifies five programming languages: Ladder Diagram (LD), Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL), and Sequential Function Chart (SFC). The updated IEC 61131-3:2013 version [6] introduced new data types, conversion functions, references, namespaces, and object-oriented programming features such as classes and function blocks. Although not all manufacturers fully comply with this standard (particularly with older PLC models), many modern PLCs use programming languages that completely or closely adhere to these specifications, making code adaptation between PLCs, whether from the same or different manufacturers, generally feasible even for the older ones.

Despite its potential, research into the implementation of ML algorithms on PLCs remains limited and often lacks reproducible results. Prior work has explored isolated use cases, but has not systematically addressed the constraints and potential of ML in this context. In this paper we aim to bridge this gap by demonstrating that deploying ML algorithms on industrial PLC devices is not only feasible but also practical, without external hardware support. This advancement has the potential to transform PLCs from deterministic logic executors into intelligent agents capable of performing sophisticated control and prediction tasks. By addressing this identified gap, the contribution of our manuscript can be summarized as follows:

- We implement four widely used ML algorithms – Linear Regression, Logistic Regression, k-Nearest Neighbors (kNN), and a Neural Network – directly on a Siemens S7-1516 PLC using Structured Text (ST).
- We propose two optimization strategies to address PLC limitations: minimizing cycle time through tailored algorithm adaptations and enhancing matrix computation via a custom high-performance matrix multiplication library.
- We conduct a comparative evaluation of ML models implemented with standard TIA Portal functions versus our optimized approach, using datasets of varying sizes to assess their scalability and performance.
- We demonstrate, as a central contribution of our research, that ML models can achieve acceptable execution performance on PLCs, supporting their direct integration into real-time industrial control systems without the need for external computing hardware.

The rest of this manuscript is organized as follows. Section 2 reviews related work in ML deployment on PLCs and embedded systems. Section 3 briefly pauses at concepts related to PLC that are needed to properly understand the optimization strategies proposed in the manuscript. Section 4 details such strategies, whereas the ML algorithms implemented by embracing them are described in Section 5. Section 6 presents experimental results and their discussion. Section 7 further validates our research results through the use of one of our implemented ML algorithms (linear regression) on a real-world use case. Finally, Section 8 concludes the paper and outlines directions for future work.

2. Related work

This section reviews the state of the art in applying ML within Programmable Logic Controller (PLC) environments, with a focus on how models are adapted to the computational constraints of industrial hardware. Starting from Wegmann's seminal work [7], which first explored the integration of neural networks and fuzzy logic into PLC-based applications, subsequent research has proposed a range of approaches that differ in the role assigned to the PLC. Such prior studies can be categorized based on the role of the PLC in ML training and inference. These categories are as follows: (1) *embedded ML systems*, where the PLC is solely responsible for both training and inference processes; (2) *hybrid systems*, in which inference is performed on the PLC while training occurs externally, facilitated through data exchange with a workstation or PC; and (3) *external systems*, where the PLC does not perform training nor inference, but instead serves as a communication interface, exchanging data with an external workstation or PC that performs all computational tasks.

The section is organized to first introduce relevant developments in TinyML and the complexity of deploying ML models on PLCs (Subsection 2.1). We then discuss the practical implications of different deployment scenarios, from full on-device learning (Subsection 2.2) to inference-only (Subsection 2.3) or interface-only roles (Subsection 2.4). Prior strategies for overcoming resource limitations are also reviewed (Subsection 2.5). The section concludes by highlighting the novelty of the present work (Subsection 2.6).

2.1. TinyML and the challenges of deploying machine learning on PLCs

The rapid proliferation of the Internet of Things (IoT) has fundamentally reshaped the deployment landscape of artificial intelligence (AI), pushing it beyond centralized computing environments into the realm of low-power, edge-based devices. In this context, the emergence of embedded ML and the paradigm of TinyML have been instrumental in enabling the execution of AI models on devices with highly constrained computational resources. This paradigm demonstrates that tasks traditionally requiring substantial memory and processing capabilities – typically reserved for personal computers (PCs) or cloud infrastructure – can now be effectively carried out on lightweight platforms such as field-programmable gate arrays (FPGAs), computer-on-modules (COMs), and single-board computers (SBCs).

As emphasized in the comprehensive survey by Murshed et al. [8], TinyML offers considerable advantages, including reduced energy consumption, lower hardware costs, and smaller physical footprints, making it well-suited for pervasive and embedded applications. Nonetheless, these benefits come with significant challenges, including limited memory, restricted computational throughput, and a lack of mature development environments optimized for AI on such platforms. Despite these limitations, there have been notable successes. For example, Kumar et al. [9] developed Bonsai, a compact tree-based algorithm for IoT devices, capable of running on hardware with only 2 KB of RAM, 32 KB of flash memory, and no floating-point support, while maintaining competitive prediction accuracy. Similarly, Gupta et al. [10] introduced ProtoNN, a resource-efficient variant of the K-Nearest Neighbor (KNN) algorithm,

implemented on an Arduino UNO, which achieved near state-of-the-art results with minimal storage and memory requirements.

These developments are particularly relevant to industrial environments, where programmable logic controllers (PLCs) are widely deployed to manage mechanical and electronic processes. Integrating ML capabilities into PLCs aligns naturally with the TinyML philosophy: bringing intelligence closer to the process to reduce latency, eliminate the need for external computation units, and improve robustness. The potential benefits include reduced hardware integration complexity, decreased energy usage compared to conventional computing solutions, and avoidance of costly system downtime typically associated with adding new external hardware.

However, deploying ML models directly on PLCs remains a complex endeavor due to several inherent limitations. First, PLCs are not designed to manage or store large datasets; they operate in real time and prioritize deterministic execution over data storage or complex computations. Their memory capacities are significantly smaller than those of PCs, making it infeasible to load or process large training datasets locally. Second, their computational power is limited. PLCs typically lack parallel processing capabilities and have slower CPUs, which poses challenges for executing even moderately complex ML models. Third, and perhaps most critically, PLCs do not natively support high-level programming frameworks for ML. Developers are often forced to implement algorithms at a low level, translating mathematical formulations into structured logic manually, which increases development time and the potential for implementation errors.

Although some early efforts attempted to address this gap (e.g. the now discontinued NeuroSystems software, which supported fuzzy logic and neural network inference on the Siemens S7-300 and S7-400 PLC families), these tools have not kept pace with the broader evolution of ML. A more modern solution is the Siemens TM NPU module, designed for the S7-1500 series and the ET 200MP distributed I/O system. This module is tailored for inference-only tasks and supports pre-trained models developed in TensorFlow or Caffe, which must be converted using the OpenVINO™ toolkit for compatibility. Internally, the NPU leverages the Intel®Movidius™ Myriad™ X Vision Processing Unit (VPU), which includes a dedicated Neural Compute Engine and 16 SHAVE cores, optimized for on-device inference and computer vision workloads [11].

As a result, any attempt to implement training on a Siemens PLC requires manually encoding the learning process, often involving direct translation of gradient descent or optimization routines into PLC logic blocks. This is a labor-intensive and error-prone process, making it impractical for most industrial applications. Consequently, while TinyML has enabled meaningful AI applications on constrained devices in consumer and IoT contexts, its translation to PLC environments remains partial and underdeveloped. The industrial ecosystem still lacks a standardized framework for deploying and maintaining ML models on PLC hardware. In this paper we show that this gap can be effectively bridged without relying on external hardware.

2.2. PLC performing both training and inference

Building on the limitations discussed in the previous section, executing both training and inference directly on PLCs represents the most demanding and least explored scenario within the spectrum of ML deployment in industrial controllers. Due to severe hardware and software constraints (including limited memory, lack of parallel operations support, the criticality of cycle time, and the absence of native ML libraries), realizing full on-device learning on PLCs remains rare and highly challenging. Nevertheless, a few studies have attempted to push the boundaries of what is computationally feasible in this context.

Körösi et al. [12] proposed a method that enables bidirectional neural network creation between MATLAB and a PLC using XML-based data exchange. While their primary focus lies in facilitating the communication process, the authors report deploying a neural network on a Schneider Electric PLC. However, the study does not specify the PLC model, the training methodology, or the network's performance characteristics, and the validation was conducted using simulated data. Similarly, Li et al. [13] implemented a neural network with backpropagation for PID tuning on a simulated Siemens S7-1500 PLC, where both training and inference were carried out. Although the approach demonstrates theoretical feasibility, the reliance on simulation rather than physical hardware limits its practical relevance, especially considering the known discrepancies between simulated and real-time PLC environments. A more implementation-focused example is provided by Mahmoud et al. [14], who deployed a neuro-controller (NC) on an unspecified PLC to control a DC drive. Prior to deployment, the network was simplified through PC-based simulation by reducing neuron count, minimizing input dimensionality, approximating the sigmoidal activation function with a linear one, and using an adaptive learning factor. The resulting feedforward network was then implemented on the PLC and tested under constant and adaptive learning rates.

While these studies demonstrate that training and inference on PLCs are theoretically possible, none provide a complete account of the experimental setup. Crucial details such as the specific PLC model, network structure, parameter configurations, and execution metrics on physical hardware are either missing or insufficiently reported. This lack of transparency hinders reproducibility and limits the ability to assess the real-world viability of fully embedded ML pipelines in PLC-based systems.

2.3. PLC performing only inference

Given the considerable complexity and resource demands of executing both training and inference on PLCs, as outlined in the previous section, a more practical and commonly adopted approach involves performing only the inference phase on the PLC, while offloading training to an external system. This strategy enables the deployment of pre-trained models on PLCs, maintaining real-time decision-making capabilities while circumventing the computational overhead associated with model training.

One of the earlier examples of this approach is provided by Topalova et al. [15], who used a multilayer perceptron (MLP) neural network for object image recognition. The model was trained externally on a PC using Siemens NeuroSystems software and then transferred to a Siemens S7-300 PLC for deployment. In a follow-up study, Topalova et al. [16] employed a similar setup for texture classification of marble shades, again using a pre-trained MLP network deployed on a Siemens S7-317 PLC. This methodology was extended in Topalova's later work [17], where an adaptive algorithm for marble plate classification was implemented using a trained MLP that was also transferred to the same PLC platform. Tzokev et al. [18] also demonstrated this paradigm in a marble production context by designing a hybrid system consisting of a PC and a PLC. The neural network, trained using Siemens NeuroSystems, was offloaded to the PLC to perform real-time image and data processing tasks. More recently, Dedy et al. [19] pursued a different strategy, implementing a segmented linear regression method for estimating mass from vapor pressure in saturated vapor systems. The approach involved five distinct linear formulas embedded in the PLC to perform direct inference, showcasing a lightweight alternative to neural networks for regression tasks. Another recent example by Duymazlar et al. [20] involved training an artificial neural network using backpropagation in MATLAB's Neural Network Toolbox. The trained model was converted into Structured Control Language (SCL) code via Simulink and then integrated into a Siemens S7-1200 PLC as a function block. This setup was used to control a nonlinear, multivariable system, highlighting the feasibility of deploying inference-ready ANN models on mid-tier PLC hardware. More recently, Doumanidis et al. [21] introduced the ICSML framework, a ML inference framework designed for industrial control system (ICS) environments. The framework is implemented in *Structured Text* and supports interoperability with all languages defined in the IEC 61131-3 standard. Its

main goal is to provide ML engineers with a reusable codebase and a structured development methodology that enables the efficient design, deployment, and execution of cross-platform, real-time ML solutions on programmable logic controller (PLC) hardware. The authors demonstrate a real on-PLC ML application implemented with ICSML, highlighting its effectiveness through a case study in which a defense mechanism is deployed to mitigate process-aware attacks on a Multi-Stage Flash (MSF) desalination process.

For context, the Siemens SIMATIC S7-317-2 PN/DP, used in several of the aforementioned studies, executes bit operations in approximately 25 ns. More modern PLCs, such as the S7-1516-3 PN/DP used in this study, can perform bit operations in just 10 ns, providing a substantial performance gain that further supports inference-only ML deployment strategies on PLCs.

2.4. PLC not performing training nor inference

While the previous section focused on scenarios where inference is executed locally on the PLC, another common approach entirely offloads both training and inference to external systems. In such configurations, the PLC acts solely as a data acquisition and control interface, while all ML computations are performed on external devices such as PCs or servers. This setup is often preferred in industrial applications where the computational demands of ML models exceed the capabilities of available PLC hardware, and where real-time constraints can be met through high-speed data communication.

Although this study aims to investigate the feasibility of embedding ML logic within PLCs, it is important to acknowledge this alternative architecture as a widely used baseline in industrial settings. Consequently, we do not provide an extensive literature review here but highlight a few representative examples to contextualize the landscape. In this regard, Jung et al. [22] used the PLC exclusively for data monitoring and acquisition. The collected data were exported to MATLAB, where both training and inference were carried out using an artificial neural network. Similarly, Fonseca et al. [23] implemented an industrial neural network controller in MATLAB, with the PLC serving only to manage operational tasks. In both cases, the PLC functioned as a bridge between the physical process and the external ML system, rather than as an active computational component in the ML pipeline. This architecture, while practical and widely adopted, stands in contrast to the objective of this study, which is to explore the integration of ML logic within the PLC itself.

2.5. Strategies to overcome PLC constraints for ML

As discussed in the previous sections, only a handful of studies have explored the direct implementation of ML algorithms within PLC environments, particularly those that include the training phase (typically the most computationally demanding component). Despite this, no widely adopted or systematic methods have emerged to address the core limitations of PLCs, such as strict cycle time requirements and the computational inefficiency of operations central to ML algorithms, particularly matrix multiplications.

Li et al. [13] explicitly noted the lack of support for vectorized operations in Siemens' TIA Portal environment, which significantly hinders the efficient implementation of neural networks. Likewise, Mahmoud et al. [14] emphasized both the limited memory available and the absence of parallel processing capabilities, which collectively restrict the scalability and responsiveness of ML models executed on PLC hardware. The framework proposed by Doumanidis et al. [5] enables the partitioning of computations across multiple scan cycles, thereby allowing for reduced scan cycle durations. The authors further suggest that leveraging platform-specific libraries for optimized vector and matrix operations could potentially decrease inference latency even further. These observations align closely with the challenges addressed in our work and provide additional context for the methodological advances presented in Section 2.6.

A few attempts have been made to address these challenges. Mahmoud et al. [14], for instance, proposed simplifying the activation functions using piecewise linear approximations and reducing execution time by disabling unused inputs. While such measures can marginally improve execution speed and reduce resource usage, they come at a cost. Linearizing activation functions can alter model behavior and degrade performance, and disabling inactive inputs constitutes a basic optimization rather than a comprehensive solution. Additionally, it is also suggested to minimize the number of neurons per layer and use an adaptive learning factor, but these proposals also constitute basic optimizations without any novel contribution. Overall, these approaches underscore the need for new techniques to deploy ML algorithms on industrial PLCs without compromising the fidelity and robustness of ML models.

2.6. Contribution

Building on the limitations and prior strategies discussed in the previous sections, this study contributes a novel and practical approach to implementing ML algorithms on Siemens PLCs that includes both training and inference capabilities. Addressing the critical challenges related to cycle time constraints and the limited efficiency of essential mathematical operations, our work introduces two key advancements: first, we develop a custom matrix multiplication library tailored to the PLC execution environment. This library significantly improves upon the performance of standard, naive implementations by optimizing for the architectural constraints of Siemens PLCs, particularly addressing slow execution due to the lack of vectorization and parallel processing support. Second, we introduce an iterative slicing technique that partitions the execution of algorithms across multiple cycles. This mitigates the impact of prolonged execution times on the PLC scan cycle and ensures compliance with real-time constraints. Together, these contributions provide a foundational methodology for extending the applicability of ML models in industrial automation contexts. By enabling both training and inference directly on PLCs while maintaining computational feasibility and preserving model behavior, this work addresses a critical gap in the intersection of TinyML, real-time systems, and industrial control.

From the practical point of view, enabling both inference and training directly on PLCs significantly improves the practical applicability of ML in industrial environments. Performing training on-site, where the data are generated, removes the need for external data collection or off-device model preparation, thereby simplifying the workflow and supporting rapid retraining when required. Since PLCs inherently acquire and manage process data, this capability allows training to be carried out locally, reducing latency, system complexity, and potential failure points. Such advantages are relevant in applications like that of Doumanidis et al. [21], which uses sensor and actuator variables for anomaly and process-aware attack detection. Beyond these specific applications, similar data could also be leveraged for fault anticipation through behavioral analysis. In a wider context, industrial ML deployments may experience shifts in data distribution over time, a theoretical yet realistic possibility in practice. Such changes could occur in image-classification scenarios, as explored by Topalova et al. [16,17], or from operating-point variations in control systems, as discussed by Jun Li et al [13]. While these works do not explicitly report such changes, their potential occurrence underscores the importance of enabling rapid retraining. In the use case presented in this work, the operating conditions may also vary, making retraining beneficial for re-adjusting the model parameters toward optimal values. As suggested by the authors, leveraging platform-specific libraries for optimized vector and matrix operations could potentially decrease inference latency even further. In this work, we demonstrate how our implementation of matrix multiplication is specifically tailored to maximize performance on the target hardware, validating that the use of platform-specific optimizations leads to significant performance gains.

3. Preliminaries

Before proceeding with the description of the proposed approach, we define key concepts related to PLC program execution, including cycle time (Subsection 3.1), computational load and program execution speed (Subsection 3.2), PLC memory (Subsection 3.3) and locality of reference (Subsection 3.4). Understanding these concepts is crucial for understanding the practical benefits of the present study.

3.1. Cycle time

PLC code execution is cyclical, with the cycle time defined as the duration required to complete one full iteration of the code. This cycle includes updating the process image of inputs and outputs, executing program instructions, and handling any system activities that may interrupt the cycle. In a Siemens S7-1500 PLC, each cycle comprises the following steps, in order: updating the process image of inputs, executing the cyclic program, and updating the process image partition of outputs. The process image is a memory area where the states of physical inputs and outputs are mapped, allowing the program quick access to this data during execution.

Cycle time varies based on several factors, including the size of the process image and the number and complexity of instructions executed within the program. For example, a program that performs 10 addition operations will have a shorter cycle time than one performing 1000 additions, all else being equal. Similarly, mapping a small number of inputs and outputs in the process image will have a smaller effect on cycle time than mapping a large number.

Cycle time is critical in industrial processes, as it determines the frequency of program execution. If the cycle time is too long, the program may not execute frequently enough to meet the control demands of the process, potentially leading to suboptimal or unsafe outcomes. For this reason, minimizing PLC cycle time is advisable. Exceeding the maximum allowable cycle time during execution causes the PLC to enter the Stop state, halting all tasks. In the Siemens S7-1516 model, the maximum configurable cycle time is 6 seconds.

3.2. Computational load and program execution speed

The program execution speed in a PLC depends on several factors, primarily the number of instructions and the computational load associated with them. Repetitive loops, such as WHILE or FOR loops, can substantially impact cycle time, as they prevent the cycle from completing until all instructions within the loop have executed and the program reaches the cycle's end. Additionally, not all mathematical operations have the same computational cost; for instance, multiplications and divisions are more computationally intensive than additions or subtractions.

A particularly demanding operation in ML applications is matrix multiplication, which is computationally expensive. In the Siemens S7-1500 PLC family (except for the S7-1518, which provides an additional second independent runtime environment in order to execute C/C++ applications in parallel to the STEP 7 program) and the S7-1200, S7-400, and S7-300 families, parallel computation is not possible, imposing further constraints on execution speed. Due to these limitations, optimizing matrix multiplication performance is of utmost importance for effectively implementing ML algorithms on Siemens PLCs.

3.3. PLC memory architecture and access performance

Efficient implementation of ML algorithms on PLCs requires a detailed understanding of the memory architecture and access speeds, particularly in time-critical applications. In Siemens S7-1500 PLCs, memory is structured into four primary areas: load memory, work memory, retentive memory, and additional memory areas [24]:

- *Load memory*: This non-volatile memory, located on the SIMATIC memory card, stores code blocks, data blocks, technology objects, and hardware configuration. In the S7-1500 series, the load memory reside on an external SIMATIC memory card. When data is transferred from the computer to the PLC it is initially stored in the load memory.
- *Work memory*: This volatile memory, integrated within the CPU, consists of code work memory and data work memory and cannot be expanded. The code work memory stores the runtime-relevant parts of the program code, while the data work memory holds the runtime-relevant parts of data blocks and technology objects
- *Retentive memory*: This non-volatile memory enables the storage of data throughout program execution, with the distinctive capability of retaining data in the event of a power-off or power failure. The available amount of retentive memory is limited.
- *Additional memory areas*: Additional areas store bit memories, timers, counters, process images, and temporary local data. Temporary local data is of particular note due to its high-speed access, making it especially suitable for time-critical operations. However, like retentive memory, its capacity is limited, restricting its use for large data storage.

Memory access time is heavily influenced by how data is stored and accessed. A critical distinction exists between standard and optimized data blocks [25]:

- *Standard vs. optimized blocks*: Standard blocks (used for compatibility with S7-300/400 PLCs) follow the Big Endian format, while optimized blocks (default in S7-1200/1500 PLCs) use the Little Endian format. Optimized blocks support larger sizes (up to 16 MB vs. 64 KB) and offer faster access. For instance, accessing a 4-byte REAL value in a standard block may require two 16-bit reads with byte reordering, whereas optimized blocks support direct 32-bit access. Moreover, optimized blocks eliminate byte-level locking during bit access by assigning each bit to a separate byte. Tag arrangement in optimized blocks is also type-aligned and gap-free, which enhances processor efficiency.
- *Local memory*: Local memory consists of static and temporary tags. Static tags are used when values must persist across cycles. Temporary tags, having faster access times, are ideal for values used only within a single cycle. For frequently accessed inputs/outputs, using a temporary tag as an intermediary can improve runtime performance.
- *Non-optimized blocks*: These blocks result in slower access times due to inefficient data alignment and the need for additional processing steps.
- *Arrays and indexed access*: Indexed accesses such as using a variable index (e.g., `Motor[i]`) introduce runtime overhead. Performance can be improved by declaring index variables as `DINT` types and storing them in temporary tags within the local memory area.

Understanding and leveraging these architectural and performance characteristics is essential for optimizing memory operations when deploying computationally intensive tasks (e.g., matrix multiplications and iterative ML updates) within the cycle-time constraints imposed by PLCs.

3.4. Locality of reference

A processor typically exhibits a tendency to access certain memory locations repeatedly within a short time frame. This concept, known as *locality of reference*, is crucial for optimizing memory access efficiency. Denning formally established the foundational principles underlying temporal and spatial locality in [26], although he did not employ these terms explicitly in the original exposition. While often discussed in the context of personal computers, locality of reference is also highly relevant in PLCs, which similarly include a CPU and memory. Two primary types of locality of reference can be found:

- *Temporal Locality*, hinging on the premise that if a memory location is accessed at a given time, it is likely that the same location will be accessed again in the near future. To enhance the speed of subsequent

accesses, a copy of the referenced data is stored in the fastest storage memory.
- *Spatial Locality*, which is based on the premise that if a memory location is accessed at a given time, it is likely that nearby locations will be accessed shortly thereafter. Therefore, to expedite future accesses, copies of these adjacent memory locations are stored in the fastest storage memory.

4. Proposed approaches to implement ML algorithms on PLC devices

As outlined in previous sections, implementing ML algorithms on PLCs (particularly those involving both training and inference) requires overcoming severe hardware constraints, including limited memory, restricted computational power, and real-time cycle time requirements. Many of these constraints manifest during low-level operations that are fundamental to ML workloads, such as matrix multiplications, loop-based iterations, and memory access.

This section presents a series of approaches designed to address these constraints through low-level optimization techniques. In Subsection 4.1 we first propose slicing algorithm execution across PLC cycles to prevent overrun conditions and maintain real-time responsiveness. Next, in Subsections 4.2 and 4.3 we explore how optimizing memory access patterns through the use of row-major ordering and loop interchange can significantly reduce access latency and improve cache utilization. Subsection 4.4 examines the impact of array size declaration on execution performance, showing how explicitly defined dimensions can lead to internal optimizations. Finally, in Subsection 4.5 we apply loop unrolling to reduce control overhead and improve computation throughput, demonstrating how execution time can be substantially decreased for core operations such as matrix multiplication.

4.1. Slicing the algorithm execution

When an algorithm employs repetitive loops to perform mathematical operations, the cycle time may extend to problematic levels if all operations are executed consecutively. This situation can arise during the training phase of a ML algorithm, which requires repetitive mathematical computations over the samples in the training dataset. If the dataset is sufficiently large, the cycle time may not only reach undesirable latencies, but could also exceed the maximum allowable limits for the PLC, potentially resulting in the CPU entering a STOP state.

To address this issue, it is essential to implement a mechanism that prevents the cycle time from exceeding acceptable thresholds during both the training and inference phases of an ML algorithm. Two approaches have been considered for developing this mechanism: partitioning the execution of the algorithm by either iterations or time. This paper implements the partitioning of the algorithm by iterations, allowing the specification of the number of iterations to perform within each PLC cycle. Consequently, upon completion of these iterations, the PLC can resume normal operations, thereby avoiding delays caused by excessively long mathematical computations.

4.2. Ordered memory access

To gauge the significance of an ordered access to memory and to leverage its advantages, it is important to understand the concepts of row-major order and column-major order. In row-major ordering, successive elements are assigned to consecutive memory locations by traversing across the rows before moving down to the next row, resulting in a row-wise data storage format. Conversely, column-major ordering assigns successive elements by traversing down the columns, and then proceeding to the next column, thereby storing data in a column-wise manner. To achieve optimal performance, it is essential to access memory locations sequentially, in accordance with the order of memory storage whenever possible. This approach helps minimize cache misses, thereby enhancing overall performance.

4.3. Loop interchange

In compiler theory, loop interchange is a technique that involves exchanging the execution order of two or more loops when they are part of a nested loop structure. This optimization can transform non-sequential memory access patterns into sequential ones, thereby enhancing locality of reference. By accessing elements in a sequential manner, this technique takes advantage of CPU cache usage, which can lead to significant performance improvements. If the compiler does not automatically apply this optimization, it is possible to implement it manually provided that certain conditions are met, especially when the programmer is aware of the data storage format in memory (e.g., row-major or column-major order).

As discussed in the previous section (*Ordered Memory Access*), if it is necessary to iterate over all the values of a two-dimensional array in a system that stores data in row-major order, greater performance can be achieved by employing the loop order illustrated below in Source Code 1. Conversely, in a memory system that utilizes column-major order for data storage, optimal performance requires reversing the execution order of the two loops; the loop that was previously outer should become inner, and the loop that was previously inner should become outer.

To evaluate the effectiveness of this technique, we conducted a test on the Siemens S7-1516 PLC involving two-dimensional square matrices **A** and **B** of size 450×450. The test consisted of sequentially accessing each position of matrix **A** by either rows or columns and storing the results in matrix **B**, accessed in the same fashion. The primary objective of this test was to assess the performance differences based on the method of accessing the matrices. The Structured Control Language (SCL) code utilized for this experiment is presented in Source Codes 1 and 2.

```
1 FOR #i:= 1 TO 450 BY 1 DO
2 FOR #j:= 1 TO 450 BY 1 DO
3 ‘‘Matrix’’.B[#i, #j] := ‘‘Matrix’’.A[#i, #j];
4 END_FOR;
5 END_FOR;
```

Source Code 1: SCL code used for Row Major Order.

```
1 FOR #j:= 1 TO 450 BY 1 DO
2 FOR #i:= 1 TO 450 BY 1 DO
3 ‘‘Matrix’’.B[#i, #j] := ‘‘Matrix’’.A[#i, #j];
4 END_FOR;
5 END_FOR;
```

Source Code 2: SCL code used for Column Major Order.

Additionally, in accordance with the recommendations provided in the Siemens manual regarding Access Speed of Memory Areas, we also tested whether the use of an array index of type `DINT` demonstrates better performance compared to other data types. To this end, we analyzed both row-major and column-major order accesses using `DInt` and `Int` array index data types to access data stored in an optimized Data Block (DB).

The results of this analysis are presented in Table 1. From this test, we can deduce two key findings: first, accessing two-dimensional matrices in row-major order is faster than accessing them in column-major order; second, the use of an array index of data type `DINT` yields better performance compared to using the `INT` data type.

4.4. Array size discovery

Regarding the S7-1200 and S7-1500 series CPUs, there are two options for passing an array to a function: either by specifying the size of the array or by omitting the size and using asterisks as placeholders.

To determine the size of an array that has been passed without specifying the dimensions, the upper_bound and lower_bound instructions

Table 1
Matrix access and copy time test [milliseconds].

Access Mode	Data Type	Array Index Type	Time
Row Major Order	LReal	Int	59.94
Row Major Order	LReal	DInt	54.58
Row Major Order	DInt	Int	41.64
Row Major Order	DInt	DInt	38.11
Column Major Order	LReal	Int	67.53
Column Major Order	LReal	DInt	63.77
Column Major Order	DInt	Int	53.78
Column Major Order	DInt	DInt	48.19

are available on S7-1200 series CPUs with firmware version 4.2 or higher, and on S7-1500 series CPUs with firmware version 2.0 or higher. These instructions allow for the determination of the requested dimension size. It is important to highlight this aspect, as it pertains not only to a programming style but also to its direct impact on code execution performance.

We observed that during matrix multiplication, performance was significantly improved when the matrices were passed with their sizes explicitly specified. Our analysis indicates that this performance difference is not attributable to the time the Siemens functions take to determine the array limits; rather, it suggests that some form of optimization may occur at the compilation level or internally when the sizes of the arrays are defined. Table 2 presented and discussed in the next section illustrates these performance differences.

4.5. Loop unrolling

The loop unrolling technique is attributed to Tom Duff, who introduced it in 1983 to improve the performance of a real-time animation program. This approach later became known as Duff's Device. It can be formally defined as a compiler optimization technique that involves transforming loops to improve execution speed, at the cost of increased code size. This transformation can be applied either automatically by the compiler or manually by the programmer. The technique works by partially or fully unwinding the loop, effectively reducing or eliminating the loop control instructions needed for each iteration. By doing so, it minimizes the frequency of "end-of-loop" condition checks, thereby enhancing performance. Additionally, loop unrolling can mitigate branch penalties and reduce latencies associated with memory access delays.

However, this technique also has potential drawbacks. The increase in code size may lead to greater memory consumption, potentially causing higher instruction cache miss rates. Furthermore, unrolling can increase register pressure due to the need to store additional temporary variables within each unrolled iteration, which can lead to performance degradation. An example of code unrolled by a factor of 2 for a square matrix is provided below.

```
1 #jBlocked := 2;
2 #iBlocked := 2;
3 FOR #j := #C_LowCol TO #C_UppCol-1 BY #jBlocked DO
4 FOR #i := #C_LowRow TO #C_UppRow-1 BY #iBlocked DO
5 #acc00 := #acc01 := #acc10 := #acc11 := 0;
6 FOR #k := #A_LowCol TO #A_UppCol DO
7 #C[#i,#j] += #A[#i,#k] * #B[#k,#j];
8 #C[#i,#j+1] += #A[#i,#k] * #B[#k,#j+1];
9 #C[#i+1,#j] += #A[#i+1,#k] * #B[#k,#j];
10 #C[#i+1,#j+1] += #A[#i+1,#k] * #B[#k,#j+1];
11 END_FOR;
12 END_FOR;
13 END_FOR;
```

Source Code 3: Unrolling of 2 elements.

Table 2
Time required for matrix multiplication through the unrolling technique [seconds].

Mode	72×72	120×120	180×180
AUTO (without unrolling)	0.3402	1.5627	5.4707
AUTO (without unrolling, with Accs)	0.3033	1.4420	4.8640
MAN (without unrolling)	0.2229	1.0650	3.6158
MAN (without unrolling, with Accs)	0.1995	0.9052	3.0564
AUTO (unrolling, factor of 2)	0.1895	0.8530	2.9301
AUTO (unrolling, factor of 2, with Accs)	0.1614	0.7092	2.5084
MAN (unrolling, factor of 2)	0.1436	0.6736	2.2837
MAN (unrolling, factor of 2, with Accs)	0.1202	0.5604	1.9179
AUTO (unrolling, factor of 3)	0.1538	0.6890	2.3559
AUTO (unrolling, factor of 3, with Accs)	0.1251	0.5628	1.9425
MAN (unrolling, factor of 3)	0.1236	0.5775	1.9556
MAN (unrolling, factor of 3, with Accs)	0.0997	0.4605	1.5847
AUTO (unrolling, factor of 4)	0.1407	0.6340	2.1808
AUTO (unrolling, factor of 4, with Accs)	0.1094	0.4997	1.7100
MAN (unrolling, factor of 4)	0.1170	0.5379	1.8570
MAN (unrolling, factor of 4, with Accs)	0.0911	0.4237	1.4444

```
1 #jBlocked := 2;
2 #iBlocked := 2;
3 FOR #j := #C_LowCol TO #C_UppCol-1 BY #jBlocked DO
4 FOR #i := #C_LowRow TO #C_UppRow-1 BY #iBlocked DO
5 #acc00 := #acc01 := #acc10 := #acc11 := 0;
6 FOR #k := #A_LowCol TO #A_UppCol DO
7 #acc00 += #A[#i,#k] * #B[#k,#j];
8 #acc01 += #A[#i,#k] * #B[#k,#j+1];
9 #acc10 += #A[#i+1,#k] * #B[#k,#j];
10 #acc11 += #A[#i+1,#k] * #B[#k,#j+1];
11 END_FOR;
12 #C[#i,#j] := #acc00;
13 #C[#i,#j+1] := #acc01;
14 #C[#i+1,#j] := #acc10;
15 #C[#i+1,#j+1] := #acc11;
16 END_FOR;
17 END_FOR;
```

Source Code 4: Unrolling of 2 elements with accumulators.

To preliminarily evaluate the effectiveness of loop unrolling, we implemented and tested the dot product algorithm for multiplying two-dimensional matrices, first without unrolling and then with unrolling factors of 2, 3, and 4. For simplicity, tests were conducted, to allow testing over different matrix sizes, on square matrices of sizes 72×72, 120×120, and 180×180. Table 2 presents the execution times measured for each configuration. The table also indicates whether the arrays were passed with explicitly specified matrix dimensions (MAN) or without specified dimensions (AUTO), and whether temporary local variables, used as accumulators, were used to store intermediate results.

The results obtained in this preliminary experiment lead to two primary conclusions. First, there is a significant performance improvement when loop unrolling is applied. Specifically, using loop unrolling yields execution speeds up to three times faster compared to non-unrolled loops, representing a substantial increase in performance. Second, performance varies notably depending on whether the array dimensions are explicitly specified. When matrix dimensions are specified, execution times are significantly reduced, leading to improved performance.

5. Experimental setup

We run experiments to evaluate the performance gains of the proposed improvements in the implementation of several ML algorithms on PLC devices. Specifically, the experiments aim to provide empirical evidence to address three different research questions (RQ):

- RQ1: Which is the performance of different ML algorithms running on a PLC during their training and inference phases?

- RQ2: Which design trade-offs emerge in the implementation and deployment of ML algorithms regarding cycle time and number of iterations per cycle?
- RQ3: What is the impact of the number of iterations per cycle on the training execution speed?

To answer these RQs, the following experimental setup was configured:

PLC under consideration. A Siemens S7-1516F-3 PN/DP CPU (part number 6ES7 516-3FN01-0AB0) was utilized in conjunction with a digital input card (part number 6ES7 521-1BL00-0AB0) and a digital output card (part number 6ES7 522-1BL01-0AB0). The minimum programmed cycle time was established at 1 millisecond, while the maximum allowed cycle time was set to 6 seconds, which is the maximum cycle time allowed for this system [27]. Although communications was not used during program execution, the load time for them was configured to 50%, which is, as indicated in [27], the default preset time for the S7-1500 automation system. The software employed for programming was TIA Portal version 15.1. This configuration has been implemented to facilitate the tests to be conducted, with a particular focus on enabling time measurements even in the most unfavorable scenarios. In real-life applications, it is rare to find PLCs with the maximum cycle time set to several seconds instead of just a few milliseconds. However, this configuration is necessary to carry out the time measurements to collect the necessary data for further analysis.

Dataset Loading. There are two primary methods for loading datasets into a Siemens PLC: pre-runtime loading and runtime loading. Runtime loading occurs when the PLC stores data it receives through inputs or communication channels during program execution. In contrast, pre-runtime loading involves loading the dataset prior to program download and execution, so the dataset is stored directly in a Data Block that is subsequently loaded into the PLC memory along with the program.

Both methods were evaluated, and each has distinct advantages and limitations. Loading datasets via communication channels offers the advantage of automation; a program on a PC can be used to load data into the PLC rapidly when required. However, this method has limitations, including the need for a preliminary setup and restrictions on the amount of data that can be transmitted at once. These limitations can be problematic for large datasets, necessitating the development of a PLC program to combine dataset segments loaded in multiple steps.

In the case of directly copying the dataset into the data block, the main advantage is the ease and speed of a simple copy-paste operation. However, when storing data in an array structure (as required in our implementations), copying the dataset is unfeasible beyond a certain array size. Furthermore, it becomes impossible to view all loaded data directly. To address this limitation, a PLC program must be created to consolidate segments of the dataset loaded into separate, smaller Arrays, similar to the runtime loading approach.

Implemented ML Algorithms. Four different ML algorithms were considered for the experiments:

- *Model 1: Linear Regression* (`LR`), which predicts the value of the dependent variable based on the weighted sum of the input features and an additional bias term. To train the model, it is necessary to determine the value of the weights and bias that minimize the prediction error. For this purpose, the mean squared error (MSE) was considered as the cost function. To determine the value of the weights that minimize the cost function, various optimization techniques can be employed. In this paper, we implement two such methods: a closed-form solution based on the so-called *normal* equation (`NORM`), and batch gradient descent (`BGD`).
- *Model 2: Logistic Regression* (`LOG`), which tackles binary classification tasks by predicting the probability that an instance belongs to a particular class. The model produces a binary outcome, yielding one of two possible categories (often referred to as *positive* and *negative*). To estimate such probabilities, logistic regression computes a weighted sum of the input features plus a bias term, and applies the sigmoid function to produce an output ranging between 0 and 1. To train the model, it is necessary to adjust the parameters of the weights so that the model can estimate high probabilities for positive instances and low probabilities for negative instances. To this end, the training algorithm of this model minimizes a log loss function over the entire training set. Since there is no closed-form solution for adjusting the value of the parameters and the cost function is convex, an optimization algorithm must be employed. Similarly to linear regression, we use batch gradient descent for this purpose.
- *Model 3: K-Nearest Neighbors* (`KNN`), which employs a measure of distance to address both regression and classification tasks. We consider Euclidean distance and non-weighted (uniform) voting/averaging.
- *Model 4: Artificial Neural Network* (`ANN`), specifically a standard feed-forward ANN architecture comprising an input layer with 3 units, a single hidden layer with 3 units, and an output layer with 1 unit. Each unit in the network corresponds to an artificial neuron, with weighted connections facilitating signal propagation across layers. During training, the weights are iteratively updated to optimize performance on the given task. The architecture and activation dynamics follow conventional neural network formulations. The activation function employed in this study is the sigmoid function. Standard backpropagation is used to adjust the weights of the connections, to compensate for the errors incurred by each neuron during the learning process.

Implementation. The implemented library is publicly available at the following GitHub repository: Matrix Multiplication Library (MML).

6. Results and discussion

We now present the results obtained to address RQ1 (Subsection 6.1), RQ2 (Subsection 6.2) and RQ3 (Subsection 6.3).

6.1. RQ1: how do different ML algorithms perform on a PLC during both training and inference?

We first recall that the primary goal of RQ1 is to assess the performance of PLCs in terms of execution speed during both model training and inference phases. In order to accomplish this objective, we have designed an experimental scenario involving three datasets, each containing 100, 1,000, and 10,000 examples, for evaluating the training and inference performance of `LR`, `LOG`, and `ANN` models. In the case of the `KNN` model and for the sake of consistency with other ML algorithms, we retain the term *training* in our discussions of `KNN` to denote the dataset containing all elements (neighbors) and *inference* for the dataset containing the values to be evaluated, even though this learning algorithm does not require a training phase. For the `KNN` algorithm, we utilized three training datasets with 100, 250, and 500 examples, and three inference datasets with 100, 250, and 500 examples, using four classes and testing it with $K = 50$ and $K = 100$ nearest neighbors. The performance evaluation was conducted using the proposed strategies described in Section 4, which enhances the efficiency of matrix multiplication. We hereafter refer to the library implementing these approaches as Matrix Multiplication Library (`MML`). Additionally, we evaluated an alternative implementation using a naïve matrix multiplication operation, consistent with Siemens' `LGF` library (referred to as `LGF` in what follows).

We first inspect the time required by the PLC to perform the training phase (or execution in the case of `KNN`) of the implemented models using the `LGF` and `MML` libraries under the most favorable configuration identified in our evaluation, which involves executing the maximum possible number of iterations per cycle from the assessed cases. The effect of varying the number of iterations per cycle will be analyzed in RQ2. A summary of the results obtained during the `KNN` execution is presented in

Table 3
KNN execution duration with a 100 examples, 100 test examples, 4 labels dataset (in seconds).

Model	AUT	MAN
KNN (50 neighbors)	0.174567	0.161022 (−7.76%)
KNN (100 neighbors)	0.183976	0.168318 (−8.51%)

Table 4
KNN execution duration with a 250 examples, 250 test examples, 4 labels dataset (in seconds).

Model	AUT	MAN
KNN (50 neighbors)	1.159803	1.080765 (−6.81%)
KNN (100 neighbors)	1.181968	1.099178 (−7%)

Table 5
KNN execution duration with a 500 examples, 500 test examples, 4 labels dataset (in seconds).

Model	AUT	MAN
KNN (50 neighbors)	5.079164	4.779432 (−5.90%)
KNN (100 neighbors)	5.124098	4.815875 (−6.02%)

Table 6
Training duration with a 100 examples dataset (in seconds).

Model	LGF	MML, AUT	MML, MAN
LR (NORM)	0.001087	0.000645 (−40.66%)	0.000503 (−53.73%)
LR (BGD)	1.135192	0.985711 (−13.17%)	0.782709 (−31.05%)
LOG	2.082104	2.023350 (−2.82%)	1.376346 (−33.90%)
ANN	8.434508	7.251318 (−14.03%)	5.642015 (−33.11%)

Table 7
Training duration with a 1,000 examples dataset (in seconds).

Model	LGF	MML, AUT	MML, MAN
LR (NORM)	0.010975	0.006317 (−42.44%)	0.004150 (−62.19%)
LR (BGD)	11.006060	9.469784 (−13.96%)	7.490139 (−31.95%)
LOG	20.619547	20.087653 (−2.58%)	13.602865 (−34.03%)
ANN	85.028064	72.967473 (−14.18%)	57.099099 (−32.85%)

Table 8
Training duration with a 10,000 examples dataset (in seconds).

Model	LGF	MML, AUT	MML, MAN
LR (NORM)	0.112655	0.059574 (−47.12%)	0.043434 (−61.45%)
LR (BGD)	110.274792	94.196645 (−14.58%)	74.480953 (−32.46%)
LOG	206.885669	200.702957 (−2.99%)	136.297452 (−34.12%)
ANN	846.113391	725.726850 (−14.23%)	565.438540 (−33.17%)

Tables 3, 4 and 5 and the training phase of the other models is presented in Tables 6, 7, and 8. The legend **MAN** means that the dimension of the matrices has been explicitly indicated, whereas **AUT** denotes that the dimension of the matrices has not been indicated and that has been inferred through the use of functions. The legend **BGD** denotes Batch Gradient Descent, whereas **NORM** denotes the Normal Equation. The training time difference between models trained with the MML library and those trained with LGF (used as the reference framework) is indicated in parentheses for each instance. In the case of KNN, since it does not apply heavy matrix algebra, the same basic code was utilized, with tests conducted using parameters explicitly passed (**MAN**) or not (**AUT**) and the speed improvement is compared to AUT. Although the closed-form solution of LR using the normal equation does not require a training phase, we have included its computation time in the training time tables for consistency and to facilitate a uniform comparative analysis.

For the inference phase, due to the simplicity of the mathematical implementation and the absence of large matrix multiplications, it was not possible to differentiate between the performance of different libraries

Table 9
Inference duration with a 100 examples dataset (in seconds).

Model	LGF	MML, AUT	MML, MAN
LR	0.000116	0.000071 (−38.79%)	
LOG	0.001044	0.001044 (0%)	
ANN	0.003111	0.002804 (−9.87%)	0.002193 (−29.51%)

Table 10
Inference duration with a 1,000 examples dataset (in seconds).

Model	LGF	MML, AUT	MML, MAN
LR	0.001166	0.000670 (−42.54%)	
LOG	0.008113	0.006138 (−24.34%)	
ANN	0.030661	0.027066 (−11.72%)	0.021121(−31.11%)

Table 11
Inference duration with a 10,000 examples dataset (in seconds).

Model	LGF	MML, AUT	MML, MAN
LR	0.011156	0.006692 (−40.01%)	
LOG	0.065545	0.044923 (−31.46%)	
ANN	0.311171	0.265253 (−14.76%)	0.208760 (−32.91%)

for LR, LOG, and KNN. Instead, the same code was utilized, with tests conducted using parameters explicitly passed (**MAN**) or not (**AUT**). The time required to perform inference for the implemented models are presented in Tables 9, 10, and 11. For ANN, which involves significant matrix multiplication for the inference phase, the same approach used during training was applied (**LGF**, **MML, AUT**, and **MML, MAN**). Similarly to the training phase, the inference time difference between models trained with the MML library and those trained with LGF (used as the reference framework)is indicated in parentheses for each instance

Our evaluation provides evidence of the PLC's performance by presenting the time required to complete all training and inference computations using both the Siemens LGF library, which employs a naïve matrix multiplication algorithm, and the proposed MML library. These execution times offer insights into the PLC's capability to run the ML models. The results presented in the tables demonstrate that the MML library outperforms the LGF library providing the capability of executing the algorithms faster.

6.2. RQ2: which design trade-offs emerge in the implementation and deployment of ML algorithms regarding cycle time and number of iterations per cycle?

We now focus on RQ2, which aims to evaluate the impact of the number of iterations per cycle on the cycle time during the training phase. To this end, we designed a scenario in which training is executed for each implemented model, while varying the number of iterations per cycle (1, 2, 3, 4, 5, 25, 125, 250, 500, 750, and 1000), and monitoring the longest cycle time in each execution. This experiment examines the model's behavior by correlating the influence of the number of iterations per cycle with the duration of the longest cycle time.

LR. We begin our analysis with LR. Figs. 1.a, 1.b, and 1.c illustrate the training times obtained from the LR model using the BGD method, performed over 1000 iterations with datasets containing 100, 1,000, and 10,000 examples. We present the case of LR using only the BGD method, as the computation of the normal equation is completed in a single cycle due to the closed-form solution, which does not require iteration.

LOG. We proceed with our analysis of LOG. Figs. 1.d, 1.e, and 1.f illustrate the training times recorded for the LOG model using the BGD method, executed over 1,000 iterations with datasets containing 100, 1,000, and 10,000 examples. Analogous to the case of LR, the three datasets comprise a sufficient number of elements to permit model evaluation within the PLC's storage constraints, while preserving representativeness for rigorous performance assessment.

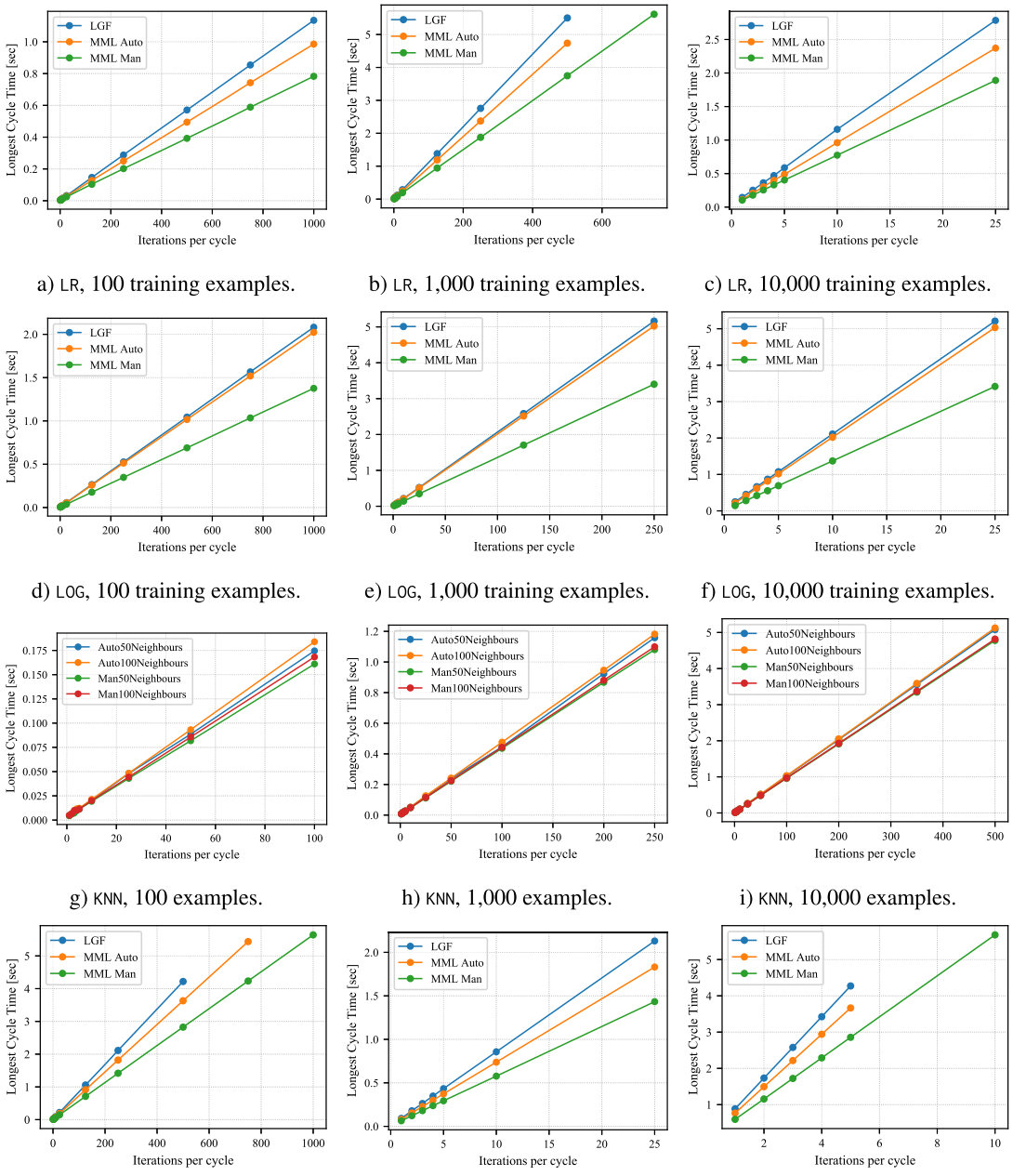


Fig. 1. Longest cycle time (in seconds) required to train the implemented ML algorithms (LR, LOG, KNN and ANN) as a function of the number of iterations per cycle, computed for datasets of varying sizes.

KNN. We now turn the focus of our analysis on the KNN algorithm. Figs. 1.g, 1.h, and 1.i report the training times recorded for the KNN model, where we utilized three datasets containing 100, 250, and 500 elements (neighbors), and three datasets of the same sizes containing the values to be evaluated. The model was tested using four labels and evaluated with 50 and 100 neighbors. Although the KNN algorithm does not employ an iterative approach like BGD, we have adapted its execution to allow for partitioned runs. This modification enables the selection of the number of examples to be processed in each cycle.

ANN. We finally present the analysis conducted with the ANN. Figs. 1.j, 1.k, and 1.l illustrate the training times recorded for the ANN model using the BGD method, executed over 1,000 iterations with datasets containing 100, 1,000, and 10,000 examples. Consistent with the LR and LOG cases, the dataset sizes enables both comparative analysis under analogous conditions and assessment of memory and performance constraints of the PLC, facilitating the identification of potential linearity patterns with respect to dataset sizes.

Based on the results obtained in RQ2, the following conclusions can be drawn:

- First, a directly proportional relationship was observed between the number of iterations per cycle and the maximum cycle time incurred during execution. Consequently, selecting the number of iterations per cycle during the training phase serves as an effective mechanism for controlling the maximum cycle time.
- Second, the MML library consistently outperformed the LGF library across all tested scenarios. Moreover, within the MML library, manual mode demonstrated superior performance compared to automatic mode for all models evaluated.

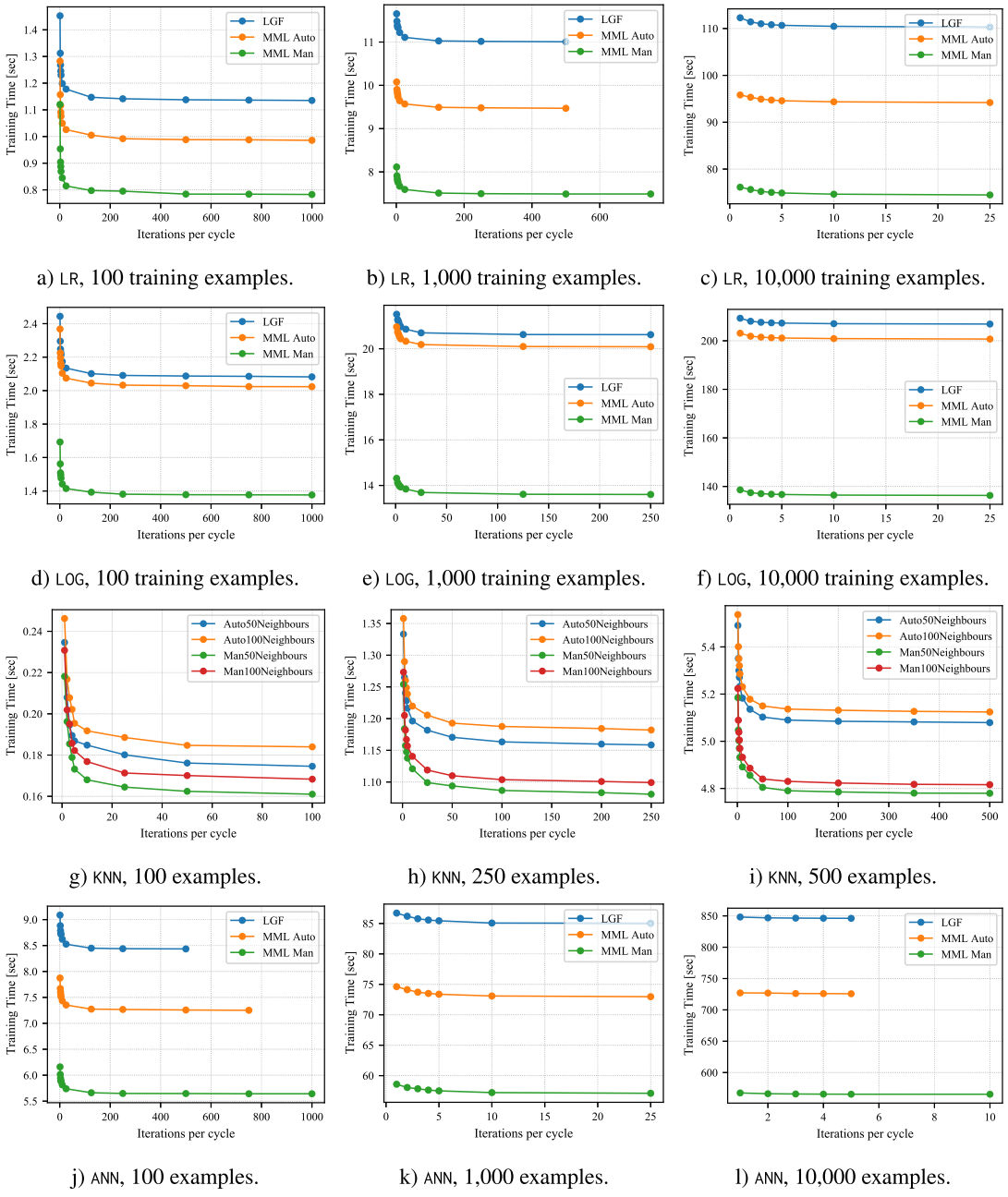


Fig. 2. Training Execution Time (in seconds) required to train the implemented ML algorithms as a function of the number of iterations per cycle, computed for datasets of varying sizes.

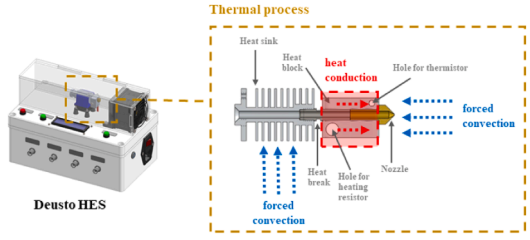


Fig. 3. Detailed view of the thermal process occurring in the *Deusto HES* experimental prototype.

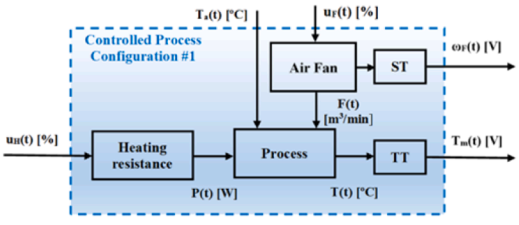
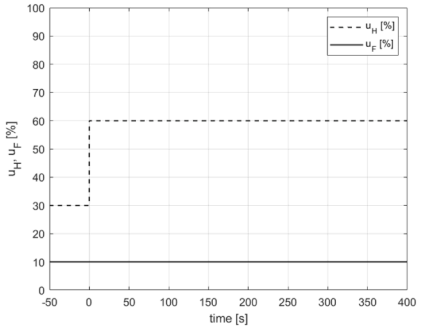


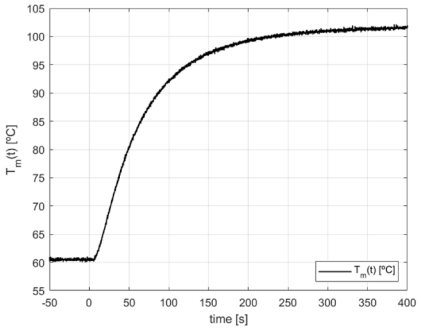
Fig. 4. Block diagram describing the controlled thermal process for the selected configuration, including all relevant variables and their corresponding units.

- Finally, the models that benefited most from the proposed library were those with a high computational demand for matrix multiplication, such as LR and ANN.

Therefore, the results demonstrate that the use of the MML library, combined with model configurations that allow the selection of the number of iterations per cycle, constitutes a powerful approach for



a) Input signals $u_H(t)$ and $u_F(t)$.



b) Process reaction curve $T_m(t)$.

Fig. 5. Experimental data obtained from a step test for process-model identification at a given operating point under the considered controlled-process configuration.

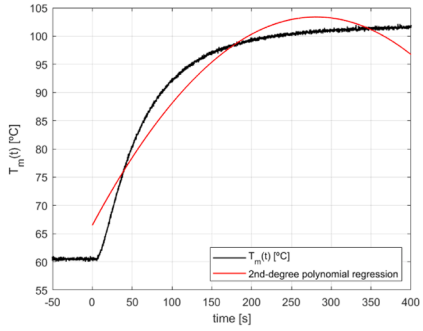
enhancing performance while maintaining precise control over cycle time – an essential requirement for PLCs operating in industrial environments.

6.3. RQ3: what is the impact of the number of iterations per cycle on the training execution speed?

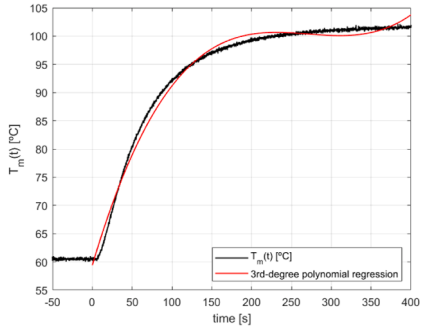
While the results presented in RQ1 provide a reference for validating the performance of the PLC in terms of execution time for ML algorithms, RQ2 focuses on analyzing the relationship between cycle time and the number of iterations per cycle. RQ3 aims to extend this analysis by correlating the number of iterations per cycle with the total time required to complete the training of the examined models. For this purpose, we reuse the scenario outlined in RQ2 and present, in the following figures, a direct comparison based on the data collected from the three datasets for each implemented model. This analysis will yield valuable insights into the impact of the number of iterations per cycle on the total execution time of the model, enabling a more comprehensive understanding of which iteration count per cycle is most beneficial for specific requirements

LR. We begin our analysis with `LR`. Figs. 2.a, 2.b and 2.c present the training times, in relation to iterations per cycle, obtained from the `LR` model using the BGD method. The model was trained for 1000 iterations using datasets comprising 100, 1,000, and 10,000 samples, with each size selected to balance representativeness and compliance with the PLC's computational constraints, thereby ensuring validity of the evaluation.

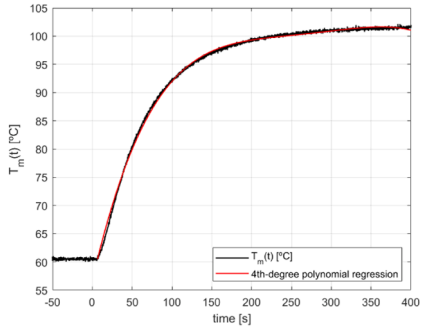
LOG. We continue with our analysis of `LOG`. Figs. 2.d, 2.e, and 2.f depict the training times, in relation to iterations per cycle, recorded for the `LOG` model using the BGD method. This analysis was conducted over 1,000 iterations with datasets containing 100, 1,000, and 10,000 examples. The sample sizes of the datasets employed ensure robust and equitable comparisons both among the evaluated models and with the outcomes reported in RQ2, thereby supporting the validity and consistency of the experimental analysis.



(a) Second-degree polynomial regression: $R^2 = 0.9289$.



(b) Third-degree polynomial regression: $R^2 = 0.9907$.



(c) Fourth-degree polynomial regression: $R^2 = 0.9980$.

Fig. 6. Second-, third-, and fourth-degree polynomial regression models learned by using our developed library.

KNN. We continue our analysis with the `KNN` algorithm. Figs. 2.g, 2.h, and 2.i present the training times, in relation to iterations per cycle, recorded for the `KNN` model. The analysis utilized three datasets containing 100, 250, and 500 elements (neighbors), as well as three datasets of the same sizes containing the values to be evaluated. The model was tested with four labels and evaluated using 50 and 100 neighbors. Although `KNN` does not employ an iterative approach like BGD, we have adapted its execution to allow for partitioned runs. This modification enables the selection of the number of examples to be processed in each cycle.

ANN. We conclude by presenting the analysis conducted with this ML model. Similarly to the previous models, Figs. 2.j, 2.k, and 2.l show the training times, in relation to iterations per cycle, recorded for the Neural Network model using the BGD method, executed over 1,000 iterations with datasets containing 100, 1,000, and 10,000 examples.

The following conclusions can be drawn from the results obtained in RQ3:

- First, the number of iterations per cycle influences the total training time. Specifically, increasing the number of iterations per cycle reduces the overall time required to complete the training process.
- Second, the relationship between the number of iterations per cycle and total training time is not linear. Beyond a certain threshold, additional iterations yield diminishing returns in terms of training time reduction. Considering the findings from RQ2 (namely, that increasing the number of iterations per cycle also increases cycle time) and the characteristic shape of the graphs obtained, selecting a number of iterations per cycle corresponding to the "elbow point" in the curves appears to offer a suitable trade-off between longest cycle time and training speed.
- Third, consistent with the conclusions of RQ1 and RQ2, the `MML` library outperforms the `LGF` library across all models evaluated. Additionally, the `MML` library in manual mode yields better performance than in automatic mode.

7. Experimental validation using a laboratory prototype

To complement the previous performance analysis and demonstrate the practical feasibility of implementing ML algorithms on PLCs, we validated our approach using a real-world thermal process. This experimental validation aims to confirm that the proposed methods are not only theoretically sound but also applicable in realistic industrial scenarios.

The laboratory prototype employed in this study is the Deusto Heater Experimental Setup (*Deusto HES*), designed and built at the Faculty of Engineering of the University of Deusto [28,29]. As illustrated in Fig. 3, the upper part of the device incorporates a 3D-printer extruder head housed inside a methacrylate duct, open at one end and equipped with a fan positioned in front of the hot end. The thermal process of interest occurs in the extruder head, which serves exclusively as a heating element. Its dynamics result from heat conduction within the block, driven by an embedded resistive heater, together with natural convection to surrounding air and forced convection generated by the fan. A temperature sensor integrated into the heat block enables continuous monitoring of the temperature. The system is reconfigurable, since both the heating power and the airflow can be adjusted by modifying the duty cycle of the PWM control signals. As demonstrated in [30], this thermal process exhibits characteristic fractional-order behavior.

Fig. 4 illustrates the block diagram of the thermal process along with its main variables. The controlled variable is the heat-block temperature, denoted as $T(t)$. The manipulated variable is the power delivered to the heating resistor, $P(t)$, which is regulated through the PWM control signal $u_H(t)$. The measured variable corresponds to the thermistor-based temperature reading, $T_m(t)$. Two primary disturbances affect the system: the fan command signal $u_F(t)$ and the ambient temperature $T_a(t)$. In the configuration adopted for this work, the heating resistor acts as the final control element, while the fan speed remains constant.

Building on the block diagram described above, Fig. 5 shows the input signals and the corresponding reaction curve obtained from an open-loop step test. The initial conditions were $u_H = 30\%$ and $u_F = 10\%$. At $t = 0$ s, a step of amplitude $\Delta u_H = 30\%$ was applied, while u_F remained unchanged. The measured temperature T_m increased from 60.50 °C to 102.50 °C, resulting in an increment of $\Delta T_m = 42$ °C. These time-series data characterize the thermal dynamics of the process under specific operating conditions and have been used to validate different fractional-order model identification approaches, including analytical methods [30,31] and hybrid methodologies combining Particle Swarm Optimization (PSO) with ML models such as ANN [32].

A representative use case is then considered, in which the PLC described earlier processes data acquired from the real-world process using the proposed laboratory prototype. In this experiment, a total of 4,501 samples were collected with a sampling interval of 0.1 s. Since the process reaction curve was obtained from the instant at which the step input was applied to the heating resistor ($t = 0$), a total of 4,001 samples were effectively used for training. A *polynomial regression model* was employed, and the corresponding parameter vector θ was estimated using the normal equation method. The coefficient of determination R^2 was computed to assess the goodness of fit of second-, third-, and fourth-degree polynomial regression models.

Figs. 6.a, 6.b, and 6.c display the fitted regression curves and illustrate the quality of the obtained fits, with the corresponding R^2 values reported in the captions. The results indicate that the third- and fourth-degree polynomial models achieve a satisfactory fit, thereby supporting the applicability of the proposed method in practical scenarios. This validation demonstrates that ML-based modeling can be effectively executed on a PLC using real process data, confirming the feasibility of integrating advanced learning techniques into industrial control systems without any dependence on specialized external hardware.

8. Conclusions and research directions

This manuscript has been elaborated on ML techniques within a Siemens PLC to demonstrate the feasibility of leveraging the potential of these devices. In this context, we present an approach aimed at mitigating two of the most significant challenges associated with such implementations: computational speed, with particular emphasis on the efficiency of matrix multiplication, and cycle time, which must be maintained at acceptable levels to ensure the system performs its functions effectively.

To this end, two key areas have been investigated: First, to enhance computation speed, particularly for matrix multiplication, the memory access times of the PLC were studied and analyzed. This analysis revealed that memory access significantly impacts overall computation time. Consequently, the `MML` was developed to incorporate improvements based on these findings, thereby increasing the speed of this mathematical operation on the PLC. Second, the challenge of maintaining an acceptable cycle time was addressed, even when an algorithm requires intensive calculations that could be time-consuming. It was demonstrated that partitioning the execution of these operations across multiple cycles is particularly effective in sustaining low cycle times.

The results obtained from three different scenarios, which examined the relationship between total execution time, longest cycle time, and the number of iterations per cycle, have been conclusive regarding the capabilities of the proposed library and the implementation strategies for the algorithms. These findings provide evidence that the presented approach can effectively address the inherent limitations of PLCs, indicating a viable direction for overcoming these challenges.

In the future we anticipate further improvements in efficiency through refined loop unrolling techniques in matrix multiplication. We also plan to explore alternative strategies for optimizing memory access speed, given its demonstrated influence on performance. Another important direction for future research involves establishing a fully quantitative comparison between PC-based and PLC-based implementations of ML algorithms. Such a study would allow for a systematic assessment of the computational trade-offs between conventional AI execution environments and the constrained, real-time setting of industrial PLCs. The methodological foundations developed in this paper (particularly the optimized matrix operations,) constitute an enabling stepping stone toward this objective. Expanding the experimental framework to incorporate more demanding ML workloads, including deep-learning-based inference or hybrid data-driven/process-driven models, therefore represents a natural continuation of our research. These avenues, along with additional approaches to enhance computation speed, constitute the core of our future research agenda.

Declaration of generative AI technologies in the manuscript preparation process

During the preparation of this work the authors used GPT5 in order to improve the language and readability of the manuscript. After using it, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

CRediT authorship contribution statement

David Zamora-Arranz: Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization; **Pablo Garcia-Bringas:** Writing – review & editing, Validation, Project administration, Methodology, Funding acquisition, Conceptualization; **Juan J. Gude:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization; **Javier Del Ser:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Funding acquisition.

Data availability

Data will be made available on request.

Declaration of competing interest

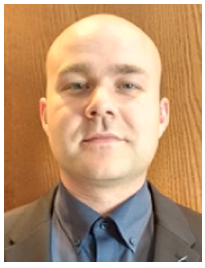
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

P. García-Bringas and Juan J. Gude acknowledge funding support from the Basque Government through the consolidated research group D4K-Deusto for Knowledge IT1528-22, IT1796-26. The work of J. Del Ser-is supported by the same institution (consolidated research group MATHMODE, IT1866-26).

References

[1] B.I. Adekunle, E.C. Chukwuma-Eke, E.D. Balogun, K.O. Ogunsola, Machine learning for automation: developing data-driven solutions for process optimization and accuracy improvement, Mach. Learn. 2 (1) (2021).
[2] F. Bachinger, J. Zenisek, M. Affenzeller, Automated machine learning for industrial applications–challenges and opportunities, Proced. Comput. Sci. 232 (2024) 1701–1710.
[3] R. Rai, M.K. Tiwari, D. Ivanov, A. Dolgui, Machine learning in manufacturing and industry 4.0 applications, 2021.
[4] V.S. Gurav, et al., The impact of industrial automation on the manufacturing industry in the era of industry 4.0, in: IEEE International Conference on Computing Communication and Networking Technologies (ICCCNT), 2024, pp. 1–6.
[5] International Electrotechnical Commission, IEC 61131-3:2003 – Programmable Controllers – part 3: programming languages, Int. Stand., Geneva, Switzerland (2003).
[6] International Electrotechnical Commission, IEC 61131-3:2013 – Programmable Controllers – part 3: programming languages, Int. Stan., Third Edition, Geneva, Switzerland (2013).
[7] H. Wegmann, Fuzzy control and neural networks industrial applications in the world of PLCs, in: IEEE International Conference on Control and Applications, 1994, pp. 1245–1249.
[8] M.G.S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, F. Hussain, Machine learning at the network edge: a survey, ACM Comput. Surv. (CSUR) 54 (8) (2021) 1–37.
[9] A. Kumar, S. Goyal, M. Varma, Resource-efficient machine learning in 2 kb RAM for the internet of things, in: International Conference on Machine Learning, PMLR, 2017, pp. 1935–1944.
[10] C. Gupta, et al., ProtoNN: compressed and accurate kNN for resource-scarce devices, in: International Conference on Machine Learning, PMLR, 2017, pp. 1331–1340.
[11] Siemens, SIMATIC S7-1500 TM NPU: Manual, 2024, (https://support.industry.siemens.com/cs/attachments/109765877/S71500_tm_npu_manual_en-US_en-US.pdf?download=true). Accessed: 2024-11-24.
[12] L. Körösi, J. Paulusová, Neural network for PLC, Tech. Comput. Bratislava 2014 (2014) 22nd.
[13] J. Li, A. Gómez-Espinosa, Improving PID control based on neural network, in: IEEE International Conference on Mechatronics, Electronics and Automotive Engineering (ICMEAE), 2018, pp. 186–191.
[14] M.I. Mahmoud, B.A. Zalam, M.A. Bardiny, E.A. Gomah, A simplification technique for an adaptive neural network based speed controller for implementation on PLC for DC drive, in: AIML 06 International Conference, 2006.
[15] I. Topalova, A. Tzokev, Optimization of a MLP network structure for a real-time PLC application, in: IEEE Convention of Electrical and Electronics Engineers in Israel, 2008, pp. 396–398.
[16] I. Topalova, A. Tzokev, Automated texture classification of marble shades with real-time PLC neural network implementation, in: IEEE International Joint Conference on Neural Networks (IJCNN), 2010, pp. 1–8.
[17] I. Topalova, Automated marble plate classification system based on different neural network input training sets and PLC implementation, Int. J. Adv. Res. Artif. Intell. 1 (2) (2012). https://doi.org/10.14569/IJARAI.2012.010209
[18] A. Tzokev, I. Topalova, Image and data pre-processing model for real-time communication between dedicated PC and PLC neural network application in marble production, in: IEEE Mediterranean Electrotechnical Conference (MELECON), 2010, pp. 41–46.
[19] A.P. Dedy, M.F. Zambak, A.A. Ahmad, S. Suwarno, PLC Implementation as a flow computer for calculation of saturated steam mass meetings with the linear divided regression method (application: PT. XYZ-Kuala tanjung), J. Comp. Sci., Infor. Technol. Telecommun. Eng. 1 (1) (2020) 8–16.
[20] O. Duymazlar, M. Engin, D. Engin, Embedded artificial neural network on PLCs to predict nonlinear system responses, in: IEEE Mediterranean Conference on Embedded Computing (MECO), 2020, pp. 1–4.
[21] C. Doumanidis, P.H.N. Rajput, M. Maniatakos, ICSML: Industrial control systems ML framework for native inference using IEC 61131-3 code, in: Proceedings of the 9Th ACM Cyber-Physical System Security Workshop, 2023, pp. 60–71.
[22] I.-S. Jung, et al., PLC Control logic error monitoring and prediction using neural network, in: IEEE International Conference on Natural Computation, 2, 2008, pp. 484–488.
[23] J.V. Fonseca, E.F.M. Ferreira, Increase of PLC computability with neural network for recovery of faults in electrical distribution substation, in: IEEE International Instrumentation and Measurement Technology Conference (I2MTC), 2013, pp. 511–516.
[24] Siemens, Structure and Use of the CPU Memory, 2013, Available at: Accessed: 2024-11-24. https://cache.industry.siemens.com/dl/files/101/59193101/att_80162/v1/s71500_structure_and_use_of_the_PLC_memory_function_manual_en-US_en-US.pdf.
[25] J. Smith, J. Doe, Programming Guideline for S7-1200/S7-1500, Siemens AG, 2017. Available at: https://cache.industry.siemens.com/dl/files/040/90885040/att_970576/v1/81318674_Programming_guideline_DOC_v16_en.pdf.
[26] P.J. Denning, The working set model for program behavior, Commun. ACM 11 (5) (1968) 323–333.
[27] J. Smith, J. Doe, Function Manual S7-1500, Siemens AG, 2014. Available at: https://cache.industry.siemens.com/dl/files/558/59193558/att_112303/v1/s71500_cycle_and_reaction_times_function_manual_en-US_en-US.pdf.
[28] J.J. Gude, P. García Bringas, A novel control hardware architecture for implementation of fractional-order identification and control algorithms applied to a temperature prototype, Mathematics 11 (1) (2022) 143.
[29] J.J. Gude, P. García Bringas, Proposal of a control hardware architecture for implementation of fractional-Order controllers, Perspectives in Dynamical Systems II—Numerical and Analytical Approaches: DSTA, Łódź, Poland December 6–9, 2021 454 (2024) 229.
[30] J.J. Gude, P. Garcia Bringas, Proposal of a general identification method for fractional-order processes based on the process reaction curve, Fract. Fract. 6 (9) (2022) 526.
[31] J.J. Gude, A. Di Teodoro, O. Camacho, P. García Bringas, A new fractional reduced-order model-inspired system identification method for dynamical systems, IEEE Access 11 (2023) 103214–103231.
[32] I. Fidalgo Astorquia, N. Gómez-Larrakoetxea, J.J. Gude, I. Pastor, Fractional-Order system identification: efficient reduced-Order modeling with particle swarm optimization and AI-Based algorithms for edge computing applications, Mathematics 13 (8) (2025) 1308.



David Zamora-Arranz studied Computer Engineering for Management and Information Systems at the University of the Basque Country (UPV/EHU, Spain). He obtained his Master's degree in Automation, Electronics and Industrial Control from the University of Deusto in 2020. His academic excellence has been recognized with the Araba 4.0 Award for the best Bachelor thesis related to Industry 4.0, as well as the award for the best Master thesis in his graduating class. His research interests include AI, Industrial Automation, and Robotics.

Pablo García-Bringas is a Full Professor of Engineering at the University of Deusto. He holds an Executive MBA, a PhD in Computer Science and Artificial Intelligence (specialized on Cybersecurity), and Master's degrees in Telecommunications and Industrial Informatics. He currently serves as Vice-Dean of External Affairs and Head Researcher of the Deusto for Knowledge (D4K) research group. He previously served as Director of DeustoTech - Deusto Institute of Technology and Director of Research at the Faculty of Engineering. His research focuses AI applications in information security and industrial processes. He has led numerous projects and technology transfer initiatives, published over 300 papers, and supervised 30 PhD dissertations.

Juan J. Gude holds a degree in Applied Physics with a major in Electronics and Automation from the University of the Basque Country (UPV/EHU), and a Ph.D. in Engineering from the University of Deusto. He is a Senior Lecturer at the University of Deusto (Spain), where he has been with the Department of Computing, Electronics and Communication Technologies since 2001 and has led the Laboratory of Measuring Systems and Control at the Faculty of Engineering for over 20 years. His activity centers on the development of industrial prototypes for engineering education and research. His research interests include applied industrial control and dynamic systems modeling, with emphasis on fractional-order systems and industrial applications.

Javier Del Ser holds a Ph.D. in Control Engineering from the University of Navarra (2006), and a second Ph.D. in Information and Communication Technologies from the University of Alcalá de Henares (2013, Extraordinary PhD Award). He is Chief AI Scientist at TECNALIA and a Distinguished Researcher at the Dept. of Mathematics of the University of the Basque Country (UPV/EHU). His research focuses on applied AI. He has coauthored more than 480 scientific papers, edited 6 books, supervised 20 doctoral theses, and led more than 60 research projects and contracts.