Approaches for Integrating Deep Learning Models for Inference using AUTOSAR in ECU

Rudraksha Kelkar
Mercedes Benz
Research and Development India
Bengaluru, India
rudraksha.kelkar@mercedes-benz.com

Manishankar Rao
Mercedes Benz
Research and Development India
Bengaluru, India
manishankar.rao@mercedes-benz.com

Immanuel Utchula
Mercedes Benz
Research and Development India
Bengaluru, India
immanuel.utchula@mercedes-benz.com

Akshata Kulkarni
Mercedes Benz
Research and Development India
Bengaluru, India
akshata.kulkarni@mercedes-benz.com

Kaushik Raghunath
Mercedes Benz
Research and Development India
Bengaluru, India
kaushik.raghunath@mercedes-benz.com

***Abstract*—Deep Learning (DL) models are typically deployed on high-computation platforms like GPUs and FPGAs in cloud environments, leading to high latency and network dependency. In contrast, performing deep learning inference on micro-controllers in automotive applications reduces latency, minimizes network dependency, enhances security, and offers a cost-effective solution.**

This paper investigates the integration of deep learning models into AUTOSAR-compliant C-code for automotive Electronic Control Units (ECUs). Adhering to AUTOSAR standards guarantees compatibility, scalability, maintainability, and efficient resource management. The study emphasizes toolchains that transform deep learning models, initially trained and developed in Python with frameworks such as TensorFlow or PyTorch, or using other tools like MATLAB, into high-quality AUTOSAR-compliant C-code, tailored for specific compilers and hardware optimizations. Upon deployment, the deep learning software module within the ECU must satisfy rigorous real-time system requirements.

In this paper, we demonstrate the integration of a deep learning model into an ECU, emphasizing automation and seamless incorporation into existing workflows. The workflow setup can be extended to various deep learning models, but we use a simple neural network for estimating the battery's state of health to illustrate the process. The model is trained in Python, quantized for ECU deployment, and converted to AUTOSAR-compliant C-code through three methods: manual implementation with TargetLink, Simulink combined with the Deep Learning Toolbox, and a hybrid approach integrating Simulink-generated code into existing TargetLink logic using S-functions. This comparison focuses on fixed-point scaling and hardware optimization to ensure code efficiency, reliability, and correctness, verified through static code analysis and unit testing. Functional tests in a Software-in-the-Loop (SIL) environment validate system performance and accuracy by comparing results to a physics-based model. Hardware-in-the-Loop (HIL) testing assesses real-time behavior and robustness, ensuring reliable performance with actual hardware components under real-world conditions.

***Index Terms*—Deep Learning, Simulink, TargetLink, S-Function, Electronic control unit (ECU), AUTomotive Open System ARchitecture (AUTOSAR), Software-in-the-Loop (SIL), Hardware-in-the-Loop (HIL)**

I. Introduction

Integrating Deep Learning (DL) into embedded systems is crucial for modern, cost-effective solutions. TinyML, a subfield of ML, enables sophisticated DL models to run directly on micro-controllers managing sensors and actuators. This offers real-time inference on-device, reducing latency, network dependency, and enhancing security and privacy by avoiding cloud computation.

While DL models typically deploy on powerful platforms like GPUs or FPGAs in the cloud, this introduces latency and network reliance. On-device inference, especially in automotive scenarios, mitigates these issues and provides a cost-effective alternative. Recent successes include real-time keyword spotting and Human Activity Recognition on 32-bit low-power micro-controllers. Deploying these models involves extracting and encoding weights/biases, developing an inference program, and flashing it onto the micro-controller's memory [1].

Despite the benefits, micro-controller DL faces challenges like limited hardware resources and a lack of unified TinyML frameworks. For instance, a Binary Neural Network (BNN) for sound event detection on a RISC-V GAP8 micro-controller showed 10x better computational efficiency than an ARM Cortex-M7 int16 implementation, despite a 7.3% accuracy drop. However, RISC-V micro-controllers are scarce, and BNNs require manual data packing/unpacking, adding overhead [2].

Various studies offer solutions. Falaschetti et al. optimized ECG-Based Arrhythmia Classification on STM32 micro-controllers [3], while Perego et al.'s AutoTinyML framework efficiently identifies accurate and deployable models with reduced hardware usage [4]. Yu et al. proposed TinyCNN for low-latency, accurate feature extraction on ultra-low-power micro-controllers using quantization [5]. Falaschetti et al. also developed a lightweight CNN vision system for real-time

2325-1301/25/$31.00 ©2025 IEEE
DOI 10.1109/RTCSA66114.2025.00013

concrete crack detection on low-cost platforms [6]. Ghibellini et al. used dynamic range quantization for Human Activity Recognition on an Arduino BLE 33 Sense, reducing model size [7]. Meißl et al. demonstrated real-time DL on micro-controllers using LSTMs and 1D CNNs for handwritten character classification, significantly improving inference speed [8]. Most DL models use floating-point parameters, demanding high memory and computation. This is addressed by converting float32 models to int8 via post-training quantization. While frameworks like TensorFlow Lite and Intel OpenVINO support quantized models for higher-end edge devices, they don't typically cater to micro-controllers, which have far more limited RAM and flash memory than mobile phones.

Various methods exist for writing C code for deep learning models, each with its own set of challenges. Manual coding, although offering complete control, is highly error-prone and time-consuming, often lacking essential checks like saturation and division by zero, and failing to optimize fixed-point scaling. Libraries such as CMSIS-NN and frameworks like TensorFlow Lite for Microcontrollers provide efficient implementations but may not fully adhere to automotive standards or offer the same level of integration and optimization for specific hardware. Framework-specific code export and hybrid approaches can lead to fragmented workflows and increased complexity in maintaining code consistency and reliability. Embedded AI frameworks like Edge Impulse, while user-friendly, may not provide the same depth of customization and rigorous testing required for automotive applications.

Currently, several frameworks like TensorFlow Lite for Microcontrollers (TFLM), X-Cube-AI, and Texas Instruments Deep Learning support various micro-controller platforms. However, these tools often fall short of expectations. TFLM generates C++ inference libraries, which can be complex for embedded developers who prefer C, making debugging and customization in specific IDEs challenging. X-Cube-AI's proprietary nature restricts manipulation and extension, while many existing tools rely on hardware-specific runtime libraries, lacking a unified, vendor-independent framework for integrating quantized models into embedded software development environments.

Given these challenges, manual coding and frameworks often lead to fragmented workflows and reliability issues. In contrast, Simulink and TargetLink provide robust solutions with built-in mechanisms for fixed-point scaling, saturation checks, and error handling. These tools ensure that the generated code is reliable, efficient, and compliant with automotive standards, effectively addressing integration challenges and meeting industry expectations.

Implementing deep learning models onto an Electronic Control Unit (ECU) presents challenges. ECUs manage multiple modules using time-sharing schedulers for tasks like engine management and drive coordination. Integrating deep learning models requires seamless communication, interoperability, and efficient resource management with other software components, while meeting stringent real-time system requirements. Compatibility issues arise because different modules may use distinct communication protocols and data formats. Scalability becomes complex as the number of integrated modules increases. Efficient resource management is critical to ensure optimal use of computational and memory resources while maintaining real-time performance. Maintainability and upgradability are significant challenges, as the system must be easily maintained and upgraded without disrupting existing functionalities. Safety and security are paramount, requiring the integrated system to meet stringent standards.

To address these challenges, the automotive industry has widely adopted the AUTOSAR (AUTomotive Open System ARchitecture) standard. AUTOSAR provides a comprehensive framework for developing and integrating software components in ECUs. Key benefits include standardized architecture, modularity, reusability, abstraction layers, efficient resource management, and enhanced safety and security. AUTOSAR promotes compatibility and interoperability among components from different suppliers, simplifying integration and reducing development time. Its modular approach allows for reusable software components, enhancing scalability and reducing costs. Abstraction layers separate application software from hardware-specific details, facilitating easier maintenance and upgrades. AUTOSAR ensures optimal utilization of ECU resources to maintain real-time performance and incorporates safety and security features to meet industry standards, including functional safety (ISO 26262) and cybersecurity (ISO/SAE 21434).

This paper explores the implementation of deep learning models using AUTOSAR-compliant C-code, enhancing compatibility, scalability, maintainability, and resource management. The study investigates toolchains that convert Python-trained models from frameworks like TensorFlow or PyTorch into production-quality C-code, customizable for specific compilers and optimized for efficient ECU deployment.

The primary use case for this study focuses on estimating the battery state of health by considering factors such as charging and discharging power, energy throughput, battery temperature, state of charge, and depth of discharge. Real-time estimation of battery SoH at high temporal resolution offers substantial benefits for electric vehicles by enabling intelligent, data-driven battery management. High-frequency SoH monitoring allows for early detection of degradation or anomalies, enhancing operational safety through timely intervention by the Battery Management System. It enables dynamic optimization of charging, discharging, and thermal control strategies, ensuring efficient energy usage while extending battery lifespan. Accurate, real-time SoH information also improves state of charge (SOC) estimation and range prediction, contributing to a more reliable driving experience. Additionally, such detailed monitoring supports predictive maintenance and warranty validation by capturing degradation patterns under real-world conditions. However, estimating SoH at fine time intervals requires robust, noise-tolerant algorithms and computational efficiency to ensure meaningful insights. To support this, we reviewed literature on deep learning-based SoH estimation.

Saad El Fallah et al. highlight the degradation of lithium-ion batteries and the increasing use of deep learning, including CNN and RNN, for predicting State of Health (SoH) and State of Charge (SoC). Their work stresses the need for high-quality training data and continuous model adaptation for reliable predictions, suggesting hybrid models for future research [9]. Similarly, Kusumika Krori Dutta et al. explored FNN and CNN for estimating SoH and capacity in electric vehicle batteries, using temperature, voltage, and current data. Their CNN model outperformed FNN, achieving 98.46% accuracy for SoH and 97.86% for capacity, underlining the importance of diverse data for accurate predictions [10].

This study focuses on integrating a deep learning model into an Electronic Control Unit, emphasizing an automated toolchain and seamless integration into existing industry workflows. We use a simple neural network to estimate battery SoH, trained in Python and then quantized for memory optimization. Our primary goal is to compare three methodologies for modeling and code generation: manual implementation via TargetLink, Simulink with the Deep Learning Toolbox, and a hybrid approach combining Simulink-generated code into TargetLink using S-functions. The comparison centers on fixed-point scaling and hardware optimization. We assess code efficiency, reliability, and correctness through static code analysis and unit testing. Functional tests in a Software-in-the-Loop (SIL) environment validate the inference process and system performance. Finally, Hardware-in-the-Loop (HIL) testing assesses real-time behavior, robustness, latency, and ECU load, confirming reliable performance under real-world conditions. This paper ultimately aims to explore different approaches to integrate deep learning models into ECUs, reducing manual effort and maintaining an automated toolchain.

II. Methodology

The toolchain provides production-quality AUTOSAR-compliant C-code for deep learning models trained in Python, MATLAB, or other tools, ensuring seamless integration and optimization for automotive ECUs.

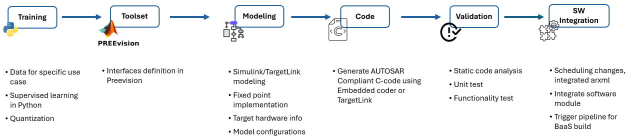

Training
Toolset
PREEvision
Modeling
Code
Validation
SW Integration
• Data for specific use case
• Supervised learning in Python
• Quantization
• Interfaces definition in Preevision
• Simulink/TargetLink modeling
• Fixed point implementation
• Target hardware info
• Model configurations
• Generate AUTOSAR Compliant C-code using Embedded coder or TargetLink
• Static code analysis
• Unit test
• Functionality test
• Scheduling changes, integrated arxml
• Integrate software module
• Trigger pipeline for BaaS build

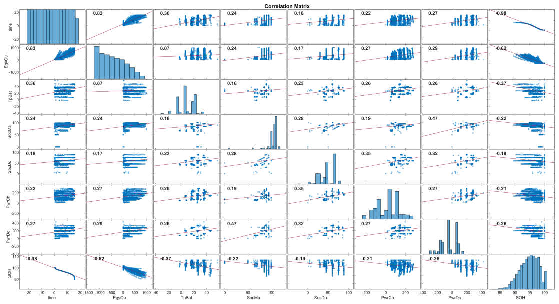
Figure 1: Workflow to integrate deep learning models in ECU

The workflow for deploying a deep learning model onto an automotive ECU is illustrated in Figure 1. The process involves several key steps:

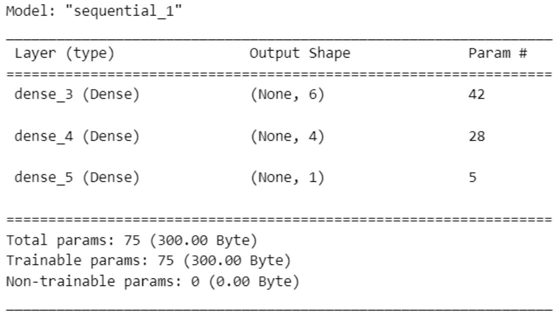
- Data Collection and Training: Gather data for the specific use case, train the deep learning model, and apply quantization to conserve storage resources.
- Toolset: Utilize essential tools such as PREEvision, MATLAB, Simulink, TargetLink, Deep Learning Toolbox, AUTOSAR Component Designer, Fixed-Point Designer, and Embedded Coder.
- Model Implementation: Generate the framework and import the quantized deep learning model into Simulink, or implement the deep learning model logic in TargetLink.
- Code Generation: Use AUTOSAR and Embedded Coder in Simulink or TargetLink Coder to generate the production-quality code.
- Validation: Conduct static code analysis, unit testing, and functionality testing to benchmark the model's performance, real-time behavior, robustness, latency, and ECU load.
- Integration and Deployment: Make necessary scheduling changes, integrate the module into the ECU software code base, and flash the validated software onto the target hardware.

A. Training

For effective battery state of health estimation, it is crucial to utilize a comprehensive set of features that accurately reflect the battery's performance and condition. Key features include Energy throughput, battery temperature, maximum state of charge, depth of discharge, charging and discharging power. These features provide valuable insights into the battery's operational patterns and degradation mechanisms, which are essential for accurate SoH estimations.



(a) Feature correlation matrix



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 6)                 42

 dense_4 (Dense)             (None, 4)                 28

 dense_5 (Dense)             (None, 1)                 5

=================================================================
Total params: 75 (300.00 Byte)
Trainable params: 75 (300.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

(b) Summary of deep learning model for battery state of health estimation

Figure 2: Feature correlation and summary of deep learning model

Figure 2a illustrates the correlation matrix for all features, analyzed using MATLAB toolbox, which helps in understanding the relationships between different variables. For example, a negative correlation coefficient of -0.82 between the state of health and energy throughput in a battery system indicates a strong inverse relationship. As energy throughput increases, SoH decreases significantly. This highlights the trade-off between usage and longevity, where extensive use leads to faster degradation of the battery's health.

The initial step in developing a deep learning model for SoH estimation involves gathering data specific to the use case from sources such as public datasets, APIs, through simulations and experiments, in this case vehicle measurements. This raw data must then be preprocessed to construct a dataset with relevant features by cleaning, handling missing values, normalizing, and performing feature engineering. Preprocessing these features is vital to enhance data quality and model performance. Techniques such as moving time averages can smooth out short-term fluctuations and highlight long-term trends, making the data more suitable for machine learning algorithms. By preprocessing the features, noise is reduced, missing values are handled, and data is normalized, ensuring that the neural network receives clean and consistent inputs for reliable SoH estimation.

Once the dataset is prepared, the deep learning model is trained using Python libraries such as TensorFlow, Keras, or PyTorch. The summary of the developed model, including its architecture and performance metrics, is presented in Figure 2b. After training, the model is saved in .pb format for seamless import into MATLAB with Simulink. This step ensures that the deep learning model's behavior is accurately captured and can be effectively executed within the automotive system, providing robust and precise State of Health estimation for the battery. However, for TargetLink implementation, the learnable parameters of the deep learning model must be manually copied into parameter interfaces, which increases the effort and time overhead.

B. Toolset

In the field of automotive software development, adhering to AUTOSAR standards is essential for ensuring interoperability and scalability. This guide briefly illustrates several key tools and licenses necessary for this workflow.

PREEvision is a versatile tool for modeling and designing automotive electrical/electronic (E/E) systems, enabling interface definition, requirement management, and the creation of detailed system architectures. MATLAB 2024b offers a high-level programming environment for algorithm development, data analysis, and numerical computation, incorporating the latest enhancements. Simulink provides a block diagram environment for modeling, simulating, and analyzing dynamic systems, and integrates seamlessly with MATLAB.

The AUTOSAR Component Designer toolbox assists in designing and generating AUTOSAR-compliant software components, supporting the import of arxml files and the creation of frame models. The Deep Learning Toolbox facilitates the integration of deep learning models into the Simulink environment, providing blocks and functions for designing, training, and simulating neural networks. The Fixed-Point Designer is crucial for optimizing models for fixed-point arithmetic, allowing for efficient implementation on hardware with limited precision. Embedded Coder generates production-quality C and C++ code from Simulink models, ensuring compliance with AUTOSAR standards for reliable and efficient code generation. All these toolboxes are provided by MathWorks.

Additionally, TargetLink, a tool from dSPACE, supports model-based development of embedded systems by automatically generating production code C code directly from MATLAB/Simulink models. This process reduces development time and minimizes errors, ensuring that the generated code is both efficient and reliable for automotive and other embedded applications.

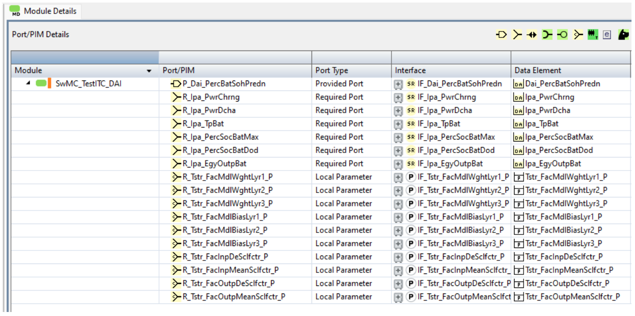
By utilizing these tools, developers can create robust and scalable AUTOSAR-compliant systems, ensuring seamless integration and high performance in automotive applications.

C. Modeling

In this paper, we explore and compare three methodologies for generating AUTOSAR-compliant C code for deep learning model inferencing: manual implementation with TargetLink, Simulink combined with the Deep Learning Toolbox, and a hybrid approach integrating Simulink-generated code into existing TargetLink logic using S-functions. Each approach has its own unique characteristics, which are discussed in detail. The choice between these methodologies depends on the specific requirements of the application, such as performance, resource constraints, precision, and development complexity. By understanding the nuances of each approach, developers can make informed decisions to optimize their embedded system design and ensure efficient and reliable implementation.

1) TargetLink approach: The first step involves defining necessary interfaces in PREEvision, specifying data types, signal properties, and communication protocols. This establishes a clear communication framework for seamless interaction between components. Figure 3 illustrates the definition of interfaces for deep learning model parameters, including weights and biases, which are manually fed with their default values, requiring significant effort and time. These interfaces are crucial as they ensure the deep learning model can be accurately integrated and function correctly within the automotive system. Once defined, these interfaces are exported as arxml files, which standardize the system's architecture and facilitate subsequent development stages.

The next step involves manually generating a frame model and developing a TargetLink model for inferencing, which executes with fixed-point arithmetic. This process uses simple algebraic equations to replicate deep learning model logic. Figure 4 illustrates the implementation, outlining tasks such as normalizing input values, performing neural network inferencing, and denormalizing output values. This manual approach with fixed-point arithmetic, though labor-intensive, ensures precise control over the model's implementation. It is crucial for meeting the specific requirements of the automotive application, providing robust and reliable performance within the ECU.

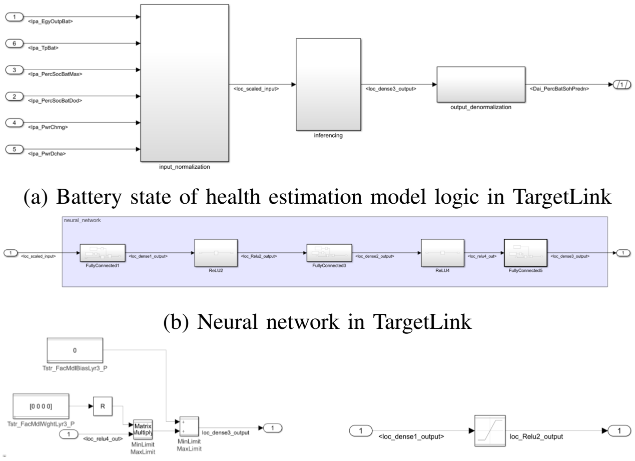


Module Details

Port/PIM Details

Module	Port/PIM	Port Type	Interface	Data Element
SwMC_TestITC_DAI	P_Dai_PercBatSohPredn	Provided Port	IF_Dai_PercBatSohPredn	Dai_PercBatSohPredn
	R_Ipa_PwrChrng	Required Port	IF_Ipa_PwrChrng	Ipa_PwrChrng
	R_Ipa_PwrDcha	Required Port	IF_Ipa_PwrDcha	Ipa_PwrDcha
	R_Ipa_TpBat	Required Port	IF_Ipa_TpBat	Ipa_TpBat
	R_Ipa_PercSocBatMax	Required Port	IF_Ipa_PercSocBatMax	Ipa_PercSocBatMax
	R_Ipa_PercSocBatDod	Required Port	IF_Ipa_PercSocBatDod	Ipa_PercSocBatDod
	R_Ipa_EgyOutpBat	Required Port	IF_Ipa_EgyOutpBat	Ipa_EgyOutpBat
	R_Tstr_FacMdlWghtLyr1_P	Local Parameter	IF_Tstr_FacMdlWghtLyr1_P	Tstr_FacMdlWghtLyr1_P
	R_Tstr_FacMdlWghtLyr2_P	Local Parameter	IF_Tstr_FacMdlWghtLyr2_P	Tstr_FacMdlWghtLyr2_P
	R_Tstr_FacMdlWghtLyr3_P	Local Parameter	IF_Tstr_FacMdlWghtLyr3_P	Tstr_FacMdlWghtLyr3_P
	R_Tstr_FacMdlBiasLyr1_P	Local Parameter	IF_Tstr_FacMdlBiasLyr1_P	Tstr_FacMdlBiasLyr1_P
	R_Tstr_FacMdlBiasLyr2_P	Local Parameter	IF_Tstr_FacMdlBiasLyr2_P	Tstr_FacMdlBiasLyr2_P
	R_Tstr_FacMdlBiasLyr3_P	Local Parameter	IF_Tstr_FacMdlBiasLyr3_P	Tstr_FacMdlBiasLyr3_P
	R_Tstr_FacInpDeSclfctr_P	Local Parameter	IF_Tstr_FacInpDeSclfctr_P	Tstr_FacInpDeSclfctr_P
	R_Tstr_FacInpMeanSclfctr_P	Local Parameter	IF_Tstr_FacInpMeanSclfctr_	Tstr_FacInpMeanSclfctr_P
	R_Tstr_FacOutpDeSclfctr_P	Local Parameter	IF_Tstr_FacOutpDeSclfctr_P	Tstr_FacOutpDeSclfctr_P
	R_Tstr_FacOutpMeanSclfctr_P	Local Parameter	IF_Tstr_FacOutpMeanSclfct	Tstr_FacOutpMeanSclfctr_P

Figure 3: Sample interface definition in PREEvision for TargetLink



(a) Battery state of health estimation model logic in TargetLink

(b) Neural network in TargetLink

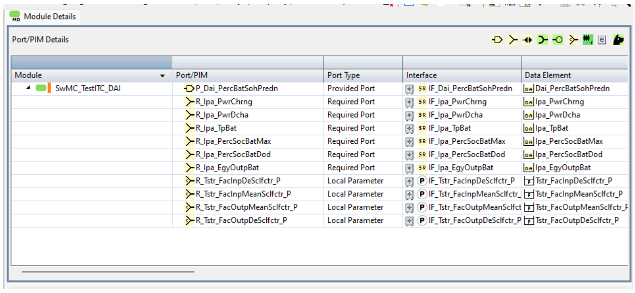
(c) Dense layer logic

(d) Activation function logic

Figure 4: TargetLink model for inferencing, includes logic for hidden layer and activation

The final step is to generate AUTOSAR-compliant C-code. This begins with opening the TargetLink Data Dictionary (TLDD) and configuring the necessary AUTOSAR properties. Essential AUTOSAR elements such as runnables, ports, and interfaces are defined within the TLDD. The appropriate AUTOSAR version is selected, and settings such as memory sections, optimization levels, and compiler options are adjusted to meet specific requirements. With these configurations in place, TargetLink generates AUTOSAR-compliant C-code based on the Simulink model and the specified TLDD settings. This code is optimized for deployment on automotive Electronic Control Units, ensuring that the deep learning model can be efficiently executed within the target hardware environment.

2) Simulink + Deep learning toolbox: The initial step involves defining necessary interfaces in PREEvision, specifying data types, signal properties, and communication protocols to establish a clear communication framework. Figure 5 illustrates that, in this approach, interfaces for weights and biases of deep learning models are not required, simplifying the process. This focus on essential communication elements streamlines the setup. Once these interfaces are meticulously defined, they are exported as arxml files, which standardize the system's architecture and facilitate subsequent development stages.



Module Details

Port/PIM Details

Module	Port/PIM	Port Type	Interface	Data Element
SwMC_TestITC_DAI	P_Dai_PercBatSohPredn	Provided Port	IF_Dai_PercBatSohPredn	Dai_PercBatSohPredn
	R_Ipa_PwrChrng	Required Port	IF_Ipa_PwrChrng	Ipa_PwrChrng
	R_Ipa_PwrDcha	Required Port	IF_Ipa_PwrDcha	Ipa_PwrDcha
	R_Ipa_TpBat	Required Port	IF_Ipa_TpBat	Ipa_TpBat
	R_Ipa_PercSocBatMax	Required Port	IF_Ipa_PercSocBatMax	Ipa_PercSocBatMax
	R_Ipa_PercSocBatDod	Required Port	IF_Ipa_PercSocBatDod	Ipa_PercSocBatDod
	R_Ipa_EgyOutpBat	Required Port	IF_Ipa_EgyOutpBat	Ipa_EgyOutpBat
	R_Tstr_FacInpDeSclfctr_P	Local Parameter	IF_Tstr_FacInpDeSclfctr_P	Tstr_FacInpDeSclfctr_P
	R_Tstr_FacInpMeanSclfctr_P	Local Parameter	IF_Tstr_FacInpMeanSclfctr_	Tstr_FacInpMeanSclfctr_P
	R_Tstr_FacOutpMeanSclfctr_P	Local Parameter	IF_Tstr_FacOutpMeanSclfct	Tstr_FacOutpMeanSclfctr_F
	R_Tstr_FacOutpDeSclfctr_P	Local Parameter	IF_Tstr_FacOutpDeSclfctr_P	Tstr_FacOutpDeSclfctr_P

Figure 5: Sample interface definition in PREEvision for Simulink

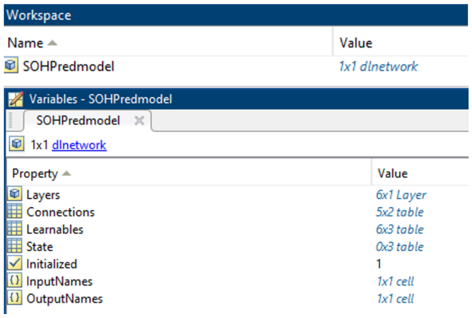
The process starts by loading arxml files into MATLAB using the AUTOSAR Component Designer, as shown in Figure 6a. This tool interprets the interface definitions and creates a frame model, providing a structured foundation for development. This step is crucial as it ensures that all communication protocols and data types are accurately represented, facilitating seamless integration with other system components..

```
    arxml_obj = arxml.importer('SwMC_TestITC_DAI
        .arxml')
    arxml_comp = arxml_obj.getComponentNames
    arxml_obj.createComponentAsModel(arxml_comp
        {1},'ModelPeriodicRunnablesAs','
        FunctionCallSubsystem')
```

(a) Import arxml and generate frame model

```
    SOHPredmodel = importNetworkFromTensorFlow('
        path_to_.pb_model')
    analyzeNetwork(SOHPredmodel)
    exportNetworkToSimulink(SOHPredmodel)
```

(b) Import and analyze .pb model into MATLAB workspace



Workspace

Name	Value
SOHPredmodel	1x1 dlnetwork

Variables - SOHPredmodel

SOHPredmodel

1x1 dlnetwork

Property	Value
Layers	6x1 Layer
Connections	5x2 table
Learnables	6x3 table
State	0x3 table
Initialized	1
InputNames	1x1 cell
OutputNames	1x1 cell

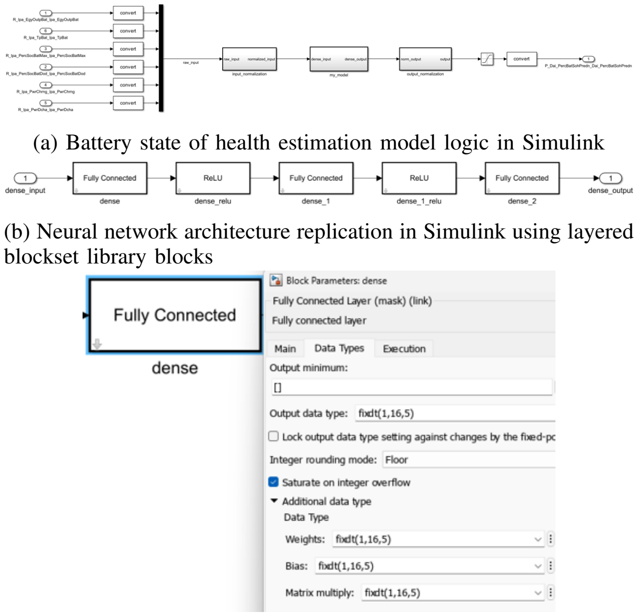
(c) MATLAB workspace with imported deep learning model

Figure 6: Sample code to import trained model from Python TensorFlow to MATLAB workspace

Next, the Simulink model is developed using the Deep Learning Toolbox. This automated approach converts the deep learning model in .pb format into a Simulink model via an

API, preserving the architecture and the values of weights and biases, as illustrated in Figure 6b and Figure 6c. This conversion is essential for maintaining the integrity of the deep learning model during deployment. MATLAB's functions for loading TensorFlow, PyTorch, ONNX models enable the use of its powerful computational and visualization tools for detailed analysis, simulation, and deployment, ensuring robust performance and accurate results within the automotive systems.

The Deep Learning Toolbox provides a comprehensive suite of blocks for constructing and training neural networks. It includes input layer normalization techniques such as Zero-Center, Zscore, and Rescale, applicable in 1D, 2D, and 3D formats. The toolbox offers convolution layers in 1D, 2D, and 3D, along with various pooling layers, including max, average, and global pooling. It supports a range of activation functions like ReLU, Leaky ReLU, Clipped ReLU, Tanh, Sigmoid, and Softmax, as well as normalization layers such as batch and layer normalization. Additionally, the toolbox provides sequence layers like LSTM and Flatten, combination layers for addition, multiplication, and concatenation, fully connected layers, and utility layers like Dropout. Post-processing capabilities are also included to identify label indices, making the toolbox a versatile and powerful tool for deep learning applications.


1
R_Ipa_EgyOutpBat_Ipa_EgyOutpBat
6
R_Ipa_TpBat_Ipa_TpBat
3
R_Ipa_PercSocBatMax_Ipa_PercSocBatMax
2
R_Ipa_PercSocBatDod_Ipa_PercSocBatDod
4
R_Ipa_PwrChrng_Ipa_PwrChrng
5
R_Ipa_PwrDcha_Ipa_PwrDcha
convert
raw_input
raw_input
normalized_input
input_normalization
dense_input
dense_output
my_model
norm_output
output
output_normalization
convert
1
P_Dai_PercBatSohPredn_Dai_PercBatSohPredn

(a) Battery state of health estimation model logic in Simulink

(b) Neural network architecture replication in Simulink using layered blockset library blocks

Block Parameters: dense
Fully Connected Layer (mask) (link)
Fully connected layer

Output minimum: []

Output data type: fixdt(1,16,5)

□ Lock output data type setting against changes by the fixed-pc

Integer rounding mode: Floor

☑ Saturate on integer overflow

▾ Additional data type

Data Type

Weights: fixdt(1,16,5)

Bias: fixdt(1,16,5)

Matrix multiply: fixdt(1,16,5)

(c) Fixed point scaling configuration for layered blockset library blocks

Figure 7: Simulink model for inferencing, includes logic for hidden layer and activation

To integrate the trained deep learning model into Simulink, blocks like the Predict block are used, but they function as a black box, hiding internal workings. Figure 7 illustrates an alternative approach using a layered blockset, which unrolls the neural network into individual layers. This method allows for monitoring signals after each layer and easier debugging, providing greater transparency and control over the model's operation. Additionally, parameters and operations can be optimized for fixed-point implementation using the Fixed-Point Designer, which is not possible with the Predict block. This optimization is crucial for ensuring efficient and reliable performance in automotive applications. The model logic mirrors the Python implementation, including tasks such as normalizing input values, performing neural network inferencing, and denormalizing outputs. This approach ensures that the deep learning model functions accurately and effectively within the Simulink environment, providing robust and precise results.

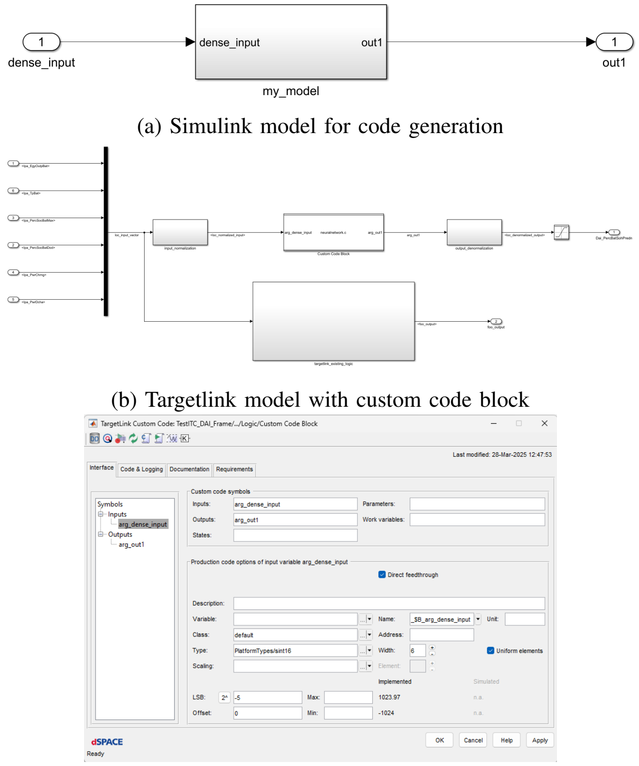
To ensure that the generated code is optimized for the specific hardware, detailed information about the target hardware platform is provided. This includes configuring hardware settings in the Simulink model, specifying the target hardware, clock speed, memory constraints, and other hardware-specific parameters. This ensures that the generated code is tailored to the capabilities and limitations of the target hardware. Additionally, the code generation configuration is set up to ensure adherence to AUTOSAR standards. This involves selecting the appropriate AUTOSAR schema, specifying optimization levels, data type settings, and naming conventions. Memory code sections are defined in the custom storage class designer, setting attributes such as memory section name, type, and alignment to optimize access speed and memory usage.

Finally, Embedded Coder is utilized to generate C-code that adheres to AUTOSAR standards. Embedded Coder offers advanced code generation capabilities, ensuring that the produced code is efficient, reliable, and compliant with industry standards. This comprehensive process ensures that the deep learning model is effectively integrated and executed within the automotive system, providing robust and accurate state of health estimation for the battery.

3) Hybrid approach: The hybrid approach integrates deep learning models developed using Simulink into existing software logic in TargetLink. The model is trained and quantized in Python, then converted into a dlnetwork object via an API, as shown in Figure 6b. This API directly generates a Simulink model, similar to the one illustrated in Figure 8a, eliminating the need to create a separate frame model for this specific approach. From this Simulink model, a custom C-code is generated using Embedded Coder, ensuring efficient and reliable code tailored to hardware requirements.

The conversion of the Python-trained and quantized model into a dlnetwork object leverages specialized APIs designed to bridge the gap between high-level deep learning frameworks and embedded deployment environments. During this conversion, the API meticulously translates the network architecture, weights, and biases into a format compatible with Simulink's code generation capabilities. This includes optimizing the model for fixed-point arithmetic, which is often essential for resource-constrained embedded systems. The API also handles the creation of necessary input and output interfaces within the generated Simulink model, simplifying subsequent integration.

Once the custom C-code is generated, it is integrated into the existing TargetLink model using S-functions, which serve as an interface between the TargetLink model and the custom code, as shown in Figure 8b. Figure 8c illustrates the necessary configurations for integration, ensuring that the custom code can correctly receive and send data within the TargetLink environment. These configurations include matching input and output widths to the model's dimensions, ensuring data type compatibility, and defining scaling factors for fixed-point representations. The S-function acts as a wrapper, allowing the hand-coded or generated C-code to behave like a standard TargetLink block, enabling seamless simulation and code generation within the TargetLink ecosystem. This intricate integration process ensures that the deep learning model operates harmoniously within the existing control system, leveraging the established verification and validation workflows of TargetLink.



(a) Simulink model for code generation

(b) Targetlink model with custom code block

TargetLink Custom Code: TestITC_DAI_Frame/.../Logic/Custom Code Block

Last modified: 28-Mar-2025 12:47:53

Interface | Code & Logging | Documentation | Requirements

Symbols
- Inputs
  - arg_dense_input
- Outputs
  - arg_out1

Custom code symbols

Inputs: arg_dense_input
Outputs: arg_out1
States:
Parameters:
Work variables:

Production code options of input variable arg_dense_input

☑ Direct feedthrough

Description:
Variable:
Name: _$B_arg_dense_input
Unit:
Class: default
Address:
Type: PlatformTypes/sint16
Width: 6
☑ Uniform elements
Scaling:
Element:

			Implemented	Simulated
LSB:	2^ -5	Max:	1023.97	n.a.
Offset:	0	Min:	-1024	n.a.

dSPACE
Ready

OK | Cancel | Help | Apply

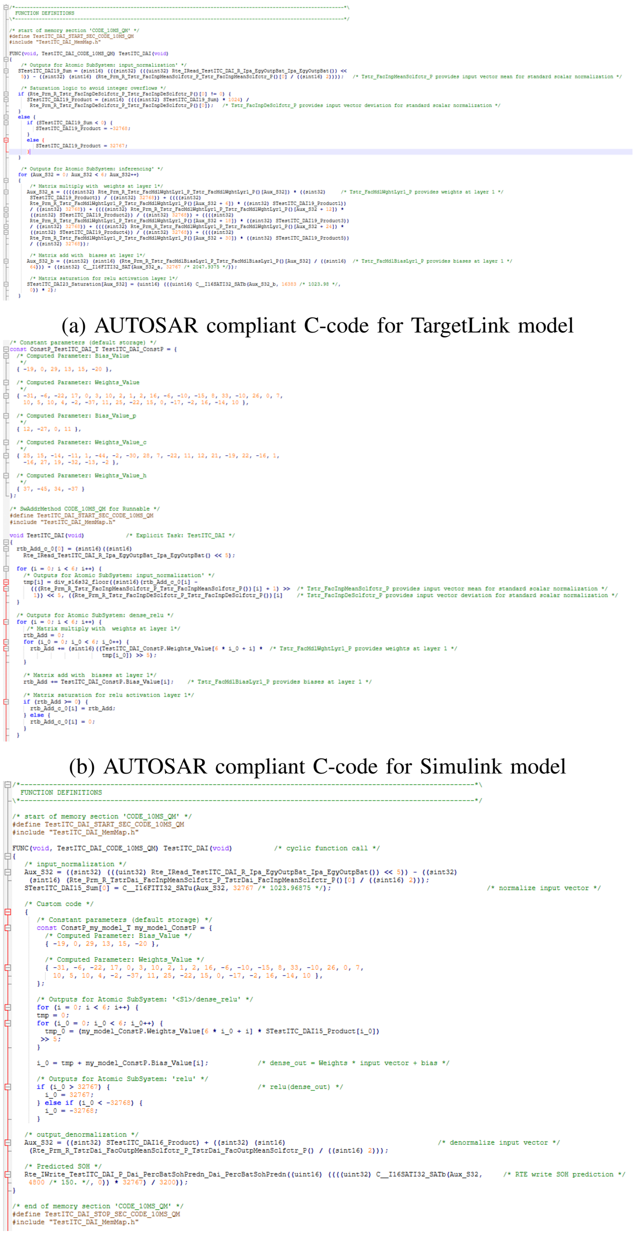
(c) Custom code block configuration

Figure 8: Targetlink model with custom code for deep learning model

D. Code

When integrating deep learning models into AUTOSAR-compliant systems, the generated code must meet specific standards for interoperability and efficiency.

Here's a breakdown of the key components:



```
/*---------------------------------------------------------------------------------------------*\
  FUNCTION DEFINITIONS
\*---------------------------------------------------------------------------------------------*/

/* start of memory section 'CODE_10MS_QM' */
#define TestITC_DAI_START_SEC_CODE_10MS_QM
#include "TestITC_DAI_MemMap.h"

FUNC(void, TestITC_DAI_CODE_10MS_QM) TestITC_DAI(void)
{
   /* Outputs for Atomic SubSystem: input_normalization' */
  STestITC_DAI19_Sum = (sint16) (((sint32) (((uint32) Rte_IRead_TestITC_DAI_R_Ipa_EgyOutpBat_Ipa_EgyOutpBat()) <<
   5)) - ((sint32) (sint16) (Rte_Prm_R_Tstr_FacInpMeanSclfctr_P_Tstr_FacInpMeanSclfctr_P()[0] / ((sint16) 2))));   /* Tstr_FacInpMeanSclfctr_P provides input vector mean for standard scalar normalization */

   /* Saturation logic to avoid integer overflows */
  if (Rte_Prm_R_Tstr_FacInpDeSclfctr_P_Tstr_FacInpDeSclfctr_P()[0] != 0) {
    STestITC_DAI19_Product = (sint16) ((((sint32) STestITC_DAI19_Sum) * 1024) /
     Rte_Prm_R_Tstr_FacInpDeSclfctr_P_Tstr_FacInpDeSclfctr_P()[0]);   /* Tstr_FacInpDeSclfctr_P provides input vector deviation for standard scalar normalization */
  }
  else {
    if (STestITC_DAI19_Sum < 0) {
      STestITC_DAI19_Product = -32768;
    }
    else {
      STestITC_DAI19_Product = 32767;
    }
  }

   /* Outputs for Atomic SubSystem: inferencing' */
  for (Aux_S32 = 0; Aux_S32 < 6; Aux_S32++)
  {
     /* Matrix multiply with  weights at layer 1*/
    Aux_S32_a = ((((sint32) Rte_Prm_R_Tstr_FacMdlWghtLyr1_P_Tstr_FacMdlWghtLyr1_P()[Aux_S32]) * ((sint32)     /* Tstr_FacMdlWghtLyr1_P provides weights at layer 1 */
     STestITC_DAI19_Product)) / ((sint32) 32768)) + ((((sint32)
     Rte_Prm_R_Tstr_FacMdlWghtLyr1_P_Tstr_FacMdlWghtLyr1_P()[Aux_S32 + 6]) * ((sint32) STestITC_DAI19_Product1))
     / ((sint32) 32768)) + ((((sint32) Rte_Prm_R_Tstr_FacMdlWghtLyr1_P_Tstr_FacMdlWghtLyr1_P()[Aux_S32 + 12]) *
     ((sint32) STestITC_DAI19_Product2)) / ((sint32) 32768)) + ((((sint32)
     Rte_Prm_R_Tstr_FacMdlWghtLyr1_P_Tstr_FacMdlWghtLyr1_P()[Aux_S32 + 18]) * ((sint32) STestITC_DAI19_Product3))
     / ((sint32) 32768)) + ((((sint32) Rte_Prm_R_Tstr_FacMdlWghtLyr1_P_Tstr_FacMdlWghtLyr1_P()[Aux_S32 + 24]) *
     ((sint32) STestITC_DAI19_Product4)) / ((sint32) 32768)) + ((((sint32)
     Rte_Prm_R_Tstr_FacMdlWghtLyr1_P_Tstr_FacMdlWghtLyr1_P()[Aux_S32 + 30]) * ((sint32) STestITC_DAI19_Product5))
     / ((sint32) 32768));

     /* Matrix add with  biases at layer 1*/
    Aux_S32_b = ((sint32) (sint16) (Rte_Prm_R_Tstr_FacMdlBiasLyr1_P_Tstr_FacMdlBiasLyr1_P()[Aux_S32] / ((sint16)  /* Tstr_FacMdlBiasLyr1_P provides biases at layer 1 */
     64))) + ((sint32) C__I16FITI32_SAT(Aux_S32_a, 32767 /* 2047.9375 */));

     /* Matrix saturation for relu activation layer 1*/
    STestITC_DAI23_Saturation[Aux_S32] = (uint16) (((uint16) C__I16SATI32_SATb(Aux_S32_b, 16383 /* 1023.98 */,
     0)) * 2);
  }
```

(a) AUTOSAR compliant C-code for TargetLink model

```
/* Constant parameters (default storage) */
const ConstP_TestITC_DAI_T TestITC_DAI_ConstP = {
  /* Computed Parameter: Bias_Value
   */
  { -19, 0, 29, 13, 15, -20 },

  /* Computed Parameter: Weights_Value
   */
  { -31, -6, -22, 17, 0, 3, 10, 2, 1, 2, 16, -6, -10, -15, 8, 33, -10, 26, 0, 7,
    10, 5, 10, 4, -2, -37, 11, 25, -22, 15, 0, -17, -2, 16, -14, 10 },

  /* Computed Parameter: Bias_Value_p
   */
  { 12, -27, 0, 11 },

  /* Computed Parameter: Weights_Value_c
   */
  { 25, 15, -14, -11, 1, -44, -2, -30, 28, 7, -22, 11, 12, 21, -19, 22, -16, 1,
    -16, 27, 19, -32, -13, -2 },

  /* Computed Parameter: Weights_Value_h
   */
  { 37, -45, 34, -37 }
};

/* SwAddrMethod CODE_10MS_QM for Runnable */
#define TestITC_DAI_START_SEC_CODE_10MS_QM
#include "TestITC_DAI_MemMap.h"

void TestITC_DAI(void)            /* Explicit Task: TestITC_DAI */
{
  rtb_Add_c_0[0] = (sint16)((sint16)
    Rte_IRead_TestITC_DAI_R_Ipa_EgyOutpBat_Ipa_EgyOutpBat() << 5);

  for (i = 0; i < 6; i++) {
    /* Outputs for Atomic SubSystem: input_normalization' */
    tmp[i] = div_s16s32_floor((sint16)(rtb_Add_c_0[i] -
      (((Rte_Prm_R_Tstr_FacInpMeanSclfctr_P_Tstr_FacInpMeanSclfctr_P())[i] + 1) >>  /* Tstr_FacInpMeanSclfctr_P provides input vector mean for standard scalar normalization */
       1)) << 5, ((Rte_Prm_R_Tstr_FacInpDeSclfctr_P_Tstr_FacInpDeSclfctr_P())[i]    /* Tstr_FacInpDeSclfctr_P provides input vector deviation for standard scalar normalization */
  }

  /* Outputs for Atomic SubSystem: dense_relu */
  for (i = 0; i < 6; i++) {
     /* Matrix multiply with  weights at layer 1*/
    rtb_Add = 0;
    for (i_0 = 0; i_0 < 6; i_0++) {
      rtb_Add += (sint16)((TestITC_DAI_ConstP.Weights_Value[6 * i_0 + i] *  /* Tstr_FacMdlWghtLyr1_P provides weights at layer 1 */
                          tmp[i_0]) >> 5);
    }

    /* Matrix add with  biases at layer 1*/
    rtb_Add += TestITC_DAI_ConstP.Bias_Value[i];    /* Tstr_FacMdlBiasLyr1_P provides biases at layer 1 */

    /* Matrix saturation for relu activation layer 1*/
    if (rtb_Add >= 0) {
      rtb_Add_c_0[i] = rtb_Add;
    } else {
      rtb_Add_c_0[i] = 0;
    }
  }
```

(b) AUTOSAR compliant C-code for Simulink model

```
/*---------------------------------------------------------------------------------------------*\
  FUNCTION DEFINITIONS
\*---------------------------------------------------------------------------------------------*/

/* start of memory section 'CODE_10MS_QM' */
#define TestITC_DAI_START_SEC_CODE_10MS_QM
#include "TestITC_DAI_MemMap.h"

FUNC(void, TestITC_DAI_CODE_10MS_QM) TestITC_DAI(void)          /* cyclic function call */
{
   /* input_normalization */
   Aux_S32 = ((sint32) (((uint32) Rte_IRead_TestITC_DAI_R_Ipa_EgyOutpBat_Ipa_EgyOutpBat()) << 5)) - ((sint32)
    (sint16) (Rte_Prm_R_TstrDai_FacInpMeanSclfctr_P_TstrDai_FacInpMeanSclfctr_P()[0] / ((sint16) 2)));
   STestITC_DAI15_Sum[0] = C__I16FITI32_SATu(Aux_S32, 32767 /* 1023.96875 */);                        /* normalize input vector */

   /* Custom code */
   {
     /* Constant parameters (default storage) */
     const ConstP_my_model_T my_model_ConstP = {
       /* Computed Parameter: Bias_Value */
       { -19, 0, 29, 13, 15, -20 },

       /* Computed Parameter: Weights_Value */
       { -31, -6, -22, 17, 0, 3, 10, 2, 1, 2, 16, -6, -10, -15, 8, 33, -10, 26, 0, 7,
         10, 5, 10, 4, -2, -37, 11, 25, -22, 15, 0, -17, -2, 16, -14, 10 },
     };

     /* Outputs for Atomic SubSystem: '<S1>/dense_relu' */
     for (i = 0; i < 6; i++) {
     tmp = 0;
     for (i_0 = 0; i_0 < 6; i_0++) {
       tmp_0 = (my_model_ConstP.Weights_Value[6 * i_0 + i] * STestITC_DAI15_Product[i_0])
      >> 5;
     }

     i_0 = tmp + my_model_ConstP.Bias_Value[i];           /* dense_out = Weights * input vector + bias */

     /* Outputs for Atomic SubSystem: 'relu' */
     if (i_0 > 32767) {                                   /* relu(dense_out) */
       i_0 = 32767;
     } else if (i_0 < -32768) {
       i_0 = -32768;
     }

   /* output_denormalization */
   Aux_S32 = ((sint32) STestITC_DAI16_Product) + ((sint32) (sint16)                        /* denormalize input vector */
    (Rte_Prm_R_TstrDai_FacOutpMeanSclfctr_P_TstrDai_FacOutpMeanSclfctr_P() / ((sint16) 2)));

   /* Predicted SOH */
   Rte_IWrite_TestITC_DAI_P_Dai_PercBatSohPredn_Dai_PercBatSohPredn((uint16) ((((uint32) C__I16SATI32_SATb(Aux_S32,     /* RTE write SOH prediction */
    4800 /* 150. */, 0)) * 32767) / 3200));
}

/* end of memory section 'CODE_10MS_QM' */
#define TestITC_DAI_STOP_SEC_CODE_10MS_QM
#include "TestITC_DAI_MemMap.h"
```

(c) AUTOSAR compliant C-code for TargetLink model with custom code block

Figure 9: Comparison of code generated from three approaches

- Cyclic function: This core part of the software module executes the model's logic at regular intervals, ensuring timely input processing and output predictions. Implemented as a periodic task, it is triggered by a timer or scheduler within the AUTOSAR framework.
- Inference Logic: This section runs the deep learning inference, processing input data through neural network layers to generate predictions. It includes matrix multiplications, activation functions, and other operations for the neural

network's forward pass.

- Weights and biases: These parameters, learned during training, are crucial for model performance and accuracy. In the generated code, weights and biases are stored as variables or constants, typically defined in a separate section or dedicated data files for easy access and modification.

The key difference between code generated for deep learning models using TargetLink, Simulink and hybrid approach illustrated in figure 9 is how they handle weights and biases for the deep learning model. In TargetLink, these are retrieved through parameter interfaces via the Runtime Environment (RTE), allowing easy updates through .dcm or calibration files without altering the core code. In contrast, Simulink defines weights and biases as constants, requiring complete code regeneration and reflashing for any updates, making it less flexible for frequent changes.

E. Validation

After generating AUTOSAR-compliant code for your deep learning model, the next critical step is validation. This ensures the code meets quality standards, performs as expected, and integrates seamlessly into the automotive system. The validation process includes:

- Static Code Analysis: In this initial step, the code is examined without execution to identify issues such as coding standard violations, security vulnerabilities, and logical errors. Standard static code analysis tools, such as Polyspace or Astrée, are employed in this study to detect runtime errors, data races, and other critical issues in the embedded software.
- Unit Testing: In this step, individual components of the code are tested to ensure they function correctly in isolation. A standard unit testing tool, such as TESSY by Razorcat Development GmbH, is used in this study to create and execute test cases for each unit of the AUTOSAR code. This process generates detailed reports on test results and highlights any issues, ensuring the reliability and correctness of each code unit.
- Functionality testing: This step ensures that integrated software modules work together correctly. Standard tools can be employed to validate the software's overall functionality, analyzing detailed results and logs to ensure it meets specifications and performs as expected.

  In this study, Silver by Synopsys was used to perform Software-in-the-Loop testing in a simulated environment. During functionality testing, the C code is executed with the same inputs, and the outputs are compared against the ground truth from the physics-based model. This direct comparison verifies that the C code accurately replicates the behavior and performance of the original physics-based model. HIL testing, in this case using INCA software from ETAS, emphasizes on assessment of system's real-time behavior and robustness, ensuring that the model interacts correctly with hardware components and meets the required specifications under operational scenarios. This comprehensive approach guarantees that both software and hardware components function seamlessly together in real-world applications.

F. SW Integration

After validating your AUTOSAR-compliant deep learning code, the final step is integrating it into the larger existing software code base. This ensures the new software module is properly scheduled and built within the existing framework.

- Scheduling changes: Adjust the system's schedule to include the new module. Review and modify task periods, deadlines, and execution order to fit the new module without disrupting existing functionality.
- Integrated Arxml: The integrated arxml contains information regarding all the software modules in the code base and provides interfaces between these software modules. Use an AUTOSAR authoring tool to update the arxml file with definitions for the new module, its interfaces, and scheduling information. Ensure all dependencies and interactions are correctly specified.
- Integrate software module: Add the new module's source code and configuration files to the project directories. Update build scripts and configuration files to include the new module, resolving all dependencies and references.
- Trigger build pipeline: Use a CI/CD pipeline to automate the build process. Set up a pipeline with tools like Jenkins, GitLab CI, or Azure DevOps to compile the code, run tests, and generate final build artifacts.

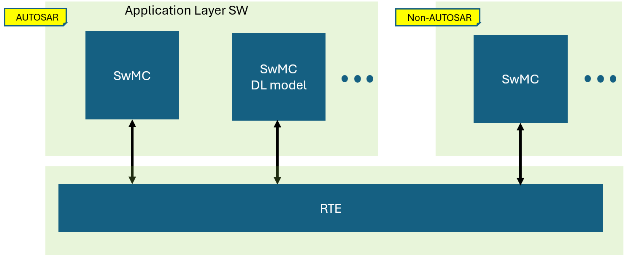


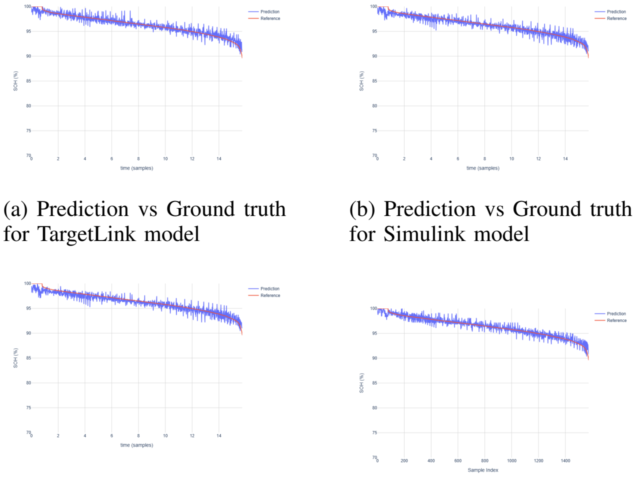
Figure 10: Integrating Deep learning model in existing ECU software with AUTOSAR and non-AUTOSAR modules

After development, the CI/CD pipeline generates A2L and S19 hex files, essential for flashing the software onto the ECU and ensuring correct integration. Subsequently, Hardware-in-the-Loop and vehicle tests are conducted to evaluate real-time behavior and monitor system performance on the target hardware, crucial for validating functionality and reliability in real-world conditions.

Once integrated into the ECU software, the deep learning module can seamlessly interact with other AUTOSAR or non-AUTOSAR application modules via the Runtime Environment as shown in Figure 10. This enables real-time data processing and analysis, enhancing ECU operations for versatile and adaptable automotive solutions.

III. Results

In this section, we present the results of the functionality test conducted in a Hardware-in-the-Loop environment for the battery degradation use case. The primary goal of this use case is to estimate the battery's state of health by taking into account various factors such as charging and discharging power, energy throughput, battery temperature, state of charge, and depth of discharge.



(a) Prediction vs Ground truth for TargetLink model

(b) Prediction vs Ground truth for Simulink model

(c) Prediction vs Ground truth for Targetlink model with custom code

(d) Prediction vs Ground truth for floating point python model

Figure 11: Comparison of Targetlink, Simulink, Hybrid approaches and floating point python deep learning model predictions with ground truth

The actual state of health is determined in the vehicle using a physics-based model, which serves as the benchmark for comparison. The deep learning model is developed to accurately replicate the results produced by the physics-based model. However, it is important to note that a detailed discussion of the physics-based model itself is beyond the scope of this paper. Our focus remains on the development and validation of the deep learning model to ensure it achieves comparable accuracy and reliability.

Initially, the deep learning model is trained in Python using historical data and relevant features. To optimize memory usage for deployment on the Electronic Control Unit , the model is quantized, reducing its size while maintaining performance. This paper compares three different modeling and code generation approaches: manual implementation with TargetLink, Simulink combined with the Deep Learning Toolbox, and a hybrid approach integrating Simulink-generated code into existing TargetLink logic using S-functions. The comparison focuses on fixed-point scaling and hardware optimization, which are critical for efficient ECU deployment.

Figure 11 compares the predictions of TargetLink, Simulink, hybrid approaches, and a floating-point deep learning model in Python against the ground truth from a physics-based model. All approaches use the same weights and biases from the Python-trained deep learning model, effectively replicating the physics-based model's results with minor performance differences. The similar predictions and ground truth indicate comparable accuracy, suggesting all methods are viable for estimating battery SoH in automotive applications. Additionally, fixed-point scaling and hardware optimization techniques ensure the models are suitable for real-time deployment on ECUs, balancing accuracy and computational efficiency.

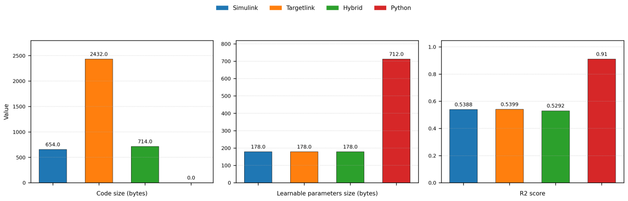


Figure 12: Resource utilization and performance of Simulink and TargetLink approaches

Additionally, Figure 12 compares resource utilization and performance across different approaches, including the floating-point deep learning model in Python, considering code size, learnable parameters size, and R² score. Resource utilization encompasses code size (memory required for executable code) and learnable parameters size (memory needed for model parameters); the R² score indicates how well the model's predictions match actual data.

The Python-based floating-point model demonstrates exceptional performance with an R² score of 0.91 and utilizes 712 bytes for its learnable parameters. However, its significant computational resource demands make it unsuitable for Electronic Control Units, which necessitate fixed-point models for real-time application viability. Notably, fixed-point implementations across the three approaches require only 178 bytes of data for learnable parameters stored in ROM, making them far more appropriate for such resource-constrained environments.

Among these fixed-point approaches for ECUs, Simulink's optimized code generation results in a code size of 654 bytes and an R² score of 0.5388. TargetLink, with a larger code size of 2432 bytes due to manual saturation logic and robust error handling, achieves a slightly higher R² score of 0.5399. The Hybrid approach, combining elements of both, has a code size of 714 bytes and a lower R² score of 0.5292, which may be affected by interface inefficiencies via the S-function.

These results highlight Python's performance strength but limitations in ECU deployment, while Simulink, TargetLink, and Hybrid offer practical solutions for real-time applications with varying trade-offs in resource utilization and accuracy.

During Hardware-in-the-Loop testing, the latency observed was minimal, with output results accurately reflected at each sample time, and the ECU load remained insignificant. However, the ECU load may increase proportionally with neural network complexity, potentially impacting performance.

IV. Conclusion

This study demonstrates the feasibility and effectiveness of deploying deep learning models on automotive Electronic Control Units using AUTOSAR-compliant C-code. Focusing on battery state of health estimation, we show that deep learning models can be trained in Python, quantized for memory optimization, and implemented using three approaches: manual implementation with TargetLink, Simulink combined with the Deep Learning Toolbox, and a hybrid approach integrating Simulink-generated code into existing TargetLink logic using S-functions.

Integrating deep learning models into automotive systems via TargetLink, Simulink, or a hybrid approach presents distinct challenges. TargetLink necessitates manually defining network weights and parameters as Rte calls and building the model manually through basic arithmetic blocks, making it increasingly difficult as network size and complexity grow (e.g., for CNNs or RNNs). Conversely, Simulink simplifies this by directly importing weights from the MATLAB workspace's ‘dlnetwork‘ object as constants and automatically creating a Simulink subsystem regardless of the network's complexity or size.

TargetLink allows easy updates via calibration files for weights and biases, making it ideal for long-term maintenance. However, it is labor-intensive, requiring manual design of each hidden layer, which increases potential errors and effort. Simulink requires complete code regeneration for updates but offers high automation, reducing manual effort and errors, thus facilitating rapid development and deployment for time-sensitive projects. The hybrid approach balances automation with flexibility, using Simulink for model development and Embedded Coder for code generation, then integrating the code into TargetLink via S-functions, preserving existing logic without needing parameter interfaces.

Both Simulink and TargetLink approaches require manual handling of project-related settings, such as target hardware configuration and code generation configurations, to optimize the generated code for specific hardware and ensure adherence to project standards. The hybrid approach similarly requires careful configuration to integrate custom code in S-functions. Key differences among the TargetLink, Simulink, and hybrid approaches are summarized in Table I, emphasizing their methods for handling deep learning network weights and parameters, the ease of updating these weights, the level of automation involved, and the overall efficiency in development and deployment.

The Hardware-in-the-Loop tests indicate that TargetLink, Simulink, and hybrid approaches can accurately replicate the results of a physics-based model, with comparable R² scores. Resource utilization analysis reveals that all three approaches successfully integrate the floating-point deep learning model trained in Python into ECUs with reduced resource demands due to quantization and fixed-point implementation.

Overall, this study emphasizes the potential of deep learning in automotive applications, offering key insights into converting models from frameworks like TensorFlow into production-quality AUTOSAR-compliant C code. It showcases the effective integration and deployment of these models on ECUs, ensuring robust performance and reliability.

Aspect	**TargetLink**	**Simulink**	**Hybrid**
Training	Python	Python	Python
Modeling	Manual, TargetLink blocks	Automated, Simulink & Deep Learning Toolbox	Automated, Simulink & Deep Learning Toolbox, retains TargetLink logic
Code Generation	TargetLink AUTOSAR	Embedded Coder	Embedded Coder, integrated via S-functions
Parameters	Defined as parameter port interfaces, Rte calls	Imported as constants from MATLAB workspace	Imported as constants, no parameter interfaces
Updating Weights	Easy via calibration files	Requires full code regeneration	Requires full code regeneration
Automation	Manual, high effort	High, reduces effort	High, balances automation and manual control
Development	Labor-intensive, prone to errors	Rapid, suitable for fast deployment	Efficient, combines automated code with existing logic
Resource Utilization	Larger code size, robust error handling	Optimized code size	Moderate code size, interface inefficiencies
Error Handling	Robust	Standard	Standard
Maintenance	Easy with calibration files	Complex due to regeneration	Complex due to regeneration
User Expertise Required	High	Medium	Medium

TABLE I: Comparison of TargetLink, Simulink, and Hybrid Approaches

Future research could explore the scalability of these integration approaches to support more complex and larger deep neural networks on ECUs. It would be particularly insightful to investigate the limits of current methodologies when faced with models demanding higher computational and memory resources. Furthermore, extending beyond mere inference, there's a growing need for edge learning capabilities on ECUs. This would involve enabling models to continually adapt and learn from new data directly on the device, rather than relying solely on pre-trained models. This shift to on-device learning could unlock new possibilities for autonomous systems, enhancing their adaptability and robustness in dynamic real-world environments.

Acknowledgment

We extend our heartfelt gratitude to Mercedes-Benz Research and Development India for the invaluable support and contributions to this study. The expertise and resources provided have been essential to the successful completion of this research. We also wish to thank MathWorks for the collaboration and support, particularly with Simulink and the Deep Learning Toolbox, which have been pivotal in the development and deployment of our deep learning models. This collaboration has greatly enhanced the quality and impact of our work.

References

[1] S. M. et al., ”A novel framework for deployment of CNN models using post-training quantization on micro-controller,” *Microprocessors and Microsystems*, vol. 94, 2022, Art. no. 104634, doi: 10.1016/j.micpro.2022.104634.

[2] G. Cerutti et al., ”Sound Event Detection with Binary Neural Networks on Tightly Power-Constrained IoT Devices,” in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design (ISLPED)*, Boston, NY, USA, 2020, pp. 1-6, doi: 10.1145/3370748.3406588.

[3] L. Falaschetti et al., ”ECG-Based Arrhythmia Classification using Recurrent Neural Networks in Embedded Systems,” *Procedia Comput. Sci.*, vol. 207, pp. 3479-3487, 2022, doi: 10.1016/j.procs.2022.09.406.

[4] R. Perego et al., ”AutoTinyML for micro-controllers: Dealing with black-box deployability,” *Expert Syst. Appl.*, vol. 207, 2022, Art. no. 117876, doi: 10.1016/j.eswa.2022.117876.

[5] X. Yu et al., ”A practical wearable fall detection system based on tiny convolutional neural networks,” *Biomed. Signal Process. Control*, vol. 86, 2023, Art. no. 105325, doi: 10.1016/j.bspc.2023.105325.

[6] L. Falaschetti et al., ”A Lightweight CNN-Based Vision System for Concrete Crack Detection on a Low-Power Embedded micro-controller Platform,” *Procedia Comput. Sci.*, vol. 207, pp. 3948-3956, 2022, doi: 10.1016/j.procs.2022.09.457.

[7] A. Ghibellini et al., ”Intelligence at the IoT Edge: Activity Recognition with Low-Power Microcontrollers and Convolutional Neural Networks,” in *Proc. IEEE 19th Annu. Consum. Commun. Netw. Conf. (CCNC)*, Las Vegas, NV, USA, 2022, pp. 707-710, doi: 10.1109/CCNC49033.2022.9700665.

[8] F. Meißl et al., ”Online Handwriting Recognition using LSTM on micro-controller and IMU Sensors,” in *Proc. 21st IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Nassau, Bahamas, 2022, pp. 999-1004, doi: 10.1109/ICMLA55696.2022.00167.

[9] S. El Fallah et al., ”State of charge estimation of an electric vehicle's battery using Deep Neural Networks: Simulation and experimental results,” *J. Energy Storage*, vol. 62, 2023, Art. no. 106904, doi: 10.1016/j.est.2023.106904.

[10] K. Dutta et al., ”Evaluation of Lithium Ion Battery State of Health Using Deep Leaning Methods,” in *Proc. IEEE 4th Int. Conf. Sustain. Energy Future Electr. Transp. (SEFET)*, 2024, pp. 1-6, doi: 10.1109/SEFET61574.2024.10718085.

[11] A. Gozuoglu et al., ”CNN-LSTM based deep learning application on Jetson Nano: Estimating electrical energy consumption for future smart homes,” *Internet Things*, vol. 26, 2024, Art. no. 101148, doi: 10.1016/j.iot.2024.101148.

[12] D. Nadalinis et al., ”Reduced precision floating-point optimization for Deep Neural Network On-Device Learning on microcontrollers,” *Future Gener. Comput. Syst.*, vol. 149, pp. 212-226, 2023, doi: 10.1016/j.future.2023.07.020.