



CHALMERS
UNIVERSITY OF TECHNOLOGY



Optimization of Battery AI Models for Edge Deployment: A Systematic Study of Model Compression Techniques and Hardware-Aware Efficiency

Master's thesis in Embedded Electronic System Design

Yerui Liu
Yuxin Xia

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

MASTER'S THESIS 2026

Optimization of Battery AI Models for Edge Deployment: A Systematic Study of Model Compression Techniques and Hardware-Aware Efficiency

Yerui Liu
Yuxin Xia



Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Optimization of Battery AI Models for Edge Deployment: A Systematic Study of
Model Compression Techniques and Hardware-Aware Efficiency

Yerui Liu

Yuxin Xia

© Yerui Liu, Yuxin Xia, 2026.

Supervisor: Torbjörn Thiringer, Department of Electrical Engineering

Company advisor: Christian Fleischer, Cognivity AI

Examiner: Lena Peterson, Department of Microtechnology and Nanoscience

Master's Thesis 2026

Department of Microtechnology and Nanoscience

Chalmers University of Technology

SE-412 96 Gothenburg

Telephone +46 31 772 1000

Typeset in L^AT_EX

Gothenburg, Sweden 2026

Optimization of Battery AI Models for Edge Deployment: A Systematic Study of Model Compression Techniques and Hardware-Aware Efficiency

Yerui Liu

Yuxin Xia

Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Battery management systems (BMS) increasingly employ data-driven models to estimate battery state of health (SoH), but deploying such models on automotive microcontrollers remains challenging because of limited memory and computational resources. This thesis evaluates a hardware-aware workflow for transferring, compressing, and preparing a long short-term memory (LSTM)-based SoH estimation model for embedded execution. The baseline model and its original training were provided by Cognivity AI; model development is therefore outside the scope of this thesis.

The study uses the provided model and its established preprocessing pipeline, with battery-level data separation for evaluation. The workflow covers PyTorch to MATLAB model transfer, numerical equivalence verification, pruning and quantization evaluation, Simulink integration, and embedded C++ code generation.

Two complementary compression directions are evaluated. Structured pruning reduces the number of learnable parameters while retaining useful predictive accuracy in the reported experiment. INT8 post-training quantization (PTQ) and quantization-aware training (QAT) reduce the theoretical weight storage to approximately one quarter of the FP32 value; QAT performs better than PTQ for the unpruned baseline in the reported comparison. Compression configurations are selected with validation data and evaluated on the held-out test data only after they are frozen.

The selected models are exported to Simulink for model-in-the-loop functional checks, and statically allocated C++ code is generated using Embedded Coder. The generated artifacts were cross-compiled for the Infineon KIT_A3G_TC4D7_LITE evaluation platform. These results demonstrate conversion and compilation feasibility, but do not establish INT8 arithmetic in the generated implementation, on-target execution time, power consumption, or completed hardware deployment. Those measurements remain future work.

Keywords: Battery Management System, Deep Learning, Edge Computing, Model Pruning, Model Compression, Quantization-Aware Training

Acknowledgements

Time flies, and before we knew it, our two-year journey of studying in Sweden drew to a close. These two years have been a truly unique experience—leaving our respective hometowns to pursue our studies in a country with a completely different culture. Fortunately, we managed to overcome various challenges and successfully complete this master’s thesis. First, we would like to thank our company advisor, Christian; we are grateful to him and Cognivity AI for providing this project and offering invaluable support regarding hardware, software, and conceptual guidance—his contribution to the project was immense. We also wish to thank our Chalmers advisor, Torbjörn, for providing practical and insightful advice on maintaining proper thesis standards and ensuring academic rigor. Additionally, we thank our examiner, Lena, for offering constructive feedback on every report and proposal we submitted.

Finally, we would like to thank everyone who has taken the time to read our thesis; thank you for your interest and your feedback.

Yerui Liu, Yuxin Xia, Gothenburg, June 2026

List of Abbreviations

Abbreviation	Definition
AI	Artificial intelligence
BMS	Battery management system
CC	Constant current
CNN	Convolutional neural network
CPU	Central processing unit
CR	Compression ratio
CV	Constant voltage
DSPR	Data scratchpad RAM
ECM	Equivalent circuit model
FP32	32-bit floating-point representation
INT8	8-bit integer representation
LCO	Lithium cobalt oxide
LFP	Lithium iron phosphate
LSTM	Long short-term memory
MAE	Mean absolute error
MAPE	Mean absolute percentage error
MCU	Microcontroller unit
NMC	Nickel manganese cobalt oxide
OCV	Open-circuit voltage
PTQ	Post-training quantization
QAT	Quantization-aware training
RAM	Random-access memory
RC	Resistor-capacitor
ReLU	Rectified linear unit
RMSE	Root mean square error
RNN	Recurrent neural network
SG	Savitzky–Golay
SoC	State of charge
SoH	State of health
TCN	Temporal convolutional network

Contents

1	Introduction	1
1.1	Related Work	2
1.1.1	Evolution of Battery State Estimation Paradigm: From Electrochemical Physics Models to Machine Learning	2
1.1.2	Current Status of Research on Model Compression and Edge Optimization	3
1.1.3	Another possibility for edge acceleration of time series data: Temporal Convolutional Networks	4
1.2	Purpose and Goal	4
1.2.1	Purpose	4
1.2.2	Goal	4
1.3	Contribution	6
1.4	Organization of the Thesis	6
2	Technical Background	9
2.1	Battery Modeling Fundamentals	9
2.1.1	State of Charge	9
2.1.2	State of Health	9
2.1.3	Equivalent Circuit Model	10
2.2	Recurrent Neural Networks & LSTM	11
2.2.1	Basics of Artificial Neurons and Data Flow	11
2.2.2	Recurrent Neural Networks	12
2.2.3	Long Short-Term Memory	13
2.3	Compressing Method	14
2.3.1	Pruning	14
2.3.2	Quantization	15
2.4	Evaluation Framework	15
3	Method	17
3.1	Data and Evaluation Protocol	17
3.2	PyTorch-to-MATLAB Transfer	17
3.3	Compression Evaluation	17
3.4	Simulink and Code-Generation Checks	18

4	Design	19
4.1	Framework	19
4.2	Feature Layer	20
4.3	Model Layer	21
4.4	Compression Layer	22
4.4.1	Magnitude-Based Weight Pruning	22
4.4.2	INT8 Post-Training Quantization	24
4.5	Deployment Layer: Code generation and cross-compilation feasibility verification	25
5	Results	29
5.1	Data Preprocessing and Electrochemical Feature Engineering Result .	29
5.1.1	Dataset Overview	29
5.1.2	Preprocessing visualization	31
5.2	Baseline LSTM Performance	32
5.2.1	Numerical Equivalence Verification	32
5.2.2	Evaluation Metrics and Baseline Performance	32
5.3	Pruning Result	35
5.4	Result	37
5.5	Combination results	40
5.6	Simulink Closed-loop Verification	44
5.6.1	Real-Time Streaming Simulation and Verification	44
5.6.2	C++ Embedded Code Generation	47
5.7	Summary	47
6	Discussion	49
6.1	Interpretation of the Results	49
6.2	Deployment Scope and Limitations	49
6.3	Ethical Statement	49
7	Conclusion	51
	Bibliography	53

1

Introduction

As the global transportation industry accelerates its electrification transition, advanced battery intelligence technology has become a core driver for ensuring battery safety, efficiency, and optimizing lifespan.

Modern battery management systems (BMS) increasingly rely on data-driven methods to estimate internal battery states that are difficult to observe directly through physical sensors, such as State of Charge (SoC) and State of Health (SoH).

Deep learning models, particularly recurrent neural network (RNN) architectures such as long short-term Memory (LSTM) networks, have demonstrated exceptional efficiency in capturing the nonlinear electrochemical dynamics inherent in time-series measurements such as voltage and current. However, this improvement in prediction accuracy typically comes at the cost of a significant increase in model complexity, posing a significant challenge in the resource&energy-sensitive field of automotive BMS. An automotive BMS is a typical "always-on" system, continuously monitoring cell safety even when the vehicle is parked. Running highly complex deep learning models on low-power embedded platforms can lead to significant "phantom drain", which means unnecessary waste of battery energy, potentially depleting the main battery over weeks of inactivity.

In modern automotive electronic architectures, while centralized onboard supercomputing platforms, such as the NVIDIA Orin series, provide powerful GPU acceleration support for high-computing scenarios like autonomous driving, the main control unit of the BMS, responsible for battery safety, still widely employs automotive-grade microcontroller units (MCUs) [1]. This distributed architecture leads to a "supply-demand imbalance" between advanced data-driven artificial intelligence (AI) algorithms and extremely constrained underlying hardware.

The resource bottlenecks of automotive-grade MCUs are mainly reflected in the following two dimensions:

- Extreme limitations of the memory hierarchy: Automotive-grade MCUs provide substantially less memory than cloud or PC platforms. These constraints motivate evaluating model-storage reduction, but this thesis does not measure paging, latency, or power consumption on a target device.
- Limited arithmetic acceleration: LSTM inference includes matrix multiplications and nonlinear functions, while available hardware support varies by MCU. The actual arithmetic type and runtime behavior must therefore be verified on the target platform and are not claimed here.

1.1 Related Work

1.1.1 Evolution of Battery State Estimation Paradigm: From Electrochemical Physics Models to Machine Learning

Internal state estimation of lithium-ion batteries (including SoC and SoH) is not only a complex multi-scale electrochemical multiphysics modeling problem, but also, in engineering practice, a typical dynamic system state observation problem deeply coupled with weak signal processing, nonlinear feature extraction, and pattern recognition. Accurate and robust SoC and SoH estimations form the core underlying technology pillars of complex electronic systems such as active safety defenses for electric vehicles [2], long-range flight path planning for drones, and thermal runaway early warning systems for industrial energy storage systems. In recent years, benefiting from the emphasis placed on open source data and benchmarking by academia and industry, research based on battery state estimation has emerged in large numbers, including open source datasets, code libraries, and benchmark deep learning methods for lithium-ion battery SoH estimation [3].

From a technological evolution perspective, early battery state estimation mainly relied on empirical models and physical models based on electrochemical mechanisms (for example, the classic Physics-based Electrochemical Models [4] and Reduced-order Models [5]). Among these methods, the ampere-hour integration method (ampere-hour counting method) remains widely used as a benchmark due to its extremely simplified computational architecture and near-zero hardware resource consumption. However, this method is essentially an open-loop estimation topology, and its long-term accuracy is limited by sensor drift and initial measurement errors. Therefore, in industrial applications, Coulomb counting is rarely executed in a purely open-loop manner; instead, it is conventionally enforced with periodic corrections from time to time, such as open-circuit voltage (OCV) resets during prolonged rest periods, to recalibrate the baseline and suppress cumulative errors. Nevertheless, these intermittent corrections encounter practical bottlenecks under continuous and dynamic driving cycles, where prolonged rest windows are unavailable.

To introduce a closed-loop correction mechanism, methods based on equivalent circuit models (ECMs) have become the mainstream research approach in industry over the past decade. These methods typically utilize a resistor-capacitor (RC) network and a controlled voltage source to establish a lumped parameter model, and nest extended Kalman filter, unscented Kalman filter, or adaptive particle filter algorithms to dynamically update the battery’s internal state through the residual between the online measured terminal voltage and the theoretical model output. Although the ECM-Kalman filter pipeline exhibits excellent convergence boundary and local robustness under mild laboratory isothermal environment and standard charge and discharge conditions, it encounters significant practical physical obstacles when deployed at actual vehicle edge. Under complex dynamic loads and various degradation modes, the robustness of the physical equations is often difficult to guarantee.

With the explosive growth of historical log data from vehicle sensors and BMS, the

use of machine learning (ML) and modern deep learning (DL) paradigms to solve electrochemical state estimation has become an inevitable technological trend in the field of battery intelligence. Machine learning methods in BMS have made leaps and bounds, evolving from early simple linear regression, support vector machines, and Gaussian process regression [6], [7] to complex deep RNN and highly hybrid convolutional neural networks-long short-term memory networks (CNN-LSTM) [8]. These data-driven deep time series algorithms do not require explicit solution of complex physical differential equations, but instead transform the battery estimation problem into a nonlinear mapping between a high-dimensional time series feature space and scalar regression invariants. While these deep time-series models significantly outperform traditional Kalman filters in predictive accuracy and generalization performance, their advantages come at a cost. They demonstrate strong performance under dynamic operating conditions, wide temperature ranges, and nonlinear accelerated aging stages, particularly around the Knee Point. Deep compression techniques, such as pruning, quantization, and Huffman coding, have proven effective for CNNs in image processing. However, their application to deep time-series models remains challenging due to the increased model parameter size and computational overhead of these models. These complex models involve millions of matrix multiply-accumulate operations during runtime, and their high hardware storage size and computational deployment costs make them extremely difficult to directly port to resource-constrained underlying controllers. Furthermore, the application of these processes to time series regression in safety-critical "always-on" battery systems remains insufficiently investigated. Recent survey [9], [10] highlight the need for hardware-aware optimization specifically for embedded battery applications. Moreover, in the field of temporal prediction, temporal convolutional networks (TCN) has been considered to have advantages over RNN&LSTM at the edge in recent years because convolutional operations are easier to parallelize or accelerate on some hardware. TCN can be combined with LSTM [11].

1.1.2 Current Status of Research on Model Compression and Edge Optimization

In the general technology field of edge deployment of deep learning, in order to break down the natural physical barrier between "large models" and "lightweight hardware," the academic community has gradually formed a relatively mature "deep compression" pipeline. Han et al. systematically proposed a three-stage joint optimization paradigm that integrates "synaptic pruning, lossless quantization, and efficient Huffman coding" [12]. In the current field of computer vision (CV), this technology pipeline has achieved disruptive success and has been widely used in the spatial feature compression of various classic convolutional neural networks. It can reduce the physical storage volume of the model by tens or even hundreds of times while maintaining almost no loss in Top-1 or Top-5 classification accuracy, as confirmed by the research of some Chinese and South Korean scholars in recent years [13, 14].

However, a thorough analysis of the existing literature reveals that while deep compression of CNN models for image processing has reached a high level of sophis-

tication, directly applying these highly mature compression optimization pipelines to time series regression tasks in safety-critical, always-on battery management systems remains an extremely inadequate, unsystematic, and even fragmented research stage in the current international academic and engineering communities.

Recent extensive literature reviews and industry assessments (primarily focusing on ultra-low power consumption and energy awareness, battery state estimation [15], and battery-aware resource management [16]) have consistently emphasized and frequently called for in-depth, low-level hardware-aware optimization for embedded battery intelligence applications. This urgency stems from the fundamental technical contradictions between the physical scenarios of batteries and traditional image recognition scenarios.

1.1.3 Another possibility for edge acceleration of time series data: Temporal Convolutional Networks

Besides deep compression of recurrent connection-based architectures like LSTM networks, TCNs have been widely recognized in recent years as having significant underlying hardware advantages over traditional recurrent architectures (RNNs and LSTMs) in edge computing, embedded deployment, and hardware acceleration, particularly in the cutting-edge fields of time series prediction and sequence modeling.

Meanwhile, according to the latest cutting-edge literature, TCN and LSTM are not mutually exclusive [17], [18], [19]. When processing dual-scale time series such as batteries, which exhibit long-term degradation and short-term complex fluctuations, TCN's powerful local multi-scale parallel feature extraction capability can be deeply integrated with LSTM's excellent long-term global historical momentum preservation capability (TCN-LSTM combined architecture). Although this hybrid architecture can further squeeze precision ceiling, its parameter scale at the edge is much larger, making "hardware-aware deep compression" an essential path to overcome the "last mile" of industrial implementation.

1.2 Purpose and Goal

1.2.1 Purpose

The purpose of this research is to evaluate a hardware-aware workflow for transferring and compressing a data-driven battery state-estimation model for resource-constrained microcontrollers. The study focuses on accuracy and theoretical storage trade-offs; latency, energy consumption, and real-time behavior are outside the demonstrated scope.

1.2.2 Goal

Specifically, this thesis seeks to answer the following research questions:

- To what numerical extent do individual model compression techniques—such as magnitude-based pruning and post-training quantization—affect the accu-

racy of battery SoH estimation models, and what are their respective error boundaries?

- What are the quantitative trade-offs between prediction accuracy and resource consumption across different optimization strategies?
- Which optimization pipelines achieve an optimal Pareto balance between model storage size and prediction error for battery AI inference on resource-constrained platforms?

The primary objective is to quantify accuracy and theoretical parameter-storage changes for the evaluated compression configurations and to document the software steps needed for Simulink integration and code-generation feasibility.

1.3 Contribution

The baseline PyTorch LSTM model and its initial implementation in this report were provided by Cognivity AI. The core contribution of this thesis lies in establishing and evaluating a complete tool chain from PyTorch model import to embedded code generation, specifically in the following four aspects:

First, a model transfer and numerical equivalence verification process from PyTorch to MATLAB was established. For the PyTorch LSTM model provided by Cognivity AI, cross-framework import was completed—addressing key issues such as LSTM layer mapping, weight arrangement transformation, and gating order alignment. By constructing a unified test vector set, the numerical deviations between the original PyTorch input, the MATLAB input, and the difference between MATLAB output and Simulink simulation output, providing a quantitative benchmark for the correctness of subsequent compression and code generation.

Second, pruning and quantization strategies were systematically evaluated for the supplied model. Unstructured and structured pruning were compared with PTQ and QAT using validation-based configuration selection and a final held-out test evaluation. The combined configurations are reported as limitations where their accuracy degradation is large.

Thirdly, automated generation of embedded C++ code from the compressed model is achieved. In the MATLAB/Simulink environment, the compressed LSTM model was exported as a Simulink module. Using Embedded Coder, general-purpose C++ code independent of vendor-specific deep learning libraries was generated. The feasibility of code generation and compilation was verified using the Infineon AURIX TC4D7 Lite evaluation board as the cross-compilation target.

Last, an accuracy–theoretical-storage comparison was constructed across the evaluated configurations. Runtime computation, Flash/RAM usage, and energy are left for future on-target measurements.

1.4 Organization of the Thesis

This thesis is organized into seven chapters. Chapter 1 introduces the research background and related work on battery state estimation, model compression, and edge deployment. It presents the purpose and goals of the study and summarizes the main contributions of the thesis.

Chapter 2 provides the technical background required for the subsequent analysis. It introduces the fundamentals of battery modeling, including State of Charge (SoC), State of Health (SoH), and equivalent circuit models. The chapter then presents the basic principles of recurrent neural networks and Long Short-Term Memory (LSTM) networks, followed by an introduction to model compression techniques, including pruning and quantization. Finally, the evaluation framework used to analyze the trade-off between prediction accuracy and theoretical model storage is described.

Chapter 3 presents the methodology adopted in this study. It describes the data and evaluation protocol, the transfer of the baseline LSTM model from PyTorch

to MATLAB, the procedures used to evaluate pruning and quantization strategies, and the Simulink and code-generation checks performed as part of the deployment workflow.

Chapter 4 describes the design and implementation of the proposed deployment workflow. The overall framework is divided into data, feature, model, compression, and deployment layers. The chapter further details the feature-processing interface, the provided LSTM model, the applied pruning and quantization procedures, and the workflow for Simulink integration and embedded C++ code generation.

Chapter 5 presents the experimental results. It first reports the dataset and preprocessing results, followed by the numerical equivalence verification and baseline LSTM performance. The effects of unstructured and structured pruning are then evaluated, followed by a comparison of post-training quantization (PTQ) and quantization-aware training (QAT). The chapter also investigates combined compression configurations and presents the Simulink closed-loop verification and embedded C++ code-generation results.

Chapter 6 discusses the experimental findings and their implications for embedded deployment. The results are interpreted in terms of prediction accuracy, parameter reduction, and theoretical storage efficiency. The chapter also discusses the scope and limitations of the demonstrated deployment workflow, including the absence of on-target measurements for execution time, power consumption, and other hardware-level performance metrics.

Chapter 7 concludes the thesis by summarizing the main findings and the demonstrated transfer, compression, and deployment workflow. It also highlights the limitations identified in the study and outlines directions for future work.

2

Technical Background

This chapter provides the theoretical foundation necessary to understand battery state estimation and neural network-based modeling. We first introduce fundamental battery metrics (SoC and SoH) and traditional modeling approaches. We then explain the mechanics of RNNs and LSTMs. Finally, we define the evaluation framework used to quantify the trade-offs between model accuracy and hardware efficiency.

2.1 Battery Modeling Fundamentals

2.1.1 State of Charge

State of Charge, also known as charge level or remaining charge, is the current amount of charge in a battery, expressed as a percentage. In electric vehicles, whether battery-powered or plug-in hybrid, the SoC of the battery pack is similar to the fuel gauge in a gasoline car, indicating the maximum remaining driving range. In consumer-level electronics such as mobile phones and laptops, SoC indicates the remaining time the battery can be used without a power source.

To measuring SoC, it is usually measured online or locally. Local measurements use a constant current method for charging and discharging (coulomb counting). This method estimates the remaining capacity by continuously integrating the current flowing in and out of the battery over time. While computationally simple, it is prone to cumulative sensor drift over long periods. While online measurement technologies to estimate the battery's capacity during use. Indirect methods for estimating SoC generally fall into five categories: Mass of substances before and after a chemical reaction (unit: mole), voltage, current integral (charge), Kalman Filtering, and pressure.

2.1.2 State of Health

The State of Health is a performance indicator of a battery or battery pack, comparing its condition to its ideal state. SoH is also measured as a percentage (100% indicates the battery's state matches its specifications). Generally, a new battery has the same capacity as its nominal capacity, and its SoH is 100%. During battery use, its health, capacity, and other parameters gradually decline until the end of its lifespan (SoH is generally less than 70%). Therefore, unstable or unreliable batter-

ies are periodically replaced with healthy, new batteries, which can be evaluated as following,

$$\text{SoH} = \frac{\text{condition}}{\text{condition}_{\text{ideal}}} \times 100\% \quad (2.1)$$

When a battery-powered device is put into use, the battery management system assesses the health of the managed batteries and reports this information to other systems. Subsequently, a threshold is determined based on the application, and the battery's health is compared to this threshold to assess its suitability. Understanding the health of a specific battery and the threshold for a specific application confirms whether the battery is suitable for that application and estimates its usable lifespan.

2.1.3 Equivalent Circuit Model

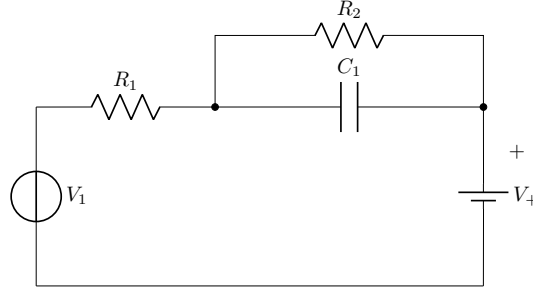


Figure 2.1: First-order equivalent circuit model for Li-ion cell

The equivalent circuit model (ECM) of a lithium-ion battery is a lumped circuit model of the battery. It uses passive components such as resistors and capacitors, along with a voltage source, to simulate the terminal voltage characteristics of the lithium-ion battery. The ECM is applicable in many fields due to its simple structure, low computational requirements, ease of representing battery characteristics, and structural flexibility, making it suitable for computer simulations. Figure 2.1 shows a basic first-order equivalent circuit model for Li-ion cell [20].

The characteristics of the ECM also make it suitable for real-time BMS to monitor parameters such as SoC and SoH, and for battery thermal management.

The ECM is used to simulate the voltage across the battery during charging or discharging. The most common circuit consists of three components connected in series: a variable voltage source representing the battery's OCV, a resistor representing the battery's internal ohmic resistance, and a resistor-capacitor (RC) circuit representing the battery's dynamic voltage drop [21], [22].

- **Open-circuit voltage:** The open-circuit voltage of a lithium-ion battery refers to the voltage measured under equilibrium conditions (no load current, after a long period of rest). Open-circuit voltage is a decreasing non-linear function of its SoC, and is related to the battery's anode (usually graphite) and cathode

(materials such as LFP). In the equivalent circuit, open-circuit voltage is the voltage generated by charge; it acts as the ideal voltage source representing the static open-circuit voltage of the cell.

- **Internal resistance:** Internal resistance in a circuit is represented by a resistor, used to represent the voltage drop caused by the Ohmic effect. Internal resistance is affected by temperature, SoH, and SoC.
- **RC parallel circuit:** One or more RC parallel circuits are added to the model to simulate its dynamic voltage changes. The number of parallel circuits is determined during modeling: generally, more RC circuits result in a more accurate model, but increase the complexity of the identification process and computational burden; fewer RC circuits result in less computational burden and easier identification of features, but less accurate dynamic voltage estimation. Generally, the best practice is to use one or two RC circuits in parallel.

2.2 Recurrent Neural Networks & LSTM

2.2.1 Basics of Artificial Neurons and Data Flow

For readers less familiar with AI, before diving into the complex recurrent architectures used in this study, it is helpful to understand the basic operation of an artificial neural network and its fundamental building block: the artificial neuron.

At a macro level, an AI algorithm operates as a highly complex mathematical mapping function. Data flows sequentially from an *input layer*, through one or more intermediate *hidden layers*, and finally to an *output layer*. In the context of this battery management thesis, the input data series consists of the 12-dimensional electrochemical features extracted from a battery's charge-discharge cycles (e.g., charging duration, voltage variations). The network processes these arrays of numbers layer by layer, and the output layer ultimately computes a single scalar value: the predicted SoH percentage.

Inside the hidden layers, the actual computation is distributed across hundreds or thousands of basic mathematical units called "neurons" (or nodes). Typically, a single artificial neuron performs a straightforward two-step operation to process incoming data:

1. **Weighted Sum:** First, the neuron receives multiple input signals ($x_1, x_2, \dots, x_{(n-1)}, x_n$) from the previous layer. It multiplies each input by a specific learnable coefficient called a *weight* (w_i), and adds a constant offset called a *bias* (b). The weights and biases represent the "knowledge" of the network, which are continuously updated during the training process to minimize prediction errors. This step is a simple linear combination can be represented as

$$z = \sum_{i=1}^n w_i x_i + b \quad (2.2)$$

2. **Activation Function (The "Neuron Function"):** Second, to allow the network to learn complex real-world physical dynamics rather than just straight lines, the linear result z is passed through a non-linear mathematical equation called an *activation function*, denoted as $\sigma(z)$. For a non-AI-knowledgeable reader, this function can be thought of as a biological neuron's "firing rule"—deciding how strongly the neuron should pass its signal to the next layer.

Typical examples of these "neuron functions" include:

- **Rectified linear unit (ReLU):** Simply outputs zero if the input is negative, and passes the input directly if it is positive ($y = \max(0, z)$). This is widely used for its computational simplicity.
- **Sigmoid / Tanh:** These functions squash the output into a strict numerical range, such as $[0, 1]$ or $[-1, 1]$. They are shaped like an "S" curve and are heavily used in LSTM networks to act as "gates" that decide what proportion of information to remember or forget.

By stacking thousands of these simple neuron functions together into deep layers, the AI model gains the remarkable ability to approximate the highly non-linear, path-dependent electrochemical degradation trajectories of lithium-ion batteries.

2.2.2 Recurrent Neural Networks

Recurrent neural networks, which are proposed by Jeffrey L. Elman in 1990, are a class of neural networks designed to process sequential data. Unlike traditional feedforward neural networks, RNNs introduce recurrent connections within layers, allowing the network to retain information from previous inputs [23], [24].

Logically, an RNN can be viewed as a neuron continuously passing its own information to itself at the next time step through recurrent connections; computationally, it is typically unfolded along time steps, forming a chain-like deep network.

The core of an RNN is the hidden state h_t , which acts as the network's "internal memory," responsible for storing historical information up to time t . Figure 2.2 illustrates a simple and direct structure of RNN (Elman Network) .

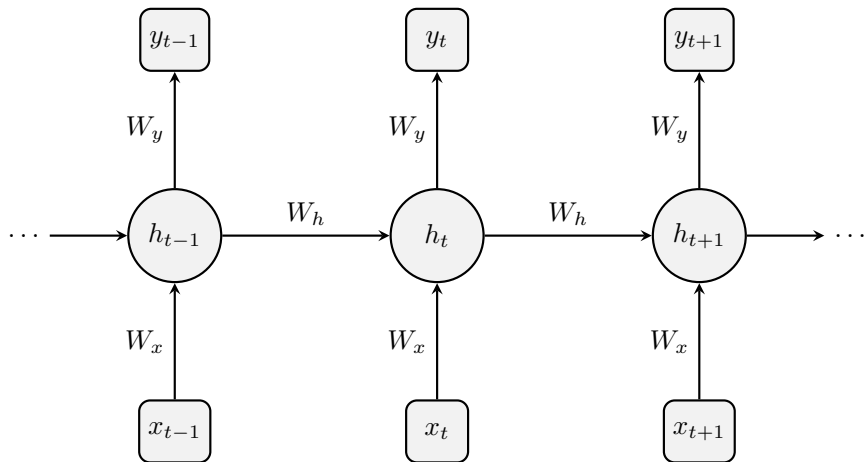


Figure 2.2: Recurrent Neural Network Structure (Elman Network)

Regarding the training mechanism, RNNs are typically trained using the backpropagation through time algorithm.

- **Forward Propagation:** The entire sequence is expanded time-step by time, and the output at each time step is calculated.
- **Loss Calculation:** The sum of the losses over all time steps (or at a specific time step) is calculated.
- **Gradient Update:** Gradients propagate not only between layers but also along the time steps from back to front. This means that the update of the current weights depends on the states of all past time steps.

Mathematically, it is

$$h_t = \phi(W_h h_{t-1} + W_x x_t + b_h) \quad (2.3)$$

$$y_t = \sigma(W_y h_t + b_y) \quad (2.4)$$

at each time step t , we can get such a hidden state h_t and output y_t .

2.2.3 Long Short-Term Memory

LSTM are a variant of RNNs designed to address the vanishing and exploding gradient problems during long sequence training [25]. The simplified version of the LSTM unit structure used in this study is shown in the figure below. Its core mechanism utilizes a "gating" system to preserve long-term information and filter short-term information. Figure 2.3 indicates the basic LSTM structure in time t .

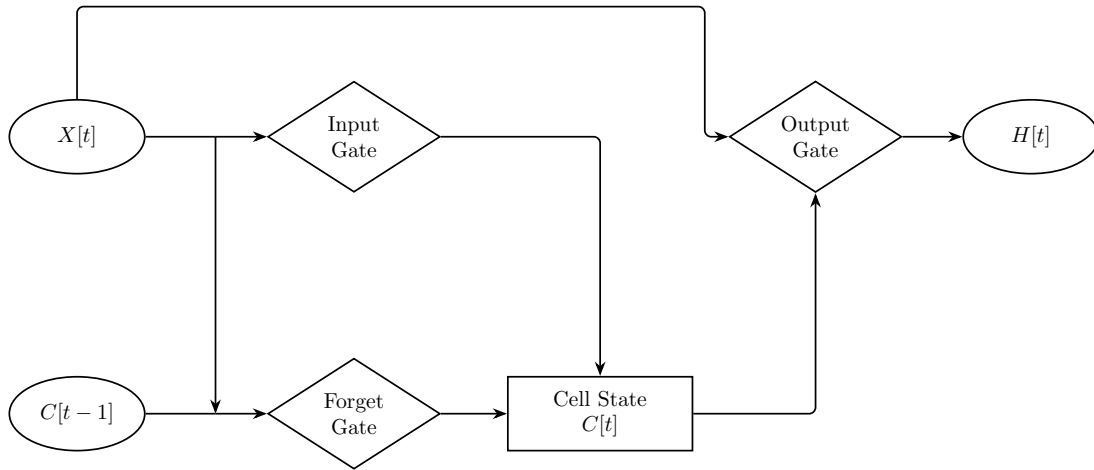


Figure 2.3: LSTM Structure

Core Components and Signal Inputs:

- **Input signal (X_t):** Represents the input feature vector at the current time t .

- Previous cell state (C_{t-1}): Carries long-term memory information since the beginning of the sequence.
- Current cell state (C_t): The core "conveyor belt" of the LSTM, continuously updating and transmitting information through the operations of various gating mechanisms.
- Output signal (H_t): The hidden state at the current time step; it is both the output of this unit and will be passed to the computation of the next time step.

LSTM controls the update of the cell state C_t through the following three key gating mechanisms [26]:

- Forget Gate: This gate determines which information to discard from the previous cell state C_{t-1} . It processes X_t and H_{t-1} (implicitly shown in the diagram) to generate a "Forget old" filter coefficient, which is applied to C_{t-1} . Usually in forget gate, it can be expressed as $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$.
- Input Gate: This gate determines which new information from the current X_t is stored in the cell state. It generates "Add new information" and adds it to the filtered cell state, thus generating the current C_t .
- Output Gate: After the cell state C_t is updated, the output gate, combined with the current input X_t , determines the final output. It filters and transforms C_t to ultimately produce the hidden state H_t at the current moment.

2.3 Compressing Method

2.3.1 Pruning

In fields such as edge AI and large language model compression and lightweighting, the billions to hundreds of billions of parameters in large models bring extremely stringent requirements for storage space and memory bandwidth, creating a severe contradiction with the limited hardware resources at the edge. To overcome the bottleneck of "computing power/bandwidth wall," model pruning and model quantization have become the two most core edge compression and acceleration techniques. Model pruning aims to reduce the number of parameters while preserving core representation capabilities by evaluating the significance of model weights and removing redundant or unimportant parameters. Based on the regularity of the parameter matrix after pruning, it is mainly divided into unstructured pruning and structured pruning. Unstructured pruning uses a single weight element-wise as the smallest unit, directly setting weights below a threshold to zero based on the absolute value of the weight or gradient sensitivity. Elias Frantar et al. first proved that large models with hundreds of billions or trillions of parameters can undergo up to 50%-60% unstructured pruning directly without retraining, and with almost no decrease in accuracy [27]. This significantly reduces the computational cost of pruning, enabling large-scale weight sparsification to be completed within hours by utilizing approximate information from the second-order derivative Hessian matrix.

Structured pruning uses regular structural blocks as the smallest unit, directly pruning entire rows, columns, attention heads, MLP hidden channels, and even entire Transformer layers. Mengzhou Xia et al., through structured pruning and dynamic data loading, precisely "pruned" and compressed larger parameter models (such as 13B) into standard, smaller models (such as 3B/7B) [28].

2.3.2 Quantization

Model quantization maps high-precision floating-point parameters (such as single-precision FP32 and half-precision FP16) in a model to low-bit fixed-point numbers or integer formats (such as INT8, INT4, and INT2). Quantization can not only directly compress the model size to 50% to 12.5% of its original size, but also transform expensive floating-point multiplication into low-power integer addition and multiplication operations. Based on whether model retraining is required, quantization techniques can be divided into post-training quantization (PTQ) and quantization-aware training (QAT). PTQ completes quantization after model pre-training using only a very small amount of calibration data or no data at all, by directly calculating the quantization scaling factor and zero-point by statistically analyzing the numerical distribution of weights and activation values. QAT, on the other hand, introduces quantization simulation operators (such as the pass-through estimator STE) during model fine-tuning or training [29]. It introduces simulated quantization errors during forward propagation, allowing the model network weights to actively adapt to rounding noise caused by low bit depth, and updates the weights during backpropagation.

2.4 Evaluation Framework

In order to quantify the trade-offs between model performance and hardware constraints realistically evaluated in this study, we shift the evaluation focus from dynamic runtime metrics (such as power and latency, which require physical on-target profiling) to static memory metrics. The core evaluation framework relies on a Storage-Accuracy Pareto analysis [30], [31].

The efficiency of the compression is quantified by the *Compression Ratio* (CR)

$$CR = \frac{\text{Size}_{\text{FP32}}}{\text{Size}_{\text{Compressed}}} \quad (2.5)$$

It is defined based on the theoretical parameter memory footprint [29].

Simultaneously, the prediction penalty is monitored via the RMSE degradation relative to the baseline model. The goal of this framework is to maximize the CR while keeping the RMSE strictly under a predefined engineering tolerance threshold (e.g., $1.15 \times \text{RMSE}_{\text{base}}$). This multi-objective perspective directly addresses the static memory bottlenecks (SRAM and Flash limits) of automotive MCUs, providing actionable guidance for edge deployment without overclaiming runtime acceleration.

3

Method

This chapter describes the workflow used to transfer and evaluate a battery SoH model for embedded deployment. The baseline LSTM model, its original implementation, training, feature definitions, and preprocessing were provided by Cognivity AI. The work reported here does not claim development of that model; it evaluates transfer fidelity, compression behavior, and deployment-toolchain feasibility.

3.1 Data and Evaluation Protocol

The experiments use the datasets and preprocessing supplied with the baseline model. Data are separated at battery level to avoid sharing a battery between development and evaluation. One baseline checkpoint, one preprocessing version, and one evaluation script are used throughout the comparisons. Compression parameters and quantization calibration data are selected from the training and validation partitions. The held-out test partition is evaluated only after the model configuration has been selected and frozen.

3.2 PyTorch-to-MATLAB Transfer

The supplied PyTorch network is imported into MATLAB and checked with shared input sequences. The transfer check covers tensor layout, recurrent-state handling, weight mapping, and output scaling. Numerical differences are summarized before any compression is applied, providing a reference for later Simulink checks.

3.3 Compression Evaluation

Unstructured pruning, structured pruning, PTQ, and QAT are evaluated as separate configurations and, where applicable, in combination. Pruning candidates are selected using validation RMSE. QAT uses training data for optimization, while PTQ uses a calibration subset drawn from the training partition. The reported INT8 storage values are theoretical parameter-storage estimates based on one byte per quantized parameter; they do not by themselves establish the data types used by generated code.

3.4 Simulink and Code-Generation Checks

Selected networks are exported to Simulink for model-in-the-loop functional comparison with the MATLAB implementation. Embedded Coder is then used to generate statically allocated C++ artifacts, and the artifacts are cross-compiled for the Infineon KIT_A3G_TC4D7_LITE evaluation platform. These checks establish software conversion and compilation feasibility only. No on-target latency, CPU load, power, energy, or real-time schedulability measurements are claimed.

4

Design

Building upon the methodology outlined in Chapter 3, this chapter describes the engineering workflow used in this project. The baseline LSTM model, its architecture, preprocessing, and original training were provided by Cognivity AI. The design work reported here therefore focuses on model transfer, compression evaluation, Simulink integration, and code-generation feasibility rather than on developing a new predictive model.

4.1 Framework

This project designed an intelligent estimation system for the SoH of lithium batteries for embedded deployment. Its overall architecture consists of five layers: a data layer, a feature layer, a model layer, a compression layer, and a deployment layer. These layers are interconnected through standardized interfaces, forming an end-to-end automated pipeline.

The data layer is responsible for the unified access and cleaning of heterogeneous data from multiple sources. The system integrates five publicly available battery datasets: CALCE (University of Maryland) [32], HUST (Huazhong University of Science & Technology) [33], MATR (National Renewable Energy Laboratory) [34], Tongji (Tongji University) [35], and XJTU (Xi'an Jiaotong University) [36], covering three mainstream chemical systems: LCO, NMC, and LFP. To address the differences in sampling rate, voltage range, and field naming among the different datasets, an adaptive parser was designed—emphasizing voltage-current normalization based on electrochemical boundaries for common formats and recursive expansion and timestamp reconstruction algorithms for the nested cell structure unique to XJTU. After deduplication, interpolation, and resampling, the original cyclic data is uniformly aligned to a standard representation of 200 uniform time points per cycle.

The feature layer uses the cycle-level features and preprocessing supplied with the baseline model. Only the resulting interface and the battery-level separation required for reproducible evaluation are described here; Cognivity-specific feature definitions, thresholds, and normalization constants are omitted.

The model layer is the sequence-to-one LSTM supplied by Cognivity AI. Its internal layer sizes, window length, feature definitions, and training configuration are treated as provided model details and are not repeated. The experiments use one frozen baseline checkpoint and one consistent preprocessing and evaluation implementation.

The compression layer evaluates unstructured pruning, structured pruning, PTQ, and QAT. Candidate configurations and calibration settings are selected using training/validation data; the held-out test set is used only for the final comparison. The deployment layer exports the selected network to Simulink and generates general-purpose C++ with Embedded Coder. Reported INT8 storage values are theoretical parameter-storage estimates unless an implementation data type is explicitly verified. Cross-compilation is treated as a feasibility check, not as evidence of on-target performance or hardware deployment.

4.2 Feature Layer

The feature layer uses the cycle-level feature interface supplied with the baseline model. Detailed feature definitions and Cognivity-specific preprocessing are omitted because they are not contributions of this thesis.

Table 4.1: Definition of the 12 engineered electrochemical features extracted per cycle.

ID	Feature Name	Definition
<i>Constant-current/Constant-voltage (CC/CV) Charge Characteristics</i>		
F1	CC_Charge	CC-phase charge capacity (C-rate normalized), $\int I_{\text{norm}} dt / 3600$
F2	CV_Charge	CV-phase charge capacity (Ah)
F3	CC_Time	CC-phase duration (s)
F4	CV_Time	CV-phase duration (s)
<i>Voltage Statistical Characteristics</i>		
F5	V_Std	Standard deviation of normalized voltage within the cycle
F6	V_Max	Maximum normalized voltage
F7	V_Slope	Discharge voltage decay slope (linear fit on V_{norm} vs. t)
F8	V_Kurtosis	Kurtosis of the voltage distribution
<i>Current and Temperature Characteristics</i>		
F9	I_Max	Maximum absolute discharge C-rate
F10	T_Std	Standard deviation of temperature
F11	T_Max	Maximum cycle temperature
<i>Capacity Window Characteristic</i>		
F12	Cap_Window	Discharge capacity (Ah) within normalized voltage window $[0.2, 0.8]$

When extracting cycle-level features, the raw data often contains local oscillations and pseudo-spikes due to sensor noise or the "capacity regeneration" effect (a physical

phenomenon where batteries temporarily recover some capacity after a long rest). Traditional moving average filters tend to "smear" or blur critical high-frequency electrochemical changes, such as the sudden voltage drop at the end of a battery's life (the "Knee Point"). Therefore, we utilize the Savitzky-Golay (SG) filter. By fitting a local polynomial, the SG filter successfully flattens sensor spikes while preserving the true physical waveforms and local maxima of the degradation trend.

After feature extraction, a three-stage noise reduction pipeline is applied: (1) percentile clipping at the 0.5% and 99.5% levels to remove extreme outliers, (2) moving median trend-line deviation detection with a 15-cycle window—any point deviating more than 5% from the trend is replaced by the trend value, and (3) a Savitzky-Golay polynomial smoothing to produce the final feature trajectories while preserving degradation local maxima

SoH labels are consumed from the supplied preprocessing pipeline. The implementation details and thresholds are not repeated here.

$$\text{SoH}(c) = \frac{Q(c)}{Q_{\text{ref}}}, \quad Q_{\text{ref}} = \text{median}\left(\max_5\{Q(c) \mid c \in [4, 50]\}\right). \quad (4.1)$$

4.3 Model Layer

The model layer employs a two-layer LSTM architecture whose detailed specification is provided in Table 4.2. The input is a sliding window of length $W = 15$, allowing the model to infer current health using only recent history. The first LSTM layer with 64-dimensional hidden states captures long-range temporal dependencies, while the second LSTM layer with 32-dimensional hidden states performs feature compression. A fully connected layer with ReLU activation maps the compressed representation to a scalar SoH prediction.

Table 4.2: Layer-by-layer specification of the LSTM model architecture.

Layer	Type	Input Shape	Output Shape	Parameters
Input	Sequence Input	15×12	15×12	0
LSTM_1	LSTM (hidden size = 64)	15×12	15×64	19,712
Dropout	Dropout ($p = 0$, disabled)	15×64	15×64	0
LSTM_2	LSTM (hidden size = 32)	15×64	15×32	12,416
FC_1	Fully Connected (32)	32	32	1,056
ReLU	ReLU Activation	—	—	0
FC_2	Fully Connected (1)	32	1	33
Total: 33,217 parameters (weights + biases)				

A critical design decision in the training strategy is the normalization procedure, which is specifically engineered to prevent test-set information leakage. As summarized in Table 4.3, a five-step per-battery relative normalization pipeline is applied before global scaling.

Table 4.3: Five-step normalization strategy for leakage-free preprocessing.

Step	Operation	Description
1	Per-battery baseline	Compute the mean feature vector over the first 3 cycles of each battery
2	Explosion guard	If baseline absolute value $< 10^{-2}$, force baseline to 1 (avoids division errors)
3	Relative change	Divide all cycles of that battery by its baseline vector
4	Hard clipping	Clip relative change rates to $[-5, 5]$ to suppress sensor noise amplification
5	Global Min-Max	Apply Min-Max scaling computed solely from training set statistics

The original training configuration was provided by Cognivity AI and is outside the scope of this thesis. Evaluation uses battery-level separation and derives selection decisions from training/validation data only; the test partition remains held out until final evaluation.

4.4 Compression Layer

Automotive-grade MCUs have constrained memory and compute resources. This project evaluates pruning and quantization as candidate methods for reducing theoretical model storage; runtime and power effects are not measured.

Specifically, the compression layer evaluates pruning followed by INT8 quantization, as shown in Figure 4.1. The resulting accuracy-storage trade-offs, including substantial degradation for combined configurations, are reported in Chapter 5.

4.4.1 Magnitude-Based Weight Pruning

Neural networks typically contain massive redundant connections, meaning not all weights contribute equally to the final SoH prediction. To address this, our project implements magnitude pruning on the full-precision (FP32) network. In the context of this thesis, the primary objective of this unstructured pruning phase is to enhance model compressibility (parameter footprint reduction) to fit within extremely limited ROM/Flash storage, thereby significantly reducing the parameter burden for subsequent quantization processes.

To scientifically determine the optimal balance between sparsity and precision degradation, we systematically sweep target sparsity levels from 0% to 90% in 10% increments, rather than relying on a fixed threshold. This scanning approach allows us to map the empirical *Pareto Front* on multi-source time series datasets, where theoretical redundancy cannot always be perfectly mapped to practical performance. For a given target sparsity θ , the original weights are permanently set to zero if they fall below a magnitude threshold, yielding the pruned weight $\widetilde{W}_{ij}$ as

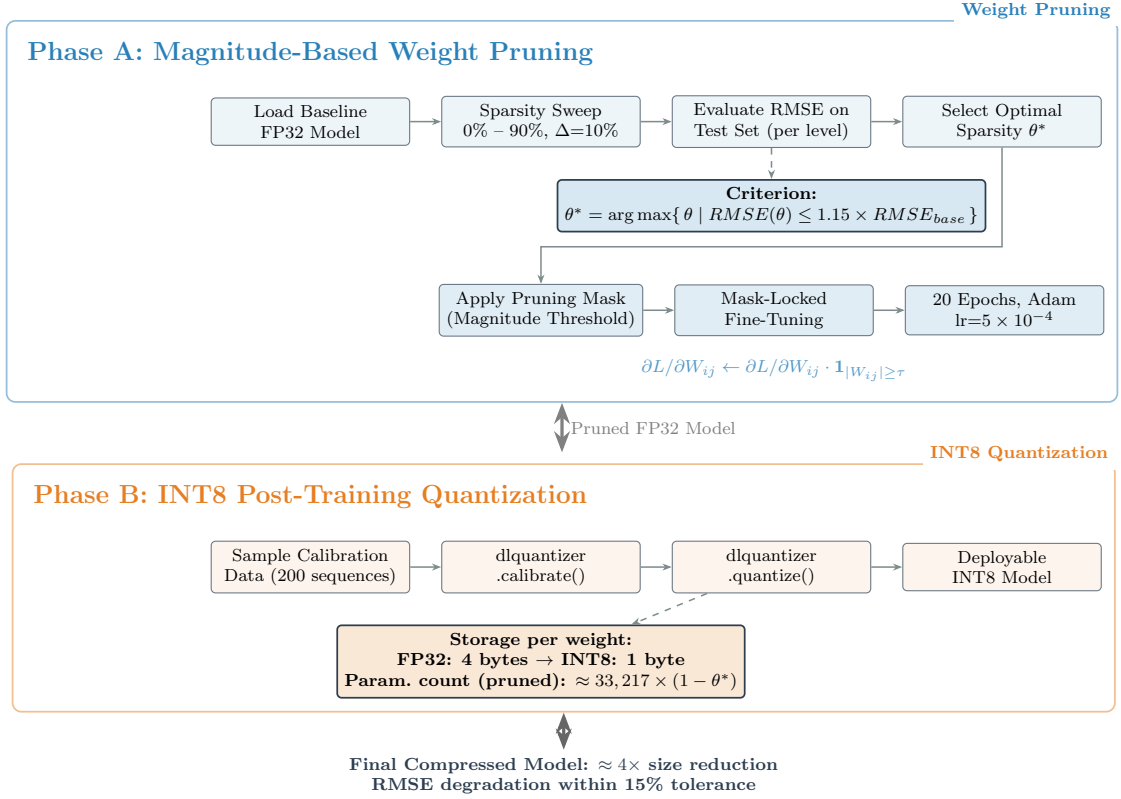


Figure 4.1: Two-stage deep compression pipeline: Phase A—magnitude-based weight pruning with mask-locked fine-tuning; Phase B—INT8 post-training quantization via MATLAB `dlquantizer`.

$$\widetilde{W}_{ij} = W_{ij} \cdot \mathbf{1}_{|W_{ij}| \geq \tau}, \quad \tau = \text{percentile}(\{|W_{ij}|\}, \theta \times 100) \quad (4.2)$$

where W_{ij} is the original element from the weight matrices (including both LSTM input/recurrent weights and fully connected weights), $\mathbf{1}$ denotes the indicator function, and τ represents the threshold defined as the θ -th percentile of the absolute values across all network weights.

For engineering purposes, a 15% zero-shot root mean square error (RMSE) degradation tolerance (evaluated immediately after applying the mask, without any re-training) is specifically set as the acceptable upper limit. The rationale for this exact selection criterion is threefold. First, zero-shot evaluation serves as a highly efficient proxy for network recoverability; evaluating the fully fine-tuned performance at every granularity step would be computationally prohibitive. Second, empirical observations in battery SoH time-series applications suggest that an initial structural damage of up to 15% can be reliably recovered during the subsequent fine-tuning phase. Exceeding this threshold typically indicates that critical feature extraction pathways—such as essential historical state retention mechanisms in the LSTM memory cells—have been irreversibly severed, leading to a permanent performance plateau that fine-tuning cannot overcome. Third, maintaining this strict

margin ensures that the network retains sufficient representational capacity to absorb additional quantization noise in subsequent deployment steps.

Consequently, the optimal sparsity θ^* is selected as the maximum sparsity that bounds the zero-shot error strictly within this $1.15\times$ baseline RMSE envelope, establishing a controlled accuracy-efficiency trade-off. Once the sweep exceeds this tolerance, the system rolls back and locks at the previous safe sparsity, termed the Pruning Lock Point.

Where fine-tuning is applied after pruning, the pruning mask is kept fixed. Optimizer and epoch settings are implementation details and are not used to claim development of the baseline model.

$$\frac{\partial L}{\partial W_{ij}} \leftarrow \frac{\partial L}{\partial W_{ij}} \cdot \mathbf{1}_{|W_{ij}| \geq \tau^*}, \quad \forall(i, j), \quad (4.3)$$

where L denotes the training loss function, and τ^* is the optimal magnitude threshold derived from the selected sparsity.

Instead of altering the sparse structure, this mask-locked setup forces the remaining non-zero, high-value weights to be adjusted to compensate for the feature information lost due to pruning.

4.4.2 INT8 Post-Training Quantization

Following pruning, fine-tuning, and topological reconstruction, the optimization bottleneck shifted to data bit-width. In the realm of neural network quantization, two primary paradigms exist: QAT and PTQ. QAT incorporates simulated quantization errors (e.g., clipping and rounding) into the training forward pass, allowing the model to adapt and compensate for precision loss via backpropagation. While QAT typically yields optimal accuracy, especially for ultra-low bit-widths, it is computationally expensive, requires access to the full training pipeline, and involves complex hyperparameter tuning. In contrast, PTQ operates directly on a pre-trained floating-point model without the need for retraining or gradient calculations, mapping weights and activations to lower precision using only a small calibration dataset.

PTQ and QAT are both evaluated. Their accuracy and storage results are treated as empirical outcomes of the reported experiment; no assumption of near-lossless compression or runtime benefit is made.

Quantization calibration uses only training-partition data. The calibration procedure determines scale factors and zero-points for the evaluated PTQ configuration; no claim is made about the data types of generated code.

With the calibration data prepared, the calibrate method was utilized to determine optimal dynamic ranges for the network. Subsequently, the quantize method was applied to linearly map the floating-point weights and activations—originally occupying 32 bits of memory—into the 8-bit integer (INT8) space.

INT8 quantization reduces the theoretical parameter-storage estimate relative to FP32. Whether generated code uses INT8 arithmetic, and any resulting band-

width, latency, or power benefit, requires separate implementation and hardware measurements.

4.5 Deployment Layer: Code generation and cross-compilation feasibility verification

The deployment layer evaluates conversion of the compressed model into embedded C++ code and its static memory mapping. The locked model is exported to Simulink, where input/output routing is checked under fixed-step simulation. Embedded Coder generates general-purpose C++ artifacts, and cross-compilation for the Infineon AURIX evaluation platform is used as a toolchain feasibility check. This stage does not establish target-device execution time, CPU load, power consumption, or generated INT8 arithmetic.

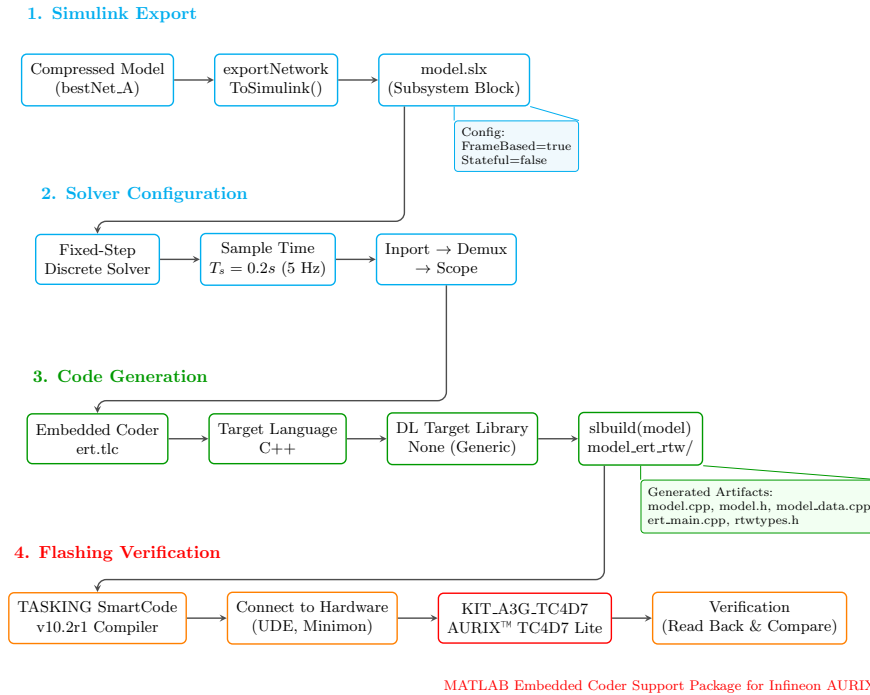


Figure 4.2: Simulink-to-hardware deployment flow

The exported C++ code architecture is shown in Table 4.4. The model class (`model`) encapsulates the full inference pipeline with a clean interface of `initialize()`, `step()`, and `terminate()` methods.

The memory footprint of the generated code is summarized in Table 4.5. All runtime buffers are stack-allocated within the model class, and all constant weights reside in read-only data sections suitable for placement in program flash memory.

Table 4.4: Generated C++ code artifacts from Embedded Coder.

File	Description
<code>model.h</code>	Model class declaration with block signals (<code>B_model_T</code>), states (<code>DW_model_T</code>), constant parameters (<code>ConstP_model_T</code>), and public interface (<code>init/step/terminate</code>)
<code>model.cpp</code>	Model step function implementing two-layer LSTM forward pass with element-wise sigmoid and tanh activation helpers
<code>model_data.cpp</code>	Constant parameter storage containing all trained weights and biases (<code>InputWeights</code> , <code>RecurrentWeights</code> , Bias for each LSTM layer, and FC weights)
<code>ert_main.cpp</code>	Standalone ERT main entry point with <code>rt_OneStep()</code> timer callback template and overrun detection
<code>rtwtypes.h</code>	Simulink Coder fixed-width type definitions (<code>real_T</code> , <code>int32_T</code> , <code>boolean_T</code> , etc.)

Table 4.5: Memory footprint estimation of the generated code (FP32 precision).

Component	Size	Recommended Placement
Constant weights (<code>ConstP_model_T</code>)	≈ 258 KB	PFLASH (Program Flash)
Block signals (<code>B_model_T</code>)	≈ 65 KB	DSPR (Data Scratchpad RAM)
Block states (<code>DW_model_T</code>)	≈ 1.5 KB	DSPR
External I/O (<code>ExtU</code> + <code>ExtY</code>)	≈ 1.5 KB	DSPR
Total runtime RAM	≈ 68 KB	

Table 4.6: Authoritative Interface Specification for Edge Deployment

Interface Attribute	Specification Details
Number of Input Features	12 (Electrochemical, voltage, and thermal features)
Sequence Length (W)	15 cycles
PyTorch Input Shape	Tensor(Batch_Size, Sequence_Length, Input_Dim)
MATLAB Input Shape	Sequence_Length $\times$ Input_Dim matrix per observation
Generated C/C++ Input	Flattened 1D array <code>real_T seq_input[180]</code> (Passed via External Input structure <code>ExtU</code>)
Output Definition	1D Scalar, Predicted SoH/SoC value (Passed via External Output structure <code>ExtY</code>)
Execution Mode	Stateless (<code>Stateful=false</code> , frame-based processing enabled)
Preprocessing Assumptions	• Features smoothed via Savitzky-Golay filter • Per-battery initial ratio normalized (Explosion guard applied) • Relative change rates hard-clipped to $[-5, 5]$ • Min-Max scaling applied based on training statistics

5

Results

5.1 Data Preprocessing and Electrochemical Feature Engineering Result

High-quality, noise-free input features are a core prerequisite for ensuring the generalization ability and convergence stability of deep learning models. This section details the physical characteristics of the datasets employed and presents the corresponding processing results, incorporating the rigorous data engineering logic applied to the raw time-series data prior to its input into the neural network.

Table 5.1: Summary of Multi-Source Battery Aging Datasets

Dataset Source	Chemistry Type	Nominal Cap. (Ah)	Number of Cycles	Voltage range
01_CALCE	LCO / Graphite	1.1	660	2.700 – 4.200 V
02_HUST	LFP / Graphite	1.1	1504	1.948 – 3.600 V
03_MATR	LFP / Graphite	1.1	1850	2.000 – 3.600 V
14_Tongji	Li0.86Ni0.86Co0.11Al0.03O2/ Graphite	3.5	487	2.650 – 4.200 V
15_XJTU	LiNi0.5Co0.2Mn0.3O2/ Graphite	2.0	389	2.500 – 4.200 V

5.1.1 Dataset Overview

To verify the robustness of the proposed edge AI deployment framework across different battery materials, discharge rates, and complex operating conditions, this study integrates five widely used open-source battery aging datasets: CALCE (University of Maryland), HUST (Huazhong University of Science & Technology), MATR (NREL), Tongji (Tongji University), and XJTU (Xi'an Jiaotong University). These datasets cover various cathode materials (e.g., Lithium Iron Phosphate, Nickel Manganese Cobalt) and different nominal capacities, with cycle lives ranging from hundreds to thousands of cycles. The diversity of the experimental data ensures that the model does not merely overfit to a specific operational profile but genuinely learns the common electrochemical characteristics of battery degradation.

The basic physical parameters and scales of each dataset are summarized in Table 5.1.

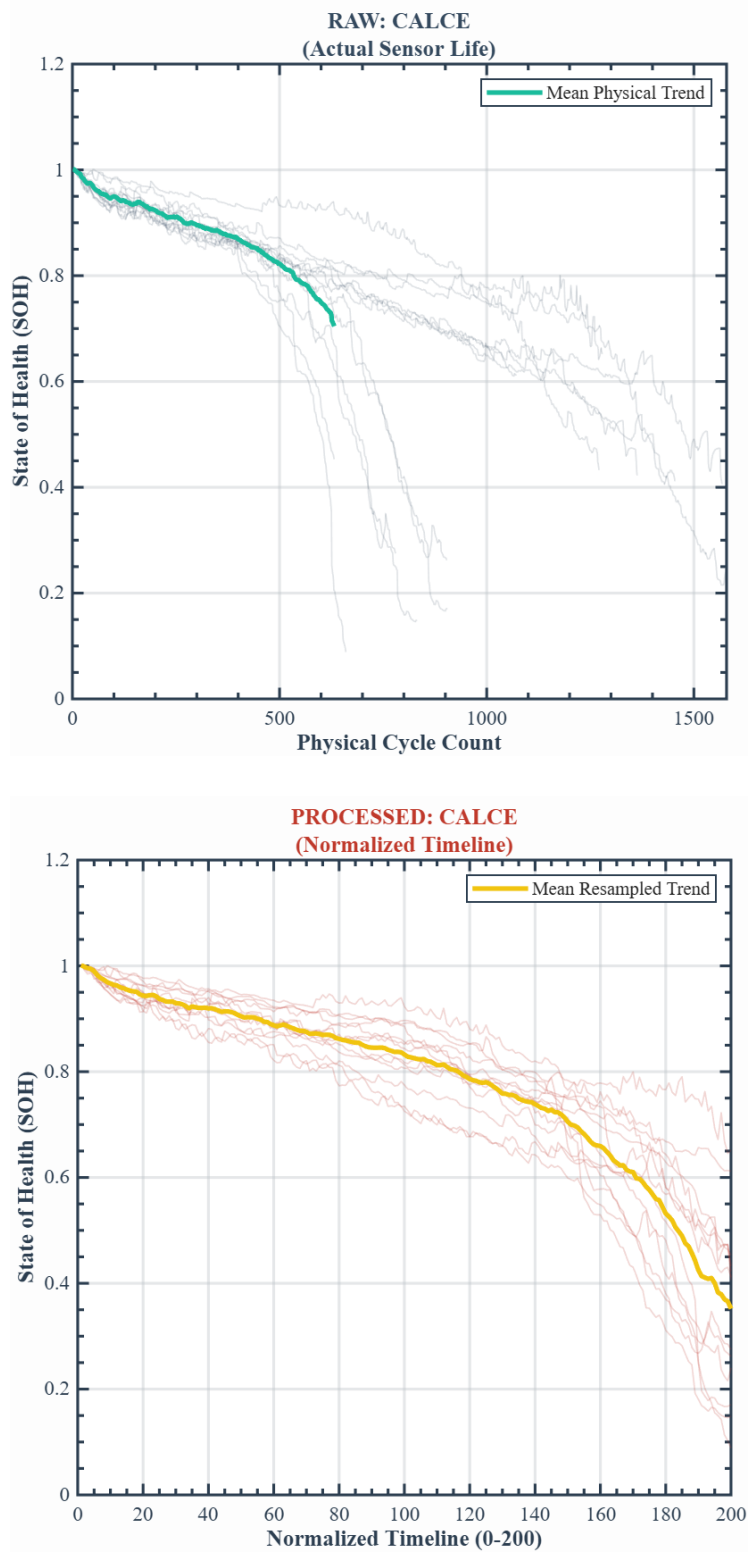


Figure 5.1: Visualization preprocessing results of the CALCE dataset.

5.1.2 Preprocessing visualization

We evaluated the supplied SoH resampling procedure on two representative datasets. The left plots of Fig. 5.1 and Fig. 5.2 show the original noisy SoH trajectories. The right plots show the smoothed and normalized timelines (0–200) after preprocessing.

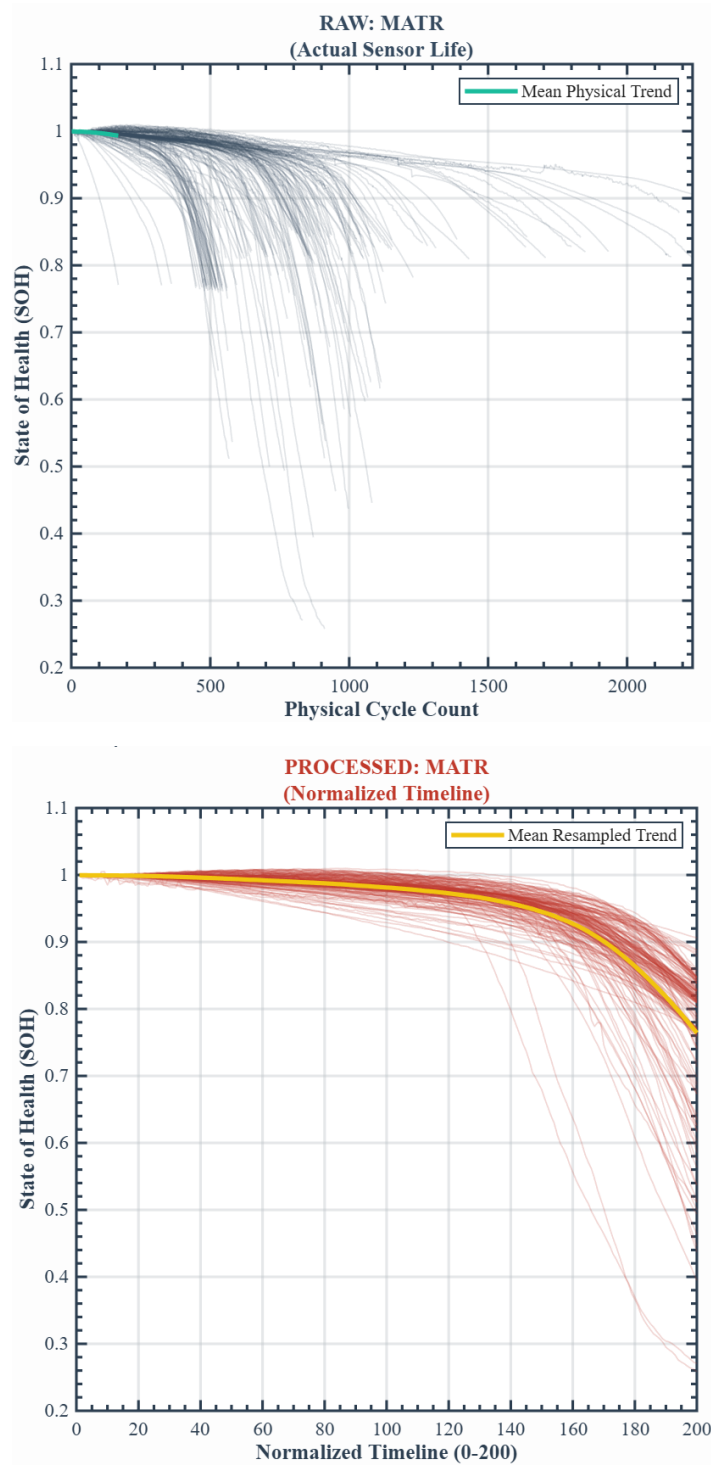


Figure 5.2: Visualization preprocessing results of the MATR dataset.

5.2 Baseline LSTM Performance

Before determining the compression constraints for edge deployment, the primary task is to build and evaluate a high-capacity floating-point (FP32) baseline model. As this project directly utilizes an LSTM model provided by Cognivity AI, the scope is limited to evaluation. This section details the predictive performance of the baseline LSTM model on the multi-source dataset.

5.2.1 Numerical Equivalence Verification

Prior to model compression and embedded deployment, the baseline PyTorch LSTM model provided by Cognivity AI was imported into the MATLAB environment. To guarantee that no structural distortion or weight realignment errors (such as LSTM gate ordering mismatches) occurred during the format migration, a cross-platform numerical verification was conducted across a standardized test cohort of $N = 200$ cycle sequence vectors. As illustrated in Figure 5.3, the prediction trajectory produced by the imported MATLAB ONNX model matches the original PyTorch baseline output with remarkable fidelity (top panel). The bottom log-scale residual profile confirms that the point-wise absolute deviation $|y_{PT} - y_{MATLAB}|$ resides strictly between 10^{-9} and 3×10^{-7} , remaining orders of magnitude below the engineering tolerance threshold of $\epsilon = 10^{-5}$. This strict floating-point numerical equivalence establishes a verified mathematical foundation for all downstream pruning, quantization, and Simulink code generation pipelines.

5.2.2 Evaluation Metrics and Baseline Performance

To comprehensively and objectively quantify the model’s predictive capabilities, this experiment selected three universally recognized evaluation metrics for regression tasks:

- **Root Mean Square Error (RMSE):** By applying a quadratic penalty to prediction deviations, RMSE is highly sensitive to outlier errors that severely deviate from the true value. In safety-critical BMS scenarios, large instantaneous errors could trigger false thermal runaway alarms; hence, RMSE is the core monitoring metric in this study.
- **Mean Absolute Error (MAE):** This measures the average magnitude of the absolute error between predicted values and true SoH labels, providing a linear baseline intuition regarding the model’s overall fitting deviation.
- **Mean Absolute Percentage Error (MAPE):** By standardizing the error relative to the true capacity, MAPE reflects the relative proportion of the error, making the prediction accuracy comparable across batteries with different capacity specifications (e.g., 1.1Ah vs. 2.0Ah).

Baseline Model Performance:

After the provided baseline checkpoint was selected, it was evaluated on an independent battery-level test set. Preprocessing and split definitions were fixed before this evaluation; no claim of newly trained model development is made.

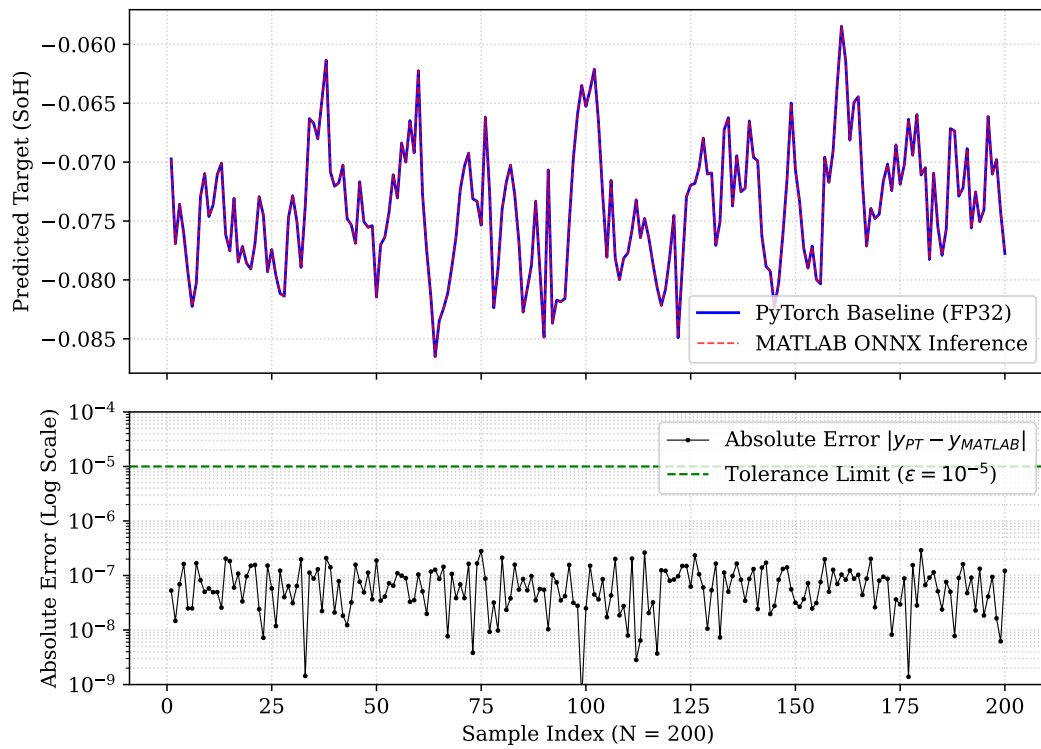


Figure 5.3: Cross-platform prediction overlap and FP32 residual error analysis between PyTorch baseline and MATLAB imported LSTM model ($N = 200$ test vectors, $\epsilon = 10^{-5}$).

5. Results

Experimental results for the supplied preprocessing and baseline checkpoint show that the FP32 model tracks the degradation trajectories in the reported examples. These curves are illustrative and do not establish generalization across all datasets or operating conditions.

The baseline model achieved the following quantitative metrics on the test set: **RMSE = 0.0102**, **MAE = 0.0069**, and **MAPE = 0.0079**. These values define the reference for the compression comparisons. The plots are example prediction curves for individual datasets and do not represent aggregated multi-source generalization performance.

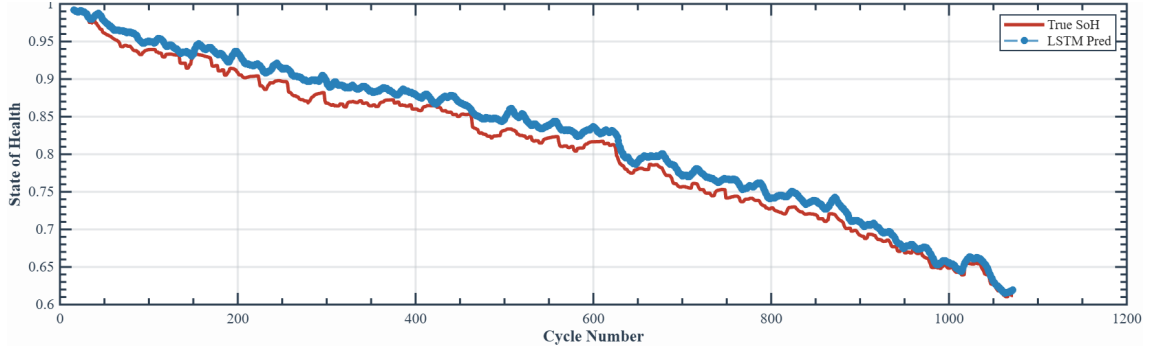


Figure 5.4: SoH Prediction Audit on the CALCE dataset.

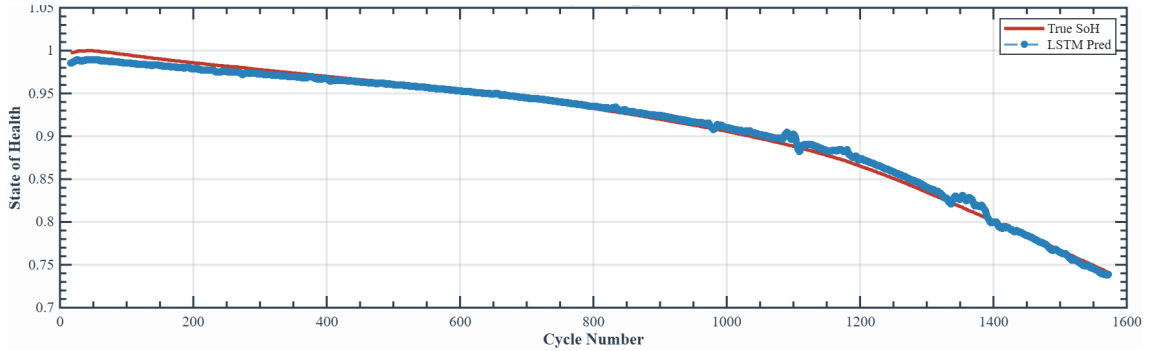


Figure 5.5: SoH Prediction Audit on the HUST dataset.

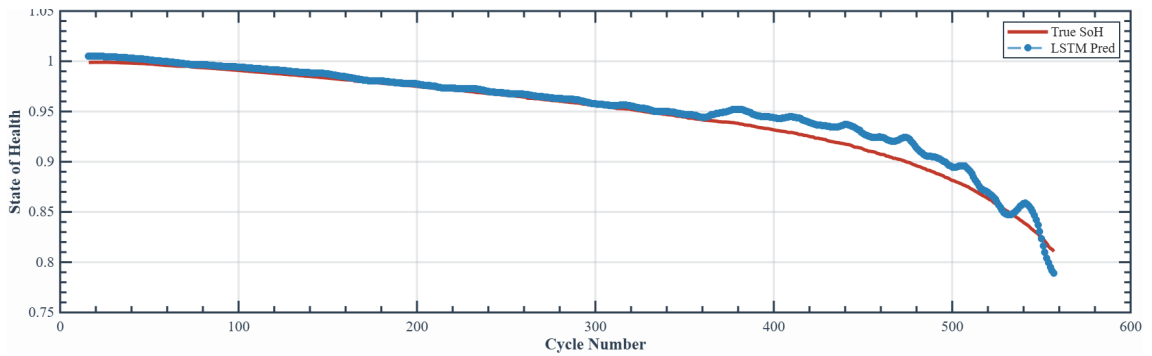


Figure 5.6: SoH Prediction Audit on the MATR dataset.

5.3 Pruning Result

As previously mentioned, this study investigates unstructured and structured pruning of the baseline LSTM model. Target pruning rates range from 0% to 90%, and changes in model size, prediction error, and SoH prediction curves are compared using MAE and RMSE.

Figure 5.7 illustrates the results of unstructured pruning applied to the baseline model at various target pruning rates. Unstructured pruning targets individual weight parameters within the network, determining the pruning threshold based on the magnitude of the absolute weight values. As the target pruning rate increases, the number of low-importance weights to be removed rises, resulting in an overall upward trend in the corresponding weight threshold.

As shown in the top-left subplot of 5.7, when the pruning rate increases from 0% to 30%, the pruning threshold rises rapidly from near zero to approximately 0.93×10^{-4} , indicating that a large number of network weights with small magnitudes can be effectively identified and removed. Beyond a pruning rate of 30%, the threshold remains relatively stable around 0.94×10^{-4} , increasing further to nearly 1.0×10^{-4} only at a 90% pruning rate. This suggests that the model contains a high proportion of parameters with small weight magnitudes, allowing for the removal of a vast number of redundant parameters at low-to-moderate pruning rates, whereas further increasing the pruning rate necessitates the removal of weights with relatively larger magnitudes.

Regarding the variation in prediction error, unstructured pruning did not exhibit a simple monotonic relationship where higher pruning rates necessarily led to larger errors. As the pruning rate increased, the prediction error dropped significantly within the 10%–20% range, reaching an optimal level at a pruning rate of around 20% (with an MAE of approximately 3.9×10^{-3} and an RMSE of approximately 5.8×10^{-3}). This indicates that appropriately removing redundant weights from the network did not severely degrade model performance at low sparsity levels.

When the pruning rate is further increased to the 30%–60% range, both MAE and RMSE fluctuate. At 80%–90%, prediction errors rise; at 90%, the MAE is approximately 5.3×10^{-3} and the RMSE approximately 7.9×10^{-3} . This indicates reduced predictive capacity at high pruning rates.

All pruning comparisons use the same frozen FP32 baseline checkpoint. A 0% pruning configuration therefore represents that baseline; any separately fine-tuned model is reported as a distinct experiment. Pruning configurations are selected using validation performance and evaluated on the test set only after selection.

Figure 5.7 compares the original model, the selected unstructured-pruned model, and the actual SoH curve. The example shows that the pruned model follows the overall degradation trend.

Unlike unstructured pruning, structured pruning targets the structured computational units within an LSTM network—such as hidden units—for removal, thereby directly altering the network’s architecture. Consequently, this method not only

5. Results

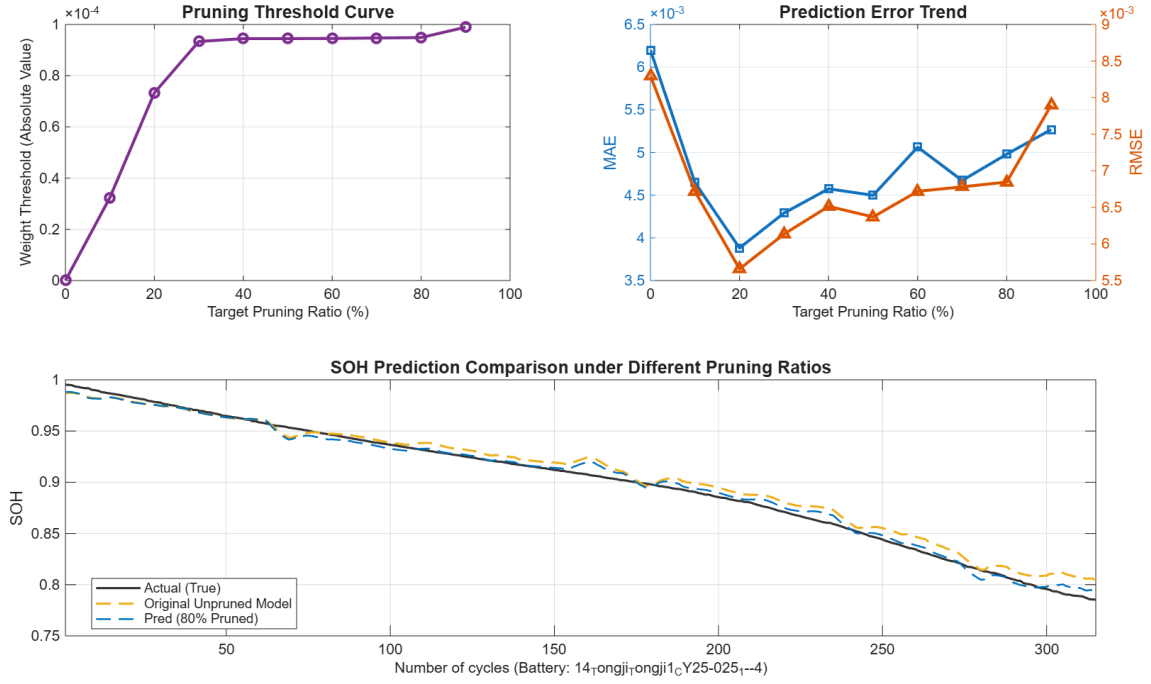


Figure 5.7: Unstructured pruning scan results of the baseline LSTM model, including the pruning threshold, MAE, RMSE, and representative SoH prediction curves over target sparsity levels from 0% to 90%

reduces the number of parameters but also directly decreases the model’s computational scale.

Figure 5.8 presents the experimental results for the LSTM model following Group Lasso structured pruning. As clearly shown in the top-left plot, the number of retained hidden units in the LSTM decreases gradually from 32 to 3 as the target pruning rate rises from 0% to 90%. At the same time, the number of learnable model parameters continuously decreases as the structured pruning rate increases. As shown by the axis on the right side of the figure, the parameter count of the original model is approximately 3.2×10^4 ; it drops to about 2.0×10^4 at a 90% pruning rate. Therefore, structured pruning enables the simultaneous compression of both the model structure and the parameter scale by directly reducing the number of LSTM hidden units.

Regarding the prediction error results, structured pruning also exhibits non-monotonic variation. When the pruning rate increases to the 10%–20% range, the prediction error decreases significantly; specifically, at a 20% pruning rate, the MAE is approximately 3.6×10^{-3} and the RMSE is approximately 5.1×10^{-3} , representing the best results observed across the structured pruning experiments. This indicates that, under moderate structured compression, removing certain redundant hidden units does not compromise the model’s primary feature extraction capabilities; instead, it may reduce redundant representations within the model.

When the pruning rate is further increased to 60%, the MAE rises to approximately 5.2×10^{-3} . At 90%, the MAE is approximately 4.3×10^{-3} and the RMSE approximately 5.9×10^{-3} . These values describe the observed SoH prediction error and do

not establish a general structural or causal explanation.

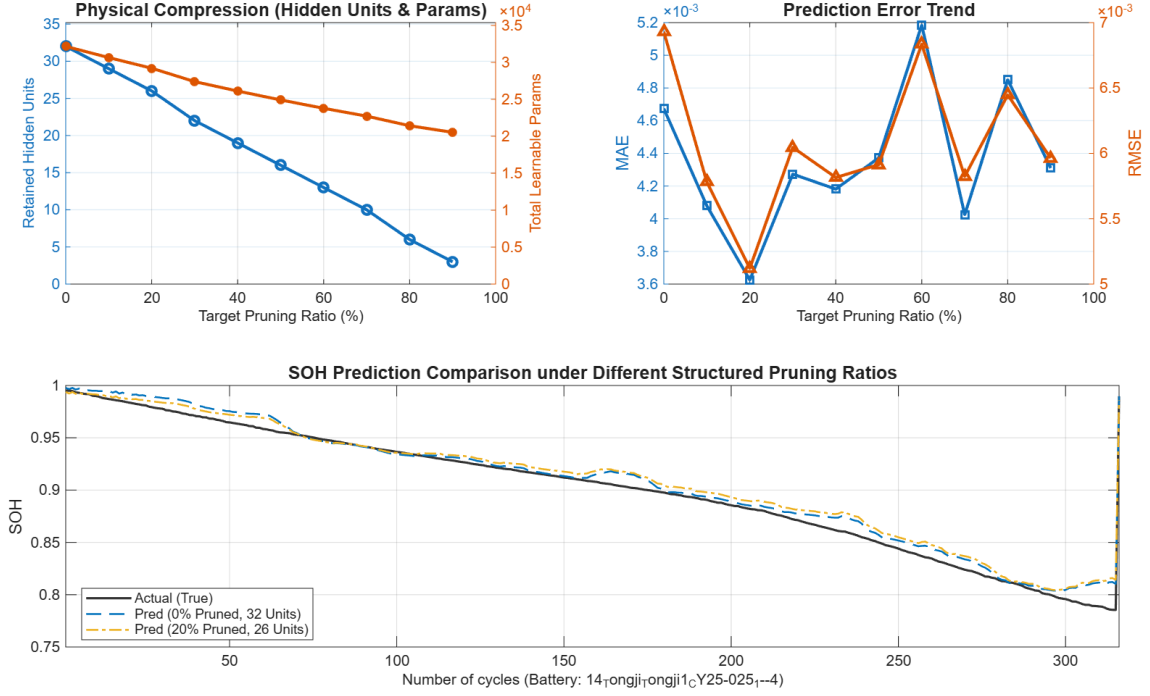


Figure 5.8: Group Lasso structured pruning results of the LSTM model, showing the relationship between the target pruning rate, the number of retained hidden units, model parameter count, prediction error (MAE and RMSE), and the resulting SoH prediction curves.

5.4 Result

To further evaluate storage reduction, this report conducts INT8 quantization experiments on the LSTM model. PTQ and QAT are compared in terms of prediction accuracy and theoretical parameter storage.

Figure 5.9 shows the INT8 model based on PTQ. In the reported example, the quantized model follows the overall SoH degradation trend of the FP32 model.

From the quantitative evaluation metrics, the global RMSE of the original FP32 model is 0.0102, while the RMSE of the PTQ-based INT8 model is 0.0124. Compared with the original model, the RMSE increased by approximately 0.0022, and the error increase was approximately 21.6%. This result indicates that directly mapping the model parameters from FP32 to INT8 without additional training introduces some quantization error, leading to a slight decrease in prediction accuracy.

Meanwhile, the model storage space was significantly compressed. As shown in the precision-storage tradeoff analysis below Figure 5.9, the theoretical storage space of the FP32 model is approximately 128 KB, while that of the INT8 model is only approximately 32 KB. The model size is reduced to 25% of its original size, achieving a storage compression rate of approximately 75%. This result is consistent with the

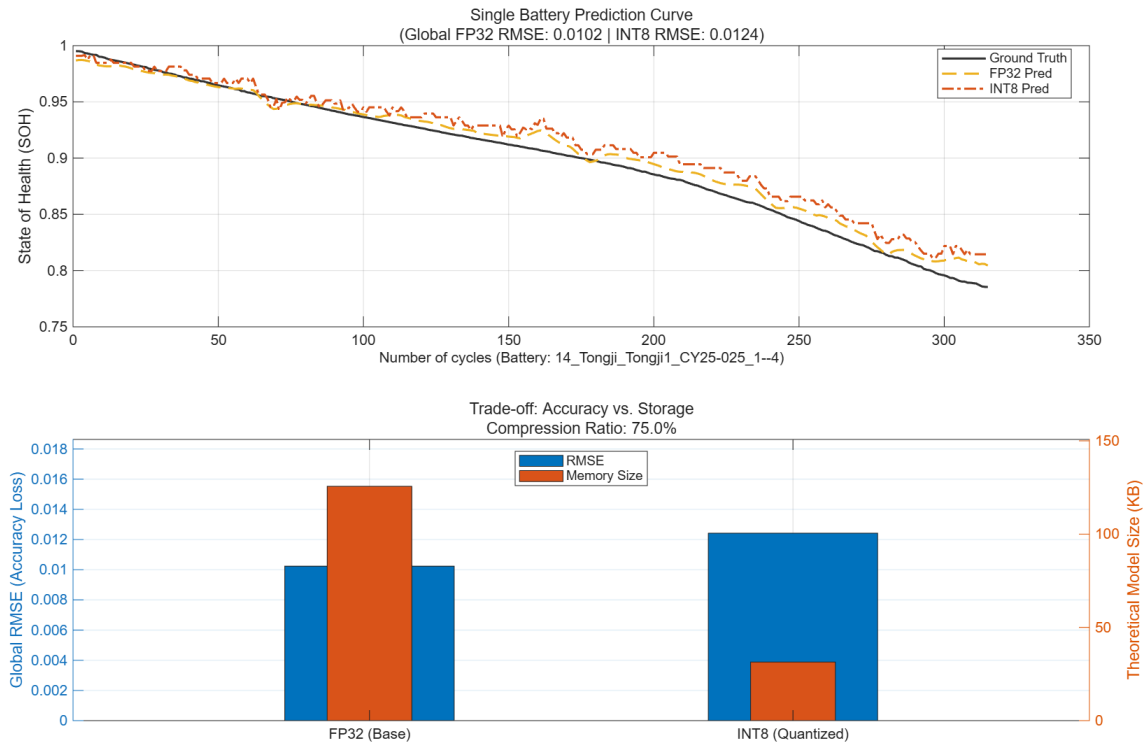


Figure 5.9: Post-training INT8 quantization results comparing the FP32 baseline and the INT8 model in terms of SoH prediction and theoretical parameter storage, including the corresponding RMSE values and the reduction in theoretical storage from approximately 128 KB to 32 KB.

theoretical compression effect brought about by the conversion from FP32 (32-bit) to INT8 (8-bit).

In comparison, this report also uses QAT to test the model. QAT introduces a simulation of quantization operations during the training phase, allowing the network parameters to adapt to INT8 quantization errors in advance, thereby improving the prediction accuracy of the quantized model.

Figure 5.10 shows the QAT INT8 model. In the reported comparison, QAT follows the true SoH trend more closely than PTQ.

From the perspective of global error metrics, the RMSE of the QAT INT8 model is 0.0113, while the RMSE of the corresponding FP32 baseline model is 0.0102. Compared to PTQ's 0.0124, QAT reduces the quantized RMSE by approximately 0.0011, resulting in an error reduction of approximately 8.9%. This demonstrates that quantization-aware training effectively compensates for the accuracy loss during INT8 quantization, enabling the quantized model to achieve better prediction performance while maintaining a high compression ratio.

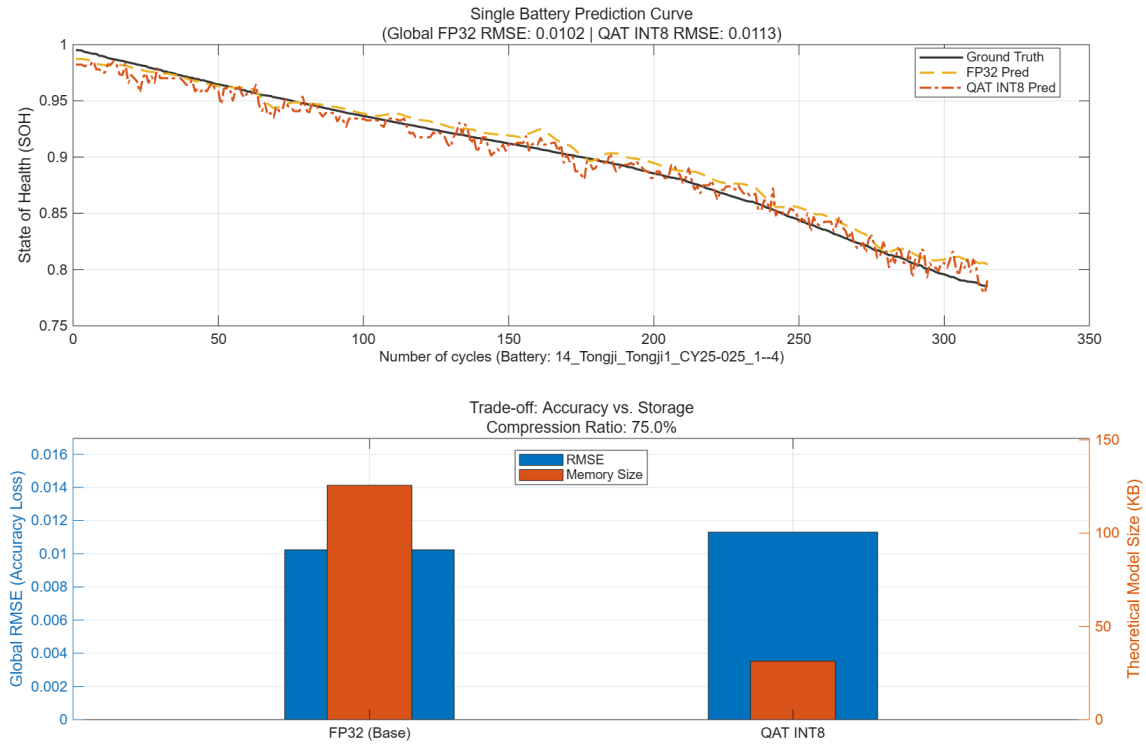


Figure 5.10: Quantization-aware training (QAT) results for the LSTM model, comparing the FP32 baseline and the INT8 quantized model in terms of SoH prediction performance and theoretical parameter storage, including the corresponding prediction curves, RMSE values, and model storage estimates.

Regarding model storage, QAT and PTQ use the same INT8 parameter representation, thus exhibiting similar compression effects. As shown in the figure, the theoretical storage space of the QAT INT8 model is also approximately 32 KB, achieving a 75% model compression ratio compared to the 128 KB FP32 model. Therefore,

QAT can further improve the prediction accuracy of the quantized model without increasing model storage overhead.

5.5 Combination results

To further explore the limits of model compression and attempt to minimize the deployment cost on resource-constrained embedded platforms, this study investigated a comprehensive compression scheme that combines pruning (both unstructured and structured) with INT8 quantization (PTQ and QAT). The goal was to evaluate whether the theoretical storage benefits of applying quantization to an already pruned model could be achieved without destroying predictive performance.

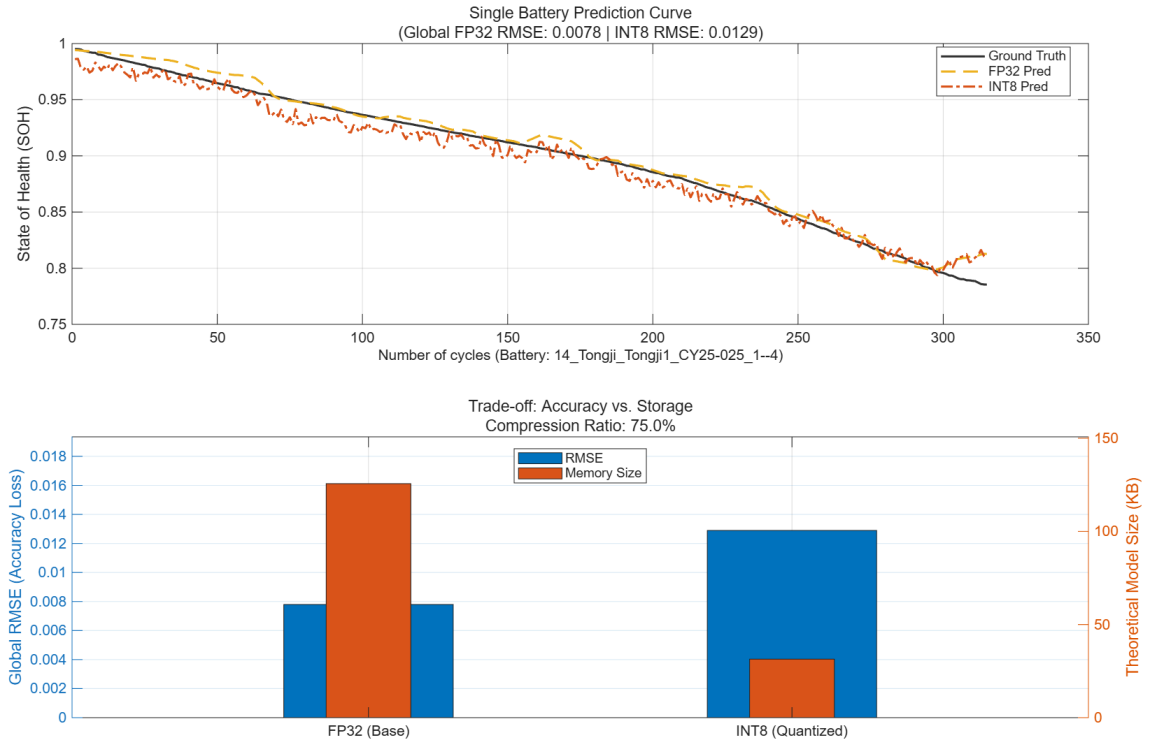


Figure 5.11: Post-training INT8 quantization results for the unstructured-pruned LSTM model, comparing the fine-tuned FP32 model and its INT8 PTQ counterpart in terms of SoH prediction behavior, RMSE, and theoretical parameter storage.

Figure 5.11 shows the experimental results of applying PTQ to the unstructured pruned model. From a quantitative perspective, the global RMSE of the fine-tuned unstructured pruned FP32 model was 0.0078. After PTQ quantization, the RMSE increased to 0.0129. This represents an error increase of approximately 65.4% compared to the unquantized pruned baseline. While the theoretical storage space is reduced to roughly 32 KB, this significant degradation in prediction performance indicates that superimposing low-precision quantization noise onto an already sparsified network severely impacts its accuracy.

The degradation is far more severe when combining structured pruning with PTQ, as shown in Figure 5.12. The global RMSE rises from 0.0069 to 0.0307, an increase

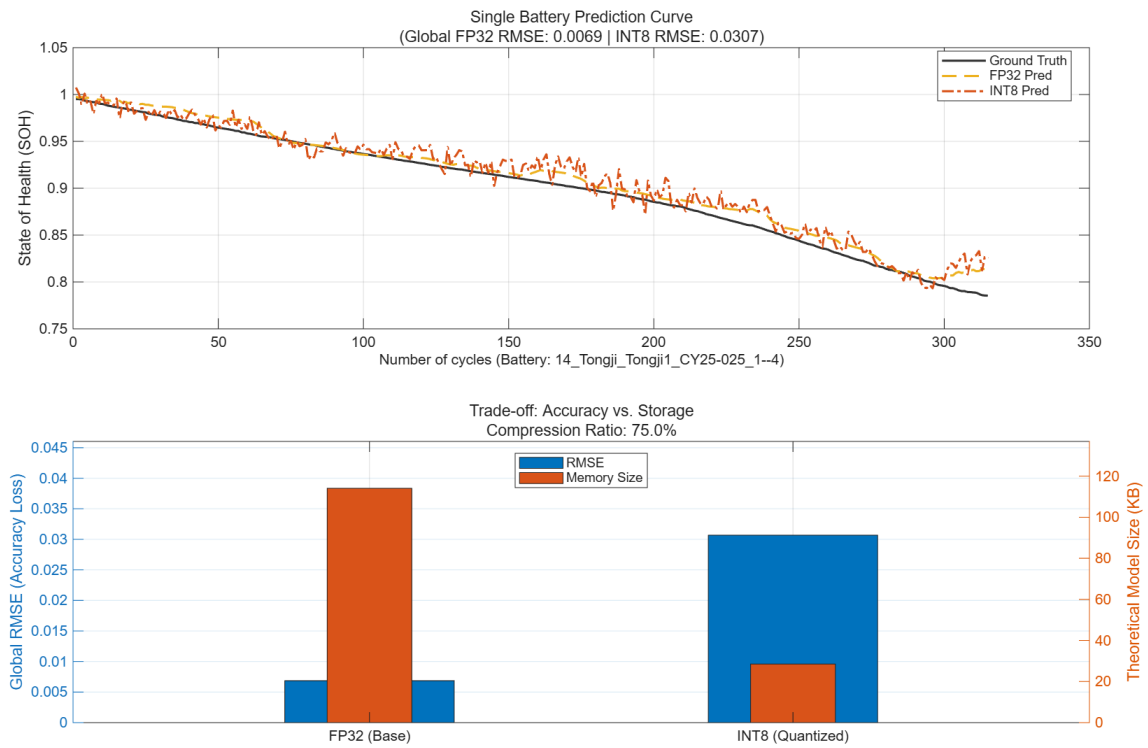


Figure 5.12: Post-training INT8 quantization results for the structured-pruned LSTM model, comparing the structured-pruned FP32 model and its INT8 PTQ counterpart in terms of SoH prediction behavior, RMSE, and theoretical parameter storage.

5. Results

exceeding 300%. Under these settings, the combined model no longer reliably tracks the SoH degradation trend.

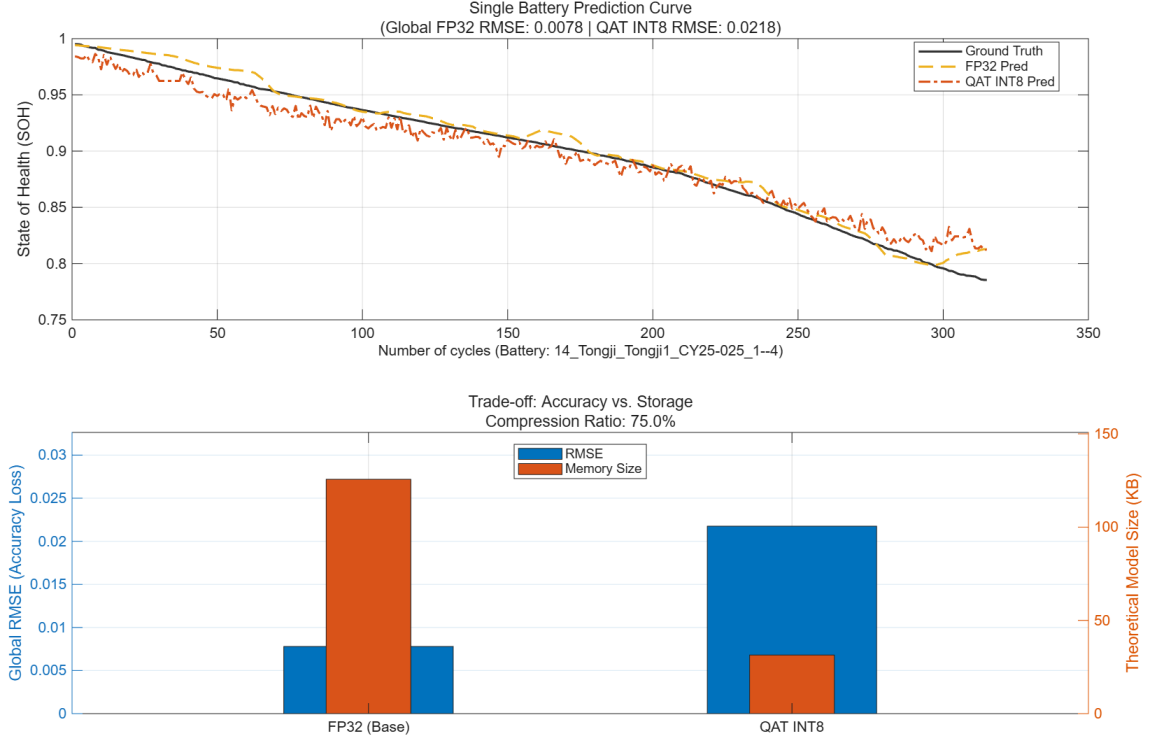


Figure 5.13: Quantization-aware training (QAT) results for the unstructured-pruned LSTM model, comparing the unstructured-pruned FP32 model and the corresponding INT8 QAT model in terms of SoH prediction performance, RMSE, and theoretical parameter storage.

To investigate whether QAT could recover the accuracy lost during combined compression, QAT was applied to the pruned models (as shown in Figure 5.13, Figure 5.14, and summarized in Table 5.2 and Table 5.3).

For the structured pruned model, QAT demonstrated a slight ability to compensate for quantization noise, reducing the quantized RMSE from 0.0307 (PTQ) to 0.0278 (an improvement of approximately 9.4%). However, the absolute error (0.0278) remains unacceptably high compared to the FP32 baseline (0.0069).

For the unstructured pruned model, QAT achieved an RMSE of 0.0218, which underperformed compared to the PTQ result (0.0129) under the current experimental settings. This suggests that applying QAT to a sparsely masked network may introduce optimization instabilities, failing to bring the expected benefits.

Contrary to initial hypotheses, the empirical results clearly demonstrate that the current pruning-plus-quantization combinations perform poorly. While stacking these techniques theoretically reduces the model size to an absolute minimum (a 75% reduction on top of the pruning reduction), it results in unacceptable accuracy degradation. The network simply lacks the representational capacity to absorb INT8 rounding noise once its structural or magnitude-based redundancy has been stripped

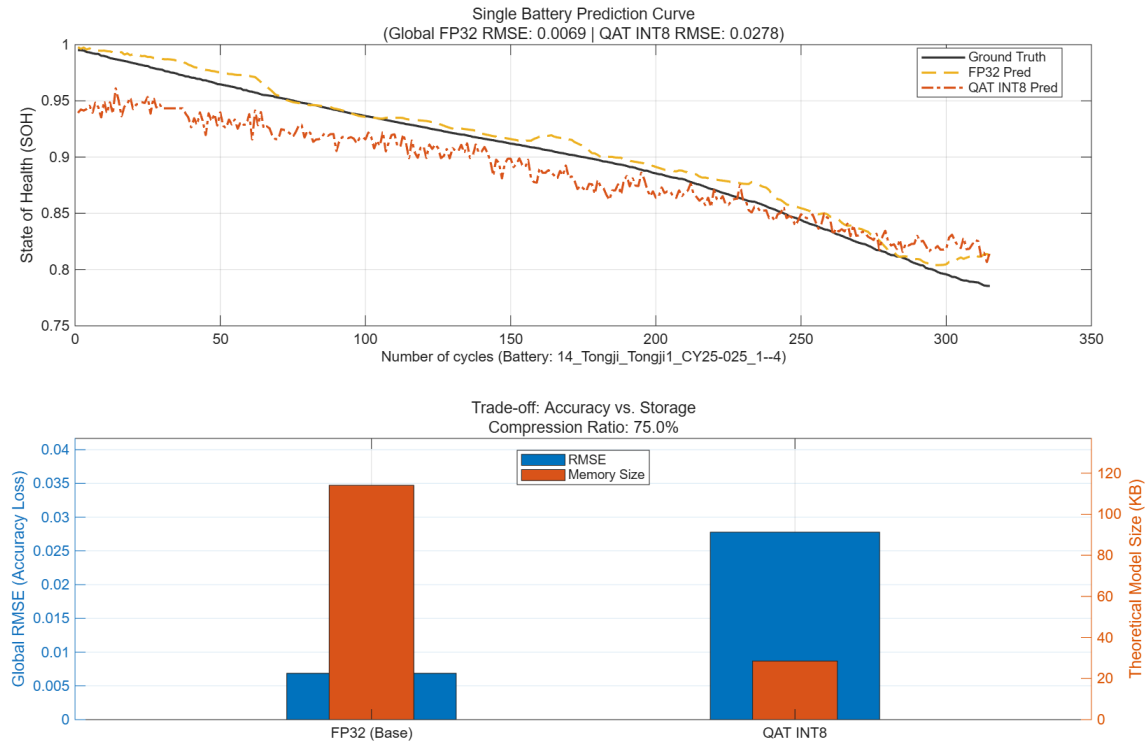


Figure 5.14: Quantization-aware training (QAT) results for the structured-pruned LSTM model, comparing the structured-pruned FP32 model and the corresponding INT8 QAT model in terms of SoH prediction performance, RMSE, and theoretical parameter storage.

Table 5.2: Performance comparison of different pruning strategies combined with QAT quantization

Compression scheme	FP32 RMSE	QAT INT8 RMSE	RMSE Increment	Storage compression ratio
Unstructured pruning + QAT	0.0078	0.0218	+0.0140	75%
Structured pruning + QAT	0.0069	0.0278	+0.0209	75%

Table 5.3: Comparison of PTQ and QAT quantification results

Compression scheme	PTQ RMSE	QAT RMSE
Unstructured pruning	0.0129	0.0218
Structured pruning	0.0307	0.0278

away. Therefore, these combined compression models are not recommended as deployment candidates under the tested settings. The results highlight a limitation of aggressively stacking compression techniques. Within this experiment, standalone QAT performed better than standalone PTQ, while structured FP32 pruning offered a useful parameter-reduction comparison; selecting a deployment configuration requires target-specific measurements that are outside this study.

5.6 Simulink Closed-loop Verification

After completing the extreme hardware-aware compression of deep models, seamlessly and safely converting deep learning operators—originally existing as dynamic graphs—into low-level machine instructions executable by automotive-grade MCUs is the final chasm that must be crossed to bridge academic research and engineering deployment. This section details the full-pipeline conversion and verification process, from the MATLAB deep learning framework to Simulink dynamic simulation verification, and finally to the generation of standalone embedded C++ code.

5.6.1 Real-Time Streaming Simulation and Verification

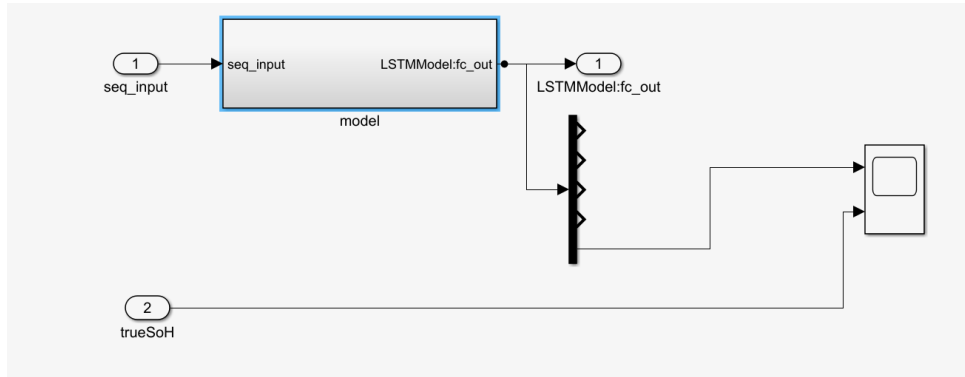


Figure 5.15: Simulink Model-in-the-Loop verification architecture for the compressed LSTM model, showing the streaming input path, stateless Predict Block, Demux-based extraction of the final time-step prediction, and output routing used for comparison with the MATLAB reference implementation.

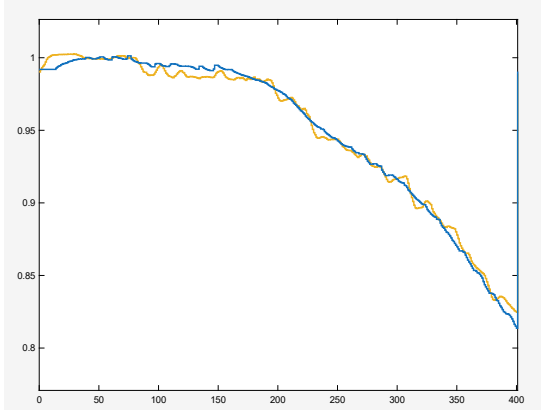
To ensure that the execution logic (e.g., operator precision, time-stepping behavior) of the compressed model in an actual in-vehicle embedded environment remains strictly consistent with that in the deep learning training framework, this study introduces rigorous Model-in-the-Loop simulation verification using Simulink prior to code generation.

The deployment layer transforms the compressed model into production embedded C++ code, as shown in Figure 4.2. The compressed network is first exported to a Simulink subsystem via `exportNetworkToSimulink` with frame-based processing enabled and stateful mode disabled. The Simulink model is then augmented with Inport, Demux, and Scope blocks for input/output routing.

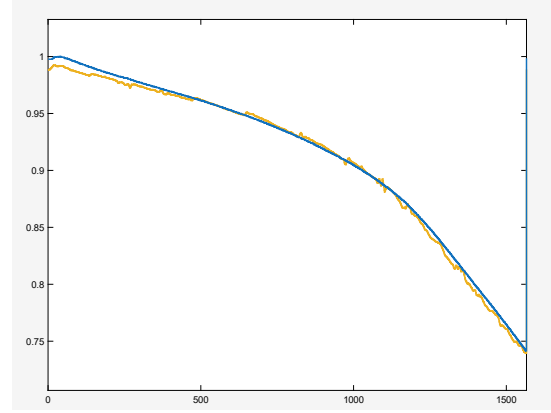
Regarding the construction of the Simulink simulation topology, to accurately capture the output at the end of the sliding window sequence, a signal separator (Demux) was dynamically connected to the output of the Predict Block. This module decouples the multiple time steps of the unrolled LSTM output and precisely extracts the SoH prediction scalar at the final moment (Last Time Step) of the sequence.

As shown in Figure 5.15, the Simulink verification stage separates model inference from input and output routing. The Predict Block performs the LSTM inference, while the Demux block separates the sequence outputs so that the prediction associated with the final time step can be extracted. This structure provides a consistent interface for streaming test sequences and allows the Simulink output to be compared with the corresponding MATLAB prediction. Therefore, the figure documents the functional integration required before proceeding to embedded C++ code generation rather than serving as a performance benchmark.

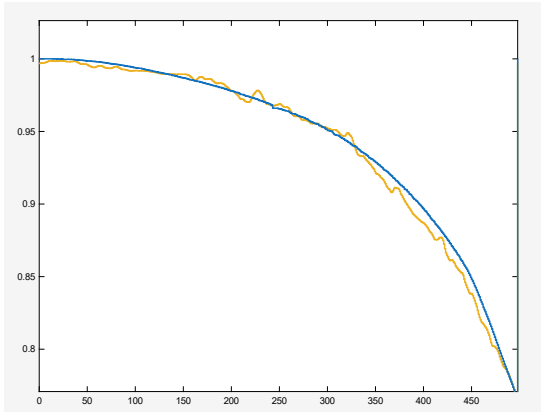
The simulation results (`simOut`) provide a functional model-in-the-loop comparison between the Simulink module and the MATLAB reference for the tested sequences. This is software-level evidence only; it is not a proof of numerical equivalence for all inputs or of on-target MCU hardware verification.



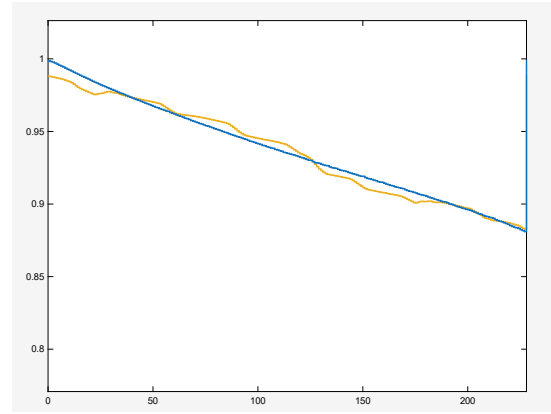
(a) Xi'an Jiaotong University



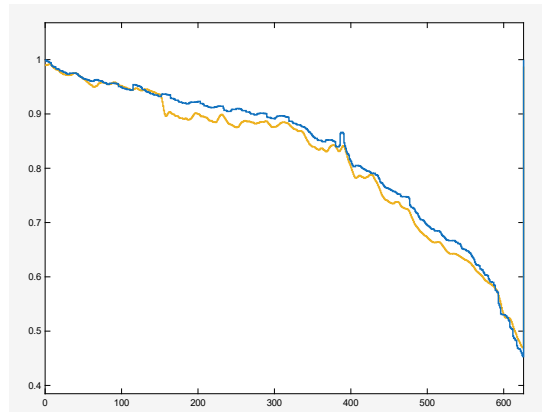
(b) Huazhong University of ST



(c) NREL (Battery Archive)



(d) Tongji University



(e) University of Maryland

Figure 5.16: Simulink architecture output illustration. This serves as a software-in-the-loop functional validation rather than proof of on-target MCU hardware verification.

5.6.2 C++ Embedded Code Generation

Following the successful Simulink MIL simulation verification, this study utilized the Embedded Coder (using the `ert.tlc` system target file) to complete automated code generation for embedded edge devices.

While the generated code is highly portable and compiles successfully via the TASK-ING toolchain targeting the AURIX TC4D7 architecture, it is important to acknowledge that physical on-target profiling (such as real-time latency cycles and dynamic power measurements) was not within the primary scope of this verification phase. The validation emphasizes the automated translation, static memory allocation mapping, and functional mathematical equivalence of the generated embedded C++ artifacts.

5.7 Summary

In summary, the experiments evaluate a practical pipeline for adapting the Cognivity-provided SoH model to resource-constrained automotive MCUs. The study covers unstructured pruning, structured pruning, PTQ, and QAT.

The pruning experiments show that structured FP32 pruning can provide a useful parameter-reduction trade-off in the reported experiment. Quantization experiments show that the theoretical storage of INT8 parameters is approximately one quarter of FP32 storage; the generated implementation data types were not established by this study.

The combined compression experiments show large accuracy degradation for both pruning-plus-quantization combinations. Although QAT partially improves the structured pruned result relative to PTQ, the absolute RMSE remains high; these configurations are therefore unsuccessful under the tested settings. Standalone QAT performs better than standalone PTQ for the unpruned baseline.

Overall, the results establish transfer and compression behavior, but do not demonstrate acceptable accuracy for the combined configurations or prove reduced computational cost, power consumption, deterministic real-time execution, or completed hardware deployment. Cross-compilation and Simulink checks should be understood as feasibility evidence; on-target profiling and compiler memory reports remain future work.

6

Discussion

6.1 Interpretation of the Results

The baseline LSTM model and its original training were provided by Cognivity AI. Accordingly, the results should be interpreted as an evaluation of transfer, compression, and deployment preparation rather than as evidence of a newly developed predictive architecture.

The PyTorch-to-MATLAB comparison supports the numerical equivalence of the transferred network within the measured error reported in Chapter 5. The prediction curves in Figures 5.4–5.6 are illustrative examples for selected datasets; they do not by themselves establish generalization across all batteries or operating conditions.

Standalone structured pruning can provide a useful FP32 accuracy–parameter trade-off in the reported experiment. Standalone QAT performs better than standalone PTQ for the unpruned baseline, while both methods have similar theoretical INT8 weight-storage requirements. These storage figures assume one byte per quantized parameter and should not be confused with the data types or memory reported by generated implementations.

The combined pruning-plus-quantization experiments show substantial RMSE degradation. They are therefore unsuccessful under the present settings and should be treated as a limitation of the evaluated workflow, not as deployment candidates. The results also do not demonstrate that pruning eliminated overfitting.

6.2 Deployment Scope and Limitations

Simulink model-in-the-loop checks and cross-compilation demonstrate software integration and compilation feasibility. They do not prove numerical or hardware verification on the target MCU, INT8 arithmetic in generated code, deterministic real-time execution, reduced power consumption, or completed hardware deployment. Processor-in-the-loop execution, compiler-reported Flash/RAM usage, measured latency, CPU load, and power or energy measurements remain future work.

6.3 Ethical Statement

The baseline model and associated model-development decisions were provided by Cognivity AI; this thesis reports the authors’ transfer, compression evaluation, and

deployment-toolchain work. Cognivity-specific model know-how is intentionally limited to the detail needed to reproduce the validation. The study does not claim measured energy savings, improved safety performance, or real-time operation. Generative AI tools were used only for language editing, and the authors remain responsible for the technical content and reported results.

7

Conclusion

The baseline LSTM model, its original implementation, and its training were provided by Cognivity AI. This thesis therefore focused on evaluating a reproducible transfer, compression, and deployment workflow rather than developing a new predictive model.

The PyTorch-to-MATLAB transfer was checked numerically, and the selected network was integrated into Simulink for model-in-the-loop functional checks. Unstructured and structured pruning, PTQ, and QAT were evaluated using a consistent baseline checkpoint, preprocessing version, data split, and evaluation procedure. Compression configurations were selected with training/validation data and tested only after they were frozen.

The reported experiments indicate that structured FP32 pruning can be useful, and that standalone QAT performs better than standalone PTQ in the unpruned comparison. In contrast, the current pruning-plus-quantization combinations cause large accuracy degradation and are not successful deployment candidates under the tested settings. No claim is made that pruning eliminated overfitting.

INT8 storage numbers in this thesis are theoretical parameter-storage estimates. They are separate from the model exported to Simulink, the data types used by generated code, and any compiler-reported Flash or RAM usage. Embedded C++ generation and cross-compilation for the Infineon KIT_A3G_TC4D7_LITE platform demonstrate toolchain feasibility only. On-target execution time, CPU load, power, energy, schedulability, and completed hardware deployment remain future work.

Bibliography

- [1] NVIDIA Corporation, *NVIDIA DRIVE Orin Architecture Technical Brief*, NVIDIA Corporation, 2020, accessed: 2026-05-28. [Online]. Available: <https://www.nvidia.com/en-us/self-driving-cars/drive-platform/>
- [2] S. Hosseini, R. Dai, and M. Mesbahi, “Optimal path planning and power allocation for a long endurance solar-powered UAV,” in *2013 American Control Conference*, 2013, pp. 2588–2593.
- [3] F. Wang, Z. Zhai, B. Liu, S. Zheng, Z. Zhao, and X. Chen, “Open access dataset, code library and benchmarking deep learning approaches for state-of-health estimation of lithium-ion batteries,” *Journal of Energy Storage*, vol. 77, p. 109884, 2024.
- [4] L. Xia, E. Najafi, H. Bergveld, and M. Donkers, “A computationally efficient implementation of an electrochemistry-based model for lithium-ion batteries,” vol. 50, 07 2017.
- [5] H. A. A. Ali, L. H. Raijmakers, K. Chayambuka, D. L. Danilov, P. H. Notten, and R.-A. Eichel, “A comparison between physics-based Li-ion battery models,” *Electrochimica Acta*, vol. 493, p. 144360, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0013468624006029>
- [6] M. A. George, A. M. George, D. V. Kamath, and C. P. Kurian, “EV speed tracking using the regression-based supervised machine learning algorithms,” in *2024 First International Conference for Women in Computing (InCoWoCo)*, 2024, pp. 1–8.
- [7] H. Mediouni, S. El Hani, A. Ezzouhri, A. Daghour, I. Aboudrar, and N. Naseri, “State of charge estimation for lithium-ion batteries using machine learning algorithms,” in *2022 IEEE International Conference on Electrical Sciences and Technologies in Maghreb (CISTEM)*, vol. 4, 2022, pp. 1–5.
- [8] G. Xu, J. Xu, and Y. Zhu, “LSTM-based estimation of lithium-ion battery SOH using data characteristics and spatio-temporal attention,” *PLoS One*, vol. 19, no. 12, p. e0312856, 2024.
- [9] D. P. Pau and A. Aniballi, “Tiny machine learning battery state-of-charge estimation hardware accelerated,” *Applied Sciences*, vol. 14, no. 14, p. 6240, 2024.
- [10] X. Wang and W. Jia, “Optimizing edge AI: a comprehensive survey on data, model, and system strategies,” *arXiv preprint arXiv:2501.03265*, 2025.

- [11] H. Kour and M. K. Gupta, "Hybrid LSTM-TCN model for predicting depression using Twitter data," in *2022 17th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, 2022, pp. 167–172.
- [12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
- [13] R. Wang, R. Li, T. Li, Y. Zheng, Y. Zhao, C. Wang, C. Cheng, and K. Zhu, "Hybrid integer post-training quantization for CNN-Transformer neural networks," in *2025 5th International Symposium on Artificial Intelligence and Big Data (AIBDF)*, 2025, pp. 40–43.
- [14] S. Seo and J. Kim, "Hybrid approach for efficient quantization of weights in convolutional neural networks," in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2018, pp. 638–641.
- [15] A. Sabovic, M. Aernouts, D. Subotic, J. Fontaine, E. De Poorter, and J. Famaey, "Towards energy-aware tinyML on battery-less IoT devices," *Internet of Things*, vol. 22, p. 100736, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2542660523000598>
- [16] M. Sudarshan, J. V. R. Vajja, and V. Tomar, "DegradAI: A scalable framework for early battery health diagnosis from limited data," *npj Clean Energy*, vol. 1, 08 2025.
- [17] L. Li, Y. Li, R. Mao, L. Li, W. Hua, and J. Zhang, "Remaining useful life prediction for lithium-ion batteries with a hybrid model based on tcn-gru-dnn and dual attention mechanism," *IEEE Transactions on Transportation Electrification*, vol. 9, no. 3, pp. 4726–4740, 2023.
- [18] S. Qiu, B. Zhang, Y. Lv, J. Zhang, and C. Zhang, "A lithium-ion battery remaining useful life prediction model based on CEEMDAN data preprocessing and HSSA-LSTM-TCN," *World Electric Vehicle Journal*, vol. 15, no. 5, 2024. [Online]. Available: <https://www.mdpi.com/2032-6653/15/5/177>
- [19] Z. Liu, Y. Wang, and J. He, "Knowledge-enhanced time series anomaly detection for lithium battery cell screening," *Processes*, vol. 14, no. 2, 2026. [Online]. Available: <https://www.mdpi.com/2227-9717/14/2/371>
- [20] X. Hu, S. Li, and H. Peng, "A comparative study of equivalent circuit models for li-ion batteries," *Journal of Power Sources*, vol. 198, pp. 359–367, 2012. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0378775311019628>
- [21] D. Deng, J. Qiao, J. Qi, S. Wang, S. Jin, X. Xiao, X. Hao, and Y. Shang, "Chapter 6 - equivalent modeling and parameter identification of power lithium-ion batteries," in *State Estimation Strategies in Lithium-ion Battery Management Systems*. Elsevier, 2023, pp. 95–124. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780443161605000019>
- [22] D. P. Laudani, D. Milillo, M. Quercio, F. Riganti Fulginei, and L. Sabino, "A comprehensive review of equivalent circuit models and neural network models

- for battery management systems,” *Batteries*, vol. 12, no. 1, 2026. [Online]. Available: <https://www.mdpi.com/2313-0105/12/1/37>
- [23] J. L. Elman, “Finding structure in time,” *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1
- [24] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.
- [25] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.
- [26] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.
- [27] E. Frantar and D. Alistarh, “SparseGPT: Massive language models can be accurately pruned in one-shot,” 2023. [Online]. Available: <https://arxiv.org/abs/2301.00774>
- [28] M. Xia, T. Gao, Z. Zeng, and D. Chen, “Sheared LLaMA: Accelerating language model pre-training via structured pruning,” 2024. [Online]. Available: <https://arxiv.org/abs/2310.06694>
- [29] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.
- [30] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” 2016. [Online]. Available: <https://arxiv.org/abs/1510.00149>
- [31] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search,” 2019. [Online]. Available: <https://arxiv.org/abs/1812.03443>
- [32] CALCE Battery Research Group, “Battery research data,” 2026, accessed: 2026-09-21. [Online]. Available: <https://calce.umd.edu/data>
- [33] Y. Yuan, G. Ma, and S. Xu, “The dataset for: Real-time personalized health status prediction of lithium-ion batteries using deep transfer learning,” 2022. [Online]. Available: <https://data.mendeley.com/datasets/nsc7hnsg4s/2>
- [34] Toyota Research Institute and collaborators, “Experimental data platform: Battery datasets,” 2019, battery datasets for fast-charging and battery cycle-life studies. [Online]. Available: <https://data.matr.io/1>
- [35] J. Zhu, Y. Wang, Y. Huang, R. B. Gopaluni, Y. Cao, M. Heere, M. J. Mühlbauer, L. Mereacre, H. Dai, X. Liu, A. Senyshyn, X. Wei, M. Knapp, and H. Ehrenberg, “Data-driven capacity estimation of commercial lithium-ion batteries from voltage relaxation,” 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.6405084>

- [36] F. Wang, “Project - physics-informed neural network for lithium-ion battery degradation stable modeling and prognosis,” 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.10963339>