Application Note

Using MSPM0G5187 in Edge Artificial Intelligence Applications

Sal Ye

Abstract

This application note presents the implementation of edge artificial intelligence (AI) applications on the MSPM0G5187 microcontroller, which integrates the TinyEngine™ neural processing unit (NPU) - a dedicated hardware accelerator designed for efficient neural network inference in resource-constrained embedded systems. The document begins with an overview of the MSPM0G5187 hardware architecture and the key features of the NPU, followed by a comprehensive introduction to TI's edge AI software ecosystem, including the CCSTUDIO™ Edge AI Studio, Tiny ML Tensorlab, and neural network compiler (NNC). Two representative application examples are then presented to demonstrate the Edge AI capabilities of the MSPM0G5187: a handwritten digit recognition design based on a LeNet-5 CNN model, and a waveform classifier design based on a time-series CNN model. Quantitative performance benchmarks comparing NPU/CPU-based inference implementations are provided, demonstrating the substantial advantages of NPU hardware acceleration in inference latency and energy efficiency. The results offer practical guidance and design reference for engineers developing edge AI designs on embedded platforms.

Table of Contents

1 Introduction 3
2 MSPM0G5187 with TinyEngine™ NPU 4
3 Edge AI Toolchains 5
3.1 TI Edge AI Studio 6
3.2 TI Tiny ML Tensorlab 7
3.3 TI Neural Network Compiler 7
4 Rapid Prototyping Through TI Edge AI Studio 9
4.1 Project Setup 9
4.2 Data Capture 11
4.3 Data Import 13
4.4 Data Annotation 16
4.5 Model Training 18
4.6 Performance Reports 22
4.7 Model Export 25
5 Advanced Development Through TI Tiny ML Tensorlab 27
5.1 Environment Setup 27
5.2 Data Acquisition 28
5.3 Model Architecture Iterative Refactoring 30
5.4 Model Export 34
5.5 Quantization-Aware Training 34
5.6 Model Compilation 35
6 Edge AI Application: Digit Recognition 38
6.1 LeNet-5 Variant CNN Model 38
6.2 NPU/CPU Performance Comparison 39
7 Edge AI Application: Waveform Classifier 40
7.1 Feature Extraction 41
7.2 Time-Series Classification Model 43
7.3 Model Memory Considerations 44
7.4 NPU/CPU Performance Comparison 45
8 Summary 46

Copyright © 2026 Texas Instruments Incorporated

9 References 46
10 Revision History 46

Trademarks

TinyEngine™, CCSTUDIO™, C7™, Code Composer Studio™, EnergyTrace™, and E2E™ are trademarks of Texas Instruments.
Arm® and Cortex® are registered trademarks of Arm Limited.
PyTorch® is a registered trademark of Linux Foundation.
Linux® is a registered trademark of Linus Torvalds.
Windows® is a registered trademark of Microsoft Corporation.
All trademarks are the property of their respective owners.

Copyright © 2026 Texas Instruments Incorporated

1 Introduction

Edge Artificial Intelligence (AI) involves executing inference models locally, at or near the data source. Compared to cloud-based AI, Edge AI delivers significant advantages, including enhanced responsiveness through reduced latency, improved operational efficiency via lower bandwidth requirements and reduced cloud dependency, increased system reliability with continued operation during network disruptions, and strengthened data security through minimized data transmission and enhanced privacy protection.

MSPM0G5187 with TinyEngine NPU have facilitated the integration of AI capabilities directly into end devices across multiple domains, including consumer electronics, smart home devices, industrial and automotive systems. The dedicated hardware NPU operating in parallel enables MCUs to achieve up to 90x lower inference latency and 120x lower energy utilization per inference compared to equivalent MCUs without hardware acceleration.

However, Edge AI is more than just hardware - it requires a complete ecosystem to truly accelerate users' development and deployment. From generic classification, regression to forecasting tasks, TI's comprehensive software ecosystem empowers engineers to rapidly prototype, optimize, and deploy AI solutions with minimal complexity.

- **TI Edge AI Studio**: offers an intuitive graphical environment for model development, training, compilation, and deployment
- **TI Tiny ML Tensorlab**: delivers an end-to-end workflow covering model training, quantization, compilation, and deployment via command line tool
- **TI Neural Network Compiler**: allows seamless deployment to either the NPU or CPU through simple configuration

MSPM0 SDK further accelerates development by providing a rich collection of ready-to-use Edge AI example projects, enabling developers to rapidly deploy reference code onto MCUs, import pre-built models for retraining, or integrate custom user-developed AI models into applications.

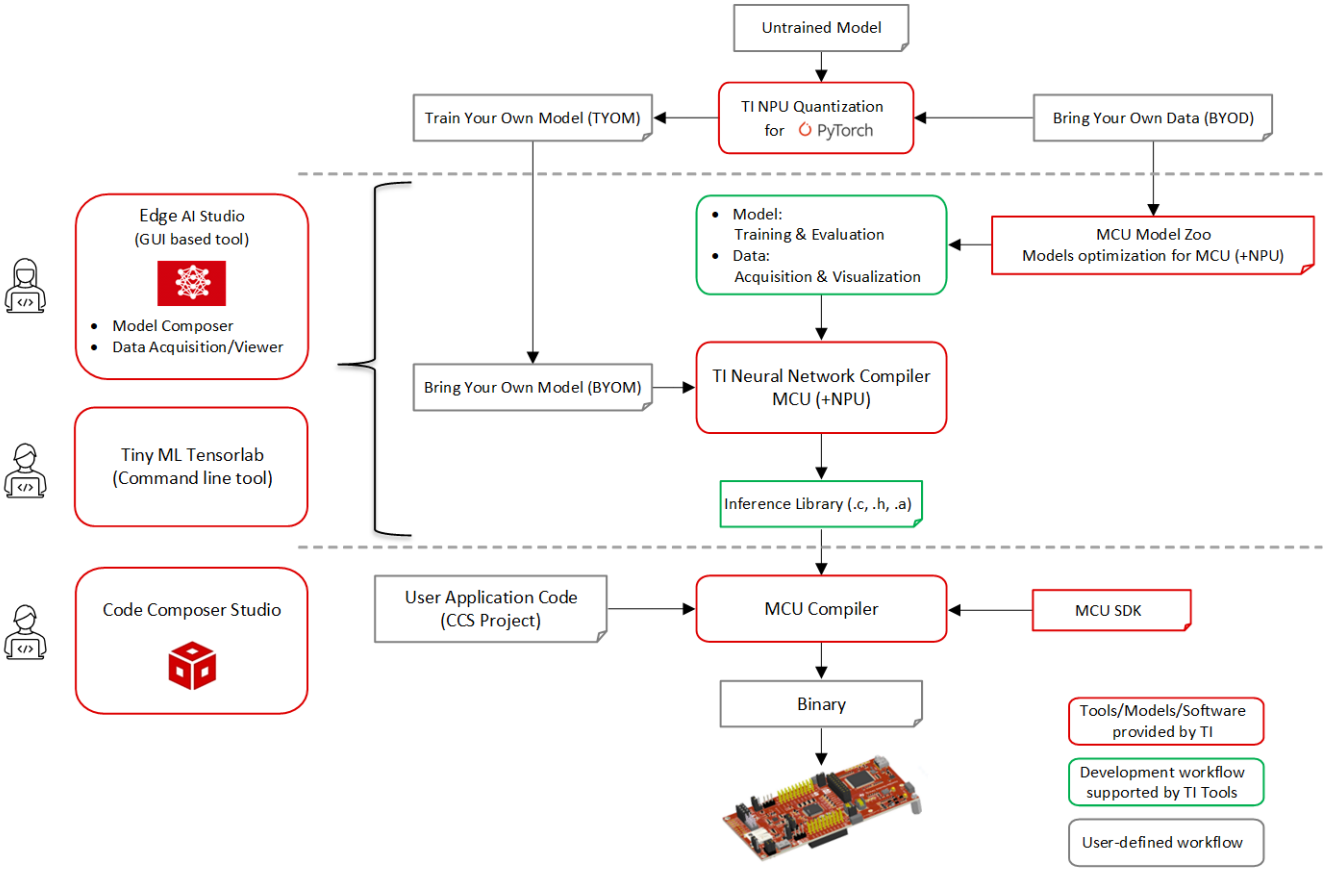


Figure 1-1. Deploy Edge AI Model on MSPM0G5187

Copyright © 2026 Texas Instruments Incorporated

2 MSPM0G5187 with TinyEngine™ NPU

MSPM0G5187 microcontrollers (MCUs) are part of the MSP highly integrated, ultra-low-power 32-bit MCU family based on the enhanced Arm® Cortex®-M0+ 32-bit core platform, operating at up to 80MHz frequency. These MCUs offer a blend of cost optimization and design flexibility for applications requiring up to 128KB of flash memory in small packages or high pin count packages (up to 64 pins). These devices include a USB2.0-FS interface, digital audio interface, TinyEngine NPU, and provide excellent low power performance across the operating temperature range.

The hardware NPU is a highly optimized core for deep convolutional neural networks (CNNs), supporting machine learning inference using pre-trained models. This works in conjunction with the on-chip CPU to provide higher performance and lower power consumption for CNN inference. With capability for 640–2560MOPS (Mega Operations Per Second), the NPU enables 90x lower latency than software implementations and consumes less than 2µA in standby mode.

Figure 2-1 shows the top-level view of the modules within the chip.

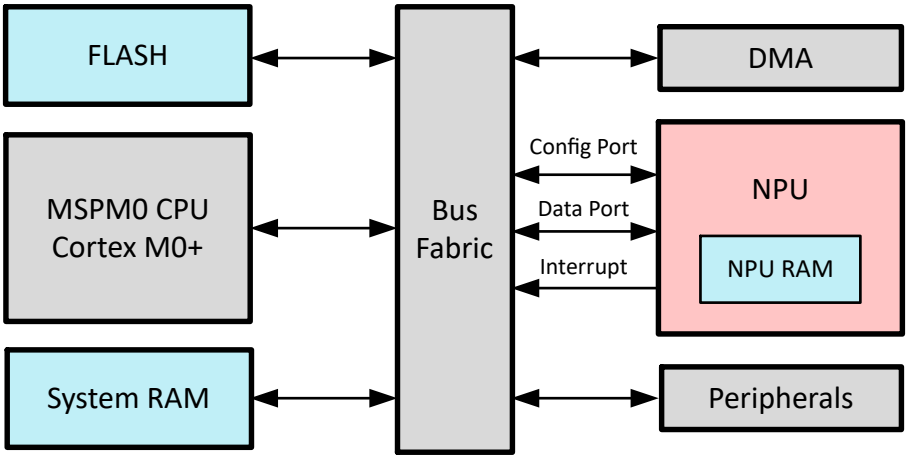


Figure 2-1. MSP CPU/NPU System Structure

MSPM0 with NPU system shares the on-chip memory including non-volatile FLASH memory and volatile RAM memory. The system RAM supports read/write with CPU or NPU. Additionally, NPU has a dedicated RAM for CNN inference. The Edge AI model must fit within the available FLASH memory range, and the peak RAM requirements must fit within the available system RAM.

Copyright © 2026 Texas Instruments Incorporated

3 Edge AI Toolchains

The implementation of Edge AI consists of two distinct phases: the offline training workflow and the on-device runtime inference, as shown in Figure 3-1.

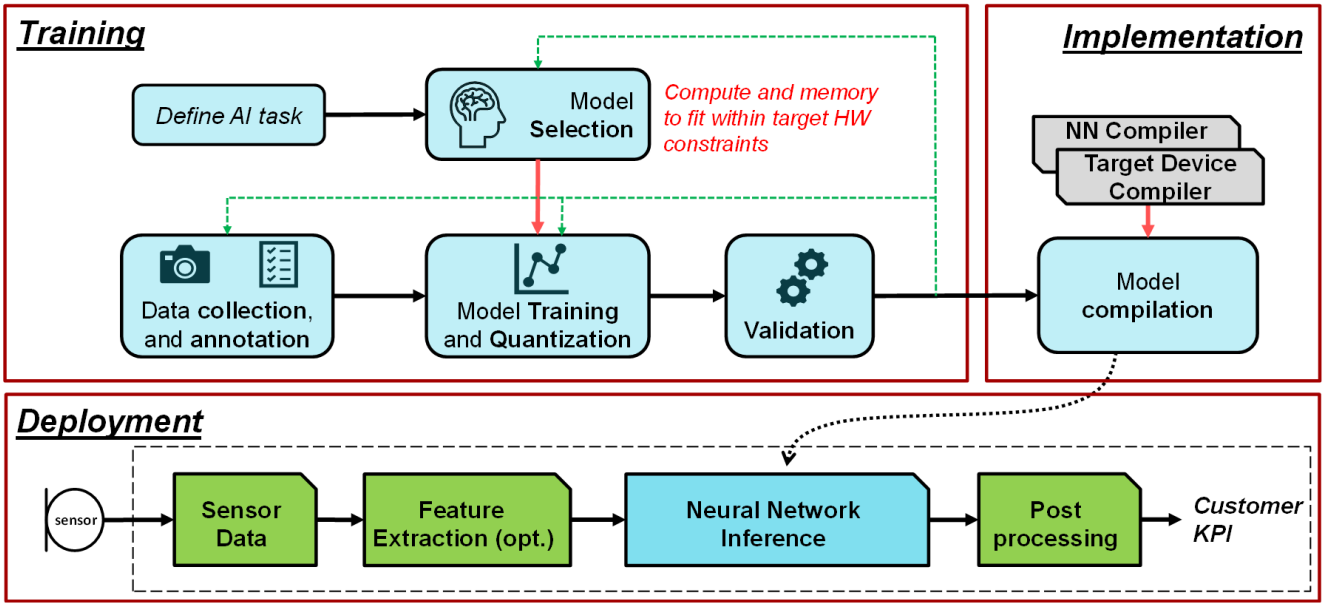


Figure 3-1. Edge AI Workflow

Model Development Workflow

The development pipeline transitions from initial application definition to finalized hardware compilation through four sequential stages:

- **Task Definition & Model Selection:** Core AI workloads are defined based on specific application requirements and operational constraints. Developers can select structurally optimized, highly parameterized reference models from the TI Model Zoo, or alternatively, import proprietary custom network architectures.
- **Dataset Management:** High-fidelity datasets are critical for maximizing model generalization in real-world environments. The Edge AI Studio graphical user interface (GUI) provides a streamlined environment for data acquisition and annotation. The platform also natively supports the ingestion of users' own datasets.
- **Training & Quantization:** Models are trained within a PyTorch® framework and exported through the standard ONNX format. To maximize hardware efficiency, the pipeline utilizes TI-optimized post-training quantization (PTQ) or quantization-aware training (QAT/QDQ) methodologies.
- **Compilation:** The compiled model is translated into highly optimized C code or binary libraries. This artifact allows seamless, zero-overhead integration directly into downstream Code Composer Studio (CCS) firmware projects.

On-Device Runtime Data Flow

Upon successful deployment to the physical hardware, the system executes real-time processing and edge inference via a deterministic pipeline:

- **Signal Acquisition:** Physical transducers (sensors) continuously sample ambient variables—such as acceleration, voltage transients, or acoustic waveforms—converting them into digitized sensor data.
- **Feature Extraction:** Depending on available computational overhead, raw data optionally undergoes localized feature extraction. This stage reduces data dimensionality and reduces the computational burden on the downstream inference engine.
- **Neural Network Inference:** This core execution phase performs high-speed computation and classification. The network workload is heavily accelerated by the hardware neural processing unit (NPU), with the CPU handling system orchestration and control-flow management to minimize latency.
- **Post-Processing:** Raw tensor outputs are decoded, normalized, and transformed into high-level actionable metrics and customer key performance indicators (KPIs).

Copyright © 2026 Texas Instruments Incorporated

In summary, this decoupled architecture enables the entire loop - spanning physical data perception, algorithmic inference, and deterministic decision-making - to be executed entirely localized at the network edge, eliminating cloud dependencies and optimizing deterministic response times.

3.1 TI Edge AI Studio

TI Edge AI Studio is a comprehensive suite of graphical and command line tools designed to simplify the development, training, compilation, and deployment of AI models onto a wide range of Texas Instruments processors, microcontrollers, and radar sensors.

To accommodate varying development environments and security requirements, Edge AI Studio is available in two distinct versions:

- Cloud Version: An entirely web-based environment that requires no local installation. This version enables rapid prototyping and quick start-up with minimal configuration.
- Desktop Version: A local installation that runs on the machine of the user. This version is optimized for projects with strict data confidentiality requirements, verifying that all datasets and trained models remain on-premises and are never accessed by TI.

Edge AI Studio provides a robust workflow for the following tasks:

- **Data Management:** Seamless data capture, annotation, and visualization.
- **Model Training:**
  - Training TI-optimized models using user-supplied datasets (Bring Your Own Data, BYOD).
  - Customization of training parameters to fine-tune model performance.
- **Model Deployment:**
  - Automated compilation of trained models for target hardware.
  - Live inference preview for real-time performance validation.

Certain advanced customization workflows are not currently supported within Edge AI Studio:

- Training user-defined models (Train Your Own Model, TYOM)
- Modification of TI model architectures
- Customization of the feature extraction pipeline or quantization flows
- Direct compilation of pre-trained or customer-provided models

Note

For workflows requiring advanced model customization (such as TYOM or architecture modification), please refer to Section 3.2, which provides extended support for these capabilities.

Edge AI Studio supports over 22 TI microcontrollers across several specialized families, including processors featuring a C7™ NPU or AI-accelerated devices such as microcontrollers featuring a TinyEngine as well as AI-supported devices without a dedicated accelerator. For more detailed information, please visit the EDGE-AI-STUDIO product page.

Copyright © 2026 Texas Instruments Incorporated

3.2 TI Tiny ML Tensorlab

Tiny ML Tensorlab is Texas Instruments' comprehensive AI toolchain for developing, training, optimizing, and deploying machine learning models on resource-constrained microcontrollers (MCUs), which enables users to:

- Train machine learning models for time series and image classification tasks
- Optimize models using quantization (2-bit, 4-bit, 8-bit) for embedded deployment
- Compile models to run efficiently on TI MCUs, with optional NPU acceleration
- Deploy models using Code Composer Studio (CCS)

The toolchain provides an end-to-end workflow that takes you from raw data to a compiled model running on a TI MCU. The toolchain handles:

- **Data Preparation & Feature Extraction:** Automated formatting, preprocessing, and transformation of raw signals into high-dimensional features.
- **Model Training:** Training of neural networks specifically architected for embedded resource constraints.
- **Model Optimization:** Implementation of quantization-aware training (QAT) or post-training quantization (PTQ) to support 2-bit, 4-bit, and 8-bit weight precision.
- **Compilation:** Generation of highly optimized, device-specific code via the TI Neural Network Compiler.
- **Performance Analysis:** Comprehensive evaluation of model accuracy and inference metrics prior to hardware deployment.

TI Tiny ML Tensorlab supports the following Machine Learning (ML) task types:

Table 3-1. Tensorlab-Supported Task Types

Task Type	Description
Time Series Classification	Categorize time-series data into discrete classes (for example, fault detection, activity recognition)
Time Series Regression	Predict continuous values from time-series inputs (for example, torque estimation)
Time Series Forecasting	Predict future values based on historical patterns (for example, temperature prediction)
Anomaly Detection	Identify abnormal patterns using autoencoder-based models (for example, equipment monitoring)
Image Classification	Categorize images into classes (for example, visual inspection, digit recognition)

Tensorlab supports a wide range of TI microcontrollers across several specialized families, including **C2000 DSP**, **MSPM0**, **MSPM33**, **AM13x/AM26x**, and **Connectivity**. For latency-critical and low-power applications, select devices (such as F28P55, AM13E2, and MSPM0G5187) feature TI's integrated Neural Processing Unit (NPU) for hardware-accelerated inference.

Note

The list of supported devices is subject to change as new hardware is integrated. For the most up-to-date and comprehensive list of supported part numbers, please refer to the Tiny ML Tensorlab User's Guide: Supported Devices.

Additional Resources

- Tiny ML Tensorlab GitHub Repository
- Tiny ML Tensorlab User's Guide

3.3 TI Neural Network Compiler

The TI Neural Network Compiler (NNC) is TI's embedded AI compilation toolchain for MCU devices, built on the TVM compiler framework. TI NNC enables flexible deployment of the AI model to either the hardware NPU accelerator or the host CPU through a simple compilation configuration, with no changes required to the application code or model side. The inference is always invoked through the unified interface, enabling developers to integrate AI inference into embedded applications without requiring deep knowledge of the underlying hardware architecture.

Copyright © 2026 Texas Instruments Incorporated

Table 3-2. NPU/CPU Deployment Configuration Comparison

Target Deployment	Configuration Symbol	Initialization Requirement	Invoke Inference
Host CPU	TVMGEN_DEFAULT_TI_NPU_SOFT	Yes, initialize NPU before Inference	tvmgen_default_run()
Hardware NPU	TVMGEN_DEFAULT_TI_NPU	No	

To compile the model for the MSPM0 series, configure your environment and execute the TVMC compilation command as shown below.

Linux® / macOS

```
export ARM_CGT_PATH=/path/to/ARM-CGT
export PATH=$ARM_CGT_PATH/bin:$PATH
export TIARMCLANG_OPTIONS_M0="-Os -mcpu=cortex-m0plus -march=thumbv6m -mtune=cortex-m0plus -mthumb
-mfloat-abi=soft -I. -Wno-return-type"
```

```
tvmc compile --target="c, ti-npu type=soft" --target-c-mcpu=cortex-m0plus ./model.onnx -o
artifacts_m0/mod.a --cross-compiler="tiarmclang" --cross-compiler-options="$TIARMCLANG_OPTIONS_M0"
```

Windows® Git BASH

```
export ARM_CGT_PATH="\path\to\ARM-CGT"    # e.g. "C:\\ti\\ti-cgt-armllvm_4.0.3.LTS"
export PATH=$ARM_CGT_PATH\\bin:$PATH
export TIARMCLANG_OPTIONS_M0="-Os -mcpu=cortex-m0plus -march=thumbv6m -mtune=cortex-m0plus -mthumb
-mfloat-abi=soft -I. -Wno-return-type"
```

```
tvmc compile --target="c, ti-npu type=soft" --target-c-mcpu=cortex-m0plus ./model.onnx -o
artifacts_m0/mod.a --cross-compiler="tiarmclang" --cross-compiler-options="$TIARMCLANG_OPTIONS_M0"
```

Windows PowerShell

```
$env:ARM_CGT_PATH="\path\to\ARM-CGT"      # e.g. "C:\\ti\\ti-cgt-
armllvm_4.0.3.LTS"$env:PATH=$env:ARM_CGT_PATH + "\bin;" + $env:PATH $env:TIARMCLANG_OPTIONS_M0="-Os
-mcpu=cortex-m0plus -march=thumbv6m -mtune=cortex-m0plus -mthumb -mfloat-abi=soft -I. -Wno-return-
type"
```

```
tvmc compile --target="c, ti-npu type=soft" --target-c-mcpu=cortex-
m0plus .\model.onnx -o artifacts_m0/mod.a --cross-compiler="tiarmclang" --cross-compiler-
options="$env:TIARMCLANG_OPTIONS_M0"
```

> If users do not require a binary output (for example, when generating C code only), the ARM Clang compiler is not required. You can omit the cross-compiler arguments from the command.

For more information, see TI Neural Network Compiler for MCUs User's Guide.

Copyright © 2026 Texas Instruments Incorporated

4 Rapid Prototyping Through TI Edge AI Studio

Edge AI Studio is engineered to accelerate prototyping and enable rapid validation of TI-optimized reference models, significantly reducing development overhead.

This section provides a comprehensive overview of the development environment. Table 4-1 summarizes the tools required for this workflow, including their respective versions and access links. Detailed, step-by-step setup instructions follow.

Table 4-1. Development Tools Requirements

Tool	Version	Function	Access Link
Edge AI Studio	1.7.0	GUI tool for ML model development	EDGE-AI-STUDIO-MCU
MSPM0 SDK	2.11	Software development kit for TI MSPM0 MCUs	MSPM0-SDK
ARM CLANG Compiler	5.1.1	Device compiler for TI MSPM0 MCUs	ARM-CGT-CLANG
Code Composer Studio IDE	21.0	Integrated development environment for MSPM0 MCUs	CCStudio™ IDE Tool

Note

The procedures described in this document are based on the Edge AI Studio Desktop version; however, the workflow is similar for the Cloud version.

4.1 Project Setup

To initiate a new AI project in Edge AI Studio, users can navigate to the *Workspace* menu. The platform offers two primary workflows. The standard workflow involves selecting the *"Create New Project"* option, as shown in Figure 4-1.

During this process, users must define the application scope by selecting one of four specialized AI Task categories:

- **Classification:** For tasks involving categorical identification and pattern recognition.
- **Regression:** For predicting continuous numerical values.
- **Anomaly Detection:** For identifying outliers or irregular patterns within a dataset.
- **Forecasting:** For forecasting future data points based on historical trends.

The selection of an *AI Task* is critical, as the platform dynamically configures the underlying training methodologies and optimization algorithms to meet the specific mathematical requirements of the chosen task.

In the *Sensor Selection* page, for best results, TI recommends to select a *Generic Sensor* and subsequently configure its specific type on the *Home* page. If the application requires data from multiple sources, select the *Multi-sensor* option on the *Data Format Definition* page.

Once the project is configured, rename the project to finalize the setup and begin the Edge AI development workflow.

Copyright © 2026 Texas Instruments Incorporated

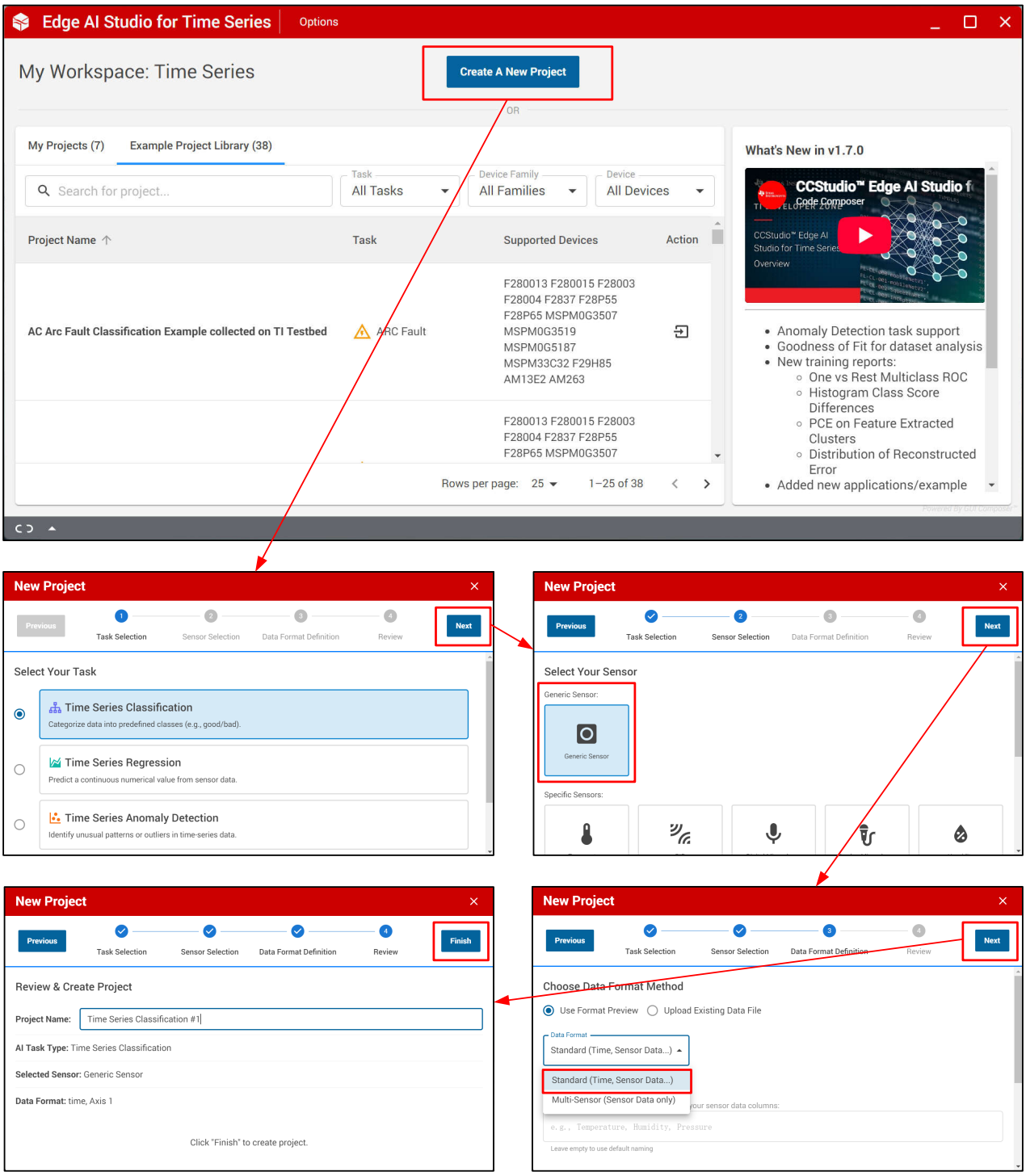


Figure 4-1. Using Edge AI Studio to Create A New Project

Alternatively, users can accelerate the development cycle by leveraging TI's library of pre-configured example projects. Currently, the platform supports 38 specialized projects, which are categorized by target application, specific datasets, or distinct model architectures.

These examples serve as a foundational reference for various industrial and consumer use cases. For instance, within the *Arc Fault Detection* domain, users can choose from several implementation paths:

- **TI Test Data Implementation:** Utilizing proprietary TI datasets for optimized performance.

Copyright © 2026 Texas Instruments Incorporated

- **Public Dataset Implementation:** Utilizing standardized community datasets for general benchmarking.
- **Anomaly Detection Implementation:** Utilizing the specialized *Anomaly Detection AI Task* to identify irregular electrical signatures.

To utilize an existing example, use the *Search* function to locate a specific project via keywords. Once found, select the *Import* button to add the project directly into your active *Workspace* for immediate development and testing.

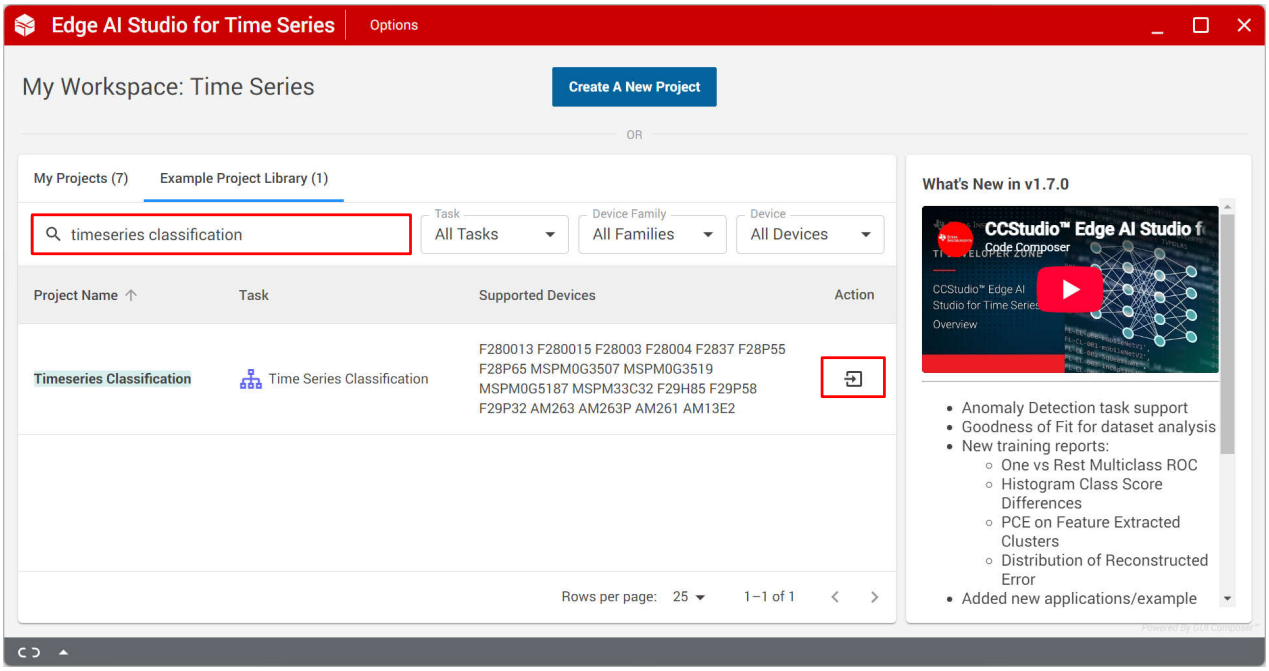


Figure 4-2. Using Edge AI Studio to Import a Reference Project

4.2 Data Capture

This section describes the procedures for acquiring real-time sensor data using a TI microcontroller (MCU) and configuring that data for AI model development within Edge AI Studio.

Two primary workflows are supported for data capture:

- **Real-Time Data Capture:** This workflow utilizes a TI MCU (such as LP-MSPM0G5187) interfaced with a compatible sensor evaluation module (for example, TIDA-010997). The system follows a standardized GUI communication protocol to capture and stream real-time sensor data during runtime.
- **Offline Data Upload:** Users can import pre-captured datasets acquired from other hardware platforms. For detailed instructions on the data import process, refer to Section 4.3.

4.2.1 Hardware Setup

To provide seamless data streaming and prevent data corruption, the target hardware and software environment must be configured according to the following procedures.

1. **Import Project:** Open **Code Composer Studio (CCS)** and import the project from the following SDK directory.

   ```
   ...\ti\mspm0_sdk_<SDK_Version>\examples\nortos\LP_MSPM0G5187\edgeAI\tida_010997_data_capture
   ```

2. **Build and Flash:** Compile the project and flash the resulting binary to the LP-MSPM0G5187 target board.

Copyright © 2026 Texas Instruments Incorporated

Note

It is critical that the serial communication parameters in the software match the firmware configuration to prevent data overflow or packet loss.

Before establishing a connection, verify the baud rate settings in both the firmware and the host application:

1. **Firmware Verification (SysConfig):** Open the *SysConfig* tool within the CCS project and verify the programmed Baud rate in the UART configuration module
2. **Edge AI Studio Setup:**
   a. Navigate to the *Options* tab
   b. Select *Select Port* to identify the active COM port assigned to the MCU
   c. Set the *Baud Rate* to match the value verified in the *SysConfig* tool

Once the parameters are synchronized, follow these steps to initiate data acquisition:

1. Navigate to the *Capture* tab
2. Select *Connect To Device*
3. Confirm the connection by verifying the *"Hardware connected"* status in the bottom-left status bar

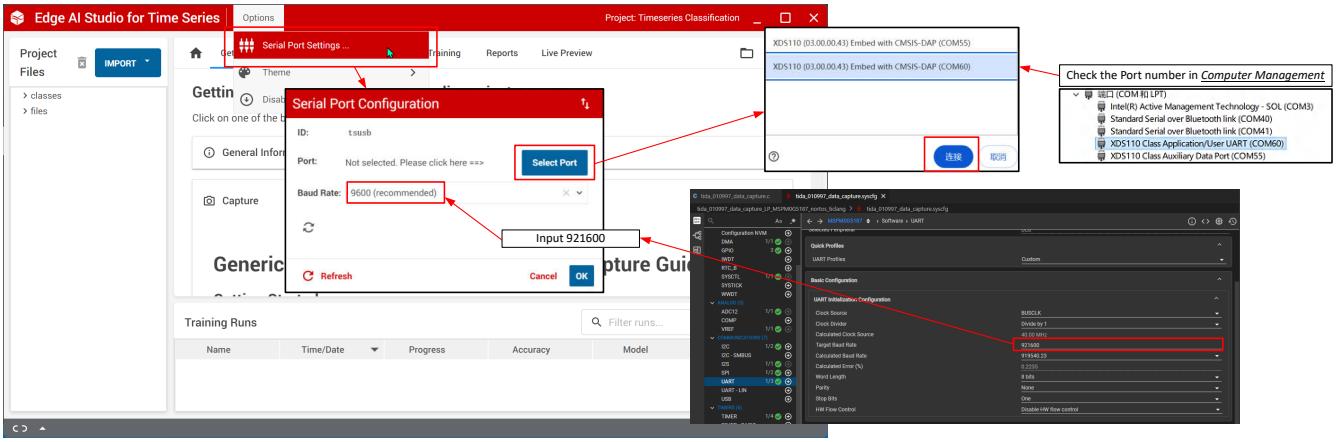


Figure 4-3. Connect Hardware to Edge AI Studio

4.2.2 Feature Overview

The **Data Capture** interface in Edge AI Studio provides a comprehensive suite of tools for configuring sensor hardware and managing data acquisition parameters. The platform supports a wide range of sensor modules. For instance, the TIDA-010997 evaluation module integrates multiple high-performance sensors, including:

- Motion
  - Dual-element PIR sensor with Fresnel lens (IRA-S210ST01 + IML-0685)
  - Low-power 6-axis accelerometer and gyroscope (BMI270)
  - High-precision linear 3D Hall-effect sensor (TMAG5170)
- Environment:
  - Digital humidity and temperature sensor (HDC3020)
  - High-speed, high-precision ambient light sensor (OPT4001)
  - Digital pressure sensor (BMI384)
- Audio:
  - Digital I2S output bottom-port microphone (ICS-43434)
  - Ultra-low noise, differential analog microphone (ICS-40740)

During the data capture process, users can configure the following parameters to provide organized and repeatable data collection:

- **Data Label:** Assigns a custom identifier to the dataset, which is utilized in the automated file-naming convention for efficient data management
- **Sample Count:** Defines the number of samples to be collected in a single capture session

Copyright © 2026 Texas Instruments Incorporated

- **Capture Interval:** Configures the number of continuous capture cycles and the time delay between each cycle, allowing for precise hardware synchronization and testing

Once the acquisition is complete, users can review the captured time-series data by navigating to: *Project Files → files*.

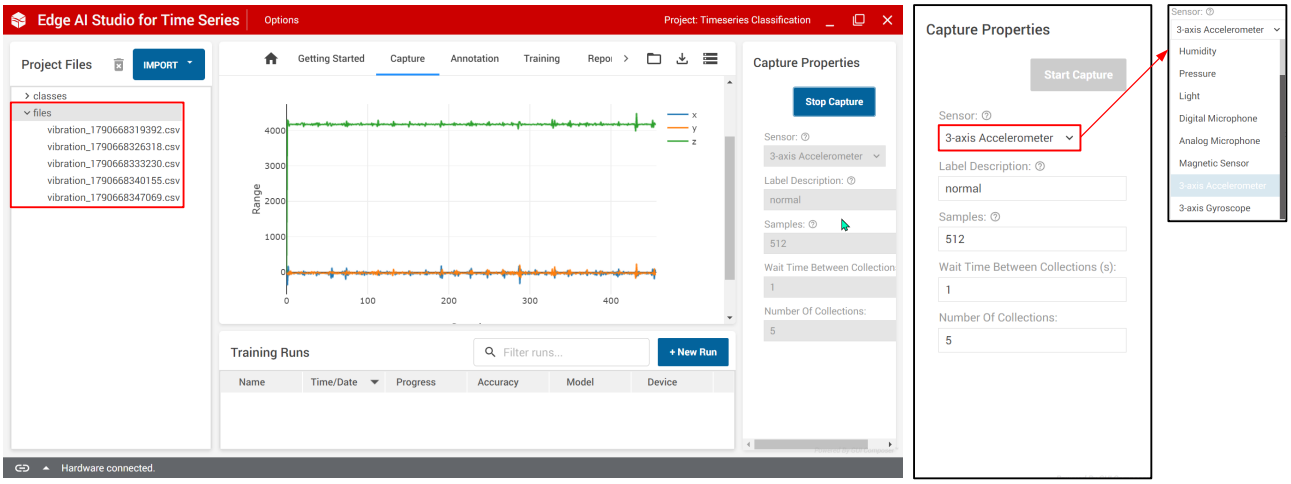


Figure 4-4. Capture Page Overview

4.3 Data Import

To facilitate direct model training, users can import pre-captured datasets from a local PC, as illustrated in Figure 4-5.

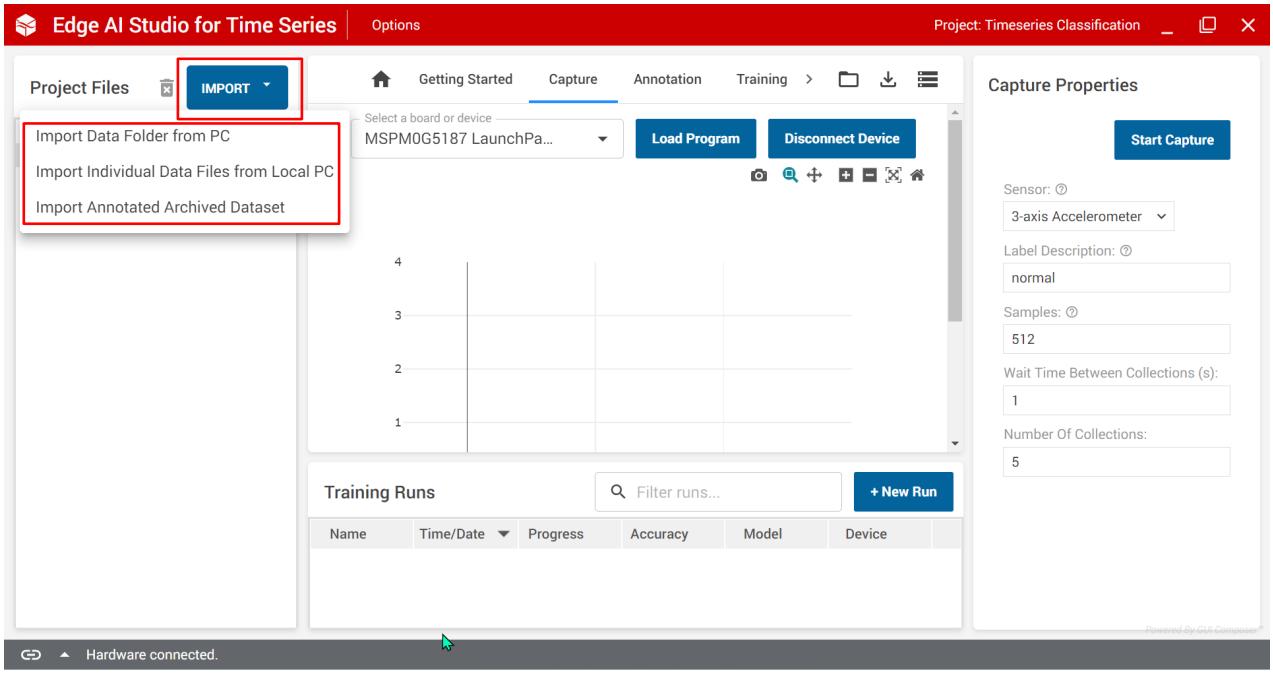


Figure 4-5. Import Data from Local PC

To provide successful data ingestion, uploaded files must comply with the specific formats required for each task complexity. For comprehensive guidance on data preparation, please refer to the TI Tiny ML Tensorlab User's Guide: Bring Your Own Data.

Copyright © 2026 Texas Instruments Incorporated

The following section details the supported data formats for various AI tasks.

4.3.1 Classification Task

Data Format

A. Headerless Format (Simple)

This format is practical for single-variable datasets. The headerless format consists of raw numeric values with one measurement per row.

Example:

```
0.523
0.612
0.498
0.701
...
```

B. Headered Format (Recommended for Multi-Variable)

This format is recommended for multi-channel sensor data. The first row must contain the column names.

Example:

```
channel_x,channel_y,channel_z
0.523,0.112,-0.234
0.612,0.098,-0.198
...
```

C. Time Column Handling

Edge AI Studio automatically detects and excludes time-series timestamps to prevent them from being treated as feature variables. Any column with a name containing the string **"time"** (case-insensitive) is automatically dropped during import.

Example:

```
Time,value1,value2
0.001,0.523,0.112
0.002,0.612,0.098
...
```

4.3.2 Regression Task

Data Format

Regression tasks involve predicting a continuous numerical value based on multiple input features. For regression tasks, the *target value (label) must be located in the final column* of the dataset. The file must follow a headered format where the first row defines the feature names and the target variable. Format structure is shown as below:

```
feature1,feature2,feature3,...,target
0.5,18.2,45.5,...,0.187
0.6,18.5,45.6,...,0.245 ...
```

Example: Motor Torque Prediction

- Columns 1-6: Input features
- Column 7 (last): Target value to predict

```
current_d,current_q,voltage_d,voltage_q,motor_speed,pm_temp,target_torque
-0.450,0.032,18.805,1.499,0.002,24.55,0.187 -0.325,0.045,18.818,1.542,0.003,24.54,0.245
-0.440,0.028,18.876,1.456,0.002,24.54,0.176
```


Copyright © 2026 Texas Instruments Incorporated

To prevent timestamps from being incorrectly processed as input features, Edge AI Studio automatically identifies and removes any column containing the Time.

Labeling Logic

Note

Simple Window transform is required for regression tasks.

Edge AI Studio applies a windowed averaging process to the target values. This process stabilizes the target label for time-series modeling:

- **Window Extraction:** A window of size N (defined by the *frame_size* parameter) is extracted from the raw target column
- **Mean Calculation:** The target values within that window are averaged
- **Label Assignment:** This calculated mean becomes the single target label for the entire window

Example Calculation (frame_size = 4):

```
Window 1: rows 0-3, targets [0.18, 0.24, 0.17, 0.19] → avg = 0.195
Window 2: rows 2-5, targets [0.17, 0.19, 0.22, 0.20] → avg = 0.195
```

4.3.3 Forecasting Task

Forecasting tasks involve predicting the future values of one or more specified variables based on historical observations.

In forecasting, all variables are organized in columns. Unlike regression, where the target is a dedicated column, forecasting utilizes the existing variables within the dataset. The user must explicitly specify which columns are to serve as input features and which are to serve as the variables to be forecasted through the configuration parameters.

Example: Temperature Forecasting

```
ambient,coolant,current,pm_temp
19.85,18.81,2.28,22.93
19.85,18.79,2.28,22.94
19.85,18.79,2.28,22.94
19.85,18.77,2.28,22.94
...
```

In this example, the dataset is structured such that the first three columns serve as input features, while the final column represents the target variable for forecasting. The column mapping is defined as follows:

- Column 0 (**ambient**): Input feature
- Column 1 (**coolant**): Input feature
- Column 2 (**current**): Input feature
- Column 3 (**pm_temp**): Target to forecast

While both Regression and Forecasting tasks utilize continuous numerical targets, they differ fundamentally in their temporal logic and target definitions, as illustrated in Table 4-2.

Table 4-2. Comparison of Regression and Forecasting Tasks

Feature	Regression Task	Forecasting Task
Primary Objective	Predict a continuous value based on current or recent windowed features.	Predict future values by identifying historical temporal patterns and trends.
Target Definition	Windowed Average: The target is defined as the mean value within the current observation window.	Future Value: The target is a specific value at a predefined future time step.
Data Requirement	Requires standard windowing transformations.	Requires explicit configuration of the prediction horizon.

Copyright © 2026 Texas Instruments Incorporated

4.3.4 Annotation List

To facilitate precise model training and evaluation, Edge AI Studio supports explicit dataset partitioning through dedicated annotation files. Users can define specific subsets for training, validation, and testing to provide robust model performance.

> For smaller datasets or rapid prototyping, users can manually assign labels and partitions directly within the Edge AI Studio graphical user interface (GUI).

Project Directory Structure

The required directory organization depends on the AI task type (Classification versus Regression/Forecasting). Refer to the structures below:

For Classification Tasks: The dataset must be organized into subdirectories representing each target class.

```
my_dataset/
├── classes/
│   ├── class_name_A/
│   │   └── sample_01.csv
│   └── class_name_B/
│       └── sample_01.csv
└── annotations/
    ├── file_list.txt              # All files (auto-generated if missing)
    ├── instances_train_list.txt   # Training subset
    ├── instances_val_list.txt     # Validation subset
    └── instances_test_list.txt    # Test subset (optional)
```

For Regression and Forecasting Tasks: Data is stored as individual files within a flat directory structure.

```
my_dataset/
├── files/
│   ├── datafile_01.csv
│   ├── datafile_02.csv
│   └── datafile_N.csv
└── annotations/
    ├── instances_train_list.txt
    ├── instances_val_list.txt
    └── instances_test_list.txt
```

Annotation File Format

Each annotation file (.txt) must contain a list of relative paths, with each path assigned to a unique line.

Example:

```
# instances_train_list.txt
class_A/sample_01.csv
class_A/sample_02.csv
class_B/sample_01.csv
```

4.4 Data Annotation

After successful data acquisition or import, the workflow proceeds to data annotation and dataset partitioning. These processes are performed through the Edge AI Studio GUI to prepare the raw data for model training and evaluation.

Copyright © 2026 Texas Instruments Incorporated

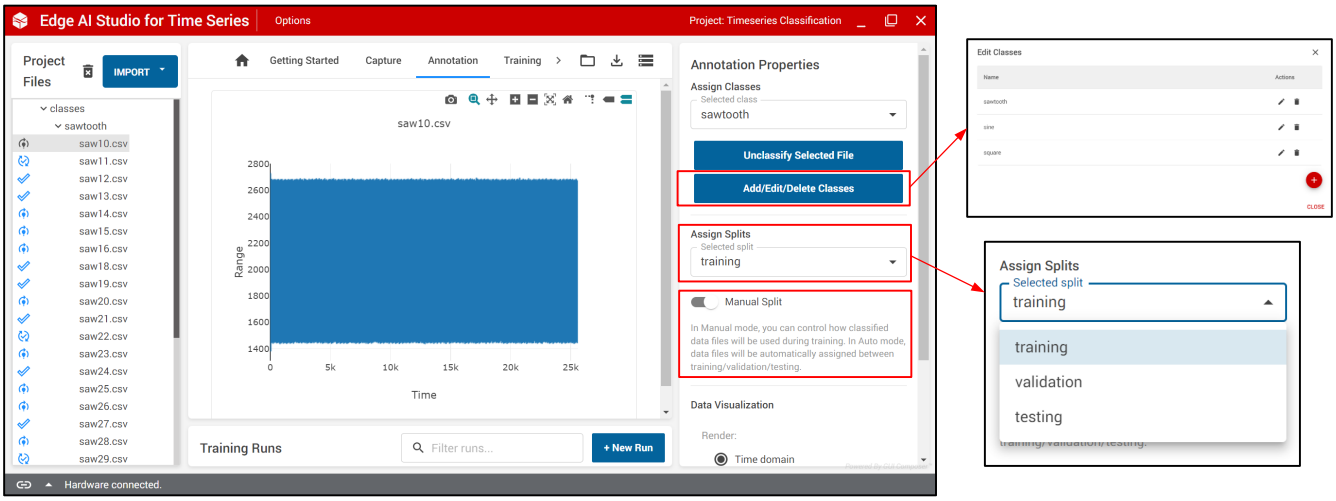


Figure 4-6. Annotation Page Overview

Annotation Workflow

Within the **Data Annotation** interface, users can perform the following operations to organize raw data into meaningful classes:

- **Class Management:** Users can create new class categories or rename existing ones to match the target application requirements.
- **Data Assignment:** Individual files can be selected and assigned to specific classes, or moved between existing categories to refine the dataset.

Dataset Partitioning

Once classes are assigned, the dataset must be partitioned into three distinct subsets: *Training*, *Validation*, and *Testing*. The requirements and purposes for each subset are summarized in Table 4-3.

Table 4-3. Dataset Requirements

Subset	Requirement	Purpose
Training dataset	Mandatory	Used by the model to learn features and optimize weights.
Validation dataset	Mandatory	Used during training to tune hyperparameters and monitor for overfitting.
Testing dataset	Optional	Used to evaluate the final model performance on unseen data.

Edge AI Studio provides a **Manual Split** mode, granting users full granular control over the distribution of samples. This mode is recommended for specialized applications where specific edge cases must be strictly reserved for testing.

The distribution of the test dataset must be carefully considered to provide statistical significance in model evaluation.

Note

To obtain the most reliable performance metrics, the test set must represent a substantial portion of the total dataset. For most applications, a test set exceeding 50% of the total data is recommended. As the overall dataset size increases, the proportion of the test set must also scale upward to make sure the evaluation results are statistically robust and representative of real-world operating conditions.

Copyright © 2026 Texas Instruments Incorporated

4.5 Model Training

The Edge AI Studio Model Training interface provides a streamlined workflow for configuring model architectures and training hyperparameters. This section outlines the configuration sequence and the underlying compilation process.

Copyright © 2026 Texas Instruments Incorporated

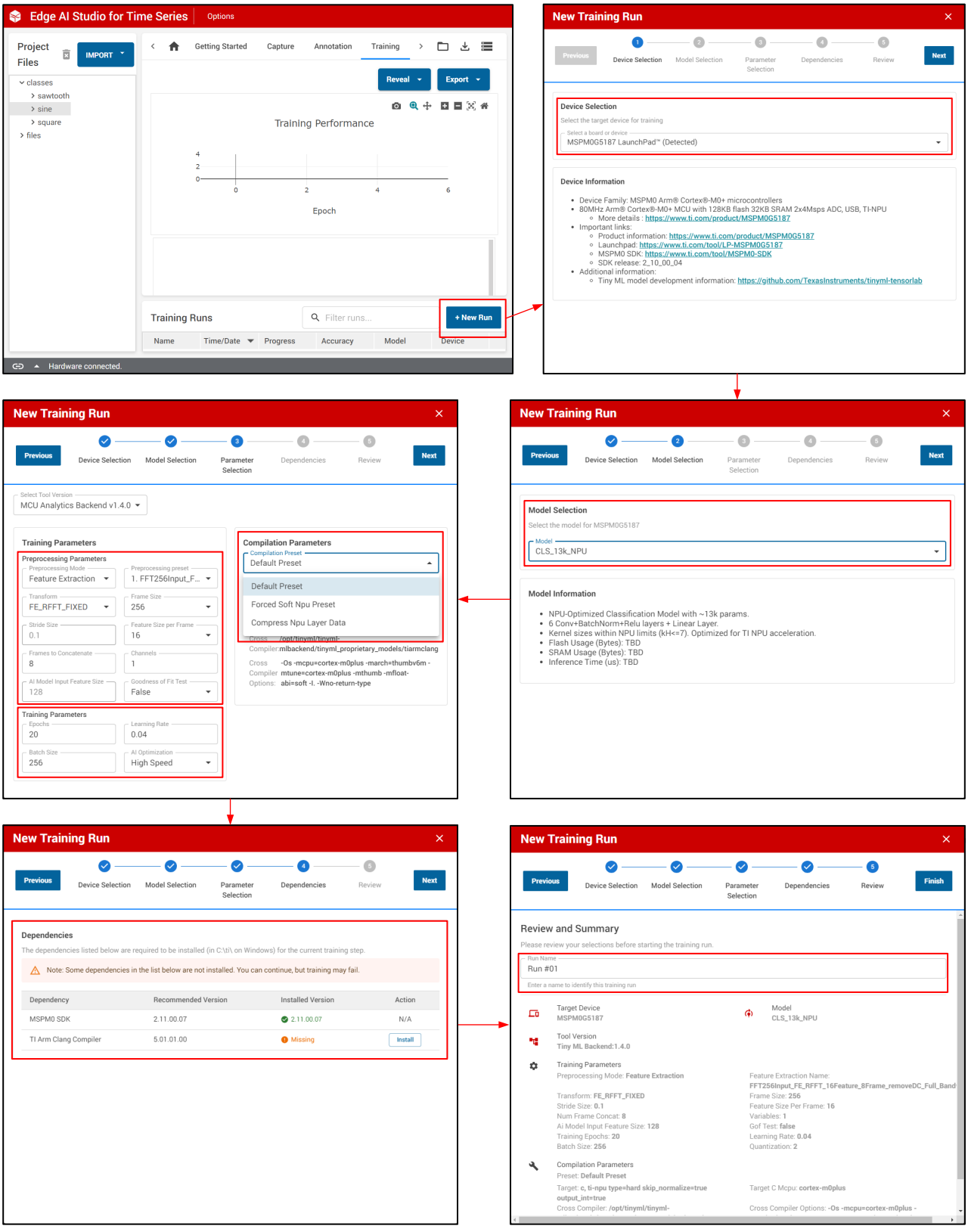


Figure 4-7. Training Page Overview

Copyright © 2026 Texas Instruments Incorporated

Model and Hardware Configuration

To initiate the training process, users must define the target hardware and the model architecture.

1. **Target Device Selection:** Users must select the target microcontroller (MCU) from the provided dropdown menu. Edge AI Studio supports a diverse ecosystem of over 20 MCU series. This cross-platform capability allows a single AI model architecture to be ported and deployed across different hardware platforms based on specific power and performance requirements.
2. **AI Model Selection:** For classification tasks, Edge AI Studio offers over 24 TI-optimized model architectures. When selecting a model, users must consider the trade-offs between computational complexity and predictive accuracy:
   a. **Model Complexity:** Larger models with more layers and parameters typically yield higher accuracy.
   b. **Resource Constraints:** Increased parameter counts directly correlate with higher memory (SRAM/ Flash) consumption and longer inference latency.

After selecting the model and defining training parameters (refer to Section 4.5.1 for detailed parameter descriptions), users must specify the execution target:

- **Default Preset (NPU):** Optimizes the model for hardware acceleration through the neural processing unit (NPU).
- **Forced Soft NPU Preset (CPU):** Routes the inference workload to the CPU, typically used for debugging or for devices without dedicated NPU hardware.

Training and Compilation Pipeline

The transition from a trained model to a target-ready binary follows a rigorous multi-stage pipeline. This process begins after the training phase is completed and the model artifact is generated.

1. **Training phase:** the system optimizes the model parameters based on the provided dataset. Upon successful completion, Edge AI Studio generates a high-level model file in ONNX format. This ONNX file serves as the standardized intermediate representation for the subsequent conversion stages
2. **ONNX to Source Code Conversion:** The generated ONNX model is then processed by the **TI NNC**. The NNC performs graph optimizations and operator mapping to transform the high-level ONNX representation into optimized source code
3. **Source Code to Binary Conversion:** In the final stage, the optimized source code is passed to the **Device Compiler** (for example, ARM CLANG). The compiler performs low-level machine code generation to produce the final binary library (.a file) tailored for the specific target MCU architecture

Note

If the Device Compiler is not installed on the PC system, the compilation pipeline terminates with a fatal error. The preceding model training phase completes normally, as training is independent of the compilation toolchain.

4.5.1 Training Parameter

The training parameter configuration is divided into two primary stages: preprocessing parameters and training hyperparameters.

Preprocessing Parameters

Edge AI Studio supports two preprocessing workflows to prepare data for model training:

- **Simple Window:** This method utilizes raw time-series data from the dataset using a sliding window mechanism. Users can configure the window size within a range of 0.01 to 1.0
- **Feature Extraction:** This workflow performs advanced data transformation. Within the Feature Extraction module, users can select between two processing modes:
  – **FE_RFFT_FIXED:** Uses FFT to extract frequency-domain information. Supported FFT sizes must be powers of 2, and the output size is defined as "frame size / 2". The FFT output is subsequently aggregated into a specific number of bins as defined by the "Feature Size per Frame" parameter
  – **Raw:** Utilizes raw time-series data without frequency transformation

Copyright © 2026 Texas Instruments Incorporated

The following parameters are used to configure the Feature Extraction workflow. These settings are applicable to both **FFT** and **Raw** processing modes:

Table 4-4. Feature Extraction Parameters

Parameter	Description
Frame Size	Specifies the number of input samples per processing frame.
Feature Size per Frame	Defines the number of bins (or features) generated per frame.
Frames to Concatenate	Specifies the number of bins from preceding frames to be retrieved from the buffer. This mechanism enables the model to capture temporal variations across the frequency spectrum.
Channels	Defines the number of sensor channels to be processed simultaneously.

For a comprehensive implementation example, please refer to Section 7.1 (Waveform Classifier Application).

Training Hyperparameters

The training hyperparameters control the optimization process and the final model characteristics:

- **Epochs:** Defines the total number of complete passes through the training dataset. Generally, increasing the number of epochs allows the model parameters to converge closer to the best design, though too many epochs can lead to overfitting.
- **Learning Rate:** Determines the step size during parameter updates following gradient descent in each epoch. A higher learning rate can accelerate convergence toward the global optimum but can cause oscillations or instability. Conversely, a lower learning rate provides more stable convergence but requires more epochs and can risk getting trapped in local optima.
- **Batch Size:** Specifies the number of training samples processed in a single iteration. Note that larger batch sizes increase the computational load and memory pressure on the host PC.
- **Optimization (Precision Mode):** Defines the target precision of the trained model:
  - **High Precision:** Trains a full-precision floating-point model.
  - **High Speed:** Utilizes Quantization-Aware Training (QAT) to produce a quantized fixed-point model optimized for high-speed inference.

4.5.2 Toolchain Dependency

The model compilation process in Edge AI Studio can fail even if the required device compiler (ARM Clang) and MSPM0 SDK are already installed on the host system. This issue is primarily observed in earlier versions of Edge AI Studio.

In certain versions, Edge AI Studio can fail to automatically detect the installation paths for the required tools, preventing the backend from correctly invoking the compiler during the compilation stage.

If the compilation fails due to tool detection issues, you can manually specify the tool installation path with the following steps:

1. Navigate to the following directory (replace [version] with your specific installation version):

```
...\ti\eaistudio[version]-timeseries\model-composer-extensions\tinyml-backend-[version]
\_internal\tinyml_modelmaker\ai_modules\timeseries
```

2. Open the file:

```
constants.py
```

3. Locate the following code block and modify the TOOLS_PATH variable:

```
TOOLS_PATH = os.path.abspath(os.getenv('TOOLS_PATH', os.path.join(f'{HOME_DIR}', 'bin')))
```

4. Change it to point directly to your TI installation directory, typically as below:

```
TOOLS_PATH = 'C:\\ti'
```


Copyright © 2026 Texas Instruments Incorporated

5. Verify the compiler version and make sure the MSPM0_CGT_VERSION matches your installed version. For example:

```
MSPM0_CGT_VERSION = 'ti-cgt-armllvm_5.1.1.LTS'
```

4.6 Performance Reports

Upon completion of the training process, users can access the *Reports* interface to evaluate the predictive performance of the model. The *Reports* interface provides three distinct analytical dimensions: confusion matrix, multi-class ROC curves, and pairwise class prediction difference, enabling a comprehensive assessment of accuracy, robustness, and decision margins, shown in Figure 4-8.

Copyright © 2026 Texas Instruments Incorporated

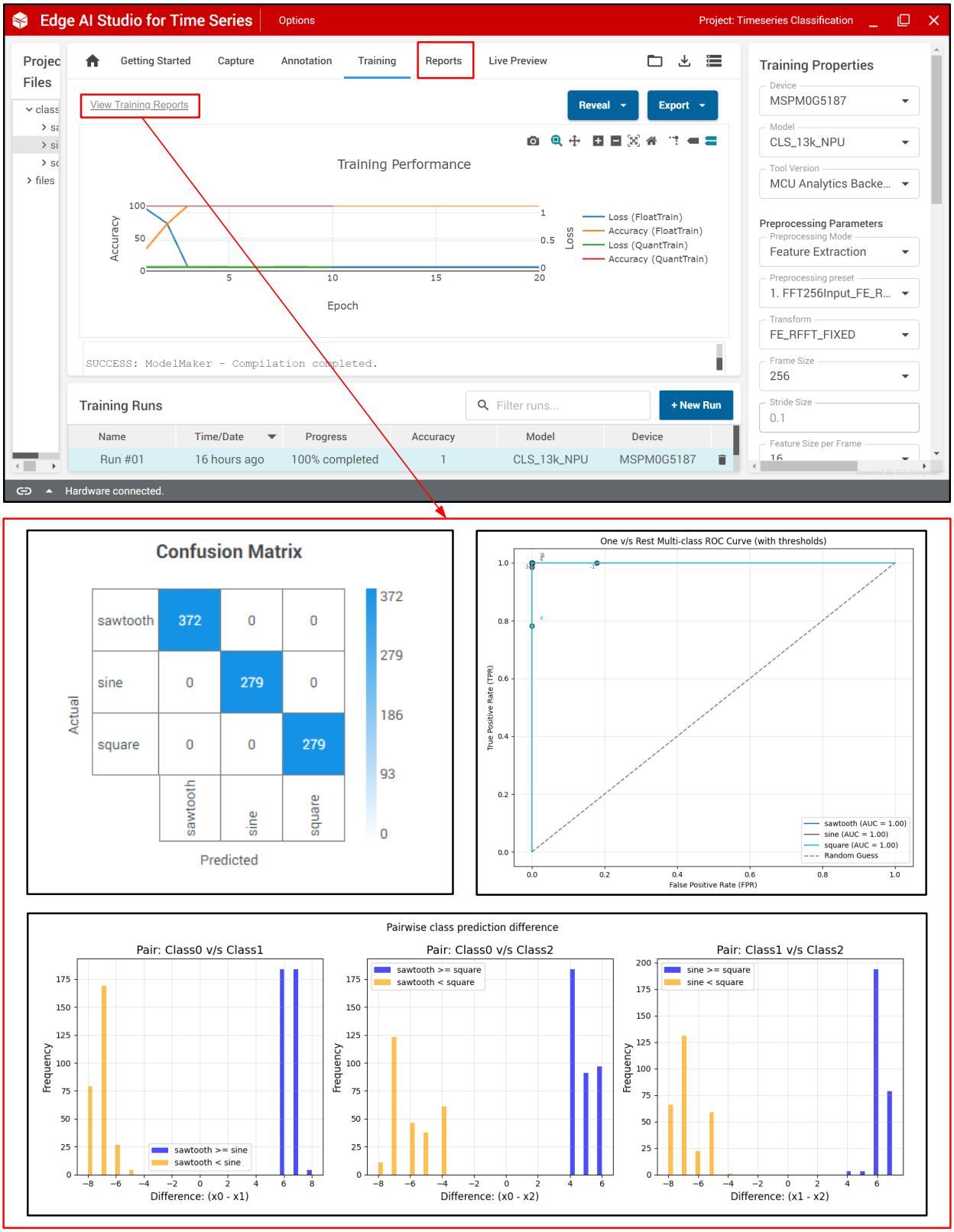


Figure 4-8. Reports Page Overview

Copyright © 2026 Texas Instruments Incorporated

Confusion Matrix

The confusion matrix is a fundamental tool for visualizing the performance of a supervised learning algorithm at a specific decision threshold. The confusion matrix provides a granular view of how the model distributes predictions across the actual dataset.

The matrix is organized into a grid where:

- **Rows (Y-axis):** Represent the **Actual/True Class** labels from the test dataset.
- **Columns (X-axis):** Represent the **Predicted Class** labels generated by the model.
- **Diagonal Elements:** Represent **Correct Predictions** (where the predicted label matches the actual label).
- **Off-diagonal Elements:** Represent **Misclassifications** (errors).

The confusion matrix serves as a multi-faceted diagnostic tool for evaluating model reliability. Primarily, the concentration of values along the main diagonal provides a direct visual representation of the model's overall accuracy, as well as the precision and recall for individual classes; a high degree of diagonal intensity indicates that the model is successfully capturing the underlying patterns of each class.

Beyond overall accuracy, the matrix allows for sophisticated error pattern diagnostics through the analysis of off-diagonal elements. By observing where misclassifications aggregate, engineers can identify specific instances of feature confusion - for instance, if Class A is frequently misidentified as Class B, the matrix indicates that the current feature set lacks the discriminative resolution required to separate these two specific categories. Such insights provide a targeted direction for optimizing upstream feature engineering or data preprocessing.

Furthermore, the matrix facilitates a critical verification of dataset integrity. By calculating the sum of elements across individual rows or columns, users can verify the sample distribution within the test set. This step is essential to make sure that the performance metrics are not skewed by class imbalance, thereby maintaining that the reported accuracy is an objective reflection of the true generalization capability of the model.

Multi-Class ROC Curve (One-vs-Rest)

The receiver operating characteristic (ROC) curve evaluates the model's ability to discriminate between classes by plotting the true positive rate (TPR) against the false positive rate (FPR). For multi-class tasks, Edge AI Studio employs the one-vs-rest (OvR) strategy. In this approach, each individual class is treated as the "positive" class, while all other classes are aggregated into a single "negative" class. This allows for the generation of a distinct ROC curve for every class in the model.

The Multi-class ROC curve provides a comprehensive view of the discriminative capability of the model across varying operating conditions. The primary metric for this assessment is the area under the curve (AUC); an AUC value approaching 1.0 for a given class signifies that its corresponding curve is tightly pushed toward the top-left corner, representing near-perfect separation between the target class and the rest of the dataset. Conversely, an AUC near 0.5 indicates that the model lacks predictive utility and performs no better than random chance.

Beyond a single scalar value, the ROC curve is an essential tool for evaluating threshold robustness. Because the curve tracks performance across a spectrum of varying classification confidence thresholds, the stability of the curve - specifically its ability to maintain a high true positive rate (TPR) while minimizing the false positive rate (FPR) across a wide range of points - is critical. The integration of explicit threshold annotations directly onto the curve allows engineers to visually pinpoint how specific scores map to true and false positive rates.

For engineers deploying models in real-time embedded systems, a stable ROC curve suggests that the model remains reliable even when the classification threshold is dynamically adjusted to meet specific application requirements, such as sensitivity or false-alarm tolerances.

Pairwise Class Prediction Difference

The pairwise class prediction difference histogram analyzes the decision margin - the confidence with which the model distinguishes between any two specific classes. This histogram plots the distribution of the difference in raw decision scores (or predicted probabilities) between two classes. The histogram captures the statistical behavior of the output distribution of the model, rather than just the final hard label.

Copyright © 2026 Texas Instruments Incorporated

While traditional metrics focus on binary correctness, the pairwise class prediction difference histogram offers a deeper analysis of the decision margin of the model and probabilistic certainty. By examining the distribution of score differences between any two classes, engineers can characterize the statistical "gap" that defines the model's classification boundaries.

A high-performance model is characterized by a distinct bimodal distribution, where the two peaks are widely separated. Such a structure indicates that the model possesses a wide decision margin, providing high confidence in its outputs. Most importantly, the density of the distribution near the zero-point - the **"Ambiguity Zone"** - serves as a critical indicator of noise immunity.

A histogram with minimal or "vacuum" density around the zero-value suggests that the model is highly resilient to minor signal perturbations. In practical deployment, this translates to excellent anti-interference capabilities, as the model is statistically less likely to trigger an erroneous state transition due to transient noise or environmental fluctuations.

4.7 Model Export

Once the training phase is completed, the trained model can be exported for integration into customized development workflows. For edge-side deployment, Edge AI Studio also supports exporting compiled models. Note that as of version 1.7.1, the supported export format is limited to binary files, which are optimized for edge devices.

The model export workflow within Edge AI Studio is illustrated in Figure 4-9.

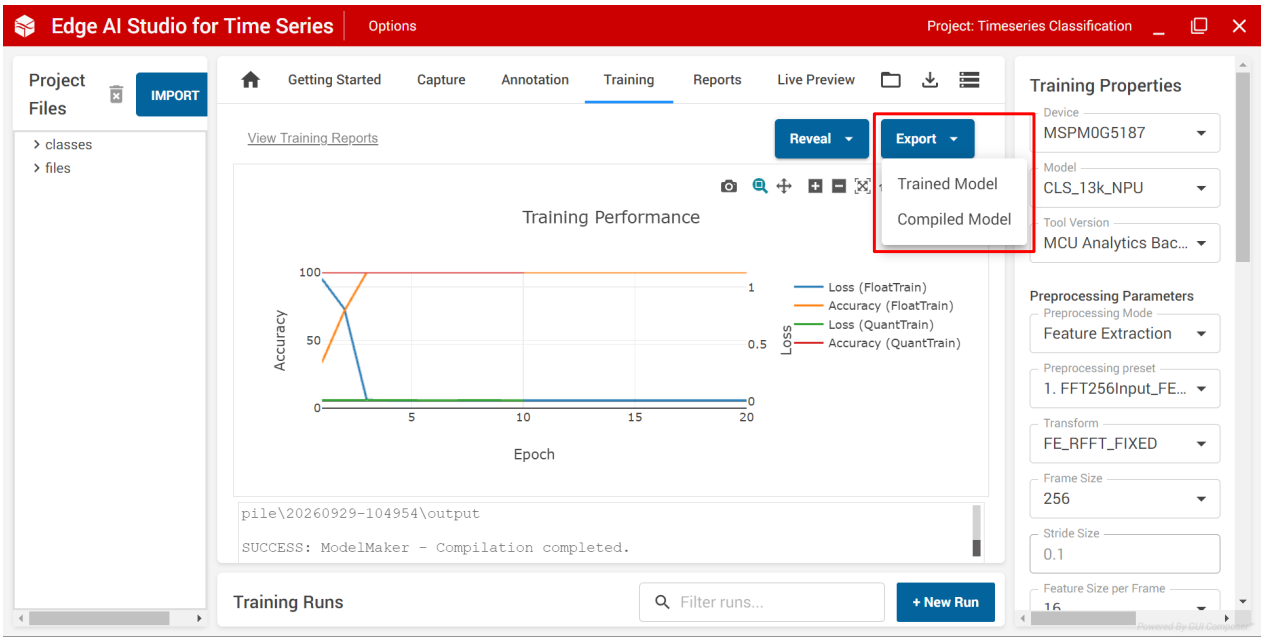


Figure 4-9. Export the Model from Edge AI Studio

Upon successful exportation, the system generates two distinct types of artifacts: **Trained** NN Model Files and **Compiled** NN Model Files. The detailed directory structure and the specific contents of these exported files are provided in Figure 4-10 and Figure 4-11, respectively.

Copyright © 2026 Texas Instruments Incorporated

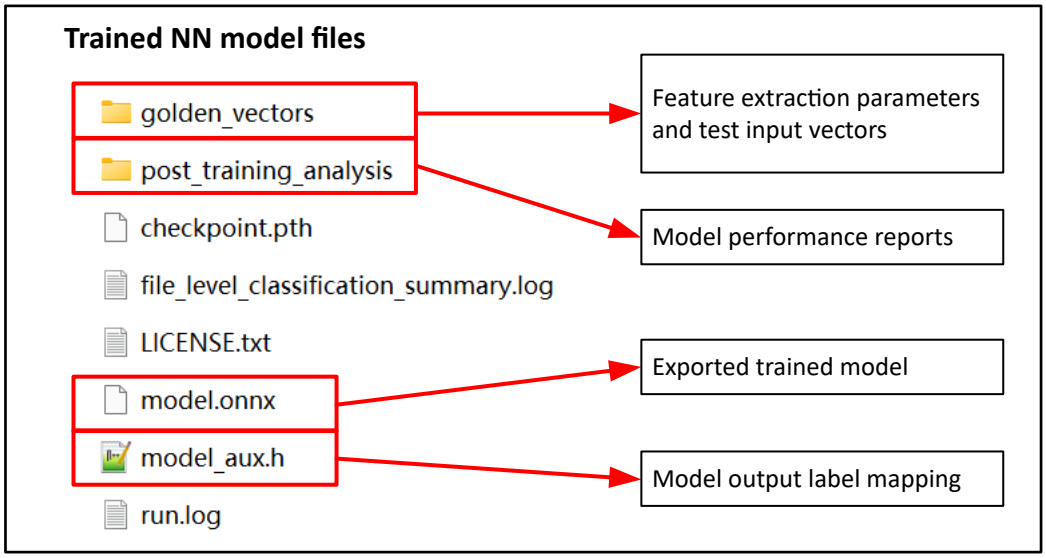


Figure 4-10. Trained NN Model Files

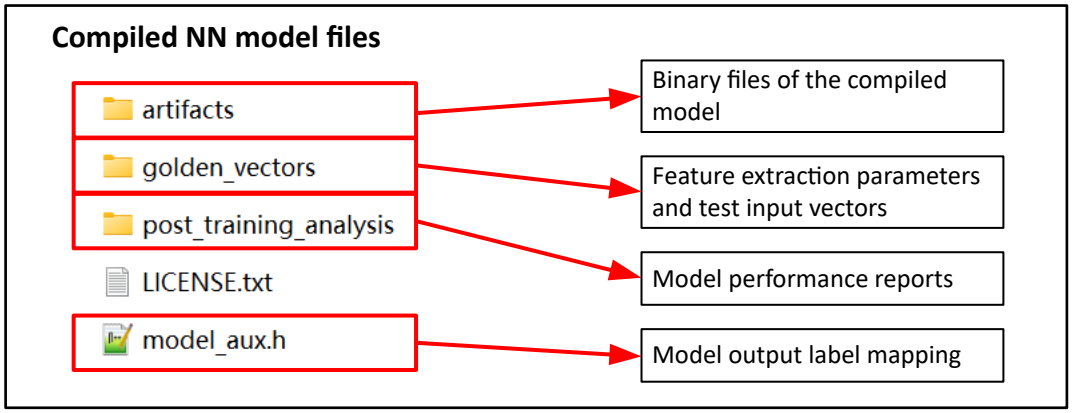


Figure 4-11. Compiled NN Model Files

Copyright © 2026 Texas Instruments Incorporated

5 Advanced Development Through TI Tiny ML Tensorlab

This section introduces Tensorlab, a workflow-oriented toolset optimized for PyTorch-based model training. By leveraging core PyTorch modules and classes, Tensorlab facilitates the end-to-end design and training of neural networks.

Unlike high-level abstraction tools, Tensorlab is engineered for professional developers requiring granular, end-to-end control over the machine learning pipeline. Tensorlab provides the necessary flexibility for highly customized development, enabling precise manipulation of data acquisition, model architecture, training hyperparameters, and the compilation workflow.

The following implementation utilizes the MNIST (Digit Recognition) dataset as a foundational benchmark to demonstrate the construction of a neural network from first principles. The technical progression begins with baseline implementations and iteratively incorporates advanced features from the PyTorch ecosystem. The workflow encompasses the complete deployment lifecycle: model exportation via ONNX, compilation using TVM (TI NNC), and PC-side inference. The section concludes with the implementation of Quantization-Aware Training (QAT), utilizing TI-specific NPU quantization techniques to optimize model performance for edge deployment.

Additional Resources

- Edge AI User's Guide (MSPM0-SDK)
- Tiny ML Tensorlab User's Guide
- PyTorch Webpage: What is torch.nn really?
- Tiny ML Workflow Example Python Script Package

5.1 Environment Setup

Installation

To install Tiny ML Tensorlab on your system, please refer to the Tiny ML Tensorlab User's Guide: Installation. Select an installation path that aligns with your development environment, noting that the workflows described herein are validated for the Developer Installation.

Virtual Environment (venv) Configuration

To provide environment isolation and prevent dependency conflicts, all Python scripts must be executed within the Tensorlab virtual environment. All operations must be initiated from the root installation directory:

```
C:\ti\tinyml-tensorlab
```

Depending on your preferred terminal interface, use the following commands to activate the environment:

A. PowerShell

```
cd C:\ti\tinyml-tensorlab
\venv\Scripts\Activate.ps1
```

B. Git Bash

```
cd C:\ti\tinyml-tensorlab
source ./venv/Scripts/activate
```

VS Code Configuration

To integrate the virtual environment with Visual Studio Code for development and debugging, follow these steps:

1. Open the Command Palette by pressing

   ```
   Ctrl + Shift + P
   ```

2. Search for and select:

   ```
   Python: Select Interpreter
   ```

3. From the resulting list, select the interpreter associated with the 'venv' directory.

Copyright © 2026 Texas Instruments Incorporated

Verifying Dependencies

Once the environment is activated, you can verify the installed packages and make sure all required dependencies are present by executing the following command:

```
pip list
```

5.2 Data Acquisition

To provide a modular and verifiable development workflow, the data acquisition process is divided into two distinct stages: data retrieval and data validation. This modular approach allows for isolated testing of network connectivity, file system integrity, and PyTorch tensor compatibility.

5.2.1 Data Retrieval

The first stage involves executing a script to download the MNIST dataset and organize the dataset within the local project structure. To prevent path-related errors during execution, the script utilizes absolute path resolution based on the current file location. This makes sure that the

```
data/
```

directory is always created and accessed relative to the script's directory, regardless of the current working directory of the terminal.

1. Create a new file named

   ```
   01_test_download_data.py
   ```

   in your project root directory.

2. Implement the following script

   ```
   from pathlib import Path
   import requests

   # 1. Define the absolute base directory and target path
   BASE_DIR = Path(__file__).resolve().parent
   PATH = BASE_DIR / "data" / "mnist"

   # 2. Ensure the target directory hierarchy exists
   PATH.mkdir(parents=True, exist_ok=True)

   # 3. Define remote source and local filename
   URL = "https://github.com/pytorch/tutorials/raw/main/_static/"
   FILENAME = "mnist.pkl.gz"

   # 4. Execute conditional download
   if not (PATH / FILENAME).exists():
       print("Downloading MNIST dataset. Please wait...")
       content = requests.get(URL + FILENAME).content
       (PATH / FILENAME).open("wb").write(content)
       print("Download complete.")
   else:
       print("Local dataset detected. Skipping download.")
   ```

3. Run the script using the terminal. Upon successful completion, the terminal displays the following log:

   ```
   Downloading MNIST dataset. Please wait...
   Download complete.
   ```

5.2.2 Data Validation

The second stage verifies the integrity of the downloaded file, visualizes the raw data, and converts the NumPy arrays into PyTorch tensors.

1. Create a new file named

   ```
   02_test_load_visual_data.py
   ```


Copyright © 2026 Texas Instruments Incorporated

2. Implement the following validation script

```
import pickle
import gzip
from pathlib import Path
from matplotlib import pyplot
import numpy as np
import torch

# 1. Synchronize path resolution with Stage 1
BASE_DIR = Path(__file__).resolve().parent
PATH = BASE_DIR / "data" / "mnist"
FILENAME = "mnist.pkl.gz"

print("--- 1. Loading and Decompressing Dataset ---")
# Validate file existence before proceeding
if not (PATH / FILENAME).exists():
    raise FileNotFoundError(f"Dataset not found at: {PATH / FILENAME}. "
                            "Please ensure Stage 1 was completed successfully.")

with gzip.open((PATH / FILENAME).as_posix(), "rb") as f:
    ((x_train, y_train), (x_valid, y_valid), _) = pickle.load(f, encoding="latin-1")

print(f"Training set shape (NumPy): {x_train.shape}")

print("\n--- 2. Visualizing Sample Data ---")
# Reshape to 28x28 for image rendering
pyplot.imshow(x_train[0].reshape((28, 28)), cmap="gray")
print("Note: Close the graphical window to proceed with tensor conversion.")
pyplot.show()

print("\n--- 3. Converting NumPy Arrays to PyTorch Tensors ---")
x_train, y_train, x_valid, y_valid = map(
    torch.tensor, (x_train, y_train, x_valid, y_valid)
)

print("\n--- 4. Verifying PyTorch Tensor Integrity ---")
print(f"Training Data Sample:\n{x_train}\n{y_train}")
print(f"Converted Tensor Shape: {x_train.shape}")
print(f"Label Range: Min={y_train.min()}, Max={y_train.max()}")
print("Success: Data pipeline and PyTorch integration verified.")
```

3. Run the script. A window displays the first handwritten digit from the MNIST set, as shown in Figure 5-1. After closing the image window, the terminal displays the converted tensor shapes and the success message. Successful execution confirms that the dataset is correctly loaded, the environment is properly configured, and the PyTorch dependency is functional

Copyright © 2026 Texas Instruments Incorporated

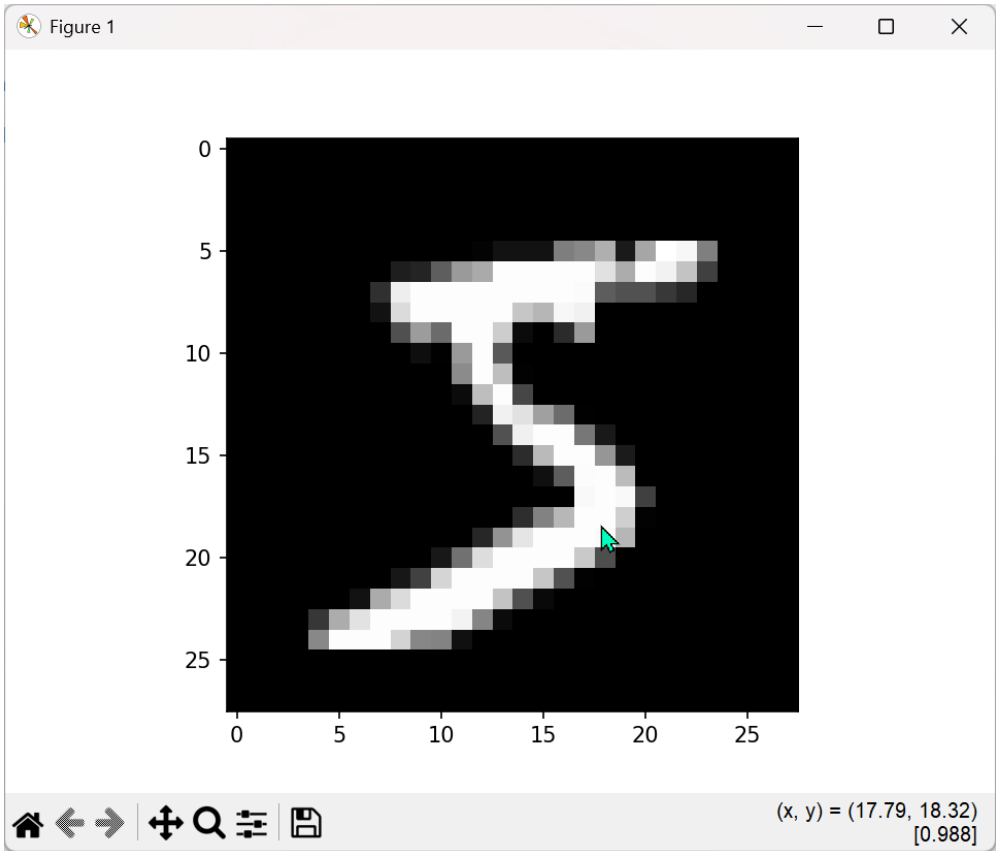


Figure 5-1. Handwritten Digit Image

Note

MNIST dataset has been scaled to a float range of [0, 1].

5.3 Model Architecture Iterative Refactoring

As the development pipeline matures, the model architecture must evolve from basic mathematical abstractions to sophisticated, high-performance structures capable of capturing complex data patterns. This section demonstrates the transition from a linear classifier to a convolutional neural network (CNN), highlighting the power of modularity in the PyTorch ecosystem.

5.3.1 Baseline Implementation

A minimal linear classifier is implemented from first principles to establish a baseline. The construction utilizes fundamental PyTorch tensor operations instead of high-level abstractions like *torch.nn*, providing transparent insight into the mathematical mechanics of neural networks.

The implementation leverages two core PyTorch functionalities:

1. **Tensor Operations:** Manual management of weights and biases as tensors, with scaled initialization applied for numerical stability.
2. **Autograd Engine:** The activation of

   ```
   requires_grad=True
   ```

   , enables PyTorch to track all tensor operations, building a dynamic computational graph for automatic gradient computation during back-propagation.

Copyright © 2026 Texas Instruments Incorporated

The definition of the forward pass, activation function (log_softmax), and loss function (NLL) via standard Python logic demonstrates the ability of PyTorch's vectorized backend to fully optimize custom-coded architectures.

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "03_test_train_from_scratch.py" script. After executing the script using the terminal, verify the training progress by monitoring the logs:

```
Data loaded successfully. Training samples: 50000, Features per sample: 784
--- Pre-training Model State ---
...
Initial Loss: 2.2634
Initial Accuracy: 0.1406 (Expected near random)
------------------------------

--- Commencing Model Training ---
...
Epoch 10/10 completed.

--- Post-training Model Performance ---
Final Loss: 0.0559 (Target: Significant decrease)
Final Accuracy: 1.0000 (Target: Significant increase)
```

Note

The observed 100% accuracy is a result of evaluating the model on the training set without a separate validation set, indicating potential **overfitting**.

5.3.2 Refactoring With PyTorch Built-in Modules

This session transitions from manual tensor manipulation to a standardized workflow by leveraging PyTorch's high-level abstractions: nn.Module and torch.optim. This refactoring shifts the development paradigm from low-level mathematical implementation to an object-oriented programming (OOP) approach.

nn.Module

To improve modularity, the model is encapsulated within a class inheriting from nn.Module. This provides several critical advantages:

- **Parameter Management:** Instead of manually defining and initializing *self.weights* and *self.bias*, the *nn.Linear* module automatically handles parameter initialization and registration.
- **Automatic Tracking:** By inheriting from nn.Module, all internal parameters are automatically registered, allowing the system to track them for gradient computation without manual intervention.

torch.optim

The manual parameter update step - which requires explicit **torch.no_grad()** contexts and per-parameter gradient zeroing - is replaced by the **torch.optim** framework

- **model.parameters()**: This method acts as a bridge, passing all registered model parameters to the optimizer. The optimizer maintains a registry of these parameter pointers for efficient updates.
- **opt.step()**: This method abstracts the update logic. This method iterates through all registered parameters and applies the chosen optimization algorithm (for example, SGD) to update the weights based on the current gradients.
- **opt.zero_grad()**: This provides a centralized mechanism to clear gradients across all optimized parameters simultaneously, maintaining the integrity of the next training iteration.

Using an optimizer allows for seamless transitions between different optimization algorithms (for example, moving from SGD to Adam) by modifying a single line of code, without altering the underlying model architecture.

Copyright © 2026 Texas Instruments Incorporated

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "04_test_pytorch_module_refactor.py" script. After executing the script using the terminal, verify the training progress by monitoring the logs:

```
--- Pre-training Model State (Optimized Version) ---
Initial Loss: 2.3087
------------------------------

--- Commencing Training with torch.optim ---
...
Epoch 10/10 completed.

--- Post-training Performance ---
Final Training Loss: 0.0554
Validation Accuracy: 0.9245
```

5.3.3 Data Pipeline Optimization

While previous sections relied on manual index calculation and explicit slicing of training and label tensors, a more robust engineering workflow is achieved by implementing the Dataset and DataLoader abstractions. These abstractions effectively decouple data storage from batch processing, enhancing both scalability and efficiency.

TensorDataset

Managing features and labels as separate entities requires careful synchronization to ensure indices remain aligned during shuffling or slicing. TensorDataset solves this by "zipping" multiple tensors into a single dataset object. Once encapsulated, accessing an index automatically returns a tuple containing the corresponding feature tensor and label, maintaining structural integrity across the pipeline.

DataLoader

While Dataset handles storage, DataLoader manages the logistics of data delivery. DataLoader provides several essential engineering capabilities:

- **Data Shuffling (shuffle = True)**: To prevent the model from learning the sequence of data (which leads to overfitting), the DataLoader reshuffles the dataset at the start of every epoch, maintaining stochasticity in the mini-batch selection.
- **Parallelized Loading (num_workers)**: In large-scale applications where I/O (disk reading) becomes a bottleneck, **DataLoader** can utilize multi-process data loading. This allows the CPU to pre-fetch and prepare the next batch in the background while the GPU processes the current one, maximizing hardware utilization.

By utilizing these tools, the training loop is simplified from complex indexing arithmetic to a clean, iterable interface.

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "05_test_data_pipeline_refactor.py" script. After executing the script using the terminal, verify the training progress by monitoring the logs:

```
--- Commencing Training ---
Epoch 0: Val Loss = 0.4051 | Val Accuracy = 87.68%
Epoch 1: Val Loss = 0.3202 | Val Accuracy = 90.73%
Epoch 2: Val Loss = 0.2923 | Val Accuracy = 91.63%
Epoch 3: Val Loss = 0.2960 | Val Accuracy = 91.53%
Epoch 4: Val Loss = 0.2821 | Val Accuracy = 92.19%
Epoch 5: Val Loss = 0.3038 | Val Accuracy = 91.28%
Epoch 6: Val Loss = 0.3009 | Val Accuracy = 91.40%
Epoch 7: Val Loss = 0.3131 | Val Accuracy = 91.14%
Epoch 8: Val Loss = 0.2663 | Val Accuracy = 92.78%
Epoch 9: Val Loss = 0.3008 | Val Accuracy = 91.59%

--- Final Test Set Evaluation ---
>> Test Loss: 0.3143
>> Test Accuracy: 91.03%
```


Copyright © 2026 Texas Instruments Incorporated

5.3.4 Transition to CNN Architecture

With a robust, modular training infrastructure in place, the pipeline can now be upgraded to support advanced computer vision (CV) tasks by replacing the linear architecture with a convolutional neural network (CNN).

The primary limitation of the linear model was the inability to preserve spatial hierarchies; flattening a 28x28 image into a 784-element vector destroys the local geometric relationships between pixels. The CNN architecture resolves this through two mechanisms:

- **Tensor Reshaping:** Using *xb.view(-1, 1, 28, 28)*, the input is reshaped into a 4D tensor (Batch, Channel, Height, Width). This restores the spatial grid required for convolutional operations.
- **Hierarchical Feature Learning:** The model utilizes a sequence of **nn.Conv2d** layers with increasing channel depths. This allows the network to extract a hierarchy of features: from low-level edges in the first layer to complex, high-level digit contours in the final convolutional layer.

To improve convergence speed and stability, the SGD optimizer is upgraded to include a momentum = 0.9 term. In standard SGD, updates can stall or oscillate heavily in flat regions of the loss landscape. Momentum introduces an accumulation of past gradients, providing a physical "inertia" that helps the optimizer traverse plateaus and navigate narrow valleys more efficiently.

Because the training logic (loss_batch and fit) is thoroughly decoupled from the network architecture, this transition requires zero modifications to the core training loop.

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "06_test_cnn_model_refactor.py" script. After executing the script using the terminal, verify the training progress by monitoring the logs:

```
--- Commencing CNN Training ---
Epoch 0: Val Loss = 0.3287 | Val Accuracy = 0.9031
Epoch 1: Val Loss = 0.2477 | Val Accuracy = 0.9233
Epoch 2: Val Loss = 0.2205 | Val Accuracy = 0.9352
Epoch 3: Val Loss = 0.2174 | Val Accuracy = 0.9335
Epoch 4: Val Loss = 0.1661 | Val Accuracy = 0.9522
Epoch 5: Val Loss = 0.1702 | Val Accuracy = 0.9488
Epoch 6: Val Loss = 0.1590 | Val Accuracy = 0.9537
Epoch 7: Val Loss = 0.1423 | Val Accuracy = 0.9606
Epoch 8: Val Loss = 0.1265 | Val Accuracy = 0.9622
Epoch 9: Val Loss = 0.1303 | Val Accuracy = 0.9611

--- Final Test Set Evaluation ---
>> Test Loss: 0.1389 | Test Accuracy: 0.9568 <<
```

Streamlining Architecture with nn.Sequential

While the previously implemented CNN architecture utilizes nn.Module to define custom layers, the *forward* method still requires explicit, manual definition of the data flow. This approach, while highly flexible, necessitates a manual orchestration of every tensor transformation within the class structure.

To streamline this process, PyTorch provides the nn.Sequential container. This utility allows developers to encapsulate multiple layers into a single modular unit, where data flows through the layers in the exact order they are defined. This "pipeline" approach significantly reduces boilerplate code and eliminates the need to manually implement the **forward** method for standard feed-forward architectures.

A common challenge when using nn.Sequential is the requirement to perform intermediate tensor operations that do not exist as standard native layers, such as executing a custom slicing operation or adapting specific shapes before subsequent processing. In a standard nn.Module, this is handled easily within the *forward* function. However, within a strict nn.Sequential pipeline, every single step must be encapsulated as a valid nn.Module object. To achieve this without writing entirely new standalone classes for every distinct transformation, a Lambda wrapper can be utilized.

Copyright © 2026 Texas Instruments Incorporated

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "07_test_cnn_sequential_refactor.py" script. After executing the script using the terminal, verify the training progress by monitoring the logs:

```
--- Commencing Sequential CNN Training ---
Epoch 0: Val Loss = 0.3077 | Val Accuracy = 0.9077
Epoch 1: Val Loss = 0.2371 | Val Accuracy = 0.9303
Epoch 2: Val Loss = 0.1858 | Val Accuracy = 0.9456
Epoch 3: Val Loss = 0.1682 | Val Accuracy = 0.9503
Epoch 4: Val Loss = 0.1618 | Val Accuracy = 0.9546

--- Final Test Set Evaluation ---
>> Test Loss: 0.1694 | Test Accuracy: 0.9501 <<
```

5.4 Model Export

Once a model has reached the desired accuracy during the training phase, it can be transitioned from the PyTorch development environment into a deployment-ready format - converting a model to the open neural network exchange (ONNX) format.

ONNX serves as an interoperable intermediate representation, enabling evaluation inference across various platforms - including C++, Android, iOS, and web browsers - using runtimes such as *ONNX Runtime*, without the overhead of a full PyTorch installation.

The following script demonstrates how to extend your existing training workflow to include a robust model exportation step. This implementation includes directory management and specific configurations to make sure the exported model is compatible with downstream inference engines.

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "08_test_export_onnx.py" script. After executing the script using the terminal, verify the training progress by monitoring the logs:

```
--- Commencing Sequential CNN Training ---
Epoch 0: Val Loss = 0.3461 | Val Accuracy = 0.8950
Epoch 1: Val Loss = 0.3137 | Val Accuracy = 0.9107
Epoch 2: Val Loss = 0.2044 | Val Accuracy = 0.9445
Epoch 3: Val Loss = 0.1736 | Val Accuracy = 0.9490
Epoch 4: Val Loss = 0.1684 | Val Accuracy = 0.9502

--- Final Test Set Evaluation ---
>> Test Loss: 0.1698 | Test Accuracy: 0.9475 <<

--- Commencing ONNX Model Exportation ---
ONNX model successfully exported to: ...\model\mnist\mnist_model.onnx
```

5.5 Quantization-Aware Training

Quantization is a critical optimization technique for deploying machine learning models on resource-constrained edge devices. Quantization involves reducing the precision of model weights and activations from 32-bit floating-point (FP32) to lower bit-widths, such as 8-bit (INT8), 4-bit (INT4), or 2-bit (INT2) integers.

This process yields three primary benefits for Edge AI:

- **Model Compression:** Significant reduction in memory footprint (for example, a 4× reduction when moving from FP32 to INT8).
- **Enhanced Inference Speed:** Integer arithmetic is computationally more efficient than floating-point operations on specialized hardware.
- **Hardware Acceleration:** TI's neural processing unit (NPU) is architecturally optimized for integer-based computations, making quantization a prerequisite for high-performance execution.

As bit-width decreases, the model gains efficiency at the cost of potential accuracy degradation. Table 5-1 outlines the typical performance characteristics observed during quantization.

Copyright © 2026 Texas Instruments Incorporated

Table 5-1. Accuracy and Efficiency Trade-offs

Precision	Size Reduction	Speed Improvement	Typical Accuracy Drop
Float32	1x (Baseline)	1x (Baseline)	0%
INT8	4x	2x – 4x	< 1%
INT4	8x	3x – 6x	1% – 5%
INT2	16x	4x – 8x	5% – 15%

To achieve high accuracy at low bit-widths, quantization-aware training (QAT) is recommended. QAT simulates quantization errors during the training process, allowing the model weights to adapt to the reduced precision. For more details, visit

- Tiny ML Tensorlab User's Guide: Quantization
- Tiny ML Tensorlab User's Guide: Standalone Quantization Examples

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "09_test_cnn_quantization_model.py" script, which demonstrates the complete pipeline: from FP32 training to INT8 Quantization-Aware Training and finally to ONNX exportation.

The following log shows a complete execution of the script, including the baseline FP32 training, the subsequent Quantization-Aware Training (QAT) phase, and the successful exportation of the quantized model.

```
--- Stage 1: FP32 Training ---
--- Commencing Model Training ---
Epoch 0: Val Loss = 0.3535 | Val Accuracy = 0.8992
Epoch 1: Val Loss = 0.2269 | Val Accuracy = 0.9410
Epoch 2: Val Loss = 0.2205 | Val Accuracy = 0.9359
Epoch 3: Val Loss = 0.1928 | Val Accuracy = 0.9449
Epoch 4: Val Loss = 0.1590 | Val Accuracy = 0.9593
--- Final Test Set Evaluation ---
>> Test Loss: 0.1732 | Test Accuracy: 0.9522 <<

--- Stage 2: INT8 QAT ---
--- Commencing Model Training ---
Freezing BN for subsequent epochs
Epoch 0: Val Loss = 0.1289 | Val Accuracy = 0.9617
Epoch 1: Val Loss = 0.1197 | Val Accuracy = 0.9641
Freezing ranges for subsequent epochs
Epoch 2: Val Loss = 0.1499 | Val Accuracy = 0.9530
Epoch 3: Val Loss = 0.1278 | Val Accuracy = 0.9623
Epoch 4: Val Loss = 0.0985 | Val Accuracy = 0.9713
--- Final Test Set Evaluation ---
>> Test Loss: 0.1041 | Test Accuracy: 0.9686 <<

--- Stage 3: Exporting to ONNX ---
ONNX model successfully exported: ...\model\mnist\mnist_model_int8.onnx
```

5.6 Model Compilation

After the model has been trained and exported to the ONNX format, the final step in the development lifecycle is compilation. This process converts the high-level ONNX computational graph into machine-readable code optimized for the target microcontroller (MCU).

The TI Neural Network Compiler (NNC) is utilized to bridge the gap between a general-purpose ONNX model and an embedded device. The NNC transforms the model into one of two primary deployment formats, depending on your project requirements:

- **Static Library (.a):** The model is compiled into a static library using a cross-compiler (for example, tiarmclang). This is the standard method for production, where the library is linked directly into a Code Composer Studio™ (CCS) project.
- **Source Code Package (.tar):** The model is decomposed into human-readable C source files. This approach is best for deep inspection of the generated kernels or for environments where direct linking of binaries is restricted. This mode does not require an external cross-compiler during the compilation phase.

Copyright © 2026 Texas Instruments Incorporated

Output File Structure

The output files generated by the compiler follow a standardized directory structure:

```
root_path/
├── lib0.c                # Optimized source code lib0
├── lib1.c                # Optimized source code lib1
├── lib2.c                # Optimized source code lib2
├── mod.a                 # Compiled static library file (Link this to CCS)
└── tvmgen_default.h      # C header file containing inference function signatures
```

Command-Line Interface (CLI)

For advanced users, the *tvmc* command-line interface provides granular control over the compilation process. Detailed documentation can be found in the TI Neural Network Compiler for MCUs User's Guide: Compilation Explained.

To compile an ONNX model into a static library for an MSPM0 (Cortex-M0+) target, execute the following command in your terminal:

```
tvmc compile \
    --target="c, ti-npu skip_normalize=true output_int=true" \
    --target-c-mcpu=cortex-m0plus \
    .\model.onnx \
    -o artifacts_m0/mod.a \
    --cross-compiler="tiarmclang" \
    --cross-compiler-options="-Os -mcpu=cortex-m0plus -march=thumbv6m -mtune=cortex-m0plus -mthumb
-mfloat-abi=soft -I. -Wno-return-type"
```

Example Script

Download the Tiny ML Workflow Example Python Script Package and use the included "10_test_cnn_quan_compile_npu_lib.py" script. The script provides a generic wrapper around the *tvmc* command-line interface. The script allows developers to manage target-specific compiler flags with minimal effort.

After executing the script using the terminal, verify the training progress by monitoring the logs:

```
============================================================
Output Mode    : LIB
Target Device  : MSPM0 (cortex-m0plus)
NNC Target     : c, ti-npu skip_normalize=true output_int=true
Input Model    : C:\ti\tinyml-tensorlab\UserTestFolder\model\mnist\mnist_model_int8.onnx
Output Dir     : C:\ti\tinyml-tensorlab\UserTestFolder\model\mnist\outputLib
============================================================
WARNING:ti_mcu_nnc:GCONV layer NOT offloaded to TI-NPU
WARNING:ti_mcu_nnc:GCONV layer config: 8-bit input [9, 9, 16], 8-bit output [4, 4, 10], 8-bit
kernel [3, 3], strides [2, 2], padding [(0, 0), (0, 0)], data layout NHWC, kernel layout HWIO
WARNING:ti_mcu_nnc:GCONV does not support 10 output channels. Please change to multiples of 4.
WARNING:ti_mcu_nnc:AVGPOOL layer NOT offloaded to TI-NPU
WARNING:ti_mcu_nnc:Global AVGPOOL layer config:8-bit input [4, 4, 10], 8-bit output [10]
WARNING:ti_mcu_nnc:Global AVGPOOL does not support 10 input channels. Please change to multiples of
4.
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc: TI NPU Offloading Report
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc: LAYER/OPERATOR COUNT SUMMARY
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc: Layer Patterns Offloaded
INFO:ti_mcu_nnc: ------------------------------------------------------------
INFO:ti_mcu_nnc: Layer Pattern                                    Count
INFO:ti_mcu_nnc: ------------------------------------------------------------
INFO:ti_mcu_nnc: FCONV                                            1
INFO:ti_mcu_nnc: PAD                                              2
INFO:ti_mcu_nnc: GCONV                                            1
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc: Operators not Offloaded
INFO:ti_mcu_nnc: ------------------------------------------------------------
INFO:ti_mcu_nnc: Operator                                         Count
```


Copyright © 2026 Texas Instruments Incorporated

```
INFO:ti_mcu_nnc: ------------------------------------------------------------
INFO:ti_mcu_nnc: reshape                                                  1
INFO:ti_mcu_nnc: nn.pad                                                   2
INFO:ti_mcu_nnc: qnn.conv2d                                               1
INFO:ti_mcu_nnc: cast                                                     4
INFO:ti_mcu_nnc: nn.bias_add                                              1
INFO:ti_mcu_nnc: multiply                                                 2
INFO:ti_mcu_nnc: right_shift                                              2
INFO:ti_mcu_nnc: clip                                                     2
INFO:ti_mcu_nnc: sum                                                      1
INFO:ti_mcu_nnc: add                                                      1
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc: MODEL PARTITION - TOPOLOGICAL VIEW
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc: Layer/Operator                                    Offloaded
INFO:ti_mcu_nnc: ------------------------------------------------------------
INFO:ti_mcu_nnc: Input
INFO:ti_mcu_nnc: reshape                                                [ ]
INFO:ti_mcu_nnc: nn.pad                                                 [ ]
INFO:ti_mcu_nnc: nn.pad                                                 [ ]
INFO:ti_mcu_nnc: FCONV                                                  [X]
INFO:ti_mcu_nnc: PAD                                                    [X]
INFO:ti_mcu_nnc: GCONV                                                  [X]
INFO:ti_mcu_nnc: PAD                                                    [X]
INFO:ti_mcu_nnc: qnn.conv2d                                             [ ]
INFO:ti_mcu_nnc: cast                                                   [ ]
INFO:ti_mcu_nnc: nn.bias_add                                            [ ]
INFO:ti_mcu_nnc: multiply                                               [ ]
INFO:ti_mcu_nnc: right_shift                                            [ ]
INFO:ti_mcu_nnc: clip                                                   [ ]
INFO:ti_mcu_nnc: cast                                                   [ ]
INFO:ti_mcu_nnc: cast                                                   [ ]
INFO:ti_mcu_nnc: sum                                                    [ ]
INFO:ti_mcu_nnc: add                                                    [ ]
INFO:ti_mcu_nnc: multiply                                               [ ]
INFO:ti_mcu_nnc: right_shift                                            [ ]
INFO:ti_mcu_nnc: clip                                                   [ ]
INFO:ti_mcu_nnc: cast                                                   [ ]
INFO:ti_mcu_nnc: Output
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc:
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc: TI Model Library Memory Usage (mod.a)
INFO:ti_mcu_nnc: ============================================================
INFO:ti_mcu_nnc: Code:                      1952 bytes (    1.91 KB)
INFO:ti_mcu_nnc: RO Data:                   7061 bytes (    6.90 KB)
INFO:ti_mcu_nnc: RW Data:                   9540 bytes (    9.32 KB)
INFO:ti_mcu_nnc: Total:                    18553 bytes (   18.12 KB)
INFO:ti_mcu_nnc: ============================================================
```


Copyright © 2026 Texas Instruments Incorporated

6 Edge AI Application: Digit Recognition

Digit recognition technology is a fundamental capability with broad applications across embedded and personal consumer markets. However, deploying reliable recognition on embedded systems presents a significant challenge: traditional template-matching approaches lack robustness against the natural variability in human handwriting. The emergence of Edge AI offers a compelling design - enabling accurate, real-time inference directly on the device itself, without reliance on cloud connectivity. CPU-based inference pipelines demand substantial computational resources that exceed the processing capabilities typically available on resource-constrained microcontrollers.

The MSPM0G5187 microcontroller addresses this challenge by integrating a dedicated TinyEngine NPU, designed to accelerate ML workloads with minimal power and memory overhead. The design demonstrates classification of single-digit character images into one of 10 categories (0-9). The application receives a 28 × 28 grayscale image as pixel data over the UART back channel, performs hardware-accelerated model inference using the on-chip NPU, and sends the recognized digit class back to a host GUI for display.

Leveraging the on-chip NPU, the digit recognition system achieves approximately 99% classification accuracy with an inference latency of only 6.05ms, while consuming just 73KB of Flash and 10.9KB of RAM, as shown in Table 6-1. These results demonstrate the viability of on-device machine learning on resource-constrained microcontrollers.

Table 6-1. Digit Recognition Edge AI Design Performance

Metric	Value
Accuracy	Approximately 99%
Flash Usage	73KB
RAM Usage	10.9KB
Inference Latency (NPU)	6.05ms
Inference Energy(1) (AVG)	170.38uJ

(1) The inference energy was measured using EnergyTrace™ software during repeated executions of the tvmgen_default_run().

Note

To maintain accurate inference energy measurements on the LP-MSPM0G5187, remove the J6 jumper. This prevents additional current consumption from the onboard microphone and audio ADC, which can otherwise skew the results.

Table 6-2 shows the model main information.

Table 6-2. Digit Recognition AI Model Information

Property	Value
Model Architecture	CNN
Number of Parameters	60,000
Input Shape	(1, 28, 28)
Output Classes	10
Quantization	INT8

6.1 LeNet-5 Variant CNN Model

LeNet is a pioneering series of Convolutional Neural Network (CNN) architectures originally designed for recognizing handwritten digits and characters from small grayscale images. The digit recognition Edge AI model is a LeNet-5 variant CNN model compiled to NPU-native instructions through the TI Neural Network Compiler.

Figure 6-1 shows its work flow. The design accepts a flattened 28 × 28 grayscale image as a 784-element float32 input and produces a 10-element float32 output representing per-class confidence scores. The network consists of two consecutive convolutional blocks, each comprising a Conv2D layer with bias, ReLU activation, and 2×2 MaxPooling. These blocks progressively extract spatial features while halving the spatial resolution at each stage. The resulting feature map is then flattened and passed through a series of fully connected layers, which map the high-dimensional features to a 10-class output. Since all layers operate on INT8 quantized

Copyright © 2026 Texas Instruments Incorporated

weights and activations, a dequantization post-processing step is applied after the final fully connected layer to rescale the output back to float32. The predicted digit is determined by an argmax over the 10 output scores on the host side.

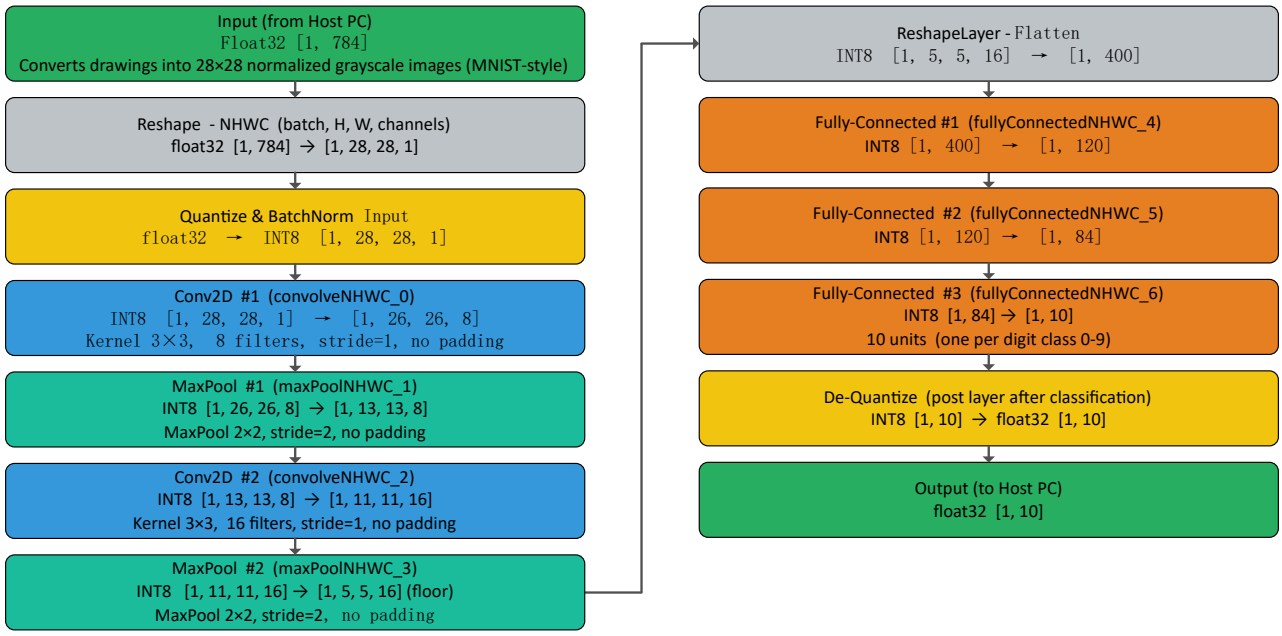


Figure 6-1. Digit Recognition AI Model Flow Chart

6.2 NPU/CPU Performance Comparison

The Edge AI model can be deployed to hardware via TI Neural Network Compiler, targeting either the dedicated hardware NPU or the host CPU. The TinyEngine NPU is a dedicated hardware accelerator specifically designed to execute neural network computations with high efficiency, delivering significantly reduced inference latency and energy compared to a general-purpose CPU.

To facilitate performance evaluation, the MSPM0 SDK provides both NPU and CPU implementation examples for user benchmarking, refers to:

- NPU-based Digit Recognition Edge AI Example
- CPU-based Digit Recognition Edge AI Example

Table 6-3 summarizes the performance comparison between the NPU-based and CPU-based Edge AI designs for the digit recognition application.

Table 6-3. NPU/CPU Performance Comparison

Performance Metric	NPU-Based Design	CPU-Based Design
Accuracy	Approximately 99%	Approximately 99%
Flash Usage	73KB	68KB
RAM Usage	10.9KB	8.1KB
Inference Latency	6.05ms	89.81ms
Inference Energy (AVG)	170.38uJ	2,447.45uJ

The NPU-based design achieves approximately 15 times lower inference latency compared to the CPU-based implementation and effectively reduces the energy consumption per inference by approximately 96%, making it a highly efficient choice for power-sensitive edge applications.

Copyright © 2026 Texas Instruments Incorporated

7 Edge AI Application: Waveform Classifier

MSPM0 provides an experimental waveform classifier design capable of performing real-time, continuous detection and classification of input signals, identifying common waveform types such as sine, triangle, and sawtooth waves. The design is built upon a 13K-parameter time-series classification model, offering strong scalability and extensibility.

The waveform classifier system achieves 100% (R square) classification accuracy with an inference latency of only 5.84ms, while consuming just 21.5KB of Flash and 3.3KB of RAM, as shown in Table 7-1.

Table 7-1. Waveform Classifier Edge AI Solution Performance

Metric	Value
Accuracy	100%
Flash Usage	21.5KB
RAM Usage	3.3KB
Inference Latency (NPU)	5.84ms
Inference Energy(1) (AVG)	236.97uJ

(1) The inference energy was measured using EnergyTrace during repeated executions of the *tvmgen_default_run()*.

Note

To maintain accurate inference energy measurements on the LP-MSPM0G5187, remove the J6 jumper. This prevents additional current consumption from the onboard microphone and audio ADC, which can otherwise skew the results.

Table 7-2 shows the model main information.

Table 7-2. Waveform Classifier AI Model Information

Property	Value
Model Architecture	CNN
Number of Parameters	14,124
Input Shape	Tensor [(1, 1, 128, 1)]
Output Classes	3 (Sawtooth, Sine, Square)
Quantization	INT8

Copyright © 2026 Texas Instruments Incorporated

7.1 Feature Extraction

Feature extraction is a critical component of common EdgeAI applications, particularly for audio and time-series signal processing. The feature extraction module provides efficient signal processing functions optimized for embedded systems, leveraging ARM libraries for optimal performance. The available feature extraction functions include:

- Real Fast Fourier Transform (RFFT): Converts time-domain signals to frequency domain representation
- Magnitude Scaling: Applies scaling for normalizing signal magnitude to appropriate range and prevent overflow/underflow for subsequent processing
- Complex Magnitude Calculation: Calculates the magnitude of complex numbers resulting from FFT
- DC Component Removal: Eliminates signal DC bias for improved feature accuracy
- Bin Calculation: Averages frequency components within defined bins for dimensionality reduction

Figure 7-1 shows the feature extraction processing pipeline.

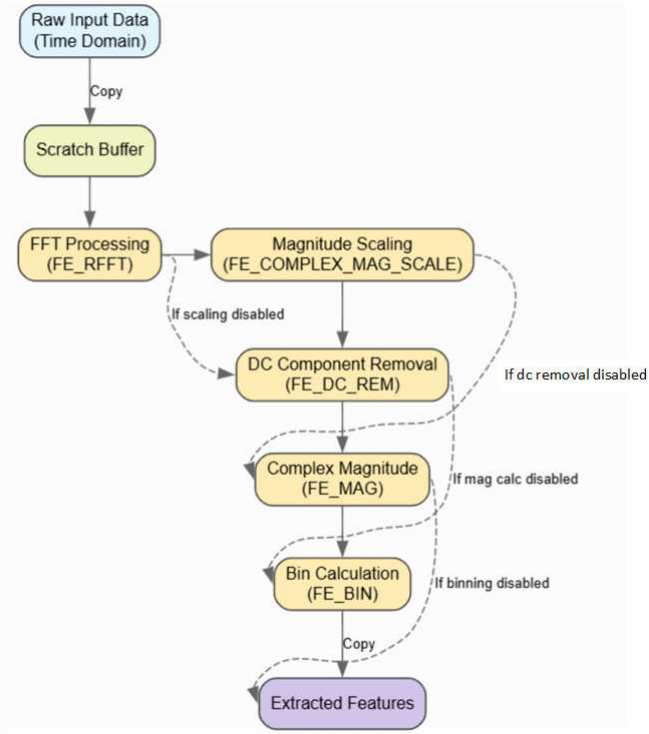


Figure 7-1. Feature Extraction Processing Pipeline

Table 7-3 shows the feature extraction processing pipeline configuration designed for waveform classifier. When the corresponding configuration symbol is defined, its associated task is executed within the processing pipeline.

Table 7-3. Pipeline Configuration for Waveform Classifier

Processing Task	Configuration Symbol	Defined or Not
RFFT	FE_RFFT	Y
Magnitude Scaling	FE_COMPLEX_MAG_SCALE	Y
DC Component Removal	FE_DC_REM	Y
Complex Magnitude Calculation	FE_MAG	Y
Bin Calculation	FE_BIN	Y

Copyright © 2026 Texas Instruments Incorporated

Table 7-4 shows the feature extraction configuration parameters designed for waveform classifier. For each inference task, eight frames are concatenated, where each frame processes 256 data points to generate 16 feature dimensions, resulting in a total of 128 feature dimensions fed into the waveform classifier AI model.

Table 7-4. Waveform Classifier Feature Extraction Parameters

Metric	Configuration Symbol	Value
Input Frame Size	FE_FRAME_SIZE	256
Feature Size per Frame	FE_FEATURE_SIZE_PER_FRAM	16
Frequency Bin Size	FE_BIN_SIZE	8
Normalize Bin Values	FE_BIN_NORMALIZE	1
Frame to Concatenate	FE_NUM_FRAME_CONCAT	8
Input Feature Size	FE_STACKING_FRAME_WIDTH	128

Note

The feature extraction parameters configured in Table 7-4 must match the parameters used during model training to verify consistent results.

Performance Consideration

- Optimized Computation: Leverages ARM-optimized libraries for efficient processing
- Memory Efficiency: Buffer swapping minimizes memory copy overhead
- Dimensionality Reduction: Bin calculation reduces feature dimensions, lowering memory usage and computation in downstream stages
- Parameters Tuning: Configuration parameters must be tuned to match available memory and compute resources

Copyright © 2026 Texas Instruments Incorporated

7.2 Time-Series Classification Model

Time-series classification identifies specific categories, operational states, or predefined labels within time-series data, and is widely applied across domains including healthcare, wireless monitoring, smart security, and automated testing.

The waveform classifier design implements a lightweight, 13K-parameter time-series classification model (CLS_13k_NPU), designed for efficient deployment on embedded neural processing units (NPUs). The design employs a 256-point Real Fast Fourier Transform (RFFT) feature extraction pipeline combined with 8-frame concatenation per AI inference task, accepting a 128-element 1D time-series feature vector as input and producing a 3-element output vector where each element represents the confidence score of the corresponding class.

The exported model file outputs an INT8 (8-bit quantized) tensor from the final fully connected layer. Since the raw output consists of per-class confidence scores in quantized form, a post-processing step is required to derive the final classification result.

Waveform Classifier Model Architecture

The model network is structured as a sequential feature transformation pipeline, where each stage builds directly upon the output of the previous to progressively refine and abstract the input representation toward a final classification decision. Figure 7-2 shows the workflow.

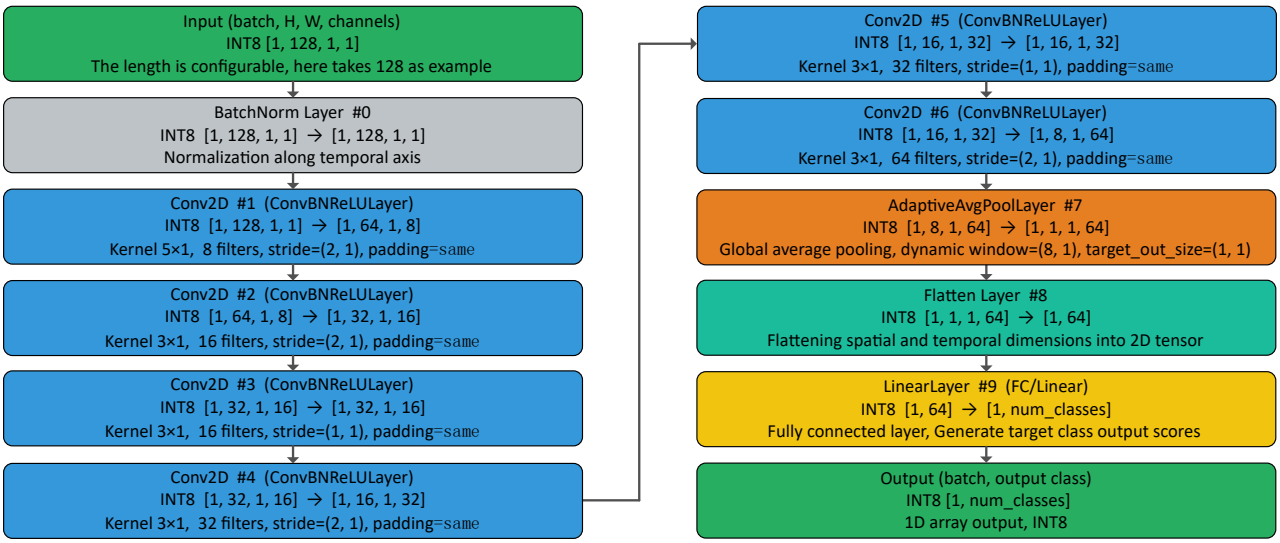


Figure 7-2. Time-series Classification Model (CLS_13k) Flow Chart

The pipeline begins with a Batch Normalization Layer, which standardizes the raw input sequence in real time, ensuring consistent feature scaling regardless of input signal amplitude variations before any learned transformation is applied. With the input stabilized, the network then progressively deepens the feature representation through six Consecutive Convolutional Blocks. Each block follows a consistent internal structure - a Conv2D Layer with bias, followed by Batch Normalization and ReLU activation, forming a repeating pattern that captures increasingly abstract temporal patterns across successive stages. The Batch Normalization within each block maintains training stability as depth increases, while the ReLU activation introduces the non-linearity necessary for the model to learn complex signal characteristics.

After the six convolutional stages, the resulting feature map passes through an Adaptive Average Pooling Layer, which collapses the spatial resolution to a fixed size of 1x1 regardless of the preceding temporal length. This design is particularly significant: by decoupling the model from any fixed temporal dimension, this inherently grants the model compatibility with varying time-series input lengths, providing flexibility for future deployment scenarios without requiring architectural changes.

Copyright © 2026 Texas Instruments Incorporated

The fixed-size pooled feature map is then flattened into a 2D tensor and passed through a final Fully Connected (FC) Layer, which maps the compressed, high-dimensional feature representation directly to the target class output scores.

Model Output and Post-Processing

The exported model file outputs an INT8 (8-bit quantized) tensor from the final fully connected layer. Since the raw output consists of per-class confidence scores in quantized form, a post-processing step is required to derive the final classification result. Typically, this is accomplished by applying an argmax operation over the output vector, which identifies the class index carrying the highest confidence score as the predicted classification outcome.

7.3 Model Memory Considerations

Developing edge AI designs on resource-constrained embedded platforms requires balancing a rigid, three-way optimization triad: algorithmic performance (accuracy, latency), non-volatile storage (FLASH/ROM), and run-time memory (SRAM).

- **Algorithmic Performance**: encompasses both inference accuracy and latency, which are primarily shaped by the model architecture - including network depth, layer width, and operator complexity.
- **Static Storage Memory (Flash/ROM)**: This is primarily determined by the model's total parameter count (weights and biases). In a fully quantized INT8 pipeline, each parameter maps directly to exactly one byte of Flash storage. Consequently, deeper networks with expanded channel widths linearly increases the static binary size. Sometimes, the variations in the input feature dimensions can also influence this static memory usage (for example, in fully-connected layer).
- **Dynamic Runtime Memory (SRAM/RAM)**: This is governed by the input feature size and the resulting activation tensors (feature maps) across intermediate layers. As a time-series slice propagates through the network, the processing core must allocate temporary workspaces to store layer inputs and outputs. Longer input temporal windows or higher feature dimensions exponentially inflate this peak runtime RAM requirement.

Driven by these hardware realities, deploying at the edge forces a departure from traditional accuracy-first mentalities. Instead, navigating a successful deployment demands a meticulous trade-off, balancing raw model performance directly against the hard physical boundaries of static Flash and peak dynamic SRAM.

To illustrate this relationship, Table 7-5 quantifies the memory footprint and resource utilization of waveform classifier across varying model sizes and input feature configurations (test with Edge AI Studio 1.6.1 Cloud Version).

Table 7-5. Model Memory Analysis

Model Variant (Parameter)	Flash (ROM) Size	RAM Size at Input = 64	RAM Size at Input = 128	RAM Size at Input = 256
TimeSeries_Generic_1K	Approximately 5.6KB	Approximately 2.7KB	Approximately 5.3KB	Approximately 10.4KB
TimeSeries_Generic_4K	Approximately 9.9KB	Approximately 1.2KB	Approximately 1.7KB	Approximately 2.7KB
TimeSeries_Generic_13K	Approximately 21.4KB	Approximately 2.3KB	Approximately 3.3KB	Approximately 5.3KB

Across all input dimensions, the smallest model (TimeSeries_Generic_1K) consistently consumes two to four times more runtime RAM than the larger models (TimeSeries_Generic_4K and TimeSeries_Generic_13K). This counterintuitive behavior stems from the fact that parameter count and runtime memory are driven by fundamentally different architectural decisions. The 1K model relies on a shallow topology that lacks early downsampling, causing large, high-resolution feature maps to persist in memory throughout the intermediate layers. In contrast, 4K and 13K models adopt deeper architectures with strided convolutions at the front-end of the network, which immediately reduce the temporal dimensions and significantly decrease the size of intermediate runtime tensors.

Copyright © 2026 Texas Instruments Incorporated

7.4 NPU/CPU Performance Comparison

The Edge AI model can be deployed to hardware via TI Neural Network Compiler, targeting either the dedicated hardware NPU or the host CPU. To facilitate performance evaluation, the MSPM0 SDK provides both NPU and CPU implementation examples for user benchmarking, refers to:

- NPU-based Waveform Classifier Edge AI Example
- CPU-based Waveform Classifier Edge AI Example

Table 7-6 summarizes the performance comparison between the NPU-based and CPU-based Edge AI designs for the waveform classifier application.

Table 7-6. NPU/CPU Performance Comparison

Performance Metric	NPU-Based Design	CPU-Based Design
Accuracy	100%	100%
Flash Usage	21.5KB	21.7KB
RAM Usage	3.3KB	3.1KB
Feature Extraction Latency	10.75ms	10.93ms
Feature Extraction Energy(1) (AVG)	386.68uJ	393.15uJ
Inference Latency	5.77ms	54.42ms
Inference Energy (AVG)	236.97uJ	2039.20uJ

(1) The feature extraction energy was measured using EnergyTrace during repeated executions of the FE_process().

In a comparative evaluation of hardware computational efficiency, the NPU-accelerated architecture exhibits substantial advantages in both performance and power metrics over the baseline CPU implementation.

During the pure inference stage, the NPU-based design delivers exceptional acceleration, achieving an approximately 8x reduction in execution latency while driving down per-inference energy consumption by approximately 88%.

When factoring the front-end feature extraction into the end-to-end system profile, the overall efficiency gains exhibit a localized attenuation due to the deterministic workload imposed by signal pre-processing on the CPU; the NPU-based pipeline still maintains a robust 3x latency compression and curtails per-execution energy consumption by approximately 74%.

Copyright © 2026 Texas Instruments Incorporated

8 Summary

This application note provides a comprehensive introduction to the Edge AI solution based on the MSPM0G5187 microcontroller, covering the hardware characteristics, software toolchains, and representative application examples.

Specifically, the document introduces two optimized workflows: TI Edge AI Studio for streamlined model development, and TI Tiny ML Tensorlab for advanced model engineering.

Two applications are validated to demonstrate the platform capability: the digit recognition design achieves approximately 99% classification accuracy, with the hardware NPU delivering 15 times lower inference latency and 96% better energy efficiency over the CPU-based implementation; the waveform classifier achieves 100% classification accuracy, with the NPU delivering 8 times lower inference latency and 88% better inference energy efficiency.

These results confirm that the MSPM0G5187, powered by the integrated TinyEngine NPU, is a compelling platform for deploying AI efficiently on resource-constrained embedded devices.

9 References

1. Texas Instruments, 80MHz Arm® Cortex®-M0+ MCU with 128kB flash, 32kB SRAM, USB 2.0 FS, I2S, ADC and TinyEngine™ NPU, product page.
2. Texas Instruments, *MSPM0 G-Series 80MHz Microcontrollers*, technical reference manual.
3. Texas Instruments, MSPM0G5187 LaunchPad™ development kit evaluation module, product page.
4. Texas Instruments, *How edge AI-accelerated Arm® Cortex®-M0+ MCUs bring more brain power to electronics*, technical article.
5. Texas Instruments, *TI's TinyEngine™ NPU unlocks edge AI acceleration in more embedded systems*, product overview.
6. Texas Instruments, Edge AI sensor BoosterPack™ reference design, product page.
7. Texas Instruments, Digit recognition with TI's microcontrollers (MCUs), product page.
8. Texas Instruments, CCStudio™ software development tools for edge AI, product page.
9. Texas Instruments, Tiny ML Tensorlab GitHub Repository, webpage.
10. Texas Instruments, Tiny ML Tensorlab User's Guide, webpage.
11. Texas Instruments, TI Neural Network Compiler for MCUs User's Guide, webpage.
12. Texas Instruments, *MSPM0 Edge AI User Guide*, resource explorer.
13. Texas Instruments, *AC Arc Fault Detection Using Edge AI on MSPM0 Low-Cost Microcontrollers*, application note.
14. Texas Instruments, Tiny ML Workflow Example Python Script Package, E2E™ support thread.
15. PyTorch, What is torch.nn really?, webpage.

10 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (June 2026) to Revision A (October 2026)	Page
• Updated *Introduction* section	3
• Updated *Edge AI Toolchains* section	5
• Added Figure 3-1 to *Edge AI Toolchains* section	5
• Updated *TI Edge AI Studio* section	6
• Updated *TI Tiny ML Tensorlab* section	7
• Updated *TI Neural Network Compiler* section	7
• Added *Rapid Prototyping Through TI Edge AI Studio* section	9
• Added *Project Setup* section	9
• Added *Data Capture* section	11
• Added *Hardware Setup* section	11
• Added *Feature Overview* section	12
• Added *Data Import* section	13

Copyright © 2026 Texas Instruments Incorporated

- Added *Classification Task* section ... 14
- Added *Regression Task* section ... 14
- Added *Forecasting Task* section ... 15
- Added *Annotation List* section ... 16
- Added *Data Annotation* section ... 16
- Added *Model Training* section ... 18
- Added *Training Parameter* section ... 20
- Added *Toolchain Dependency* section ... 21
- Added *Performance Reports* section ... 22
- Added *Model Export* section ... 25
- Added *Advanced Development Through TI Tiny ML Tensorlab* section ... 27
- Added *Environment Setup* section ... 27
- Added *Data Acquisition* section ... 28
- Added *Data Retrieval* section ... 28
- Added *Data Validation* section ... 28
- Added *Model Architecture Iterative Refactoring* section ... 30
- Added *Baseline Implementation* section ... 30
- Added *Refactoring With PyTorch Built-in Modules* section ... 31
- Added *Data Pipeline Optimization* section ... 32
- Added *Transition to CNN Architecture* section ... 33
- Added *Model Export* section ... 34
- Added *Quantization-Aware Training* section ... 34
- Added *Model Compilation* section ... 35
- Updated *Edge AI Application: Digit Recognition* section ... 38
- Updated *LeNet-5 Variant CNN Model* section ... 38
- Updated *NPU/CPU* Performance Comparison section and Table 6-3 ... 39
- Added note in *Edge AI Application: Waveform Classifier* section ... 40
- Updated title for Table 7-4 in *Feature Extraction* section ... 41
- Updated *Time-Series Classification Model* section ... 43
- Updated text and Table 7-5 in *Model Memory Considerations* section ... 44
- Updated text and Table 7-6 in *NPU/CPU Performance Comparison* section ... 45
- Updated *Summary* section ... 46
- Updated *References* section ... 46

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025