# Next-Generation Intelligent Terminal Embedded System Development Tutorial Manual

Using

The **NuMaker-X-M55M1** Development Board

Table of Contents

Part. A - M55M1 System Development Guide



Part. B - NuMaker-X-M55M1 Development Board Implementation Examples

# 1 Introduction to CPU, GPU, NPU, and Edge AI

## 1.1 Overview of CPU, GPU, and NPU

With the rapid advancement of artificial intelligence (AI) technologies, we are entering a new era of intelligent systems. Edge AI, as one of the core driving forces in this era, is revolutionizing the way we live and work at an unprecedented pace. The primary advantage of Edge AI lies in its ability to perform data processing directly on the device, significantly reducing dependency on cloud computing. This enables real-time data analysis and response, which is critical for applications that require high performance and low latency.

Compared to traditional cloud computing, Edge AI offers significantly improved efficiency and minimizes latency during the processing cycle. This capability enables a wide range of applications that rely on real-time responsiveness. For instance, autonomous vehicles depend on immediate analysis of road conditions to make split-second decisions. Smart healthcare devices require real-time interpretation of patient physiological data to support medical decision-making. Intelligent manufacturing systems can continuously monitor production line status and make adjustments on the fly. These applications are made possible through the support of Edge AI.

As technology continues to advance, Edge AI is expected to become increasingly widespread, extending beyond traditional data centers into various embedded devices and becoming an integral part of everyday life. From smart homes to industrial automation, Edge AI will not only be a component of the ongoing technological revolution but also a foundational element of how we live and work in the future.

### 1.1.1 Introduction to CPU, GPU, and NPU

1. CPU (Central Processing Unit)

   (1) Design Purpose:

   The CPU is the core processing unit of a computer, primarily responsible for executing instructions and performing logical operations. Often referred to as the "brain" of the computer, it interprets and carries out commands from software applications.

(2) Number of Cores:

Modern CPUs typically feature multiple cores, ranging from 1 to 16 or more.

a. Single-core: Suitable for basic computing tasks.
b. Multi-core: Capable of executing multiple processes concurrently, making it ideal for high-performance applications such as image processing and gaming.

(3) Performance Advantages:

CPUs are optimized for high single-core performance, focusing on complex control logic, branching decisions, and multitasking.

a. Control Logic: Manages conditional branching and sequential execution (e.g., pipelining techniques).
b. Branch Prediction: Minimizes performance penalties from conditional instructions.
c. Multitasking: Multi-core architecture allows simultaneous execution of multiple applications, enhancing overall system performance.

(4) Application Scenarios:

CPUs are mainly used in the following areas:

a. Running operating systems (e.g., Windows®, Linux®)
b. Office productivity software (e.g., Word®, Excel®)
c. General daily tasks such as web browsing

(5) Major Manufacturers:

a. Intel®: One of the world's leading CPU manufacturers, known for high-performance processors such as the Core i® series.
b. AMD®: Another major CPU vendor, recognized for its cost-effective solutions and strong multi-core capabilities, such as the Ryzen® series.

2. GPU (Graphics Processing Unit)

(1) Design Purpose:

The GPU was originally designed to accelerate graphics processing, primarily used for rendering images and visual output. However, with the increasing demand for computation, GPUs are now widely utilized in various domains, including:

a. Data Processing: Suitable for handling large-scale data workloads.
b. Scientific Computing: Ideal for high-performance applications such as simulations and weather forecasting.

(2) Number of Cores:

GPUs contain a significantly larger number of processing cores compared to CPUs, typically ranging from hundreds to thousands. This high core count makes GPUs especially well-suited for parallel computing tasks such as graphics rendering and matrix operations.

(3) Performance Advantages:

GPUs offer notable advantages in the following areas:

a. High Parallelism: Capable of executing a vast number of similar operations simultaneously, making them ideal for:
   - Matrix Operations: Such as linear algebra and neural network computations.
   - Deep Learning: Supporting large-scale data processing, including tensor operations.
b. High Throughput: Optimized for processing large volumes of data, often outperforming CPUs in specialized tasks.

(4) Application Scenarios:

Key use cases for GPUs include:

a. Graphics Rendering: Providing powerful rendering capabilities for gaming, animation, and design software.
b. Video Processing: Used for video encoding, decoding, and post-production workflows.
c. Big Data Analytics: Accelerating computation and analysis of large datasets.
d. Machine Learning: Essential for training deep learning models and executing AI applications.

(5) Major Manufacturers:

NVIDIA®: A global leader in GPU manufacturing. Its product lines, such as GeForce® and Tesla®, are widely adopted across gaming, professional design, and high-performance computing industries.

3. NPU (Neural Processing Unit)

(1) Design Purpose:

The NPU is a processor specifically designed for Artificial Intelligence (AI) and deep learning applications. Its primary objective is to accelerate the computation of neural network models, particularly tasks involving matrix multiplication, convolution operations, and other computationally intensive workloads.

a. AI Inference: Real-time processing of neural network inference, such as image and speech recognition.
b. AI Training: While some NPUs support training, most are optimized primarily for inference workloads.

(2) Number of Cores:

The architecture of an NPU typically consists of a large number of lightweight specialized cores engineered for efficient neural network computations.

a. May include hundreds to thousands of cores.
b. Delivers highly parallel computing performance, focusing on accelerating specific neural network operations.

(3) Performance Advantages:

Compared to traditional processors such as CPUs and GPUs, NPUs offer the following advantages in AI applications:

a. High Energy Efficiency: Hardware optimized for AI algorithms enables more efficient computation with lower power consumption.
b. Low Latency: Enables real-time AI inference, such as object detection in autonomous driving.
c. Custom Optimization: Equipped with built-in AI acceleration instruction sets, optimized for specific models such as CNNs and RNNs.

(4) Application Scenarios:

NPUs are widely used in the following fields:

a. Autonomous Vehicles: For object detection and path planning.
b. Smartphones: To accelerate AI functions such as image processing and voice assistants.

c. Edge Devices: Enables AI inference in Internet of Things (IoT) applications.
d. Cloud Computing: Efficiently handles inference tasks for deep learning models.

(5) Major Manufacturers:
a. Google®: Developed the TPU (Tensor Processing Unit), a dedicated AI accelerator categorized under NPUs.
b. Apple®: Includes a "Neural Engine" in its A-series chips, serving as the NPU for tasks like image processing and speech recognition.

| Component | CPU | GPU | NPU |
|---|---|---|---|
| **Design Purpose** | General-purpose computing, executes instructions and performs logical operations. | Large-scale parallel processing, primarily designed for graphics rendering. | Specifically designed to accelerate neural networks and deep learning workloads. |
| **Core Count** | Few cores (typically 1 to 16). | Many cores (hundreds to thousands). | Typically consists of hundreds of processing units optimized for matrix and convolution operations. |
| **Performance Advantages** | High versatility; limited parallel processing capability. | Highly parallel; ideal for large-scale data processing. | Optimized for neural network computation and inference tasks. |
| **Primary Application Areas** | Operating systems, office software, basic computing tasks. | Graphics rendering, video processing, big data analysis, machine learning. | Deep learning inference, neural network computation, embedded AI applications. |
| **Power Consumption** | Relatively low. | Higher, especially under heavy workloads. | Relatively low, particularly in mobile and edge devices. |

Table 1 Comparison Table of CPU, GPU, and NPU

### 1.1.2 Core Hardware: Nuvoton NuMaker-X-M55M1 Development Board

In the Edge AI revolution, hardware selection plays a critical role. The Nuvoton

M55M1 is a high-performance microcontroller designed specifically for Edge AI applications, offering powerful computing capability, low power consumption, and comprehensive development support. The NuMaker-X-M55M1 development board is an ideal platform for efficient edge computing and features the following highlights:

(1) High-Performance Microcontroller Architecture

The NuMaker-X-M55M1 development board is built on the ARM® Cortex®-M55 processor, delivering efficient computational power. It supports various mainstream AI training frameworks and enables real-time inference processing. This allows developers to run machine learning models directly on edge devices and apply them in a wide range of real-world scenarios.

(2) Low Power Consumption Design

The low-power architecture of the M55M1 is well-suited for energy-constrained embedded environments. Whether used in wearable devices or smart sensors, the board ensures extended operation without excessive power drain.

(3) Comprehensive Software and Hardware Development Support

Nuvoton provides a complete software development ecosystem for the NuMaker-X-M55M1 board, including drivers, development kits, and machine learning model support. Whether for beginners or professional developers, it offers a rapid path to creating edge AI applications.

### 1.1.3 In-Depth Learning and Applications

This book serves not only as a technical manual but also as a practical guide, helping readers master the application of Edge AI technology from both hardware and software perspectives, and from foundational concepts to advanced implementation. It covers the following key areas:

1. Hardware Configuration and Architecture

   Gain a thorough understanding of the internal architecture of the NuMaker-X-M55M1 development board, from the microcontroller and memory to various external interfaces, empowering readers to design and implement edge computing solutions using the board.

2. Model Training and Optimization

Edge AI development goes beyond hardware design and includes the selection and optimization of machine learning models. This book introduces how to select appropriate AI models and optimize them based on specific application requirements, thereby enhancing the computational performance of edge devices.

3. Application Case Studies

   Through real-world case studies in smart manufacturing, smart healthcare, and smart agriculture, this book demonstrates practical applications of Edge AI. These examples show how Edge AI can be integrated into real-world needs, transforming traditional industries, increasing productivity, reducing costs, and promoting sustainable development.

### 1.1.4 Target Audience

This book is designed for a wide range of readers. Whether you are a beginner exploring Edge AI technologies or a professional engineer looking to enhance your technical capabilities, you will find value in these pages. For beginners, foundational concepts and hands-on examples will help you quickly get started and acquire core skills. For experienced engineers, the book offers in-depth technical details and application cases to support the integration of Edge AI into real-world projects, improving work efficiency and innovation potential.

As Edge AI continues to evolve, it will drive innovation and reshape the way we live and work. This book will help you take the first step in Edge AI development and guide you into this dynamic and opportunity-rich field.

## 1.2 Introduction to Edge AI

1. What is Edge AI?

Edge Artificial Intelligence (Edge AI) refers to the deployment of AI computing capabilities directly on devices near the data source, such as smart sensors, industrial controllers, and surveillance cameras, instead of relying solely on traditional cloud servers or data centers. This distributed architecture enables edge devices to process and analyze data locally in real time, delivering low latency, enhanced privacy, and greater cost-efficiency. Pre-trained AI models are deployed on edge devices, allowing them to make fast, local decisions, ideal for time-sensitive application scenarios.

2. Advantages of Edge AI

Low latency, enhanced data privacy, offline capability, and energy efficiency.

## 1.3 Edge AI Application Domains

1. Smart Manufacturing: Real-Time Monitoring and Predictive Maintenance

Smart manufacturing is a prime application of Edge AI. Within factories, AI systems can monitor the operational status of machinery in real time and respond rapidly to anomalies. For example, predictive maintenance powered by sensor data can help identify potential equipment failures in advance, reducing downtime and maintenance costs. With computer vision, Edge AI can also be used for workplace safety monitoring, detecting unsafe behavior and ensuring compliance with safety standards.

2. Smart Home: Enhancing Quality of Life

Smart homes represent another rapidly growing field for Edge AI. By incorporating vision recognition, home devices can identify household members and adapt the environment accordingly. Edge AI enables real-time control of lighting, HVAC systems, and appliances based on environmental conditions or user preferences, offering a more comfortable and intelligent living experience. Devices such as smart locks, home surveillance systems, and voice assistants also leverage Edge AI to boost home security and convenience.

3. Smart Healthcare: Real-Time Monitoring and Health Management

Edge AI is transforming healthcare delivery through real-time monitoring of patient health. Wearable devices can continuously track physiological data such as heart rate, blood pressure, and body temperature. Edge AI processes this data locally and can trigger alerts when abnormalities are detected, enabling timely intervention. It also supports tasks like medication recognition and disease prediction. Fall detection is another critical application, particularly for elderly individuals, where the system can notify caregivers or medical services immediately upon detecting an incident.

4. Smart Agriculture: Improving Farm Management Efficiency

Edge AI is gaining momentum in the agricultural sector. Using image analysis, it monitors crop health and detects issues such as pests or nutrient deficiencies early. This facilitates precise treatment, improving productivity while reducing the

use of harmful chemicals for more sustainable farming practices. Additionally, Edge AI can continuously monitor environmental conditions, such as soil moisture and climate trends, providing actionable insights for better agricultural decision-making and enhanced crop yield and quality.

# 2 Introduction to NuMaker-X-M55M1

## 2.1 NuMaker-X-M55M1 Development Board Specifications

The NuMaker-X-M55M1 is a microcontroller series developed by Nuvoton Technology, specifically designed for edge AI applications. It offers exceptional computational performance and low power consumption, making it well-suited for high-performance requirements in edge computing. The core architecture and processing capabilities of this microcontroller exhibit the following technical features:

1. Processor Core
   (1) Arm® Cortex®-M55 Core

   The M55M1 microcontroller is powered by an Arm® Cortex®-M55 processor operating at a frequency of up to 220 MHz. It provides powerful digital signal processing (DSP) capabilities, making it suitable for various high-performance edge computing tasks. The Cortex®-M55 is designed with a focus on low power and high efficiency, fulfilling the dual demands of computation and energy efficiency in embedded systems.

   (2) Efficient Digital Signal Processing

   The core is capable of handling a wide range of DSP tasks efficiently and supports high-performance data processing. It delivers stable performance for applications such as audio processing, image processing, and other AI-related tasks.

2. Neural Processing Unit (NPU)
   (1) Arm® Ethos™-U55 NPU

   The M55M1 is equipped with an Arm® Ethos™-U55 neural processing unit (NPU), specifically designed for artificial intelligence applications, operating at 220 MHz. This NPU provides efficient acceleration for neural network computations and is particularly suited for running AI models on resource-constrained edge devices, significantly improving inference performance.

   (2) Support for Deep Learning Inference

The Ethos™-U55 NPU supports a wide range of neural network architectures, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and other deep learning models. It offers robust inference capabilities, making it highly effective in applications such as object recognition, speech recognition, and image classification.

(3) Support for YOLO Model

This NPU also supports inference for the YOLO (You Only Look Once) model, a popular real-time object detection algorithm widely used in autonomous driving, intelligent surveillance, and security systems.

3. Memory

(1) 1.5 MB SRAM

The M55M1 includes 1.5 MB of static random-access memory (SRAM), providing high-speed data access. This capacity is well-suited for large-scale computations and data processing tasks, particularly those required by high-performance AI applications.

(2) 2 MB Flash Memory

The microcontroller also integrates 2 MB of on-chip flash memory, which can be used for storing program code, configuration data, and other non-volatile information. This storage capacity enables the system to support more complex and feature-rich applications.

4. Expandability

(1) HyperBus Interface

The M55M1 supports a HyperBus interface, allowing for the expansion of external RAM and ROM. This interface is well-suited for embedded applications requiring large-scale data computation or storage, such as image processing, deep learning inference, or high-capacity data logging. With HyperBus support, this microcontroller delivers excellent scalability to meet demanding computational workloads.

5. Security Features

(1) Secure Boot

Secure Boot ensures that only authorized and unmodified code is executed during system startup, thereby preventing unauthorized code or malicious software from interfering with the boot process and ensuring system integrity from the outset.

(2) Arm® TrustZone

Arm® TrustZone provides hardware-level security by dividing the system into "secure" and "non-secure" zones. This architecture ensures that sensitive data is processed and stored exclusively within the secure zone, protecting against unauthorized access and enhancing overall system security.

(3) True Random Number Generator (TRNG)

The M55M1 includes a hardware-based True Random Number Generator (TRNG) capable of producing high-quality random numbers, which are essential for encryption and secure communications, thus improving data security.

(4) Encryption Modules (AES, ECC, RSA)

This microcontroller is equipped with integrated cryptographic modules supporting AES (Advanced Encryption Standard), ECC (Elliptic Curve Cryptography), and RSA (Rivest–Shamir–Adleman). These standard algorithms help secure data transmission and storage, ensuring that sensitive information remains protected from tampering or interception.

6. Support for Arm® Helium Technology

Arm® Helium technology is a hardware acceleration feature designed for digital signal processing (DSP) and machine learning (ML) workloads. It significantly enhances data processing efficiency and is especially valuable in low-power embedded systems. The M55M1 microcontroller incorporates Helium technology, offering the following advantages:

(1) Performance Boost

Helium enhances processing performance for ML and DSP applications, accelerating data handling and optimizing compute efficiency. It is particularly well-suited for real-time AI inference and data processing scenarios.

(2) Software Tool and Library Support: CMSIS-DSP and CMSIS-NN

Arm® provides powerful DSP and neural network (NN) libraries—CMSIS-DSP and CMSIS-NN—that fully leverage the Helium architecture. These tools simplify development and offer ready-to-use, high-performance algorithms, enabling developers to implement deep learning, audio processing, image processing, and more with ease on the M55M1.

7. Communication Interfaces

(1) UART (Universal Asynchronous Receiver/Transmitter)

Enables serial communication with other devices, offering a simple and low-cost connectivity solution.

(2) SPI (Serial Peripheral Interface)

A high-speed data transfer interface suitable for interaction with external instruments and sensor devices.

(3) $I^2C$ (Inter-Integrated Circuit)

A reliable and low-power communication protocol ideal for connecting multiple peripherals, such as sensors or display modules.

(4) USB (Universal Serial Bus)

Supports high-speed data exchange with various external devices, enhancing the system's expandability and peripheral integration.

8. Peripheral Features

(1) 24-Channel PWM (Pulse Width Modulation)

Allows control of devices such as motors, LEDs, and audio components. The multiple channels support synchronized control for precise output modulation.

(2) 2 Sets of SAR ADCs (Successive Approximation Register Analog-to-Digital Converters)

Convert analog signals to digital data, ideal for processing sensor inputs of various types.

(3) 1 Built-in Temperature Sensor

Provides real-time system temperature monitoring, making it well-suited for embedded environmental supervision and system protection tasks.

9. Low Power Design

(1) Multiple Low-Power Operating Modes

The M55M1 microcontroller supports several power-saving modes that significantly reduce energy consumption during idle or low-activity periods. This design is especially beneficial for portable devices and long-duration applications.

(2) Voltage Monitoring Modules

Equipped with multiple voltage monitoring units, the M55M1 ensures system reliability by detecting and responding to abnormal voltage levels, helping prevent system errors or crashes.

## 2.2 Hardware Configuration

The M55M1 microcontroller solution is based on a dual-core architecture consisting of the Arm® Cortex®-M55 (CPU) and the Ethos™-U55 (NPU). The main features are as follows:

1. CPU – Cortex®-M55
   (1) Operating Frequency Range: 200 to 240 MHz
2. NPU – Ethos™-U55
   (1) Performance: Supports up to 256 MACs (Multiply-Accumulate Operations) per clock cycle at 8-bit precision
   (2) Neural Network Inference Support: Capable of accelerating inference for neural network models such as CNNs (Convolutional Neural Networks) and YOLO
   (3) AI Acceleration: Provides dedicated acceleration for AI workloads in embedded applications
3. Memory
   (1) 1.5 MB SRAM: High-speed on-chip memory optimized for real-time data access in high-performance computing tasks
   (2) 2 MB Flash: Embedded flash memory for storing application code and non-volatile data
4. HyperBus Expansion – Supports One HyperBus Interface
   (1) Frequency: Half the system clock (e.g., system clock at 220 MHz, HyperBus runs at 110 MHz)
   (2) Maximum Expansion Capacity: Up to 32 MB
   (3) Use Case: Ideal for expanding high-performance embedded storage to support tasks such as image processing, deep learning inference, or large data caching
5. External Interface Support
   (1) Camera Module Interface: One dedicated interface for direct camera module connection, enabling use cases in image processing and visual recognition
   (2) PDM Digital Microphone Support: Up to 4 PDM (Pulse Density Modulation) digital microphone interfaces for audio capture and voice recognition applications

| | **M55M1** |
|---|---|
| **CPU** | Cortex®-M55 |
| **NPU** | Ethos™-U55 |

| | |
|---|---|
| **Clock** | 220MHz |
| **MAC/cc(8-bit)** | 256 |
| **Memory** | 1.5 MB SRAM |
| **HyperRAM (on-board)** | 8 MB |
| **Flash** | 2 MB Embedded |
| **HyperBus** | 1 (110 MHz, 32 MB) |
| **Camera** | 1 |
| **PDM Digital Mic** | 4 |

Table 2 Hardware Specification

### 2.2.1 Front View

1. Processor Core
   (1) Core: The M55M1 is built on the Arm® Cortex®-M55 processor with an integrated Ethos™-U55 NPU.
2. Memory Storage
   (1) HyperRAM: Supports high-performance external RAM expansion for handling larger datasets or faster data transfer.
3. Communication Interfaces
   (1) FS-USB Connector: Full-speed USB interface for data communication with external devices.
   (2) HS-USB Connector: High-speed USB interface suitable for applications requiring fast data transfers.
   (3) RJ-45 Interface: Supports Ethernet connectivity for network communication.
4. Wi-Fi Communication
   (1) ESP-12F Module: Built-in Wi-Fi functionality for wireless connectivity, ideal for IoT applications.
5. Audio and Sensors
   (1) MEMS Microphone: Supports multiple digital microphone inputs for voice processing and audio analysis.
   (2) Phone Jack (Audio Output): Provides audio output for multimedia or audio processing applications.
   (3) MPU6500: Integrated 6-axis accelerometer and gyroscope for motion detection and orientation sensing.
6. Built-in Debug and Programming Tools
   (1) Nu-Link2-Me: Onboard debugger for rapid development and firmware programming.
   (2) ICE USB Connector: For connecting development tools and performing system debugging.

(3) VCOM Switch: Virtual COM port for convenient serial communication between the developer and the board.
(4) Status LED: Indicates debugger or system status.
(5) Offline Programming Button: Supports firmware programming without a PC.

7. User Interaction and Expansion Interfaces
   (1) User Buttons: Can be used for debugging or application-level interaction.
   (2) User LEDs: Provide status indication or display test results.
   (3) mikroBUS Interface: Allows connection to various expansion modules for enhanced functionality.
   (4) M55M1 Expansion Interface: Arduino UNO-compatible headers for rapid prototyping and development.
8. Power and System Control
   (1) Reset Button: Resets the system for easier development and testing.
   (2) Power LED: Indicates board power status to verify system power is functioning correctly.
9. Other Features
   (1) CAN FD Transceiver: Supports the CAN FD protocol, ideal for industrial automation and automotive applications.

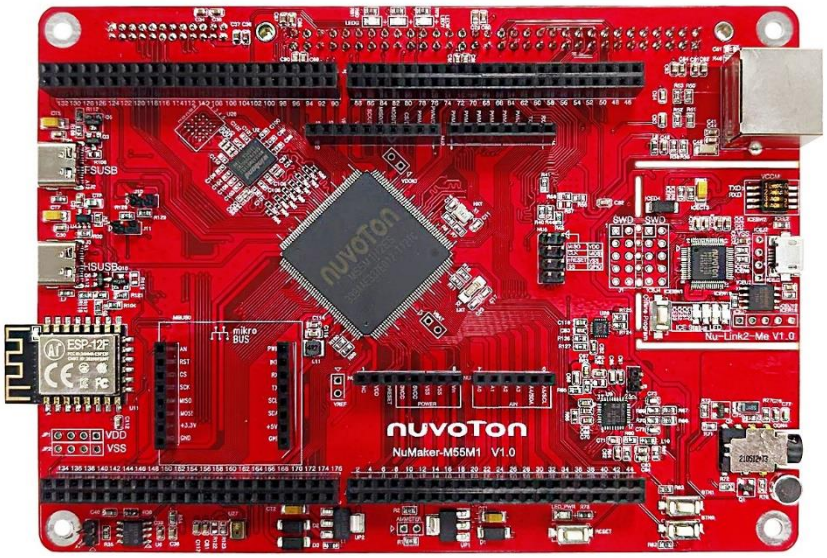


Figure 1: Front view of NuMaker-X-M55M1 development board

### 2.2.2 Rear View

1. Display Interface
   (1) TFT LCD Interface (LCD I/F Connector):

Supports connection to TFT LCD display modules, enabling visual interaction and data display. This reliable interface is useful for debugging and implementing display-related functionalities during development.

2. Camera Module Interface
   (1) CMOS I/F Connector:

   Supports the connection of CMOS camera modules, making it suitable for applications such as image processing, visual recognition, and real-time monitoring.

3. Storage Expansion
   (1) SD Card Slot:

   Provides support for external memory cards used for data storage or firmware updates. This is ideal for applications that require large-capacity storage, such as video recording or image data logging.

4. Power Management
   (1) ICEVCC Power Switch:

   Controls power supply to the ICE (In-Circuit Emulator) debugger, ensuring stable power during debugging sessions.

   (2) MCUVCC Power Switch:

   Controls the main power supply for the MCU, allowing developers to enable or disable the MCU power based on development or testing requirements.

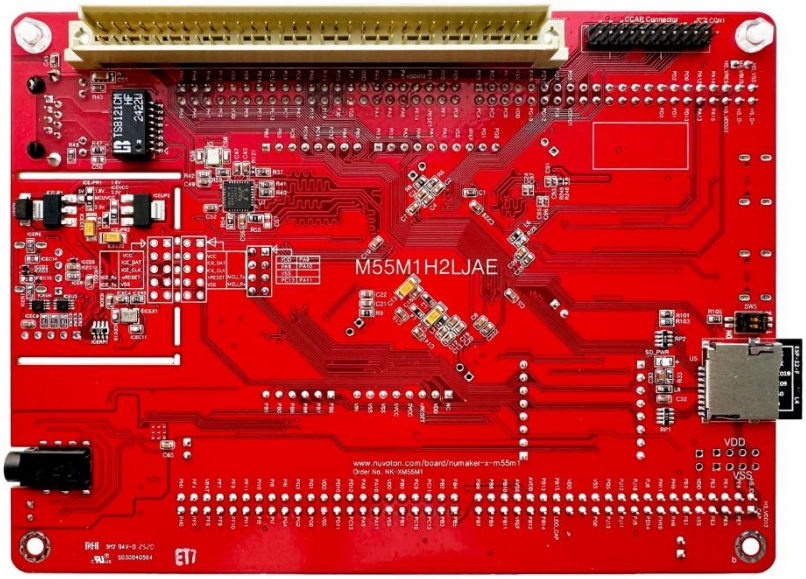


Figure 2: Back view of NuMaker-X-M55M1 development board

### 2.2.3 Power Supply Interfaces

1. FS-USB Connector
   (1) Function:

   Full-Speed USB interface that can serve as a power source for the development board.

   (2) Application Scenario:

   Suitable for standard USB power supply applications, allowing stable power delivery from a computer or other USB power source.

2. HS-USB Connector
   (1) Function:

   High-Speed USB interface that also supports power delivery.

   (2) Application Scenario:

   Ideal for embedded development requiring both high-speed data transfer and power through a single interface.

3. NU1 Pin8 (VIN)
   (1) Function:

   Provides an external DC voltage input pin for powering the development board.

   (2) Voltage Range:

   Requires a properly regulated DC input voltage as specified by the circuit design.

4. External VDD and VSS Connectors
   (1) External VDD Connector:

   Allows external positive voltage supply to the development board.

   (2) External VSS Connector:

   Ground (negative) terminal to pair with VDD, ensuring stable voltage supply.

   (3) Application Scenario:

   Suitable for custom power supply designs or special power requirements in advanced development environments.

5. ICE USB Connector

(1) Function:

Used for debugging and programming via USB, and also provides power to the board.

(2) Application Scenario:

When using the Nu-Link debugger, this interface can supply power and facilitate data communication simultaneously.


Figure 3: The power supply of NuMaker-X-M55M1 development board

## 2.3 Expansion Interfaces

The NuMaker-X-M55M1 development board supports multiple expansion interfaces, including the M55M1 Extension Connectors, Arduino UNO-compatible headers, and mikroBUS interface. These interfaces provide developers with a wide range of options for application development and peripheral integration.

### 2.3.1 Arduino UNO-Compatible Expansion Interface

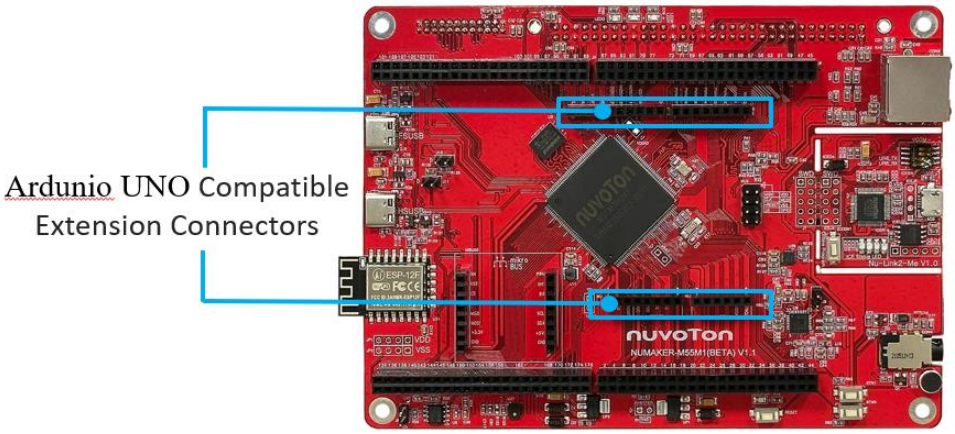


Figure 4: Arduino UNO-Compatible Expansion

1. Hardware Compatibility
   (1) Users can easily expand the board's functionality using Arduino-compatible modules such as sensors, displays, and wireless communication modules.

2. Pin Layout and Functions
   (1) Digital Signal Pins (Digital Pins)
      a. Provides up to 14 digital I/O pins (D0–D13) for input or output of digital signals.
      b. Supports multiple communication protocols such as UART and SPI, making it suitable for module control and data exchange.

   (2) Analog Signal Pins (Analog Pins)
      a. Includes 6 analog input pins (A0–A5) for reading analog sensor inputs, such as temperature, light intensity, or distance sensors.

   (3) Power Supply Pins
      a. Includes 3.3V, 5V, VIN (external power input), and GND pins to support various voltage requirements for external modules.

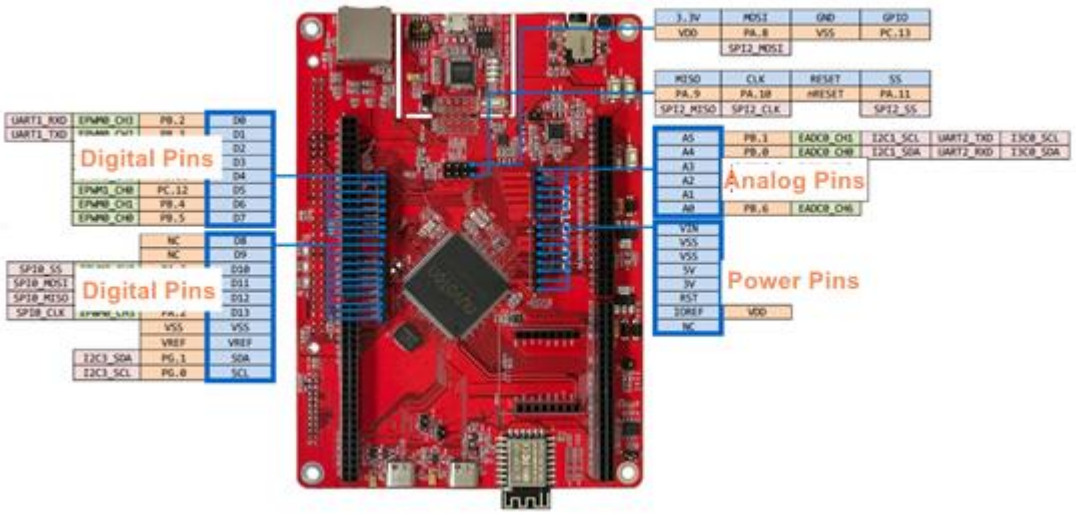


Figure 5: Pin layout of the Arduino UNO-compatible expansion interface

3. Use Cases and Module Compatibility (Supported Module Examples)

(1) GPIO Modules: Control digital signals for on/off switching or status indication.
(2) I2C/SPI Modules: Such as display panels and SD card modules.
(3) UART Communication Modules: Such as Bluetooth or LoRa wireless modules.
(4) PWM Applications: Control of servo motors or LED brightness adjustment.
(5) ADC Applications: Capture analog data from sensors such as sound detectors or environmental monitors.

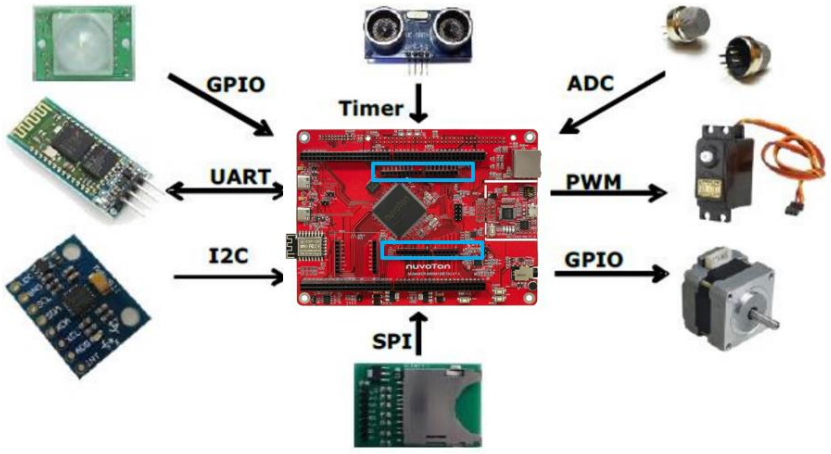


Figure 6: Examples of Arduino-compatible expansion modules

### 2.3.2 mikroBUS-Compatible Expansion Interface

1. Standardized Design
   (1) The mikroBUS interface, developed by MikroElektronika, is a standardized pinout interface designed to provide maximum expansion capability with minimal pin usage.
   (2) Its layout is simple and well-defined, supporting multiple communication protocols including I2C, SPI, and UART, making it suitable for rapid modular development.

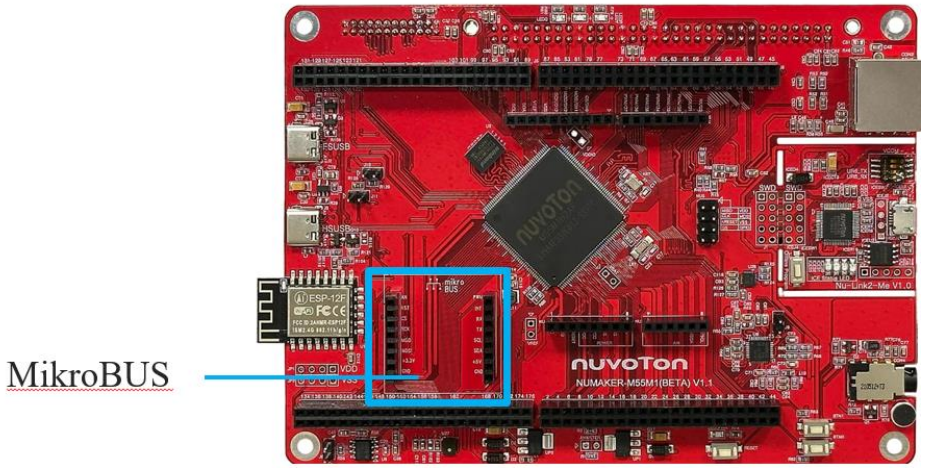


Figure 7: mikroBUS-Compatible Expansion Interface

2. Compatibility with Click Boards Series Modules
   (1) mikroBUS is primarily used for connecting to the Click Boards expansion module series.
   (2) The Click Boards ecosystem includes over 1,645 modules with various functions, covering applications such as sensing, communication, display, control, and power management.


Figure 8: Click Boards Expansion Modules

### 2.3.3 M55M1 Extension Connectors

1. Standardized Interface Design
   (1) The connectors are positioned on the top and bottom edges of the development board, providing multiple pins for external device expansion.
   (2) The configuration is customizable based on application needs and can connect to various sensors or dedicated functional modules.

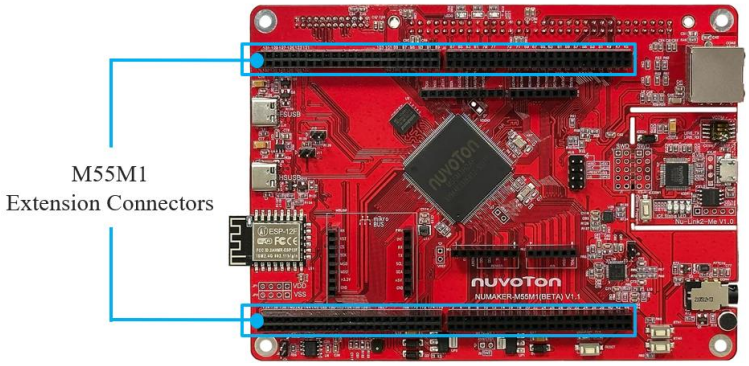


Figure 9: M55M1 Extension Connectors

2. High Compatibility
   (1) Supports a wide range of peripheral interfaces including GPIO, I2C, UART, and SPI protocols.
   (2) Enables seamless integration with various external devices, such as displays, storage modules, and data communication peripherals.

# 3 Full Deployment Workflow for Application Development

## 3.1 Overall Deployment Flowchart

The complete deployment process for the NuMaker-X-M55M1 development board is divided into five major stages (as shown in the diagram), which include: TinyML training, model quantization and conversion, Vela compilation and configuration, C language programming, and deployment/flashing to the development board. Each stage is described in detail below.

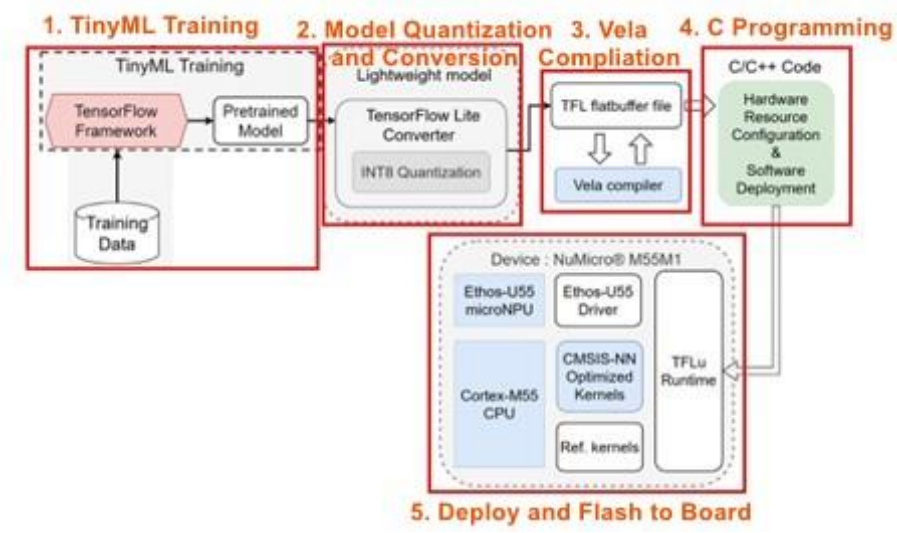


Figure 10. Using TensorFlow Framework

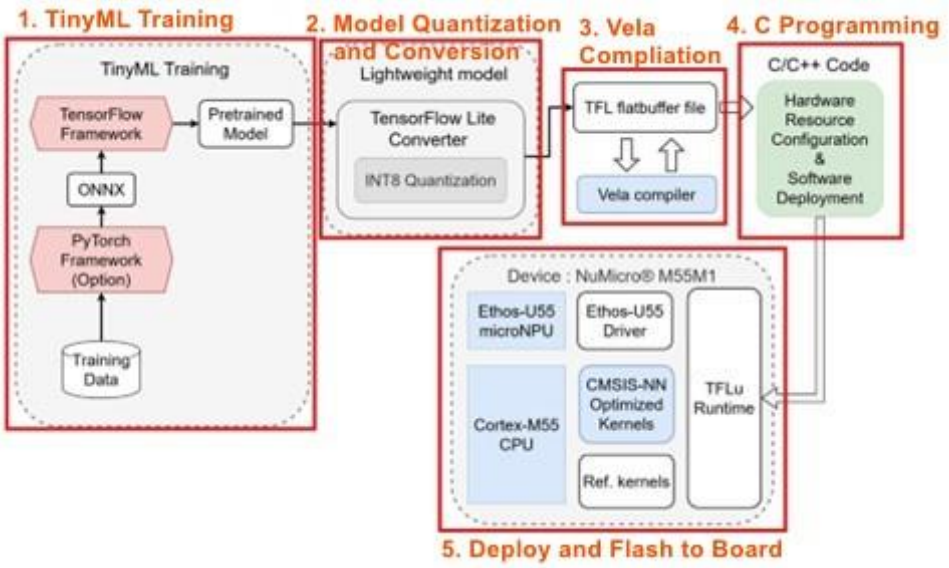


Figure 11. Using PyTorch Framework

### 3.1.1 TinyML Training

The TinyML training stage forms the foundation of the entire workflow, aiming to generate a pre-trained model. The steps include:

1. Framework Selection: Both TensorFlow and PyTorch frameworks are supported. When training with PyTorch, the model must be converted to TensorFlow format via ONNX to be compatible with the target development board.

2. Dataset Preparation: Developers need to prepare a high-quality dataset (e.g., from open sources like Kaggle or Roboflow), split into training, validation, and test sets.

3. Model Selection: Lightweight models such as YOLOX-Nano are used due to their small size and high performance, making them ideal for resource-constrained embedded systems.

4. Output: A pre-trained model is produced, serving as the foundation for subsequent quantization and deployment.

### 3.1.2 Model Quantization and Conversion

To meet the performance requirements of the M55M1 board, the trained model must be quantized and converted:

1. TensorFlow Lite Converter: Converts a TensorFlow model to TensorFlow Lite (TFLite) format, which is lightweight and optimized for embedded environments.

2. Quantization: INT8 full integer quantization is used to significantly reduce model size and memory usage, while maximizing performance on the built-in Arm® Ethos™-U55 NPU.

### 3.1.3 Vela Compilation and Configuration

Vela is a specialized compiler developed for Arm® Ethos™-U NPU to optimize TFLite models and improve inference performance:

1. Compiler Functions: Optimizes or merges supported neural network operators for execution on the Ethos™-U55 NPU. Unsupported operations are handled by the Cortex®-M55 CPU.
2. Configuration Options: Developers can choose between memory optimization (minimizing SRAM usage) or performance optimization (maximizing inference speed).
3. Output: An optimized TFLite model ready for hardware deployment.

### 3.1.4 C Language Programming

To deploy the model to the M55M1 board, developers must write C/C++ code for hardware configuration and inference logic:

1. Hardware Resource Configuration: Use C/C++ to control the M55M1 peripherals (e.g., NPU, PWM, UART).
2. Inference Logic: Implement input data preprocessing, model inference, and output result parsing.
3. Post-processing: Add logic for post-inference tasks tailored to the specific application, such as object detection and tagging.

### 3.1.5 Deployment and Flashing to the Development Board

Finally, the configured code and model are flashed to the M55M1 development board:

1. Flashing Tools: Use Arm® Keil MDK µVision 5 or other compatible tools for the M55M1.
2. Flashing Workflow: Includes project setup, hardware configuration, importing the model and code, and flashing via micro USB.
3. Validation and Debugging: After flashing, debug the system to ensure stable operation.

### 3.1.6 Summary

This complete deployment workflow integrates TinyML technology with embedded development, enabling efficient AI applications on edge devices using the M55M1 development board. The workflow is beginner-friendly while also providing the flexibility for professional developers to build sophisticated embedded AI systems.

## 3.2 TinyML Model Training

1. Application Scenarios Overview

This tutorial project covers four major smart application domains:

(1) Smart Factory: Includes personnel safety monitoring, equipment operation status detection, and predictive maintenance.
(2) Smart Home: Applied in smart appliance control and anomaly detection such as fire or intrusion alarms.
(3) Smart Healthcare: Enables fall detection and medication recognition.
(4) Smart Agriculture: Used for fruit quality analysis and fish fry counting.

These scenarios demand high real-time performance and accuracy, so the choice of model must balance both performance and resource usage.

2. Framework and Model Used
   (1) Framework: PyTorch
      a. PyTorch is one of the most popular deep learning frameworks due to its numerous advantages.

   (2) Dynamic Computation Graph:
      a. PyTorch uses a dynamic computation graph, meaning a new graph is created each time an operation is executed. This allows flexible handling of various data formats and model structures.
   (3) Intuitive API Design:
      a. Its API is designed in a Pythonic style, making model construction and debugging more intuitive.
      b. Tensor operations are similar to NumPy, making it ideal for those transitioning from data science to deep learning.
   (4) Community and Ecosystem:
      a. PyTorch has a strong developer community and extensive open-source models and tools, which reduce the learning curve.
      b. It supports various pre-trained models (e.g., ResNet, YOLO, Transformers), accelerating development.
   (5) GPU Acceleration Support:
      a. It seamlessly supports GPU computation
      b. Includes built-in distributed computing features for multi-GPU training.
   (6) ONNX Format Support:
      a. PyTorch-trained models can be exported to ONNX format for deployment on other platforms like TensorFlow or edge devices.

3. Model Selection: YOLOX-Nano

YOLOX-Nano is a lightweight version of the YOLOX series, designed for resource-constrained environments. It targets embedded systems where real-time object detection and limited computational resources must be balanced.

4. Dataset Preparation
   (1) For model training, datasets should be divided into three categories to optimize model performance at each stage:
      a. Training Set:
         Used for feature learning and parameter tuning to improve model fitting on training data.

b. Validation Set:
   Used to evaluate model performance during training and monitor overfitting.

c. Test Set:
   Used to evaluate the model's generalization ability and real-world performance on unseen data.

5. Model Training Platform Selection

Choosing the right environment is crucial for successful deep learning model training. Two common platforms are:

(1) Google Colab

A free cloud-based platform by Google, enabling users to write and run Python code in a browser—particularly well-suited for deep learning development.

Advantages:

a. Free GPU resources (Tesla K80, T4, P100, or V100)
b. Pre-configured environment (TensorFlow, PyTorch preinstalled)
c. Google Drive integration for easy file access/storage
d. Cross-platform (usable via browser on Windows/macOS/Linux)

Disadvantages:

a. Limited GPU usage time (sessions can be interrupted)
b. Shared resources lead to performance fluctuations
c. Requires stable internet and raises potential privacy concerns for cloud-stored data

(2) Local Training

Training on your own computer or server is ideal for users needing full performance and control.

Advantages:

a. Use high-performance GPUs (e.g., NVIDIA RTX series)
b. Full control over hardware/software environment

c. Higher data security (data stays local)
d. No cloud-imposed limits—suitable for long or large-scale training

Disadvantages:

a. High hardware costs and maintenance
b. More complex environment setup
c. Limited scalability compared to cloud
d. Requires regular hardware upkeep and updates

Environment Setup Using NVIDIA GPU

Proper hardware and software configuration is critical. For NVIDIA GPU-based training, follow these steps:

a. CUDA and cuDNN Installation and Configuration
   - CUDA: A parallel computing platform providing APIs for accelerating computation on NVIDIA GPUs.
   - cuDNN: A GPU-accelerated library optimized for deep learning operations (e.g., convolutional layers).
   - Installation Steps:
     1. Visit NVIDIA's official site to choose the right CUDA version based on your GPU and OS: CUDA Toolkit Archive
     2. Install the appropriate CUDA version.
     3. Visit the cuDNN download page: cuDNN Library
     4. Extract and install it into the corresponding CUDA directory.

Notes:

- Ensure CUDA and cuDNN versions are compatible with your framework (e.g., PyTorch).
- After installation, run nvcc --version to verify the setup.

6. Using Anaconda to Create Virtual Environments

Visit Anaconda Official Site and download the version suitable for your OS.

Advantages:

a. Manage multiple isolated Python environments to prevent dependency conflicts.

b. Built-in Conda package manager simplifies installation of frameworks like TensorFlow and PyTorch.
c. Simplified environment switching and maintenance for users working on multiple projects.

### 3.2.1 Introduction to TinyML Models

1. Foundational Stage – CNN

The history of Convolutional Neural Networks (CNNs) dates back to the 1980s. Yann LeCun proposed the earliest CNN architecture and introduced the LeNet-5 model in 1998. This model was successfully applied to handwritten digit recognition. LeNet-5 pioneered the use of CNNs in computer vision and inspired researchers to apply deep learning techniques to more complex image recognition tasks.

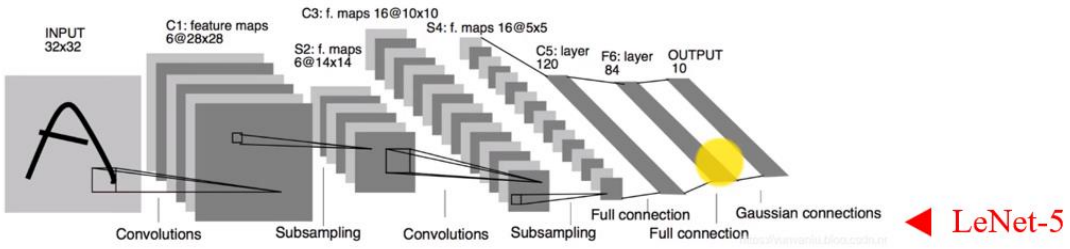


Figure 12: LeNet-5 Architecture

A major breakthrough followed in 2012 with the introduction of the AlexNet model by Alex Krizhevsky. By employing a deeper network structure and activation functions like ReLU, AlexNet significantly reduced the image classification error rate from 26.2% to 15.3%. It won the ImageNet competition and marked the rapid proliferation of deep learning in computer vision. The success of AlexNet demonstrated the potential of deep neural networks for handling large-scale image data and inspired the emergence of numerous subsequent models.

The R-CNN (Region-based CNN) series brought another significant advancement. It was the first to combine region proposals with CNNs for object localization and classification within an image. The core process of R-CNN involves using bounding boxes to locate multiple object positions, then employing deep learning models for feature extraction and classification. While R-CNN offers high

accuracy, its multi-stage process results in slower inference speed and higher hardware requirements.

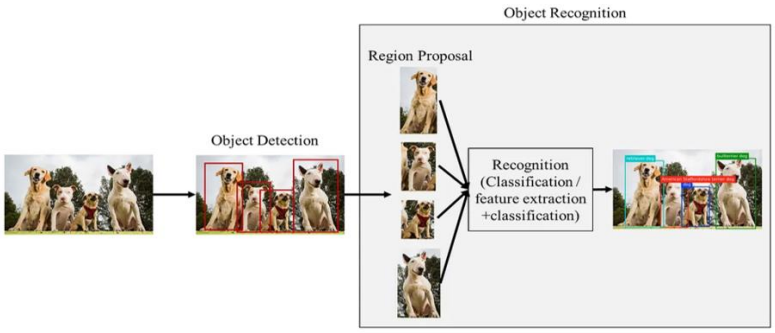


Figure 13: R-CNN Architecture Illustration

The introduction of the YOLO (You Only Look Once) series addressed the limitations of R-CNN. Proposed by Joseph Redmon and colleagues in 2016, YOLO introduced a novel one-stage object detection method. Its core concept is to complete object detection and classification in a single pass, greatly enhancing speed and efficiency. Compared to R-CNN, YOLO offers much faster performance and lower computational demands, making it well-suited for real-time applications and deployment on resource-constrained embedded systems, such as smart home devices and drones.

These technological advancements laid the foundation for the YOLO models and illustrate the progressive evolution of deep learning in object detection. This eventually led to efficient and lightweight models like YOLOX-Nano, capable of delivering high performance on low-power edge devices.

2. Core Features of the YOLO Series

The YOLO (You Only Look Once) object detection models are well-known for their efficient one-stage detection mechanism. These models divide an input image into a fixed grid, where each grid cell is responsible for detecting multiple bounding boxes, confidence scores, and class probabilities. This one-stage detection pipeline enables YOLO to perform object localization and classification in a single forward pass, making it highly suitable for applications requiring real-time responsiveness.

In contrast to traditional object detection methods, YOLO adopts a fully convolutional neural network (FCN) architecture. It simplifies the detection pipeline by integrating object localization, bounding box prediction, and classification into a single model. This approach significantly improves processing speed and makes the model more suitable for embedded deployment.

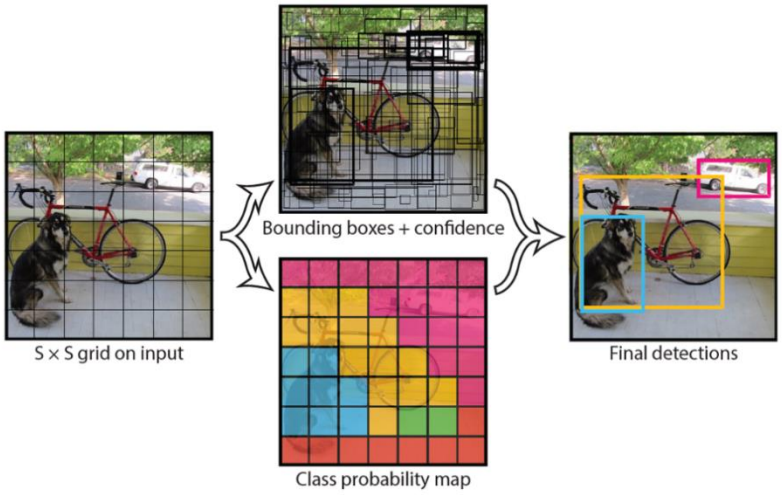


Figure 14: YOLO Object Detection Illustration

YOLO's primary advantages are its high speed and low computational overhead. Because its architecture performs detection and classification simultaneously, YOLO is ideal for embedded devices with limited computing resources. The simplified computation process also results in fewer hardware requirements, enabling efficient object detection even on edge devices with constrained resources.

## 3. Comparison Between R-CNN and YOLO

In the evolution of object detection, the R-CNN (Region-based Convolutional Neural Network) was one of the earlier deep learning models. It performs object detection by extracting region proposals from an image, then applying feature extraction and classification to each region. However, this multi-stage process—region proposal, feature extraction, and classification—results in slow inference speeds and high hardware demands.

YOLO fundamentally changed this process. Compared to R-CNN, YOLO integrates all stages into a single model using a one-stage detection method. This significantly improves inference speed and reduces system computational load.

Furthermore, YOLO adopts an anchor-free mechanism, eliminating the need for manually defined prior anchor boxes. This simplification further reduces computational complexity and enhances efficiency, making YOLO more suitable for real-time applications.

4. Evolution of the YOLO Series

- Key Features of Each Version

The YOLO series has evolved through multiple versions, each with significant improvements:

YOLOv5, developed using the PyTorch framework, is known for its ease of use and deployment. It introduces an AutoAnchor mechanism that automatically adjusts anchor box sizes based on input data, improving detection accuracy and deployment flexibility.

YOLOX-Nano is a lightweight variant specifically designed for edge computing. It features an extremely compact structure that enables efficient object detection on low-resource hardware. It uses EfficientNet-Lite as its backbone to reduce parameter count while maintaining high performance.

YOLOv7 refines the overall model architecture to improve inference speed and accuracy, making it especially suitable for real-time applications.

YOLOv8 introduces a modular design that supports multitask applications such as object detection, image segmentation, and pose estimation. This flexible architecture allows for adaptable configurations based on specific use cases.

5. YOLOX-Nano Model Overview

- Parameters and Performance

YOLOX-Nano is among the most lightweight models in the YOLO series. It contains only approximately 1.0 million parameters and has a model size of around 2MB. Despite its compact size, YOLOX-Nano strikes a strong balance between accuracy

and computational efficiency, making it an ideal choice for resource-constrained devices.

The model is specifically designed for embedded systems and IoT devices, emphasizing high performance with low compute demands.

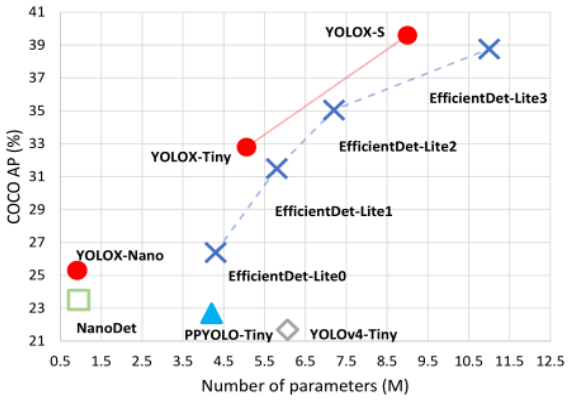


Figure 15: YOLOX-Nano Performance Comparison (Official GitHub)

6. Model Architecture

The YOLOX-Nano model consists of three main components:

Backbone: Utilizes EfficientNet-Lite, a lightweight, optimized network designed to reduce computation while maintaining performance.

Detection Head: Employs a YOLO-style detection head to handle multi-object detection tasks efficiently.

Anchor-Free Mechanism: The model adopts an anchor-free detection strategy, removing the need for predefined anchor boxes. This enhances flexibility and reduces configuration complexity.

7. Advantages and Application Scenarios

YOLOX-Nano's main advantages lie in its high-speed detection and compact architecture, making it particularly suitable for:

- Mobile devices

- Embedded systems
- IoT edge applications

Due to its small parameter count and low computational demands, YOLOX-Nano is easy to deploy and can operate on various hardware platforms. These characteristics make it well-suited for use cases that require high efficiency and low power consumption.

With its efficient design and optimized performance, YOLOX-Nano delivers high-quality object detection even on low-resource devices. Its lightweight structure and strong accuracy make it an ideal candidate for edge AI deployments, particularly in latency-sensitive and power-constrained environments.

As edge computing continues to evolve, lightweight models such as YOLOX-Nano are expected to play an increasingly vital role. They provide high-performance solutions under hardware constraints and are significant for the development of future intelligent systems. Through continuous optimization, YOLOX-Nano and its successors are poised to further enhance their capabilities and broaden their application scope—offering intelligent solutions for diverse resource-limited scenarios.

### 3.2.2 YOLOX-Nano Training Workflow


Figure 16: YOLOX Official Diagram

1. Reference Materials
   - The official YOLOX-Nano GitHub repository is available at: https://github.com/Megvii-BaseDetection/YOLOX

- This repository includes detailed documentation and example code for training, evaluation, and deployment.

2. Dataset Preparation

(1) Purpose and Key Points

a. Preparing a well-structured and diverse dataset is the first and most critical step in the YOLOX-Nano training pipeline. The goal is to provide the model with sufficient labeled image data to enhance its object detection accuracy and generalization capability.

(2) Recommended Dataset Sources

The following are three commonly used open-source platforms for image data collection and annotation:

a. Kaggle: The world's largest platform for data competitions, offering a rich selection of pre-annotated datasets, including popular object detection datasets like COCO.


Figure 17: Kaggle Dataset

b. Roboflow: A dedicated image dataset platform designed for machine learning. Provides tools for annotation, data augmentation, and dataset management, making it ideal for quickly building high-quality training datasets.


Figure 18: Roboflow Platform

c. Open Images Dataset: A large-scale open-source image dataset provided by Google. Contains diverse scenes and annotation categories, suitable for both object detection and classification tasks.


Figure 19: Open Images Dataset

Once the dataset is prepared using the above platforms, the process can move forward to the TinyML training stage, where the YOLOX-Nano model is trained using the curated data.

### 3.2.3 Machine Learning Framework (PyTorch or TensorFlow)

The model training process may utilize either PyTorch or TensorFlow as the primary framework. However, since the target development board (NuMaker-X-M55M1) operates under the TensorFlow framework for deployment, any models trained using PyTorch must be converted into the TensorFlow format via ONNX (Open Neural Network Exchange).

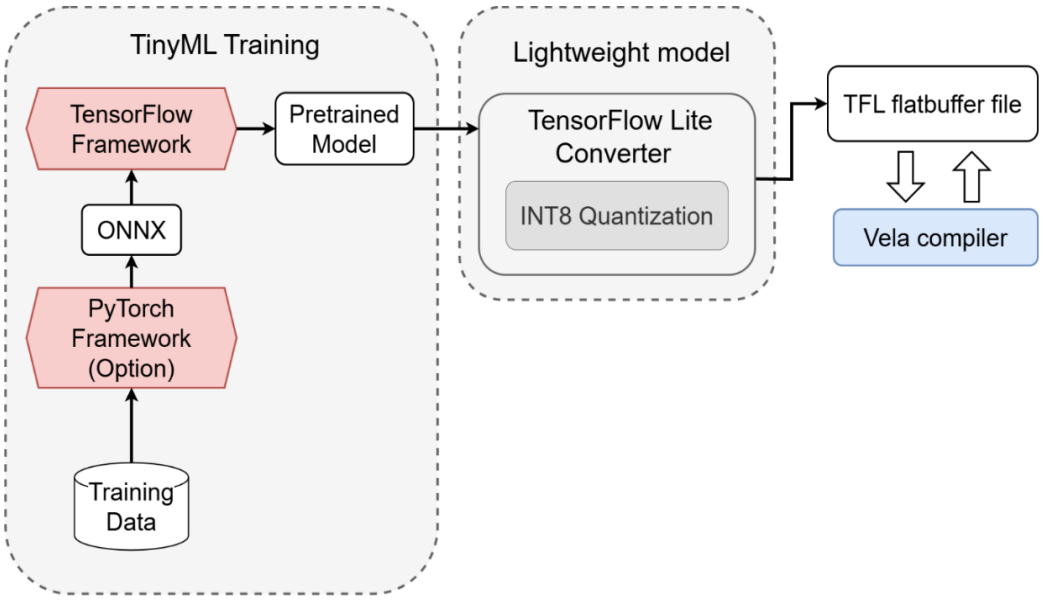


Figure 20: Model Training Workflow Diagram

### 3.2.4 Model Conversion

To deploy the model onto the embedded hardware, conversion from the original training format to TensorFlow Lite is required.

If the model is trained in PyTorch, it should be exported to ONNX format, and then converted into a TensorFlow-compatible format using ONNX tools.

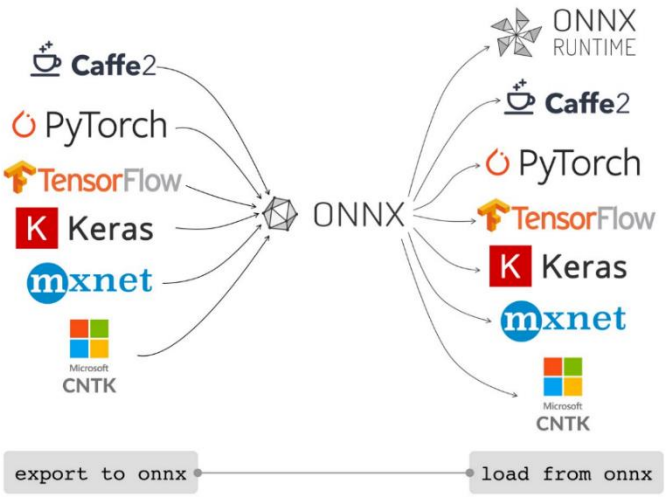


Figure 21: Model Conversion via ONNX

## 3.3 Model Optimization for Deployment (Model Quantization)

Overview

Model optimization is a key step in the TinyML workflow. Its purpose is to convert pretrained deep learning models into lightweight versions that can efficiently run on resource-constrained hardware platforms. This typically involves techniques such as model compression and quantization, which reduce computational and memory demands while preserving performance.

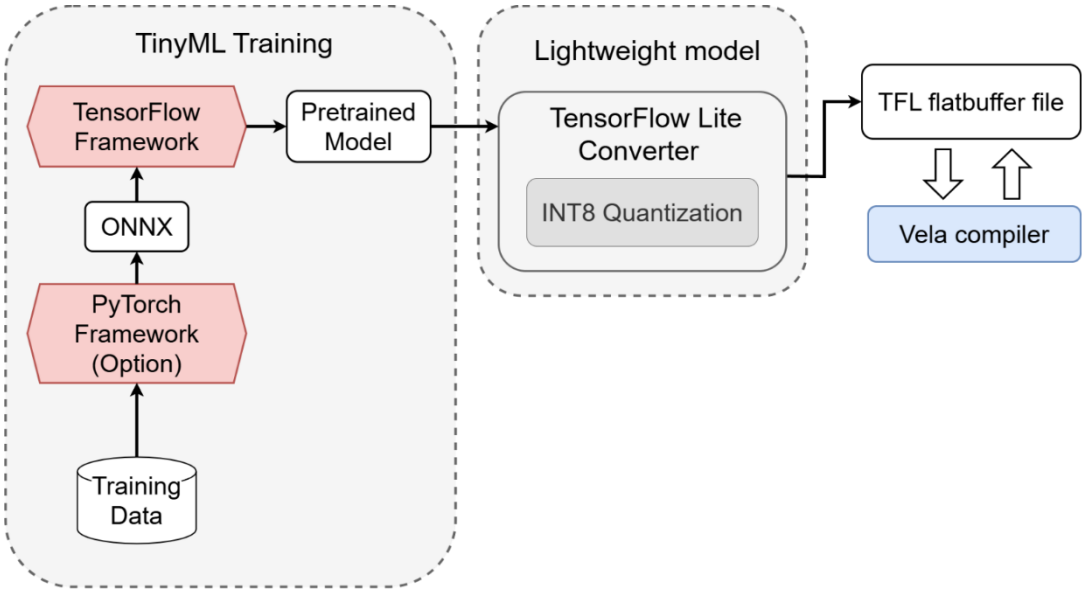


Figure 22: Training Pipeline Diagram

### 3.3.1 TensorFlow Lite Converter

1. TensorFlow Lite: Usage and Functionality
   (1) TensorFlow Lite is a mobile-focused deep learning library designed for efficiently deploying models on mobile devices, microcontrollers, and other edge platforms.
   (2) Its core goal is to preserve inference accuracy while significantly reducing memory footprint and computational overhead, making it ideal for low-resource environments.

2. Role of the TensorFlow Lite Converter

(1) TensorFlow Lite Converter is a critical component of TensorFlow Lite. It transforms standard TensorFlow models (typically in .pb or SavedModel format) into .tflite format.
(2) The resulting .tflite model is optimized for reduced storage size and is executable on embedded or edge devices.

3. Advantages of Quantization
   (1) Quantization techniques, such as INT8 quantization, further shrink model size and improve performance during inference.
   (2) These techniques are especially beneficial for edge deployment because they maintain accuracy while dramatically lowering memory and compute requirements.

### 3.3.2 Model INT8 Quantization

1. Background of Quantization Techniques
   (1) Quantization is a key method for optimizing deep learning models in environments with limited computational power, such as handheld or IoT devices.
   (2) While some accuracy loss may occur during quantization, the gains in efficiency often outweigh the drawbacks in practical applications.

2. Features and Benefits of INT8 Quantization
   (1) INT8 quantization converts model weights and activations from high-precision formats (e.g., FP32 or FP16) into 8-bit integers.
   (2) This conversion reduces memory usage and dramatically lowers computational load — a critical optimization for embedded or edge deployment.

3. Hardware Support and Use Cases
   (1) Many edge-AI processors (e.g., Arm® Ethos™ NPUs) are designed to accelerate fully quantized INT8 inference workloads.
   (2) Deploying INT8 models on such platforms unleashes their full performance potential while remaining resource efficient.

(4) Types of Quantization and Detailed Explanation
   (1) Full Integer Quantization

   Definition: Converts all weights and activations into integers. All computations are performed in integer space.

Advantages:

a. Significantly reduces model size.
b. Ideal for embedded systems, enhancing speed and efficiency.

Use Case: All examples in this tutorial use full integer quantization to demonstrate its benefits on hardware.

(2) Dynamic Range Quantization

Definition: Converts float weights to int during inference, while some parameters (e.g., initialization) remain float.

Advantages:

a. Easy to implement and quick to test.
b. Good for low-performance gain scenarios.

Trade-off: Slightly lower accuracy than full integer quantization.

(3) Hybrid Quantization

Definition: Part of the model remains in floating-point, while other parts are quantized.

Advantages:

a. Useful when some layers require floating-point precision.
b. Balances accuracy with performance and memory efficiency.

Use Case: Appropriate for models with high accuracy demands that still need compression.

(5) Quantization Workflow

Quantization involves several steps to reduce memory and compute load while preserving inference accuracy and stability:

(1) Prepare a Floating-Point Model
   a. After training, export your FP32 model (e.g., TensorFlow SavedModel).
   b. This full-precision baseline is used for accurate conversion.

(2) Calibration Dataset
   a. Use a representative dataset to simulate real-world usage.
   b. Proper calibration ensures inference accuracy is preserved post-quantization.

(3) Run Quantization Tool
   a. Use tools like TensorFlow Lite Converter to apply full integer quantization.
   b. This tool transforms weights, biases, and activations to INT8 automatically.

(4) Export as TFLite
   a. The quantized model is saved as a .tflite file.
   b. It is highly optimized and ready for deployment on edge/embedded platforms.

## 3.4 Vela Compiler (NPU Acceleration)

During the TinyML training workflow, the compiled model goes through the following process:

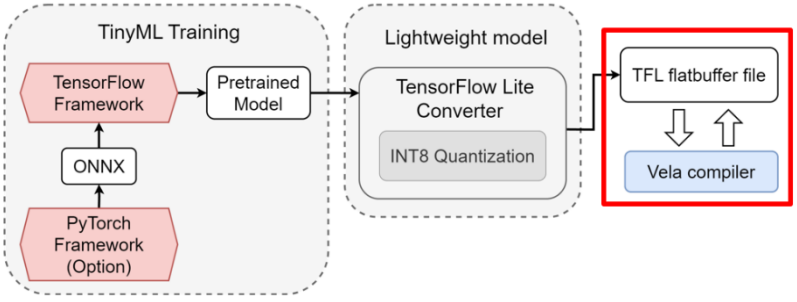


Figure 23: Training Workflow Diagram

### 3.4.1 Introduction to Vela

Vela is a neural network model compiler specifically designed for the Arm® Ethos™-U series Neural Processing Unit (NPU). It converts TensorFlow Lite models into optimized forms for execution on embedded systems that include Ethos™-U NPUs.

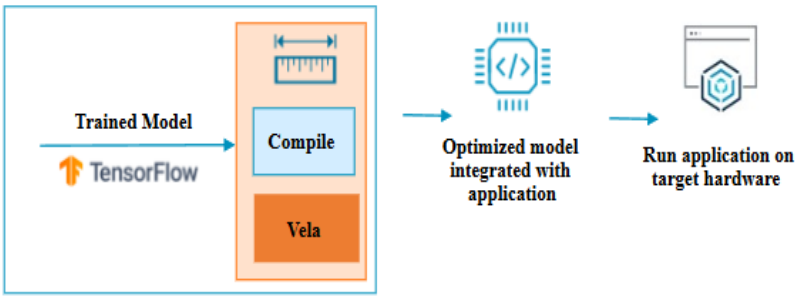


Figure 24: Vela Compilation Flow

1. Core Functions:
   (1) Optimizes TensorFlow Lite Models into Efficient TFLite Files
      a. Vela restructures models to fit embedded system constraints, trimming and optimizing layers to reduce memory usage and computational load.
      b. This ensures the model is deployable and performant on low-resource devices.
   (2) Offloads Compatible Operations to the Ethos™-U NPU
      a. Vela analyzes the model graph and automatically maps supported operators to the NPU.
      b. This accelerates inference significantly by leveraging hardware-level parallelism.
   (3) Falls Back to Cortex®-M CPUs with CMSIS-NN for Unsupported Ops
      a. Operations that can't be handled by the NPU are executed on the Cortex®-M CPU.
      b. These CPU-executed ops are optimized using the CMSIS-NN library provided by Arm®, ensuring high efficiency even on fallback paths.
   (4) Memory Access Optimization and Instruction Flow Optimization
      a. Vela optimizes how the model accesses memory, minimizing latency from memory bottlenecks.

b. It also refines the instruction stream for faster execution, improving overall model performance at runtime.

### 3.4.2 Vela Environment and Extensions

The detailed list of supported TensorFlow and Python versions for ethos-u-vela can be found at:

https://pypi.org/project/ethos-u-vela/

1. Runtime Environment:

ethos-u-vela runs on both Linux and Windows 10 operating systems, making it easy to deploy across major desktop platforms for model optimization tasks.

2. Embedded Deployment Capabilities:

ethos-u-vela is designed to enable efficient model compilation and optimization for embedded systems. It empowers developers to deploy deep learning models—especially those optimized for Arm® Ethos™-U NPUs—in a more efficient and hardware-accelerated way. This translates into significant performance improvements for inference at the edge.

### 3.4.3 Vela Command Overview

1. Common CLI Commands:
   (1) Compile for Specific Ethos™-U NPU:

```
$ vela --accelerator-config ethos-u55-256
helmet_quantized_.tflite
```

   Explanation: This command compiles and optimizes the model helmet_quantized_.tflite for the ethos-u55-256 configuration, ensuring it runs efficiently on that specific NPU.

   (2) Optimize for Minimal Peak SRAM Usage:

```
$ vela --optimise Size my_model.tflite
```

Explanation: This command minimizes the peak SRAM usage of my_model.tflite, which is ideal for deployment on memory-constrained embedded devices.

Optimize for Maximum Performance:

```
$ vela --optimise Performance my_model.tflite
```

Explanation: This focuses on maximizing inference speed. It is best suited for systems with adequate hardware resources where performance is a higher priority than memory footprint.

(3) Constrain Performance Optimization Within Memory Limits:

```
$ vela --optimise Performance --arena-cache-size 300000
my_model.tflite
```

Explanation: This command optimizes my_model.tflite for performance while keeping within a strict memory constraint of 300,000 bytes. It ensures the model remains performant even under limited memory availability.

### 3.4.4 Vela Compilation Results

1. Command:

```
$ vela --accelerator-config ethos-u55-256
helmet_quantized_.tflite
```

Explanation: In this command, ethos-u55-256 specifies the NPU model and its MAC configuration, while helmet_quantized_.tflite is the input model for compilation. The default optimization setting is used (focused on performance), ensuring the model is tuned for peak execution efficiency on the target NPU.

Figure 25: Output Results

2. Analysis:
   (1) 71.2% of the model's operations are offloaded to the NPU, indicating that the NPU handles the majority of the computational workload.
   (2) 28.8% of the operations are executed by the CPU. These may include operations unsupported by the NPU or non-accelerable functions.

3. Summary:

After compilation with Vela, a significant portion (71.2%) of computation is accelerated by the NPU, resulting in a major improvement in inference speed and efficiency. The remaining 28.8% is handled by the CPU, ensuring compatibility and maintaining performance where the NPU lacks support. This hybrid strategy allows for an optimal balance between speed and flexibility on embedded platforms.

### 3.4.5 YOLOX-Nano Model Fragment Visualization

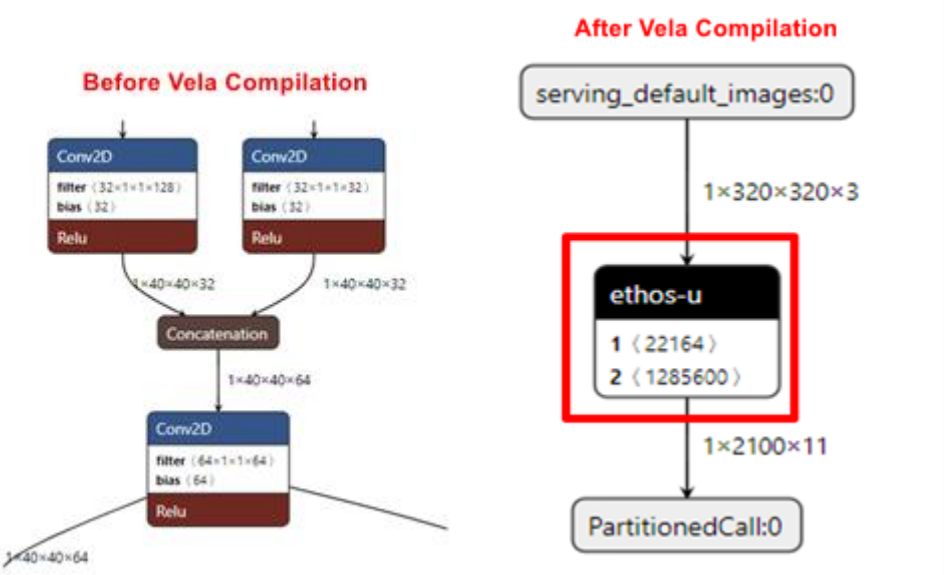


Figure 26: Visualization of Model Segments

This diagram illustrates how the computational graph of the YOLOX-Nano model changes before and after being compiled with Vela.

1. Before Compilation:

In the original version executed on the CPU, a Concatenation operation is present. This operation requires extra memory read/write steps, which negatively affect runtime performance. Performing concatenation on the CPU adds a load to SRAM (Static Random-Access Memory), slowing down inference—especially when running large models or deploying on memory-constrained embedded systems.

2. After Vela Compilation:

   Once compiled using the Arm® Ethos™-U NPU, the model structure is optimized, eliminating the need for explicit concatenation.

   The Ethos™-U NPU is better suited to handle sliced, smaller tensors, which helps minimize data shuffling and improves runtime performance.

   The NPU handles more of the computational load directly, and the optimized graph no longer relies on inefficient concatenation steps.

   As shown in the figure, the NPU can now directly process tensors without memory-heavy concatenation, reducing memory bandwidth use and speeding up inference.

3. Summary:

   By compiling the YOLOX-Nano model with Vela, previously CPU-dependent concatenation operations are removed and replaced with more efficient NPU execution.

   This significantly improves inference performance on embedded systems, reduces memory bandwidth consumption, and shortens runtime.

   Vela's optimization allows the NPU to handle more computation and reduces unnecessary memory transfers—critical for achieving high efficiency on resource-limited edge platforms.

## 3.5 Configuring Hardware and Software with C Language

The sample project for object detection, ObjectDetection_FreeRTOS_yoloxn, is available at: https://github.com/OpenNuvoton/M55M1-eBook-Sample-Code

Nuvoton will continue updating the official BSP repository, which can be found here: https://github.com/OpenNuvoton/M55M1BSP

This chapter explains the location and role of various driver configurations and files used in the object detection example (see Figure 29).

### 3.5.1 M55M1BSP

1. Document Directory (M55M1BSP/Document)
   (1) CMSIS.html: Documentation on the CMSIS directory contents.
   (2) NuMicro M55M1 Series Driver Reference Guide.chm: CHM help file explaining the driver functions in the M55M1 BSP.
   (3) Revision History.pdf: Describes the update history of the M55M1 BSP.

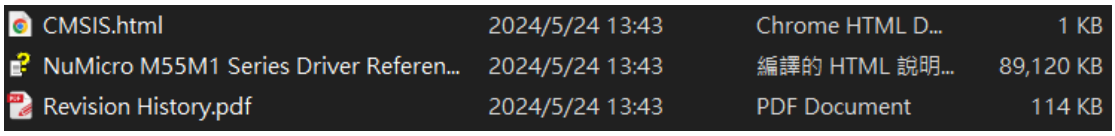


Figure 27: Document Folder Structure

2. Library Directory (M55M1BSP/Library)
   (1) CMSIS: CMSIS (Cortex® Microcontroller Software Interface Standard) related definitions and files.
   (2) Commu: Utility functions for communication protocols, such as XMODEM.
   (3) CryptoAccelerator: Source code for hardware acceleration of cryptographic functions (MbedTLS-based).
   (4) Device: Device headers following CMSIS conventions.
   (5) JpegAcceleratorLib: SIMD-optimized accelerator and headers for use with libjpeg.
   (6) PowerDeliveryLib: Power Delivery libraries and headers supporting dual-role (source and sink) functionality.
   (7) SmartcardLib: Smart card protocol support libraries and headers.
   (8) StdDriver: Source and header files for all peripheral drivers.
   (9) UsbHostLib: Source code for USB Host library support.

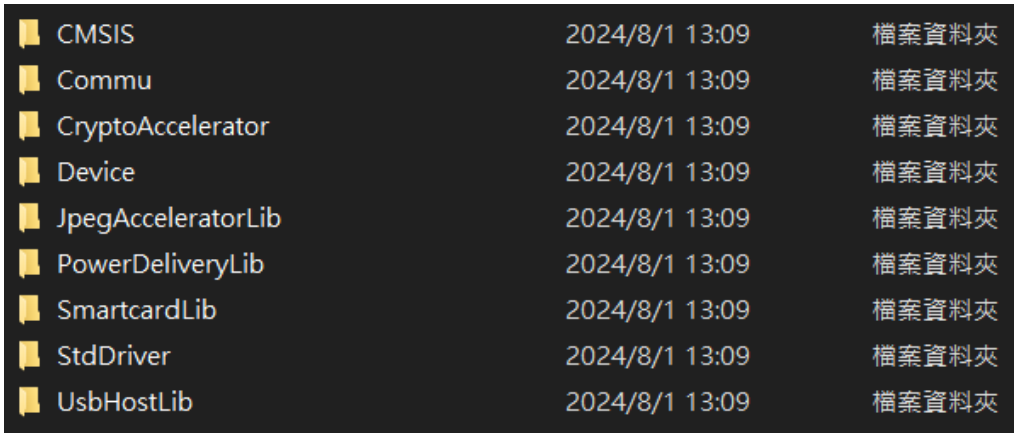


Figure 28: Library Folder Structure

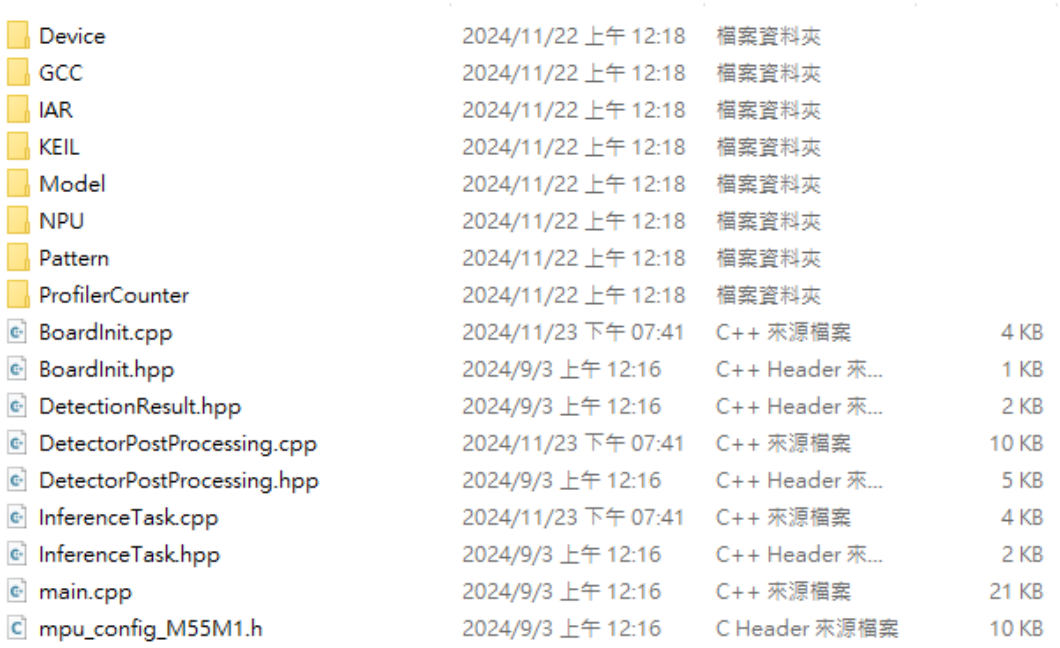


Figure 29: Folder and Source Code Overview of the Object Detection Example

### 3.5.2 Display Component Overview

- Display.c

  Responsible for the core display functionality.

  Its primary roles include:

  - Initializing the LCD hardware (including EBI interface configuration and multi-function pin setup)
  - Providing rectangle fill functions for rendering image processing results
  - Supporting character display and delay operations

Use case examples: Rendering bounding boxes for object detection results and showing system status (e.g., FPS or model name) on-screen.

- Font8_16.c

  Handles font definitions.

  - Defines 8×16 dot-matrix font data for LCD character display
  - Provides character sets in array format
  - Works in conjunction with Display_PutText() to output text to the screen

- LCD_FSA506.c

  Implements the LCD driver logic.

  - Sends commands and data to the LCD over the EBI interface
  - Controls column and page address setup (e.g., fsa506_set_column())
  - Initialization includes:
    - Hardware reset
    - Panel size configuration
    - Clock format setup
    - TFT output timing configuration

- LCD.h

  Header file defining the LCD driver interface.

  - Declares structures and configurations related to LCD functionality
  - Provides the S_LCD_INFO structure, which includes:
    - Initialization routines
    - Function pointers for display operations

### 3.5.3 HyperRAM Component Overview:

- hyperram_code.c

  This file implements the core functionality for HyperRAM operations, including:

- Erase
- Data write
- Delay calibration
- Optional PLL calibration for tuning read timing

- Function Details:
  - Erase Logic:

    Uses HyperRAM_Erase() to iteratively clear blocks of memory, ensuring the erased data is set to zero.

  - PLL Calibration:

    Performs delay step adjustment to ensure reliable data transfer by tuning internal read timing.

  - Default Configuration:

    Sets up chip select behavior and read/write access timing for proper HyperRAM interface operation.

### 3.5.4 ImageSensor Component Overview:

- ImageSensor.c

  This is the main module for handling the image sensor. Its core functions include:

  - Initializing the image sensor and the CCAP (Camera Capture) module
  - Configuring image output formats such as YUV422 or RGB565
  - Implementing image capture and cropping features
  - Acquiring input images from the sensor for inference use

### 3.5.5 KEIL Configuration Analysis:

- M55M1.scatter

  This is the memory layout configuration file, used to define the program's memory allocation—covering code segments, data sections, and stack regions.

  It also defines Flash and RAM partitioning, as shown in Figure 30.

Its primary purpose is to guide the Linker on how to place the program in the target hardware's memory.

```
131  APP_IMAGE FLASH_START FLASH_SIZE
132  {
133      ; Flash 2 MB
134      rom_exec FLASH_START FLASH_SIZE
135      {
136          *.o (RESET, +First)
137          * (InRoot$$Sections)
138          ; Make sure reset_handler ends up in root segment
139          startup_M55M1.o
140          .ANY (+RO)
141          ..\..\..\..\ThirdParty\tflite_micro\Lib\tflu.lib (.init_array)
142      }
143      rom_netwrok_executor +0
144      {
145          ..\..\..\..\ThirdParty\tflite_micro\Lib\tflu.lib (+RO)
146          ethosu_*.o
147      }
```

Figure 30: Code defining Flash and RAM partitioning in M55M1.scatter

HyperRAM:

- A type of external high-speed RAM used for:
  - Storing intermediate computation results
  - Temporary data
- Commonly used in:
  - Neural network processing
  - Graphics rendering
- Advantages:
  - Large capacity (32MB or more)
  - Cost-effective via SPI interface
- Disadvantages:
  - Slower read speed compared to internal SRAM
- See Figure 31 for HyperRAM address definition in the scatter file.

```
187  #if 0
188      HYPERRAM SPIM0_START UNINIT SPIM0_SIZE
189      {
190          ; Place tensor arena in SRAM if we do not have a fast memory area
191          .ANY (.bss.NoInit.activation_buf_sram)
192      }
193
194
195      HYPERRAM SPIM1_START UNINIT SPIM1_SIZE
196      {
197          ; Place tensor arena in SRAM if we do not have a fast memory area
198          .ANY (.bss.NoInit.activation_buf_sram)
199      }
200  #endif
```

Figure 31: Code defining HyperRAM address in M55M1.scatter

HyperFlash:

- A type of high-speed external Flash used for:
  - Storing program code
  - Read-only data (e.g., images, audio)
- Common use: Firmware or static data storage
- Advantages:
  - Large capacity (64MB or more)
  - Fast loading via SPI interface
- Disadvantages:
  - Slow write speed
- See Figure 32 for HyperFlash address definition in the scatter file.

```
94    #define FLASH_START      0x00100000
95    #define FLASH_SIZE       0x00200000
```

Figure 32: Code defining HyperFlash address in M55M1.scatter

SRAM:

- A volatile memory—contents are lost when power is removed
- Main advantage:
  - Fastest access speed among memory types
  - No need for refresh cycles like DRAM
- Main limitation:
  - Low storage capacity
- See Figure 33 for the SRAM address mapping code in the scatter file.

```
180     SRAM012 SRAM012_START SRAM012_SIZE
181     {
182         .ANY1 (.bss.sram.data)
183         ; Place tensor arena in SRAM if we do not have a fast memory area
184         .ANY (.bss.NoInit.activation_buf_sram)
185     }
```

Figure 33: Code defining SRAM address in M55M1.scatter

### 3.5.6 MODEL Folder Overview and Analysis:

- Label Module

Demonstrates how labels are loaded from a static array into a dynamic container.

An illustration shows the label structure and how it's applied during inference.

- Labels.cpp:

  Contains the list of labels used by the object detection model, enabling human-readable classification of inference results.

- YOLO Model Management

  Explains how model data is integrated into the inference process, including a diagram illustrating model initialization and runtime support.

  - YoloXnanoNu.cpp:

    Encapsulates the YOLOX-Nano inference model, including all setup and execution routines.

  - yolox_nano_*.tflite.cc:

    These files contain quantized YOLOX-Nano model data in .tflite format, designed for execution on NPU (Neural Processing Unit) environments.

- Integration of Labels with Inference

  Details how the label module integrates with the YOLO model to generate human-readable inference results.

  - Labels.hpp:

    Defines the interface for external access to the label module, supporting label retrieval and management during object detection.

### 3.5.7 NPU Folder Overview

- CPU Cache Management Module

  Describes the functionality of ethosu_cpu_cache, which ensures data consistency between the NPU and CPU during shared memory operations.

- NPU Initialization Module

  Details the initialization steps for ethosu_npu_init, with diagrams showing interrupt handling and memory configuration during NPU startup.

- Performance Profiling Module

  Uses visual charts to show how performance counters (e.g., idle cycles, data traffic) capture the runtime state of the NPU.

  Also explains how ethosu_profiler supports performance optimization.

- Memory Configuration Module

  Includes diagrams illustrating memory partition structures, showing how cache and buffer regions are laid out.

  Highlights their impact on performance and data access efficiency.

## 3.6 Program Development and Flashing on the Development Board

### 3.6.1 Integrated Development Environment – Arm® Keil MDK µVision 5

Arm® Keil MDK µVision 5 is an integrated development environment (IDE) specifically designed for embedded development, particularly for applications based on Arm® Cortex®-M microcontrollers.

1. Software Features
   (1) Integrated Workflow: µVision offers a full development flow—from editing and compiling to debugging—all within a single interface.
   (2) C/C++ Compiler: Utilizes the high-performance Arm® Compiler to generate optimized code, reducing binary size and enhancing execution efficiency.
   (3) Rich Debugging Tools:

   Supports real-time tracing, execution analysis, and hardware debugging. Includes a built-in simulator for testing code without physical hardware.

2. Available Editions
   (1) Community Edition: Free for developers, suitable for learning or small-scale project development.
   (2) Keil MDK Nuvoton Edition: Provided free by Nuvoton Technology for commercial development.

   (Learn more: Keil MDK Nuvoton Edition – Full Cortex®-M – Nuvoton)

3. Main Application Areas
   (1) Internet of Things (IoT): Development of low-power, high-performance IoT devices.
   (2) Consumer Electronics: Such as appliance controllers and handheld devices.
   (3) Industrial Control: Embedded systems for machine control and monitoring.
   (4) Medical Electronics: Includes wearable and monitoring devices.

4. Feature Overview
   (1) Device Tree Support: Simplifies chip and peripheral configuration and auto-generates config code.
   (2) Pack Management: Allows developers to download and manage Device Family Packs (DFP) for various microcontrollers.

   RTOS Support: Integrates Keil RTX Real-Time Operating System, enabling efficient multitasking application development.

### 3.6.2 Development Workflow

1. Create a New Project

   Before flashing, you need to create a new project in Keil µVision. Follow these steps:

   (1) Launch Keil µVision

   Open the Keil µVision IDE. Ensure it's properly installed and the correct version.

   (2) Open the “Project” Menu

   Click the Project tab from the top menu bar.

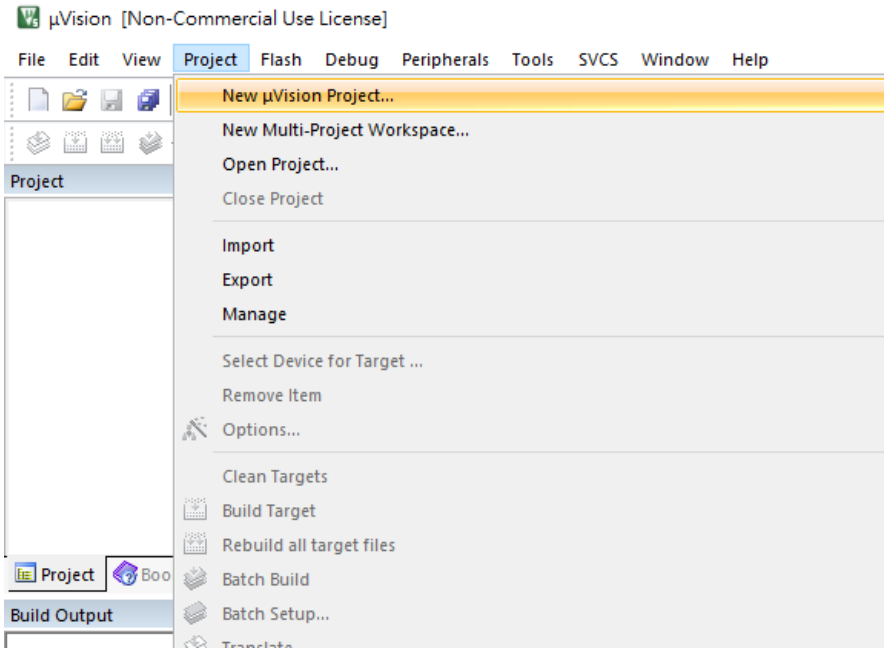


Figure 34: Open Project Menu

(3) Select “New µVision Project”

From the dropdown, select New µVision Project to start creating a new project.

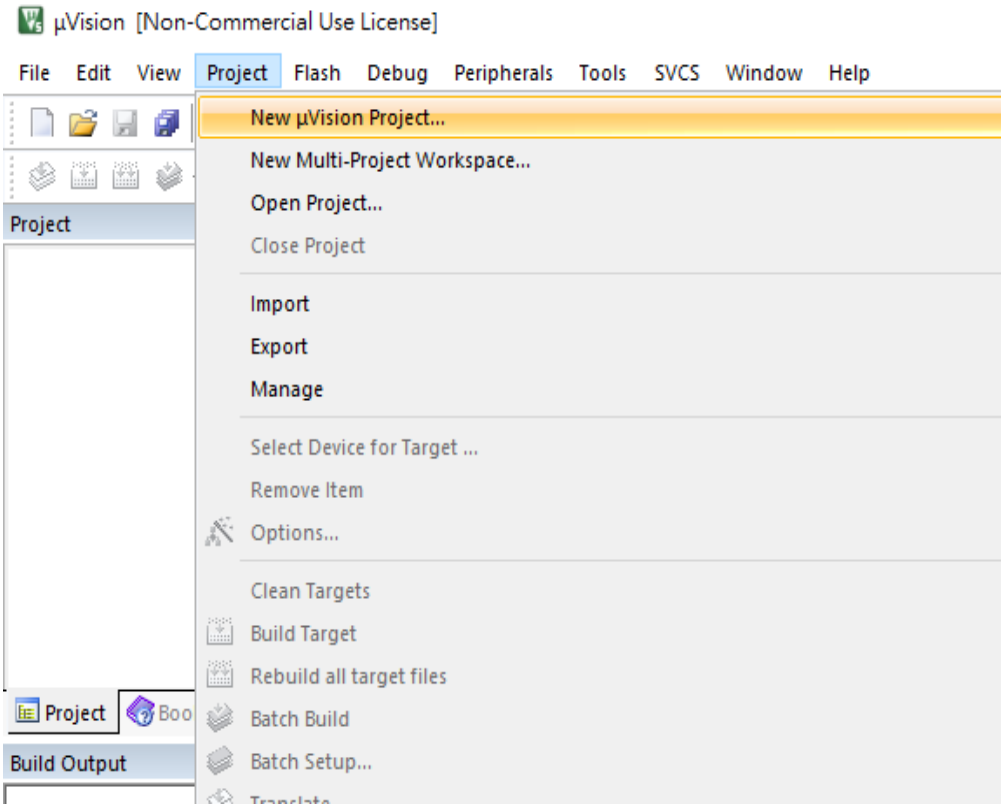

Figure 35: Select New uVision Project

(4) Name Your Project and Set Save Path

In the dialog box that pops up, give your project a name and choose a meaningful directory path to save it.

(5) Select Target Hardware

After naming, select the appropriate MCU or processor from the device list, e.g., Arm® Cortex®-M55.

2. Project Naming

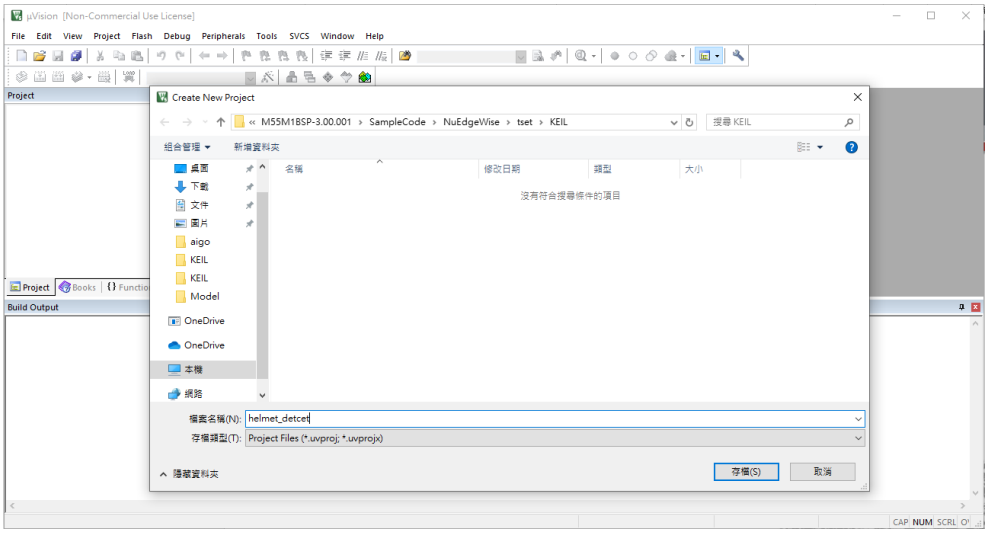


Figure 36: Naming project

(1) Enter the Project Name

In the "Create New Project" dialog box, fill in the File Name field.

(2) Choose Save Location

Navigate to your preferred directory to store the project files.

(3) Confirm and Save

Click Save to complete naming and storing the new project.

(4) Select MCU Model

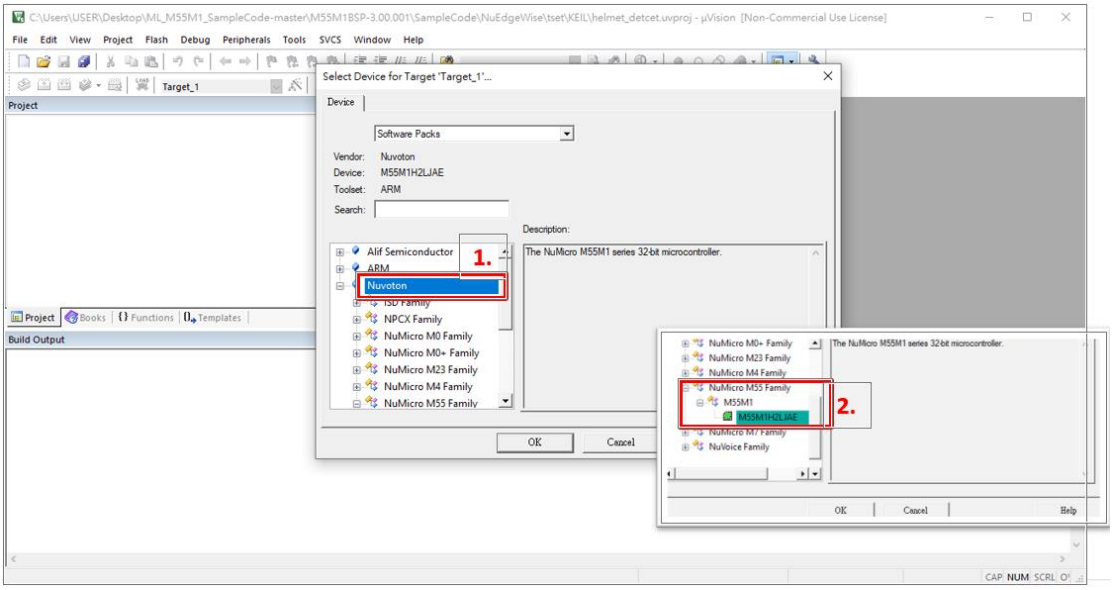


Figure 37: Select Microcontroller (select M55M1)

After saving, Keil will open the Select Device for Target dialog.

(5) Expand Nuvoton from the list.
(6) Go to NuMicro M55 Family, then select M55M1H2LJAE.
(7) Click OK to confirm.

3. Select and Configure Software Components

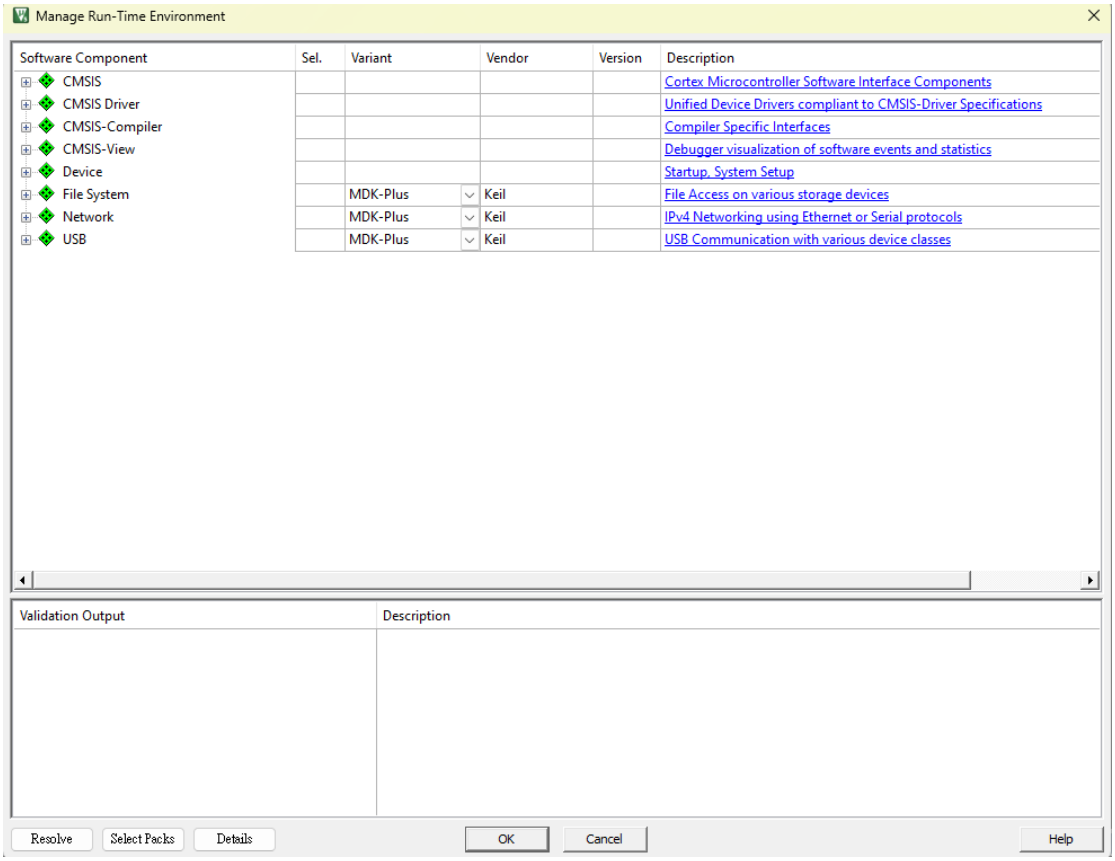


Figure 38: Select and configure required packages on the project

(1) Open “Manage Run-Time Environment”

Go to the Project menu > Manage Run-Time Environment.

(2) Choose Required Components

Select components like:

- CMSIS: Basic MCU support.
- Compiler: Arm® compiler.
- RTOS (FreeRTOS): For multitasking support.
- USB/Network: If USB or networking is needed.

(3) Review Component Details

On the right panel, check version, vendor, and feature description to ensure compatibility.

(4) Click OK to Apply

After configuration, click OK to save and load the components.

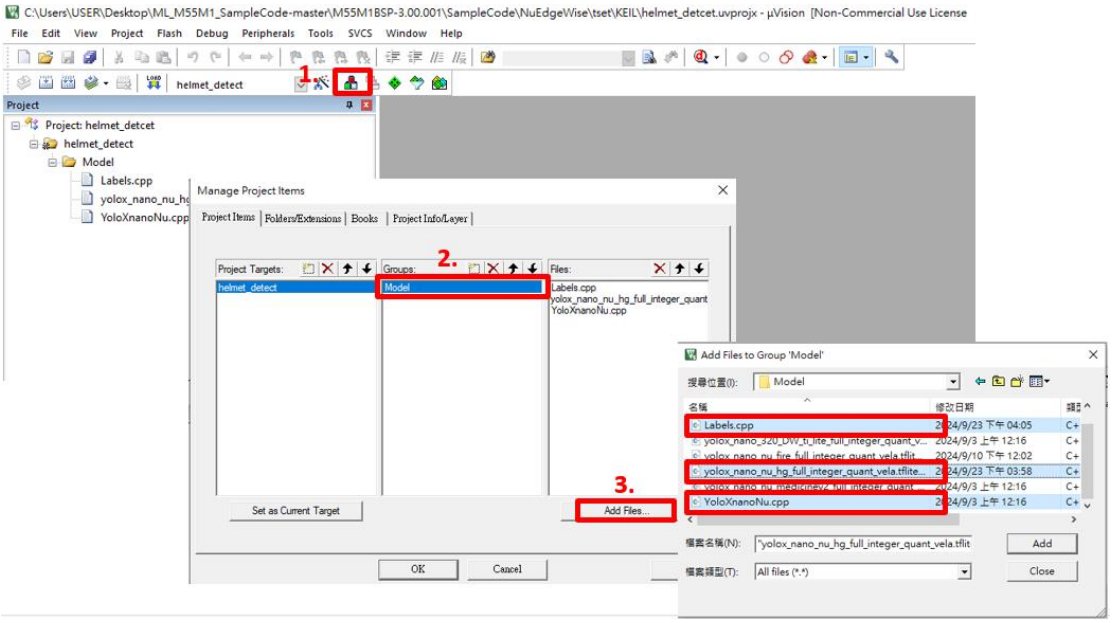


Figure 39: Add model, hardware configuration to the project

4. Add Files to the Project

(1) Open “Add Files” Dialog

In the project pane, click Add Files to open the file manager.

(2) Choose File Group

Select or create a Group (e.g., Model, Drivers) for better organization.

(3) Add Files to Group

Click Add Files...

Select required files such as:

- Labels.cpp (label definitions)
- YOLO model files (e.g., yolox_nano_*.tflite.cc)
- Other source files (e.g., YoloXNanoNu.cpp)

Click Add > OK

5. Verify Project Structure

Ensure the files appear in the project tree and compile correctly.

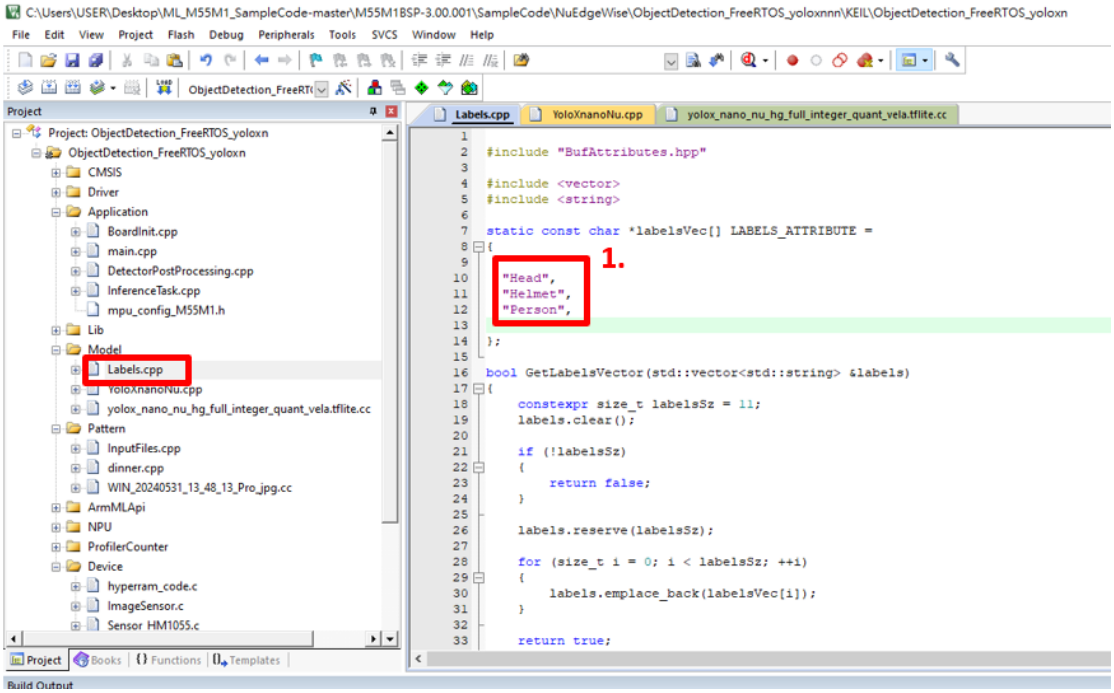


Figure 40: If project has been created, just rename labels

6. Modify Existing Project Labels (Optional)

   Locate Labels.cpp

   In the project tree: Lib > Model > Labels.cpp

7. Edit Label Array

   Modify the labelsVec[] array with your custom labels.

```
static const char *labelsVec[] LABELS_ATTRIBUTE = {
    "Head",
    "Helmet",
    "Person"
};
```

Figure 41: The array defined for labels

8. Save and Rebuild

   Click Save, then Build to apply changes.

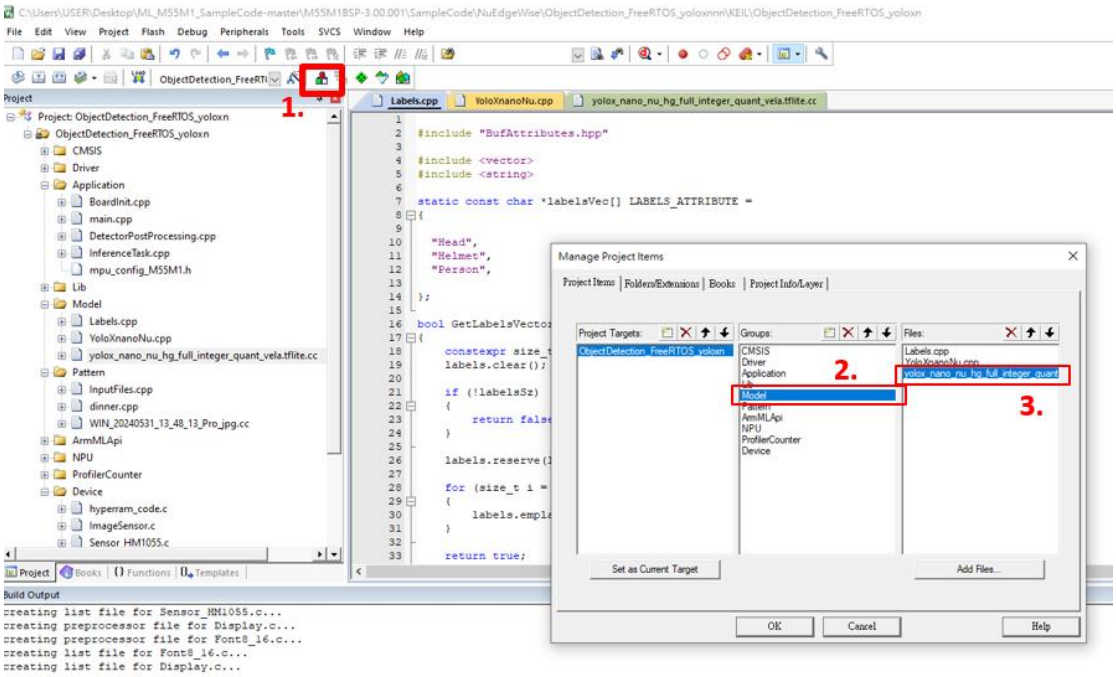


Figure 42: Custom Weights and Config Files

9. Add Custom Weights and Config Files
   (1) Open Manage Project Items
   Click Manage Project Items in the toolbar.
   (2) Create New Group (if needed)
   Click Add Group to organize new files.

Custom model weights (e.g., yolox_nano_nu_hg_full_integer_quant)

Hardware config files (e.g., audio drivers, sensor configs)

10. Verify Structure

    Ensure all new files appear and are structured properly in the project tree.

### 3.6.3 Compile and Flash the Program to the Board

1. Build the Project
   (1) Click Build Target to compile the project.
   (2) Check the Build Output pane: 0 Error(s), 0 Warning(s) confirms success.

2. Generate Flashable Binary

The build produces .bin or .hex files (e.g., .\release\ObjectDetection_FreeRTOS_yoloxn.bin)

3. Flash the Binary to the Board
   (1) Click Flash Download
   (2) Ensure your board is connected and Nu-Link selected
   (3) Click Program to flash the binary
   (4) DO NOT disconnect the board during flashing

4. Verify Execution
   (1) After flashing, the board auto-executes the program
   (2) Use Keil's Debug or serial tools to check runtime behavior

# 1 Smart Factory 1 – Safety Helmet Detection

## 1.1 Use Case Overview – Safety Helmet Wearing Detection

This use case utilizes the safety-helmet-dataset from the open-source Roboflow dataset collection. The YOLOX-Nano model is trained under the PyTorch framework. After training, the model is converted to the TensorFlow Lite format—compatible with the development board—via ONNX. To meet the model size constraints of the board, Full-INT8 quantization is applied for model optimization.

The model is further optimized using the Vela compiler, which transfers all operations originally assigned to the CPU to the NPU (Neural Processing Unit) for execution. Finally, the optimized system is programmed into the Nuvoton M55M1 development board using the Keil toolchain, successfully achieving a real-time, efficient, and power-saving safety helmet detection system.

Figure: Safety Helmet Wearing Detection

Introduction to Nuvoton Development Board and Practical Applications

1. High-Efficiency Hardware Performance
   (1) Cortex®-M55 and Ethos™-U55 microNPU

a. Capable of achieving processing speeds up to 220 MHz, enabling neural network inference.
b. Support for Full-INT8 quantization:

   After applying Full-INT8 model quantization, computation efficiency is significantly improved. At the same time, memory usage and computational resource requirements are greatly reduced, enabling more efficient real-time predictions.

2. Real-Time Performance and Low Latency
   (1) With the hardware acceleration of Ethos™-U55 and the high throughput of Cortex®-M55, the system achieves an excellent balance between real-time performance and detection accuracy.

3. Rich Expansion Interfaces
   (1) The M55M1 board provides multiple interfaces, such as the CMOS sensor interface (CCAP) and TFT-LCD display interface, allowing integration of cameras and displays for image capture and on-device inference.

4. Practical Applications in Safety Helmet Detection
   (1) Factory Scenario:

   In a factory environment, there are numerous operating machines. Wearing a safety helmet protects workers' heads from falling or flying objects and prevents injuries from protruding or moving equipment. Therefore, an inspection checkpoint is required at the factory entrance to ensure workers are wearing helmets as per safety regulations.


Figure: Person not wearing a safety helmet – Paula Bronstein/Getty Images

5. Project Objectives

This project aims to build a high-performance, real-time safety helmet detection system using the Nuvoton NuMaker-X-M55M1 development board. It leverages deep learning models to accurately detect helmet-wearing status and trigger alarms.

(1) Rapid Detection of Violations:

Use machine learning to perform real-time checks on individuals entering factory zones, ensuring those without helmets cannot access hazardous areas.

(2) High-Accuracy Recognition:

Through optimized models and hardware acceleration, the system delivers high-precision helmet detection, reducing both false alarms and missed detections.

(3) Low Power Consumption and Stability:

The low-power design of the M55M1 development board ensures long-term, stable operation, suitable for 24/7 factory surveillance.

(4) Easy Deployment and Scalability:

The system is designed for practical industrial deployment and can integrate with other safety monitoring systems.

6. Project Results

Using the Nuvoton M55M1 development board, the safety helmet detection system achieved the following results:

(1) Real-Time Helmet Detection:

The system can accurately identify helmet-wearing status and display a “No Helmet” alert to immediately notify on-site managers.

(2) High Detection Accuracy:

In current tests with 500 test images, the system achieved an overall accuracy of 95%.

(3) Visual Display:

Detection results, including personnel images and helmet status, are presented in real time via the M55M1’s display module.

(4) Optimized Model Performance:

By combining TinyML with Full-INT8 compression techniques, the system effectively reduces computational load and enhances model efficiency on edge devices.

## 1.2 Dataset and AI Model Training

Dataset Introduction: Safety Helmet Detection Dataset (safety-helmet-dataset from Roboflow)

1. Dataset Preparation
   (1) Data Collection
   a. This dataset originates from the "safety-helmet-dataset" on the Roboflow platform. It is specifically designed for helmet-wearing detection tasks and includes classifications such as wearing a helmet and not wearing a helmet.
   b. The data sources may include surveillance video recordings from factories or images captured in simulated environments to ensure diverse scenes and lighting conditions.
   c. Steps to Obtain the Dataset from Roboflow:

Step 1: Visit the Roboflow official website: https://roboflow.com

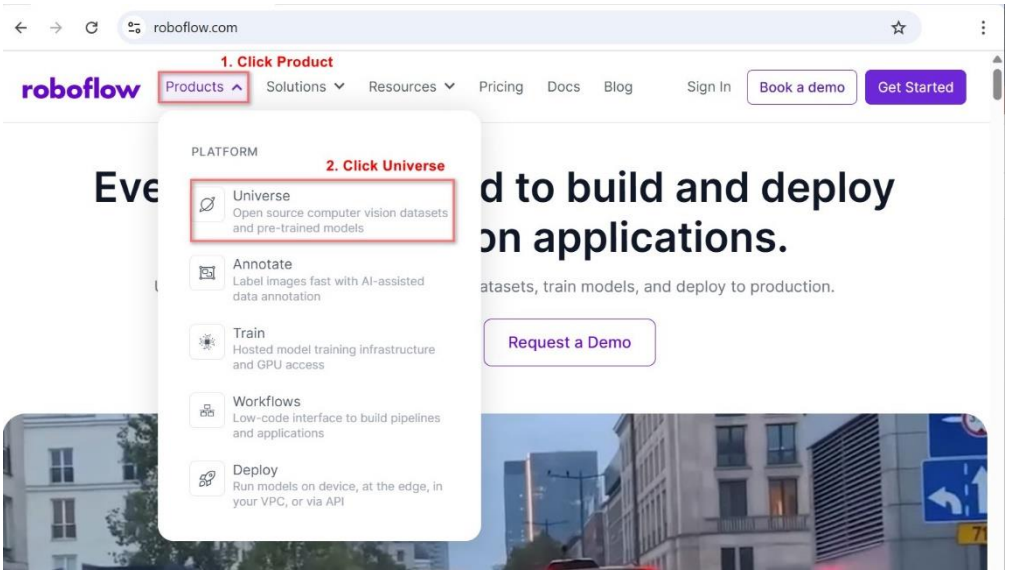


Figure: Roboflow homepage

Step 2: Click to enter the open dataset library

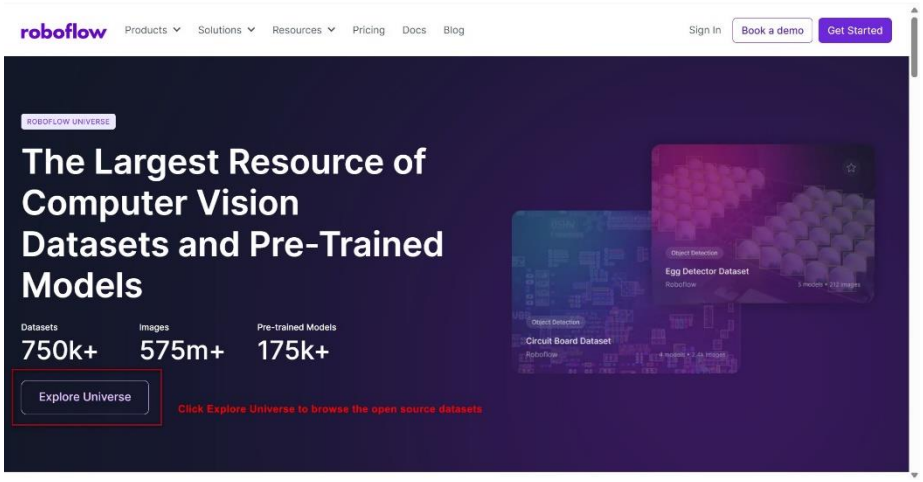


Figure: Dataset library

Step 3: Search for the desired dataset

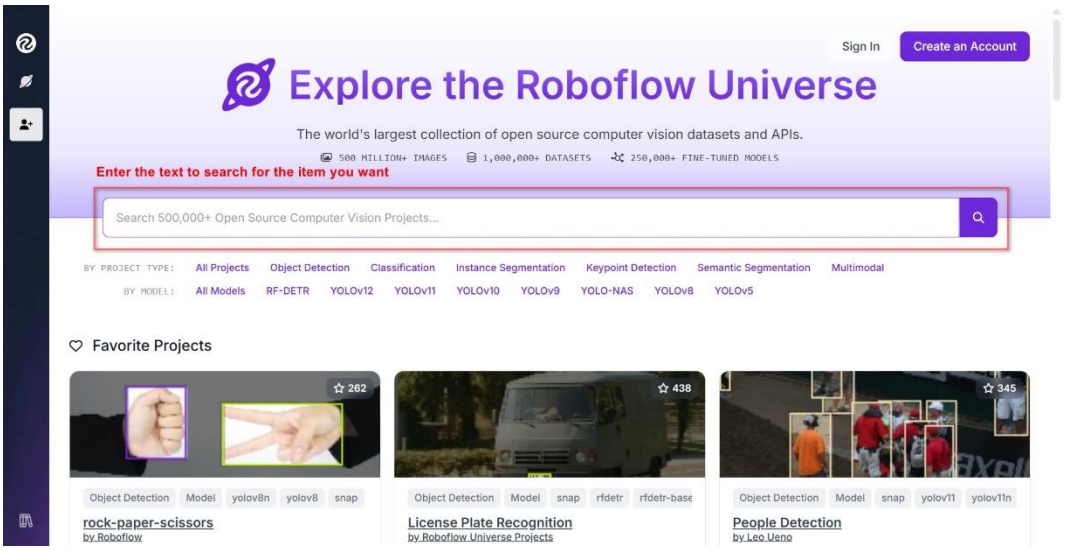


Figure: Dataset search interface

Step 4: After selecting a project, click "Download Project" to obtain the dataset


Figure: Project download page

Step 5: Choose the desired annotation format to download the dataset

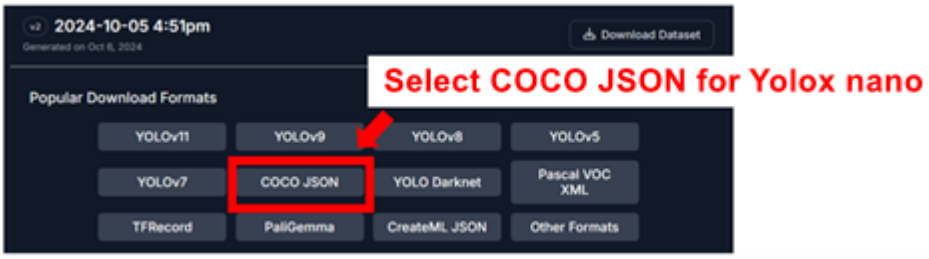


Figure: Format selection interface

d. Preprocessing includes converting BGR to RGB, resizing images, and normalizing image data to a floating-point range of 0 to 1.
e. The training and validation datasets for the YOLOX model must comply with the COCO JSON format.

(2) Data Format

a. The dataset uses the COCO JSON format, which includes the following features:
   (a) Annotation Standardization: Each image's labeled objects (e.g., helmet presence) are marked using bounding boxes. The annotation data is stored in JSON files, making it easy to load with training frameworks.
   (b) Extensibility: Supports multi-class annotation and complex scene descriptions.

b. The dataset is divided into the following subsets:
   (a) Training Set: 5145 images

   Used for primary model training and learning.

   (b) Validation Set: 1470 images

   Used to evaluate model performance during training and avoid overfitting.

   (c) Test Set: 735 images

   Independent from the training and validation sets, used for final evaluation.

   (d) Dataset Link: https://universe.roboflow.com/realdatasetfinish/safety-helmet-dataset-ep6zc/dataset/1

(3) Dataset and Training Environment Setup

a. Install and Set Up Anaconda Environment

(a) Anaconda is a widely used open-source platform for Python and R programming, ideal for machine learning, data analysis, and application deployment. It's known as an “all-in-one solution” for data science.

(b) Developers can conveniently create virtual environments within Anaconda to isolate project dependencies.

(c) Installing Anaconda is simple. Visit the official site and navigate to Products > Distribution.

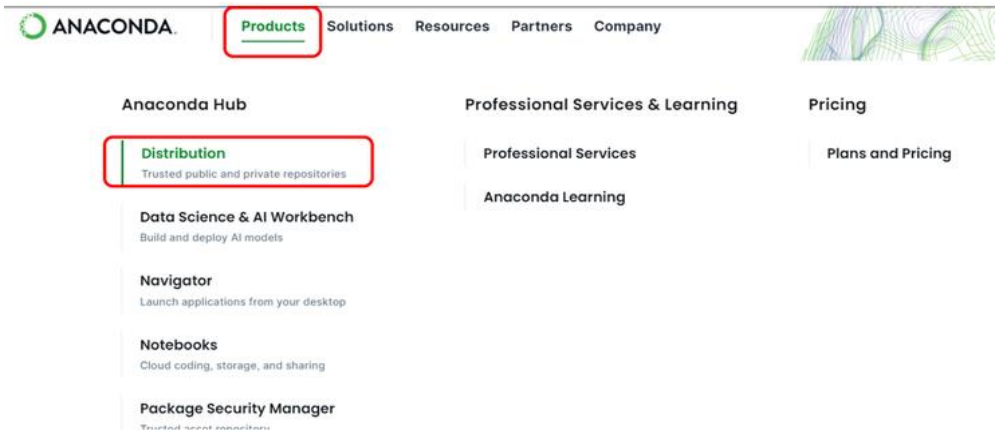


Figure: Anaconda homepage

(d) Click “Skip registration” to proceed without account setup

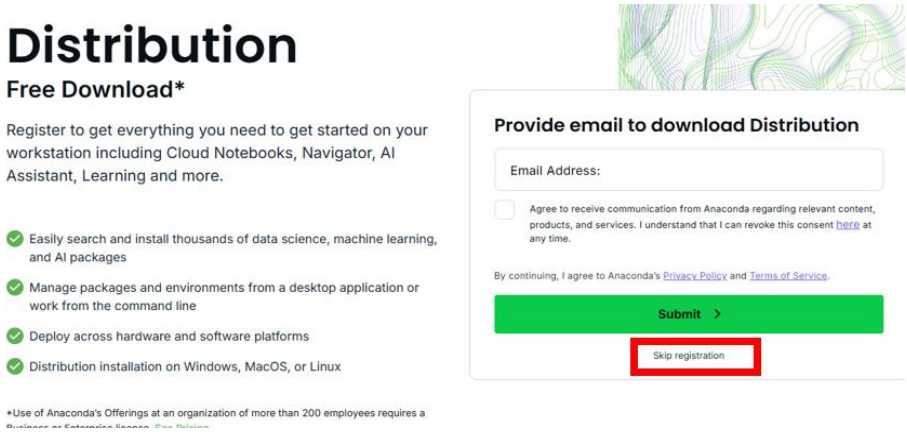


Figure: Skip registration

(e) Choose the version that matches your operating system. For this case, download the Windows version.

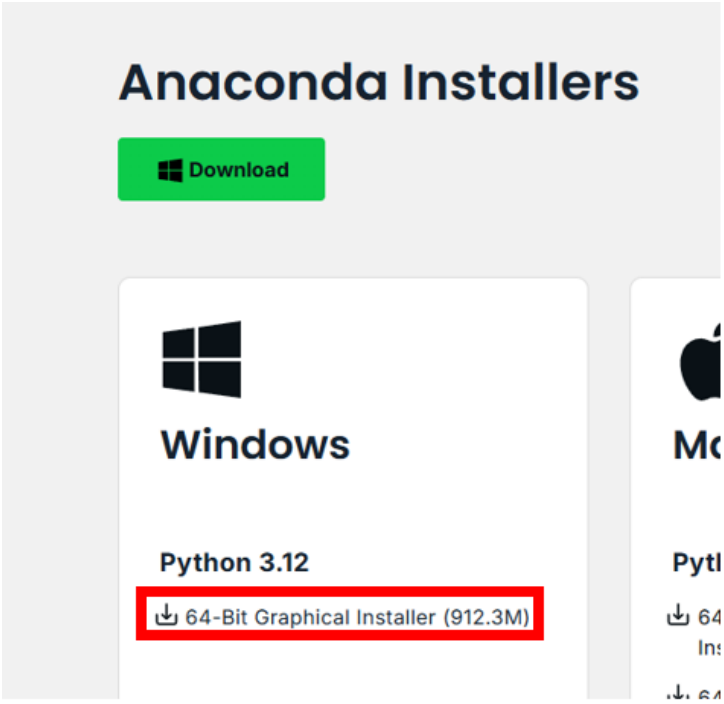


Figure: Windows installer

b. Steps to Set Up the Anaconda Environment

(a) Download YOLOX-TFLITE Project

GitHub repository:
https://github.com/OpenNuvoton/M55M1-eBook-Sample-Code

(b) Create Python Environment

- `conda create --name yolox_nu python=3.10`
- `conda activate yolox_nu`

(c) Upgrade pip and setuptools

- `python -m pip install --upgrade pip setuptools`

(d) Install CUDA, PyTorch, and MMCV

This project uses CUDA 11.8 and Torch 2.0

- `python -m pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`

(e) Install MMCV based on your hardware setup (version 2.0.1)

- `python -m pip install mmcv==2.0.1 -f https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/index.html`

(f) Install Other Required Packages

Open the requirements.txt in the downloaded directory.

Run the following to install all listed packages:

- `python -m pip install --no-input -r requirements.txt`

(g) Install YOLOX

After installing all dependencies, run:

- `python setup.py develop`

## 1.3 Model Training Using PyTorch Framework on PC with Anaconda

1. Model Selection
   (1) Choose a lightweight model suitable for the available hardware resources. This project selects YOLOX Nano due to its compact model size and excellent performance.
   (2) The model framework uses PyTorch, which supports flexible adjustment of network architecture and parameters.

2. Training Process
   (1) Split the dataset into training and validation sets: 80% for training, 20% for validation.
   (2) Use Adaptive Learning Rate to minimize overfitting.
   (3) Apply data augmentation strategies, such as image scaling, cropping, and enhancements (e.g., rotation, brightness adjustment), to improve generalization.
   (4) Utilize hardware (e.g., GPU) to accelerate the training process and enhance efficiency.

3. Training Environment
   (1) Hardware: NVIDIA GeForce RTX 3070 Ti
   (2) CUDA Version: 11.8
   (3) PyTorch Version: 2.0.0
   (4) Training Data: 5145 images
   (5) Training Parameters: Epoch = 100, Batch size = 32
   (6) Training Duration: Approximately 1.5 hours

4. Training Using Pretrained Model
   (1) Configuration File: exps/default/yolox_nano_ti_lite_nu.py
   (2) Prepare your training data and organize it as follows in the dataset directory:

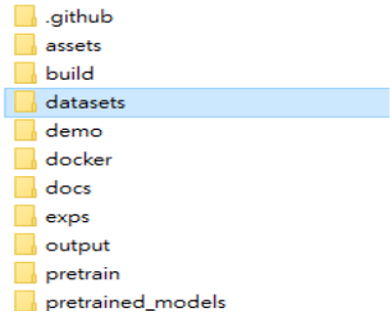


Figure: Dataset directory structure

(3) Dataset format requirements:

```
datasets/<dataset_name>
        ├── annotations
        │     ├── <train_annotation_json_file>
        │     └── <val_annotation_json_file>
        ├── train2017
        └── val2017
```

Figure: Dataset format

(4) Configure dataset path:

```
# Define yourself dataset path
self.data_dir = "datasets/COCO"
self.train_ann = "train_annotation_json_file.json"
self.val_ann = "val_annotation_json_file.json"
```

Figure: dataset path setup

(5) Modify parameters in yolox_nano_ti_lite_nu.py:

Set Classes number based on dataset

```
# ---------------- model config ---------------- #
self.num_classes = 4
self.depth = 0.33
self.width = 0.25
self.input_size = (320, 320)
self.random_size = (10, 20)
self.mosaic_scale = (0.5, 1.5)
self.mosaic_prob = 0.5
self.enable_mixup = False
self.exp_name = os.path.split(os.path.realpath(__file__))[1].split(".")[0]
self.act = "relu"
```

```
# ---------------- training config ---------------------- #
self.warmup_epochs = 3
self.max_epoch = 100
# ----------------- testing config ------------------ #
self.test_size = (320, 320)
```

Set epoch to 100

Figure: parameter settings

(6) Training command:

```
python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
<BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
```

Example:

```
python tools/train.py -f
exps/default/yolox_nano_ti_lite_nu.py -d 1 -b 32 --fp16 -
o -c
retrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti_
lite.pth
```

Parameter Explanation:

- -f: Specifies the model config file
- -d: Device count (e.g., GPU count)
- -b: Batch size
- --fp16: Enables mixed-precision (half float) training
- -o: Enables automatic mixed-precision
- -c: Path to pretrained model

5. Conversion Tools

(1) Use ONNX (Open Neural Network Exchange) to convert the framework, allowing deployment across diverse environments.

(2) Convert PyTorch model to ONNX with:

```
python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c
<TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
```

Example:

```
python tools/export_onnx.py -f
exps/default/yolox_nano_ti_lite_nu.py -c
YOLOX_outputs/yolox_nano_ti_lite_nu/best_ckpt.pth --
output-name YOLOX_outputs/ONNX/yolox_nano_nu_helmet.onnx
```

Parameter Explanation:

- <MODEL_CONFIG_FILE>: Model config file
- <TRAINED_PYTORCH_MODEL>: Trained PyTorch checkpoint
- <ONNX_MODEL_PATH>: Output ONNX model file path

6. Model Quantization

(1) Quantization is used to compress model parameters (e.g., float → int) to improve inference efficiency, crucial for resource-constrained edge devices. This project uses Full-Integer 8 (INT8) quantization.

(2) Generate Calibration Data:

```
python demo/TFLite/generate_calib_data.py --img-size
<IMG_SIZE> --n-img <NUMBER_IMG> -o <CALI_DATA_FILE> --
img-dir <TRAIN_IMAGE_DIR>
```

Example:

```
python demo/TFLite/generate_calib_data.py --img-size 320
320 --n-img 1400 -o
YOLOX_outputs/yolox_nano_ti_lite_nu/calib_data_320x320_n2
00.npy --img-dir datasets/COCO/train2017
```

Parameter Explanation:

- <IMG_SIZE>: e.g., 320x320
- <NUMBER_IMG>: Number of images used for calibration
- <CALI_DATA_FILE>: Output .npy file path
- <TRAIN_IMAGE_DIR>: Training image folder

(3) Convert ONNX to TFLite:

```
onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
<CALI_DATA_NPY_FILE> "[[[[0,0,0]]]]" "[[[[1,1,1]]]]"
```

Example:

```
onnx2tf -i YOLOX_outputs/ONNX/yolox_nano_nu_helmet.onnx -
oiqt -qcind images
YOLOX_outputs/yolox_nano_ti_lite_nu/calib_data_320x320_n2
00.npy "[[[[0,0,0]]]]" "[[[[1,1,1]]]]"
```

Flags Explanation:

- -oiqt: Full-INT8 quantization
- -qcind images: Use image-based calibration data
- [[[[0,0,0]]]] and [[[[1,1,1]]]]: Min/max tensor values

7. Vela Compiler Workflow

Step 1: Edit variables.bat (in vela/ folder)

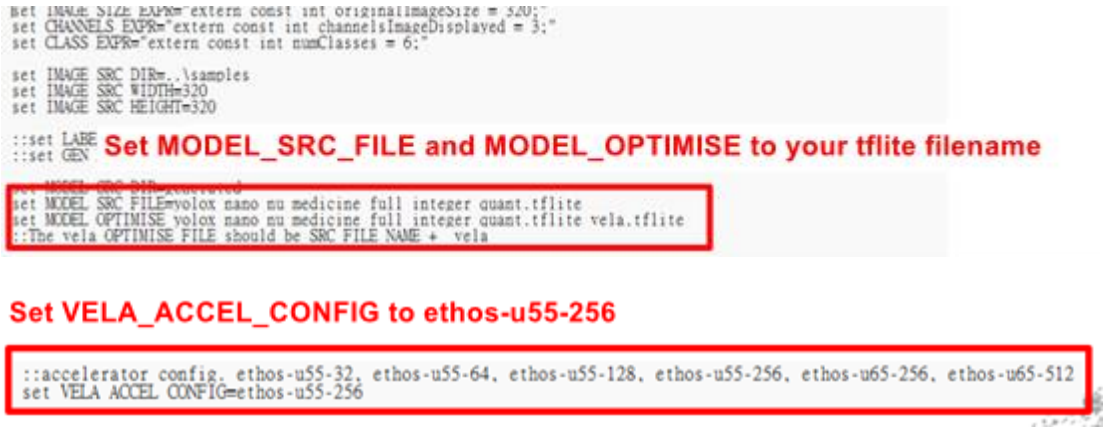


Figure: Modify parameters in variables.bat file

Set MODEL_SRC_FILE=<your tflite modem>

Example:

```
set
MODEL_SRC_FILE=yolox_nano_nu_helmet_full_integer_quant.tf
lite
```

Set MODEL_OPTIMISE_FILE=<output vela model>

Example:

```
set
MODEL_OPTIMISE_FILE=yolox_nano_nu_helmet_full_integer_qua
nt_vela.tflite
```

Run it in Anaconda Terminal:

call variables.bat

Step 2: Run gen_model_cpp.bat to generate Vela-compiled .cc file:

Output: yolox_nano_nu_helmet_full_integer_quant_vela.tflite.cc

Path: vela/generated/

This .cc file includes pretrained model weights and should be placed in Sample Code/Model directory.

The weight .cc file will be flashed along with the firmware.

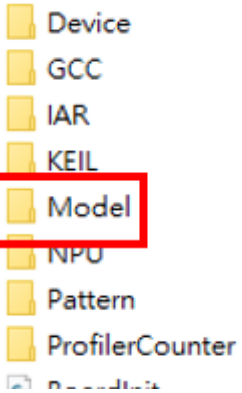


Figure: Location of Model Weights File

```
{

static const uint8_t nn_model[] MODEL_TFLITE_ATTRIBUTE =
{
    0x24, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00,
    0x00, 0x00, 0x04, 0x00, 0x12, 0x00, 0x00, 0x00, 0x18, 0x00,
    0x02, 0x00, 0x00, 0x00, 0x38, 0x00, 0x00, 0x00, 0x04, 0x00,
    0x69, 0x6e, 0x65, 0x4d, 0x65, 0x6d, 0x6f, 0x72, 0x79, 0x41,
    0x06, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x13, 0x00,
    0x08, 0x00, 0x00, 0x00, 0x00, 0x49, 0x08, 0x00, 0xf0, 0x48,
    0x04, 0x00, 0x00, 0x00, 0x26, 0xb7, 0xf7, 0xff, 0x04, 0x00,
    0x20, 0x38, 0x01, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xff, 0xff, 0x20, 0xaf, 0x02, 0x00, 0x00, 0x40,
    0x32, 0x2e, 0x33, 0x2e, 0x30, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x14, 0x00, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x43, 0x4f, 0x50, 0x31, 0x01, 0x00, 0x10, 0x00, 0x08, 0x30,
    0x0f, 0x01, 0x01, 0x00, 0x00, 0x40, 0x00, 0x00, 0x20, 0xaf,
    0x00, 0x00, 0x00, 0x00, 0x0b, 0x01, 0x13, 0x00, 0x0c, 0x01,
    0x80, 0x07, 0x00, 0x00, 0x04, 0x40, 0x00, 0x00, 0x60, 0x00,
    0x03, 0x01, 0x00, 0x00, 0x02, 0x01, 0x00, 0x00, 0x1f, 0x01,
```

Figure: The content of Model Weights File

## 1.4 Inference Program System Flow on the Nuvoton M55M1 Board

1. System Initialization

   At system startup, hardware initialization is performed using the BoardInit() function, which initializes components such as clock settings, UART, HyperRAM, and optionally, the NPU.

   Function Purpose

   (1) BoardInit()
   a. Objective:

Performs hardware-related initialization to ensure that all modules (clock, UART, memory, processing unit) are ready for subsequent operations.

b. Function Details:
   (a) SYS_Init(): Configures system clocks and parameters to maintain timing consistency across modules.
   (b) InitDebugUart(): Sets up UART (e.g., UART6) for debug output, enabling printf for diagnostics.
   (c) HyperRAM_Init(): Initializes HyperRAM and sets its operating mode and interface.
   (d) SPIM_HYPER_EnterDirectMapMode: Enters Direct Map mode for more efficient HyperRAM access.
   (e) ethosu_npu_init(): Initializes the Arm® Ethos™-U NPU if present. Returns an error status if initialization fails.
   (f) SYS_UnlockReg() / SYS_LockReg(): Unlocks and locks system protection registers for critical configurations.

```
int BoardInit(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    SYS_Init();

    /* UART init - will enable valid use of printf (std
     * re-directed at this UART (UART6) */
    InitDebugUart();

    SYS_LockReg();                    /* Unlock register

    HyperRAM_Init(HYPERRAM_SPIM_PORT);
    /* Enter direct-mapped mode to run new applications
    SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);

    info("%s: complete\n", __FUNCTION__);

#if defined(ARM_NPU)

    int state;

    /* If Arm Ethos-U NPU is to be used, we initialise
    if (0 != (state = arm_ethosu_npu_init()))
    {
        return state;
    }

#endif /* ARM_NPU */
```

Figure: BoardInit function

2. YOLO Initialization and Model Parameter Setup

Function Purpose

(1) YoloXnanoNu::Init()

a. Objective:

Initializes the YOLO model using its TensorFlow Lite version via YoloXnanoNu::Init() method.

b. Function Details:

Sets up the TensorFlow Lite runtime environment, including model data and buffer configuration.

```
bool YoloXnanoNu::Init(uint8_t *tensorArena,
                       size_t tensorArenaSize,
                       uint8_t *modelPointer,
                       size_t modelLen)
{
    info("Initializing YOLO model...\n");

    /* 設置 TensorFlow Lite 模型 */
    this->m_modelPointer = modelPointer;
    this->m_modelLen = modelLen;

    /* 初始化 TensorFlow Lite 解釋器 */
    this->m_interpreter = tflite::GetInterpreter(tensorArena, tensorArenaSize, modelPoint
    if (!this->m_interpreter)
    {
        printf_err("Failed to create TFLite interpreter.\n");
        return false;
    }
```

Figure: YoloXnanoNu::Init function

3. Capturing Image Input from Camera

Function Purpose

(1) get_empty_framebuf()

a. Objective:

Finds a buffer with status eFRAMEBUF_EMPTY to store a new image.

b. Function Details:

Iterates through the buffer array to locate and return an empty frame buffer, or NULL if none found.

```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

(Figure: get_empty_framebuf function)

(2) ImageSensor_Capture()

a. Objective:

Controls the image sensor to capture an image.

b. Function Details:

(a) CCAP_SetPacketBuf(): Sets the target buffer address.

(b) CCAP_Start(): Starts image capture.

(c) CCAP_Stop(TRUE): Stops capture and checks result.

```
int ImageSensor_Capture(uint32_t u32FrameBufAddr)
{
    int i32Ret = CCAP_OK;
    /* Set System Memory Packet Base Address Register */
    //printf("sensor capture address %x \n", u32FrameBufAddr);
    CCAP_SetPacketBuf((uint32_t)u32FrameBufAddr);

    /* Start image capture */
    CCAP_Start();

    /* Start image capture */
    i32Ret = CCAP_Stop(TRUE);

    if (i32Ret != CCAP_OK)
        return -1;

    return 0;
}
```

Figure: ImageSensor_Capture function

4. Helmet Detection Model Inference Task Execution

Function Purpose

(1) get_full_framebuf()

a. Objective:
Finds a frame buffer marked eFRAMEBUF_FULL for use in inference.

b. Function Details:
Iterates the frame buffer array s_asFramebuf and returns the first valid pointer or NULL.

```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_FULL)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_full_framebuf function

(2) inferenceJob

a. Objective:
Constructs an inference task and enqueues it for processing.

b. Structure Details:

(a) responseQueue: Queue to receive inference responses.

(b) pPostProc: Callback for post-processing, e.g., filtering low-confidence results or drawing bounding boxes.

(c) modelCols, modelRows: Input image dimensions.

(d) srcImgWidth, srcImgHeight: Source image size for coordinate scaling.

(e) results: Output structure for inference results.

```
//trigger inference
inferenceJob->responseQueue = inferenceResponseQueue;
inferenceJob->pPostProc = &postProcess;
inferenceJob->modelCols = inputImgCols;
inferenceJob->modelRows = inputImgRows;
inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
inferenceJob->results = &fullFramebuf->results;

xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
fullFramebuf->eState = eFRAMEBUF_INF;
```

Figure: inferenceJob structure

(3) PresentInferenceResult()

a. Objective:

Displays inference results by annotating detected objects with class labels and bounding boxes.

b. Function Details:

(a) results: A std::vector of DetectionResult, each with class index (m_cls), confidence (m_normalisedVal), and bounding box (m_x0, m_y0, m_w, m_h).

(b) labels: A std::vector<std::string> mapping class indices to human-readable names.

```
static bool PresentInferenceResult(const std::vector<arm::app::object_detection::DetectionResult> &results,
                                   std::vector<std::string> &labels)
{
    /* If profiling is enabled, and the time is valid. */
    info("Final results:\n");

    for (uint32_t i = 0; i < results.size(); ++i)
    {
        info("%" PRIu32 ") %s(%f) -> %s {x=%d,y=%d,w=%d,h=%d}\n", i,
             labels[results[i].m_cls].c_str(),
             results[i].m_normalisedVal, "Detection box:",
             results[i].m_x0, results[i].m_y0, results[i].m_w, results[i].m_h);
    }

    return true;
}
```

Figure: PresentInferenceResult function

5. Post-Processing Module

Performs Non-Maximum Suppression (NMS) to improve detection accuracy by removing redundant bounding boxes.

The purpose of NMS Purpose is to filter overlapping boxes to retain only the most confident detections.

Function Purpose

(1) CalculateNMS() Parameters:

a. detections: Inference outputs including positions, confidence, and class info.

b. net.numClasses: Total number of supported classes, NMS is applied per class.

c. m_nms: IoU threshold to determine box overlap—lower values apply stricter suppression.

6. Displaying Results

   Visualizes detection outcomes using DrawImageDetectionBoxes() by overlaying bounding boxes and labels on the image.

   Function Purpose

   (1) DrawImageDetectionBoxes()
      a. Objective:
         Draws object detection results with bounding boxes and labels on the image for user display.
      b. Parameters:
         (a) results: Vector of detection results.
         (b) drawImg: Target image for rendering.
         (c) labels: Class label vector (e.g., "none", "warning", "helmet", "person").

```
static void DrawImageDetectionBoxes(
    const std::vector<arm::app::object_detection::DetectionResult> &results,
    image_t *drawImg,
    std::vector<std::string> &labels)
{
    for (const auto &result : results)
    {
        imlib_draw_rectangle(drawImg, result.m_x0, result.m_y0, result.m_w, result.m_h, COLOR_B5_MAX, 1, false);
        imlib_draw_string(drawImg, result.m_x0, result.m_y0 - 16, labels[result.m_cls].c_str(), COLOR_B5_MAX, 2, 0, 0, false,
                          false, false, false, 0, false, false);
    }
```

Figure: DrawImageDetectionBoxes function

   (2) Internal Operations
      a. Drawing the Rectangle:

         The imlib_draw_rectangle function is used to draw a rectangle based on the result's m_x0, m_y0 (top-left coordinates), m_w (width), and m_h (height). The color and style of the rectangle can be customized.

      b. Displaying the Label (String):

         The imlib_draw_string function is used to display the label text (provided by labels) above the rectangle. Font color, size, and style can be configured.

# 2 Smart Factory 2 – Fire Detection

## 2.1 Use Case Overview – Fire Detection

This project utilizes a dataset from the Roboflow open-source library and applies machine learning techniques to train a model using YOLOX Nano. The trained model is then converted to the TensorFlow Lite framework via ONNX, followed by Full-INT8 quantization to meet the size constraints of the development board. Subsequently, the model is further optimized using the Vela compiler to ensure all inference computations are executed on the Neural Processing Unit (NPU). Finally, the optimized system is flashed to the Nuvoton M55M1 development board using the Keil toolchain, successfully realizing a real-time, high-performance, and energy-efficient fire detection system.

### 2.1.1 Introduction to Nuvoton Development Board and Practical Applications

1. High-Efficiency Hardware Performance

   Equipped with the Cortex®-M55 core and the Ethos™-U55 microNPU, the board supports processing speeds of up to 220 MHz for neural network inference. With support for Full-INT8 quantization, model computation becomes even more efficient after quantization, significantly reducing memory and computational resource usage—enabling highly efficient real-time inference.

2. Real-Time Performance with Low Latency

   Thanks to the hardware acceleration capabilities of the Ethos™-U55 and the high throughput of the Cortex®-M55, the system achieves a solid balance between inference speed and detection accuracy in real-time applications.

3. Rich Expansion Interfaces

   The M55M1 provides various interfaces, including a CMOS sensor interface (CCAP) and a TFT-LCD display interface, allowing integration of a camera and display for live image capture and on-device inference.

### 2.1.2 Practical Application in Fire Detection Systems

In factory environments where high temperatures prevail, fires are more likely to occur and may start in areas that are not frequently monitored. When the fire

detection system identifies the presence of flames, it triggers an alarm to alert factory personnel to potential hidden dangers.

Based on the aforementioned advantages of the development board, the benefits of using the M55M1 development board for fire detection include:

1. Rapid Response: The flame detection capability can be integrated with sensors (e.g., Arduino buzzers) to emit alerts at the scene of a fire, thereby helping to mitigate casualties and property damage.
2. Efficient Deployment: With its low power consumption and rich peripheral interfaces, the system can be deployed effectively in complex environments such as factories and warehouses.

## 2.2 Dataset Collection

### 2.2.1 Dataset Introduction: Fire Detection Dataset (Roboflow)

1. Dataset Source

   This dataset is sourced from the Fire Detection dataset on the Roboflow platform. It is specifically designed for fire detection tasks and contains a wide variety of flame scene images. The dataset encompasses diverse environments (e.g., indoor and outdoor, daytime and nighttime) to ensure model adaptability and robustness across different scenarios.

2. Dataset Format

   The dataset adopts the COCO JSON format, which includes the following features:

   (1) Standardized Annotations: Each image's flame location is annotated using bounding boxes. Annotation data is stored in JSON format, making it easily accessible for training frameworks.
   (2) Extensibility: Supports multi-class annotations and descriptions of complex scenes.
   (3) Training Set: 1,210 images, used for primary model training, providing sufficient samples for learning and fitting.
   (4) Validation Set: 347 images, used during training to evaluate model performance and prevent overfitting.
   (5) Test Set: 164 images, used for final performance evaluation of the model.

3. Dataset Characteristics
   (1) Diversity: Includes various scenes such as industrial zones, residential environments, and outdoor settings, as well as different fire forms like open flames and smoke, greatly enhancing the model's generalization capability.
   (2) High-Quality Annotations: Each flame region in the images is precisely annotated to ensure accurate bounding box localization.
   (3) Openness: The dataset is publicly available via the Roboflow platform and can be extended or augmented as needed.

4. Model Training Environment

   To process this dataset, the project employs the following environment setup:

   (1) Virtual Environment: Anaconda
   (2) Training Framework: PyTorch
   (3) Model: YOLOX-Nano

5. Dataset Application

   The dataset is primarily used for fire detection tasks and is suitable for the following application scenarios:

   (1) Smart Security: Integrated into surveillance systems for real-time fire detection and alert triggering.
   (2) Industrial Safety: Deployed in factories or hazardous areas to enhance fire emergency response capabilities.

   Model and Dataset Download Link:
   https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

### 2.2.2 Dataset Preparation

This project uses the open-source fire detection dataset from Roboflow for training. Given that YOLOX Nano is selected as the training model, the dataset is formatted in the COCO standard.

### 2.2.3 Introduction to Roboflow

Roboflow is a platform dedicated to managing image datasets, aimed at helping users efficiently handle image data throughout the lifecycle—from annotation to

transformation and deployment. The platform offers numerous public datasets contributed by users. Roboflow enables users to browse and utilize datasets that match their project needs. Its core advantages include:

1. Multiple Format Support

   Roboflow can convert datasets into various formats such as COCO, YOLO, and Pascal VOC, making integration with different training frameworks seamless.

2. Data Augmentation

   Offers a wide array of augmentation techniques including rotation, scaling, flipping, and brightness adjustment, which improve model generalization.

3. Cloud-Based Management

   Datasets are stored in the cloud, supporting team collaboration and version control.

### 2.2.4 Dataset and Training Pipeline Design: From Collection to Deployment

Steps to Acquire Dataset from Roboflow

Step 1: Visit the official Roboflow website

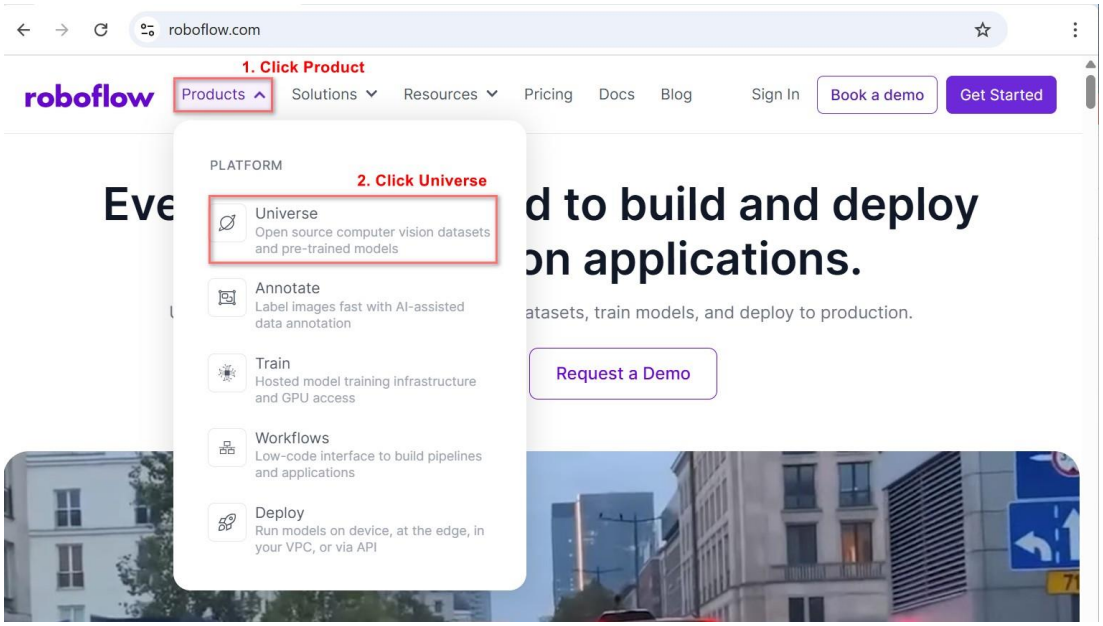


Figure 68: https://roboflow.com

Step 2: Navigate to the Open Datasets section

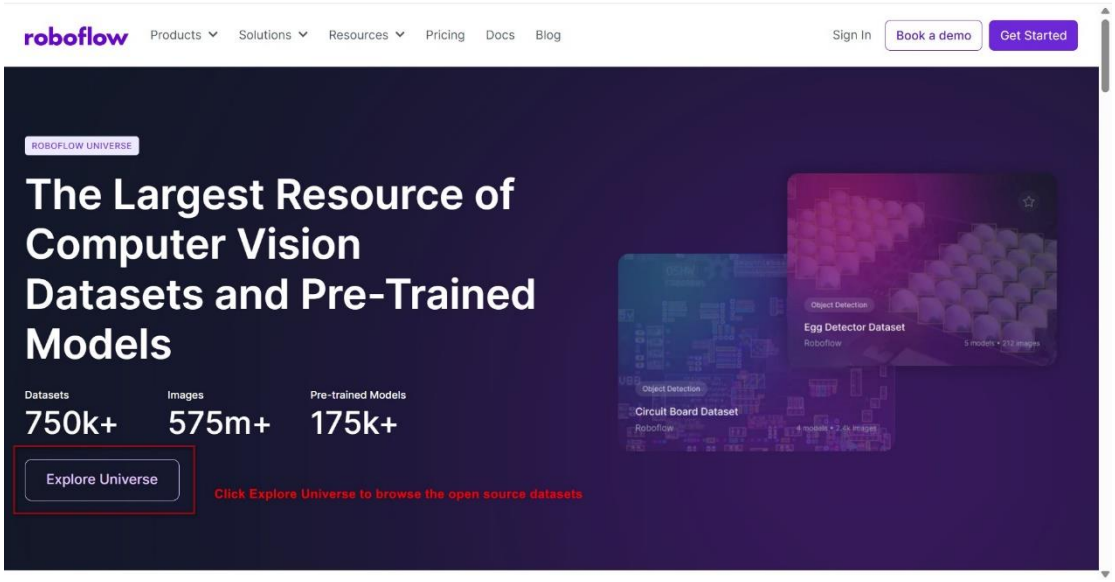


Figure 69: Accessing the Open-Source Datasets

Step 3: Search for a dataset using relevant keywords

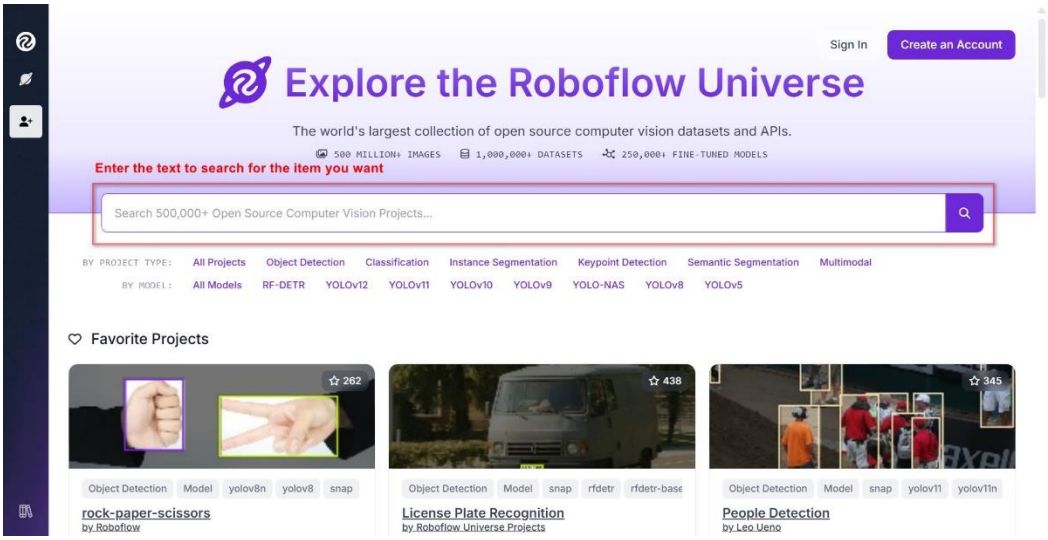


Figure 70: Search for “FIRE” dataset

Step 4: Select the desired dataset project


Figure 71: Download option for selected dataset

Step 5: Choose the desired export format

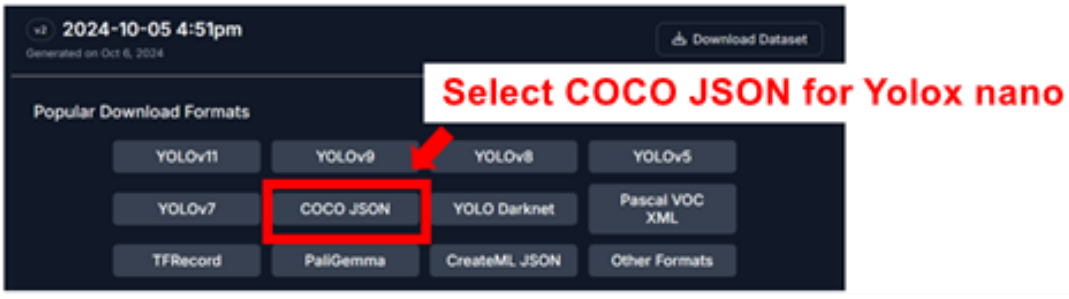


Figure 72: Select COCO format to download the dataset

## 2.3 Training the Fire Detection Model on PC Using Anaconda Environment

### 2.3.1 Training Hardware and Configuration

1. Hardware: NVIDIA GeForce RTX 4060
2. CUDA Version: 11.8
3. PyTorch Version: 2.0.0
4. Training Dataset: 1,210 images
5. Training Parameters: Epochs = 200, Batch size = 64
6. Training Duration: Approximately 1 hour

### 2.3.2 Complete Fire Detection Model Training Procedure

1. Environment Setup Steps
   (1) Create Python Virtual Environment

   Use the conda command to create a new Python virtual environment named yolox_nu with Python version 3.10:

   ```
   $ conda create --name yolox_nu python=3.10
   ```

   Activate the newly created environment:

   ```
   $ conda activate yolox_nu
   ```

   Upgrade pip and setuptools to the latest versions:

   ```
   $ python -m pip install --upgrade pip setuptools
   ```

   (2) Install CUDA, PyTorch, and Related Packages

   Install deep learning packages optimized for CUDA 11.8 and PyTorch 2.0:

   ```
   $ python -m pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
   ```

   (3) Install MMCV (Version 2.0.1)

   Install MMCV, an OpenMMLab utility library supporting model training:

   ```
   $ python -m pip install mmcv==2.0.1 -f https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/index.html
   ```

(4) Install Additional Dependencies

Change the working directory to the project root. Then install required packages listed in requirements.txt:

```
$ python -m pip install --no-input -r requirements.txt
```

(5) Install YOLOX Package

Install the YOLOX model in development mode to allow real-time code modifications without reinstallation:

```
$ python setup.py develop
```

After completing the above steps, the YOLOX Nano training environment is fully prepared. You may then proceed to dataset configuration and begin training the model.

2. Preparing for Custom Dataset Training

The following outlines the key steps required prior to model training:

(1) Training with a Pretrained Model

Use the default configuration file located at `exps/default/yolox_nano_ti_lite_nu.py`. This file includes the predefined YOLOX Nano model parameters. Ensure that it contains the appropriate settings for your training scenario.

(2) Preparing the Dataset

The dataset must be organized according to the following directory structure. In addition, modify the self.data_dir field in the YOLOX Python configuration file to specify the actual dataset path.

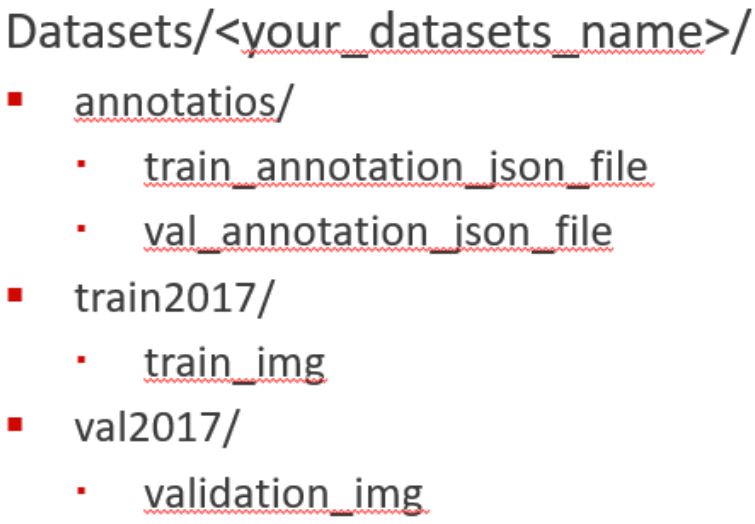


Figure: Dataset directory architecture


Figure: Set dataset path to code

Update this path accordingly in the training script.

(3) Adjusting Configuration File Parameters

Edit the parameters in yolox_nano_ti_lite_nu.py:

`self.num_classes = 6`

This line sets the number of object categories in the dataset. Change it to match the actual number of classes in your dataset.

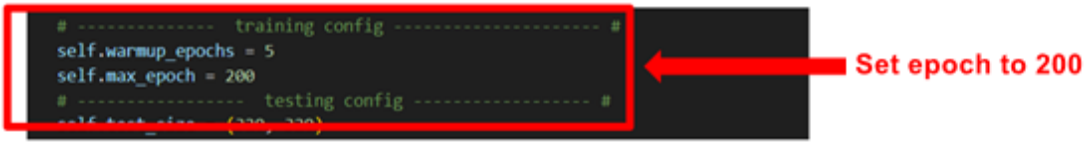


Figure: Set epoch


Figure: Set num_classes

(4) About Epochs

An epoch refers to a single pass through the entire training dataset by the neural network.

a. Effect of Epoch Count:
   - Too few: The model may underfit due to insufficient learning.
   - Too many: The model may overfit and waste training time.

   Use validation accuracy/loss to determine an optimal value and consider implementing Early Stopping to automatically halt training once performance saturates.

b. How to Choose a Proper Epoch Value:

   Consider:

   - Dataset size: Smaller datasets may require more epochs.
   - Model complexity: More complex models may benefit from more training iterations.
   - Monitoring: Track loss/accuracy on the validation set to choose the most efficient stopping point.

c. Interaction with Other Parameters:

   Batch Size: Smaller batches may require more epochs; larger batches provide stable gradients but may reduce generalization.

   Learning Rate: A high learning rate may cause instability, requiring more epochs for correction; a low learning rate may need many epochs to converge.

3. Start Training

Execute the following command in the terminal to begin model training:

```
python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
<BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
```

Parameter Explanation:

`-f <MODEL_CONFIG_FILE>`: Path to the configuration file (yolox_nano_ti_lite_nu.py)

`-d 1`: Number of devices (e.g., 1 GPU)

`-b <BATCH_SIZE>`: Batch size (e.g., 64)

`--fp16`: Enable mixed precision training for speedup

`-o`: Enable automatic mixed precision optimization

`-c <PRETRAIN_MODEL_PATH>`: Path to the pretrained weights file

This completes the preparation process for training a custom fire detection model using YOLOX Nano.

### 2.3.3 Detailed Explanation of Machine Learning Model Framework Conversion and Quantization

1. Machine Learning Model Framework Conversion

   PyTorch is a powerful deep learning framework; however, it may require conversion to a more universal format for deployment on embedded devices or cross-platform environments. ONNX (Open Neural Network Exchange) provides a standardized model format that facilitates interoperability across different frameworks.

   Execute the following command to convert the PyTorch model into the ONNX format. This step generates an ONNX model to enable subsequent optimization or deployment:

   ```
   python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c <TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
   ```

   Parameter Explanation:

   `<MODEL_CONFIG_FILE>`: The configuration file defining the model structure and parameters.

   `<TRAINED_PYTORCH_MODEL>`: The trained PyTorch model weights file.

   `<ONNX_MODEL_PATH>`: The output path for the converted ONNX model.

2. Model Quantization

Model quantization is a technique that improves execution efficiency by reducing model parameter precision (e.g., from floating point to integer format). This is particularly important for resource-constrained systems such as embedded platforms. This project applies Full Integer 8-bit Quantization (Full-INT8), where calibration data is used during quantization to fine-tune parameters and maintain model accuracy.

(1) Create Calibration Data

Execute the following command to generate calibration data:

```
python demo/TFLite/generate_calib_data.py --img-size
<IMG_SIZE> --n-img <NUMBER_IMG_FOR_CALI> -o
<CALI_DATA_NPY_FILE> --img-dir <PATH_OF_TRAIN_IMAGE_DIR>
```

Parameter Explanation:

<IMG_SIZE>: Input image dimensions, e.g., 320x320.

<NUMBER_IMG_FOR_CALI>: Number of images used for calibration.

<CALI_DATA_NPY_FILE>: Output path of the calibration data .npy file.

<PATH_OF_TRAIN_IMAGE_DIR>: Path to the training image directory.

(2) Convert ONNX to TFLite (Quantized Format)

This step compresses the ONNX model into a TensorFlow Lite format suitable for embedded systems. Use the following command to perform the conversion with quantization:

```
onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
<CALI_DATA_NPY_FILE> "[[[[0,0,0]]]]" "[[[[1,1,1]]]]"
```

Parameter Explanation:

<ONNX_MODEL_PATH>: Path to the input ONNX model.

<TFLITE_MODEL_PATH>: Path to the output quantized TFLite model.

-oiqt: Enables Full Integer 8-bit Quantization.

-qcind images: Specifies the calibration input data type as images.

<CALI_DATA_NPY_FILE>: Calibration data file used for parameter adjustment.

`"[[[[0,0,0]]]]" / "[[[[1,1,1]]]]"`: Defines the normalization range of the input tensor.

### 2.3.4 Introduction and Workflow of the Vela Compiler

1. Overview of the Vela Compiler

   Vela is a compiler designed specifically for Arm® microNPU (Neural Processing Unit). Its primary purpose is to optimize TensorFlow Lite (TFLite) models and generate high-efficiency inference formats compatible with Arm's microNPU architecture.

   (1) Core Features

   a. Model Optimization: Converts TFLite models into optimized formats for acceleration on NPU.
   b. Computation Allocation: Offloads suitable operations to the NPU, reducing CPU workload and improving inference speed.
   c. Memory Access Optimization: Enhances memory access patterns and instruction flows to boost inference efficiency.

   (2) Weight Compression

   Supports lossless compression of weight files, reducing SRAM and Flash memory usage without sacrificing accuracy.

2. Vela Compilation Workflow

   Step 1: Modify the variables.bat configuration file to set paths and parameters.

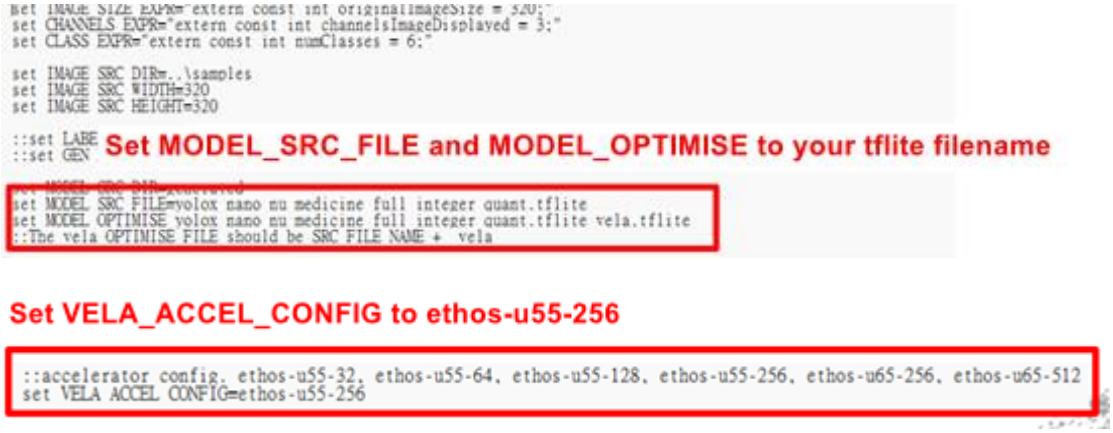

Figure: Modify the variables.bat file

Step 2: Run the gen_model_cpp.bat batch file. Upon execution, this generates a Vela-compiled and optimized TFLite model suitable for direct inference using a microNPU.


Figure: Run the gen_model_cpp.bat

The resulting TFLite model can be visualized using Netron, an open-source model visualization tool that supports various neural network formats (e.g., CNN, RNN, etc.).

**Structural Comparison Before and After Vela Compilation**:

Before Compilation: Some operations on the CPU require additional memory I/O, which slows down inference.

After Compilation: All computations are offloaded to the Ethos™-U NPU, eliminating the need for concatenation operations and reducing data transfer overhead—thus enhancing inference performance.

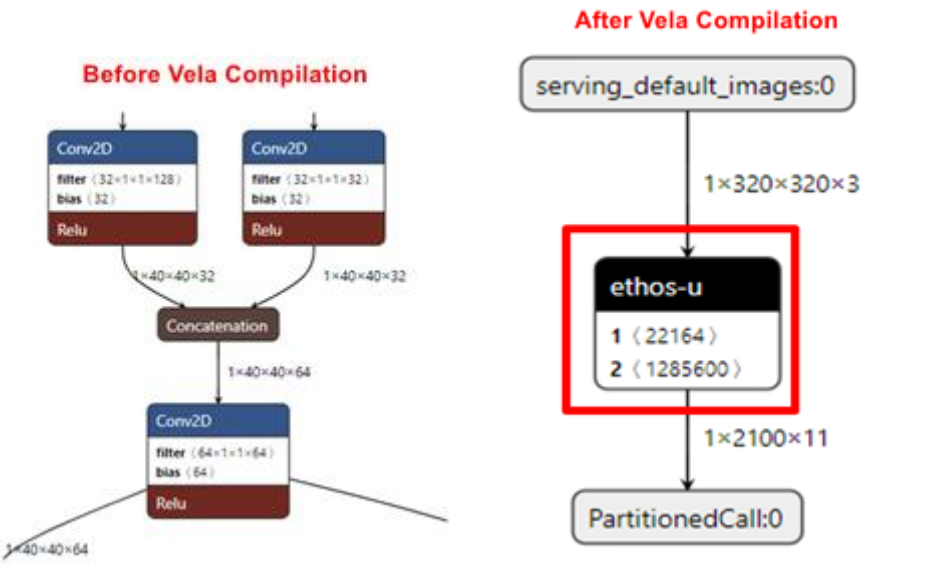


Figure: Structure comparison before/after Vela compilation

## 2.4 Inference Program System Flow on the Development Board

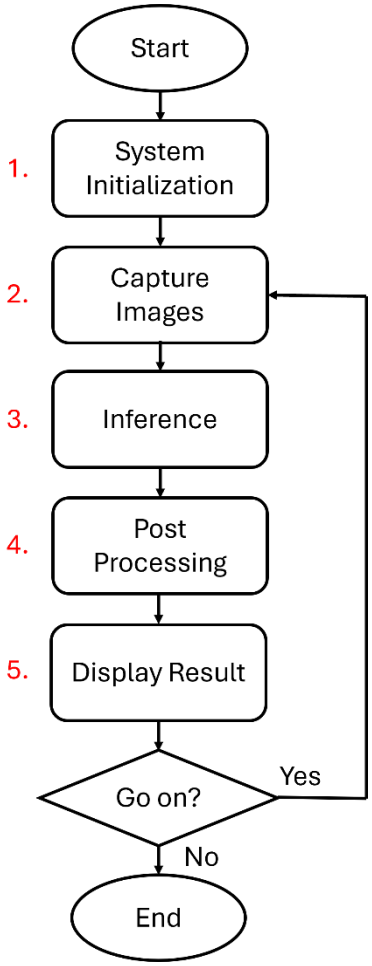


Figure: System Flowchart

### 2.4.1 System Initialization

System Initialization aims to establish a stable operational foundation to prevent unexpected issues during execution. This stage includes initializing hardware resources to ensure devices (such as sensors and processors) function properly and meet the system's runtime requirements. It also involves configuring the software environment—loading models and setting parameters—to enable optimal inference performance.

In embedded system development, initializing the hardware and system clocks is a critical step for stable operation. Below is a detailed breakdown of two code segments: system clock initialization and hardware initialization.

4. System Clock Initialization (SYS_Init function)
   (1) Enable Internal RC 12MHz Clock (HIRC):
      a. Call CLK_EnableXtalRC(CLK_SRCCTL_HIRCEN_Msk) to enable the internal oscillator.
      b. Use CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk) to ensure clock stability.
   (2) Enable External High-Frequency Clock (HXT):
      a. Enable via CLK_EnableXtalRC(CLK_SRCCTL_HXTEN_Msk)
      b. Confirm stability with CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk)
   (3) Set System Clock Source:
      a. Switch to APLL0 and configure to 180MHz using CLK_SetBusClock.
      b. Call SystemCoreClockUpdate() to update the core clock value.
   (4) Enable Module Clocks:
      a. Enable clocks for GPIO, NPU, FMC0, and CCAP0.
      b. Set UART6 clock source to HIRC using SetDebugUartCLK().
   (5) Configure Multi-Function Pins:
      a. Set UART RXD and TXD pins for serial communication.
      b. Call HyperRAM_PinConfig to configure SPI pins for HyperRAM.

```c
static void SYS_Init(void)
{
    /*---------------------------------------------------------------------------------------------------------*/
    /* Init System Clock                                                                                       */
    /*---------------------------------------------------------------------------------------------------------*/
    CLK_EnableXtalRC(CLK_SRCCTL_HIRCEN_Msk);
    CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);
    CLK_EnableXtalRC(CLK_SRCCTL_HXTEN_Msk);
    CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk);
    CLK_SetBusClock(CLK_SCLKSEL_SCLKSEL_APLL0, CLK_APLLCTL_APLLSRC_HIRC, FREQ_180MHZ);
    SystemCoreClockUpdate();
    CLK_EnableModuleClock(GPIOA_MODULE);
    CLK_EnableModuleClock(GPIOB_MODULE);
    CLK_EnableModuleClock(GPIOC_MODULE);
    CLK_EnableModuleClock(GPIOD_MODULE);
    CLK_EnableModuleClock(GPIOE_MODULE);
    CLK_EnableModuleClock(GPIOF_MODULE);
    CLK_EnableModuleClock(GPIOG_MODULE);
    CLK_EnableModuleClock(GPIOH_MODULE);
    CLK_EnableModuleClock(GPIOI_MODULE);
    CLK_EnableModuleClock(GPIOJ_MODULE);
    CLK_EnableModuleClock(FMC0_MODULE);
    CLK_EnableModuleClock(NPU0_MODULE);
    CLK_EnableModuleClock(CCAP0_MODULE);
    SetDebugUartCLK();
    /*---------------------------------------------------------------------------------------------------------*/
    /* Init I/O Multi-function                                                                                 */
    /*---------------------------------------------------------------------------------------------------------*/
    SetDebugUartMFP();
    HyperRAM_PinConfig(HYPERRAM_SPIM_PORT);
}
```

Figure: System Clock Initialization (SYS_Init function)

5. Hardware Initialization (BoardInit function)

This function invokes SYS_Init() and performs additional hardware setups:

(1) Unlock Protected Registers:

Unlock system registers using SYS_UnlockReg() for clock and module configuration.

(2) Invoke SYS_Init():

Complete system and module clock setup.

(3) Initialize UART:

Call InitDebugUart() to set up UART6 for printf output.

(4) Lock Registers:

Use SYS_LockReg() to prevent unintended future changes.

(5) Initialize HyperRAM:

Call HyperRAM_Init() and enter Direct Map Mode using SPIM_HYPER_EnterDirectMapMode().

(6) Initialize NPU (Optional):

If ARM® NPU is used, initialize Ethos™-U with ethosu_npu_init().

(7) Output System Design Name:

Use info("Target system: %s\n", DESIGN_NAME) to display the design identifier.

```
int BoardInit(void)
{
    SYS_UnlockReg();
    SYS_Init();
    InitDebugUart();
    SYS_LockReg();                  /* Unlock register lock protect */
    HyperRAM_Init(HYPERRAM_SPIM_PORT);
    SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);
    info("%s: complete\n", __FUNCTION__);
#if defined(ARM_NPU)
    int state;
    /* If Arm Ethos-U NPU is to be used, we initialise it here */
    if (0 != (state = arm_ethosu_npu_init()))
    {
        return state;
    }
#endif /* ARM_NPU */
    /* Print target design info */
    info("Target system: %s\n", DESIGN_NAME);
    return 0;
}
```

Figure: Hardware Initialization (BoardInit function)

### 2.4.2 Capture Image

The purpose of image capture is to obtain real-time input images for inference. It ensures rapid response and accurate analysis. Input accuracy and timeliness are critical for detection quality, especially in dynamic environments where real-time updates are necessary (e.g., moving objects). In main.cpp, image capture is handled via the Camera Capture (CCAP) functionality within the main_task() function.

1. Camera Initialization
   (1) Call ImageSensor_Init to set up hardware.
   (2) Call ImageSensor_Config to configure image format as eIMAGE_FMT_RGB565 and resolution using frameBuffer.w and frameBuffer.h.

```
#if defined (__USE_CCAP__)
    //Setup image senosr
    ImageSensor_Init();
    ImageSensor_Config(eIMAGE_FMT_RGB565, frameBuffer.w, frameBuffer.h);
#endif
```

Figure: Code for Camera Initialization

2. Image Capture Logic

   Check for an available buffer using get_empty_framebuf(). If available, proceed to capture.

```
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: Check the Availability of Empty Image Buffers

3. Capture Execution

   Use ImageSensor_Capture to store the image in emptyFramebuf->frameImage.data.

```
ImageSensor_Capture((uint32_t)(emptyFramebuf->frameImage.data));
```

Figure: Execute Capture Operation

4. Update Buffer State

Mark buffer status as eFRAMEBUF_FULL to indicate availability of a complete image.

```
emptyFramebuf->results.clear();
emptyFramebuf->eState = eFRAMEBUF_FULL;
```

Figure: Update Image Buffer State

5. Image Format Processing (Static Source Mode)

   In non-CCAP mode, static images are copied from pu8ImgSrc to the buffer. Image scaling and format conversion are performed by imlib_nvt_scale.

6. Frame Buffer Management (S_FRAMEBUF)
   (1) get_empty_framebuf: Obtain a free buffer for new capture.
   (2) get_full_framebuf: Identify filled frames pending processing.
   (3) get_inf_framebuf: Retrieve inference-ready frames for display.

7. Output and Preprocessing

   Captured images are resized and quantized, then passed to the inference module for object detection.

### 2.4.3 Inference

The inference stage uses a pre-trained model to analyze captured images and detect or classify objects. The system extracts features and generates inference results in real-time, leveraging on-board hardware such as the NPU for acceleration. Efficient and accurate inference is a key performance metric.

Steps:

```
arm::app::YoloXnanoNu model;
```

Figure: STEP 1: Create model instance.

```
if (!model.Init(arm::app::tensorArena,
                sizeof(arm::app::tensorArena),
                arm::app::yoloxnanonu::GetModelPointer(),
                arm::app::yoloxnanonu::GetModelLen()))
{
    printf_err("Failed to initialise model\n");
    vTaskDelete(nullptr);
    return;
}
```

Figure: STEP 2: Initialize the model.

```
TfLiteIntArray *inputShape = model.GetInputShape(0);

const int inputImgCols = inputShape->data[arm::app::YoloXnanoNu::ms_inputColsIdx];
const int inputImgRows = inputShape->data[arm::app::YoloXnanoNu::ms_inputRowsIdx];
```

Figure: STEP 3: Configure model input dimensions.

Capture image to S_FRAMEBUF *fullFramebuf. Use imlib_nvt_scale to resize the image and define ROI. Load processed image into the input tensor via TfLiteTensor *inputTensor = model.GetInputTensor(0). Call m_model->RunInference() to execute analysis.

```
S_FRAMEBUF *fullFramebuf;
imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);
TfLiteTensor *inputTensor    = model.GetInputTensor(0);
bool runInf = m_model->RunInference();
```

Figure: STEP 4: Preprocessing Image

### 2.4.4 Post-Processing & Draw Results

Post-processing refines the model output by filtering low-confidence detections and visualizing results on the image with bounding boxes and labels.

Steps:

1. Scale Bounding Boxes

Map model output (x, y, w, h) to the original image scale and clip to valid bounds.

```
for (auto &it : detections)
{

    // transfer to original img size (base on resize ratio)
    it.bbox.x = (it.bbox.x * originalImageWidth) / net.inputWidth;
    it.bbox.y = (it.bbox.y * originalImageHeight) / net.inputHeight;
    it.bbox.w = (it.bbox.w * originalImageWidth) / net.inputWidth;
    it.bbox.h = (it.bbox.h * originalImageHeight) / net.inputHeight;
```

Figure: Scale Bounding Box

2. Extract and Store Results

   For each class, if confidence > 0, save the class, score, and bounding box.

```
for (int j = 0; j < net.numClasses; ++j)
{
    if (it.prob[j] > 0)
    {

        DetectionResult tmpResult = {};
        tmpResult.m_normalisedVal = it.prob[j];
        tmpResult.m_x0 = (int)boxX;
        tmpResult.m_y0 = (int)boxY;
        tmpResult.m_w = (int)boxWidth;
        tmpResult.m_h = (int)boxHeight;
        tmpResult.m_cls = j;

        resultsOut.push_back(tmpResult);
    }
}
```

Figure: Extract Result

3. Apply Non-Maximum Suppression (NMS)

   Remove overlapping boxes using threshold m_nms.

```
/* Do nms */
CalculateNMS(detections, net.numClasses, m_nms);
```

Figure: Execute NMS

4. Store Final Results

   Results are saved in resultsOut for later display or analysis.

```
resultsOut.push_back(tmpResult);
```

Figure: Store Final Results

### 2.4.5 Display Results

Final results are displayed visually to users, enabling real-time monitoring and informed decisions (e.g., triggering alarms).

In main.cpp, the DrawImageDetectionBoxes function visualizes the detection:

1. Iterate over all detections in results.
2. Draw bounding rectangles via imlib_draw_rectangle().
3. Render class labels above boxes using imlib_draw_string(labels[result.m_cls]).

```
static void DrawImageDetectionBoxes(
    const std::vector<arm::app::object_detection::DetectionResult> &results,
    image_t *drawImg,
    std::vector<std::string> &labels)
{
    for (const auto &result : results)
    {
        imlib_draw_rectangle(drawImg, result.m_x0, result.m_y0, result.m_w, result.m_h, COLOR_B5_MAX, 1, false);
        imlib_draw_string(drawImg, result.m_x0, result.m_y0 - 16, labels[result.m_cls].c_str(), COLOR_B5_MAX, 2, 0, 0, false,
                          false, false, false, 0, false, false);
    }
}
```

Figure: DrawImageDetectionBoxes Function

### 2.4.6 Flash to Development Board and Test


Image 1: Example test image

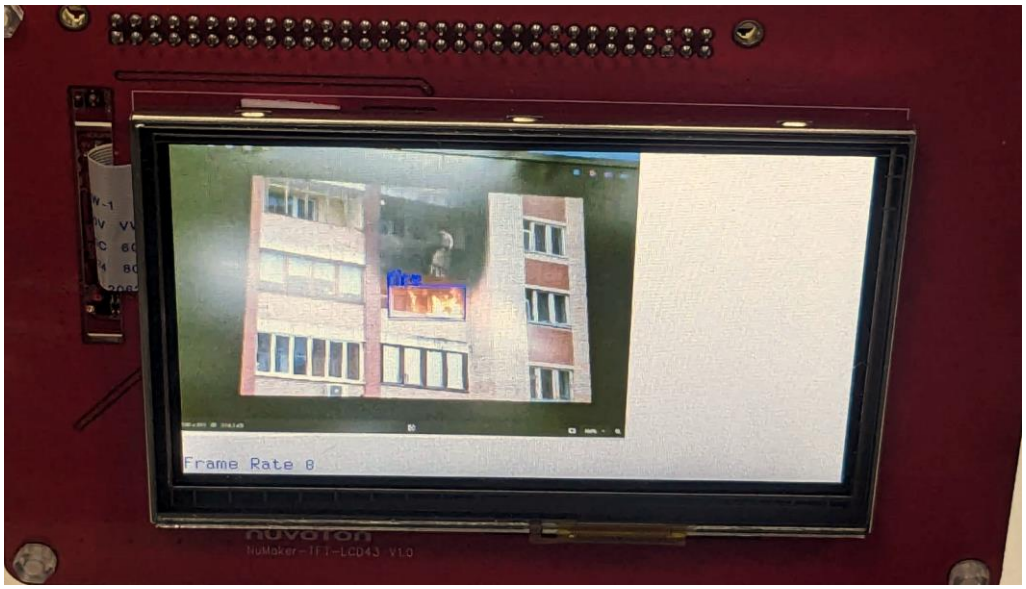


Image 2: Example test image

## 2.5 Conclusion and Future Development

In this fire detection system project, we successfully implemented a high-performance, real-time, and energy-efficient fire detection solution on the Nuvoton M55M1 development board by integrating the Roboflow dataset with the YOLOX-Nano model. Through Full-INT8 quantization and optimization using the Vela compiler, we significantly enhanced both inference speed and accuracy, achieving up to 98% detection accuracy in testing. The system accomplished the following goals:

1. Rapid Response Capability

Real-time image capture and fire region annotation provide effective early warning for factories and other high-risk environments.

2. High Performance with Low Power Consumption

   By leveraging NPU hardware acceleration, the system delivers high-performance computation within an embedded environment while reducing power consumption.

3. Scalability

   The development board's rich interface support makes it suitable for various scenarios, including industrial zones, home safety, and public facility fire prevention.

4. Multimodal Sensor Integration

   Integrating other sensing technologies (such as temperature sensors or smoke detectors) enhances detection accuracy and versatility.

5. Improved Model Generalization

   Expanding the dataset with diverse fire scenarios and environments strengthens the model's adaptability across varied real-world conditions.

6. Edge Computing Optimization

   Further research into advanced model compression and optimization techniques for embedded systems can help minimize computational resource requirements.

7. Remote Monitoring and Connectivity

   By incorporating IoT technologies, the system enables real-time cloud data upload and remote monitoring, thereby enhancing overall usability and system performance.

These future development directions will further improve the functionality and applicability of the fire detection system, contributing to broader deployment in smart factories, smart homes, and public safety domains—ultimately reducing the risk of fire-related loss of life and property.

References:

English

- Roboflow. In this blog, you will learn about | by Muhammad Faizan | Red Buffer | Medium

- https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8?tab=readme-ov-file

Traditional Chinese

- https://hackmd.io/@hohoho/rkdZNWaZh
- https://medium.com/@andy6804tw/%E5%BF%AB%E9%80%9F%E4%B8%8A%E6%89%8Byolo-%E5%88%A9%E7%94%A8-roboflow-%E5%92%8C-ultralytics-hub-%E5%AE%8C%E6%88%90%E6%A8%A1%E5%9E%8B%E8%A8%93%E7%B7%B4%E8%88%87%E7%AE%A1%E7%90%86-%E4%B8%8A-37acd110a8a0
- https://hackmd.io/@nB1rzit6Toq8WdkZosRK_Q/r16-c7Ynj
- https://ithelp.ithome.com.tw/articles/10305905

# 3 Smart Factory 3 – Noise Reduction and Keyword Detection

## 3.1 Case Overview – Noise Reduction and Keyword Detection

### 3.1.1 Project Background

In the rapidly evolving era of digitalization and intelligent technologies, audio processing has become a critical technology in IoT, smart factories, and embedded application scenarios. With the widespread adoption of smart devices, user demand for voice interaction continues to grow. However, in industrial environments or household settings, background noise often interferes with accurate voice signal recognition. This raises the bar for the user experience of smart devices.

How to effectively suppress noise while accurately recognizing voice commands is a significant research challenge in current speech processing technology. This project focuses on an embedded voice processing solution based on the Nuvoton M55M1 development board. It integrates spectral subtraction noise reduction technology with a keyword spotting (KWS) model. The goal is to address the accuracy of voice commands under noisy conditions while enhancing the system's real-time performance.

The specific objectives of this project include:

1. Development of a Noise Reduction Module:

   Utilizing spectral subtraction techniques to significantly reduce background noise, improve the clarity of speech signals, and provide high-quality data for subsequent voice processing.

2. Integration of Keyword Spotting Module:

   By incorporating keyword recognition technology, the system can quickly identify voice commands for use in voice-controlled scenarios.

3. Performance Optimization:

   Leveraging ARM® Helium technology to enhance computational efficiency and ensure the system meets low-power and real-time processing requirements in embedded environments.

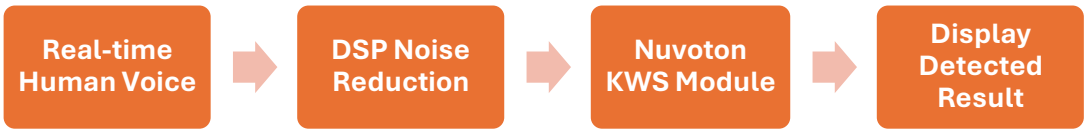


Figure: Projected Implementation Flow

### 3.1.2 Expected Goals

This project builds on the keyword recognition module provided by Nuvoton, which is capable of recognizing ten common keywords:

"yes", "no", "up", "down", "left", "right", "on", "off", "stop", "go".

Based on this foundation, a spectral subtraction-based noise reduction function will be added and accelerated using ARM® Helium technology to further improve system performance. At the same time, the terminal's interactive functionality will be enhanced, offering more operation options for user control and adjustment.

In testing under noisy environments, the original keyword recognition module achieved an accuracy rate of 69% (69/100). After incorporating the spectral subtraction-based noise reduction feature, the recognition accuracy significantly improved to 85% (85/100). This demonstrates that such noise reduction technology can indeed enhance the stability and accuracy of voice recognition.

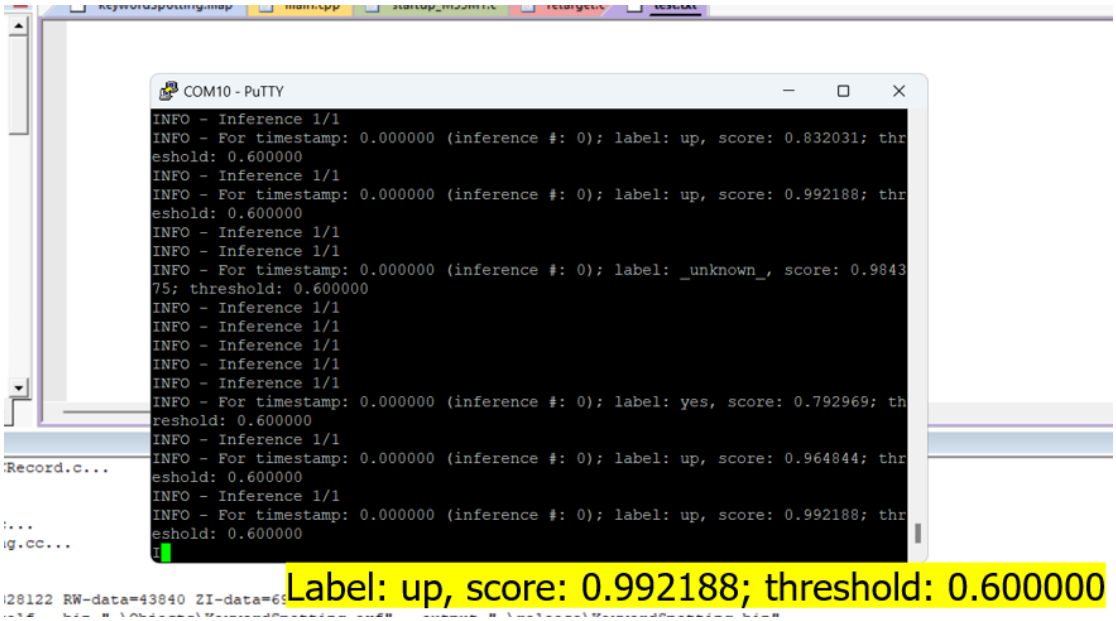


Figure: Recognition result for voice input "up" displayed on the terminal

### 3.1.3 Introduction to the Nuvoton M55M1 Development Board

The Nuvoton M55M1 development board is a high-performance hardware platform designed for embedded application scenarios. Its core is based on the ARM® Cortex®-M55 processor architecture, combined with multiple technological advantages, making it particularly well-suited for voice processing applications.

Key Features of the Nuvoton M55M1:

1. High-Performance Computing:
   (1) Supports Helium technology (M-Profile Vector Extension), significantly enhancing computation speed for Digital Signal Processing (DSP) and Machine Learning (ML) tasks.
   (2) Capable of efficiently executing compute-intensive operations such as Fast Fourier Transform (FFT) and matrix computation.

2. Diverse I/O Interfaces:
   (1) Supports digital microphones (DMIC) and Micro-Electro-Mechanical Systems (MEMS MIC), providing hardware support for high-precision voice data acquisition.
   (2) Richly equipped with GPIO, UART, and I2S interfaces to meet various application needs.

3. Ease of Development:
   (1) Provides a complete Software Development Kit (SDK) and development tools, supporting rapid deployment and testing.
   (2) Compatible with Keil, ARM CMSIS-DSP, and other toolchains, enabling developers to get started quickly.

Application Value:

In voice signal processing, the Nuvoton M55M1 development board supports multiple operations, including signal preprocessing, feature extraction, and model inference. Its outstanding performance makes it the core hardware platform of this project, providing a solid foundation for implementing noise reduction and keyword recognition functions.

### 3.1.4 Overview of Spectral Subtraction and Keyword Recognition Technologies

1. Spectral Subtraction

   Spectral subtraction is a classical noise reduction technique in audio signal processing. Its core idea is to estimate the spectrum of background noise and subtract it from the noisy speech spectrum to achieve noise suppression.

   The specific implementation involves the following steps:

   (1) Background Noise Modeling: Use the spectral characteristics of environmental background noise as a reference to generate a noise template.
   (2) Spectral Subtraction: Subtract the estimated noise spectrum from the speech spectrum to eliminate noise components.
   (3) Signal Reconstruction: Use Inverse Fast Fourier Transform (IFFT) to reconstruct a clean time-domain signal.

2. Keyword Recognition

   Keyword recognition is a crucial application in speech processing, aiming to detect specific speech commands (e.g., “turn on the light”) from continuous audio streams.

   The implementation process includes:

   (1) Voice Feature Extraction: Extract features such as Mel-Frequency Cepstral Coefficients (MFCC) from the speech signal.
   (2) Model Inference: Feed the extracted features into a pre-trained neural network model for recognition.
   (3) Application Trigger: Trigger the corresponding command or operation based on the recognition result.

3. Advantages of Technology Integration

   The integration of spectral subtraction and keyword recognition effectively eliminates noise interference in industrial or home environments, enhances the accuracy of voice command recognition, and reduces the system's false activation rate.

### 3.1.5 Features and Advantages of ARM® Helium Technology

ARM® Helium is a vector extension technology (M-Profile Vector Extension, MVE) specifically designed for the Cortex®-M processor series. It aims to significantly enhance computational performance in embedded devices, particularly in the areas of digital signal processing (DSP) and machine learning (ML).

Key Features of Helium Technology

1. High-Efficiency Computation
   (1) Supports vectorized operations, enabling simultaneous processing of multiple data elements, significantly accelerating DSP tasks such as FFT and filtering.
   (2) Provides substantial optimization for matrix operations used in noise reduction algorithms such as spectral subtraction.
2. Low Power Design
   (1) Tailored for embedded devices, it improves computational performance while maintaining low power consumption.
   (2) Suitable for long-running smart devices, such as voice-controlled systems.
3. Wide Application Range
   (1) Supports various application domains including audio processing, machine learning inference, and image processing.
   (2) Meets the needs of multiple sectors, including smart homes, industrial automation, and medical devices.

Application in This Project

In this project, Helium technology is utilized to optimize the computational efficiency of spectral subtraction and MFCC feature extraction. It significantly reduces processing latency while ensuring real-time system performance. This enables the M55M1 development board to execute voice processing tasks efficiently, making it an ideal platform for embedded voice applications.

Summary

Chapter One provides a comprehensive framework—from project background and foundational technologies to the hardware platform—establishing clear objectives and references for subsequent development processes and technical implementations.

## 3.2 Development Process Overview

This project centers on achieving effective noise reduction and accurate keyword recognition. Through a well-defined hardware architecture and software workflow, it ensures tight coordination among functional modules and stable system operation.

### 3.2.1 System Architecture Design

The system architecture is divided into two main sections: hardware architecture and software workflow. Each component handles voice signal acquisition, processing, and output, forming a complete pipeline from data collection to command triggering.

Hardware Architecture Design

1. Microphone Input Module
   (1) Function: Uses a Digital Microphone (DMIC) to capture speech signals, ensuring high-quality and stable input data.
   (2) Value: DMIC effectively reduces noise issues common in analog microphones, improving signal clarity and fidelity.
2. Core Processing Module
   (1) Function: Uses the Nuvoton M55M1 development board as the main controller to execute voice data collection, noise reduction, and keyword recognition.
   (2) Feature: The M55M1 supports ARM® Helium technology for efficient computation and fast execution of complex algorithms.
3. Output Module
   (1) Function: Outputs recognition results via terminal; users can use these results to drive external devices.
   (2) Application Scenarios: Includes smart home command triggers, industrial device control, etc.

Software Workflow Design

The software workflow is composed of three primary stages to ensure voice data is processed from raw input to command execution:

1. Audio Preprocessing

The captured speech signal is segmented and converted into the frequency domain using Short-Time Fourier Transform (STFT), laying the foundation for noise reduction and feature extraction.

2. Noise Reduction

   Spectral subtraction is applied to eliminate background noise, retaining effective frequency components in the voice signal and producing a clean audio stream.

3. Keyword Recognition

   The noise-reduced signal is used to extract speech features (e.g., MFCC) and fed into a pre-trained deep learning model for classification and accurate voice command recognition.

### 3.2.2 Hardware and Software Requirements List

To ensure smooth project execution, the following resources are required:

Hardware Requirements

1. Nuvoton M55M1 Development Board

   Serves as the main control unit with stable processing performance and rich I/O support. It is the core hardware platform of this project.

2. Digital Microphone (DMIC)

   Responsible for capturing high-quality speech signals and providing consistent input.

3. Terminal or Display Device

   Displays the results of noise reduction and recognition, allowing users to verify system performance.

4. Power Supply Module

   Provides stable power to the hardware system to ensure reliable long-term operation. Includes UART interface for communication with PC terminals.

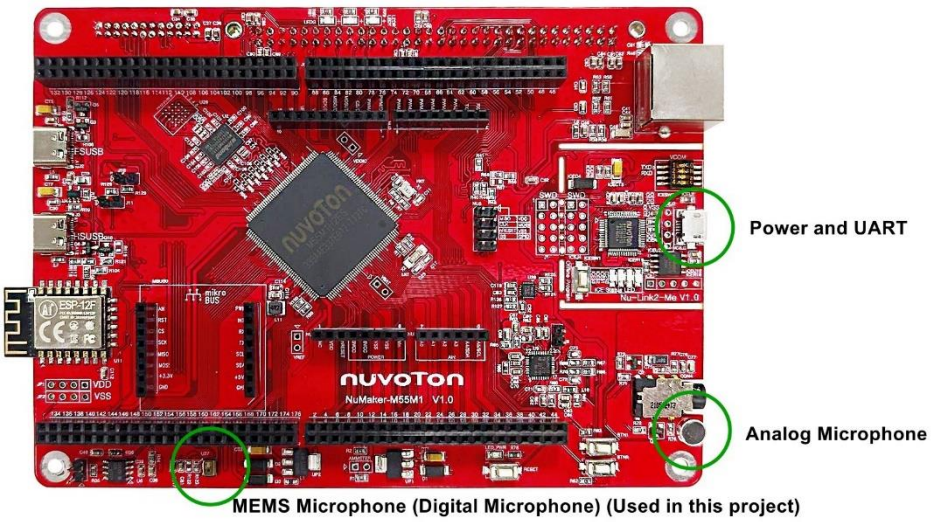


Figure: Related I/O Modules in the Project

Software Requirements

1. Keil MDK

   The primary C development environment, suitable for programming and compiling code for the M55M1 development board.

2. Python Environment

   Used to simulate spectral subtraction for validating algorithm feasibility and parameter tuning.

3. ARM® DSP Library

   Provides efficient digital signal processing tools to accelerate calculations for STFT, spectral subtraction, and feature extraction.

4. Terminal Tools

   e.g., PuTTY or Tera Term, used for system interaction, debugging, and data monitoring.

Development Procedure

To systematically complete the project, the development steps are structured as follows:

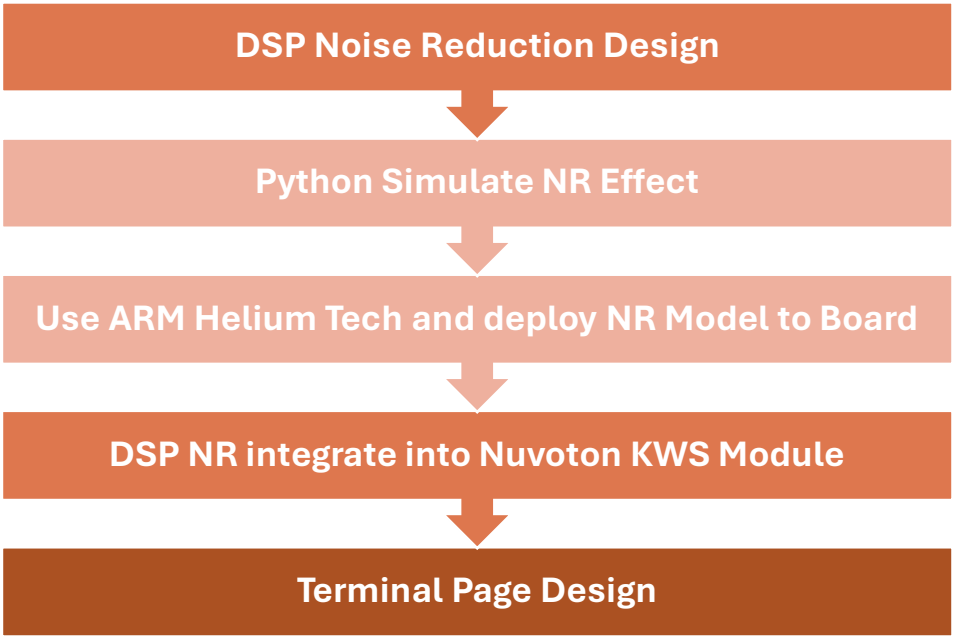


Figure: System Development Flow

Summary

Chapter 2 provides a detailed explanation of the system design and development process. Both hardware and software aspects are meticulously planned. The modular design and clearly defined steps offer an efficient development path, laying a solid foundation for the practical implementation in subsequent chapters.

## 3.3 Theoretical Foundation of Spectral Subtraction

Spectral subtraction is a classical speech denoising technique widely used in speech signal processing under noisy environments. This chapter provides a detailed explanation of the signal model, core algorithm principles, and the challenges and solutions encountered during its application.

### 3.3.1 Signal Model of Spectral Subtraction

The fundamental concept of spectral subtraction is to model the spectral characteristics of background noise and subtract them from the noisy speech spectrum to restore a clean speech signal.


Figure: Conceptual Diagram of Spectral Subtraction

Mathematical Representation of the Signal Model

In the time domain, a noisy speech signal, y(t) can be represented as:

$$y(t)=s(t)+n(t)$$

Where:

- y(t): Noisy speech signal
- s(t): Clean speech signal
- n(t): Background noise signal

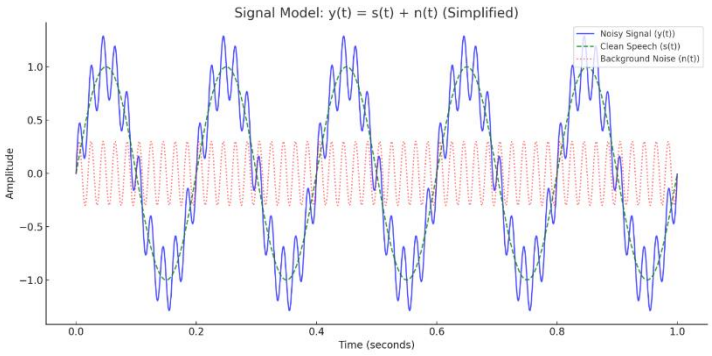


Figure: Illustration of y(t)=s(t)+n(t)

By applying Fourier Transform to convert the signal into the frequency domain, the model becomes:

$$Y(f)=S(f)+N(f)$$

Where:

- Y(f): Spectrum of the noisy speech
- S(f): Spectrum of the clean speech
- N(f): Spectrum of the background noise

Application Value

The core advantage of this denoising algorithm lies in its simplicity and efficiency, making it particularly suitable for speech denoising in embedded systems.

### 3.3.2 Introduction to STFT (Short-Time Fourier Transform)

1. Characteristics of Speech Signals

   Speech signals are time-varying signals whose frequency components change over time. Performing a Fourier Transform on the entire speech segment only provides spectral information over the entire duration and fails to capture frequency changes over time.

2. Principles and Advantages of STFT

   Short-Time Fourier Transform (STFT) performs Fourier Transforms on segmented and windowed portions of the signal, enabling observation of frequency changes over time.

   Mathematical Expression:

   For a signal, x(t), its STFT is defined as:

$$X(t,f) = \int_{-\infty}^{\infty} x(\tau) \cdot w(t-\tau) \cdot e^{-j2\pi f}\, \tau d\tau$$

Figure: STFT Mathematical Formula

   Where:

   - w(t): Window function, used to extract short-time signals
   - X(t,f): Spectrum at time t and frequency f

3. STFT Procedure
   (1) Segmentation: Divide the signal into fixed-length segments (e.g., 20 ms).
   (2) Windowing: Multiply each segment by a window function (e.g., Hamming Window) to reduce distortion.

(3) Fourier Transform: Compute the spectrum for each segment to obtain time-varying frequency information.

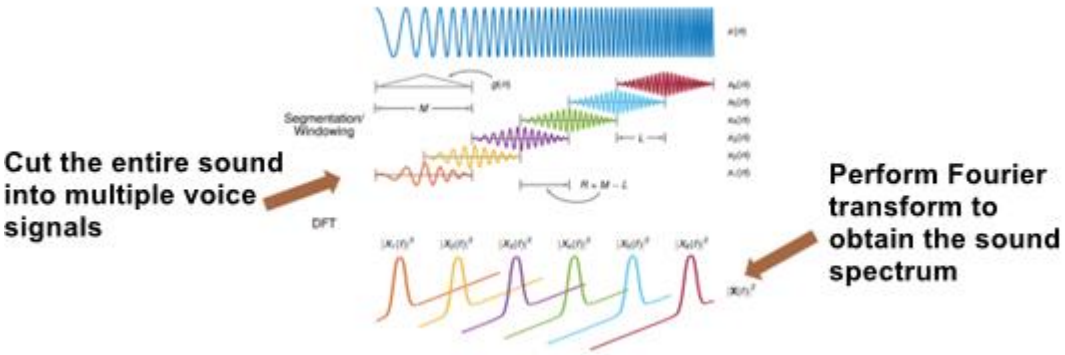


Figure: STFT Illustration Diagram

4. Technical Advantages
   (1) High Computational Efficiency: Simple spectral subtraction operations are well-suited for real-time processing.
   (2) Strong Applicability: Applicable in various noisy environments, particularly in scenarios with stable background noise.

### 3.3.3 Challenges of Spectral Subtraction in Real-Time Processing

Although spectral subtraction is theoretically simple and effective, several challenges must be addressed in practical applications.

1. Accuracy of Noise Spectrum Estimation

   The background noise spectrum may vary over time, reducing the applicability of a fixed noise template.

   Solution:

   Implement a dynamic update mechanism to re-record the noise spectrum at regular intervals and introduce adaptive algorithms to adjust the template based on environmental changes.

2. Speech Quality Degradation Due to Subtraction

   Over-subtraction may lead to loss of valid speech components, causing artifacts like “robotic” or “metallic” sounds.

   Solution:

   Precisely adjust the scaling factor $\alpha$, to balance denoising performance with speech quality, and apply spectral smoothing techniques to reduce auditory discontinuities.

3. Real-Time Processing Constraints

   In embedded systems, STFT and spectral calculations may cause processing delays, affecting real-time performance.

   Solution:

   Utilize ARM® Helium technology to optimize FFT and matrix computations, significantly improving processing efficiency.

Summary

Spectral subtraction, as a simple yet effective denoising technique, is widely used in speech signal processing due to its efficient frequency-domain operations. This chapter thoroughly explained the principles of spectral subtraction from mathematical modeling to algorithmic details, while also analyzing real-world challenges and corresponding solutions—laying a solid theoretical foundation for subsequent system implementation.

## 3.4 Python-Based Denoising Simulation

To validate the feasibility of spectral subtraction before deploying on embedded hardware, this chapter uses Python to build a simulation environment, completing the entire process from background noise modeling to denoising performance evaluation. Python's powerful audio processing tools and extensive dataset support make the simulation process efficient and intuitive.

### 3.4.1 Purpose and Value of Python Simulation

In real-world speech signal processing, embedded hardware development requires extensive testing and tuning. Using Python for simulation offers the following advantages:

1. Rapid Validation of Theoretical Feasibility

   Although spectral subtraction is mathematically grounded, its effectiveness in real-world scenarios must be validated. Python provides a rapid prototyping environment, allowing for convenient testing of denoising performance under different parameter combinations.

2. Simplified Audio Processing

   With tools such as Librosa and Scipy, audio loading, analysis, and processing can be easily accomplished without manually implementing complex functions.

3. Visualization and Parameter Tuning

   Using visualization libraries such as Matplotlib, waveform and spectrograms before and after denoising can be displayed, allowing designers to visually compare performance and adjust parameters (e.g., scaling factor α).

### 3.4.2 Overview of Relevant Python Libraries

The following Python tools are used for key tasks in this chapter:

1. Librosa
   - Provides convenient audio loading and Short-Time Fourier Transform (STFT) functionality.
   - Supports spectral processing and reconstruction, suitable for denoising and feature analysis.
2. Soundfile
   - Enables efficient reading and saving of various audio formats (e.g., WAV, FLAC).
   - Suitable for handling large-scale audio data with simplicity and high efficiency.
3. Scipy
   - Offers mathematical tools such as Fourier Transform and filter design for preprocessing.
   - Useful for spectrum analysis and computation during the denoising process.
4. Matplotlib
   - Used for plotting waveforms, spectrograms, and other visualization charts.

- Assists in showcasing denoising performance and visually comparing processed vs. raw audio.

### 3.4.3 Python Implementation

1. Implementation Flow Overview

   Step 1: Collect background noise as the baseline.

   Step 2: Perform STFT on speech signal to convert it into the frequency domain.

   Step 3: Subtract the noise spectrum from the speech spectrum.

   Step 4: Apply IFFT (Inverse Fourier Transform) to reconstruct the denoised signal.

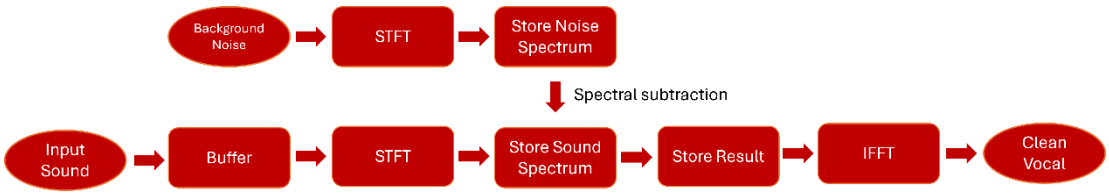


Figure: Overview of Python Implementation Workflow

2. Dataset Selection and Processing

   To simulate real-world and diverse environments, the following public datasets can be used for testing:

   (1) Mozilla Common Voice (MCV):

   Provides diversified voice samples suitable for speech processing tests.

   (2) UrbanSound8K:

   Offers urban noise data such as vehicle sounds and sirens, simulating background noise.

   Mixed samples generated by combining UrbanSound8K noise with MCV speech are used as input for spectral subtraction testing.

3. Background Noise Collection

   Accurate modeling of background noise spectrum is critical for spectral subtraction, directly impacting denoising effectiveness.

Step 1: Record an audio file containing background noise, assuming the first 0.5 seconds contain only noise.

Step 2: Use librosa.load to load the audio file and extract the noise segment.

Step 3: Analyze the noise segment's spectrum to generate a noise template.

```
# 讀取音頻文件
input_file = 'mix_noisy.wav'
output_file = 'denoise_spectral.wav'

# 加載音頻
y, sr = librosa.load(input_file, sr=None)
```

Figure: Loading a WAV Audio File

4. STFT Spectrum Computation

STFT converts time-domain signals into the frequency domain, useful for analyzing speech and noise frequency components.

Step 1: Use Librosa's stft function to compute the spectra of speech and noise.

Step 2: Decompose the spectrum into magnitude and phase for further spectral operations.

```
# compute STFT
y_stft = librosa.stft(y)
magnitude, phase = np.abs(y_stft), np.angle(y_stft)
```

Figure: Computing STFT with librosa.stft

5. Applying Spectral Subtraction

In the frequency domain, subtract the background noise spectrum from the noisy speech spectrum.

Step 1: Calculate the average of the background noise spectrum.

Step 2: Use the spectral subtraction formula with scaling factor α to control denoising strength.

```
# adjust noise reduction factor
noise_reduction_factor = 2.5  # alpha
mean_noise_spectrum = np.mean(np.abs(noise_stft), axis=1) * noise_reduction_factor
```

Figure: Noise Spectrum Calculation

6. IFFT Audio Reconstruction

   After denoising, reconstruct the time-domain signal using inverse STFT (ISTFT).

   Step 1: Combine the denoised magnitude with the original phase.

   Step 2: Use librosa.istft to reconstruct the waveform and save it as an audio file.

```
# spectral subtraction
threshold_factor = 1.0  # spectral subtraction threshold
magnitude_denoised = np.maximum(magnitude - (mean_noise_spectrum[:, np.newaxis] * threshold_factor), 0)

# restore phase information
y_stft_denoised = magnitude_denoised * np.exp(1j * phase)

# ISTFT to restore audio
y_denoised = librosa.istft(y_stft_denoised)

# save denoised audio
sf.write(output_file, y_denoised, sr)
```

Figure: Computing ISTFT with librosa.istft

### 3.4.4 Comparison of Denoising Results and STOI Evaluation

1. Visual Comparison

   Use Matplotlib to plot waveforms and spectrograms before and after denoising for intuitive comparison.

2. STOI Evaluation

   The Short-Time Objective Intelligibility (STOI) score is an important metric for evaluating speech intelligibility. The score ranges from 0 to 1, with higher values indicating better intelligibility.

```
# ensure both audio files have the same length
min_length = min(len(clean), len(denoised))
clean = clean[:min_length]
denoised = denoised[:min_length]

# compute STOI score
stoi_score = stoi(clean, denoised, sr_clean, extended=False)

print(f"STOI: {stoi_score:.2f}")

✓ 5.3s

STOI: 0.99
```

Figure: Short-Time Objective Intelligibility (STOI)

Summary

This chapter provided a detailed explanation of implementing spectral subtraction denoising in Python, from foundational theory to full process execution. Visualization and STOI evaluation validated the denoising effectiveness. This simulation process provides theoretical verification and parameter tuning references for implementation on embedded hardware, laying a solid foundation for the subsequent deployment phase.

## 3.5 C Implementation on the M55M1 Development Board

This chapter provides a detailed explanation of implementing spectral subtraction denoising on the M55M1 development board, from simplified architecture design to advanced structural optimization. The chapter emphasizes the core role of Helium technology in implementation and highlights the advantages of modular program design.

### 3.5.1 Application of Helium Technology in Denoising

Helium is ARM®'s M-Profile Vector Extension (MVE) technology, specifically designed to accelerate DSP and machine learning tasks. It features high efficiency and low power consumption. Its applications in spectral subtraction denoising primarily include the following aspects:

1. Efficient FFT Computation
    - Utilize the Arm®_rfft_fast_f32 function from the ARM® DSP library to perform STFT and IFFT computations rapidly.
    - Helium supports vectorized operations, enabling parallel processing of multiple data elements to significantly accelerate Fourier transform calculations.
2. Low-Power Operation
    - Helium's architecture is optimized for energy efficiency on embedded devices, making it ideal for prolonged operation on resource-constrained M55M1 boards.
3. Support for Vector Operations

- The matrix operations involved in spectral subtraction (e.g., spectral subtraction) are efficiently executed using Helium's SIMD (Single Instruction Multiple Data) capabilities.

4. Optimized Data Access
   - Helium provides efficient memory management features to reduce frequent data load/store operations and further improve processing performance.

### 3.5.2 Spectral Subtraction Architecture

Architecture Overview

In embedded systems, computational resources and memory are limited. Therefore, a simplified spectral subtraction architecture must be designed to ensure denoising effectiveness while minimizing computational complexity.

Characteristics of the Simplified Architecture:

1. Modular Processing
   - The overall process is divided into several modules, including audio input, STFT, spectral subtraction, IFFT, and audio output, facilitating development, testing, and debugging.
2. Fixed Parameter Configuration
   - Employ fixed configuration parameters (e.g., window size, overlap ratio) to reduce computational load and enhance implementation efficiency.
3. Focus on Core Functionality
   - Avoid overly complex denoising algorithms, focusing on background-noise-based spectral subtraction to ease embedded implementation.

Processing Flow:

1. Audio Input
   - Acquire noisy speech data and store it in a buffer.
2. STFT Spectrum Computation
   - Apply Short-Time Fourier Transform to convert time-domain signals into frequency-domain signals.
3. Spectral Subtraction
   - Subtract background noise spectra from speech spectra.

4. IFFT Spectrum Reconstruction
    - Use Inverse FFT to convert the frequency-domain signal back to the time domain.
5. Audio Output
    - Output the processed speech signal, resulting in denoised audio data.

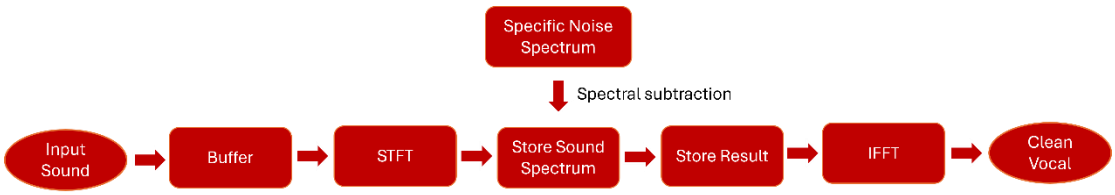


Figure: Spectral Subtraction Processing Flow

### 3.5.3 Spectral Subtraction Implementation

1. Initialization and Parameter Configuration

    Initialization is the first step in implementing spectral subtraction. It includes configuring the window size, buffers, and FFT processor. This ensures all resources are allocated and ready for processing.

    Initialization Steps:

    (1) Define FFT window size and overlap ratio.
    (2) Initialize the Helium FFT function library.
    (3) Allocate buffers to store speech data, spectral data, and results.

    Example Code:

```
55  // todo: STFT denoising function
56  void ApplySTFTDenoising(const std::vector<int16_t>& inputAudio, std::vector<int16_t>& outputAudio) {
57      size_t frameSize = 1024;  // frame size
58      size_t hopSize = frameSize / 2;  // overlap size
59      std::vector<float> frame(frameSize);  // store time-domain frame data
60      std::vector<float> fftOutput(frameSize);  // store FFT result
61
62      // initial FFT
63      arm_rfft_fast_instance_f32 fftInstance;
64      arm_rfft_fast_init_f32(&fftInstance, frameSize);
```

Figure: STFT Initialization Code

2. STFT and Spectrum Computation

    STFT is the key step that converts speech signals from the time domain to the frequency domain. Helium's optimized FFT functions can significantly accelerate this step.

Implementation Steps:

(1) Segment the speech signal using a fixed frameSize with overlap defined by hopSize.
(2) Apply a Hamming window to each frame to reduce spectral leakage.
(3) Use Arm®_rfft_fast_f32 to compute the FFT of each segment.

Example Code:

```
69      for (size_t i = 0; i + frameSize <= inputAudio.size(); i += hopSize) {
70          // 1. convert the current frame of audio data to float
71          for (size_t j = 0; j < frameSize; ++j) {
72              frame[j] = static_cast<float>(inputAudio[i + j]);
73          }
74          // 2. apply Hamming window
75          for (size_t j = 0; j < frameSize; ++j) {
76              frame[j] *= 0.54f - 0.46f * cos(2 * PI_VALUE * j / (frameSize - 1));  // Hamming window
77
78          // 3. perform FFT
79          arm_rfft_fast_f32(&fftInstance, frame.data(), fftOutput.data(), 0);
```

Figure: Calling arm_rfft_fast_f32 to Perform FFT

3. Spectral Subtraction Processing

The purpose of spectral subtraction is to remove the noise spectrum from the speech spectrum. Helium technology optimizes this step for speed and efficiency.

Implementation Steps:

(1) Compute the average background noise spectrum as a noise template.
(2) Subtract the noise spectrum from the speech spectrum, preserving only the speech content.
(3) Apply non-negative constraint to avoid signal distortion.

Example Code:

```
80          // 4. spectral subtraction: subtract the current frame's spectrum from the background spectrum
81          float noiseThreshold = 0.1f;  // set a noise threshold
82          for (size_t j = 0; j < fftOutput.size(); ++j) {
83              if (fftOutput[j] < noiseThreshold) {
84                  fftOutput[j] = 0;  // set the value to zero if it is below the threshold
85              }
86          }
```

Figure: Executing Spectral Subtraction

4. IFFT and Signal Reconstruction

After spectral subtraction, IFFT is used to reconstruct the time-domain signal. The final denoised speech signal is generated using overlap-add.

Implementation Steps:

(1) Use Arm®_rfft_fast_f32 to convert the spectrum back to the time-domain signal.
(2) Store the reconstructed signal in the output buffer and perform overlap-add to smooth transitions.

Example Code:

```
88          // 5. perform inverse FFT (iFFT) to convert the spectrum data back to time domain
89          arm_rfft_fast_f32(&fftInstance, fftOutput.data(), frame.data(), 1);
90          // 6. copy the reconstructed frame data back to the output, perform overlap-add
91          for (size_t j = 0; j < frameSize; ++j) {
92              outputAudio[i + j] += static_cast<int16_t>(frame[j]);
93          }
```

Figure: Calling arm_rfft_fast_f32 to Perform IFFT

## 3.6 Integration of the Keyword Spotting Module

This chapter describes the implementation of keyword spotting (KWS) on the M55M1 platform and the integration of the denoising module to enhance recognition accuracy. Modular design allows the system to adapt to various environmental challenges, ensuring stable keyword detection performance.

### 3.6.1 Overview of the Keyword Spotting Model

By applying speech signal processing techniques, the system can detect target keywords in real time and trigger corresponding actions.

Module Workflow:

1. Real-Time Audio Input

   Use a microphone to capture speech and convert it into digital signals for subsequent processing.

2. Buffered Input

   Segment the speech signal into fixed-length frames and store them in a buffer to minimize latency.

3. MFCC (Mel-Frequency Cepstral Coefficients) Extraction

Extract key frequency features from the speech signal to simplify model training and inference.

4. Keyword Inference

   Use a pre-trained model to classify the extracted features and determine whether the target keyword is present.

5. Display Inference Results

   Output the inference result in real-time and trigger subsequent application responses.


Figure: Keyword Spotting Model Operation Flow

### 3.6.2 Integration with the Denoising Module

Integrating the denoising module with the keyword spotting module improves recognition accuracy in noisy environments.

Integration Flow:

1. Background Noise Recording

   During system initialization, record background noise to generate a noise spectrum template.

2. Insertion of Denoising Module

   Insert the denoising module between the buffer and MFCC stages to ensure clean audio is input to the MFCC extractor.

3. Post-Denoising Processing

   The buffered data is high-quality, denoised audio ready for feature extraction.

4. Keyword Spotting Workflow

   Use the denoised audio for feature extraction and model inference.

Block One: Feature Extraction and Inference

Segment real-time audio and apply STFT denoising to enhance signal quality.

Step Analysis:

1. Acquisition of Audio Samples
   - Use DMICRecord_AvailSamples() to confirm sufficient sample availability.
   - Call DMICRecord_ReadSamples() to read audio data into the buffer.
2. Denoising Process
   - Call ApplySTFTDenoising() to apply STFT-based denoising to the data.
3. Segmented Sliding Window Processing
   - Apply a sliding window over the audio data for segment-based feature extraction.

Block Two: Feature Extraction and Inference

Function and Objective: Extract features (e.g., MFCC) from denoised audio and input them into the model for inference.

Step Analysis:

1. MFCC Feature Extraction:
   - Use the preProcess module to compute MFCC features for each segment and generate model-compatible input data.
2. Inference Execution:
   - Input the extracted features into the pre-trained model to determine if the target keyword is present.
3. Result Processing and Output:
   - If the inference result matches the target keyword, display a successful detection message and trigger related application functions.

Code Example:

```
584             info("Inference %zu/%zu\n", audioDataSlider.Index() + 1,
585                  audioDataSlider.TotalStrides() + 1);
586
587 #if defined(__PROFILE__)
588             u64StartCycle = pmu_get_systick_Count();
589 #endif
590
591             /* Run the pre-processing, inference and post-processing. */
592             if (!preProcess.DoPreProcess(inferenceWindow, arm::app::audio::KwsMFCC::ms_defaultSamplingFreq))
593             {
594                 printf_err("Pre-processing failed.");
595                 break;
596             }
```

Figure: Calling the MFCC Extraction Function

Key Explanations:

1. MFCC Extraction:
   - MFCC is a crucial feature for keyword spotting, designed to emulate the human auditory perception for capturing essential speech information.
2. Inference Model:
   - The inference process uses a pre-trained, lightweight CNN model.

### 3.6.3 End-to-End Execution Flow Summary

Step-by-Step Description:

1. Keyword Detection Initialization:
   - The system begins real-time audio recording upon user confirmation.
2. Real-Time Audio Denoising:
   - Segment and denoise the recorded audio using STFT and spectral subtraction to remove background noise.
3. MFCC Feature Extraction:
   - Extract MFCC features from the denoised audio segments as model input.
4. Keyword Inference and Result Handling:
   - Use the model to detect the specified keyword and display the inference result while triggering application functions.

### 3.6.4 Module Design Summary

The keyword spotting module, combined with denoising and speech feature extraction, significantly improves detection accuracy and system stability. Its modular design ensures good scalability and adaptability across various use cases.

## 3.7 Program Architecture and Optimization Strategies

This chapter systematically analyzes the overall program architecture, detailing the complete workflow from background noise processing to keyword spotting (KWS). It further examines the role and technical implementation of each module.

Finally, optimization and debugging strategies are proposed to further enhance system performance and stability.

### 3.7.1 Overall Program Architecture Design

1. Architecture Overview

   The program is built around three core functionalities: background noise processing, real-time speech denoising, and keyword spotting. Through a modular design, the system achieves efficient coordination between components. The main stages of the overall architecture are:

   (1) Background Noise Processing:

   Record and analyze background noise to generate a spectral template.

   (2) Real-Time Speech Denoising:

   Apply noise suppression to real-time audio recordings to enhance speech quality.

   (3) Keyword Spotting:

   Extract speech features and feed them into a deep learning model to detect the target keyword.

2. Architecture Advantages
   - Modular Design:

     Functionalities are separated into independent modules, enabling ease of development and debugging.

   - Real-Time Performance:

     Accelerated processing based on ARM® Helium technology enables low-latency audio processing.

   - High Adaptability:

     Capable of adapting to diverse noise environments and speech scenarios.

### 3.7.2 Roles and Implementation Details of Each Module

1. Real-Time Audio Processing and Denoising Module

   (1) Role:

Process real-time audio and remove background noise.

(2) Technical Implementation:
   a. Apply a sliding window technique to segment audio while maintaining data continuity.
   b. Use a spectral subtraction algorithm optimized with Helium-accelerated FFT computation.

(3) Advantage:

Improves the speech signal-to-noise ratio (SNR) and reduces noise interference in keyword spotting.

2. Keyword Spotting Module

(1) Role:

Extract speech features and perform inference to detect the target keyword.

(2) Technical Implementation:
   a. Use MFCC (Mel-Frequency Cepstral Coefficients) to extract core speech features.
   b. Perform inference using a pre-trained deep learning model such as DS-CNN.

(3) Advantage:

High recognition accuracy, suitable for real-time speech recognition in embedded environments.

3. UART Interaction Module

(1) Role:

Facilitate user interaction to control background noise recording and keyword detection initiation.

(2) Technical Implementation:
   - Implement user command reception and feedback via the UART interface.
   - Include input error detection to improve system stability.

(3) Advantage:

Simple operation, suitable for human–machine interaction (HMI) in embedded systems.

### 3.7.3 Optimization and Debugging Strategies

1. Denoising Module Optimization
   (1) Dynamic Background Spectrum Update:
   Periodically record background noise to adapt to environmental changes.
   (2) Spectral Smoothing:
   Use smoothing algorithms to reduce the impact of abrupt frequency changes on speech quality.
2. Improving Keyword Spotting Accuracy
   (1) Enhanced Feature Extraction:
   Adjust MFCC parameters to extract more precise features.
   (2) Data Augmentation for Model Training:
   Add training samples with varied background noise and speech rates to improve model generalization.
3. System Performance Tuning
   (1) Computation Efficiency Improvements:
      a. Use Helium acceleration for FFT, matrix computation, and other high-frequency operations.
      b. Reduce memory copy operations by leveraging pointer and reference usage.
   (2) Time and Resource Management:
   Log execution times for each module to locate performance bottlenecks.

### 3.7.4 Logical Flow of System Operation

1. Initialization

   Upon startup, the system prompts the user to record background noise, then calculates and stores the spectral template.

2. Real-Time Audio Processing

   Capture real-time audio, segment it with a sliding window, and apply spectral subtraction for denoising.

3. Keyword Spotting

   Extract features from denoised audio and perform model inference to check for the target keyword.

4. Result Output

If the target keyword is detected, display a success message on the terminal and trigger the application function.

### 3.7.5 Future Expansion Directions

1. Further Model Optimization:

   Adopt more efficient model architectures (e.g., MobileNet) to balance performance and accuracy.

2. Support for More Voice Commands:

   Expand the model's recognition scope to support multiple languages and command sets.

3. Module Function Expansion:

   Add noise classification capabilities to further improve the adaptability of the denoising module.

Conclusion

This system integrates core technologies including background noise recording, real-time audio processing, keyword spotting (KWS), and speech denoising, fully leveraging the potential of embedded systems in speech processing applications. Through optimization using the ARM® Helium instruction set, the system achieves efficient Short-Time Fourier Transform (STFT) and spectral subtraction processing. Combined with MFCC (Mel-Frequency Cepstral Coefficient) feature extraction and machine learning models, it successfully delivers high-accuracy keyword recognition.

During the design process, emphasis was placed on a modular system architecture and a logically structured workflow, ensuring code readability and maintainability. At the same time, by fine-tuning the denoising module and improving inference accuracy, the system adapts to diverse noisy environments, thereby enhancing its practicality and robustness.

Looking ahead, this architecture can be further enhanced by integrating more advanced denoising algorithms—such as deep learning-based speech enhancement—alongside more efficient inference models. This would enable its expansion into a wider range of speech-related application scenarios, including voice assistants, smart home control, and far-field speech recognition. Ultimately,

this design sets a benchmark for embedded systems in intelligent speech applications and provides a solid foundation for future research.

References

1. STFT – MathWorks

   https://ww2.mathworks.cn/help/signal/ref/stft.html

# 4 Smart Healthcare 1 – Drug Recognition

## 4.1 Case Introduction – Drug Recognition

Background:

In pharmaceutical manufacturing plants or medical environments, drug recognition is a critical requirement.

1. Accuracy and Efficiency

   (1) Drug recognition is directly related to patient health and safety. In large-scale production, traditional methods can no longer meet operational demands.
   (2) Leveraging intelligent tools for automated recognition can significantly reduce human error and improve efficiency.

2. Challenges of Traditional Methods

   (1) Manual recognition is time-consuming and susceptible to fatigue or environmental interference, which can lead to errors.
   (2) Traditional methods cannot efficiently process large-scale drug data, creating the need for embedded AI-based intelligent solutions.

3. Example Usage Scenarios

   (1) Medical Applications:
      a. Provide medical professionals and patients with real-time drug identification to prevent health risks caused by incorrect medication.
      b. Support patient self-service query systems, where scanning a drug instantly displays detailed information such as dosage and precautions.

   (2) Pharmaceutical Manufacturing Plants:
      a. Real-time labeling and sorting of different drugs during production to prevent mixing.
      b. Integrating a machine learning system into the production line to monitor processes, reducing manual intervention and error rates.

4. Further Expansion

   The potential of an intelligent drug recognition system goes beyond recognition itself — it can be integrated with big data analytics to establish a traceable drug circulation management platform.

   Future enhancements could include voice-assisted functions to help visually impaired individuals identify drugs, improving accessibility and usability.

   (1) Support for Diverse Recognition:

   The system can recognize multiple drugs with various appearances, including capsules, tablets, and drugs of different colors and shapes.

   (2) High Precision and High Efficiency:
   a. Achieves a recognition accuracy rate of 95%, meeting the high standards of the medical and pharmaceutical industries.
   b. Equipped with high-speed processing capabilities to respond instantly to drug recognition demands in production and medical scenarios.

   (3) Embedded Hardware Support:

   Utilizes NuMicro® microcontrollers for processing, ensuring high performance and reliability in drug recognition.

   This hardware platform features low power consumption and high performance, making it suitable for embedded applications in medical and industrial domains.

5. Demonstration and Results
   (1) Tested Object Imaging:

   The system can accurately identify and label captured capsules and tablets, demonstrating high accuracy and reliability.

   (2) Expected Development Board LCD Output:

   On the embedded platform, an LCD displays drug recognition results in real time, labeling each drug's name and type for easy user verification.

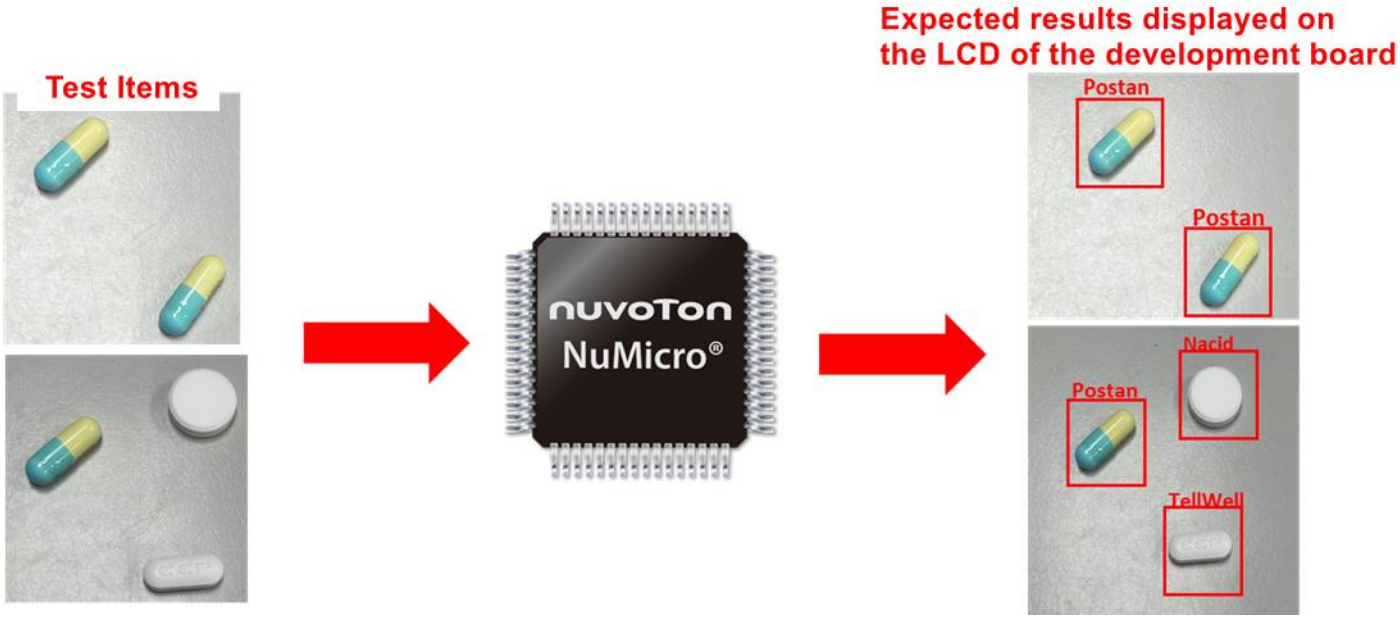


Figure: Drug identification shows result.

6. Dataset and Model Training Process

Tool Selection:

(1) LabelImg: An open-source image annotation tool for drawing bounding boxes and generating label files.
(2) Roboflow: An online tool that supports annotation, data augmentation, and exporting datasets in multiple formats for different model training pipelines.

Data Augmentation:

(1) Apply techniques such as rotation, cropping, and color transformation to the dataset to increase diversity and improve model generalization.

Model Training

(1) Model Selection:
    a. Utilize YOLOX Nano as the base model — a lightweight object detection model optimized for embedded applications.
    b. Offers a balanced trade-off between speed and accuracy, suitable for deployment on resource-constrained embedded devices.
(2) Training Process:

a. Fine-tune the model with training data to adapt to the specific drug recognition scenario.
b. Apply data augmentation and regularization techniques during training to mitigate overfitting.

Model Framework Conversion

(1) From PyTorch to TensorFlow Lite:
a. Train the model in PyTorch, then perform an intermediate conversion to ONNX format.
b. Convert the ONNX model to TensorFlow Lite format for optimized performance and embedded device compatibility.

(2) Optimization and Testing:
a. Use TensorFlow Lite optimization tools (e.g., quantization, weight pruning) to further reduce model size and improve runtime efficiency.
b. Deploy and test the model on the embedded development board to validate accuracy and inference speed.

## 4.2 Dataset and Model Overview

### 4.2.1 Training Framework and Model

1. Training Framework:

   PyTorch is employed as the training framework, leveraging its flexibility and powerful model development capabilities to efficiently perform deep learning model training and fine-tuning.

2. Training Model:

   The training is based on the YOLOX-Nano model, a lightweight object detection model specifically designed for embedded devices. It offers high performance with low resource consumption.

### 4.2.2 Dataset Preparation and Annotation Method

1. Dataset Source:

   The dataset is sourced from the National Cheng Kung University Hospital drug dataset, containing diverse images of capsules and tablets to meet variability requirements.

2. Annotation Method:

   Annotations are performed using the LabelImg tool, generating label files in COCO JSON format to facilitate data loading and processing during model training.

3. Dataset:
   (1) Training Set: 1,500 images
   (2) Validation Set: 300 images

## 4.3 Training Environment and Hardware Overview

1. Hardware Equipment
   (1) NVIDIA RTX 3060 Ti GPU, providing high-performance GPU computing capabilities.
   (2) Supports CUDA technology, enabling full utilization of GPU acceleration for deep learning model training.

2. CUDA and PyTorch Versions
   (1) CUDA Version: 11.8
   (2) PyTorch Version: 2.4.1

3. Training Data

   The training dataset contains a total of 1,800 images, covering various drug appearance features.

4. Training Parameter Settings
   (1) Epochs: 200

(2) Batch Size: 32

5. Training Duration

   Each complete training session takes approximately 1.5 hours on average.

## 4.4 Model Training

### 4.4.1 Model Training Workflow

1. Environment Setup:

   Prepare the necessary software and hardware environment, including training frameworks (PyTorch, TensorFlow) and hardware (e.g., GPU).

2. Dataset Preparation:

   Use annotated datasets to ensure data accuracy and diversity for model training.

3. TinyML Training:

   Train the YOLOX-Nano model based on the PyTorch framework, generating the trained model.

4. Model Quantization:

   Convert the model using TensorFlow Lite, including INT8 quantization, to produce a lightweight model suitable for embedded environments.

5. Vela Compilation:

   Use the Vela compiler to further optimize the quantized model, generating a TFLite flatbuffer file to improve execution performance.

### 4.4.2 Model Training Flow Diagram Explanation

Step 1: YOLOX-Nano Training

(1) The dataset is trained using the PyTorch framework and a pre-trained model to generate the target model.
(2) Conversion from PyTorch to TensorFlow Lite.

Step 2: Quantization

Use TensorFlow Lite to perform INT8 quantization and generate a lightweight model.

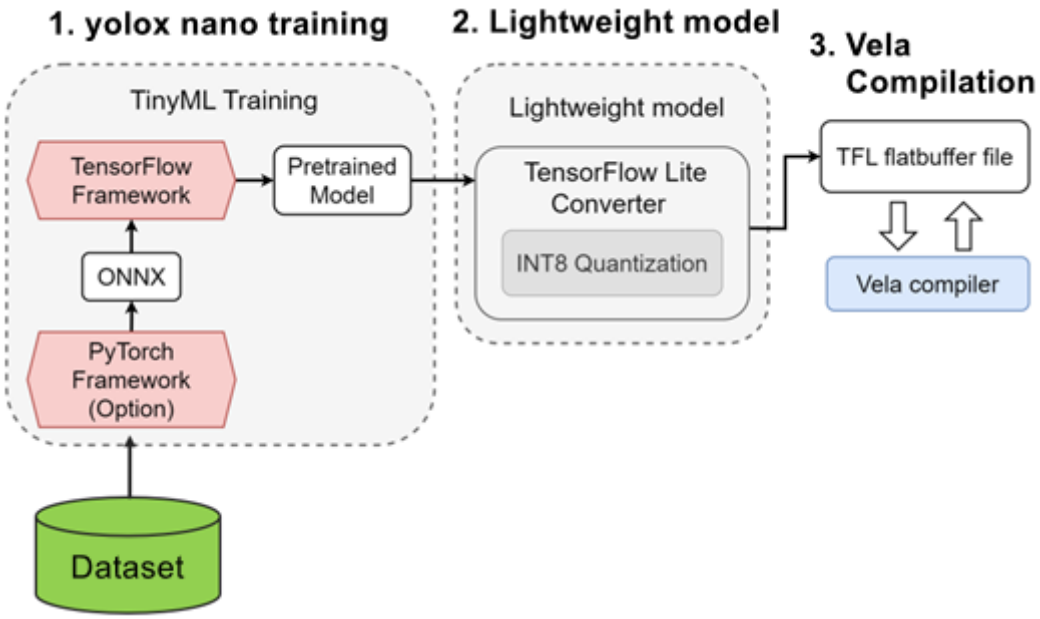


Figure: Model training flow chart

Step 3: Vela Compilation

Compile the model with Vela to produce a TFLite flatbuffer optimized for embedded devices.

### 4.4.3 Detailed Steps to Start Training

1. Create Python Environment:
   (1) Create a Python 3.10 environment named yolox_nu:

   ```
   conda create --name yolox_nu python=3.10
   ```

   (2) Activate the created environment:

   ```
   conda activate yolox_nu
   ```

   (3) This step creates an isolated environment in Anaconda for the project.

2. Upgrade pip:

   Upgrade pip and related packages:

   ```
   python -m pip install --upgrade pip setuptools
   ```

3. Install PyTorch:
   (1) Install the correct CUDA and PyTorch versions; example for CUDA 11.8:

   ```
   python -m pip install torch torchvision torchaudio --
   index-url https://download.pytorch.org/whl/cu118
   ```

   (2) Ensure the CUDA version (e.g., cu118) matches your hardware configuration.

4. Install MMCV:

   Choose proper version depends on your hardware. For CUDA 11.8 as an example:

   ```
   python -m pip install mmcv==2.0.1 -f
   https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/inde
   x.html
   ```

5. Install Other Required Packages:

   Navigate to the project directory and install required packages:

   ```
   python -m pip install --no-input -r requirements.txt
   ```

6. Install YOLOX:

   In the YOLOX project directory:

   ```
   python setup.py develop
   ```

### 4.4.4 YOLOX-Nano Dataset Organization

1. COCO JSON Format:

   The YOLOX-Nano dataset must follow the COCO format, including training and validation annotation files.

2. Directory Structure:
   (1) Ensure that data are put in directory as follows.

```
Datasets/<your_dataset_name>/
├── annotations/
│   ├── train_annotation_json_file
│   ├── val_annotation_json_file
├── train2017/
│   ├── train_img
├── val2017/
│   ├── validation_img
```

(2) `annotations/` stores JSON annotation files for training and validation.
(3) `train2017/` stores training images.
(4) `val2017/` stores validation images.

### 4.4.5 YOLOX-Nano Model Training Steps

1. Set Model Training Path:

   Edit exps/default/yolox_nano_ti_lite_nu.py.

2. Update Dataset Paths:

   ```
   self.data_dir = "datasets/your_coco"
   self.train_ann = "your_train.json"
   self.val_ann = "your_val.json"
   ```

3. Pre-trained Model Path:

   ```
   retrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti_lit
   e.pth
   ```

4. Start Training:

   ```
   python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
   <BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
   ```

   (1) `<MODEL_CONFIG_FILE>`: Model configuration path, e.g., yolox_nano_ti_lite_nu.py
   (2) `<BATCH_SIZE>`: Size of each training batch, e.g., 32
   (3) `<PRETRAIN_MODEL_PATH>`: Full path to the pre-trained model

### 4.4.6 Model Conversion

1. PyTorch to ONNX:

(1) Use the following command to convert the PyTorch model to ONNX format

```
python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c
<TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
```

(2) Parameter Description:
   a. <MODEL_CONFIG_FILE>: Model configuration file path, for example, yolox_nano_ti_lite_nu.py.
   b. <TRAINED_PYTORCH_MODEL>: Trained PyTorch model path.
   c. <ONNX_MODEL_PATH>: Output ONNX model storage path.

2. Create Calibration Data:
(1) Create calibration data to support quantification

```
python demo/TFLite/generate_calib_data.py --img-size
<IMG_SIZE> --n-img <NUMBER_IMG_FOR_CALI> -o
<CALI_DATA_NPY_FILE> --img-dir <PATH_OF_TRAIN_IMAGE_DIR>
```

(2) Parameter Description:
   a. <IMG_SIZE>: Input image size (e.g. 320).
   b. <NUMBER_IMG_FOR_CALI>: Number of images used for calibration.
   c. <CALI_DATA_NPY_FILE>: Output file path for calibration data.
   d. <PATH_OF_TRAIN_IMAGE_DIR>: Image directory for generating calibration data.

3. ONNX to TensorFlow Lite:
(1) Use the following command to convert the ONNX model to TensorFlow Lite format:

```
nx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
<CALI_DATA_NPY_FILE> "[[[[0,0,0]]]]"
```

(2) Parameter Description:
   a. <ONNX_MODEL_PATH>: The storage path of the ONNX model.
   b. <CALI_DATA_NPY_FILE>: The path of the calibration data file.

### 4.4.7 Vela Compilation

1. Prepare Quantized Model:

   Place the quantized model in `vela/generated/`.

2. Modify Variables File:

Open the `variables.bat` file in the vela directory and set the following:

```
set MODEL_SRC_FILE=<your tflite model>
set MODEL_OPTIMISE_FILE=<output vela model>
```

3. Execute Compilation:
   (1) Run the `gen_modle_cpp` script.
   (2) Output is generated at:
   `vela/generated/yolox_nano_ti_lite_nu_full_integer_quant_vela.tflite.cc`

## 4.5 C++ Software Implementation

### 4.5.1 C++ Software Functional Design

1. Overview of C++ Sample Code:

   Provides sample code based on the hardware platform, demonstrating the model inference workflow.

2. C++ Software Flowchart:

   Describes the complete process from data loading to model inference.

3. Introduction to C++ Software Design:

   Details the division of program modules, including the data processing module, model loading module, and inference module.

### 4.5.2 System Flowchart

1. Model Training (Blue Section)
   (1) TinyML Training
      a. Train the initial model using the PyTorch framework.
      b. Convert the trained model to ONNX format, then perform quantization (e.g., INT8 quantization) using TensorFlow Lite to generate a lightweight model.

   (2) Vela Compilation

Optimize the TensorFlow Lite model using the Vela tool to generate a compiled file (TFL flatbuffer file) suitable for embedded devices.

2. Hardware and Software Implementation (Red Section)
   (1) C/C++ Software Implementation
   Develop C/C++ software for the hardware platform to handle resource configuration and model deployment.

   (2) Hardware Development Board Programming
   Use a hardware development board (e.g., NuMicro® M55M1) integrating hardware accelerators (e.g., Ethos™-U55 microNPU) and CMSIS-NN optimized kernels to execute TensorFlow Lite Runtime.

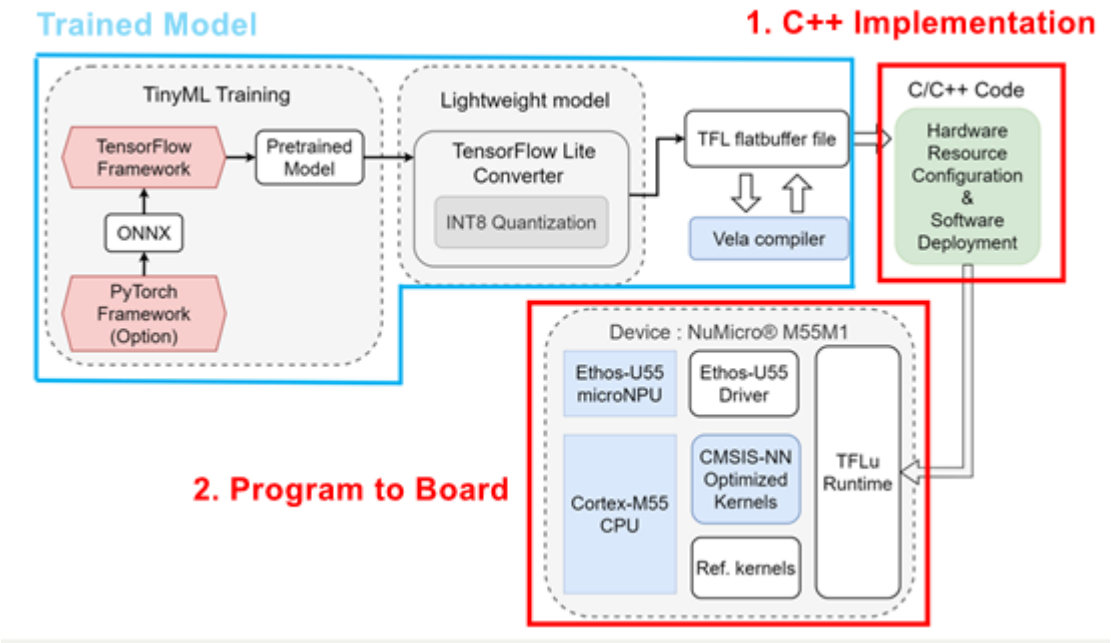


Figure: Overall System Flowchart

### 4.5.3 Verifying Development Board Hardware Requirements

1. LCD Module:
   (1) Displays the user interface, providing a visual representation of the drug recognition results.
   (2) Configured to communicate with the EBI (External Bus Interface) for data transfer.

(3) Supports multiple resolution settings to suit various application scenarios.

2. Image Sensor:
   (1) Captures images of the drug appearance for recognition and classification.
   (2) Resolution settings ensure recognition accuracy.
   (3) Configured to communicate via $I^2C$ (Inter-Integrated Circuit) for image data transfer and module configuration.


Figure: Development Board Module Usage

### 4.5.4 C++ Software Execution Flow

3. Initialization of Hardware, Model, and Framebuffer:

   Initialize hardware resources to ensure the execution environment is ready.

   (1) Configure resources required for model loading and inference.
   (2) Initialize the framebuffer to manage image data streams.

4. Three Functions Corresponding to Framebuffer States:
   (1) Image Capture: Use the camera module to capture drug images and store them in the framebuffer.
   (2) Image Preprocessing and Inference: Send the image data to the YOLOX-Nano model for inference to obtain recognition results.

(3) Model Result Post-Processing: Format the inference results and display them on the development board LCD.

### 4.5.5 Internal Software Flow on the Development Board

1. Initialization:
   (1) Hardware module initialization
   (2) Model initialization
   (3) Framebuffer initialization

2. Main Execution Process:
   (1) Image Capture: If the framebuffer is empty, capture an image and store it.
   (2) Model Inference: If the framebuffer is FULL, pass the image to YOLOXNano for processing to obtain inference results.
   (3) Post-Processing and Display: Format the results and display them on the LCD.

3. Loop Execution:

   In the main loop, reset the framebuffer and repeat the process.

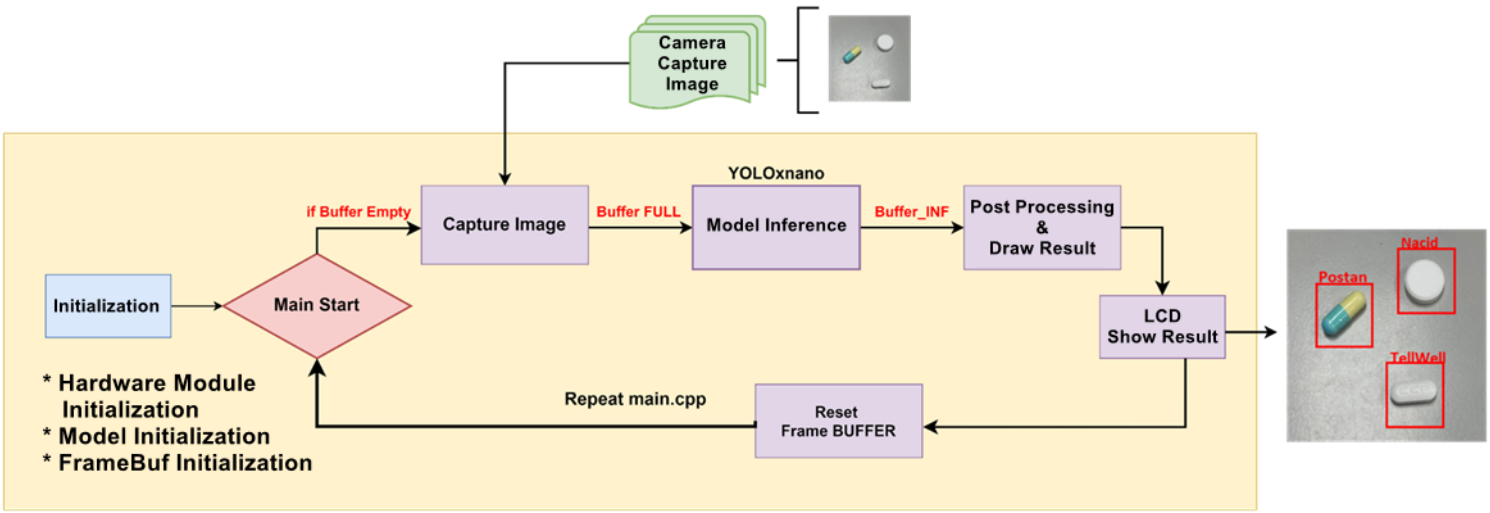


Figure: Flowchart of internal software of the development board

### 4.5.6 Image Processing Functions Overview

1. Camera Module Configuration (Image Sensor)
   (1) Setup Steps:

   In main.cpp, include ImageSensor.h to use the function prototypes defined in this header file.

2. ImageSensor Functional Overview:

   Function: int ImageSensor_Init()

   Initializes the image sensor, setting its frequency and timing parameters.

3. Function Definitions in ImageSensor.h:
   (1) ImageSensor_Init(void) – Initializes the image sensor.
   (2) ImageSensor_Capture() – Starts image capture and saves data to the specified buffer.
   (3) ImageSensor_Config() – Configures image format, width, and height.
   (4) ImageSensor_TriggerCapture() – Triggers an image capture operation.
   (5) ImageSensor_WaitCaptureDone() – Waits for capture completion.

4. ImageSensor – Capturing Images
   (1) ImageSensor_Capture() – Starts capturing images and stores them at a specified memory address.
   (2) ImageSensor_TriggerCapture() – Triggers image capture and sets the shutter signal.
   (3) ImageSensor_WaitCaptureDone() – Waits until image capture is complete and handles timeout conditions.

### 4.5.7 Input Image Processing

1. Framebuffer in main.cpp
   (1) In the main_task() function loop, the program continuously acquires new frames from the camera.
   (2) Uses the framebuffer to store captured frames before passing them to the YOLOX-Nano model for inference.
   (3) S_FRAMEBUF is used to manage both image data and detection results.

2. Framebuffer Functional Overview
   (1) eFRAMEBUF_EMPTY (Empty Buffer):
      a. When the framebuffer is empty, the main loop searches for an available buffer to store new image data.
      b. get_empty_framebuf() returns an empty buffer, after which:

(a) If using the image sensor (CCAP), triggers ImageSensor_Capture() to obtain new image data and store it in the buffer.
(b) If using static images, copies image data from memory into the buffer.

c. The buffer state is then updated to eFRAMEBUF_FULL for subsequent processing.

(2) eFRAMEBUF_FULL (Full Buffer):
a. When the framebuffer is full, the main loop checks and processes these buffers.
b. get_full_framebuf() returns a full buffer, after which:
(a) The image is scaled to the model's required input size (e.g., using imlib_nvt_scale()).
(b) If quantization is required, image data is quantized to integer format.
(c) The data is placed into the inference processing queue, and the buffer state is updated to eFRAMEBUF_INF.

(3) eFRAMEBUF_INF (Inference Buffer):
a. When the framebuffer is in inference state, the main loop processes completed inference results.
b. get_inf_framebuf() returns an inference buffer, after which:
(a) The inference results undergo post-processing (e.g., using DetectorPostprocessing) to draw bounding boxes on the image.
(b) If a display device is present, the results are shown on the screen.
(c) Finally, the buffer state is reset to eFRAMEBUF_EMPTY for the next processing cycle.

## 4.6 C++ Software Design – Model Inference

### 4.6.1 Inference Task Creation

1. Using FreeRTOS xTaskCreate Function
(1) Creates an inference task responsible for processing image data required for inference.
(2) The task retrieves complete image data from the framebuffer when its state is eFRAMEBUF_FULL and passes it to the inference process.

### 4.6.2 Model Inference Workflow

1. Execution Condition:

   When the framebuffer state is eFRAMEBUF_FULL, it indicates that the image data is ready, and inference can begin.

2. Inference Steps:
   (1) Set inference task parameters:
      a. Assign the inference output buffer (responseQueue) and the post-processing function (PostProcess).
      b. Pass the number of columns and rows of the input image (inputImgCols, inputImgRows).
      c. Bind the source image dimensions (srcImgWidth, srcImgHeight) with the complete image buffer.
   (2) Add inference task to queue:
      a. Use xQueueSend to pass the configured inferenceJob to the inference processing queue.
      b. Update the framebuffer state to eFRAMEBUF_INF, indicating that the frame has entered the inference stage.
3. After inference completion:

   The buffer state is updated to a new value (for example, reset or proceed to the next stage of processing).

4. Inference completion flag:

   The buffer state is set to eFRAMEBUF_INF, marking the inference result as ready.

5. Data structure transfer:

   The output results from the inference module are stored in the results section of the buffer, awaiting calls from the post-processing module.

6. Post-processing trigger:

   Based on the eFRAMEBUF_INF state, the post-processing module automatically invokes the corresponding processing function to perform tasks such as bounding box scaling and filtering.

### 4.6.3 Post-Processing and Drawing Workflow

After completing model inference, the detected bounding boxes must be processed, scaled back to the original image size, and filtered to retain valid object detection results. The main steps are:

1. Scaling Bounding Boxes
   (1) Function:
      a. Scale the detected bounding boxes output by the inference network (imgNetRows, imgNetCols) proportionally back to the original image size (imgSrcRows, imgSrcCols).
      b. Ensure bounding boxes are accurately mapped to their positions in the original image.

2. Extracting Detection Results
   (1) Function:

      Parse the model's output tensor to extract each object's bounding box, class ID, and confidence score.

   (2) Note:

      Exclude detection results below the predefined confidence threshold (e.g., only keep bounding boxes with confidence > 0.5).

3. Performing NMS (Non-Maximum Suppression)
   (1) Function:
      a. Apply Non-Maximum Suppression (NMS) to remove highly overlapping bounding boxes, retaining only those with the highest confidence scores.
      b. This step prevents multiple detections of the same object, improving accuracy.
   (2) Algorithm Steps:
      a. Sort detection boxes in descending order by confidence score.
      b. Compare the Intersection Over Union (IOU) values; if greater than the threshold (e.g., 0.5), remove overlapping boxes.
      c. Retain the remaining bounding boxes as the final results.

4. Result Output
   (1) Function:
      a. Store the final detection results (including bounding box coordinates and object class) into the output vector (resultsOut) for subsequent drawing and display.

(2) Application:

a. Display results on the development board's LCD or pass them to other processing modules.

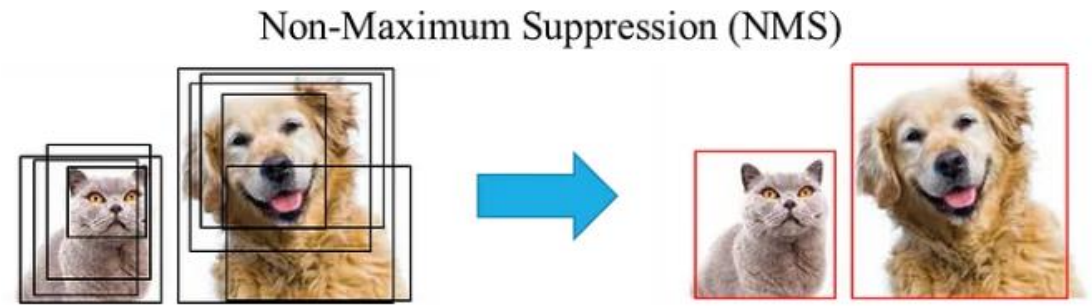


Figure: NMS Result

### 4.6.4 Configuring Label.cpp File

1. Adding Label Data:
   (1) In Label.cpp, use the labelVec[] array to define all class names the model can recognize.
   (2) Based on the trained dataset, list all class names (e.g., TellWell, Postan, Nacid) in sequence to ensure that inference results match the correct object names.

2. Label Naming Rules:
   (1) Each class name must be clear and accurate to match the dataset's labels.
   (2) Avoid duplicate or ambiguous names to prevent confusion in the model's output.

3. Function Description:
   (1) The values in the labelVec[] array are used for result mapping after inference.

   For each detected object, the class ID will be matched to the corresponding label in the array to display the correct name.

### 4.6.5 Implementation Example

```
static const char *labelsVec[] LABELS_ATTRIBUTE = {
    "TellWell",
    "Postan",
    "Nacid",
};
```

1. This code defines a static constant character array containing all possible label names the model can detect.
2. LABELS_ATTRIBUTE is an attribute specifier, possibly used for compiler optimization or special memory alignment.

### 4.6.6 Notes

1. If the model's training dataset is updated or modified, remember to update the label array accordingly; otherwise, inference results may not match the correct class names.
2. After updating Label.cpp, recompile the program to ensure the new labels are effective at runtime.

Completing the above steps allows you to continue with the subsequent deployment steps as described in the firmware programming guide.

References:

YOLOX-Nano Training: https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

COCO Dataset: https://cocodataset.org/#home

CUDA Toolkit: https://developer.nvidia.com/cuda-toolkit

OpenCV: https://opencv.org/

NMS Algorithm: https://ieeexplore.ieee.org/document/8100168

# 5 Smart Healthcare 2 – Fall Detection

## 5.1 Case Introduction – Fall Detection

### 5.1.1 Project Overview

This system utilizes the Nuvoton M55M1 development board to classify three human postures: fall-down, sitting, and standing.

The dataset was sourced from Roboflow, and the YOLOX-Nano model was trained under the PyTorch framework.

The trained model was subsequently converted and quantized into the TensorFlow Lite format, followed by Vela compilation for deployment on the development board.

Finally, both the C software and the model were programmed onto the board using the KEIL IDE.

By applying machine learning to determine whether a person has fallen within the detection area, this system not only reduces the demand for medical personnel but also enables faster detection of emergency situations.

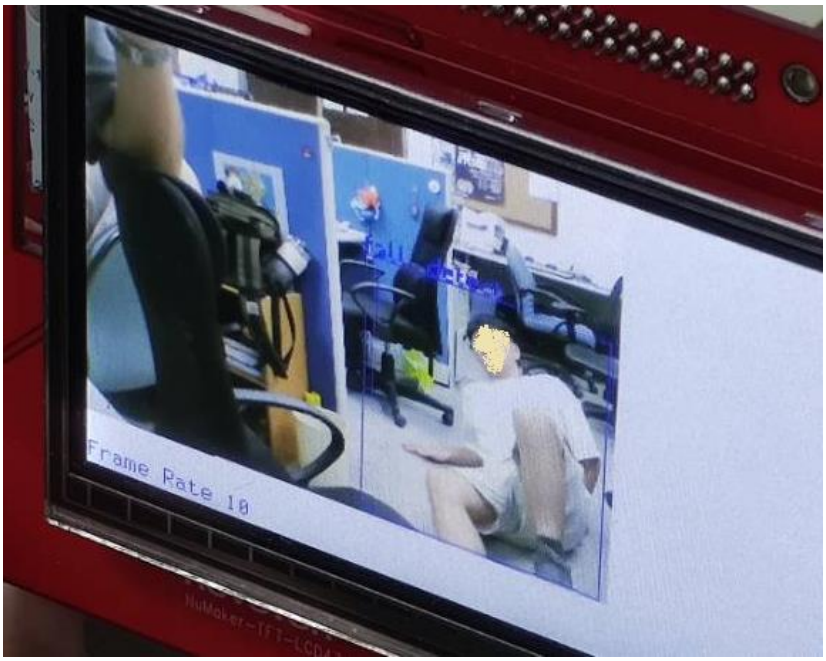


Figure: Successful detection of fall-down state on development board

Figure: Successful detection of sitting state on development board

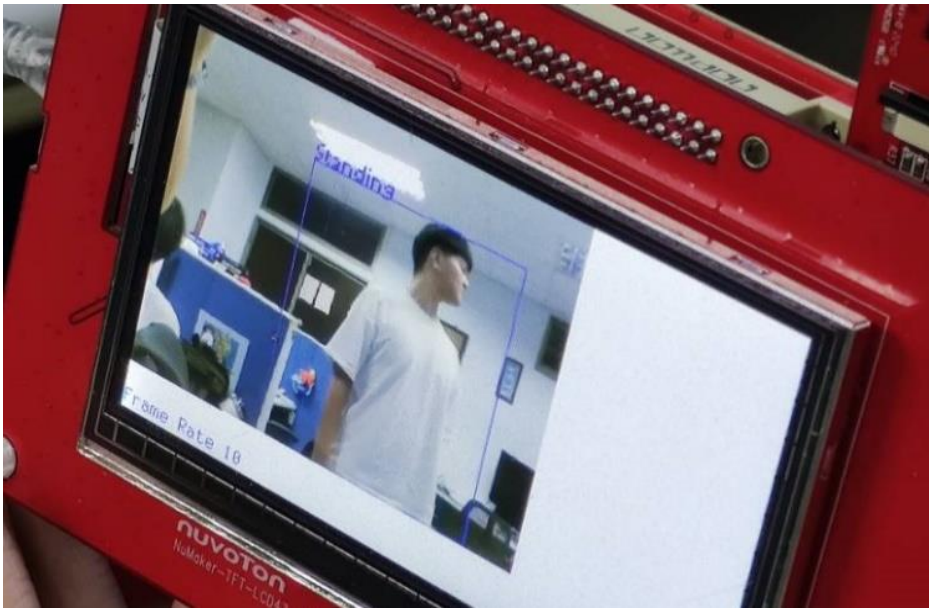


Figure: Successful detection of standing state on development board

## 5.2 Overview of Fall Detection Techniques

Machine learning–based fall detection models can generally be implemented using two methods: Keypoint Detection and Object Detection.

This report describes both approaches and explains why Object Detection was selected.

1. Keypoint Detection

In the keypoint detection approach, the human torso and limb positions are annotated in the image.

Figure: Example annotation using Keypoint Detection

This method offers higher accuracy for posture recognition but requires a more complex labeling process and a significantly larger dataset to achieve reliable classification.

It is more suitable for applications where posture variations are minimal—such as distinguishing between good and bad sitting postures.

2. Object Detection

The object detection approach follows the same principles as general object detection.

During annotation, the target in the image is simply enclosed in a bounding box and labeled with its posture class.

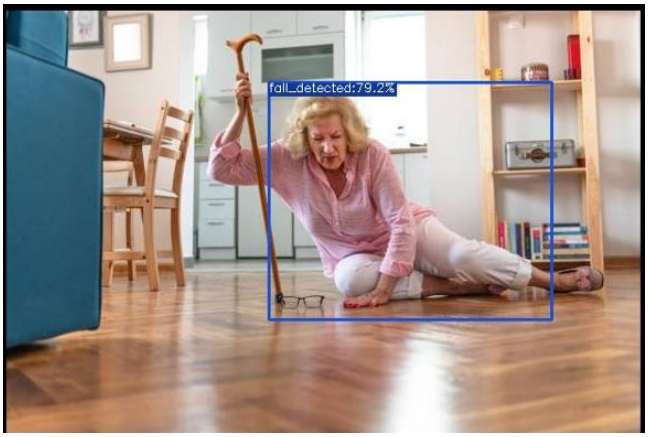


Figure: Example annotation using Object Detection

This process is relatively straightforward, and in this application—where the three posture classes (fall-detected, sitting, standing) differ significantly—it still delivers excellent accuracy.

## 5.3 Using an Open-Source Dataset from Roboflow

The dataset was sourced from Roboflow Universe, where a suitable dataset for fall detection training was identified. It was downloaded in COCO JSON format to be directly used for training under the PyTorch framework. The selected dataset, Fall Detection Dataset, categorizes target postures into three classes:


Figure: Example training image for “Fall-detected” in Fall Detection Dataset

Figure: Example training image for “Sitting” in Fall Detection Dataset


Figure: Example training image for “Standing” in Fall Detection Dataset

The dataset is organized as follows:

- Training Set: 731 images
- Validation Set: 208 images
- Test Set: 104 images

## 5.4 Model Training on PC Using the Anaconda Environment

### 5.4.1 Training Environment Installation Method

Follow the steps below in sequence:

1. Create a Python environment

   ```
   $ conda create --name yolox_nano python=3.12
   $ conda activate yolox_nano
   ```

   This command creates and activates a new environment named yolox_nano in Anaconda.

2. Upgrade pip

   ```
   $ python -m pip install --upgrade pip setuptools
   ```

3. Install PyTorch from the official website

   ```
   $ python -m pip install torch torchvision torchaudio --
   index-url https://download.pytorch.org/whl/cu118
   ```

4. Install mmcv

   ```
   $ python -m pip install mmcv==2.0.1 -f
   https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/inde
   x.html
   ```

5. Install other required packages

   Navigate to the designated directory and execute:

   ```
   $ python -m pip install --no-input -r requirements.txt
   ```

6. Install YOLOX

   ```
   $ python setup.py develop
   ```

### 5.4.2 Model Training Configuration

Organize the dataset in the following structure under the Datasets directory:

```
Datasets/
    <your_dataset_name>/
        annotations/
            <train_annotation_json_file>
            <val_annotation_json_file>
        train2017/
            <train_img>
        val2017/
            <validation_img>
```

In the training configuration file yolox_nano_ti_lite_nu.py. Specify the paths for the training and validation datasets. Configure training parameters.

```
27          # Define yourself dataset path
28          self.data_dir = "datasets/COCO"
29          self.train_ann = "train_annotations.coco.json"
30          self.val_ann = "val_annotations.coco.json"
31
32          # --------------  training config --------------------- #
33          self.num_classes = 4
34          self.warmup_epochs = 5
35          self.max_epoch = 500
```

Figure Setting relevant code in the training file yolox_nano_ti_lite_nu.py

Set the number of iterations appropriately (e.g., 500 iterations requiring approximately 3 hours). In the class definition file coco_classes.py. Define the dataset categories: none, fall-detected, sitting, standing.

```
5   COCO_CLASSES = (
6       "none",
7       "fall_detected",
8       "sitting",
9       "standing"
10  )
```

Figure: Set the relevant code in the class file coco_classes.py

Finally, execute the training command:

```
$ python tools/train.py -f
exps/default/yolox_nano_ti_lite_nu.py -d 1 -b 64 --fp16 -o -c
pretrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti_lite
.pth
```

After training, inference can be performed using the test set images to verify accuracy.


Figure: Model inference using the test set image Standing


Figure: Model inference using test set images Sitting and Fall-down

In this example, the PyTorch-trained YOLOX-Nano model achieved 90.3% accuracy, with only 10 misclassified images out of 104 test images.

### 5.4.3 Model Framework Conversion

1. Convert the trained checkpoint (.ckpt) to ONNX (.onnx):

   ```
   $ python tools/export_onnx.py -f
   exps/default/yolox_nano_ti_lite_nu.py -c
   YOLOX_outputs/yolox_nano_ti_lite_nu/latest_ckpt.pth --
   output-name
   YOLOX_outputs/yolox_nano_ti_lite_nu/yolox_nano_nu_fall.onnx
   ```

2. Convert the ONNX file to TensorFlow format and quantize to TensorFlow Lite (.tflite):

   ```
   $ python demo/TFLite/generate_calib_data.py --img-size 320
   320 --n-img 200 -o
   YOLOX_outputs\yolox_nano_ti_lite_nu\calib_data_320x320_n200.
   npy --img-dir datasets\COCO\train2017
   ```

```
$ onnx2tf -i
YOLOX_outputs/yolox_nano_ti_lite_nu/yolox_nano_nu_fall2.onnx
-oiqt -qcind images
YOLOX_outputs\yolox_nano_ti_lite_nu\calib_data_320x320_n200.
npy "[[[[0,0,0]]]]" "[[[[1,1,1]]]]"
```

### 5.4.4 Vela Compilation Procedure

1. Place the quantized model into the vela/generated/ directory and edit the variables.bat file in the vela directory:

   ```
   $ set
   MODEL_SRC_FILE=yolox_nano_nu_fall_full_integer_quant.tflite
   ```

   ```
   $ set
   MODEL_OPTIMISE_FILE=yolox_nano_nu_fall_full_integer_quant_ve
   la.tflite
   ```

2. Execute the gen_model_cpp script.

   The generated output will appear in:

   ```
   vela\generated\yolox_nano_ti_lite_nu_fall_full_integer_quant
   _vela.tflite.cc
   ```

## 5.5 Inference Program System Flow on Development Board

The C language program system flow and its explanation mainly consist of three steps:

1. Initialization of hardware and model.
2. Execution of operations depending on different states of the frame buffer.
3. Updating the frame buffer state, then repeating steps 2–3.

The following provides detailed explanations along with relevant code.

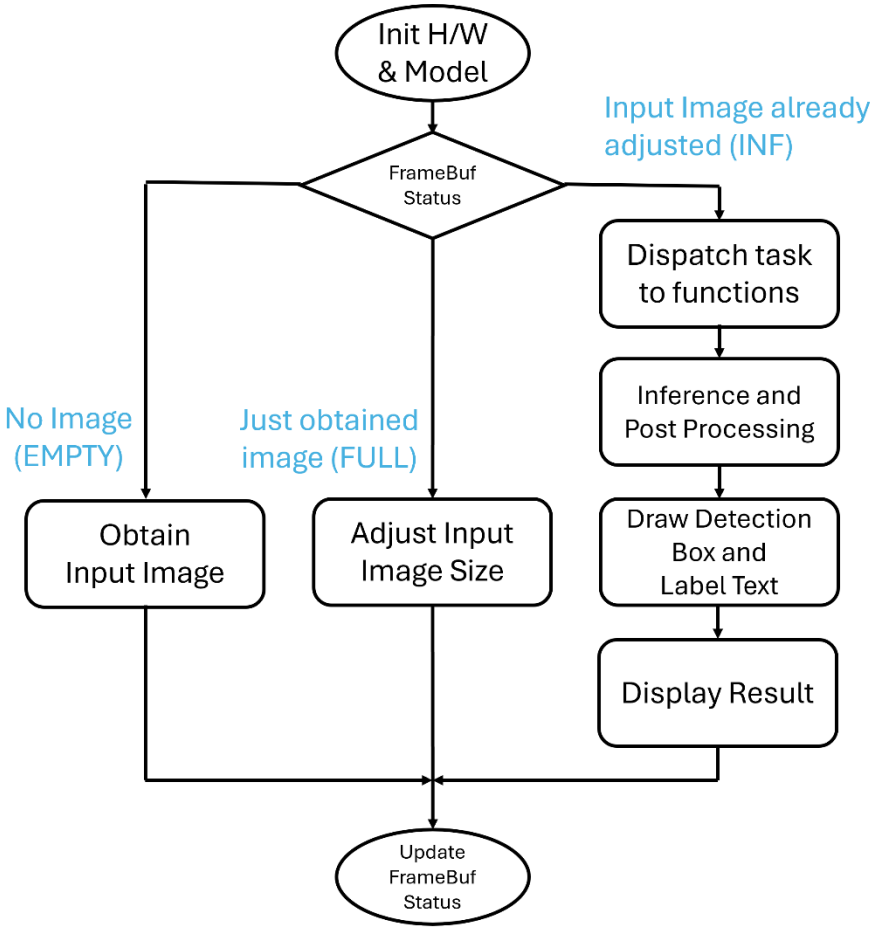


Figure: C language program system flowchart

### 5.5.1 Initialization of Hardware and Model

Initialize hardware resources and the model, includ configuring the frame buffer size and format, and setting the frame buffer state to EMPTY, indicating that it is available for storing new images.

```
150 static void omv_init()
151 {
152     image_t frameBuffer;
153     int i;
154 
155     frameBuffer.w = GLCD_WIDTH;
156     frameBuffer.h = GLCD_HEIGHT;
157     frameBuffer.size = GLCD_WIDTH * GLCD_HEIGHT * 2;
158     frameBuffer.pixfmt = PIXFORMAT_RGB565;
159 
160     _fb_base = fb_array;
161     _fb_end =  fb_array + OMV_FB_SIZE - 1;
162     _fballoc = _fb_base + OMV_FB_SIZE + OMV_FB_ALLOC_SIZE;
163     _jpeg_buf = jpeg_array;
164 
165     fb_alloc_init0();
166 
167     framebuffer_init0();
168     framebuffer_init_from_image(&frameBuffer);
169 
170     for (i = 0 ; i < NUM_FRAMEBUF; i++)
171     {
172         s_asFramebuf[i].eState = eFRAMEBUF_EMPTY;
173     }
174 
175     framebuffer_init_image(&s_asFramebuf[0].frameImage);
```

Figure: Initialization of hardware and model code

### 5.5.2 Image Input Acquisition

Capture images from the camera or load images from embedded storage and update the frame buffer state to FULL.

```
#if defined (__USE_CCAP__)
            //capture frame from CCAP
#if defined(__PROFILE__)
            u64CCAPStartCycle = pmu_get_systick_Count();
#endif

            ImageSensor_Capture((uint32_t)(emptyFramebuf->frameImage.data));

#if defined(__PROFILE__)
            u64CCAPEndCycle = pmu_get_systick_Count();
            info("ccap capture cycles %llu \n", (u64CCAPEndCycle - u64CCAPStartCycle));
#endif

#else
            //copy source image to frame buffer
            image_t srcImg;

            srcImg.w = IMAGE_WIDTH;
            srcImg.h = IMAGE_HEIGHT;
            srcImg.data = (uint8_t *)pu8ImgSrc;
            srcImg.pixfmt = PIXFORMAT_RGB888;

            roi.x = 0;
            roi.y = 0;
            roi.w = IMAGE_WIDTH;
            roi.h = IMAGE_HEIGHT;

            imlib_nvt_scale(&srcImg, &emptyFramebuf->frameImage, &roi);
#endif
```

Figure: Image input acquisition code

### 5.5.3 Input Image Resizing

Resize the input image to the model input size and update the frame buffer state to INF.

```
395         if (fullFramebuf)
396         {
397             //resize full image to input tensor
398             image_t resizeImg;
399 
400             roi.x = 0;
401             roi.y = 0;
402             roi.w = fullFramebuf->frameImage.w;
403             roi.h = fullFramebuf->frameImage.h;
404 
405             resizeImg.w = inputImgCols;
406             resizeImg.h = inputImgRows;
407             resizeImg.data = (uint8_t *)inputTensor->data.data; //direct resize to input tensor buffer
408             resizeImg.pixfmt = PIXFORMAT_RGB888;
409 
410 #if defined(__PROFILE__)
411             u64StartCycle = pmu_get_systick_Count();
412 #endif
413             imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);
```

Figure: Input image resizing code

### 5.5.4 Processing Images Pending Inference

Assign tasks to relevant sub-functions. The operations include performing inference and post-processing, drawing object detection bounding boxes, displaying results on the LCD monitor, and updating the frame buffer state back to EMPTY.

```
395         if (fullFramebuf)
396         {
397             //resize full image to input tensor
398             image_t resizeImg;
399 
400             roi.x = 0;
401             roi.y = 0;
402             roi.w = fullFramebuf->frameImage.w;
403             roi.h = fullFramebuf->frameImage.h;
404 
405             resizeImg.w = inputImgCols;
406             resizeImg.h = inputImgRows;
407             resizeImg.data = (uint8_t *)inputTensor->data.data; //direct resize to input tensor buffer
408             resizeImg.pixfmt = PIXFORMAT_RGB888;
409 
410 #if defined(__PROFILE__)
411             u64StartCycle = pmu_get_systick_Count();
412 #endif
413             imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);
```

Figure: Image processing for inference code

```
        if (infFramebuf)
        {
            //draw bbox and render
            /* Draw boxes. */
            DrawImageDetectionBoxes(infFramebuf->results, &infFramebuf->frameImage, labels);

            //display result image
#if defined (__USE_DISPLAY__)
            //Display image on LCD
            sDispRect.u32TopLeftX = 0;
            sDispRect.u32TopLeftY = 0;
            sDispRect.u32BottonRightX = (infFramebuf->frameImage.w - 1);
            sDispRect.u32BottonRightY = (infFramebuf->frameImage.h - 1);

#if defined(__PROFILE__)
            u64StartCycle = pmu_get_systick_Count();
#endif

            Display_FillRect((uint16_t *)infFramebuf->frameImage.data, &sDispRect);

#if defined(__PROFILE__)
            u64EndCycle = pmu_get_systick_Count();
            info("display image cycles %llu \n", (u64EndCycle - u64StartCycle));
#endif

#endif

            u64PerfFrames ++;
```

Figure: Image processing for inference code

```
        if ((uint64_t) pmu_get_systick_Count() > u64PerfCycle)
        {
            info("Total inference rate: %llu\n", u64PerfFrames / EACH_PERF_SEC);
#if defined (__USE_DISPLAY__)
            sprintf(szDisplayText, "Frame Rate %llu", u64PerfFrames / EACH_PERF_SEC);
            //            sprintf(szDisplayText,"Time %llu",(uint64_t) pmu_get_systick_Count() / (uint64_t)SystemCoreClock);

            sDispRect.u32TopLeftX = 0;
            sDispRect.u32TopLeftY = frameBuffer.h + FONT_HTIGHT;
            sDispRect.u32BottonRightX = (frameBuffer.w);
            sDispRect.u32BottonRightY = (frameBuffer.h + (2 * FONT_HTIGHT) - 1);

            Display_ClearRect(C_WHITE, &sDispRect);
            Display_PutText(
                szDisplayText,
                strlen(szDisplayText),
                0,
                frameBuffer.h + FONT_HTIGHT,
                C_BLUE,
                C_WHITE,
                false
            );
#endif
            u64PerfCycle = (uint64_t)pmu_get_systick_Count() + (uint64_t)(SystemCoreClock * EACH_PERF_SEC);
            u64PerfFrames = 0;
        }

        PresentInferenceResult(infFramebuf->results, labels);
        infFramebuf->eState = eFRAMEBUF_EMPTY;
    }
```

Figure: Image processing for inference code

### 5.5.5 Model Inference and Post-processing

The program sequentially:

1. Acquires the model input tensor.
2. Executes inference.
3. Obtains the model output tensor.
4. Applies post-processing to interpret inference results.

```
300     TfLiteTensor *inputTensor   = model.GetInputTensor(0);
301     TfLiteTensor *outputTensor = model.GetOutputTensor(0);
302
303     if (!inputTensor->dims)
304     {
305         printf_err("Invalid input tensor dims\n");
306         vTaskDelete(nullptr);
307         return;
308     }
309     else if (inputTensor->dims->size < 3)
310     {
311         printf_err("Input tensor dimension should be >= 3\n");
312         vTaskDelete(nullptr);
313         return;
314     }
315
316     TfLiteIntArray *inputShape = model.GetInputShape(0);
317
318     const int inputImgCols = inputShape->data[arm::app::YoloXnanoNu::ms_inputColsIdx];
319     const int inputImgRows = inputShape->data[arm::app::YoloXnanoNu::ms_inputRowsIdx];
320
321     // postProcess
322     arm::app::object_detection::DetectorPostprocessing postProcess(0.6, 0.65, numClasses, 0);
```

Figure: Model inference and post-processing code

### 5.5.6 Drawing Detection Bounding Boxes and Label Text

Retrieve relevant information from inference results and render bounding boxes and labels on the image.

```
211         imlib_draw_rectangle(drawImg, result.m_x0, result.m_y0, result.m_w, result.m_h, COLOR_B5_MAX, 1, false);
```

Figure: Drawing detection bounding boxes code

```
212         imlib_draw_string(drawImg, result.m_x0, result.m_y0 - 16, labels[result.m_cls].c_str(), COLOR_B5_MAX, 2, 0, 0, false,
213                           false, false, false, 0, false, false);
```

Figure: Drawing label text code

### 5.5.7 Displaying Results on LCD Monitor

Output the processed image with annotations to the LCD monitor.

```
            //display result image
#if defined (__USE_DISPLAY__)
            //Display image on LCD
            sDispRect.u32TopLeftX = 0;
            sDispRect.u32TopLeftY = 0;
            sDispRect.u32BottonRightX = (infFramebuf->frameImage.w - 1);
            sDispRect.u32BottonRightY = (infFramebuf->frameImage.h - 1);

#if defined(__PROFILE__)
            u64StartCycle = pmu_get_systick_Count();
#endif

            Display_FillRect((uint16_t *)infFramebuf->frameImage.data, &sDispRect);

#if defined(__PROFILE__)
            u64EndCycle = pmu_get_systick_Count();
            info("display image cycles %llu \n", (u64EndCycle - u64StartCycle));
#endif

#endif

            u64PerfFrames ++;

            if ((uint64_t) pmu_get_systick_Count() > u64PerfCycle)
            {
                info("Total inference rate: %llu\n", u64PerfFrames / EACH_PERF_SEC);
```

Figure: Displaying results on LCD monitor code

## 5.6 Conclusion and Future Development

### 5.6.1 Conclusion

This fall detection system effectively leverages the Roboflow dataset in combination with the YOLOX-Nano model on the Nuvoton M55M1 development board. The solution provides a faster and more resource-efficient approach to detecting falls, thereby improving healthcare quality. By applying Full-INT8 model quantization technology and Vela compiler optimization, both inference speed and accuracy were improved, achieving over 90% detection accuracy in testing.

### 5.6.2 Future Development

This current development only uses a simple camera for image capture. Future enhancements may incorporate more advanced cameras (e.g., 3D cameras) or adopt Keypoint-Detection methods to increase the variety of pose recognition and accuracy, thereby enabling more professional and precise applications.

References:


- YOLOX-Nano Training
  https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

- Roboflow Fall Detection Dataset
  https://universe.roboflow.com/search?q=class:fall-detected

- Roboflow Keypoint-Detection Introduction
  https://blog.roboflow.com/pose-estimation-algorithms-history/

# 6 Smart Home Application 1 – Waste Classification

## 6.1 Case Study – Waste Classification

### 6.1.1 Introduction and Application

1. Technical Foundation:

   (1) Implementation of waste classification image recognition using the Nuvoton M55M1 development board. This system integrates a camera module, enabling effective recognition of different types of waste and providing rapid classification. Through real-time display, the system offers detailed information on the identified waste categories.
   (2) YOLOX-Nano Model Training: By applying deep learning techniques, a waste classification model was trained to achieve accurate recognition across multiple waste categories. The trained model was then deployed onto the M55M1 development board for application.

2. Practical Application Scenarios:

   (1) In a household environment, waste classification often poses challenges for children, primarily due to significant differences in color, material, and size among various types of waste. This may result in misclassification or incomplete sorting.
   (2) With this waste classification system, children simply place the waste item in front of the camera. The screen instantly displays the waste category and provides visual reminders and guidance, thereby ensuring accurate classification.

### 6.1.2 Advantages and Features

1. Real-time Performance: Enables rapid image processing and result display.
2. Educational Value: Enhances children's awareness of waste classification and fosters environmentally conscious behavior from an early age.
3. Accuracy: Relies on the high precision of deep learning models, effectively minimizing errors inherent in manual classification.

### 6.1.3 Objectives and Achievements

1. Accuracy:

   The system was able to classify four categories of waste (paper, plastic bottles, aluminum cans, and paper cups) with an accuracy rate of 90.1%, demonstrating high effectiveness in practical applications.

2. Model Size:

   The trained model was kept under 2 MB, fully compliant with development board flash memory standards, and capable of running smoothly in embedded environments.

3. Classification Capability:

   Supports simultaneous recognition of multiple waste types, significantly improving the efficiency and practicality of waste classification.

### 6.1.4 Illustration

The figure below demonstrates the expected results in practical applications. The display shows real-time image processing results where the system successfully identifies and annotates different waste types (e.g., aluminum cans, plastic bottles).


Figure: Image recognition demo results

## 6.2 Dataset and Model Training

### 6.2.1 Dataset

1. Tool: The annotation and labeling of data were performed using LabelImg.
2. Data Splitting:
   (1) Training Set: A total of 378 images.
   (2) Validation Set: A total of 36 images.
   (3) Test Set: A total of 19 images.

### 6.2.2 Model Training

1. Model: The YOLOX-Nano (Light Model) was adopted, which is well-suited for lightweight embedded systems.
2. Virtual Environment: A virtual environment was created using Anaconda to ensure modularity and isolation during development.
3. Training Framework: Model training was conducted using the PyTorch deep learning framework, providing flexibility in model construction and training workflows.

4. Dataset Format: The dataset was managed in COCO JSON format to maintain compatibility with mainstream model frameworks.

### 6.2.3 Dataset Preparation Workflow

The dataset was primarily prepared through self-collected photographs, followed by annotation and training. The detailed steps are as follows:

1. Photo Collection:
   (1) Collect a sufficient number of images as training data (approximately 400 photographs were used in this case). A larger dataset generally results in better model training performance.

2. Data Annotation:
   (1) Use the LabelImg tool for image labeling, marking the target objects in each photograph.
   (2) This tool can be downloaded from the Internet. It is simple and user-friendly, making it suitable for annotating custom datasets.

3. Format Consistency Check:
   (1) Ensure that the annotated dataset matches the format requirements for model training. In this case, the COCO JSON format was adopted, which is supported by most deep learning frameworks.

4. Format Conversion:
   (1) Convert the original YOLO format annotations into COCO JSON format using a format conversion tool, so that the dataset can be used for model training.

### 6.2.4 Downloading and Installing LabelImg

1. Download Link:
   (1) Navigate to the following GitHub page to download the LabelImg tool: LabelImg Releases.

2. Version Selection:
   (1) On the release page, click to download the compressed file windows_v1.8.1.zip corresponding to Binary v1.8.1.

3. Installation and Execution:
   (1) Extract the downloaded file, enter the extracted directory, locate the executable labelImg.exe, and double-click to launch the annotation tool interface.

4. Tool Features:
   (1) Cross-platform Support: Compatible with Windows, Linux, and macOS.
   (2) Ease of Use: Provides rapid image annotation, making it suitable for beginners performing image labeling tasks.
   (3) Flexible Format Options: Supports multiple output formats (e.g., PASCAL VOC and COCO JSON), facilitating integration with mainstream deep learning frameworks.

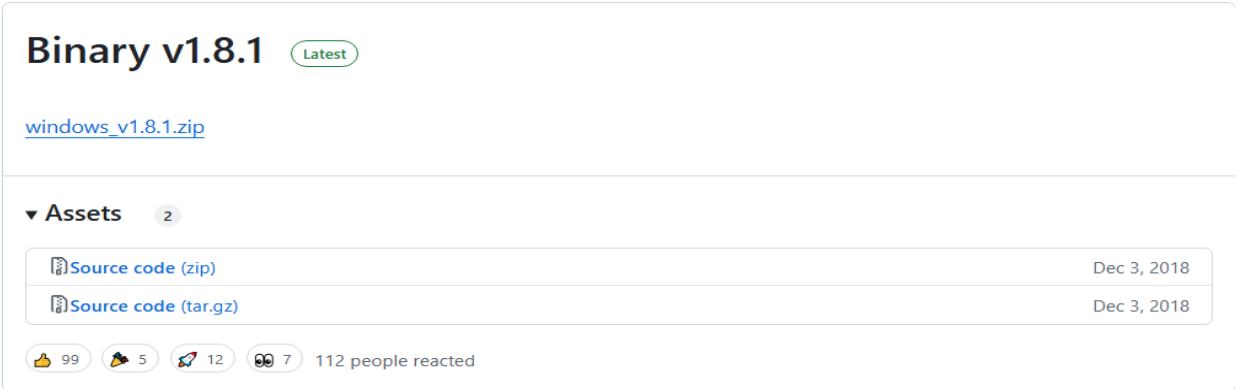


Figure: Label tool download

5. Setting Dataset Path and Storage Location
   (1) Select the Dataset Source Folder:

   Click the Open Dir button and choose the source folder of the dataset. This folder should contain all the image files to be annotated.

   (2) Select the Annotation Results Storage Location:

   Click the Change Save Dir button to specify the directory where the annotation files will be saved after completion.

   (3) Confirm the Settings:

   Once the paths are configured, the file list area at the bottom right will display all images pending annotation. Verify that all images are correctly loaded.

6. Illustration

The figure illustrates the positions and functions of the operational buttons, guiding users to quickly complete the configuration of the dataset directory and the storage directory.

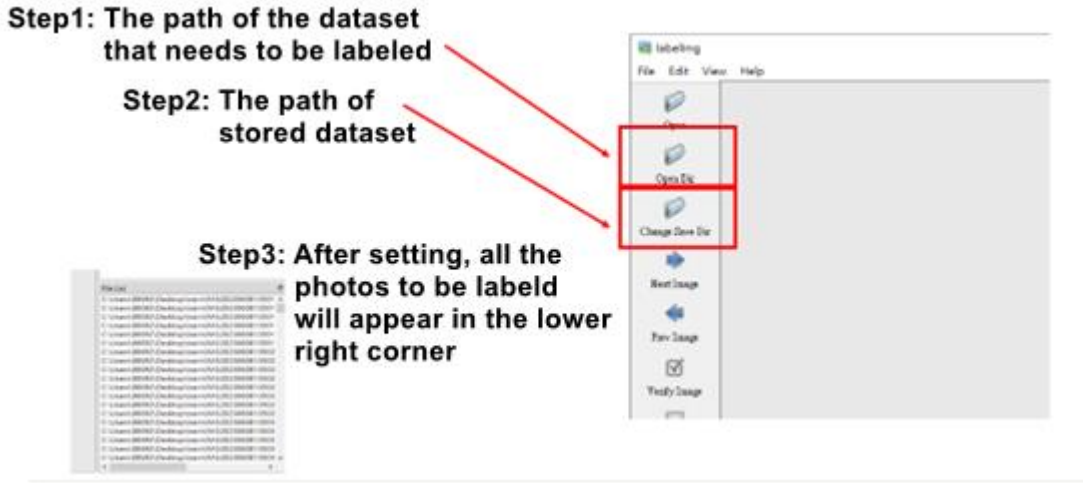


Figure: Label Tool Tutorial

7. Selecting the Annotation Format

Select the annotation format according to the model training requirements. Commonly used formats include PascalVOC, YOLO, and CreateML. Their characteristics and use cases are as follows:

(1) PascalVOC Format:
   a. Storage Method: Stored in XML files, including image file name, dimensions (height, width), and details of each bounding box (class and coordinates).
   b. Application Scenario: Suitable for training deep learning models using TensorFlow and Keras frameworks.

(2) YOLO Format (Used in This Case):
   a. Storage Method: Plain text files, simple and efficient. Each line represents one bounding box, in the format: <class_id> <x_center> <y_center> <width> <height>
      - class_id: The class ID of the bounding box.
      - x_center, y_center: Coordinates of the bounding box center (relative to image size).
      - width, height: Width and height of the bounding box (relative to image size).

b. Application Scenario: Designed for the YOLO (You Only Look Once) family of models, optimized for real-time object detection.

(3) CreateML Format:
   a. Storage Method: Stored in JSON files, describing bounding box positions, classes, and related data for each image.
   b. Application Scenario: Suitable for Apple's CreateML platform, facilitating training and deployment in macOS environments.

(4) Format Selection Recommendation:
   a. For TensorFlow or Keras: Use PascalVOC format.
   b. For YOLO training: Use YOLO format (selected in this case).
   c. For iOS application development: Use CreateML format.

8. Setting Annotation Categories and Performing Annotations
   (1) Create the Annotation Categories File:
      a. Create a text file named label.txt and list all the categories to be annotated within the file (e.g., Aluminum Can, Paper, Paper Cup, Plastic Bottle).
      b. This file will serve as the category reference for the annotation tool, ensuring correct selection of categories during annotation.

   (2) Start Annotation:
      a. Click the Create RectBox button on the toolbar (shortcut key: W) and use the mouse to drag and draw bounding boxes around the target objects.
      b. Once the bounding box is created, a prompt window will appear, allowing the user to select or input the category name for the annotated object.

   (3) Annotation Verification:
      a. After annotating an image, the Box Labels area on the right will display all annotated objects and their corresponding categories.
      b. The user can verify whether the bounding boxes are correctly assigned to the intended categories and make adjustments if necessary.

   (4) Illustration:
      a. Left Image: Displays the category list within the tool, including items such as Aluminum Can, Paper, Paper Cup.

b. Right Image: Shows an example of a bounding box and its corresponding category input operation, where the selected object is labeled as Aluminum Can.

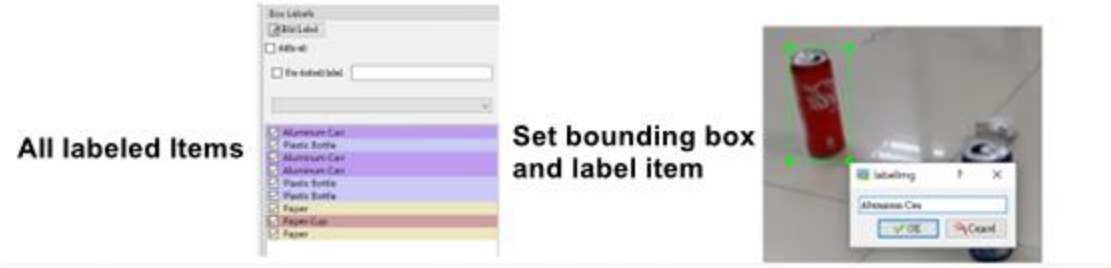


Figure: Annotation Method and Class Usage

9. Converting YOLO Format to COCO JSON Format
   (1) Format Conversion Requirement:
      a. After completing annotations in YOLO format, a Python script must be used to convert YOLO format into COCO JSON format.
      b. Reason: The COCO JSON format is widely supported across deep learning frameworks and offers higher compatibility for complex object detection tasks.

   (2) Using the Conversion Script:
      a. The provided script file `yolo2cocojson.py` can be used for format conversion: https://drive.google.com/file/d/15cSq0MiBjgwUIwJn0Z_0t2c51vUY2jh6/view?usp=sharing
      b. Users only need to make minor modifications according to their specific dataset requirements, after which the script can automatically complete the conversion process.

10. Configuring the Conversion Script
   (1) Add Annotation Categories:

   Update the categories section of the script according to the annotation categories and their order. Example illustration provided in the figure.

```
categories = [
    {"id": 0, "name": "class1"},
    {"id": 1, "name": "class2"},
    {"id": 2, "name": "class3"},
    {"id": 3, "name": "class4"}
]
```

Figure: Setting Classes and Their IDs

(2) Configure Dataset Paths:

a. Update the script with the correct paths to the images and annotation files.

b. The directory names must be exactly Images and Labels, and each image must have a corresponding annotation file with the same filename.

```
images_dir = "/images"  # 圖片資料夾
labels_dir = "/labels"  # YOLO 標註檔案資料夾
```

Figure: Configuring Image and Annotation File Paths

## 6.3 Evaluate TFLite int8/float Model (Overview)

### 6.3.1 Model Performance Evaluation

1. Purpose of Testing:

(1) To evaluate the accuracy and efficiency of the trained model, focusing on:

a. Accuracy (AP, Average Precision): Measures prediction accuracy for each class.

b. Intersection over Union (IoU): Evaluates the overlap between the predicted bounding box and the ground truth bounding box.

$$\text{IoU} = \frac{\text{預測框與真實框的交集面積}}{\text{預測框與真實框的聯集面積}}$$

Figure: Definition of IoU

2. Evaluation Method:
   (1) A Python script is used to run inference on the model and gather performance metrics for each category.
   (2) Both integer-quantized TFLite models and float TFLite models are supported for evaluation.

### 6.3.2 Evaluation via Command Line

1. Basic Command Format:

```
$ python demo/TFLite/tflite_inference.py \
  -m <FULL_INTEGER_QUANT_TFLITE> \
  -s <SCORE_THR> \
  -i <PATH_OF_IMAGE> \
  -a <PATH_OF_VAL_ANNOTATION_FILE>
```

2. Example Command:

```
python demo/TFLite/tflite_inference.py \
  -m
YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_hg_full_inte
ger_quant.tflite \
  -s 0.6 \
  -i datasets/hagrid_coco/val2017/0001.jpg \
  -a hagrid_coco/annotations/hagrid_val.json
```

### 6.3.3 Command Parameter Explanation

1. FULL_INTEGER_QUANT_TFLITE:

Path to the integer-quantized TFLite model file (.tflite).

2. SCORE_THR:

   Detection confidence threshold; only results with scores above this value will be kept (example: 0.6).

3. PATH_OF_IMAGE:

   Path to the image file used for inference testing.

4. PATH_OF_VAL_ANNOTATION_FILE:

   Path to the validation annotation file containing ground truth bounding box data (.json format).

### 6.3.4 Model Performance Testing

1. Single Image Test Command:

```
$ python demo/TFLite/tflite_inference.py \
  -m <FULL_INTEGER_QUANT_TFLITE> \
  -s <SCORE_THR> \
  -i <PATH_OF_IMAGE> \
  -a <PATH_OF_VAL_ANNOTATION_FILE>
```

2. Example Command:

```
python demo/TFLite/tflite_inference.py \
  -m
YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_ti_lite_nu_full
_integer_quant.tflite \
  -s 0.6 \
  -i datasets/hagrid_coco/val2017/0001.jpg \
  -a hagrid_coco/annotations/hagrid_val.json
```

### 6.3.5 Parameter Summary

1. FULL_INTEGER_QUANT_TFLITE:

   File path of the integer-quantized TFLite model (.tflite).

2. SCORE_THR:

Detection confidence threshold; only results above this threshold are retained (example: 0.6).

3. PATH_OF_IMAGE:

   Full path of the test image (example: datasets/hagrid_coco/val2017/0001.jpg).

4. PATH_OF_VAL_ANNOTATION_FILE:

   Validation annotation file containing ground truth bounding boxes (example: hagrid_coco/annotations/hagrid_val.json).

## 6.4 C++ Software Flow

### 6.4.1 System Initialization

1. Main File Locations:

   `BoardInit.cpp` and `mpu_config_M55M1.h`

2. Hardware Resource Configuration:
   (1) Configure essential system hardware resources such as clock, UART, memory, and NPU (Neural Processing Unit).
   (2) Configure the Memory Protection Unit (MPU), including memory region allocation and access permissions. The detailed definitions are specified in mpu_config_M55M1.

3. Objective:

   Ensure that the hardware environment is properly initialized, providing a stable foundation for subsequent program execution.

### 6.4.2 Data Preparation

1. Loading and Processing Data:
   (1) Load image data and perform necessary preprocessing, such as resizing or format conversion.
   (2) Ensure that input data meets the requirements of model inference, thereby improving accuracy and efficiency.

2. Task Scheduling:

   Handled by InferenceTask.cpp. This module manages data reception and preliminary scheduling to provide valid inputs for the inference module.

### 6.4.3 Model Inference

1. Main File Location:

   `InferenceTask.cpp`

2. Execution Details:
   (1) Perform inference using the model core function (`m_model->RunInference()`) and retrieve the output tensor via `GetOutputTensor()`.
   (2) During inference, compute class probabilities and predicted bounding boxes, converting raw predictions into interpretable results.

3. Objective:

   Achieve efficient inference execution while ensuring that the results accurately reflect the input data.

### 6.4.4 Result Output and Post-Processing

1. Main File Location:

   `DetectorPostProcessing.cpp`

2. Execution Details:
   (1) Apply Non-Maximum Suppression (NMS) to filter out redundant prediction boxes, thereby improving detection accuracy.
   (2) Map model outputs back to the coordinates of the original image for further processing and visualization.

3. Objective:

   Refine output results to precisely locate and classify targets, ultimately yielding accurate detection outputs.

### 6.4.5 Main Loop

1. Functionality:

   Contains control logic for continuous system execution until an exit condition is triggered (e.g., receiving a termination signal).

2. Key Checks:

   Ensure proper resource deallocation during exit to prevent resource leaks and maintain overall system stability.

## 6.5 System Program Analysis

### 6.5.1 Main Files

This section effectively details the boot-up sequence of the M55M1 system, covering clock → UART → HyperRAM → NPU → security. Each step is modularized in `BoardInit.cpp` and parameterized in `mpu_config_M55M1.h`.

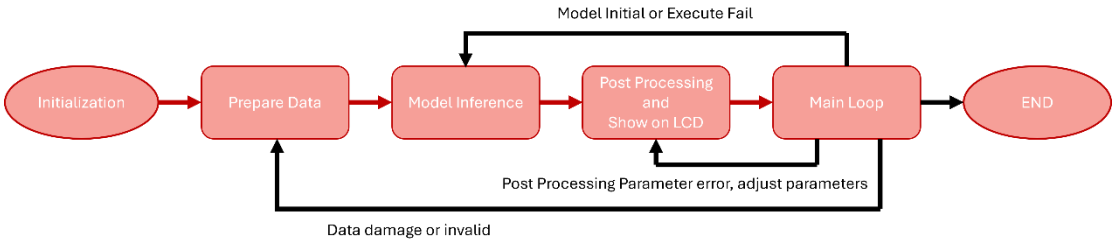


Figure: System flow chart.

1. System and Module Clock Initialization (SYS_Init)
   (1) Functionality:

   Ensure both system clock and external oscillators operate stably, providing clock sources to all modules.

   (2) Steps:
   a. Enable internal and external oscillators, wait for stabilization.

b. Configure APLL (Phase-Locked Loop) as the system clock source at 180 MHz
c. Enable clocks for all required hardware modules.

(3) Example Code:

```
/* Enable internal RC and external Crystal Osc */
CLK_EnableXtalRC(CLK_SRCTL_HIRCEN_Msk);
CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);
CLK_EnableXtalRC(CLK_SRCTL_HXTEN_Msk);
CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk);

/* Set and update system clock */
CLK_SetBusClock(CLK_SCLKSRC_APLL, FREQ_180MHZ);
SystemCoreClockUpdate();
```

Figure: System Initialization Code

2. UART Initialization
(1) Functionality:

Configure UART6 as the standard output port to support debugging via printf.

(2) Steps:
a. Set the clock source and multi-function pins for UART.
b. Initialize UART to enable standard output functions.
(3) Example Code:

```
/* Configure UART Clock */
SetDebugUartCLK();
SetDebugUartMFP();
InitDebugUart();  // Initialze UART and Standard Out
```

Figure: UART Code Example

3. HyperRAM Initialization and Mode Configuration
(1) Functionality:

Configure the HyperRAM communication interface and switch to Direct Map Mode for higher access efficiency.

(2) Steps:

a. Configure HyperRAM pin assignments and perform basic initialization.
b. Switch to Direct Map Mode to optimize memory access.

(3) Example Code:

```
HyperRAM_PinConfig(HYPERRAM_SPIM_PORT);
HyperRAM_Init(HYPERRAM_SPIM_PORT);
SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);
```

Figure: HyperRAM Code Example

4. Arm® Ethos™-U NPU Initialization

(1) Functionality:

Initialize the Arm® Ethos™-U NPU and validate status to ensure embedded AI models can execute properly.

(2) Steps:

a. Enable the NPU.
b. Verify initialization status; return error codes for debugging if initialization fails.

(3) Example Code:

```
#if defined(ARM_NPU)
int state;

/* Init Arm Ethos-U NPU, result error code if failed */
if (0 != (state = arm_ethosu_npu_init())) {
    return state;
}
#endif
```

Figure: NPU Initialization Code Example

5. Hardware Protection Configuration

(1) Functionality:

Unlock protected registers, configure hardware protection, and lock registers again to ensure system security.

(2) Steps:

a. Unlock protected registers.
b. Apply hardware protection configurations.
c. Lock registers and print a message confirming system initialization completion.

(3) Example Code:

```
SYS_UnlockReg();  // Unlock protected registers
SYS_LockReg();    // Lock protected registers
info("%s: complete\n", __FUNCTION__); // Output the completion
```

Figure: Hardware Protection Code Example

### 6.5.2 Model Inference Program Analysis

This section explains the end-to-end inference workflow, from input handling → inference → post-processing → multi-task coordination under FreeRTOS. It emphasizes both the computational flow (RunInference) and the concurrency management (task queues, semaphores, mutexes).

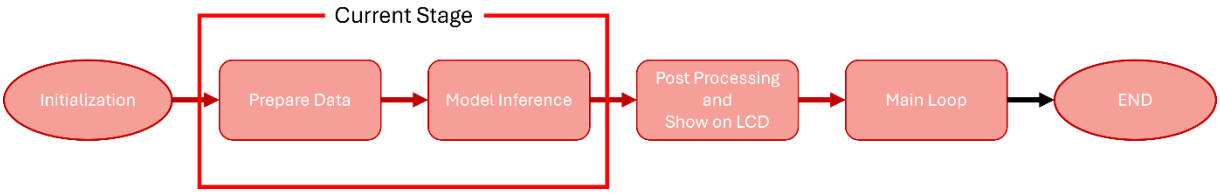


Figure: Model Inference Program Flowchart

1. Key Functional Overview (InferenceTask.cpp)
   (1) Inference Execution
      a. Model Inference Call:

         Executes model inference on input data using the core function `m_model->RunInference()`. Generates inference results for subsequent processing.

      b. Post-Processing:

         Utilizes DetectorPostProcessing to parse the output tensor data into object detection results. Extracts bounding boxes and associated labels for identified objects.

   (2) Task Handling

a. Provides a multi-task execution framework under FreeRTOS, responsible for handling data from the inference task queue.
b. Synchronization is achieved through semaphores and mutexes, ensuring safe and efficient execution across tasks.

2. Inference and Execution (RunInference)

(1) Function Description

a. Performs inference using the YOLO model, invoking the core method `m_model->RunInference()`.
b. Executes post-processing via pPostProc->RunPostProcessing(), which transforms tensor outputs into interpretable object detection results, including bounding boxes and classification labels.

(2) Example Code

```
bool InferenceProcess::RunJob(
    object_detection::DetectorPostprocessing *pPostProc,
    int modelCols,
    int modelRows,
    int srcImgWidth,
    int srcImgHeight,
    std::vector<object_detection::DetectionResult> *results)
{
    bool runInf = m_model->RunInference();  // 執行模型推論
    pPostProc->RunPostProcessing(
        modelRows,
        modelCols,
        srcImgHeight,
        srcImgWidth,
        modelOutput0,
        *results);  // 執行後處理
}
```

Figure: Inference and Execution Code Example

(3) Objective

To structure inference outputs into a usable format, enabling subsequent visualization and application.

3. FreeRTOS Multi-Task Architecture

(1) Functional Purpose

a. Receives inference requests from the task queue (xQueueReceive) and processes them sequentially.
b. After inference and post-processing, results are returned to the requesting task (xQueueSend).

(2) Example Code

```
void inferenceProcessTask(void *pvParameters)
{
    struct ProcessTaskParams params = *reinterpret_cast<struct ProcessTaskParams *>(pvParameters);

    InferenceProcess::InferenceProcess inferenceProcess(params.model);

    for (;;)
    {
        xInferenceJob *xJob;

        xQueueReceive(params.queueHandle, &xJob, portMAX_DELAY);  // 等待推論任務

        inferenceProcess.RunJob(
            xJob->pPostProc,
            xJob->modelCols,
            xJob->mode1Rows,
            xJob->srcImgWidth,
            xJob->srcImgHeight,
            xJob->results
        );

        xQueueSend(xJob->responseQueue, &xJob, portMAX_DELAY);  // 回傳結果
    }
}
```

Figure: Multi-Task Architecture Code Example

(3) Architectural Advantages

a. Enhances execution efficiency by using a task queue to maintain orderly data processing.
b. Ensures resource safety during multi-task execution, preventing conflicts between concurrent tasks.

### 6.5.3 Post-Processing Program Analysis (DetectorProcessing.cpp)

This section explains how YOLO inference outputs are refined into final usable detection results, ensuring accuracy by filtering low-confidence boxes, applying NMS, and rescaling detections to the original image resolution.

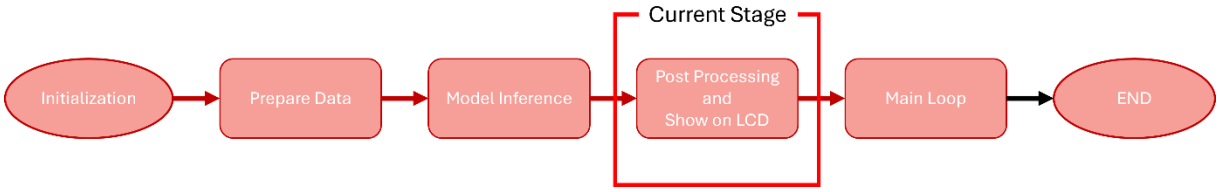


Figure: Post-Processing flowchart

1. Key Functional Overview

(1) Initialization of Post-Processing Parameters

Defines critical parameters for detection, such as detection threshold, Non-Maximum Suppression (NMS) Intersection-over-Union (IoU) threshold, number of classes, and Top-N selection.

(2) Model Output Handling

Parses bounding box positions and sizes from the YOLO model output tensor. Filters detections that meet or exceed the confidence threshold.

(3) Non-Maximum Suppression (NMS)

Eliminates bounding boxes with high overlap (IoU above the threshold). Retains only the most relevant detection results.

(4) Rescaling to Original Image Size

Converts model output bounding box coordinates (relative to resized input) back to the original image dimensions. Produces accurate physical object locations.

2. Post-Processing Initialization

(1) Parameter Definitions and Significance

a. threshold: Detection threshold; boxes below this confidence level are discarded.
b. nms: IoU threshold for NMS, controlling overlap tolerance between boxes.
c. numClasses: Number of supported object categories.
d. topN: Maximum number of highest-confidence detections returned.

(2) Example Code

```
DetectorPostprocessing::DetectorPostprocessing(
    const float threshold,
    const float nms,
    int numClasses,
    int topN)
    : m_threshold(threshold),
      m_nms(nms),
      m_numClasses(numClasses),
      m_topN(topN) {}
```

Figure: Post-Processing Initialization Example

(3) Purpose

To establish baseline configurations for the YOLO model's post-processing phase.

3. Execution of Post-Processing

(1) Function Description

a. Initializes structured data representation of YOLO network outputs.
b. Extracts tensors (e.g., bounding box coordinates and class scores) from modelOutput0.
c. Converts quantized model outputs to floating-point values using scale and zeroPoint parameters.

(2) Example Code

```
void DetectorPostprocessing::RunPostProcessing(
    uint32_t imgNetRows,
    uint32_t imgNetCols,
    uint32_t imgSrcRows,
    uint32_t imgSrcCols,
    TfLiteTensor *modelOutput0,
    std::vector<DetectionResult> &resultsOut    /* init postprocessing */
)
```

```
Network net
{
    .inputWidth = static_cast<int>(imgNetCols),
    .inputHeight = static_cast<int>(imgNetRows),
    .numClasses = m_numClasses,
    .branches = {
        Branch {
            .resolution = modelOutput0->dims->data[1],
            .modelOutput = modelOutput0->data.int8,
            .scale = ((TfLiteAffineQuantization *)(modelOutput0->quantization.params))->scale->data[0],
            .zeroPoint = ((TfLiteAffineQuantization *)(modelOutput0->quantization.params))->zero_point->data[0],
            .size = modelOutput0->bytes
        }
    },
    .topN = m_topN
```

Figure: Post-Processing Execution Example

4. Bounding Box Parsing and Filtering

   (1) Processing Logic
      a. Parses each detection's bounding box coordinates, dimensions, and confidence score.
      b. Filters boxes with confidence above the defined threshold.
      c. Converts bounding box positions from model input scale back to original image scale.

   (2) Example Code

```
GetNetworkBoxes(
    net,
    originalImageWidth,
    originalImageHeight,
    m_threshold,
    detections);
```

Figure: Bounding Box Parsing Example

5. Non-Maximum Suppression (NMS)
   (1) Objective
      a. Removes redundant bounding boxes with IoU greater than the NMS threshold, keeping only the box with the highest confidence.
      b. Ensures that each detected object is represented by only one bounding box.

6. Rescaling to Original Image Size

(1) Conversion Logic

Rescales bounding box coordinates from model input resolution (e.g., 320×320 or 416×416) to match the original image resolution. Ensures detections correspond to true object positions in the input image.

(2) Example Code

```
for (auto &it : detections)
{
    it.bbox.x = (it.bbox.x * originalImageWidth) / net.inputWidth;
    it.bbox.y = (it.bbox.y * originalImageHeight) / net.inputHeight;
    it.bbox.w = (it.bbox.w * originalImageWidth) / net.inputWidth;
    it.bbox.h = (it.bbox.h * originalImageHeight) / net.inputHeight;
}
```

Figure: Rescaling to Original Image Size Example

7. Final Result Generation

(1) Output Contents

a. Bounding box coordinates: starting point (m_x0, m_y0) and dimensions (m_w, m_h).
b. Detection confidence score (m_normalisedVal).
c. Object classification label (m_cls).

(2) Example Code

```
DetectionResult tmpResult = {};
tmpResult.m_normalisedVal = it.prob[j];
tmpResult.m_x0 = (int)boxX;
tmpResult.m_y0 = (int)boxY;
tmpResult.m_w = (int)boxWidth;
tmpResult.m_h = (int)boxHeight;
tmpResult.m_cls = j;

resultsOut.push_back(tmpResult);
```

Figure: Final Result Generation Example

### 6.5.4 Main Loop Analysis and Implementation

The main loop serves as the core control module of the system, responsible for orchestrating data flow, scheduling inference tasks, and handling output results. Below is a detailed breakdown of its design and functionality.

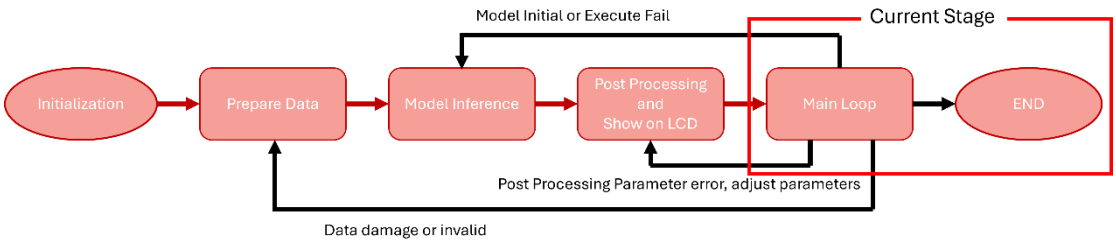


Figure: Main Loop Flowchart

Main Loop Structure and Flow

1. Data Reception (Back to Dataset Preparation Stage)
   (1) Purpose: Check for new data to process and retrieve a complete frame image from the buffer.
   (2) Steps:
      a. Obtain an empty buffer (get_empty_framebuf()).
      b. Obtain a filled buffer (get_full_framebuf()).

2. Inference Execution (Back to Model Inference Stage)
   (1) Purpose: Once the buffer data is ready, send it to the inference task for execution.
   (2) Implementation: Uses TensorFlow Lite model inference via RunInference().
   (3) Main loop role: Waits until inference completes and then proceeds with post-processing.

3. Post-Processing (Back to Post-Processing Stage)
   (1) Purpose:
      a. Apply Non-Maximum Suppression (NMS) to remove redundant bounding boxes.
      b. Rescale coordinates from model input space back to original image dimensions.

c. Refine valid detections for efficient display.

4. Display and Output (Back to Result Handling Stage)
   (1) Purpose:
      a. Render processed detection results to the output device (e.g., LCD display or serial console).
      b. Update buffer status to eFRAMEBUF_EMPTY to mark processing completion.
5. Error Handling and Summary
   (1) Error Scenarios:
      a. If data is corrupted or invalid → retry acquisition of valid data.
      b. If model initialization or execution fails → reinitialize model and restart inference task.
      c. If post-processing parameters are incorrect → adjust input parameters or model output configuration.

### 6.5.5 Key Program Analysis (Main.cpp)

1. Image Buffer Management
   (1) Purpose: Manages buffers of different states for image filling, inference, and result processing.
   (2) Functions:
      a. get_empty_framebuf() → Fetch an empty buffer.
      b. get_full_framebuf() → Fetch a filled buffer.
      c. get_inf_framebuf() → Fetch a buffer currently under inference.

```
static S_FRAMEBUF *get_empty_framebuf();
static S_FRAMEBUF *get_full_framebuf();
static S_FRAMEBUF *get_inf_framebuf();
```

Figure: Image Buffer Management Example

2. Model Initialization
   (1) Steps:
      a. Initialize YOLO model and allocate tensor buffers.
      b. Load model labels and length information.

```
arm::app::YoloXnanoNu model;
model.Init(arm::app::tensorArena, sizeof(arm::app::tensorArena),
           arm::app::yoloxnanonu::GetModelPointer(), arm::app::yoloxnanonu::GetModelLen());
```

Figure: Model Initialization Example.

3. Input Image Processing
   (1) Purpose: Resize input images to the model's required dimensions and apply format conversion (e.g., int8).
   (2) Example Code:

```
imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);
if (model.IsDataSigned()) {
    arm::app::image::ConvertImgToInt8(inputTensor->data.data, inputTensor->bytes);
}
```

Figure: Image Preprocessing Example

4. Inference Trigger
   (1) Purpose: Use FreeRTOS queues to send inference tasks, enabling object detection on the current image.

5. Post-Processing and Result Presentation
   (2) Purpose:
      a. Draw detection bounding boxes and labels (DrawImageDetectionBoxes()).
      b. Present inference results (PresentInferenceResult()).

```
imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);
if (model.IsDataSigned()) {
    arm::app::image::ConvertImgToInt8(inputTensor->data.data, inputTensor->bytes);
}
```

Figure: Result Display Example

6. Main Loop Logic
   (1) Execution Logic:
      a. Continuously check buffer status and perform corresponding processing.
      b. Insert delay (vTaskDelay(1)) between iterations to avoid excessive CPU usage.

```
while (1) {
    infFramebuf = get_inf_framebuf();
    fullFramebuf = get_full_framebuf();
    emptyFramebuf = get_empty_framebuf();

    vTaskDelay(1);
}
```

Figure: Main Loop Logic Example

## 6.6 Future Outlook

In future development, HyperRAM can be leveraged to enable the training of larger models. This expansion would support recognition across a greater number of classes, thereby improving accuracy. Ultimately, such advancements can facilitate deployment on a waste conveyor belt system, where the model, in combination with automated machinery, would enable fast and precise fully automated waste sorting.

References:

English


- CUDA Toolkit
  https://developer.nvidia.com/cuda-toolkit
- Netron
  https://netron.app/


Traditional Chinese


- LabelImg Tutorial
  https://hackmd.io/@osense-rd-public/H1ekDPqBt
- Anaconda Environment Setup
  https://hackmd.io/@nB1rzit6Toq8WdkZosRK_Q/r16-c7Ynj

# 7 Smart Home Application 2 – Personnel Tracking

## 7.1 Case Introduction – Personnel Tracking

The core objective of this case is to implement a smart home personnel detection system based on the Nuvoton M55M1 hardware platform combined with a machine learning model.

Key characteristics:

- High-precision target detection: Accurately identifies personnel in residential environments.
- Real-time inference: Suitable for edge computing scenarios.
- Low-power design: Optimized for long-term continuous operation.

Application scenarios include home security monitoring, elderly care, and resource management.

The accompanying illustration demonstrates personnel detection running on the M55M1 platform, showing its ability to process and display detection results in real time. The system design is aligned with embedded applications, offering excellent computational performance and hardware compatibility.

**System Requirements**

The main requirement of the personnel detection system is safety enhancement, especially in potentially hazardous environments such as factories and construction sites. The system must be able to accurately monitor entries and exits within designated areas to ensure compliance with safety regulations.

Core requirements:

- Safety: Monitor restricted or hazardous areas and prevent unauthorized personnel access.
- Real-time operation: Provide immediate detection and alarm functionality.
- Ease of use: Must be simple to install and operate, adaptable to diverse environments.

**Example Use Cases**

1. Industrial production line
   - Restrict entry of unauthorized personnel.
   - Install detection equipment in operational zones to monitor access.
   - Trigger immediate alarms upon abnormal entry.
2. Construction site
   - Monitor access to high-risk areas to prevent accidents.
   - Record activity data for subsequent analysis and safety compliance review

**Case Results**

This case is based on TinyML techniques for model training and deployment. Testing was performed both on the PC side and on the development board.

Results:

PC-side verification: Achieved 97% accuracy (143/148) on the validation dataset, demonstrating high reliability under ideal conditions.

Development board deployment: Achieved 92.5% accuracy (136/148), confirming robust performance in real-world embedded environments.

**Technical Workflow**

To achieve efficient model operation on the development board, the following workflow was adopted:

(1) TinyML Training

Develop and train a lightweight model tailored for edge computing requirements.

(2) Model Quantization and Optimization

Apply compression and optimizations to adapt the model for low-compute embedded devices.

(3) Model Vela Compilation and Configuration

Use Arm® Vela to compile the model, ensuring optimized support for the M55M1 platform.

(4) C Language Inference Implementation

Develop inference code integrating the model with the hardware interfaces.

Deployment to Development Board: Flash the final model and code into the board. Conduct on-site testing and debugging to validate performance.

## 7.2 Model Comparison

1. Model Size and Inference Speed

   In this project, the model selection prioritizes both performance and resource efficiency, especially for low-compute embedded platforms.

   (1) YOLOX Nano (used in this project)
   a. Extremely small model size: ~0.91M parameters, making it suitable for resource-constrained devices.
   b. High efficiency and low power: Optimized for low-compute platforms, enabling real-time applications such as smart cameras or drones.
   c. Fast inference: Runs efficiently on platforms with built-in NPUs (e.g., Nuvoton M55M1), delivering quick detection results.

2. Training Efficiency

   Training efficiency is an important factor in evaluating practical usability.

   (1) YOLOX Nano (used in this project)
   a. Anchor-free design: Simplifies training by removing anchor dependency, reducing the need for complex hyperparameter tuning.
   b. Strong dataset adaptability: Adapts easily to new datasets, especially for dynamic or rapidly changing data.

   (2) Other YOLO models
   a. YOLOv4 / YOLOv5:

      Anchor-based models, requiring careful anchor configuration and dataset-specific tuning. More hyperparameters to adjust, leading to higher training complexity.

   b. YOLOX Standard versions:

Also anchor-free like YOLOX Nano. However, larger in size and require higher computational resources for training, making them more suitable for high-performance systems.

3. Application Scenarios

   Different YOLO variants are tailored for different levels of compute resources and application needs.

   (1) YOLOX Nano (used in this project)

   Best suited for low-compute or latency-sensitive environments:

   a. Smart Home: Personnel detection, smart door locks, rapid response to environmental changes.
   b. Mobile Devices: Embedded object detection in smartphone apps with real-time requirements.
   c. Industrial Edge Computing: Compact devices like IP cameras on production lines requiring fast object recognition.

   (2) Other YOLO models

   YOLOX Standard versions: Ideal for professional-grade use cases requiring higher precision and throughput.

   a. Autonomous driving: High-reliability perception systems.
   b. Commercial surveillance: Large-scale monitoring with strict accuracy demands.

## 7.3 LabelImg

LabelImg is an open-source image annotation tool widely used for preparing datasets in object detection tasks. It supports exporting annotations in both COCO JSON and PASCAL VOC XML formats, making it compatible with mainstream deep learning frameworks.

### 7.3.1 Installation

Before installation, ensure that a Python environment is properly configured. Execute the following command in the terminal to install the LabelImg package:

```
pip install labelImg
```

This command automatically retrieves and installs the LabelImg package without the need for additional manual configuration.

### 7.3.2 Execution

Once installed, LabelImg can be launched directly through the terminal by running:

```
labelImg
```

Upon startup, the interface allows users to load images and annotate objects via bounding boxes. The tool then generates the corresponding annotation files. Supported Output Formats:

- PASCAL VOC (XML format)
- COCO (JSON format)

This ensures smooth integration into popular AI workflows, including TensorFlow, PyTorch, and other object detection pipelines.

### 7.3.3 Reference Resources

For extended guidance and detailed operation examples, the following resource may be consulted:

- (English) Installing and using Labelimg to annotate images | by Samin Karki | Medium
  https://medium.com/@samn.krki/installing-and-using-labelimg-to-annotate-images-92f754e910a5

- (Simplified Chinese) LabelImg Usage Tutorial (CSDN Blog)
  https://blog.csdn.net/knighthood2001/article/details/125883343

The tutorial provides comprehensive instructions, including installation, execution, annotation examples, and export format configuration, making it suitable for both beginners and experienced practitioners in AI dataset preparation.

## 7.4 Model Training on PC with Anaconda Environment

### 7.4.1 Training Hardware Specifications

1. GPU: Nvidia GTX 1080 Ti

   To verify GPU specifications:

   - Method 1: Execute the nvidia-smi command in the terminal to display GPU model and details.
   - Method 2: Open Task Manager, switch to the Performance tab, and check GPU information.

2. GPU determines the compatible CUDA and Torch versions.

   To verify supported CUDA versions for your GPU, refer to: PyTorch CUDA Compatibility Table.

### 7.4.2 Software and Version Requirements

- CUDA: 11.8
- Torch: 2.0.0
- Python: 3.10

### 7.4.3 Training Configuration and Results

- Training Duration: Approximately 1.5 hours.
- Training Dataset: ~600 labeled images used.

### 7.4.4 Environment Setup

Step 1: Install Anaconda3

Download and install from the Anaconda official site. Ensure the conda command is available after installation.

Step 2: Create Virtual Environment

```
conda create --name yolox_nu python=3.10
conda activate yolox_nu
```

Step 3: Clone YOLOX Nano Repository

```
gh repo clone MaxCYCHEN/yolox-ti-lite_tflite_int8
```

Step 4: Upgrade Pip and Tools

```
python -m pip install --upgrade pip setuptools
```

### 7.4.5 Environment Installation

Step 1: Install Core Frameworks

PyTorch, CUDA, and MMCV versions must match.

For GPU training (CUDA 11.8 + Torch 2.0.0), install MMCV as follows:

```
python -m pip install mmcv==2.0.1 -f
https://download.openmmlab.com/mmcv/dist/cu118/torch
2.0/index.html
```

Step 2: Install Additional Dependencies

```
python -m pip install --no-input -r requirements.txt
```

Step 3: Install YOLOX

```
python setup.py develop
```

### 7.4.6 YOLOX Nano Model Training

1. Model Constraints
   (1) Without external HyperRAM, only models <2MB are supported.
   (2) Selected model: YOLOX_Nano_ti_lite_nu (0.91MB), optimal for the M55M1 development board.

2. Training Command

```
python tools/train.py -f
exps/default/yolox_nano_ti_lite_nu.py -d 1 -b 64 --fp16 -o -
c
pretrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti_li
te.pth
```

Customizable Parameters

```
python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
<BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
```

Explanation of Training Parameters

- MODEL_CONFIG_FILE

  Refers to the model configuration file.

  This must be specified to point to the downloaded YOLOX Nano yolox-nano-ti-nu configuration file.

- BATCH_SIZE

  Defines the training batch size.

  Default: 64

  If system resources are insufficient, this value can be reduced to 32 or 16.

- PRETRAIN_MODEL_PATH

  Indicates the location of the pre-trained model weight file.

  This file can be downloaded from the YOLOX Nano pre-trained model directory.

### 7.4.7 Training Dataset Format

1. Dataset Structure
   (1) The dataset is organized according to the following structure and primarily includes these directories and files:
   - **Annotations** (annotation files): Contains training and validation annotation files in COCO JSON format.
   - **Train2017** (training images): Stores the training images.
   - **Val2017** (validation images): Stores the validation images.

   Example of dataset directory structure

```
datasets/<dataset_name>/
└── annotations/
    ├── train_annotation_json_file
    └── val_annotation_json_file
└── train2017/
    └── train img
└── val2017/
    └── validation img
```

   (2) Set the dataset paths

   Configure dataset paths in the program as follows:

```
self.data_dir = "datasets/hagrid_coco"
self.train_ann = "hagrid_train.json"
self.val_ann = "hagrid_val.json"
```

### 7.4.8 Custom Training Parameters (in train.py)

1. Primary adjustable parameters
   (1) **Image size** (Resolution):

   `self.input_size` and `self.test_size` define the input resolution for training and testing. The example setting is 320 × 320.

   (2) **Number of classes** (Classes):

   `self.num_classes` defines the total number of label categories to recognize. The example setting is 11 classes.

   (3) **Number of epochs** (Epochs):

`self.max_epoch` specifies the number of complete passes through the training dataset. The example setting is 150.

Note:

The number of epochs is not “the more, the better.” A recommended default range is 150–200. Excessive epochs may lead to overfitting, which can reduce model accuracy.

2. Parameter code example

```
self.input_size = (320, 320)
self.test_size  = (320, 320)
self.num_classes = 1
self.max_epoch  = 150
```

Corresponding code settings — example explanation

**Number of classes**: Set to 11 classes, meaning the model needs to recognize 11 object categories.

**Epochs**: Training runs for 150 full epochs, which is suitable for avoiding overfitting.

### 7.4.9 PyTorch to ONNX

Model conversion workflow

1. Convert a PyTorch model to ONNX format

   Use the following command:

   ```
   python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c
   <TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
   ```

   **`MODEL_CONFIG_FILE`**: Describes the model architecture and parameters. This is typically located in the YOLOX Nano folder.

   **`TRAINED_PYTORCH_MODEL`**: The trained PyTorch weights file (with .pth extension) used as the source for conversion.

   **`ONNX_MODEL_PATH`**: The output path and filename for the converted ONNX model.

2. Example command

   ```
   python tools/export_onnx.py -f
   exps/default/yolox_nano_ti_lite_nu.py -c
   YOLOX_outputs/yolox_nano_ti_lite_nu/latest_ckpt.pth --
   output-name
   YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onn
   x
   ```

3. Example explanation

   **exps/default/yolox_nano_ti_lite_nu.py**: Path to the model configuration file.

   **YOLOX_outputs/yolox_nano_ti_lite_nu/latest_ckpt.pth**: The most recent trained PyTorch checkpoint.

   **YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onn** x: Output path and filename for the converted ONNX model.

### 7.4.10 ONNX to TFLite

1. Model conversion workflow

   Convert the ONNX model into a format compatible with TensorFlow Lite. Execute the following command to perform the conversion.

2. Command template

   ```
   onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
   <CALI_DATA_NPY_FILE> "[[[0,0,0]]]"
   ```

   **ONNX_MODEL_PATH**: Path to the ONNX model file.
   **CALI_DATA_NPY_FILE**: Path to the calibration data file, usually a NumPy .npy file.

3. Concrete example command

   ```
   onnx2tf -i
   YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onn
   x -oiqt -qcind images
   YOLOX_outputs/yolox_nano_ti_lite_nu/calib_data_320x320_n200.
   npy "[[[0,0,0]]]"
   ```

   **ONNX_MODEL_PATH**: Replace with your actual ONNX file path, for example:

`YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onnx`.

**CALI_DATA_NPY_FILE**: Path to the calibration NumPy file, for example:

`YOLOX_outputs/yolox_nano_ti_lite_nu/calib_data_320x320_n200.npy`.

Calibration parameters:

[0,0,0] serves as the calibration value and can be adjusted based on specific requirements.

### 7.4.11 Vela Compiler and Conversion to Deployment Format

Use the Vela Compiler for model conversion.

1. Set model paths and output paths

   Configure the environment variables as follows:

   ```
   set MODEL_SRC_FILE=<your tflite model>
   set MODEL_OPTIMISE_FILE=<output vela model>
   ```

   `<your tflite model>`: Points to the original TFLite model file (.tflite).
   `<output vela model>`: Specifies the optimized model file produced by the Vela compiler.

2. Output file

   After compilation, the output file will be located at:

   `vela\generated\yolox_nano_ti_lite_nu_full_integer_quant_vela.tflite`

   This file will be used for subsequent deployment, enabling execution on the embedded platform.

## 7.5 Inference Program System Flow on the Development Board

This section describes the complete system workflow of the inference program on the development board, covering each step from system initialization to result display.

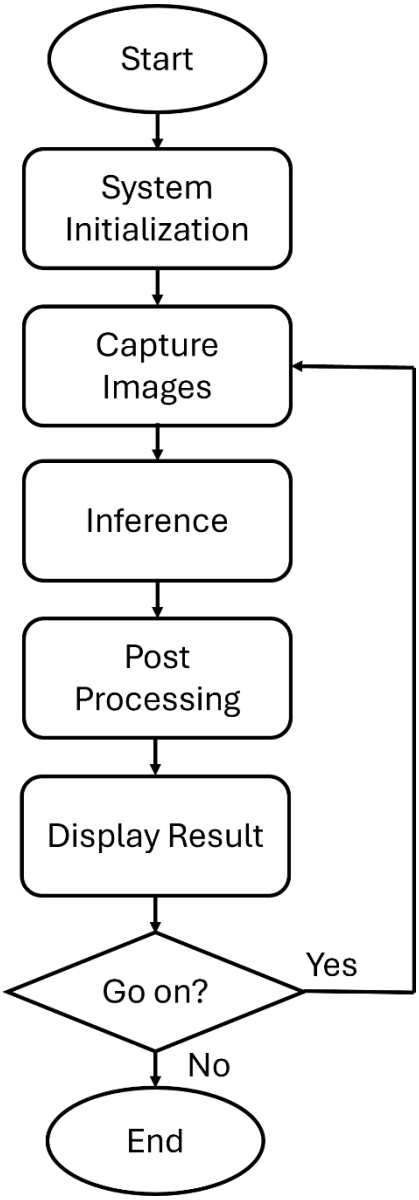


Figure: System Flow Diagram

1. System Workflow

(1) System Initialization

Initialize both hardware and software environments, load the model and configuration files, and prepare the inference runtime environment.

(2) Capture Image

Acquire real-time image input from the camera module or other image input devices to be used as input data for the model.

(3) Inference

Run the deployed model on the captured image to perform inference, generating preliminary detection results.

(4) Post-Processing & Draw Results

Apply post-processing to the raw model outputs, such as decoding bounding boxes and class labels, and overlay the detection results onto the image.

(5) Display Results

Render the processed image with detection results on the display device for user observation.

At the end of the workflow, the system will either continue with the next inference cycle or terminate the process based on user requirements.

2. Flowchart Explanation

This section provides a detailed explanation of each step in the inference program flowchart and its corresponding operational logic.

(1) Peripheral Initialization on the Development Board

Validate whether peripheral devices (e.g., sensors, camera modules) are successfully initialized. If initialization succeeds, the system enters image detection mode. If initialization fails, the workflow returns to hardware or software configuration checks.

(2) Start Image Capture

Activate the image capture function through the camera module to prepare real-time input data. The captured image is immediately passed to the deployed deep learning model for processing.

(3) Perform Inference

The system executes inference using the deployed model on the captured image. The model identifies and localizes target objects (e.g., persons). Outputs include preliminary detection results such as object positions and class labels.

(4) Enter Post-processing

Apply post-processing operations to the inference results. Detected targets are annotated with bounding boxes and classification labels.

For example, if a person is detected, the individual is highlighted with a bounding box to indicate position and size.

(5) Output Results

Render the processed image with detection annotations on the display screen. Provide users with a visual representation of the inference results.

(6) Flow Control

Based on user requirements:

If continuous detection is requested, the workflow loops back to the image capture step.

If termination is requested, the program execution ends.

3. System Initialization

System initialization is the first step in running the inference system. Its primary responsibility is to configure hardware resources to ensure that other modules operate reliably. The following describes the initialization workflow and the specific functionality and implementation of the `BoardInit()` function.

(1) `BoardInit()` Function

The `BoardInit()` function is a critical routine responsible for executing hardware-related initialization tasks, ensuring that fundamental hardware resources are fully prepared.

Objectives of execution:

- Provide a stable system clock (CLK) and reliable communication mechanisms to support the operation of other modules, such as image processing and neural network inference.

- Enable basic input/output (I/O) functionality to ensure proper interaction with peripherals.

(2) Core Code Explanation

Below is an excerpt of the `BoardInit()` function code with explanatory notes:

```
static void SYS_Init(void)
{
    /*---------------------------------------------------------------------------------------------------------*/
    /* Init System Clock                                                                                       */
    /*---------------------------------------------------------------------------------------------------------*/

    /* Enable Internal RC 12MHz clock */
    CLK_EnableXtalRC(CLK_SRCCTL_HIRCEN_Msk);

    /* Waiting for Internal RC clock ready */
    CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);

    /* Enable HXT clock */
    CLK_EnableXtalRC(CLK_SRCCTL_HXTEN_Msk);

    /* Waiting for HXT clock ready */
    CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk);

    /* Switch SCLK clock source to APLL0 and Enable APLL0 180MHz clock */
    CLK_SetBusClock(CLK_SCLKSEL_SCLKSEL_APLL0, CLK_APLLCTL_APLLSRC_HIRC, FREQ_180MHZ);

    /* Update System Core Clock */
    /* User can use SystemCoreClockUpdate() to calculate SystemCoreClock. */
    SystemCoreClockUpdate();
```

```
int BoardInit(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    SYS_Init();

    /* UART init - will enable valid use of printf (stdout
     * re-directed at this UART (UART6) */
    InitDebugUart();

    SYS_LockReg();                  /* Unlock register lock protect */

    HyperRAM_Init(HYPERRAM_SPIM_PORT);
    /* Enter direct-mapped mode to run new applications */
    SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);

    info("%s: complete\n", __FUNCTION__);
```

Figure: Example of `BoardInit()` function implementation

These steps collectively guarantee the stability and availability of the system hardware, providing the necessary foundation for higher-level functions such as image acquisition, neural network inference, and result visualization.

4. Explanation of Image Capture Program Segment

Image capture is one of the key steps in the inference system. It relies on the frame buffer to temporarily store the acquired image data. This section introduces the core logic and function implementations used in the image capture process.

Frame Buffer

The frame buffer is a memory region used to temporarily store captured image data. During system operation, the state of the frame buffer must be checked in order to locate an available buffer that can store new image frames.

(1) get_empty_framebuf Function

The primary purpose of the get_empty_framebuf function is to:

Traverse the frame buffer list to identify a buffer in the empty state (eFRAMEBUF_EMPTY).

Return a pointer to the available buffer, which will be used to store the next captured image frame.

Code Example:

```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_empty_framebuf() function implementation

This function iterates through all frame buffer states and returns the first available buffer found. If no buffer is available, it returns NULL.

(2) ImageSensor_Capture Function

The ImageSensor_Capture function performs the core operations of the image capture routine. Its workflow is as follows:

a. Invoke the get_empty_framebuf function to obtain an available frame buffer.
b. Use the image sensor to capture a new frame and store it into the selected buffer.
c. Handle potential errors during the capture process, such as reporting a failure when no valid image data is acquired.

Code Example:

```
void ImageSensor_Capture(uint32_t* frameImageData) {
    S_FRAMEBUF* emptyFramebuf = get_empty_framebuf();
    if (emptyFramebuf) {
        const uint8_t* pu8ImgSrc = get_img_array(u8ImgIdx);
        if (pu8ImgSrc == NULL) {
            printf("Failed to get image index");
            return;
        }
        memcpy(emptyFramebuf->frameImage.data, pu8ImgSrc, IMAGE_SIZE);
    }
}
```

Figure: ImageSensor_Capture() function implementation

Through this function, the captured image data is reliably stored in the frame buffer. The stored frames are then made available for the subsequent inference module, ensuring a smooth pipeline from data acquisition to neural network inference.

5. Inference

Inference is the core component of the entire system. It is responsible for analyzing the captured images and generating detection results. This section provides detailed explanations of the core functions and logic involved in the inference process.

(1) get_full_framebuf Function

The get_full_framebuf function retrieves an image frame from a filled frame buffer, which is then used as the input for inference operations.

Logic Overview:

- Iterates through all frame buffers.
- Returns the first buffer found in the filled state (eFRAMEBUF_FULL).
- If no buffer is found, returns NULL.

```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_FULL)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_full_framebuf() function implementation

(2) Inference Program Logic

The inference procedure consists of the following steps:

a. Obtain a filled frame buffer as the input image data.
b. Configure the inference task, including:
    - Input image size
    - Model dimensions (rows and columns)
    - Post-processing parameters
c. Submit the configured inference task into the inference queue, where it will wait for execution.

These steps enable the system to efficiently complete inference tasks and provide base data for subsequent post-processing.

(3) PresentInferenceResult Function

The final stage of inference is to visualize and present the detection results so that users can understand them.

The PresentInferenceResult function:

- Formats and outputs each detected object.
- Includes the object's class label and bounding box coordinates.

Code Example:

```
static bool PresentInferenceResult(const std::vector<arm::app::object_detection::DetectionResult> &results,
                                   std::vector<std::string> &labels)
{
    /* If profiling is enabled, and the time is valid. */
    info("Final results:\n");

    for (uint32_t i = 0; i < results.size(); ++i)
    {
        info("%" PRIu32 ") %s(%f) -> %s {x=%d,y=%d,w=%d,h=%d}\n", i,
             labels[results[i].m_cls].c_str(),
             results[i].m_normalisedVal, "Detection box:",
             results[i].m_x0, results[i].m_y0, results[i].m_w, results[i].m_h);
    }

    return true;
}
```

Figure: PresentInferenceResult() function implementation

This function generates a user-friendly representation of inference results. It is typically used for debugging or for presenting a concise summary of detection outputs.

6. Post-Processing — Non-Maximum Suppression (NMS)

Post-processing is a critical part of the object detection pipeline, aimed at refining inference outputs. The workflow includes:

(1) Scaling Detection Boxes

Rescale the detection boxes from model output space back to the original image dimensions.

Ensure bounding boxes align with the scale of the input image.

(2) Extracting Detection Results

Parse model output tensors to extract:

- Bounding box coordinates
- Classification probabilities

Sort detected objects by their confidence scores.

(3) Executing Non-Maximum Suppression (NMS)

NMS is the core step in object detection post-processing. Its purpose is to eliminate redundant detections.

Select the bounding box with the highest classification confidence.

Suppress overlapping boxes whose IoU (Intersection over Union) exceeds a predefined threshold.

Repeat until all boxes are processed.

This step significantly improves detection accuracy and reduces duplicate detections.

(4) Output Results

Store the refined detection results into a designated data structure (e.g., resultsOut vector).

Results can then be visualized or passed to downstream applications.

(5) Basic NMS Logic Summary

Sort all detection boxes by classification score (descending order).

Select the box with the highest score as a final box. Remove boxes with IoU greater than the threshold relative to the selected box.

Repeat until no boxes remain.

# 8 Smart Agriculture 1 – Apple Quality Recognition

## 8.1 Case Introduction – Apple Quality Recognition

**Abstract**

The core objective of this case study is to implement an intelligent apple quality recognition system based on the Nuvoton NuMicro® M55M1 hardware platform combined with a machine learning model. The key characteristics of this solution include:

- High-precision object detection, enabling accurate recognition of apple quality.
- Real-time inference support, meeting the requirements of edge computing applications.
- Low-power design, suitable for long-duration operation.

The platform is optimized for embedded system applications, offering strong computational performance and broad hardware compatibility.

**Requirements**

1. Standardization

   In apple quality recognition, standardization is a core requirement. Traditionally, apple grading has relied on manual inspection, which is prone to subjective bias and inconsistencies. By leveraging machine learning and computer vision technologies, the system applies unified grading criteria, eliminating subjective human error. Each apple is graded against the same standard, ensuring accuracy, consistency, and improved quality control.

2. High Efficiency

   Modern large-scale production lines demand high efficiency. The apple quality recognition system must support high-speed detection and process large volumes of apple image data within short timeframes. This capability:

   - Increases overall production throughput.
   - Reduces waiting time and resource waste.

- Significantly improves production line performance and economic benefits.

**Application Scenario**

Apple Sorting and Packaging Production Line:

Sorting and packaging are critical steps in apple processing. Traditional grading methods are time-consuming, labor-intensive, and prone to human bias, which reduces product consistency and market value.

By integrating machine learning and computer vision, the quality recognition system enables automated apple grading.

Workflow:

- A high-resolution camera captures images of each apple.
- A deep learning model analyzes external features such as color, shape, defects (e.g., spots, scratches, bruises)
- Based on extracted features, apples are automatically classified into quality tiers (e.g., high, medium, low).
- Classification results guide the automated packaging and labeling process.

Advantages:

(1) Increased efficiency – real-time grading without manual intervention improves sorting speed and accuracy.
(2) Reduced labor costs – automation minimizes the need for manual work, cutting costs and reducing human error caused by fatigue.
(3) Enhanced standardization – unified classification criteria ensure consistent product quality, improving consumer satisfaction and market competitiveness.

**Expected Results**

This case leverages TinyML technology for model training and deployment. Testing was performed both on PC and after deployment to the development board.

Results:

- After deployment to the development board, the recognition system achieved an accuracy rate of 92% (272/296).
- This demonstrates the model's strong real-world performance under practical application conditions.

## 8.2 Model Comparison

The model comparison in this case study is introduced in three aspects:

### 8.2.1 Model Size and Inference Speed

A key factor in model selection for this project is balancing performance and resource requirements, especially when optimizing for low-compute-power devices. The detailed comparison is as follows:

1. YOLOX Nano (Used in This Project)
   (1) Extremely small model size: Only ~0.91M parameters, making it highly suitable for deployment on hardware with limited resources.
   (2) High efficiency and low power consumption: Optimized for low-compute platforms, making it suitable for real-time applications such as smart cameras and drones.
   (3) Fast inference speed: Demonstrates excellent performance on low-power devices (e.g., M55M1), enabling rapid task execution with real-time results.

2. Other YOLO Models

   YOLOX Standard Versions: Support multiple model sizes (e.g., YOLOX-S, YOLOX-M).

   Advantage: Higher accuracy for demanding applications.

   Trade-off: Slower inference speed, especially with larger model variants.

### 8.2.2 Training Efficiency

Training efficiency is an important indicator when evaluating a model's practicality. The comparison between YOLOX Nano and other YOLO models is outlined below:

1. YOLOX Nano (Used in This Project)
   (1) Anchor-free architecture: Simplifies the training process by reducing dependence on anchors, thereby lowering hyperparameter tuning complexity.
   (2) Strong dataset adaptability: Easier to adapt to new datasets compared to other YOLO models, especially for dynamic or changing data sources.

2. Other YOLO Models
   (1) YOLOv4 / YOLOv5:

   Use anchor-based methods, requiring stricter dataset labeling and anchor configuration.

   Often need more hyperparameter adjustments to achieve optimal performance.

   (2) YOLOX Standard Versions:

   Also anchor-free, similar to YOLOX Nano in training workflow.

   However, due to larger model sizes, require greater computational resources for training. More suitable for high-performance hardware environments.

### 8.2.3 Application Scenarios

Different YOLO models are suited for different use cases depending on their design and performance requirements.

1. YOLOX Nano (Used in This Project)

Best suited for low-power or real-time applications, including:

(1) Smart home: e.g., smart locks and human presence detection, enabling quick responses to environmental changes.
(2) Mobile devices: On-device object detection for mobile apps with real-time processing requirements.
(3) Industrial edge computing: Real-time detection tasks on compact devices such as small cameras, e.g., for industrial production line monitoring.

## 8.3 LabelImg

LabelImg is an open-source image annotation tool designed to create labeled datasets for object detection tasks. Below are the installation and usage steps:

### 8.3.1 Installation of LabelImg

Ensure that a Python environment is available. Install LabelImg using pip:

```
$ pip install labelImg
```

### 8.3.2 Running LabelImg

After installation, launch LabelImg with the following command in the terminal:

```
$ labelImg
```

Once started, LabelImg allows you to load images, draw bounding boxes, and assign labels. The tool generates annotation files in formats such as COCO JSON or PASCAL VOC XML.

### 8.3.3 Reference Tutorials

For more detailed usage instructions, refer to the following guide:

- (English) Installing and using Labelimg to annotate images | by Samin Karki | Medium https://medium.com/@samn.krki/installing-and-using-labelimg-to-annotate-images-92f754e910a5

- (Simplified Chinese) LabelImg Usage Tutorial (CSDN Blog) https://blog.csdn.net/knighthood2001/article/details/125883343

## 8.4 Model Training in Anaconda Environment on PC

### 8.4.1 Training Hardware Specifications

1. GPU: Nvidia GTX 1080 Ti

   The GPU model and specifications can be verified through the following methods:

   - Method 1: Run the command nvidia-smi in the terminal to display GPU model and specifications.
   - Method 2: Open Task Manager, switch to the Performance tab, and check the GPU information to confirm the model and utilization status.
2. GPU determines the compatible CUDA and Torch versions.

   To verify supported CUDA versions for your GPU, refer to: PyTorch CUDA Compatibility Table.

### 8.4.2 Software and Version Requirements

- CUDA: 11.8
- Torch: 2.0.0
- Python: 3.10

### 8.4.3 Training Configuration and Results

- Training Duration: The model requires approximately 1.5 hours of training time.
- Training Dataset: Around 600 images were used as training data.

### 8.4.4 Training Environment Setup

Step 1: Install Anaconda3

Download and install Anaconda3 from the official website: Anaconda Download Page. After installation, verify that the conda command is accessible and functioning correctly.

Step 2: Create a New Environment

Open Anaconda Prompt (or terminal).

Create a dedicated environment with Python 3.10:

```
conda create --name yolox_nu python=3.10
```

Activate the environment:

```
conda activate yolox_nu
```

Step 3: Download YOLOX Nano Resources

Clone the YOLOX Nano GitHub repository:

```
gh repo clone MaxCYCHEN/yolox-ti-lite_tflite_int8
```

Step 4: Update Pip and Tooling

Ensure Pip and Setuptools are updated to the latest versions to avoid dependency conflicts:

```
python -m pip install --upgrade pip setuptools
```

### 8.4.5 Environment Installation

Step 1: Install Core Frameworks

Install PyTorch, CUDA, and MMCV.

PyTorch, CUDA, and MMCV versions must be compatible.

For CPU-only training, CUDA installation can be skipped.

For GPU training, install versions corresponding to your GPU specifications:

CUDA: 11.8
Torch: 2.0.0

Example MMCV installation command:

```
python -m pip install mmcv==2.0.1 -f
https://download.openmmlab.com/mmcv/dist/cu118/torch
2.0/index.html
```

Step 2: Install Additional Python Dependencies

Use the provided requirements.txt file to automatically install all dependencies:

```
python -m pip install --no-input -r requirements.txt
```

Step 3: Install YOLOX

From within the YOLOX project directory, execute the following to complete installation:

```
python setup.py develop
```

### 8.4.6 YOLOX Nano Model Training

1. Model Characteristics – Constraints

   Memory Limitation:

   Due to the absence of external HyperRAM, only models smaller than 2 MB can be deployed.

   Selected Model:

   The project uses YOLOX_Nano_ti_lite_nu, with a size of only 0.91 MB, making it the most suitable model for the M55M1 development board.

2. Training Procedure

   Execute Training Command:

The following command initiates training based on the specified model configuration file and pretrained model:

```
python tools/train.py -f
exps/default/yolox_nano_ti_lite_nu.py -d 1 -b 64 --fp16 -
o -c
pretrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti
_lite.pth
```

Custom Training Parameters:

Training parameters can be modified using the following general format:

```
python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
<BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
```

3. Parameter Definitions

`MODEL_CONFIG_FILE`

Path to the model configuration file. Must point to the configuration of `yolox_nano_ti_lite_nu` in the YOLOX Nano repository.

`BATCH_SIZE`

Defines the training batch size. Default: 64.
If system resources are limited, reduce to 32 or 16.

`PRETRAIN_MODEL_PATH`

Path to the pretrained model weights. These weights can be downloaded from the YOLOX Nano pretrained model directory.

### 8.4.7 Training Dataset Format

1. Dataset Structure

The dataset is organized into the following directories and files:

- **Annotations** (Label Files): Contains training and validation annotation files in COCO JSON format.
- **Train2017** (Training Images): Stores training images.
- **Val2017** (Validation Images): Stores validation images.

Example Directory Layout

```
datasets/<dataset_name>
  annotations/
      train_annotation_json_file
      val_annotation_json_file
  train2017/
      training images
  val2017/
      validation images
```

2. Dataset Path Configuration

   In the program, the dataset path should be specified. Example:

```
self.data_dir = "datasets/hagrid_coco"
self.train_ann = "hagrid_train.json"
self.val_ann = "hagrid_val.json"
```

### 8.4.8 Custom Training Parameters (train.py)

1. Primary Adjustable Parameters
   (1) Image Resolution (self.input_size & self.test_size):

   Defines the resolution for training and testing images. Example setting: 320 × 320.

   (2) Number of Classes (self.num_classes):

   Specifies the total number of annotated categories to be recognized. Example: 11 classes.

   (3) Epochs (self.max_epoch):

   Defines the number of full training iterations over the dataset. Example: 150.

   Note:

   More epochs are not always better. Recommended range: 150–200. Excessive epochs may lead to overfitting, reducing model accuracy.

2. Example Parameter Code

```
self.input_size = (320, 320)
self.test_size  = (320, 320)
self.num_classes = 1
self.max_epoch   = 150
```

Example Explanation:

**Number of Classes**: Set to 11, meaning the model is trained to recognize 11 object categories.

**Epochs**: Training runs for 150 complete epochs, a balanced choice to avoid overfitting while ensuring adequate training.

### 8.4.9 PyTorch to ONNX

Model Conversion Workflow

1. Convert PyTorch Model to ONNX Format

   Run the following command to perform the conversion:

   ```
   $ python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c
   <TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
   ```

   **`MODEL_CONFIG_FILE`**: Defines the model architecture and parameters. Typically located in the YOLOX Nano folder.

   **`TRAINED_PYTORCH_MODEL`**: Trained PyTorch model file (.pth format), used as the source for conversion.

   **`ONNX_MODEL_PATH`**: Path and filename for saving the converted ONNX model.

2. Example Command

   ```
   $ python tools/export_onnx.py -f
   exps/default/yolox_nano_ti_lite_nu.py -c
   YOLOX_outputs/yolox_nano_ti_lite_nu/latest_ckpt.pth --
   output-name
   YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onn
   x
   ```

3. Example explanation

   **`exps/default/yolox_nano_ti_lite_nu.py`**: Path to the model configuration file.

**YOLOX_outputs/yolox_nano_ti_lite_nu/latest_ckpt.pth**: Latest trained PyTorch checkpoint file.

**YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onnx**: Destination path and filename for the converted ONNX model.

### 8.4.10 ONNX to TFLite

1. Model Conversion Workflow

   Convert the ONNX model into TensorFlow Lite (TFLite) format using the following command:

   ```
   onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
   <CALI_DATA_NPY_FILE> "[[[0,0,0]]]"
   ```

   **ONNX_MODEL_PATH**: Path to the ONNX model file.
   **CALI_DATA_NPY_FILE**: Calibration dataset in .npy format, used for quantization.

2. Example Command

   onnx2tf -i YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onnx -oiqt -qcind images YOLOX_outputs/yolox_nano_ti_lite_nu/calib_data_320x320_n200.npy "[[[0,0,0]]]"

   Parameter Details

   **ONNX_MODEL_PATH**: Example file path YOLOX_outputs/yolox_nano_ti_lite_nu/vox_nano_nu_medicine.onnx

   **CALI_DATA_NPY_FILE**: Example calibration data file: YOLOX_outputs/yolox_nano_ti_lite_nu/calib_data_320x320_n200.npy

   Calibration Parameters:

   [0,0,0] specifies calibration mean values and can be adjusted as needed.

### 8.4.11 Vela Compiler – Conversion for Deployment

Use the Vela Compiler for model conversion.

1. Define Model Path and Output Path

   Use the following commands to configure model paths:

   ```
   $ set MODEL_SRC_FILE = <your tflite model>
   $ set MODEL_OPTIMISE_FILE = <output vela model>
   ```

   `<your tflite model>`: Path to the original TFLite model file (.tflite).
   `<output vela model>`: Path for saving the optimized model generated by Vela Compiler.

2. Output File

   After compilation, the optimized output file will be located at:

   `vela/generated/yolox_nano_ti_lite_nu_full_integer_quant_vela.tflite`

   This optimized .tflite model is used for deployment on embedded platforms, ensuring efficient inference execution.

## 8.5 Future Outlook

With the rapid advancement of technologies—particularly in the fields of Artificial Intelligence (AI), Machine Learning, the Internet of Things (IoT), and Embedded Systems—the future of apple quality recognition technology appears highly promising. From smart agriculture to automated production, such systems will not only enhance the market competitiveness of agricultural products but also significantly improve efficiency and quality control throughout the agricultural production process. The following outlines several potential directions for future development:

1. Higher-Precision Recognition Technology

   Future apple quality recognition systems will benefit from more advanced deep learning models and high-resolution image processing technologies. With continuous improvements in image sensor technology, next-generation systems will achieve finer-grained feature extraction for apples, enabling precise identification of attributes such as color, shape, and surface defects

(e.g., spots, scratches). Emerging models such as Convolutional Neural Networks (CNNs) and their variants—including self-attention mechanisms and Generative Adversarial Networks (GANs)—will further enhance recognition accuracy and processing speed.

2. Cross-Device and Cloud Integration

   With the growth of IoT, apple quality recognition will no longer be confined to a single embedded development board or device. Instead, it will be integrated across a broader range of smart devices, enabling collaborative operation. For example, sensors, cameras, and drones could work together to monitor orchards and perform quality inspection tasks. These devices will generate large-scale datasets, which can be processed via cloud platforms for remote monitoring and intelligent decision support, thereby increasing the level of automation and intelligence in agricultural production management.

3. Real-Time Processing and Edge Computing

   Driven by the rise of Edge Computing, future apple quality recognition systems will increasingly emphasize real-time processing capabilities, particularly in resource-constrained environments such as orchards. Embedded platforms such as Nuvoton M55M1 will continue to undergo optimization to support faster data processing. This enables on-device, real-time recognition without relying on cloud connectivity, which reduces latency while improving reliability and operational efficiency.

# 9 Smart Agriculture 2 – Fish Fry Counting

## 9.1 Case Introduction – Fish Fry Counting

### 9.1.1 Application Scenario

Using a camera module to capture underwater fish fry images, the Nuvoton M55M1 development board performs real-time inference. The system displays the fish fry classification results on the onboard LCD panel, while simultaneously outputting the current fry count via the terminal interface.

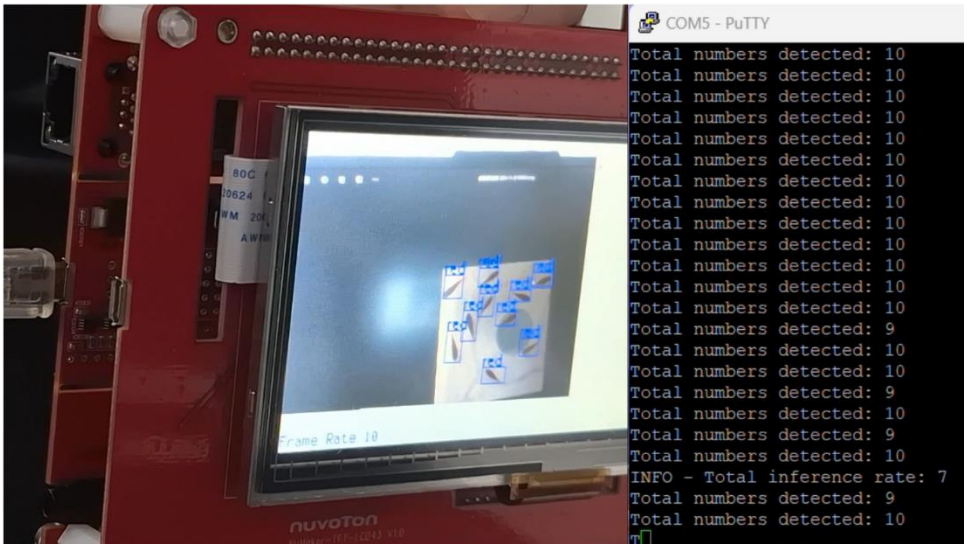


Figure: Fish Fry Counting Result

### 9.1.2 Project Summary

The dataset was trained using the YOLOX-ti-lite_tflite_int8 model. The model underwent a complete framework conversion pipeline, starting from PyTorch → ONNX → TensorFlow Lite, followed by quantization into INT8 format to reduce model size and enhance inference speed. This optimization improved both model efficiency and deployment feasibility.

Subsequently, the Vela Compiler was applied to allocate supported operations onto the NPU (Neural Processing Unit), further accelerating inference. The

optimized model was then compiled using Keil, and the resulting firmware was deployed to the Nuvoton M55M1 development board.

Upon execution, the system displays fry classification results on the LCD panel, and the fry count is reported on the PC terminal (PuTTY) in real time.

### 9.1.3 Data Processing and Inference Workflow

1. Dataset Preparation
2. Model Conversion and Optimization
3. PyTorch → ONNX → TensorFlow Lite
4. INT8 quantization and Vela compilation for NPU acceleration
5. Deployment – Model burned into M55M1 board
6. Counting Function Integration
7. Inference Output – Display fry classification on LCD, with real-time count displayed on PuTTY terminal


Figure: Data Processing and Model Inference Workflow

### 9.1.4 Results and Accuracy

Validation results demonstrated a recognition accuracy of 95.7% (180 correctly identified out of 188 samples in the test set).

## 9.2 System Workflow and Program Modules

1. Core Source Files

   `main.cpp`

   Responsible for system initialization and orchestrating the execution flow. Invokes other functional modules as the program entry point.

`BoardInit.cpp` / `BoardInit.hpp`

Handles hardware board initialization, including configuration of GPIO, I2C, SPI, and other peripheral interfaces.

`InferenceTask.cpp` / `InferenceTask.hpp`

Manages model loading and execution of inference tasks. Encapsulates the logic required to run AI models on the M55M1 development board.

`DetectorPostProcessing.cpp` / `DetectorPostProcessing.hpp`

Processes raw model outputs and converts them into usable bounding box results. Implements data parsing, Non-Maximum Suppression (NMS), and coordinate transformation. Required because models typically output numerical coordinates (e.g., x, y, w, h) or preformatted bounding boxes that must be post-processed before visualization.

`mpu_config_M55M1.h`

Defines hardware parameters and processor-specific configuration settings for the Nuvoton M55M1 development board.

2. Supporting Directories

Device / GCC / Keil / IAR:

Provide device drivers and compiler-specific project configuration files. Ensure portability across multiple toolchains.

Model:

Stores machine learning models and corresponding training results. Typically includes pre-trained .tflite or .onnx models optimized for deployment.

ProfilerCounter:

Contains performance analysis utilities. Used to measure execution time, memory usage, and inference latency, supporting system optimization.

Pattern:

Holds test samples or template datasets used for functional validation. Facilitates unit testing and benchmarking during development.

## 9.3 Dataset and Model Training

### 9.3.1 Dataset Preparation

1. Annotation Tool: Use LabelImg for dataset annotation.
2. Data Source: Self-captured images of Xiphophorus maculatus (Platyfish).
3. Annotation Format: COCO JSON format, with the following dataset distribution:
   - Training Set: 752 images
   - Validation Set: 188 images

### 9.3.2 Model Training

1. Framework: PyTorch
2. Model Selection: YOLOX-ti-lite_tflite_int8 – featuring high efficiency and lightweight properties, well-suited for embedded deployment.
3. Training Environment: NVIDIA GeForce RTX 4060 GPU; each training session requires approximately 60 minutes.
4. Model Source & Reference Link:
   - GitHub: https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

### 9.3.3 Model Framework Conversion

1. Convert from PyTorch → ONNX → TensorFlow Lite, followed by INT8 quantization to optimize model size and inference speed.
2. Quantization Objective: Balance accuracy with efficiency, ensuring suitability for embedded device deployment.

### 9.3.4 Environment Setup Steps

1. Create Python Environment
   (1) Using Anaconda:

```
conda create --name yolox_nu python=3.10
```

```
conda activate yolox_nu
```

(2) Upgrade pip and setuptools

```
python -m pip install --upgrade pip setuptools
```

(3) Install CUDA, PyTorch, and MMCV

Select CUDA version according to GPU configuration.

Actual versions used: CUDA 11.8, PyTorch 2.0, MMCV 2.0.1.

Installation command:

```
pip install mmcv==2.0.1 -f
https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/i
ndex.html
```

(4) Install Project Dependencies

```
python -m pip install --no-input -r requirements.txt
```

(5) Install YOLOX

```
python setup.py develop
```

### 9.3.5 Dataset Directory Structure

Prepare training data in the following structure under Datasets

```
Datasets/<your_datasets_name>/
├── annotations/
│   ├── train_annotation_json_file
│   └── val_annotation_json_file
├── train2017/   (training images)
├── val2017/     (validation images)
```

### 9.3.6 Training Parameter Configuration

In yolox_nano_ti_lite_nu.py:

```
self.input_size = (320, 320)
self.test_size = (320, 320)
self.num_classes = 1
self.max_epoch = 150
```

```
self.num_classes = 1
self.depth = 0.33
self.width = 0.25
self.input_size = (320, 320)
self.random_size = (10, 20)
self.mosaic_scale = (0.5, 1.5)
self.mosaic_prob = 0.5
self.enable_mixup = False
self.exp_name = os.path.split(os.path.realpath(__file__))[1].split(".")[0]
self.act = "relu"

# Define yourself dataset path
self.data_dir = "datasets/red_coco"
self.train_ann = "train2017_red.json"
self.val_ann = "val2017_red.json"

# --------------  training config --------------------- #
self.warmup_epochs = 5
self.max_epoch = 150
# -----------------  testing config ------------------ #
self.test_size = (320, 320)
```

Figure: Training Parameter Configuration

### 9.3.7 Pre-trained Model Training

1. Use `exps/default/yolox_nano_ti_lite_nu.py` as the default configuration file.
2. Update dataset paths:

```
self.data_dir = "datasets/your_coco"
self.train_ann = "your_train.json"
self.val_ann   = "your_val.json"
```

### 9.3.8 Start Training

```
python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
<BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
```

### 9.3.9 Model Conversion and Optimization

PyTorch → ONNX

```
python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c
<TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
```

Calibration Data Generation

```
python demo/TFLite/generate_calib_data.py --img-size
<IMG_SIZE> --n-img <NUMBER_IMG_FOR_CALI> -o
<CALI_DATA_NPY_FILE> --img-dir <PATH_OF_TRAIN_IMAGE_DIR>
```

ONNX → TFLite (with INT8 Quantization)

```
onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
<CALI_DATA_NPY_FILE> "[[[[1,1,1]]]]"
```

### 9.3.10 Vela Compilation

1. Place the quantized model into `vela/generated/`.
2. Edit `variables.bat` under the vela directory:

   ```
   set MODEL_SRC_FILE=<your tflite model>
   set MODEL_OPTIMISE_FILE=<output vela model>
   ```

3. Run `gen_model_cpp` to generate compilation results.
4. Output will appear at:

   ```
   vela/generated/yolox_nano_ti_lite_nu_full_integer_quant_vela
   .tflite.cc
   ```

## 9.4 C++ Design, Deployment, and Flashing

### 9.4.1 Program Workflow Overview

1. Software Function Design

   Main.cpp: Responsible for invoking board modules, including:

   - Camera module
   - LCD display module
   - Detection bounding-box rendering

2. Flashing Process

(1) Keil IDE: Used for embedded development and firmware flashing.
(2) Project Setup: Establish a project environment to support both hardware and software co-development.
(3) Model File Flashing: Deploy the trained ML model onto the target hardware using Keil's flashing utilities.

3. Execution Workflow
   (1) System Startup & Initialization
      a. Call `BoardInit.cpp` to initialize hardware resources.
      b. Initialize other subsystems such as memory allocation and peripheral activation.
   (2) Model Loading & Inference Task
      a. Load ML model from the `/Model` directory.
      b. Call InferenceTask.cpp to execute inference:
         (a) Pass image input data into the model.
         (b) Receive detection outputs (bounding boxes, labels).
   (3) Inference Result Post-Processing
      a. Call DetectorPostProcessing.cpp to handle model output:
         (a) Filtering and formatting (Non-Maximum Suppression, bounding-box decoding).
         (b) Convert results into interpretable bounding boxes.
   (4) Output & Response
      a. Return results to the main program.
      b. Display detection results on the LCD or transmit over a network interface.

### 9.4.2 Hardware Resource Configuration and Main Program Implementation

1. Main.cpp Responsibilities
   (1) Board Initialization: Initialize clocks, GPIO, and other fundamental peripherals.
   (2) Constant Definitions: Use USE_CCAP and USE_DISPLAY to configure camera sensor and LCD modules.
   (3) AI Model Initialization: Instantiate Arm®::app::YoloXnanoNu model and allocate Tensor Arena buffer.
   (4) Inference Task Creation: Use FreeRTOS xTaskCreate to establish inference tasks for processing image data.
   (5) Image Processing Workflow:

- Image acquisition, scaling, format conversion
- Model inference
- Post-processing: detection box drawing and label rendering

2. Main Program Flow
   - System Boot → Main Loop → Image Capture → Inference → Post-processing → Display → Repeat
   - When buffer empties, terminate execution.

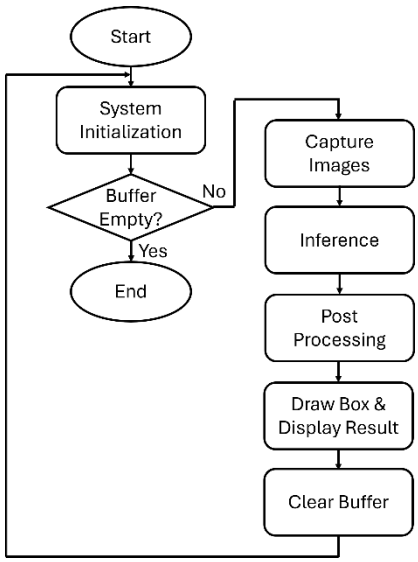


Figure: Main Program Flow

### 9.4.3 System Initialization – BoardInit()

1. Ensure reliable clock (CLK) sources and communication mechanisms for NN inference.
2. Initialize HyperRAM, UART, and Core Clock (CLK).
3. Example steps:
   - System Clock (SYS_Init): enable external RC oscillator and HXT clock source.

```c
static void SYS_Init(void)
{
    /*---------------------------------------------------------------------------------------------------------*/
    /* Init System Clock                                                                                       */
    /*---------------------------------------------------------------------------------------------------------*/

    /* Enable Internal RC 12MHz clock */
    CLK_EnableXtalRC(CLK_SRCCTL_HIRCEN_Msk);

    /* Waiting for Internal RC clock ready */
    CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);

    /* Enable HXT clock */
    CLK_EnableXtalRC(CLK_SRCCTL_HXTEN_Msk);

    /* Waiting for HXT clock ready */
    CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk);

    /* Switch SCLK clock source to APLL0 and Enable APLL0 180MHz clock */
    CLK_SetBusClock(CLK_SCLKSEL_SCLKSEL_APLL0, CLK_APLLCTL_APLLSRC_HIRC, FREQ_180MHZ);

    /* Update System Core Clock */
    /* User can use SystemCoreClockUpdate() to calculate SystemCoreClock. */
    SystemCoreClockUpdate();
```

Figure: CLK Initialization

- Update core frequency.
- Peripheral Initialization: camera, display, GPIO.
- UART setup for debugging/logging.
- HyperRAM configuration for large buffer allocation.

```c
int BoardInit(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    SYS_Init();

    /* UART init - will enable valid use of printf (stdout
     * re-directed at this UART (UART6) */
    InitDebugUart();

    SYS_LockReg();                  /* Unlock register lock protect */

    HyperRAM_Init(HYPERRAM_SPIM_PORT);
    /* Enter direct-mapped mode to run new applications */
    SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);

    info("%s: complete\n", __FUNCTION__);
```

Figure: Peripheral Initialization

### 9.4.4 Main Loop

The main loop in main_task() orchestrates the capture-inference-display cycle:

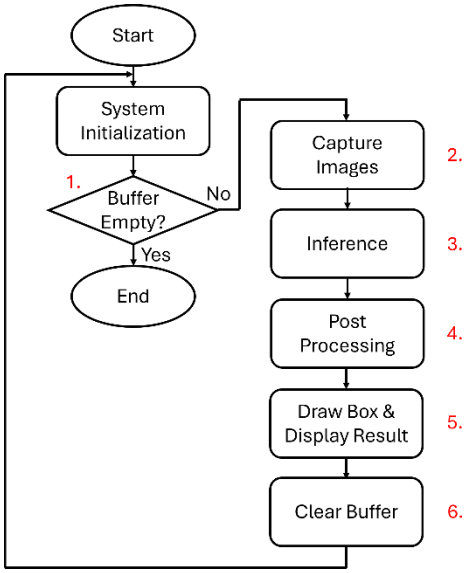


Figure: Main loop

1. Check Buffers
   - If no empty frame buffer is available, terminate loop.
2. Image Capture
   - Acquire images from the camera sensor into available frame buffers.
3. Inference Execution
   - Retrieve data from full buffers → run YOLOX model inference.
4. Post-Processing
   - Scale detection boxes to match original resolution.
   - Filter by confidence thresholds.
   - Apply Non-Maximum Suppression (NMS) to eliminate redundant boxes.
5. Result Visualization
   - Draw bounding boxes via DrawImageDetectionBoxes.
   - Render class labels with imlib_draw_string.
6. Buffer Recycling
   - Clear used frame buffers for next capture cycle.

[Notes]

1. Main Loop Multithreading Optimization

   The main loop can be optimized through a multithreaded approach, for example by allocating image acquisition, inference, and post-processing to separate threads. This parallelization enhances overall system efficiency.

2. Model Inference and Dynamic Memory Management

   After each inference execution, dynamically release memory resources to prevent memory leaks and ensure stable long-term system operation.

### 9.4.5 Buffer Management Functions

1. Frame Buffer

   The frame buffer (framebuf) in the main_task() function's main loop is a critical component for managing image data acquired from the camera. Its primary function is to temporarily store captured frames before performing format conversion, inference, and subsequent processing.

   (1) Actual Process

   a. get_empty_framebuf(): Obtains an empty buffer frame to store new image data.

```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_empty_framebuf() function

   b. get_full_framebuf(): Retrieves a filled buffer frame, resizes it, and converts it into the YOLOX-Nano model input format.

```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_FULL)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_full_framebuf() function

c. get_inf_framebuf(): Retrieves a frame that has completed inference, overlays detection results (e.g., bounding boxes and labels), and displays it on the screen.

```
static S_FRAMEBUF *get_inf_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_INF)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_inf_framebuf() function

(2) Function Implementations

a. get_empty_framebuf(): Iteratively checks for available empty frames and returns one.
b. get_full_framebuf(): Uses similar logic to find frames already populated with captured data.
c. get_inf_framebuf(): Handles frames that have completed inference and passes them back to the display module.

2. Image Acquisition

During image acquisition, buffer management is crucial. The state of each buffer frame (empty, filled, or inference completed) determines whether capture and processing can proceed correctly.

Before capturing a new frame, the system must invoke get_empty_framebuf() to ensure an available buffer is allocated for new data.

```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: Avoid overwriting existing data when capturing images

Complete Acquisition Process

- Confirm buffer availability to avoid overwriting existing data.
- Acquire image data via the camera module interface and store it in the allocated buffer.
- If acquisition fails (e.g., unable to read data), execute error handling and retry logic.

[Notes]

- Multi-buffer Management: Introducing multiple buffer frames significantly improves throughput, particularly in high-frame-rate camera scenarios.
- Error Handling: Image acquisition errors must be logged clearly, with retry mechanisms to improve system stability.

3. Inference Execution

Inference involves extracting image data from filled buffers, feeding it to the YOLOX model, and storing or displaying the results.

(1) get_full_framebuf()

- Extracts a data block from the filled buffer for inference.

- Verifies availability by checking the buffer's state.
- The extracted data is then passed to the YOLOX model for inference.

```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_FULL)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_full_framebuf() function

```
    //trigger inference
    inferenceJob->responseQueue = inferenceResponseQueue;
    inferenceJob->pPostProc = &postProcess;
    inferenceJob->modelCols = inputImgCols;
    inferenceJob->mode1Rows = inputImgRows;
    inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
    inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
    inferenceJob->results = &fullFramebuf->results;

    xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
    fullFramebuf->eState = eFRAMEBUF_INF;
}
```

Figure: Execute inference code

(2) PresentInferenceResult Function

- Displays inference results, including detected object class labels and bounding box positions.
- Iterates over each detection result, formats it, and outputs to the console or display.
- Results can also be forwarded to other modules for further processing.

```cpp
static bool PresentInferenceResult(const std::vector<arm::app::object_detection::DetectionResult> &results,
                                   std::vector<std::string> &labels)
{
    /* If profiling is enabled, and the time is valid. */
    info("Final results:\n");

    for (uint32_t i = 0; i < results.size(); ++i)
    {
        info("%" PRIu32 ") %s(%f) -> %s {x=%d,y=%d,w=%d,h=%d}\n", i,
             labels[results[i].m_cls].c_str(),
             results[i].m_normalisedVal, "Detection box:",
             results[i].m_x0, results[i].m_y0, results[i].m_w, results[i].m_h);
    }

    return true;
}
```

Figure: PresentInferenceResult function

### 9.4.6 Post-Processing

Post-processing extracts useful information from the raw model output and visualizes it.

1. Core Steps
   (1) Bounding Box Scaling: Ensures detection boxes match the original image resolution.
   (2) Result Extraction: Retrieves bounding box coordinates and confidence scores.
   (3) Non-Maximum Suppression (NMS): Removes redundant overlapping boxes, retaining only the most relevant detections.
   (4) Result Storage: Stores finalized detections into structured result data.

2. Key Functions
   - DetectorPostprocessing::RunPostProcessing

     Performs full post-processing: initializes network structures, extracts bounding box data, executes NMS, and adjusts final results.

     Parameters:

     - `imgNetRows`, `imgNetCols`: Input image size for the network.
     - `modelOutput0`: Model output tensor.
     - `resultsOut`: Output vector to store processed results.

     Enhancements:

- `totalBoundingBoxes`: Tracks the number of detected bounding boxes.
- Increment totalBoundingBoxes when pushing results into resultsOut.
- Display total bounding box count via std::cout.
- Ensure only detections above the confidence threshold are counted (if (it.prob[j] > 0)).
- Example thresholds: Confidence = 0.6, NMS = 0.25.

- DetectorPostprocessing::InsertTopNDetections

  Inserts detection results into a ranked list by confidence score. Maintains a fixed-size list to avoid excessive memory usage.

- DetectorPostprocessing::GetNetworkBoxes

  Converts raw model outputs into bounding box coordinates. Filters detections by confidence.

  Processing Logic:

  - Computes stride values across scales.
  - Extracts coordinates (x, y, w, h).
  - Adjusts bounding box size and position according to stride.

### 9.4.7 Bounding Box Rendering and Display

Visualization is achieved by overlaying detection results directly on the image.

- `DrawImageDetectionBoxes` Function

  Draws bounding boxes and class labels on the image for each detection result.

  Parameters:

  `drawImg`: Pointer to the image buffer for rendering.
  `results`: List of detection results with bounding box information.
  `labels`: Class labels corresponding to detected objects.

```
static void DrawImageDetectionBoxes(
    const std::vector<arm::app::object_detection::DetectionResult> &results,
    image_t *drawImg,
    std::vector<std::string> &labels)
{
    for (const auto &result : results)
    {
        imlib_draw_rectangle(drawImg, result.m_x0, result.m_y0, result.m_w, result.m_h, COLOR_B5_MAX, 1, false);
        imlib_draw_string(drawImg, result.m_x0, result.m_y0 - 16, labels[result.m_cls].c_str(), COLOR_B5_MAX, 2, 0, 0, false,
                          false, false, false, 0, false, false);
    }
}
```

Figure: `DrawImageDetectionBoxes` function

Implementation:

- imlib_draw_rectangle: Draws bounding boxes on the image.
- imlib_draw_string: Renders object class labels near bounding boxes.

This visualization step ensures detection results are human-interpretable, making them suitable for analysis, debugging, or end-user display.

## 9.5 Conclusion and Future Outlook

### 9.5.1 Conclusion

Fingerling counting is a critical management process in aquaculture. Accurate fingerling enumeration not only enables farmers to optimize resource allocation but also enhances farming efficiency and ensures the healthy growth of aquatic species. With the advancement of technology, traditional manual counting methods are gradually being replaced by automated and intelligent techniques. In particular, image processing and artificial intelligence (AI)-based counting methods have demonstrated significant advantages in practical applications. These methods not only improve accuracy and efficiency but also meet the high-volume demands of large-scale aquaculture facilities.

This study introduces modern image processing techniques to design and implement a deep learning-based fingerling counting system. The system effectively identifies and counts fingerlings, significantly reducing human error while improving counting speed and accuracy. Experimental results demonstrate

that, compared with traditional methods, image recognition-based approaches better address the dynamic movements of fingerlings in water, thereby overcoming limitations of conventional counting techniques.

### 9.5.2 Future Outlook

Despite notable progress, several challenges remain for fingerling counting technologies. First, system accuracy requires improvement across varying water conditions and species. Factors such as underwater lighting, turbidity, and overlapping or densely packed fingerlings can adversely affect results. Future research may focus on enhancing system adaptability in complex environments, for example, by developing more efficient image processing algorithms to improve accuracy under low-light or turbid water conditions.

Second, with the expansion of aquaculture operations, the demand for big data processing and real-time monitoring is growing. Future fingerling counting systems may integrate cloud computing and Internet of Things (IoT) technologies to achieve full-process monitoring and data analytics. Such systems would continuously collect environmental data, fingerling growth status, and other relevant metrics, providing farmers with fine-grained decision support. Through advanced data analysis, optimal operational strategies could be recommended to further enhance aquaculture productivity.

Finally, with the continuous evolution of deep learning and AI, future fingerling counting systems are expected to become increasingly self-learning and adaptive. By leveraging ongoing data training, these systems will autonomously adapt to diverse aquaculture environments, progressively improving accuracy and efficiency. Moreover, the scope of application may expand beyond fingerlings to include monitoring of juvenile and adult fish populations, as well as contributing to conservation and ecological monitoring of aquatic species.

In summary, fingerling counting technologies possess substantial development potential. As technological advancements continue, these solutions are expected to further transform traditional aquaculture management models, delivering greater productivity gains and supporting the sustainable growth of the aquaculture industry.

### 9.5.3 References

English

- PyTorch: Previous Versions
  https://pytorch.org/get-started/previous-versions/

- YOLOX-TI-Lite (TFLite INT8) Implementation – GitHub Repository
  https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

Traditional Chinese

- Utilizing Conda for Establishing and Managing Python Virtual Environments
  https://medium.com/ai-for-k12/%E5%88%A9%E7%94%A8-conda-%E5%BB%BA%E7%AB%8B%E5%8F%8A%E7%AE%A1%E7%90%86-python-%E8%99%9B%E6%93%AC%E7%92%B0%E5%A2%83-cc1c89d96fa9

- HackMD: YOLOX Deployment Notes
  https://hackmd.io/@zxcasd89525/Syw8BypDi

# 10 Smart Agriculture 3 – Fish Species Recognition

## 10.1 Case Introduction – Fish Species Recognition

### 10.1.1 Application Scenario

Using a camera module to capture images of fish in water, the captured frames are processed by the development board. The recognition results are then displayed on the LCD panel of the Nuvoton M55M1 development board. In this case, 13 fish species were used to simulate real-world conditions (dataset images collected at Farglory Ocean Park).

Figure: Actual Recognition Results

### 10.1.2 Project Summary

The project adopts the YOLOX-ti-lite_tflite_int8 model to train the fish species dataset. The model undergoes a series of framework conversions: PyTorch → ONNX → TensorFlow Lite, followed by quantization into INT8 format to reduce model size and improve inference speed. This process optimizes both performance and footprint.

Next, the Vela compiler is applied to allocate suitable operations to the NPU, further accelerating inference. The optimized model is then compiled with Keil and programmed into the Nuvoton M55M1 development board, where recognition results are displayed in real time.

### 10.1.3 Data Processing and Inference Workflow

1. Dataset Preparation
2. Model Framework Conversion
3. Model Quantization & Vela Compilation
4. Deployment to Development Board
5. On-board Inference and Result Display


Figure: Data Processing and Inference Workflow

### 10.1.4 Results Summary

The experimental evaluation achieved a validation accuracy of 96.5% (250/259), demonstrating the system's effectiveness in recognizing multiple fish species under practical conditions.

## 10.2 System Workflow and Program Modules

1. Core Files

`main.cpp`
Responsible for system initialization and invoking other functional modules.

`BoardInit.cpp / BoardInit.hpp`
Handles hardware board initialization, including GPIO, $I^2C$, SPI, and other peripheral interfaces.

`InferenceTask.cpp / InferenceTask.hpp`
Manages model loading and execution of inference tasks.

`DetectorPostProcessing.cpp / DetectorPostProcessing.hpp`
Processes model output results and converts them into actual bounding boxes. Since the raw output of the model consists of coordinate data (e.g., x, y, w, h or pre-defined bounding boxes), this module performs:

- Output parsing

- NMS (Non-Maximum Suppression)
- Coordinate transformation logic

mpu_config_M55M1.h

Defines hardware parameters and processor-related configurations for the M55M1 development board.

2. Supporting Directories

Device, GCC, Keil, IAR
Provide device drivers and compiler-specific project configuration files.

Model:
Contains machine learning models and training artifacts.

ProfilerCounter:
Implements performance profiling and runtime analysis utilities.

Pattern:
Stores test cases, sample inputs, or template datasets for validation and benchmarking.

## 10.3 Dataset and Model Training

### 10.3.1 Dataset Preparation

1. Annotation Tool: The dataset was annotated using LabelImg.
2. Data Source: Images were collected on-site at Farglory Ocean Park, covering 13 fish species.
3. Annotation Format: COCO JSON format, with the following distribution:
   - Training Set: 1,031 images
   - Validation Set: 259 images

### 10.3.2 Model Training

1. Framework: PyTorch

2. Model Selection: YOLOX-ti-lite_tflite_int8 – optimized for high performance and lightweight deployment on embedded devices.
3. Training Environment: NVIDIA GeForce RTX 4060 GPU, training time ≈ 60 minutes per session.
4. Model Source & Reference:
   - GitHub Repository: https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

### 10.3.3 Model Framework Conversion

1. Conversion pipeline: PyTorch → ONNX → TensorFlow Lite.
2. Applied INT8 quantization to reduce model size and accelerate inference while preserving accuracy, tailored for embedded deployment.

### 10.3.4 Environment Setup Steps

1. Create Python Environment

```
conda create --name yolox_nu python=3.10
conda activate yolox_nu
```

2. Upgrade pip and setuptools

```
python -m pip install --upgrade pip setuptools
```

3. Install CUDA, PyTorch, and MMCV
   Selected versions based on GPU compatibility:
   - CUDA: 11.8
   - PyTorch: 2.0
   - MMCV: 2.0.1

   Installation:

```
pip install mmcv==2.0.1 -f
https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/inde
x.html
```

4. Install dependencies

```
python -m pip install --no-input -r requirements.txt
```

5. Install YOLOX

```
python setup.py develop
```

### 10.3.5 Dataset Directory Structure

Prepare training data in the following structure under Datasets

```
Datasets/<your_dataset_name>/
    annotations/
        train_annotation_json_file
        val_annotation_json_file
    train2017/   # training images
    val2017/     # validation images
```

### 10.3.6 Training Parameters

In yolox_nano_ti_lite_nu.py:

```
self.input_size = (320, 320)
self.test_size  = (320, 320)
self.num_classes = 13
self.max_epoch   = 150
```

```python
class Exp(MyExp):
    def __init__(self):
        super(Exp, self).__init__()
        # --------------- model config --------------- #
        self.num_classes = 13
        self.depth = 0.33
        self.width = 0.25
        self.input_size = (320, 320)
        self.random_size = (10, 20)
        self.mosaic_scale = (0.5, 1.5)
        self.mosaic_prob = 0.5
        self.enable_mixup = False
        self.exp_name = os.path.split(os.path.realpath(__file__))[1].split(".")[0]
        self.act = "relu"

        # Define yourself dataset path
        self.data_dir = "datasets/fish_coco_final"
        self.train_ann = "train2017.json"
        self.val_ann = "val2017.json"
```

Figure: Training Parameter Configuration

### 10.3.7 Use Pretrained Model

1. Configuration file: `exps/default/yolox_nano_ti_lite_nu.py`
2. Update dataset paths:

```
self.data_dir = "datasets/your_coco"
self.train_ann = "your_train.json"
self.val_ann   = "your_val.json"
```

### 10.3.8 Start Training

```
python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b
<BATCH_SIZE> --fp16 -o -c <PRETRAIN_MODEL_PATH>
```

### 10.3.9 Model Conversion & Quantization

PyTorch to ONNX

```
python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c
<TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
```

Generate Calibration Data

```
python demo/TFLite/generate_calib_data.py --img-size
<IMG_SIZE> --n-img <NUM_IMAGES> -o <CALI_DATA_NPY_FILE> --img-
dir <TRAIN_IMAGE_DIR>
```

Convert ONNX to TFLite

```
onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images
<CALI_DATA_NPY_FILE> "[[[[1,1,1]]]]"
```

### 10.3.10 Vela Compilation

1. Place quantized model under `vela/generated/`.
2. Update `variables.bat` with:

```
set MODEL_SRC_FILE=<your_tflite_model>
```

```
set MODEL_OPTIMISE_FILE=<output_vela_model>
```

3. Run gen_model_cpp to generate compiled output.
4. Final model output:

```
vela/generated/yolox_nano_ti_lite_nu_full_integer_quant_vela
.tflite.cc
```

## 10.4 C++ Design, Deployment, and Flashing

### 10.4.1 Program Flow Overview

1. Software Functional Design
   Main.cpp: Responsible for managing the core board modules, including:
   - Camera module
   - LCD display module
   - Detection box drawing functionality

2. Flashing Process
   (1) Keil IDE: Utilized for embedded development and flashing modules onto the target device.
   (2) Project Setup: Create a project supporting hardware–software co-development.
   (3) Model Deployment: Use Keil to flash the compiled model file into the target hardware.

3. Program Workflow
   (1) System Startup and Initialization
      a. Calls BoardInit.cpp to initialize hardware resources.
      b. Initializes memory allocation and peripheral modules.
   (2) Model Loading and Inference Execution
      a. Loads the machine learning model from the Model directory.
      b. Calls InferenceTask.cpp to execute inference:
         (a) Inputs image data into the model.
         (b) Receives inference outputs (e.g., detection results).
   (3) Inference Result Post-Processing
      a. Calls DetectorPostProcessing.cpp to:

(a) Apply filtering (e.g., Non-Maximum Suppression, bounding box refinement).
(b) Store or return results to the main program.

(4) Output and Response
a. Returns processed inference results to the main program.
b. Displays results on the LCD or transmits them over the network.

### 10.4.2 Hardware Resource Configuration and Main Program Implementation

Main.cpp Responsibilities

1. Board Initialization: Configure system clock, GPIO, and essential hardware modules.

2. Feature Flags:
   - USE_CCAP for enabling the image sensor (camera capture).
   - USE_DISPLAY for enabling the LCD display module.

3. AI Model Setup:
   - Instantiates the YOLO model using Arm®::app::YoloXnanoNu.
   - Initializes the model and allocates Tensor Arena memory buffers.

4. Inference Task Creation:
   - Uses FreeRTOS xTaskCreate() to spawn inference tasks, handling input image data.

5. Image Processing Pipeline:
   - Image capture
   - Resizing and format conversion
   - Inference execution
   - Post-processing (object detection, bounding box drawing, labeling)

### 10.4.3 Main Program Execution Flow

Steps:
1. System Startup and Initialization

- Hardware initialization
- Peripheral configuration

2. Main Loop Execution
   - Image Capture: Acquire frames from the camera module.
   - Inference: Feed frames into the AI model and obtain detection results.
   - Post-Processing: Apply bounding box decoding, filtering, and formatting.
   - Display Results: Draw detection boxes and labels, update LCD output.
   - Buffer Recycling: Clear memory buffers and repeat.

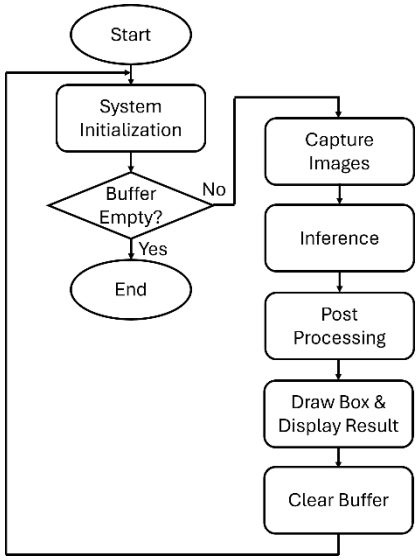


Figure: Main Program Flow

### 10.4.4 System Initialization

BoardInit() Responsibilities

1. Ensure stable system clocks (CLK) and reliable communication mechanisms.
2. Provide base-level I/O for downstream modules (image processing, neural inference).
3. Initialize:
   - System Clock (SYS_Init) with external RC and HXT sources.

```c
int BoardInit(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    SYS_Init();

    /* UART init - will enable valid use of printf (stdout
     * re-directed at this UART (UART6) */
    InitDebugUart();

    SYS_LockReg();                  /* Unlock register lock protect */

    HyperRAM_Init(HYPERRAM_SPIM_PORT);
    /* Enter direct-mapped mode to run new applications */
    SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);

    info("%s: complete\n", __FUNCTION__);
```

Figure: CLK Initialization

- Clock routing and update of system core frequency.
- Peripherals: UART, HyperRAM, and board-specific hardware.

Key Components

- UART Configuration: Supports communication and debugging output.
- HyperRAM Mapping: Allocates large buffers required for neural network workloads.

### 10.4.5 Main Loop Details

The main loop executes the system's full vision pipeline:

1. Buffer Availability Check
   - If no image data is available, exit the loop.

2. Image Acquisition
   - Capture new frame from camera sensor and push into buffer.

3. Inference Execution
   - Run detection model on preprocessed frame.

4. Post-Processing
   - Decode detection results, filter bounding boxes, finalize outputs.

5. Drawing and Display
   - Overlay bounding boxes on the image.
   - Output the processed image via LCD display.

6. Buffer Recycling
   - Clear memory to prepare for next capture cycle.

[Notes]

1. Multithreading Optimization

   Main loop tasks (image capture, inference, post-processing) can be parallelized across multiple threads to enhance throughput and system responsiveness.

2. Dynamic Memory Management

   Memory should be released after each inference cycle to prevent memory leaks and ensure long-term stability.

### 10.4.6 Buffer Management Functions

1. Frame Buffer

   The frame buffer (framebuf) in the main_task() function's main loop is a critical component for managing image data acquired from the camera. Its primary function is to temporarily store captured frames before performing format conversion, inference, and subsequent processing.

   (1) Actual Process
      a. get_empty_framebuf(): Obtains an empty buffer frame to store new image data.

```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_empty_framebuf() function

b. get_full_framebuf(): Retrieves a filled buffer frame, resizes it, and converts it into the YOLOX-Nano model input format.

```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_FULL)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_full_framebuf() function

c. get_inf_framebuf(): Retrieves a frame that has completed inference, overlays detection results (e.g., bounding boxes and labels), and displays it on the screen.

```c
static S_FRAMEBUF *get_inf_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_INF)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_inf_framebuf() function

(2) Function Implementations

a. get_empty_framebuf(): Iteratively checks for available empty frames and returns one.
b. get_full_framebuf(): Uses similar logic to find frames already populated with captured data.
c. get_inf_framebuf(): Handles frames that have completed inference and passes them back to the display module.

2. Image Acquisition

During image acquisition, buffer management is crucial. The state of each buffer frame (empty, filled, or inference completed) determines whether capture and processing can proceed correctly.

Before capturing a new frame, the system must invoke get_empty_framebuf() to ensure an available buffer is allocated for new data.

```c
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: Avoid overwriting existing data when capturing images

Complete Acquisition Process

- Confirm buffer availability to avoid overwriting existing data.
- Acquire image data via the camera module interface and store it in the allocated buffer.
- If acquisition fails (e.g., unable to read data), execute error handling and retry logic.

[Notes]

- Multi-buffer Management: Introducing multiple buffer frames significantly improves throughput, particularly in high-frame-rate camera scenarios.
- Error Handling: Image acquisition errors must be logged clearly, with retry mechanisms to improve system stability.

3. Inference Execution

   Inference involves extracting image data from filled buffers, feeding it to the YOLOX model, and storing or displaying the results.

   (1) get_full_framebuf()
   - Extracts a data block from the filled buffer for inference.
   - Verifies availability by checking the buffer's state.
   - The extracted data is then passed to the YOLOX model for inference.

```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++)
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_FULL)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```

Figure: get_full_framebuf() function

```
    //trigger inference
    inferenceJob->responseQueue = inferenceResponseQueue;
    inferenceJob->pPostProc = &postProcess;
    inferenceJob->modelCols = inputImgCols;
    inferenceJob->mode1Rows = inputImgRows;
    inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
    inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
    inferenceJob->results = &fullFramebuf->results;

    xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
    fullFramebuf->eState = eFRAMEBUF_INF;
}
```

Figure: Execute inference code

(2) PresentInferenceResult Function

- Displays inference results, including detected object class labels and bounding box positions.
- Iterates over each detection result, formats it, and outputs to the console or display.
- Results can also be forwarded to other modules for further processing.

```
static bool PresentInferenceResult(const std::vector<arm::app::object_detection::DetectionResult> &results,
                                   std::vector<std::string> &labels)
{
    /* If profiling is enabled, and the time is valid. */
    info("Final results:\n");

    for (uint32_t i = 0; i < results.size(); ++i)
    {
        info("%" PRIu32 ") %s(%f) -> %s {x=%d,y=%d,w=%d,h=%d}\n", i,
             labels[results[i].m_cls].c_str(),
             results[i].m_normalisedVal, "Detection box:",
             results[i].m_x0, results[i].m_y0, results[i].m_w, results[i].m_h);
    }

    return true;
}
```

Figure: PresentInferenceResult function

### 10.4.7 Post-Processing

Post-processing extracts useful information from the raw model output and visualizes it.

1. Core Steps
   (1) Bounding Box Scaling: Ensures detection boxes match the original image resolution.
   (2) Result Extraction: Retrieves bounding box coordinates and confidence scores.
   (3) Non-Maximum Suppression (NMS): Removes redundant overlapping boxes, retaining only the most relevant detections.
   (4) Result Storage: Stores finalized detections into structured result data.

2. Key Functions
   - DetectorPostprocessing::RunPostProcessing

     Performs full post-processing: initializes network structures, extracts bounding box data, executes NMS, and adjusts final results.

     Parameters:

     - `imgNetRows`, `imgNetCols`: Input image size for the network.
     - `modelOutput0`: Model output tensor.
     - `resultsOut`: Output vector to store processed results.

   - DetectorPostprocessing::InsertTopNDetections

     Inserts detection results into a ranked list by confidence score. Maintains a fixed-size list to avoid excessive memory usage.

   - DetectorPostprocessing::GetNetworkBoxes

     Converts raw model outputs into bounding box coordinates. Filters detections by confidence.

     Processing Logic:

     - Computes stride values across scales.
     - Extracts coordinates (x, y, w, h).
     - Adjusts bounding box size and position according to stride.

### 10.4.8 Bounding Box Rendering and Display

Visualization is achieved by overlaying detection results directly on the image.

- `DrawImageDetectionBoxes` Function

Draws bounding boxes and class labels on the image for each detection result.

Parameters:

`drawImg`: Pointer to the image buffer for rendering.
`results`: List of detection results with bounding box information.
`labels`: Class labels corresponding to detected objects.

```
static void DrawImageDetectionBoxes(
    const std::vector<arm::app::object_detection::DetectionResult> &results,
    image_t *drawImg,
    std::vector<std::string> &labels)
{
    for (const auto &result : results)
    {
        imlib_draw_rectangle(drawImg, result.m_x0, result.m_y0, result.m_w, result.m_h, COLOR_B5_MAX, 1, false);
        imlib_draw_string(drawImg, result.m_x0, result.m_y0 - 16, labels[result.m_cls].c_str(), COLOR_B5_MAX, 2, 0, 0, false,
                          false, false, false, 0, false, false);
    }
}
```

Figure: `DrawImageDetectionBoxes` function

Implementation:

- imlib_draw_rectangle: Draws bounding boxes on the image.
- imlib_draw_string: Renders object class labels near bounding boxes.

This visualization step ensures detection results are human-interpretable, making them suitable for analysis, debugging, or end-user display.

## 10.5 Conclusion and Future Outlook

### 10.5.1 Conclusion

This study successfully developed an efficient fish species recognition system based on YOLOX-ti-lite_tflite_int8, achieving real-time inference on embedded hardware. Through multi-stage model training, conversion, and optimization, we not only improved inference speed but also significantly reduced model size, making the system well-suited for deployment on resource-constrained devices.

1. Model Training and Conversion

   The fish species recognition model was initially trained using PyTorch, then converted to the ONNX format before being further transformed into a TensorFlow Lite model. Using INT8 quantization, we effectively reduced model size while accelerating inference. This quantization process minimized storage requirements and improved runtime efficiency, without sacrificing recognition accuracy, thus laying a foundation for deployment on embedded and low-power platforms.

2. NPU Acceleration and Performance Optimization

   To further optimize performance, the Vela compiler was applied to partition operations between the CPU and Neural Processing Unit (NPU). By offloading suitable operators onto the NPU, inference latency was substantially reduced. This optimization improved system responsiveness, demonstrating that real-time fish recognition is feasible on low-power hardware.

3. Development Board Deployment and Real-Time Visualization

   The optimized model was deployed to the Nuvoton M55M1 development board, where real-time inference and visualization of fish recognition results were successfully achieved. This deployment validates the practicality of the proposed system and highlights the potential of embedded AI inference technology for real-world, low-power applications.

### 10.5.2 Future Outlook

While the research achieved promising results, several areas warrant further exploration and enhancement:

1. Balancing Accuracy and Model Size

   Although quantization successfully reduced storage requirements and improved inference speed, precision degradation may occur under certain conditions. Future work could investigate advanced quantization methods (e.g., mixed precision or quantization-aware training) to achieve an optimal trade-off between accuracy and efficiency.

2. Improving Model Generalization

   Current performance is dataset-specific. To extend recognition robustness, future studies should incorporate diverse datasets and consider techniques such as domain adaptation or reinforcement learning to improve generalization in unseen environments.

3. Real-Time Processing and Latency Optimization

   Although real-time inference was achieved, additional optimization opportunities remain. Future work could leverage more complex model architectures, operator fusion, or software–hardware co-design to further minimize latency and deliver smoother real-time video processing.

4. Expansion to Broader Applications

   While the focus was on fish species identification, the framework can be extended to other domains such as biological species recognition, industrial quality inspection, or medical image analysis, demonstrating broad applicability in embedded AI.

5. Advancement in Low-Power Edge Devices

   With the rise of IoT and edge computing, low-power hardware will play a critical role. Future efforts should emphasize optimization for ultra-low-power platforms, enabling deployment of recognition systems in smaller, battery-powered, or field-deployed devices.

### 10.5.3 References

English

- PyTorch: Previous Versions
  https://pytorch.org/get-started/previous-versions/

- YOLOX-TI-Lite (TFLite INT8) Implementation – GitHub Repository
  https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

Traditional Chinese

- Utilizing Conda for Establishing and Managing Python Virtual Environments
  https://medium.com/ai-for-k12/%E5%88%A9%E7%94%A8-conda-%E5%BB%BA%E7%AB%8B%E5%8F%8A%E7%AE%A1%E7%90%86-python-%E8%99%9B%E6%93%AC%E7%92%B0%E5%A2%83-cc1c89d96fa9

- HackMD: YOLOX Deployment Notes
  https://hackmd.io/@zxcasd89525/Syw8BypDi