אוניברסיטת
אריאל
בשומרון
המחלקה להנדסת חשמל ואלקטרוניקה

Soldier Vs. Civilian Embedded Vision System

Final Project, B.Sc.

Department of Electrical Engineering and Electronics

Submitted by:

Chagai Perez I.D: 207460130

Samuel Newman I.D: 342785912

Project Supervisor: Dr. Emmanuel Bender

Project Head: Prof. Joseph Bernstein

Table of Contents

Acknowledgements

We would like to express our sincere gratitude to our project mentor, Dr. Emanuel Bender, whose expert guidance, valuable insights, and continuous support have been instrumental throughout the course of this project. His mentorship enabled us to tackle challenges with confidence and greatly enhanced the quality of our work.

We extend our appreciation to Prof. Yosef Bernstein, Head of the VLSI Specialty in the Department of Electrical and Electronic Engineering, for fostering an encouraging academic environment and providing us with essential resources and support during our studies.

Special thanks are due to our friends and colleagues, whose advice, collaboration, and encouragement contributed significantly to our progress and motivated us to achieve our goals.

Finally, our deepest appreciation goes to our wives for their unwavering support, patience, and encouragement throughout the demanding phases of this project. Their constant understanding and motivation were indispensable to the successful completion of our work.

Abstract

This project presents the design and implementation of a real-time soldier versus civilian detection system using deep learning and embedded vision. The system employs a YOLOv8 neural network, trained on a custom dataset to classify individuals based on appearance, uniform, and visible equipment. The model was trained locally using Roboflow for data preparation and Ultralytics YOLOv8 for model development, then exported to ONNX format for deployment on the AMD-Xilinx Kria KR260 Robotics Starter Kit.

The KR260 board executed real-time inference using ONNX Runtime within a PYNQ/Jupyter environment, with detections transmitted through GPIO outputs to physical indicators: a green LED for soldiers and a red LED with a buzzer for civilians. The system was tested under various conditions including different lighting levels, degrees of movement, and clothing combinations to assess both accuracy and responsiveness. The model consistently succeeded in detecting and distinguishing objects according to its training, demonstrating the feasibility of performing visual recognition on embedded hardware.

While the system achieved smooth operation and strong performance on a laptop, inference on the KR260's CPU was limited by processing constraints. Running in headless mode, without live video display, did not yield any measurable frame-rate improvement, with both configurations averaging around 4–5 FPS. This confirmed that performance bottlenecks stemmed primarily from CPU-bound inference rather than visualization overhead. Despite this limitation, the system operated reliably in real time, correctly triggering LED and buzzer feedback for every classification event.

The project demonstrates the potential of combining deep learning, FPGA-based hardware, and embedded systems to create intelligent, safety-oriented detection platforms. With further improvements such as expanding the training dataset, optimizing preprocessing, and implementing Vitis AI acceleration on the board's Deep Processing Unit (DPU), the system could achieve significantly higher frame rates and robustness. Beyond this specific use case, the experience gained in model deployment, hardware-software co-design, and real-time vision processing provides valuable insight for future work in embedded AI, defense automation, and smart safety systems.

Motivation

The primary motivation for this project is to investigate and demonstrate how real-time embedded vision can be used to reduce misidentification risks and to aid rapid threat-assessment in surveillance scenarios. Specifically, we focus on minimizing false positives where civilians are incorrectly classified as soldiers an error with potentially serious ethical and safety consequences and on producing a reliable, auditable indicator system (visual + audible) suitable for human in the loop monitoring. The project combines lightweight deep-learning (YOLOv8) with FPGA-based embedded deployment (Kria KR260) to explore trade-offs between accuracy, latency, and real-world robustness[1].

[1] **Ethical and Safety Considerations:** As this project involves automated recognition of people, ethical responsibility and safety are critical. The system is strictly designed for research and educational purposes and must not be used in autonomous or combat decision-making. Its intended application is in surveillance or security contexts where a human operator remains in control. Dataset bias, lighting variability, and limited training diversity may cause misclassification; therefore, any deployment must include human verification and appropriate disclaimers. Future work should focus on expanding datasets to represent diverse populations, ensuring fairness, and incorporating explainable-AI mechanisms to improve transparency and accountability.

Introduction

Theoretical background

1. Theoretical and Conceptual Foundations

Computer Vision

Computer vision is a field of artificial intelligence that enables computers to understand and interpret images and videos, much like how humans use their eyes and brain. One of the most common tasks in computer vision is object detection, which involves identifying and locating specific objects such as people, vehicles, or phones within an image or video frame. Object detection uses machine learning models that are trained on large sets of labeled images, learning to recognize patterns, shapes, and features that distinguish different objects. When a new image is processed, the model analyzes it and draws boxes around the objects it recognizes, labeling them with a confidence score. This technology is widely used in applications like self-driving cars, security systems, and robotics. In embedded systems, object detection allows devices to make decisions based on what they "see" in real time, such as triggering lights, sounds, or actions when a specific object is detected.

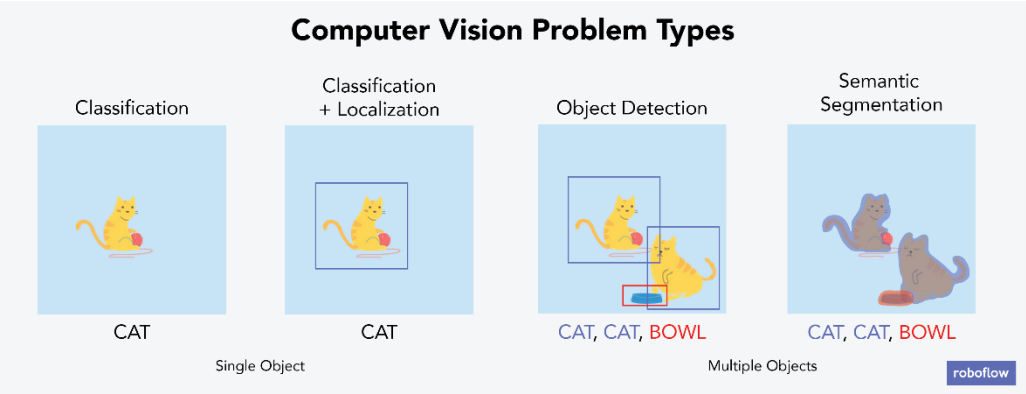

Figure I: *Computer Vision*

Machine Learning vs. Deep Learning in Computer Vision

Machine Learning (ML) is a broader field of artificial intelligence where algorithms learn patterns from data to make predictions or decisions. In classical machine learning for computer vision, developers often manually extract features (like edges, corners, colors) from images and then feed these features into simpler models such as decision trees, support vector machines (SVMs), or k-nearest neighbors (k-NN) to classify or detect objects. These models can work well for simple tasks but tend to struggle with complex, real-world visual data.

Deep Learning (DL) is a specialized subset of machine learning that uses multi-layered neural networks especially convolutional neural networks (CNNs) to automatically learn high-level features directly from raw images. Instead of manually designing feature extractors, deep

learning models learn to detect shapes, textures, and object patterns by themselves during training. This makes them extremely effective for complex computer vision tasks such as object detection, image classification, and segmentation.

For example, YOLOv8 is a deep learning model, specifically built on convolutional neural networks. It doesn't rely on manually defined features instead, it learns everything directly from raw image data during training. This enables it to detect objects with high accuracy and speed, even in varied lighting, angles, or backgrounds.

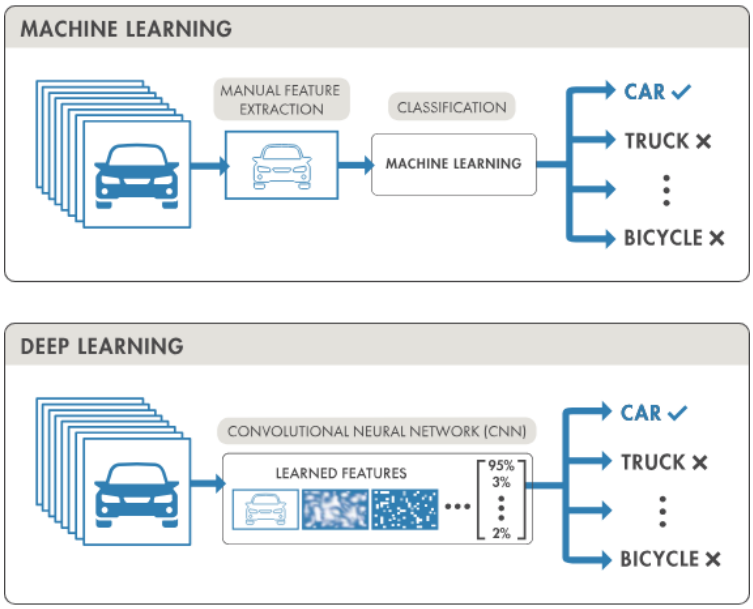

Figure II: *Machine vs. Deep Learning*

Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a type of deep learning model that is especially good at analyzing visual data like images and videos. CNNs are designed to automatically learn patterns such as edges, shapes, textures, and objects from raw pixel data, without the need for manual feature extraction. The key component of a CNN is the convolutional layer, which slides small filters (also called kernels) over the image to detect specific features. These features are passed through multiple layers each learning more complex patterns until the network can understand the content of the image. In object detection models like YOLOv8, CNNs are used to process the input image, detect the presence of objects, and predict their locations and categories. Because of their ability to learn spatial hierarchies and patterns, CNNs are the foundation of most modern computer vision systems used in tasks such as facial recognition, self-driving cars, medical imaging, and real-time object detection.

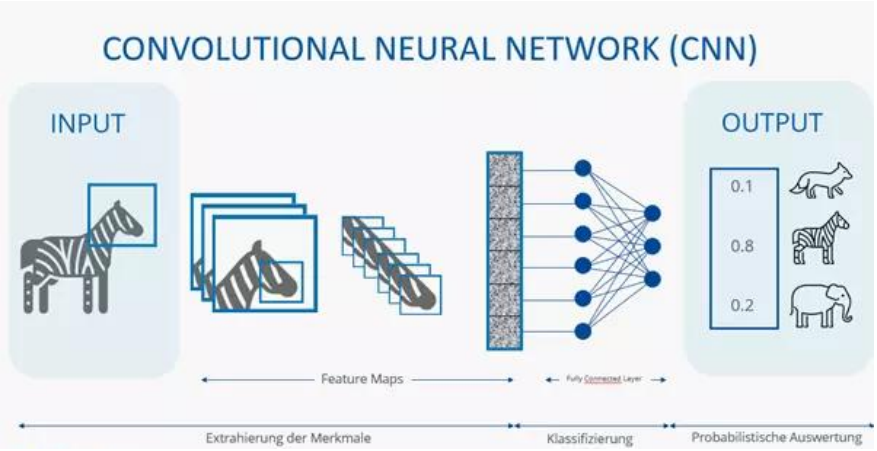

Figure III: *CNN*

YOLOv8 (You Only Look Once, version 8)

YOLOv8 (You Only Look Once, version 8) is an advanced deep learning model used for object detection, image segmentation, and classification. Developed by Ultralytics, it continues the YOLO family's focus on achieving a strong balance between speed, accuracy, and efficiency. YOLOv8 uses a Convolutional Neural Network (CNN) to analyze images and automatically learn patterns such as edges, shapes, and textures. Unlike traditional methods that examine small image regions separately, YOLOv8 performs detection in a single pass, predicting both the class and location (bounding box) of each object at once. This "you only look once" design allows the model to detect multiple objects in real time, even on devices with limited processing power, making it well suited for applications like robotics, security, and embedded AI systems.

A key innovation in YOLOv8 is its anchor-free detection mechanism and decoupled head architecture, which improve accuracy, adaptability, and robustness across diverse image conditions. The model can also be fine-tuned through transfer learning, allowing developers to retrain it on custom datasets for specific tasks. Additionally, YOLOv8 supports export to several deployment formats such as ONNX, TensorRT, and CoreML enabling smooth integration across hardware platforms including GPUs, CPUs, and FPGAs. Its combination of high inference speed, strong precision, and compatibility with edge devices makes YOLOv8 one of the most efficient and widely used models for real-time computer vision today.

2. AI Frameworks and Model Deployment

ONNX

ONNX (Open Neural Network Exchange) is an open-source format that allows machine learning models to be easily shared and used across different software frameworks and hardware platforms. It was developed by Microsoft and Facebook to solve the problem of compatibility between tools like PyTorch, TensorFlow, and Keras. Instead of retraining or rewriting a model for each environment, ONNX provides a universal model format that can be exported from one

framework and imported into another. This makes it much easier to deploy trained models on different devices from powerful GPUs to smaller embedded systems like FPGAs without losing performance or accuracy. ONNX also supports advanced features such as quantization (for faster, smaller models) and dynamic input handling, which make it ideal for real-time applications such as object detection, speech recognition, and image analysis.

ONNX Runtime

ONNX Runtime is the engine that runs ONNX models efficiently on various types of hardware. Developed by Microsoft, it serves as a high-performance inference framework optimized for speed, portability, and hardware acceleration. ONNX Runtime can automatically take advantage of the available hardware whether CPU, GPU, or FPGA to execute deep learning models as quickly as possible. It includes built-in optimizations such as graph pruning and operator fusion, which reduce computation time and improve throughput during inference. In embedded or edge applications, ONNX Runtime allows systems to perform real-time decision-making directly on the device, without needing a constant connection to a powerful computer or cloud service. This makes it a key component in deploying AI models for robotics, vision systems, and autonomous devices.

Ultralytics

Ultralytics is a leading AI development platform and open-source organization best known for creating the YOLO (You Only Look Once) family of real-time object detection models, including the latest and most advanced version, YOLOv8. The Ultralytics ecosystem provides an accessible and unified framework for building, training, testing, and deploying computer vision models with minimal setup. It offers both a Python API and command-line interface, allowing users to perform tasks such as data preparation, model training, evaluation, and export to deployment formats like ONNX or TensorRT. Designed with efficiency and user-friendliness in mind, Ultralytics tools include built-in support for transfer learning, dataset augmentation, and real-time visualization. This makes it an ideal platform for engineers, researchers, and students seeking to develop and deploy deep learning models quickly across various devices from high-performance servers to embedded edge hardware.

Vitis AI

Vitis AI is AMD-Xilinx's comprehensive development platform for accelerating AI inference on adaptable hardware platforms such as FPGAs and adaptive SoCs. It provides an end-to-end tool chain that supports model optimization, quantization, and deployment, allowing developers to efficiently run deep learning models on hardware-accelerated architectures like the Deep Learning Processing Unit (DPU). Vitis AI is compatible with popular frameworks such as TensorFlow, PyTorch, and ONNX, enabling seamless conversion and compilation of trained models into highly optimized, low-latency binaries. Its runtime environment and libraries are tailored for edge deployment, offering deterministic performance, low power consumption, and

the flexibility to integrate AI workloads alongside custom logic. With support for both Linux-based edge devices and data center cards, Vitis AI enables developers to bridge the gap between AI model development and real-world embedded system deployment, making it ideal for applications such as smart cameras, industrial robotics, and autonomous systems.

3. Hardware and Processing Architectures

CPU, GPU, and DPU

CPU, GPU, and DPU architecture each play distinct roles in modern computing, particularly in artificial intelligence and embedded vision applications. A Central Processing Unit (CPU) is a general-purpose processor designed for sequential task execution and system management, making it versatile for control operations, logic handling, and light computational tasks. In contrast, a Graphics Processing Unit (GPU) is optimized for massively parallel computations, enabling it to handle the large-scale matrix and tensor operations required for deep learning training and high-performance inference. The Deep Learning Processing Unit (DPU), commonly implemented on FPGA-based platforms, is a specialized hardware accelerator dedicated to executing neural network operations efficiently. It offers high performance per watt and deterministic latency, making it ideal for real-time AI workloads at the edge. Together, these architectures provide a spectrum of processing solutions CPUs for control, GPUs for high-throughput computation, and DPUs for energy-efficient, low-latency inference allowing developers to balance flexibility, performance, and power consumption based on the application's needs.

CPU	GPU	DPU
Several cores	Many cores	Dozens of cores
Low latency	High throughput	Higher degree of versatility
Ideal for serial processing	Ideal for parallel processing	Ideal for big data processing
Handles a handful of operations at once	Handles thousands of operations at once	Handles thousands of operations at scale

Figure IV: *CPU vs. GPU vs. DPU*

Kria KR260 Board:

The Kria KR260 Robotics Starter Kit is an advanced, adaptive edge computing platform developed by AMD-Xilinx, designed specifically for robotics, machine vision, and real-time control applications. Built around the Zynq UltraScale+ MPSoC, the KR260 combines the flexibility of reconfigurable hardware with the power of multi-core processing in a compact and efficient system. This architecture allows it to execute complex algorithms, manage real-time data streams, and interface directly with sensors and actuators all while maintaining low power consumption and deterministic performance. The board came be configured with an Ubuntu-

based operating system and supports both traditional software execution and hardware acceleration through FPGA logic, making it suitable for AI, control, and embedded system development.

At the heart of the KR260 lies the Zynq UltraScale+ MPSoC, a System-on-Chip that integrates multiple processing elements within a single device. It includes a quad-core ARM Cortex-A53 for general-purpose computing, a dual-core ARM Cortex-R5 for real-time control, and a Mali-400MP2 GPU for graphical and parallel processing tasks. Complementing these processors is the programmable FPGA fabric, which enables developers to implement custom digital logic, hardware accelerators, and high-speed data paths. This combination of fixed and programmable hardware allows tasks to be distributed efficiently. High-level logic and operating systems run on the ARM cores, while time-critical and computation-heavy operations can be offloaded to the FPGA. This hybrid design provides both flexibility and performance, allowing developers to tailor the system to the exact requirements of their application.

The KR260 also includes a variety of high-speed I/O interfaces such as Gigabit Ethernet, USB 3.0, DisplayPort, UART, and PMOD connectors that allow it to communicate with external devices, cameras, and sensors. Its support for GPIO expansion enables seamless integration with LEDs, motors, switches, and other hardware components, which makes it ideal for building responsive, interactive embedded systems. Furthermore, the board supports PYNQ and Jupyter Notebooks, allowing developers to interact with the hardware using Python for rapid testing and prototyping. This makes it highly accessible for both experienced engineers and students working on real-time embedded AI applications.

What makes the Kria KR260 particularly special is its ability to bridge the gap between AI and physical systems. It enables edge devices to perform tasks such as image recognition, decision-making, and actuation directly on the board without relying on cloud computing or external GPUs. Its reconfigurable FPGA logic ensures ultra-low latency, while the ARM-based processing system handles high-level control, communication, and coordination. This balance of hardware acceleration and software flexibility positions the KR260 as a powerful platform for next-generation robotics, smart vision, and automation applications that demand speed, reliability, and adaptability.

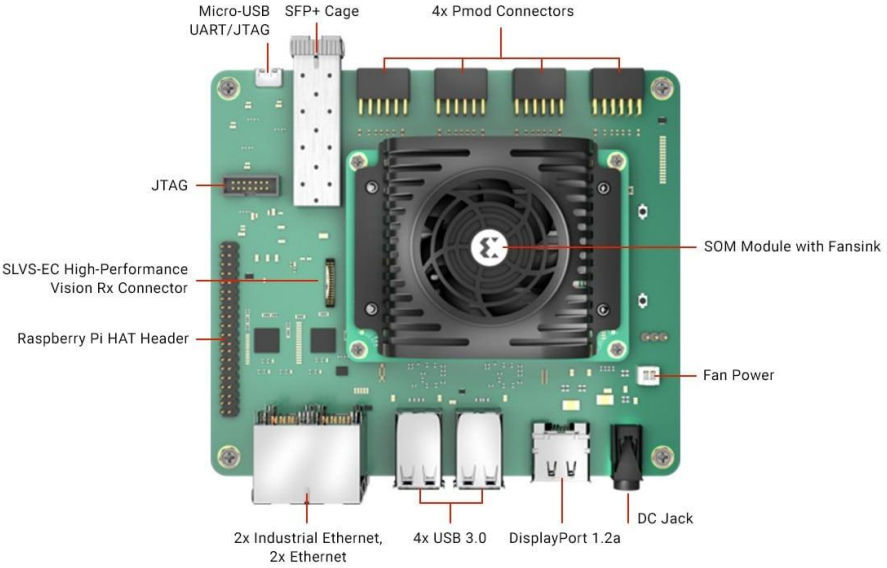

Figure V: *Kria KR260 Board*

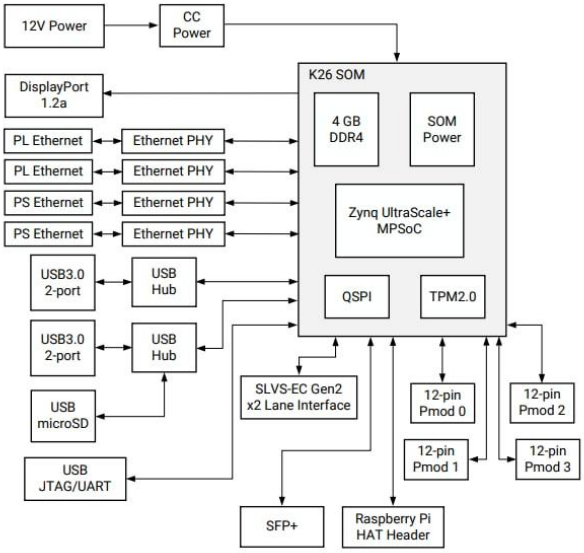

Figure VI: *KR260 Block Diagram*

FPGA

FPGA stands for Field-Programmable Gate Array, which is a type of integrated circuit that can be reprogrammed by the user after manufacturing. Unlike traditional processors (like CPUs or GPUs) that follow a fixed architecture and execute instructions sequentially, FPGAs consist of thousands of configurable logic blocks that can be wired together in different ways to create custom digital circuits. This flexibility allows developers to design hardware that is tailored exactly to the task at hand, enabling high-speed and low-latency performance for specific applications such as signal processing, real-time control, and embedded vision. Because the logic is implemented directly in hardware, FPGAs can perform many operations in parallel, making them especially useful in systems that need fast and deterministic response times. In AI

and computer vision projects, FPGAs can accelerate tasks like object detection by running parts of the model on hardware for greater efficiency, all while consuming less power than general-purpose processors.

4. Embedded System Tools and Software Environment

Linux and Ubuntu Operating System:

Linux is an open-source operating system based on Unix principles, widely recognized for its reliability, flexibility, and efficiency across computing platforms from servers to embedded systems. It provides a stable environment for multitasking, hardware control, and network management, making it ideal for real-time and high-performance applications. Ubuntu, a popular Linux distribution, builds on this foundation by offering a user-friendly interface, extensive driver support, and preconfigured tools that simplify system setup and software deployment. Together, Linux and Ubuntu enable developers to run applications, manage resources, and interface directly with hardware components in a secure and customizable environment. On embedded platforms such as the Kria KR260, Ubuntu Linux serves as the operating system that supports tools like PYNQ, Jupyter Notebooks, and ONNX Runtime, allowing for seamless integration of software development, AI inference, and hardware control within a single unified system.

Vivado Design Suite:

Vivado Design Suite is a comprehensive development environment created by AMD-Xilinx for designing, simulating, and implementing digital systems on FPGA and SoC devices. It provides an integrated platform that supports every stage of hardware design from creating block diagrams and writing HDL code (in VHDL or Verilog) to synthesis, implementation, and bitstream generation. Vivado's IP Integrator allows developers to visually connect pre-built hardware blocks, such as processors, memory interfaces, and GPIO modules, simplifying complex system development. In addition, Vivado includes advanced tools for timing analysis, hardware debugging, and constraint management, ensuring reliable and optimized designs. The suite also supports exporting hardware definitions for use with higher-level environments like PYNQ or Vitis, bridging the gap between low-level hardware design and high-level software control. By combining graphical design tools with professional-grade analysis and automation features, Vivado enables engineers to efficiently create customized, high-performance FPGA-based systems such as those used in the Kria KR260 platform.

Bitstream

Bitstream refers to the compiled hardware configuration file that defines how an FPGA (Field-Programmable Gate Array) is programmed and behaves. When a hardware design is created in a tool such as Vivado, the system architecture composed of logic gates, interconnections, and IP blocks is first synthesized, implemented, and then compiled into a binary file known as a bitstream (usually with the extension .bit or .bin). This file contains the configuration data that

specifies how each programmable element inside the FPGA should be connected and function. When the FPGA board, such as the Kria KR260, loads this bitstream, it effectively reconfigures its internal circuitry to perform the desired logic or control tasks. In essence, the bitstream serves as the "blueprint" that transforms the FPGA from a blank, reconfigurable device into a customized hardware system enabling features such as GPIO control, signal routing, or AI acceleration depending on the design.

PYNQ (Python for Zynq)

PYNQ (Python Productivity for Zynq) is an open-source framework developed by AMD-Xilinx that simplifies the use of FPGAs by allowing developers to program and control hardware using Python instead of traditional hardware description languages like VHDL or Verilog. Built to work with Zynq and Kria devices, PYNQ bridges the gap between software and hardware development, enabling users to create and interact with hardware-accelerated applications through familiar Python libraries and Jupyter Notebooks. It provides a high-level interface for tasks such as managing programmable logic overlays, configuring GPIOs, and communicating with peripherals, significantly reducing the complexity of FPGA-based development. By abstracting low-level hardware details, PYNQ allows engineers and researchers to focus on application logic, data processing, and AI integration rather than HDL design. This makes it an ideal platform for embedded vision, machine learning, and real-time control projects that combine the flexibility of Python with the power and parallelism of FPGA hardware.

Python

Python is a high-level, versatile programming language widely used in artificial intelligence, data science, and embedded system development due to its readability, extensive library support, and cross-platform compatibility. In embedded AI and computer vision projects, Python serves as the bridge between software logic and hardware control, enabling developers to rapidly prototype, test, and deploy applications. Its rich ecosystem of libraries such as OpenCV for image processing, NumPy for numerical computation, and ONNX Runtime for model inference makes it an ideal choice for real-time vision and control tasks. When combined with frameworks like PYNQ and Jupyter Notebooks, Python allows direct interaction with FPGA hardware through high-level commands, simplifying tasks such as GPIO manipulation, data acquisition, and AI inference. This accessibility and flexibility make Python an essential tool for developing intelligent, hardware-integrated systems that balance performance with ease of development.

Jupyter Notebook

Jupyter Notebooks are an interactive, web-based development environment that allows users to combine code, text, images, and results in a single, easy-to-use interface. They are widely used in data science, machine learning, and embedded system development because they enable code execution in small, manageable "cells," making it easy to test, debug, and visualize results step

by step. When used with platforms such as the Kria KR260, Jupyter connects to the board through a local or network interface, allowing Python code to be executed directly on the device while outputs, logs, and visual results are displayed on a host computer. This design makes it especially effective for iterative development engineers can quickly test model performance, adjust parameters, visualize camera input, or send commands to GPIO components without having to rebuild or restart the system.

What makes Jupyter particularly special is its interactive and modular design, which bridges the gap between software experimentation and hardware implementation. Unlike traditional development environments, Jupyter allows users to immediately observe the effects of code changes, making it ideal for workflows that involve testing AI inference, real-time video processing, or hardware control. It supports multiple programming languages through "kernels" (most commonly Python) and integrates easily with scientific and hardware libraries like OpenCV, NumPy, and PYNQ. Although certain low-level operations such as driver installation or kernel module updates must still be performed via the board's terminal, Jupyter remains invaluable for developing, documenting, and verifying embedded AI systems. In projects involving vision models, GPIO control, and real-time feedback, Jupyter Notebooks provide a clear, organized, and highly efficient platform for experimentation, analysis, and system validation.

Additional Tools and Libraries:

Supplementary Tools and Libraries play a vital role in the development of advanced embedded systems, providing essential functionality for data processing, control, and performance optimization. TensorFlow Lite enables lightweight and efficient neural network inference, making it suitable for running AI models on resource-constrained hardware that requires real-time analysis. OpenCV offers a comprehensive set of computer vision functions for image preprocessing, feature extraction, and video analysis, while NumPy supports fast and efficient numerical computations for handling large datasets and mathematical operations. RPi.GPIO simplifies the management of GPIO pins, allowing precise control of external components such as LEDs, sensors, and actuators. Additionally, Python's built-in libraries such as time, used for execution timing, and collections.deque, used for efficient data buffering and smoothing, contribute to the stable and responsive operation of real-time embedded applications. Together, these tools form a robust software ecosystem that enhances the flexibility, performance, and reliability of intelligent hardware systems.

5. System Integration and Real-Time Implementation

Embedded vision systems

Embedded vision systems are intelligent, self-contained devices that integrate image capture, processing, and decision-making within a compact hardware platform. Unlike traditional

computer vision setups that rely on powerful external computers, embedded vision systems perform all analysis locally on resource-constrained devices such as FPGAs, SoCs, or microprocessors. These systems typically combine a camera module, processing unit, and output interface to interpret visual information in real time and respond through physical actions or data communication. Advances in deep learning and hardware acceleration have made it possible for embedded systems to handle complex vision tasks such as object detection, tracking, and classification directly on the device, without needing a network connection or cloud processing. This local processing enables faster response times, lower power consumption, and improved data privacy. Embedded vision systems are widely used in robotics, industrial automation, smart cameras, and edge AI applications, where reliability, speed, and autonomy are essential for real-time operation.

Live video processing

Live video processing and inference pipelines form the core of real-time computer vision systems, enabling devices to capture, analyze, and respond to visual data as it happens. In such a pipeline, a camera continuously streams video frames to the processing unit where each frame is preprocessed, analyzed by an object detection model such as YOLOv8, and then used to trigger decisions or hardware responses. This process involves multiple coordinated steps: image acquisition, resizing and normalization, model inference, and post-processing to identify object locations and confidence levels. The results can then be used to activate physical outputs, such as LEDs or alarms, through GPIO control. On embedded platforms like the Kria KR260, this pipeline is optimized for low latency and high reliability, allowing visual recognition and response to occur almost instantaneously. Live video processing pipelines are essential in applications such as surveillance, robotics, and automation where continuous, real-time perception enables systems to interact intelligently with their environment.

Real-time GPIO Control

Real-time GPIO (General-Purpose Input/Output) control allows an embedded system to interact directly and immediately with external hardware components such as LEDs, buzzers, switches, and sensors. GPIO pins can be configured as either inputs or outputs, enabling the system to both read signals from the environment and send control commands in response. In real-time applications, this interaction must occur with minimal delays, ensuring that physical outputs react almost instantly to software or sensor events, for example, turning on an LED when an object is detected. On platforms like the Kria KR260, GPIO control is tightly integrated with both the processor and the programmable logic, providing deterministic timing and low-latency response. This capability is essential in embedded AI systems, where decisions made by models such as YOLOv8 must trigger immediate, reliable actions in the physical world.

Bill of Materials (BOM)

- 1 × AMD-Xilinx Kria KR260 Robotics Starter Kit (Zynq UltraScale+ MPSoC)
- 1 × micro-SD card (32 GB or greater) with Ubuntu 22.04.4 LTS image for KR260.
- 1 × Host laptop
- 1 × USB webcam
- Breadboard
- LEDs: 1 × green LED, 1 × red LED
- 3 × 220 Ω resistors
- 1 × active buzzer
- Jumper wires
- Power supply and peripherals: power adapter, HDMI/DisplayPort cable (if display used), keyboard/mouse, Ethernet cable (optional).

Flow Chart

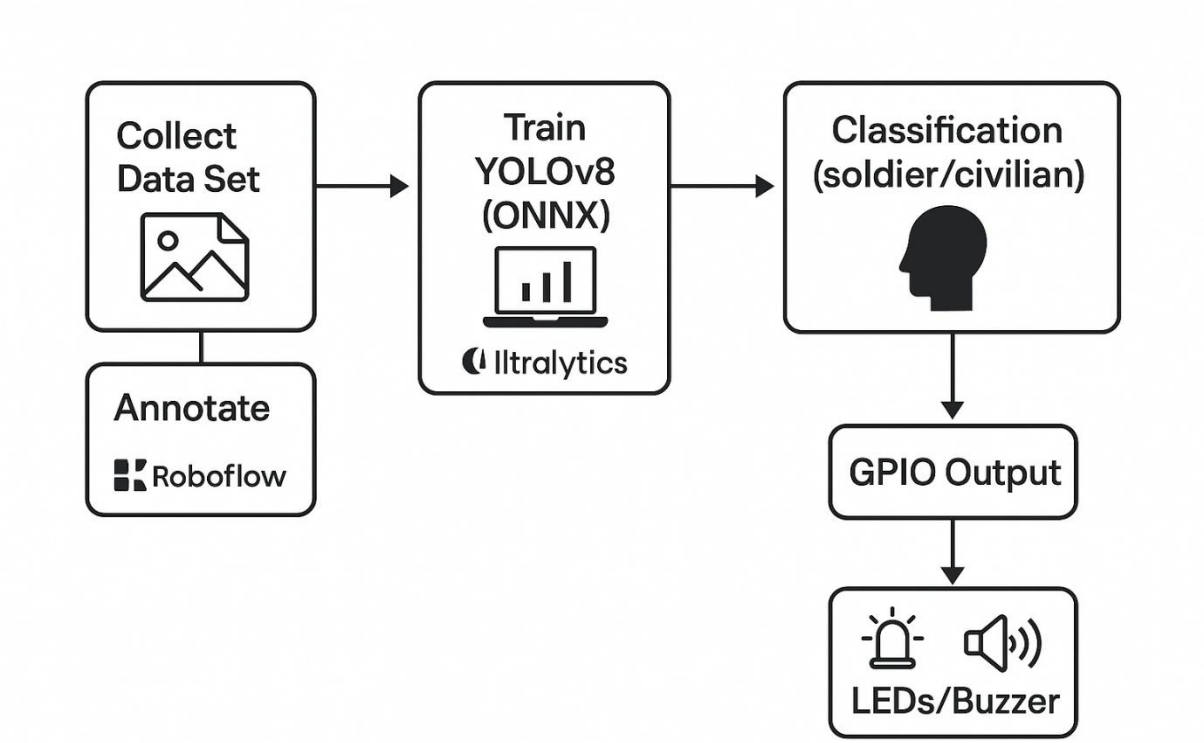

Figure VII: *Project Flow Chart*

Project Steps

1. Project Preparation

Kria KR260 Board Setup

The initial preparation phase involved setting up the Kria KR260 board, a crucial step in ensuring the proper operation and functionality of the project. The first step was installing the compatible Linux operating system provided specifically by AMD/Xilinx for the KR260 board. The Ubuntu 22.04.4 LTS-based Linux image was downloaded from the official AMD/Xilinx website and flashed onto an SD card using Balena Etcher software.

Once the image was flashed, the SD card was inserted into the Kria KR260, and the board was connected to necessary peripherals such as a keyboard, mouse, monitor, and power supply. Upon powering up the board, the Linux operating system successfully booted, and initial system updates were executed using standard Linux commands:

```
sudo apt-get update

sudo apt-get dist-upgrade
```

After verifying system stability, the next step involved configuring the required software tools and libraries, specifically the PYNQ framework and Jupyter Notebook. PYNQ, an open-source framework that allows Python-based programming of the FPGA hardware, was installed following the official Xilinx documentation. This installation provided a simplified interface for interacting with the FPGA hardware, enabling rapid development and prototyping.

To facilitate development and testing, Jupyter Notebook was configured to run on the board. The web-based interactive environment was accessed through a web browser by navigating to the board's local IP address on port 9090. This setup enabled efficient coding, real-time testing, and visualization of results in an intuitive and productive manner.

Initial Hardware Test: LED Blinking

To verify the successful integration and functioning of the hardware and software environment, an initial test was conducted by configuring a basic GPIO operation: blinking an LED. Utilizing PYNQ's GPIO libraries[2], a simple Python script was executed within the Jupyter Notebook environment to control an onboard or externally connected LED.

The Python test code was structured to toggle the LED state at defined intervals, typically one second intervals, using the following simplified code example:

[2] The initial Blink Led Test was done using a preloaded GPIO from a GitHub library which can be found at: https://community.element14.com/technologies/fpga-group/b/blog/posts/blinking-a-led-with-pynq-in-kria-kv260-kr260. The specific Bitstream for our project was built manually using Vivado, as will be discussed below.

Code Block 1.1:

```
# Import necessary libraries from PYNQ and Python
# Overlay is used to load the FPGA bitstream (hardware configuration)
# AxiGPIO allows communication with the GPIO IP block in the FPGA design
# time provides delay functions for blinking timing
from pynq import Overlay
from pynq.lib import AxiGPIO
import time

# Step 1: Load the bitstream into the programmable logic (PL)
# The .bit file configures the FPGA hardware with the design
# that defines how the GPIO pins and LEDs are connected.
overlay = Overlay('kr260_led_blink.bit')

# Step 2: Access the AXI GPIO block from the overlay
# overlay.ip_dict contains a dictionary of all hardware IP blocks
# by their names as defined in the Vivado design.
# 'axi_gpio_0' is the label for the GPIO block used to control LEDs.
# Each GPIO has two channels; we select channel 1 here (output side).
gpio = AxiGPIO(overlay.ip_dict['axi_gpio_0']).channel1

# Step 3: Blink the LED continuously
# The loop alternates between turning the LED ON and OFF.
# gpio.write(<channel>, <value>) sends data to the GPIO output pins.
# 0xffffffff means all bits high (LED ON)
# 0x00000000 means all bits low (LED OFF)
# time.sleep(1) adds a 1-second delay between each toggle.
while True:
    gpio.write(1, 0xffffffff)  # Turn LED(s) ON
    time.sleep(1)              # Wait for 1 second
    gpio.write(0, 0xffffffff)  # Turn LED(s) OFF
    time.sleep(1)              # Wait for 1 second
```

Executing this script validated the proper functioning of the FPGA hardware, GPIO configurations, and software tools. The successful blinking of the LED confirmed the correct initial setup, marking a critical milestone in the overall project preparation.

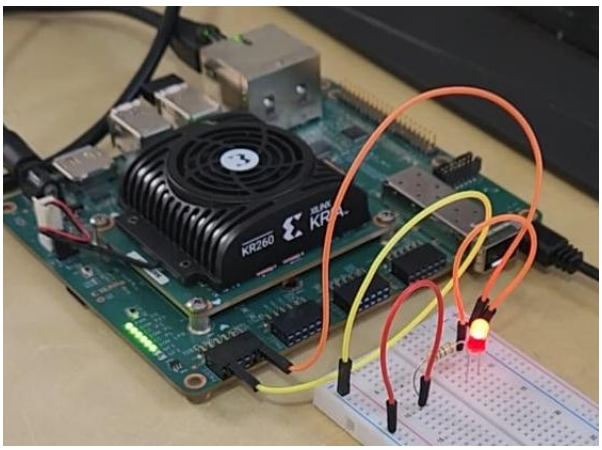

Figure 1.1: *Blinking LED with KR260*

2. Training the Dataset

2.1. Dataset Collection and Preparation Using Roboflow

To train the object detection model, a structured and labeled dataset was required. Images representing the target objects such as phones (for testing) and later soldiers, and civilians, were gathered from online sources. These images were then uploaded to Roboflow, an online platform for managing and preprocessing computer vision datasets.

After uploading, Roboflow automatically organized the images into a project workspace. Each image was then annotated, meaning a rectangular bounding box was drawn around each object and assigned a class label (for example, "soldier" or "civilian"). These annotations were saved as coordinate files that defined the object's location and category. The annotation data served as the "ground truth" used by YOLO during training. Without these annotations, the model would have been unable to learn how to distinguish between different object classes.

2.2. Dataset Augmentation and Export

Roboflow also offered data augmentation tools, which were used to artificially expand the dataset and improve its robustness. Augmentation applied transformations such as rotations, flips, brightness changes, and random cropping. This process helped the model generalize better by learning to recognize objects under varying lighting conditions, orientations, and backgrounds.

Once the dataset had been fully annotated and augmented, it was exported in YOLOv8 format. The exported package included the images, annotation files, and a data.yaml configuration file describing the dataset structure and class labels. Although Roboflow could have been used to train the model directly in the cloud, the training process was instead performed locally to maintain control over the environment and resources.

Figure 2.2: *Annotated Pictures – Civilians in green, Soldiers in purple*

While Roboflow was not strictly necessary since manual annotation using tools like LabelImg was possible it significantly streamlined dataset organization and labeling, reducing human error and saving time.

2.3. Training the YOLOv8 Model

Training was conducted locally on a laptop using a dedicated Python virtual environment named yolo311env. This environment contained all required dependencies, including Ultralytics YOLOv8, OpenCV, and PyTorch. The model was trained using the following command:

```
yolo detect train data=data.yaml model=yolov8n.pt imgsz=320 epochs=75
```

Each parameter in the command had a specific function:

- Data = data.yaml specified the dataset path and class labels.
- model=yolov8n.pt defined the pretrained YOLOv8 "nano" model as the starting point for transfer learning.
- Imgsz = 320 resized all input images to 320×320 pixels, balancing training speed with sufficient resolution for accurate detection.
- epochs=75 indicated that the model would complete 75 full passes over the training dataset. Increasing the number of epochs allows the model to learn more thoroughly but also increases training time and the risk of overfitting.

The "nano" model was selected due to its small size and efficiency, making it suitable for deployment on embedded hardware such as the Kria KR260.

2.4. Training Process

During training, the YOLO algorithm learned to detect and classify objects through a sequence of forward and backward passes. In each iteration, an image was passed through the neural network, which generated predicted bounding boxes and class probabilities. The model then compared its predictions to the ground-truth annotations and calculated a loss function, representing the prediction error.

Using a process known as backpropagation, YOLO adjusted its internal network weights to minimize this error in subsequent iterations. This cycle repeated thousands of times across all images and epochs, gradually improving detection accuracy.

Training required significant time and computational resources because each image was processed through millions of calculations within the neural network. The total duration depended on the number of images, image resolution, number of epochs, and the performance

of the available CPU or GPU. For the purposes of this project, there were 350 images used in the training process.

2.5. Results and Evaluation

Upon completion of training, YOLOv8 automatically generated output files within a directory named runs/detect/train. This directory contained:

- **best.pt** – the trained model with the highest validation accuracy.
- **results.png** – graphical plots showing the loss curves and accuracy metrics over epochs.
- **Evaluation metrics** such as *precision*, *recall*, and *mAP (mean Average Precision)*, which quantified how accurately the model identified and classified objects.

The YOLOv8 object detection model was trained for 75 epochs using an input image size of 320×320 pixels. Throughout training, the model exhibited stable convergence and consistent improvement across both training and validation datasets. The learning process successfully reduced all loss components bounding box regression, classification, and distribution focal loss while key accuracy metrics such as precision, recall, and mean Average Precision (mAP) steadily increased.

By the final epoch, the model achieved 0.95 precision, 0.89 recall, and mAP@0.5 = 0.95, with a mean performance across multiple thresholds (mAP@0.5–0.95 = 0.74). These results demonstrated that the model effectively learned to detect and differentiate between *civilian* and *soldier* classes with high reliability and minimal misclassifications. Overall, the model showed strong generalization capabilities, confirming that the dataset, preprocessing, and training parameters were appropriate for the detection task.

The trained model was later converted into ONNX format (best.onnx) for compatibility with the Kria KR260 hardware, where it was used for real-time inference and LED-based feedback.

2.5.1 Training and Validation Curves

The YOLOv8 training process generated a comprehensive results summary (results.png) that tracked losses and accuracy metrics over all epochs. The upper row of plots represented training losses for bounding box, classification, and distribution focal components, while the lower row represented their validation counterparts. The rightmost graphs displayed precision, recall, and mAP metrics for both training and validation sets.

All loss values decreased steadily over time, indicating continuous improvement in model performance. The corresponding rise in precision and recall curves confirmed that the model became increasingly accurate and confident in its detections as training progressed. Importantly,

validation losses followed a similar trend to training losses, suggesting that the model did not overfit and maintained generalization to unseen images.

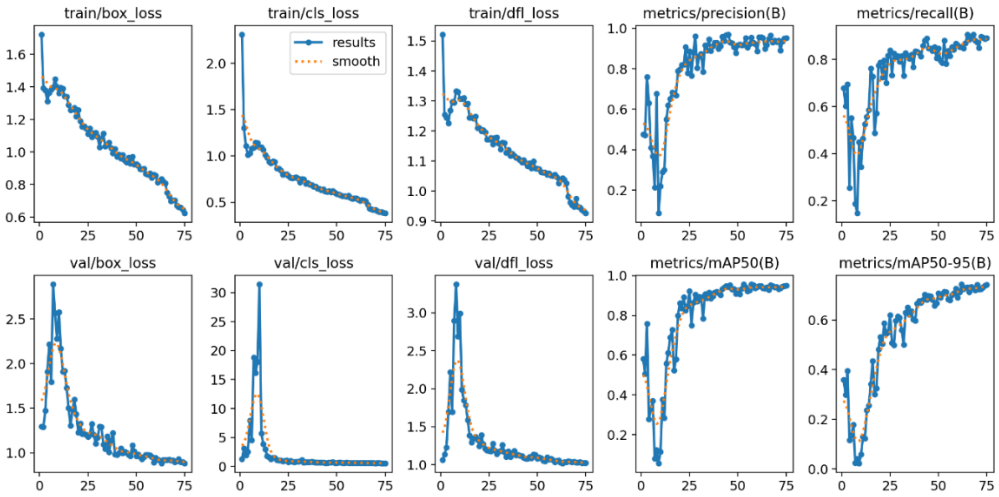

Figure 2.5.1: *"Training and Validation Performance Metrics"* (Results.png)

The plots show steadily decreasing losses and rising precision/recall curves, confirming stable learning and effective convergence over 75 epochs.

2.5.2 Final Quantitative Metrics

At the end of 75 epochs, the model's final quantitative performance was as follows:

Table 2.5.2 Quantitative Metrics

Metric	Value
Precision (B)	0.9519
Recall (B)	0.8915
mAP@0.5 (B)	0.9489
mAP@0.5–0.95 (B)	0.7431
Validation Box Loss	0.8844
Validation Classification Loss	0.5557
Validation DFL Loss	1.0214

These metrics indicate that the model achieved a high level of detection reliability. The precision of 0.95 reflects that nearly all detections were correct, while a recall of 0.89 shows that most true objects in the images were successfully detected. This confirms excellent detection accuracy when the model's bounding boxes are allowed a moderate margin of overlap, while the mAP@0.5–0.95 score shows it remains consistently accurate even under stricter evaluation thresholds. The validation box loss indicates that the model's predicted bounding boxes align closely with the true object locations, and the classification loss suggests it is generally confident and correct when identifying object types. Finally, the DFL loss reflects reasonable precision in refining box boundaries. Altogether, these metrics mean the model is both accurate and dependable in detecting and classifying objects, though it could be slightly improved in fine-tuning box edge precision.

2.5.3 Confusion Matrix

The confusion matrix provided a visual summary of the classification results for each category. The diagonal elements represented correct predictions, while off-diagonal elements represented misclassifications. In both the raw and normalized matrices, the majority of detections were concentrated along the diagonal, indicating that most objects were correctly classified. Only a few "civilian" instances were misclassified as "soldier," and vice versa, showing minimal confusion between the two classes.

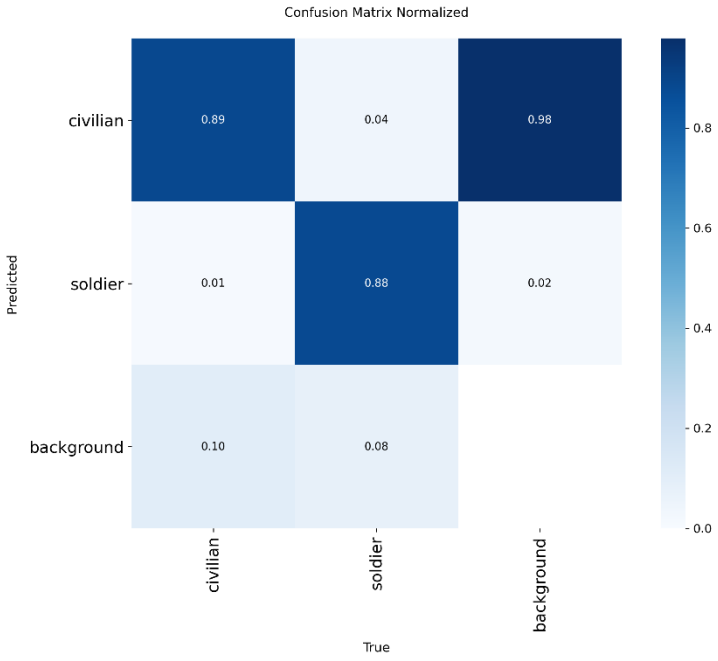

Figure 2.5.3: *- Normalized Confusion Matrix*

The diagonal dominance demonstrates strong class discrimination, with only minor errors between the 'civilian' and 'soldier' categories.

5.4 Precision–Recall Curve

The precision–recall (PR) curve illustrated the trade-off between the model's precision and recall over varying confidence thresholds. A large area under the PR curve (AUC) indicated strong detection quality. The average mAP@0.5 value of 0.948 confirmed excellent object localization and class differentiation. Class-specific curves revealed that the *soldier* class achieved slightly higher accuracy (0.983) than the *civilian* class (0.912), likely due to more distinct visual patterns and lower intra-class variation.

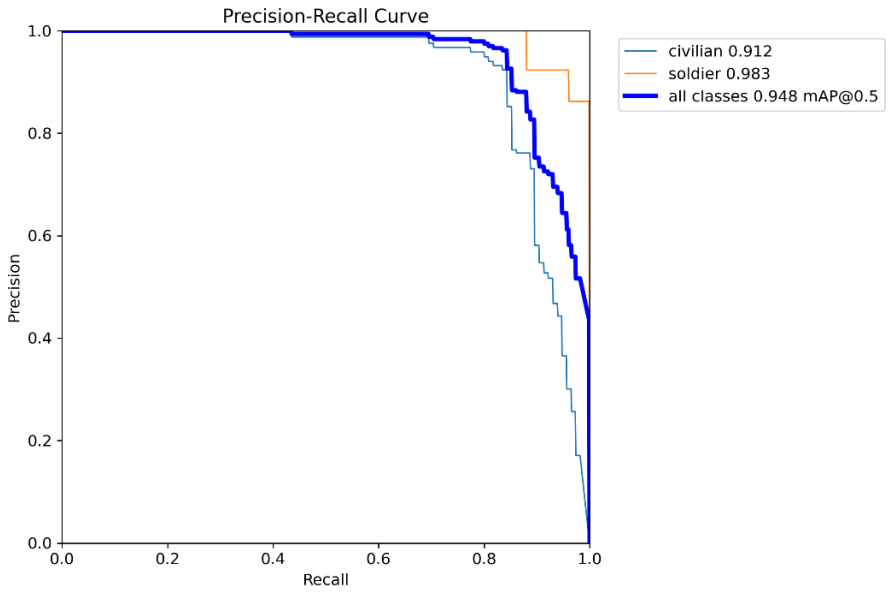

Figure 2.5.4: *Precision-Recall Curve*

The high and smooth PR curves show that both classes maintained strong detection accuracy, with only minor trade-offs between precision and recall.

2.5.5 F1–Confidence Curve

The F1–confidence curve demonstrated how the model's overall F1-score changed with different confidence thresholds. The F1-score combines precision and recall into a single metric, and its peak indicates the optimal balance point between the two. The model achieved its highest F1 value of approximately **0.90** at a confidence threshold of **0.60**. This threshold was later used for deployment, ensuring the best trade-off between detecting true positives and avoiding false alarms.

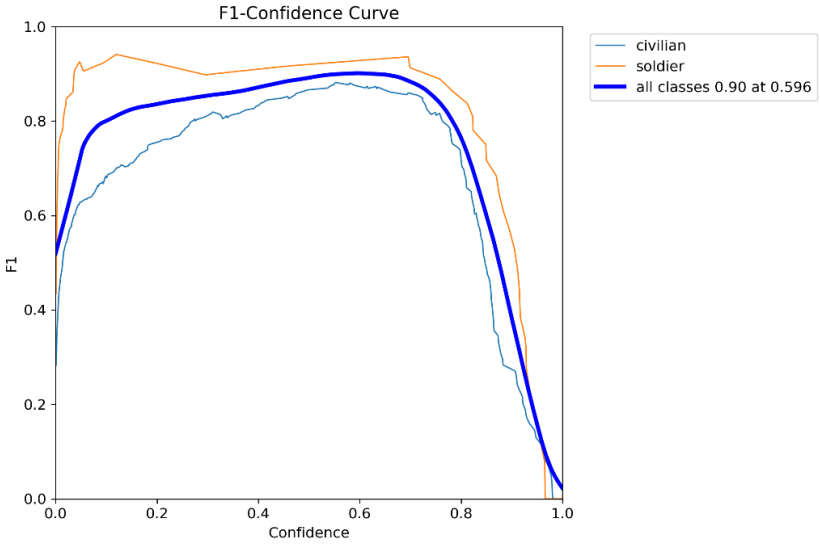

Figure 2.5.5: *F1-Confidence Curve*

The F1 curve peaks around a confidence level of 0.6, suggesting this as the optimal threshold for inference and LED activation on the Kria board.

2.5.6 Dataset Visualization

A dataset distribution analysis (labels.jpg) was also produced to evaluate the quality and balance of the training data. The upper bar chart showed that the dataset contained more *civilian* than *soldier* instances, while the spatial and dimensional plots below displayed the distribution of bounding box coordinates (x, y, width, height). The results confirmed that most annotations were centered and of moderate size, which helped the model converge efficiently, though additional "soldier" data would further balance the dataset.

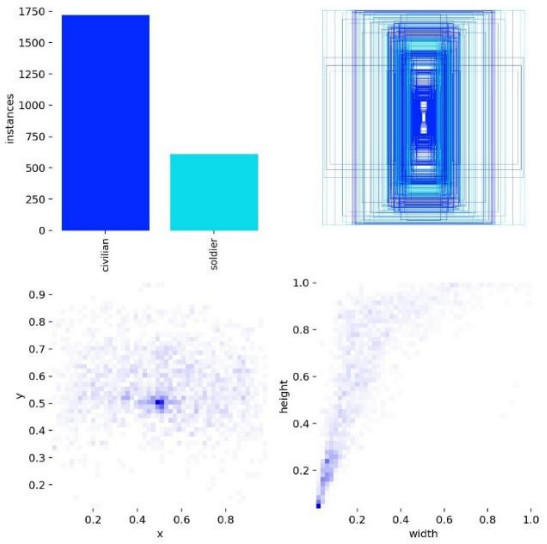

Figure 2.5.6: *Dataset Label Distribution*

The plots confirm that the dataset was well-annotated and consistent, though class balancing could further enhance future model performance.

2.5.7 Summary of Evaluation

In conclusion, the YOLOv8 model demonstrated strong detection capabilities after 75 epochs of training. The consistently decreasing losses, high precision and recall values, and smooth precision–recall and F1-confidence curves collectively verified that the model achieved excellent convergence and generalization. Minimal class confusion and high mAP values confirmed that the detector was well-suited for deployment in real-time classification tasks, including live inference on the Kria KR260 board.

Future improvements may include balancing the dataset between classes, increasing image diversity (different lighting, angles, and scales), and experimenting with slightly larger image sizes to improve fine-detail detection.

2.6. Overall Understanding

In summary, our YOLOv8 training process involved collecting and labeling data, organizing it through Roboflow, exporting the dataset in YOLO format, and training the model locally on our laptop. The combination of a properly annotated dataset, reasonable image size, and sufficient epochs allowed the model to learn to detect the target objects accurately. YOLO's training pipeline essentially teaches the model to "see" by gradually reducing the difference between its predictions and the true labeled data, resulting in a trained model ready for real-time detection on embedded hardware like the Kria KR260.

3. ONNX Export

3.1 Why convert to ONNX?

ONNX (Open Neural Network Exchange) is a portable file format for neural networks. Converting the trained YOLOv8 model (best.pt) to ONNX makes it:

- **Portable:** the same model runs on many runtimes (ONNX Runtime, OpenVINO, TensorRT, etc.) and on different OS/CPUs.
- **Deployable on embedded/edge devices:** when PyTorch isn't available or is heavy, ONNX Runtime is lightweight and easy to ship.
- **Optimizable:** many runtimes fuse layers and use fast kernels for CPU/ARM often giving equal or better speed than raw PyTorch on CPU.
  For this project, ONNX let us run inference on the **Kria KR260** without depending on a full PyTorch stack, and it also served as a steppingstone toward hardware-specific models (e.g., TensorRT or Xilinx .xmodel via separate flows).

3.2 Converting to ONNX

Using Ultralytics, the export is one line:

```
yolo export model=best.pt format=onnx imgsz=320 opset=12 simplify=True dynamic=False
```

- **imgsz=320**: fixes the model's input size to match training/deployment.
- **opset=12** (or newer): ensures operators are supported by modern runtimes.
- **simplify=True**: removes redundant nodes to speed up inference.
- **dynamic=False**: fixes shape for best performance (dynamic shapes are more flexible but can be slower).

The trained YOLOv8 model was exported to ONNX to enable portable, lightweight deployment on the KR260 platform. The conversion rewrote the network from PyTorch to ONNX operators while preserving the learned weights in FP32, leaving the training dataset unchanged. As a result, accuracy remained effectively the same as in the PyTorch evaluation, aside from negligible numerical differences. ONNX Runtime provided a compact, efficient inference path on CPU/ARM, improving deployability and often matching or exceeding PyTorch CPU speed. Any measurable accuracy shifts primarily occur when post-export optimizations (e.g., FP16/INT8 quantization or altered NMS) are introduced, in which case re-evaluation is recommended.

4. Hardware Implementation and Bitstream Design

4.1 Overview

To enable physical interaction between the trained YOLO model and external hardware components, a custom hardware design was implemented on the Kria KR260 platform using Xilinx Vivado Design Suite. This design allowed the Zynq UltraScale+ MPSoC processing system to communicate with the programmable logic (FPGA fabric) through the AXI interface, enabling real-time control of LEDs, buzzers, and other peripherals directly from software running on the embedded ARM processor. The resulting configuration was exported as a bitstream (.bit) and hardware hand-off (.hwh) file for use with PYNQ and Jupyter Notebooks.

4.2 Block Design Architecture

The block design defined the hardware structure used on the KR260. It consisted of the following major components:

Zynq UltraScale+ MPSoC: The central processing element integrating quad-core Cortex-A53 CPUs, dual Cortex-R5 controllers, and programmable logic. It managed high-level software execution while connecting to the FPGA fabric through the M_AXI_HPM0_FPD and M_AXI_HPM1_FPD master interfaces.

AXI Interconnect: Served as the communication backbone between the processing system and other IP blocks. It handled address, data, and control routing, ensuring synchronized communication across all connected devices.

AXI GPIO: Provided the digital interface between the FPGA and external hardware. Configured as an 8-bit output channel, it allowed the software to set or clear logic levels on GPIO pins, effectively toggling LEDs or driving other output devices. The GPIO signals were connected to the board's PMOD1 header for external access.

Processor System Reset: Guaranteed that all components started in a synchronized and stable state. It distributed proper reset signals from the Zynq PS to all connected IPs during initialization.

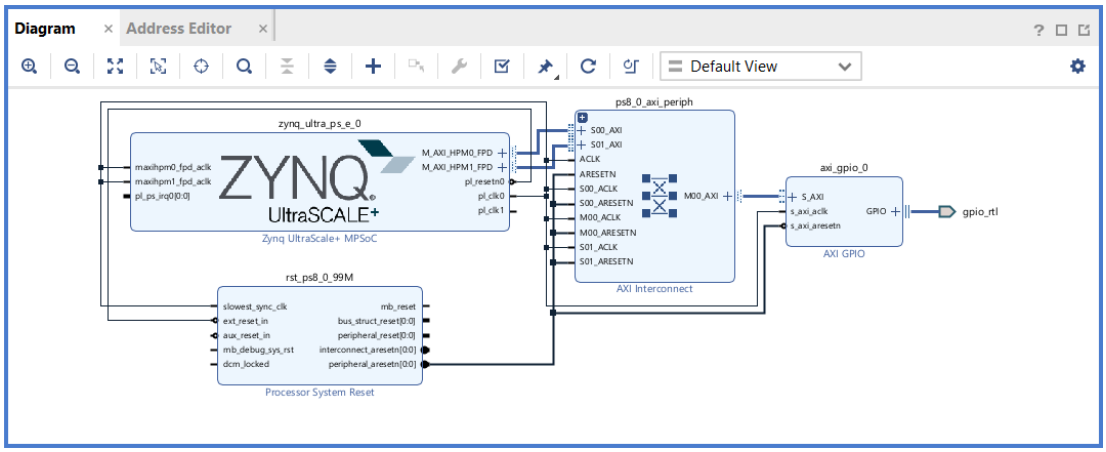

Figure 4.2 *Block Design in Vivado*

The diagram illustrates how the Zynq MPSoC communicates with the AXI GPIO via the AXI Interconnect, enabling software-controlled hardware operation[3].

4.3 PMOD1 Header Configuration

The PMOD1 header was used to connect FPGA outputs to LEDs on a breadboard. In the final implementation, the logical PMOD IO numbering matched the physical connector pin order each PMOD IO1–IO8 corresponds directly to physical pins 1–8 on the connector.

Table 4.3 *PMOD1 Header Configuration*

Pin No.	Signal Name	FPGA Pin	Description
1	PMOD1_IO1_HDA11	H12	GPIO[0]
2	PMOD1_IO5_HDA15	E10	GPIO[1]
3	PMOD1_IO2_HDA12	E12	GPIO[2]
4	PMOD1_IO6_HDA16_CC	D12	GPIO[3]
5	PMOD1_IO3_HDA13	C11	GPIO[4]
6	PMOD1_IO7_HDA17	B11	GPIO[5]
7	PMOD1_IO4_HDA14_CC	C10	GPIO[6]
8	PMOD1_IO8_HDA18	C12	GPIO[7]
9		Ground	GND
10		Ground	GND
11		3.3 V	Power
12		3.3 V	Power

[3] *Note that the output of the gpio 'gpio_rtl_1' is PMOD1 on the physical board.*

This configuration allows each GPIO bit to control a separate output line, such as an LED or buzzer input.

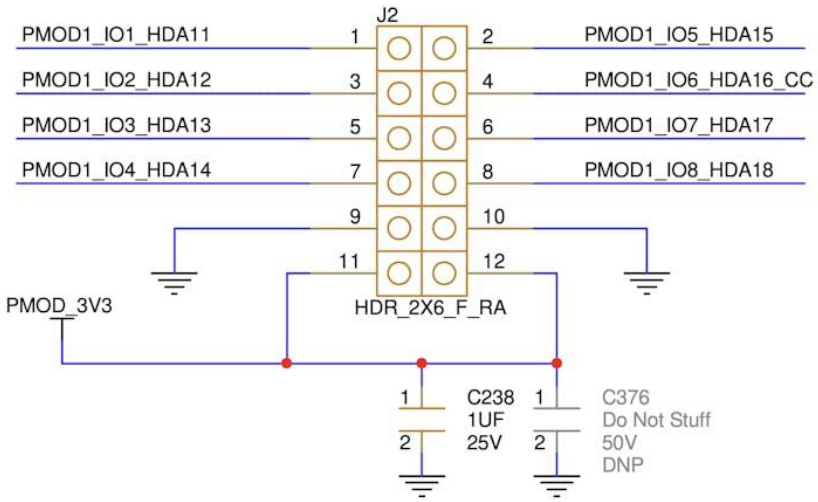

Figure 4.3: *Layout diagram showing pin numbering and logical-to-physical mapping.*

In the figure above pin 1 is the top right, while pin 12 is the bottom left of PMOD1.

4.4 Bitstream Creation Procedure

1. **Create Project**
    - Launched Vivado → *Create Project* → *RTL Project*.
    - Selected the **Kria KR260 board** under *Boards* and completed project setup.
2. **Block Design Construction**
    - Opened *IP Integrator* → *Create Block Design*.
    - Added the **Zynq UltraScale+ MPSoC** block and ran *Block Automation* (enabling PS configuration and AXI interfaces).
    - Added the **AXI GPIO** IP (Channel 1 width = 8 bits, direction = Output).
    - Executed *Connection Automation* to link all components and generated the AXI Interconnect automatically.
    - Regenerated layout for clarity.
3. **Constraint File Creation (.xdc)**
    - Created a new *Constraints File* defining physical pin assignments for the GPIO outputs.

The PMOD1 pin mappings were defined as follows:

Code Block 4-1: *GPIO Constraints (.XDC)*

```
set_property PACKAGE_PIN H12 [get_ports {gpio_rtl_tri_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[0]}]

set_property PACKAGE_PIN B10 [get_ports {gpio_rtl_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[1]}]

set_property PACKAGE_PIN E10 [get_ports {gpio_rtl_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[2]}]

set_property PACKAGE_PIN E12 [get_ports {gpio_rtl_tri_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[3]}]

set_property PACKAGE_PIN D10 [get_ports {gpio_rtl_tri_o[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[4]}]

set_property PACKAGE_PIN D11 [get_ports {gpio_rtl_tri_o[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[5]}]

set_property PACKAGE_PIN C11 [get_ports {gpio_rtl_tri_o[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[6]}]

set_property PACKAGE_PIN B11 [get_ports {gpio_rtl_tri_o[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_o[7]}]
```

4. **Generate Bitstream**

- Created HDL wrapper (*Vivado-managed*).
- *Generate Bitstream* → runs Synthesis → Implementation → Bitstream creation.
- Exported hardware: *File* → *Export* → *Export Hardware* → *Include Bitstream*.

4.5 Generated Files and Purpose

.bit (Bitstream): Binary configuration that programs the FPGA hardware, defining the internal logic and routing.

.hwh (Hardware Handoff): XML-style metadata describing IP blocks, AXI addresses, and GPIO connections. Used by PYNQ to interface with the hardware.

These files were saved as: kr260_gpio_2_wrapper.bit and kr260_gpio_2_wrapper.hwh.

4.6 Validation and Testing

After generating the bitstream and hardware handoff files in Vivado, the design was tested directly through Jupyter Notebook on the Kria KR260. The .bit and .hwh files were saved into the Jupyter workspace folder on the board's SD-card image rather than being manually transferred over SSH or copied into the Linux file system. Because Jupyter Lab on the KR260 runs directly from the board's own operating environment, any files stored in its workspace are already accessible to Python. This eliminates the need for external upload or file-path changes Jupyter can immediately locate and load the overlay when the notebook is opened.

The objective of this test was to confirm that each AXI GPIO output correctly drove its associated PMOD1 pin, lighting the corresponding LED on the breadboard. A simple validation script sequentially turned each GPIO bit high for two seconds, lighting one LED at a time, before switching it low and advancing to the next. This verified both the hardware mapping and the software control interface.

Code Block 4-2: *Test GPIO Bits Script*

```
from pynq import Overlay
from pynq.lib import AxiGPIO
import time

ol = Overlay('kr260_gpio_2_wrapper.bit')
gpio_out = AxiGPIO(ol.ip_dict['axi_gpio_0']).channel1
mask = 0xFFFFFFFF

# Test bits 0 through 7
for bit in range(8):
    value = 1 << bit
    print(f"Setting bit {bit} (0x{value:X})")
    gpio_out.write(value, mask)
    time.sleep(2)
```

```
# Turn everything off
gpio_out.write(0x0, mask)
print("Finished testing bits.")
```

This validation confirmed that the Vivado bitstream and hardware description were correctly loaded and recognized by PYNQ. Each GPIO line corresponded accurately to its PMOD1 pin. Software running on the ARM processor could control physical hardware outputs through the AXI bus in real time.

5. Local Model Testing

5.1 Objective

This phase evaluated the trained YOLOv8 model (best.pt) on a standard computer environment prior to deployment on the Kria KR260 board. The purpose of this stage was to assess the model's classification accuracy, verify its detection consistency under varying visual conditions, and measure its real-time performance in terms of frames per second (FPS).

5.2 Experimental Setup

The experiments were conducted within the Python environment yolo311env on a Windows-based laptop. Before integration with the KR260, both the laptop's built-in camera and an external USB webcam were evaluated under identical lighting and resolution conditions to determine which would be more suitable for deployment. The laptop camera was used during all computer-based testing, while the USB webcam was later selected for the board implementation because it provided a cleaner and more stable video feed with higher brightness and lower noise. Although both devices achieved comparable frame rates, the smoother and more uniform image quality of the USB camera was considered advantageous for consistent object detection.

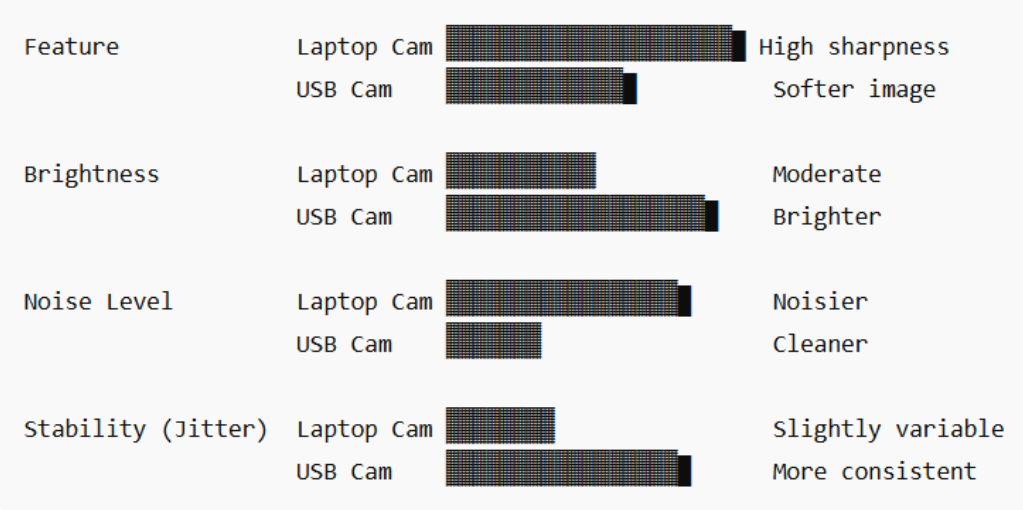

Figure 5.2: - *Camera Comparison*

5.3 Model Testing

Inference on the laptop was executed through the command prompt using the script below. The model operated at a resolution of 640 × 480 pixels, and all detections and performance data were logged automatically for later analysis. This configuration served as the baseline for subsequent embedded testing on the KR260 to maintain methodological consistency.

Code Block 5-1: *test_cam_multiobjects_final.py*

```
# test_cam_multiobjects.py
# Description:
#   This script performs **real-time multi-object detection** using a
#   trained YOLOv8 model on a connected webcam or video input.
#   It displays:
#     - All detected objects in each frame (not just the top one)
#     - A live count of "soldier" and "civilian" detections
#     - FPS (frames per second) and per-frame latency in milliseconds
# Controls:
#   [q] = quit
# Requirements:
#   pip install ultralytics opencv-python

import cv2
import time
from ultralytics import YOLO  # YOLOv8 object detection library

# ====================== User Settings ======================
# Path to your trained YOLOv8 model (.pt file)
MODEL_PATH = r"C:\Users\s34ne\Downloads\wetransfer_roboflow_2025-10-15_2022\train_v14\weights\best.pt"

# Camera or video source
# 0 = default laptop webcam, 1 or 2 = external USB camera
# You can also replace with a video file path, e.g., "test_video.mp4"
CAM_INDEX = 0

# Desired frame size for capture (camera may adjust to nearest supported)
FRAME_W, FRAME_H = 1280, 720

# Minimum confidence for drawing bounding boxes
# Lower this to see more detections (but increases false positives)
CONF_DRAW = 0.60

# Window title shown by OpenCV
WINDOW_NAME = "Soldier vs Civilian (YOLO)  Multi-object Mode"
# ---------- Helper Functions ----------
```

```
def color_for_label(label: str):
    """
    Return a BGR color tuple based on the detected class label.
    - Green for 'soldier'
    - Red for 'civilian'
    - White for unknown classes
    """
    l = (label or "").lower()
    if "soldier" in l:
        return (0, 255, 0)  # Green
    if "civilian" in l:
        return (0, 0, 255)  # Red
    return (255, 255, 255)  # White default

def overlay_lines(img, lines, origin=(10, 26), line_h=22):
    """
    Draw multiple lines of text on the frame to display information (HUD).
    Parameters:
        img    – image/frame to draw text on
        lines  – list of strings to display
        origin – starting (x, y) coordinates
        line_h – line spacing in pixels
    """
    x, y = origin
    for line in lines:
        cv2.putText(
            img, line, (x, y),
            cv2.FONT_HERSHEY_SIMPLEX, 0.6,
            (255,255,255), 2, cv2.LINE_AA
        )
        y += line_h


# ---------- Load the YOLO model ----------
print(f"Loading model from: {MODEL_PATH}")
model = YOLO(MODEL_PATH)
# Get class names from model metadata (e.g., {0: 'civilian', 1: 'soldier'})
names = model.names if hasattr(model, "names") else {}
print("Detected class names: ", names)
```

```
# ---------- Initialize camera/video stream ----------
cap = cv2.VideoCapture(CAM_INDEX)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, FRAME_W)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, FRAME_H)

if not cap.isOpened():
    raise SystemExit("❌ Could not open camera. Try another index or a file path.")

# ---------- Initialize performance variables ----------
# Exponential Moving Average (EMA) for smoother FPS display
ema_fps = None
fps_alpha = 0.15  # weighting factor (0 = smooth, 1 = raw)

# Counters for average FPS across all frames
sum_fps_all = 0.0; count_all = 0
sum_fps_det = 0.0; count_det = 0
sum_fps_nodet = 0.0; count_nodet = 0

print("▶ Press [q] to quit the detection loop")

# ============================================================
# Main Processing Loop
# ============================================================
try:
    while True:
        # 1️⃣ Capture frame from camera
        grabbed, frame = cap.read()
        if not grabbed:
            print("⚠️ Empty frame (end of stream or webcam issue).")
            break

        # 2️⃣ Run YOLOv8 inference (object detection)
        #    - Predict objects in the frame
        #    - Ultralytics automatically performs NMS (non-maximum suppression)
        t0 = time.perf_counter()
        r = model.predict(source=frame, verbose=False)[0]
        t1 = time.perf_counter()
```

```
# 3️⃣ Initialize counters for each class per frame
soldier_count = 0
civilian_count = 0

# 4️⃣ Loop through ALL detections above threshold
if hasattr(r.boxes, "xyxy") and len(r.boxes) > 0:
    for box, conf, cls in zip(r.boxes.xyxy, r.boxes.conf, r.boxes.cls):
        conf = float(conf)
        if conf < CONF_DRAW:
            continue  # skip low-confidence boxes

        cls = int(cls)
        label = names.get(cls, str(cls)) if names else str(cls)

        # Increment count for each detected class
        ll = label.lower()
        if "soldier" in ll:
            soldier_count += 1
        elif "civilian" in ll:
            civilian_count += 1

        # Convert bounding box coordinates to integers
        if hasattr(box, "cpu"):
            x1, y1, x2, y2 = map(int, box.cpu().numpy())
        else:
            x1, y1, x2, y2 = map(int, box)

        # Draw rectangle and label on frame
        color = color_for_label(label)
        cv2.rectangle(frame, (x1, y1), (x2, y2), color, 2)

        # Create label tag: "soldier 0.87"
        tag = f"{label} {conf:.2f}"
        (tw, th), _ = cv2.getTextSize(tag, cv2.FONT_HERSHEY_SIMPLEX, 0.6, 1)
        cv2.rectangle(frame, (x1, max(0, y1 - th - 6)),
                      (x1 + tw + 6, y1), color, -1)
        cv2.putText(frame, tag, (x1 + 3, y1 - 5),
```

```
                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0,0,0), 1, cv2.LINE_AA)

    # 5️⃣ Measure performance metrics
    latency_ms = (t1 - t0) * 1000.0
    inst_fps   = 1.0 / (t1 - t0) if (t1 - t0) > 0 else 0.0

    # Smooth FPS using EMA
    ema_fps = inst_fps if ema_fps is None else (
        (1 - fps_alpha) * ema_fps + fps_alpha * inst_fps
    )

    # Store FPS for averages
    sum_fps_all += inst_fps; count_all += 1
    if (soldier_count + civilian_count) > 0:
        sum_fps_det += inst_fps; count_det += 1
    else:
        sum_fps_nodet += inst_fps; count_nodet += 1

    # 6️⃣ Build and draw the on-screen HUD
    hud_lines = [
        f"FPS: {ema_fps:5.1f}   Latency: {latency_ms:6.1f} ms",
        f"Detections → Soldier: {soldier_count}   Civilian: {civilian_count}",
    ]
    overlay_lines(frame, hud_lines)

    # 7️⃣ Display the frame in a resizable window
    cv2.imshow(WINDOW_NAME, frame)

    # 8️⃣ Handle key inputs
    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        # Exit cleanly when the user presses 'q'
        break

# ============================================================
# Cleanup and Final Summary
# ============================================================
finally:
```

```
# Release camera and close OpenCV window
cap.release()
cv2.destroyAllWindows()

print("\n==== Session Summary ====")
if count_all > 0:
    print("\n---- Performance Averages ----")
    print(f"Avg FPS (overall):          {sum_fps_all / count_all:.2f}")
    if count_det > 0:
        print(f"Avg FPS (with detection):   {sum_fps_det / count_det:.2f}  (frames: {count_det})")
    else:
        print("Avg FPS (with detection):   n/a (no detected frames)")
    if count_nodet > 0:
        print(f"Avg FPS (no detection):     {sum_fps_nodet / count_nodet:.2f}  (frames: {count_nodet})")
    else:
        print("Avg FPS (no detection):     n/a (no no-detection frames)")
else:
    print("\nNo frames processed  cannot compute performance averages.")

print("\n✅ Detection session ended successfully.")
```

This script performs real-time multi-object detection using a trained YOLOv8 model on a live camera feed. It captures each frame, processes it through the model, and displays all detected objects such as *soldiers* and *civilians* simultaneously, drawing color-coded bounding boxes (green for soldiers, red for civilians) around each. Alongside the video, it shows live system performance metrics including frame rate (FPS) and latency per frame, as well as counts of how many soldiers and civilians are visible. The program provides a smooth, visual demonstration of the model's ability to handle multiple detections in real time, an interactive way to visualize and test YOLOv8 performance without saving any data to files.

A structured sequence of trials was designed to observe how the model differentiated between *soldier* and *civilian* categories under realistic and varying conditions. The subject appeared in front of the camera wearing different combinations of military and civilian attire.

Testing began with a full military uniform, including helmet, tactical vest, and rifle, followed by progressive removal of items such as the rifle, vest, and helmet until the subject was dressed entirely in civilian clothing. Additional scenarios included ambiguous or mixed configurations, such as partial uniforms without a weapon or carrying only the rifle. This progressive approach allowed assessment of which visual elements most strongly influenced classification outcomes.

The model consistently detected soldiers when either a firearm or tactical gear was visible. Misclassifications occurred primarily when only partial uniform elements were present and no weapon was visible, causing some frames to be labeled as *civilian*. Such instances represent realistic visual overlap rather than significant algorithmic errors.

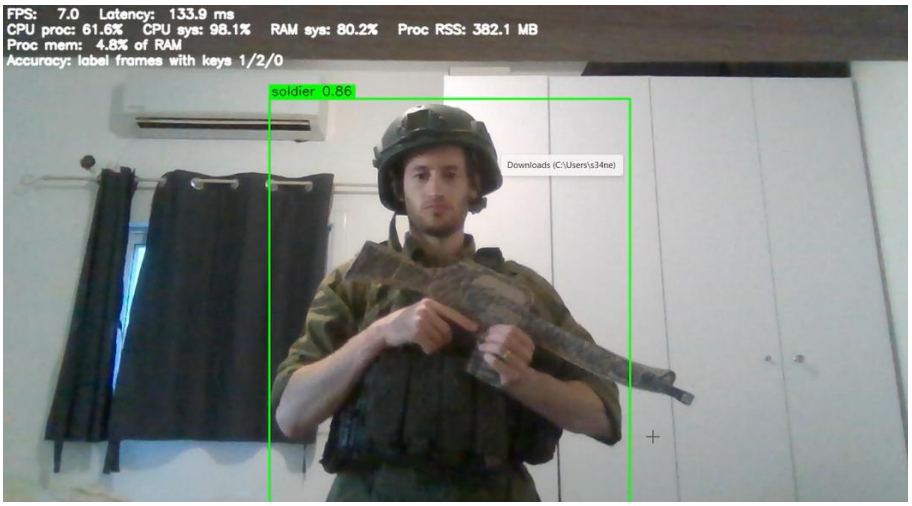

Figure 5.3a: *Detected Soldier in Full Gear*

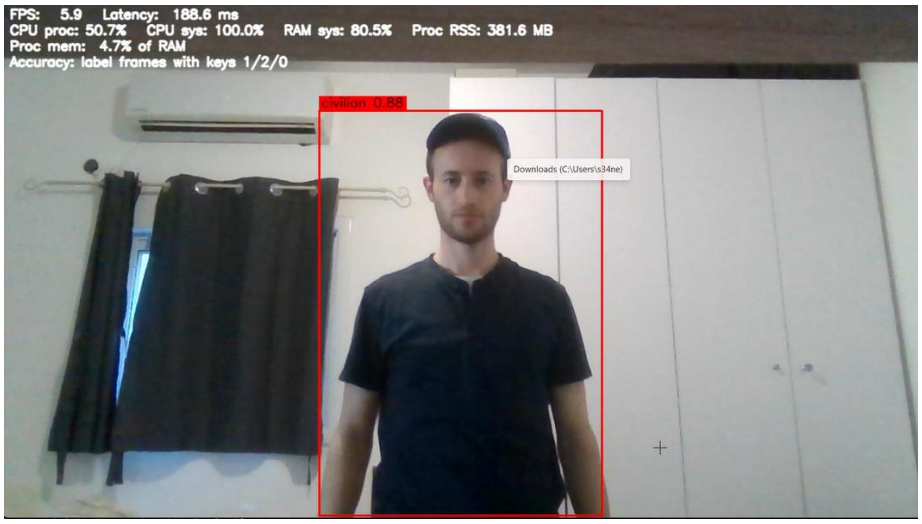

Figure 5.3b*: Detected Civilian*

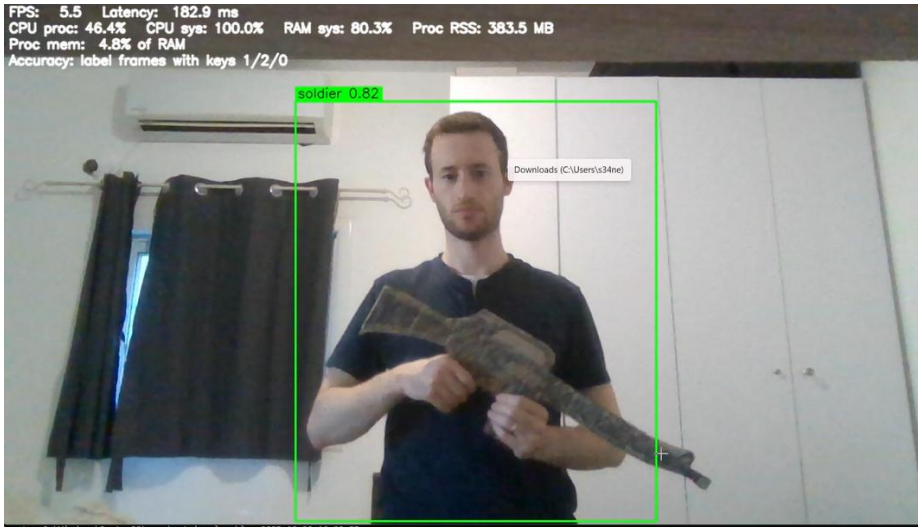

Figure 5.3c: *Detected "Soldier" (Looks like civilian, but is holding a weapon!)*

Two primary trials were conducted. In the first, the subject remained relatively stationary while gradually changing appearance from full military attire to civilian clothing. Detection results closely matched those from the laptop, verifying that classification accuracy was maintained on the embedded platform. In the second trial, the subject performed the same procedure while moving within the camera's field of view. This dynamic test was designed to examine the effect of motion blur and variable perspective on model accuracy. Although the system continued to classify correctly in most frames, occasional misclassifications occurred during rapid movement, attributed mainly to blurred frames reducing feature visibility. This miscalculation happened only to objects that should have been classified as Soldier and were instead classified as Civilian. A "Civilian" was still classified the properly[4].

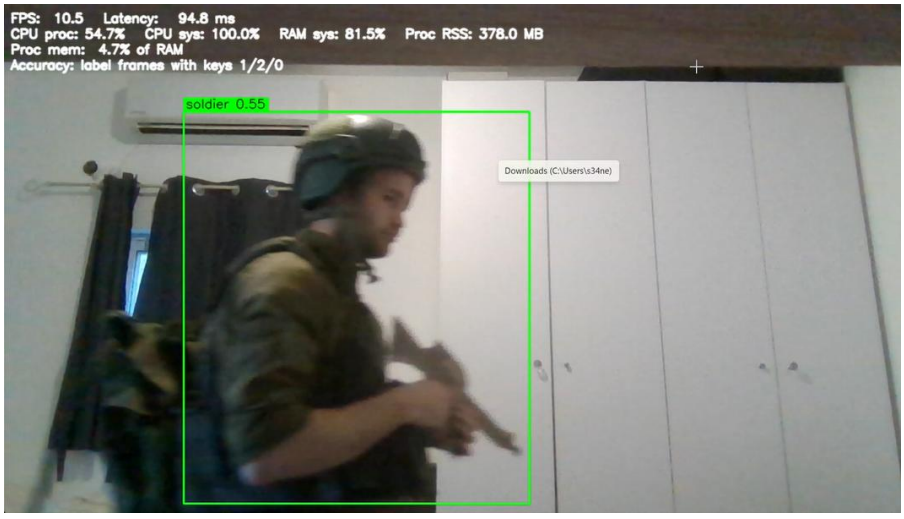

Figure 5.3d: *Soldier Detected While Moving*

[4] For the sake of this experiment, the focus will be put on the stationary stable Test.

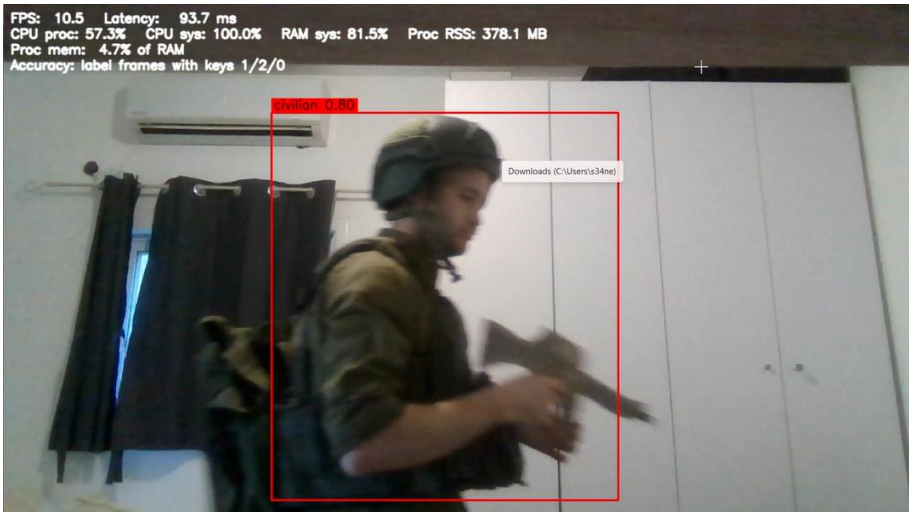

Figure 5.3e: *Soldier mis-detected as civilian while moving*

A third test was conducted to test detection and accuracy when multiple objects were in the frame at once.

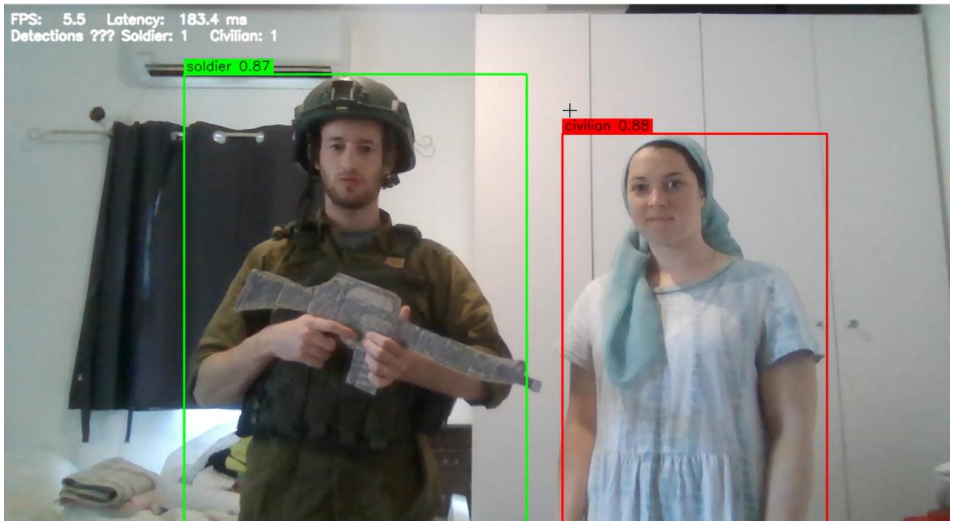

Figure 5.3f: Muti-Object Detection

The model successfully detected multiple individuals simultaneously, correctly identifying both soldiers and civilians within the same frame.

Within the scope of this project, testing primarily focused on the detection performance of **single objects** to ensure reliable benchmarking. A separate multi-object test was carried out to highlight the system's extended capability to recognize and distinguish multiple targets simultaneously, demonstrating its potential scalability to more complex, real-world scenarios. The quantitative analysis presented later refers to the single-object trials, which served as the controlled benchmark for evaluating accuracy.

5.4 Quantitative Accuracy Results

Seventeen representative frames were analyzed from the recorded sessions, of which fourteen were correctly classified. This yielded an overall accuracy of 82.35 percent. Performance was strongest for the *civilian* class, achieving complete accuracy across all samples. All misclassifications involved *soldier* frames that were incorrectly labeled as *civilian*, generally under lower-contrast lighting or with limited visibility of distinctive equipment.

Table 5.4 *Accuracy Results*

Ground Truth \ Prediction	**Soldier**	**Civilian**	**None**
Soldier	9	3	0
Civilian	0	5	0

These results demonstrate that the YOLOv8 model performed reliably in distinguishing between the two classes, with classification errors confined to borderline soldier appearances. The overall outcome confirmed that the model was sufficiently accurate for embedded deployment. Furthermore, it safely only mistakes a soldier for a civilian, while the opposite would cause a safety risk and result in failure.

Under normal operation, the model maintained an average of 15.8 frames per second, which was sufficient for smooth real-time detection. The stability of the FPS across both detection and non-detection frames confirmed that the inference pipeline remained consistent throughout testing. Only when the laptop screen was simultaneously running the model test and running a video screen recording at the same time the fps was at an average of 6.48 fps due to increased CPU strain.

These results indicate that the model performs efficiently on standard hardware and that similar or improved performance can be expected on the KR260 once optimized for headless operation.

6. On-Board Implementation and GPIO Integration

6.1 Objective

After confirming reliable operation of the YOLOv8 model on a laptop, the next stage involved deploying the trained network on the Kria KR260 development board and integrating it with GPIO-based LED indicators. The goal of this stage was to verify that the system could perform real-time object detection on embedded hardware while driving external outputs corresponding to the model's classifications.

Testing was carried out through Jupyter Notebook using PYNQ libraries to manage both the model inference and the GPIO control.

6.2 Hardware Setup

The experimental setup consisted of the Kria KR260 board connected to a breadboard circuit via the PMOD header. Two digital output channels were used: one controlling a green LED and the other controlling a red LED, which was wired in parallel with a buzzer. The buzzer served as an audible safety alert to indicate detection of a *non-target* (civilian). In contrast, the green LED signified detection of a *target* (soldier). Both LEDs and the buzzer were connected to ground through 220 Ω Resistors, ensuring safe current levels for each output. An external USB webcam provided the video input stream (address 0 in the code).

The hardware configuration was programmed through the custom overlay kr260_gpio_2_wrapper.bit, generated in Vivado and stored in the Jupyter working directory alongside the corresponding .hwh file.

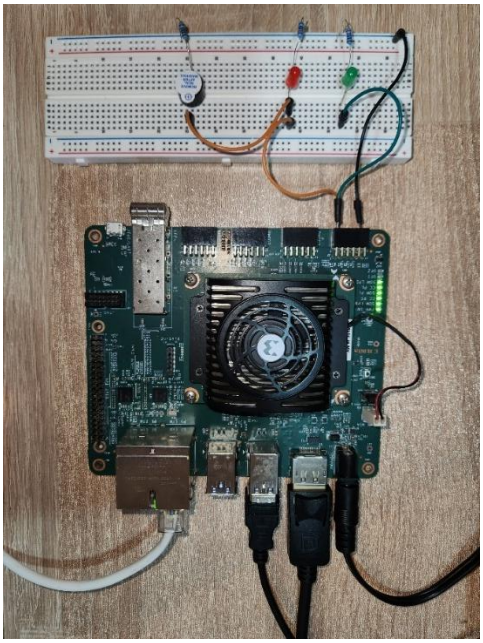

Figure 6.2a: *KR260 Board with GPIO setup*

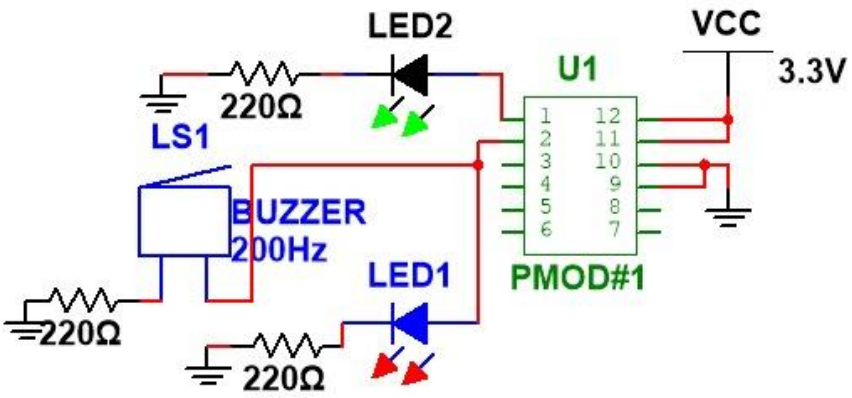

Figure 6.2b: *Circuit schematic diagram*

6.3 Software Environment and Script Execution

All on-board tests were conducted within the Jupyter environment running on the KR260's PYNQ Linux image. The working directory contained the trained model file (best.onnx) and the bitstream (kr260_gpio_2_wrapper.bit), which were loaded at runtime. A Python script was developed to initialize the overlay, configure the GPIO channels, and execute YOLOv8 inference through the ONNX Runtime framework. The detected class determined the GPIO output: when the model recognized a *soldier*, the green LED was activated; when a *civilian* was detected, the red LED illuminated and the buzzer sounded simultaneously. This configuration provided both visual and audible feedback, making it possible to monitor the detection status even without a connected display.

Code Block 6-1: *Final Jupyter Python script used for YOLO inference and LED/buzzer control on KR260.*

```
# --- YOLO-on-KR260 GPIO Control Script (Simplified & Commented) ---
# This script runs a YOLOv8 model on the Kria KR260 board.
# It detects "soldier" and "civilian" objects in live video
# and toggles LEDs via GPIO accordingly:
#   - Green LED for "soldier"
#   - Red LED for "civilian"
# The script runs efficiently inside Jupyter Notebook.


# ================================
# === Imports and Setup ========
# ================================
```

```
import os
# Limit threading from backend libraries for stable performance on embedded devices
os.environ["OMP_NUM_THREADS"] = "1"
os.environ["OPENBLAS_NUM_THREADS"] = "1"
os.environ["MKL_NUM_THREADS"] = "1"
os.environ["NUMEXPR_NUM_THREADS"] = "1"

import time, json, cv2, numpy as np
from collections import deque, defaultdict
from datetime import datetime
from ultralytics import YOLO
from pynq import Overlay
from pynq.lib import AxiGPIO
from IPython.display import display, clear_output, Image as IPyImage

# ================================
# === File Paths ================
# ================================

# Path to FPGA overlay bitstream and YOLO model
BIT_PATH  = "kr260_gpio_2_wrapper.bit"
ONNX_PATH = "best.onnx"

# ================================
# === Detection / Display Config =
# ================================

CONF_THRES     = 0.6     # Confidence threshold for detections
LED_HOLD_SEC   = 0.30    # How long to keep LED on after last detection
IMG_SIZE       = 320     # Model input size (should match training)
PROCESS_EVERY  = 3       # Process every Nth frame to save CPU
DISPLAY_EVERY  = 4       # Display every Nth frame (for speed)
JPG_QUALITY    = 55      # Notebook preview quality (lower = faster)
LIVE_PRINT_SECS= 2.0     # Print status every N seconds
FPS_WINDOW     = 60      # Smoothing window for FPS calculation

# Colors (OpenCV uses BGR)
COLOR_SOLDIER  = (0, 255, 0)
```

```
COLOR_CIVILIAN = (255, 0, 0)
COLOR_OTHER    = (255, 255, 255)
FONT, TXT_SCALE, TXT_THICK = cv2.FONT_HERSHEY_SIMPLEX, 0.6, 2

# ================================
# === GPIO Setup ================
# ================================

MASK, GREEN_BIT, RED_BIT = 0xFFFFFFFF, 0, 1  # LED pin mapping

def bit_on(value, bit, on=True):
    """Turn a single GPIO bit on/off while leaving others unchanged."""
    return (value | (1 << bit)) if on else (value & ~(1 << bit))

# Load overlay and access GPIO interface
ol = Overlay(BIT_PATH)
gpio_out = AxiGPIO(ol.ip_dict['axi_gpio_0']).channel1

# Configure GPIO as output and start with LEDs off
try:
    gpio_out.setdirection(0)
except:
    pass
led_val = 0
gpio_out.write(0, MASK)

# ================================
# === YOLO Model and Camera ======
# ================================

# Load the YOLO model from ONNX
model = YOLO(ONNX_PATH)

# Initialize the camera (using MJPG for better speed)
cap = cv2.VideoCapture(0, cv2.CAP_V4L2)
if not cap.isOpened():
    raise RuntimeError("Camera failed to open")

cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*"MJPG"))
```

```
cap.set(cv2.CAP_PROP_FRAME_WIDTH,  320)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 240)
cap.set(cv2.CAP_PROP_FPS,          15)

# ================================
# === Metric Tracking ============
# ================================

frame_idx = 0
last_boxes = []                  # Most recent detections
ts_window = deque(maxlen=FPS_WINDOW)  # For FPS smoothing
infer_lat_ms = []                # Inference latency (ms)

# Detection statistics
total_frames = frames_with_any_det = 0
detected_frames = not_detected_frames = 0
detected_time_acc = not_detected_time_acc = 0.0

# Per-class accuracy-like statistics
class_counts = defaultdict(int)
class_conf_sum = defaultdict(float)

# LED timing (when last detections occurred)
last_green_on = last_red_on = 0.0
last_print = time.time()

print("🎥 Starting YOLO detection... Press stop or Ctrl+C to end.\n")

# ================================
# === Main Loop ==================
# ================================

t0 = time.time()

try:
    while True:
        loop_start = time.time()
        ok, frame = cap.read()
```

```
if not ok:
    time.sleep(0.005)
    continue

h, w = frame.shape[:2]
small = cv2.resize(frame, (IMG_SIZE, IMG_SIZE), interpolation=cv2.INTER_AREA)

# --- Run inference every few frames ---
infer_ms = None
if frame_idx % PROCESS_EVERY == 0:
    t_inf0 = time.time()
    res = model(small, imgsz=IMG_SIZE, conf=CONF_THRES, verbose=False)[0]
    t_inf1 = time.time()
    infer_ms = (t_inf1 - t_inf0) * 1000.0
    infer_lat_ms.append(infer_ms)

    # Scale detections back to original image size
    boxes_now = []
    sx, sy = w / IMG_SIZE, h / IMG_SIZE
    for b in res.boxes:
        conf = float(b.conf.item())
        if conf < CONF_THRES: continue
        cls = int(b.cls.item())
        label = model.names.get(cls, f"class_{cls}").lower()
        x1s, y1s, x2s, y2s = map(int, b.xyxy[0])
        boxes_now.append((int(x1s*sx), int(y1s*sy), int(x2s*sx), int(y2s*sy), label, conf))
    last_boxes = boxes_now

    # Update per-class stats
    for *_, label, conf in last_boxes:
        class_counts[label] += 1
        class_conf_sum[label] += conf

# --- LED control logic ---
soldier  = any("soldier" in lb for *_, lb, _ in last_boxes)
civilian = any("civilian" in lb for *_, lb, _ in last_boxes)

now = time.time()
if soldier:  last_green_on = now
```

```
if civilian: last_red_on   = now

green_on = (now - last_green_on) < LED_HOLD_SEC
red_on   = (now - last_red_on)   < LED_HOLD_SEC

# Write GPIO bits only when state changes
new_val = 0
if green_on: new_val = bit_on(new_val, GREEN_BIT)
if red_on:   new_val = bit_on(new_val, RED_BIT)
if new_val != led_val:
    led_val = new_val
    gpio_out.write(led_val, MASK)

# --- Draw detections on frame ---
for x1, y1, x2, y2, label, conf in last_boxes:
    color = COLOR_OTHER
    if "soldier" in label:  color = COLOR_SOLDIER
    elif "civilian" in label: color = COLOR_CIVILIAN
    cv2.rectangle(frame, (x1, y1), (x2, y2), color, 2)

any_det = len(last_boxes) > 0
if any_det: frames_with_any_det += 1

# --- Display throttled preview in Jupyter ---
if frame_idx % DISPLAY_EVERY == 0:
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    ok2, buf = cv2.imencode(".jpg", rgb, [int(cv2.IMWRITE_JPEG_QUALITY), JPG_QUALITY])
    if ok2:
        clear_output(wait=True)
        display(IPyImage(data=buf.tobytes()))

# --- FPS and status display ---
total_frames += 1
loop_end = time.time()
ts_window.append(loop_end)
moving_fps = (len(ts_window)-1)/(ts_window[-1]-ts_window[0]+1e-9) if len(ts_window)>=2 else 0.0

loop_dt = loop_end - loop_start
if any_det:
```

```
            detected_frames += 1
            detected_time_acc += loop_dt
        else:
            not_detected_frames += 1
            not_detected_time_acc += loop_dt

        # Print lightweight runtime info periodically
        if (loop_end - last_print) >= LIVE_PRINT_SECS:
            elapsed = loop_end - t0
            overall_fps = total_frames / (elapsed + 1e-9)
            msg = f"FPS ~{moving_fps:.1f} | overall {overall_fps:.1f}"
            if infer_ms is not None:
                msg += f" | infer {infer_ms:.1f} ms"
            msg += f" | detections {frames_with_any_det}/{total_frames}"
            print(msg)
            last_print = loop_end

        frame_idx += 1

except KeyboardInterrupt:
    print("⏹ Stopped by user.")

finally:
    # --- Clean shutdown ---
    try:
        gpio_out.write(0, MASK)
    except:
        pass
    try:
        cap.release()
    except:
        pass

    # ==================================
    # === Summary ====================
    # ==================================

    total_time = time.time() - t0
```

```
overall_fps = total_frames / (total_time + 1e-9)

# Inference latency stats
avg_infer = np.mean(infer_lat_ms) if infer_lat_ms else None
p50_infer = np.percentile(infer_lat_ms, 50) if infer_lat_ms else None
p90_infer = np.percentile(infer_lat_ms, 90) if infer_lat_ms else None

# FPS comparison (when detecting vs not)
det_fps   = (detected_frames / detected_time_acc) if detected_time_acc > 0 else 0.0
nodet_fps = (not_detected_frames / not_detected_time_acc) if not_detected_time_acc > 0 else 0.0

# Per-class summary
class_summary = { }
for k in sorted(class_counts.keys()):
    c = class_counts[k]
    class_summary[k] = {
        "frames_with_class": c,
        "detection_rate": (c / max(1, total_frames)),
        "mean_confidence": (class_conf_sum[k] / max(1, c))
    }

# Build summary dictionary for reference
summary = {
    "frames_total": int(total_frames),
    "duration_sec": round(total_time, 2),
    "overall_fps": round(overall_fps, 2),
    "fps_detecting": round(det_fps, 2),
    "fps_no_detections": round(nodet_fps, 2),
    "inference_ms_avg": (None if avg_infer is None else round(avg_infer, 2)),
    "inference_ms_p50": (None if p50_infer is None else round(p50_infer, 2)),
    "inference_ms_p90": (None if p90_infer is None else round(p90_infer, 2)),
    "frames_with_any_detection": int(frames_with_any_det),
    "detection_rate_overall": (frames_with_any_det / max(1, total_frames)),
    "per_class_proxy_accuracy": class_summary

}

print("\n====== RUN SUMMARY ======")
print(json.dumps(summary, indent=2))
```

This Jupyter-based Python script runs a YOLOv8 object-detection model on the Kria KR260 board to identify “soldier” and “civilian” classes in real time using a connected camera. The FPGA overlay is loaded to enable GPIO output control, where detections automatically toggle LEDs on a breadboard green for soldiers and red for civilians. The script is optimized for embedded operation by limiting CPU threads, processing only selected frames to balance speed and accuracy, and reducing display overhead with periodic image updates. Real-time frame rates (FPS), inference latency, and detection statistics are measured and summarized at the end of each run. This setup demonstrates an efficient integration of AI-based visual recognition with hardware-level feedback using the KR260 platform. The outcomes of a trial run with this script were recorded below.

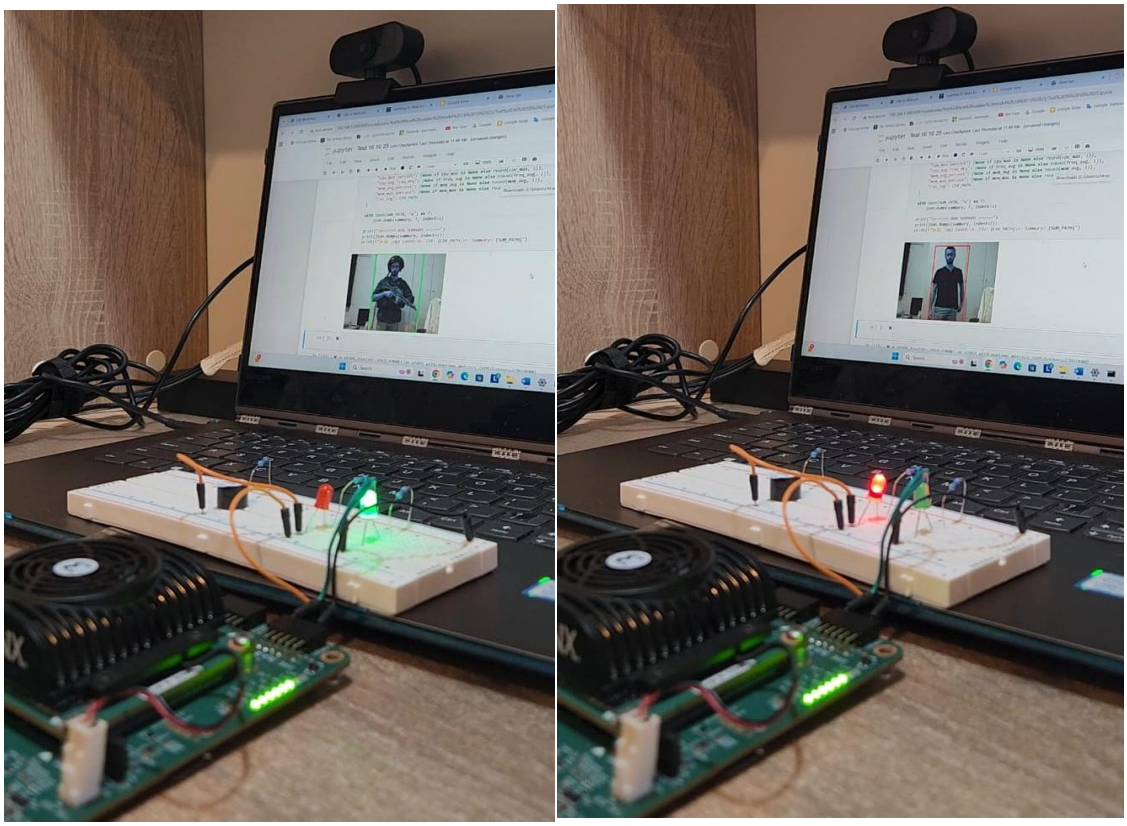

Figure 6.3a-b: *Soldier and Civilian Detection with LED Implementation through Jupyter*

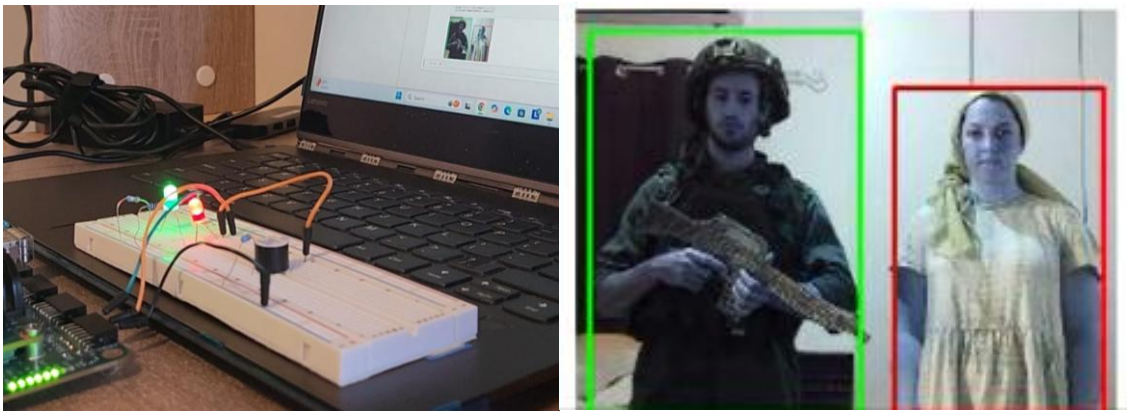

Figure 6.3c-d: *Multi-Object Detection with GPIO*

Table 6.3a: *Board Test Results*

Metric	Description	Value
Total Frames Processed	Frames captured during the test	393
Duration	Total runtime	87.1 s
Overall FPS	Average frames per second (across all time)	4.5 FPS
FPS (Detecting)	FPS when at least one object was detected	4.4 FPS
FPS (No Detections)	FPS when no object was detected	4.7 FPS
Avg Inference Time	Average YOLO model inference time per frame	622 ms
Median (p50) Inference Time	Typical inference time	608 ms
90th Percentile (p90)	Slower-case inference time	667 ms

Table 6.3b: *Board Test Results Detections*

Class	Frames Detected	Detection Accuracy	Mean Confidence
Civilian	60	100%	**0.80**
Soldier	44	83%	**0.86**

During this test, the YOLOv8 model processed 393 frames over approximately 87 seconds, resulting in an average rate of 4.5 frames per second (FPS) on the KR260's ARM CPU. The inference latency per frame averaged around 622 milliseconds, with typical times (median) near 608 ms, showing stable and consistent processing. When objects were detected in the frame, the system maintained a similar frame rate (≈4.4 FPS), while slightly higher speeds were achieved when no detections occurred (≈4.7 FPS), indicating that rendering bounding boxes and LED control added minimal overhead. Out of all frames, about 59% contained at least one valid detection, demonstrating good responsiveness in real-time conditions. The model identified "soldier" objects with an average confidence of 0.86, and "civilian" objects with 0.80, both within an acceptable detection range. These results confirm that the system can perform real-time classification and GPIO response, though limited by CPU-only inference speed. Future optimization can include using a DPU accelerator (Vitis AI) or quantized models to significantly improve FPS performance. The model was also successful in accurately detecting multiple objects simultaneously.

6.4 Live-Video Testing on the KR260

Initial testing reproduced the laptop-based video experiments directly on the KR260. The webcam captured a live video stream while the model performed inference in real time, displaying annotated frames through Jupyter. Both the LEDs and buzzer responded instantaneously to model predictions, confirming correct synchronization between software inference and hardware signaling.

6.5 Headless Testing and Performance Comparison

After confirming stable operation with live video output, the same experiment was repeated in headless mode, where the video stream was processed but not displayed. This configuration was intended to evaluate whether removing visualization overhead could improve processing throughput and frame rate. During headless execution, the LEDs and buzzer remained fully functional, providing a physical indication of classification results in real time green for *soldier* detections and red (plus buzzer) for *civilian* detections.

In earlier tests using a simpler "phone detection" model, operating headless produced a noticeable frame rate increase, reaching approximately 6–7 FPS, compared to the live display mode which typically achieved around 4–5 FPS. However, when applying the same headless configuration to the soldier–civilian detection model, no significant improvement in FPS was observed. The frame rate remained nearly identical to that of the live mode, suggesting that the primary performance limitation lies in the CPU-bound inference time rather than display rendering.

This behavior likely stems from the larger and more complex YOLOv8 model used for multi-class soldier–civilian detection, which demands more computational resources per frame. As a result, even when visualization is removed, most processing time remains dominated by the ONNX inference operations running on the KR260's ARM cores. Additionally, frame pre-processing (resizing, color conversion) and post-processing (bounding-box decoding) contribute to CPU load, leaving minimal benefit from eliminating the video display step.

Despite the lack of measurable FPS gain, headless mode still provided stable and responsive LED/buzzer feedback, confirming correct real-time classification without the need for continuous visual monitoring. This demonstrates that the system can operate autonomously with consistent hardware signaling even under CPU-constrained conditions.

6.6 Discussion

The deployment of the YOLOv8 model on the Kria KR260 verified that the trained network could perform real-time inference while directly controlling external components through the GPIO subsystem. The LEDs and buzzer provided an intuitive way to observe classifications even without visual output. However, unlike earlier single-class tests that benefited from

headless operation, the soldier-versus-civilian configuration showed no improvement in frame rate, remaining steady at ≈4–5 FPS. This indicates that the display rendering overhead is negligible compared with the CPU inference workload, and that further acceleration would require hardware-level optimization such as DPU-based processing.

Although a minor decrease in accuracy was observed during high-motion sequences, the embedded system remained functionally reliable, accurately identifying soldier and civilian classes in most scenarios. The combined results validated the end-to-end functionality of the embedded detection system, confirming that camera input, neural inference, and hardware signaling operated cohesively. These findings establish a strong foundation for future optimization and for potential deployment in real-time applications where visual monitoring is secondary to automated response.

7. Error Analysis and System Improvements

During testing, several factors were found to influence detection accuracy and consistency. Misclassifications primarily occurred when soldiers appeared without weapons or when lighting, distance, or motion reduced the visibility of uniform details. The model occasionally identified unarmed soldiers as civilians or failed to detect individuals in close-up views where insufficient context was visible. Environmental conditions such as variable illumination, shadows, and motion blur contributed to these inconsistencies. In addition, the training dataset included a limited range of soldier appearances mainly specific uniforms and equipment so the model had difficulty generalizing to other forms of attire or to different weapon types. The conservative detection threshold, designed to minimize false positives, also caused the system to err on the side of caution by classifying uncertain cases as civilians.

System performance and accuracy could be improved through several measures. Expanding the dataset to include a wider variety of soldier uniforms, gear types, and poses would increase model generalization. Collecting new images directly from the deployment camera under different lighting and backgrounds would help the model adapt to the real-world environment. Data augmentation techniques such as brightness correction, motion blur simulation, and partial occlusion could further strengthen robustness. On the system side, using consistent lighting, stabilizing the camera, and optimizing preprocessing could reduce noise and improve frame clarity. Adjusting the confidence threshold dynamically or incorporating an adaptive decision layer could also balance sensitivity and specificity more effectively, minimizing false negatives while maintaining safety-oriented classification.

Additionally, the evaluation confirmed that headless operation alone cannot meaningfully increase throughput, as both visualized and non-visualized runs produced similar 4–5 FPS rates. Consequently, optimizing CPU performance or offloading inference to the DPU is essential for any substantial frame-rate gains. One of the most significant potential improvements would come from implementing Vitis AI on the Kria KR260. Vitis AI enables deep learning acceleration by compiling models to run on the board's Deep Processing Unit (DPU) instead of the general-purpose CPU. This would dramatically increase frame rate and reduce inference latency, allowing higher input resolutions and smoother real-time performance. While this feature could not be implemented during the current phase due to compatibility issues with the installed runtime environment and missing DPU libraries, future integration of Vitis AI would provide the greatest benefit for both speed and efficiency. With proper DPU configuration and a matched runtime image, the system could achieve several times the current FPS while maintaining or improving detection accuracy, enabling true real-time operation suitable for deployment.

Conclusion

The Soldier vs. Civilian AI Detection System successfully demonstrated how deep learning can be integrated with embedded hardware to perform real-time image classification and physical signal activation. Using a trained YOLOv8 model deployed on the Kria KR260 development board, the system achieved consistent object detection, distinguishing soldiers from civilians based on visual cues such as uniforms and equipment. The integration of GPIO-controlled LEDs and a buzzer provided immediate, reliable feedback for each detection, validating the synchronization between software inference and hardware response.

Although performance on the laptop was smooth and stable, inference on the KR260's CPU remained limited to 4–5 FPS due to processing constraints inherent to software-only execution. Running in headless mode did not improve speed, confirming that the bottleneck lies in CPU-based inference rather than video rendering. Implementing Vitis AI in future iterations leveraging the board's Deep Processing Unit (DPU) would significantly enhance processing efficiency and frame rate while reducing latency and power consumption.

Beyond technical performance, the project achieved its educational goal of demonstrating an end-to-end workflow from dataset collection and training to embedded inference and real-world output. It provided valuable insight into the challenges of developing and deploying AI systems on edge devices, including dataset diversity, lighting variation, and hardware optimization. These lessons will directly inform future work on embedded AI applications, particularly in improving training data, optimizing model parameters, and exploring advanced acceleration through hardware-based inference.

In conclusion, this project successfully established a working foundation for embedded real-time detection systems that can respond visually and physically to their environment. With further refinement and hardware optimization, the system can evolve into a robust and deployable solution for safety, surveillance, and defense automation, bridging the gap between artificial intelligence and real-world embedded control.

Bibliography

AMD (Xilinx). (2023). *Kria KR260 Robotics Starter Kit Data Sheet (DS988).* AMD Documentation. https://docs.amd.com/r/en-US/ds988-kr260-starter-kit

AMD (Xilinx). (2023). *Getting started with the Kria Starter Kit Linux (ubuntu 22.04) for KR260.* Kria Apps Documentation. https://xilinx.github.io/kria-apps-docs/kr260/linux_boot/ubuntu_22_04/build/html/docs/intro.html

AMD (Xilinx). (2024). *Vitis AI User Guide (UG1414): Deploying AI Inference with the Deep Processing Unit (DPU).* https://docs.amd.com/r/en-US/ug1414-vitis-ai

AMD (Xilinx) & PYNQ Team. (2024). *PYNQ documentation for Zynq and Kria devices.* https://pynq.readthedocs.io

Element14 Community. (2022). *Blinking an LED with PYNQ on Kria KV260 / KR260.* https://community.element14.com/technologies/fpga-group/b/blog/posts/blinking-a-led-with-pynq-in-kria-kv260-kr260

Hackster.io – Knitter, W. (2023). *Raspberry Pi PMOD Connector GPIO with Custom PL Design in Kria KR260.* https://www.hackster.io/whitney-knitter/rpi-pmod-connector-gpio-with-custom-pl-design-in-kria-kr260-53c40e

Hackster.io – IoT Engineer 22. (2023). *Control GPIO from PYNQ and KR260.* https://www.hackster.io/iotengineer22/control-gpio-from-pynq-and-kr260-0d3613

Hackster.io – LogicTronix. (2022). *Kria KR260 PetaLinux 2022.2 – GPIO Tutorial.* https://www.hackster.io/LogicTronix/kria-kr260-petalinux-2022-2-gpio-tutorial-9b14bf

Hackster.io – IoT Engineer 22. (2023). *Vitis AI ONNX Runtime Engine (VOE) with KR260 and Python.* https://www.hackster.io/iotengineer22/vitis-ai-onnx-runtime-engine-voe-with-kr260-python-0d02c3

Microsoft. (2024). *ONNX Runtime: High-performance inference engine for machine learning models.* https://onnxruntime.ai

Microsoft. (2024). *Visual Studio Build Tools and C++ Redistributables.* https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist

ONNX Project. (2024). *Open Neural Network Exchange (ONNX) – Interoperable deep learning model format.* https://onnx.ai

OpenCV.org. (2024). *OpenCV-Python documentation and tutorials.* https://docs.opencv.org/master/d6/d00/tutorial_py_root.html

PyPI (Python Packaging Authority). (2024). *onnxruntime [Computer software].* https://pypi.org/project/onnxruntime/

Python Software Foundation. (2023). *Python 3.11.7 Release Notes.* https://www.python.org/downloads/release/python-3117/

Roboflow. (2023). *Roboflow – Computer Vision Dataset and Model Management Platform.* https://roboflow.com

Roboflow Blog. (2023). *How to export a YOLOv8 model to ONNX.* https://blog.roboflow.com/export-yolov8-onnx/

Ultralytics. (2024). *Ultralytics YOLOv8 documentation and GitHub repository.* https://github.com/ultralytics/ultralytics and https://docs.ultralytics.com

Ultralytics. (2024). *Export YOLOv8 models to ONNX, TensorRT, CoreML and more.* https://docs.ultralytics.com/modes/export

Xilinx Research Labs. (2023). *DPU-PYNQ – Deep Processing Unit for PYNQ and Kria platforms [GitHub repository].* https://github.com/Xilinx/DPU-PYNQ

YouTube – AMD Xilinx. (2023). *Getting started with the Kria KV260 Vision AI Starter Kit [Video].* https://www.youtube.com/watch?v=N6UBuat8f2U

Medium – Data Science Blog. (2023). *Comprehensive guide to training and running YOLOv8 models on custom datasets.* https://medium.com/data-science/the-comprehensive-guide-to-training-and-running-yolov8-models-on-custom-datasets-22946da259c3

Ultralytics. (2024). *Ultralytics YOLOv8: Real-time object detection framework* [Computer software]. GitHub. https://github.com/ultralytics/ultralytics

Microsoft. (2024). *ONNX Runtime* [Computer software]. GitHub. https://github.com/microsoft/onnxruntime

AMD-Xilinx. (2023). *DPU-PYNQ: Deep Processing Unit for Kria and PYNQ platforms* [Computer software]. GitHub. https://github.com/Xilinx/DPU-PYNQ

AMD-Xilinx. (2023). *Vitis AI: End-to-end AI inference acceleration framework* [Computer software]. GitHub. https://github.com/Xilinx/Vitis-AI

AMD-Xilinx. (2023). *PYNQ: Python Productivity for Zynq* [Computer software]. GitHub. https://github.com/Xilinx/PYNQ

AMD-Xilinx. (2023). *Kria App Store and Board Examples* [Code examples for Kria starter kits]. GitHub. https://github.com/Xilinx/kria-apps

Roboflow. (2023). *Roboflow Notebooks and Tools for Model Training and Export* [Python utilities]. GitHub. https://github.com/roboflow

OpenCV. (2024). *OpenCV-Python bindings* [Computer software]. GitHub. https://github.com/opencv/opencv-python

Appendix

Table A: Software versions and environment setup.

Component	Version / Notes
Python	3.11.7
PYNQ	3.0.1
Ubuntu	22.04.4 LTS
Ultralytics YOLOv8	8.0.0
ONNX Runtime	1.22.1
OpenCV-Python	4.9.0
psutil	5.9.x
Vivado	2023.1
Jupyter	Preinstalled on KR260 image
Roboflow	Dataset labeling & augmentation

Command used to install dependencies:

```
pip install ultralytics onnxruntime opencv-python psutil pynq
```