Running YOLOv8 Object Detection on Jetson Boards with OpenCV DNN C++

Welcome to this comprehensive guide on implementing YOLOv8 object detection on NVIDIA Jetson platforms using the OpenCV DNN module with C++. This presentation will walk you through the entire process from setup to optimization, providing practical insights for developing high-performance computer vision applications on edge devices.

Throughout this presentation, we'll cover essential aspects of deploying state-of-the-art object detection models on resource-constrained yet powerful Jetson hardware, enabling real-time inference for robotics, autonomous systems, and smart camera applications.

by RAGHUNATH.N

Overview

YOLOv8 Capabilities

State-of-the-art object detection framework offering superior accuracy and speed with support for detection, segmentation, and classification tasks.

Jetson Board Advantages

Power-efficient edge computing platforms with dedicated GPU acceleration for AI workloads, perfect for deploying computer vision applications.

OpenCV DNN Benefits

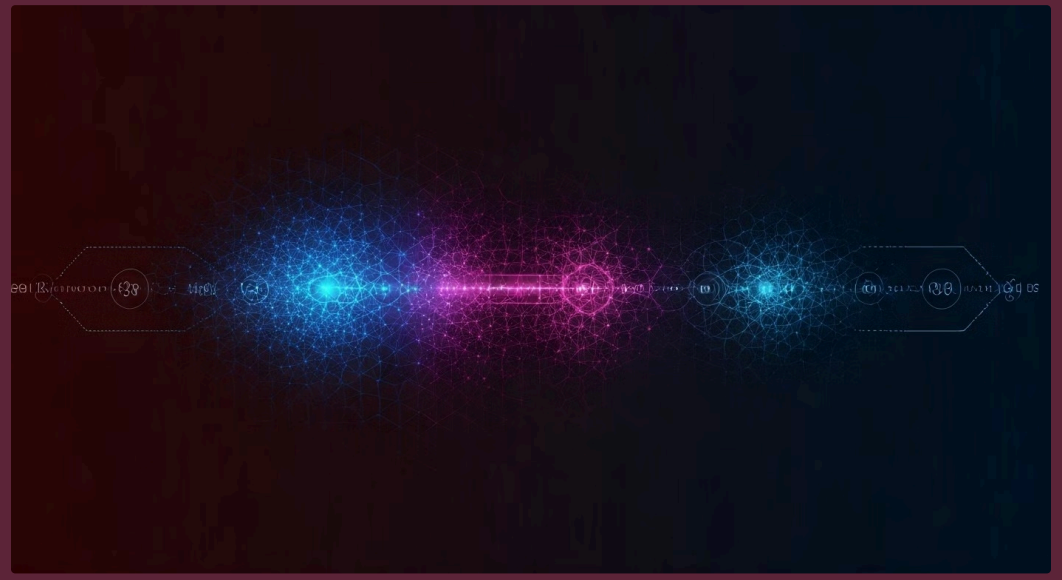
Cross-platform inference framework providing hardware acceleration support and simplified API for deep learning model deployment in C++.

YOLOv8 Introduction

State-of-the-Art Performance

YOLOv8 represents a significant advancement in the YOLO (You Only Look Once) family of object detection models. It offers improvements in both speed and accuracy over previous versions, making it ideal for real-time applications with limited computational resources.

The model architecture has been refined to better handle small objects and reduce false positives, addressing common challenges in previous YOLO iterations.



Built on the Ultralytics framework, YOLOv8 provides easy training, validation, and deployment workflows. It supports multiple tasks beyond detection, including instance segmentation and classification, all within a unified architecture.

NVIDIA Jetson Platforms

Jetson Nano

Entry-level AI computing device with 128 CUDA cores and 4GB RAM. Delivers up to 472 GFLOPS with power consumption of 5-10W. Ideal for beginners and projects with moderate computational requirements.

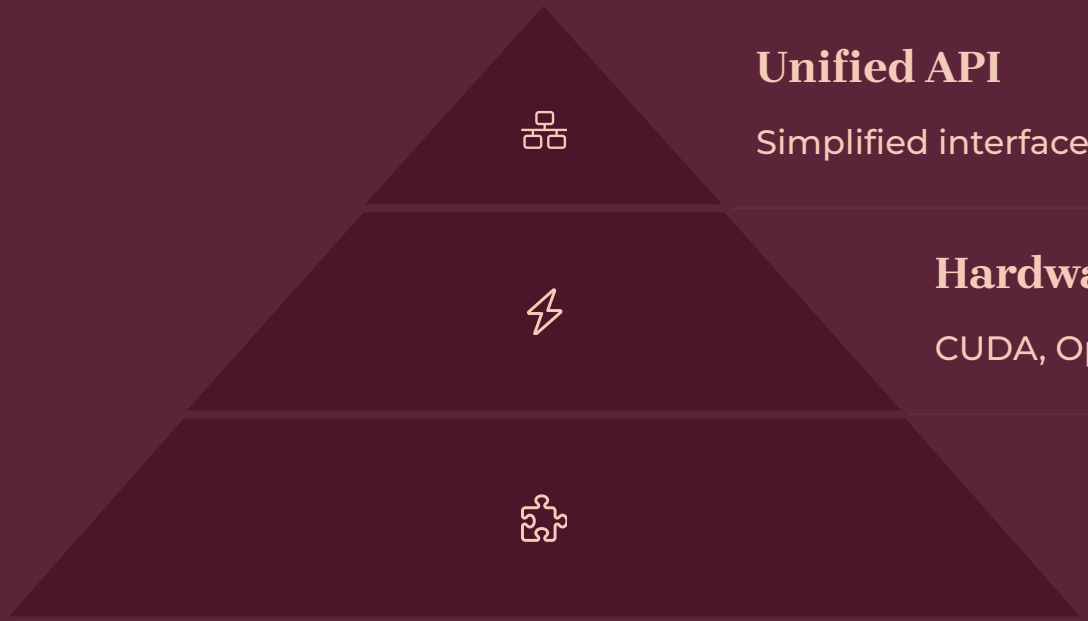
Jetson Xavier NX

Mid-range platform featuring 384 CUDA cores, 48 Tensor cores, and 8GB RAM. Delivers up to 21 TOPS of AI performance with 10-15W power draw. Great balance of performance and power efficiency.

Jetson AGX Orin

Flagship developer kit with 2048 CUDA cores, 64 Tensor cores, and up to 64GB RAM. Achieves 275 TOPS with 15-60W power consumption. Designed for the most demanding edge AI applications.

OpenCV DNN Module



Unified API

Simplified interface for different frameworks

Hardware Acceleration

CUDA, OpenCL, and FPGA support

Framework Support

Compatible with ONNX, TensorFlow, PyTorch, etc.

The OpenCV Deep Neural Network (DNN) module provides a comprehensive and high-level API for deploying deep learning models across platforms. It acts as an abstraction layer between your application code and various AI frameworks, allowing developers to focus on application logic rather than framework-specific details.

On Jetson platforms, the DNN module can leverage CUDA acceleration to significantly improve inference performance. This enables efficient deployment of complex models like YOLOv8 with minimal code complexity.

Setup Requirements

JetPack SDK Installation

Flash your Jetson board with the latest JetPack SDK (minimum 4.6+) using NVIDIA SDK Manager. This includes the Linux OS, CUDA toolkit, cuDNN, TensorRT, and other essential libraries for AI development.

OpenCV with CUDA Support

Build OpenCV 4.5+ from source with CUDA, cuDNN, and GStreamer support enabled. This ensures optimal performance for the DNN module and video processing capabilities required for real-time object detection.

Development Tools

Install build essentials, CMake 3.10+, and other development tools needed for compiling C++ applications. Set up proper environment variables for CUDA and OpenCV paths to simplify the build process.

YOLOv8 Model Preparation

Obtain Model

Download pre-trained YOLOv8 weights from Ultralytics repository or train custom model using your dataset

Convert to ONNX

Export PyTorch model to ONNX format using Ultralytics export utility with appropriate parameters

Optimize for Jetson

Use TensorRT to optimize ONNX model for specific Jetson hardware target

The ONNX format acts as a bridge between PyTorch and OpenCV DNN, allowing for framework-independent model deployment. When exporting the model, consider simplifying the network by removing training-specific layers and ensuring proper input/output node configurations to match what the OpenCV DNN module expects.

For the best performance on Jetson platforms, further optimization using TensorRT is recommended. This can be done directly during inference or as a preprocessing step to create an optimized engine file.

C++ Environment Setup

Project Structure

Create a well-organized directory structure with separate folders for source code, build files, models, and resources. This promotes maintainability and easier collaboration.

CMake Configuration

Set up CMakeLists.txt to find and link OpenCV and CUDA dependencies. Configure build flags for optimization and specify C++11 or higher standard support.

Build Environment

Configure environment variables for CUDA, cuDNN, and OpenCV paths. Install any additional dependencies required for your specific application needs.

Test Build

Verify your setup with a simple test program that initializes OpenCV and checks CUDA availability before implementing the full application.

Loading YOLOv8 ONNX Model

```
// Load YOLOv8 ONNX model
cv::dnn::Net net = cv::dnn::readNetFromONNX("yolov8n.onnx");

// Set computation backend and target
net.setPreferableBackend(cv::dnn::DNN_BACKEND_CUDA);
net.setPreferableTarget(cv::dnn::DNN_TARGET_CUDA);

// Configure model parameters
const float inputWidth = 640.0;
const float inputHeight = 640.0;
const float scale = 1.0/255.0;
const cv::Scalar mean = cv::Scalar(0, 0, 0);
const bool swapRB = true;
```

The code above demonstrates how to load a YOLOv8 ONNX model using the OpenCV DNN module. The critical configuration involves selecting the appropriate backend and target to enable hardware acceleration on Jetson platforms. This ensures the inference will leverage the GPU rather than running on the CPU.

When setting up the model parameters, pay careful attention to the input dimensions, scaling factors, and channel ordering. These must match the preprocessing steps used during model training and export to ensure accurate results.

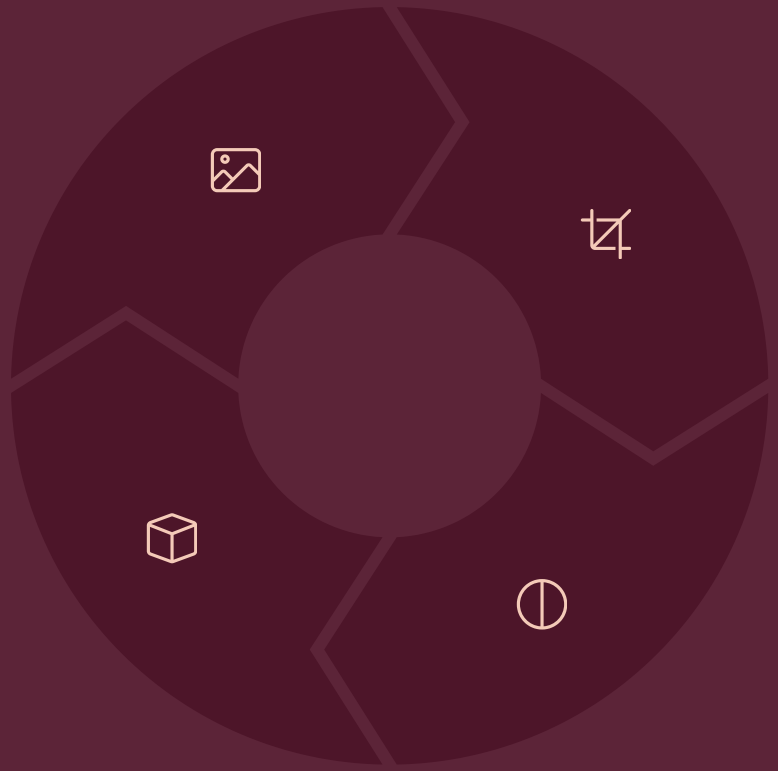
Image Preprocessing

Load Image

Read input using cv::imread or camera feed

Resize & Crop

Maintain aspect ratio with letterboxing



Create Blob

Convert to network input format

Normalize

Scale pixel values to [0,1] range

Proper preprocessing is critical for achieving accurate detection results. The input image must be transformed to match the expected format of the YOLOv8 model. This typically involves resizing to the model's expected dimensions (usually 640×640) while preserving the aspect ratio to avoid distortion.

After resizing, the image pixels must be normalized by scaling values from the [0,255] range to [0,1]. The cv::dnn::blobFromImage function handles these transformations efficiently, creating a properly formatted 4D tensor (NCHW format) required by the network.

Running Inference

1	8-30	4
Forward Pass	**FPS Range**	**Output Matrices**
Push preprocessed image through the neural network	Typical performance on Jetson platforms	Number of tensors to process from YOLOv8

```
// Set the prepared blob as input to the network
net.setInput(blob);

// Perform forward pass and measure time
auto start = std::chrono::high_resolution_clock::now();
std::vector outputs;
net.forward(outputs, net.getUnconnectedOutLayersNames());
auto end = std::chrono::high_resolution_clock::now();

// Calculate inference time
float inferenceTime = std::chrono::duration(end - start).count();
std::cout << "Inference time: " << inferenceTime << " ms"
          << " (" << 1000.0/inferenceTime << " FPS)" << std::endl;
```

Post-processing Detections

Parse Output Tensors

Extract detection data from network output

Apply Confidence Threshold

Filter low-confidence predictions

Non-Maximum Suppression

Remove overlapping bounding boxes

Map Class Labels

Assign class names to detections

YOLOv8's output format differs from previous versions, requiring careful parsing. The network outputs detection results as a single array with each row representing a detection. This array contains bounding box coordinates, confidence scores, and class probabilities for each detected object.

Non-maximum suppression (NMS) is essential to eliminate duplicate detections of the same object. By comparing the Intersection over Union (IoU) of bounding boxes and keeping only the highest confidence detection where significant overlap occurs, we can clean up the results for a more accurate visualization.

Drawing Bounding Boxes

```
// Function to draw detection results
void drawDetections(cv::Mat& image,
                    const std::vector& detections) {
    for (const auto& detection : detections) {
        // Convert normalized coordinates to image coordinates
        int xmin = static_cast(detection.box.x * image.cols);
        int ymin = static_cast(detection.box.y * image.rows);
        int width = static_cast(detection.box.width * image.cols);
        int height = static_cast(detection.box.height * image.rows);

        // Draw rectangle and label
        cv::rectangle(image, cv::Rect(xmin, ymin, width, height),
                      colors[detection.classId], 2);

        std::string label = cv::format("%s: %.2f",
                           classes[detection.classId].c_str(),
                           detection.confidence);

        int baseLine;
        cv::Size labelSize = cv::getTextSize(label,
                             cv::FONT_HERSHEY_SIMPLEX,
                             0.5, 1, &baseLine);

        // Draw background for text
        cv::rectangle(image,
                      cv::Point(xmin, ymin - labelSize.height - 5),
                      cv::Point(xmin + labelSize.width, ymin),
                      colors[detection.classId], cv::FILLED);

        // Put text on the image
        cv::putText(image, label, cv::Point(xmin, ymin - 5),
                    cv::FONT_HERSHEY_SIMPLEX, 0.5,
                    cv::Scalar(255, 255, 255), 1);
    }
}
```

Optimizing Performance

TensorRT Integration

Convert ONNX model to TensorRT engine for up to 2-3x performance improvement. Configure precision (FP32, FP16, or INT8) based on accuracy requirements and hardware support.

Batch Processing

Process multiple frames simultaneously to amortize model loading and optimization overhead. Especially useful for multi-camera systems or video processing applications.

Asynchronous Pipeline

Implement separate threads for image acquisition, preprocessing, inference, and visualization to maximize throughput. Use producer-consumer patterns with thread-safe queues.

When targeting maximum performance on Jetson platforms, it's essential to balance model complexity with inference speed. Consider using smaller YOLOv8 variants (YOLOv8n or YOLOv8s) for real-time applications where latency is critical, and larger variants for offline processing where accuracy is paramount.

Handling Video Streams

Video Capture Setup

OpenCV provides versatile video capturing capabilities through the cv::VideoCapture class. On Jetson platforms, leveraging GStreamer backends allows for hardware-accelerated decoding of various video formats, reducing CPU load during capture.

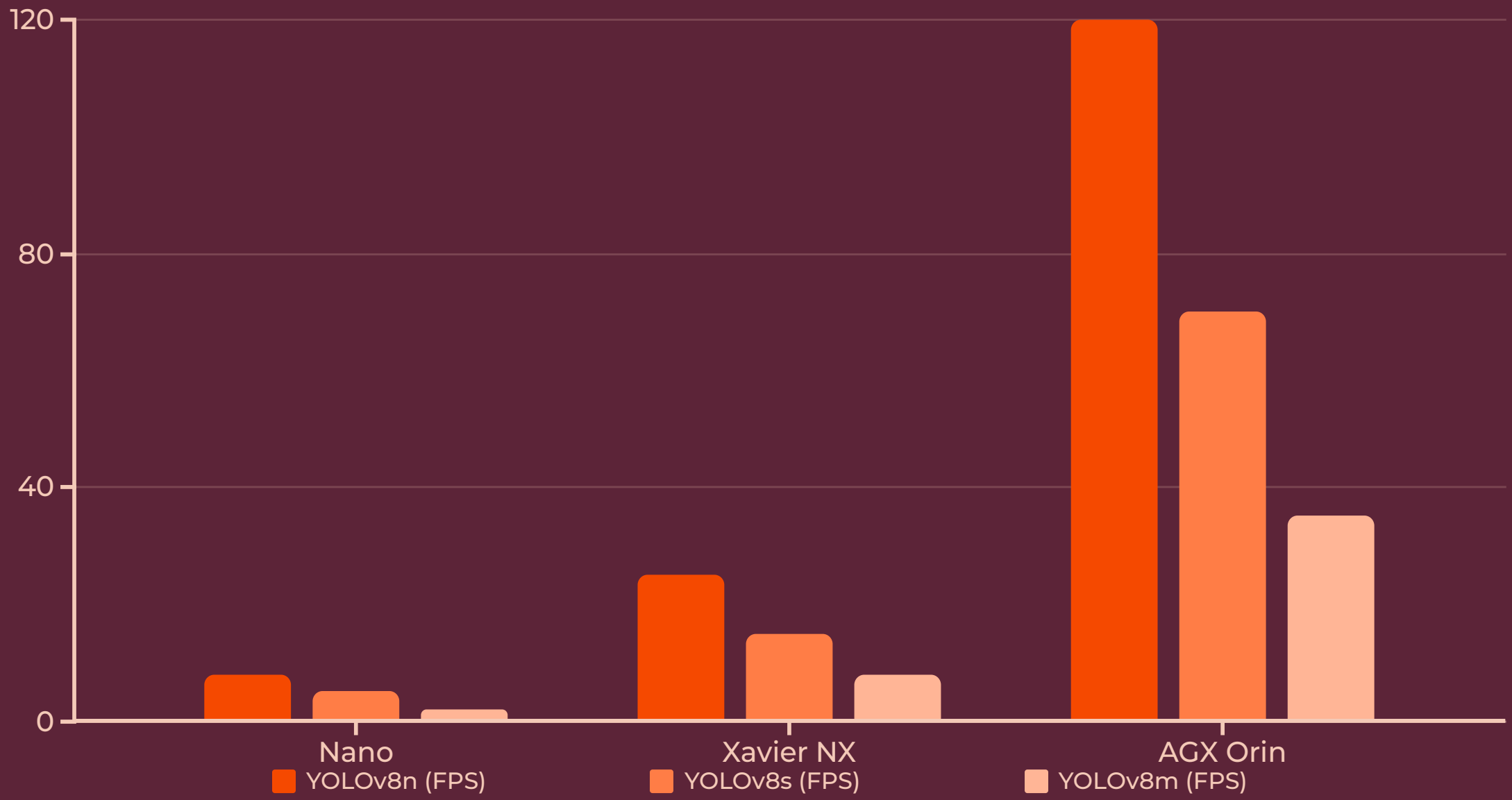
```
// Configure GStreamer pipeline for hardware-accelerated video capture
std::string pipeline = "nvcamerasrc ! "
                       "video/x-raw(memory:NVMM), "
                       "width=1280, height=720, "
                       "format=NV12 ! "
                       "nvvidconv ! video/x-raw, "
                       "format=BGRx ! "
                       "videoconvert ! video/x-raw, "
                       "format=BGR ! appsink";
cv::VideoCapture cap(pipeline, cv::CAP_GSTREAMER);
```

Real-time Processing Loop

The main processing loop reads frames, performs detection, and displays results in real-time. Implementing a frame-skipping strategy can help maintain consistent frame rates on resource-constrained platforms, sacrificing some frames to ensure timely processing of others.

For applications requiring temporal consistency, consider implementing simple tracking algorithms to maintain object identities across frames, even when processing at lower frame rates. This can provide smoother visual results and more stable detection outputs.

Benchmarking



The performance of YOLOv8 varies significantly across different Jetson platforms and model sizes. The benchmark results above demonstrate the frames per second (FPS) achievable with different configurations using TensorRT optimization at FP16 precision with a 640×640 input resolution.

When benchmarking on your specific hardware, consider measuring not just average FPS but also inference latency variance. High variance can indicate thermal throttling or resource contention issues. Also evaluate detection accuracy using metrics like mAP (mean Average Precision) to ensure optimizations don't significantly degrade detection quality.

Common Challenges

Memory Limitations

Jetson platforms have limited RAM, which can constrain the size of models and batch processing capabilities. The Nano with just 4GB RAM may struggle with larger YOLOv8 variants like YOLOv8l or YOLOv8x, while the AGX Orin with up to 64GB can handle them comfortably.

Strategies: Use smaller model variants, implement model pruning or quantization, optimize memory usage in your application code.

Thermal Management

Sustained high GPU utilization can cause thermal throttling, reducing performance over time. This is particularly noticeable in compact enclosures or environments with poor airflow.

Strategies: Implement proper cooling solutions, use power modes appropriately, monitor temperatures and adjust workloads dynamically.

Power Constraints

For battery-powered applications, balancing performance with power consumption is crucial. Higher performance typically comes at the cost of increased power draw.

Strategies: Use NVIDIA's power modes (e.g., 10W mode for Xavier NX), implement duty cycling for intermittent processing, optimize model efficiency.

Best Practices

Model Optimization

Use quantization (FP16/INT8), pruning, and knowledge distillation to reduce model size and computational requirements while maintaining acceptable accuracy.

System Configuration

Set appropriate CUDA thread count, maximize clocks with jetson_clocks, configure proper cooling, and enable maximum performance mode when thermal solutions permit.

Memory Management

Minimize memory allocations during inference, reuse buffers, implement proper resource cleanup, and monitor memory usage to prevent leaks.

Future Developments

YOLOv8 Enhancements

Ultralytics continues to refine YOLOv8 with regular updates improving accuracy, speed, and additional task support. Expect better feature extractors, more efficient head designs, and improved training methodologies.

1

OpenCV DNN Improvements

Future OpenCV releases will likely include better TensorRT integration, support for newer model architectures, and improved quantization support for edge deployment.

2

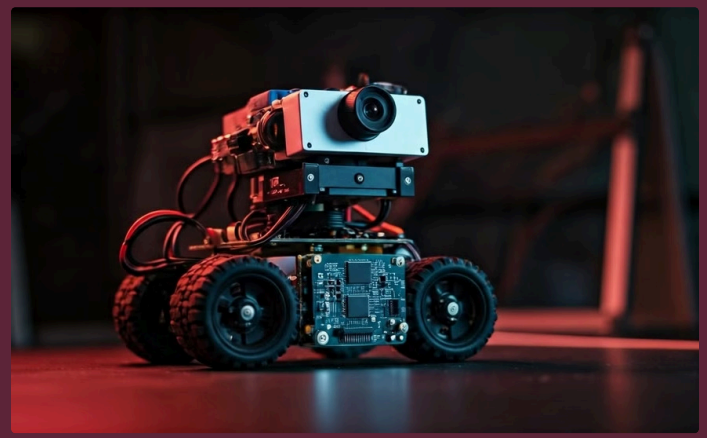
Jetson Platform Evolution

NVIDIA's roadmap includes more powerful Jetson modules with enhanced AI performance, greater memory bandwidth, and improved power efficiency. Watch for new SoC architectures incorporating the latest GPU technology.

3

As the field of edge AI continues to evolve rapidly, staying current with these developments will be crucial for maintaining competitive performance. Upcoming advancements in hardware-specific neural architecture search and automated deployment optimization tools promise to further simplify the deployment of complex models on resource-constrained edge devices.

Conclusion



Robotics & Automation

Implement YOLOv8 on Jetson platforms for autonomous navigation, object manipulation, and human-robot interaction scenarios requiring real-time perception capabilities.

Smart Retail

Deploy in-store analytics solutions for customer behavior analysis, automated inventory management, and checkout-free shopping experiences with privacy-preserving edge processing.

Industrial Inspection

Create quality control systems capable of identifying defects, monitoring production lines, and ensuring worker safety through real-time monitoring and anomaly detection.

We've covered the complete workflow for implementing YOLOv8 object detection on Jetson platforms using OpenCV DNN in C++. From environment setup and model preparation to optimization techniques and practical applications, you now have the knowledge to deploy efficient computer vision solutions at the edge.

For further learning, explore the NVIDIA Developer Zone, Ultralytics GitHub repository, and OpenCV documentation. Join community forums to share experiences and stay updated on best practices in this rapidly evolving field.