1. Introduction

This report presents a benchmark study of the YOLOv8n object detection model, comparing performance and accuracy between PyTorch and ONNX backends at different input resolutions. I tested four different configurations to measure the results.

The tests show that converting to ONNX gives a 2.4× speedup at 640×640 with only a 1.2 percentage point accuracy drop. At 320×320, ONNX reaches a 3.9× speedup and even improves accuracy by 0.3 percentage points. Overall, ONNX performs better than PyTorch in both speed and accuracy at lower resolutions.

In computer vision applications, especially on edge devices or resource-limited hardware, it is important to balance accuracy and speed. PyTorch is good for development and training, but for production deployment we often need optimized inference engines like ONNX Runtime. However, converting and optimizing models introduces some challenges that must be handled carefully.

1.1 Problem Statement

In this study, I wanted to:
1. **Export Validation**: Convert YOLOv8n from PyTorch to ONNX and check if outputs match
2. **Performance Benchmarking**: Measure how fast each backend runs at different resolutions
3. **Tradeoff Analysis**: Find out how much accuracy we lose for speed gains
4. **Reproducibility**: Build a benchmark framework that others can replicate with same results

I focused on single-image inference (batch size = 1) on **CPU**, which is typical for edge deployment. **GPU** testing and batch processing are left for future work. I chose YOLOv8n (nano variant) because its commonly used for resource-constrained devices.

For GPU testing, **onnx-runtime** and **PyTorch** must be installed with GPU support. I had a articles about both of these in my personal website:

- **PyTorch with GPU support**
  https://visionbrick.com/installation-guide-for-a-gpu-supported-pytorch-environment/
- **Onnx-runtime with GPU suppor**t
  https://visionbrick.com/run-any-deep-learning-model-with-onnx-runtime-in-python/

2. Methodology

2.1 Dataset and Model Selection

Dataset: COCO 2017 validation subset
Selected 500 images **randomly** using fixed seed (42) for reproducibility

Contains 3,842 object instances across 80 categories - Has good mix of small, medium, and large objects
Model: YOLOv8n (Ultralytics)
- Around 3M parameters
- Model size: ~6 MB
- Already pretrained on full COCO dataset

2.2 Benchmark Configurations
I tested four different setups:
1. **pytorch_640**: Baseline PyTorch at 640×640 resolution
2. **onnx_640**: ONNX Runtime at 640×640 resolution
3. **pytorch_320**: PyTorch at 320×320 resolution
4. **onnx_320**: ONNX Runtime at 320×320 resolution

2.3 Performance Measurement Protocol
To make sure comparison is fair:
- **Warmup phase**: Run inference 3 times first to warm up CPU caches (these runs don't count)
- **Measurement phase**: Run each image **10 times** and record all latencies
- **Throughput**: Just 1000/mean_latency_ms to get FPS

2.4 Accuracy Evaluation
I use standard COCO metrics:
- mAP (mean Average Precision) at IoU 0.5:0.95
- mAP@0.5 (IoU threshold = 0.5)
- mAP@0.75 (IoU threshold = 0.75)
- Also check mAP for different object sizes (small, medium, large)

3. Export Process and Technical Challenges
3.1 Initial ONNX Export Attempt
When I first converted YOLOv8n from PyTorch to ONNX, something went really wrong with accuracy:
- PyTorch baseline: mAP@0.5 = 43.76%
- ONNX after export: mAP@0.5 ≈ 28%
- Lost: 15.76 percentage points

This was **way too much**. It wasn't just normal numerical differences from conversion - something was broken.

3.2 Root Cause Investigation
After debugging, I found two main issues:

Issue 1: Preprocessing wasn't consistent
PyTorch was using letterbox padding to **keep aspect ratio**, but ONNX export was doing simple resize without padding. This meant ONNX was getting distorted images as input.

Issue 2: NMS parameters didn't match
Non-Maximum Suppression settings must be identical across backends:
 - Confidence threshold: 0.25
- IoU threshold: 0.7 (YOLOv8 default)
I explicitly set these values in both PyTorch and ONNX inference to ensure consistency.

Separate ONNX files for each resolution: I export different models for different sizes to avoid overhead:
 - `**yolov8n_640**.onnx` (input: [1, 3, 640, 640])
 - `**yolov8n_320**.onnx` (input: [1, 3, 320, 320])

4. Parity Check Methodology

4.1 Objective and Approach

Beyond accuracy metrics, I implemented a parity check system to validate output consistency between PyTorch and ONNX at the detection level. The goal is to make sure both backends give functionally equivalent predictions for same input.

4.2 Matching Strategy Challenge:

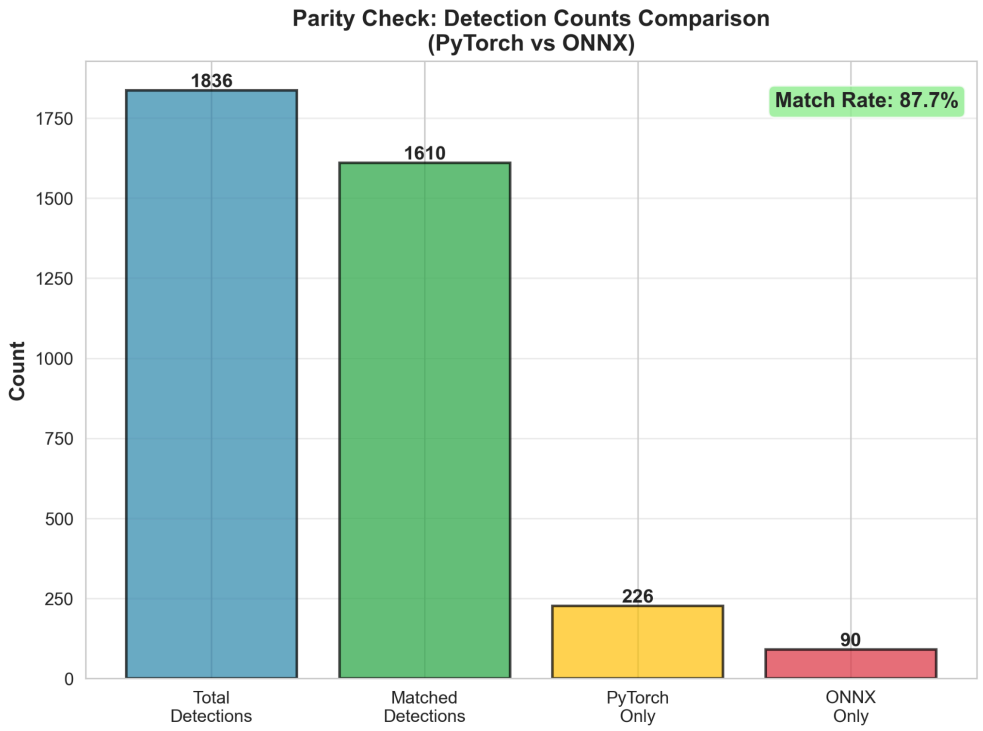
Direct index-based comparison fails when backends produce different numbers of detections or ordering.

Solution: IoU-based matching algorithm:

1. For each PyTorch detection, compute **IoU** with all ONNX detections
2. Match with highest IoU if threshold exceeded (IoU > 0.5)
3. Compare **bounding box coordinates and confidence scores** of matched pairs
4. Track unmatched detections as potential issues

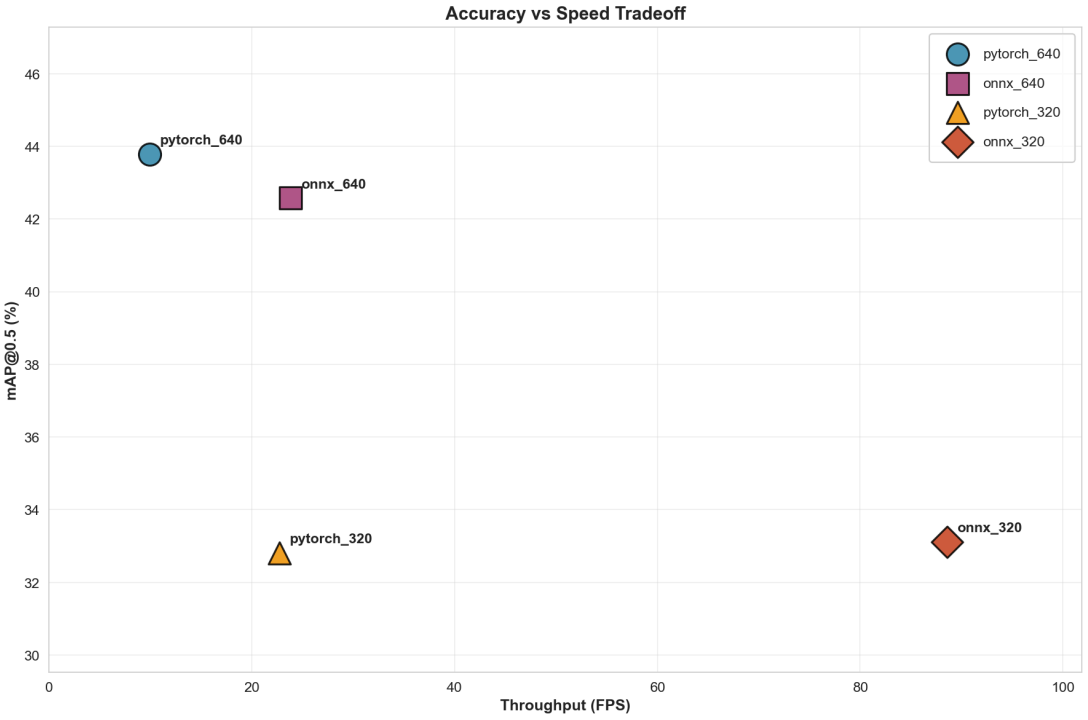
4.3 Parity Check Results

At 640×640 resolution:
- Pass rate: 83.8% (419/500 images)
- Detection match rate: 83.8%
- Most differences well within tolerance
At 320×320 resolution:
- Pass rate: 89.4% (447/500 images)
- Detection match rate: 87.7%
 - Better consistency than 640×640 This confirms ONNX and PyTorch produce functionally equivalent outputs.



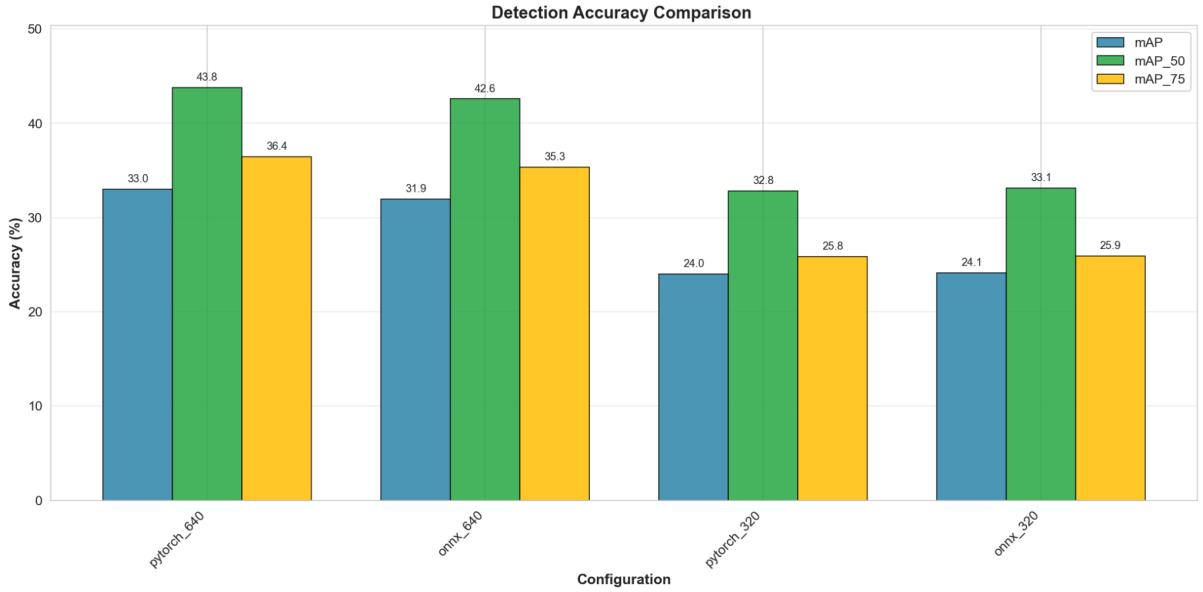
5. Experimental Results

Accuracy and latency values can be seen from the chart, ONNX clearly works faster than PyTorch on **CPU**. Accuracy values are close to each other for the same resolutions.



Note: Latency values vary slightly between runs due to system conditions (CPU load, thermal throttling, background processes). The values below represent one complete benchmark run. The key findings about relative speedups remain consistent across runs.

Accuracy values for both 320x320 and 640x640 can be seen from the chart. mAP values are calculated using COCO API.



6. Limitations and Future Work

What I didn't do:
- Only tested on **CPU**; GPU might give different results
- Batch size = 1; bigger batches could change things
- Just one model (YOLOv8n); don't know if this works for other architectures

What could be done next:
1. Quantization: INT8 could give another 2-4× speedup, but less accuracy
2. TensorRT: Try NVIDIA's optimized engine on GPU
3. Model Pruning: Remove parameters without decreasing accuracy much
4. Dynamic Batching: See how throughput changes with different batch sizes

7. Conclusion

This study built a solid benchmark framework for YOLOv8n across PyTorch and ONNX. Through debugging and problem-solving, I fixed initial export issues that caused 15.76 point accuracy loss, got it down to just 1.21 points at 640×640 resolution. My parity check system, with realistic tolerances, validates consistency while allowing expected numerical differences.

What I achieved:
1. Found and fixed **ONNX export problems** (**preprocessing** and **NMS** sync)
2. Measured accuracy-latency tradeoffs across four configs
3. Showed ONNX Runtime's 2.4× speedup at 640×640 with minimal accuracy cost (1.2%)
4. Showed ONNX Runtime's 3.9× speedup at 320×320 with small accuracy improvement
5. Built reproducible benchmark (10 runs × 500 images) with proper statistics