Deploying Real Time Object Detection Inference Systems

Yves Kreimerman, 2025

Abstract. Computer vision is a field in machine learning that aims to understand a give picture, identify what's going on and classify those into classes(patterns) it understands. There are couple of main fields in the computer vision world, those being:

1. **Classification** - The model classifies a whole picture into classes. It is possible for the model to classify the picture as multiple classes(Multi-Label Classification), or one class, being the most dominant(Multi-Class Classification)

2. **Object Detection** - The model omits a series of boxes around various object it identifies(based on the data it was trained), pointing to the coordinates where the object is found on the picture.

3. **Semantic Segmentation** - The model identifies pixels as part of a class, but does not distinct between different objects sharing the same class

4. **Instance Segmentation**- The model identifies pixels as part of a class, and has the distinction between objects. it is able to distinct two objects from the same class that are nearby.

1. Introduction

Live streaming video is a broad subject, of producing a live video feed which is constantly broadcasted from various sources, with consumers watching on the other end. Live streaming is usually split into couple of main categories, the main ones being:

Type	**Delay Range (Seconds)**	**Use**
WebRTC	0.5 - 2	Real Time (video calls, surveillance)
Broadcast Live	2 - 5	TV (News, Sports)
RTMP/HLS	5 - 30+	Live Media (Youtube, Twitch)

Table 1: Main categories of live streaming

Having Machine Learning fit model into the chain of live streaming can be difficult, as it requires us to fit under very slim time constraints.
With Real Time being our main subject, we will go over the constraints we should take into count, and how they project on our system design:

1. **Source Capture & Extraction of raw frame** - Getting raw frames from source, before any manipulations - 30ms

2. **Machine Learning Inference** - Processing, model inference, postprocessing

3. **Video encoding & Processing** - Processing frames into actual video, encoding to various formats(e.g. H264) - 20ms

4. **Network Transport** - Transporting the processed video to consumer(Over LAN/WAN) - 100ms-150ms

5. **Consumer Decoding & Display** - Displaying the video to the end consumer via streaming platform - 100ms-130ms

The numbers shown are for an ideal scenario, and in reality numbers are much higher. That requires us to fit under very low constraints, ideally around **15ms-75ms**, depending on video frame rate and network throughput.

3. Methodology

Having tight time constraints to take into consideration, we would have to design a low-latency, high throughput system, that can serve as many video sources as possible, while utilizing the most from the given resources.

3.1 Model Precision

With the goal of minimizing latency of inferance, and maximizing processing throughput, we need a way to optimize the given ML models and squeeze the best preformance out of those.
By default, machine learning models use FP32(Floating point 32) datatypes for weights and inputs/outputs by default, allowing higher precision for training, therefore maximizing accuracy.
When running inference on the same models, we don't necessarily need all that precision and can "compromise" on lower floating points, i.e. FP16. FP16 should give us much better preformance, while almost not compromising on accuracy.
We first want to test all model versions for accuracy, determine how lossy is the conversion of floating points and is compromising on floating points even worth it.
We will do so with a pre-made evaluation script, while models are evaluated on MS-COCO Val 2017 dataset, for consistency purposes(all models are trained on the same data too).
The goal is to demonstrate how one model compares to another, so the actual performance of the model(how well it performs on the data) is trivial.

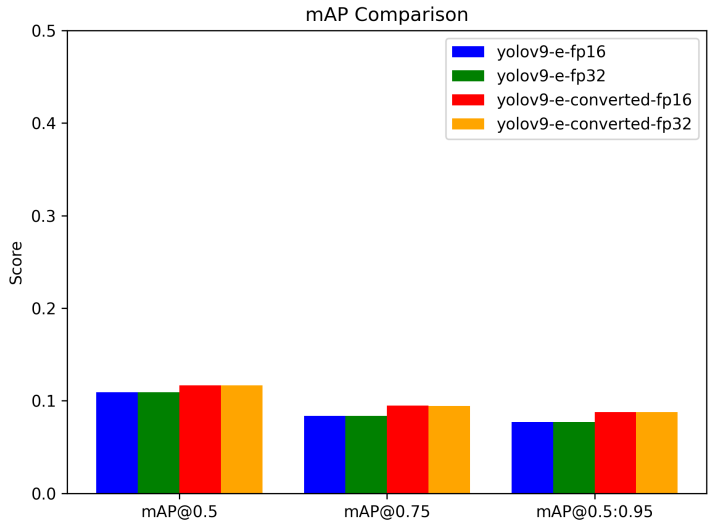

Figure 1: mAP (under different IOUs)

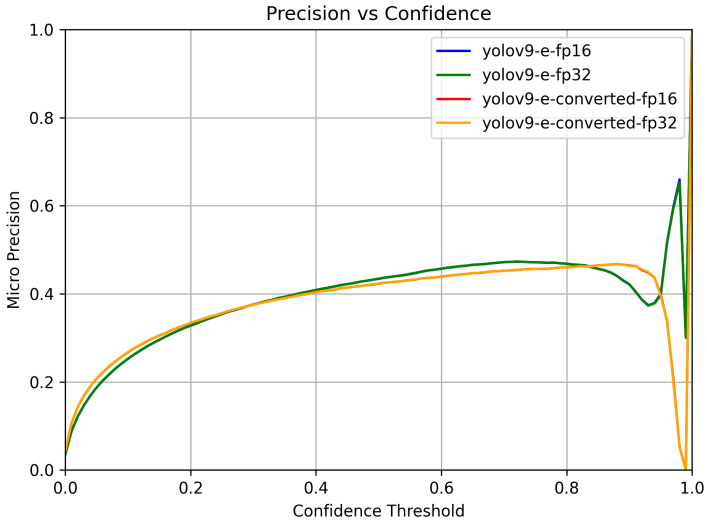

Figure 2: Precision - Confidence Curve

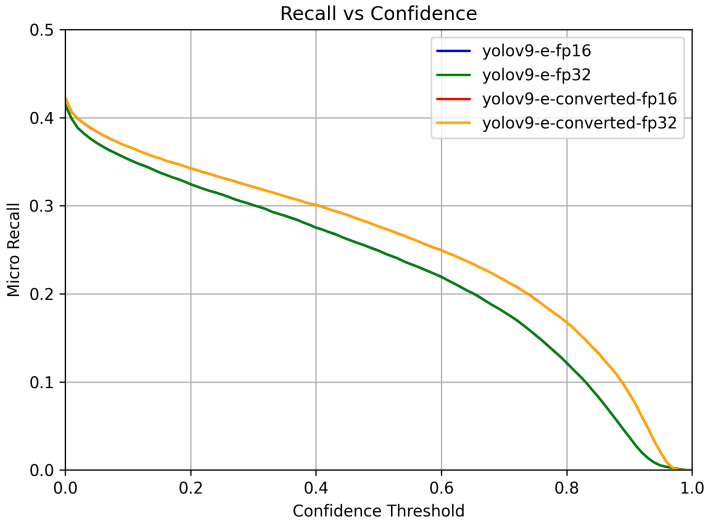

Figure 3: Recall - Confidence Curve

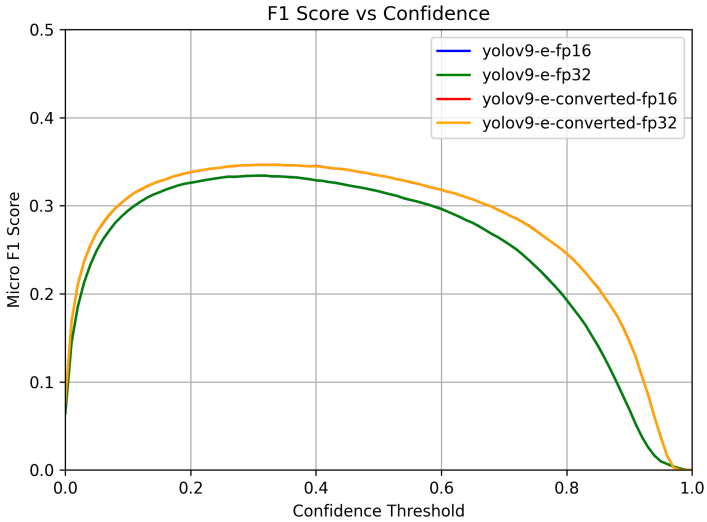

Figure 4: F1 Score - Confidence Curve

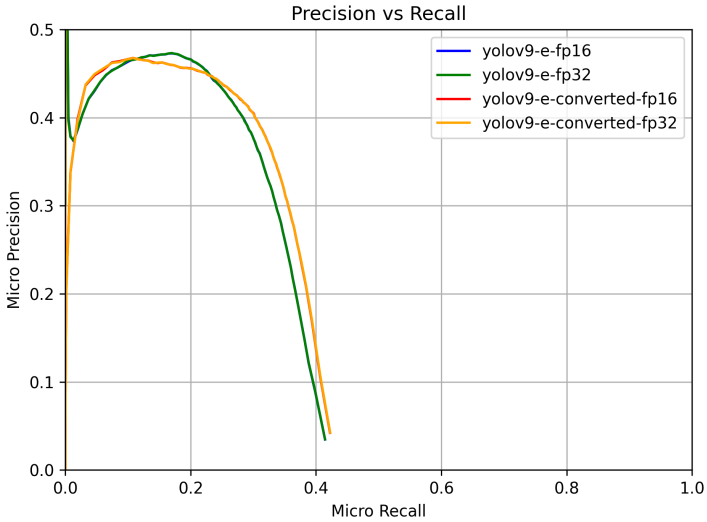

Figure 5: Precision - Recall Curve

While it may not be visible right away, we can see that FP16 and FP32 results for the same kind of model are overlapping in all curves. This means that they both reach identical results(with minor changes after the 4th floating digit).
With that said, **FP16 models will be our choice of deployment** as they allow us to have the same inference accuracy while significantly lowering inference time(higher throughput).

3.2 Model Performance

With the fact that FP16 models allow us to have same precision, while being much faster at inference time, we will be having all further tests made on FP16 models only.
Along with optimizing model internals(the actual neural network)(using efficient code, fp16 models), we need to have some tools to be able to boost the models to a higher level of performance, and **NVIDIA TensorRT** solves just that. NVIDIA TensorRT allows us to compile a model specific to our GPU hardware, therefore very efficient with memory management inside the GPU, giving us peak performance at almost no hard work. One of TensorRT's main functions is allowing batch-processing of inputs, therefore allowing us to get multiple results and process much more data in less time.
Coupled with NVIDIA TensorRT, we also have **NVIDIA Triton Server**, which is a platform used to serve those models and run efficient inference on. All models are served through HTTP/gRPC requests, and allow us to perform inference through those protocols. Triton Server comes hand to hand with TensorRT, leveraging all TensorRT advantages(batch-processing, multiple model instances running at the same time), therefore boosting our performance even more.
For the following experiment we have compiled our models with TensorRT, specifically with dynamic inputs(Unlocking the batch-processing features) and FP16(for lighter and faster models). We have tested the model throughput for different amounts of concurrent requests, to test how much we can squeeze out of our GPU.
Experiment ran with the following environment specifications:

- CUDA 12.9
- TensorRT 10.11.0
- Triton Server 25.06
- RTX 4070 GPU

Note - The model we deployed on Triton Server uses a specific config tailored to squeeze the best out of our model. It involves dynamic batching, special GPU settings and more. Will be provided alongside this document.

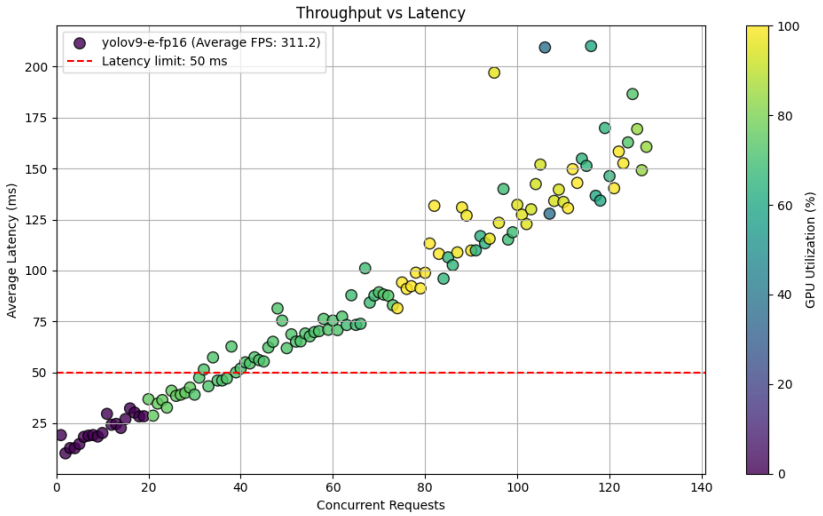

Figure 6: YOLOV9-e FP16 Throughput

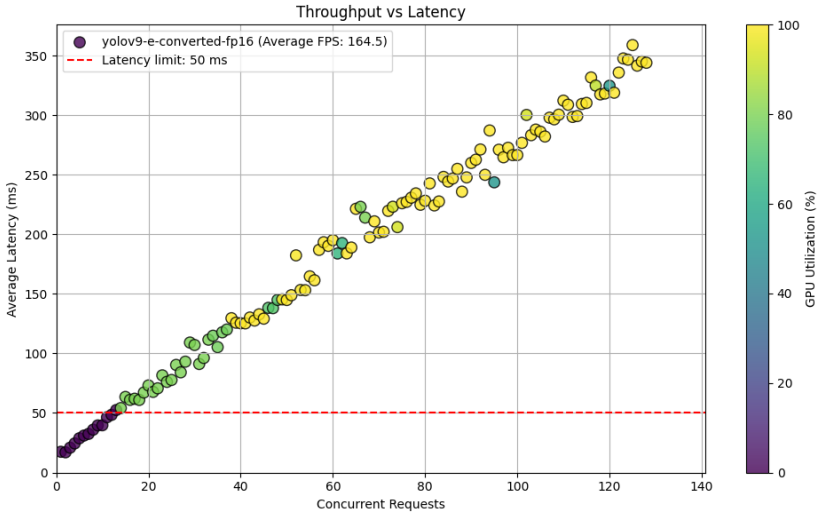

Figure 7: YOLOV9-e-converted FP16 Throughput

Each graph shows a model performance("stress test") under different amounts of concurrent requests, essentially simulating our "real-time video" scenario. Each dot represents a test conducted for X amount of concurrent requests at once(X-axis), with the average latency we got per request(Y-axis). The color indicates the GPU utilization at the different points, with a scale allowing us to measure how well our GPU is doing at every point. There is also an indication at the top left corner, showing us how much FPS our model was able to serve per second. With our real time video context, it will allow us to calculate how many video streams our model will be able to serve at once:

$$\text{Total Streams} = \frac{\text{Model FPS}}{\text{Video Stream FPS}}$$

Looking at our results, we can see that YOLOV9-e outperforms YOLOV9-e-converted by almost double, while maintaining much lower GPU utilization throughout the tests. This leads us to our final conclusion - **YOLOV9-e-fp16 will be our model of choice** for building the real-time video inference application.

3.1 Architecture overview

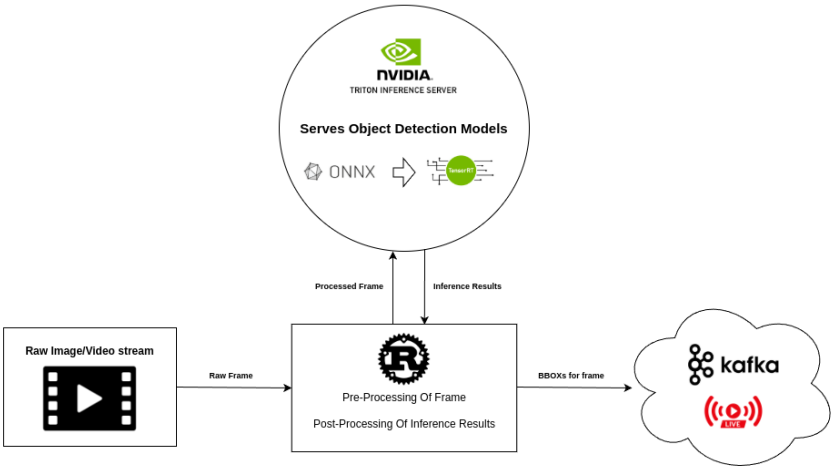

Figure 8: Architecture for a real time video inference system

Having our model pushed to it's limit is really great, but we also need a steady system to back it up. We must build a strong, low latency inference system, that doesn't create an overhead for pre-processing video input or post-processing model output, and allows us to serve inference results at near model time. for that we have two main options:

- **Python** - Has a very rich ML/processing tools such as pytorch, numpy, openCV, numba and much more, allowing us to run code in times close to C language. Having said that, it has many downsides, which we should take in mind:
  - Poor optimization - Python is not a low level programming language, therefore has many overhead components that hurt runtime(garbage collector, datatypes) and prevent us from getting instant response.
  - Poor Multi-Threading - Running multiple threads(for multiple video streams for example) is very limited, as Python's GIL(Global Interpreter Lock) prevents us to run more than one python bytecode at a given time. Even with multithread packages(i.g. threading), will have to wait in queue for GIL to finish - not ideal for real time scenarios.
- **Rust** - Has a growing set of ML tools, such as openCV, ndarray(Numpy-like) and more, providing sufficient tools in our case. Rust has multiple upsides in case of real time inference:
  - C-like performance - Rust is very efficient, having no overhead components(No garbage collector) and other datatypes that can hurt runtime, rust allows us to write code in high level syntax with low level performance. Great for achieving real-time computations.
  - Real multi-threading - Rust allows us run our program in real parallel, on multiple cores at a time natively. This allows us to serve multiple streams seperately without them depending on each other to finish running.

  With that said, rust has a rather steep learning curve, and it takes time for an average programmer to be able to write production grade code.

With that said, **Rust will be our language of choice**, having its capabilities overcome Python's, and with the idea of having our system be stable, fast and reliable over time.