Journal of COMPUTER AND FORENSIC SCIENCES

CFS 2025, Vol. 4, Issue 1, pp. 3-19
https://doi.org/10.5937/jcfs4-58032
Original research paper

# Comparative Analysis of Machine Learning Models for Real-time Object Detection

**Merlin Wittenhagen**[1*]

[1] Student, School of Engineering and Architecture, SRH University, Heidelberg, Germany

* Corresponding author: merlinwittenhagen@icloud.com

Received: April 6, 2025 • Accepted: May 6, 2025 • Published: June 20, 2025

**Abstract:** Object detection is a fundamental task in computer vision with applications ranging from autonomous driving to industrial automation and medical imaging. This report presents a comparative analysis of six well-known object detection models: three small models for edge computing and three large models likely more suited for usage on high-performance systems. The models YOLOv10-Nano, MobileNetV3-SSDLite, EfficientDet-D0, Faster R-CNN, YOLOv10-Large, and DETR were evaluated and compared based on their performance in terms of inference speed, accuracy, and computational efficiency. The evaluation is conducted through both literature-based benchmarks and empirical tests on two different systems: an Apple Silicon M1 Pro-based system and an NVIDIA RTX 3080Ti-powered computer. Results show that YOLOv10 models consistently outperform the other models in real-time object detection as well as achieving superior accuracy in general while maintaining significantly lower inference times. The analysis further highlights compatibility issues with certain hardware, particularly focusing on PyTorch's MPS backend on Apple Silicon, which leads to serious performance drops in some models. The findings highlight the importance of choosing the right model and appropriate hardware for specific application scenarios.

**Keywords:** object detection; model benchmarking; inference efficiency, hardware-aware evaluation.

## 1. INTRODUCTION

Object detection has become one of the most important tasks in computer vision, with applications ranging from autonomous driving to medical imaging and automation in industrial facilities. While some use cases allow for complex object detection models with high computational demands, others are constrained by limited system capabilities (edge computing or mobile devices), requiring more efficient solutions.

Over the last years, multiple object detection models have been developed and evolved, each having unique advantages and tradeoffs. These can be speed, accuracy, and computational cost. Besides common CNN-based architectures like YOLO and Faster R-CNN, which have been the most obvious choice for most problems in recent years, new Transformer-based models such as DETR (DEtection TRansformer) were developed to improve object detection in complex scenes.

In the scope of this comparative analysis, six well-known object detection models – YOLOv10nano, MobileNetV3-SSD Lite, EfficientDet-D0, Faster R-CNN, YOLOv8-large, and DETR – will be evaluated to determine their suitability for various applications. This will be done by comparing performance metrics like inference speed, accuracy, computational efficiency, and their applicability in edge or high-performance environments.

The primary aim of this work is to give insights into how these models perform under various constraints, such as limited computational power compared to high-performance setups. In addition to that, it will be highlighted whether Transformer-based models like DETR can compete with or surpass more traditional CNN-based approaches. The paper aims to provide information on which model is best for certain use cases, keeping metrics like accuracy, computational demand, and speed in mind.

## 2. MODEL OVERVIEW

### 2.1. YOLO

For this work, two sub models of the YOLO-model (You Only Look Once) were used. In fact, the chosen models are based on the tenth generation of YOLO, which is the result of improvements in performance over the years. The following subsection will give an insight into the general function of earlier YOLO models as well as YOLOv10.

#### *2.1.1. YOLO – Basic Functionality*

As a single-stage model, YOLO processes the entire image in one step, simultaneously predicting class labels and bounding boxes. This sets it apart from other models that use a two-stage approach (like Faster R-CNN, which is explained later), which works by proposing regions and then classifying an object separately [1, 42].

The model divides an input image into a grid of S x S dimensions, where each grid cell is detecting if an object lies within that cell. Each cell then predicts a set number of bounding boxes, which are described by coordinates (x, y, w, h), representing the position of the bounding box as well as its dimensions relative to the image size [2]. Those bounding boxes are then getting assigned a confidence score, which combines the probability of an object being present and the accuracy of the bounding box. This accuracy is measured by Intersection over Union (IoU). In addition to that, the network also predicts class probabilities to identify the detected object [2].

After calculating all confidence scores and bounding boxes, a threshold is applied to filter out predictions with low confidence. In previous YOLO versions (v1-v8), the Non-Maximum Suppression (NMS) was used to remove needless bounding boxes by only selecting the ones with the highest confidence scores while preventing keeping strongly overlapping boxes based on the Intersection over Union metric [3, 42].

With the introduction of YOLOv10, NMS was dropped to further increase computational efficiency. This was done by optimizing processes during training, which allows the model to learn to predict only the most accurate bounding boxes directly [4]. For final detection

(classification and locating the bounding boxes), three different CNNs are used. They are optimized to detect large, medium, and small objects in an image [6]. The general architecture of YOLOv10 is shown in Figure 1.

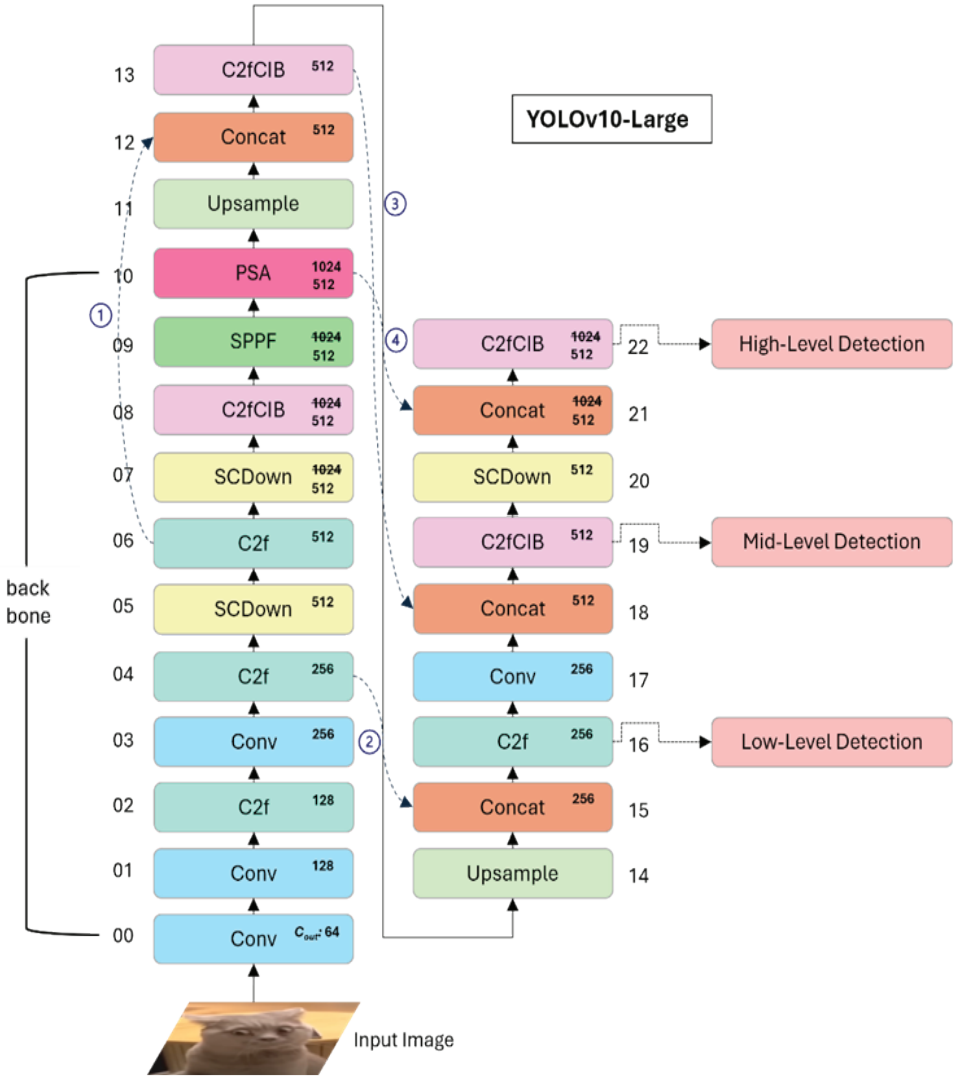


**Figure 1.** *Architecture of YOLOv10. Adapted from [7].*

### 2.2. Faster R-CNN – Basic Functionality

Other than YOLO, Faster R-CNN (FRCNN) is a two-stage object detection model, which separates the process of localizing objects and classifying them. To achieve this, FRCNN utilizes a Region Proposal Network (RPN), which is responsible for accurately predicting the regions where objects are likely to be present. These region proposals are then used to generate bounding boxes. The second stage of the pipeline classifies the object within the identified region and refines the bounding box for more precise localization [8]. In detail,

the model processes an input image through a CNN backbone (in this case, ResNet-101) to extract feature maps. These feature maps are passed into the RPN, which generates region proposals that likely contain objects. Similar to earlier YOLO models, the RPN assigns anchor boxes to different locations in the image and predicts whether each anchor contains an object or not. The bounding boxes are further refined to better frame the detected object [8].

Once the RPN generates a set of region proposals, they undergo Region of Interest (RoI) Pooling, which ensures that all proposals are transformed into a fixed-size feature representation. These feature maps serve as the input for a fully connected layer, which performs object classification and further adjusts the bounding box for more precise localization [9].

After object classification and bounding box refinement, a confidence threshold is applied to filter out low-confidence predictions. To avoid multiple overlapping bounding boxes for the same object, Non-Maximum Suppression (NMS), as described in the previous chapter, is applied [8]. Figure 2 shows a rough depiction of the Faster R-CNN model.

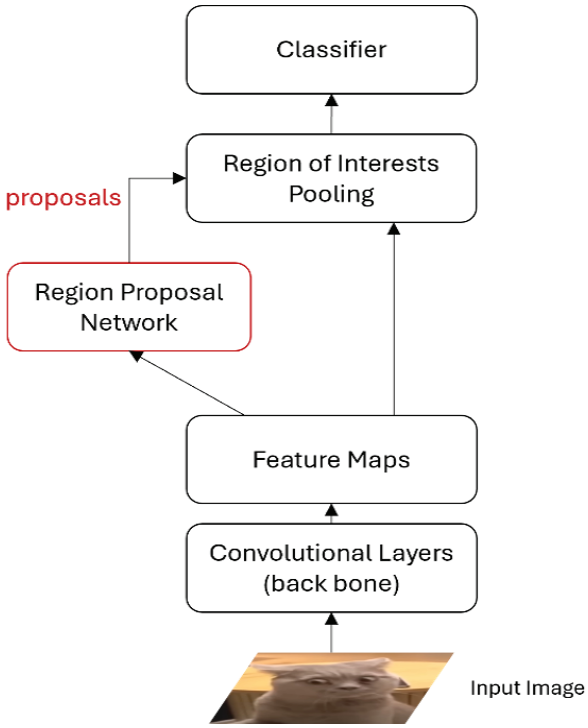


**Figure 2.** *Basic Architecture of Faster R-CNN.* Adjusted from [10].

### 2.3. MobileNetV3 – Basic Functionality

The MobileNetV3 model that was chosen for this work is the SSDlite320 MobileNetV3-Large. It is a lightweight object detection architecture that combines the MobileNetV-Large backbone with the so-called Single Shot MultiBox Detector (SSD) framework. This setup was designed to deliver high-performance object detection suitable for mobile

and embedded systems. MobileNetV3 is a convolutional neural network architecture that is optimized for mobile applications. It uses different techniques to balance accuracy and latency effectively. The "Large" variant is adapted for scenarios requiring higher accuracy while remaining computationally efficient [11]. The SSD makes the combined model a single-stage object detector like YOLO. It does not rely on a separate RPN to propose regions. Instead, the SSD predicts object classes and bounding boxes directly from the feature maps extracted from the MobileNetV3 CNN [12, 13].

### 2.4. EfficientDet-D0 – Basic Functionality

EfficientDet-D0 is also an efficient, lightweight object detection model. Like other models of its kind, it aims to optimize the trade-off between accuracy and computational efficiency to make it suitable for mobile applications. Unlike traditional object detection architectures that scale only in depth or width, EfficientDet-D0 uses compound scaling, which scales the backbone network, the feature fusion layers, and the prediction heads to improve overall efficiency [14]. EfficientDet combines a classic CNN architecture with a Bidirectional Feature Pyramid Network (BiFPN) to efficiently fuse multi-scale features. This way the feature extraction as well as the prediction heads are optimized, which provides a balanced trade-off between computational cost and accuracy. [14] Figure 3 shows a simplified version of the EfficientDet network, which utilizes the EfficientNet backbone (CNN) and extracts features from multiple layers to feed them into the BiFPN. The output of the BiFPN is then used as input for two fully connected layers for class prediction and box prediction [14].

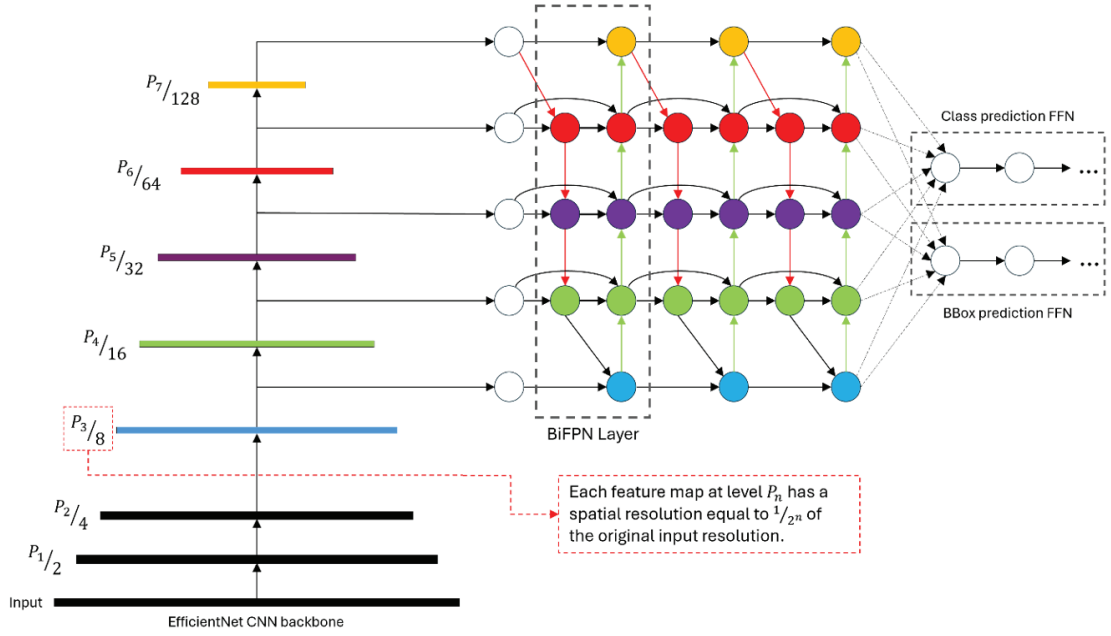


**Figure 3.** *Architecture of EffiecientDet-D0. Adjusted from [15].*

### 2.5. DETR – Basic Functionality

Contrary to the models described before, the DETR (DEtection TRansformer), as the name states, uses a Transformer architecture to interpret features extracted by a CNN. Like Faster R-CNN, the backbone used to extract features from input images is typically ResNet-50 or ResNet-101. The generated feature maps are used as input for the Transformer Encoder. The Transformer Encoder processes the extracted feature maps and learns long-range dependencies between different regions of the image. Unlike other models that generate a fixed or dynamic number of regional proposals or divide the image into a grid, like YOLO does, DETR uses a fixed number of learnable object queries. Each of those queries represents a potential object in the image, even if some queries remain unused since no object is detected. Those queries are in general tensors with n dimensions, which are initialized randomly and then are optimized during the training phase of the model. There could be, for example, 100 query tensors, which all search for a different object in the feature map (the amount of object queries determines the number of detectable objects). These queries are processed by the Transformer Decoder, which relates them to the encoded feature map. Since each query focuses on a specific region in the image, the model is able to determine the location of an object and its class. The final predictions are technically done by using two separate prediction heads in the form of two different fully connected layers to determine the class and the bounding box of the object [16, 17, 5].

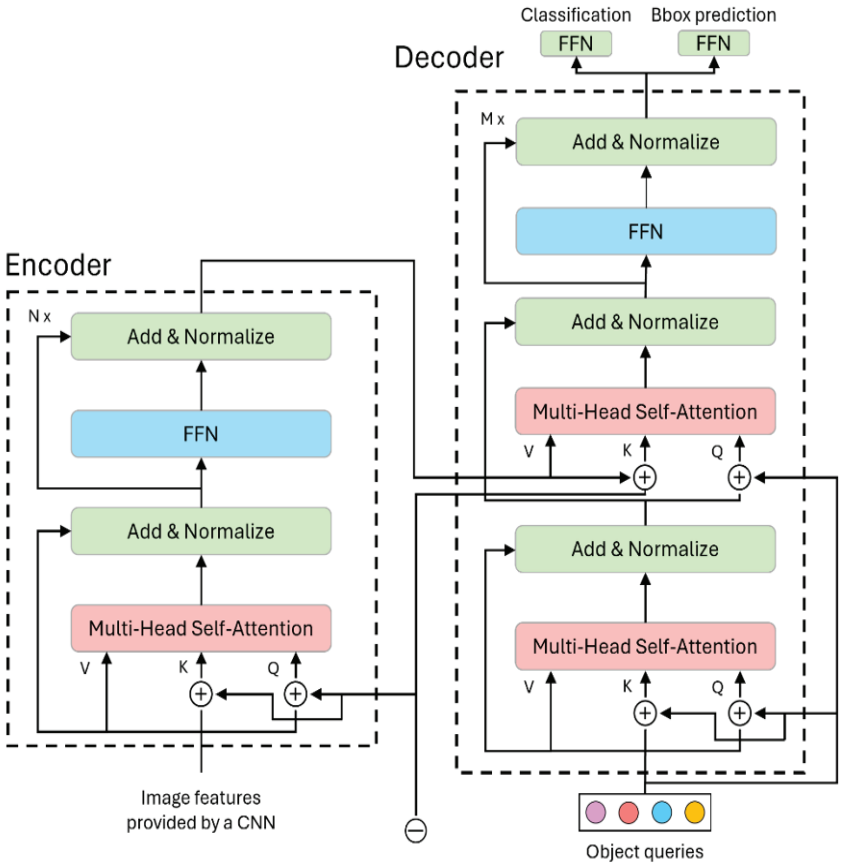


**Figure 4.** *Encoder-Decoder Architecture of DETR. Adapted from [18].*

# 3. METHODS

## 3.1. Comparison of the Models Based on Existing Evaluation Data

The first comparison was made by choosing three low-size and high-speed as well as three high-size and relatively low-speed models, which still can be used for real-time processing. Chosen were popular models that are widely used in the realm of object detection. The six models that were described in the previous chapter were then compared based on their number of parameters, their computational cost in FLOPS (Floating Point Operations per Second), the model size in terms of storage demand, the inference time in milliseconds, and the accuracy based on the mAP@0.5:0.95 (Mean Average Precision across the defined range of Intersection over Union, detailed explanation in the chapter "Explanation of the Mean Average Precision"). The models were compared overall and with respect to their respective aim (separated by small and large models).

The information about the researched models was taken from different sources, since there is no comparison between all of them. This means that the evaluation results for each model were probably obtained on a different system. To still provide useful insight, it was made sure that all models were trained and evaluated based on the 2017 COCO dataset [19].

## 3.2. Testing the Models on Different Systems

To give an insight into the performance of the tested models when run on systems that are more likely to be found in a real-word application, the performance in terms of inference time was measured on two different devices. The computers used were chosen because of their availability while conducting this work.

The first system was an Apple MacBook Pro with an ARM64-based M1 Pro chip. This chip consists of 8 performance and 2 efficiency CPU cores working at 2.06-3.22 GHz as well as 16 GPU cores. The system uses a unified memory of 16 GB [20]. In addition to that, the chip inherits a 16-core Apple Neural Engine, which is technically an NPU (Neural Processing Unit), which is optimized for convolutions and matrix multiplications (tensor operations). However, the latter was not used since the tested models were all based on the PyTorch framework, which does not support the ANE (Apple Neural Engine). To still run the models efficiently, Apple's MPS (Metal Performance Shaders) API was used to run them on the GPU. PyTorch contained MPS backend support since version 1.12 while still being unable to offer the same functionality as other APIs, which results in poorer performance [21].

As a second test system, a more "standard" tower PC was used. It operates on the more common 64-bit x64-CPU-architecture. The CPU was an Intel Core i7 12700k with a total of 12 cores (8 performance, 4 efficiency) running at 3.6-4.9 and 2.7-3.8 GHz [22]. The system memory consisted of 32 GB DDR4 at 4000 MHz. These specifications were only included in this description to be precise and transparent. Their influence would be marginal since the models were run exclusively on the GPU. The used GPU was an NVIDIA

GeForce RTX 3080Ti with 12 GB of video memory and 10240 CUDA Cores, which makes it very suitable for AI and ML related tensor/matrix calculations [23].

The two systems differ significantly in their intended application purpose, as one is optimized for low energy consumption mobile use and the other is set up for energy-intensive, high-performance operation. This comparison was done to show how insightful commonly available information on the performance of object detection models really is, since most systems are not able to match the performance of benchmark systems. This comparison aimed to show how much the inference times actually differ from the proposed benchmark values generated with high-end systems if less powerful hardware is used. In addition to that, this was intended to also highlight possible compatibility issues with certain hardware, which could cause an object detection model to become unusable on a device.

## 3.3. Description of the Mean Average Precision

The mean Average Precision as introduced in the COCO benchmark [43] was used to evaluate the performance of the models in terms of box prediction and object-classification accuracy. This chapter will describe how the mAP is calculated and which subtypes are commonly used to evaluate object-detection models. In general, the mAP is dependent on the Intersection over Unit, which is then used to calculate the share of correctly detected boxes based on precision and recall. It is used as a metric to describe the robustness of an object-detection model.

### *3.3.1. Intersection Over Unit (IoU)*

The IoU describes the accuracy of a predicted bounding box relative to the actual box (Ground Truth). This is calculated by dividing the area of overlap between the predicted and the actual box by the area of their union [24]. This can be mathematically described as:

$$IoU = \frac{|A \cap B|}{|A \cup B|} = \frac{|I|}{|U|}$$

1where:

- *IoU* is the intersection over unit,
- *A* is the area of the first box,
- *B* is the area of the second box.

The expected result is a number between zero and one, with higher values showing a higher overlap and therefore a better result. A graphical example can be seen in Figure 5. Precision and Recall are used to describe the portion of correctly predicted boxes relative to the total number of predictions and to describe the portion of correctly predicted boxes relative to the total number of existing boxes.

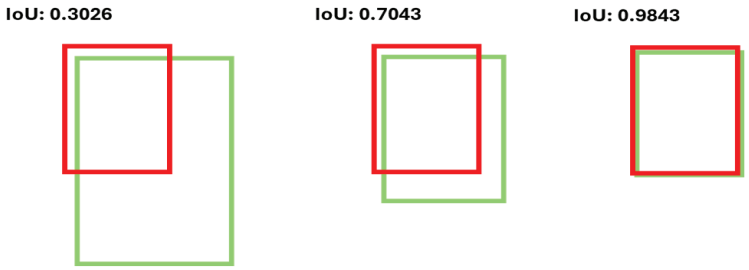


**Figure 5.** *Example – IoU Bounding Boxes. Adjusted from [25].*

### *3.3.2. Calculation of the mAP*

As mentioned in the introduction to this chapter, there are multiple versions of the mAP. In general, the AP (Average Precision) is the area under the precision-recall curve. Earlier object detection models were evaluated using the mAP@0.5 metric, where a bounding box is rated as true positive if the IoU is equal to or larger than the threshold of 0.5. The downside of a fixed threshold of 0.5 is that it evaluates rather poor predictions as true positives, even if the bounding box is relatively far off. To get a more robust way of evaluating the performance of an object-detection model, the mAP@0.5:0.95 is used. It describes the mean Average Precision over a range of 10 thresholds between 0.5 and 0.95. This shows not only the model's capability of detecting an object, but it also gives insight into the ability to find accurate bounding boxes [26].

The final equations used to calculate the mAP@0.5 and the mAP@0.5:0.95 are shown in Equation 2 and Equation 3.

To make the comparison between the six selected models, only the mAP@0.5:95 was chosen as the accuracy metric since it provides the more meaningful insight. This method is currently the most used for the evaluation of object-detection models, often as a benchmark metric after training and evaluation with the COCO data set [27]. Therefore, the mAP@0.5 was not used as a metric for the comparison conducted for this paper.

$$mAP@0.5 = \frac{1}{N}\sum_{c=1}^{N} AP_c@0.5$$

2where:

- $mAP@0.5$ is the mean Average Precision at an $IoU$ threshold of 0.5,
- $N$ is the total number of object classes,
- $AP_c@0.5$ is the Average Precision for class $c$ at an $IoU$ threshold of 0.5.

$$\text{mAP@0.5:0.95} = \frac{1}{N}\sum_{c=1}^{N}\left(\frac{1}{10}\sum_{k=1}^{10}\text{AP}_c(IoU_k)\right)$$

3where:

- *mAP*@0.5:0.95 is the mean Average Precision, averaged over 10 *IoU* thresholds from 0.5 to 0.95 in steps of 0.05,
- *N* is the total number of object classes,
- $AP_c(IoU_k)$ is the Average Precision for class *c* at a specific *IoU* threshold $IoU_k$,
- *k* represents the index for the *IoU* thresholds, where $IoU_k \in \{0.5, 0.55, 0.6, \ldots, 0.95\}$,
- $\frac{1}{10}$ ensures the average over the 10 *IoU* thresholds.

## 4. RESULTS

### 4.1. Comparison Results Based on Available Evaluation Data

Comparing the models based on already available data first of all shows the clear differences in light models for edge computing and heavy models for high-performance applications. One of the key factors is the size of the model in terms of parameters and total file size. The small models range from 2.3 to 3.9 million parameters, while the large models have around ten times as many. The small models, namely the YOLOv10-Nano, the MobileNetV3-SSDLite, and the EfficientDet-D0, are showing the typical trade-off between low computational demand, low inference, and reasonable accuracy. YOLOv10-Nano has the lowest inference time of all models, with the highest accuracy of the smaller models based on the mAP@0.5:0.95, while requiring more FLOPS than other fast models. This seems counterintuitive since more FLOPS mean more calculations, which should increase the inference time. In the specific case of YOLOv10, this seems not to be an issue. This is because YOLOv10's architecture is highly optimized for parallel calculations on modern hardware platforms, which makes it faster than other models with less computational demand in terms of FLOPS [2]. For the large models where Faster R-CNN, YOLOv10-Large, and DETR were compared, it is also noticeable that YOLOv10 outperforms its competitors in terms of speed, accuracy, and memory demand. In this particular case, even the computational demand is with around 120 GFLOPS lower than that of Faster R-CNN and DETR, with 134 and 180 GFLOPS. The YOLOv10-Large model is by far the most accurate of all six models, outperforming the second best (DETR) by roughly ten percent points (53.3 % vs 42 %). Even the YOLOv10-Nano model outperforms every compared model excluding the DETR and YOLOv10-Large at around 39.5 % accuracy.

Based on the provided data, YOLOv10-Nano is the best small model for edge computing and mobile applications. It is by far the fastest model, provides the highest accuracy, and has the smallest memory demand, which makes it more suitable for devices with limited memory.

The result for the large models is similar. YOLOv10-Large outperforms Faster R-CNN and DETR in every point. It is faster, more accurate and needs less memory as well as less computational power. In terms of real-time detection capabilities, it even competes with the small models, having an inference of around 7.28 ms.

To conclude the data interpretation, it is apparent that the YOLOv10 models are at the moment the best choice to solve object detection problems, especially when it comes to

real-time detection. Both the Nano and the Large model outperform their competitors in almost every metric, which leads to the stated conclusion.

The comparison data in Table 1 was obtained from the following sources: [4, 28, 29, 30, 31, 32, 33, and 34]. The table was also divided into small models and large models, as can be observed.

**Table 1.** *Comparison of Object Detection Models.*

| Model | Params (Million) | FLOPs (GFLOPs) | Size (MB) | Inference Time (ms) | Accuracy (mAP@0.5:0.95) |
|---|---|---|---|---|---|
| YOLOv10-Nano | 2.3 | 6.7 | 4 | 1.56 | 39.5 % |
| MobileNetV3-SSDLite | 3.9 | 0.55 | 5 | 3.5 | 39 % |
| EfficientDet-D0 | 3.9 | 2.54 | 16 | 3.92 | 34.6 % |
| Faster R-CNN (ResNet-101) | 41.8 | 134.4 | 160 | 54 | 37 % |
| YOLOv10-Large | 24.4 | 120.3 | 100 | 7.28 | 53.3 % |
| DETR (ResNet-50) | 41.6 | 180 | 159 | 36 | 42 % |

*Rounded values*

### 4.2. Performance Results on Different Systems

Running the models on the two selected systems resulted in the inference times displayed in Table 2. Like before, the table is split into small and large models. Looking at the M1 Pro results for the small models – YOLOv10-Nano, MobileNetV3-SSDLite, EfficientDet-D0 – a very large discrepancy in inference times becomes apparent. The YOLO-Nano model is by far the fastest of the three, with an average inference time of around 17 ms, while the EfficientDet-D0 model comes second with an average inference of 70 ms. The third small model, however, did not perform well at all, processing the images with an average inference of roughly 760 ms, which is significantly worse than the other two models.

Looking at the M1 Pro results for the large models – Faster R-CNN, YOLOv10-Large, DETR – the Large YOLO model also shows the best average inference, with around 37 ms. DETR performed well for a large model at 140 ms. Faster R-CNN, however, performed so significantly badly that it can be deemed unusable on this platform. It has an average inference of circa 100,000 ms, which is extremely high. The reason for this behavior will be discussed in detail, since it highlights a significant flaw of using PyTorch models on some devices.

Coming to the results generated with the RTX 3080Ti and first looking at the performance of the small models, each can be evaluated as working properly. The YOLOv10-Nano – like on the M1 Pro – performed the best with a noticeable lead over the other two models. With an average inference of ~3.5 ms, it runs more than seven times faster than MobileNet with ~25 ms and almost six times faster than EfficientDet with ~20 ms inference.

The results for the large models run on the 3080Ti show that YOLOv10-Large also performed the best in terms of speed. It only needed roughly 7.3 ms to process an image, while

Faster R-CNN needed 36 ms and DETR 26.5 ms on average. This time, Faster R-CNN worked as intended and performed rather well compared to the test on the M1 Pro.

Concluding the results of the system comparison, it is possible to say that in terms of speed, both YOLO-models outperformed all other four models regardless of being compared to the small or large ones. This is especially interesting since YOLOv10-Large not only outperforms MobileNet and EfficientDet at raw speed, it also tremendously outperforms them in terms of accuracy, as can be seen in Table 1. The results reinforce the statement that, out of the compared models, YOLOv10 currently seems to be the best choice for small and large models in terms of speed or, more precisely, for real-time object detection.

**Table 2.** *Average Inference Times on M1 Pro and Nvidia RTX 3080Ti.*

| Model | Apple Silicon M1 Pro | Nvidia RTX 3080Ti |
|---|---|---|
| YOLOv10-Nano | 17 | 3.5 |
| MobileNetV3-SSDLite | 760 | 25 |
| EfficientDet-D0 | 70 | 20 |
| Faster R-CNN (ResNet-101) | 100000 | 36 |
| YOLOv10-Large | 37 | 7.3 |
| DETR (ResNet-50) | 140 | 26.5 |

*Average values in milliseconds*

## 5. DISCUSSION

### 5.1. Interpretation of the Results

If it comes to real-time object detection, out of the tested models there is currently no better choice than YOLOv10+ (v10 or newer). This is due to its described lightweight architecture with vast optimizations to allow for fast calculations resulting in very low inference times while providing benchmark setting accuracy.

Looking a bit further and taking more models into account which were not compared in this paper, depending on the task there are serious competitors for YOLOv10 and newer versions. For real-time detection one alternative would be RTDETR (Real-Time DEtection TRansformer) which offers comparable performance. It could be further researched, if this model even has some distinct advantages over YOLO [35].

To not give the impression that YOLOv10+ is always the best choice, it does not set the bar when it comes to high-precision-detection models and zero-shot object detection models. In the realm of high precision models YOLOv10+ can be used but there are better alternatives, namely Co-DETR (Check out DETR) and DETA (Detection Transformers with Assignment) [36]. Both models provide an mAP@0.5:0.95 higher than 62 [36]. Regarding zero-shot object detection (detection objects without being explicitly trained on those classes) there are significantly better models than YOLOv10+. If such a problem must be solved, Grounding Dino 1.5 Pro or OWLv2 should be taken as possible alternatives [36]. As always, it should be researched which model would provide the best performance in terms of accuracy and speed for a specific task.

### 5.2. Compatibility and Performance Issues

Looking at the Results chapter, and especially at the results for the average inference times of the M1 Pro tests, the bad results for the MobileNet and Faster R-CNN tests should raise some questions. The first should be why the inference times are significantly higher than expected and why it only happened on the M1 Pro and not on the RTX 3080Ti.

As already mentioned before, there is MPS backend support in PyTorch. Unfortunately, there are still some undisclosed issues where some operations are not yet implemented using MPS. If this happens and the CPU fallback (utilizing the CPU instead of the GPU) does not work, it could result in such high inference times as observed in the case of Faster R-CNN. These problems are widely reported and will probably be solved in the future [37, 21]. One possible alternative would be to search for TensorFlow-based models, as the MPS backend support of TF is more refined than that of PyTorch since it is not in the beta phase anymore [21, 38].

Issues like this are less likely to occur on NVIDIA GPUs, as most machine learning frameworks are built around NVIDIA's CUDA (Compute Unified Device Architecture) API [39, 40]. Deploying AI models in Edge Computing across different hardware and software ecosystems can impose new hurdles that need to be recognized and overcome. Model size, computational demand, and format can be some of them, especially when running AI models on not optimized platforms like Raspberry Pi, Arduino, or STM32, since they do not have hardware acceleration or suitable processors at all. Luckily there are some specially developed Edge Computing solutions like NVIDIA Jetson, which, for example, supports CUDA, PyTorch and is optimized for deep learning, using special Tensor-Core GPUs or ARM-based chips [41]. Finally, it is always necessary to choose the right model for the right task and to make sure that the system used does support such a model, as the conducted tests showed.

## 6. CONCLUSION

The comparison showed that out of the selected models, YOLOv10 performed the best with regard to every metric. That goes for the Nano version in the realm of small models and for the large version regarding the other tested large models. Additional to this, an inference test and comparison showed how high the inference is likely to be if the models are run on more standard devices and not on high-end benchmark systems. It was proved that some of the models provided very satisfying performance in terms of accuracy and speed on a low-energy-consumption laptop and a relatively high-end computer. If run on powerful hardware, all tested models could be used for real time-object detection with acceptable inference times or frames per second. If run on a lower performance system, only the YOLO models and – with significantly less performance – the EfficientDet-D0 could be used for real-time detection. The issues regarding the use of PyTorch with the Apple MPS API made it obvious that not every model is suited for every device, even if the technical specifications match on the first glance.

To finish this paper, it should be mentioned that object detection models are still evolving fast, and the insights gained in the scope of this work might be obsolete in just a matter

of time. Also worth mentioning is the diverse use of different approaches to solve quite similar problems, which leaves room for improvement and new ideas.

## FUNDING
This research received no external funding.

## INSTITUTIONAL REVIEW BOARD STATEMENT
Not applicable.

## INFORMED CONSENT STATEMENT
Not applicable.

## CONFLICTS OF INTEREST
The author declares no conflict of interest.

## REFERENCES

[1] R. Kundu, "YOLO: Algorithm for object detection explained [+examples]," V7 Labs Blog, 2023.

[2] DataCamp, "YOLO object detection explained: A beginner's guide," DataCamp Blog, 2024.

[3] Ultralytics, "Nicht-maximum-unterdru¨ckung (nms)," Ultralytics Glossar, 2024.

[4] Ultralytics, "Yolov10: End-to-end-objekterkennung in echtzeit," Ultralytics YOLO Docs, 2024.

[5] H. Face, "Deformable DETR: Deformable transformers for end-to-end object detection." https://huggingface.co/docs/transformers/en/model_doc/deformable_detr, 2025. Accessed: 2025-03-11.

[6] A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, and G. Ding, "YOLOv10: Real-time end-to-end object detection." https://github.com/THU-MIG/yolov10/blob/main/ultralytics/nn/modules/head.py, 2024. Accessed: 2025-03-11.

[7] d4r6j, "YOLOv10: Model review." https://velog.io/@d4r6j/YOLOv10-1.-Model-Review, 2024. Accessed: 2025-03-11.

[8] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," arXiv preprint arXiv:1506.01497, 2015.

[9] R. B. Girshick, "Fast R-CNN," in Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 1440–1448, 2015.

[10] R. User, "Network structure diagram of Faster R-CNN." https://www.researchgate.net/ fig-ure/Network-structure-diagram-of-Faster-R-CNN-Faster-R-CNN-is-mainly-divided-into-the_ fig1_341871095. Accessed: 2025-03-07.

[11] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for MobileNetV3," arXiv preprint arXiv:1905.02244, 2019.

[12] V. Vryniotis, "Everything you need to know about Torchvision's SSDlite implementation." https://pytorch.org/blog/torchvision-ssdlite-implementation/, 2021. Accessed: 2025-03-07.

[13] P. Team, "SSDlite implementation in Torchvision." https://github.com/pytorch/vision/blob/main/torchvision/models/detection/ssdlite.py, 2025. Accessed: 2025-03-07.

[14] M. Tan, R. Pang, and Q. V. Le, "EfficientDeT: Scalable and efficient object detection," in Proceed-ings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 10781–10790, 2020.

[15] D. Buongiorno, D. Caramia, L. D. Ruscio, and A. Brunetti, "EfficientDet-D0 architecture: Effi-cientNet-B0 as backbone network with multiple BiFPN layers." https://www.researchgate.net/figure/EfficientDet-D0-architecture-EfficientNet-B0-34-is-the-backbone-network-multiple_ fig2_365439360, 2022. Accessed: 2025-03-07.

[16] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in European Conference on Computer Vision (ECCV), pp. 213–229, 2020.

[17] H. Face, "DETR: End-to-end object detection with transformers." https://huggingface.co/ docs/transformers/en/model_doc/detr, 2025. Accessed: 2025-03-11.

[18] G. Boesch, "DETR: End-to-end object detection with transformers." https://viso.ai/ deep-learning/detr-end-to-end-object-detection-with-transformers/, 2024. Accessed: 2025-03-11.

[19] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dol-lár, and C. L. Zitnick, "Microsoft COCO: Common objects in context." https: //cocodataset.org/#download, 2014. Accessed: 2025-03-07.

[20] Notebookcheck, "Apple M1 Pro prozessor – benchmarks und specs," 2021.

[21] A. Inc., "Accelerated PyTorch training on mac." https://developer.apple.com/metal/pytorch/, 2025. Accessed: 2025-03-10.

[22] I. Corporation, "Intel® core™ i7-12700k prozessor (25 mb cache, bis zu 5,00 ghz) spezifikationen." https://www.intel.de/content/www/de/de/products/sku/134594/intel-core-i712700k-processor-25m-cache-up-to-5-00-ghz/specifications.html, 2021. Zugegriffen: 2025-03-10.

[23] NVIDIA, "GeForce RTX 3080 and RTX 3080 Ti graphics cards," 2024.

[24] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," Proceedings of the IEEE/CVF Con-ference on Computer Vision and Pattern Recognition (CVPR), pp. 658–666, 2019.

[25] A. Rosebrock, "Intersection over union (IoU) for object detection." https://pyimagesearch. com/2016/11/07/intersection-over-union-iou-for-object-detection/, 2016. Accessed: 2025-03-10.

[26] D. Shah, "Mean average precision (mAP) explained: Everything you need to know." https: //www.v7labs.com/blog/mean-average-precision, 2022. Accessed: 2025-03-10.

[27] C. Consortium, "COCO object detection evaluation." https://cocodataset.org/ #detection-eval, 2025. Accessed: 2025-03-10.

[28] Ultralytics, "YOLOv10 vs EfficientDet comparison." https://docs.ultralytics.com/com-pare/yolov10-vs-efficientdet/, 2024. Accessed: 2025-03-09.

[29] Ultralytics, "YOLOv10 model comparisons." https://docs.ultralytics.com/de/models/yolov10/#comparisons, 2024. Accessed: 2025-03-09.

[30] P. Team, "MobileNetV3 small model." https://pytorch.org/vision/main/models/ generat-ed/torchvision.models.mobilenet_v3_small.html#torchvision.models. mobilenet_v3_small, 2025. Accessed: 2025-03-09.

[31] O. Toolkit, "EfficientDet-D0 TensorFlow model." https://github.com/openvinotoolkit/open_model_zoo/blob/master/models/public/efficientdet-d0-tf/README.md, 2025. Accessed: 2025-03-09.

[32] P. Team, "Faster R-CNN with ResNet-50 FPN." https://pytorch.org/vision/main/models/generat-ed/torchvision.models.detection.fasterrcnn_resnet50_fpn.html, 2025. Accessed: 2025-03-09.

[33] F. Research, "DETR: End-to-end object detection with transformers." https://github.com/ face-bookresearch/detr/blob/main/README.md, 2025. Accessed: 2025-03-09.

[34] PromptLayer, "DETR ResNet-50 model overview." https://www.promptlayer.com/models/ detr-resnet-50-6671, 2025. Accessed: 2025-03-09.

[35] Ultralytics, "RTDETRv2 vs YOLOv10: A technical comparison for object detection." https:// docs.ultralytics.com/compare/rtdetr-vs-yolov10/, 2024. Accessed: 2025-03-10.

[36] A. Kouidri, "Top object detection models in 2024." https://www.ikomia.ai/blog/top-object-detection-models-review, 2024. Accessed: 2025-03-10.

[37] PyTorch Community, "Metal performance shader (MPS) discussion forum." https://dis-cuss.pytorch.org/c/metal-performance-shader/38, 2025. Accessed: 2025-03-10.

[38] A. Inc., "Accelerated TensorFlow training with metal." https://developer.apple.com/met-al/tensorflow-plugin/, 2025. Accessed: 2025-03-10.

[39] R. Kumar, "List of CUDA-aware frameworks in machine learning." https://www. devopss-chool.com/blog/list-of-cuda-aware-framework-in-machine-learning/, 2024. Accessed: 2025-03-10.

[40] N. Corporation, "Deep learning software." https://developer.nvidia.com/ deep-learning-software, 2025. Accessed: 2025-03-10.

[41]N. Corporation, “Eingebettete systeme: Entwicklerkits und module von nvidia jetson.” https: //www.nvidia.com/de-de/autonomous-machines/embedded-systems/, 2025. Zugegriffen: 2025-03-10.

[42]J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time ob-ject detection,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recogni-tion (CVPR), pp. 779–788, 2016.

[43]T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Mi-crosoft COCO: Common objects in context,” in European Conference on Computer Vision (ECCV), pp. 740–755, Springer, 2014.