This ICCV workshop paper is the Open Access version, provided by the Computer Vision Foundation. Except for this watermark, it is identical to the accepted version; the final published version of the proceedings is available on IEEE Xplore.

YOLOBench: Benchmarking Efficient Object Detectors on Embedded Systems

Ivan Lazarevich Matteo Grimaldi Ravish Kumar Saptarshi Mitra
Shahrukh Khan Sudhakar Sah
Deeplite
ivan.lazarevich@deeplite.ai

Abstract

We present YOLOBench, a benchmark comprised of 550+ YOLO-based object detection models on 4 different datasets and 4 different embedded hardware platforms (x86 CPU, ARM CPU, Nvidia GPU, NPU). We collect accuracy and latency numbers for a variety of YOLO-based one-stage detectors at different model scales by performing a fair, controlled comparison of these detectors with a fixed training environment (code and training hyperparameters). Pareto-optimality analysis of the collected data reveals that, if modern detection heads and training techniques are incorporated into the learning process, multiple architectures of the YOLO series achieve a good accuracy-latency trade-off, including older models like YOLOv3 and YOLOv4. We also evaluate training-free accuracy estimators used in neural architecture search on YOLOBench and demonstrate that, while most state-of-the-art zero-cost accuracy estimators are outperformed by a simple baseline like MAC count, some of them can be effectively used to predict Pareto-optimal detection models. We showcase that by using a zero-cost proxy to identify a YOLO architecture competitive against a state-of-the-art YOLOv8 model on a Raspberry Pi 4 CPU. The code and data are available at https://github.com/Deeplite/deeplite-torch-zoo.

1. Introduction

Object detection constitutes a pivotal task in the field of computer vision, entailing the critical process of identifying and localizing objects present within an image. Applications of object detection models include autonomous vehicles, surveillance, robotics, and augmented reality [4]. The central problem of deploying deep learning-based object detection solutions on embedded hardware platforms is the amount of computation, memory, and power required for their inference [15]. This necessitates the development of efficient object detection models specialized for low-footprint hardware devices to achieve an optimal trade-off of accuracy and latency. For years, the state-of-the-art (SOTA) deep learning approach to object detection has been the series of YOLO architectures [28]. In recent years, remarkable strides have been taken in advancing YOLO-like single-stage object detectors, prioritizing real-time operation while simultaneously striving for higher accuracy and deployability on low-power devices. These advancements have primarily focused on enhancing various components of the detection pipeline. Key areas of improvement include the design of accurate and efficient backbone and neck structures within the network [31], exploration of different detection head designs (e.g. anchor-based [31] vs. anchor-free [8]), utilization of diverse loss functions [19], and implementation of novel training procedures including innovative data augmentation techniques [13]. These collective efforts have continually refined and evolved YOLO-like architectures, enhancing object detection effectiveness and efficiency in real-time scenarios. The differences between consecutive YOLO versions, such as YOLOv5 [11] and YOLOv6 [17], span various pipeline components, making it challenging to isolate their individual contributions. This paper aims to address these challenges by providing a fair comparison of recent YOLO versions under controlled conditions (e.g. same training loop for all models) to demonstrate the impact of the backbone and neck structure of YOLO-based models in embedded inference applications. We also use the collected accuracy and latency data for multiple YOLO-based detector variations to empirically evaluate training-free performance predictors commonly used in neural architecture search [1]. We summarize our contributions as follows:

- We provide a latency-accuracy benchmark of 550+ YOLO-based object detection models on 4 different datasets, called *YOLOBench*. All the models are validated on 4 different embedded hardware platforms (Intel CPU, ARM CPU, Nvidia GPU, NPU),
- We show that if modern detection heads and training techniques are implemented for the detector training pipeline, multiple backbone and neck variations, including those of older architectures (e.g. YOLOv3

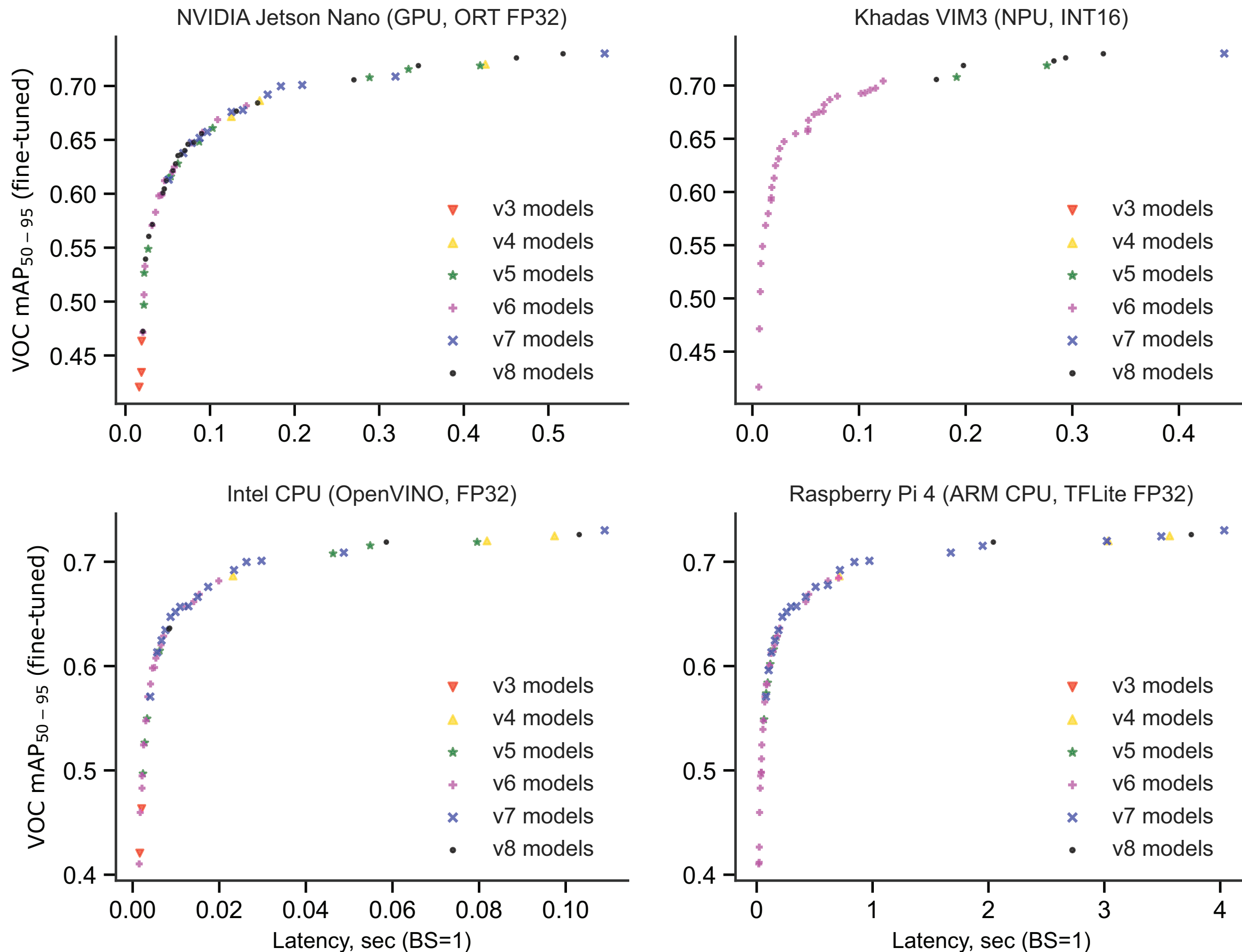

Figure 1. Pareto frontiers of *YOLOBench* models fine-tuned on the VOC dataset (on several target resolutions) from COCO-pretrained weights on 4 different hardware platforms. Each point represents a single model in the mAP-latency space, with the model family coded with color and marker shape (all YOLOv6-3.0 models are represented by the same color). Refer to Appendix B for Pareto frontier plots on other datasets.

and YOLOv4), can be used to achieve state-of-the-art latency-accuracy trade-off,

- Looking at *YOLOBench* as a neural architecture search (NAS) space, we demonstrate that, while most of the state-of-the-art zero-cost (training-free) proxies for model accuracy estimation are outperformed by simple baselines such as MAC count, the NWOT estimator [24] can be effectively used to identify potential Pareto-optimal YOLO detectors in a training-free manner,

- We showcase the effectiveness of the NWOT estimator for optimal detector prediction by using it to identify a YOLO-like model with FBNetV3 backbone that outperforms YOLOv8 on the Raspberry Pi 4 ARM CPU.

2. Related Work

There has been a tremendous amount of progress in efficient object detection in recent years pushing the accuracy-latency frontier, including architectures like YOLOv7 [31], YOLOv6-3.0 [17], DAMO-YOLO [34], RTMDet [23], RT-DETR [22] and PP-YOLOE [33]. These works oftentimes improve upon state-of-the-art latency-accuracy trade-offs, providing comparisons of several generations of detectors on the COCO dataset. Benchmarks of different model families are also provided by framework developers, such as MMYOLO [3] and Ultralytics [12]. Additionally, there exist third-party benchmarks of several architectures from the YOLO series on server-grade and embedded GPUs as well as specialized accelerators [7, 14, 25, 36]. We identify a few limitations of the existing efficient detector benchmarks that have served as motivation for *YOLOBench*:

Table 1. Pareto-optimal *YOLOBench* models on 3 datasets and 3 hardware platforms. Shown are the best models in terms of mAP_{50-95} under a given latency threshold (max. latency). For each model, the scaling parameters are given (d33w25 means depth factor $= 0.33$ and width factor $= 0.25$), corresponding input resolution of the models is indicated in brackets.

HW/max. latency	VOC model	VOC mAP_{50-95}	SKU-110k model	SKU-110k mAP_{50-95}	WIDERFACE model	WIDERFACE mAP_{50-95}
Nano/0.1 sec	YOLOv7 d1w5 (288)	0.657	YOLOv8 d1w25 (480)	0.567	YOLOv7 d1w25 (480)	0.336
VIM3/0.05 sec	YOLOv6l d67w25 (416)	0.620	YOLOv6s d33w25 (480)	0.556	YOLOv6m d67w25 (480)	0.318
Raspi4/0.5 sec	YOLOv6l d67w5 (384)	0.669	YOLOv4 d1w25 (480)	0.569	YOLOv7 d1w25 (480)	0.336

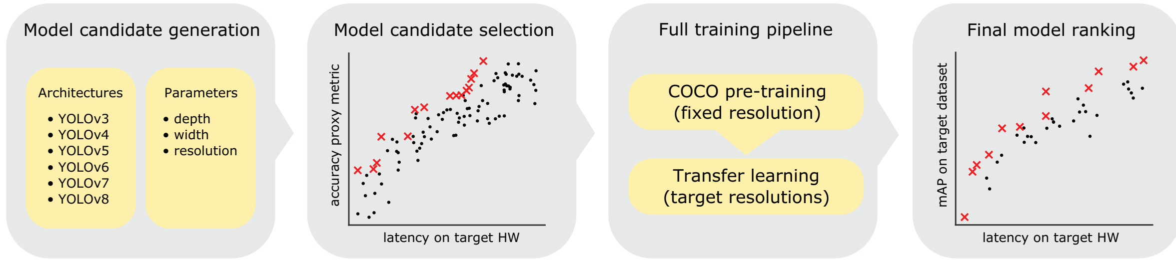

Figure 2. Flowchart of the *YOLOBench* process for model candidate generation, pre-selection and ranking. Pareto-optimal points are depicted as red crosses.

- Comparisons of different YOLO versions are frequently done either by using a proxy metric for the actual latency like MAC count and number of parameters or by reporting latency values on server-grade GPUs, neither of which is directly indicative of latency on embedded devices,
- Accuracy metrics are usually reported on the COCO dataset, which could be considered too large-scale with respect to actual practical use cases,
- Some architecture parameters (like input resolution) are often considered to be fixed in detector benchmarking, while it is known that they serve as important factors in optimal CNN scaling [10],
- Different YOLO variations being compared to one another are typically trained with different training codebases, training techniques (loss functions, data augmentations), and hyperparameter values, making it hard to disentangle the contribution of the training pipeline improvements vs. better architecture design.

To address these issues, we conduct a thorough accuracy and latency benchmarking of state-of-the-art YOLO detector versions in controlled, fixed conditions to study the impact of backbone and neck design proposed by several YOLO model families.

Table 2. *YOLOBench* architecture space (variation of backbone/neck, depth, width, and input resolution).

Model	Backbone	Neck
YOLOv3 [26]	DarkNet53	FPN
YOLOv4 [2]	CSPDNet53	SPP-PAN
YOLOv5 [11]	CSPDNet53-C3	SPPF-PAN-C3
YOLOv6s-3 [17]	EfficientRep	RepBiFPAN
YOLOv6m-3 [17]	CSPBep (e=2/3)	CSPRepBiFPAN
YOLOv6l-3 [17]	CSPBep (e=1/2)	CSPRepBiFPAN
YOLOv7 [31]	E-ELAN	SPPF-ELAN-PAN
YOLOv8 [12]	CSPDNet53-C2f	SPPF-PAN-C2f

Width factor $\in \{0.25, 0.5, 0.75, 1.0\}$
Depth factor $\in \{0.33, 0.67, 1.0\}$
Input resolution $\in \{160{:}480{:}32\}$

3. Methodology

The purpose of the current study is to thoroughly study the impact of the backbone and neck and its parameters (width, depth, input resolution) on the performance of YOLO detectors in terms of their accuracy and latency. For the rest of the factors influencing the accuracy-latency trade-off, such as choice of the detection head, loss function, training pipeline, and hyperparameters, we aim to have a fixed, controlled setup, so that we can isolate the effect of backbone and

neck design on model performance. For this reason, we use the anchor-free decoupled detection head of YOLOv8 [28], as well as CIoU and DFL losses for bounding box prediction used in YOLOv8, as they have been shown to produce state-of-the-art results on the COCO dataset. Anchor-free detection in YOLO models has been also shown to provide latency benefits in the end-to-end detection pipelines [22]. Hence, the main source of variation in *YOLOBench* models is the structure and parameters of the backbone and neck. We also use the same training code and hyperparameters for all models, as set by default in the YOLOv8 training code released by Ultralytics [12], which provides a relatively simple training loop capable of producing SOTA results.

The flow of candidate model generation, pre-selection, and training is shown in Figure 2. First, we generate the full architecture space consisting of about 1000 models by independently varying the backbone/neck structure, depth factor, width factor, and input resolution (Table 2). For each architecture, we consider its variations trained and tested on 11 different input resolutions (from 160x160 to 480x480 with a step of 32) and 12 variations in depth and width, aside from 4 usually considered scaling variants (n, s, m and l). The only exception is the YOLOv7 models, for which we only vary the width factor producing 4 variations of the model. For YOLOv6 models, we use the v3.0 version [17], for which provided s, m and l variations actually represent different architectures aside from different depth and width factors (see Table 2). Hence, we consider YOLOv6s, YOLOv6m and YOLOv6l as different model families and generate the same 12 depth-width combinations for each one.

Latency measurements. The actual inference latency for each model might vary significantly depending on the deployment environment and runtime. Therefore, we collect the latency measurements for each of the models by running inference on 4 different hardware platforms (runtime and inference precisions specified in brackets): (i) NVIDIA Jetson Nano GPU (ONNX Runtime, FP32), (ii) Khadas VIM3 NPU (AML NPU SDK, INT16), (iii) Raspberry Pi 4 Model B CPU (TFLite with XNNPACK, FP32), (iv) Intel® Core™i7-10875H CPU (OpenVINO, FP32).

We did not consider latency measurements for INT8 precision, as depending on the quantization scheme (e.g. per-tensor vs. per-channel) and approach (e.g. post-training quantization vs. quantization-aware training), there can be a varied impact of INT8 quantization on accuracy. Adding INT8 results for both accuracy and latency in *YOLOBench* is a matter of future work. All latency measurements were performed with a batch size of 1 averaged over 200 inference cycles (with 5 warmup steps). We measured the inference time required to execute the YOLO model graph, without taking bounding box post-processing (e.g. non-maximum suppression) into account. Note that for VIM3 NPU measurements, the bounding box decoding post-processing operations (operations after the last convolutional layers of the network) were also skipped due to the limitations of VIM3 SDK.

Training pipeline. To obtain the accuracy metric values for the models, we consider the following 4 datasets: (i) PASCAL VOC (20 object categories) [6], (ii) SKU-110k (1 class, retail item detection) [9], (iii) WIDER FACE (1 class, face detection) [35], (iv) COCO (80 object categories) [21]. Our motivation to include several smaller-scale (with respect to COCO) but challenging datasets stems from the fact that for many practical deployment use cases, the number of object categories to detect and the amount of available data might be limited. The metric of interest for all datasets is mAP_{50-95}. For all selected models, the training procedure starts with pretraining on the COCO dataset (for 300 epochs, with a batch size of 64 and 640x640 resolution), afterward the best COCO weights are used as initialization for other datasets, on which we perform fine-tuning for 100 epochs (batch size of 64) on all 11 *YOLOBench* resolutions and select the best weights (in terms of mAP_{50-95} value) for each one. For the COCO dataset, we do not perform fine-tuning on target resolutions, rather we evaluate the model trained on 640x640 images on all target resolutions (to mimic the deployment of pre-trained COCO weights). All other training hyperparameters are set as per default values of the Ultralytics YOLOv8 codebase [12]. Model weights are randomly initialized for all experiments (e.g. no transfer of ImageNet weights for the backbone is performed).

Candidate model pre-selection. In order to reduce the number of training runs on the COCO dataset, we filter out some of the least promising model candidates from the *YOLOBench* architecture space as an initial step of our benchmarking procedure. To determine the most promising models in terms of the accuracy-latency trade-off, we compute a proxy metric that is well correlated with the final mAP values of the models fine-tuned on the target datasets. A natural choice for such a metric is model performance when trained on a smaller-scale representative dataset. For this purpose, we use the VOC dataset to train all the model candidates from scratch (random initialization) for 100 epochs and use the resulting mAP_{50-95} value as a proxy metric to predict performance on all target datasets (with models fine-tuned on these datasets from COCO pre-trained weights). We observe a good correlation of such a training-based accuracy proxy with final metrics on all considered datasets (even on datasets from other domains, like SKU-110k; see Appendix C). We also examine the performance of training-free accuracy estimators for this task and compare it to mAP of VOC training from scratch (see Section 4.2).

Once we have the accuracy proxy values and latency measurements for all models in the dataset, we determine the models with the best accuracy-latency trade-off (the Pareto frontier models). We use the OApackage software library [5]

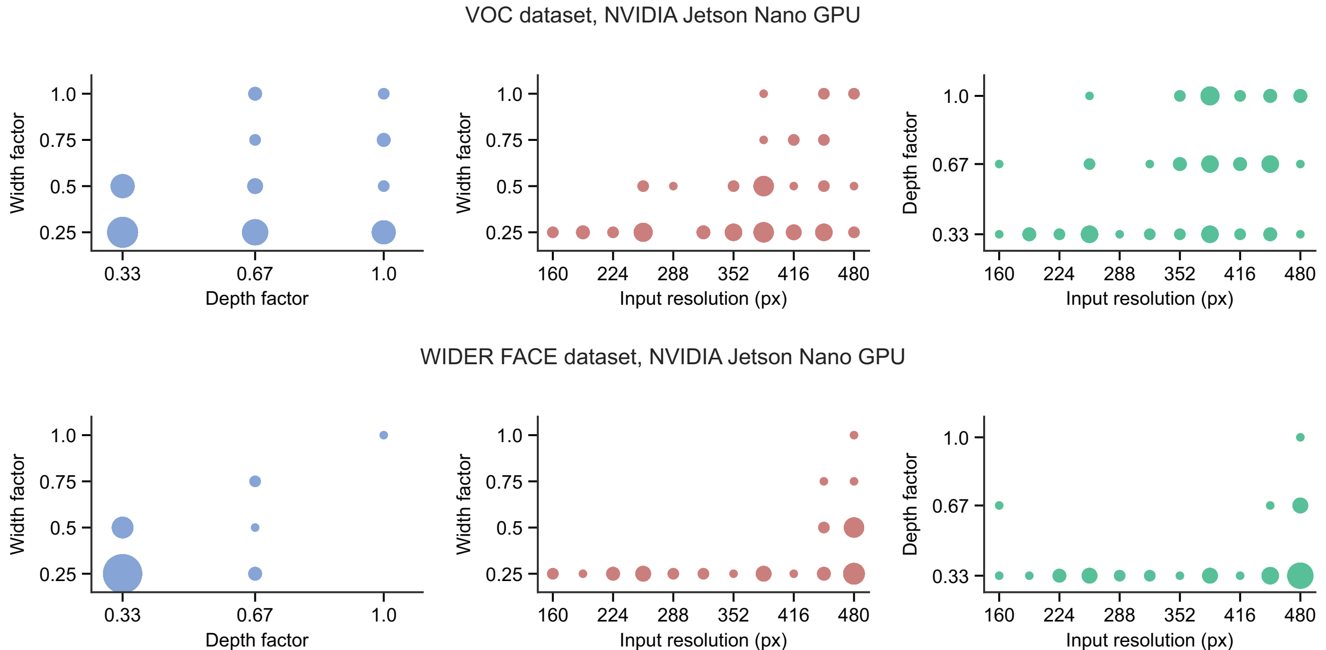

Figure 3. Statistics of model scaling parameters (depth factor, width factor, input resolutions) in Pareto-optimal models on VOC and WIDER FACE datasets with latency measured on the Jetson Nano GPU. The size of each point (circle) is proportional to the number of models for that parameter combination.

to determine the Pareto optimal elements in the latency-accuracy space. We define the second Pareto set as the set of models that are Pareto-optimal if the initial Pareto set models are removed (so that the "second best" models in terms of latency-accuracy trade-off become the best). Correspondingly, we define the N-th Pareto set.

For our model pre-selection procedure, we consider the models contained in the first and second Pareto fronts (in terms of mAP_{50-95} in VOC training from scratch), with latency for each considered hardware platform separately. We merge all the first and second Pareto sets for each HW platform to form the list of promising architectures to be selected for COCO pre-training. After the COCO pre-training phase is finished for a model, variations of that architecture on multiple resolutions are considered in the benchmark.

4. Results

4.1 Pareto-optimal YOLO models

By computing the proxy metric for model accuracy (mAP_{50-95} in VOC training from scratch) and latency values for the whole *YOLOBench* architecture space on several hardware platforms, we determine the Pareto sets containing the most promising models (in terms of latency-accuracy trade-off) for each HW platform. The first and second Pareto sets for each device are merged into a unified list of best architectures, which is comprised of 52 backbone/neck combinations for COCO pre-training. Same architectures with different input resolutions are considered as the same data points in this list since COCO pre-training is regardless done on a fixed resolution of 640x640. The COCO pre-training phase is followed by fine-tuning at 11 different resolutions (from 160x160 to 480x480 with a step of 32) on all downstream datasets (except for COCO), resulting in 572 models total for each dataset.

Finally, with the obtained fine-tuned model accuracy on several datasets and latency measurements on several devices, we compute the actual Pareto sets for each particular dataset/HW platform combination. Figure 1 shows the Pareto frontiers of *YOLOBench* models fine-tuned on the VOC dataset on 4 different devices. Notably, significant differences emerge in these Pareto frontiers between different devices. In particular, the Pareto-optimal set for VIM3 NPU is mostly comprised of YOLOv6 models, with some YOLOv5, YOLOv7, and YOLOv8 models present in the higher accuracy region. This is not found to be the case for the Pareto sets of Intel and ARM CPUs. Despite containing a few YOLOv6 models in the lower latency region, these sets also encompass numerous YOLOv5 and YOLOv7 variations, with limited representation from other model families such as YOLOv3 and YOLOv4. While the Pareto sets for Intel and ARM CPUs exhibit a certain degree of similar-

Table 3. Performance of training-free accuracy predictors on *YOLOBench* models and two datasets (VOC and SKU-110k, from COCO-pretrained weights) compared to using metrics of models trained from scratch on the VOC dataset as a predictor. Refer to Appendix C for the data on all considered zero-cost metrics.

	VOC, mAP_{50-95}			SKU-110k, mAP_{50-95}		
Predictor metric	global τ	top-15% τ	%Pareto pred. (GPU)	global τ	top-15% τ	%Pareto pred. (GPU)
JacobCov	0.095	-0.078	0.015	0.541	0.136	0.025
ZiCo	0.195	0.016	0.015	0.115	0.081	0.025
Zen	0.255	0.092	0.062	0.146	0.121	0.050
Fisher	0.280	0.156	0.015	-0.380	-0.096	0.025
SNIP	0.336	0.217	0.015	-0.290	-0.059	0.025
#params	0.399	0.372	0.031	0.256	0.119	0.050
SynFlow	0.558	0.227	0.062	0.512	0.254	0.100
MACs	0.739	0.520	0.123	0.604	0.314	0.125
NWOT	0.756	0.622	0.262	0.703	0.321	**0.200**
NWOT (pre-act)	**0.827**	**0.623**	**0.292**	**0.765**	**0.406**	**0.200**
VOC training from scratch (mAP_{50-95})	0.847	0.665	0.369	0.739	0.374	0.425

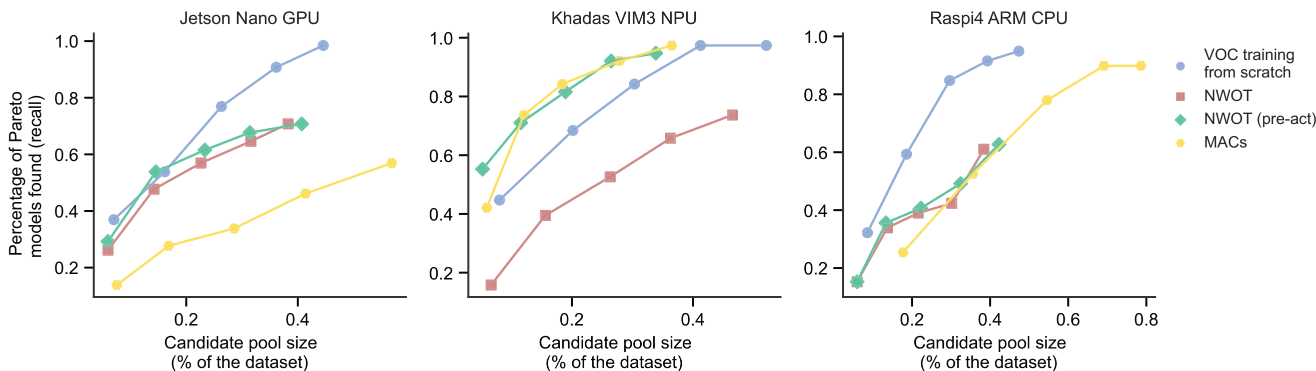

Figure 4. Percentage of all actual Pareto models (recall) found in the candidate pools consisting of first N (N from 1 to 5) ZC-based Pareto sets depending on the HW platform and ZC metric. The data shown is for Pareto-optimal models on the VOC dataset. The x-axis shows the candidate pool size as a percentage of the full dataset size.

ity, the Jetson Nano GPU stands out from the rest of the devices. It showcases a non-uniform distribution of model families, with YOLOv5, YOLOv6, YOLOv7, and YOLOv8 models all represented across the entire accuracy/latency space. Table 1 shows representative Pareto-optimal models for 3 different datasets (VOC, SKU-110k, WIDERFACE) and 3 hardware platforms under certain latency thresholds. Note that although there are similarities of model family distributions in Pareto sets computed for different datasets (see Appendix B), the exact optimal model for a given latency threshold depends on the specific dataset of interest.

Next, we analyze the statistics of Pareto-optimal models depending on the dataset and hardware platform. Figure 3 shows the distribution of depth factor, width factor, and input resolution values in Pareto frontier models for VOC and SKU-110k datasets on Jetson Nano GPU (data for other datasets and devices are available in Appendix B). The general trend indicates that models at lower input resolutions mostly have lower depth and width factors. This suggests that achieving an optimal latency-accuracy trade-off involves scaling down both the architecture's depth and width before reducing the input resolution. This effect is more pronounced in some datasets (SKU-110k and WIDERFACE), where almost all optimal models are either at the maximal resolution we considered (480x480) with variation in width and depth, or at lower resolutions with minimal width and depth factors. This effect is dataset-dependent, as a more relaxed trend is observed for VOC and COCO datasets, where many optimal models with a variation in width and depth factor are found at resolutions lower than 480x480.

To summarize, we demonstrate that with a state-of-the-art training pipeline and detection head structure, YOLO-based

models with various backbone/neck combinations could achieve good latency-accuracy trade-offs in various deployment scenarios, including older backbone/neck structures from YOLOv4 and YOLOv3 models. Furthermore, we show that depth/width reduction precedes input resolution downscaling in optimal YOLO-based detectors.

4.2 Ranking training-free accuracy predictors

With an increasing number of architecture blocks and hyperparameter combinations, the size of the candidate model space in *YOLOBench* can further grow exponentially. Hence, it is important to develop efficient methods of filtering out bad architecture proposals before running them through the full training pipeline, including pre-training on the COCO dataset. In the field of neural architecture search, recent works have proposed a handful of training-free, *zero-cost* (ZC) estimators that have been shown to perform well on various (relatively simple) benchmarks [1, 18, 24].

Zero-cost estimators were originally proposed by Mellor et al. [24], and later expanded by Abdelfattah et al. [1] as a means to quickly evaluate the performance of an architecture using only a mini-batch of data. These estimators work by extracting statistics obtained from a forward (and/or backward) pass of a few mini-batches of data through the network, hence eliminating the need for full training of the model. Despite the fact that over 20 different zero-cost accuracy estimators have been introduced in recent years, simple baselines like the number of parameters and MAC count are still found to be hard to outperform [18].

The vast search space of YOLO-like architectures necessitates the development of effective training-free estimators to filter out bad candidates and reduce the search space. We examine the performance of a representative subset of zero-cost estimators on *YOLOBench*, namely: Fisher [29], GradNorm [1], GraSP [30], JacobCov [1], Plain [1], SNIP [16], SynFlow [27], ZiCo [18], Zen-score [20] and NWOT [24]. The NWOT metric is computed by measuring the Hamming distance between binary codes produced by each layer's activations [24]. Although originally proposed for ReLU-based networks, we observe that it works well in practice for YOLO variations, most of which contain SiLU activations. The NWOT metric can also be computed by taking the signs of each layer's output features before the activation layer to form the binary code. We refer to that version of the NWOT metric as *NWOT (pre-act)* ("pre-activation") and find that its performance might differ significantly from the original NWOT metric, primarily because the binary codes are computed before the normalization layers followed by the activations. We also compare the performance of the zero-cost predictors with simple baselines such as the number of trainable parameters and MAC count, as well as with a training-based proxy that we have used to pre-select models for *YOLOBench* (mAP_{50-95} in training from scratch on the VOC dataset).

All zero-cost metrics are computed on randomly initialized models with the same loss function as used for training of all *YOLOBench* models and using a single mini-batch of data with a corresponding image resolution (except for ZiCo, which requires two different mini-batches of data [18]). We empirically evaluate the considered set of zero-cost proxies on *YOLOBench* using the following metrics:

- Kendall τ (global): Kendall rank correlation coefficient evaluated on all *YOLOBench* models
- Kendall τ (top-15%): Kendall rank correlation coefficient evaluated on the top-15% performing *YOLOBench* models (in terms of mAP_{50-95} value)
- Percentage of all actual Pareto-optimal models in the Pareto set determined with the zero-cost estimator in the zero-cost proxy-latency space (recall for Pareto-optimal model prediction using the ZC-based Pareto set)

The last metric effectively measures how accurate the computed Pareto set would be if the proxy values are used instead of actual mAP to rank models. It is calculated by determining Pareto fronts for model rankings based on zero-cost proxies (and real latency measurements) and then estimating how many models present in the actual Pareto set are also present in the ZC-based Pareto set. In other words, a recall value of 0.7 would mean that by taking the models from the ZC-based Pareto set as candidates, we find 70% of all actual Pareto-optimal models in that candidate set. We report values for Pareto fronts computed with latency measurements on the Jetson Nano GPU in Table 3.

We generally find that all of the zero-cost predictors we consider (except for NWOT) are outperformed by the simple baseline of MAC count both in terms of Kendall-Tau scores as well as in the percentage of predicted Pareto-optimal models (see Table 3). Furthermore, when compared with using mAP_{50-95} on VOC training from scratch as a predictor, we observe that only NWOT comes close to it in terms of ranking scores. We also find that the pre-activation version of NWOT tends to work better than standard NWOT on *YOLOBench*. For the task of predicting mAP_{50-95} of models fine-tuned on SKU-110k, we notably observe that pre-activation NWOT outperforms VOC training from scratch metric in terms of Kendall-Tau scores (possibly due to domain difference between VOC and SKU-110k datasets), but the VOC-based proxy metric still performs better for Pareto-optimal model prediction on SKU-110k. For the data on the sensitivity of NWOT predictions to hyperparameter values please refer to Appendix C.

In trying to capture all the real Pareto-optimal models using ZC scores, one could expand the ZC-based candidate

Table 4. COCO test and minival mAP and inference latency on Raspberry Pi 4 CPU (TFLite, FP32) for YOLOv8s vs. a model identified from the NWOT-latency Pareto frontier. For latency, mean and standard deviation over 5 runs (each run done for 100 iterations) are shown, with 640x640 input resolution. For mAP, the mean and standard deviation over three random seeds are shown.

Model	$\mathrm{mAP}^{test}_{50-95}$	$\mathrm{mAP}^{val}_{50-95}$	Latency, ms
YOLOv8s	43.17% (0.12%)	44.43% (0.23%)	1476.09 (1.49)
YOLOv8s (HSwish)	42.90% (0.00%)	44.23% (0.10%)	1381.62 (7.34)
YOLO-FBNetV3-D-PAN-C3	**43.87%** (0.05%)	**45.30%** (0.08%)	**1355.21** (9.93)

pool by calculating subsequent Pareto sets (second, third, fourth, and so forth) and incorporating them into the candidate pool. By applying this strategy, it's possible to identify the complete set of actual Pareto-optimal models while examining only a subset of the entire dataset (e.g., the first N ZC-based Pareto fronts). In this context, we compute candidate pools consisting of N ZC Pareto fronts for each ZC metric and look at the percentage of actual Pareto-optimal models found in the pool versus the pool size (as % of the full dataset size). Looking at the pool size is motivated by the observation that the number of models in ZC-based Pareto fronts can significantly vary depending on the specific ZC metric used.

Figure 4 shows the percentage of predicted real Pareto-optimal models on the VOC dataset contained in pools of N first Pareto fronts for 4 different predictors (VOC training from scratch, NWOT, pre-activation NWOT, and MAC count). For ARM and Intel CPUs, we observe a general trend of VOC training from scratch being the best predictor and MAC count being the worst at all points. Interestingly, for Jetson Nano GPU NWOT performs close to VOC training from scratch for $N = 1, 2$ but starts to perform worse with more models in the pool. Surprisingly, MAC count and pre-activation NWOT, which are training-free predictors, outperform VOC training from scratch in predicting Pareto-optimal models on VIM3 NPU.

4.3 Pareto-optimal detector identification using NWOT score

To demonstrate the potential of using ZC-based Pareto sets in identifying promising detector architectures with good accuracy-latency trade-off, we additionally generate multiple candidate architectures based on CNN backbones provided by the `timm` library [32]. The architectures are generated by using one of the 347 CNN-based backbones available in `timm` as a feature extractor followed by a modified Path Aggregation Network (PAN) (same structure with C3 blocks as in YOLOv5 is used, with the number of channels corresponding to YOLOv5s, without the SPPF block) and a YOLOv8 detection head, as in all other *YOLOBench* models.

We compute the pre-activation NWOT scores as well as measure inference latency on Raspberry Pi 4 ARM CPU with TFLite for all candidate models. We then use the NWOT score and latency values for each model to compute the Pareto frontier in the NWOT-latency space (see Appendix D). We then train one of the models identified to belong to the NWOT-based Pareto frontier (YOLO with FBNetV3-D backbone) on the COCO dataset with a similar setup used to pre-train *YOLOBench* models (640x640 input resolution, 500 epochs, batch size 256, other hyperparameters set to default of Ultralytics YOLOv8 [12])[1]. The resulting model is found to be more accurate and faster than YOLOv8s (a model in a similar latency range) when tested on Raspberry Pi 4 CPU with TFLite (FP32, XNNPACK backend) (see Table 4). Furthermore, we look at the accuracy and latency of a YOLOv8s modification with SiLU activations replaced with Hardswish activations (Table 4), as we observe the choice of activation function to be a significant factor affecting TFLite inference latency. We find that the identified NWOT-Pareto model (also containing Hardswish activations in the backbone, neck, and head) still outperforms YOLOv8s-HSwish in terms of latency and accuracy.

5. Conclusion

In this work, we present *YOLOBench*, a latency-accuracy benchmark of several hundred YOLO-based models on 4 different object detection datasets and 4 different hardware platforms. The accuracy and latency data are collected in a fixed, controlled environment with the only variation in backbone/neck structure and input image resolution of the detectors. We use these data to demonstrate that it is possible to achieve Pareto-optimal results with a range of different backbone structures, including those of the older architectures in the YOLO series, such as YOLOv3 and YOLOv4. We also observe that depth and width scaling precede input resolution scaling in optimal YOLO-based detectors.

Finally, we use *YOLOBench* to evaluate zero-cost accuracy predictors, and find that, while many of the existing state-of-the-art predictors perform poorly, pre-activation NWOT score can be effectively used to identify Pareto-optimal detectors for specific target devices of interest. We demonstrate that by using NWOT to find a YOLO backbone (FBNetV3-D) that outperforms a state-of-the-art YOLOv8 model when deployed on a Raspberry Pi 4 ARM CPU.

[1] Note that YOLOv8s results provided by Ultralytics [12] are slightly higher than the ones we report. However, no script to reproduce those results has been released to date.

References

[1] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*, 2021. 1, 7

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020. 3

[3] MMYOLO Contributors. MMYOLO: OpenMMLab YOLO series toolbox and benchmark. https://github.com/open-mmlab/mmyolo, 2022. 2

[4] Tausif Diwan, G Anirudh, and Jitendra V Tembhurne. Object detection using yolo: Challenges, architectural successors, datasets and applications. *multimedia Tools and Applications*, 82(6):9243–9275, 2023. 1

[5] Pieter Thijs Eendebak and Alan Roberto Vazquez. Oapackage: A python package for generation and analysis of orthogonal arrays, optimal designs and conference designs. *Journal of Open Source Software*, 4(34):1097, 2019. 4

[6] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results, 2012. 4

[7] Haogang Feng, Gaoze Mu, Shida Zhong, Peichang Zhang, and Tao Yuan. Benchmark analysis of yolo performance on edge intelligence devices. *Cryptography*, 6(2):16, 2022. 2

[8] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021. 1

[9] Eran Goldman, Roei Herzig, Aviv Eisenschtat, Jacob Goldberger, and Tal Hassner. Precise detection in densely packed scenes. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5227–5236, 2019. 4

[10] Kai Han, Yunhe Wang, Qiulin Zhang, Wei Zhang, Chunjing Xu, and Tong Zhang. Model rubik's cube: Twisting resolution, depth and width for tinynets. *Advances in Neural Information Processing Systems*, 33:19353–19364, 2020. 3

[11] Glenn Jocher. YOLOv5 by Ultralytics, May 2020. 1, 3

[12] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. YOLO by Ultralytics, Jan. 2023. 2, 3, 4, 8

[13] Parvinder Kaur, Baljit Singh Khehra, and Er Bhupinder Singh Mavi. Data augmentation for object detection: A review. In *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 537–543. IEEE, 2021. 1

[14] Stereo Labs. Performance benchmark of yolo v5, v7 and v8. https://www.stereolabs.com/blog/performance-of-yolo-v5-v7-and-v8/, 2023. 2

[15] Nicholas D Lane, Sourav Bhattacharya, Akhil Mathur, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing*, 16(3):82–88, 2017. 1

[16] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018. 7

[17] Chuyi Li, Lulu Li, Yifei Geng, Hongliang Jiang, Meng Cheng, Bo Zhang, Zaidan Ke, Xiaoming Xu, and Xiangxiang Chu. Yolov6 v3. 0: A full-scale reloading. *arXiv preprint arXiv:2301.05586*, 2023. 1, 2, 3, 4

[18] Guihong Li, Yuedong Yang, Kartikeya Bhardwaj, and Radu Marculescu. Zico: Zero-shot nas via inverse coefficient of variation on gradients. *arXiv preprint arXiv:2301.11300*, 2023. 7

[19] Xiang Li, Wenhai Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection. *Advances in Neural Information Processing Systems*, 33:21002–21012, 2020. 1

[20] Ming Lin, Pichao Wang, Zhenhong Sun, Hesen Chen, Xiuyu Sun, Qi Qian, Hao Li, and Rong Jin. Zen-nas: A zero-shot nas for high-performance image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 347–356, 2021. 7

[21] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014. cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list. 4

[22] Wenyu Lv, Shangliang Xu, Yian Zhao, Guanzhong Wang, Jinman Wei, Cheng Cui, Yuning Du, Qingqing Dang, and Yi Liu. Detrs beat yolos on real-time object detection. *arXiv preprint arXiv:2304.08069*, 2023. 2, 4

[23] Chengqi Lyu, Wenwei Zhang, Haian Huang, Yue Zhou, Yudong Wang, Yanyi Liu, Shilong Zhang, and Kai Chen. Rtmdet: An empirical study of designing real-time object detectors. *arXiv preprint arXiv:2212.07784*, 2022. 2

[24] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021. 2, 7

[25] Sovit Rath and Vikas Gupta. Performance comparison of yolo object detection models – an intensive study. https://learnopencv.com/performance-comparison-of-yolo-models/, 2022. 2

[26] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018. 3

[27] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in neural information processing systems*, 33:6377–6389, 2020. 7

[28] Juan Terven and D Cordova-Esparza. A comprehensive review of yolo: From yolov1 and beyond. *arXiv preprint arXiv:2304.00501*, 2023. 1, 4

[29] Jack Turner, Elliot J Crowley, Michael O'Boyle, Amos Storkey, and Gavin Gray. Blockswap: Fisher-guided block substitution for network compression on a budget. *arXiv preprint arXiv:1906.04113*, 2019. 7

[30] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*, 2020. 7

[31] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7464–7475, 2023. 1, 2, 3

[32] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019. 8

[33] Shangliang Xu, Xinxin Wang, Wenyu Lv, Qinyao Chang, Cheng Cui, Kaipeng Deng, Guanzhong Wang, Qingqing Dang, Shengyu Wei, Yuning Du, et al. Pp-yoloe: An evolved version of yolo. *arXiv preprint arXiv:2203.16250*, 2022. 2

[34] Xianzhe Xu, Yiqi Jiang, Weihua Chen, Yilun Huang, Yuan Zhang, and Xiuyu Sun. Damo-yolo: A report on real-time object detection design. *arXiv preprint arXiv:2211.15444*, 2022. 2

[35] Shuo Yang, Ping Luo, Chen-Change Loy, and Xiaoou Tang. Wider face: A face detection benchmark. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5525–5533, 2016. 4

[36] Jiawei Zhu, Haogang Feng, Shida Zhong, and Tao Yuan. Performance analysis of real-time object detection on jetson device. In *2022 IEEE/ACIS 22nd International Conference on Computer and Information Science (ICIS)*, pages 156–161. IEEE, 2022. 2