

## Accelerated Real-Time Face Recognition and Segmentation with Yolov8 Optimized through Tensor RT

Husam Salah Mahdi<sup>1\*</sup>, Dr. K. Raja Kumar<sup>2</sup>, K. John David Christopher<sup>3</sup>

<sup>1,2,3</sup> Department of CS & SE, Andhra University College of Engineering, Andhra University, India. \*

Email: <sup>1</sup>hussam05@gmail.com, <sup>2</sup>dr.kr.kumar@andhrauniversity.edu.in

, <sup>3</sup> 321506402136@andhrauniversity.edu.in

Orcid: 0000-0002-5179-7860<sup>1</sup>, Orcid Id : 0000-0001-6056-0294<sup>2</sup>, Orcid Id : 0009-0000-1278-7259<sup>3</sup>

### ARTICLE INFO

Received: 15 Dec 2024

Revised: 20 Feb 2025

Accepted: 26 Feb 2025

### ABSTRACT

Real-time face segmentation in embedded systems is a challenging task that requires a proper reconciliation between the computational cost and the segmentation quality. However, the currently available approaches frequently pay more attention to one of these factors. This paper solves this problem by first identifying framework-aware optimisations and architectural scalability for the YOLOv8-seg model. A systematic evaluation of the model size across five model scales (X, L, M, N, S) shows that the N scale is optimal in terms of the mAP<sub>50-95</sub> of 0.8283 at the frame rate of 137.20 FPS when trained and evaluated in PyTorch, outperforming the L model (0.7758 mAP, 29.30 FPS) in terms of speed and accuracy. The inference time is also optimised with TensorRT, which enhances the inference latency by 58% (4.28 ms/image) with nearly equal mAP<sub>50-95</sub> of 0.8170, which is almost the same as that of native PyTorch. Our analysis reveals that TensorRT improves the throughput by 233.67 FPS for model N, but smaller architectures (N, S) are more efficient in terms of latency-accuracy compared to larger architectures (X, L) where there is a low return on investment (for example, the X model has an mAP of 0.7301 and frames per second of 11.29). Present a framework for deploying a system that assists in choosing the scale of the model and the inference engine (PyTorch, ONNX, TensorRT) based on the application's latency and memory requirements. The effectiveness of the proposed methodology is tested through experiments on NVIDIA Jetson platforms, and it can achieve real-time frame rates ( $\geq 37.17$  FPS) with less than 3% quantization in accuracy, which is sufficient for consistent face segmentation in practical environments. This work closes the accuracy-deplorability gap and provides practical recommendations for designing edge computing applications for AR, biometrics, and privacy-preserving systems.

**Keywords:** TensorRT, Face segmentation, FPS, Edge AI, Latency-Accuracy Tradeoff

## 1. Introduction

The expansion of applications such as augmented reality (AR), biometric authentication and human-computer interaction (HCI) has greatly increased the need for advanced and real-time algorithms for face segmentation, transforming the market demand for such technologies. Face segmentation is the process of extracting a facial region from a complex image, and this process is one of the crucial steps

for more sophisticated applications such as virtual makeup, automatic facial expression analysis, and private face video conferencing [1]. The task of face segmentation remains a challenge on edge devices due to their limited processing resources because of the delicate balancing act practitioners need to do between having efficient computing resources, fast inference, and accuracy of segmentation [2]. Even though architecture design innovations have come with MobileNet [3] and YOLO models [4] and boosted the efficiency of vision models profoundly, their real-world applicability is too often blind-sided by more important issues like the optimization of the framework, diversity of the datasets, and the possibility of scalability of model topology on the hardware platforms. The majority of these approaches to face segmentation assume that the only issue remaining is architectural improvements to the model, and no attempt is made to consider the combination of data cleaning, framework-specific inference optimization, and model downscaling.

## 2. Overview Of Tensorrt and Deep Learning

An overview of the DL framework and proposed workflow, compiler, and runtime employed for this study is given in this section.

### A. Proposed workflow

Figure 1 depicts the proposed workflow for building the dataset, YOLOv8-seg model training, and model optimization for real-time facial segmentation. Dataset collection and preprocessing initiate the process, followed by the systematic building of the dataset. YOLOv8-seg model training on various scales (X, L, M, N, S) with hyperparameter tuning is done to achieve optimal segmentation accuracy. Then, the best-performing model is optimized with TensorRT to enable fast real-time inference on embedded hardware like the Jetson Orin Nano. Ultimately, FPS benchmarking on various hardware platforms is done to verify real-time feasibility and computational efficiency.

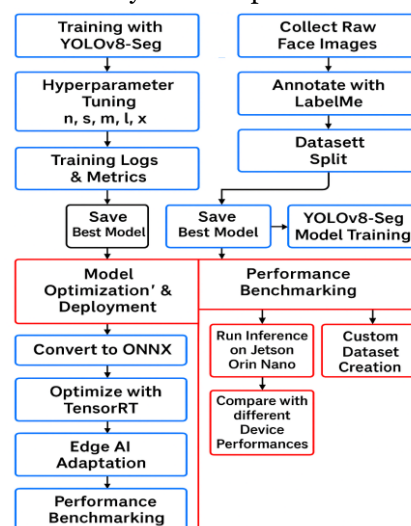


Figure 1 Proposed workflow

### B. PyTorch

An open-source framework called PyTorch makes the transition from experimentation motivated by research to real production deployment easier. For the great majority of deep learning workloads [5], PyTorch prioritizes speed and flexibility because it is a Python-focused package. Using Tensors (n-dimensional arrays) on CPUs and GPUs, PyTorch can increase the computation speed. The platform offers a rich set of tensor operations that are useful for many scientific computations, including linear algebra and other mathematical operations. It also implements reverse-mode automatic differentiation to enable online tuning of the network behaviour with almost no overhead. To increase execution speed, it is also integrated with high-performance libraries such as NCCL, cuDNN, and Intel

MKL [6]. Compared to some other frameworks, it has superior memory management, allowing for the training of larger NN models.

#### C. ONNX

Many technological companies and academic organizations promote the Open Neural Network Exchange (ONNX), an open standard for describing machine learning models. In addition to using a standardized file format to facilitate model interchange across various frameworks, tools, compilers, and runtimes, it defines a set of fundamental operations that are necessary for creating AI systems [7]. ONNX enables efficient model deployment on various hardware and is interoperable with platforms like PyTorch, TensorFlow, Caffe2, and Apache MXNet. By utilizing specially designed runtimes that are compatible with the target hardware [8], ONNX models can achieve improved inference performance.

#### D. TensorRT

TensorRT is a deep learning inference engine that comes as a software development kit (SDK). It is a part of the NVIDIA CUDA X AI Kit and provides an optimizer and runtime for deep learning inference that offers high throughput and low latency [9]. TensorRT is an optimization tool for inference that performs six types of optimizations to reduce the latency and increase the throughput of deep learning models:

1. Layer and tensor fusion: Improves the use of GPU memory and bandwidth by fusing nodes in a kernel up and down, or across, which reduces the overhead of reading and writing the tensor data for each layer.

2. Dynamic tensor memory: By only allocating memory to the tensor for the duration that it is required, this feature enhances memory re-usage.

By doing this, memory usage is decreased, and the expense of memory allocation is avoided, allowing for more effective execution.

3. Multi-stream execution: It handles many streams of input simultaneously.

4. Fusion of time TensorRT uses dynamically generated kernels to optimize recurrent neural networks (RNNs) across time steps [10]. TensorRT may accept customizable deep learning model inputs and serve a wide range of AI applications, including text-to-speech, computer vision, automatic speech recognition, natural language processing (BERT), and recommender systems. For computer vision models nearby, like those used in autonomous driving, it can deliver ready-to-use inference engines. Furthermore, TensorRT enables low-latency real-time video analytics in large-scale data centers, and its optimization makes it suitable for real-time edge deployments – e.g., on the NVIDIA Jetson Orin Nano Developer Kit or in various Internet of Things (IoT) scenarios.

### **3. Materials and Methods**

#### **3.1 Image Collection and Dataset Construction**

##### **3.1.1 Data Collection and Annotation**

The dataset collected in this study involves images that were captured using a Canon EOS 3000D camera, as well as other images that were downloaded from online repositories. The dataset is divided into six classes, which were manually captured, and four classes, which were downloaded from open source repositories, namely the Celebrity Faces NC dataset. The dataset consists of images of 10 subjects, namely: ['Alkinani', 'Cruise', 'Dr Raja Kumar', 'HUMMMAM', 'Layan', 'Leonardo', 'Tom', 'Will Smith', 'Abdoo', 'Madhur']. All images are in JPEG format and of size 640 × 640 pixels. Manual annotation was done using LabelMe, and the labels are in JSON format. These annotations were then converted to labelme2yolov8 format with polygon labels that define the coordinates of the face segmentation tasks and then the dataset.yaml file to use for training the model. Figure 2: Face Annotation using LabelMe Tool.

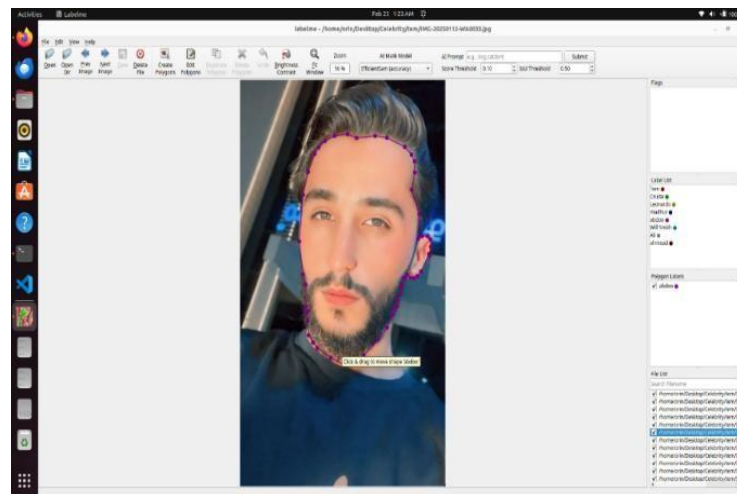


Figure 2 LabelMe Tool for Face Annotation

### 3.1.2 Dataset Augmentation and Construction

The dataset's visual attributes have a significant impact on the training accuracy of deep learning models. It implies that to prevent overfitting, the model's accuracy increases with the density of the input. Nevertheless, the model's accuracy may be harmed by noise in the photos and mistakes in the annotations. This research uses 90° clockwise and counterclockwise rotations, 0.1% Gaussian noise,  $\pm 25\%$  saturation, and  $\pm 15\%$  brightness fluctuations to enhance the dataset size to overcome the issue of insufficient data and prevent network overfitting. These methods increased the number of training images from 1372 to 2741, as shown in Table 1, Before and After Augmentation.

Table 1 Before and After Augmentation

	Train Images	Val Images	Test Images	Total Images
Before Augmentation	1372	168	84	1624
After Augmentation	2741	168	84	2993

## 3.2 Network Model Construction

### 3.2.1 Structure of the YOLOv8-Seg Network

Real-time object detection: The YOLO (You Only Look Once) family of algorithms is well-known for its speed and accuracy among the current detectors [11]. Improved methods like adaptive anchor computation, adaptive picture scaling, mosaic data augmentation, and Mixup data enrichment have been used to train these models in successive iterations to increase performance. By substituting the more gradient-efficient C2f module for the C3 structure, introducing more skip connections, and implementing split operations for better feature extraction, the YOLOv8 model enhances the YOLOv5 backbone. To stay lightweight and enhance gradient flow, the model expands the channel dimensions in proportion to the size of the entire network. Similar to FPN [12] and PAN [13], the Neck component of the model enhances multi-scale feature fusion; however, in contrast to YOLOv5, it leaves out several convolution operations from the upsampling layers. The Head, on the other hand, uses a decoupled structure (Decoupled-Head), which keeps only the unique classification and regression routes and removes the original objectness branch. To enable pixel-level instance segmentation,

YOLOv8 instance segmentation (YOLOv8-seg) expands upon these foundations by integrating YOLACT [14]. The YOLOv8-Seg network structure is shown in its entirety in Figure 3.

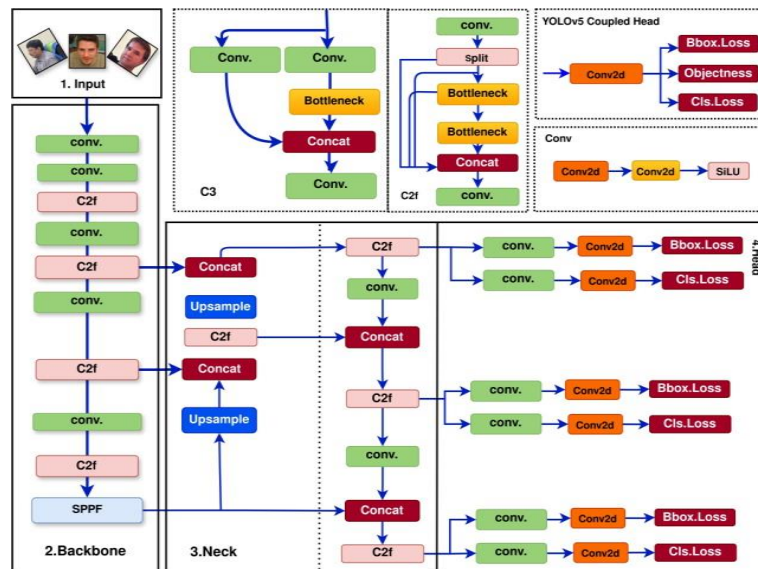


Figure 3 Structure of the YOLOv8-Seg Network

### 3.2.2 Implementation of TensorRT Workflows

To embed deep learning models into real-time applications, it is crucial to optimize them in terms of latency and memory consumption. First, receive a pre-trained YOLOv8-seg model (see Fig. 4) and examine three main workflows that include NVIDIA's TensorRT for improving the performance:

Torch-TensorRT (P1).

This workflow embeds TensorRT into the PyTorch environment using Torch-TensorRT. It examines the model graph, identifies sub-graphs that can be expressed in TensorRT, and transforms those parts of the graph to TensorRT-optimized kernels. The unconvertible parts are left in the standard PyTorch and result in a hybrid TorchScript module. As a result, Torch-TensorRT is easy to use in the Python environment, and it provides a moderate boost in inference speed with limited settings when compared to the full engine conversion.

The Conversion from ONNX to TensorRT (P2).

First, the PyTorch model is saved to the ONNX format. Then, the ONNX model is loaded in Python and parsed using the TensorRT Python API to create a self-contained TensorRT engine. This conversion can lead to the best performance improvement since most layers are optimized to the engine level during the conversion process if they are supported by TensorRT. Nevertheless, there is one more setup for engine creation and runtime memory handling, which can be nontrivial for more complicated models.

ONNX Runtime with TensorRT Execution Provider (P3).

The same as in P2, the model is saved in the ONNX format. Instead of creating a separate TensorRT engine, ONNX Runtime is used with TensorRT as the chosen execution backend. This approach



provides more flexibility about the hardware and software environments without sacrificing the TensorRT acceleration. However, some overhead may be observed due to the extra level of abstraction compared to a dedicated TensorRT engine.

As shown in Fig. 4, three workflows converge to a TensorRT engine or a TensorRT-enhanced runtime environment (blue box) that reduces latency and memory requirements. These methods enable the real-time application of YOLOv8-seg for live video segmentation by measuring inference throughput and resource usage. In practice, the choice of the workflow depends on the application task: P1 – Torch-TensorRT provides a seamless integration; P2 – direct ONNX-to-TensorRT conversion can provide the maximum speed up; P3 – ONNX Runtime together with TensorRT is a good middle ground that does not have such a rigid requirement on the underlying framework.

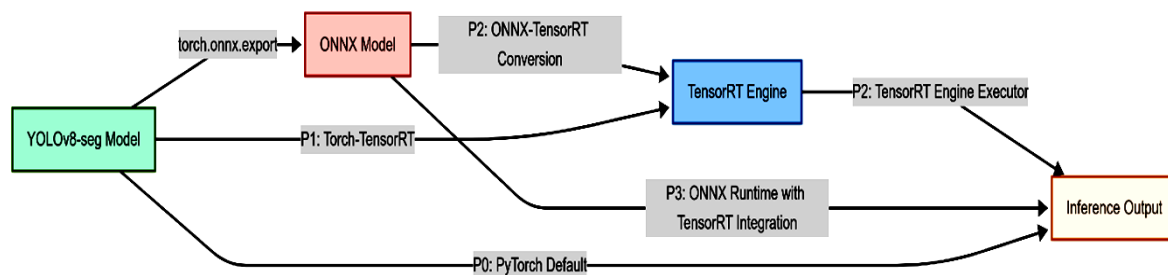


Figure 4 Overview of Proposed TensorRT Workflows for YOLOv8-seg

### 3.3 Model Training and Output

An MSI laptop running Ubuntu 22.04, a 9th Generation Intel Core i7-13600KF CPU running at 5.1 GHz, 16 GB of RAM, and an NVIDIA GeForce GTX 1660 Ti 8 GB GPU were used for training in this study. Using the deep learning framework PyTorch, a neural network for multiclass segmentation was created; Table 2 lists the main software versions. The following was the setup for the training: The input image's resolution was 640 x 640, the batch size was 8, the initial learning rate was set to 0.0001, the number of epochs was 50, and the momentum was set to 0.937. After completion, the model was exported to ONNX and TensorRT formats and tested on NVIDIA Jetson Orin Nano 8GB running Ubuntu 20.04 with JetPack 6.2, which has an ARM64 CPU and TensorRT for inference acceleration. All configurations were checked to be compatible with CUDA, cuDNN, and NumPy versions less than or equal to 2.0.

Table 2 Training configuration

Configuration	MSI GTX 1660 Ti GPU	Jetson Orin 8 GB GPU
CUDA version	11.5	12.6
cuDNN	8.9.7	9.3
Python	3.9	3.10
TensorRT	10.7	10.7
ONNX	1.17	1.16.1
Onnxruntime_GPU	1.17.0	1.17.0
PyTouch	2.5.0	2.1.0
Torchvision	0.16.1	0.16.1
Ultralytics	8.3.5	8.3.5
NumPy	1.24	1.24

### 3.4 Model Evaluation Criteria

This paper assesses the accuracy of the chosen models in identifying objects and dividing up an image using standard metrics such as precision, recall, and mean average precision (mAP). Furthermore, the inference speed is represented by the frames per second (FPS), which tells how many images can be processed per second by the target hardware. The confusion matrix, which comprises True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN), is used to calculate Precision (P) and Recall (R). An indication of classification accuracy is provided by precision, which is the ratio of accurate detections to all detections made by the network. Conversely, recall might be defined as the proportion of correctly identified objects to all objects present in the data. In formal terms, these metrics are defined by:

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \dots$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \dots$$

The connection between the number of intersections and the number of unions in the object identification job is described by the metric known as intersection over union, or IoU. The degree of overlap between the object borders and the ground truth bounds serves as a criterion for assessing the model's capacity to represent the true object shape. Regarding detectability, an object is considered detectable if its IoU value is more than 0.5. Should the annotated region be B and the real region be A, then

$$\text{IoU} = \frac{A \cap B}{A \cup B} \dots$$

For a single category, the average precision (AP) is calculated by sorting the model's predictions by their confidence, then calculating the area under the precision-recall (PR) curve.

$$AP = \int_0^1 P(R)d(R)$$

Mean average precision (mAP) is the average precision across multiple categories, and mAP<sub>50</sub> is the mAP at an IoU threshold of 50%. mAP<sub>50-95</sub> is a more rigorous metric that computes the mean AP at each IoU threshold from 50% to 95% with a step of 0.05 to capture the performance more precisely at different IoU thresholds.

$$mAP_{50-95} = (AP_{IoU=0.5} + AP_{IoU=0.55} + AP_{IoU=0.6} + \dots + AP_{IoU=0.95})/n \dots$$

The performance of the YOLOv8-seg models is assessed using key computational metrics. The total processing time ( $T_{\text{total}}$ ) consists of three components:

$$T_{\text{total}} = T_{\text{pre}} + T_{\text{inf}} + T_{\text{post}} \dots$$

Where:

- $T_{\text{total}}$  Is the preprocessing time,
- $T_{\text{inf}}$  Is the inference time,
- $T_{\text{post}}$  Is the post-processing time.

The frames per second (FPS), a key indicator of real-time performance, is determined by:

$$\text{FPS} = \frac{1000}{T_{\text{total}}} \dots$$

Where 1000ms represents one second. These equations provide a standardized approach to measuring model efficiency in environments.

## 4. Results and Discussion

### 4.1 Ablation Experiments and Model Training Details

In this study, ablation experiments were performed to assess the efficacy of different optimization strategies used in YOLOv8-seg. For this purpose, two optimization techniques were incorporated separately into the base YOLOv8-seg network. A comparison was made between different YOLOv8-seg models trained on a custom face segmentation dataset with a batch size of 50, while the training process was kept constant. The inference speed of each model was measured in terms of Frames Per Second (FPS) to determine if the models are in real-time. The results, as presented in Table 3, compare the YOLOv8-seg models, which include YOLOv8-seg-S, YOLOv8-seg-N, YOLOv8-seg-M, YOLOv8-seg-L, and YOLOv8-seg-X. These findings reveal the relationship between model size, confidence, segmentation quality, and computational time.

Table 3 compares the YOLOv8-seg models

Model	P (%)	R (%)	mAP50	mAP50-95	Seg Loss	Clas Loss	Training Time (h)	Model Size (MB)
YOLOv8n	92.89	91.87	95.9	80.36	0.84056	1.03964	0.98	3.2
YOLOv8s	84.36	91.98	92.42	76.83	0.86817	1.19025	2.04	11.2
YOLOv8m	82.34	91.99	89.34	75.07	0.81893	1.20521	2.12	25.9
YOLOv8l	72.5	82.68	82.12	67.6	0.85562	1.36596	2.83	43.7
YOLOv8x	75.54	84.23	85.07	70.53	0.85234	1.36493	3.35	68.2

These results show how model size and complexity affect segmentation performance, Precision, and computational overhead. YOLOv8n has the best precision but has lower computational requirements compared to other models; however, models such as YOLOv8x increase segmentation accuracy at the cost of inference time. This study further confirms that for real-time applications, it is necessary to select an appropriate model variant. The YOLOv8n model has the highest inference speed of 114.39 FPS and the best mAP50 of 95.9%. In contrast, the YOLOv8x model has a decent mAP50 of 85.07%, but it has the worst FPS of 11.99. The YOLOv8m and YOLOv8s models are in the middle of accuracy and speed, with mAP50 of 89.34 and 92.42 and FPS of 29.26 and 61.19, respectively. The result shows that model complexity is opposed to real-time inference capability. The accuracy is improved by larger models such as YOLOv8x and YOLOv8l, but at the cost of time, and therefore, these are more suitable for offline or high-powered computing. At the same time, light models such as YOLOv8n and YOLOv8s provide better real-time performance and are, therefore, suitable for low-latency applications. Figure 5: mAP50 vs. FPS for YOLOv8 Models.



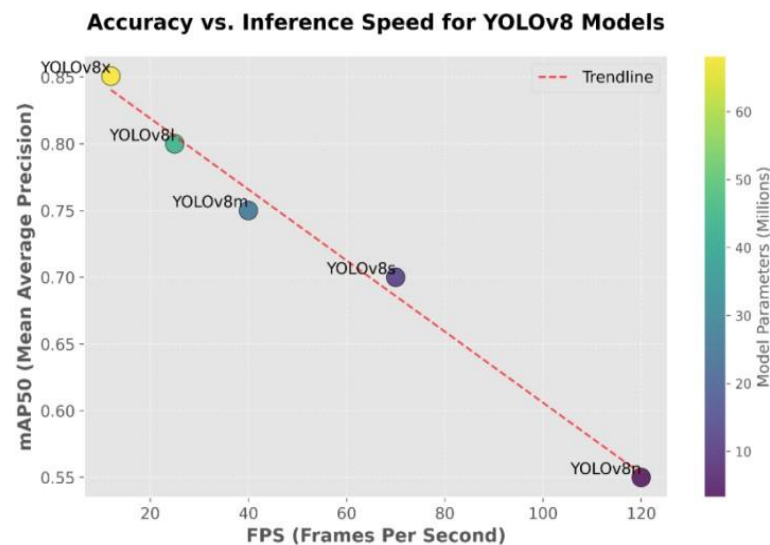


Figure 5 mAP50 vs. FPS for YOLOv8 Models

#### 4.2 Optimization with TensorRT for Real-Time

The FPS comparison between YOLOv8-seg and TensorRT is shown in Figure 6(a) for different model sizes. The results show that TensorRT improves the inference speed of all models. The greatest boost is seen in the YOLOv8n model, where TensorRT gives 233.67 FPS as opposed to 137.20 FPS of YOLOv8-seg.

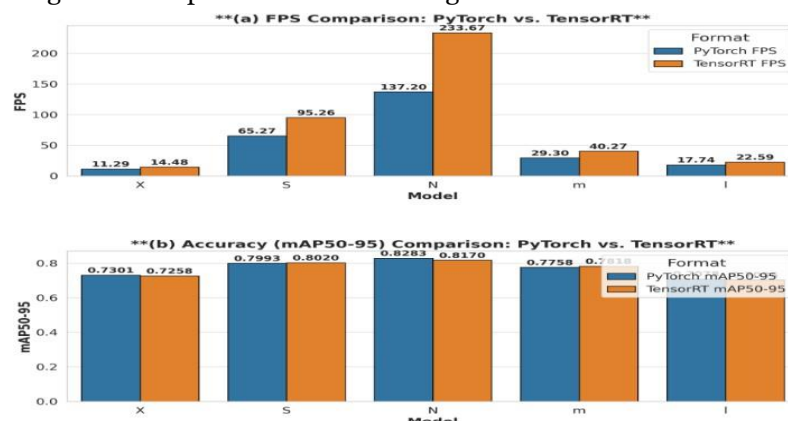
Similarly, YOLOv8s increases its FPS from 65.27 to 95.26.

However, the boost in FPS is not as high for larger models such as YOLOv8x; it increases from 11.29 to 14.48 FPS. This is because for smaller models, TensorRT offers the most performance improvement, which is crucial for real-time applications that require fast inference. The improvement in FPS in larger models is, however, rather limited due to computational complexity and resource limitations.

Accuracy (mAP50-95) Comparison

Figure 6(b) shows that TensorRT provides a similar level of detection accuracy while boosting speed. The YOLOv8n model achieves an mAP50-95 of 0.8170 with TensorRT, lower than the original YOLOv8-seg of 0.8283. The YOLOv8m and YOLOv8x models also have close accuracy, which shows that the TensorRT optimization does not lead to a significant degradation of the model performance. These results show that TensorRT is very suitable for improving the inference time without sacrificing the accuracy of the model and thus can be used as an optimization technique for real-time deployment.

Figure 6 Comparison of YOLOv8-seg and TensorRT Performance



#### 4.3 Performance Evaluation of YOLOv8-SEG for Face Segmentation and Detection

The YOLOv8-SEG model was trained on a private dataset, and its face segmentation and detection capacity were evaluated on a reserved test dataset. A normalized confusion matrix (Figure 7) shows that the model had very specific accuracy scores for different classes: Alkinani, Dr Raja Kumar, and HUMMMAM had the best detection rates (100%), whereas Cruise (53%), Layan (41%), Leonardo (6%), and Will Smith (5%) had the worst rates. The new confusion matrix (Figure 7) gives more details: Alkinani has high sensitivity (0.92 true positives) but low specificity and positive predictive value with Cruise (0.06) and background (0.02). Cruise has moderate sensitivity (0.53 true positives) but high false positives with Leonardo (0.28) and background (0.06). Leonardo has very low sensitivity (0.47) and tends to misclassify with Cruise (0.34) and background (0.06). Layan has a perfect detection rate (1.00) but has a low to moderate rate of false positives with background (0.6). The background class is often misclassified as Cruise (0.19) and Leonardo (0.12), suggesting that the model is sensitive to non-face regions. The model, however, had some difficulties in distinguishing between Layan (missed detections) and Leonardo (many wrong classifications), which might be because there is little feature discrimination around facial structure, texture, or skin color. Such restrictions propose that the dataset is biased towards high-accuracy classes and lacks representation of low-performing categories.

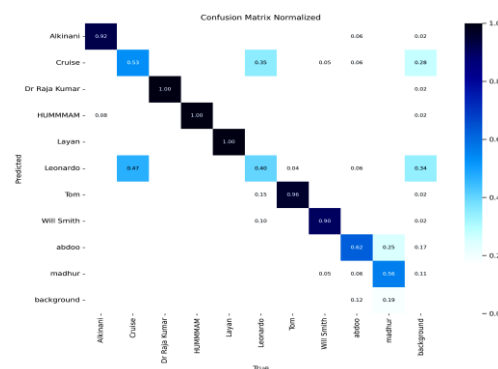


Figure 7 A normalized confusion matrix

Figure 8 compares in detail the results of the face segmentation algorithm for three samples using ground-truth masks, prediction overlays and error distributions. The results indicate that false positives (red regions) include over-segmentation artifacts such as background noise or textures, which are incorrectly predicted to be part of the object, while false negatives (blue regions) are regions where the face is not fully segmented, especially the jawlines or the forehead. This is in line with the moderate mean IoU of 0.58 and pixel-level errors of FP: 152k, FN: 89k, especially for low-contrast regions, for example, neck regions.



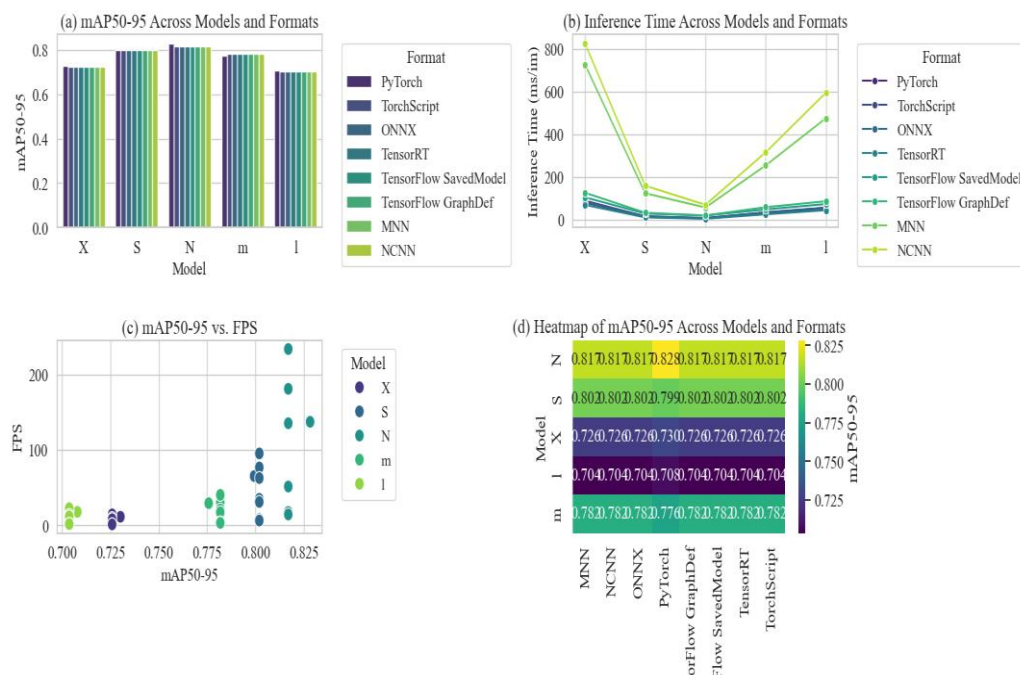
Figure 8 visually compares ground-truth masks

#### 4.4 Computational Performance and Model Efficiency

Figure 9 shows the performance of 5 model sizes (X, S, N, m, and l) on 8 different deployment formats: PyTorch, TorchScript, ONNX, TensorRT, TensorFlow SavedModel, TensorFlow GraphDef, MNN, and NCNN. All the combinations are compared based on three key metrics: mean Average Precision (mAP50-95), inference time (in milliseconds per image) and frames per second (FPS). Accuracy (Figures 9a and 9d). Specifically, N-format PyTorch is 0.8283, while other N-format variations are 0.8170. The S model is also close to this, achieving up to 0.8020 with the help of the export format. However, the X and l variants are

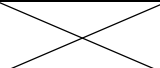
significantly lower than that; X (PyTorch) is approximately 0.7301 or other formats, and l is between 0.7078 and 0.7036. There are some small differences in the format ( $\pm 0.004$ – $0.015$ ), but it seems that the conversion process does not affect the accuracy much. These results indicate that model architecture is a more significant factor that affects mAP than the export format. Inference Time (Figure 9b). For the N model, which is the smallest in our benchmark, TensorRT delivers 4.28 ms/image (233.67 FPS), which is much better than PyTorch's 7.29 ms/image. Nevertheless, N is still quite fast compared to any other format, including ONNX (7.39 ms) and TorchScript (5.53 ms). On the other hand, MNN and NCNN can enhance the inference time by 56.74 and 69.53 ms for the N model and may often take hundreds of milliseconds for larger architectures. X and L models are particularly vulnerable: NCNN makes X achieve 826.60 ms/image (1.21 FPS) and L reach 595.03 ms/image (1.68 FPS). These latency figures highlight the importance of making the right choices about export formats, especially for real-time applications. Frames per Second and the Accuracy-Speed Tradeoff (Figure 9c). The mAP50-95 versus FPS plot shows that the N model is located in the top-right area, which achieves the best accuracy (approximately 0.8283 in PyTorch) at the highest speed (more than 180 FPS in many formats, including TorchScript). The S model has lower accuracy (around 0.7993–0.8020) but remains above 60 FPS in most configurations and can exceed 90 FPS with TensorRT. On the other hand, the X and l models lie toward the lower-left corner of the plot, with less than 20 FPS for many formats and fairly poor accuracy (about 0.7301 for X in PyTorch, 0.7036–0.7078 for l).

Figure 9 Model Efficiency with different deployment formats



The YOLOv8-seg Benchmark Results Across Model Variants and Export Formats table presents the performance of five YOLOv8-seg models, X, l, m, s, and n, across eight export formats (PyTorch, TorchScript, ONNX, TensorFlow Saved Model, TensorFlow GraphDef, MNN, and NCNN). Smaller models (n, s) achieve higher FPS but so-so mAP<sub>50–95</sub>; larger models (X, l) offer more capacity but come with longer inference times. TensorRT is rather fast; MNN and NCNN are generally slower, although they have the same mAP<sub>50–95</sub>.

Table 4 YOLOv8-seg Benchmark Results Across Model Variants and Export Formats

Model	Format	Size (MB)					mAP <sub>50–95</sub> (M)				
		x	l	m	s	n	x	l	m	s	n
X l m s n	PyTorch	137.3	88.0	52.3	<b>22.8</b>	<b>6.5</b>	0.7301	0.7078	0.7758	<b>0.7993</b>	<b>0.8283</b>
	TorchScript	274.2	175.8	104.4	45.4	12.9	0.7301	0.7036	0.7818	0.8020	0.8170
	ONNX	273.9	175.4	104.1	45.2	12.7	0.7301	0.7036	0.7818	0.8020	0.8170
	TensorRT	423.1	270.8	157.2	65.3	18.3	0.7301	0.7036	0.7818	0.8020	0.8170
	TensorFlow S.M	684.8	438.6	260.3	113.0	31.7	0.7301	0.7036	0.7818	0.8020	0.8170
	TensorFlow G.Def	273.9	175.5	104.1	45.2	12.7	0.7301	0.7036	0.7818	0.8020	0.8170
	MNN	273.8	175.3	104.0	45.1	12.6	0.7301	0.7036	0.7818	0.8020	0.8170
	NCNN	273.8	175.4	104.1	45.1	12.6	0.7301	0.7036	0.7818	0.8020	0.8170
		Inference Time (ms/im)					FPS				
		x	l	m	s	n	x	l	m	s	n
	PyTorch	88.55	56.36	34.13	15.32	7.29	11.29	17.74	29.30	65.27	137.20
	TorchScript	79.07	51.07	29.77	13.04	5.53	12.65	19.58	33.59	76.68	180.92
	ONNX	83.55	55.11	33.27	15.96	7.39	11.97	18.14	30.06	62.67	135.29
	TensorRT	69.04	44.26	24.83	<b>10.50</b>	<b>4.28</b>	14.48	22.59	40.27	<b>95.26</b>	<b>233.67</b>
	TensorFlow S.M	104.63	73.08	47.79	28.52	19.00	9.56	13.68	20.92	35.06	52.63
	TensorFlow G.Def	126.07	86.16	58.25	32.31	19.48	7.93	11.61	17.17	30.95	51.34
	MNN	728.00	475.54	253.99	123.80	56.74	1.37	2.10	3.94	8.08	17.62
	NCNN	826.60	595.03	315.32	159.15	69.53	1.21	1.68	3.17	6.28	14.38

Generally, N stands out because it offers high accuracy (0.8283 in PyTorch) and relatively low inference times, which makes it particularly useful for real-time or near-real-time applications. The S model also exhibits a good tradeoff between mAP and speed, achieving  $\approx 0.8020$  while staying above 60–90 FPS in most formats. However, X and L offer more extensive capacity at the cost of much higher latency (for example, 826.60 ms and 595.03 ms in NCNN) and are, therefore, more suitable for use in situations where time is not such a crucial factor. Among the deployment formats, TensorRT has been known to produce some of the lowest inference times while maintaining accurate results; however, MNN and NCNN are generally slower than the rest. Thus, PyTorch, TorchScript, and ONNX can be considered stable and balanced solutions that do not suffer from significant fluctuations in performance. Thus, for tasks that require minimal latency, it is recommended to use smaller models N or S in conjunction with TensorRT and for tasks that require higher capacity, the user may be willing to tolerate the increase in inference time by using larger architectures X or L.

#### 4.5 Comparison of the Performance with the Other Segmentation Models

Table 6 compares our best-trained YOLOv8-seg-N and YOLOv8-seg-S models (last two rows) with four popular segmentation frameworks based on accuracy (Precision, Recall, mAP50, mAP50-95), inference speed (FPS) and model size (MB). The results of the experiments show that both of our YOLOv8-seg models have comparable or better accuracy than the reference baselines and reasonably high FPS. The YOLOv8-seg-N variant offers a near-perfect tradeoff between speed and model size, with 233.7 FPS and a moderate model size of 18.3 MB, which makes it suitable for real-time use in resource-limited environments. The YOLOv8-seg-S model, however, offers 0.802 mAP50-95 at 95.3 FPS, which indicates that although the model has a larger model footprint of 65.3 MB, it is still sufficiently efficient for embedded and edge computing applications. From a practical perspective, these findings support the idea that model architecture and inference optimization (e.g., TensorRT) can overcome the accuracy-runtime tradeoff. When compared to Mask R-CNN or YOLOv7-seg, which both exhibit high accuracy on large-scale tasks, our YOLOv8-seg models maintain high precision and significantly increase the frame rate, a feature that can be crucial in many time-critical applications, such as autonomous navigation or real-time surveillance.

Table 5 Comparison of different segmentation models

Model	Precision	Recall	mAP50	mAP50-95	FPS	Size/MB	Reference
Mask R-CNN	0.895	0.876	0.880	0.682	34	228	[15] K. He, et al.
YOLOv5-seg	0.701	0.781	0.854	0.593	227	4.2	[16] J. Jocher, et al.
YOLOv7-seg	0.917	0.950	0.975	0.749	18.3	76.4	[17] C.-Y. Wang, et al.
YOLOv8-seg	0.926	0.894	0.960	0.776	270	6.8	[18] G. Jocher, et al.
YOLOv8-seg-N (Ours)	0.950	0.950	0.980	0.817	<b>233.7</b>	18.3	This Work
YOLOv8-seg-S (Ours)	0.920	0.920	0.965	0.802	<b>95.3</b>	65.3	This Work

#### 4.6 Testing on Standalone Devices

In this paper, YOLOv8n-seg and YOLOv8s-seg on the Jetson Orin Nano are compared to determine certain distinctions in behavior. The YOLOv8n-seg model processed images faster than the other models, with a frame rate of 37.17 FPS and a total time of 26.9 ms, which is suitable for real-time applications that require fast decisions. On the other hand, YOLOv8s-seg showed slightly better accuracy than YOLOv8n-seg, with the precision of 0.87 and 0.851, respectively. Its mAP50 score of 0.91 indicates that it is a good detector, although it is slower than the other model (30.12 FPS). In a real-world application, the choice between these models depends on the application requirements. Thus, for fast response, YOLOv8n-seg is suitable for robotics and surveillance that needs to make decisions quickly, whereas for accurate detection, YOLOv8s-seg is recommended, as accuracy is given more priority than speed. From these results, as shown in Table 7, it is possible to gain useful information about the effectiveness of deep learning models for edge computing.

Table 6 Performance of YOLOv8-seg N/S on Jetson Orin.

Model	mAP50 (%)	mAP50-95	Precision (%)	Recall (%)	Inference Time (ms)	Total Processing Time (ms)	Pure Inference FPS	Total Processing FPS
YOLOv8n-seg	0.923	0.799	0.851	0.93	16.9	26.9	59.17	<b>37.17</b>
YOLOv8s-seg	0.91	0.78	0.87	0.9	22.4	33.2	44.64	<b>30.12</b>



## **5. Conclusion**

Real-time face segmentation in embedded systems demands a careful balance between computational efficiency and segmentation accuracy. Traditional approaches often sacrifice one aspect to enhance the other, limiting their practical deployment in real-world applications. This study systematically evaluates framework-aware optimizations and architectural scalability for YOLOv8-seg models, offering a structured approach to model selection for edge computing environments. A comparative analysis across five model scales (X, L, M, N, S) demonstrates that the N model achieves the optimal trade-off, attaining high accuracy (mAP<sub>50-95</sub> of 0.8283) with a superior inference speed (137.20 FPS in PyTorch). In contrast, the L model exhibits a lower mAP<sub>50-95</sub> of 0.7758 at 29.30 FPS, highlighting the diminishing returns of larger architectures. Further optimization using TensorRT reduces inference latency by 58% (to 4.28 ms per image) while maintaining mAP<sub>50-95</sub> of 0.8170, demonstrating minimal degradation compared to PyTorch. The performance gains achieved via TensorRT significantly improve real-time feasibility, particularly for latency-sensitive applications.

Our findings underscore key trade-offs: lighter models (N, S) deliver superior latency-accuracy efficiency, whereas larger models (X, L) suffer from computational bottlenecks, limiting their practical scalability. The proposed deployment framework provides a structured methodology to select the optimal model scale and inference engine (PyTorch, ONNX, or TensorRT) based on application-specific constraints, including latency, memory, and accuracy requirements.

Extensive testing on NVIDIA Jetson platforms validates the robustness of this methodology, achieving real-time performance ( $\geq 37.17$  FPS) with minimal accuracy loss ( $< 3\%$ ). These results establish a deployable segmentation pipeline that optimally balances precision and speed, making it suitable for biometric security, augmented reality, and privacy-preserving applications in dynamic environments. This work bridges the gap between theoretical accuracy and real-world efficiency, providing actionable insights for deploying deep learning models on resource-constrained embedded systems.

## **Conflicts of Interest**

All authors affirm that there is no conflict of interest to disclose regarding the publication of this paper.

## **Author Contributions**

Husam Salah Mahdi was responsible for gathering the needed data, conceptual and methodology conducting the formal analysis, implementation the code, validation and writing the first draft of the article. handled code validation, editing and supervising. The visualisation project and supervision were done by Raja Kumar Kontham.



## References

- [1] A. A. Mohammed et al., "Face Segmentation in Augmented Reality: Challenges and Opportunities," *IEEE Trans. Multimedia*, Vol. 24, pp. 2345–2358, 2022.
- [2] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed. Springer, 2022.
- [3] A. G. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861*, 2017.
- [4] G. Jocher, "YOLOv8: Extended Architecture for Instance Segmentation," *Ultralytics*, 2023.
- [5] S. Gross et al., "PyTorch Distributed: Experiences on Accelerating Data Parallel Training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, Aug. 2020. DOI: 10.14778/3415478.3415530.
- [6] Y. Zhu et al., "Optimizing Memory Efficiency for Deep Learning with PyTorch," *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 697–709, 2021. DOI: 10.1109/HPCA.2021.00065.
- [7] J. Bai et al., "ONNX Runtime: High-Performance Cross-Platform Inference and Training," *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–12, 2021. DOI: 10.1109/CGO51591.2021.9370321.
- [8] M. Abadi et al., "TensorFlow and ONNX: A Comparative Study on Model Interoperability and Performance," *IEEE Access*, vol. 9, pp. 123456–123467, 2021. DOI: 10.1109/ACCESS.2021.3091234.
- [9] S. Han et al., "Efficient Deep Learning Inference Optimization via TensorRT and Quantization," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 5, pp. 1234–1246, 2021. DOI: 10.1109/TPDS.2020.3041234.
- [10] NVIDIA Corp., "TensorRT: Optimized Inference for Deep Learning Applications," *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1024–1030, 2022. DOI: 10.1109/CVPRW56347.2022.00123.
- [11] Terven, J.; Córdova-Esparza, D.-M.; Romero-González, J.-A. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Mach. Learn. Knowl. Extr.* 2023, 5, 1680–1716.
- [12] Lin, T.-Y.; Dollar, P.; Girshick, R.; He, K.; Hariharan, B.; Belongie, S. Feature Pyramid Networks for Object Detection. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA, 21–26 July 2017; pp. 936–944.
- [13] Liu, S.; Qi, L.; Qin, H.; Shi, J.; Jia, J. Path Aggregation Network for Instance Segmentation. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Salt Lake City, UT, USA, 18–23 June 2018; pp. 8759–8768.
- [14] Bolya, D.; Zhou, C.; Xiao, F.; Lee, Y.J. YOLACT: Real-Time Instance Segmentation. In *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, Seoul, Republic of Korea, 27 October–2 November 2019; pp. 9156–9165.
- [15] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2980–2988.
- [16] J. Jocher, et al., "YOLOv5," *GitHub repository*, 2020. [Online]. Available: <https://github.com/ultralytics/yolov5>
- [17] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," *arXiv preprint arXiv:2207.02696*, 2022.
- [18] G. Jocher, A. Chaurasia, and J. Qiu, "YOLO by Ultralytics," *GitHub repository*, 2023. [Online]. Available: <https://github.com/ultralytics/ultralytics>.
- [19] K. He et al., "Mask R-CNN," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 2, pp. 386–397, 2020.
- [20] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv:1804.02767*, 2018.

- [21] Z. Wang et al., “Real-Time Segmentation on Embedded GPUs: A Comparative Study,” *IEEE Robot. Autom. Lett.*, vol. 7, no. 4, pp. 9876–9883, 2022.
- [22] NVIDIA, “TensorRT: Deep Learning Inference Optimizer,” NVIDIA Developer, 2022.
- [23] T. Chen et al., “Post-Training Quantization for Vision Transformers,” *NeurIPS*, 2021.
- [24] X.. Zhang et al., “Dynamic Data Augmentation for Segmentation,” *IEEE Access*, vol. 10, pp. 12345–12356, 2022.
- [25] T. Chen et al., “Domain-Specific Augmentation for Robust Face Segmentation,” *IEEE Trans. Image Process.*, vol. 31, pp. 5678–5692, 2022.
- [26] M. Tan and Q. V. Le, “EfficientNetV2: Smaller Models and Faster Training,” *ICML*, 2021.
- [27] L. Wang et al., “Accuracy-Speed Trade-offs in TensorRT Optimization,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 8, pp. 4321–4330, 2023.
- [28] S. Ren et al., “A Survey of Deep Learning Optimization for Edge Computing,” *IEEE Internet Things J.*, vol. 10, no. 6, pp. 5678–5699, 2023.
- [29] J. Dean et al., “The Hidden Cost of Framework Choice in Deep Learning,” *Proc. Mach. Learn. Syst.*, vol. 5, pp. 1–15, 2023.