

# Towards Real-Time and Efficient Perception Workflows in Software-Defined Vehicles

Sumaiya, Reza Jafarpourmarzouni, Yichen Luo, Sidi Lu *Member, IEEE* and Zheng Dong *Member, IEEE*

**Abstract**—With the growing demand for software-defined vehicles (SDVs), deep learning-based perception models have become increasingly important in intelligent transportation systems. However, these models face significant challenges in enabling real-time and efficient SDV solutions due to their substantial computational requirements, which are often unavailable in resource-constrained vehicles. As a result, these models typically suffer from low throughput, high latency, and excessive GPU/memory usage, making them impractical for real-time SDV applications.

To address these challenges, our research focuses on optimizing model and workflow performance through the integration of pruning and quantization techniques across various computational environments, utilizing frameworks such as PyTorch, ONNX, ONNX Runtime, and TensorRT. We systematically explore and evaluate three distinct pruning methods in combination with multi-precision quantization workflows (FP32, FP16, and INT8) and present the results based on four evaluation metrics: inference throughput, latency, GPU/memory usage, and accuracy. Our designed techniques, including pruning and quantization, along with optimized workflows, can achieve up to  $18\times$  faster inference speed and  $16.5\times$  higher throughput, while reducing GPU/memory usage by up to 30%, all with minimal impact on accuracy. Our work suggests using the Torch-ONNX-TensorRT workflow quantized with FP16 precision and group pruning as the optimal strategy for maximizing inference performance. It demonstrates great potential in optimizing real-time, efficient perception workflows in SDVs, contributing to the enhanced application of deep learning models in resource-constrained environments.

**Index Terms**—Software-defined vehicle, real-time, pruning, quantization, workflow, FP32, FP16, INT8, throughput, latency, GPU/memory usage, accuracy.

## I. INTRODUCTION

**ROLE of Vision-Based Systems in SDVs.** With the continuous advancements in computer vision, decision-making, and control technologies, software-defined vehicles (SDVs) have become a highly significant topic in intelligent transportation [1]. These SDVs rely on precise, low latency, high throughput, and GPU/memory efficient applications/software to manage a wide range of functions throughout the vehicle's lifecycle [2], [3]. As a key component of SDVs, vision-based perception systems play a vital role in gathering environmental information and providing the necessary per-

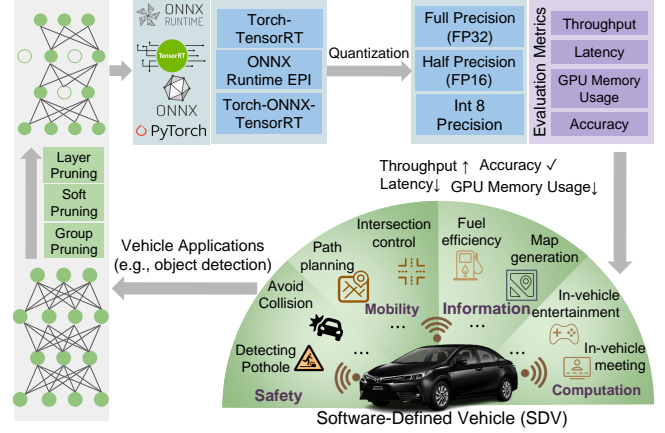


Fig. 1: Overview of the software-defined vehicles (SDVs), integrating object detection model undergoes techniques like pruning, quantization, and frameworks like TensorRT ONNX Runtime, for the optimized model inference performance.

ception for route planning and decision-making [4]. These systems use cameras to gather real-time information about the surrounding environment and enhance driving safety by quickly alerting drivers to unusual conditions through the detection of objects like vehicles and pedestrians, which is crucial for SDVs.

**Market Growth and Background.** The global SDV market is valued at around \$213 billion in 2024 and is projected to reach \$1.2K billion by 2030. Despite its great potential, ensuring the efficiency and accuracy of vehicle software remains challenging. A recent report noted nearly 4,000 incidents in 2023, many linked to high latency and slow throughput. Large, over-parameterized models—ranging from 61 million to over 2 billion parameters,—are a key contributor to these issues, with models processing up to 40 TB of data in just 8 hours of driving, leading to excessive GPU memory consumption and delayed response in SDVs.

**Limitations of Perception System Algorithms.** Perception systems are the key component of SDVs. The evolution of perception systems algorithms has transitioned from earlier two-stage methods, such as R-CNN [5], [6], SPP-Net [7], Fast R-CNN [8], and Faster R-CNN [9], to the more popular one-stage methods, such as Single Shot MultiBox Detector (SSD) [10] series and the You Only Look Once (YOLO) [11], [12] series. The entire YOLO series, from YOLOv1 [11], YOLOv2 [13], YOLOv3 [14] to the most popular version for real-world application YOLOv5 [15], are all one-stage detectors. Two-stage networks are renowned for their fine accuracy but their slower processing speeds, making them less suitable for real-time applications in SDVs [16]. In contrast, one-stage detectors focus on speed, making them more suitable

This work was supported in part by the U.S. National Science Foundation under Grants CNS-2103604, CNS-2140346, CNS-2231523, CNS-2348151 and Commonwealth Cyber Initiative grant HC-3Q24-048.

Sumaiya is with the Computer Science Department, Wayne State University, Detroit, MI 48202 (email: sum@wayne.edu).

Reza Jafarpourmarzoun is with the Computer Science Department, Wayne State University, Detroit, MI 48202 (email: rezajafarpour@wayne.edu).

Yichen Luo is with the Computer Science Department, William & Mary, Williamsburg, VA 23185 (email: yluo11@wm.edu).

Sidi Lu is with the Computer Science Department, William & Mary, Williamsburg, VA 23185 (email: sidi@wm.edu).

Zheng Dong is with the Computer Science Department, Wayne State University, Detroit, MI 48202 (email: dong@wayne.edu).

for real-time SDV applications. However, this speed comes at the cost of reduced accuracy, especially in complex scenes involving small or overlapping objects. For example, YOLO, a type of one-stage network, may struggle to achieve the precision required for safety-critical systems. Hence, the trade-off between **accuracy** and **latency** is a key challenge in SDV perception systems.

**Challenges in Real-World SDVs.** In addition to the trade-off between accuracy and latency [17], real-world applications of deep learning-based perception system models also face significant challenges in **throughput** [18] and **GPU/memory usage** [19], which are key factors for the safety and effectiveness of SDV applications, due to the increasing size and complexity of the involved models. While larger models often require substantial processing power and memory, such resources may not be readily available in the constrained environments of SDVs. These limited computational capabilities can lead to bottlenecks in processing. For example, the growing size of models like YOLOv5 increases computational demands and results in higher GPU memory consumption, presenting an additional obstacle for resource-constrained SDV systems. Hence, optimization is necessary not only to maintain the safety and reliability of the vehicles but also to ensure that the advanced capabilities of these models can be effectively utilized in real-world scenarios.

**Insights of Optimization Techniques.** To overcome the aforementioned challenges in SDVs [20], a range of optimization strategies have been developed. One of the most significant advancements is the use of NVIDIA GPUs, which enable parallel processing [21] to accelerate execution, further enhanced by CUDA optimizations [22]. In addition, there are various hardware acceleration techniques [23] like ONNX [24], TensorRT [1], [25], and ONNX Runtime EPI (Execution Provider Interface) [26], which have been introduced to enhance model performance. Another key set of techniques includes model quantization [27], hyper-parameter optimization [28], [29], and model pruning [30]. Model quantization reduces the precision of the model's weights and activations, which decreases the model's size and computational requirements without significantly compromising accuracy. It works by reducing the precision of weights and activations from floating-point formats (e.g., FP32) [31] to lower-precision formats such as FP16 or INT8. On the other hand, model pruning [30], [32] reduces the model's size by removing redundant or less important parameters. By systematically removing these parameters, the model becomes more efficient, leading to faster inference times and reduced memory usage without heavily impacting the model's performance. Both of these techniques are essential in optimizing deep learning models for real-time, resource-constrained applications in SDVs.

**Contributions of This Work.** In this paper, we designed approaches that combine both pruning and quantization techniques, specialized to our designed workflows for various precision modes presented in Fig. 1. These workflows were developed to address the need for precision calibration across different computational environments. By integrating pruning, which systematically reduces model complexity by eliminating less important parameters, with quantization, which lowers the precision of model weights and activations, we aimed to achieve a balanced optimization of model performance.

This approach not only maximizes the inference performance but also reduces the computational load of the models. The combination of these (pruning and quantization) techniques was applied across different precision modes, including FP32, FP16, and INT8, allowing us to thoroughly explore the impact of precision calibration on the overall efficiency and effectiveness of the models in real-world scenarios.

To be concrete, the specific contributions of this work are illustrated as follows:

- Our work presents the effectiveness of enhancing the inference performance of real-time SDV perception algorithms without compromising significantly in accuracy. To achieve this, we utilize video streams captured by the vehicle's cameras to perform a comparative analysis. This involves generating three pruning techniques (e.g., layer pruning, soft pruning, and group pruning) and also determining the optimal pruning method.
- We design and implement three distinct quantization workflows (e.g., Torch-TensorRT, ONNX Runtime Quantization, Torch-ONNX-TensorRT) each supporting three precision modes: FP32, FP16, and INT8. Each pruned models are quantized using these workflows resulting in nine optimized workflows in total that are evaluated across all precision levels.
- We conduct a comprehensive evaluation using four key metrics: inference throughput, inference latency, GPU/memory usage, and accuracy. The optimized workflow can achieve up to  $18\times$  faster inference speed,  $16.5\times$  higher throughput, and reduce the GPU/memory usage by up to 30% without a noticeable drop in accuracy. These metrics were assessed both after pruning (prior quantization) and post-quantization, providing a thorough analysis of the impact of our optimization techniques on model performance.
- Upon our evaluation metrics, we recommend the Torch-ONNX-TensorRT workflow quantized with FP16 precision and group pruning as the ideal solution for achieving maximum inference efficiency (e.g., throughput, latency, and GPU/memory usage) balancing with accuracy in resource-constrained SDVs.

**Organization.** We organize our research throughout the paper as follows. Sec. II provides the background of frameworks, techniques, and methods which is denoted further in our paper. Sec. III presents the experimental design of our paper and a high-level overview of pruning and quantization workflows. In Sec. IV we describe each pruning technique, fine-tuning, and knowledge distillation in detail. Then, We elaborately describe each of our workflows with each precision mode in Sec. V. The Dataset description, System Configuration, evaluation metrics, and experimental results of our research are shown in Sec. VI and Conclusion in Sec. VII.

## II. BACKGROUND AND RELATED WORK

In this section, we present a comprehensive overview of the frameworks employed in our research, emphasizing the key components such as pruning, knowledge distillation, and quantization. We also provide an insightful review of prior research advancements in time-critical real-world applications.

### A. PyTorch

PyTorch [33] is an open-source deep learning framework developed by Facebook's AI Research Lab, known for its flexibility and ease of use. It provides efficient GPU-accelerated tensor operations and an Autograd library for automatic gradient computation.

### B. ONNX

ONNX [34] (Open Neural Network Exchange) is an open standard that enables the transfer of machine learning models between different frameworks, such as PyTorch and TensorFlow, by defining a common model format.

### C. TensorRT

TensorRT [35] is a high-performance deep learning inference optimizer and runtime library developed by NVIDIA. TensorRT optimizes deep learning models using several key techniques. Layer fusion combines multiple layers into a single operation to reduce computation. Precision calibration supports mixed-precision inference (FP16 and INT8), reducing resource use while maintaining accuracy. Kernel auto-tuning selects the best kernels for target hardware, enhancing execution efficiency. Dynamic tensor memory manages memory allocation efficiently, minimizing footprint. FP32, FP16, and INT8 refer to different numerical precision levels used in TensorRT to optimize inference performance.

FP32, or 32-bit floating point, provides the highest level of precision and is typically used during the training phase of deep learning models to ensure accurate gradient calculations. FP16, or 16-bit floating point, offers a middle ground by reducing the data size and computational demands while still maintaining a high level of precision. This reduction in precision allows for faster processing speeds and lower memory usage. INT8, or 8-bit integer, represents an even lower precision level, which drastically reduces both memory and computational requirements. INT8 calibration involves quantizing the model's weights and activations to 8-bit integers, which can lead to significant performance gains.

### D. Pruning

Pruning [36] is a technique that reduces the complexity of neural networks by removing less important or redundant parameters. This process makes models lighter, faster, and more efficient, while ideally maintaining or improving accuracy. By eliminating redundant parameters, pruning [37] significantly reduces model size, making it more suitable for deployment on resource-constrained devices. With fewer parameters, the model requires less memory for storage and execution, which is essential for edge devices with limited memory capacity. Below are key terms associated with pruning:

- **Parameters:** Learnable weights in a neural network. Pruning reduces the total number of parameters by removing those considered unnecessary.
- **Weights:** Weights with minimal contribution to the model's predictions (such as those with small magnitudes) are pruned to create a sparser network.
- **Sparsity:** The ratio of zero-valued weights or pruned connections in a model after pruning. Higher sparsity indicates fewer active weights, resulting in a more compact model.
- **Pruning Ratio:** The proportion of parameters removed from the model during the pruning process.

- **Weight Regularization:** Techniques such as L1 or L2 regularization, applied during training, encourage smaller weights, which can be more easily pruned later.

### E. Quantization

Quantization [38] in deep learning refers to the process of reducing the precision of the numbers used to represent a model's weights and activations. There are two primary methods for model quantization: post-training quantization (PTQ) and quantization-aware training (QAT).

In PTQ [39], a model is first trained using standard, high-precision techniques to achieve the desired accuracy. After training, the model's weights and biases initially represented as 32-bit floating-point numbers, are converted to lower-precision numbers, such as 8-bit integers. PTQ is quicker to implement as it does not require retraining, but may result in a slight accuracy drop. A fine-tuning step is often applied afterward to adjust the quantized weights and biases, aiming to recover any lost accuracy due to the quantization process. QAT [40], on the other hand, integrates quantization into the training phase, allowing the model to adapt to low-precision arithmetic. By simulating quantization effects on weights and activations during training, QAT produces models that are more robust to the precision constraints of deployment hardware. QAT typically yields better performance as it accounts for precision constraints throughout the entire training process.

### F. Knowledge Distillation

Knowledge distillation [41] is a machine learning technique where a smaller model, the "Child," is trained to replicate the performance of a larger model, the "Mother." The goal is to transfer the knowledge from the mother model to the child model using the mother's outputs, typically probability distributions over classes (soft targets) [42]. This technique is especially valuable for smaller or pruned models, which often struggle to achieve high accuracy due to their reduced capacity. In perception systems, traditional models are trained using hard targets, such as labeled bounding boxes and object classes. However, knowledge distillation [43] enhances this process by using soft targets from a larger Mother model, which includes probability distributions [44] over object classes and detailed bounding box predictions. These soft targets capture the Mother model's confidence and insights into object relationships, such as assigning probabilities to similar classes like "car" and "truck." This richer information allows the Child model to learn not only the object classifications but also the spatial nuances in object localization more effectively. Through distillation, the smaller Child model approximates the Mother model's performance, improving both class predictions and bounding box accuracy, even with reduced capacity.

### G. Fine-Tuning

Fine-tuning [45] in deep learning is the process of adjusting a model to suit a specific task or dataset better. This technique leverages the knowledge gained from the initial training on a large dataset, making the model more effective with minimal additional training. It is particularly important for smaller or pruned models, as it helps them regain performance that might be lost due to their reduced size or complexity. To fine-tune a pruned model, a set of hyperparameters such as learning rate, batch size, and number of epochs must be optimized to balance maintaining the model's efficiency and recovering lost accuracy. One common approach is to use a

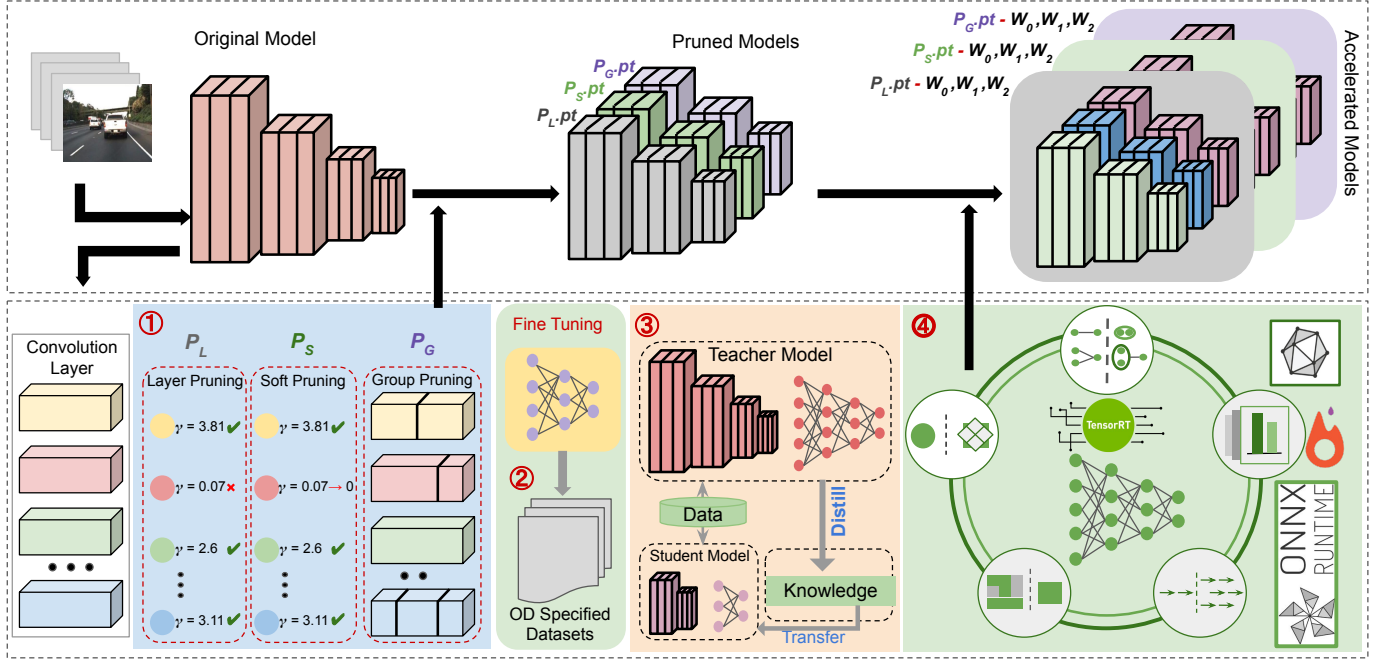


Fig. 2: An overview of methodologies - (1) three pruning techniques, (2) fine-tuning, (3) knowledge distillation, and (4) quantization through workflows (three workflows: Torch-TensorRT, ONNX Runtime Quantization, and Torch-ONNX-TensorRT).

low learning rate (e.g., one-tenth of the original learning rate) to ensure that the weights are updated cautiously, preventing drastic changes that could lead to overfitting. Fine-tuning can be done with or without freezing layers [46]. When freezing the initial layers, only the later layers are updated during training, which allows the model to retain the low-level feature representations learned from the original training dataset. Alternatively, full-model fine-tuning retrain all layers of the network, which is more suitable when the new task or dataset is significantly different from the original one. By fine-tuning, these models can achieve high accuracy while remaining efficient, making them suitable for deployment in resource-constrained environments [47].

#### H. Prior Related Study

Several approaches including parallel processing [48], [49], hardware acceleration [50], model compression [51], [52], and model optimization [53], [54] have been proposed by authors to maximize the inference performance suitable for real-world SDVs. Techniques like parallelism [55] and pipelining [56] are introduced to maximize usage of the limited resource [57]. TensorRT-based frameworks on Jetson Boards [58] are utilized for optimization. Some authors worked on modifying the model's architecture to create a lighter model suitable for edge devices [16], [59], [60]. Deep computation model [61] was proposed of lightweight tensor to make the model suitable for resource-constrained edge devices [62]. Adjusting weight sparsity in pruning for real-time execution on edge devices [63] was introduced to solve the problem. Some authors also suggest one-shot pruning for efficient real-time object tracking [64], [65]. Dynamic resource allocation approach was proposed to solve resource-constrained problems in edge-assisted software-defined vehicles [66]. Tools like TensorRT were explored to improve DL real-time inference [67] by researchers. Also, techniques like image offloading to the

cloud were introduced to reduce computational cost on edge-assisted real-world driving systems [68]. An efficient parallel pruning approach proposed for efficient task-scheduling on resource constraint systems [69]. Some authors accelerated Neural Networks by intra-kernel pipelining for time-triggered transportation systems [70]. DNN layering techniques were introduced to maximize the inference with minimization in memory usage [71]. However, existing studies often overlook a comprehensive analysis of integrating pruning methods with all quantization precision modes. While many focus on a single precision mode, our approach conducts an extensive investigation into the impact of three pruning techniques combined with FP32, FP16, and INT8 across three workflows using various frameworks. This thorough examination aims to optimize inference performance and provide deeper insights into SDVs.

### III. EXPERIMENT DESIGN

In this section, we present the notations of our methodology, which will be referenced and explained in detail in the subsequent sections of the paper. These notations form the foundation for understanding the key processes and techniques applied throughout the study. The overview of methodology from Fig. 2, we first train a YOLOv5s and load the model. Then we develop three pruning techniques that are applied to the YOLOv5s for model optimization. After successfully applying the pruning techniques, our experimental framework includes fine-tuning and knowledge distillation. For each pruned version of the model, we then implemented three comprehensive workflows for quantization which is designed using TensorRT, ONNX, and ONNX-RunTime [72] while evaluating the model across three precision modes: FP32, FP16, and INT8. From Fig. 2, the composed notations and brief descriptions are listed below:



- $P_L$ : Layer pruning (selectively prune a layer)
- $P_S$ : Soft pruning (certain weights or channels set to zero)
- $P_G$ : Group pruning (remove entire groups of connected parameters)
- $P_L.pt$ : Layer pruned model
- $P_S.pt$ : Soft pruned model
- $P_G.pt$ : Group pruned model
- $(P_L, P_S, P_G).pt$ : the layer pruned model ( $P_L.pt$ ) or the soft pruned model ( $P_S.pt$ ) or the group pruned model ( $P_G.pt$ )
- $W_{default}$ : PyTorch default workflow
- $W_0$ : Torch - TensorRT (the workflow with a collaborative effort combining PyTorch with NVIDIA's TensorRT)
- $W_1$ : ONNX Runtime Quantization (the workflow with a high-performance inference engine developed by Microsoft)
- $W_2$ : Torch - ONNX - TensorRT (the workflow that combines the flexibility of PyTorch, the interoperability of ONNX, and the high-performance inference capabilities of TensorRT).

$W_{default}$ : PyTorch Default: By default, in the PyTorch framework, we load our YOLOv5s model trained with the COCO dataset and ensure that its inference runs on the GPU.

$P_L$ : Layer Pruning: Layer pruning in deep learning models, such as YOLOv5, involves selectively removing certain layers or channels to reduce the computational load during inference while preserving essential layers, such as the detection layer.

$P_S$ : Soft Pruning: Soft pruning reduces the influence of certain weights or channels without removing them, preserving the model's structure for easier fine-tuning. It targets less important channels by setting their weights and biases to zero, allowing for recovery of performance after pruning.

$P_G$ : Group Pruning: Group pruning is a technique used in neural networks to remove entire groups of connected parameters, such as channels in convolutional layers, in a coordinated way. By considering layer interconnections, it maintains the network's structural integrity and results in more hardware-efficient sparsity patterns. This method involves using a dependency graph to guide systematic pruning across layers.

$(P_L, P_S, P_G).pt - W_0$ : Pruning - (Torch - TensorRT): In this workflow, we improved pruned model performance using Torch-TensorRT quantization, consisting of three phases: ❶ simplifying the TorchScript module, ❷ transforming it for optimized execution, and ❸ executing the optimized graph for efficient inference performance.

$(P_L, P_S, P_G).pt - W_1$ : Pruning - (ONNX Runtime Quantization): In this workflow, the ONNX Runtime Execution Provider optimizes pruned model execution by leveraging hardware acceleration libraries, enabling deployment across various environments. The process involves ❶ loading pruned models, converting them from PyTorch to ONNX, and verifying the conversion, and ❷ quantizing the models into FP32, FP16, and INT8 precision modes, optimizing them for deployment across diverse hardware setups.

$(P_L, P_S, P_G).pt - W_2$ : Pruning - (Torch - ONNX - TensorRT): In this workflow, we accelerate pruned model performance using quantization with our Torch-ONNX-TensorRT pipeline, organized into three stages: ❶ exporting the PyTorch model to the ONNX format for interoperability, ❷ building the TensorRT engine for optimized execution on NVIDIA GPUs,

and ❸ deploying the model to enhance inference performance for efficient real-world operation.

Notation	Description	Organization
$P_L, P_S$ and $P_G$	Layer Pruning, Soft Pruning and Group Pruning	Section IV
$P_L.pt, P_S.pt$ and $P_G.pt$	Saved pruned models after applying each pruning techniques $P_L, P_S$ and $P_G$	Section IV
Fine-tuning	Applied to maintain the accuracy of pruned models	Section IV
Knowledge Distillation	Applied to maintain the accuracy once again	Section IV
$P_L, P_S, P_G).pt - W_0$	Quantization of all pruned models( $P_L, P_S, P_G).pt$ ) through $W_0$ -(Torch-TensorRT)	Section V
$P_L, P_S, P_G).pt - W_1$	Quantization of all pruned models( $P_L, P_S, P_G).pt$ ) through $W_1$ -(ONNX Runtime EP)	Section V
$P_L, P_S, P_G).pt - W_2$	Quantization of all pruned models( $P_L, P_S, P_G).pt$ ) through $W_2$ -(Torch-ONNX-TensorRT)	Section V

TABLE I: The summary of the key processes, including pruning methods (layer, soft, group), fine-tuning and knowledge distillation for accuracy recovery, and quantization using various workflows ( $W_0, W_1, W_2$ ), along with their corresponding sections for further details.

The overview of our methodology Fig. 2, we note that first, the original model goes through ❶ three different pruning types  $P_L, P_S$  and  $P_G$ . After Pruning performance, we save each of the pruned models ( $P_L.pt, P_S.pt$  and  $P_G.pt$ ), ❷ we then fine-tune the pruned models and ❸ distill knowledge from the large YOLO model to keep up the accuracy with accelerated performance. At this point, our pruned model is lighter than the original model without significant compromise in accuracy. From Fig. 2 ❹, Then we utilize our designed workflow  $W_0, W_1$  and  $W_2$  (elaborately described in Fig. 4) for FP32, FP16 and INT8 precision mode with PTQ and QTA and calibration dataset for INT8. Table I provides a concise summary of the key elements from Fig. 2, including an overview of pruning and quantization techniques, along with references to the sections where these processes are described in detail.

#### IV. PRUNING OVERVIEW AND DESCRIPTION

In this section, we present a detailed explanation of the pruning methods we developed, along with the subsequent fine-tuning and knowledge-distillation processes. Following the Torch-Pruning framework (provided by PyTorch) [73], we extended and created three customized pruning methods for our specific criteria and objectives. The three pruning methods are summarized in Table II, and we provide a comprehensive description of Fig. 3 for each pruning methods description following fine-tuning and knowledge distillation in the following subsections.

Notation	Description
$P_L$	Pruning technique to eliminate specific layer.
$P_S$	Pruning technique to zeros out redundant parameters and weights.
$P_G$	Pruning technique to remove groups of connected parameters that are redundant.

TABLE II: The summary of three pruning techniques which includes  $P_L$  (layer pruning),  $P_S$  (soft pruning) and  $P_G$  (group pruning).

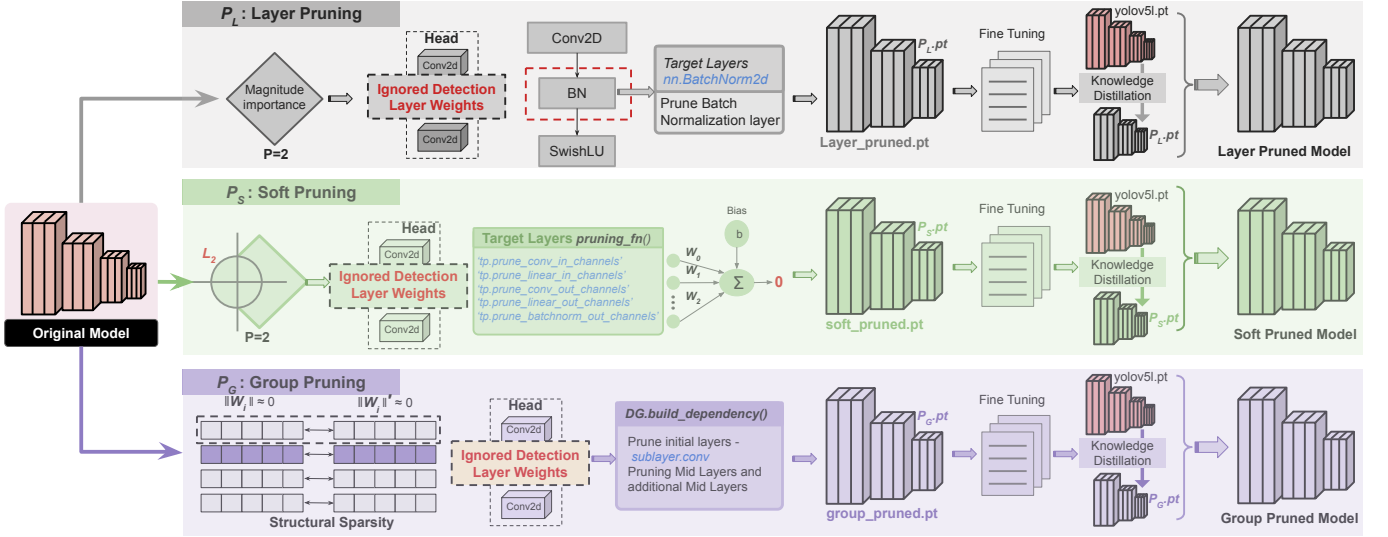


Fig. 3: An overview of pruning methods descriptively,  $P_L$ : layer pruning,  $P_S$ : soft pruning and  $P_G$ : group pruning combined with Fine-Tuning and Knowledge Distillation which illustrates the pruning process for each method, followed by fine-tuning to recover lost accuracy and knowledge distillation from a larger model (YOLOv5l) to further enhance the performance of the pruned models.

#### A. $P_L$ : Layer Pruning Procedures

Batch normalization (BN) [74] layers in deep learning models like YOLOv5 normalize mini-batch inputs, speeding up training. These layers have parameters that can be pruned, followed by activation and convolutional layers. By pruning BN layers and removing corresponding channels in subsequent layers, we reduce the number of operations needed during inference. The process of applying Layer pruning in the Object detection model is shown in Fig. 3.

**Defining the importance measure:** We use PyTorch’s TorchPruning to create an importance measure object, *MagnitudeImportance*, which determines the importance of each channel based on the L2 norm. This calculation is used to decide which channels to prune.

**Identifying layers to ignore:** We carefully Identify and mark layers that should not be pruned. We ignore the detection layer (Detect) during pruning to maintain the model’s detection functionality.

**Initializing the pruner:** We define the number of iterative pruning steps and the pruning ratio, indicating the fraction of channels to be pruned in each step. Then, we create a *MagnitudePruner* object using the model, including inputs, importance measure, number of steps, pruning ratio, and ignored layers. We set the pruning ratio to 20%.

**Iterative pruning performance:** For each pruning group we identify the target layer and the pruning function. We select the target batch normalization layers (*nn.BatchNorm2d*) for pruning.

**Pruned model performance:** In the performance evaluation step we save the pruned model  $P_L.pt$  (layer pruned model) in the disk and measure the evaluation metrics.

#### B. $P_S$ : Soft Pruning Procedures

Soft pruning reduces the influence of certain weights or channels in a neural network without removing them, preserving the model’s structure for easier fine-tuning and performance recovery. Instead of eliminating weights, we set probable pruned weights and biases to zero. As shown in Fig. 3, we

initiate soft pruning by measuring the L2 norm to assess channel importance, targeting those with lower values, and to maintain essential detection functions we exclude the detect layers from pruning. The process if soft pruning is as follows:  
**Initialization of pruner:** We initialize a *MagnitudePruner* object with the following parameters. *i)* Importance: The *MagnitudeImportance* measure. *ii)* We set the *global\_pruning*: Set to False to perform local pruning within each layer. *iii)* Pruning\_ratio: Set to 0.2 to prune 20% of the channels in each iteration.

**Soft pruning process:** For each pruning group, the target layer and pruning function were identified. Depending on the pruning function, we set the weights and biases of the selected channels to zero:

(a) For convolutional layers (Conv2d) and linear layers (Linear), We set the weights of the pruned input channels to zero by multiplying the corresponding weights with zero.

(b) For convolutional layers and linear layers, the weights and biases of the pruned output channels were set to zero.

(c) For batch normalization layers (BatchNorm2d), both weights and biases of the pruned channels were set to zero.

We set the weights of the pruned Convolutional, Linear Layers, and Batch Normalization Layers (Input Channels and Output Channels) to zero. And, if biases existed for these layers, they were also set to zero.

**After pruning performance:** After pruning, the model  $P_S.pt$  (soft pruned model) is saved to the disk. This saved model retains the pruned weights and biases set to zero, facilitating further fine-tuning if necessary.

#### C. $P_G$ : Group Pruning Procedures

Group pruning removes groups of connected parameters, such as channels in convolutional layers, in a coordinated way to preserve the network’s structural integrity and dependencies. This approach enforces structured sparsity by pruning entire groups together, leading to more predictable and hardware-friendly sparsity patterns. The process we follow to apply group pruning is as follows:

**Building dependency graph:** From Fig. 3 we constructed a dependency graph using Torch-Pruning’s DependencyGraph to understand the dependencies between layers and ensure that pruning actions maintain the structural sparsity of the network.

**Pruning initial layers:** For the initial layers of the model, specifically the first two layers, we identified sublayers with convolutional operations (conv). We pruned 20% of the channels in these layers by selecting indices for pruning and obtaining a pruning group using the dependency graph. The group was checked for consistency before applying the pruning operation.

**Pruning mid-layers:** For mid-layers (*indices 2, 4, 6, and 8*), we applied the same strategy to sublayers (*cv1, cv2, cv3*) containing convolutional operations. We pruned 20% of the channels, ensuring the pruning actions were consistent with the dependency graph.

**Pruning additional layers:** For other layers (*indices 3, 5, and 7*) and the detection layers (*indices 9 to 25*), we continued the same pruning strategy. For each layer with convolutional operations, we selected 20% of the channels for pruning and validated the pruning group before application.

**Performance evaluation after pruning:** Finally, the pruned model  $P_G.pt$  (group pruned model) was saved to a file and ready to calculate the evaluation metrics.

#### D. Fine Tuning Procedures

After applying the three pruning techniques described earlier, we obtained three distinct pruned models  $P_L.pt$  (layer pruned model),  $P_S.pt$  (soft pruned model), and  $P_G.pt$  (group pruned model). To evaluate the effectiveness of each pruning method, we executed the *detect.py* script on each pruned model. This script allowed us to assess the mean Average Precision (mAP) as well as the inference performance. However, the initial results revealed that the mAP of the pruned models was significantly lower than desired. From Fig. 3, the fine-tuning was performed on the pruned models  $P_L.pt$  (layer pruned),  $P_S.pt$  (soft pruned), and  $P_G.pt$  (group pruned) using the COCO dataset.

**Execution of build.sh Script and key parameters.** We started the fine-tuning process for each pruned model by running the *build.sh* script -that utilizes a training command specifying essential parameters such as image resolution, batch size, number of epochs, and the optimization method. Then, we created a custom script and custom script invoke it using a command specifying the input image size (*-img 640*), batch size (*-batch 128*), and the number of training epochs (*-epochs 300*).

**Optimization method:** As an optimizer we used the AdamW optimizer, which adapts the learning rate and applies weight decay to prevent overfitting. This optimizer is particularly effective for fine-tuning as it provides better convergence properties for pruned models.

**Configuration and hyperparameters:** We then employed a configuration file (*hyp.finetune.yaml*) to adjust hyperparameters, including learning rate, momentum, and weight decay.

**Dataset:** Finally, we performed the fine-tuning on the COCO dataset, which provides a diverse range of object categories. This allowed the pruned models to generalize well and recover lost accuracy across a variety of object detection tasks.

By fine-tuning the pruned models, we were able to significantly improve their mAP, recovering much of the accuracy

lost during pruning.

#### E. Knowledge Distillation Procedures

In addition to fine-tuning, we further distilled knowledge to the pruned models to mitigate the accuracy drop caused by pruning further. The distillation, as illustrated in Fig. 3 significantly enhanced the performance of the pruned models, allowing them to retain high accuracy despite their streamlined structure. To improve the accuracy of the pruned models, further, we applied a knowledge distillation technique, where the knowledge from a larger model was used as the teacher model for our pruned models. This distillation process transferred the knowledge from the YOLOv5l large model into the pruned models, helping them recover lost accuracy while maintaining their reduced size and faster inference speed. We incorporated Kullback-Leibler (KL) divergence as the distillation loss, in addition to the traditional classification loss, to guide the student models in mimicking the output distribution of the teacher. By balancing the task loss and distillation loss during training, the pruned models were able to absorb valuable information from the larger YOLOv5l model, helping them recover lost accuracy while maintaining their reduced size and faster inference speed. **Teacher Model:** The YOLOv5l large model acted as the teacher model, while the pruned models (which had been fine-tuned earlier) served as the student models. **Distillation process:** The distillation transferred knowledge from the YOLOv5l model into the pruned models by minimizing the Kullback-Leibler (KL) divergence between the teacher’s softened output logits and the student’s output, alongside the traditional classification loss. **Key parameters:** We set batch size and initialized teacher model weight (YOLOv5l) and student model (pruned models) weight for the distillation process in the script as - (*-batch-size 64*), (*-teacher\_weight yolov5l.pt*). The remaining process involves training the pruned model (student) with the help of the YOLOv5l teacher model.

### V. QUANTIZATION WORKFLOW DESCRIPTION

In this section, we present a brief summary of the selected quantization workflows in Table III. Detailed descriptions of each workflow are provided in subsection V-A, subsection V-B and subsection V-C.

Notation	Description
$W_0$	Torch-TensorRT is a collaboration between Meta AI and NVIDIA that combines PyTorch and TensorRT to optimize DL models supporting quantization for FP32, FP16, and INT8 precision modes.
$W_1$	ONNX Runtime EP enables efficient model execution by leveraging hardware acceleration libraries across different environments, supporting quantization for FP32, FP16, and INT8 precision modes.
$W_2$	Torch-ONNX-TensorRT improves inference by converting a PyTorch model to ONNX, then optimizing it into a TensorRT engine for FP32, FP16, and INT8 precision modes.

TABLE III: The summary of three quantization workflows.  $W_0$  represent The (Torch - TensorRT) quantization workflow,  $W_1$  represent (ONNX Runtime Execution Provider), and  $W_2$  represent (Torch - ONNX - TensorRT) quantization respectively.

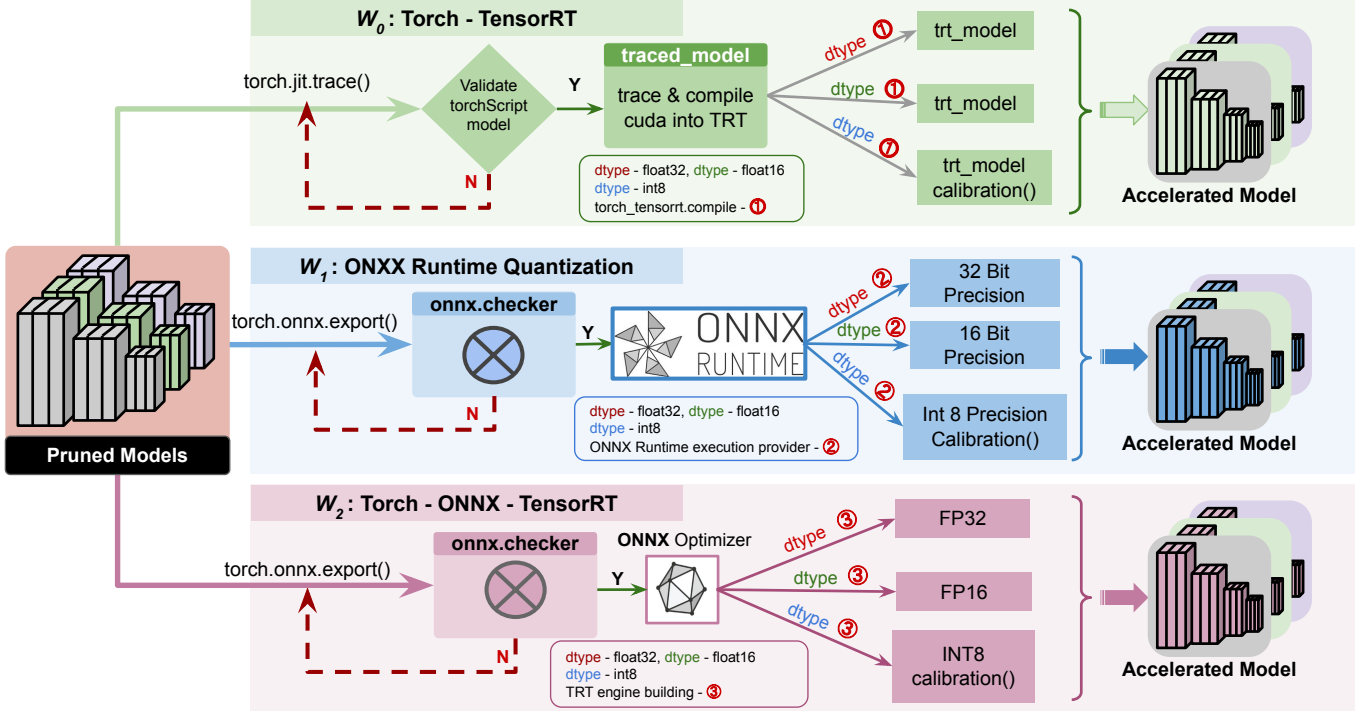


Fig. 4: An overview of quantization workflows for acceleration across multiple precision modes (FP32, FP16, INT8) applied to pruned models. Workflow  $W_0$ : Torch-TensorRT performs quantization using the Torch-TensorRT framework to improve model performance. Workflow  $W_1$ : ONNX Runtime Quantization demonstrates the steps for executing quantization via the ONNX Runtime Execution Provider Interface (EPI). Workflow  $W_2$ : Torch-ONNX-TensorRT integrates Torch and ONNX with TensorRT for further quantization.

A.  $(P_L, P_S, P_G).pt$  -  $W_0$ : Pruned models Quantization with Torch - TensorRT.

In this workflow  $W_0$ , we accelerate the inference performance of the object detection model YOLO by employing Torch-TensorRT. The description of quantization through Torch-TensorRT  $W_0$ , is as follows:

- 1) **Loading pruned models and dataset selection:** We start the workflow  $W_0$ , by loading our pruned models from three pruning techniques:  $P_L.pt$  (layer pruned model),  $P_S.pt$  (soft pruned model) and  $P_G.pt$  (group pruned model) which we saved in the disk after pruning correctly. Now, we select our dataset, ensuring that the image size is fixed at 640x640 pixels, and convert it into tensors.
- 2) **Model tracing:** Upon resizing the images to the specified dimensions, we proceed to transform our pruned models into TorchScript modules utilizing PyTorch's Just-In-Time (JIT) compiler. After transforming to TorchScript, each pruned model— soft pruned, layer pruned, and group pruned—is stored as a traced model shown in Fig. 4 and preserved on disk for further use.

**Float32** - For optimization using 32-bit floating point precision, we compile each traced model with TensorRT, explicitly setting the precision to FP32. This process requires specifying both the input shape and the data type as `torch.float32`.

**Float16** - We also create alternative versions of the models utilizing 16-bit floating point precision (FP16). This is accomplished by changing the data type to `torch.half`, which reduces the precision but significantly enhances computational efficiency.

**Int8** - To optimize the models for 8-bit integer precision, We select a representative calibration dataset that captures the range of inputs the model will encounter during inference. In Post-Training Quantization (PTQ), a fixed range is chosen for each quantizer, often determined through a calibration process. The calibration process adjusts the scale and zero-point for each tensor. This involves passing the calibration dataset through the model to compute the dynamic range (min and max values) of activations and weights. During Quantization-Aware Training (QAT), we again fine-tune the calibrated model to further enhance its accuracy. After calibration, we compile the traced models with TensorRT, specifying INT8 precision using `torch.tensorrt.compile(model, inputs=[torch.tensorrt.Input(example`input.shape`)], enabled_precisions=torch.int8, calibrator=calibrator)`.

- 3) **Inference performance:** Next, we benchmark the FP32, FP16, and INT8 optimized models to compare their performance with the original model. The process begins by synchronizing the CUDA device for consistency. For each image, we capture the start time using `torch.cuda.Event()`, run the model to assess inference performance, and then record the end time, followed by resynchronizing the CUDA device to ensure accurate timing.

B.  $(P_L, P_S, P_G).pt$  -  $W_1$ : Pruned models Quantization with ONNX Runtime Execution Provider (EP)

ONNX Runtime Execution Provider framework allows us to ensure that ONNX models can be deployed across different environments and can take full advantage of the underlying hardware's computational capabilities for quantization. The in-



teraction between ONNX Runtime and the execution providers is managed through an API that assigns specific nodes or sub-graphs for execution by the EP library on supported hardware. The process of quantization using  $W_1$  is outlined below:

- 1) **Loading pruned models and dataset selection:** In this workflow, we begin by loading the saved pruned models:  $P_L.pt$  (layer pruned model),  $P_S.pt$  (soft pruned model) and  $P_G.pt$  (group pruned model) and specified dataset just like the previous workflow  $W_0$ .
- 2) **Pruned models to .onnx conversion:** We export the pruned models from PyTorch to the ONNX format. This involves converting each of the three pruned models—soft pruned, layer pruned, and group pruned—into ONNX models. To export the model into onnx file format we used `torch.onnx.export()` function. Following the conversion, we verified the file format’s correctness using the ONNX checker tool which is done by invoking `onnx.checker.check_model`.
- 3) **FP32 (single-precision floating point):** For FP32, we use the default settings, as this precision mode does not require any specific flag or additional configuration for quantization. The ONNX model remains in its original floating-point representation.
- 4) **FP16 (half-precision floating point):** To enable FP16 quantization, we configure the ONNX Runtime to use the ORT Execution Provider with FP16 settings. This involves specifying the (`QuantType.QInt16`, `QuantType.QUInt16`) in the ORT settings. FP16 provides a balance between performance and accuracy.
- 5) **INT8 (8-bit integer):** First, we configure the ONNX Runtime settings to enable INT8 precision using the Execution Provider Interface (EPI). This process involves setting the quantization parameters within ONNX Runtime, enabling both activation and weight quantization to INT8 precision. We implement a calibration process by defining a custom `CalibrationDataReader()` class, which reads and processes a representative calibration dataset. This dataset is then used to compute the appropriate quantization parameters through ONNX Runtime’s calibration methods. Next, the `quantize_static` function is used to apply these quantization parameters to the model. Specifically, we provide the function with the original full-precision ONNX model, and the calibration data reader, and specify the desired quantization settings. In this case, the activations are quantized using unsigned 8-bit integers (`QUInt8`), while the weights are quantized using signed 8-bit integers (`QInt8`). The `per_channel` parameter is set to True, enabling per-channel quantization for weights, which improves accuracy. Additionally, the `reduce_range` parameter is enabled to further enhance precision by reducing the quantization range. The calibration method selected is `MinMax`, which calculates the quantization parameters based on the minimum and maximum values observed in the calibration data.

Finally, along with the original full-precision model, FP16 and INT8 are then used by the ONNX Runtime Execution Provider to optimize and produce quantized models ready for efficient inference performance.

#### C. ( $P_L, P_S, P_G$ ).pt - $W_2$ : Pruned models Quantization with Torch - ONNX - TensorRT

In this workflow, we enhance the inference performance of the object detection model by converting the PyTorch model to the ONNX format and then the TensorRT engine. After saving the model in ONNX, we further optimize it by converting the ONNX model into a TensorRT engine for three precision modes: FP32, FP16, and INT8. The process is described as follows:

- 1) **Loading pruned models and dataset selection:** Like previous workflows  $W_0$  and  $W_1$ , we start this workflow  $W_2$ , after loading our pruned models saved in the disk:  $P_L.pt$  (layer pruned model),  $P_S.pt$  (soft pruned model) and  $P_G.pt$  (group pruned model). And, the dataset is also fixed at 640×640 pixels ensuring the image size.
- 2) **Pruned models to .onnx conversion:** After loading the pruned models from disk, we convert each model to ONNX format using the `torch.onnx.export` function, which preserves the model’s architecture. The converted model is then verified using the `onnx.checker.check_model` tool. Once verified, the final `yolo.onnx` model is saved, making it ready for deployment in ONNX-supported environments.
- 3) **TensorRT engine building:** To create the TRT engine for a specific precision mode (e.g., FP32, FP16, or INT8), we start the process by importing the necessary libraries and setting up the TensorRT logger. The engine-building procedure is outlined below and divided into two sections: (i) the general engine-building process and (ii) the distinct steps we took to build the engine for each specific precision mode.

**Generic engine building.** The generic TensorRT engine building consists of a series of strategic steps: (i) We initialize the TensorRT builder and configure its settings. (ii) We then optimize resource allocation by setting the maximum workspace size. (iii) Next, we define the network using the explicit batch flag to enhance batch processing. (iv) We parse the ONNX model, identify and mark the key output layer, and build a serialized network by integrating the TensorRT builder, network, and configuration, resulting in an optimized model ready for deployment. (v) Finally, we create a function to serialize and save the engine, defining paths for the ONNX model and engine, and deserializing the engine for future use.

**Precision modes for engine building.** FP32 (Default Precision): The FP32 precision mode is the default setting, and no specific precision flag needs to be set. FP16 (Half Precision): To enable FP16 precision, we activate the FP16 flag in the builder configuration by using `builder_config.set_flag(trt.BuilderFlag.FP16)`. This mode strikes a balance between performance and accuracy. INT8 (Integer Precision): Enabling INT8 precision requires setting the INT8 flag with `builder_config.set_flag(trt.BuilderFlag.INT8)`. Additionally, to maintain accuracy, an extra calibration step is performed using a dataset. This involves the `ImageBatchStream` and `Calibrator` classes. INT8 offers the highest performance in terms of throughput and latency, but careful calibration is essential to preserve model accuracy.

- 4) **Deployment of the inference performance:** We calcu-

late performance metrics using the `benchmark_trt_FP32` function, which measures inference with a TensorRT engine in FP32 precision. Instead of passing the model, we pass the engine and adjust the function accordingly. We create a CUDA execution context for TensorRT inference, managing resources and the pipeline. The input tensor is converted to a NumPy array, with GPU memory allocated for both input and output. A CUDA stream handles asynchronous operations, and input data is flattened to `np.float32`. After a warm-up loop, benchmarking involves transferring data to the GPU, running inference, and transferring the output back to the CPU. Each iteration is timed within an `n_runs` loop, and the times are used to calculate the average inference time and FPS. Finally, the CUDA stream is synchronized, and the optimization profile is set to 0 to complete all operations.

We implemented similar benchmark functions for FP16 (`benchmark_trt_FP16`) and INT8 (`benchmark_trt_INT8`) precision. The primary difference in `benchmark_trt_FP16` is the precision mode used for inference, which benchmarks the model in FP16. This mode can offer faster performance and reduced memory usage compared to FP32, especially on GPUs with Tensor Cores optimized for FP16 calculations. The `benchmark_trt_INT8` function benchmarks the engine in INT8 mode, which is faster and more memory-efficient than FP32 and FP16, particularly on GPUs optimized for INT8 calculations.

## VI. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we first describe the datasets used in performance analysis (presented in subsection VI-A) and experimental setup (Our hardware/software setup is presented in subsection VI-B). Later, we provide a brief description of evaluation metrics in subsection VI-C and then present the experimental results and analysis (presented in subsection VI-D, subsection VI-E, subsection VI-F and subsection VI-G).

For the performance analysis, we first evaluate the original (non-pruned, non-quantized) model on the validation dataset. We then assess the pruned models (soft pruned, layer pruned, and group pruned) and their quantized versions (FP32, FP16, INT8) on the same dataset. This analysis is organized into distinct parts, each focusing on different aspects of the model optimization process. After applying the three pruning methods outlined in Fig. 3, we obtained three distinct pruned models. In the first subsection VI-D, we evaluate the performance of pruned models referred to as  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$ . This comparison was made prior to applying any quantization techniques. Then, in subsection VI-E, subsection VI-F and subsection VI-G we present the accelerated performance of quantized workflow (denoted as  $W_0$ ,  $W_1$ , and  $W_2$ ) on pruned models. These workflows (Fig. 4) represent distinct paths for implementing quantization, each with three precision modes.  $W_0$  involves the use of Torch integrated with TensorRT,  $W_1$  employs ONNX Runtime Quantization, and  $W_2$  incorporates a combination of Torch, ONNX, and TensorRT. Hence, we evaluated in total nine unique workflow metrics across three pruned models using FP32, FP16, and INT8 precision modes. The results, consistently analyzed for 640×640 images with a batch size of 128, highlight the impact of pruning and quantization.

### A. Dataset Selection

To thoroughly evaluate our experimental results, we selected datasets. To evaluate our results, we used the KITTI [75], BDD100K [76], and COCO datasets [77], all well-suited for object detection in SDVs. The KITTI dataset offers detailed annotations in various urban scenarios, while BDD100K provides over 100,000 video sequences covering diverse driving conditions. KITTI and BDD100K are particularly well-suited for vehicle-related studies, offering real-world driving scenarios across various road types, traffic conditions, and environments. The COCO dataset further enriches our analysis with its extensive annotations and variety making it highly valuable for training and testing. These datasets are crucial for assessing object detection performance in SDVs.

### B. Experiment Setup

**Hardware Setup.** Our hardware configuration in Fig. 5 includes an NVIDIA GPU Workstation with an Intel Xeon CPU featuring substantial memory for handling large datasets and running multiple programs simultaneously. The setup includes four NVIDIA GeForce RTX 2080 Ti graphics cards, each with 11 GB of memory, providing robust parallel processing capabilities essential for deep learning, rendering, and advanced graphical computations. The RTX 2080 Ti is particularly valued for its performance in AI and machine learning workloads.

	NVIDIA GPU Workstation
CPU	Intel Xeon E5-2690 v4
GPU	4 x 11 GB GeForce RTX 2080 Ti
Frequency	2.6 GHZ
Core	14
Memory	64 GB
OS	Ubuntu 18.04 LTS




Fig. 5: The configuration of NVIDIA GPU workstation.

**Software Setup.** We create a dedicated software environment for our experiments using a Docker image called Torch-TensorRT. This setup ensures consistency across systems. Inside the Docker container, we configured ONNX, ONNX-Runtime-GPU, and essential Python libraries required for our tasks. The environmental software specifications are detailed in Table IV.

TABLE IV: Software configuration.

CUDA	PyTorch	ONNX	ONNX Runtime	TensorRT	Torch-TensorRT
12.1	2.2.0	1.15.0	1.17	8.6.5	2.2.0

### C. Evaluation Metrics

In this work, we utilize four evaluation metrics for our performance analysis: inference throughput, inference latency, GPU usage, and accuracy. These metrics are discussed in detail below:

(i) Inference throughput measures how many inputs, like video frames or images, an object detection model can process per second, typically in FPS or inferences per second. To evaluate throughput, we initialized various pruned and quantized models, prepared input batches, recorded the total inferences processed, and calculated throughput by dividing the total inferences by the processing time.

(ii) Inference latency is the time, measured in milliseconds (ms), that an object detection system takes to process an input, such as a video frame, and generate an output. To measure

TABLE V: Performance comparison across three datasets (COCO, KITTI, and BDD100K) in terms of MACs (Multiply-Accumulate Operations), parameters, mAP (Mean Average Precision), and GPU/memory usage for YOLOv5s model and three pruned models:  $P_L.pt$  (layer pruned model),  $P_S.pt$  (soft pruned model) and  $P_G.pt$  (group pruned model). This table highlights the impact of each pruning technique on accuracy and resource efficiency, comparing them with the original YOLOv5s model.

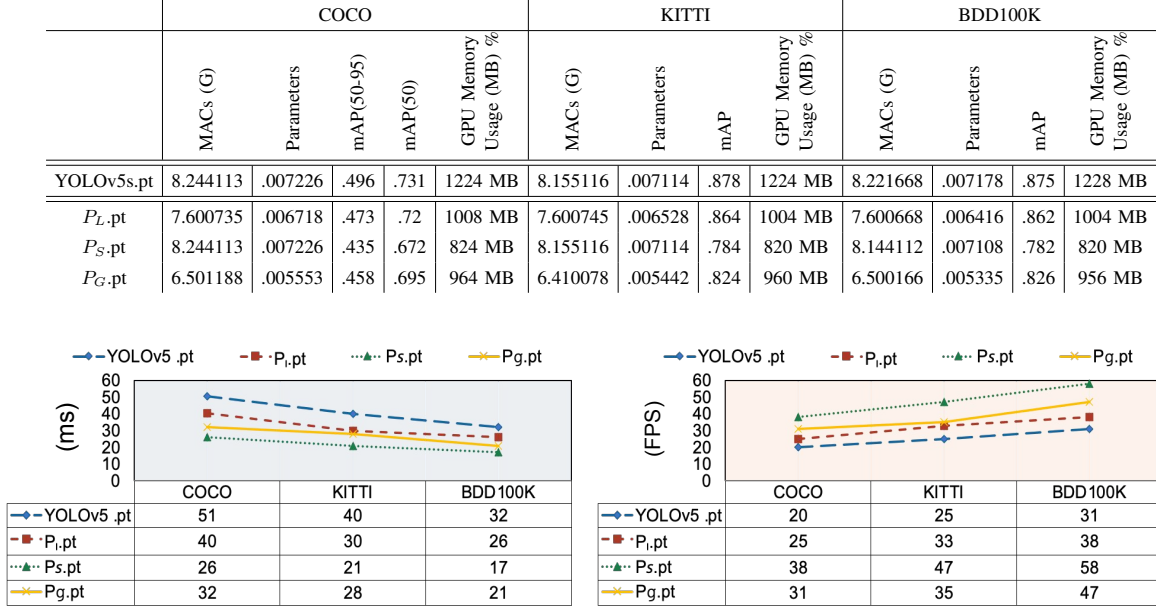


Fig. 6: The inference latency and throughput for pruned models ( $P_L.pt$ ,  $P_S.pt$  and  $P_G.pt$ ) on COCO, KITTI, and BDD100K datasets. (i) shows the latency reduction achieved by the pruned models in milliseconds (ms), highlighting the faster efficiency over the original YOLOv5 model. (ii) displays the throughput in frames per second (FPS), demonstrating the improved processing speed of the pruned models compared to the baseline YOLOv5, with notable gains in each dataset.

latency, models (original, pruned, and quantized) were initialized after a warm-up phase. Input data was preprocessed, and for each run, start and end times were recorded using `torch.cuda.synchronize()`. Latency was computed as the average time difference over multiple iterations for each inference.

(iii) GPU/Memory usage measurement involves monitoring and recording GPU resource usage, such as memory and computational capacity, during model inference. We initialized pruned and quantized models, using a concurrent thread to monitor GPU usage with libraries like GPUUtil, recording data at regular intervals for analysis.

(iv) Accuracy evaluates model performance using metrics like Precision (correct positive predictions), Recall (correctly identified positives), mAP (mean Average Precision across all classes), and IoU (Intersection over Union) to assess localization accuracy.

#### D. After Pruning Performance Analysis

As demonstrated above, we applied three pruning techniques to the original YOLOv5s object detection model, resulting in three pruned models. Each of these pruned models is lighter than the original model in terms of Multiply-Accumulate Operations (MACs) and the number of parameters. From Table V, we observe the evaluation metrics, including MACs, parameters, GPU/memory usage, and mAP for pruned models across the COCO, KITTI, and BDD100K datasets.

**Pruning Impact on Model Efficiency and Accuracy.** The application of three pruning techniques to the YOLOv5s model demonstrates consistent trends across the COCO, KITTI, and BDD100K datasets.

The layer-pruned model  $P_L.pt$  achieves MACs of 7.600735 G and parameters of 0.006718 G, reflecting a notable reduction compared to the original YOLOv5s model, which has MACs of 8.244113 G and parameters of 0.007226 G. The mAP(50-95) of  $P_L.pt$  is 0.473, which is slightly lower than the original model's mAP50 of 0.496, but still retains strong detection capability. Additionally, the GPU/memory usage for  $P_L.pt$  is reduced to 1008MB. The layer-pruned model offers a modest reduction in Multiply-Accumulate Operations (MACs) and parameters, while maintaining accuracy close to the original model, making it an effective option for optimizing memory usage without significantly compromising detection performance.

In contrast, the soft-pruned model, which retains the original MACs and parameters due to zeroed weights, experiences a more notable drop in accuracy. However, this model improves GPU memory efficiency, positioning it as a strong choice where efficiency is prioritized over precision.

The group-pruned model consistently strikes the best balance between model size reduction and accuracy, achieving the largest decrease in MACs and memory usage with only a moderate impact on performance. This trend holds across all datasets (COCO, KITTI, and BDD100K), with the group-pruned model demonstrating the most memory usage optimization, making it suitable for applications where both efficiency and accuracy are important.

**Pruned models latency and throughput.** From Fig.6, In terms of latency, the soft-pruned model consistently achieves the lowest latency across all datasets, outperforming the other

pruning methods. This can be attributed to its structure, which remains intact while zeroing out weights, allowing for smoother computation flow. The group-pruned model also shows a notable reduction in latency, making it the second-best option in this regard, while the layer-pruned model, though still an improvement over the original YOLOv5s, has a slightly higher latency compared to the other pruned models.

When evaluating throughput, the soft-pruned model once again stands out, delivering the highest frames per second (FPS) across the COCO, KITTI, and BDD100K datasets. This highlights the method’s ability to maintain computational speed while reducing unnecessary operations. The group-pruned model follows closely behind, offering incremental gains in throughput compared to the original model. And, the layer-pruned model shows moderate improvements in throughput.

Although, the soft pruning proves to be the most speed-up pruned model in terms of latency and throughput, the group-pruned model strikes the balance between speeding up (e.g., latency and throughput) and maintaining accuracy.

#### E. Performance Analysis on Pruned Models to (Torch-TensorRT) Quantization: $(P_L, P_S, P_G).pt - W_0$

Figure 7 illustrates the inference latency and throughput of the Torch-TensorRT workflow  $W_0$  on three pruned models:  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$  across the COCO, KITTI, and BDD100K datasets.

**Inference latency.** For quantization of the layer-pruned model  $P_L.pt$ , using Torch-TensorRT  $W_0$  from Fig. 7(i) on the COCO dataset, significantly reduces latency across all precision modes. The most notable improvement is seen with INT8 precision, where the latency is reduced by nearly  $5\times$  compared to the Pruned model, and  $6\times$  faster than the original model highlighting the effectiveness of quantization in improving performance for real-time applications. Similar improvements are observed with FP16 and FP32 precision modes, where the layer-pruned model experiences 3-4 $\times$  faster latency than the original. From KITTI and BDD100K Dataset we also observe up to 6.2 $\times$  faster speed after quantizing with INT8 precision.

For the soft-pruned model  $P_S.pt - W_0$ , the results show that INT8 precision quantization achieves the most significant latency reduction. For the COCO dataset, The INT8 precision mode reduces latency by nearly  $5\times$  compared to the soft-pruned model and almost  $9\times$  faster than the original model. Meanwhile, FP16 and FP32 also result in considerable latency reductions. Similar trends are observed in the KITTI dataset and BDD100K Dataset. Once again, INT8 precision mode outperforms others, offering the highest speedup. FP16 mode also performs efficiently, making it a strong alternative when a balance between accuracy and speed is required.

For the group-pruned model  $P_G.pt - W_0$ , quantization using Torch-TensorRT also results in noticeable latency reductions across all precision modes. On the COCO dataset, INT8 precision reduces latency by  $5\times$  compared to the unquantized group-pruned model and more than  $8\times$  faster than the original model. FP16 and FP32 also show significant latency improvements. Similar results are observed on the KITTI dataset and in the BDD100K dataset, where INT8 precision continues to offer the greatest speedup, delivering nearly  $5\times$  faster latency than the unquantized group-pruned model, while FP16 provides a good alternative with approximately  $3\times$  faster

performance.

**Inference throughput.** The inference throughput of the Torch-TensorRT workflow  $W_0$  on three pruned models:  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$  across the COCO, KITTI, and BDD100K datasets.

The quantization of soft pruned model  $P_S.pt - W_0$  archives the best performance in terms of FPS among all three precision modes. In the COCO dataset, INT8 precision achieves a  $5\times$  improvement in throughput compared to the pruned model, and nearly  $10\times$  faster than the original model. FP16 continues to offer solid performance gains ( $7.5\times$  faster). the layer-pruned model  $P_L.pt$ , quantization using Torch-TensorRT significantly boosts throughput. INT8 precision increases throughput by nearly  $5\times$  compared to the pruned model and more than  $6\times$  faster than the original model. FP16 also shows strong improvements, delivering over  $3\times$  higher throughput than the pruned model. In the group-pruned model  $P_G.pt$ , INT8 precision provides the most substantial throughput increase, with more than  $5\times$  higher throughput compared to the pruned model and over  $7.6\times$  faster than the original. FP16 also offers considerable improvements, making it a reliable alternative for efficient inference. Similar trends were observed in KITTI and BDD100K datasets.

#### Observation 1

- $W_0$  - Torch-TensorRT (PyTorch integration with TensorRT), on  $(P_L, P_S, P_G).pt$  (layer, soft, and group pruned models) produces  $2\times$  faster performance in FP32,  $3.5\times$  faster in FP16, and  $5\times$  faster in INT8 compared to the pruned models. Besides, it delivers up to  $4\times$  faster performance in FP32,  $6\times$  faster in FP16, and  $10\times$  faster in INT8 compared to the original model.

**Accuracy.** Table VI (i), presents the mAP results across three datasets: COCO, KITTI, and BDD100K. The column  $P_L.pt - W_0$  shows the mAP achieved by the quantized Torch-TensorRT workflow at FP32, FP16, and INT8 precision on the layer-pruned model  $P_L.pt$ . Similarly,  $P_S.pt - W_0$  and  $P_G.pt - W_0$  present the mAP for the quantized Torch-TensorRT workflow at these three precision modes on the soft-pruned and group-pruned models, respectively. The mAP results across the COCO, KITTI, and BDD100K datasets show that FP32 and FP16 precision modes maintain accuracy close to the original pruned models after quantization. However, INT8 precision generally leads to a more noticeable drop in accuracy. For example, in the COCO dataset, the layer-pruned model sees minimal changes with FP32 and FP16, while INT8 shows a more significant reduction in mAP. Similar trends are observed in the KITTI dataset, where the soft-pruned model retains accuracy with FP32 and FP16 but experiences a drop with INT8.

Overall, INT8 quantization offers strong inference performance but typically results in a modest reduction in accuracy compared to FP32 and FP16 across all datasets and models.

**GPU/Memory Usage.** Table VI (ii), presents the GPU/memory usage after quantization of workflow  $W_0$  across three precision modes on three pruned models. The data shows that GPU/memory usage decreases most significantly for INT8 precision mode. For instance, in the COCO dataset, the layer-pruned model uses 1008 MB of GPU memory. After quantization with  $P_L.pt - W_0$  in INT8 precision mode, the memory usage reduces to 860 MB. This indicates that



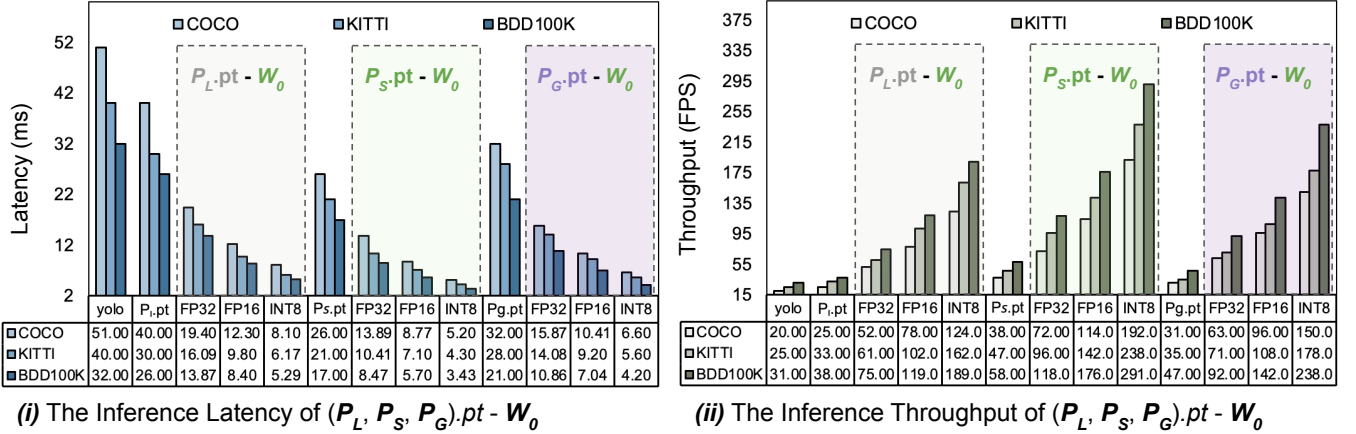


Fig. 7: Comparison of inference latency and throughput using the Torch-TensorRT workflow  $W_0$ , based on three pruned models  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$ . (i) Latency comparison shows the reduction in inference time in milliseconds (ms) for different precision modes (FP32, FP16, and INT8). (ii) Throughput comparison displays the improvement in frames per second (FPS) across the same precision modes.

TABLE VI: (i) Accuracy: Mean Average Precision (mAP) results and (ii) GPU/memory usage (batch size 128) for Torch-TensorRT workflow  $W_0$ , evaluated across COCO, KITTI, and BDD100K datasets. The table shows the mAP across different precision modes (FP32, FP16, INT8) and their corresponding memory usage.

		Yolo	$P_L.pt$	$P_L.pt - W_0$			$P_S.pt$	$P_S.pt - W_0$			$P_G.pt$	$P_G.pt - W_0$		
				FP32	FP16	INT8		FP32	FP16	INT8		FP32	FP16	INT8
(i)	COCO(50)	.731	.720	.720	.710	.650	.672	.670	.665	.601	.695	.692	.680	.620
	KITTI	.878	.864	.861	.854	.790	.784	.784	.781	.726	.824	.823	.820	.776
	BDD100K	.875	.862	.862	.858	.810	.782	.782	.780	.721	.826	.826	.817	.749
(ii)	COCO	1224 MB	1008 MB	1004 MB	980 MB	860 MB	824 MB	820 MB	796 MB	674 MB	964 MB	960 MB	938 MB	816 MB
	KITTI	1224 MB	1004 MB	1004 MB	976 MB	852 MB	820 MB	816 MB	792 MB	668 MB	960 MB	960 MB	936 MB	812 MB
	BDD100K	1228 MB	1004 MB	1004 MB	980 MB	864 MB	820 MB	816 MB	792 MB	672 MB	956 MB	960 MB	936 MB	816 MB

quantization to INT8 results in GPU memory usage that is 85% of that required by the layer-pruned model (which is about a 15% reduction compared to layer layer-pruned model).

#### F. Performance Analysis on Pruned Models to (ONNX Runtime) Quantization: $(P_L, P_S, P_G).pt - W_1$

Fig. 8, presents the inference latency and inference throughput of the ONNX Runtime EPI workflow  $W_1$  on three pruned models:  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$  on the COCO, KITTI, and BDD100K datasets.

**Inference latency.** From Fig. 8(i), we observe that quantization using ONNX Runtime EPI  $W_1$ , significantly reduces latency for the layer-pruned model  $P_L.pt$  across all datasets. On the COCO dataset, quantization notably decreases latency, particularly in INT8 precision, where performance improves nearly threefold compared to the pruned model and fourfold compared to the original. Similar trends are evident in the KITTI dataset, where INT8 precision provides the most significant speedup, achieving up to a  $3.1\times$  improvement over the pruned model. The BDD100K dataset follows the same pattern, with INT8 offering the largest latency reduction, demonstrating consistent gains in efficiency across all precision modes.

For the soft-pruned model  $P_S.pt$ , quantization with  $W_1$  leads to latency reductions across all datasets. On the COCO dataset, pruning alone provides a notable reduction in latency, and after quantization, the INT8 precision mode delivers the most significant improvement, achieving nearly  $3\times$  faster performance compared to the pruned model. Similar trends are

observed in the KITTI dataset and BDD100K dataset, INT8 precision continues to outperform other modes, offering the highest latency reduction, making it nearly  $3\times$  faster than the pruned model and over  $5\times$  faster than the original model. These results highlight the efficiency gains provided by INT8 quantization, with FP16 also offering a strong balance between speed and precision.

For the group-pruned model  $P_G.pt$ , quantization with ONNX Runtime EPI yields significant latency reductions across the datasets. On the BDD100K dataset, pruning itself leads to a reduction in latency, and further quantization with INT8 precision achieves the most pronounced reduction, providing a  $3\times$  speedup over the pruned model and nearly  $4.7\times$  faster than the original model. A similar pattern is observed in the KITTI dataset where FP16 also demonstrates considerable latency improvements, offering a balanced alternative for both datasets.

**Inference throughput.** Fig. 8(ii) illustrates the inference throughput of the ONNX Runtime workflow  $W_1$  on three pruned models:  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$  across the COCO, KITTI, and BDD100K datasets.

Quantization using ONNX Runtime  $W_1$  leads to noticeable improvements in throughput across all models and datasets, with INT8 precision delivering the most performance gains. Using the COCO dataset, the layer-pruned model  $P_L.pt$ , INT8 precision provides the most substantial improvement, delivering over  $3\times$  higher throughput compared to the pruned model. Similarly, the soft-pruned model  $P_S.pt$  sees a major



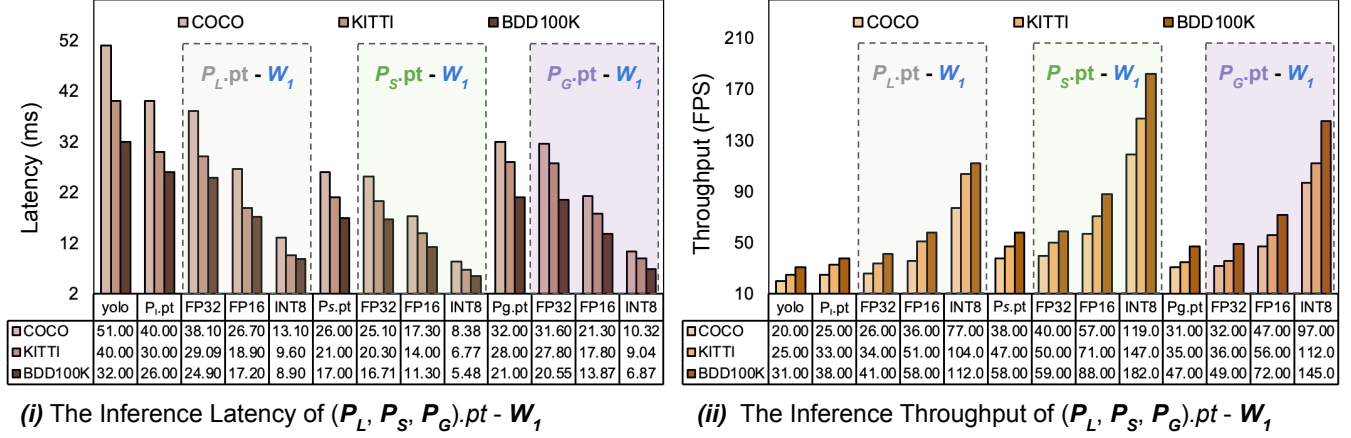


Fig. 8: Comparison of inference latency and throughput using the ONNX Runtime EP workflow  $W_1$ , based on three pruned models  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$ . (i) Latency comparison shows the reduction in inference time in milliseconds (ms) for different precision modes (FP32, FP16, and INT8). (ii) Throughput comparison highlights the improvement in frames per second (FPS) across the same precision modes.

TABLE VII: (i) Accuracy: Mean Average Precision (mAP) results and (ii) GPU/memory usage (batch size 128) for ONNX Runtime workflow  $W_1$ , evaluated across COCO, KITTI, and BDD100K datasets. The table shows mAP results across different precision modes (FP32, FP16, INT8) and their corresponding memory usage.

		Yolo	$P_L.pt$	$P_L.pt - W_1$			$P_S.pt$	$P_S.pt - W_1$			$P_G.pt$	$P_G.pt - W_1$		
				FP32	FP16	INT8		FP32	FP16	INT8		FP32	FP16	INT8
(i)	COCO(50)	.731	.720	.720	.718	.687	.672	.672	.672	.649	.695	.695	.690	.650
	KITTI	.878	.864	.864	.861	.836	.784	.784	.784	.748	.824	.824	.824	.792
	BDD100K	.875	.862	.862	.861	.829	.782	.782	.782	.750	.826	.826	.826	.790
(ii)	COCO(50)	1224 MB	1008 MB	998 MB	956 MB	712 MB	824 MB	812 MB	776 MB	524 MB	964 MB	956 MB	908 MB	656 MB
	KITTI	1224 MB	1004 MB	996 MB	956 MB	716 MB	820 MB	812 MB	772 MB	520 MB	960 MB	956 MB	904 MB	652 MB
	BDD100K	1228 MB	1004 MB	996 MB	952 MB	712 MB	820 MB	816 MB	772 MB	516 MB	956 MB	952 MB	902 MB	648 MB

boost in throughput, with INT8 precision achieving more than  $3\times$  the throughput of the pruned model and nearly  $6\times$  that of the original. The group-pruned model  $P_G.pt$  on the BDD100K dataset follows the same trend, with INT8 precision offering the largest gains. Across the COCO, KITTI, and BDD100K datasets, quantization consistently enhances throughput, particularly with INT8 precision.

### Observation 2

- $W_1$  - ONNX Runtime Quantization (Post training Static Quantization), on  $(P_L, P_S, P_G).pt$  (layer, soft, and group pruned models) produces  $1.5\times$  faster in FP16, and  $3\times$  faster in INT8 compared to the pruned models. Additionally, it delivers up to  $3\times$  faster performance in FP16, and  $6.1\times$  faster in INT8 compared to the original model.

**Accuracy.** Table VII (i), provides a detailed overview of the mAP results across three prominent datasets: COCO, KITTI, and BDD100K. The column  $P_L.pt - W_1$ , showcases the mAP obtained through the quantized ONNX Runtime workflow at different precision levels—FP32, FP16, and INT8—on the layer-pruned model  $P_L.pt$ . In a similar way,  $P_S.pt - W_1$  and  $P_G.pt - W_1$  reflect the mAP performance for the quantized workflow applied to soft-pruned and group-pruned models, respectively, across the same precision modes. Analyzing the data, The mAP results across the COCO, KITTI, and BDD100K datasets demonstrate that the ONNX Runtime quantization workflow ( $W_1$ ) maintains accuracy well at FP32

and FP16 precision levels, closely aligning with the performance of the original pruned models. For example, on the COCO dataset, the layer-pruned model shows no change in mAP at FP32, with only a slight reduction at FP16. However, a more noticeable decrease occurs when transitioning to INT8 precision. This pattern is consistent across the KITTI and BDD100K datasets. Overall, INT8 precision leads to reduced accuracy across all models and datasets, while FP32 and FP16 remain stable in terms of mAP performance.

**GPU/Memory Usage.** Table VII (ii) illustrates the GPU/memory usage following the quantization of workflow  $W_1$  across three different precision modes applied to three pruned models. The data reveals a reduction in GPU/memory usage, particularly for the FP16 and INT8 precision modes. For example, in the KITTI dataset, the soft-pruned model originally consumes 820 MB of GPU memory. After quantization with  $P_S.pt - W_1$ , the memory usage drops to 772 MB in FP16 mode and further decreases to 520 MB in INT8 mode. This demonstrates that quantization to INT8 reduces the GPU memory requirement to approximately 70% of what is needed by the soft-pruned model, (30% reduction compared with soft pruned model) highlighting the efficiency gains achievable through INT8 quantization.

### G. Performance Analysis on Pruned Models to (Torch-ONNX-TensorRT) Quantization: $(P_L, P_S, P_G).pt - W_2$

Fig. 9 showcases the inference throughput and latency of the Torch-ONNX-TensorRT workflow  $W_2$  applied to three pruned models:  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$  across the COCO, KITTI, and

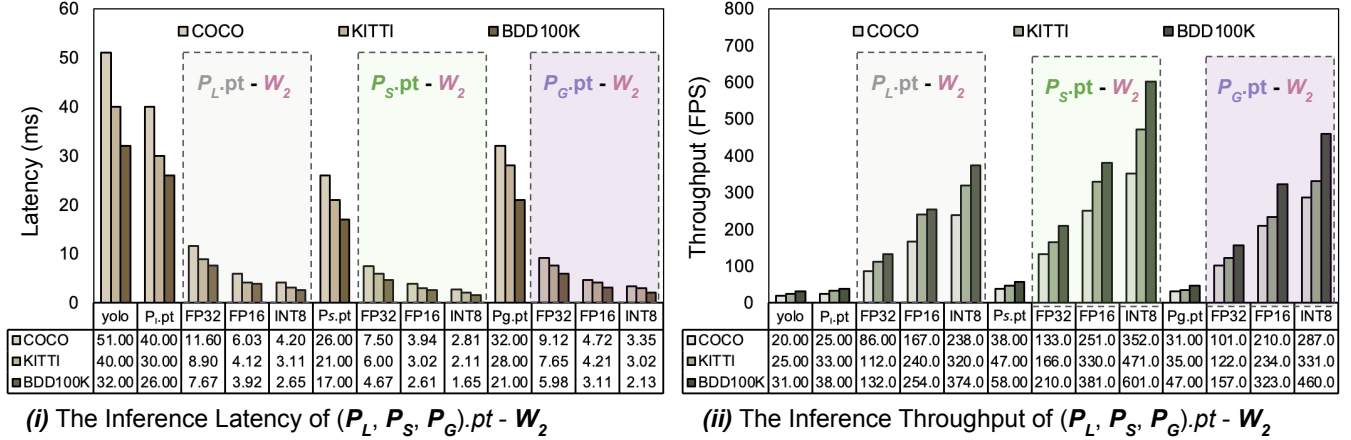


Fig. 9: The inference latency and throughput of Torch-ONNX-TensorRT workflow  $W_2$ , based on three pruned models  $P_L.pt$ ,  $P_S.pt$ , and  $P_G.pt$ . (i) Latency comparison shows the reduction in inference time in milliseconds (ms) across different precision modes (FP32, FP16, INT8). (ii) Throughput comparison demonstrates the increase in frames per second (FPS) across the same precision modes

TABLE VIII: (i) Accuracy: Mean Average Precision (mAP) results and (ii) GPU/memory usage (batch size 128) for Torch-ONNX-TensorRT workflow  $W_2$ , evaluated across COCO, KITTI, and BDD100K datasets. The table displays mAP results across different precision modes ( $FP32$ ,  $FP16$ ,  $INT8$ ) and their corresponding memory usage.

		Yolo	$P_L.pt$	$P_L.pt - W_2$			$P_S.pt$	$P_S.pt - W_2$			$P_G.pt$	$P_G.pt - W_2$		
				FP32	FP16	INT8		FP32	FP16	INT8		FP32	FP16	INT8
(i)	COCO(50)	.731	.720	.720	.714	.632	.672	.671	.670	.596	.695	.695	.692	.616
	KITTI	.878	.864	.862	.861	.781	.784	.784	.782	.702	.824	.824	.823	.748
	BDD100K	.875	.862	.862	.861	.780	.782	.782	.780	.701	.826	.826	.822	.745
(ii)	COCO(50)	1224 MB	1008 MB	1002 MB	976 MB	828 MB	824 MB	818 MB	796 MB	660 MB	964 MB	958 MB	932 MB	824 MB
	KITTI	1224 MB	1004 MB	1000 MB	976 MB	826 MB	816 MB	812 MB	792 MB	656 MB	960 MB	956 MB	928 MB	820 MB
	BDD100K	1228 MB	1004 MB	1004 MB	978 MB	828 MB	820 MB	816 MB	792 MB	652 MB	956 MB	952 MB	926 MB	818 MB

BDD100K datasets.

**Inference latency.** The results from Fig. 9(i), show significant latency reductions across all datasets when applying the Torch-ONNX-TensorRT workflow ( $W_2$ ) to pruned models. For the layer-pruned model  $P_L.pt$ , quantization leads to substantial improvements, particularly with INT8 precision, achieving a nearly  $9.5\times$  faster latency compared to the pruned model and a  $12\times$  reduction over the original model on the COCO dataset. Similar trends are observed across the KITTI and BDD100K datasets, where INT8 precision consistently delivers the greatest speedup, followed by FP16 and FP32.

The soft-pruned model  $P_S.pt$  after quantizing with  $W_2$ , also exhibits considerable latency reductions after quantization, with INT8 precision once again providing the most significant improvement, reducing latency by up to  $10\times$  compared to the pruned model. FP16 and FP32 show strong gains as well, further enhancing the efficiency of the pruned model across the datasets.

For the group-pruned model  $P_G.pt$ , quantization with  $W_2$  delivers consistent latency reductions, particularly in INT8 precision, which results in a  $10\times$  improvement over the pruned model. FP16 and FP32 also show notable reductions, enhancing the performance significantly across COCO, KITTI, and BDD100K datasets. Overall, the application of  $W_2$  leads to dramatic latency improvements, especially with INT8 precision, offering up to  $20\times$  faster performance compared to the original model and significant speedups across all precision modes.

**Inference throughput.** In the COCO dataset, the throughput for Fig. 9(ii), the Torch-ONNX-TensorRT workflow ( $W_2$ ) delivers substantial throughput improvements across all pruned models. For the layer-pruned model ( $P_L.pt$ ), quantization significantly boosts throughput, with INT8 precision offering the largest gains, achieving more than  $9\times$  improvement over the pruned model and nearly  $12\times$  higher than the original model. FP16 also demonstrates notable improvements, with throughput nearly doubling compared to the pruned model.

Similarly, for the soft-pruned model  $P_S.pt$ , quantization leads to dramatic increases in throughput, particularly with INT8 precision, which achieves more than  $9\times$  higher throughput than the pruned model and over  $17\times$  higher than the original. FP16 continues to provide strong gains, showcasing the efficiency of the workflow in accelerating model performance.

Quantizing the group-pruned model  $P_G.pt$  follows the same pattern, with INT8 precision delivering the highest speedup, exceeding  $9\times$  the throughput of the pruned model and  $12\times$  that of the original. Across all datasets, including KITTI and BDD100K, the quantization consistently accelerates throughput, with INT8 precision providing the most substantial gains, followed by FP16.

**Accuracy.** Table VIII (i), provides a comprehensive summary of the mAP results for three key datasets: COCO, KITTI, and BDD100K. The column  $P_L.pt - W_2$  highlights the mAP achieved using the quantized Torch-ONNX-TensorRT workflow across different precision levels—FP32, FP16, and INT8—on the layer-pruned model  $P_L.pt$ . Similarly,  $P_S.pt -$

$W_2$  and  $P_G.pt - W_2$  present the mAP performance for the quantized workflow applied to soft-pruned and group-pruned models, respectively, across the same precision modes.

The mAP results from the workflow  $W_2$  indicate that quantization at FP32 and FP16 precision levels maintains performance closely aligned with the original pruned models across the COCO, KITTI, and BDD100K datasets. For the layer-pruned model ( $P_L.pt$ ) on the COCO dataset, the mAP remains unchanged after quantization with FP32, with only a slight reduction observed at FP16. However, a more noticeable drop in mAP is seen when transitioning to INT8 precision. A similar pattern is observed in the KITTI dataset. The group-pruned model ( $P_G.pt$ ) on the BDD100K dataset follows the same trend, with FP32 and FP16 maintaining the original accuracy, but INT8 precision leading to a more substantial reduction. Overall, while FP32 and FP16 quantization preserves accuracy, INT8 precision tends to result in a measurable drop in mAP across all models and datasets.

### Observation 3

- $W_2$  - Torch-ONNX-TensorRT (PyTorch to ONNX export with TensorRT), on ( $P_L, P_S, P_G$ ).pt (layer, soft, and group pruned models) produces  $3.5\times$  faster in FP32,  $7\times$  faster in FP16, and up to  $10\times$  faster in INT8 compared to the pruned models. Additionally, it delivers up to  $7\times$  faster performance in FP32,  $14\times$  faster performance in FP16, and  $18\times$  faster in INT8 compared to the original model.

**GPU/Memory usage.** Table VIII (ii) shows the GPU/memory usage after quantizing workflow  $W_2$  across three precision modes (FP32, FP16, and INT8) on three pruned models. The results indicate a significant reduction in memory consumption, especially for the FP16 and INT8 precision modes. For instance, in the BDD100K dataset, the group-pruned model initially uses 956 MB of GPU memory. After quantization with  $P_G.pt - W_2$ , the memory usage decreases to 926 MB in FP16 mode and further reduces to 818 MB in INT8 mode. This is 82% usage of GPU/memory (reduces 20%) compared to the original memory usage demonstrating the substantial efficiency gains achieved through INT8 quantization.

## VII. CONCLUSION AND FUTURE PLAN

This paper provides a comprehensive analysis of object detection model inference performance using PyTorch frameworks, with a focus on acceleration through pruning and quantization within various workflows designed for resource-constrained SDVs. Our evaluation covered key performance metrics, including latency, throughput, GPU/memory usage, and accuracy, following both pruning and subsequent quantization (using our designed workflows) of the models.

Our study offers an in-depth examination of the performance of different pruned models, highlighting the strengths and weaknesses of each workflow across three precision modes: FP32, FP16, and INT8. This analysis provides valuable insights for selecting the most suitable pruning method and workflow for time-critical systems. Among the pruned models, the group-pruned model strikes the best balance between inference performance (throughput, latency) and accuracy, while the soft-pruned model, though fastest in inference, exhibits a notable drop in accuracy. Additionally, our results demonstrate

that the Torch-ONNX-TensorRT workflow yields the most effective acceleration of model inference performance.

After evaluating the three precision modes, we conclude that FP16 offers the optimal trade-off between inference performance (throughput, latency, and GPU/memory usage) and accuracy (mAP). This balance is crucial for real-world, time-critical SDVs. Therefore, we recommend the **Torch-ONNX-TensorRT workflow quantized with FP16 precision and group pruning as the optimal solution for applications requiring maximum inference performance in resource-limited SDVs**. It can achieve up to  $18\times$  faster inference speed and  $16.5\times$  higher throughput while reducing GPU/memory usage by up to 30%, all with minimal impact on accuracy.

In the future, we will include the evaluation of real-time performance metrics such as jitter (the variability in response time), deadline miss rate, real-time inference accuracy (RTIA), and memory bandwidth utilization, which will provide a more comprehensive evaluation of the model's performance in real-time and resource-constrained systems like SDVs.

## REFERENCES

- [1] R. Jafarpourmarzouni, S. Lu, Z. Dong et al., "Enhancing real-time inference performance for time-critical software-defined vehicles," in *2024 IEEE International Conference on Mobility, Operations, Services and Technologies (MOST)*. IEEE, 2024, pp. 101–113.
- [2] Y. Luo, D. Xu, G. Zhou, Y. Sun, and S. Lu, "Impact of rain-drops on camera-based detection in software-defined vehicles," in *2024 IEEE International Conference on Mobility, Operations, Services and Technologies (MOST)*. IEEE, 2024, pp. 193–205.
- [3] S. Lu and W. Shi, "Vehicle as a mobile computing platform: Opportunities and challenges," *IEEE Network*, 2023.
- [4] J. Chen and S. Lu, "An advanced driving agent with the multimodal large language model for autonomous vehicles," in *2024 IEEE International Conference on Mobility, Operations, Services and Technologies (MOST)*. IEEE, 2024, pp. 1–11.
- [5] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 6, pp. 1137–1149, 2016.
- [6] P. Li, X. Chen, and S. Shen, "Stereo r-cnn based 3d object detection for autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7644–7652.
- [7] P. Purkait, C. Zhao, and C. Zach, "Spp-net: Deep absolute pose regression with synthetic views," *arXiv preprint arXiv:1712.03452*, 2017.
- [8] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.
- [9] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.
- [10] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.
- [11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.
- [12] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, "A review of yolo algorithm developments," *Procedia computer science*, vol. 199, pp. 1066–1073, 2022.
- [13] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.
- [14] J. Redmon, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.
- [15] "YOLOv5," <https://pytorch.org/hub/ultralytics/yolov5/>, 2017.
- [16] S. Liang, H. Wu, L. Zhen, Q. Hua, S. Garg, G. Kaddoum, M. M. Hassan, and K. Yu, "Edge yolo: Real-time intelligent object detection system based on edge-cloud cooperation in autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 25 345–25 360, 2022.
- [17] S. Jiang, Z. Lin, Y. Li, Y. Shu, and Y. Liu, "Flexible high-resolution object detection on edge devices with tunable latency," in *Proceedings*

- of the 27th Annual International Conference on Mobile Computing and Networking, 2021, pp. 559–572.
- [18] A. Anupreetham, M. Ibrahim, M. Hall, A. Boutros, A. Kuzhiveli, A. Mohanty, E. Nurvitadhi, V. Betz, Y. Cao, and J.-S. Seo, “High throughput fpga-based object detection via algorithm-hardware co-design,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, no. 1, pp. 1–20, 2024.
  - [19] K. Jeziorek, A. Pinna, and T. Kryjak, “Memory-efficient graph convolutional networks for object classification and detection with event cameras,” in *2023 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. IEEE, 2023, pp. 160–165.
  - [20] L. Zhen, Y. Zhang, K. Yu, N. Kumar, A. Barnawi, and Y. Xie, “Early collision detection for massive random access in satellite-based internet of things,” *IEEE Transactions on Vehicular Technology*, vol. 70, no. 5, pp. 5184–5189, 2021.
  - [21] Y. Xiang and H. Kim, “Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference,” in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 392–405.
  - [22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded gpu using cuda,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.
  - [23] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, “Model compression and hardware acceleration for neural networks: A comprehensive survey,” *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.
  - [24] W.-F. Lin, D.-Y. Tsai, L. Tang, C.-T. Hsieh, C.-Y. Chou, P.-H. Chang, and L. Hsu, “ONNC: A compilation framework connecting ONNX to proprietary deep learning accelerators,” in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2019, pp. 214–218.
  - [25] S. Arabi, A. Haghighat, and A. Sharma, “A deep-learning-based computer vision solution for construction vehicle detection,” *Computer-Aided Civil and Infrastructure Engineering*, vol. 35, no. 7, pp. 753–767, 2020.
  - [26] F. Plesinger, A. Ivora, E. Vargova, R. Smisek, J. Pavlus, Z. Koscova, P. Nejedly, V. Bulkova, R. Kozubik, J. Halamek et al., “Scalable, multiplatform, and autonomous ecg processor supported by ai for telemedicine center,” in *2022 Computing in Cardiology (CinC)*, vol. 498. IEEE, 2022, pp. 1–4.
  - [27] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *arXiv preprint arXiv:1802.05668*, 2018.
  - [28] L. Yang and A. Shami, “On hyperparameter optimization of machine learning algorithms: Theory and practice,” *Neurocomputing*, vol. 415, pp. 295–316, 2020.
  - [29] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyperparameter optimization,” *Advances in neural information processing systems*, vol. 24, 2011.
  - [30] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression,” *arXiv preprint arXiv:1710.01878*, 2017.
  - [31] E. Michaud, Z. Liu, U. Girit, and M. Tegmark, “The quantization model of neural scaling,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
  - [32] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, “Rethinking the value of network pruning,” *arXiv preprint arXiv:1810.05270*, 2018.
  - [33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
  - [34] “ONNX (Open Neural Network Exchange),” <https://onnx.ai/>, 2017.
  - [35] “NVIDIA TensorRT,” <https://developer.nvidia.com/tensorrt>, 2017.
  - [36] R. Reed, “Pruning algorithms—a survey,” *IEEE transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.
  - [37] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Gutttag, “What is the state of neural network pruning?” *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.
  - [38] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X.-s. Hua, “Quantization networks,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 7308–7316.
  - [39] “Post Training Quantization (PTQ),” <https://pytorch.org/TensorRT/tutorials/ptq.html>, 2017.
  - [40] “Quantization Aware Training (QAT),” <https://pytorch.org/docs/stable/quantization.html>, 2017.
  - [41] J. Gou, B. Yu, S. J. Maybank, and D. Tao, “Knowledge distillation: A survey,” *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.
  - [42] Z. Li, H. Liang, H. Wang, M. Zhao, J. Wang, and X. Zheng, “MKD-Cooper: Cooperative 3d object detection for autonomous driving via multi-teacher knowledge distillation,” *IEEE Transactions on Intelligent Vehicles*, 2023.
  - [43] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, “Learning efficient object detection models with knowledge distillation,” *Advances in neural information processing systems*, vol. 30, 2017.
  - [44] P. De Rijk, L. Schneider, M. Cordts, and D. Gavrilu, “Structural knowledge distillation for object detection,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 3858–3870, 2022.
  - [45] C. Käding, E. Rodner, A. Freytag, and J. Denzler, “Fine-tuning deep neural networks in continuous learning scenarios,” in *Computer Vision—ACCV 2016 Workshops: ACCV 2016 International Workshops, Taipei, Taiwan, November 20–24, 2016, Revised Selected Papers, Part III* 13. Springer, 2017, pp. 588–605.
  - [46] F. C. Akyon, S. O. Altinuc, and A. Temizel, “Slicing aided hyper inference and fine-tuning for small object detection,” in *2022 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2022, pp. 966–970.
  - [47] Y. Wang, Z. Huang, Q. Liu, Y. Zheng, J. Hong, J. Chen, L. Xiong, B. Gao, and H. Chen, “Drive as veteran: Fine-tuning of an onboard large language model for highway autonomous driving,” in *2024 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2024, pp. 502–508.
  - [48] X. Hu, S. Li, T. Huang, B. Tang, R. Huai, and L. Chen, “How simulation helps autonomous driving: A survey of sim2real, digital twins, and parallel intelligence,” *IEEE Transactions on Intelligent Vehicles*, 2023.
  - [49] S. Lu, Y. Yao, and W. Shi, “Clone: Collaborative learning on the edges,” *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 222–10 236, 2020.
  - [50] G. Tatar, S. Bayar et al., “Real-time multi-learning deep neural network on an mpso-c-fpga for intelligent vehicles: Harnessing hardware acceleration with pipeline,” *IEEE Transactions on Intelligent Vehicles*, 2024.
  - [51] H. Liu, Y. He, F. R. Yu, and J. James, “Flexi-compression: A flexible model compression method for autonomous driving,” in *Proceedings of the 11th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications*, 2021, pp. 19–26.
  - [52] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “A survey of model compression and acceleration for deep neural networks,” *arXiv preprint arXiv:1710.09282*, 2017.
  - [53] J. Seo and S. Park, “Optimizing model parameters of artificial neural networks to predict vehicle emissions,” *Atmospheric Environment*, vol. 294, p. 119508, 2023.
  - [54] U. K. Das, K. S. Tey, M. Seyedmahmoudian, S. Mekhilef, M. Y. I. Idris, W. Van Deventer, B. Horan, and A. Stojcevski, “Forecasting of photovoltaic power generation and model optimization: A review,” *Renewable and Sustainable Energy Reviews*, vol. 81, pp. 912–928, 2018.
  - [55] M. Han, H. Zhang, R. Chen, and H. Chen, “Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 539–558.
  - [56] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, “Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks,” in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 329–341.
  - [57] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm, “Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 305–317.
  - [58] E. Jeong, J. Kim, and S. Ha, “Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 21, no. 5, pp. 1–26, 2022.
  - [59] H. Xu, M. Guo, N. Nedjah, J. Zhang, and P. Li, “Vehicle and pedestrian detection algorithm based on lightweight yolov3-promote and semi-precision acceleration,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 10, pp. 19 760–19 771, 2022.
  - [60] Z. Yao, Q. Liu, Q. Xie, and Q. Li, “Tl-detector: Lightweight based real-time traffic light detection model for intelligent vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 9, pp. 9736–9750, 2023.
  - [61] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, “An improved deep computation model based on canonical polyadic decomposition,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 10, pp. 1657–1666, 2017.
  - [62] D. Liu, L. T. Yang, R. Zhao, J. Wang, and X. Xie, “Lightweight tensor deep computation model with its application in intelligent transportation systems,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 3, pp. 2678–2687, 2022.
  - [63] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang, “Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5117–5124.



- [64] I. Jung, K. You, H. Noh, M. Cho, and B. Han, "Real-time object tracking via meta-learning: Efficient model adaptation and one-shot channel pruning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 11 205–11 212.
- [65] O. Andersson, F. Heintz, and P. Doherty, "Model-based reinforcement learning in continuous environments using real-time constrained optimization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.
- [66] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 196–209.
- [67] Y. Zhou and K. Yang, "Exploring tensorrt to improve real-time inference for deep learning," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2022, pp. 2011–2018.
- [68] R. Wang, H. Liu, J. Qiu, M. Xu, R. Guérin, and C. Lu, "Progressive neural compression for adaptive image offloading under timing constraints," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 118–130.
- [69] Y. Yang, N. Zhang, D. Yan, X. Wei, J. Zhou, H. Liu, and M. Chen, "Brief industry paper: Towards efficient task scheduling for autosar using parallel pruning," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 484–488.
- [70] Y. Wang, B. Feng, Z. Wang, T. Geng, K. Barker, A. Li, and Y. Ding, "{MGG}: Accelerating graph neural networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} pipelining on {Multi-GPU} platforms," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 779–795.
- [71] M. Ji, S. Yi, C. Koo, S. Ahn, D. Seo, N. Dutt, and J.-C. Kim, "Demand layering for real-time dnn inference with minimized memory usage," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 291–304.
- [72] "ONNX Runtime GPU," <https://onnxruntime.ai/>, 2017.
- [73] G. Fang, X. Ma, M. Song, M. B. Mi, and X. Wang, "Depgraph: Towards any structural pruning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 16 091–16 101.
- [74] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" *Advances in neural information processing systems*, vol. 31, 2018.
- [75] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the KITTI vision benchmark suite," in *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012, pp. 3354–3361.
- [76] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell, "BDD100K: A diverse driving dataset for heterogeneous multitask learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 2636–2645.
- [77] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755.



**Reza Jafarpourmarzouni** received his B.Sc in Mechanical Engineering from Babol Noshirvani University of Technology (BNUT), Iran, in 2021. He is currently working toward a Ph.D degree in Computer Science at Wayne State University, Detroit, USA. His academic advisor at Wayne State University is Prof. Zheng Dong. His research interests include real-time system, edge computing, cyber-physical systems and autonomous driving system.



**Yichen Luo** received her B.E. degree from Shanghai Dianji University, Shanghai, China, in 2023. She is currently pursuing a Ph.D. in Computer Science at William & Mary, Virginia, USA, under the supervision of Prof. Sidi Lu. Her academic research interests include the Internet of Things, autonomous driving, sensor systems, edge intelligence, and vehicle computing.



**Sidi Lu** is the assistant professor in the Department of Computer Science at William & Mary, Virginia, USA. She received her Ph.D. degree at Wayne State University, Detroit, USA in 2023. Her research interests broadly encompass edge computing, vehicle computing, emerging mobility, and applied AI and data science, aimed at enhancing the reliability, scalability, security, and efficiency of networked, distributed, and autonomous systems. More information can be found at <http://sidilu.org>.



**Zheng Dong** received a BS degree from Wuhan University, China, in 2007, the MS degree from the University of Science and Technology of China, in 2011, and the PhD degree from the University of Texas at Dallas, USA, in 2019. He is an assistant professor with the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests are in real-time embedded computer systems and connected autonomous driving systems. His current research focus is on multiprocessor scheduling theory and hardware-software co-design for real-time applications. He received the Outstanding Paper Award at the 38th IEEE RTSS. He is a member of the IEEE Computer Society.



**Sumaiya** received her B.Sc in Engineering Degree from Khulna University of Engineering and Technology (KUET), Bangladesh in 2020. Currently, she is a PhD candidate working towards a degree in Computer science at Wayne State University, Detroit, USA, under the supervision of Dr. Zheng Dong. Her research interests include Autonomous Driving, Real-time systems, Deep Learning, Edge assisted Machine Learning and vision systems, image processing, etc.