28th *Computer Vision Winter Workshop*
Graz, Austria, February 12–14, 2025

DOI: 10.3217/978-3-99161-022-9-006

Real-time object detection in diverse weather conditions through adaptive model selection on embedded devices

Mohammad Milad Tufan[1,2,3] Christian Fruhwirth-Reisinger[1,2] M. Jehanzeb Mirza[1,2]
Darko Štern[2,3]

[1]Institute of Visual Computing, Graz University of Technology
[2] Christian Doppler Laboratory for Embedded Machine Learning, Austria
[3]AVL List GmbH

m.tufan@student.tugraz.at, {reisinger, mirza}@tugraz.at, darko.stern@avl.com

Abstract

The perception system is a critical component of Advanced Driver Assistance Systems (ADAS) and Automated Driving (AD), playing a pivotal role in reducing traffic accidents caused by human error. For ADAS/AD systems to be seamlessly integrated into everyday life, it is essential to ensure the reliable operation of their perception systems, even under challenging conditions such as adverse weather. This paper presents a novel perception pipeline for real-time object detection with YOLOv3 across diverse weather scenarios. The pipeline incorporates adaptive model selection based on current conditions to optimize detection performance dynamically. To address the computational limitations of embedded systems in constraint environments, we propose a three-step approach: (1) reduction of YOLOv3 complexity using L^1 regularization for feature selection, followed by (2) weight pruning and (3) knowledge distillation to recover precision lost in earlier steps. This results in lightweight models up to 70% smaller than the base model while maintaining high precision through knowledge distillation. Finally, the optimized models are evaluated on resource-constrained embedded devices, including the NVIDIA Jetson AGX Orin, NVIDIA Jetson Nano, and Raspberry Pi 4, demonstrating robust and efficient performance under real-world conditions.

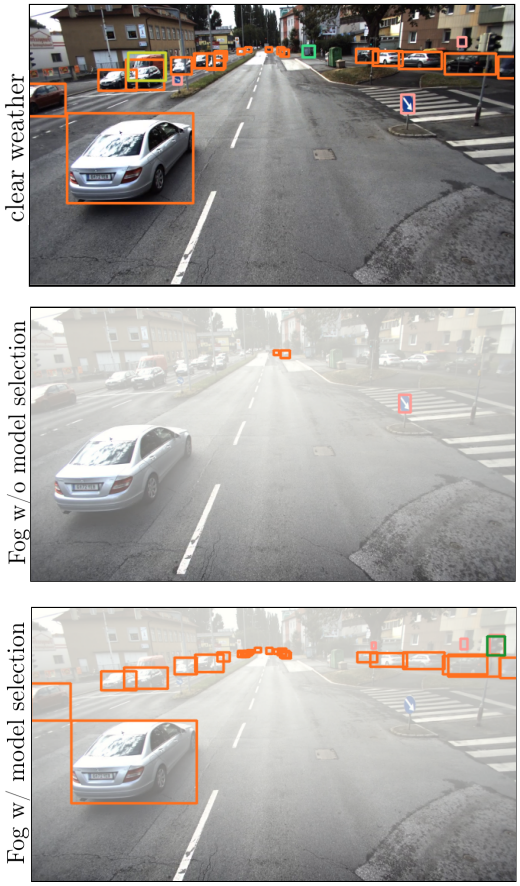

Figure 1. The detection examples demonstrate the need for adaptation in real-time. The second row shows detections with the clear weather model on a foggy image, while the last row shows detections with our adaptive perception pipeline on a foggy image. Fog was injected artificially into the clear weather image.

1. Introduction

Advanced Driver-Assistance Systems (ADAS) play a crucial role in enhancing road safety by mitigating risks associated with human error [2], which remains a leading cause of traffic accidents. According to the European Commission's 2021 accident report [8], approximately 100,000 traffic accidents involving personal injury occurred in the EU, with 20% resulting in fatalities. Human factors such as distraction, fatigue, or delayed reactions account for a significant proportion of these incidents. ADAS technologies have the potential to prevent many of these accidents or at least reduce their severity,

making their development and deployment critical.

Traditional ADAS functionalities, such as *forward collision warning*, *automatic emergency braking*, and *traffic sign recognition*, rely predominantly on rule-based systems. While effective in specific scenarios, these systems are highly application-specific and lack adaptability to diverse environments or evolving requirements. With the advent of deep learning, ADAS have gained significant versatility and accuracy, enabling tasks such as object detection, scene understanding, and environment perception. These capabilities form the foundation for both ADAS and automated driving (AD), where reliable detection of traffic participants is essential for safe and efficient operation. Despite the advancements brought by deep learning, these methods often require large-scale, meticulously annotated datasets to perform reliably. The process of collecting, storing, and labeling such data poses significant challenges. Memory and processing constraints further complicate the ability to save and review all recorded scenes, particularly in dynamic environments. Determining which scenes should be labeled for training or analysis is a complex task that relies on exhaustive data exploration, increasing time and resource costs.

A practical approach to this challenge is deploying lightweight, real-time object detection systems directly on the recording platforms. These systems serve as an initial filter to pre-select scenes containing relevant objects or events for further processing. By focusing on critical areas of interest, such as scenes with traffic participants or specific environmental conditions, such systems reduce the burden of exhaustive data storage and labeling while ensuring that the most informative samples are identified. Although accuracy is not the primary goal in this context, detectors with higher precision naturally lead to better-informed data selection decisions, ultimately enhancing the performance of subsequent deep learning pipelines.

In this paper, we propose a perception pipeline that dynamically adapts to various weather conditions via model selection and runs in real-time on embedded platforms with constrained resources. In particular, we train a YOLO [26] expert for each weather scenario, *i.e.*, *clear*, *rain*, and *fog* to deal with occurring distribution shifts. While inference, a weather domain classifier decides which model to use. The proposed expert selection design ensures precise detections in dynamic environments, as shown in Fig. 1. To reach real-time performance on embedded devices, we follow two strategies: 1) model pruning and 2) tiny models. In both cases, we perform knowledge distillation from the base model to achieve adequate performance. Our contributions are as follows:

- We propose a real-time perception pipeline, depicted in Fig. 2, deployable on various embedded devices. This pipeline tackles distribution shifts by first recognizing the domain and, secondly, switching to an appropriate expert model.
- With a sparse training and pruning procedure, we reduce model size and complexity to perform real-time perception on edge devices. Afterward, we distill knowledge [15] from the base model to regain the precision lost in the pruning process.
- Finally, we perform exhaustive evaluations in clear and adverse weather conditions and provide a detailed runtime and memory analysis on various edge devices.

2. Related work

Object detection. The localization and classification of objects is a crucial task in many challenging real-world applications like robotics [13, 30] or autonomous driving [1]. It becomes even more challenging when applied in constrained environments like embedded devices [33], especially when real-time processing is required. To that end, object detection has been extensively researched in the past. Examples are EfficientDet [31], DEtection TRansformer (DETR) [4], or CenterNet [10]. Object detectors can be separated into two categories: By a two-stage detector using region proposals [12, 27] or by a one-stage detector with a unified network architecture that treats object detection as a regression problem [19, 20]. In our pipeline, we apply the YOLO [26] object detector. It has a lightweight architecture that applies only a single stage and provides satisfactory results. With appropriate optimization, it can run in real time on embedded devices.

Distribution shifts. In Machine Learning, we draw training samples from a distribution p_{train} that we assume to be independent and identically distributed (i.i.d.). Further, we assume our inference data to be drawn from the same or at least very similar distribution $p_{inference}$. However, since not all samples of a highly dynamic environment are known in advance, *e.g.*, training on clear weather and inference on rainy weather, unknown samples inevitably lead to a distribution shift such that $p_{inference} \neq p_{train}$. The need for adaptive model selection arises to handle adverse weather conditions properly. Pérez-Gállego et al. [23] tackle model selection for quantification tasks. Due to distribution shifts in data for quantification problems, they employ dynamic quantifier ensemble selection to select a model trained on a dataset most similar to the given test sample. To effectively select a model that precisely predicts the next sample in time series forecasting on data streams, Boulegane et al. [3] employ Multi-Target Regression (MTR). Given an ensemble of models, they assume that for temporal data streams, each model in the ensemble is an expert in some area of the stream. To select the model, they simultaneously assess each model in an ensemble based on its ability to produce a good result for the given test sample.

DILAM [18] addresses distribution shifts using incremental learning through activation matching to prevent catastrophic forgetting. They store affine transformations (scale γ and shift β) of batch normalization layers [16] in a memory bank. For each target domain, the models are adapted, and their corresponding affine transfor-

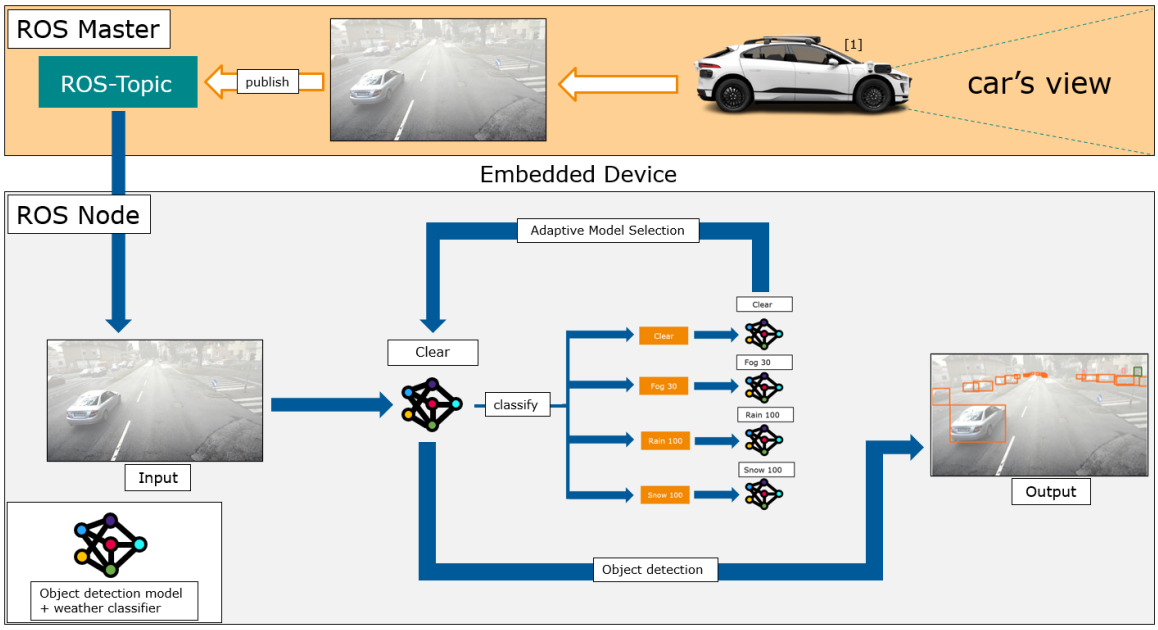

Figure 2. The Pipeline framework consists of two stages: data acquisition (top) and perception (bottom). In the data acquisition stage (ROS-master), a sensor setup composed of multiple cameras and other sensors captures road images. The second stage (ROS-node), receives the data, recognizes the current domain, and performs object detection with an expert domain model. Image marked with [1] was adapted from [1].

mations are stored in a memory bank. During inference, their plug-and-play framework seamlessly substitutes the model's current transformations with the pre-stored transformations specific to the target domain based on a learned domain classifier. However, changing weights during runtime as in [18] is inefficient. For embedded devices, models are converted to TensorRT, which makes changing parameters during runtime inefficient. Instead, we adopted model selection, which ensures better real-time performance and responsiveness on embedded devices.

Model compression. The need for model compression arises due to the immense network parameters in recent detection architectures. Due to memory and processing power restrictions, state-of-the-art models are unsuitable for edge devices. Reducing the model complexity and, thus, model size requires less memory to store the model. This enables us to store the model directly on the much faster on-chip memory, compared to the slow off-chip DRAM, a large storage area outside the CPU [28].

LeCun et al. [17] first show that a network can be pruned by removing weights that do not significantly affect the models' performance. However, this approach is an unstructured pruning technique that does not consider the network structure and its layers. This method is suitable for dense layers where the weights are independent. However, this leads to problems for convolution layers where kernels share weights across spatial locations. In contrast to [17], Polyak and Wolf [24] preserve the structure of the neural network and apply channel-level pruning. They either prune each layer's input or output channels. The task is to first identify the importance of each channel by looking at the activation output variance and then filter and eliminate insignificant ones. Polyak and Wolf [24] tackle this issue by eliminating channels with the least contribution variance. Unlike unstructured pruning [17] or channel-level pruning [24], our approach simplifies pruning by pre-filtering insignificant weights using an L^1 penalty, making structured pruning more effective.

Knowledge distillation. The idea of knowledge distillation [15] is to transfer knowledge from a larger, highly accurate teacher model to a smaller, less accurate student model by computing a soft loss with the predictions of the teacher network on top of the data loss. Sau and Balasubramanian [29] extend knowledge distillation by making a student network learn from multiple teachers via logit perturbation. However, they do not directly employ multiple teacher networks but inject noise and perturbations into the teacher outputs. By doing so, they effectively simulate multiple teachers. Moreover, injecting noise into the teacher outputs introduces noise in the loss, thus creating a regularization effect. Chen et al. [6] extend the knowledge distillation workflow from [15] by considering activation responses from intermediate layers of the teacher network. This guides the student network in the correct direction and improves its accuracy. Our work draws inspiration from [6] in using knowledge distillation to improve object detection. However, instead of using intermediate layer activations, we focus on distilling knowledge through a combination of classification and bounding box loss com-

[1] https://www.roadtoautonomy.com/metaverse-waymo-spending/.

puted with the final teacher and student outputs.

3. Method

During data recording, on-device filtering of data samples is crucial to minimize unnecessary memory consumption and processing costs for subsequent labeling or inspection. We present our object detection pipeline designed for diverse weather conditions through adaptive model selection. In addition, we provide a pruning and knowledge distillation strategy for real-time detection on embedded devices that creates highly optimized models.

3.1. Perception pipeline

Framework. With our detection pipeline, we aim to detect objects in real-time on embedded devices. It consists of various YOLOv3 detection models, each an expert for a specific weather condition (*i.e.*, *clear*, *rain* and *fog*), and a weather recognition module. After classifying the prevailing weather, the appropriate model for detection is selected and applied. We integrate our pipeline into a framework based on the Robot Operating System (ROS) [25] as illustrated in Fig. 2. It consists of two stages: data acquisition (top) and perception (bottom). In the first stage, data acquisition, a sensor setup composed of multiple cameras and optional other sensors such as LiDAR acts as the ROS master, providing sensor recordings as a data stream. The second stage is a ROS node that receives incoming data from the previous stage and runs our perception pipeline on embedded devices like the Jetson AGX Orin, Jetson Nano, or Raspberry PI 4.

Weather recognition. Similar to Leitner et al. [18], we reuse layers from the YOLOv3 backbone and employ a linear classification head to recognize the weather conditions. Therefore, we reduce the additional overhead from a separate model to a tiny linear layer. Furthermore, to get a model that performs one single forward pass, we integrate the weather recognition into the forward pass of the object detection. During the weather recognition model training, we only adjust the weights of the classification head and leave the rest of the network frozen.

3.2. Model compression

Embedded devices are constrained in resources and performance. Therefore, model compression is needed to reduce the network complexity, speeding up the inference. As illustrated in Fig. 3, model compression consists of multiple steps. To effectively reduce the model size, we first need to eliminate redundant weights from the model. However, the question of which weights are essential and which can be safely pruned without affecting the performance of the resulting pruned model arises. We start by looking at our base model, YOLOv3, which follows the YOLO network architecture and shares a common pattern throughout the network: a convolution layer followed by a batch normalization (BN) [16] layer. To effectively leverage this structure, we look closer at the BN layers.

Batch normalization (BN) [16] in deep neural networks improves stability while training and speeds up the convergence of the model. The BN layers first normalize the input and afterward scale and shift it to reduce internal covariance shifts [32]. The normalization process of BN layers is described as follows:

$$z = \frac{x_{in} - \mu_{x_{in}}}{\sqrt{\sigma_{in}^2 + \epsilon}}, \tag{1}$$

where x_{in} is the output of the previous convolution layers, μ_{in} the mean of x_{in} and σ_{in}^2 the variance. By normalizing the input, we will have zero mean and unit variance. This, however, decreases the representational power of the network. BN layers introduce γ and β parameters to retain the representational power. These parameters need to be learned by the network, where the β parameter learns the optimal shift for each BN layer and γ the optimal scaling factor. The output of BN Layers is described as follows:

$$z_{out} = \gamma \cdot z + \beta, \tag{2}$$

where z is the normalized data [21].

Sparse training. Inspecting Equ. 2 in detail, we can conclude that a smaller BN scale factor γ indicates less influence on the corresponding channel in the convolution layer. Hence, we aim to get a network structure where only a few key features of the network (high γ) are responsible for the final detection result while most channels have a γ close to zero [21]. However, the model should still learn a meaningful representation and provide highly accurate detections. Afterward, we can safely prune away the convolution channels along with the corresponding scale and shift factors γ and β for channels contributing minimally to the final detections.

We employ sparse training to get such a sparse network representation of the YOLOv3 model, where the detection result depends only on a small number of key features within the network. Leveraging a technique called proximal gradients, we compute the gradients w.r.t the regular YOLOv3 loss function. Afterward, we apply a proximal operator, namely, a soft-thresholded L^1 penalty on the γ factors of the BN layers. By applying soft thresholding, we encourage the γ factors to become zero or close to zero and hence induce sparsity into the network. The final loss function during sparse training is described as follows:

$$L = \sum_{(x,y)} l(f(x,\theta), y) + \lambda \sum g(\gamma), \tag{3}$$

where $l(f(x,\theta), y)$ is the YOLOv3 loss using the parameter vector θ and $g(\gamma)$ is the soft-thresholded L^1 penalty on the BN γ factors described as follows:

$$g(\gamma) = sign(\gamma) \cdot max(|\gamma| - \tau, 0). \tag{4}$$

τ denotes the threshold. Every value below this threshold will be set to zero. The hyperparameter λ represents the balance between YOLOv3 loss and L^1 penalty [21].

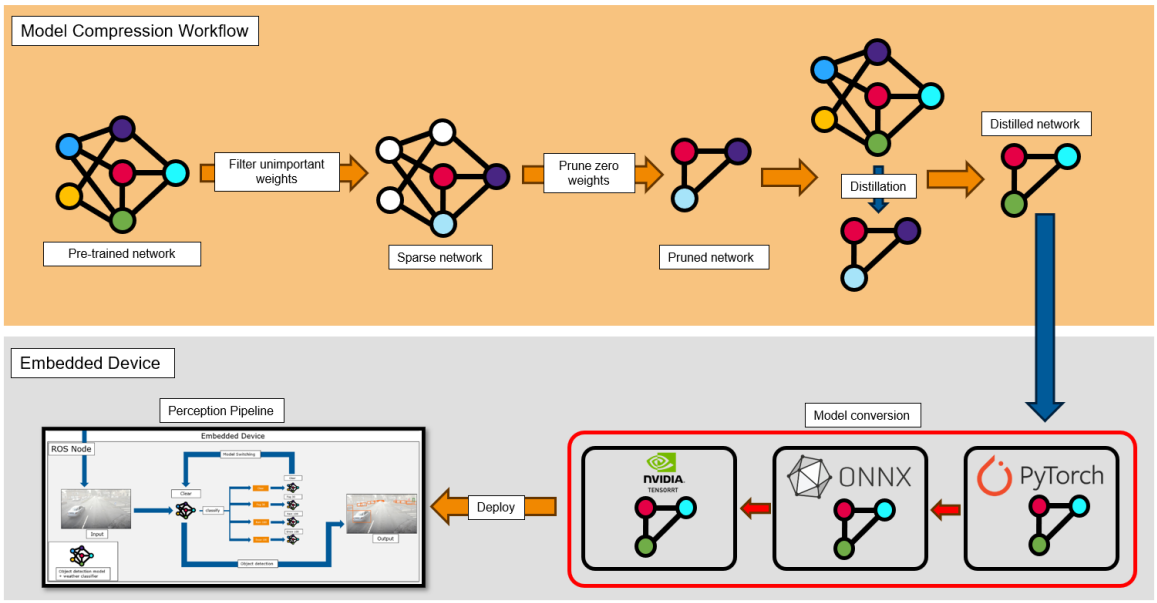

Figure 3. Overview of our method. Firstly, we sparsely train a network to filter unimportant weights. Secondly, the insignificant weights from the sparse network are pruned. Thirdly, the initial large pre-trained network distills knowledge into the pruned network. Afterward, we convert this distilled network from PyTorch to ONNX and TensorRT on the embedded device. Finally, we apply the pruned and converted model within our perception pipeline.

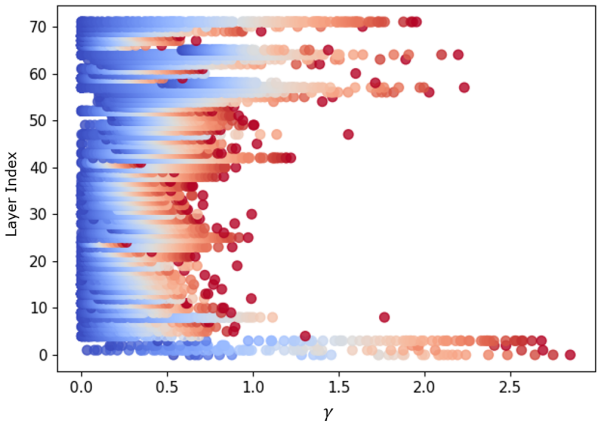

Figure 4. Distribution of BN scale factor γ after sparse training. Here we can see that not all layers have the same importance. The earlier layers have high γ values and thus are more important than the middle layers. We can also see that nearly most of the weights of the middle layers go to 0 after sparse training and hence can be pruned away safely.

Pruning. In the context of neural networks, pruning describes a technique used to eliminate weights from a network based on their importance selectively. After sparse training, we can start pruning insignificant weights from the network. The result is a model reduced in size, which also increases the inference speed. However, not all layers of the YOLO network have the same importance as shown in Fig. 4. Hence, it is crucial to prune weights also depending on the importance of layers. Following Chu et al. [7], we flatten all BN γ factor vectors in the network, concatenate them into one vector, and set a pruning percentage to determine a threshold. Every value below this threshold will be set to zero. Afterward, we remove every zero weight in bias, scale γ, and shift β vectors. In a subsequent step, we create a mask of the same length as each layer's corresponding γ vector. This mask has the value one if there is a non-zero entry at the corresponding γ vector index and zero if there is a zero entry. We can safely prune convolution layers with this mask by removing each convolution channel, where the corresponding mask entry is zero [7].

Knowledge distillation. After reducing the complexity of the model and pruning weights that we have deemed insignificant in previous sections, the model's performance may deteriorate compared to the larger base version of the network. The reason is that we may have pruned too much of the network or that weights we had deemed insignificant since they were relatively small were, in fact, significant. To counter that, we use a technique called knowledge distillation [6, 15]. Knowledge distillation describes a technique in Machine Learning where a minor student network is taught by a more extensive teacher network how to perform a specific task. In our case, the small student network is the pruned version of the base network, and the teacher network is the base version. We sample from the training dataset and run this sample through both the teacher and the student network. In the first step, we use the student network predictions and ground truth labels to compute the YOLOv3 loss. The second step is to compute the loss with the predictions of the teacher network. We first transform the teacher predictions into soft labels using the softmax function and a temperature pa-

rameter T, which controls the smoothness of the output distribution [5, 15]. The computation of the soft labels is described as follows:

$$t_{out} = \sigma\left(\frac{e^{z_i/T}}{\sum_{j=1}^{N} e^{z_j/T}}\right)_i, \tag{5}$$

where z_i denotes the i^{th} output of the teacher network and σ the softmax activation function. Using these soft labels t_{out}, we compute Kullback-Leibler (KL) divergence with the student class predictions s_{out} as follows:

$$KL_{loss} = \frac{KL(\log(\sigma(s_{out})), \sigma(t_{out}) \cdot T^2}{batch\_size}, \tag{6}$$

where

$$KL(P||Q) = \int_{-\infty}^{\infty} p(x) \cdot \log\left(\frac{p(x)}{q(x)}\right) dx. \tag{7}$$

We use the teacher output in the loss calculation because the output of the teacher network carries significant information about relations and similarities of the predicted output. Objects similar to the actual label will have high probabilities. For instance, we expect a detector trained on a dataset with three classes, *i.e.*, *car*, *truck*, *pedestrian*, to have class label predictions for a car be close to that of a small truck but far apart from pedestrians. This example demonstrates a semantic relation between cars and trucks, which can not be incorporated with only the ground truth labels for loss calculation. Furthermore, to also consider the bounding box error, we compute a box loss with teacher-predicted bounding boxes and student bounding boxes as follows:

$$box_{loss} = \frac{1}{N}\sum_{i=1}^{N}(BS_i - BT_i)^2, \tag{8}$$

where BS_i is the i^{th} student bounding box and BT_i the i^{th} teacher bounding box.

Model conversion. It is insufficient to only reduce the model complexity to deploy object detection models onto embedded devices. Therefore, we need to change their structure to use the acceleration provided by embedded devices efficiently. As illustrated in Fig. 3, after successfully compressing the models, the next step is to optimize them for use on the embedded devices. In this paper, we achieve this by converting the models into the TensorRT [9] format that automatically derives essential information on how to use the underlying GPUs efficiently or accelerate inference times by restructuring the model.

4. Experiments

The experiments are split into two parts: the performance of the models tested on a workstation with high-end GPU and the performance of the models on embedded devices. First, we provide model performances in terms of precision and model size. Afterward, we investigate the inference speed of our models measured in FPS on various embedded devices.

4.1. Dataset

In this paper, we conduct experiments on the KITTI-Dataset [11]. This dataset provides a comprehensive resource for developing and evaluating autonomous driving systems. The KITTI dataset is recorded in sunny weather conditions. However, we need additional data alongside the sunny dataset to conduct experiments for adverse weather conditions. Mai et al. [22] add artificial fog and Halder et al. [14] add artificial rain to the KITTI-clear images. KITTI-rain consists of eight severities of rain, and KITTI-fog consists of seven different severities of fog.

4.2. Implementation details

For our initial training of the YOLOv3 network, we use a batch size of 32 and a learning rate of 0.0001. We train the network for 500 epochs in total. For the sparse training, we set α=0.01, which controls the step size for our proximal gradient, and sr=0.001, which controls the sparsity level. In the knowledge distillation setup, we use a batch size of 8 and train for 2000 epochs to give the teacher network enough time to teach the student network.

All models and training scripts are implemented in Python 3.8 and PyTorch 2.0.1. We train and test our models on an NVIDIA RTX 4090 GPU. For experiments on embedded devices, we run the models on three different platforms: NVIDIA Jetson AGX Orin, NVIDIA Jetson Nano, and Raspberry PI 4 Model B.

4.3. Baselines

The first step for our experiments is to get baseline models for validating the performance of our pruned and distilled models. Table 1 shows the results for our models trained on their respective domain, *e.g.*, clear, fog 30m, fog 50m, and rain 200mm/h. From these experiments, we can see that we achieve a mean average precision of 96.24 at an IoU threshold of 0.5 ($mAP@.5$) for a large float32 precision model (large-32) trained on the clear domain. Furthermore, we observe that the performance of our model for the most challenging weather condition, fog 30m, is sufficiently good, with a $mAP@.5$ of 93.36.

We can see that the model's mAP does not decrease significantly when we quantize the weights from float32 precision to float16. Additionally, we can see that the tiny models denoted as tiny-32 and tiny-16 (for float32 and float16 precision) have significantly lower mAP than our large YOLOv3 model. This is because these tiny models are 86% smaller than their base variants. Considering the model's small size, its performance is remarkable.

4.4. Performance evaluation

Before deploying our models on the NVIDIA Jetson AGX Orin, NVIDIA Jetson Nano, and the Raspberry PI 4

Model	KITTI-clear	KITTI-fog 30m	KITTI-fog 50m	KITTI-rain 200mm/h	model size
large-32	96.24	93.36	94.64	95.72	246.70 MB
large-16	96.26	93.32	94.78	95.81	123.35 MB
tiny-32	70.68	57.58	60.32	67.00	34.8 MB
tiny-16	70.70	57.55	60.26	66.93	17.8 MB

Table 1. Mean average precision at an IoU threshold of 0.5 ($mAP@.5$), when testing models on their respective domains.

Model B, we need to make sure our model's performance measured in Average Precision (AP) and mean Average Precision (mAP) is satisfactory.

Pruned Models. Firstly, we need to mention that sparse training before pruning is absolutely crucial. The reason is that before sparse training unimportant weights have not been identified, therefore leading the pruning process to eliminate weights that are crucial to do object detection. During our experiments, we could not recover lost precision during pruning if we did not perform sparse training before pruning.

To show the efficiency of our pruning pipeline, which keeps the structure of the network intact, and our knowledge distillation pipeline, which recovers lost precision during pruning, we prune our large-32 and large-16 baseline models with different percentages, *i.e.*, 30%, 50%, and 70%. Table 2 shows the $AP@.5$ and the $mAP@.5$ for models pruned with different percentages and tested on the KITTI-clear weather domain. We can see that the Pruned models, denoted as pruned-32-30 and pruned-16-30, perform well compared to our baseline models named large-32 and large-16. Both models perform pretty well on the clear domain with a $mAP@.5$ of 93.79 and 93.78. Even if we prune 50% of the network weights, the AP does not decrease drastically.

In Fig. 4 we can see that the earlier and later layers in the YOLO network are the most important for the object detection task after the sparse training. By pruning 70% of the network, we can observe a significant drop in AP. This observation indicates that we have already eliminated significant weights from the earlier and later layers. When pruning 90% of the weights, our model degenerates and cannot detect objects anymore.

To increase the performance of the tiny models (tiny-32 and tiny-16) we apply our proposed knowledge distillation pipeline to boost its ability to detect objects. By doing so, we increase the mAP of the YOLOv3-tiny models by around 12 points, from 70.70 to 82.48. When comparing this tiny model to our large-32 baseline, we can see that we only have a 14-point difference in $mAP@.5$. This precision is particularly good considering that the tiny model is 86% smaller than the large-32 model.

4.5. Performance on embedded devices

After verifying our models' results on a regular PC and reducing the model's complexity, we deploy them onto resource- and performance-constrained embedded devices. In this case, we are mainly interested in the inference speed of our models when tested on the KITTI dataset. Table 3 shows our results. We can see that for the large models (246.7 MB), we achieve 93.45 FPS when running inference on images of size 416×416 on the Jetson AGX Orin. The FPS drops considerably when we deploy the same model onto a smaller embedded device like the Jetson Nano or Raspberry PI 4. Due to the compact size and performance constraints of both of these smaller embedded devices, the performance drops by 97.52% and 99.78%, respectively. To reach the domain of real-time inference, we need to achieve at least 24 FPS to outperform the sensor recording frequency. Pruning 30% of the network weights increases our FPS from 93.45 to 100.00 on the Jetson Orin. We are more than twice as fast on the Jetson Nano and the Raspberry PI 4.

To finally reach real-time inference even on the Jetson Nano, we utilize the YOLOv3-tiny model. This model has a significantly smaller network architecture. We can see that this model is 86% smaller than our large-32 model. With this model, we achieve 23.8 FPS, and by quantizing the model's weights to float16, we achieve 31.25 FPS. On the Raspberry PI 4, we increased the performance by more than 9 times compared to the large-32 model by using the tiny model.

Furthermore, we can also see that for the Raspberry PI 4, the float32 precision models are faster than the float16 precision models even though the latter are smaller. This is because The Raspberry has no GPU and thus does not provide GPU acceleration. Furthermore, CPUs have native support for float32 and can, therefore, handle them more efficiently than float16. Reducing the complexity of our models leads to a reduction in the number of floating point operations (FLOPS).

5. Limitations and future work

Our object detection and weather classification model is limited by how many domains it can effectively work with. We have trained both the object detection part and the weather classification part with synthetic data. Therefore, we will encounter a significant drop in precision for any domain unavailable at training time. To ensure the seamless operation of the perception system under a broader range of adverse weather conditions, including rain and snow, it is crucial to have large and diverse publicly available datasets. These datasets will support the

Model	Tested on KITTI-clear								mAP@.5	Model size
	Car	Van	Truck	Ped	Psit	Cyc	Tram	Misc		
large-32	98.26	98.81	99.28	90.53	90.38	96.30	98.97	97.37	96.24	246.70 MB
large-16	98.26	98.79	99.27	90.60	90.38	96.54	98.93	97.35	96.26	123.35 MB
pruned-32-30	97.84	97.91	98.91	86.11	80.60	93.47	99.03	96.46	93.79	173.40 MB
pruned-16-30	97.82	97.93	98.91	86.14	80.62	93.36	99.03	96.45	93.78	86.70 MB
pruned-32-50	97.22	97.41	98.35	84.80	86.20	91.64	98.40	94.60	93.56	134.00 MB
pruned-16-50	97.20	97.40	98.34	84.61	86.23	91.51	98.41	94.60	93.52	67.00 MB
pruned-32-70	93.28	91.97	93.35	70.61	60.94	77.88	87.32	75.82	81.40	76.30 MB
pruned-16-70	93.24	91.93	93.36	70.39	61.28	77.76	87.20	76.06	81.40	38.15 MB
tiny-32	86.32	74.83	81.97	59.96	59.46	65.15	80.83	56.95	70.68	34.8 MB
tiny-16	86.41	75.07	82.09	60.20	59.00	64.89	80.82	57.12	70.70	17.8 MB
tiny-32-kd	92.71	90.14	93.51	68.66	64.18	78.68	90.87	81.10	82.48	34.8 MB
tiny-16-kd	92.67	90.02	93.55	68.79	66.03	78.61	90.88	81.02	82.70	17.8 MB

Table 2. Average Precision (AP) and mean Average Precision ($mAP@.5$) for all KITTI classes using models with different pruning percentages.

Model	Tested on Architecture			Model size	GFLOPS
	Jetson AGX Orin	Jetson Nano	Raspberry PI 4		
large-32	93.45 / 80.65	2.32 / 2.04	0.32 / 0.21	246.70 MB	32.75 / 49.61
large-16	111.11 / 103.09	3.90 / 3.26	0.26 / 0.19	124.30 MB	32.75 / 49.61
pruned-32-30	100.00 / 90.90	5.55 / 4.78	0.61 / 0.49	173.40 MB	14.10 / 21.36
pruned-16-30	116.27 / 108.70	8.69 / 7.29	0.43 / 0.41	86.50 MB	14.10 / 21.36
pruned-32-50	103.09 / 94.33	7.09 / 5.99	0.83 / 0.57	134.00 MB	10.36 / 15.69
pruned-16-50	120.04 / 117.64	10.75 / 9.09	0.72 / 0.48	67.00 MB	10.36 / 15.69
pruned-32-70	111.11 / 100.00	9.90 / 8.00	1.31 / 0.84	76.30 MB	6.31 / 9.55
pruned-16-70	149.25 / 125.03	15.38 / 11.76	1.08 / 0.75	38.15 MB	6.31 / 9.55
tiny-32	181.82 / 142.85	23.80 / 20.19	3.09 / 2.32	34.80 MB	2.75 / 4.15
tiny-16	212.76 / 176.42	31.25 / 27.89	3.04 / 1.53	17.80 MB	2.75 / 4.15

Table 3. Average FPS and model size on different architectures. The values in the Model column consist of the model name, the precision, and the percentage of weights pruned. The values in the Tested on Architecture column represent the FPS. The first value denotes the FPS when running inference on images of 416×416 pixels; the second value represents the same but with images of 512×512 pixels. The FPS values are averaged over 3781 (test set) iterations. The values in the GFLOPs column represent the Giga Floating Point Operations per Second. Again, the first value denotes inference using an image of size 416×416 and the second represents inference using an image of size 512×512.

advancement of unsupervised domain adaptation by enabling improvements in robust domain recognition and the refinement of object detection capabilities.

6. Conclusion

In this paper we studied real-time object detection in diverse weather conditions through adaptive model selection on embedded devices. We particularly focused on gaining a deeper understanding of model compression techniques to reduce model complexity and enable real-time applications on performance- and resource-constrained embedded devices. The key findings entail:

- Filtering insignificant network weights is essential to reduce precision loss during pruning and to make the model rely on key features for object detection only.
- Knowledge distillation is a suitable technique to regain the lost precision after pruning.
- Our proposed perception pipeline ensures real-time object detection on embedded devices. It recognizes known domains, selects a suitable model, and performs robust real-time object detection without interruptions.

These findings emphasize the significance of a structured approach that reduces model size to increase inference speed on embedded devices. This enables automatic driving applications to run robustly in real time and adapt to adverse weather conditions, such as fog or rain.

Acknowledgements

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged. Furthermore, we gratefully acknowledged the financial support of the Austrian Federal Ministry of Climate Action, Environment, Energy, Mobility, Innovation and Technology, the Austrian Federal Ministry of Digital and Economic Affairs, in the frame

of the Important Project of Common European Interest (IPCEI) on Microelectronics and Communication Technologies (ME/CT) implemented by austria wirtschaftsservice (aws) and the Austrian Research Promotion Agency (FFG).

References

[1] Pedro Azevedo and Vítor Santos. Yolo-based object detection and tracking for autonomous vehicles using edge devices. In *Proc. ROBOT*, 2022. 2

[2] Juan Borrego-Carazo, David Castells-Rufas, Ernesto Biempica, and Jordi Carrabina. Resource-constrained machine learning for ADAS: A systematic review. *IEEE Access*, 8:40573–40598, 2020. 1

[3] Dihia Boulegane, Albert Bifet, Haytham Elghazel, and Giyyarpuram Madhusudan. Streaming time series forecasting using multi-target regression with dynamic ensemble selection. In *IEEE BigData*, 2020. 2

[4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *Proc. ECCV*, 2020. 2

[5] Alexandros Chariton. Knowledge Distillation Tutorial. https://pytorch.org/tutorials/beginner/knowledge_distillation_tutorial.html, 2024. Online; accessed March 14, 2024. 6

[6] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In *Proc. NeurIPS*, 2017. 3, 5

[7] Yun Chu, Pu Li, Yong Bai, Zhuhua Hu, Yongqing Chen, and Jiafeng Lu. Group channel pruning and spatial attention distilling for object detection. *AI*, 52(14):16246–16264, 2022. 5

[8] European Commision. Annual accident report. https://road-safety.transport.ec.europa.eu/european-road-safety-observatory/statistics-and-analysis-archive/annual-accident-report_en, 2021. Online; accessed May 23, 2024. 1

[9] NVIDIA Developer. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2024. Online; accessed April 15, 2024. 6

[10] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *Proc. ICCV*, 2019. 2

[11] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? the KITTI Vision Benchmark Suite. In *Proc. CVPR*, 2012. 6

[12] Ross Girshick. Fast r-cnn. In *Proc. ICCV*, 2015. 2

[13] Brent Griffin. Mobile robot manipulation using pure object detection. In *Proc. WACV*, 2023. 2

[14] Shirsendu Sukanta Halder, Jean-François Lalonde, and Raoul de Charette. Physics-Based Rendering for Improving Robustness to Rain. In *Proc. ICCV*, 2019. 6

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv*, abs/1503.02531, 2015. 2, 3, 5, 6

[16] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. ICML*, 2015. 2, 4

[17] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In *Proc. NeurIPS*, 1989. 3

[18] Stefan Leitner, M Jehanzeb Mirza, Wei Lin, Jakub Micorek, Marc Masana, Mateusz Kozinski, Horst Possegger, and Horst Bischof. Sit Back and Relax: Learning to Drive Incrementally in All Weather Conditions. In *Proc. IV*, 2023. 2, 3, 4

[19] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proc. ICCV*, 2017. 2

[20] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Proc. ECCV*. Springer, 2016. 2

[21] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proc. ICCV*, 2017. 4

[22] Nguyen Anh Minh Mai, Pierre Duthon, Louahdi Khoudour, Alain Crouzil, and Sergio A. Velastin. 3D Object Detection with SLS-Fusion Network in Foggy Weather Conditions. *Sensors*, 21(20), 2021. 6

[23] Pablo Pérez-Gállego, Alberto Castaño, José Ramón Quevedo, and Juan José del Coz. Dynamic ensemble selection for quantification tasks. *IF*, 45:1–15, 2019. 2

[24] Adam Polyak and Lior Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015. 3

[25] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *Proc. ICRAW*, 2009. 4

[26] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proc. CVPR*, 2016. 2

[27] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proc. NeurIPS*, 2015. 2

[28] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. A Comprehensive Survey on Model Quantization for Deep Neural Networks in Image Classification. *ACM TIST*, 14 (6), 2023. 3

[29] Bharat Bhusan Sau and Vineeth N Balasubramanian. Deep model compression: Distilling knowledge from noisy teachers. *arXiv*, abs/1610.09650, 2016. 3

[30] Ruohuai Sun, Chengdong Wu, Xue Zhao, Bin Zhao, and Yang Jiang. Object recognition and grasping for collaborative robots based on vision. *Sensors*, 24(1):195, 2023. 2

[31] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proc. CVPR*, 2020. 2

[32] Eric Wong. Distribution shift. https://riceric22.github.io/assets/debugml/distribution_shift.pdf, 2022. Online; accessed April 27, 2024. 4

[33] Zhengxia Zou, Keyan Chen, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *IEEE*, 111(3):257–276, 2023. 2