TUM School of Computation, Information and Technology
Technische Universität München

Hardware Aware Robust Compression of Neural Networks

Manoj Rohit Vemparala

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr.-Ing. habil. Erwin Biebl

Prüfende der Dissertation:
1. apl. Prof. Dr.-Ing. Walter Stechele
2. Prof. Maurizio Martina, Ph.D.,
   Politecnico di Torino

Die Dissertation wurde am 12.07.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 13.12.2022 angenommen.

Contents

List of Figures

List of Tables

Acronyms

AD Autonomous Driving.

AOC Altera Offline Compiler.

ASIC Application-Specific Integrated Circuit.

BEV Bird's Eye View.

BNN Binarized Neural Network.

BOPs Bit Operations.

CAM Class Activation Map.

CNN Convolutional Neural Network.

CR Compression Ratio.

CTC Computation-to-Communication.

CUDA Compute Unified Device Architecture.

CV Computer Vision.

DDPG Deep Deterministic Policy Gradient.

DNN Deep Neural Network.

ES Evolutionary Search.

FastAT Fast Adversarial Training.

FC Fully Connected.

FGSM Fast Gradient Sign Method.

FPGA field programmable gate array.

FPS Frames Per Second.

GA Genetic Algorithm.

GEMM General Matrix-Matrix Multiplication.

GOPS Giga Operations Per Second.

GPU Graphics Processing Unit.

HDL hardware description language.

HIL Hardware-In-the-Loop.

HLS High-Level Synthesis.

HOF Hall Of Fame.

HW hardware.

Ifmap Input Feature Map.

IRO Input Reuse Order.

KD Knowledge Distillation.

KITTI Karlsruhe Institute of Technology and Toyota Technological Institute.

KL Kullback Leibler.

KPI Key Performance Indicator.

LiDAR Light Detection and Ranging.

MAC Multiply–Accumulate.

mAP Mean Average Precision.

MBN Multi Bit Networks.

NAS Neural Architecture Search.

NCC normalized compute complexity.

NSGA Non Sorting Genetic Algorithm.

Ofmap Output Feature Map.

Op Operation.

ORO Output Reuse Order.

Param parameter.

PE Processing Element.

PGD Projected Gradient Descent.

PR Pruning Rate.

PTQ Post Train Quantization.

QAT Quantization Aware Training.

ReLU Rectified Linear Unit.

RL reinforcement learning.

RoI region of interest.

RTL register-transfer level.

SGD Stochastic Gradient Descent.

SPG Stochastic Policy Gradient.

STE Straight Through Estimator.

WRO Weight Reuse Order.

Acknowledgements

I am deeply grateful for the amazing journey that has led to the completion of this thesis. I have been blessed with the support and encouragement of so many incredible people, and I want to express my heartfelt thanks to each and every one of them.

First and foremost, I would like to thank Prof. Walter Stechele for supervising my thesis and providing me with the opportunity to work on this amazing topic. His careful and precious guidance was extremely valuable during the course of the thesis.

I also would like to thank my colleagues, partners and dear friends Alexander Frickenstein, Nael Fasfous for their dedicated help, advice, inspiration, encouragement and continuous support, throughout the project. Their precious feedback was instrumental in making this thesis a reality.

I am also deeply indebted to the amazing team at BMW, who have accompanied and supported my thesis in the past years. Especially, my association with Naveen Shankar Nagaraja, Lukas Frickenstein, Pierpaolo Mori, Anmol Singh and Shambhavi Sampath helped me to improve my knowledge in the research topics investigated in the scope of the thesis.

I would like to thank my department heads Hariolf Gentner, Florain Homm and Hans Joerg Voegel who created an environment that allowed me to thrive and grow.

To my dear friends Kiran and Rohith, thank you for your unwavering support and encouragement during this entire journey.

Finally, I want to express my deepest gratitude to my family, including my mom, dad, wife, and sister, whose love, patience, and support have sustained me during my doctoral studies.

Zusammenfassung

CNN erreichen höchste Leistungen bei Computer-Vision-Anwendungen wie Bildklassifizierung, semantische Segmentierung und Objekterkennung. Die verbesserte Leistung dieser Netze geht jedoch auf Kosten der Größe des Modells und der Komplexität der Berechnungen. Eine effiziente Verarbeitung dieser Netze ist von entscheidender Bedeutung, um ihren Einsatz auf rechen- und speicherbeschränkten Zielhardwareplattformen zu ermöglichen. Mit Hilfe von Kompressionsverfahren wie Pruning und Quantisierung können wir den Rechen- und Speicherbedarf der CNN-Inferenz reduzieren. Wir nutzen Hardware-Modelle und Scheduling-Schemata zur Abschätzung der Ausführung, um Suchalgorithmen Rückmeldung zu geben, die effiziente Kompressionskonfigurationen bestimmen. Wir modellieren speziell einen CNN-Beschleuniger auf Basis eines Spatial Arrays und nutzen die auf Reinforcement Learning basierende Exploration, um hardwarebewusste Pruning-Konfigurationen abzuleiten. Bestehende Suchalgorithmen, die Kompressionskonfigurationen ableiten, erfordern ein vortrainiertes Modell in Fließkommazahlen und einen hohen Rechenaufwand aufgrund der iterativen Feinabstimmungsphase. Wir reduzieren den Rechenaufwand der Kompressionsphase, indem wir In-Train-Optimierungstechniken vorschlagen, die während des gradientenbasierten Lernprozesses effiziente Beschneidungs- oder Quantisierungsstrategien bestimmen. Wir untersuchen die Robustheit verschiedener komprimierter CNN, indem wir sehr kleine, für das menschliche Auge nicht wahrnehmbare Störungen in die Bilder einspeisen. Wir verbessern die Robustheit gegen Angreifer, indem wir defensive Kompressionsmethoden vorschlagen, die lernbare Pruning-Masken und Bitweiten in die Trainingsverfahren gegen Angreifer integrieren. Das Ziel dieser Arbeit ist es, Kompressionsverfahren zu entwickeln, die den Kompromiss zwischen aufgabenspezifischer Genauigkeit, Robustheit gegen Angreifer und Ausführungsmetriken bei weniger rechnerischem Aufwand verbessern.

Abstract

Convolutional Neural Networks (CNNs) achieve state-of-the-art performance on computer vision applications such as image classification, semantic segmentation and object detection. The improved performance of these networks comes at the cost of huge model size and compute complexity. Efficient processing of these networks is critical to allow its deployment in compute and memory-constrained target hardware (HW) platforms. Using compression techniques like pruning and quantization, we can reduce the compute and memory demand of CNN inference. We derive execution estimates using HW-models and scheduling schemes to provide feedback to search algorithms which determine efficient compression configurations. We specifically model a spatial array based CNN accelerator and leverage reinforcement learning (RL)-based exploration to derive HW-aware pruning configurations. Existing search algorithms which derive compression configurations require floating point pretrained model and high computational effort due to iterative fine-tuning phase. We reduce the computational effort of the compression phase by proposing in-train optimization techniques which determine efficient pruning or quantization strategies during the gradient-based learning process. We finally investigate the adversarial robustness of different compressed CNNs by injecting very small perturbations to input images that are imperceptible to the human eye. We improve the adversarial robustness by proposing defensive compression methods which integrate learnable pruning masks and bit-widths in the adversarial training schemes. The goal of this work is to investigate compression techniques which improve the trade-off between task specific accuracy, adversarial robustness and execution metrics with less computational effort.

1 Introduction

1.1 Motivation

Technological advances in the fields of automation and robotics are changing our world in a significant way. Autonomous Driving (AD) is revolutionizing mobility by offering more flexibility and comfort. One of the main challenges of autonomous driving is its dependency on a reliable perception and understanding of the environmental surroundings. Perception is acquired using dedicated sensors such as cameras, LiDAR and radar. These sensors produce raw data in the form of signals, which have to be processed to extract meaningful information, crucial for the planning of the driving trajectory. The process of classifying and detecting objects using sensor data is therefore a crucial building block in the perception stack of AD functionality.

Neural network based algorithms have gained growing interest in the last years in several perception based applications. They achieve impressive results for state-of-the-art Computer Vision (CV) applications like image classification [1], semantic segmentation [2] and object detection [3]. The high memory demand, energy consumption and latency of high-performance CNNs portray the main challenges for the deployment of such neural networks on resource-constrained embedded hardware, such as autonomously driving cars or mobile devices, for real-world applications. Therefore, neural network compression and acceleration represents a key aspect to enable CNN in applications with strict latency, memory and energy requirements.

Many approaches have been developed in the past few years for reducing the model size and computational complexity of CNNs. Some effective and popular techniques include pruning [4, 5, 6], quantization [7, 8] or knowledge distillation [9]. Typical CNN hardware accelerators include general purpose and data center Graphics Processing Units (GPUs) [10], which are the most popular choice due to their high computing capabilities and support to various machine learning frameworks. Although model compression reduces the total compute complexity, the custom CNN configuration caused by compression hinders the efficient acceleration on general-purpose processors like GPUs. In these cases Application-Specific Integrated Circuits (ASICs) [11] and field programmable gate arrays (FPGAs) [12] provide custom solutions to leverage HW benefits in the form of latency and energy consumption for these compression techniques. Most of the compression works in literature highlight the benefits in terms of proxy metrics such as Operations (Ops) and parameters (Params) , which are not guaranteed to produce benefits on HW metrics such as latency for the target task. In-order to obtain *HW-aware compressed CNNs*, we directly incorporate HW metrics instead of proxies in the target objective function of the optimization scheme.

Given a CNN model, the search space to realize an efficient compression configuration grows exponentially with increasing number of layers. To obtain HW-friendly layer-wise compression strategies, post-training based compression methods use search techniques, such as, RL [5, 13] or Evolutionary Search (ES) [14, 15]. They require a pretrained baseline model and additionally

demand costly post-train *GPU hours*. The search space for compression techniques, such as quantization alone consists of $|q|^{2L}$ solutions, where q is the set of possible quantization levels and L is the number of layers. Quantizing the operands in certain layers leads to larger drops in accuracy than others, and different accuracy drops can take place at different quantization levels for the same layer. Moreover, quantization strategies change the mapping and scheduling space of the accelerator. For example, a quantization strategy might make new schedules possible, which leads to sudden drops in latency and energy, leading to better HW implementation. In-order to realize a *Fast Compression* with lower GPU hours, we directly determine the efficient compression strategy along with the learnable parameters *during* the training process and produce dominant solutions in terms of prediction accuracy, target HW metrics.

With the continuous progress in the development of AD, the associated features come increasingly into the spotlight. Self-evidently, the safety critical environment for AD maintains zero-tolerance for potential threats for the CNN-based perception algorithms. With the advent of adversarial attacks, Szegedy et al. [16] unveiled the vulnerability of CNNs against malicious perturbations added to inputs, resulting to fool neural networks. Defense methods [17, 18] incorporate the attacked samples into the training process, making the deployment secure against the adversarial perturbations. Most compression schemes are only evaluated on the test dataset without adversarial perturbations. In-order to obtain *HW-aware robust CNNs*, we conduct detailed investigations to analyze robustness of compressed CNNs and furthermore formulate a defensive compression scheme which is resistant against adversarial attacks.

1.2 Objectives

We aim to achieve *accurate and robust CNNs* which produce *low HW execution metrics* by using a *fast optimization* scheme. Specifically, we optimize our CNN f with model parameters θ to maximize prediction accuracy Acc for a validation dataset $\mathcal{D}_{\text{val}}$ consisting of inputs x and corresponding labels y, given in Eq. 1.1.

$$\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{val}}}\left[f(x,\theta,\alpha)\right] \longmapsto Acc \tag{1.1}$$

Our first objective is to realize *HW-aware CNN models* by exploring compression opportunities in the form of α and directly minimize the specific execution metrics (given as $HW_{metrics}$) like latency, throughput, memory access and energy consumption as highlighted in Eq. 1.2. We would like to determine the efficient compression configuration α^* indicating the pruning and quantization strategy for different layers of the CNN model to obtain Pareto-dominant solutions with respect to the prediction accuracy and the underlying HW platform.

$$\begin{aligned} \alpha^* &= \arg\min_{\alpha} \;\; HW_{metrics}(\texttt{Target}_{\texttt{HW}}, f(x,\theta,\alpha)) \\ \theta^* &= \arg\max_{\theta} \;\; Acc \;\; \text{ with } \; Acc = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{val}}}\left[f(x,\theta,\alpha^*)\right] \end{aligned} \tag{1.2}$$

Target platform dependencies and scheduling schemes imposed by the compiler must be taken into account while selecting the CNN compression configuration α to obtain Pareto-dominant solutions. For e.g. GPUs can obtain benefits in execution metrics using regular channel pruning

configuration. Bit serial accelerators can leverage execution benefits using mixed precision quantization strategies.

Our second objective is to procure an efficient compression strategy with reduced computational effort. Specifically, we aim to achieve a *HW-friendly CNN configuration* $f(x, \theta, \alpha)$ with reduced number of GPU hours. We formulate a search problem in Eq. 1.3 to obtain an efficient CNN compression configuration α. Specifically, we split the dataset $\mathcal{D}$ into training ($\mathcal{D}_{\text{train}}$) and validation ($\mathcal{D}_{\text{val}}$) sets. We aim to reduce HW metrics and maximize the prediction accuracy on validation set by searching efficient pruning and/or quantization strategies α^*. However, the post-train search problem involves expensive inner loop convergence (determining θ^* in Eq. 1.3 for every compression scheme α in the search space).

$$\begin{aligned}
\alpha^* &= \arg\min_{\alpha} \;\; HW_{metrics}(\texttt{Target}_{\texttt{HW}}, f(x, \theta, \alpha)) \\
\alpha^* &= \arg\max_{\alpha} \;\; ValAcc \;\; \text{with} \;\; ValAcc = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{val}}}\left[f(x, \theta, \alpha)\right] \\
\theta^* &= \arg\max_{\theta} \;\; TrainAcc \;\; \text{with} \;\; TrainAcc = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{train}}}\left[f(x, \theta, \alpha^*)\right]
\end{aligned} \tag{1.3}$$

To reduce the computational effort, measured in GPU hours, required for optimization, we need to determine $\theta^*(\alpha)$ which can provide sufficient judgement about the compression strategy α in the form of $ValAcc$. However, the search space for compression through pruning and quantization grows exponentially with deeper CNN models. This would demand an increased number of *GPU hours* due to iterative fine-tuning and extensive model exploration. Therefore, we reduce the GPU hours by integrating the compression opportunities and task specific training scheme into a single optimization problem.

Our third objective is to further improve the adversarial robustness of the compressed CNNs. We formulate defensive compression schemes by learning efficient compression configurations during the adversarial training procedure. We expose the compressed CNN f to adversarial images X_{adv}, resulting in the adversarial accuracy Acc_{adv}. This represents the *adversarial robustness* of the CNN against a specified threat model τ, see Eq. 1.4.

$$\begin{aligned}
f(X_{adv}, \theta, \alpha) &\longmapsto \text{Acc}_{adv} \;\; s.t. \;\; \tau \longmapsto X_{adv}, f \\
\theta^*, \alpha^* &= \arg\max_{\theta,\alpha} \;\; \text{Acc}_{adv} \;\; \text{with} \;\; \text{Acc}_{adv} = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{val}}}\left[f(X_{adv}, \theta, \alpha)\right] \\
\alpha^* &= \arg\min_{\alpha} \;\; HW_{metrics} \;\; \text{with} \;\; HW_{metrics} = Target_HW(f(X_{adv}, \theta^*(\alpha), \alpha))
\end{aligned} \tag{1.4}$$

Determining the influence between the compression scheme α and HW metrics and further formulating a defensive compression scheme to improve Acc_{adv} aids us to realize *HW-aware Robust CNNs*.

1.3 Contributions

The thesis realizes HW-aware robust CNNs using *HW-aware compression*, *Fast Compression* and *Robust Compression* techniques as highlighted in Fig. 1.1. In-order to obtain benefits in HW-execution metrics, the compression scheme must account for the correlation between the workload

dimensions and its corresponding schedule on target HW-platform. In Fig. 1.1, we observe that the convolutional layer consists of input, output feature maps and weights. The first contribution of the thesis, i.e. *HW-aware compression* determines efficient compression configurations for each layer of the CNN to satisfy target HW constraints. As an example for compression method, Fig. 1.1 determines efficient quantization configuration for input and weights to improve the compute density in the underlying HW-accelerator. The second contribution of the thesis proposes *Fast Compression* methods, which determine the compression configurations for various workload present in the CNN model. As shown in the Fig. 1.1, pertubtating the input image with adversarial noise distorts the prediction quality of the CNN model. The third contribution of the thesis i.e. *Robust Compression*, investigates unified training schemes to achieve adversarial robust compression of CNN models.

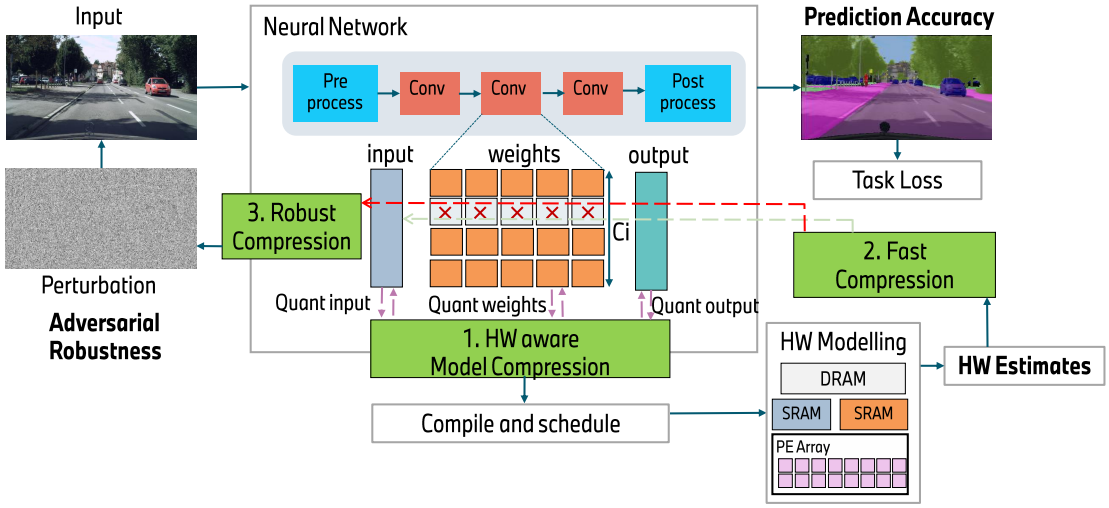


Figure 1.1: Overview of this thesis document summarizing the three contributions and various objectives. Input image is fed into the neural network with several layers to generate predictions. The three contributions of the thesis enable an end-end HW-aware robust compression to achieve an efficient inference on target accelerator.

HW aware Neural Network Compression: Most CNN compression approaches in literature [19, 4, 20, 21] such as pruning do not consider target HW benefits such as energy consumption and latency as optimization goals, instead focus on proxy optimization targets such as Ops and Params. Further pruning approaches such as AMC [5], ChamNet [22] require target HW accelerator during the compression process. Limited works in literature [23] realize an end-end automated channel pruning pipeline for complex CNN applications such as semantic segmentation and 3D object detection. Therefore,

- We realize channel pruned configurations of an over-parameterized CNN by imposing constraints on execution metrics such as energy consumption and latency for spatial array based CNN accelerator. We obtain execution estimates by realizing HW models providing us the flexibility to explore different components such as compute array sizes,

memory hierarchies and dataflow choices. We further reduce the time required to obtain the execution estimates by determining an efficient schedule using analytical search approaches.

- We investigate the latency-accuracy trade-offs on channel pruned configurations for PointPillars [24], a fast LiDAR-based 3D object detection model. We conduct detailed experimental evaluation using RL-agent based channel pruning using HW metrics from a GPU.

Reduced GPU Hours for Model Compression: Most pruning and quantization based CNN compression pipelines in literature [21, 5, 13] follow a three stage pipeline to search for an efficient compression configuration. The increasing number of layers due to the growing task complexity demands more GPU hours to search for a pruning and/or quantization strategy. Therefore,

- We reduce the number of GPU hours for CNN pruning using multi-task RL agent, identifying layer's redundant features and adequate fine-tuning time concurrently.
- We propose an *in-train pruning* approach, which identifies redundant weights by minimizing a hardware-aware auxiliary loss when updating the network's connections. Our approach realizes a pruned model for the same number of epochs compared to the baseline training, requiring no additional overhead for searching pruning configuration or fine-tuning.
- We introduce a novel training scheme which jointly learns the model parameters and the number of unique values required to represent weights and activations for all the layers, thereby identifying optimal word length assignments. We further incorporate HW-awareness by appending a differentiable auxiliary HW-loss objective using Gaussian process regression.

Improved Adversarial Robustness: Extensive amount of work in literature such as [17, 18] propose defensive training schemes to produce robust neural networks countering adversarial attacks. Very few works investigate the influence between compression and adversarial robustness. Therefore,

- We conduct a detailed study on adversarial robustness of compressed CNNs variants. We analyze adversarial robustness for various pruned, quantized and knowledge distilled CNN configurations using four white box and two black box attacks.
- We obtain robust pruned and mixed precision configurations by augmenting the trainable pruning masks and bit-widths during the adversarial training.

1.4 Organization of Thesis

This thesis is organized into five chapters. This Chapter (Chapter.1) deals with the motivation behind the thesis and gives a very high-level overview of the problem statement and our contributions.

In Chapter. 2, we introduce various building blocks of CNNs required to perform training and inference. We discuss three CV tasks, namely image classification, semantic segmentation and object detection. We introduce popular CNN architectures, datasets and metrics associated with the three tasks. Further, we cover the preliminaries of compression techniques such as quantization and pruning. We discuss various choices for CNN inference accelerator and additionally provide an introduction to adversarial robustness.

In Chapter. 3, we realize HW-aware CNNs by incorporating HW-metrics directly during the compression process. We first discuss various components in the traditional three stage compression pipeline. We further discuss pruning schemes in literature and point out their limitations. In-order to obtain HW-estimates, we formulate a HW-model and search for efficient scheduling scheme for individual layers of a CNN model. We alternatively derive the HW-estimates by directly executing a CNN model on an inference HW-platform. We provide these estimates as a feedback in the three stage compression pipeline to highlight the benefits of HW-aware CNNs. The content of this chapter is based on the following publications:

- M. Vemparala, N. Fasfous, A. Frickenstein, E. Valpreda, M. Camalleri, Q. Zhao, C. Unger, NS. Nagaraja, M. Martina and W. Stechele, "HW-Flow: A Multi-Abstraction Level HW-CNN Codesign Pruning Methodology", Leibniz Transaction of Embedded Systems (LITES), 2022 [25].

- M. Vemparala, A. Singh, A. Mzid, N. Fasfous, A. Frickenstein, F.Mirus, HJ.Voegel, NS. Nagaraja, W. Stechele, "Pruning CNNs for LiDAR-based Perception in Resource Constrained Environments", In IEEE Intelligent Vehicles Symposium Workshops, 3D-Deep Learning for Automated Driving (3D-DLAD) 2021 [26].

In Chapter. 4, we explore various fast compression methods to reduce the computational effort for the optimization process. Firstly, we reduce the amount of fine-tuning time required to evaluate a compressed configuration during the RL-based search process. We further propose to jointly train the weights and determine the compression strategy by devising an *in-train* pruning and quantization scheme. We demonstrate the effectiveness of our in-train compression method by highlighting the HW-benefits and reduction in *GPU-hours*. The content of this chapter is based on the following publications:

- M. Vemparala, N. Fasfous, A. Frickenstein, MA. Moraly, A. Jamal, L. Frickenstein, C. Unger, NS. Nagaraja, and W. Stechele, "L2PF - Learning to Prune Faster. In International Conference on Computer Vision Image Processing (CVIP), 2020 [27]".

- M. Vemparala, N. Fasfous, A. Frickenstein, S.Sarkar, Q.Zhao, S.Kuhn, L.Frickenstein, A.Singh, C.Unger, NS.Nagaraja, C.Wressnegger, W.Stechele, "Adversarial robust model compression using in-train pruning", In IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2021 [28].

- M. Vemparala, N. Fasfous, L.Frickenstein, A. Frickenstein, A.Singh, D.Salihu, C.Unger, NS.Nagaraja, W.Stechele, "Hardware-aware mixed-precision neural networks using in-train quantization". In British Machine Vision Conference (BMVC), 2021 [29].

In Chapter. 5, we study the impact of adversarial attacks on state of the art compression techniques. We thereby formulate defensive compression schemes to derive HW-benefits and resilience towards adversarial attacks. The content of this chapter is based on the following publications:

- M. Vemparala, A. Frickenstein, N. Fasfous, L.Frickenstein, Q.Zhao, S.Kuhn, D.Ehrhardt, Y.Wu, and W. Stechele, C.Unger, NS.Nagaraja, and W.Stechele, "Breakingbed - breaking binary and efficient deep neural networks by adversarial attacks", In Intelligent Systems Conference (IntelliSys), 2021 [30].
- M. Vemparala, N. Fasfous, A. Frickenstein, S.Sarkar, Q.Zhao, S.Kuhn, L.Frickenstein, A.Singh, C.Unger, NS.Nagaraja, C.Wressnegger and W.Stechele, "Adversarial robust model compression using in-train pruning", In IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2021 [28].
- M. Vemparala, N. Fasfous, L.Frickenstein, A. Frickenstein, A.Singh, D.Salihu, C.Unger, NS.Nagaraja and W.Stechele, "Hardware-aware mixed-precision neural networks using in-train quantization", In British Machine Vision Conference (BMVC), 2021 [29].

With this thesis, we significantly *improve the HW-efficiency* of CNN with respect to target platform, *reduce the computational effort* (in GPU-hours) for the optimization process and *defend against adversarial attacks*, facilitating the broad adoption of these models in many practical problems.

2 Background

2.1 Convolutional Neural Networks

Neural networks that have more than three layers, that is, more than one hidden layer are referred to as Deep Neural Networks (DNNs). CNNs constitute a special class of DNNs with multiple windowed and weight-shared layers called convolutional layers. Successive layers detect various features in the input image at different scales. The first layers are usually responsible of recognizing simple shapes, edges and patterns, while complex features can be detected at the deeper stages of the network. CNNs are well-suited for generating predictions based on multi-dimensional, localized input features, e.g. image processing applications. In this section, we discuss in detail the basic building blocks of CNN required during training and inference.

2.1.1 Convolutional layer

The convolution of an input activation A^{l-1} with the convolution kernel W^l produces an output feature map A^l, where each pixel of the feature map A^l can be computed as shown in Eq. 2.1.

$$A^l[c_o][h_o][w_o] = \overbrace{\sum_{c_i}^{C_i}}^{\text{Inp.Ch}} \overbrace{\sum_{k_w}^{K_w} \sum_{k_h}^{K_h}}^{\text{Kernel.dim}} a^{l-1}_{c_i, w_o \cdot s + k_w, h_o \cdot s + k_h} \cdot w^l_{c_o, c_i, k_w, k_h}, \text{ where } A^l \in \mathbb{R}^{C_o \times H_o \times W_o} \tag{2.1}$$

The Input Feature Maps (Ifmaps) denoted by A^{l-1} are composed of multiple channels C_i and spatial dimensions W_i, H_i. To compute the convolution operation, the kernel of dimensions $K_w \times K_h$ slides across the input 2-D map with stride size s. A dot-product is performed between the kernel pixels $w^l \in W^l$ and a sub-set of pixels $a^{l-1} \in A^{l-1}$ from the input volume. The dot-product accumulates the values across all input channels resulting in an output pixel. The convolution operation is the repetition of the aforementioned dot-product operation for the entire Ifmap with C_o filters, generating Output Feature Map (Ofmap) $A^l \in \mathbb{R}^{W_o \times H_o \times C_o}$ as demonstrated in Fig. 2.1.

Zero padding can be applied to the Ifmaps to increase the horizontal and vertical size of the feature map, by adding zeros around the original volume. The horizontal and vertical dimensions of the output volume can be computed using the formulas detailed in Eq. 2.2. Padding is leveraged to Ifmap to regulate the dimensions of the Ofmap in the subsequent layers of the CNN. Stride and Padding are used to control the spatial output dimensions of the Ofmap that are determined by Eq. 2.2.

$$W_o = \lfloor \frac{W_i - K_w + 2 \cdot P_w}{S_w} \rfloor + 1 \quad ; \quad H_o = \lfloor \frac{H_i - K_h + 2 \cdot P_h}{S_h} \rfloor + 1 \tag{2.2}$$

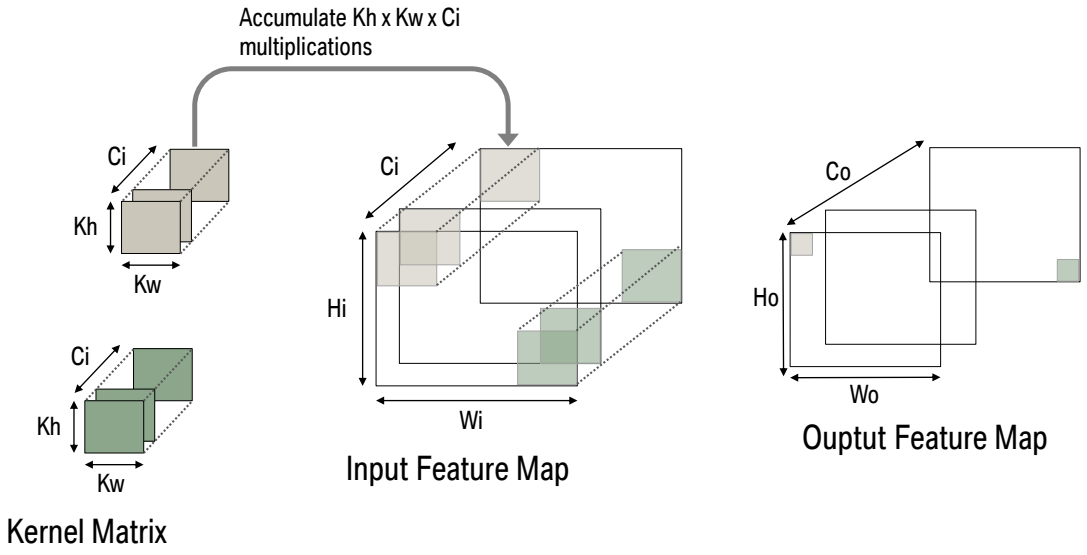


Figure 2.1: Sliding window based convolution using kernel matrix $W^l \in K_w \times K_h \times C_i \times C_o$ and Ifmap $A^{l-1} \in H_i \times W_i \times C_i$

The number of Multiply–Accumulates (MACs) in the convolution operation is given by Eq. 2.3.

$$\#MACs = C_o \times C_i \times K_w \times K_h \times H_o \times W_o \tag{2.3}$$

The convolution algorithm shown in Eq. 2.1 and Fig. 2.1 offers several data reuse opportunities. Due to the weight sharing property in the convolutional layers, the same Ifmap is leveraged to produce C_o filters. Therefore, the **reuse factor of Ifmap** in a convolutional layer is given by C_o. Each kernel window $K_h \times K_w$ is iterated $W_o \times H_o$ to generate the entire Ofmap. Therefore, the **reuse factor of kernel matrix** is given by $W_o \times H_o$.

Different forms of convolution: In order to obtain a high training and testing accuracy on various datasets, CNNs with multiple kernels per layer lead to multiple filters. Each kernel filter convolves on all the Ifmaps obtained on the previous layer, resulting in lots of mulitplications, some of which may be redundant. Training deeper models would get challenging due to difficulties in fitting these models on a single GPU. Therefore, many works such as Alexnet [31], MobileNetv2 [32], EfficientNet [33] use a slightly different variant of Eq. 2.1 by reducing the number of parameters and operations, improving the throughput of training and inference.

Grouped Convolution: In the vanilla form of convolution described in Eq. 2.1, each Ofmap is generated using $K_h \times K_w \times C_i$ MAC operations. Grouped convolutions divides the input channels C_i into g groups and demands $K_h \times K_w \times C_i/g$ MAC operations to generate each Ofmap as shown in Eq. 2.4 and Fig. 2.2.

$$A^l[c_o][h_o][w_o] = \sum_{c_i}^{\overbrace{C_i/g}^{\text{Inp.Ch split into g groups}}} \sum_{k_w}^{K_w} \sum_{k_h}^{K_h} a^{l-1}_{c_i, w_o \cdot s + k_w, h_o \cdot s + k_h} \cdot w^l_{c_o, c_i, k_w, k_h} \tag{2.4}$$

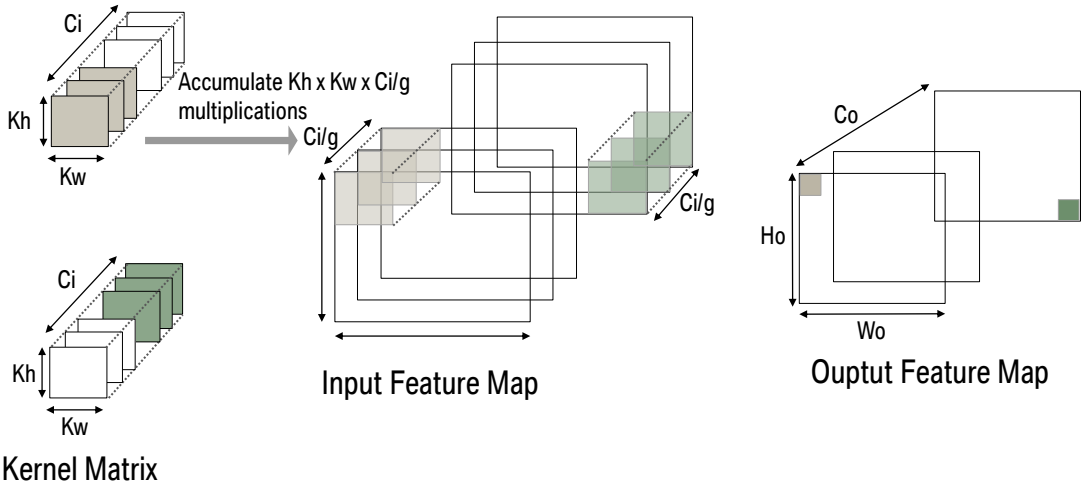


Figure 2.2: Computing the grouped convolution using kernel matrix $W^l \in K_w \times K_h \times C_i \times C_o$ and Ifmap $A^{l-1} \in H_i \times W_i \times C_i$ with g groups.

Grouped convolutions were first used in Alexnet [31] to fit the training on less powerful GPUs with smaller DRAM available in the past. Using the grouped convolutions, we can increase the width of the CNN seamlessly as the resulting Ofmaps is only related to subset of pixels in Ifmap. The number of MAC operations for grouped convolutions is given by Eq. 2.5.

$$\#MACs = C_o \times C_i/g \times K_w \times K_h \times H_o \times W_o \tag{2.5}$$

The **reuse factor of Ifmap** in a grouped convolutional layer is given by C_o/g. Each kernel window $K_h \times K_w$ is iterated $W_o \times H_o$ to generate the entire Ofmap. Therefore, the **reuse factor of kernel matrix** is given by $W_o \times H_o$.

Depthwise separable convolution can be understood as the extreme form of grouped convolution with $g = C_i = C_o$. The name depthwise separable indicates that the pixels in Ofmap directly depends on the corresponding spatial dimension but not the input channel depth as shown in Fig. 2.3.

$$A^l[c_o][h_o][w_o] = \sum_{k_w}^{K_w} \sum_{k_h}^{K_h} a^{l-1}_{c_i, w_o \cdot s + k_w, h_o \cdot s + k_h} \cdot w^l_{c_o, c_i, k_w, k_h} \tag{2.6}$$

The number of MAC operations for depthwise convolutions described in Eq. 2.6 is given by Eq. 2.7.

$$\#MACs = C_o \times K_w \times K_h \times H_o \times W_o \tag{2.7}$$

There is **no reuse factor for Ifmap** in a depthwise seperable convolutional layer as each Ofmap is related to its corresponding Ifmap. Each kernel window $K_h \times K_w$ is iterated $W_o \times H_o$ to generate the entire Ofmap. Therefore, the **reuse factor of kernel matrix** is given by $W_o \times H_o$.

Transpose Convolution: Different forms of convolutions discussed above reduce/maintain the spatial dimensions of the Ofmap relative to its Ifmap. However, in applications such as semantic segmentation [34], certain convolutional layers are responsible to upsample the Ifmap.

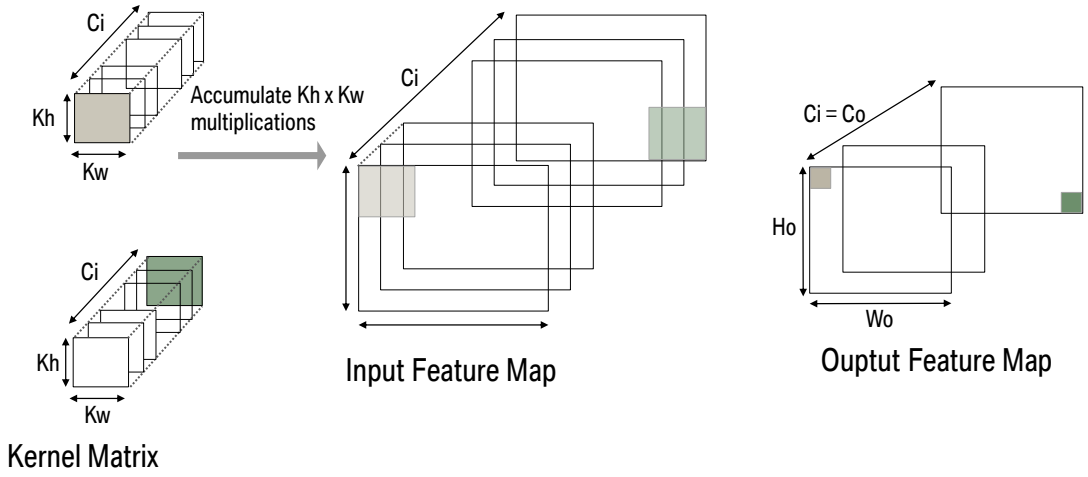


Figure 2.3: Depthwise separable convolution of kernel matrix and Ifmap.

Consider a 4×4 Ifmap. Inorder to perform transpose convolution with a kernel size of 3×3 and stride 2, we pad zeros to the Ifmap as shown in Fig. 2.4. It is equivalent to performing a vanilla convolution described in Eq. 2.1 with a 3×3 kernel over a 7×7 Ifmap using unit stride.

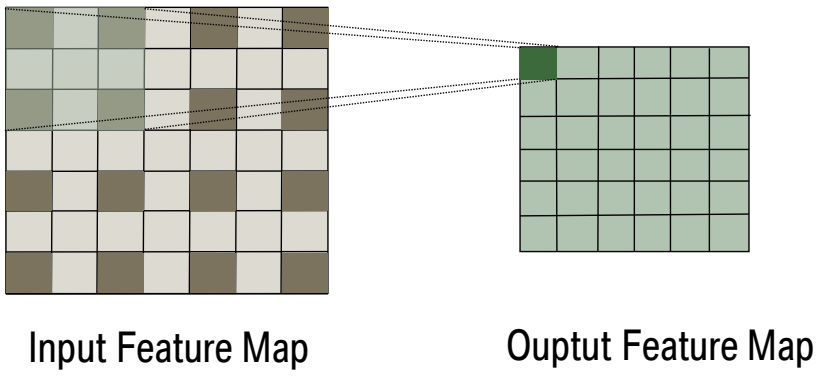


Figure 2.4: Example of transpose convolution with stride = 2 for kernel size 3×3 on input feature map dimension 4×4.

The number of MAC operations for transpose convolution is given by Eq. 2.8 which is similar to Eq. 2.1. It is important to note that the majority of operations in the transpose convolution has zero multiplications which can be skipped using efficient software implementations.

$$\#MACs = C_o \times C_i \times K_w \times K_h \times H_o \times W_o \tag{2.8}$$

Similar to the vanilla form of convolution, the **reuse factor of Ifmap** in a transpose convolutional layer is given by C_o. Each kernel window $K_h \times K_w$ is iterated $W_o \times H_o$ to generate the entire Ofmap. Therefore, the **reuse factor of kernel matrix** is given by $W_o \times H_o$.

Dilated Convolution: Dilated convolution also known atrous convolution is commonly employed in applications such as semantic segmentation [35]. The motivation behind this dilated convolution is to vary the projected area of kernel matrix on the Ifmaps and capture information at multiple scales. Compared to the vanilla form of convolution, there is an additional parameter which controls the number of zero elements inserted in the kernel to *inflate* it. Let us refer this by D_w and D_h. When $D_w > 1$, $D_w - 1$ number of zero elements are inserted in the spatial dimension of the weight kernels to extend it. For a weight kernel with dimension N_{kx} and N_{ky} the extended dimension after dilation applied is given by Eq. 2.9.

$$\hat{K_w} = K_w + (K_w - 1) \cdot (D_w - 1) \qquad \hat{K_h} = K_h + (K_h - 1) \cdot (D_h - 1) \tag{2.9}$$

Consider a 7×7 Ifmap. Inorder to perform dilated convolution with a kernel size of 3×3 and dilation rate of 2, we pad zeros to the kernel matrix as shown in Fig. 2.5. It is equivalent to performing a vanilla convolution described in Eq. 2.1 with a 5×5 kernel over a 7×7 Ifmap using unit stride.

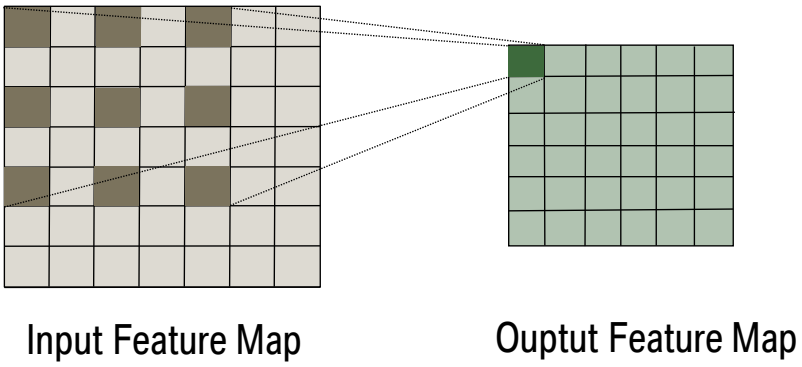


Figure 2.5: Example of dilated convolution with dilation rate = 2 for kernel size 3×3 on input feature map dimension 7×7.

The number of MAC operations for dilated convolution is given by Eq. 2.10. It is important to note that the majority of operations in the dilated convolution has zero multiplications which can be skipped using efficient software implementations.

$$\#MACs = C_o \times C_i \times \hat{K_w} \times \hat{K_w} \times H_o \times W_o \tag{2.10}$$

Similar to the vanilla form of convolution, the **reuse factor of Ifmap** in a dilated convolutional layer is given by C_o. Each kernel window $K_h \times K_w$ is iterated $W_o \times H_o$ to generate the entire Ofmap. Therefore, the **reuse factor of kernel matrix** is given by $W_o \times H_o$.

2.1.2 Fully Connected layer

Fully Connected (FC) refers to a layers where each pixel of Ifmap is connected to all elements in its previous Ofmap as well as the next layer. FC layers are simplified by considering as a special case of the convolution operation described in Eq. 2.1 by setting $W_i = K_w$, $H_i = K_h$, $W_o = 1$ and $H_o = 1$. These layers restrict weight reuse opportunities and demand high memory bandwidth during the inference. Nonetheless, batch optimization can be leveraged to reuse the same filters on different input images to reduce the energy consumption and data transfers across various memory levels. The last layers in the CNN leverage the FC layers to obtain the classification output. The number of MAC operations for FC layer is given by Eq. 2.11.

$$\#MACs = C_o \times C_i \tag{2.11}$$

2.1.3 Pooling layer

Pooling layers reduce the dimensionality of feature maps by performing a sub-sampling of the input volume. This sub-sampling is performed by sliding a 2D feature map across the input and deriving a single pixel from it, like the maximum or the average value in the window. The dimenisonality of Ofmap in the pooling volume can be evaluated with the same formulas used for a convolutional layer as discussed in Eq. 2.2. Pooling is applied to each channel separately and the commonly used configuration is $S_w = K_w$ (here K_w denotes the pool receptive field) such that no overlapping is present. *Max Pooling* extracts the maximum value from its receptive field, whereas, *Average Pooling* computes an average of the values in its receptive field as shown in the Fig. 2.6.

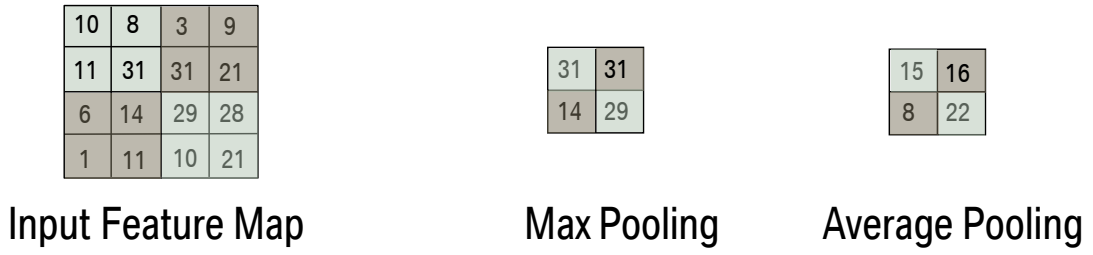


Figure 2.6: Depiction of Average Pooling and Max Pooling for a 4×4 Ifmap and 2×2 pooling window with stride 2.

2.1.4 Non linear activation

Convolutional or FC layers are usually followed by activation functions to introduce non-linearity into the networks. Non-linear functions, which are commonly used, include **Sigmoid**, **Hyperbolic Tangent** or **Tanh** and **Rectified Linear Unit (ReLU)**. Sigmoid activation is not zero-centric. Besides, sigmoid and tanh functions suffer from vanishing gradient problem during back-propagation.

State-of-the-art neural networks tend to use ReLU [36] since it is simple and facilitates faster training. Different non-linear activation functions have been illustrated in Fig. 2.7.

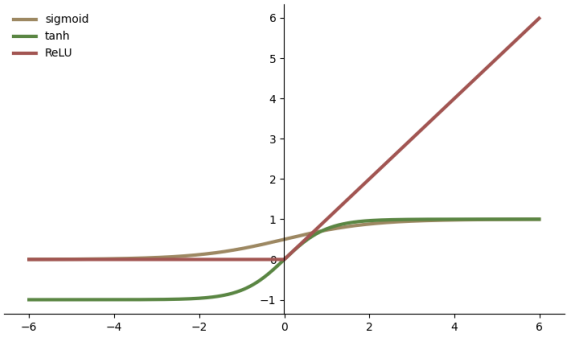


Figure 2.7: Non linear activation functions used in CNNs. X-axis represents the input of the non-linear activation function and Y-axis represents the output.

2.1.5 Batch Normalization

The batch normalization layer normalizes the distribution at the input of each layer. This produces guaranteed speed-up and accuracy improvement during training. Each layer in a CNN consists of several channels. During batch normalization, each channel is normalized and then transformed linearly using parameters β and γ, as shown in Eq. 2.12. The parameters (β, γ) are learned during training. $\mu_{l,c}$ and $\sigma_{l,c}$ denote the mean and variance of layer l and channel c for the current mini-batch during the training. Since normalization is performed over mini-batches, this method is called batch normalization. During inference of CNN, instead of deriving the l, c from current batch statistics, moving mean and moving variance acquired across the training are used.

$$y_{l,c} = \frac{x_{l,c} - \mu_{l,c}}{\sqrt{\sigma_{l,c}^2 + \epsilon}}\gamma + \beta \tag{2.12}$$

2.1.6 Training Neural Networks

Loss Function: The loss function is a differentiable metric which computes the error between the outputs predicted by the CNN and the ground truth labels g. The choice of the loss function is an important step in the model design because it reflects the quality of the predictions. The formulation depends on the application task. For instance, it is typical to use the cross-entropy loss for classification tasks while it is common to use the Mean Squared Error or the L1-Error for regression tasks. Several applications also append mean squared sum of all the weights to avoid over-fitting on the training data, also known as regularization loss.

Optimizer: The learning process (also called training) for CNN is based on updating the trainable weights w. This process happens in two stages:

- Forward Pass: The CNN predicts the outputs for a given input sample or batch θ using the model parameters $w^{(t)}$ at each training step t. Given the model predictions, the loss value $L(\theta, w^{(t)}, g)$ can be evaluated using the ground truth labels as reference.
- Backward Pass: The computed loss is back propagated iteratively through each layer of the CNN in order to update the parameters $w^{(t)}$. For this purpose, the gradient of the loss function $\Delta_w L$ is calculated with respect to different trainable parameters. The scaling factor η is called learning rate. The weights are therefore updated according to the following Eq. 2.13.

$$w^{(t+1)} = w^{(t)} - \eta \overbrace{\Delta_w L(\theta, w^{(t)}, g))}^{\text{Backpropagation}} \tag{2.13}$$

  This equation is repeated until convergence, i.e. we update the trainable parameters, for each training example, until we reach a local minimum. For faster convergence, it is common to use the momentum optimizer [37]. For adaptive learning, one of the most popular and fast optimizers is Adam [38].

2.2 Neural Networks in Computer Vision

CNNs have produced better predictions than humans on computer vision applications such as image classification [39], semantic segmentation [35] and object detection [24] using supervised ground truth labels.

2.2.1 Image Classification

Out of O possible classes, the input image is predicted based on the output $\tilde{Y} \in \mathbb{R}^O$. It is typical to translate the classification problem into predicting the probability of each possible class given an input image, so that the output layer produces a vector with a fixed dimension of O. Several CNN topologies were proposed in the last decade to solve the image classification problem. For instance, AlexNet was introduced by Krizhevsky et al. [31] as the first CNN topology for classifying the ImageNet dataset. Other examples which followed in the next years include VGG-16 [40], Inception-Net [41], ResNet [39] and EfficientNet [33].

AlexNet [31] primarily used for image classification tasks consists of 5 convolutional layers, 3 FC layers has 60 million parameters requiring approximately 1.1 billion multiply-accumulate (MAC) operations for one forward pass. AlexNet uses LRN operation in first and second convolutional layers. It further uses split convolutional mode in layers 2, 3, 4 to reduce the number of computations and perform the convolutions parallely. The most popular variant of VGG from Visual Geometry group is the VGG-16 [40], which has a depth of 16 layers, and a very regular structure, consisting exclusively of 3×3 convolution and 2×2 max-pooling layers. The layer dimensions are gradually reduced from 224×224 pixels to 7×7 pixels till the last convolutional layer, while the number of output channels is simultaneously increased from 3 to 4096. However, VGG-16 contains more than 140 million weights and one forward pass

requires nearly 16 billion MAC operations. GoogleNet [41] has 7 million parameters across 57 convolutional layers and only one fully connected layer. GoogleNet has nine inception modules. Each inception module consists of four branches with 1×1, 3×3, 5×5 convolutions and down-sampling layer. Two auxiliary loss layers inject loss from the intermediate layers and prevent gradient vanishing. At inference time, the auxiliary layers can be removed.

Residual networks [39] are stacked convolutional layers with skip (shortcut) connections to jump over specific layers, organized in blocks called residual blocks. In conventional (plain) architectures, convolutional layers are stacked directly. In plain architecture, stacking up to 30 layers is correlated to an increase in model accuracy, however, going deeper will cause accuracy to saturate and even to starts degrading. Therefore, residual networks leverage identity mapping across multiple layers such that the gradients back-propagate easily through them, allowing faster learning. Each block consists of two convolutional layers, batch normalization, and a nonlinear function. The end of the block is element-wise summation followed by batch normalization. The element-wise summation is performed between the input block and output of batch normalization of second convolutional layer. In recent years, we observe efficient CNN models using depth-wise convolutions [32] and compound model scaling techniques [33] to improve accuracy of the models with increased depth.

CIFAR-10 [42] represents one of the most common datasets for image classification. Within the CIFAR-10 dataset, 10 classes are represented ranging from images of airplanes to trucks, where each class is expressed by 6000 images. In total, $60,000$ color images of the size of 32×32 pixels are divided into $50,000$ training and $10,000$ test images. ImageNet is a large-scale dataset from ILSVRC challenge [1]. The training dataset contains 1000 classes and 1.2 million images. The validation dataset contains 50,000 images, 50 images per class. The classification performance is reported using Top-1 and Top-5 accuracy. Top-1 accuracy measures the proportion of correctly-labeled images. If one of the five labels with the largest probability is a correct label, then this image is considered to have a correct label for Top-5 accuracy.

2.2.2 Semantic Segmentation

Segmentation-based CNNs such as FCN [34] and DeepLab [35] predict the class of each pixel in the input image from O possible categories. The semantic maps are derived from the logits $\tilde{Y} \in \mathbb{R}^{W \times H \times O}$ with O probability values per pixel. The CNN topology for this task follows an encoder-decoder architecture. The encoder network is a feature extractor having a similar architecture as image classification CNNs and the decoder network is a set of upsampling layers which restore the original image resolution in order to predict the pixel-wise class output. FCN uses transpose convolution to upsample features whereas DeepLab uses the bilinear upsampling method.

Long et al. [34] have proposed Fully Convolutional Networks (FCN) for Semantic Segmentation. FCN processes the entire image and produces dense predictions in the form of probability maps as outputs. For training, the ground truth labels are required. The main advantage of FCN is to convert the existing classification models to perform semantic segmentation. The FC layers have to be converted to convolutional layers with 1×1 kernels. The FC layer of a CNN model completely removes the information about the features of the image and just produce one classification output at the end. Partly, the pooling layers are also responsible for down

sampling. They have the potential to remove information about small objects. However, FCN produces classification output for ever pixel. To restore the original image size from the last convolutional layer, the feature maps must be up-sampled using transpose convolutional layers. The skip architecture is used to fuse the features from pooling layers and output of transpose convolutions. As an example, FCN-8s is a CNN model for semantic segmentation using VGG-16 as the feature extractor. It consists of 16 convolutional layers, 3 upscore layers, 3 score layers and 3 fuse layers to provide pixel wise prediction for each image.

The Cityscapes [2] is a large-scale dataset with images captured by a camera mounted on the front of a car. It consists of a mixed set of video sequences recorded in street scenes from 50 different cities. The images are annotated pixel-wise so that each pixel is labeled as one of the classes. The dataset contains 5000 finely annotated images and about 20,000 coarsely annotated images. The images were captured and provided at a resolution of 2048×1024 pixels. The finely annotated data is divided into training, validation and testing sets, consisting of 2975, 500 and 1525 images. Fig. 2.8 highlights an example image and its corresponding semantic label in the dataset. Cityscapes is an important benchmark for urban semantic scene understanding.

(a) Example of a raw image.

(b) Example of ground truth labels.

Figure 2.8: Example of ground truth and raw image in Cityscapes dataset [2].

The Intersection over Union (IOU) is an essential metric which calculates the number of pixels overlapped between ground truth labels and the obtained predictions. The IOU score is evaluated for each class in the dataset separately and finally mean Intersection over Union (mIOU) is obtained by taking average over all the IOU values for individual classes.

2.2.3 Object Detection

Object detection can be defined as the process of extracting instances of objects from an input image including their classes from a predefined set of categories as well as their spatial locations and extents. The problem formulation for the object detection task is more challenging as it is a two objective problem. It simultaneously considers the classification and localization of the detected objects. This challenge is usually addressed by combining two equally important metrics for the final loss computation. We can define the goal of 2D object detection as predicting a set of 2D axis-aligned bounding boxes $B_i = \{b_k^{(i)}\}_{k=1}^{N_i}$. Each 2D bounding box b can be characterized by its center coordinates (x, y), its width w and its height h. Therefore, it can be written as $b = (x, y, w, h)$.

Two-stage RCNN-based object detection One of the earlier CNN architectures to solve the object detection problem is RCNN [43]. This approach enumerates a large set of region candidates using the selective search algorithm. Each region of interest (RoI) is cropped directly from the image and a CNN is applied individually and iteratively on each cropped RoI. The extracted features for each RoI are the input to a support vector machine which performs the classification. Usually, the initial estimation of the object boundaries using the selective search algorithm does not tightly encompass the object. This is why a bounding box regression is performed for each RoI based on the extracted features in order to refine the original estimates. Even though RCNN achieves successful results in the detection task, this iterative process of extracting features and processing each RoI individually is computationally expensive and makes the network performance extremely slow. Fast RCNN [44] introduced the idea of extracting features from the original image using a single CNN and then cropping the extracted features instead for each RoI. However, both methods still rely on computationally inefficient traditional region proposal methods such as selective search. Faster RCNN [45] introduces region proposal network to improve the latency of the CNN model. The region proposal network operates directly on the down-sampled image features and uses convolutional layers to make category-agnostic bounding box candidates which are the input of the second stage (detection stage consisting of classification of each region proposal and regression of final bounding box).

One-stage anchor-based object detection Two-stage object detectors [43, 44, 45] predict category-agnostic region proposals to classify and refine the bounding box candidates. One stage object detection approaches merge both steps by changing the region proposal process into a multi-class classification problem. This is done by introducing the concept of anchors which are a set of predefined bounding boxes describing different possible object shapes, scales, orientations and positions in the image. This is done by introducing anchors, which are a set of predefined bounding boxes describing different possible object shapes, scales, orientations and positions in the image. YOLO [46] is one of the most important work in one-stage object detectors. The main idea behind YOLO is to coarsely divide the image into an $S \times S$ grid, where each cell predicts K class probabilities, B bounding box regression attributes, and objectiveness scores. It is therefore much faster than two-stage detection methods, whereas Faster RCNN for instance is more accurate than YOLO. The coarse grid division leads to some extent to a failure in detecting small objects or to greater localization errors, in particular if a grid cell contains more than one object. This problem is the focus of the second version of YOLO called YOLOv2 [47] which uses anchor boxes with different aspect ratios for each grid cell to give the network the ability to detect more objects with less localization errors. SSD [48] introduces the idea of considering multi-scale convolutional feature maps in order to achieve a competitively fast detection speed, but also maintain a high detection accuracy even for fine-grained features and objects. It combines detections from different down-sampling levels where at each level category scores and bounding box regression attributes are generated using convolutional detection heads. The final detections are generated using Non-Max Suppression.

Center-based object detection The center-based approach is recently proposed object detection methods. It introduces a alternative way to represent objects. Anchor-based methods

use a large number of anchor boxes causing a huge imbalance between positive and negative examples. Additionally, they introduce more hyperparameters requiring an expert knowledge about the dataset in order to correctly tune the number and aspect ratios of the used anchors. Finally, they result in a larger pre- and postprocessing overhead slowing down the training and inference pipeline. CenterNet [49] represents objects by their center points in order to transform the object detection problem to a standard keypoint estimation problem. Instead of the use of an anchor grid to encode objects, the center-based approach generates the training targets in form of heatmap where the peak locations correspond to object center locations. Based on this representation, non-max suppression can be omitted and the detection accuracy, speed can be improved compared to anchor-based object detectors.

Lidar based 3D object detection Image-based detection approaches have limited 3D detection capabilities compared to LiDAR-based object detection. Point clouds are indeed challenging geometric data structures due to their irregular and sparse format. The object detection approaches described in the previous parts of this work expect a dense image-like representation $I \in \mathbb{R}^{W \times H \times C}$. A point cloud is however an unordered set of points $P = \{(x, y, z, r)_i\}_{i=0}^{N}$ where N is the total number of points, r is the reflectance and x, y and z are the spatial coordinates in the 3D space. This is why LiDAR-based approaches differ from image-based methods by the requirement of a point cloud encoder which is responsible of transforming the sparse point cloud $P = \{(x, y, z, r)_i\}_{i=0}^{N}$ to a dense representation $I \in \mathbb{R}^{W \times H \times C}$. Different kinds of CNN models for LiDAR based 3D object detection are studied in Sec. 3.5.

KITTI dataset Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI) [3] pioneered multimodal dataset providing front-facing stereo images, dense pointclouds from a lidar sensor as well as GPS/IMU data. The dense point clouds are generated using Velodyne HDL-64E rotating 3D laser scanner with a frequency of 10 Hz and 64 beams. The dataset consists of 7481 annotated samples which we split between training and validation with 200k 3D boxes over 22 scenes. Fig. 2.9 highlights an example of a sample from KITTI dataset along with its 3D bounding box annotations and CNN predictions.

Evaluation Metric The effectiveness of the object detection task is measured using the Mean Average Precision (mAP) metric. The metric depends on an IoU-based overlap criterion between the ground truth and predicted bounding boxes. Each class includes an IoU threshold condition (e.g. IoU greater than 0.7 for the car class) usually defined by the dataset suite [3]. If the defined IoU threshold condition is satisfied, then the predicted bounding is considered as a True Positive (TP). Similarly, if the overlap criterion is not satisfied the predicted bounding box is considered as a False Positive (FP). The average precision (AP) for a class c is given by Eq. 2.14. Taking the mean of these average individual-class-precision producesthe mean average precision (*mAP*).

$$AP_c = \frac{\#TP(c)}{\#TP(c) + \#FP(c)} \tag{2.14}$$

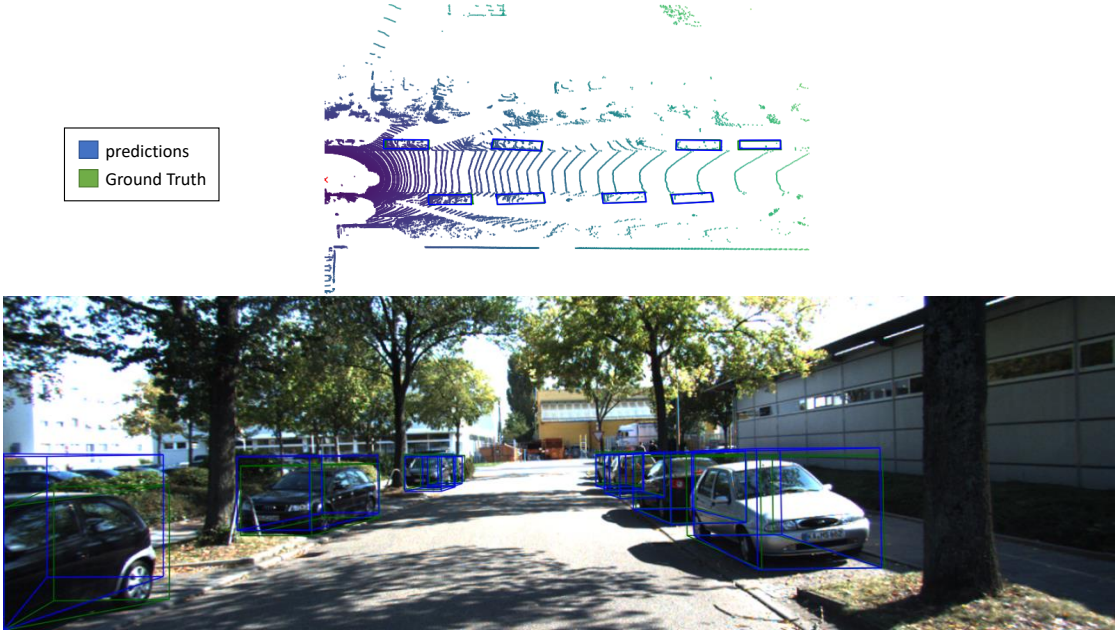


Figure 2.9: Example 3D bounding box predictions along with ground truth labels on KITTI dataset [3].

2.3 Compression Techniques

The improved performance of CNNs comes at the cost of becoming increasingly deeper and larger making it difficult to deploy them on edge devices. This necessitates compressing CNNs without loss of prediction accuracy so as to make them computationally more efficient. Various techniques for finding optimized configurations of CNN architectures have been proposed in the last few years. Among these, two popular methods, namely *Pruning and Quantization* have been discussed in this section.

2.3.1 Quantization

Quantization has become a standard technique in both industry and academia, typically applied before deploying CNNs in embedded settings. The benefits of quantization are manyfold, ranging from reducing the bit-width of weights and activations to shrink the model's size, to simplifying the arithmetic computation units on HW, thereby lowering the energy consumed by on-chip and off-chip data movement.

During the CNN training, it is common to represent weights, activations and their gradient using 32-bit floating point quantization. There are two main approaches of the quantization for CNNs : Post Train Quantization (PTQ) and Quantization Aware Training (QAT). In the PTQ, a full-precision pretrained model is quantized to a lower bit-width representation with few iterations of fine-tuning. This approach allows us to achieve the floating point accuracy at 8-bits, while below 8-bits, this results in significant accuracy degradation. Alternatively, quantization-aware training QAT methods are capable of producing quantized CNNs during the training process [7, 50].

Let $Q(x, b)$ be a quantization function with number of bits b, denoted by $Q : \mathcal{R} \rightarrow \{q_0, .., q_{2^b-1}\}$, consisting of 2^b discrete output values. Let us consider an operand $x \in \mathcal{R}$. The mapped value x_q can be derived using linear quantization based on number of assigned bits b as shown in Eq. 2.15. We calculate the absolute maximum value c in the data distribution. We ensure that the quantized data is in the range of $[-2^{b-1}, -2^{b-1} - 1, .., 0, .. 2^{b-1} - 1]$. We scale the quantized values using a scaling factor $\frac{c}{(2^{b-1}-1)}$.

$$\begin{aligned} c &= max|x| \\ x_q &= Floor(x \cdot \frac{(2^{b-1}-1)}{c}) \cdot \frac{c}{(2^{b-1}-1)} \end{aligned} \tag{2.15}$$

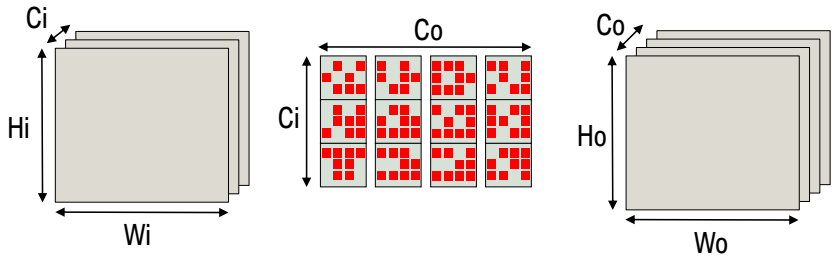
For positive data distributions, e.g. outputs upon ReLU based non linear activation, we map $x \in \mathcal{R}$ to positive discrete values in the range of $[0, 2^b - 1]$. Most methods such as [7, 50] use uniform quantization techniques. However, works such as [51, 52] use non uniform logarithmic quantization to reduce the error between full precision and quantized data types. In Eq. 2.16, we derive discrete values x_q using non linear quantization with first step size Δ, bit-width b and log base a.

$$\begin{aligned} x_q &= sign(x) \times \texttt{Clip}(\texttt{Floor}(\log_a \frac{x}{\Delta}), -2^{b-1}, 2^{b-1} - 1) \\ a^* &= \underset{a}{\operatorname{argmin}} |x - x_q| \end{aligned} \tag{2.16}$$

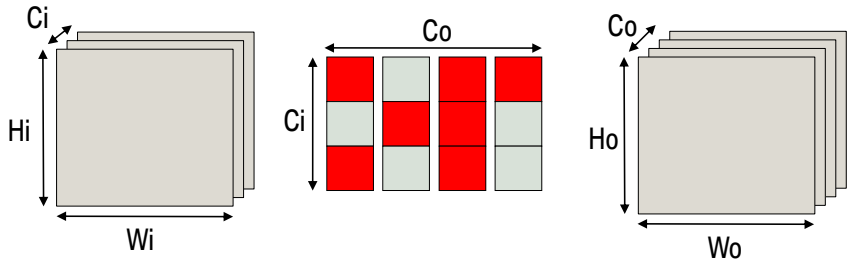
Binarization is an extreme form of quantization where the weights and activations are represented using a single bit. Courbariaux and Bengio proposed Binarized Neural Networks (BNNs) [53, 54], where the weights and activations are restricted to $\{+1, -1\}$. Since the weights and activations are binary, multiply and accumulate (MAC) operations typically used in convolution can be replaced with XNOR and popcount operations. Efficient BNN training schemes can be realized using STE [55, 56], multiple bases [57, 8] or latent free weights [58, 59]. BNNs drastically reduce memory size, accesses and improves power efficiency substantially without a significant degradation in prediction accuracy. Rastegari et al. [55] proposes an approach to enhance the performance of binarized CNNs, by multiplying the absolute mean of weights and activations with the 1-bit weight and activation. Liu et al. [56] improves the training scheme of BNN by approximating the discrete gradient of sign function with a magnitude aware piecewise polynomial function. To further mitigate the accuracy degradation of BNNs, Lin et al. [8] extended BNNs by approximating the full-precision convolutions in CNNs by using linear combinations of multiple binary bases for both weights M and activations N, resulting in Accurate Binary Convolutional Neural Networks (ABC-Nets). Thus, the convolutions of Multi Bit Networkss (MBNs) can be implemented by computing $M \times N$ bit-wise convolutions in parallel. Group-Net decomposes the network into mulitple groups, e.g. multiple binary residual blocks and improves the prediction accuracy. Recent efforts by Helwegen et al. [58] improves the training scheme of BNNs by avoiding latent weights and gradient approximation.

2.3.2 Pruning

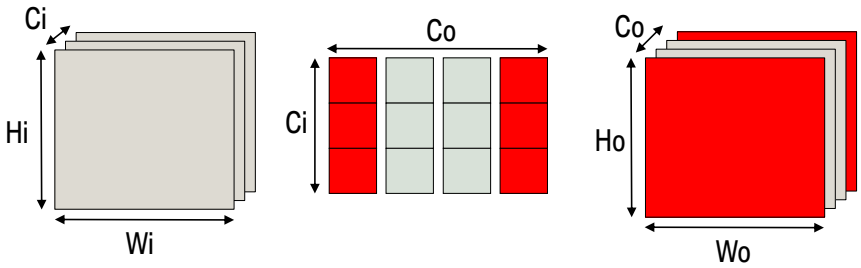
Pruning is a standard technique for removing redundancies in neural networks that do not affect the network performance. This involves removing weights, kernels or channels from CNNs. The granularity of CNNs pruning is defined by the *regularity*. Based on this, pruning can be classified into: *Structured Pruning* and *Unstructured Pruning*. Different pruning regularities have been illustrated in Fig.2.10.



(a) Irregular Pruning.



(b) Kernel Pruning.



(c) Filter Pruning.

Figure 2.10: Illustration of various pruning regularities. Irregular pruning identifies compression opportunities in an unstructured manner. Regular pruning identifies redundant kernel windows $K_h \times K_w$ in the weight matrix. Filter pruning identifies redundant output channels $K_h \times K_w \times C_i$ in the weight matrix.

Unstructured Pruning: Unstructured pruning involves identifying redundant individual weights from networks as illustrated in Fig. 2.10-a. This is relatively simpler to obtain high compression ratios and does not cause much accuracy degradation. However, weight pruning leads to a irregular structure which does not yield benefits when deployed on general-purpose hardware.

Structured Pruning: Structured pruning involves removing kernels, channels or filters as a whole from the network as illustrated in Fig. 2.10-b and Fig. 2.10-c. This is more challenging as compared to weight pruning but yields greater hardware advantages. When deployed on general purpose structured hardware like GPUs, the advantages of structured pruning can be exploited using existing Compute Unified Device Architecture (CUDA) kernels. Pruning input channels from a convolutional layer automatically leads to the filters in its previous layer getting pruned. Thus, the terms channel pruning and filter pruning can be used reversibly. However, this also makes it slightly complicated for implementing channel pruning on CNNs having complicated structures like residual blocks with convolutional shortcuts. During implementation, it has to be ensured that there is no mismatch in filters or channels for the convolutional layers. It is challenging to obtain higher compression ratios for structured pruning compared to weight pruning with minimal degradation in prediction accuracy. Apart from pruning regularity, pruning can also be classified into *handcrafted* or *automated pruning techniques*. Also, pruning approaches may be *post-train* or *in-train*. More details on these pruning techniques have been provided in chapters 3 and 4.

2.4 Hardware Accelerators for Neural Networks

The complexity of state-of-the-art CNNs increase by incorporating large number of parameters and a deeper structure in order to yield better prediction accuracy. This capability of a better performance comes at the expense of increasing computational requirements. Performing real-time inference of CNNs on mobile and embedded platforms, especially imposing real-time constraints is hard and critical. A possible way to accelerate CNNs is by leveraging the parallelization of the MAC operations on various compute units. This requires however specific hardware architectures offering a native support of multi-threading and a more efficient performance of convolution operations. The more efficient performance arises from the following optimizations embedded in the hardware architecture:

- Native support for parallel computations: This is an extremely important feature since convolutional layers apply a huge number of independent MAC operations. Increasing the utilization of Processing Elements (PEs) and maintaining the computational efficiency results in a lower latency.

- Low-precision arithmetic: CNNs can yield similar accuracy values even when the precision of activations and weights is decreased. A native support for low-precision MAC operations can therefore result in an overall improvement in efficiency (in terms of latency and energy) without affecting much the prediction accuracy.

- Efficient dataflow architectures: Inference is a memory-intensive operation if the memory architecture is not properly dimensioned. The amount of data volumes that are loaded and stored in memory requires a smooth dataflow and stall-free access patterns across different hierarchies.

These hardware platforms specifically designed to accelerate the required computations include GPUs, ASICs and FPGAs.

2.4.1 Graphical Processing Units

GPUs are the most commonly used hardware accelerator in the field of CNNs and are supported by the vast majority of Machine Learning software frameworks. They deliver the best computation speed and memory bandwidth when it comes to the training and inference of CNNs. The rapid development resulted in superior clock frequency (*GHz*) and memory bandwidth [60] (hundreds of *GB/s*) that allow to achieve the best results in terms of latency and prediction accuracy. Furthermore, GPUs offer user-friendly frameworks for developers such as CUDA. However, GPUs are greedy in terms of energy and are therefore not always suited for many embedded and mobile applications. Additionally, many optimization techniques and the use of custom data types are not supported in most current GPU-based platforms. This motivates the need for an alternative accelerator architecture which have a lower energy consumption and is more flexible in supporting custom data types and dataflow based optimization algorithms.

2.4.2 Application Specific Integrated Circuits

ASICs are designed to specialize in a specific use case and will permanently be only deployed for that single task, i.e. once it is desinged, it cannot be changed afterwards. The design process of ASICs is similar to one of FPGAs, where low-level hardware description languages such as Verilog or VHDL are used. ASIC-based CNN accelerators have a high performance (throughput), and lower energy consumption. In fact, convolutions can be computed using low-precision data types, and memory access patterns can be customized and accelerated using different hardware configurations and optimization techniques. However, the design process of ASICs is relatively slow and inflexible. With the fast evolving requirements of CNN deployment, the used hardware accelerator is required to be more adaptive and responsive to quick changes and facilitates the integration of state-of-the-art optimization patterns.

Custom ASIC architectures have achieved significant improvement in throughput and energy efficiency. DaDianNao [61] relies on large on-chip memory (consumes nearly half of the area) and achieves reduction in latency and power savings compared to the GPU. A flexible accelerator Eyeriss [11] was able to accelerate convolutional layers of AlexNet, VGG-16 with low power consumption. Eyeriss uses 16-bit fixed point with a flexibility of zero gating in the PEs. As discussed in Sec. 2.3, it is simple to produce high compression rates using irregular pruning. Accelerators such as EIE [62], SCNN [63] leverage the irregular sparsity to avoid the scheduling of zero multiplications with specialized circuits. EIE [62] is an efficient method for highly irregular pruning, introducing compressed representation for weights and activation to bypass zero multiplications. The implementation is limited to FC layers. SCNN [63] discusses an accelerator design that can be used on convolutional layers, with sparse dataflow capable of

detecting and scheduling only MAC operation with non-zero activation and weights greatly reducing latency and energy consumption.

2.4.3 Field Programmable Gate Arrays

FPGA is an integrated circuit which can be programmed to have a custom design capable of performing a specific task or functionality. FPGAs allow a high degree of flexibility and customizability. FPGA is an array of interconnected circuits containing components dedicated for specific functions as well as generic configurable logic units. This combination of custom and generic units allows lower power consumption and higher performance.

The core unit of an FPGA is the configurable Adaptive Logic Module (ALM) which consists of combinational Look-Up Table (LUT) and registers. These programmable logic elements can perform custom functions designed by the developer. In addition to these configurable logic units, FPGAs contain custom hard blocks optimized for a specific functionality such as the DSP blocks which can perform 8-bit/ 16-bit or floating point MAC operations per unit. These blocks can mainly be leveraged to execute the convolution operations.

In Arria-10 FPGA [64, 65, 66], there are two types of embedded memory blocks: 640 bit Memory Logic Array Blocks (MLABs) and 20 Kb M20K blocks. The MLABs are enhanced SRAM blocks optimized for specific functionalities such as shift registers and wide FIFO buffers. The M20Ks are used to store large memory arrays while offering enough independent access ports. These will be the main memory blocks used for storing input and weight features. The Arria 10 FPGA offers 2713 M20Ks resulting in a storage size of 54260 Kbits. However, the offered SRAM blocks are not enough for storing all input features, filters and partial sums of several state-of-the-art CNNs. Therefore, the external memory bandwidth is of great importance because the overall performance of the hardware accelerator is bounded by the speed of memory accesses. For Arria 10 GX board, the external memory bandwidth is limited to 19.2 GB/s. This makes the design task of a performant accelerator a challenging one. Xilinx FPGA such as the ZCU102 evaluation board [67] consists of 2520 DSP48E2 slices [68]. Each DSP block contains $27bit \times 18bit$ multiplier. It consists of 600K logic blocks and 32.1 Mb of BRAM.

The traditional FPGA design process requires expert HW knowledge as the logic blocks and connections need to be described manually using a hardware description language (HDL) like Verilog or VHDL. The FPGA bit stream is synthesized after mapping all the described components to functions either in form of hard-logic blocks if the function is available or otherwise using soft logic blocks. The synthesis requires furthermore the functions to be placed using the available hardware resources and connected during the routing process. If all these processes are successful, the FPGA bit stream is generated. However, the HDL-based low level languages make the development of the FPGA design demanding in terms of knowledge, development time and cost.

A high-level alternative language to describe the FPGA-based accelerator design is OpenCL. This language is very similar to C in terms of syntax, but it supports the parallelization of multiple instructions. OpenCL supports different platforms such as GPUs and FPGAs and is therefore widely used as a language. Even though the syntax of this high-level language is generic to multiple platforms, the compiler is always specific to the target device. The OpenCL program which describes the accelerator on the FPGA board is called kernel, and is compiled using the Altera Offline Compiler (AOC) for the Arria-10 FPGA.

2.5 Adversarial Robustness

Adversarial Robustness illustrates the capability of a network or system to be resilient against a attacks imposed by an adversary. An adversary in this context tries to modify the input of the CNN slightly to manipulate the prediction of a system to malfunction. These slight variations are often imperceptible for humans, so we expect the system not to alternate its predictions. There are adversarial attack schemes that differ in approach, effectiveness, and efficiency. The adversarial attack schemes can be categorized into *black-* and *white-box* attacks [69]. They differ in the knowledge that the attacker has about the CNN model. For black-box attacks, attackers have no information about the intrinsic of a CNN, such as its architecture or weights. However, the attack scheme has access to a variable number of predictions. The number of predictions is mostly dependent on inference latency, network traffic (if online), or defense mechanisms protecting against numerous queries from the same source. However, multiple smart queries with slightly manipulated inputs disclose some model intrinsics. For white-box attacks, the attacker has full knowledge about the network structure and can use the gradient information to construct effective adversaries. The following sections explain white box and black box attacks as it is necessary to understand against what compressed CNN needs to get defended. The defensive compression schemes are explained more in Sec. 5.4.

2.5.1 White Box Attacks

White-box attacks are more fierce attack schemes as the adversary is aware of the details about the model and do not need to reconstruct a model or its gradient. Therefore, the most significant benefit for white-box attacks is their reduction in queries, as most common white-box approaches use the model gradient with respect to the input directly. Using this gradient turns out to be a very effective and efficient (fast) way to construct successful and strong perturbations. The mentioned points are the primary motivation to focus on and protect against white-box attacks that are gradient-based in the scope of this thesis.

Gradient-based approaches still differ in multiple aspects. First, an attack can be targeted or untargeted. The latter means the attack is successful as soon as the model is making a wrong prediction. Targeted attacks try to trick the model into making a specifically wanted, wrong prediction. Therefore, the targeted attacks are more dangerous for real-world scenarios like AD, if they are successful. In the following, the most common approaches are briefly explained.

Fast Gradient Sign Method: The most commonly used attack to verify the robustness of neural networks against input perturbations is the Fast Gradient Sign Method (FGSM) [70]. FGSM linearizes the loss function of a neural network around θ by calculating its gradient $\nabla\mathcal{L}(I, L, \theta)$ to generate adversarial examples I^{Adv}. The input variation parameter ϵ controls the perturbation's amplitude [71], as expressed in Eq. 2.17.

$$I^{Adv} = I + \epsilon \cdot \text{sign}\left(\nabla\mathcal{L}\left(I, L, \theta\right)\right) \tag{2.17}$$

The attack is strengthened when performed iteratively. This can be considered as an extension of FGSM, generating adversarial samples using a small step-size [71].

Projected Gradient Descent: An even more effective variant is iterative Projected Gradient Descent (PGD) on the loss function with uniform random noise initialization [72], expressed in

Eq. 2.18.

$$I_{i+1}^{Adv} = \pi_{\mathcal{S}} \left(I_i^{Adv} + \alpha \cdot \nabla \mathcal{L} \left(I_i^{Adv}, L, \theta \right) \right) \tag{2.18}$$

Here, adversary examples I_{i+1}^{Adv} are generated by taking one step into the ascent direction of the loss gradient $\nabla \mathcal{L}(I_i^{Adv}, L, \theta)$ with respect to the previous image I_i^{Adv} at iteration i, where the step-size is scaled by α, followed by a potential projection π onto the legal set $\mathcal{S}$. Legal adversaries are ensured by a projection π onto the legal set $\mathcal{I} + \mathcal{S}$ with $\mathcal{S} = \{\delta : ||\delta||_p \leq \epsilon\}$. A projection onto the legal set π_S is performed by clipping δ to the interval $[-\epsilon, \epsilon]$. I_i defines the image at iteration step i and α defines the step-size of the optimization step in the ascent direction of $\nabla_I \mathcal{L}$. In the chapter 5, the PGD analysis is performed using ℓ_∞-norm as a distance metric between natural image I, and adversarial example I^{Adv}. The iterative multi-step optimization method is able to converge to local maxima of the non-concave and constrained maximization problem representing possible worst-case adversaries for the underlying model. By considering random uniform initialization, arbitrary starting points on the corresponding loss surface are ensured, thus resulting in an exploration of potentially varying local maxima and lastly giving rise to the structural behavior of the corresponding loss surface. This renders the PGD attack as the *ultimate* first-order adversary, as stated by Madry et al. [18]. In Sec. 5.3, we analyze various hyperparameters of PGD attack for different compression techniques.

DeepFool: With the DeepFool [73] attack, the authors propose a method to generate adversarial examples that fool classifiers on large-scale datasets by estimating the distance of an input instance I to the closest decision boundary. The iterative method estimates the perturbation δ_i at each iteration i till the classifier $f(I_i)$ changes its prediction ($f(I_i) \neq L$). In practice, once an adversarial perturbation δ is found, the adversarial example is pushed further beyond the decision boundary. The algorithm is not guaranteed to converge to the optimal perturbation, nevertheless it generates adversarial examples with good approximations of the minimal perturbation. The size of the calculated perturbation can also be interpreted as a metric for the model's robustness against adversarial attacks [74].

Carlini & Wagner: Carlini and Wagner (C&W) [75] presented a targeted attack, to refute the promising defensive approach of defensive distillation [76]. The proposed C&W attack emerged as one of the strongest attacks in literature [77]. C&W finds perturbations δ with minimal distance $D(I, I + \delta)$ that will change the classification of image I to the target class t. This is a challenging non-linear optimization problem and therefore the authors introduce a function g, such that $g(I + \delta) = 0$ when the classifier gets fooled towards the target class. The attack constructs adversarial examples which try to minimize the objective as mentioned in Eq. 2.19.

$$\begin{gathered} \min(\|\delta\|_p + \epsilon \cdot g(I + \delta)), \\ \text{where } g(I) = ((\max_{j \neq t} Z(I)_j) - Z(I)_t)^+ \end{gathered} \tag{2.19}$$

$Z(I)_j$ indicates the output of the CNN for class j before the softmax layer. The minimum condition $g(I) = 0$ occurs when $Z(I)_t \leq Z(I)_j \; \forall j \neq t$. The choice of ϵ maintains a trade-off between the attacked image similarity and the success rate of the target class. Using ℓ_2 distance metric, the objective function is minimized through the gradient decent. In Sec. 5.4, we formulate defensive compression methods, which are robust against C&W attack.

2.5.2 Black Box Attacks

Black-box attacks tend to be closer and more relevant for the deployment phase of neural networks as most attackers will not have additional information about the intrinsic of a CNN. In this thesis, we explore two black box attacks namely, GenAttack and LocalSearch.

GenAttack: GenAttack [78] is a gradient-free optimization strategy based on a genetic algorithm. The initial population of perturbed image examples is generated by adding uniform random noise. The best individuals survive the generation based on their fitness evaluation, the selection strategy and the crossover and mutation probabilities. Fitness evaluation reflects the optimization objective, while the selection strategy allows elite individuals in the population to generate new children perturbations through crossover and mutation mechanisms. GenAttack is a faster search algorithm when compared to LocalSearch [79], and generates perturbations which are imperceptible to the human eye.

LocalSearch: LocalSearch [79] is a simple gradient-free adversarial black-box attack, which is based on random perturbation and a *greedy search algorithm* around the perturbed pixels. The LocalSearch procedure works in iterations, where each iteration consists of two steps. The first step is to select and evaluate a small subset of points P_i, referred to as the *local neighborhood*. In the second step, a new solution P_{i+1} is selected by taking the evaluation of the previous solution P_i into account. LocalSearch is simple to implement, but is computationally expensive, similar to most greedy search algorithms. In Sec. 5.3, we perform detailed analysis to understand the influence of various white box and black box adversarial attacks on various compressed CNN variants.

3 Hardware Aware Neural Network Compression

Modern CNNs demand enormous latency and memory access, making them both computationally and memory intensive. CNN compression is essential to realize an efficient embedded application. The architecture of the CNNs is *fixed* before the training process starts. Therefore, the standard training which traditionally leverages cross entropy and regularization loss does not influence the architectural configuration. Identifying the redundant computations and parameters which produces minimum degradation in task specific accuracy and improving the HW specific metrics is refereed as *HW aware CNN compression*. There has been an increased focus on model compression [19, 80, 6, 27] and acceleration techniques [81, 82] for CNNs. These techniques tackle the compute complexity for deployment needs in embedded scenarios, like reduced storage requirements [83, 84] and inference time [85]. Evaluating a compression strategy based on proxy metrics, such as parameter and operation counts is loosely correlated to HW benefits and can lead to sub-optimal deployment setups. This has influenced recent works to optimize neural networks with Hardware-In-the-Loop (HIL) approaches [5, 13, 14]. Reliance on proxy metrics oversimplifies the problem at hand and does not always guarantee improvements in energy or latency when deployed on real HW. In this chapter, we realize *HW aware CNNs* by incorporating HW-metrics directly in the compression loop.

The upcoming sections are structured as follows: Sec. 3.1 introduces three stage pipeline to attain HW-aware compressed model. Sec. 3.2 discusses related work on CNN pruning and its impact on HW-metrics. Sec. 3.3 formulates a HW-model for spatial CNN accelerator to predict estimates required during the compression stage. Sec. 3.4 demonstrates a RL based channel pruning pipeline using estimates from the HW-model. Sec. 3.5 highlights the effectiveness of the search pipeline on LiDAR-based 3D object detection. The chapter is based on the publication of Vemparala et al. [25] and Vemparala et al. [26].

3.1 Post Train Compression

In this section, we describe various components in the CNN optimization pipeline (Sec. 3.1.1) which **considers a pre-trained model** and determines suitable compression configuration. We further discuss about automated search methods (Sec. 3.1.2) such as RL [5, 13], Genetic Algorithm (GA) [14, 15] which are commonly employed to determine the compression configuration. We finally discuss HW-based Key Performance Indicators (KPIs) in Sec. 3.1.3, which can provide feedback to accomplish *HW-awareness* for the automated search techniques.

3.1.1 Three Stage Pipeline

Finding the correct layer-wise compression strategy using pruning and quantization methods with respect to a target HW platform is a complex problem. Earlier works use uniform pruning and quantization strategy. Uniform pruning techniques rely on constant pruning rates across the layers and handcrafted heuristics such as magnitude-based [19] or geometric mean based [20] to compute the saliency of weight matrix. Uniform quantization methods assign equal bit-width to weights and activations for the entire CNN's representation. These techniques rely on efficient QAT methods [50, 7] which are capable of modeling the error introduced by the discretization of weights and activations. However, different layers contribute differently to the accuracy and HW efficiency of a network [15], justifying the use of different pruning and quantization degrees for various layers of the CNN. To obtain an efficient HW-aware compressed model, we introduce the three stage compression pipeline as shown in the Fig. 3.1.

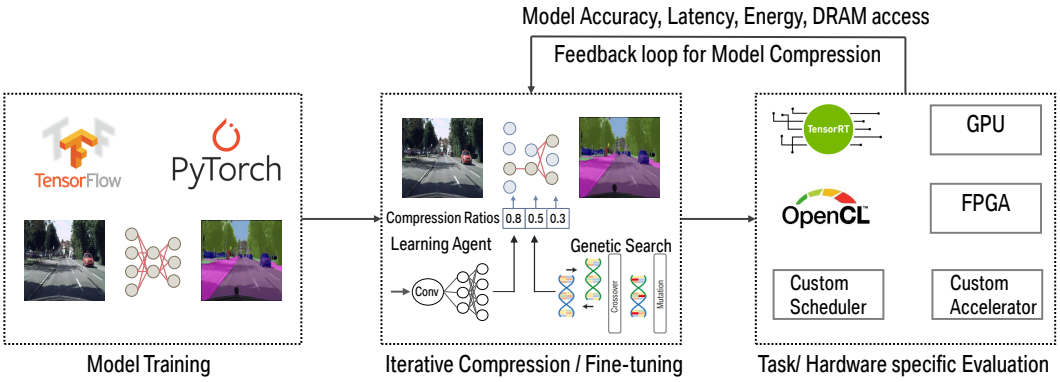


Figure 3.1: Illustration of three stage hardware aware compression pipeline.

Model Training: The first stage of the pipeline refers to the training of the L layer baseline model with weights $\{W_l\}_{l=1}^{L}$. Here $\{W_l\}$ denotes the weight parameter tensor of the l-th layer. The convolutional layer $l \in \{1, ..., L\}$ receives an input feature map $A^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$, where H_i, W_i, and C_i indicate the spatial height, width, and input channels respectively. The weights $\{\hat{W}_l\} \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o}$ are the trainable parameters of the individual layers, here K_h, K_w, and C_o are the kernel dimensions and the number of output channels (filters) respectively. The baseline model is trained for different applications as highlighted in Sec. 2.2. We obtain the pretrained weights by following the training procedure and optimizers as discussed in Sec. 2.1.6. The baseline model is typically trained using full/half precision data types using large scale training frameworks such as TensorFlow [86] and Pytorch [87]. The baseline model without the compression pipeline is over-parameterized and demands more latency, energy consumption and DRAM access.

Search for optimal compression: As discussed in Sec. 2.3, we rely on pruning and quantization to compress CNN model. In pruning, the weight matrix is zeroized using binary masks $\mathcal{M}^l = \{0,1\}^{K_w \times K_h \times C_i \times C_o}$ (irregular pruning), $\mathcal{M}^l = \{0,1\}^{1 \times 1 \times C_i \times C_o}$ (kernel pruning), $\mathcal{M}^l = \{0,1\}^{1 \times 1 \times C_i \times 1}$ (channel pruning). The Hadamard product $\odot$ is applied to realize a

sparse representation of weight matrix $\tilde{W}^l \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o} = W^l \odot \mathcal{M}^l$. To improve the trade-off between prediction accuracy and HW-metrics, search algorithms such as RL [5] and GA [14] determine layer-wise compression ratios and locations of redundant weight elements. To mitigate the search complexity, works such as AMC [5] determine layer-wise compression ratios. In-order to effectively guide the search algorithm, iterative fine-tuning becomes necessary [15, 26] for few epochs (e.g. 0.5 - 2 epochs) to recover the loss in accuracy. This is particularly important for channel pruning and quantization search.

Model Evaluation: The goal of compression pipeline is to complement well-established proxies, such as number of operations and parameter counts, with more elaborate HW-model based estimates, which are conducive to finding efficient CNNs for embedded applications. The HW metrics are determined by executing the model inference on different target platforms as shown in the Fig. 3.1. The model inference on NVIDIA-GPUs can be efficiently performed using TensorRT [88]. The connections and HW architecture in the FPGA can be described using High-Level Synthesis (HLS) [89] or OpenCL [85]. In embedded applications such as autonomous driving and robotics, the design of neural networks and the target HW accelerator goes hand in hand. During the early development phases, it is likely that the target platform is not fully defined, the HW is not available, or compilers are prone to errors, making a HIL-based approach challenging. In these scenarios, HW model is an essential tool for CNN optimization. The HW model takes CNN architecture description as input and provides HW metrics such as latency, energy or DRAM access as output.

3.1.2 Automated Search Methods for Channel Pruning

The goal of the search algorithms is to improve the trade-off between prediction accuracy and model compression compared to baseline models. Inorder to determine an efficient compression strategy, search algorithms such as RL and GA need to determine the locations of redundant weight elements. Furthermore, fine-tuning is required to reduce the gap for prediction accuracy between baseline and compressed model.

Search space complexity: In-order to systematically determine a compressed CNN model, it is important to formulate a search space indicating the compression opportunities for the search algorithm. The size of the search space depends upon the chosen pruning regularity. In irregular weight pruning, the weight matrix at layer l i.e. $\{W_l\} \in \mathbb{R}^{H_i \times W_i \times C_i \times C_o}$, requires binary pruning masks $\mathcal{M}^l = \{0,1\}^{K_w \times K_h \times C_i \times C_o}$. For a L layer CNN model, the total number of compression possibilities or search space complexity $s = \prod_{l=1}^{L}(2^{H_i^l \times W_i^l \times C_i^l \times C_o^l})$. The search space can be reduced by using pruning regularities such as kernel pruning ($s = \prod_{l=1}^{L}(2^{1\times 1\times C_i^l \times C_o^l})$) or channel pruning ($s = \prod_{l=1}^{L}(2^{1\times 1\times C_i^l \times 1})$). Even though irregular and kernel pruning have greater search space and lead to to better prediction accuracies for a given compression ratio, they require specific hardware designs to extract HW benefits. Channel/Filter pruning removes whole filters from each layer and is therefore the most hardware-friendly pruning type. Furthermore, higher search space complexity requires huge number of GPU hours for the compression pipeline. In this work, we study the search space for channel/filter pruning and understand the improvements in HW benefits.

RL based pruning: RL based search algorithms consists of three components, namely environment, agent, reward model. In the context of model compression, we consider the environment

f as CNN model, the pruning agent π deciding the compression configuration and HW model μ providing reward $\mathcal{R}$. The environment provides state $\mathcal{S}$ as the input to the RL agent. The state space contains all the information about the environment that the agent observes [90]. In the work of AMC [5], the state of the art RL agent is encoded using key architectural information such as layer number l,stride s, filters C_o, input channels C_i, number of remaining channels $\sum_{j=l+1}^{L} \varphi^j$. RL-agent used in [21] uses fully-trained filters matrix $W^l|_{l=0}^{L}$ as state space. The agent produces actions $\mathcal{A}^l$ to the CNN environment for each layer $l \in \{1, L\}$ and obtains reward $\mathcal{R}$ indicating the effectiveness of compression configuration by analyzing the trade-off between prediction accuracy and HW benefits. The pruning action $\mathcal{A}^l$ at layer l can be either discrete or continuous decision. Discrete pruning action refers to specifying the redundant filter locations as $\{a_1^\ell, a_2^\ell, ..., a_{N^l}^\ell\}$, where $a_i^\ell \in \{0, 1\}$ is equivalent to $\{prune, keep\}$ and N^ℓ is the number of filters in the ℓ^{th} layer [21]. Using this scheme, the agent is able to explore both sparsity ratio and to select the exact position of filters to prune. The work in AMC produces continuous actions in the form of layer-wise compression ratios as $\{c^1, c^2, ..., c^L\}$, where $c^\ell \in (0, 1]$ for each layer ℓ. Based on the compression ratio, magnitude based heuristic is employed to identify redundant elements in kernel matrix. The quality of an action is determined by the reward signal. The goal of RL is to learn and acquire an agent which is capable of selecting the optimal action with current given state. This agent is also called *policy* and can be expressed as given in Equation 3.1, where $\pi(a|s)$ represents the policy.

$$\pi(\mathcal{A}|\mathcal{S}) = \mathbb{P}[A_t = \mathcal{A}|S_t = \mathcal{S}] \tag{3.1}$$

Huang et al. [21] use the Stochastic Policy Gradient (SPG) [90] method to find an optimal policy π^*. The learnable parameters of the agent are updated with gradient ascent so that actions with higher rewards are more probable to be sampled [91]. Deterministic policy is also defined as $\mathcal{A} = \pi(\mathcal{S})$, which means the current action of the agent is deterministic and depends only on the current state. Deterministic policy is commonly used for applications with continuous action requirement or extraordinary large discrete action space. In deeper layers of CNNs such as ResNets [39], the number of discrete actions for channel pruning exceeds 10^3. However with continuous action, there is only one value which is suitable for deeper and wider CNNs. In the work of AMC [5], actor-critic based Deep Deterministic Policy Gradient (DDPG) agent is used.

GA based pruning: GAs are a class of evolutionary algorithms inspired from the process of natural selection. In natural selection, species that can adapt well to changes in the environment survive and go to the next generation, a concept also termed as survival of the fittest. Various variants of GAs are available in literature. Using Non Sorting Genetic Algorithm (NSGA)-II [92], mixed precision [15] and pruned [93] CNNs are derived. Channel pruning can be formulated as a search problem, where redundant filters are pruned based on layer-wise compression ratios and a magnitude-based heuristic. The pruning rates result in an integer number of remaining channels for each layer. Pruning certain filters leads to large degradation in prediction accuracy, highlighting different sensitivities for various pruning choices. Moreover, pruning makes large portions of the total computations possible in a single tile, leading to schedule-dependent improvements in latency. These aspects make the search space *discrete* and *non-differentiable*. GAs are useful to tackle this search problem, as they are known to be resilient to noisy search spaces, quick to prototype, and do not need smooth, continuous search spaces to perform well (gradient-free).

This also relieves the burden of handcrafting a cost function to combine the multiple criteria (accuracy, latency, OPs), by using NSGA-II, which follows a multi-objective Pareto-optimal selection approach. This ultimately facilitates better design space exploration, and maintains diverse non-dominated trade-off solutions which may fit different deployment scenarios.

The pruning problem at hand demands solving a multi-criteria optimization problem with two opposing objectives. The aim is to find the layer-wise sparsity ratios such that the computation effort in terms of number of operations or hardware estimate like latency is minimized. At the same time, we need to ensure that this does not affect the prediction accuracy of task at hand. Aggressive pruning leads to a drop in prediction accuracy, whereas, higher accuracy is costly in terms of hardware. Hence, there is no unique best solution but a trade-off between accuracy and latency has to be made. NSGA provides a set of pareto-optimal solutions that balances accuracy and hardware estimate.

In Sec. 3.4, we incorporate execution metrics in AMC based RL pruning search [5] using HW-model to obtain HW-aware pruning configurations. In Sec. 4.3, we also reduce the number of GPU hours for try and learn based RL pruning search [21] by appending fine-tuning based continuous reward.

3.1.3 Hardware Metrics for Model Inference

To understand the benefits in HW metrics for the compressed CNN model inference, we analyze the following metrics.

Operations and Parameters: Without the knowledge of HW accelerator, the complexity of CNN model inference can be measured by the total number of operations (`OPs`) and parameters (`Params`). These metrics help the CNN designer to understand the possibility of CNN deployment with latency constraints under peak utilization of compute units and memory blocks. In Sec.2.1, we discussed the number of operations for different kinds of layers.

External Memory access: External memory is also refereed as off-chip, where the weight matrix, inputs and partial outputs are stored. As the on-chip memory is limited (typically KBs or MBs), weights and Ifmaps are loaded as tiles from external memory (typically GBs). The amount of memory traffic measured in MBs between off-chip memory and on-chip memory is refereed as external memory access. Due to the limited memory bandwidth in CNN accelerators, the higher number external memory access creates stalls in the inference pipeline. The most power consuming component of the CNN accelerator is the off-chip memory access. The cost of accessing data from external memory is 200$\times$ more costlier than performing a MAC operation [11].

Latency: The latency of the CNN is the time interval between the stimulation of the input image and its response as prediction logits from the accelerator. The latency is measured in milliseconds (ms). Inference latency depends upon various factors such as HW platform, software environment, compiler tools, CNN architecture description and layer-wise quantization strategy. The latency is fed to optimization pipelines through HW model based predictors, look up tables or HIL measurements.

Throughput: The processing throughput is measured in Frames Per Second (FPS). Moreover, the computational throughput is measured in Giga Operations Per Second (GOPS). Processing throughput is an important metric to understand the effectiveness of batch processing or parallel

execution of multiple layers. Computational throughput provides an understanding of pipeline stalls and compute utilization of the accelerator.

Energy Consumption: The energy consumption of the CNN accelerator is determined using the data movement costs at various memory hierarchies and compute cost in PE array. The energy consumption is measured in Joules (J). The energy consumption depends upon various factors such as HW architecture, CNN architecture, compiler tools and scheduling scheme.

3.2 Related Work

Based on the target optimization metrics, we classify pruning techniques in the literature into three categories: HW-agnostic, HW-aware, and HW-modeling-based pruning techniques. Additionally, we discuss HW-modeling works that compute the HW estimates of CNN accelerators in literature.

3.2.1 Hardware Agnostic Pruning

The advantages of pruning were investigated in early works such as [94, 95]. Subsequent works determined the redundant weights based on an iterative method, without considering any target hardware resource constraints, *e.g.* magnitude-based pruning [96]. Recently, He et al. [20] pruned redundant filters using a geometric median heuristic. However, the efficiency term was limited to the Pruning Rate (PR), i.e., the ratio of pruned to total parameters. The PR was set constant to all the layers, which does not capture the energy or latency requirements of the target inference HW. The work by Guo et al. [97], dynamically pruned CNNs irregularly based on a saliency function during training to produce efficient networks. Recently, Frickenstein et al. [98] proposed the auto-encoder-based low-rank filter-sharing technique (ALF), which utilizes sparse auto-encoders to extract the most salient features of convolutional layers, pruning redundant filters. ALF approximates weight filters of existing CNNs and thus, reduces the gap between increasing hardware requirements of state-of-the-art networks and the constrained setup of embedded applications. The authors of [4] proposed structured channel pruning, where the saliency of individual channels is determined through Lasso regression. The pruning ratio for each layer is based on handcrafted heuristics which targets lower *proxy* metrics such as `OPs` and `Params`. In more recent works, automated pruning methods have gained popularity. Huang et al. [21] uses the REINFORCE algorithm [99] to formulate layer-specific agents, which receive the kernel matrix as a state and produce actions to prune exact filters. Different to other RL-agent based pruning work [5], here the agent has a more complex task of learning the features of a layer rather than simply its sparsity ratio. The agent's reward is formulated using a multi-objective cost function, which aims to find CNN models with both high accuracy and low *proxy* metrics. These works do not search for a pruning configuration specific to particular target HW-platform but instead identify redundant trainable weight parameters.

3.2.2 Hardware Aware Pruning

As HW platforms tend to be complex, the effects of arbitration, stalls, etc., may be severely understated if HW estimations purely rely on proxy metrics such as `OPs` and `Params`. Moreover, the highly complex design problem of task-to-resource mapping, dataflow scheduling, and

memory management further complicates the issue of using simple estimation proxies. By considering real HW metrics, HIL based compression pipelines have been used to verify the advantages of CNN optimization techniques pragmatically [100, 22, 5]. NetAdapt [100] prunes filters based on a pre-existing look-up table of HW metrics obtained ahead of time from a mobile device. This is a costly approach, as building the look-up table is tedious and time consuming, requiring the designer to execute all possible workloads and layer dimensions to be accurate and complete. Furthermore, the look-up table is rendered useless once there are changes in HW architecture or compiler updates. For this method to work, the HW would need to be decided and readily available before the CNN optimization process starts. Another drawback to the approach is that the pruning technique is performed in a layer-wise manner, which is susceptible to local minima, as inter-layer effects on the hardware platform and prediction accuracy are not considered. ChamNet [22] also adopts a look-up table strategy to estimate the latency with a Bayesian energy predictor and performs neural architectural search. The predictors for the HW metrics also require the "ready-to-use" target HW platform to perform optimization. Furthermore, if the target HW is changed, the effort to recollect the data for the new look-up table and the Bayesian optimizer needs to be taken into account. HW-NAS-Bench [101] presents a dataset to evaluate various CNN configurations on different HW platforms. The dataset is generated by performing extensive real HW measurements on Neural Architecture Search (NAS)-specific search spaces [102, 103]. Furthermore, the dataset does not cover exploration of HW specific hyper-parameters which impacts the CNN compilation/scheduling procedures. The work in AMC-AutoML [5] demonstrated an RL pruning agent, producing channel sparsity ratios for each layer as its action after every episode. Based on the magnitude obtained from the L2-norm heuristic and the sparsity ratio of each layer given by the RL agent, the redundant channels are pruned. The work demonstrated results of both proxy metrics (`OPs` and `Params`) and HIL-based timing evaluation using TF-Lite. Another HIL-based optimization technique, HAQ [13], resorts to RL-based exploration to determine suitable, layer-wise quantization levels for weights and activations in the CNN model. The realized mixed precision CNNs require dedicated processing elements to derive benefits in HW metrics. The reward function, including real HW metrics, is generated by directly executing the inference of a CNN model on a Field Programmable Gate Array (FPGA) design which supports quantized computations [104]. In this chapter, we only discuss channel pruning based compression as it eases out the additional HW implementation effort.

3.2.3 Hardware Modeling

HW-modelling offers the CNN designer the ability to explore and modify specific details of the accelerator, such as the memory hierarchy, PE dimensioning and the scheduling strategies to obtain new estimates. These estimates guide the CNN pruning based search algorithms without requiring the costly synthesized HW design. The deterministic nature of CNN inference execution on hardware makes analytical hardware modeling an intuitive approach to simulate aspects of the synthesis and deployment phases. Timeloop [105] is a HW-modeling tool that exploits CNN execution determinism to offer accurate estimates of a given description. The strength of modeling is to circumventing the need for cycle-accurate simulators and/or synthesized hardware in the early phases of development. The tool provides the flexibility of changing the cost of

operations (*e.g.* read, write, multiply-accumulate) and the memory hierarchy, among other design parameters. Based on the data movement constraints set by the designer, the tool searches the scheduling solution space in an exhaustive or randomly sampled manner, thereby providing the HW estimates. The schedule search time could either last significantly long with exhaustive search or lead to a sub-optimal solution with random sampling. Interstellar [106] proposes formal dataflow definitions. Unlike Timeloop, the authors of Interstellar use the Halide programming language to represent the HW-architecture and data movement constraints. The influence of memory hierarchy and dataflows on energy efficiency and latency is investigated thoroughly. MAGNet [107] considers various CNN architectures and hardware constraints generating an optimal register-transfer level (RTL) and mapping strategy to execute the CNNs efficiently. It explores various tiling strategies and dataflows by proposing a highly configurable processing element array. Yang et al. [108] leverage a HW-model to estimate the energy requirements of each layer. The layers with the highest energy contribution present a good starting point for the pruning process, based on the L2-norm heuristic. However, energy estimates do not influence the sparsity ratio directly. The work is also limited to optimizing normalized energy, but not latency, which is an equally important parameter for real-time applications.

In Sec. 3.3 and Sec. 3.4, we remove the limitations of pure proxy and HIL-based neural network pruning by introducing a HW-model for estimating the efficiency of CNN architectures. Instead of exhaustive and random sampling search techniques, we analytically reduce the size of the search space, and thereby the search-time, without sacrificing schedule efficiency. We integrate the HW-model into compression pipeline using DDPG based RL search, GA search to obtain efficient pruning configurations. In Sec. 3.5, we investigate an end-end HW-aware channel pruning based CNN deployment pipeline for 3D object detection on NVIDIA-1080TI GPU.

3.3 Modelling Neural Network Accelerator

The core components in the HW-model of CNN accelerator are the compute and memory blocks. Fig. 3.2 demonstrates an example deployment of a convolutional layer on a HW-model. In-order to obtain HW-estimates, the layers of CNN need to be *scheduled* into memory blocks ensuring minimal data movement at various levels of hierarchy. The tiles at the lowest level of memory hierarchy are *mapped* into the compute block consisting of PEs. In Sec. 3.3.1, we discuss about compute core and memory hierarchy of spatial CNN accelerators. We discuss about scheduling (Sec. 3.3.2), mapping methods (Sec. 3.3.3) which reduce energy consumption and latency based on the formulated model. Furthermore, we formulate the schedule search space and propose analytical search method to generate HW estimates quickly in Sec. 3.3.4. In Sec. 3.3.5, we explore different parameters of search space to determine efficient schedule and validate our HW-estimates with Eyeriss [11] accelerator.

3.3.1 Compute and Memory Architectures

The compute block is defined by several parameters, including the number of PEs, interconnect dimensions, register file sizes and quantization support. The register files in each PE can be

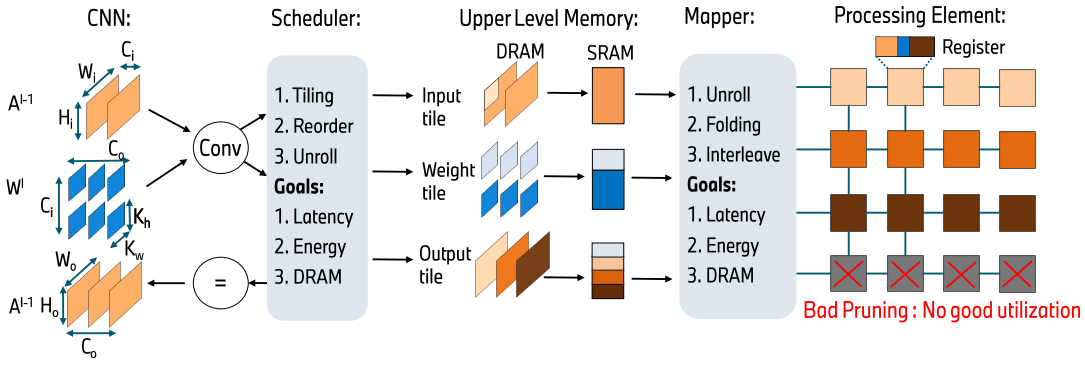


Figure 3.2: Scheduling and mapping CNN workload on different components of HW-model.

specified according to their size and its partition for Ifmaps, Ofmaps and weights as shown in Fig. 3.2 (top/right). Using these blocks, diverse compute architectures can be described.

A memory hierarchy can be created using generic memory blocks. Each block is accounted for its position in the hierarchy by referring the memory below it and the level at which it is placed. The highest level represents the largest memory, where all the data fits. Its usually refereed as DRAM. Memories existing on the same level can hold replicated and/or unique data. Memory blocks can be detailed with their total or datatype-wise segmented size. Below last-level memories (i.e. SRAM as shown in Fig. 3.2), a compute block with an array of processing elements can be instantiated. Finally, system-wide specifics can be set, such as clock rate, off-chip memory bandwidth and the costs of multi-level memory accesses. For validating our HW-model, we use a similar energy-cost model as the work in Eyeriss [11].

3.3.2 Scheduling Schemes

Inorder to obtain HW-estimates such as latency and energy consumption, the CNN model needs to be scheduled and mapped onto the HW-accelerator. The energy contribution of data movement cannot be disregarded for efficient execution of CNNs. For most cases, it constitutes the majority of the total power consumption to execute CNN models. CNNs consist of convolutional layers, which are commonly represented in a nested loop format, as expressed in Fig. 3.3. The for-loops shown present many reuse opportunities.

The main computation is at the core of the inner-most loop and many elements are accessed in multiple iterations of the higher loops. Specifically, reuse occurs when the indices of the parameters involved in the inner-most computation remain fixed for some loops before iterating in others. In HW, this translates to a single element being stored at a lower level memory for multiple iterations before being purged to make space for new data. For optimal reuse to occur, no single element should be read more than once from a higher level memory. This implies that during all the iterations that a single element is involved in, all the other elements that it is reused against also fit in the lower level memory. Practically, due to memory constraints, the parameters required by the entire nested-loop do not fit in the lowest-level of the memory hierarchy. A standard method of exploiting the entire hierarchy is to relax this constraint and split

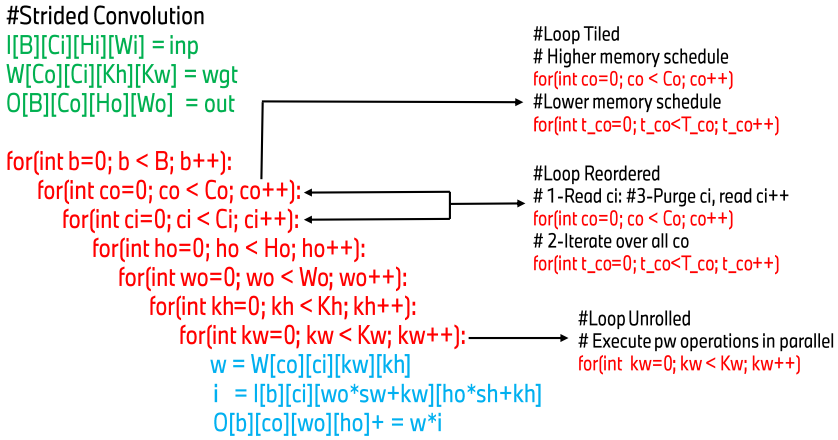


Figure 3.3: Nested for-loop representation of strided convolution.

the for-loops into shallower loops through a technique called *loop-tiling*. As shown in Fig. 3.2, the loop tiling strategy effectively decides which tiles $T \in \{T_{C_i}, T_{C_o}, T_{H_o}, T_{W_o}, T_{K_w}, T_{K_h}, T_B\}$ of CNN computation will take place in one round of communication with a lower level memory. Note that T_B is the tiling along the batch dimension when performing batch processing. The tiling strategy is selected based on the amount of on-chip buffer Buf, respecting the inequality in Eq. 3.2.

$$\underbrace{T_{H_i} \times T_{W_i} \times T_{C_i} \times T_B}_{\text{Input Tile}} + \underbrace{T_{H_o} \times T_{W_o} \times T_{C_o} \times T_B}_{\text{Output Tile}} + \underbrace{K_h \times K_w \times T_{C_i} \times T_{C_o}}_{\text{Weight Tile}} \leq Buf \quad (3.2)$$

To generate an output tile with spatial dimensions T_{W_o}, T_{H_o} with stride s and kernel k, an input tile with spatial dimensions T_{W_i}, T_{H_i} are required. The relation between input and output tiles is given in Eq. 3.3.

$$\begin{aligned} T_{W_i} &= (T_{W_o} - 1) \cdot s + K_w \\ T_{H_i} &= (T_{H_o} - 1) \cdot s + K_h \end{aligned} \quad (3.3)$$

The order of the loops can also be manipulated dynamically for each layer without affecting the algorithm through *loop-reordering*. As an example in Fig. 3.3, loop C_i can be swapped with loop C_o, allowing a single element c_i to reside longer on the lower-level memory while iterating over all possible elements $c_o \in C_o$. This can help extract improved reuse opportunities since the lower-level loops remain on the lower-level memories of the hardware architecture, thus closer to the compute units. This section considers three loop orders, namely Input Reuse Order (IRO), Weight Reuse Order (WRO), and Output Reuse Order (ORO) schemes inspired by the work in [109]. Switching dynamically between these three reuse schemes allows to schedule the entire CNN exploiting the reuse opportunities of different layers. As an example, layers with a very large kernel can benefit from ORO and WRO schedules, whereas layers with large feature maps (e.g. the first layers of most conventional CNN) will benefit the most with IRO schedule. This is refereed to as dynamic loop tiling. Finally, once a memory level is distributed spatially, further loops can be unrolled over the parallelism degree offered by the hardware architecture

through *loop-unrolling*. In Fig. 3.3, the kernel's elements can be assigned to spatially distributed processing elements, executing several K_w loop iterations in parallel during a single clock cycle.

Using the aforementioned strategies, the scheduler builds a matrix of possible tilings and loop orders. Each potential solution in the matrix is checked for *legality*, by assessing whether its transfer size breaches the memory restrictions at lower levels. The volumes $\mathcal{V}$ moved between the memory levels are calculated based on the memory occupation and the number of invocations required. To analytically reduce the size of the search space, a Computation-to-Communication (CTC) *hall-of-fame* is constructed after the evaluation of all legal solutions, which contains only a top percentage of the highest CTC loop tilings/orderings. Eq. 3.4 represents a CTC ratio formula, inspired by the work in [110]. γ represents a bandwidth-correction term to account for the burst-length of the memory transfers. The numerator is the number of operations/complexity of a particular workload. The denominator is the overall DRAM access along with bandwidth scaling for input, weights, and outputs for a particular workload.

$$\begin{gathered}\texttt{CTC} = \frac{2 \cdot H_o \cdot W_o \cdot K_h \cdot K_w \cdot C_o \cdot C_i}{\sum_{\texttt{dtype}} \gamma_{\texttt{dtype}} \cdot \mathcal{V}_{\texttt{dtype}}}, \\ \texttt{dtype} \in \{\texttt{ifmap}, \texttt{ofmap}, \texttt{psum}, \texttt{weight}\}\end{gathered} \tag{3.4}$$

The hall-of-fame solutions are passed on to the mapper. An analysis on the hall-of-fame size and the efficiency of the final schedule produced is presented in Sec. 3.3.4.

3.3.3 Mapping Methods

In-order to derive the HW-estimates, the scheduled tiles must be mapped into the PE-array. We refer mapping to dataflow between PE array and the last memory block in the hierarchy. In the case of Eyeriss accelerator [11], the last memory block in the hierarchy refers to SRAM. Many dataflow strategies have been explored in literature [11, 106]. Reuse opportunities in CNNs include convolutional, weight, input, and partial sum reuse. In this work, we focus on three dataflows, namely weight-stationary, output-stationary, and row-stationary. The weight-stationary dataflow unrolls the dimensions T_{C_i} and T_{C_o} as P_{C_i} and P_{C_o} across the spatially distributed computation array. Each PE holds complete kernels ($K_w \times K_h$) and corresponding input feature slices. Spatial reduction of partial sums can occur inside the PE, however, accumulation across input channels requires `psum` traversal over the spatial computation array. The output-stationary dataflow similarly unrolls P_{C_i} and P_{C_o}, however, the `psums` remain stationary in each PE, while input feature map pixels traverse the array and kernel pixels are updated once they are exhaustively used over the tile. Finally, row-stationary as introduced in [11] unrolls the T_{H_o} dimension horizontally across the array as P_{Ho}. Each K_h column of PEs is responsible for the complete computation of an entire row of the output W_o, while the neighboring set of K_h PEs computes the output row below that. Folding and replication techniques are applied to fit this unrolling method on the physical array dimensions. All three dataflows enable interleaving of channel computation within a single PE to maximize the use of the register files.

The mapper analytically determines the viability of a particular dataflow, based on the HW-details such as the interconnect dimensions, PE array size, and scratchpad configuration. The HW-modelling framework attempts to find a mapping that optimizes a given criterion (energy, latency, or a trade-off) while respecting the dataflow's restrictions. As presented in Fig. 3.3, unrolling a

subset of a loop's iterations as P spatially distributed computations, improves execution time. Assuming a filled pipeline, the latency of a layer φ_L can be estimated as the product of intertile and intratile latency as shown in Eq. 3.5. The intertile latency is computed based on the number of tiles required to transfer from off-chip memory to on-chip memory. Based on the PE unrolling procedure of the tiles available in the on-chip memory, the intratile latency is calculated. In Eq. 3.5, the kernel dimensions K_h and K_w are not tiled, as such granular tilings result in performance degradation for modern CNN models with small kernel sizes.

$$\begin{aligned}
\tilde{\varphi}_{L,\text{interTile}} &= \left\lceil \frac{C_o}{T_{C_o}} \right\rceil \cdot \left\lceil \frac{C_i}{T_{C_i}} \right\rceil \cdot \left\lceil \frac{H_o}{T_{H_o}} \right\rceil \cdot \left\lceil \frac{W_o}{T_{W_o}} \right\rceil \\
\tilde{\varphi}_{L,\text{intraTile}} &= \left\lceil \frac{T_{C_o}}{P_{C_o}} \right\rceil \cdot \left\lceil \frac{T_{C_i}}{P_{C_i}} \right\rceil \cdot \left\lceil \frac{T_{H_o}}{P_{H_o}} \right\rceil \cdot \left\lceil \frac{T_{W_o}}{P_{W_o}} \right\rceil \cdot \left\lceil \frac{T_{K_h}}{P_{K_h}} \right\rceil \cdot \left\lceil \frac{T_{K_w}}{P_{K_w}} \right\rceil \\
\tilde{\varphi}_{L,\text{total}} &= \tilde{\varphi}_{L,\text{interTile}} \times \tilde{\varphi}_{L,\text{intraTile}}
\end{aligned} \tag{3.5}$$

A particular mapping produces reuse factors for each datatype at different memory levels. We denote a reuse factor with $R^{\texttt{dtype}}_{\texttt{level}}$, where $\texttt{level} \in \{\texttt{Offchip}, \texttt{Onchip}, \texttt{Array}, \texttt{Registers}\}$. Reuse factors are dependent on tiling and unrolling strategies, as well as data interleaving [11], where a single computation element switches between multiple sets of the same datatype in order to extend the utilization of its registers. Once a legal mapping is found, the energy contributions of each datatype at each memory level can be computed. Eq. 3.6 shows an example of the energy consumption calculation at a particular memory level for a single datatype [11]. The read/write cost term $\mathcal{C}$ of a particular memory level can be set based on the fabrication technology or a relative normalized cost to other memory types in the hardware architecture. The energy estimates of all datatypes at all memory levels can be calculated similarly and summed up to obtain the total layer energy φ_E.

$$\begin{aligned}
&\varphi_{E,\text{Level}}(\texttt{dtype}) = \Big(\prod_{\texttt{off-chip}}^{\texttt{Level}} R^{\texttt{dtype}}_{\texttt{level}} \Big) \cdot \mathcal{C}_{\texttt{Level}} \\
&\forall\ \texttt{dtype} \in \{\texttt{ifmap}, \texttt{ofmap}, \texttt{psum}, \texttt{weight}\}
\end{aligned} \tag{3.6}$$

Finally, the mapping found is fed back to the scheduler, determining whether the tiling factors it provided were adequate. The possible combinations for legal schedules are evaluated and compared. These two optimization problems are codependent, as a tiling strategy that optimizes off-chip data movement may result in a mapping that under-utilizes the processing elements for a particular dataflow and vice versa.

3.3.4 Search Space Formulation for Efficient Schedule

For HW-estimates, creating a complete schedule implies choosing a fixed set of tiling factors $\{T_{C_i}, T_{C_o}, T_{H_o}, T_{W_o}\}$ and unrolling factors $\{P_{C_i}, P_{C_o}, P_{H_o}, P_{W_o}, P_{K_h}, P_{K_w}\}$. We restrict $T_{K_h} = K_h$ and $T_{K_w} = K_w$, and therefore omit them from the tiling factors set. Modern CNNs employ small kernel sizes, making it unreasonable to tile them during computation. Furthermore, tiling the kernel dimension generates a large amount of partial sums which can quickly become *parasitic* due to memory consumption and on-chip movement, if not collapsed into an output pixel. We

can define two subspaces in the scheduling search space: tiling space $\mathcal{T}$ and mapping space $\mathcal{P}$. Eq. 3.7 defines the size of the subspaces. Ord defines the reordering possibilities of the outer (off-chip memory) loops of the convolution. In this section, we consider three distinct orderings, IRO, OROs, and WROs, relating to inputs, outputs, or weights being kept longer on the on-chip memory respectively.

$$\begin{aligned} |\mathcal{T}| &= C_i \cdot C_o \cdot H_o \cdot W_o \cdot Ord \\ |\mathcal{P}_\tau| &= T_{C_i} \cdot T_{C_o} \cdot T_{H_o} \cdot T_{W_o} \cdot T_{K_h} \cdot T_{K_w} \quad \forall \tau \in \mathcal{T} \end{aligned} \tag{3.7}$$

$|\mathcal{T}|$ and $|\mathcal{P}_\tau|$ represent the cardinality of the tiling space and mapping space associated with a single tiling $\tau \in \mathcal{T}$ respectively. Therefore, the size of $\mathcal{P}_\tau$ is directly dependent on a single solution $\tau = \{T_{C_i}, T_{C_o}, T_{H_o}, T_{W_o}, Ord\} \in \mathcal{T}$. Restricting $\mathcal{T}$ directly reduces the number of total $\mathcal{P}_\tau$ searches necessary for finding a schedule. $\mathcal{T}$ may contain a single solution τ which results in a single mapping $\rho \in \mathcal{P}_\tau$, that is optimal for the overall schedule, in terms of latency, energy, or both. The trade-off in restricting the size of $\mathcal{T}$ is between schedule search speed and the optimality of the found schedule. To avoid evaluating drastically sub-optimal tilings, we analytically reduce the size of $\mathcal{T}$, and maintain solutions τ, which have a higher probability of producing efficient ρ mappings. The search for the optimal mapping ρ can also be expedited with further sampling techniques.

A straightforward approach to restricting the search space is to uniformly sample equidistant solutions in $\mathcal{T}$. We choose uniform sampling over random sampling to consider, at a minimum, a single candidate from each neighborhood in the search space. For fixed dimensions C_i, C_o, H_o, and W_o, the distance between two solutions depends on the sampling step. For small search spaces, the sampling step can be set to a small integer value. Therefore, for all experiments on the CIFAR-10 dataset, the sampling step was set to 2, effectively halving the number of tiles from each dimension. For the larger CNN models, better suited for the ImageNet dataset, integer steps are less effective. The size of a particular dimension C_i, C_o, H_o, and W_o, varies greatly between the first layer of the CNN towards the last. This makes the choice of a single integer step-size for all dimensions either grossly large to maintain simulation speed or small to maintain optimality at the cost of prohibitively increased search time. We use a ratio-based sampling to overcome this problem, where the step-size is a fixed fraction of the total dimension. This decouples the dimensions of the CNN from the number of τ mappings to be evaluated. We also allow each dimension to have its own ratio, providing more flexibility in finely searching smaller dimensions and coarsely searching larger ones. One more technique to aggressively reduce the search space is to find all the factors (divisors) of a particular dimension and declare those as the possible tiling factors. Since a factor will always give an integer number of tiles, this method usually leads to near-optimal results and is scalable to larger CNNs.

The CTC ratio metric is elaborated in Sec. 3.3.2 for choosing a reasonable tiling solution. Based on the intuition that a high CTC tiling solution τ could result in an efficient mapping, we analytically reduce the search space by creating a CTC Hall Of Fame (HOF). In the first step, we evaluate the CTC ratio for all $\tau \in \mathcal{T}$, which is a fast and parallelizable operation. A set percentage of $\mathcal{T}$ with the highest CTC ratios among all the solutions is entered in the HOF. Only members of the HOF have their respective $\mathcal{P}$ searched for mapping solutions.

We have a total number of full schedule evaluations equal to $\sum_{\tau}^{\mathcal{T}} |\mathcal{P}_{\tau}|$, which rapidly grows with $\mathcal{T}$, emphasizing the importance of good search space reduction techniques to maintain reasonable search time, without cutting out the optimal solutions in the space.

3.3.5 Search Space Exploration for Hardware Estimates

We evaluate the HW-model by exploring the HW estimations using different kinds of data reuse schemes. We improve the schedule search time for an efficient schedule and mapping solution by systematically reducing the search space of the proposed HW-model optimizer using a detailed ablation study in Sec. 3.3.4. An advantage of using HW-models over HIL-based methods is the flexibility of prototyping and testing multiple target architectures before committing to a final design for synthesis and fabrication. We report various HW configurations with different PE array sizes, memory costs, SRAM buffer and register sizes in Tab. 3.1. Column 3 indicates the data access cost from higher memory levels (DRAM) to lower levels (RF) relative to one MAC operation. This section uses the HW-Flow-Val model with 16-bit word length to explore various dataflows and compare the modeling estimates with Eyeriss [11] accelerator.

Architecture Spec / Hardware Model	**PE Array**	**Memory Cost** DRAM, SRAM, Array, RF	**SRAM size** <KB>	**Register Words** filter, ifmap, psums
HW-Flow - Val	16×16	200, 6, 2, 1	128	192, 12, 16
Timeloop [105]	16×16	200, 7.41, 0, 1	128	192, 12, 16
Eyeriss-like-168 PE (RS)	12×14	200, 6, 2, 1	128	224, 12, 14
Eyeriss-like-256 PE (RS)	16×16	200, 13.84, 2, 1	256	224, 12, 14
Eyeriss-like-1024 PE (RS)	32×32	200, 155.35, 2,1	3072	224, 12, 14
Eyeriss-like-Deeplab (RS)	32×32	200, 155.35, 2,1	3072	224, 37, 16

Table 3.1: Hardware configurations used for experiments and validation. RS refers to row-stationary dataflow.

Tab. 3.2 shows the search time needed for different analytical search strategies to produce a schedule. The quality of the search method can be measured by its corresponding mapping goal. In the first three rows, we produce various schedules targeting different optimization goals. We observe lower energy consumption or latency, when the search goals are changed accordingly. Three different $\mathcal{T}$ sampling rates (5%, 10%, 20%) with 1% CTC-HOF are explored. We observe that the normalized energy increases as we limit the exploration by increasing the sampling rate. Based on the trade-off between evaluation speed (see Tab. 3.2), we sample the tile space with 10% for ImageNet experiments to obtain HW estimates for the pruning process. This results in an overall shorter search time (up to $\times 2.5$) at the cost of degradation in mapping optimization goal. We also highlight the tile sampling method by computing divisors in Tab. 3.2. We observe that the divisors-based sampling method produces a schedule with the lowest energy consumption. However, this method could produce sub-optimal results in case of channel pruning when the

agent finds prime-number of filters for a particular layer. For CIFAR-10 experiments, we use smaller, integer steps of 2, as these CNNs have a small scheduling search space.

Search Strategy	**Search Time** [s]	**DRAM Energy** [$\times 10^9$]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^6$cycles]
10% $\mathcal{T}$, 1% HOF*	9.87	10.76	**83.25**	379
10% $\mathcal{T}$, 1% HOF**	10.13	155.82	257.49	**65**
10% $\mathcal{T}$, 1% HOF	10.23	11.06	91.89	67
5% $\mathcal{T}$, 1% HOF	25.59	12.10	83.96	60
10% $\mathcal{T}$, 1% HOF	10.23	11.06	91.89	67
20% $\mathcal{T}$, 1% HOF	**7.23**	10.47	104.61	118
divisors $\mathcal{T}$, 1% HOF	12.86	14.60	**71.61**	65
5% $\mathcal{T}$, 100% HOF	215.42	12.10	83.96	**60**
5% $\mathcal{T}$, 1% HOF	25.59	12.10	83.96	60
10% $\mathcal{T}$, 100% HOF	23.03	11.06	91.89	67
10% $\mathcal{T}$, 10% HOF	15.24	11.06	91.89	67
10% $\mathcal{T}$, 1% HOF	**10.76**	11.06	91.89	67
divisors $\mathcal{T}$, 100% HOF	51.47	9.67	**72.44**	65
divisors $\mathcal{T}$, 1% HOF	12.86	14.60	71.61	65

All simulations were run with 24 threads on an Intel Xeon E5-2698 Process

Mapping goal : *energy, **latency, ***dram access

Table 3.2: Schedule search duration and optimality under different search space reduction strategies for AlexNet on Eyeriss-like-256. All schedules optimize for a trade-off between latency and energy, unless marked otherwise. Similar to Eyeriss [11], we normlize DRAM energy (column 3) and total energy (column 4) to the cost of one MAC operation.

The sensitivity analysis of the CTC-HOF tile space reduction technique is shown in Table 3.2. The results show that the (10% $\mathcal{T}$, 1% HOF) strategy is very effective, providing a speedup of $2.14\times$ compared to (10% $\mathcal{T}$, 100% HOF) schedule without sacrificing the optimality of the schedule. Combining these methods is critical in maintaining a reasonable exploration time for multiple pruning experiments. We finally use the sampling strategy (10% $\mathcal{T}$, 1% HOF) with an overall search time reduction of $20\times$ compared to the search strategy (5% $\mathcal{T}$, 100% HOF). Once a HW-CNN pair is found, the HW-optimizer can run with a more exhaustive search strategy and provide an improved schedule for the final deployment stage.

We demonstrate the HW estimates and compare them with ideal performance at peak compute utilization and unbounded on-chip memory size. We obtain the ideal estimations for DRAM access counts (indicated in red) by simply summing-up the layer-wise transfer volumes of Ifmaps, Ofmaps and weights. These ideal assumptions in the initial phases of development allow the designer to choose the CNN topologies that suit the application under consideration. We perform the measurements for four CNN models, namely AlexNet, VGG-16, ResNet18, and ResNet152. We observe that as the on-chip buffer size increases, larger tiles of input, weights, and outputs can be stored on the buffer, thereby decreasing the number of DRAM Accesses. We observe that the HW-model estimations for all the CNN architectures could meet the ideal estimations

at higher buffer sizes ($\geq$ 512KB). We notice that the HW-model estimations produce a higher number of DRAM accesses, as the schedule must consider more complex design details and constraints at this level. The AlexNet architecture achieves the least throughput (Ops/Cycle) among other architectures, as a considerable number of operations and parameters are assigned to fully-connected layers. The throughput produced at the HW-model considers the dataflow and the underlying unrolling scheme and therefore achieves closer estimates compared to real target deployment. We observe that the throughput saturates at 128KB buffer size for HW estimations of different CNN architectures. A slight decrease in throughput happens for AlexNet and ResNet18 at scheduling for on-chip memories larger than 128KB. This is due to the schedule simultaneously optimizing for inference energy (not shown in the figure) as the on-chip memory grows.

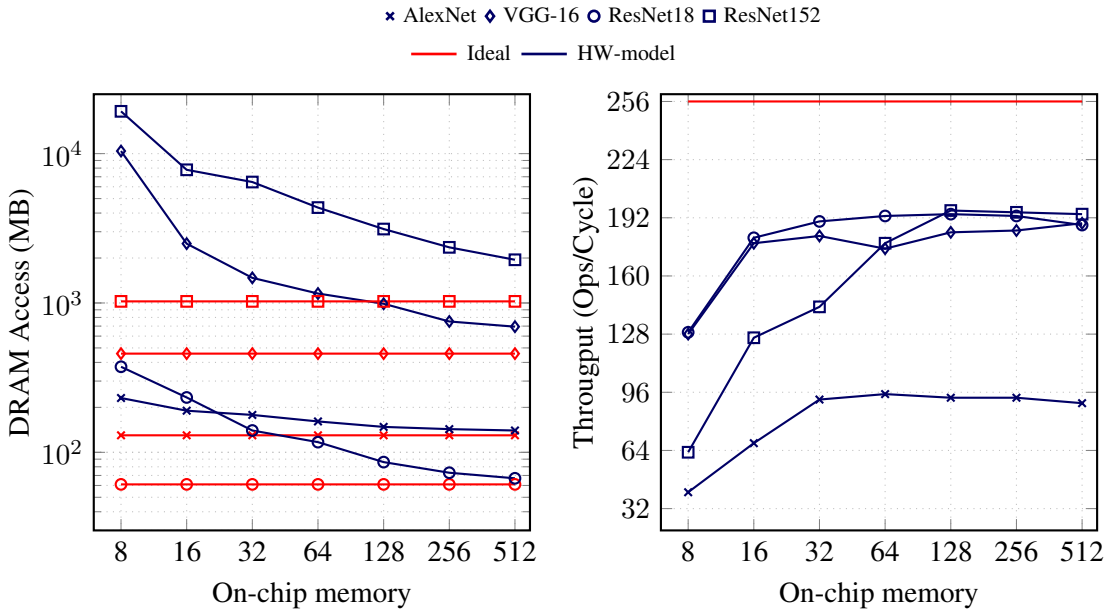


Figure 3.4: Analysis of DRAM Access and Throughput on varying the on-chip buffer size and different CNN architectures.

The row-stationary (RS), weight-stationary (WS) and output-stationary (OS) dataflows are used to explore the HW estimates. The mapper searches for a trade-off between normalized energy and latency while respecting each dataflow's unrolling rules, the HW's memory and compute capacity checks. We obtain the HW estimates for AlexNet and ResNet18 models for different on-chip buffer sizes. We observe that the RS dataflow is the most energy efficient at all the buffer sizes. This is due to RS maximizing the data reuse at the register-level, for all the datatypes [11]. OS and WS dataflows maximize the compute utilization of PE arrays, albeit with higher normalized energy requirements. The WS dataflow achieves higher throughput using larger buffer sizes ($\geq 32KB$) for ResNet18.

To validate the correctness of estimates of the HW-model and mapping components, we compare its estimates with the Eyeriss architecture [11] and its Timeloop model [105] for AlexNet [31] inference, which has diversified kernel sizes, strides and input/output dimensions.

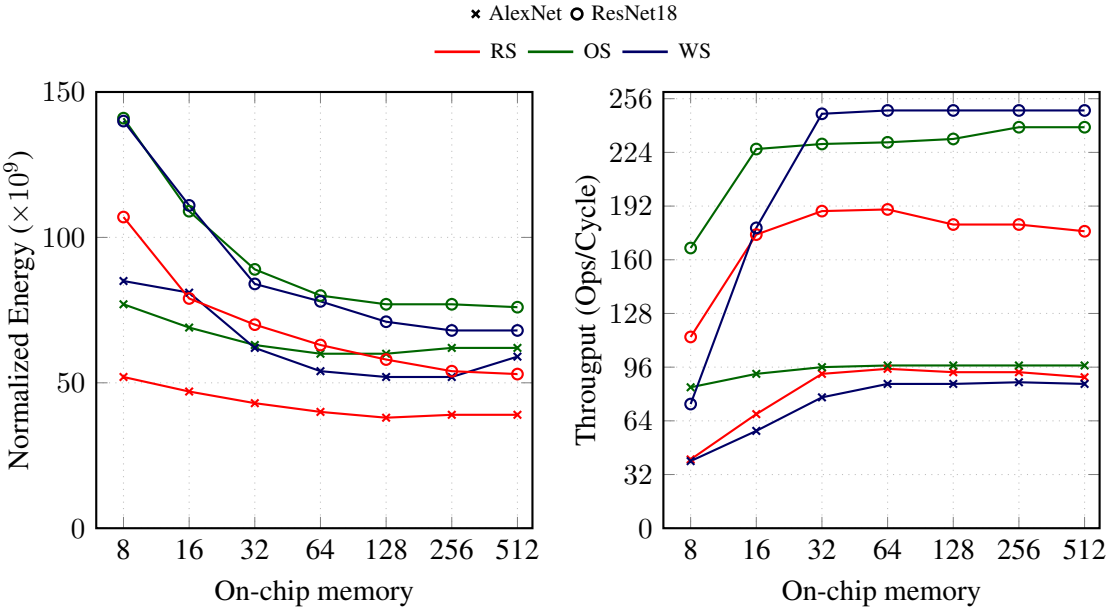


Figure 3.5: Influence of dataflows selection on normalized energy and throughput of the HW accelerator.

Fig. 3.6 shows a breakdown of normalized energy contributions of each datatype at each memory level for the convolutional layers. We observe that our HW-model tracks the original Eyeriss results similar to Timeloop in Fig. 3.6. A slight offset is observed, which can be attributed to small differences in the energy references used during the search. The overlapping line charts show the latency estimates of both frameworks.

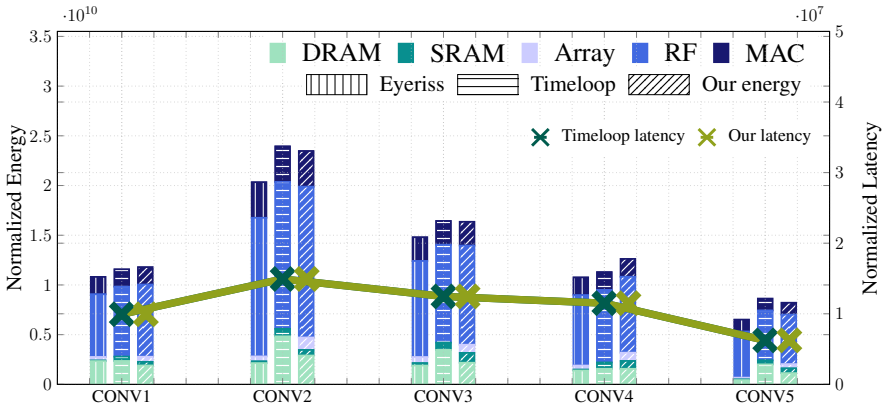


Figure 3.6: Validation with the Eyeriss accelerator [11] and Timeloop [105]. Note: [11] does not report layerwise latencies.

3.3.6 Discussion

In this section, we formulated a HW-model of a flexible CNN accelerator and determined an efficient scheduling scheme to generate HW-estimates. The HW-model requires specification of compute and memory hierarchy. The off-chip communication scheduling requires exploration of loop-parameters which influence different latency and energy contributions. In particular, we have explored loop-tiling and loop reordering of the convolutional loop applied to all the layers in the CNNs. These optimizations generate efficient communication and computation schedules for different variants of the Eyeriss [11] accelerator. In-order to compute detailed HW-metrics, the scheduled tiles are mapped onto the PE array using different different dataflows such as RS, OS and WS. We formulate an energy cost model to compute the contributions of energy consumption from different memory sources. We compute the number of processing passes to calculate the latency of the accelerator. The aim of this chapter is to leverage the HW-estimates during the compression loop to produce HW-aware CNNs.

To leverage the HW-estimates during the compression loop, it is important to reduce the schedule search time to quickly provide feedback. HW-models in literature such as Timeloop [105] only investigate uniform or random sampling based search. Thus, we realize different sampling opportunities at various levels in HW-model to produce execution metrics. We explore the effectiveness of sampling the tiling, interleaving space with respect to the schedule search time and the final HW-metrics. We further understand the appropriate search heuristics based on the CNN model and its workloads. Based on the study conducted in Tab. 3.2, we reduce the schedule search time by 20$\times$ compared to uniform sampling procedure. The discussed model and reduced search time in this section plays an important role to inject HW-awareness for the automated RL-based pruning in Sec. 3.4. Furthermore, the usage of HW-model based estimates from this section is not only limited to the procedure detailed in Sec. 3.4 but also applicable in Sec. 4.3, 4.4 and 4.5.

3.4 Hardware Model based Automated Pruning

Without loss of generality, in an L-layer CNN, the convolutional layer $l \in \{1, ..., L\}$ receives an Ifmap $A^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$, where H_i, W_i and C_i indicate the spatial height, width and input channels respectively. A^0 is the input image I to the CNN, as shown in Fig. 3.7 (bottom). The weights $W \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o}$ are the trainable parameters of the individual layers, here K_h, K_w and C_o are the kernel dimensions and the number of output channels (filters) respectively. The input A^{l-1} is convolved with the weights W^l, where the kernels are moved over the input with stride s. In detail, the task of the agent π is to prune the input channels C_i of the environment by zeroizing the binary pruning mask $\mathcal{A}^l = \{0, 1\}^{1 \times 1 \times C_i \times 1}$. To select the most salient channels, the Hadamard product $\odot$ is applied, giving a sparse representation $\tilde{W}^l \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o} = W^l \odot \mathcal{A}^l$. Referring to Fig. 3.7, zeroizing an input channel in the l^{nth} layer will zero out the corresponding output feature map from the l^{n-1} layer. Consequently the kernels of all filters in the l^{n-1} layer are also zeroed-out. Channel-wise pruning removes several weights from the CNN at once, causing a significant loss in accuracy. To mitigate this negative effect and guarantee an energy and latency efficient compression, the learning-based agent π has to learn good actions $\mathcal{A}^l$. In this section, we complement well-established proxies, such as `OPs` and `Params` count, with

more elaborate HW-model based estimates, which are conducive to finding efficient CNNs for embedded applications.

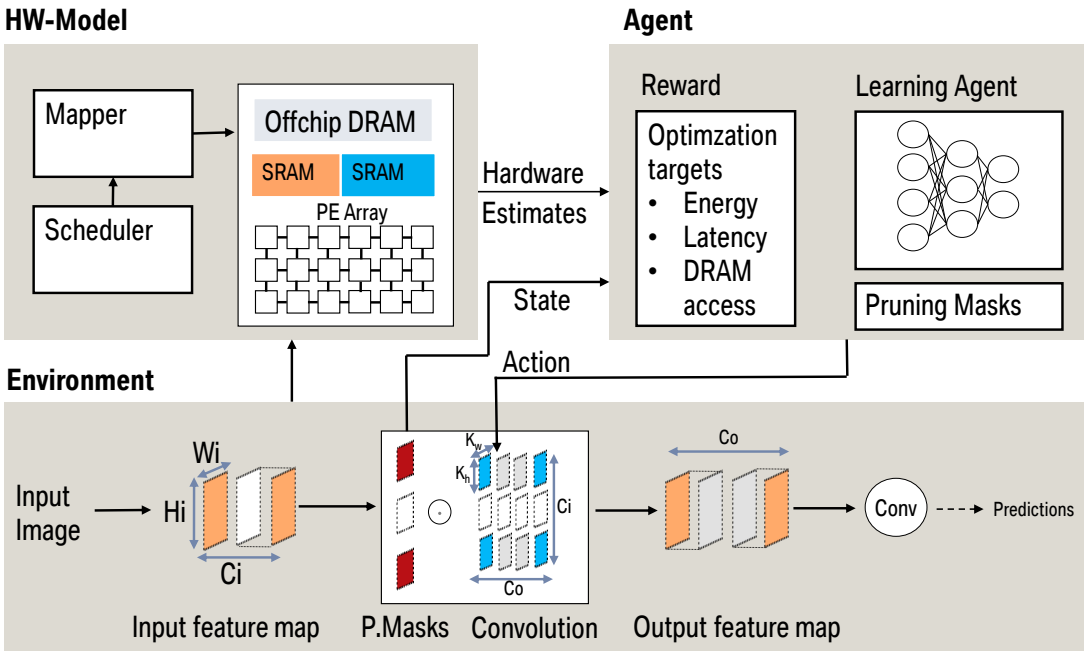


Figure 3.7: Overview of automated pruning using HW-model. The CNN environment (bottom) is pruned by a RL agent (right).

3.4.1 Hardware Model based Channel Pruning using RL

The DDPG agent's architecture, including the actor and the critic, is adopted from He et al. [5]. The agent is augmented with key rewards and state information, allowing it to understand the influence of its pruning actions on the inference hardware with respect to the energy estimates φ_E and the latency estimates φ_L as elaborated in Sec. 3.3. The newly adapted state $\mathcal{S}$ is composed of the following layer information of the environment's f: the index of the layer l, stride s and the layer dimensions after pruning $\tilde{C}_o, \tilde{C}_i, W_i, H_i$. It should be noted that the estimates φ obtained from the HW-models are considered to be part of the state $\mathcal{S}$, where $\varphi = [\varphi^0, ..., \varphi^l, ..., \varphi^L]$ ensembles either layer-wise energy estimates φ_E or latency estimates φ_L. As expressed in Eq. 3.8, the action $\mathcal{A}^{l-1}$ is applied for composing the input state $\mathcal{S}$ of the agent.

$$\mathcal{S}^l =< l, s, \tilde{C}_o, \tilde{C}_i, W_i, H, \varphi^l, \sum_{i=0}^{l-1} \varphi^i, \sum_{j=l+1}^{L} \varphi^j, \mathcal{A}^{l-1} > \tag{3.8}$$

In this section, the agent is trained using one of the two reward functions, either the estimate *balanced* or estimate *constrained* reward function, as defined in Eq. 3.9. The *balanced* reward of Eq. 3.9 is inspired by [21]. When the HW-constraints are unknown in the early stages of the

design, the reward function can be formulated to achieve at least a target accuracy ψ^* before optimizing the performance estimate term. A trade-off between the accuracy term $(1-(\psi^*-\psi)/b)$ and the estimate term $\log(\varphi^*/\varphi)$ after each pruning action is the goal of the estimate *balanced* reward function. Parameter b influences the turning point between a negative and positive reward $\mathcal{R}$, encouraging the agent to improve the accuracy when the difference between ψ^* and ψ is larger than b. When this condition is met, the agent starts to optimize the trade-off between accuracy and hardware estimates. This reward can also be extended to optimize multiple KPI's by appending several logarithmic terms. The estimate *constrained* compression improves the reward $\mathcal{R}$ by maintaining higher prediction accuracy ψ after each pruning action. This encourages the agent to prune the CNN model while minimizing the accuracy degradation when the HW-constraints are strictly stipulated.

$$\mathcal{R} = \begin{cases} (1 - \frac{\psi^* - \psi}{b}) \cdot \log\left(\frac{\varphi^*}{\varphi}\right), & \text{if } \textit{balanced} \\ \psi, & \text{otherwise } \textit{constrained} \end{cases} . \tag{3.9}$$

The estimate term in the reward represents the benefits obtained from pruning with respect to the HW-model μ, giving the estimate φ^* of the unpruned base model and φ after each episode of the agent.

3.4.2 Pruning with Proxy Metrics and Hardware Estimates

The experiments in Tab. 3.3 serve as an example on how the HW-estimates can improve the optimal task-to-resource mapping. In Tab. 3.3, we evaluate pruning configurations for ResNet56 on CIFAR-10 dataset and ResNet50 on ImageNet dataset.

Prune configuration (< `constraint` >)	**Acc** [%]	**PR** [%]	**Memory** [MB]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^3$cycles]
ResNet56					
Baseline (not pruned)	93.59	-	1.69	3.76	2350
-50% Ops *;	93.03	50.00	1.21	2.08	1219
-50% Energy*;	93.14	54.00	1.11	1.88	1159
-50% Latency*;	93.24	50.89	1.15	2.05	1176
ResNet50					
Baseline (not pruned)	76.06	-	51.00	361.12	206873
-50% Ops *;	73.25	50.00	22.67	178.55	103968
-50% Energy*;	73.69	49.82	24.12	180.91	104411
-50% Latency*;	74.35	49.68	25.62	180.93	103576

(matched constraint)

Table 3.3: Constrained and balanced pruning configurations on ResNet variants, compared to other works in literature. HW estimates measured on Eyeriss-256.

We search for pruning configurations to reduce 50% of the `OPs` , 50% of the energy and 50% of the latency and report the estimates in the subsequent rows. We observe that the target constraints are met for all the pruning experiments. Consequently, the HW-aware pruning experiments preserves the network's accuracy equivalent to the `OPs` based pruning configurations, while achieving optimal constrained estimates. For ResNet56, the energy and latency constrained

solutions produce 0.11 pp and 0.21 pp better prediction accuracy compared to the work in `OPs` constrained pruning configurations with improved HW estimates. For ResNet50, the energy and latency constrained solutions produce 0.44 pp and 1.10 pp better prediction accuracy compared to the work in `OPs` constrained pruning configurations.

3.4.3 Pruning for different hardware dimensions

To emphaisze the importance of HW-models, we consider the three candidate Eyeriss-like hardware accelerators with 168, 256, and 1024 PEs. In this experiment, the agent performs pruning based on the two types of reward functions proposed in Eq. 3.9, namely estimate *constrained* and *balanced*.

Estimate Constrained: The agent is tasked with pruning the ResNet56 model trained on CIFAR-10 such that it meets a given fixed constraint while minimizing the accuracy degradation of the compressed network. The constraint is set to 50% energy or latency reduction relative to the baseline leader, *i.e*. the accelerator which performs the best for the target metric. The results in Tab. 3.4 show several interesting trends. We observe that the 168 PE variant is the baseline leader for energy-constrained pruning and the 1024 PE accelerator as a baseline leader for latency constrained pruning. With 1024 PEs, there is an ample capacity to improve latency, requiring a lower pruning rate to meet the application constraint. Conversely, the CNN can be pruned more effectively for 168 and 256 PEs when considering an energy-constrained application. For both cases, choosing the correct hardware platform results in a pruned network with higher accuracy. These critical observations can facilitate the choice of a suitable hardware for a given application constraint.

Prune configuration (< `constraint` >;< `level` >;< `hw_model` >)	**Acc** [%]	**PR** [%]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^3$cycles]
Baseline (not pruned); Fine; 168 PE - RS	93.59	-	3.72	3377
Baseline (not pruned); Fine; 256 PE - RS	93.59	-	3.76	2350
Baseline (not pruned); Fine; 1024 PE - RS	93.59	-	5.52	588
Target Energy (-50%)**; Fine; 168 PE - RS*	92.63	58.16	1.85	1644
Target Energy (-50%)**; Fine; 256 PE - RS	93.14	54.00	1.88	1159
Target Energy (-66%)**; Fine; 1024 PE - RS	91.09	75.22	1.88	170
Target Latency (-92%)**; Fine; 168 PE - RS	86.89	93.14	0.40	269
Target Latency (-87%)**; Fine; 256 PE - RS	89.66	87.94	0.59	306
Target Latency (-50%)**; Fine; 1024 PE - RS*	92.92	52.68	3.07	294

*: Baseline leader — **: reduction required to meet constraint — (violated constraint) (matched constraint)

Table 3.4: Pruning ResNet56 on CIFAR-10 using estimate *constrained* reward $\mathcal{R}$ on Eyeriss-like accelerators.

Estimate Balanced: As detailed in Sec. 3.4.1 and Eq. 3.9, the balanced estimate reward encourages the agent to maintain the target accuracy ψ^*, while minimizing the estimates φ. Here, ψ^* and b are set to 0.5, 0.125 respectively. From Tab. 3.5, we observe that all the configurations optimized for energy and latency undergo minimal degradation in prediction accuracy with

different latency and energy estimates. The Eyeriss-like 168 PE configuration achieves the best energy, whereas 1024 PEs achieves the best latency.

Prune configuration (< `reward` >;< `level` >;< `hw_model` >)	**Acc** [%]	**PR** [%]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^3$cycles]
Baseline (not pruned); Fine; 168 PE - RS	93.59	-	3.72	3377
Baseline (not pruned); Fine; 256 PE - RS	93.59	-	3.76	2350
Baseline (not pruned); Fine; 1024 PE - RS	93.59	-	5.52	588
Energy balanced; Fine; 168 PE - RS	91.94	69.14	1.41	1309
Energy balanced; Fine; 256 PE - RS	92.56	62.22	1.61	913
Energy balanced; Fine; 1024 PE - RS	92.30	62.09	2.69	238
Latency balanced; Fine; 168 PE - RS	92.64	56.50	1.94	1658
Latency balanced; Fine; 256 PE - RS	92.58	59.75	1.69	975
Latency balanced; Fine; 1024 PE - RS	92.97	57.14	3.18	276

Table 3.5: Pruning ResNet56 on CIFAR-10 using the estimate balanced reward function on Eyeriss-like accelerators.

3.4.4 Pruning for different dataflows

The following experiment is performed to evaluate the relationship between effective pruning configuration with different dataflows. We compare the target hardware model, with 256 PEs, against two variants with identical specification, except for their dataflows. Here, the three dataflows, weight-stationary (WS), output-stationary (OS), and row-stationary (RS), described in Sec. 3.3.3, are compared in their potential for improved execution of pruned CNNs.

Prune configuration (< `constraint` >;< `level` >;< `hw_model` >)	**Acc** [%]	**PR** [%]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^3$cycles]
Baseline (not pruned); Fine; 256 PE - OS	93.59	-	5.87	1960
Baseline (not pruned); Fine; 256 PE - WS	93.59	-	5.77	1991
Baseline (not pruned); Fine; 256 PE - RS	93.59	-	3.76	2350
Target Energy (-68%); Fine; 256 PE - OS	91.84	72.05	1.88	584
Target Energy (-68%); Fine; 256 PE - WS	90.06	84.53	1.75	1308
Target Energy (-50%); Fine; 256 PE - RS *	93.14	54.00	1.88	1159
Target Latency (-50%); Fine; 256 PE - OS *	92.91	52.11	3.06	981
Target Latency (-51%); Fine; 256 PE - WS	84.17	96.20	0.71	1612
Target Latency (-58%); Fine; 256 PE - RS	92.36	61.05	1.72	984

*: Baseline leader — (violated constraint) (matched constraint)

Table 3.6: Constraining dataflows relative to 50% of the baseline leader (RS for energy and OS for latency).

The baseline estimates of the unpruned network show the energy and latency variation caused by dataflows (Tab. 3.6). All three dataflows present unique non-dominated solutions for baseline

energy and latency. Similar to the estimate constrained experiment, we set the constraint with respect to the baseline leader dataflow. RS results in the lowest baseline energy, whereas the OS has the lowest baseline latency. We can observe that the agent obtains minimum accuracy degradation for RS when constraining for energy. When constraining for latency, the agent achieves better accuracy for OS and RS. WS demands higher pruning rate when constraining both energy and latency thereby resulting in lower accuracy (marked as red in Tab. 3.6). Thus, we can conclude that the row stationary dataflow is an optimal mapping scheme to achieve efficient energy and latency.

3.4.5 Pruning DeepLabV3+ for Semantic Segmentation

Using the HW-model estimations, we prune DeepLabv3 [35] (using ResNet18 backbone) on the CityScapes [2] dataset. For the DeepLab-based CNN, the bottleneck layers consist of two residual blocks with a dilation rate of 2 and an Atrous Spatial Pyramid Pooling (ASPP) block with dilation rates $\{1, 8, 12, 18\}$. To obtain HW-estimates for dilated convolutional layers, we adapt the row-stationary dataflow. The rows of PEs responsible for the dilated parts of the kernel can either be clock-gated or removed from the logical mapping. This implies that the diagonal reuse of input pixels across the spatial array is disrupted. This phenomenon is equivalent to a regular convolution with a large stride, where not every row of the input feature is shared directly with the diagonal neighbor PE [11]. Nevertheless, a non-direct neighbor PE may still reuse the Ifmap row. In this case, the potential to reuse an Ifmap row at the PE array-level depends on the degree of unrolling P_{Ho}, the dilation rate, and the stride. We use an Eyeriss-like architecture with a large PE array to perform inference of the DeepLabv3 model. In Tab. 3.7, we highlight that the DeepLabv3 cannot be scheduled on the standard Eyeriss architecture [11] (Eyeriss-168). This is due to the Ifmap register files being dimensioned to hold at-most 12 pixels at a time (see Tab. 3.1), which is a decision made by the designers in [11] to support the largest kernel size row in AlexNet (11 pixels). The dilated convolution layers in DeepLabv3, can have up to 36 pixel rows at a time, for a 3×3 kernel with a dilation rate of 18. Increasing the PE array dimensions would not resolve this issue, as it is inherent to the pipeline and dataflow constraints of the Eyeriss-like architecture. We increase the Ifmap register sizes to 37 pixels per PE (i.e. 36 + 1) to make all layers schedulable on the accelerator and obtain baseline estimates.

Prune configuration (< `reward` >;< `level` >;< `hw_model` >)	**mIOU** [%]	**PR** [%]	**Memory** [MB]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^6$cycles]
Baseline (not pruned); Eyeriss-like 168 PE	69.68	-	33.26	NS	NS
Baseline (not pruned); Eyeriss-like 1024 PE	69.68	-	33.26	NS	NS
Baseline (not pruned); Eyeriss-like-Deeplab	69.68	-	33.26	1541	267.4
Ops Constrained (Ours); Eyeriss-like-Deeplab	69.69	50.00	25.48	954	174.9
Energy Constrained (Ours);Eyeriss-like-Deeplab	69.88	51.90	29.05	820	161.5
Latency Constrained(Ours); Eyeriss-like-Deeplab	69.79	60.36	16.87	677	119.6

(NS: Not Schedulable) (matched constraint)

Table 3.7: Pruning DeepLabv3 on the CityScapes dataset.

We constrain the number of operations, energy, and latency during the pruning process to 50% as shown in Tab. 3.7. There is no degradation in the mIOU (mean intersection over union)

evaluation metric for different pruning constraints. We could derive that the unpruned DeepLabv3 model is over-parameterized for the CityScapes dataset. We observe that a higher pruning rate is required to constrain latency to 50%. We highlight the pruned models' effectiveness by demonstrating the semantic predictions on three sample images of the CityScapes dataset in Fig. 3.8. We observe that the pruned models could produce better predictions (terrain in column 1, bikers in column 2, motorcycle and terrain in column 3) due to their higher generalization capability. By analyzing the layer-wise pruning ratios for different target constraints, we observe that the agent heavily prunes the ASPP and decoder blocks. For energy-constrained pruning, the agent only finds redundant operations in the decoder blocks.

3.4.6 Discussion

In this section, we propose a HW-model based automated pruning approach using RL-agent. We formulate the search space using the channel pruning to easily obtain HW-benefits without specialized implementations. We augmented the HW-estimates obtained from the models as discussed in Sec. 3.3 to the state input of the RL-agent. We explored two kinds of rewards, namely the constrained and balanced formulations in Eq. 3.9. We replace the proxy estimates such as the `OPs` and `Params` to HW-estimates with energy and latency in the state and reward formulations.

Using the RL-agent, we conduct detailed experiments to highlight the benefits of HW-aware compression from Sec. 3.4.2 to Sec. 3.4.5. We derive *energy-aware* and *latency-aware* CNN models for CIFAR-10 and ImageNet datasets in Tab. 3.3. By incorporating HW-estimates in the compression loop, we are able to obtain lower energy and latency with minimal change in the prediction accuracy. We explore various HW-configurations with different PE and on-chip memory sizes in Tab. 3.4. We show that the pruning rate varies based on the HW configuration to satisfy a given target constraint. We also access our pruning framework on different dataflow configurations for a given HW-constraint. We observe that the RS, OS dataflows require lower pruning rates to satisfy energy and latency constraints in Tab. 3.6 respectively. Finally, we observe the effectiveness of our HW-model based pruning on the task of semantic segmentation. Using HW-model exploration, we dimension our accelerator such that all the layers for the DeepLab model have valid schedules. We finally derive *energy* and *latency* aware Deeplab architectures using our pruning framework. Throughout the experiments, we highlight the need to provide HW-estimates during the compression loop and realize HW-friendly CNN models which execute efficiently on the target deployment platform. We also establish a similar HW-aware compression pipelines in Sec. 3.5 and Sec.4.5.

3.5 HIL based Pruning for LiDAR Processing

Over the last decade, several CNN based architectures have been proposed for object detection. While object detection CNN architectures were originally proposed for image data [111, 48, 112, 113], there have been several advances in investigating their applicability to LiDAR point clouds [114, 115, 116, 117, 118, 119, 120, 121, 24]. While images provide dense 2D measurements, 3D localization is challenging due to loss of the depth dimension during image creation. LiDAR point clouds are an intrinsically 3D data that capture rich geometric information

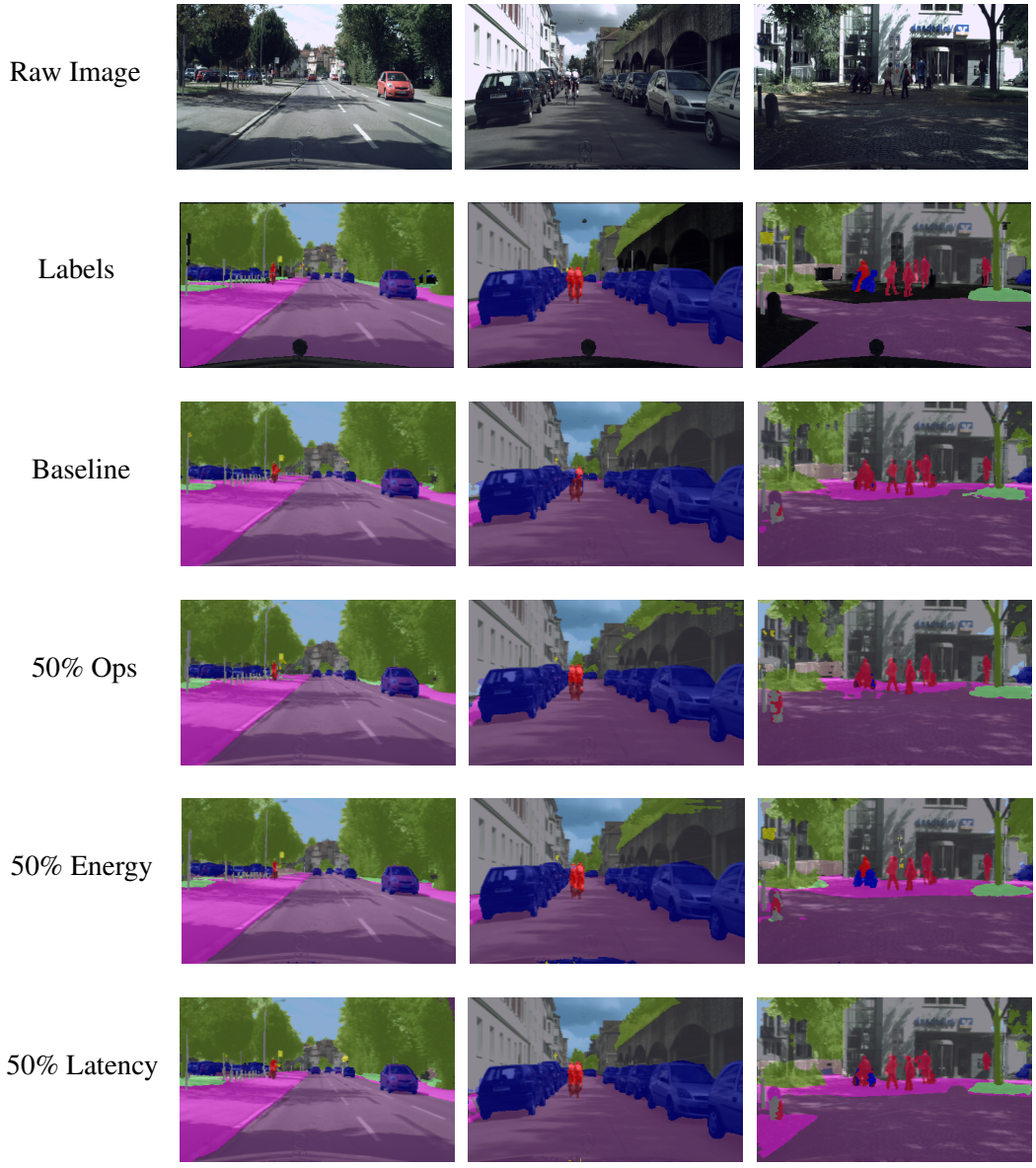


Figure 3.8: Qualitative results for pruned models on different scenarios in the CityScapes dataset. Black regions are unlabeled in the original dataset.

such as shape and scale. Due to the geometric and sparse nature of point clouds, they require extra processing and transformations to make use of conventional 3D object detection pipelines. The models used are typically compute- and memory-intensive with strict latency constraints, hindering their deployment in *resource-constrained* environments. The automation of CNN pruning for hardware deployment allows the exploration of the pruning configurations in terms of hardware estimates. Inorder to obtain HW-friendly CNN configurations on general purpose

accelerators such as GPUs, it is important to incorporate latency measurements in the RL-based pruning approach. This can be performed using HIL approach during the optimization phase as discussed in this section.

3.5.1 LiDAR based 3D object detection

Current approaches for LiDAR based 3D object detection are either raw point-cloud based [118, 122, 123, 124], discretization based [117, 115, 125, 126], or fusion based [117, 116]. These approaches differ mainly in their point-cloud processing and model architecture, offering several design choices with a trade-off between accuracy and efficiency. Point cloud is an unordered set of points $P = \{(x, y, z, r)_i\}_{i=0}^{N}$ where N is the total number of points, r is the reflectance and x, y and z are the spatial coordinates in the 3D space. This is why LiDAR-based approaches differ from image-based methods by the requirement of a point cloud encoder which is responsible of transforming the sparse point cloud $P = \{(x, y, z, r)_i\}_{i=0}^{N}$ to a dense representation $I \in \mathbb{R}^{W \times H \times C}$.

A common approach that was already mentioned in the context of fusion-based approaches is the **projection of statistical handcrafted features** extracted from the point cloud on a 2D representation. ComplexYolo [121] is the adaption of the well-known 2D YOLO detection method for the 3D application use case. ComplexYolo projects the point cloud using the Bird's Eye View (BEV) plane projection. Each element in this representation has 3 channels describing the maximal height, intensity and density of all points that project to that grid cell. The backbone and detection head are adapted versions from the 2D-based YOLO network giving the ability to regress 3D bounding boxes from projected point clouds. The main limitation of LiDAR-based approaches with handcrafted projected representations is the considerable information loss mainly on the vertical axis resulting from dropping many data points (due to projection of point cloud on the ground plane) and also from handcrafted feature space design.

A different approach is the generation of **voxel-based volumetric** descriptions of the point cloud that preserve the 3D nature of the input feature space. VoxelNet [115] uses a volumetric representation of the point cloud data. However, instead of using hand-crafted statistical features for each voxel, it applies a PointNet on all points within each non-empty voxel to extract a voxel-wise learned feature representation. The obtained volumetric feature maps are processed using 3D convolutions followed by a region proposal network to predict the final detections.

In order to speed up the feature extraction process even more, PointPillars [24] replaces the volumetric voxel representation with a **pillar representation** by collapsing the height dimension already at the initial stage. Similarly, pillar-wise feature descriptions are extracted directly from raw points within each non-empty pillar. The obtained representation is a 2D image-like data structure. The main advantage of this representation is to drop the need for 3D convolutions which are costly in terms of runtime. We compare ComplexYolo [121], VoxelNet [115] and PointPillars [24] based CNN architectures in Tab. 3.8.

3.5.2 End-End pruning pipeline

The RL-based pruning approach discussed in Sec. 3.4, allows reduction of model complexity either based on proxy estimations or HW-estimates. However, the compression of the resulting

Model	Encoder		Detection Head	Advantage/limitation
	type	Method		
Complex-YOLO [121]	Handcrafted, projection	projection in BEV grid (height, intensity, density)	YOLO	High inference speed, relatively good detection accuracy
VoxelNet [115]	Learned, voxel-based	voxelization in 3D grid, voxel-wise feature extraction, 3D convolutions	SSD	better accuracy than pillar-based encoding but costly 3D convolutions
PointPillars [24]	Learned, pillar-based	pillarwise feature extraction, 2D backbone	SSD	faster than other learned encoders for comparable accuracy

Table 3.8: Summary of popular LiDAR-based object detection models.

CNN graph for HW deployment using proxy estimations has lower HW-benefits compared to the baseline implementation. In order to get a reduction with respect to inference latency, the over-parameterized graph with pruned masks is translated to a slim graph with less parameters and connections.

To automate network pruning with HIL, the proxy-based estimations are substituted with a client that communicates with a HW setup dedicated for model inference (Fig. 3.9). The client sends a serialized CNN model definition that is parsed by the inference setup. The over-parameterized model along with pruning masks are translated into a slim model which is then executed by the hardware accelerator. After several warm-up inference cycles, the compiler optimizations converge and the accuracy of the pruned model is evaluated on the validation set. The hardware measurement of latency and runtime memory for each CNN layer are profiled during the inference step. The accuracy and runtime of the pruned model is returned from the inference setup to the client for reward computation. Similar to Eq. 3.9, the reward function can be expressed in terms of mAP and the HW latency measurements, as in Eq. 3.10.

$$R_{\text{Balanced}} = (1 - \frac{map^{baseline} - map^{pruned}}{bound}) \log\left(\frac{latency^{baseline}}{latency^{pruned}}\right) \tag{3.10}$$

3.5.3 Baseline models for 3D object detection

We train Complex-YOLO [121] and PointPillars [24] as baseline models. Complex-YOLO uses DarkNet-19 as a backbone that is a popular feature extractor for YOLO-based detection models known for its reduced complexity. The detection head of Complex-YOLO is an adapted version of YOLOv2 [127]. It uses the mean squared error as a regression loss and the binary cross entropy as a classification loss. The regression targets consist of the spatial coordinates and dimensions projected to the ground plane (x, y, w and h) and the orientation angle characterized by its real and imaginary parts. We use 3 anchor dimensions [1.6, 3.9, 1.56], [0.6, 0.8, 1.73] and [0.6, 1.76, 1.73] and two orientations 0 and $\frac{\pi}{2}$. The height and z-coordinate are regressed in a handcrafted manner using the anchor dimensions. ComplexYolo uses a handcrafted point cloud encoder. The point cloud is projected to the ground plane in a 1024×512 grid. Similarly to the original paper, the considered point cloud ranges are [0, 40m], [-40m, 40m] and [-2m, 1.25m] respectively in

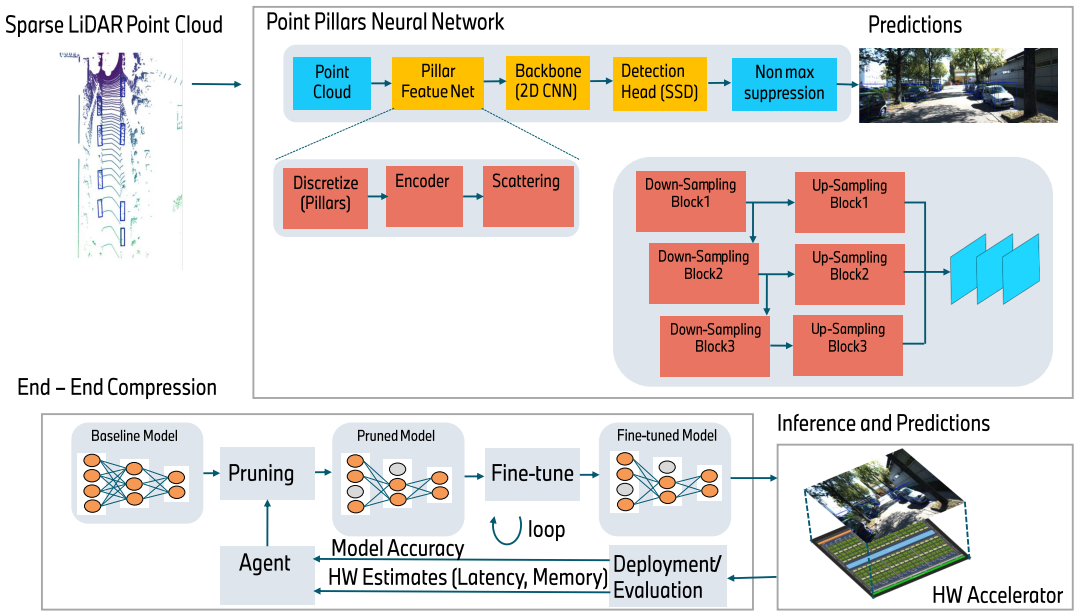


Figure 3.9: Overview of the RL-based pruning method for efficient LiDAR point cloud inference. The baseline classification model is pruned by an RL agent. The agent can directly observe the effect of its pruning actions on accuracy and inference latency through a HIL setup.

the x, y and z directions. Each cell in the input grid is encoded with 3 attributes which are the maximal intensity, the maximal height and the density of the points included in the grid cell. The model is trained for 100 epochs with a batch size of 16.

PointPillars uses the same backbone and detection head as VoxelNet [115]. The PointPillars encoder uses a pillar-based representation for the point cloud. Similar to the original work [24], the considered point cloud ranges are [0, 69.12m], [-39.68m, 39.68m], and [-3m, 1m] respectively in the x, y and z directions. The used input grid resolution is 432×496. Tab. 3.9 compares the complexity and latency of Complex-YOLO and PointPillars baselines. We observe that PointPillars is more computationally expensive than Complex-YOLO in terms of operations and inference time due to its learned encoder in contrast to the handcrafted encoder of Complex-YOLO. This encoder can be leveraged for pruning along with the backbone for better compression performance. However, note that the number of parameters of Complex-YOLO is much higher as its deeper layers have a higher number of output channels (1024, 2048) than PointPillars (256).

Model	Complexity (GFLOPs)	#params	Runtime Memory (MBs)	Latency (ms)
ComplexYolo	**50**	33.8 M	**674**	**7**
PointPillars	62	**4.8 M**	1502	19

Table 3.9: Model Complexity for the baseline Complex Yolo and Point Pillars based CNN models for 3D object detection.

Model	Car			Pedestrian			Cyclist		
	Easy	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
ComplexYolo	85.80	78.12	78.91	35.53	38.96	35.55	76.70	71.23	65.17
PointPillars	**89.59**	**89.14**	**89.14**	**63.22**	**61.1**	**58.52**	**80.83**	**72.83**	**69.86**

Table 3.10: mAP according to BEV IoU for explored models on the validation split of KITTI.

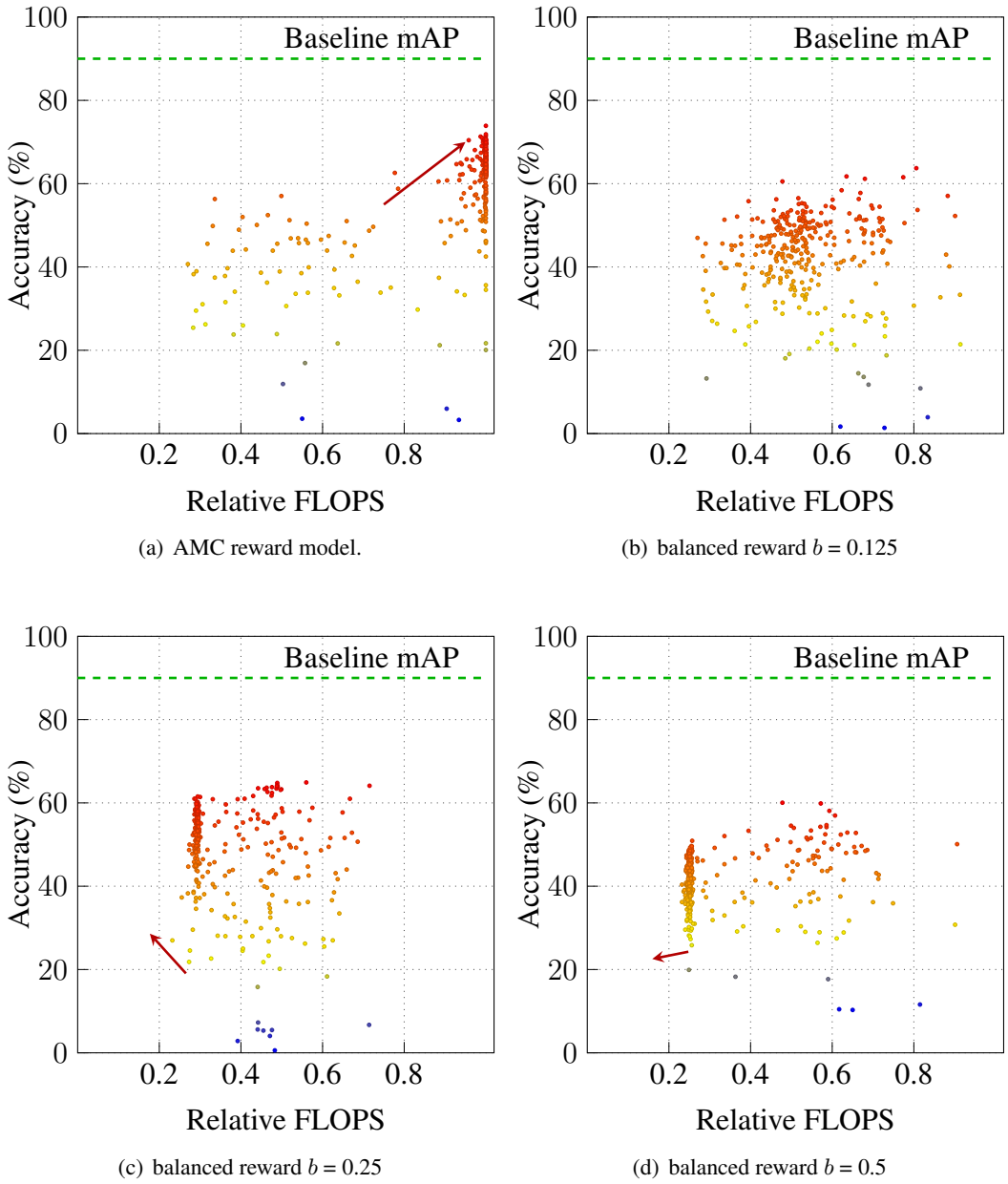
Tab. 3.10 compares the mAP in BEV for the Complex-YOLO and PointPillars models. We observe that PointPillars has the higher mAP values with different difficulty levels (*Easy*/ *Moderate*/ *Hard*) on the KITTI dataset [3]. Complex-YOLO also has a poor performance in terms of 2D metrics as the height dimension and z-coordinate are not learned attributes, rather *handcrafted* based on the chosen anchor dimensions.

3.5.4 RL Pruning using proxy metrics

This section obtains the pruning configuration based on FLOPs. In order to train the RL agent, we perform 50 warm-up episodes and 250 training episodes. During the warm-up phase, pruning ratios are set randomly. The used object detection model is highly sensitive to pruning. The agent's pruning actions with significant compression ratios often result in a mAP value close to zero. A meaningful reward computation with the agent's actions based on these mAP values is challenging. We observe that **one epoch of episode-wise fine-tuning** recovers the mAP value, thus enabling a better evaluation of the agent's actions with the chosen reward formulation without adding excessive GPU hours for exploration.

Reward Model. The reward formulation is an important aspect of an RL-based pruning setup. In Fig 3.10, we compare the accuracy guaranteed reward model in AMC [5] and several variants our balanced reward formulation (Eq. 3.10). We highlight the pruning exploration using a scatter plot. The solutions obtained with early episodes and those obtained with late episodes are shown in *blue* and *red* respectively. Fig. 3.10a shows the exploration of the agent with the AMC reward model. The agent is more focused on accuracy and converges with an insignificant reduction in FLOPS. The convergence areas obtained using the balanced reward model (shown in Fig. 3.10b, 3.10c, 3.10d) deliver a better tradeoff between detection accuracy and model complexity. This is due to the introduction of the accuracy bound, which reduces the contribution of the accuracy term to the reward model. Using different accuracy bounds, we obtain different convergence areas. When the bound value is higher, the convergence leads to lower model complexity. The scatter plot for an accuracy bound of 0.125 (Fig. 3.10b) shows more solutions converged in the right region indicating higher model complexity. Increasing the bound from 0.125 to 0.25 (Fig. 3.10c) shifts the convergence area to the left (lower model complexity) and slightly to the bottom (lower accuracy value). Going beyond 0.25 (Fig. 3.10d) results in an accuracy decay without any significant reduction in complexity.

Fine-Tuning. Additional fine-tuning is performed selectively for the Pareto optimal solutions of pruning schedules for 30 epochs. In Fig 3.11a, we highlight the Pareto dominant solutions (red) for the balanced reward with an accuracy bound of 0.25. The resulting design choices have



(a) AMC reward model.

(b) balanced reward $b = 0.125$

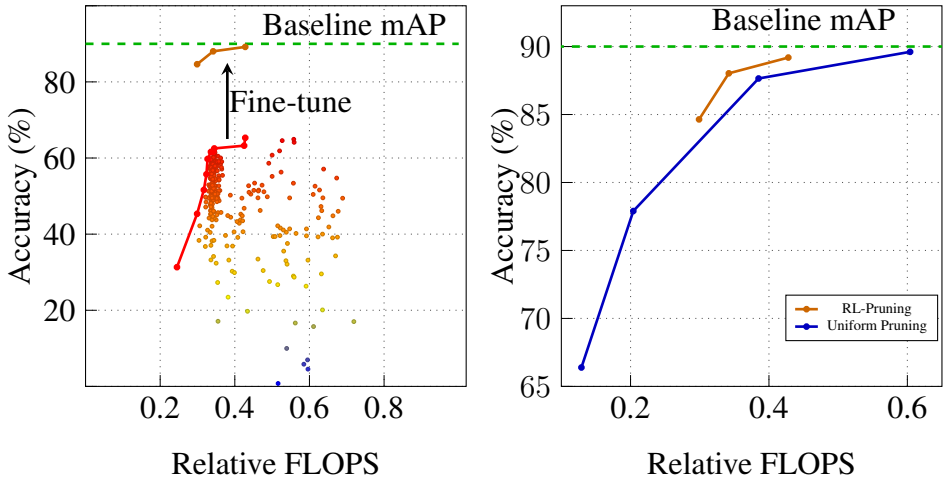
(c) balanced reward $b = 0.25$

(d) balanced reward $b = 0.5$

Figure 3.10: RL-based exploration with different reward bounds. The solutions in blue and red color indicates the pruning configurations explored during early and final stages of RL search respectively.

comparable baseline mAP with around 30-45% drop in complexity depending on the considered design choice.

Comparison with Uniform Pruning. In order to evaluate the effectiveness RL-based pruning, we uniformly prune the baseline PointPillars model using the magnitude-based pruning heuristic and layer-wise uniform pruning ratios. The uniform pruning ratios range from 0.1 to 0.9. The resulting pareto-optimal uniformly pruned solutions are fine-tuned for 30 epochs and shown in blue color in Fig. 3.11b. The comparison of the RL-based solutions (orange) with uniformly pruned ones shows the dominance of the design choices given by the data-driven approach.



(a) Baseline vs fine-tuned pareto optimal pruned solutions.

(b) Uniform vs RL pruning.

Figure 3.11: Comparison of PointPillars [24] baseline and uniformly pruned solutions against an RL-based approach with a reward formulation based on FLOPs reduction.

3.5.5 RL pruning using Hardware metrics

Automated pruning based on proxy metrics such as FLOPs significantly reduce the model's complexity with a slight degradation in the prediction accuracy. However, the reduction in FLOPs does not always reflect a proportional latency reduction. Fig. 3.12 analyzes the latency and number of FLOPs for each layer of the PointPillars baseline. Even though the first two layers have comparable complexity to other layers, the profiling results show that their latencies are larger than the subsequent layers. The first layer represents the encoder and its latency, as depicted, is the sum of the latencies of the underlying operations. The encoder consists of dense matrix multiplication, batch normalization, ReLU, max pooling and scatter operations. Furthermore, the first layers involve a high volume of data transfer, resulting in a memory-bounded execution. The computation of FLOPs does not take the HW execution aspects into consideration. A FLOPs-based pruning setup will not distinguish the encoder from other layers in computational complexity, which practically have much lower latencies.

The results obtained from the HIL setup are shown in Fig. 3.13. These solutions are obtained with the same method as FLOPs-based pruning, and then fine-tuning the Pareto-optimal solutions for 30 epochs. In Fig 3.13a, we show that the hardware-aware pruning configurations dominate

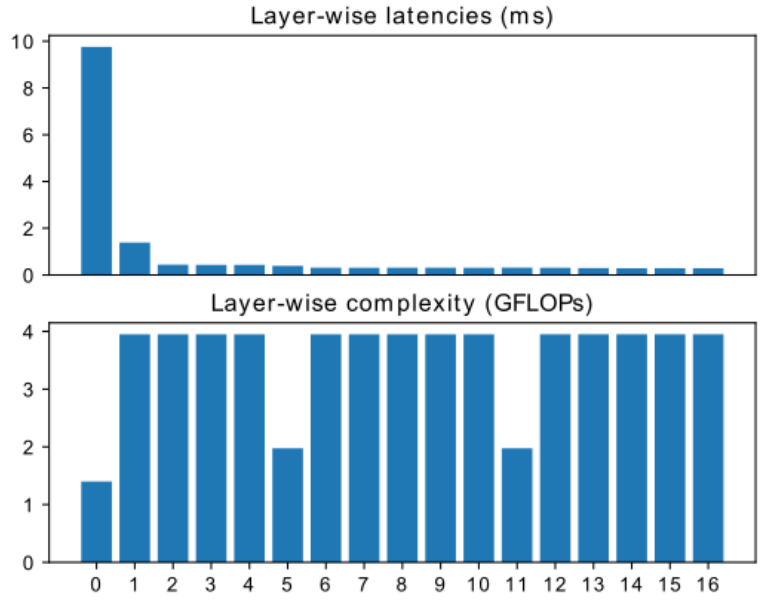
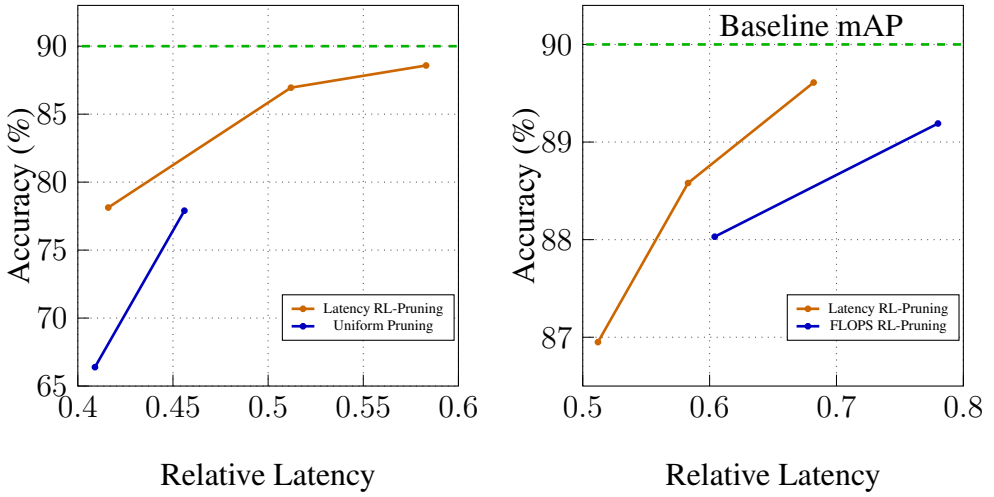


Figure 3.12: Latency in milliseconds (top) and complexity in GFLOPS (bottom) for each layer of the PointPillars model. The x-axis represents the layer numbers in both charts.

the uniformly pruned solutions. The HW-aware RL-based approach makes it possible to achieve more accuracy for the same relative latency to the baseline. We obtain an improved mAP (66% $\rightarrow$ 78%) with a latency reduction of 59%. In Fig. 3.13b, we show that solutions obtained from hardware-aware pruning dominate FLOPs-based pruning as well. Using the HW-aware pruning approach, we observe a latency reduction of 59%, while the best latency reduction obtained for a FLOPs-based setup is only 40% compared to the baseline.

We present a qualitative comparison between two models compressed with a HW-aware pruning approach, offering $1.92\times$ and $2.38\times$ latency reductions, versus their baseline. Each row of Fig. 3.14 is a scene, and we highlight the predictions of the point cloud on different models in the columns. The ground truth bounding boxes is present in green and predictions in blue. In the first row, we observe that both pruned models have equivalent qualitative predictions, when compared to the baseline model. The scene in the second row involves a large number of vehicles driving in both directions and parking on both sides of the road. The pruned model with lower latency benefits ($1.92\times$) performs similar to the baseline model. However, the aggressively pruned model ($2.38\times$) shows prediction quality degradation, where vehicles which are too far to the left are not detected. In the third row, we observe that the pruned models are able to detect cars which are not predicted by the baseline model. Even though the baseline predictions for bounding box attributes are better in these cases (location and dimensions), this improvement in delivering accurate bounding boxes which come at the cost of the inability to detect other objects in the scene. In this case, pruned models show a better trade-off between classification and regression attributes.



(a) Uniform vs Latency.

(b) FLOPs vs Latency.

Figure 3.13: Comparison of HW-aware pruned solutions with uniform and FLOPs based approaches. We highlight the relative reduction in latency (x-axis) w.r.t. baseline.

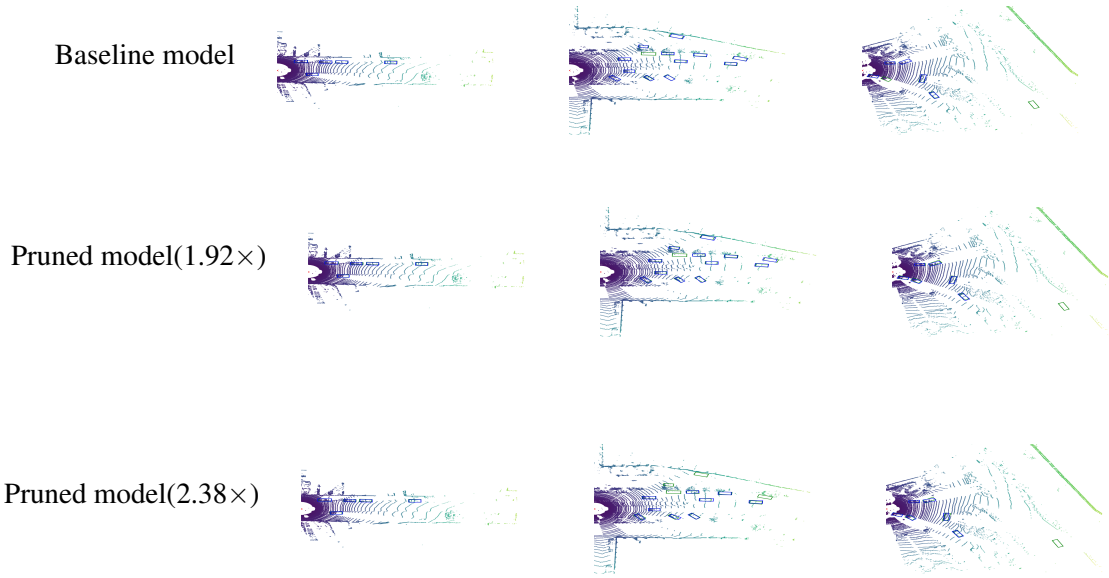


Figure 3.14: Qualitative results of baseline and two different channel pruning configurations.

3.5.6 Comparison to State of the Art

The comparison to the state of the art is challenging due to the use of different validation splits and different inference hardware across various approaches. We use the validation splits similar

to Complex-YOLO [121]. The latency measurements for our approach as well as Complex-YOLO are performed on GTX 1080 Ti with a batch size of 8. The latency is measured for the entire CNN model which includes the encoder, backbone and detection head. It excludes the pre-processing and post-processing parts which are in general not specific to the used approach and less computationally expensive. We directly adopt latency (also evaluated on GTX 1080 Ti) and accuracy evaluation of [118], [116], [128], [129], [120] from [23] in Tab. 3.11. LiDAR-based and fusion-based 3D object detection methods outperform Centernet [113] which is a recent and popular image-based 3D object detection model. Compared to fusion-based methods such as F-PointNet [118] and AVOD-FPN [116], voxel-based LiDAR-only systems have a reduced model complexity. The point-based model Point-GNN [128] which uses graph neural networks for point-wise feature extraction has the highest accuracy among all the considered models. However, its inference time exceeds all other models by a large margin. PointPillars is at least $2.5\times$ faster than other end-to-end trainable CNNs reported in Table 3.11. Our hardware-aware pruned model based on the PointPillars baseline achieves $1.5\times$ faster inference speed, while maintaining competitive accuracy results.

Approach	**Modality**	**Complexity (GFLOPs)**	**Latency (ms)**	**Car (BEV IoU)**			**Car (3D IoU)**		
				Easy	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
Centernet [113]	Camera	119.8	-	31.5	29.7	28.1	-	-	-
F-PointNet[1] [118]	Fusion	-	170	90.58	84.73	75.12	82.13	69.22	60.78
AVOD-FPN[1][116]		-	100	90.64	84.37	80.04	82.77	71.94	66.31
Point-GNN[1][128]	Lidar	-	643	**92.04**	**88.20**	**81.97**	**87.25**	**78.34**	**72.29**
Fast Point R-CNN[1][129]		-	95	89.97	87.08	80.40	85.39	77.46	70.21
SECOND[1][120]		-	50	89.39	83.77	78.59	83.34	72.55	65.82
PointPillars[1][24]		62	20	-	87.98	-	84.05	74.99	68.30
PointPillars[compiler-aware NAS [23]][1]		**3.8**	18	90.02	86.79	80.80	85.20	75.57	68.37
PointPillars [Ours]		19.8	**13**	90.18	88.13	80.29	85.42	76.20	68.50
Complex-YOLO [121]		50	**7**	85.80	78.12	78.91	-	-	-

Table 3.11: Comparison of our HW-aware pruned PointPillars model with the state-of-the-art 3D object detection methods. [1] indicates that the accuracy and latency measurements are reported based on the work on compiler-aware NAS [23].

3.5.7 Discussion

In this section, we propose an end-end HW-aware pruning pipeline for 3D object detection using LiDAR data. The approach adapts the RL-agent from the work of AMC [5] to predict the layer-wise compression ratios. A magnitude-based pruning is performed in order to reduce the complexity of the baseline model. The compression pipeline includes a HIL setup which translates the pruning configuration into a CNN graph with reduced parameters and connections. This automated HW translation allows the evaluation of the accuracy and latency of the pruned graph while taking into account HW-specific and compiler optimizations. A HW-aware balanced reward model is formulated based on the measured values and used in order to train the agent.

The baseline model was trained using the implemented model exploration framework which supports different point cloud encoders and anchor-based detection heads. PointPillars is an end-to-end trainable CNN which uses a learned pillar-based encoder to extract features from the

point cloud. It achieves around 11% better accuracy than CNN using handcrafted encoders such as ComplexYolo in terms of mAP in BEV for the car class. Furthermore, it outperforms other end-to-end trainable CNNs by at least $2.5\times$ in terms of inference time.

The baseline model is further pruned using the HW-aware compression approach. During the search process, the accuracy of each pruning configuration is scattered along with latency to form a Pareto plot. We should also note that the **pruned configuration is fine-tuned for one epoch** to realize the reward value. Iterative pruning and fine-tuning could result in enormous amount of GPU-hours, especially for larger datasets such as NuScenes [130]. Therefore, we reduce the number of GPU-hours using In-Train optimization techniques which is discussed in Chapter 4. After the search is complete, different design choices can be obtained from fine-tuning the Pareto-optimal solutions. The latency can be reduced up to a factor of $2.38\times$ with competitive accuracy values compared to baseline model. The obtained pruned configurations outperform uniform solutions and FLOPs-based pruned models.

3.6 Conclusion

Optimization of CNNs and the design of resource-constrained HW platforms go hand in hand. In this chapter, we introduce the three stage compression pipeline consisting of *model-training*, *compression* and *HW-evaluation*. We point out the limitation of considering proxy estimates during the compression loop and formulate HW-models to generate estimates such energy, latency. We identify the challenges to schedule and map certain CNN workloads and propose fast scheduling methods based on HW-heuristics. We leverage the estimates of the HW-model for optimizing and exploring CNN models. With HW-aware pruning using HW estimates, we achieved $\times 2$ energy and latency reduction with minimal loss in prediction accuracy compared to its baseline unpruned models. We extend the investigation to segmentation tasks, where observations on pruning rates of decoder and ASPP blocks were made with respect to the pruning target. DeepLabv3's energy and latency were reduced by $\sim$50%, while improving the accuracy of the baseline, over-parameterized model. We further investigated the accuracy-latency trade-offs for LiDAR based CNN processing. We compressed the PointPillars baseline using an automated HW-aware RL-based pruning approach. We obtained different design choices by fine-tuning the Pareto-optimal pruned solutions. We reduced the latency up to a factor of $1.5\times$ on NVIDIA GTX 1080 TI with comparable accuracy values. We also observed that the pruned models with HW awareness outperform the uniformly pruned solutions as well as the RL-based pruned solutions with proxy rewards.

4 Fast Compression

CNNs are challenging to deploy on a target HW with constrained resource budget. As discussed in Sec. 3.1, the traditional model compression works have used the sequential three stage pipeline to reduce the size of the models. All the stages require careful hyper-parameter tuning to obtain optimal performance. Furthermore, modern CNNs increase the depth and width of the model architecture to improve the prediction accuracy. This would increase the search space to find an effective compressed configuration, resulting in high amount of optimization time. The *major computational effort* in the three stage compression pipeline is spent for iterative fine-tuning of every search candidate CNN to recover prediction accuracy. The prediction accuracy after fine-tuning serves as a good metric to guide the search algorithm to produce near-optimal compression configurations. In-order to atleast explore 100 compressed configurations on ImageNet dataset [1] for ResNet18 [39], approximately 50 GPU hours are required using one NVIDIA-V100 GPU [15]. Furthermore, the compressed configurations found during the search must be re-trained/fine-tuned till convergence again to report the validation/test accuracy.

In this chapter, Sec. 4.1 elaborates the need for huge number of GPU hours in the three stage compression pipeline. Sec. 4.2 discusses related work on post-train/in-train compression methods. We specifically contextualize these works to compare the trade-off between **compression rates** and **GPU hours**. Sec. 4.3 introduces an approach on formulating a RL-agent which produces channel pruning configurations with lower GPU hours. Sec. 4.4 highlights a novel training scheme which produces sparse CNNs using learnable prune masks during the training process. Finally, Sec. 4.5 discusses a novel training scheme incorporating HW-awareness to produce mixed precision configurations. The chapter is based on the publications of Vemparala et al. [27], Vemparala et al. [28] and Vemparala et al. [29].

4.1 Reducing GPU hours

4.1.1 Search space for Model Compression

We aim to obtain an efficient compression strategy for the CNN model with lower GPU hours. Using pruning methods, we identify the redundant elements in the weight matrix W^l for all layers $l \in \{1, ..., L\}$. As discussed in Sec. 2.3.2, Sec. 3.1.1, we use prune masks to derive a sparse weight matrix for each layer l as $\tilde{W}^l = W^l \odot \mathcal{M}^l$, where mask $\mathcal{M}^l$ can have different dimensionality based on the pruning regularity (Fig. 2.10). For irregular sparsity, the dimensionality of the pruning mask is given by, $\mathcal{M}^l \in \{0,1\}^{K_w \times K_h \times C_i \times C_o}$. For kernel pruning, the dimensionality of the mask is given by, $\mathcal{M}^l \in \{0,1\}^{1 \times 1 \times C_i \times C_o}$. Subsequently, for channel pruning, the dimensionality of the mask is given by, $\mathcal{M}^l \in \{0,1\}^{1 \times 1 \times C_i \times 1}$. Another way to find an efficient compressed configuration is to perform model quantization. Using model quantization, each convolutional layer $l \in \{1, ..., L\}$, can be assigned bit-widths for both weights and activations

as $\{b_w^l, b_a^l\}$. Using b_{dtype} bit-width, dtype consists of at-most $2^{b_{\text{dtype}}}$ unique values. For a bit-serial accelerator such as BISMO [131], allowed bit-width range $b_{\text{dtype}} \in \{2, 3, 4, 5, 6, 7, 8\}$. Leveraging higher quantization levels in different layers, produces better task specific metrics for the CNN model. On the other hand, using higher bit-widths for convolutions gets challenging to execute on real-time HW due to resource/latency constraints.

The search algorithms discussed in Sec. 3.1.2 attempt to obtain efficient pruning and quantization configurations, dominating the baseline CNN models in terms of prediction accuracy and compression rates. The search space complexity for model pruning depends on the regularity. The search complexity for simplest case, i.e. channel pruning is given by $2^{C_i{}^l}$. To obtain an efficient mixed precision quantized CNN, the search space complexity for a flexible bit-serial accelerator is given by $|b_w|^l \times |b_a|^l$. Here, $|b_w|$ and $|b_a|$ refers to number of bit-width configurations for weights and activations respectively. For commonly used CNNs such as ResNet18, the compression search space consists of $7^{19^2} \times 2^{64^5} \times 2^{128^5} \times 2^{256^5} \times 2^{512^4}$ CNN candidates. Furthermore, for deeper and modern CNNs, the search space grows exponentially larger. Therefore, in this chapter, we investigate methods which reduce the GPU hours and still obtain efficient compressed configurations with minimal degradation of prediction accuracy.

4.1.2 Iterative fine-tuning in Compression Pipeline

We intend to maximize the validation accuracy $Valacc$, minimize the HW-metrics $HW_{metrics}$ by searching for an efficient compressed configuration α. Evaluating a compressed configuration could get challenging due to the inner loop optimization specified in Eq. 4.1. The inner loop involves ensuring convergence in the training behaviour to determine optimal weights $\theta(\alpha*)$. The outer loop involves searching for a near optimal compression configuration α.

$$\begin{aligned} \alpha^* &= \arg\min_{\alpha} \;\; HW_{metrics}(\texttt{Target}_{\texttt{HW}}, f(x, \theta, \alpha)) \\ \alpha^* &= \arg\max_{\alpha} \;\; ValAcc \;\; \text{with} \;\; ValAcc = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{val}}}\left[f(x, \theta, \alpha)\right] \\ \theta^* &= \arg\min_{\theta} \mathcal{L}_{train}(x, \theta, \alpha^*) \end{aligned} \tag{4.1}$$

In-order to evaluate a compressed configuration α during the search stage, fine-tuning is performed for few epochs to quickly evaluate the architecture. This becomes important to compression methods like channel pruning, quantization which modify the architecture and impacts the distribution of Ofmaps at every layer. Furthermore, for tasks such as object detection which produces outputs from different branches, e.g. bounding box coordinates, classes, fine-tuning becomes necessary to guide the search algorithm to achieve near optimal solutions [26]. In case of quantization search on ImageNet dataset, approximatly half an epoch is needed to distinguish the validation accuracy for various mixed precision configurations [15]. In case of channel pruning search on KITTI [3], one epoch is needed to distinguish the validation accuracy (Sec. 3.5) for various sparse models. The iterative fine-tuning becomes further essential when the complexity of training increases, e.g. including multi-objective optimization tasks such as improving adversarial robustness. Therefore, in this chapter we explore search techniques which reduce the number of fine-tuning epochs (Sec. 4.3) and in-train optimization techniques (Sec. 4.4, 4.5) which consume GPU hours similar to baseline model.

4.2 Related Work

When the compression is performed after training the baseline model, this approach is referred to as *post-train compression*. Recent works in literature identified limitations in terms of computational effort for this approach and proposed to integrate the compression method into the training phase to jointly optimize the weights, pruning connections and quantization choices. We refer to these unified training and compression schemes as *in-train compression*. Related work in post-train compression and in-train compression have been discussed in details in the subsections 4.2.1 and 4.2.2 along with the advantages and limitations of each approach.

4.2.1 Post-train Compression

Post-train compression usually adopts a standard three stage approach as discussed in Sec. 3.1. A wide range of post-train pruning and mixed precision quantization strategies have been proposed to determine near-optimal layer-wise compression ratios. Huang et al. [21] demonstrated a try-and-learn RL-based filter-pruning method to learn both sparsity ratio and the exact position of redundant filters, but it leaves out the number of fine-tuning epochs for every search candidate as a hyper-parameter. Here, the compression strategy changes, depending on the model's architecture and the dataset at hand. In AMC [5], a DDPG based RL-agent is utilized in regular filter pruning. The RL-agent provides the environment with a continuous action that can be defined as the compression ratio of each layer. Based on the magnitude obtained from the L2-norm heuristic and the sparsity ratio of each layer given by the RL-agent, the redundant channels are pruned. However, in complex applications like object detection [26], AMC pruning requires fine-tuning at every episode to efficiently explore the pruning search space.

ReLeQ [132], HAQ [13] and AutoQ [133] propose RL-based exploration schemes to determine HW-aware layer-wise quantization strategies. ReLeQ searches for bit-width configurations only for the weights of each layer, while HAQ searches for both weights and activations. AutoQ determines a fine-grained quantization strategy for each filter in every layer. The reward function is evaluated after executing the inference of the quantized CNN on a target HW. This involves finding a bit-width strategy in a large search space, demanding high training effort due to the iterative fine-tuning of every solution during the exploration. In APQ [14], a joint search is conducted to determine model's architecture, pruning, quantization configuration. The model configuration is found through evolutionary search and achieves a remarkable reduction in search time as it uses a quantization-aware accuracy predictor for the CNN to estimate accuracy of each candidate. The accuracy predictor still needs to be trained in order to perform reasonable predictions of the CNN's accuracy under different quantization strategies. This incurs an additional overhead of GPU hours when collecting the predictor's training dataset. Furthermore, the accuracy predictor is very costly to train (2400 hours required for ImageNet dataset) and needs to be retrained when new CNN workloads or datasets are adopted. In Sec. 4.3, we extend the work of Huang et al. [21] incorporating GPU-hour awareness by proposing an RL-agent which learns fine-tuning iterations making it a hyper-parameter free method.

4.2.2 In-train Compression

Post-train compression approaches can reduce the model size without a significant drop in prediction accuracy. This helps in reducing energy consumption and storage during inference. However, in-train compression approaches reduces the number of GPU hours required for optimization process. There have been efforts to integrate the pruning process into the training phase to jointly optimize the weights and pruned connections. The autoencoder-based low-rank filter-sharing technique (ALF) proposed by Frickenstein et al. [98] utilizes sparse autoencoders that extract the most salient features of convolutional layers. ALF discards filters in an unsupervised manner, decides the sparsity ratio and location of pruned filters for each layer. ALF is limited to filter pruning and does not support other pruning regularities. ALF also adds an additional expansion layer which prohibits the extraction of inter-layer filter pruning benefits. Zhang et al. [134] present a systematic weight pruning framework for CNNs, where pruning is formulated as a constrained non-convex optimization problem subject to limiting the cardinality of weights in each layer, resulting in a high degree of sparsity. By leveraging alternating direction method of multipliers (ADMM), the optimization problem can be decomposed into two subproblems which are then solved separately. The first subproblem uses Stochastic Gradient Descent (SGD) to optimize the model weights θ with respect to cross-entropy loss and an adaptive regularization loss $\frac{\rho}{2}\|\theta - Z\|_F^2$. Z is a duplicate variable for weight matrix θ that satisfies the cardinality constraint $|Z| <= n$, n denoting the number of non-zero values in the weight matrix. The second subproblem is solved analytically by applying projection on Z such that it is closest to θ but still satisfies the cardinality constraint. The authors subsequently extend their work in StructADMM [135] to structured sparsity and provide analysis on row pruning, column pruning and filter pruning. Although the task-specific and pruning objectives are solved simultaneously, there are some drawbacks of this approach. This work requires prior knowledge of layerwise sparsity ratios from other pruning works which might not always be available. The cardinality constraint is not guaranteed to be met which might require hard pruning at the end, relying again on the magnitude heuristic. Furthermore, to ensure convergence, initialization of the duplicate variable Z plays a vital role. ADMM [134] uses pre-trained model weights to initialize Z.

Sparse learning or training sparse networks from scratch [136, 137, 138] can also be considered as an in-train pruning technique, which has achieved extremely high pruning rates with negligible accuracy degradation. This method does not require a pretrained dense model and the network topology is updated during training through pruning and regrowing connections. Parameters are pruned based on magnitude and grown back at random [136] or based on gradient [138] or momentum [137] information. However, these methods often require predefined layer-wise sparsities and are mostly effective in reducing model size through weight pruning, rather than focusing on the HW advantages through structured pruning. In Sec. 4.4, we aim at eliminating the need of a pretrained model and devise a method that can learn the layerwise sparsities on its own, given the target sparsity for the entire model.

WaveQ [139] formulates a gradient-based optimization problem by introducing a sinusoidal regularization loss, pushing the weights to optimal quantization levels. However, the quantization level of activations is set uniformly. Wu et al. [140] learns quantization levels through a path selection-based neural architectural search formulation. Each quantization level is treated as a different type of layer and importance of each level is captured using a Gumbel-Softmax function

formulation. This method is challenging to scale for larger CNN models, as the super-network leads to larger search and training costs. The work in LBS [141] alleviates the compute cost by devising a single-path scheme that captures different quantization and filter pruning strategies using binary gates. Instead of allowing the entire search space to determine optimal bit-widths, LBS limits the quantization choices to ensure training stability during the reassignment process (e.g. 2, 4, 8 bits). The work in [142] investigates a suitable parameterization of the quantization operation to learn bit-widths and avoid unbounded gradient updates. In particular, the authors learn step-size and dynamic range for quantized weights and activations to determine the optimal quantization strategy. Differently, in Sec. 4.5, we learn the number of unique values required to represent weights and activations by progressively reducing the bit-widths using a differentiable loss formulation. Furthermore, we determine HW-aware mixed precision CNNs by leveraging guassian process estimator to predict latency.

4.3 GPU Hours Aware RL Pruning

In this section, we discuss a pruning approach, namely *Leaning to Prune Faster* (L2PF) [27] involving a RL-agent, which learns a layer's redundant features and adequate fine-tuning time concurrently across the search phase. In this regard, the pruning problem, environment, state space and action spaces in the context of a RL are formulated in Sec. 4.3.1. We discuss the agent's design in Sec. 4.3.2 and reward formulation in Sec. 4.3.3. Specifically, we build upon a learning-based pruning approach [21] by appending the GPU hour awareness in the form of continuous reward. We conduct various experiments in Sec. 4.3.4 to analyze the GPU hour savings using the proposed RL search approach. Finally, we discuss the key findings in the post train GPU hour aware RL search in Sec. 4.3.5.

4.3.1 RL Search Formulation

Channel Pruning: The structured channel-pruning task within a RL framework can be expressed as a try-and-learn problem, similar to the work from Huang et al. [21]. We aim to search an efficient filter pruning configuration that achieve the highest Compression Ratio (CR), while incurring a minimum loss of prediction accuracy and requiring a minimum number of fine-tuning epochs during the exploration episodes. Fig. 4.1 demonstrates the interplay between the proposed pruning agent and CNN environment. The proposed method is able to learn three aspects: First, the minimum number of epochs required to explore each pruning strategy. Second, the degree of sparsity of each layer in the model. Third, the exact position of the least important filters to be pruned.

Formally, let f be a fully-trained model with L layers and the input of the ℓ^{th} convolutional layer has a shape $[c^\ell \times w^\ell \times h^\ell]$, where c^ℓ, h^ℓ and w^ℓ represents number of input channels, height and width. The ℓ^{th} layer is convolved with the weight tensor $\mathbf{W}^\ell$, *i.e.* 2D convolutional layer's trainable parameters, with shape $\left[N^\ell \times c^\ell \times k^\ell \times k^\ell\right]$, where k^ℓ represents the kernel size and N^ℓ is number of filters. After pruning n^ℓ filters, the weight tensor is of shape $\left[(N^\ell - n^\ell) \times c^\ell \times k^\ell \times k^\ell\right]$. The layer's CR is defined as $\frac{c^\ell - n^{\ell-1}}{c^\ell}$. Additionally, we define model CR to be the total number of weights divided by the number of non-zero weights.

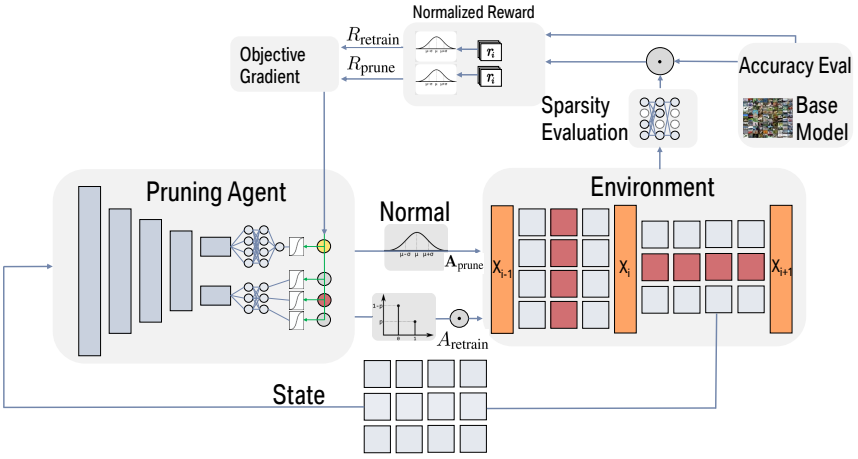


Figure 4.1: Agent receives rewards and weights as input state, whereas environment receives both prune and epoch actions. In each prune episode, M=5 Monte-Carlo set of actions are sampled ($\mathbf{A}_{\text{prune}}$ and A_{retrain}). The corresponding M rewards (R_{prune} and R_{retrain}) are normalized to zero mean and unit variance [21].

Environment: The environment is the pretrained CNN model to be pruned. The *state space* is the fully trained weight tensor $\mathbf{W}^{\ell}$ of the layer to be pruned, which is used as an input for the agent, similar to Huang et al. [21]. For each layer (or residual block), a new agent is trained from scratch. The environment receives two actions from the agent: pruning action $\mathbf{A}_{\text{prune}}$ and fine-tuning epoch action A_{retrain}. Subsequently, it generates a reward $R = R_{\text{prune}} + R_{\text{retrain}}$. For each filter there is a binary mask $\mathbf{m}_i^{\ell} \in \{0,1\}^{c^{\ell} \times k^{\ell} \times k^{\ell}}$. Pruning the i^{th} filter $\mathbf{W}_i^{\ell}$ in layer ℓ is performed by element-wise multiplication between the filter $\mathbf{W}_i^{\ell}$ and its corresponding mask $\mathbf{m}_i^{\ell}$. When pruning $\mathbf{W}_i^{\ell}$, the i^{th} kernel of all filters in the $(\ell+1)^{th}$ layer are also pruned. At each pruning step, masks are updated according to $\mathbf{A}_{\text{prune}}$ and the environment is fine-tuned for a few epochs e_{retrain}.

Action Space: The action space of the proposed RL-framework is split into two distinct spaces to satisfy the discrete and continuous requirements of actions for pruning and epoch learning respectively. The discrete pruning action space is the combination of all possible prune actions $\mathbf{A}_{\text{prune}}$. It is clear that action space dimension grows exponentially as $\mathcal{O}(2^N)$, where N is the number of filters in a layer. Discrete actions are sampled from N independent stochastic Bernoulli units [91]. Each Bernoulli requires a parameter p which presents the probability of keeping the filter as shown in Eq. 4.2.

$$P(a,p) = \begin{cases} p & \text{if a = 1 (keep filter)} \\ 1-p & \text{if a = 0 (prune filter)} \end{cases} \tag{4.2}$$

The continuous epoch-learning is used to determine the number of fine-tuning epochs e_{retrain}. Continuous actions are typically sampled from continuous distributions like normal $\mathcal{N}(\mu,\sigma)$ or beta distributions $Beta(\alpha,\beta)$ [143]. Beta distribution requires two parameters to be learned, which complicates gradient propagation implementation. Normal distribution has also two

parameters as shown in Eq. 4.3, the mean μ is a learnable parameter, while σ is chosen to be non-learnable. Here, $\mu \in (0, 1)$ since μ value is fed in from sigmoid unit.

$$\mathcal{N}\left(\mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right) \tag{4.3}$$

4.3.2 Agent Design

The agent is a non-linear stochastic functional approximator parameterized by θ. It is composed of four convolutional layers, two classifiers each with two fully connected layers [21], and two types of stochastic output units, *i.e*. Bernoulli and Normal. The agent parameters are $\theta = \{\mathbf{w}, \mu, \mathbf{P}\}$, where parameters $\mathbf{w}$ are the agent weights, μ is a learnable parameter to sample the fine-tuning action A_{retrain}, and $\mathbf{P}$ is the set of probabilities for Bernoulli units. The agent outputs two actions: discrete action $\mathbf{A}_{\text{prune}}$ for pruning, and continuous action A_{retrain} for fine-tuning epochs.

The *pruning action* $\mathbf{A}_{\text{prune}}$ is a set $\{a_1^\ell, a_2^\ell, ..., a_{Nl}^\ell\}$, where $a_i^\ell \in \{0, 1\}$ is equivalent to $\{prune, keep\}$ and N^ℓ is the number of filters in the ℓ^{th} layer [21]. Using this scheme, the agent is able to explore both sparsity ratio and to select the exact position of filters to prune.

The *fine-tuning action* A_{retrain} is a continuous action sampled from a normal distribution with two parameters - μ, σ. The mean μ is a learnable parameter, while σ is chosen to be non-learnable and set to be proportional to $|R_{\text{retrain}}|$ [91, 144]. The value of σ controls how far a sample can be from the mean. When reward signal R_{retrain} is low indicating bad actions, then σ takes higher value which allows the agent to explore actions further away from μ. Actions $A_{\text{retrain}} \notin [0, 1]$ are considered *bad* and given negative rewards. The environment fine-tunes for the number of epochs given in Eq. 4.4, where β is an upper limit for e_{retrain}.

$$e_{\text{retrain}} = \min[\max[0, A_{\text{retrain}}], 1] \times \beta \tag{4.4}$$

We leverage the SPG method to find an optimal policy π^*. SPG is guaranteed to converge at least to a local optimum without requiring the state space distribution [90]. Our objective function $J(\theta)$ is the expected sum of all rewards over one episode. The objective gradient w.r.t. the policy parameters is given in Eq. 4.5. Both terms in Eq. 4.5 can be solved approximately using the policy gradient theorem. Specifically, we implement a variant of SPG called REINFORCE [91, 21]. The agent parameters θ are updated with gradient ascent so that actions with higher rewards are more probable to be sampled [91].

$$\nabla_\theta J\left(\theta\right) = \nabla_\theta \mathbb{E}\left[r_{\text{prune}}\right] + \nabla_\theta \mathbb{E}\left[r_{\text{retrain}}\right] \tag{4.5}$$

The first term in Eq. 4.5 has the Bernoulli policy $\pi_B\left(\mathbf{A}_{\text{prune}}|\mathbf{W}^\ell, \mathbf{P}, \mathbf{w}\right)$, while the second has the normal policy $\pi_N\left(\mathbf{A}_{\text{retrain}}|\mathbf{W}^\ell, \mu, \mathbf{w}\right)$, where $\mathbf{W}^\ell$ are weights for layer to prune. Finding a closed-form solution for the expectation is not feasible, so it is approximated using M samples of a Monte-Carlo gradient estimator with score function [145, 90].

The gradient of our objective function is given by Eq. 4.6.

$$\nabla_\theta J\left(\theta\right) \approx \sum_{j=1}^{M}\left[\left(R_{\text{prune}}\right)_j \cdot \sum_{i=1}^{n} \frac{a_{ij} - p_{ij}}{p_{ij}(1-p_{ij})} \cdot \frac{\partial p_{ij}}{\partial \mathbf{w}} + \left(R_{\text{retrain}}\right)_j \cdot \frac{a_j - \mu_j}{\sigma_j^2} \cdot \frac{\partial \mu_j}{\partial \mathbf{w}}\right] \tag{4.6}$$

4.3.3 Multi-Objective Reward

The quality of agent's action is conveyed back as a reward signal, R_{prune} and R_{retrain} for $\boldsymbol{A}_{prune}$ and $A_{retrain}$ respectively.

Prune Reward: The prune reward R_{prune} is a measure for sparsity level and model accuracy acc_{pruned}. It promotes actions that remove filters with minimum accuracy loss of the pruned model w.r.t. the validation set. Following the work of Huang et al. [21], we define the prune reward as a product of two terms, *i.e.* acc_{term} and eff_{term}, as stated in Eq. 4.7.

$$R_{\text{prune}}\left(\mathbf{A}^{\ell}_{\text{prune}}, acc_{\text{pruned}}\right) = acc_{\text{term}} \cdot eff_{\text{term}} \tag{4.7}$$

Accuracy Term: Similar to Huang et al. [21], acc_{term} is defined in Eq. 4.8. The bound b is a hyper-parameter introduced in the reward function to allow control over the trade-off between model compression and tolerable accuracy drop. When the accuracy drop is greater than b, acc_{term} is negative, otherwise it lies in the range $[0, 1]$.

$$acc_{\text{term}} = \frac{b - \max\left[0, acc_{\text{base}} - acc_{\text{pruned}}\right]}{b} \tag{4.8}$$

Efficiency Term: To prevent the agent from choosing huge compression ratios, the efficiency term eff_{term} proposed by Huang et al. [21] is extended as shown in Eq. 4.9. If the prune action is aggressive, the accuracy drop will be less than the bound b resulting in a negative reward. If layer sparsity ratio is low, eff_{term} will drive reward to zero.

$$eff_{\text{term}} = \begin{cases} \log \dfrac{N}{(N-n)} & \text{if } \ (N-n) \leq N \\ -1 & \text{if } \ (N-n) = 0 \end{cases} \tag{4.9}$$

Fine-tuning Epoch Reward: The fine-tuning epoch reward R_{retrain} is responsible for promoting a lower number of fine-tuning epochs. The reward is expressed in Eq. 4.10. An action is considered *good* when $|A_{\text{retrain}}|$ is low without causing an intolerable accuracy drop. If the environment incurs no accuracy loss then $R_{\text{retrain}} = 0$, when loss is incurred then it will be a negative value scaled by the absolute value of A_{retrain}.

$$R_{\text{retrain}}\left(A_{\text{retrain}}, acc_{\text{pruned}}\right) = |A_{\text{retrain}}| \times \left(acc_{\text{pruned}} - acc_{\text{base}}\right) \tag{4.10}$$

In each prune episode, M Monte-Carlo set of actions are sampled again resulting in M corresponding rewards R_{prune} and R_{retrain}. The reward values are normalized to zero mean and unit variance for both set of rewards [146, 145].

4.3.4 Experiments

Setup: Our experiments are conducted on the CIFAR-10 [42] dataset. One-time random splitting of the 50k images into 45k training and 5k evaluation is performed. Agent reward is evaluated on 5k images. To ensure that our pruning method generalizes, the 10k images in the test dataset are held separate and only used after the agent learns to prune a layer, to report actual model accuracy. No training or reward evaluation is performed using the test dataset. As a baseline,

ResNet-20 is trained from scratch as described in [39] until convergence with validation accuracy 92.0%, and test accuracy of 90.8%. After each pruning episode, the environment is retrained for a few epochs (8 w/o epochs learning) using mini-batch momentum SGD [37] with learning rate of 0.001, gamma 0.5, step size of 1900, batch size of 128, and $l2$ regularization. After learning to prune a layer, the model is fine-tuned for 150 epochs before moving to the next layer. The agent is also trained using mini-batch momentum SGD with fixed learning rate of 0.005 and batch size equal to the number of Monte-Carlo samples $M = 5$.

Pruning Order: We investigate four different strategies based on the pruning order and the agent's capability to prune layers simultaneously (layerwise or blockwise). We exclude the first convolutional layer since pruning it offers insignicant compression benefits, while damaging the learning ability of the model. When pruning a full residual block, we preserve the element-wise addition by zero-padding the output channels of the second layer in a residual block to restore the original number of output channels, such that the order of pruned channels is preserved. Fig. 4.2 shows results of pruning ResNet20 with loss bound b of 2%.

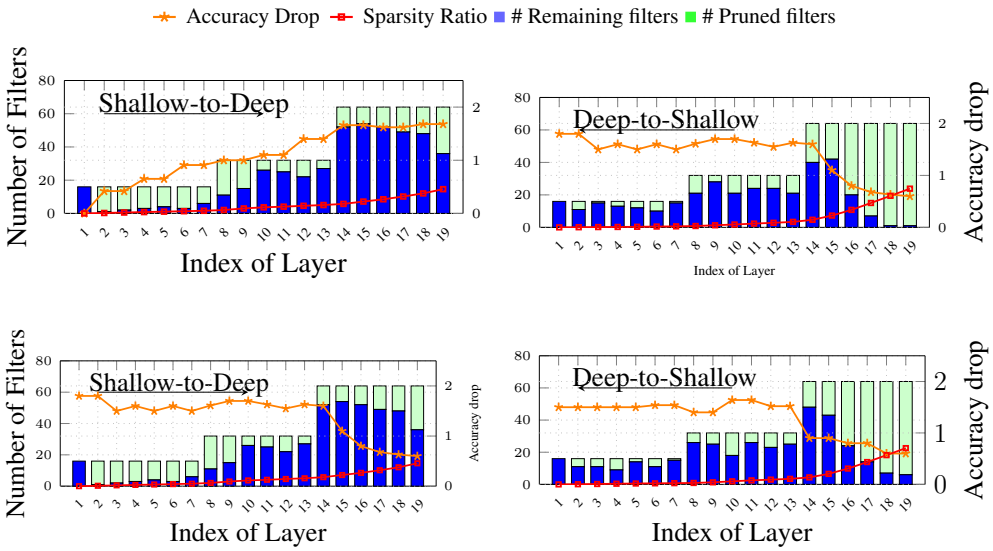


Figure 4.2: Exploration of filter pruning configuration based on order and agent complexity. Top-left and top-right subfigures indicate layer-wise pruning. Bottom-left and bottom-right subfigures indicate module-wise pruning.

In Fig. 4.2 (top-left), we perform layer-wise pruning following the forward pruning order (`conv2_1_1` $\rightarrow$ `conv4_3_2`). The agent starts pruning initial layers aggressively and struggles to find redundant filters in the deep layers (indicated by large blue bars). In Fig. 4.2 (top-right), we perform a similar layer wise pruning analysis for the backward pruning order (`conv2_1_1` $\leftarrow$ `conv4_3_2`). From Tab. 4.1, we observe that the backward pruning order results in higher CR with lower accuracy degradation (0.9%). In Fig. 4.2 (bottom-left) and 4.2 (bottom-right), we perform block wise pruning allowing the agent to prune the entire residual block simultaneously.

Similar to layer-wise pruning, we prune the residual blocks both in forward and backward orders. Lower compression ratio is observed when compared to layer-wise pruning, see also Tab. 4.1. Thus, we prune layer-wise in backward order as it results in lower accuracy degradation and high CR for the subsequent experiments.

Configuration	Pruning Order	Bound [%]	Learnable Epochs	Acc [%]	CR [×]
ResNet-20 [39]	-	-	-	90.8	1.00
L2PF (Block-wise)	Forwards	2.0	✗	89.9 (-0.9)	1.84
L2PF (Layer-wise)	Forwards	2.0	✗	89.6 (-1.2)	1.79
L2PF (Block-wise)	Backwards	2.0	✗	89.5 (-1.3)	3.38
L2PF (**Layer-wise**)	**Backwards**	2.0	✗	**89.9** (-0.9)	**3.90**
L2PF (Layer-wise)	Backwards	1.0	✗	90.2 (-0.6)	2.52
L2PF (**Layer-wise**)	**Backwards**	**2.0**	✗	**89.9** (-0.9)	**3.90**
L2PF (Layer-wise)	Backwards	3.0	✗	89.2 (-1.0)	4.53
L2PF (Layer-wise)	Backwards	4.0	✗	88.5 (-2.3)	7.23
L2PF (Layer-wise)	Backwards	2.0	✗	89.9 (-0.9)	3.90
L2PF (Layer-wise)	**Backwards**	**2.0**	✓	**89.9** (-0.9)	**3.84**

Table 4.1: Evaluating various configurations for L2PF to analyze the influence of exploration granularity, pruning order, accuracy bound w.r.t. prediction accuracy and compression ratio.

Reward Bound: In Tab. 4.1, we also evaluate the impact of the prediction accuracy and compression ratio by varying the agent's loss bound b. As we increase b, we obtain higher CR with lower prediction accuracy after fine-tuning. We choose b as 2% in the next experiments to maintain a trade-off between accuracy degradation and CR.

Reducing Fine-tuning: Previous experiments were conducted with fine-tuning epochs set manually to 8 at each exploration step. We allow the agent to decide the amount of fine-tuning time required to evaluate the pruning strategy based on the retrain epoch reward presented in Eq. 4.10. Fig. 4.3 shows a comparison of number of fine-tuning epochs required to decide the pruning strategy for each layer. Pruning with epochs learning achieves $1.71\times$ speedup in search time with a slight reduction in compression ratio, see Tab. 4.1.

Class Activation Maps: The discrete action space proposed by Huang et al. [21] and applied in L2PF allows the integration of Class Activation Maps (CAMs) [147] into the design process. CAM allows the visualization of RoI in an input image to identify the corresponding prediction label. Regions with red color denote the part with higher interest for CNN model and blue denotes regions with less importance w.r.t. the target label. Tab. 4.2 shows three example CAMs for the learned features of vanilla ResNet20 and the influence of L2PF pruning (backwards) on the learned features and thus the RoIs. The progression of discriminative regions of classes can be compared across pruning steps.

In the first row, the vanilla ResNet20 predicts the wrong class, *i.e. deer*. After pruning layer `conv3_3_2`, the RoI shifts towards the *trunk* of the *car* indicating the correct class. In the second row, the vanilla ResNet20 predicts the *ship* class. The agent tries to retain the prediction across different stages of pruning with high confidence. In the third row, the vanilla ResNet20 predicts the *truck* class. Accordingly, the pruned model at different stages also predict a *truck*. However,

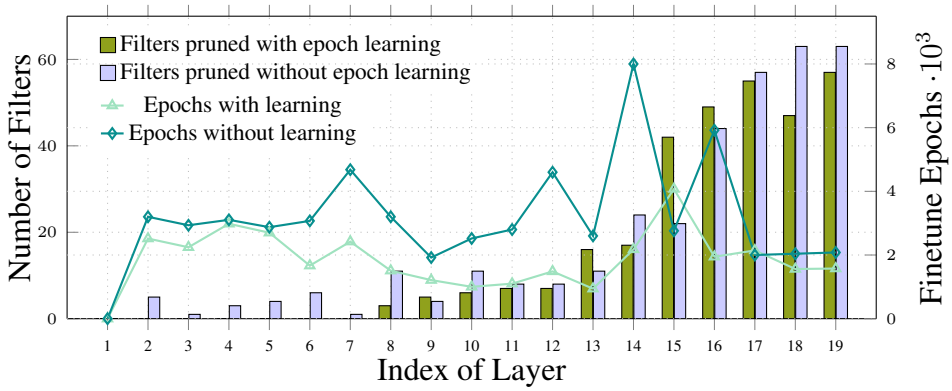


Figure 4.3: Reduction of GPU hours using epoch-learning based reward formulation. We highlight the number of filter pruned and fine-tuning epochs.

we can observe that the RoI becomes narrower indicating that the pruned model requires only few concentrated regions due to lower model capacity.

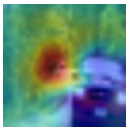
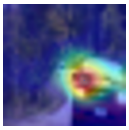
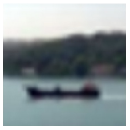
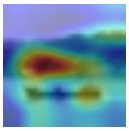
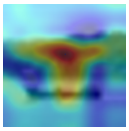
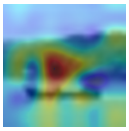
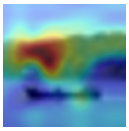
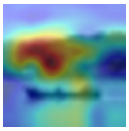
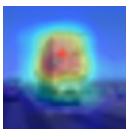
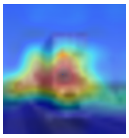
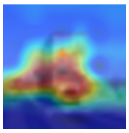
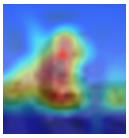
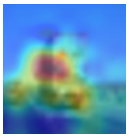
Input image	**ResNet-20 unpruned**	**Learning to Prune Faster (Backwards)** `conv4_3_2` →	`conv3_3_2` →	`conv2_3_2` →	`conv2_1_2`
 raw	 *deer*(0.53)	 *car*(0.99) →	 *car*(0.99) →	 *car*(0.99) →	 *car*(0.88)
 raw	 *ship*(0.99)	 *ship*(0.51) →	 *ship*(0.98) →	 *ship*(0.81) →	 *ship*(0.99)
 raw	 *truck*(0.99)	 *truck*(0.98) →	 *truck*(0.77) →	 *truck*(0.62) →	 *truck*(0.67)

Table 4.2: CAM visualization for three examples images from the validation dataset. Each column shows the CAM output after pruning, using backwards pruning order before model fine-tuning.

State of the Art Comparison: In this section, we compare the proposed L2PF with other RL-based state-of-the-art filter pruning works proposed in literature.

In Fig. 4.4 and Tab. 4.3, we compare our pruning configuration using layer-wise CR and final prediction accuracy with AMC [5], L2P [21], ALF [98]. We re-implemented L2P using forward pruning order with an accuracy bound b=2% to obtain pruning results for ResNet20. Compared to

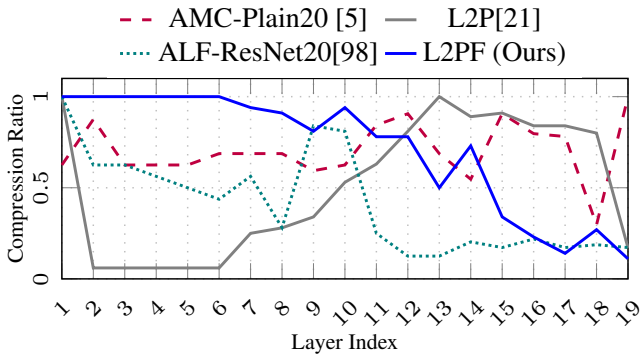


Figure 4.4: Comparing L2PF pruning statistics on ResNet-20 with State-of-the-Art.

Configuration	**Pruning Type**	**Acc** [%]	**CR** [×]	**Fine-tune [epochs]**
ResNet-20 [39]	-	90.8	1.00	-
AMC-Plain20 [5]	RL-agent	90.2	1.84	-
ALF [98]	In-train	89.4	3.99	-
L2P [21]	RL-agent	89.6	1.79	60.3K
L2PF (Ours)	**RL-agent**	**89.9**	**3.84**	**35.2K**

Table 4.3: Evaluating various configurations for L2PF to analyze the influence of exploration granularity, pruning order, accuracy bound w.r.t. prediction accuracy and compression ratio.

L2P, we obtain 0.3% better prediction accuracy, 2.11× higher CR and 1.71× less fine-tune epochs. ALF and AMC do not require fine-tuning during the pruning process. Compared to AMC's pruning implementation for Plain-20, we obtain 2.08× higher CR with 0.3% lower prediction accuracy. Compared to ALF, we achieve 0.5% better accuracy with comparable CR.

4.3.5 Discussion

In this section, we demonstrated an RL-based filter-wise pruning method which is both feature and time-aware. Our multi-task approach achieved high CRs, while minimizing the required GPU-hours and the accuracy degradation. The analysis on the sequence of layer-wise pruning led to the conclusion that backward (deep-to-shallow) pruning can produce dominating results compared with the existing state-of-the-art CRs, with minimal degradation in task specific accuracy. Finally, we visually analyzed the effect of our pruning technique with the help of CAMs to build a better understanding of our agent's pruning decisions. GPU hours for CNN compression can have many negative consequences on development cycles, profitability and fast exploration. The GPU-hour-aware approach presented can help mitigate this impediment and achieve a competitive advantage in active research fields such as autonomous driving.

4.4 In-Train Pruning

Techniques for pruning neural networks aim to remove redundant structural parameters like channels, kernels or individual weight elements of neural networks in order to decrease the memory requirements and accelerate the computation of the network at hand, while maintaining the network's accuracy. Most pruning methods [4, 5, 6] follow a three step approach: First, a model is learned to solve a task at hand. Second, this very model is pruned according to a separate objective function. Third, the model is fine-tuned to maintain the overall accuracy. This, however, significantly increases the computation effort (the GPU hours) for the pruning process. To improve upon this, recent research proposes RL agents to automate the process of finding the optimal model pruning strategies [5, 21, 27]. While, these learning-based compression techniques outperform pure heuristic-based approaches both in efficiency and compression ratio, they often do not yield an optimal solution.

In this section, we propose to incorporate the pruning process, i.e., learning an appropriate pruning mask, in the underlying optimization function of the training. We thereby break through the barrier between training and pruning, and circumvent the need for magnitude-based heuristics. The remainder of the section is structured as follows: In Sec. 4.4.1, we introduce trainable pruning masks which are appended during the CNN learning process. We further formulate task, HW specific loss terms to improve prediction accuracy and impose resource constraints during the training process. Sec. 4.4.4 provides detailed experiments to show the effectiveness of our *in-train* pruning approach. We further conduct a discussion about the proposed in-train pruning approach in Sec. 4.4.5.

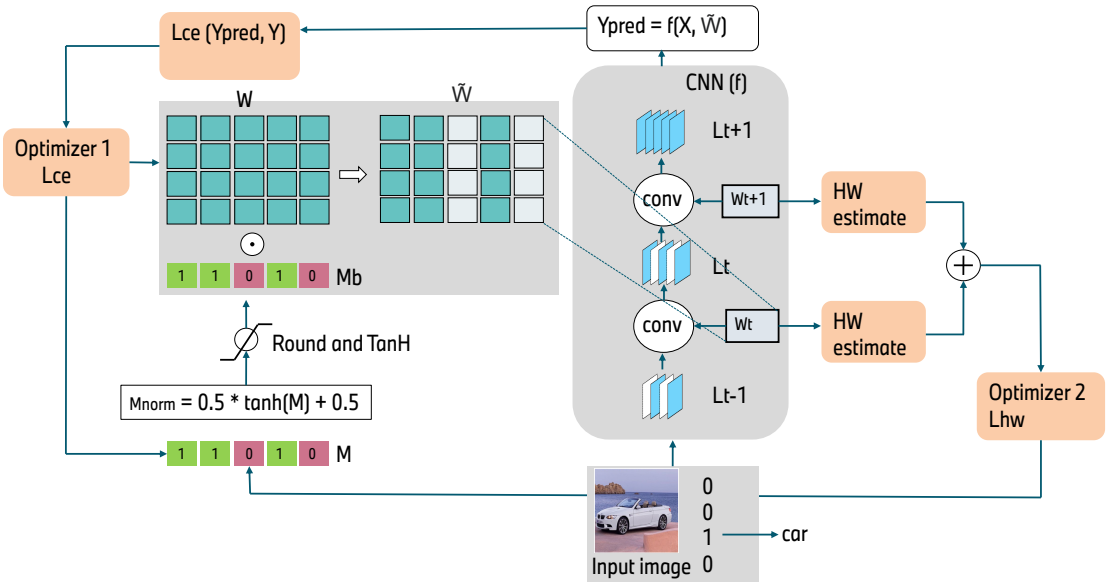


Figure 4.5: Methodology of the proposed In-train Pruning approach [28]

4.4.1 Trainable Prune Masks

Our approach compresses a model for reducing the computational complexity of a CNN as shown in Fig. 4.5. We aim for obtaining a pruning strategy directly when optimizing the CNN's weights W during the training process and thus save the optimization effort of additional post-train pruning. We adopt trainable pruning edges in the network using a binary mask $M_b \in \{0,1\}$ derived from a trainable continuous mask M. The weights W are canceled out if the corresponding dimension of the mask is 0 and left unchanged if it is set to 1: $W \odot M_b$. At each layer $l \in \{1, ..., N\}$ of an N-layer CNN, we append a binary pruning mask M_b^l to the network's weights W^l. All but the fully-connected layers have an input shape $L^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$, where H_i, W_i, and C_i indicate the spatial height, width and input channels, respectively. L^0, L^N represents the the input image I and classification output of the CNN respectively. The weights $W \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o}$ are the trainable parameters of the individual layers, where K_h, K_w and C_o refer to the kernel's dimensions, and the number of output channels/filters, respectively.

The binary masks for irregular weight pruning are structured as $M_b^l = \{0,1\}^{K_h \times K_w \times C_i \times C_o}$, kernel pruning requires masks as $M_b^l = \{0,1\}^{1 \times 1 \times C_i \times C_o}$, channel pruning requires masks $M_b^l = \{0,1\}^{1 \times 1 \times C_i \times 1}$ and filter pruning masks have the structure $M_b^l = \{0,1\}^{1 \times 1 \times 1 \times C_o}$. The size of the binary mask increases as the pruning tends to become more irregular leading to higher compression rates. However, irregular and kernel pruning demands dedicated hardware implementation [62] for load balancing and additional memory for mask indices, resulting in sub-optimal benefits on general-purpose platforms. The masked weights are obtained using the the Hadamard product $\odot$ along the pruning dimension as $\tilde{W}^l$ as shown in Eq. 4.11, that is, weights W are cancelled out if the corresponding dimension of the mask is 0 and left unchanged if it is set to 1.

$$\begin{aligned} \tilde{W}^l &= W^l \odot M_b,\ \tilde{W}^l \in \mathbb{R}^{K_h \times K_w \times C_{in} \times C_o} \\ M_b &= \mathrm{round}(M_{\text{norm}}) \\ M_{\text{norm}} &= 0.5 \cdot \tanh(\mathrm{M}) + 0.5 \end{aligned} \tag{4.11}$$

Our training scheme influences M_b using cross-entropy and HW objectives, by updating the continuous-valued and trainable masks M with the same shape as M_b. The trainable masks are introduced to incorporate the pruning objective into the training process. An optimizer, usually a Momentum optimizer (indicated as Optimizer 2 in Fig. 4.5) is used to update the trainable masks M with respect to prune loss $\mathcal{L}_{prune}$ (Eq. 4.12). We use $\tanh$, scale, and shift operations to derive the normalized masks M_{norm} in the value range of $[0,1]$ from the continuous-valued masks M. We then apply the round operation to restrict the mask values to the binary set $\{0,1\}$ as shown in Eq. 4.11. Any discrete parameter with a limited range set would introduce zero gradients. We use Straight Through Estimator (STE) similar to [148] to overcome the vanishing gradient effect and obtain updates for continuous masks M, later discretized to M_b for applying pruning decisions on the weights.

The adopted methodology for in-train pruning has been illustrated in Fig. 4.5. It shows a randomly sampled image from the training set being fed to a CNN $f(\cdot)$. Based on the network's prediction Y_{pred} and the true label Y, the cross-entropy loss is calculated. The trainable weight and mask parameters (W, M) are optimized with respect to cross-entropy and regularization loss.

Based on continuous-valued masks (M), the binary masks (M_b) are derived as shown in Eq. 4.11. The Fig. 4.5 also illustrates a channel pruning scenario where the channels in the weight matrix, corresponding to the 0 values in M_b are pruned. This automatically leads to filters in the previous layer getting pruned. The number of channels or filters remaining after applying the prune masks influences the layerwise estimates (number of parameters or MAC operations). These layerwise estimates are added to obtain the hardware estimate for the whole network. The hardware loss $\mathcal{L}_{HW}$ is formulated such that the specified target hardware constraints are met during pruning. The trainable masks are optimized at regular intervals jointly with respect to this HW loss $\mathcal{L}_{HW}$ and cross-entropy loss using a second optimizer. Hence, this method can automatically learn the layerwise sparsities through trainable masks. Also, it does not rely of magnitude of weights as the pruning heuristic as lowest magnitude weight pruning is sometimes found to be sub-optimal [149].

4.4.2 Task Specific and Hardware Specific Loss

We define the loss function that allows us to account for HW-specific compression objectives. The inference complexity of the CNN depends on the number of non-zero values in the binary pruning masks $\mathrm{sum}(M_b^l)$ at every layer l. We represent the HW inference complexity as a function of M_b^l given as $\psi_l(M_b^l)$. Increasing the number of zeros in the prune masks leads to a lower number of computations and parameters. However, this impacts prediction accuracy for extreme compression rates.

The latent weights W and the trainable masks M are optimized to improve the task-specific accuracy with respect to the sum of the cross-entropy loss $\mathcal{L}_{ce}$ and regularization loss $\mathcal{L}_{reg}$. The trainable prune masks M are also considered in the regularization loss to avoid too many binary masks M_b elements biased at the early stages due to exploding magnitude. We provide more details about the regularization $\mathcal{L}_{reg}$ in Fig. 4.8. In addition to this, we optimize the trainable masks M based on an auxiliary loss term $\mathcal{L}_{HW}$, which captures hardware HW benefits. It is important to select pruning masks which not only produce HW benefits but also allow smooth minimization of cross-entropy loss during the training process. Therefore, we formulate prune loss $\mathcal{L}^i_{\text{prune}}$ at step i in Eq. 4.12, which is an accumulation of $\mathcal{L}_{ce}$ and $\mathcal{L}_{HW}$. The HW loss $\mathcal{L}_{HW}$ is the difference between the complexity of neural networks at iteration i and a target constraint ψ^*. We accumulate the complexity of all the N layers to obtain the complexity of the whole network.

$$\begin{aligned} \mathcal{L}^i_{\text{Prune}} &= \mathcal{L}^i_{ce} + b \times \mathcal{L}^i_{HW} \\ \mathcal{L}^i_{HW} &= max(\frac{\sum_{l=1}^{N}(\psi_l^i)}{\sum_{l=1}^{N}(\psi_l^0)} - \psi^*, 0) \end{aligned} \tag{4.12}$$

We use the scaling factor b to control the convergence speed for the prune masks M during the training process. For extreme constraints such as 70% HW reductions, we use higher b=50 (more details in Tab. 4.6). The complexity of the neural network can be represented using the number of parameters or MAC operations. In Eq. 4.13, we represent the complexity by also incorporating the binary prune masks M_b. We first calculate the compression ratio μ_l for every layer l based on the number of non zeros present in the weight matrix. For this purpose, we introduce M^l_{base}

having the same dimensionality as M^l_{b}, consisting of all ones, representing the unpruned model. We observe that the number of zeros in the binary prune masks directly affect the complexity of layer l, which can be represented using either parameters $\psi_l(\text{params})$ or operations $\psi_l(\text{ops})$.

$$\begin{aligned} \mu_l &= sum(M^l_b)/sum(M^l_{\text{base}}) \\ \psi_l(\text{params}) &= K^l_w \times K^l_h \times C^l_{in} \times C^l_o \times \mu^l \\ \psi_l(\text{ops}) &= A^l_o \times B^l_o \times K^l_w \times K^l_h \times C^l_{in} \times C^l_o \times \mu^l \end{aligned} \tag{4.13}$$

Eq. 4.13 can be extended to pruning regularities such as channel/filter pruning, where inter-layer HW benefits must be taken into consideration. For channel pruning, we capture the inter-layer benefits by incorporating μ_l and μ_{l+1}, thereby reducing C^l_i and C^l_o respectively. We use an optimizer similar to that of standard training, such as Momentum/ADAM, to update the prune masks. As shown in Eq. 4.14 and Eq. 4.15, we approximate the gradients gm_{ce} and gm_{HW} derived from $\mathcal{L}_{ce}$ and $\mathcal{L}_{HW}$ to update the continuous prune mask M, incorporating STE as shown in the Fig. 4.5.

$$\begin{aligned} gm^l_{ce} = \frac{\partial \mathcal{L}_{ce}}{\partial M^l} &= \frac{\partial \mathcal{L}_{ce}}{\partial \tilde{W}} \cdot \frac{\partial \tilde{W}}{\partial M^l_b} \cdot \frac{\partial M^l_b}{\partial M^l_{\text{norm}}} \cdot \frac{\partial M^l_{\text{norm}}}{\partial M^l} \\ &\overset{\text{STE!}}{=} \frac{\partial \mathcal{L}_{ce}}{\partial \tilde{W}} \cdot \frac{\partial \tilde{W}}{\partial M^l_b} \cdot \frac{\partial M^l_{\text{norm}}}{\partial M^l} \end{aligned} \tag{4.14}$$

As shown in Eq. 4.15, the gradients updating prune masks due to $\mathcal{L}_{HW}$ scales depending on the baseline complexity ψ^l_{base} of the layer l. We derive ψ^l_{base} by setting μ_l = 1.

$$\begin{aligned} gm^l_{HW} = \frac{\partial \mathcal{L}_{HW}}{\partial M^l} = \frac{\partial \psi^l}{\partial M^l} &= \frac{\partial \psi^l}{\partial M^l_b} \cdot \frac{\partial M^l_b}{\partial M^l_{\text{norm}}} \cdot \frac{\partial M^l_{\text{norm}}}{\partial M^l} \\ &\overset{\text{STE!}}{=} \frac{\partial \psi^l}{\partial M^l_b} \cdot \frac{\partial M^l_{\text{norm}}}{\partial M^l} = \frac{\psi^l_{\text{base}}}{||M^l_{\text{base}}||} \cdot \frac{\partial M^l_{\text{norm}}}{\partial M^l} \end{aligned} \tag{4.15}$$

4.4.3 Choice of hyperparameters

The hyperparameters for in-train pruning are listed below:

- Prune begin step: Updating the trainable masks M with respect to the prune loss $\mathcal{L}_{\text{prune}}$ is started a few epochs after the normal training begins. For CIFAR-10 dataset, the mask update begin step is set at the 20^{th} epoch.
- Prune end step: The pruning masks are frozen some time prior to the end of the training process. This helps to determine the learned layerwise sparsities as well as fine-tune the pruned model during training itself.
- Mask update frequency: The trainable masks M are updated along with the weights, with respect to the cross-entropy loss $\mathcal{L}_{\text{ce}}$ and regularization loss $\mathcal{L}_{\text{reg}}$ during every training step till they are frozen at the *mask update end step*. The number of times the prune masks are updated with respect to the prune loss $\mathcal{L}_{\text{prune}}$ at every epoch is determined by the mask update frequency.

- Choice of Mask Optimizer: A suitable gradient descent optimizer like SGD, ADAM or Momentum optimizer can be chosen for optimization of the HW objective. We found that Momentum optimizer is well suited for this purpose.
- Learning Rate of Mask Optimizer: Since the weight optimizer follows a step learning rate, setting a fixed learning rate for the mask optimizer for prune loss optimization is easier to handle. A Momentum optimizer with a learning rate of 0.01 is found to perform reasonably well.
- Prune Loss constant b: b controls the trade-off between cross-entropy loss $\mathcal{L}_{\text{ce}}$ and hardware loss $\mathcal{L}_{\text{HW}}$. A large value of b accelerates the rate of optimization of the HW objective but may lead to sub-optimal results due to insufficient exploration. A small value of b, on the other hand, is not able to meet aggressive target constraints. Exploration for values of b for different target constraints have been presented in Table 4.6.

4.4.4 Experiments

The aim of in-train pruning is to train models from scratch while optimizing task accuracy and resource requirements in parallel. Model weights, as well as trainable prune masks (or importance values attached to masks), are trained on cross-entropy loss on the given task, using a Momentum optimizer with an initial learning rate to 0.1. For CIFAR-10 datasets, the training is performed for 300 epochs. The learning rate is scaled down by a factor of 10 at the 80^{th} and 160^{th} epoch. The trainable prune masks are updated with respect to HW estimates loss using another Momentum optimizer with a fixed learning rate of 0.01. The prune mask update with respect to the hardware objective starts at $E_{\text{Prune, Start}}$ = 20 and is done once every epoch. At $E_{\text{Prune, End}}$ = 240, the pruning masks are frozen, that is, the layer sparsities no longer change after $E_{\text{Prune, End}}$. For Imagenet dataset, $E_{\text{Prune, Start}}$ and $E_{\text{Prune, End}}$ are set to the 10^{th} and 80^{th} epochs respectively. The training is performed for 100 epochs with a learning rate drop at epochs 30, 60 and 90. However, these hyperparameters may be slightly adjusted based on training performance. Achieving high task-specific performance as well as the hardware objective depends on properly setting hyperparameter values, which have been studied in details in Sec. 4.4.3.

Pruning under FLOPs Constraints: We investigate the effectiveness of in-train channel pruning in Tab. 4.4 based on different constraints on the operation reduction metric. As shown in column 4 of Tab. 4.4, we set the target reduction factor for operations ψ^* from Eq. 4.12 to $\{1.0, 0.4, 0.3, 0.2\}$ for ResNet20 and ResNet56 on the CIFAR-10 dataset and $\{1.0, 0.7, 0.5\}$ for ResNet18 on the ImageNet dataset. We observe -2.91 pp and -0.53 pp of accuracy degradation for operation constraint ψ^* = 0.4 in ResNet20 and ResNet56 respectively. For an extreme target constraint ψ^* = 0.2, we observe an accuracy degradation of -4.3 and -1.99 pp for ResNet20 and ResNet56 respectively. We report the corresponding parameter reduction in column 6. In Tab. 4.4, we also investigate the consistency of these trends on more challenging datasets such as ImageNet. We observe a minor degradation of -1.31 and -3.47 pp for operation constraints of 0.7 and 0.5 on the ResNet18 model trained on the ImageNet dataset. However, it is difficult to achieve more drastic constraints for ResNet18 on ImageNet without considerable accuracy degradation.

Exploration of Pruning Regularities: We show different pruning regularity schemes for the proposed in-train pruning scheme. We observe that irregular weight pruning produces lower

Model	Dataset	Acc [%]	Ops Reduction Target	Ops Reduction Actual	Param Reduction
ResNet20	CIFAR-10	**92.47**	1.0	-	1.0
		89.56	0.4	0.38	0.68
		88.67	0.3	0.31	0.58
		88.17	**0.2**	**0.17**	**0.30**
ResNet56	CIFAR-10	**93.56**	1.0	-	1.0
		93.03	0.4	0.35	0.55
		92.38	0.3	0.28	0.50
		91.57	**0.2**	**0.18**	**0.37**
ResNet18	ImageNet	**68.53**	1.0	-	1.0
		67.22	0.7	0.69	0.88
		65.06	**0.5**	**0.45**	**0.78**

Table 4.4: In-train pruning for various operation constraints. We use ResNet20 and ResNet56 on CIFAR-10 dataset and ResNet18 on ImageNet dataset.

accuracy degradation (-1.16 pp, -0.38 pp) compared to structured channel pruning (-4.30 pp, -2.22 pp) for ResNet20 and ResNet56. Although weight pruning shows lower accuracy degradation for extreme target reductions, it is challenging to obtain inference benefits from such pruning regularities on general-purpose structured execution hardware like GPUs.

Model	Prune Regularity	Acc [%]	Ops Reduction Target	Ops Reduction Actual
ResNet20	baseline	92.47	1.0	-
	weight	91.31	0.2	0.16
	kernel	89.78	0.2	0.19
	channel	**88.17**	0.2	0.17
ResNet56	baseline	93.56	1.0	-
	weight	93.18	0.2	0.19
	kernel	92.25	0.2	0.21
	channel	**91.34**	0.2	0.21

Table 4.5: Exploring different pruning regularities for operation reduction factor ψ^*=0.2.

Training Behavior on CIFAR-10: The training behaviour which incorporates joint optimization of trainable weights and prune masks is analyzed in Fig. 4.6 for the CIFAR-10 dataset and in Fig 4.7 for the ImageNet dataset. We plot the Top1 accuracy and HW loss $\mathcal{L}_{hw}$, detailed in Eq. 4.12, across the training steps. The noisy behaviour in accuracy improvement can be seen across the iterations, indicating the joint optimization of the compression task (prune masks) and the learning task (weights). For the CIFAR-10 dataset, all the pruning constraints can be comfortably

met with accuracy curves following the same trend as in vanilla training, the operation constraint of ψ^* = 0.2 being relatively more challenging than the other two.

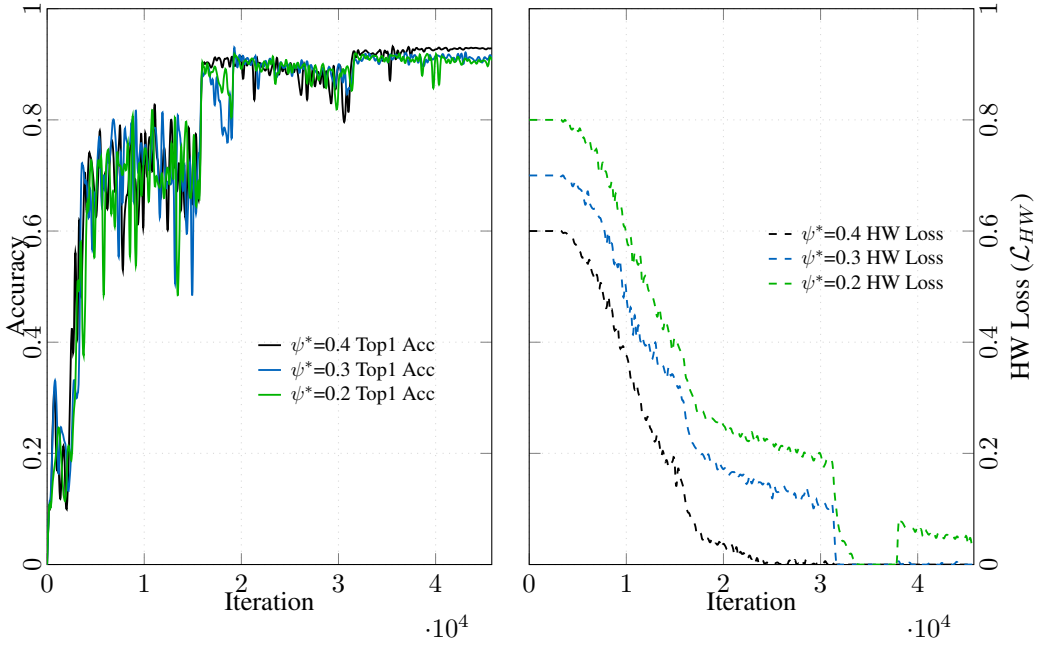


Figure 4.6: Comparison of in-train pruning behaviour across several training iterations for different operation constraints ψ^*=0.4, 0.3, 0.2 for CIFAR-10.

Training Behavior on ImageNet: The training behaviour for pruning constraints ψ^*=0.7, 0.5, 0.3 for ResNet18 on the ImageNet dataset is illustrated in Fig. 4.7. While the curves for ψ^*=0.7 display roughly the same behavior as models on CIFAR-10, higher constraints on FLOPs show considerable deviation. The fluctuations in HW Loss, particularly for 0.5 and 0.3 FLOPs constraints, indicate that scaling up to large datasets like ImageNet is much more challenging. While the optimizer for HW loss attempts to prune some channels, optimizer for cross-entropy loss attempts to grow them back, thereby resulting in the fluctuations in HW loss. At the 30^{th} epoch, when the learning rate for optimization w.r.t cross-entropy loss drops, the constraints are met more aggressively and at the same time, there is a drop is accuracy. The accuracy drops further at the 60^{th} epoch when learning rate w.r.t cross-entropy loss is further reduced. For a constraint of ψ^*=0.5, the accuracy values oscillate largely and is finally able to recover after the pruning masks are frozen. For ψ^*=0.3, although the accuracy is restored to some extent, it is still about $18\,\mathrm{pp}$ lower than the unpruned baseline model. This is a scenario where the RL based post-train pruning approach proved to be more effective that our proposed in-train approach.

Effect of Regularizing Prune Masks: As specified in Sec. 4.4.2, we incorporate trainable prune masks M in the regularization loss $\mathcal{L}_{reg}$ along with weights W. Regularizing the trainable masks M avoids early bias of binary masks M_b. We demonstrate this effect by studying the training behaviour for the proposed pruning scheme in Fig. 4.8. We constrain the number of operations to 30% of the baseline model and set b=10 to understand the behaviour of $\mathcal{L}_{HW}$ across training iterations. We observe that regularizing the trainable prune masks M (blue), achieves the

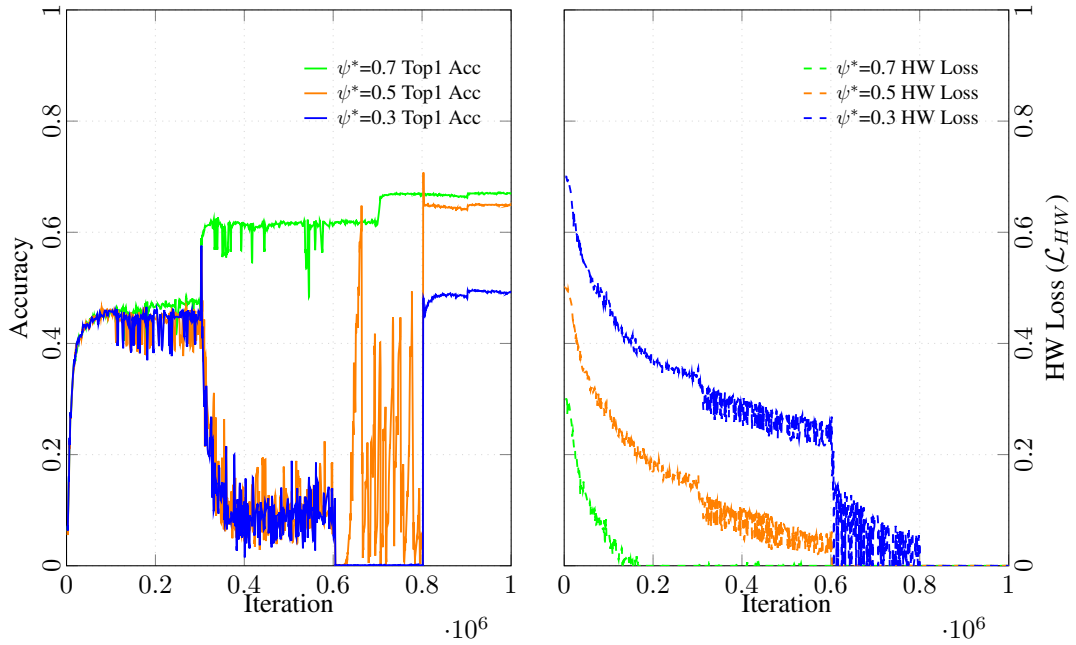


Figure 4.7: Comparison of in-train pruning behaviour across several training iterations for different operation constraints ψ^*=0.7, 0.5, 0.3 for ResNet18 on ImageNet.

target constraint ($\mathcal{L}_{HW} = 0$). We observe that there is only slight reduction in $\mathcal{L}_{HW}$, when prune mask M is not regularized. This occurs as the initialization of the trainable mask M is set to 1.0. Without regularizing the prune masks M and using a lower initialization such as $M_{\text{init}} = 0.3$ (green) would result in bias for the pruning decision during the early stages of training process. This would cause longer training time to achieve target constraints.

Exploration of b: The scalar constant b in prune loss $\mathcal{L}_{\text{Prune}}$ (Eq. 4.12) determines the relative importance of cross-entropy loss $\mathcal{L}_{ce}$ and hardware loss $\mathcal{L}_{HW}$. The exploration for b has been presented in Tab. 4.6. The value of b determines how fast or if at all the HW constraints are met. Tab. 4.6 shows that using lower values of b, drastic HW constraints cannot be met. In order to achieve 80% FLOPs reduction on CIFAR-10, the b value is set to 50. For ResNet18 model trained on ImageNet, a b value of 200 is necessary for 50% FLOPs reduction. Again, very high values of b might lead to very fast pruning without sufficient exploration, which in turn might affect the model accuracy.

Exploration of learning rate: Learning rate exploration plays a key role is ensuring that the maximum accuracy levels are reached. Vanilla training for ResNet20 with a step learning rate usually uses a learning rate drop every 80 epochs. However, for in-train pruning, we observed that avoiding a learning rate drop after the prune masks are frozen, that is after the 240^{th} epoch, always produced better results. In-train pruning results for ResNet20 with learning rate steps 80,

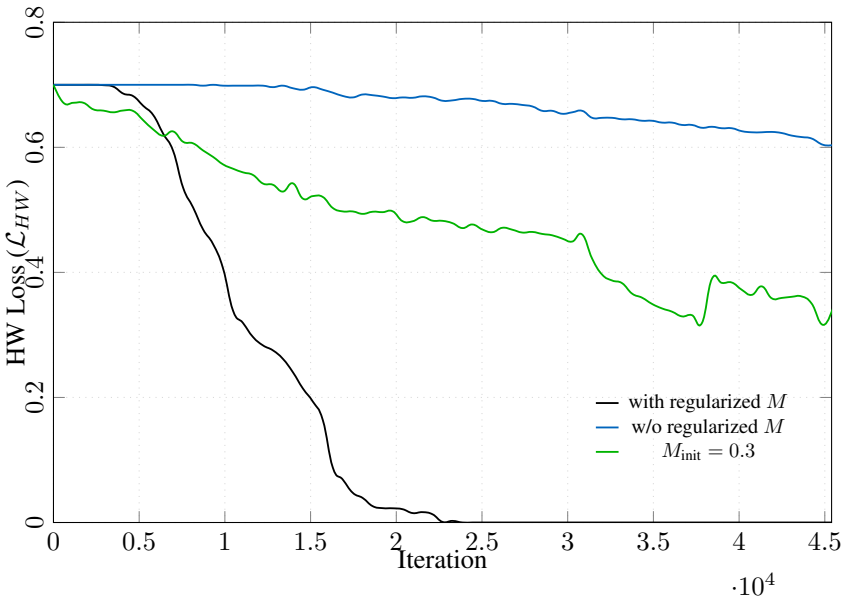


Figure 4.8: Comparison of HW loss $\mathcal{L}_{HW}$ across several training iterations for different operation constraints ψ^*=0.3 for different settings of continuous masks M.

160 and 80, 160, 250 have been presented in Tab.4.7. This portrays the importance of learning rate exploration to achieve optimum performance.

Comparison with Post-train Pruning We compare our in-train pruning approach with two state-of-the-art post-train pruning approaches: AMC [5].and ALF [98]. AMC is a RL-based pruning approach, whereas ALF uses autoencoders to learn the pruning strategy. Comparison of in-train pruning with these two approaches under FLOPs constraints is given below. Our method is found to achieve better performance than ALF for higher reduction in operations. For ResNet20 on CIFAR-10, our method achieves 0.16 pp improvement in accuracy for the same level of operations. Our method far surpasses ALF for ResNet18 on ImageNet. For the same level of reduction in operations (30%), the accuracy is better for our approach by 2.93 pp. In-train pruning achieves better accuracy on ResNet18 with more than 50% reduction in operations than ALF with 30% operations reduction. However, ALF achieves higher model size reduction for the same level of reduction in operations. Due to the use of autoencoders in each layer, it is difficult to scale ALF for very deep networks. Moreover, ALF supports only filter pruning while our method supports all pruning regularities.

Comparsion with AMC: The RL-agent proposed in the original work of AMC [5] is only suited for channel-wise pruning. It follows the standard three stage pipeline: *Taking a pre-trained model* $\rightarrow$ *Searching for a pruning strategy using a DDPG agent* $\rightarrow$ *Fine-tuning the pruned graph*. After an initial warmup phase of 100 episodes, the RL agent is allowed to search for 400 episodes to find the appropriate pruning strategy. The strategy or the layerwise prune ratios that yields the best reward, in this case accuracy on the validation set, is chosen. The pre-trained model is pruned based on the strategy devised by the RL agent. Finally, the pruned model is fine-tuned for 40 epochs with a learning rate of 0.01 for the first 20 epochs and 0.001 for the next 20 epochs.

Model	Dataset	b	Ops Reduction Target	Ops Reduction Actual	Acc [%]
ResNet20	CIFAR-10	10	0.2	0.33	88.65
		50	0.2	**0.17**	88.17
ResNet56	CIFAR-10	10	0.2	0.21	91.34
		50	0.2	**0.18**	91.57
ResNet18	ImageNet	10	0.5	0.80	65.98
		100	0.5	0.58	66.09
		200	0.5	**0.45**	65.06

Table 4.6: Exploring different values of scalar constant b in Prune Loss.

Model/ Dataset	learning rate drop epochs	Ops Reduction Target	Ops Reduction Actual	Acc [%]
ResNet20 CIFAR-10	80, 160	0.4	0.38	**89.56**
	80, 160, 250	0.4	0.35	89.35
	80, 160	0.3	0.31	**88.67**
	80, 160, 250	0.3	0.32	87.21

Table 4.7: Exploring different learning rates for In-train Pruning on ResNet20.

The in-train pruning results are compared with AMC based post-train pruning results in Tab. 4.9 for operations constraints of ψ^*=0.4, 0.3, 0.2 on CIFAR-10 and ψ^*=0.7, 0.5, 0.3 on ImageNet. In most of the cases, our approach performs as good as AMC based pruning. The difference in accuracies between the two approaches is marginal and can sometimes also be attributed to the higher level of compression that is achieved compared to the specified constraints. However, for 70% operations constraint on ResNet18, RL based pruning clearly outperforms in-train pruning to a large extent.

We also highlight the effectiveness of in-train pruning approach for a far more challenging and relevant task in autonomous driving, *Object Detection.* CenterNet with DLA-34 [113] backbone on the KITTI dataset [3] is used as the baseline model. We use a 75% and 25% split for the training and validation set respectively. We constrain the number of operations of CenterNet [113] model to 50% and compare the performance of post-train and in-train pruning approaches on the task of object detection. For baseline training as well as in-train pruning, the models are trained for 200 epochs using ADAM optimizer and step learning rate policy. We use an initial learning rate of 0.001 and decrease the learning rate by 0.01 at 60, 90 and 120 epochs. For in-train pruning, the trainable mask values are updated using a Momentum optimizer with a fixed learning rate of 0.01. The mask update begins at the 20^{th} epoch, stops at the 140^{th} epoch and uses a scalar constant b value of 50 for the prune loss $\mathcal{L}_{prune}$. AMC based post-train pruning uses 50 initial warmup episodes and 200 search episodes for the RL agent. Final fine-tuning is performed for 40

Model	Dataset	Method	Acc [%]	Ops [10^6]
ResNet20	CIFAR-10	ALF	89.4	15.8 (-61%)
		In-train	**89.56**	15.4 (-62%)
ResNet18	ImageNet	ALF	64.3	1239 (-32%)
		In-train	**67.23**	1251.7 (-31%)
		In-train	**65.06**	**816.33 (-55%)**

Table 4.8: Comparison of ALF with In-train Pruning.

Model	Dataset	Method	Operations Reduction		
			0.4	**0.3**	**0.2**
ResNet56	CIFAR-10	AMC [5]	92.86	92.68	91.66
		In-train	**93.03**	92.38	91.57
ResNet20	CIFAR-10	AMC [5]	90.70	89.25	87.61
		In-train	89.56	88.67	**88.17**
			0.3	**0.5**	**0.7**
ResNet18	ImageNet	AMC [5]	67.66	65.59	62.98
		In-train	67.23	65.06	49.62

Table 4.9: Comparison of AMC [5] Pruning with In-train Pruning.

epochs with a learning rate of 10^{-3} for the first 20 epochs and 10^{-5} for the remaining 20 epochs. We report the 2D mAP for validation data on easy, medium, and hard constraints of the car class in Tab.4.10.

For AMC based pruning, we report the GPU hours required for pre-training the baseline model, pruning search and the final fine-tuning. We observe that the pruning search with episode-wise fine-tuning (2 epochs of fine-tuning per episode) produces the best mAP among pruned models for easy and medium constraints but is extremely costly in terms of GPU hours. Our approach performs pruning during the training process, thereby saving GPU hours by 1.49× and 4.45×, as compared to AMC with and without fine-tuning during the pruning steps respectively. Also, it produces the best mAP values for hard constraints, exceeding the baseline by 6.56 pp.

In Fig. 4.9, we perform qualitative comparison of the predictions obtained using in-train pruning (right) with the baseline model (left). *Green* boxes indicate *ground truths* and *Blue* boxes indicate *predictions*. In the first row, we observe that the pruned model doesn't predict the bounding box for the car present at the far right corner. However, we observe that the overlap between in-train pruned predictions and ground truth bounding boxes is higher even when compared to the baseline model. This also justifies the higher mAP for the hard constraint in Tab. 4.10.

Method	mAP (car) [%]			GPU Hours
	Easy	Medium	Hard	
Baseline	89.24	**80.56**	71.65	22.87
AMC (w/o episode finetuning)	87.38	79.44	70.79	22.87+20.72+4.3 = 47.89
AMC (with episode finetuning)	**89.26**	80.46	71.55	22.87+112.05+7.7 = 142.62
In-train Prune	85.83	78.94	**78.21**	**32**

Table 4.10: Kitti validation for post-train vs in-train pruned CenterNet with 50% operations constraint.

Figure 4.9: Qualitative comparison of In-train pruning (right) with its baseline predictions (left).

4.4.5 Discussion

In this section, we incorporate the pruning process in the underlying optimization function of the training, an approach which we term as *in-train pruning*. We thereby break through the barrier between training and pruning, and save the computational effort required during the search stage. The pruning search stage played an important role in the approaches discussed in Sec. 3.4,3.5 and 4.3. Our joint formulation of the learning and pruning objectives eliminates the search stage and thereby allows us to find a trade-off between task-specific accuracy and compression rate. Moreover, this method is found to be most effective for structured pruning regularities. As structured pruning is more advantageous on general-purpose accelerators such as GPUs, this strengthens the motivation for the proposed in-train pruning scheme. We also eliminate the need for a pretrained model. We demonstrate that our method achieves 80% reduction of MAC operations in ResNet models. The scalability of this method has been tested for image classification on large datasets like ImageNet. This method also yields promising results for CenterNet pruning for object detection on the KITTI dataset.

The proposed approach in this section is limited to model pruning. In Sec. 4.5, we realize a mixed precision quantized CNN configurations using a similar in-train approach. As discussed in Chapter.3, it is important to incorporate HW-metrics during the compression process. The HW-estimates need to be differentiable during the training process to influence the CNN model. We also achieve this objective in Sec. 4.5. We further extend this approach to realize robust pruning configurations in Sec. 5.4.

4.5 In-train Mixed Precision Quantization

Model compression techniques such as quantization [7, 15, 8], pruning [5, 82, 27], and knowledge distillation [9] reduce the memory footprint of CNNs and speed up their computation. Quantization, in particular, has become a standard technique in both industry [54, 28] and academia [7, 131], typically applied before deploying CNNs in embedded settings. The benefits of quantization are manyfold, ranging from reducing the bit-width of weights and activations to shrink the model's size, to simplifying the arithmetic computation units on HW and lowering the energy consumed by on-chip and off-chip data movement [89]. Quantization is typically applied as a post-training calibration method, which takes in a full-precision pretrained model and compresses it to a lower bit-width representation with iterative steps of fine-tuning.

Alternatively, QAT methods are capable of producing quantized CNNs *during* the training process [50, 7]. These are typically uniform in terms of bit-width assignment, fixing the entire CNN's representation to a pre-determined number of bits. However, different layers contribute differently to the accuracy and efficiency of a network [15], justifying the use of different quantization degrees for different layers of the CNN. To obtain HW-friendly layer-wise quantization strategies, post-training quantization methods use search techniques, such as RL [13] or ES [14], bringing back the costly post-training GPU hours. In this section, we reduce the execution metrics for model inference by searching for optimal bit-widths directly *during* the training process and produce **dominant solutions** in terms of prediction accuracy and model complexity, when compared to uniform bit-width assignments and post-training methods. With our approach, we ensure similar amount of GPU-hours as training as baseline model. We also incorporate the HW-awareness by predicting estimates from a bit serial accelerator using guassian regressor.

4.5.1 Trainable Bitwidths

Without loss of generality, the activation feature map $A^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$ is considered as the input to a convolutional layer $l \in [1, ..., L]$, where W_i, H_i and C_i describe the dimensions of width, height and input channels. A^0 and A^L are the input image and the prediction of the CNN, respectively. The weight matrix $W^l \in \mathbb{R}^{K_w \times K_h \times C_i \times C_o}$ consists of kernels of shape $K_w \times K_h$, and C_o output channels. A convolution operation of tensors W^l and A^{l-1} results in the output $A^l \in R^{H_o \times W_o \times C_o}$. The output features of the final layer A^L can be used for the computation of the task-specific accuracy ψ, by comparing it to the dataset labels.

Every layer l is associated with weight and input activation bit-widths, represented as b_w^l and b_a^l respectively. The compressed CNN can be obtained by selecting the optimal quantization tuple $(b_w^l, b_a^l)\ \forall\ l$. In this paper, we target two objectives: (1) reducing the bit-width of weights and activations during the training process to lower the computational complexity of a neural network, and (2) improving the HW-awareness by directly finding the quantization strategy based on real HW metrics, such as latency and memory accesses. Both can be efficiently achieved by formulating a joint optimization problem as shown in Fig 4.10.

We aim to obtain an efficient quantization strategy directly when training the network's weights W to circumvent any additional effort of post-train quantization search. More generally, b is the bit-width of the quantized datatype, weights or activations. The real-valued data are represented by 2^b unique values in the fixed-point quantization domain. The mapping of a data element x

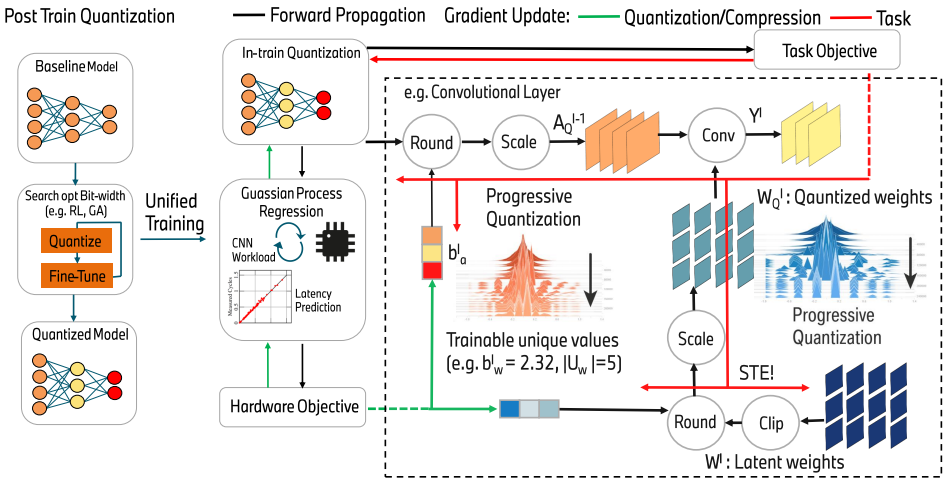


Figure 4.10: Depiction of post-train quantization approaches (left) in comparison to the proposed approach (right). Optimal precisions are determined through progressive quantization.

(*i.e.* weight or activation) onto a quantized value $\mathtt{x_q}$ is expressed in Eq. 4.16. This is similar to the linear quantization methods proposed in [50, 7]. Firstly, x is clipped between $[-c, +c]$, where c is a trainable variable for every layer and is determined by the task-specific loss function of the CNN model [7]. Based on the determined c for a given datatype, we define a scaling factor $s = c/(2^{b-1} - 1)$. For activations, we clip the values between the range of $[0, +c]$, instead of $[-c, +c]$, due to the non-linear activation function (ReLU). $\mathtt{x_q}$ approximates the continuous domain of x into the discrete values $\mathtt{x_q} \in \{0,\ s,\ 2s,\ 3s, ...,\ (2^b - 1) \cdot s\}$. During backpropagation, the gradient of the $Round$ operation vanishes, therefore we estimate it in order to update the real-valued weights during the training phase. In the simplest case, the estimated gradient g_x could be obtained by replacing the derivative of $Round$ with the identity function (see Eq. 4.16). This is referred to as the STE [150].

$$\mathtt{x_q} = Round(\mathtt{Clip}(x, 0, c) \cdot \frac{(2^b - 1)}{c}) \cdot \frac{c}{(2^b - 1)} \ ;\ g_x g_{\mathtt{x_q}} \cdot 1_{x \leq c} \tag{4.16}$$

Varying the number of bits b of each datatype, for every layer, can change the prediction accuracy ψ by several percentage points. One way to determine the best configuration is to perform exploration using an RL-agent [13] or an ES algorithm [14, 15]. However, this search could lead to excessive GPU hours, given the need for iterative fine-tuning for every exploration step. This motivates *learning* the most efficient allocation of datatype precision of each layer during the training process itself. There are challenges to achieve this task due to two important reasons: (1) Devising a training scheme which directly changes the bit-width b could lead to sudden fluctuations in the discrete weight distribution, resulting in unstable gradient updates, (2) The bit-width b only considers integer values, e.g. $b \in [1, 2, 3, ..., 8]$ for an accelerator supporting maximum of 8-bit fixed-point multiplications. Using another STE to round the parameters in the forward pass while retaining the original float values in the backward propagation could lead to

further gradient approximation, producing sub-optimal prediction accuracies ψ. We tackle these challenges by determining the set of unique values U required to represent all x, as shown in Eq. 4.17.

$$\begin{aligned} \tilde{\mathtt{x}}_{\mathtt{q}} \quad &= \quad Round(\mathtt{Clip}(x, 0, c) \cdot \frac{|U|}{c}) \times \frac{c}{|U|} \\ g_{|U|} \quad &g_{\tilde{\mathtt{x}}_{\mathtt{q}}} \cdot \big(\frac{-c}{|U|^2} \cdot Round(\mathtt{Clip}(x, 0, c) \cdot \frac{|U|}{c}) + \frac{\mathtt{Clip}(x, 0, c)}{|U|}\big) \end{aligned} \tag{4.17}$$

We avoid using another gradient approximation by allowing the cardinality $|U|$ to be a real-valued trainable parameter. Note that our approach in Eq. 4.17 differs from Eq. 4.16 by not restricting the values to an integer number b in 2^b, allowing $|U|$ to have an arbitrary number of unique elements to represent the trainable parameters. We introduce a hyperparameter $E_{\text{Quant, Start}}$ which represents the epoch at which the learning process for the bit-widths is started. This allows smooth, float-point values for the size of $|U|$, progressively lowering the *projected* cardinality of U for each layer's datatypes, until $E_{\text{Quant, Stop}}$ is reached. The number of unique values $|U|$ is updated by cross-entropy loss $\mathcal{L}_{ce}$ based on the gradient $g_{|U|}$ as shown in Eq. 4.17. The unique values in U required to represent x increases based on the rounding error $(\frac{x \cdot |U|}{c} - Round(\frac{x \cdot |U|}{c}))$. The HW loss objective $\mathcal{L}_{HW}$ is captured by fractional bit-widths (e.g. 3.5-bits) during the initial stages of training (i.e. between $E_{\text{Quant, Start}}$ and $E_{\text{Quant, Stop}}$ epochs). This also leads to progressive quantization which lowers the gradient approximation at the initial stages of the training, thereby improving the learning capability of the neural network. We gradually determine the optimal bit-widths based on $\mathcal{L}_{ce}$ and $\mathcal{L}_{HW}$ objectives as we approach the end of the training. We round the number of unique values $|U|$ to the nearest power-of-two in the middle of training process ($E_{\text{Quant, Stop}}$), deriving the optimal bit-width b as $Round(\log_2 |U|)$.

4.5.2 Task Specific and Hardware Specific Loss

We define the constrained loss function $\mathcal{L}_{HW}$ as shown in Eq. 4.18, to account for HW-specific compression objectives. b^* and b^{max} represent constrained and maximum supported bit-widths, respectively. The inference complexity of the CNN depends on the number of bits assigned to the weights b_w and activations b_a for each layer $l \in [1, ..., L]$. We represent the workload shape of layer l as l_{shape} and HW inference complexity as a function of l_{shape}, b_w^l and b_a^l given as $\varphi_l(l_{\text{shape}}, b_w^l, b_a^l)$. We use the scaling factor v to control the convergence speed for the optimal quantization strategies $\{(b_w^1, b_a^1), ..., (b_w^L, b_a^L)\}$ during the training process.

$$\begin{aligned} \mathcal{L}_{\text{total}} &= \mathcal{L}_{ce} + \mathcal{L}_{reg} + v \times \mathcal{L}_{HW} \\ \mathcal{L}_{HW} &= max\big(\frac{\sum_{l=1}^{L}(\varphi_l(l_{\text{shape}}, b_w^i, b_a^i) - \sum_{l=1}^{L} \varphi_l(l_{\text{shape}}, b_w^*, b_a^*))}{\sum_{l=1}^{L}(\varphi_l(l_{\text{shape}}, b_w^{max}, b_a^{max}))}, 0\big) \end{aligned} \tag{4.18}$$

Decreasing the number of bits b_w^l and b_a^l leads to a lower number of Bit Operations (BOPs) for a given layer l, as defined in Eq. 4.19.

$$\varphi_{\text{BOPs}}^l = X_o^l \times Y_o^l \times K_x^l \times K_y^l \times C_i^l \times C_o^l \times b_w^l \times b_a^l \tag{4.19}$$

4.5.3 Differentiable HW awareness

Many HW-aware compression works use HIL setups and/or HW-measurement look-up tables to integrate the actual HW performance into the optimization loop [14, 100, 13, 5]. Although this approach is valid for agents which require a simple reward value for their CNN optimization decisions, it cannot be extended to the gradient-descent optimization used in this work for in-train quantization. For the training optimizer to work seamlessly with HW-based loss minimization, the HW measurements need to be provided through a differentiable function. By nature of a differentiable function providing the HW-estimates, intermediate values for quantization can also be supported during the in-train quantization's progressive bit-width reduction. For example, 3.5-bits does not reflect any executable computation bit-width on real HW. During smooth, progressive in-train quantization, such bit-widths may appear, which need to have a sensible loss value associated with them to guide the gradient-descent-based training optimizer.

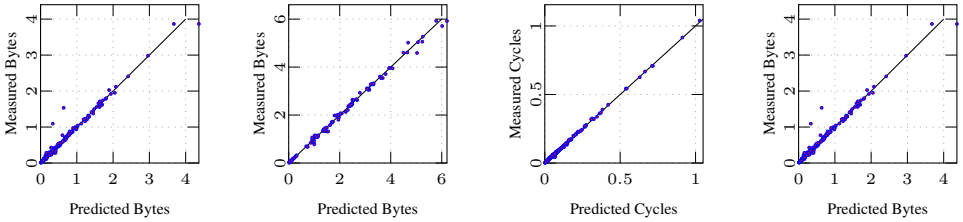
Gaussian process regression provides the means to construct a differentiable HW estimator. This injects HW-awareness into the chain-rule for the SGD, allowing it to set the layer-wise quantization values $(b_w^l, b_a^l)\ \forall\ l$. A GP prior is trained on measurements φ_{HW} collected on real HW, with respect to different computation workloads ρ. Starting with the covariance matrix K shown in Eq. 4.20, we use a squared exponential kernel, inspired by the approach in [22]. σ and ℓ represent the amplitude and lengthscale of the GP's kernel, respectively. ρ indicates the workload features, i.e. the convolutional layer's dimensions and bit-widths. Considering a General Matrix-Matrix Multiplication (GEMM) execution of a convolutional layer, ρ is the vector of features representing rows, columns and inner product (depth) of the matricies, as well as the bit-widths of weights and activations.

$$\begin{gathered} K(\rho,\rho') = \sigma^2 exp(-\frac{||\rho-\rho'||^2}{2\ell^2}), \text{ where } \rho = (row, depth, col, b_w, b_a) \\ \varphi_{HW} \sim \mathcal{GP}(m(\rho), K(\rho,\rho')) \end{gathered} \tag{4.20}$$

Based on the GP prior in Eq. 4.20, a predictive function can also be described by a mean and a covariance matrix. To guarantee the GP regressor is differentiable, we must assert that the covariance function K is differentiable. This condition is fulfilled by our squared exponential kernel, as shown in Eq. 4.21.

$$\frac{\partial K(\rho,\rho')}{\partial\rho\partial\rho'} = \frac{\sigma^2}{\ell^4}(\ell^2 - (\rho-\rho')^2)exp(-\frac{||\rho-\rho'||^2}{2\ell^2}) \tag{4.21}$$

The GP regressor's HW predictions φ_{HW} can be used in the HW loss formulation $\mathcal{L}_{HW}$, presented in Eq. 4.18. The differentiable GP regressor provides $\frac{\partial\varphi}{\partial\rho}$ during backpropagation, which links in the chain-rule, allowing the in-train quantization SGD to manipulate the $(b_w^l$, $b_a^l)\ \forall\ l$ through the ρ gradients, thereby minimizing the latency and/or DRAM accesses of the inference execution on HW. In Fig. 4.11, we present the performance of the GP regressor on unseen validation workloads from ResNet20-CIFAR and ResNet18-ImageNet, with varying b_w^l, b_a^l. The high-accuracy of the HW measurement predictions can clearly guide the in-train quantization algorithm to make decisions on minimizing the HW loss $\mathcal{L}_{HW}$, which reflect real reductions in latency and DRAM accesses on the final HW accelerator.



((a)) ResNet-CIFAR Latency **((b))** ResNet-CIFAR DRAM **((c))** ResNet-ImageNet Latency **((d))** ResNet-ImageNet DRAM

Figure 4.11: Validating the performance of the GP regressor on unseen CNN workloads from ResNet20-CIFAR and ResNet18-ImageNet for prediction of HW latency and DRAM accesses.

The BISMO [131] based FPGA design allows us to derive HW-metrics for convolutional and dense workloads with different quantization configurations. The convolutional and dense workloads are transformed into GEMM workloads. The convolution workloads can be lowered into a general matrix multiplication (GEMM), by representing the W^l and A^{l-1} tensors as 2-D matrices $\texttt{Mat}_W$ and $\texttt{Mat}_A$, according to Eq. 4.22. The dimensions m and n represent the rows and columns of each matrix.

$$\begin{gathered}\texttt{Mat}_W \in \mathbb{R}^{m_W \times n_W},\ \texttt{Mat}_A \in \mathbb{R}^{m_A \times n_A} \\ m_A = n_W = C_i \times K_x \times K_y, \\ m_W = C_o,\ \ n_A = X_o \times Y_o\end{gathered} \tag{4.22}$$

$$A^l = \texttt{Conv}(W^l, A^{l-1}) = \texttt{Mat}_W \times \texttt{Mat}_A \tag{4.23}$$

Note that in Eq. 4.23, transposing both matrices and switching their order would also produce the convolution result. HW accelerators typically exploit data reuse to minimize the number of costly off-chip DRAM calls they need to perform during execution. For example, if one column from the right-hand side (RHS) matrix is to be computed against every row from the left-hand side (LHS) matrix, the accelerator can load the RHS column once and stream through all the LHS rows, until the column has been used exhaustively for the GEMM computation. The BISMO scheduler executes GEMM workloads in this manner, although it is agnostic to the workload being provided. We could then consider forcing the weights $W^l\ \forall\ l \in L$ to remain in the RHS matrix throughout the execution, which would result in a weight-reuse schedule (WRS). Conversely, maintaining activations A^{l-1} in the RHS, would result in an activation-reuse schedule (ARS).

4.5.4 Experiments

We evaluate the proposed in-train quantization technique on CIFAR-10, CIFAR-100 [42], and ImageNet [1] datasets. We use ResNet20 and ResNet56 as baseline models for the CIFAR-10, CIFAR-100 datasets, and ResNet18 as a baseline model for the ImageNet dataset. In all the experiments, we set the first and last layer as 8-bit quantization. If not otherwise mentioned, all hyper-parameters specifying the task-related training were adopted from ResNet's base

implementation [39]. To train the GP regressor presented in Sec. 4.5.3, latency and DRAM accesses measurements were collected from two BISMO [131] bit-serial accelerators synthesized on an FPGA board with a Z7020 SoC. For CIFAR-10 and CIFAR-100 experiments, an array of 8×8 processing elements (PEs), with 256 lanes was synthesized. For ImageNet workloads, we synthesize HW with 6×6 PEs and 256 lanes each. The collected measurements include all layer shapes in the considered CNNs, and all possible bit-width combinations for weights and activations. For Tab. 4.14 and 4.15, we report the DRAM accesses and latency by executing the quantized CNN workloads directly on the BISMO accelerator.

Mixed Precision Quantization: We investigate the effectiveness of our in-train quantization approach in Tab. 4.11, based on different constraints φ^* on the number of target BOPs. We perform in-train quantization by constraining the number of BOPs to the complexity equivalent of uniform 4-bit (see row 3, 4 in Tab. 4.11). By introducing trainable bit-widths for both weights and activations, we produce $0.1\,\mathrm{pp}$ and $1.1\,\mathrm{pp}$ better prediction accuracy than uniform 4-bit quantization for ResNet20 and ResNet56, respectively. We highlight the importance of the scaling factor v (Eq. 4.18) for lower BOPs constraints φ^* = 333, 1024. We observe that the target constraint φ^* is only met when the scaling factor is increased. We observe a reduction in BOPs by $1.8\times$ and $1.5\times$ with negligible accuracy degradation compared to uniform 4-bit quantization for ResNet20 and ResNet56, respectively. Furthermore, we report the training cost required to obtain the baseline and mixed precision models. Our in-train quantization scheme learns optimal bit-widths with minimal overhead in training time, i.e. 6%, 3% extra cost compared to uniformly quantized ResNet20 and ResNet56 respectively.

Model/ Dataset	Mixed Precision Weight	Mixed Precision Activation	Scaling Factor (v)	Avg. Bitwidth W_{bit}	Avg. Bitwidth A_{bit}	Constraint φ^* BOPs (M)	Actual φ BOPs (M)	Top-1 (%)	Training Cost (GPU hours)**
ResNet20 CIFAR-10	✗	✗	-	8	8	-	2592	92.4	2hr 43min
	✗	✗	-	4	4	-	666	92.2	
	✓	✗	1.0	4.0	4	666	679	91.0	2hr 53min
	✓	✓	1.0	4.0	4.0	666	651	**92.3**	
	✓	✓	0.1	7.1	7.1	333	2028	92.3	
	✓	✓	0.5	2.9	4.8	333	575	92.3	
	✓	✓	1.0	2.3	3.7	333	**376**	**91.5**	
ResNet56 CIFAR-100	✗	✗	-	8	8	-	8025	71.1	7hr 29min
	✗	✗	-	4	4	-	2029	70.5	
	✓	✗	1.0	4.0	4	2029	2019	**72.2**	7hr 46min
	✓	✓	1.0	3.7	5.1	2029	2123	71.6	
	✓	✓	0.5	3.7	4.8	1024	2201	71.9	
	✓	✓	1.0	3.7	4.5	1024	1739	71.3	
	✓	✓	2.0	2.6	3.2	1024	**1311**	**69.9**	

** Training cost is measured on a NVIDIA TITAN-X GPU

Table 4.11: Influence of scaling factor v in $\mathcal{L}_{HW}$ for BOPs-constrained in-train quantization.

Training behaviour: In Fig. 4.12, we demonstrate the training behaviour of the proposed in-train quantization approach. We highlight the training curves for the ablation study performed in Tab. 4.11, to understand the influence of the scaling factor v on the final prediction accuracy and BOPs reduction. We consider ResNet20 and ResNet56 trained on the CIFAR-10 and CIFAR-100

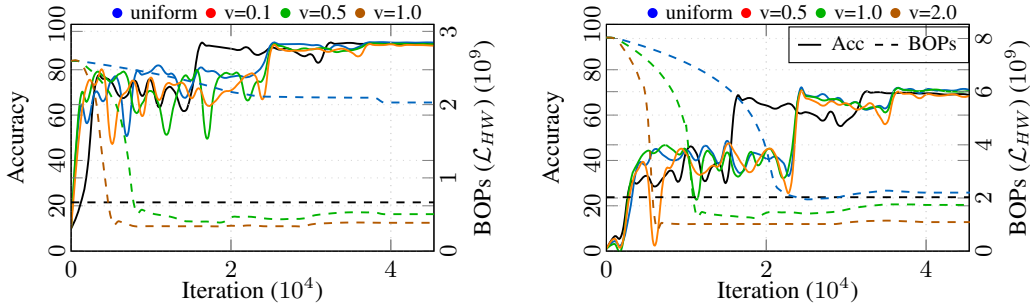
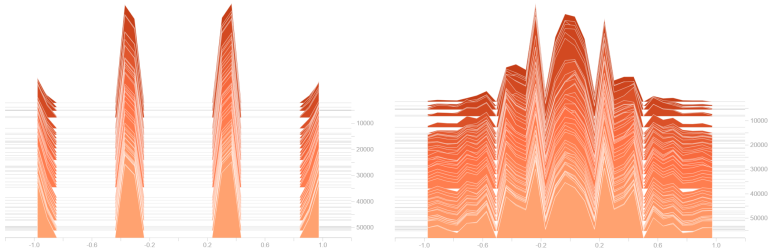


Figure 4.12: Comparison of in-train quantization behaviour for ResNet20 (left) and ResNet56 (right) with uniform quantization and different scaling factors.

datasets to observe the improvement in accuracy and reduction in BOPs. For the 4-bit uniform quantization (indicated in blue), we observe that the BOPs remain constant across the training steps. We constrain our mixed-precision models to a achieve 2× BOPs reduction compared to the uniform 4-bit model (see Tab. 4.11). We train uniformly quantized and constrained mixed-precision models for 300 epochs with a step learning decay policy. For uniform quantization, we decay the learning rates by 0.1 at 80, 160 and 240 epochs. For the constrained mixed-precision models, we decay the learning rate at 120, 180 and 240 epochs. We ensure convergence in the quantization strategy before the learning rate decay. For our constrained mixed-precision models (indicated by red, green, orange), we observe reduction in BOPs across the training steps. We increase the scaling factor v to achieve the desired BOPs constraint. For ResNet56, we observe 1.2×, 1.6× reduction in BOPs compared to the uniform 4-bit configuration for scaling factor v= 1.0, 2.0, respectively, with no degradation in accuracy.

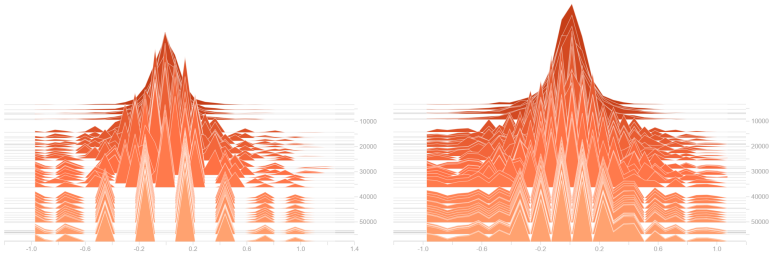
In Fig. 4.13 and Fig 4.14, we compare the distribution of W_q for our approach against the distribution under uniform quantization. We observe that uniform quantization training [7] does not change the number of unique values (peaks) across the training steps, allocating a *fixed* number of values from the start of the training. In our approach, we observe *progressive* quantization, starting from a smooth normal distribution and slowly converging to discrete peaks of quantized values. This allows larger gradient flow during the initial stages of the training and improves the trade-off between prediction accuracy and HW complexity for the resulting mixed-precision neural network.

Mixed Precision Segmentation: The semantic segmentation task is critical to applications in robotics and autonomous driving. High-quality segmentation is computationally complex by several orders of magnitude, when compared to classification tasks. This complexity is due to the typically larger input image resolution and the additional layers needed for semantic segmentation (bottleneck, ASPP block and decoder layers). For the DeepLab-based CNN architecture, we use ResNet18 as the backbone network and the last two residual blocks use dilation rate of 2. The Atrous Spatial Pyramid Pooling (ASPP) block contains dilation rates of $\{1, 8, 12, 18\}$. Our approach produces 0.7 pp better mean average over union (mIOU) and 15% lower BOPs with similar training cost as shown in Tab. 4.12.



((a)) uniform 2-bit `conv2_9` **((b))** uniform 4-bit `conv2_9`

Figure 4.13: Distribution of quantized weights W_q for uniform PACT [7]



((a)) 8bit→3-bit `conv2_7` **((b))** 8bit→4-bit `conv2_9`

Figure 4.14: Distribution of quantized weights W_q for our proposed approach

Model/ Dataset	Quant.	BOPs (G)	mIOU (%)	Training Cost (GPU hours)**
DeepLab CityScapes	8bit [7]	5365	66.6	26hrs 1min
	4bit [7]	1469	65.4	
	Ours	**1254**	**66.1**	28hrs 23min

** Training cost is measured on a NVIDIA V100 GPU

Table 4.12: Comparison of our in-train quantization approach with state-of-the-art methods.

Comparison with State of the Art: In Tab. 4.13, we compare our approach with state-of-the-art uniform quantization approaches, such as PACT [7] and ABC-Net [8]. We also compare with works which produce variable bit-widths for weights and activations such as HAQ [13], DNAS [140], and LBS [141]. HAQ [13] determines layer-wise bit-widths using RL. Such methods have a high computational search cost as the bit-width policy must be learned, involving iterative fine-tuning/evaluation for different bit-width combinations at each episode. DNAS [140] and LBS [141] determine the quantization strategy using gradient based optimization. DNAS constructs a super net consisting of several parallel edges representing search convolution operations with different quantization levels. LBS reduces the search complexity compared to the multi-path DNAS and also exploits filter pruning to further extract compression benefits.

Model/ Dataset	Method	Mixed Precision Weight	Mixed Precision Activation	BOPs (M)	Top-1 (%)
ResNet20 CIFAR-100	PACT-8 [7]	fixed	fixed	2592	68.3
	PACT-4 [7]	fixed	fixed	666	67.0
	PACT-2 [7]	fixed	fixed	189	61.6
	ABCNet-3x3 [8]	fixed	fixed	390	61.0
	HAQ (RL)* [13]	learned	learned	653	67.7
	DNAS* [140]	learned	learned	660	67.8
	LBS* [141]	learned	learned	630	68.1
	Ours	learned	learned	646	**68.3**
ResNet56 CIFAR-100	PACT-8 [7]	fixed	fixed	8025	71.1
	PACT-4 [7]	fixed	fixed	2029	70.4
	PACT-2 [7]	fixed	fixed	528	67.8
	ABCNet-3x3 [8]	fixed	fixed	1153	68.4
	HAQ (RL)* [13]	learned	learned	2015	71.2
	DNAS* [140]	learned	learned	2035	71.2
	LBS* [141]	learned	learned	1918	71.6
	Ours	learned	learned	**1739**	71.3

Table 4.13: Comparison of our in-train quantization approach with state-of-the-art methods. $*$ indicates that the accuracy and BOPs measurements are reported from [141].

However, all the three approaches retrain the sampled quantized strategies obtained from the search phase indicating higher GPU hours compared to our approach, which requires only the regular training time of the CNN as shown in Tab. 4.11. Our quantization scheme produces dominating solutions in the number of BOPs and prediction accuracy compared to HAQ and DNAS. Compared to LBS, our ResNet20 model has slightly higher accuracy ($0.2\,\mathrm{pp}$), with a slight increase in BOPs.

In-train HW-aware Quantization We investigate the effectiveness of the proposed in-train optimization scheme in Tab. 4.14, based on pseudo-HW-aware constraints (BOPs), as well as real HW constraints, i.e. inference latency. Using the GP regressor introduced in Sec. 4.5.3, our approach produces bit-widths based on the target metric and target HW. We observe that constraining the number of BOPs does not necessarily produce optimal latency benefits in all the three networks, making it a pseudo-HW-aware metric. Our approach directly reduces the latency by $1.3\times$ with respect to all 4-bit CNNs, with negligible degradation in prediction accuracy ($<1\,\mathrm{pp}$). In case of ResNet56 based latency constrained model, we obtain lower BOPs and prediction accuracy than BOPs based optimization. This can be attributed to the strict latency constraint imposed by the HW model demanding high compression ratios across several layers.

In Fig. 4.15, we compare the various quantization strategies optimized for BOPs (top) and latency (bottom). We observe lower bit-widths for the latency optimized mixed-precision strategy in the shallower layers. In the deeper layers, we observe higher assignment of bit-width for the activations. This is due to deeper layers being less computationally intensive, allowing more bit-widths to be allocated to maintain prediction accuracy, without adding too much latency. Particularly, activation bit-widths are increased, which have smaller volumes in the deeper layers of the CNN.

Model/ Dataset	Constraint	BOPs (M)	Mem Access (MB)	Latency (KCycles)	Top-1 (%)
ResNet20 CIFAR-10	PACT-4 [7]	666	6.6	1135	**92.2**
	PACT-2 [7]	189	4.1	769	89.5
	Ours (BOPs)	**448**	5.1	951	91.3
	Ours (Latency)	530	4.8	**875**	91.2
ResNet56 CIFAR-100	PACT-4 [7]	2029	18.7	3134	70.4
	PACT-2 [7]	528	10.8	2025	67.8
	Ours (BOPs)	1739	15.6	2753	**71.3**
	Ours (Latency)	**1498**	13.3	**2374**	70.7
ResNet18 ImageNet	PACT-4 [7]	34714	216	39637	**65.4**
	PACT-2 [7]	14984	132	28939	60.4
	Ours (BOPs)	**27424**	192	36930	64.6
	Ours (Latency)	35356	151	**31112**	64.5

Table 4.14: Pseudo-HW-aware constraints and real HW constraints for various CNN models on CIFAR-10, CIFAR-100, and ImageNet datasets.

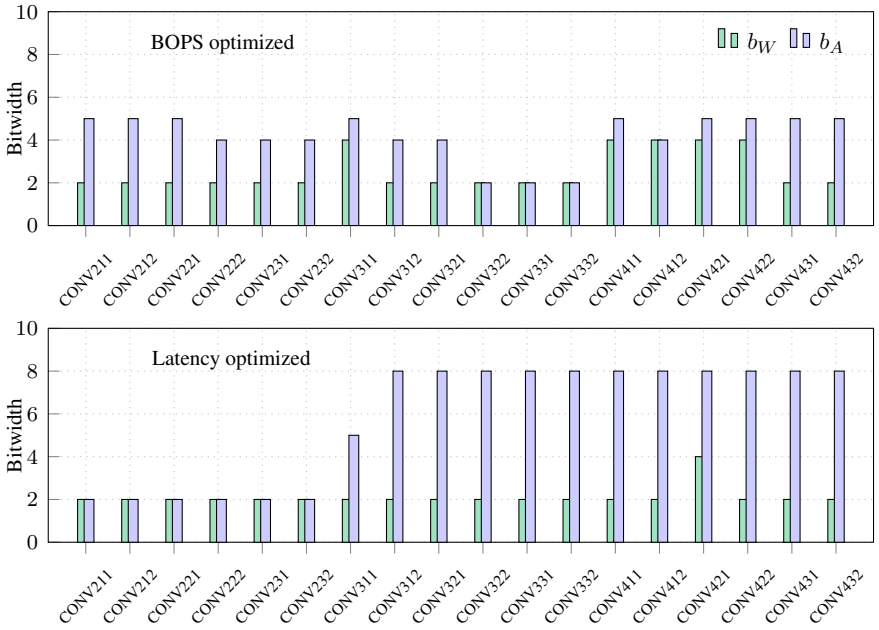
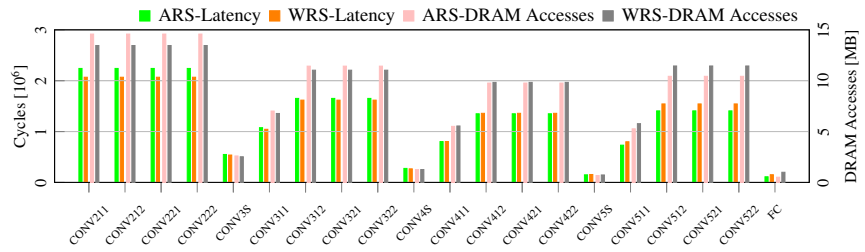


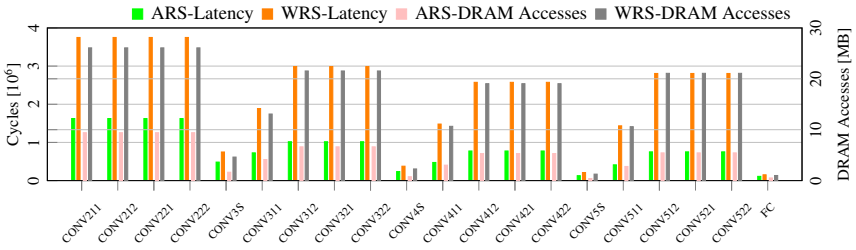
Figure 4.15: Comparsion of layerwise bit-width strategy for BOPs (top) and latency (bottom) optimized in-train quantization.

To demonstrate the sensitivity of the determined quantization strategy with respect to the target HW, we can formulate GP regressors which capture different scheduling schemes on the inference hardware. For the target BISMO bit-serial accelerator, convolutional and dense workloads are transformed into GEMM workloads. To verify the HW-awareness of our method,

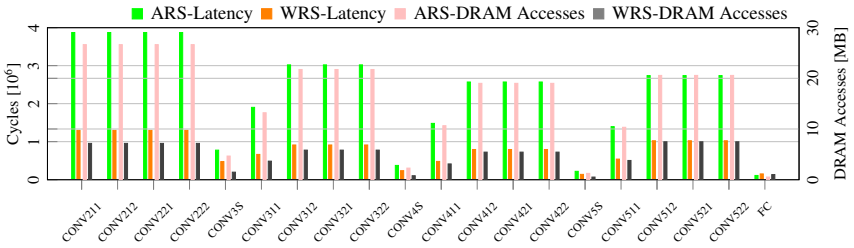
we consider the contrasting ARS and WRS schemes, which favor the reuse of either activations or weights, respectively. We construct GP regressors which can capture the differences between these schedules and analyse the effect of these subtle HW-specific details on our in-train quantization method. With this, we verify the degree of HW-awareness possible through our differentiable GP regressors and in-train quantization technique.



(a) ResNet18-ImageNet $(b_W, b_A) = (4, 4)$



(b) ResNet18-ImageNet $(b_W, b_A) = (2, 8)$



(c) ResNet18-ImageNet $(b_W, b_A) = (8, 2)$

Figure 4.16: ARS and WRS execution of ResNet-18 for ImageNet with uniform quantization. The results motivate the in-train quantization method to make schedule-aware decisions on layer-wise, datatype bit-widths,

In Fig. 4.16-a, we execute ResNet18 for ImageNet on an 6×6×256 BISMO array [131]. The weight and activation bits are set to 4-bits for all layers (uniform). We see that reusing the smaller volume of weights at the start of the CNN leads to WRS performing better, while ARS improves the execution for the latter half of the network, where reusing the activations is more beneficial. This is due to the fact that the total redundant reads are reduced when reusing the smaller volume

datatype, as a larger portion of it can be stored on-chip and used exhaustively, leading to fewer total processing passes.

In Fig. 4.16-b and -c, we execute ResNet18-ImageNet again, once with (b_W, b_A) = (2, 8) and then with (b_W, b_A) = (8, 2), for all layers. It can clearly be observed that with the expensive cost for reading the 8-bit wide datatype, the corresponding schedule which reuses that datatype has a significantly improved execution compared to the other. For example, ARS performs better than WRS for all layers, when the activations are 8-bit wide (Fig. 4.16-b). Conversely, WRS beats ARS when b_W = 8 (Fig. 4.16-c). Therefore, reusing the more *costly* datatype brings an advantage to the execution schedule. It is important to note, that all runs presented in Fig. 4.16 are on the exact same HW, but with a different schedule. This indicates that simply knowing the theoretical peak operations per second (OPS) of a hardware accelerator is not sufficient to have *real* HW-awareness. Subtleties, such as the schedule and datatype reuse highly influence the execution.

In Tab. 4.15, we observe that our quantization approach assigns higher bit-widths to the favorable datatype being reused by the HW, thus *learning* to exploit the inherent efficiency of the chosen schedule, without knowing its details. The average bit-width of weights in the WRS-based mixed-precision strategy remains at the highest value (8-bits), while the activations are quantized more aggressively, as they are costly and not reused by the WRS schedule. Conversely, we observe higher average bit-width for activations in ARS-based mixed-precision strategy. We also observe $1.33\times$ and $1.01\times$ reduction in DRAM accesses for ARS and WRS-based quantization strategies in ResNet56, with $0.4\,\mathrm{pp}$ better accuracy.

Model/ Dataset	Training Scheme	Avg. bit-width Weight	Activation	ARS Mem (MB)	WRS Mem (MB)	DRS Mem (MB)	Top-1 (%)
ResNet20 CIFAR-10	PACT-4 [7]	4	4	6.6	5.4	5.4	**92.2**
	PACT-2 [7]	2	2	4.1	3.3	3.3	89.7
	Ours (ARS-Opt)	2.3	5.3	**5.0**	6.3	4.6	91.8
	Ours (WRS-Opt)	8.0	3.3	10.3	**4.7**	4.7	91.6
ResNet56 CIFAR-100	PACT-4 [7]	4	4	18.7	15.3	15.3	70.4
	PACT-2 [7]	2	2	10.8	8.9	8.9	67.8
	Ours (ARS-Opt)	2.1	4.2	**14.0**	19.2	13.0	**70.8**
	Ours (WRS-Opt)	8.0	3.7	30.3	**15.1**	15.1	**70.8**

Table 4.15: Influence of quantization strategies based on the ARS and WRS compiler schedules.

4.5.5 Discussion

In this section, we propose an *in-train* quantization technique, which eliminates the need for computationally expensive model exploration time, typically required in post-training compression methods. We specifically eliminate the iterative fine-tuning and retraining phases used in RL or evolutionary search based optimization pipelines. In-order to learn the quantization strategy directly during the training, we formulate a QAT scheme using learnable bit-widths. We modify the parameterization of the fixed point quantizer used in uniform QAT schemes such as PACT [7] to determine efficient bit-width assignments. We avoid gradient explosion by determining set

of unique values required to represent weights and activations for all the layers. We also formulate HW-loss using fractional bit-widths. Furthermore, we determine HW-aware bitwidths by formulating a differentiable loss object using GP regression. The GP regression automatically supports HW-estimates for fractional bits. Compared to the uniform 4-bit quantization, our approach reduces the number of BOPs by $1.5\times$ for ResNet56 with minimal accuracy degradation on CIFAR-100 dataset. Compared to state-of-the-art mixed-precision approaches, we reduce the number of BOPs while improving the prediction accuracy, producing dominating solutions. With no degradation in task accuracy, our approach reduces the inference latency by $1.3\times$, compared to a uniformly quantized ResNet56.

This approach is similar to the in-train pruning procedure specified in Sec. 4.4. We leverage the quantization as a compression opportunity instead of determining redundant elements in the weight matrix. We additionally formulate differentiable HW-estimator to easily attain HW-awareness during the training process. We also determine compiler-aware/schedule-aware quantization strategy similar to Sec. 3.4. We observe higher bit-width assignments to the Ifmaps for the ARS compiler based quantization strategy. Similarly, we observe higher bit-width assignments to the weight matrix for the WRS compiler based quantization strategy. These experiments assure the HW-awarenes due to the GP-regressor for determining an efficient quantization strategy during the training process. We also realize defensive mixed precision strategies by integrating QAT with state of the art adversarial training scheme in Sec. 5.4.

4.6 Conclusion

In this chapter, we investigate efficient CNN optimization methods which produce HW-aware compression configurations with lower GPU hours. In Sec. 4.3.1, we presented pruning agent that learns to prune filters of CNN in iterative procedure. The agent requires no human-in-loop, does not use any handcrafted heuristics for filter selection criteria. Filter position and importance is leaned by the agent to derive the most efficient model in term of compression ratio with minimum accuracy loss. The proposed agent learns two kind of actions: Firstly, action to prune filters, Secondly, number of retraining epochs to be conducted between pruning steps. With epoch learning, we achieve reduction in optimization time by $2\times$ compared to pruning without epoch learning. We also achieve $3.8\times$ compression ratio with only 1% accuracy loss.

In Sec. 4.4, we incorporate the pruning process in the underlying optimization function of the training, an approach which we term as *in-train pruning*. We thereby break through the barrier between training and pruning, and save the computational effort for additional post-train pruning. We also eliminate the need for a pre-trained model. We demonstrate that our method achieves 80% reduction of multiply and accumulate (MAC) operations in ResNet models. Our method is found to be most effective for structured pruning regularities. As structured pruning is more advantageous on general-purpose accelerators such as GPUs, this strengthens the motivation for the proposed in-train pruning scheme. Also, since the robust pruned model is achieved during training itself, this method saves the GPU hours for model exploration and final fine-tuning. This method also yields promising results for CenterNet pruning for object detection on the KITTI dataset.

In Sec. 4.5, we propose an *in-train* quantization technique, which eliminates the need for computationally expensive model exploration time, typically required in post-training compression methods. We directly optimize our quantization strategy by formulating a HW-aware differentiable loss objective using Gaussian process regression. We show that our approach can determine efficient quantization strategies based on the underlying schedules of the HW-accelerator. Compared to state-of-the-art mixed-precision approaches, we reduce the number of BOPs while improving the prediction accuracy, producing pareto dominating solutions.

5 Adversarial Robust Compression

Neural network compression is an extensively studied topic for reducing the computational complexity [55, 8, 7], the memory demand [6, 5] and/or the energy consumption [100] of CNNs deployed on embedded systems. These aspects widen the potential for CNN applications in real-world scenarios. For e.g. an autonomous car must not get tricked to take wrong (potentially dangerous) decisions by slightest variations of the image-based input that are not perceptible by humans. One particular example of these variations (or perturbations) is the group of adversarial attacks against which networks should be robust. Attacking neural networks can be done by injecting small perturbations to the inputs, using white-box and black-box methods as discussed in Sec. 2.5. Understanding these threats help to develop pro-active [151] and re-active [152] methods to defend against adversarial examples and thereby improve CNN robustness. We also propose compression methods which jointly optimize the CNN's compute complexity and its vulnerability to adversarial attacks. We specifically realize adversarially robust pruned and quantized CNNs.

In this chapter, Sec. 5.1 elaborates the need for adversarial training methods to defend against perturbations. Sec. 5.2 discusses related works on analyzing adversarial robustness of compressed Neural Networks. We further discuss robust compression approaches in literature. Sec. 5.3 illustrates detailed experiments, which investigates the adversarial robustness of different compressed variants. Sec. 5.4 describes defense methods to achieve robust compressed models through in-train pruning and quantization. Sec. 5.5 concludes the chapter by providing important take-aways. The chapter is based on the publications of Vemparala et al. [30], Vemparala et al. [28] and Vemparala et al. [29].

5.1 Robustness of Compressed Networks

Most CNN compression methods in the literature are evaluated only in terms of prediction accuracy on the test dataset. Despite their satisfactory performance, these methods *do not* compare the change in adversarial robustness with respect to the baseline model. In this chapter, we would like to first determine CNN compressed variants, which are inherently robust against adversarial attacks. Furthermore, we would like propose a unified constrained optimization method to compress large-scale CNNs into both compact and adversarially robust models. In Sec. 5.1.1, we formulate the three objectives for realizing robust compressed models. In Sec. 5.1.2, we discuss a defensive training method from literature, which achieves robust CNNs with lower GPU hours.

5.1.1 Joint Objective formulation

Achieving a balanced trade-off between the optimization objectives *natural accuracy*, *compression rate* and *adversarial robustness* of CNNs, portrays an extensive optimization space.

Natural Accuracy: One of the three optimization aspects considers the *natural accuracy* of underlying CNNs used to solve image classification tasks. Given a CNN $\mathcal{F}$ with model parameters θ that classifies original images X_{org} with its corresponding labels Y for a given image classification task results in an natural accuracy Acc_{nat}. In this chapter, we focus on preserving the baseline natural accuracy Acc_{nat} using compression techniques like pruning and quantization. The compressed model $\mathcal{F}_c$ aims to maintain the natural accuracy Acc_{nat} similar to to its full precision CNNs $\mathcal{F}$ with model parameters θ_c as shown in Eq. 5.1.

$$\mathcal{F}_c(\theta_c, X_{org}, Y) \longmapsto Acc_{nat} \tag{5.1}$$

Compression Rate: The second optimization objective represents the *compression rate* of underlying CNNs. A compressed CNN $\mathcal{F}_c$ with model parameters θ_c classifying input images X results in a certain compute complexity on the target hardware after deployment. We measure the compute complexity of pruned CNNs using FLOPs. To represent the processing advantages of quantized CNNs, we report the number of BOPs. We aim to reduce the compute complexity of pruned and quantized CNNs with respect to a baseline power hungry full precision model.

Adversarial Robustness: Lastly, exposing the given compressed CNN $\mathcal{F}_c$ to adversarial images X_{adv} results in the adversarial accuracy Acc_{adv}, representing the *adversarial robustness* of the underlying model against a specified threat model τ, see Eq. 5.2.

$$\mathcal{F}_c(\theta_c, X_{adv}, Y) \longmapsto \text{Acc}_{adv} \;\; s.t. \;\; \tau \longmapsto X_{adv}, \mathcal{F}_c \tag{5.2}$$

5.1.2 Fast Adversarial Training

A number of defense mechanisms have been proposed, that provide robustness to CNNs against the adversarial attacks. Adversarial training refers to incorporating attacked images, generated by a threat model, in order to ensure that the model loss on all attacked images of the same class is low. Based on the attack mechanism used for generating image perturbations during training, there are different defense procedures, some of which are more effective than the others. Adversarial training proposed by Madry et al. [18] provides security guarantees against all attacks of the *first-order*.

Wong et al. [17] introduced Fast Adversarial Training (FastAT) to train models adversarially, a technique that was previously deemed expensive. One of the earliest adversarial training techniques, proposed by Goodfellow et al. [70] used FGSM attack for generating adversarial examples during training. Though this method was fast, it was sub-optimal. Wong et al. [17] found that FGSM attack with random initialization is as effective for defence as PGD-based

training. PGD based adversarial training is costly due to the necessity of constructing adversarial examples using iterative attack during training. With FastAT, the cost of training with one iteration of FGSM is significantly lower than the other variants. They also introduced several techniques like cyclic learning rate and mixed-precision arithmetic to accelerate the convergence of the training process. Using these techniques they brought down the training time for adversarial training to the same level as vanilla training. In this work, FastAT has been used to achieve robust optimization.

5.2 Related Work

Robust compressed models have been achieved through compression techniques by pruning and quantization. Galloway et al. [153] evaluated and interpreted the adversarial robustness of BNNs. They highlight the most commonly mentioned benefits of BNNs, i.e. the reduced memory consumption and the faster inference. Their work complements the two benefits with a third aspect, namely the robustness of BNNs. They show an improved or on par robustness of BNNs against adversarial attacks compared to full-precision models. The inherently discontinuous and approximated gradients of BNNs gives them an advantage over full-precision networks for adversarial attacks. In Sec. 5.3, we conduct extensive experiments on different compression variants such as pruning, quantization and knowledge distillation on various white box and black box attacks.

Inspired by the work of Zhang et al. [134], Ye et al. [154] incorporated adversarial robustness into the ADMM objective to produce *RobustADMM*. One of the main findings of the work for improving adversarial robustness of a compact model is to concurrently prune and adversarially train an over-parameterized network from scratch. They infer that adversarial robustness requires a significantly large capacity of the network. Directly training a sparse model from scratch cannot reach the same level of natural accuracy and adversarial robustness, that can be achieved by training an over-parameterized model (4 or 8 times its size) and then pruning it.

Contrary to the hypothesis of Ye et al., Kundu et al. [155] proposed *Dynamic Network Rewiring (DNR)* following the SNFS [137] based sparse training approach to realize a robust network from scratch. It achieves ultra high levels of compression and can learn the layerwise sparsities of the network on its own through momentum redistribution. To combat accuracy degradation, it uses a dynamic L_2 regularizer, inspired by a relaxed version of the ADMM regularization loss. Unlike SNFS, DNR framework also supports channel pruning regularities. DNR combines the advantages of both sparse learning as well as ADMM. To the best of our knowledge, it is the only sparse learning approach that has been extended for robust compression. However, the robustness of these models is not guaranteed as they evaluate their attacks against weak adversaries (PGD attack for 7 iterations with step size $\alpha = 0.01$).

Gui et al. [156] proposed *Adversarially trained Model Compression (ATMC)* framework, a unified constrained model compression formulation where existing compression techniques like pruning, quantization and factorization are incorporated into the constraints. However, a pretrained model with adversarial robustness is required before the compression.

Sehwag et al. [149] proposed a post-train robust pruning approach entitled *Hydra*, which achieves extremely high levels of compression for weight pruning. It learns the pruning strategy

based on robustness aware optimization objective. The authors show that it performs better than the lowest magnitude-based weight pruning [6] heuristic. After pretraining an over-parameterized model using normal vanilla training, the pruning is based on the importance score computed from the adversarial loss. Fine-tuning is performed on the pruned subgraph. They succeed in obtaining robust subnetworks within the non-robust pretrained network. In Sec. 5.4, we produce robust compressed models, which does not require a pre trained model and aims at obtaining a pruning/quantization strategy without relying on handcrafted heuristics.

5.3 Robustness of Quantized and Pruned Networks

In order to get a deeper understanding of the effectiveness of adversarial attacks (Sec. 2.5), applied to binary and efficient CNNs, we perform an extensive set of robustness evaluation experiments. In this section, we expose vanilla full-precision, distilled, pruned and binary CNNs to a variety of adversarial attacks.

5.3.1 Robustness Evaluation Setup

Although a successful attack could easily be carried out by adding large perturbations, the requirement of finding the minimum necessary perturbation in each case is typically desirable to perform the attack in an inconspicuous manner. This justifies CNNs to being particularly robust against adversarial attacks that are relevant or expected in practice. However, despite the requirement to keep the perturbation as small as possible, the target for training against an attack structure can be to maximize a corresponding loss function. A prior analysis on the robustness of real world compressed CNNs provides insights which facilitate the realization of strong adversarial defense methods. We perform white box and black box attacks on compressed variants as shown in Fig. 5.1. Compressed CNNs aim to mitigate the challenges of their deployment on edge devices. Compression techniques such as knowledge distillation, pruning, and binarization can potentially make CNNs more efficient in embedded settings.

Knowledge Distillation: Knowledge Distillation (KD) is the transfer of knowledge from a teacher to a student network [157, 9]. The student can be a smaller CNN, which is trained on the soft labels of the larger teacher network, achieving an improvement in accuracy-efficiency trade-off. The student represents a compressed version of the teacher, condensing its knowledge. We focus on KD training, using Kullback Leibler (KL) divergence between the teacher and the student output distribution formulated as the loss function in Eq. 5.3. Here, $\sigma(f_t(I))$ and $\sigma(f_s(I))$ represent the softmax output logits of the teacher and student network respectively, computed for a sample image I in a mini batch of N samples.

$$\mathcal{L}_{\text{KD}}^{KL}(f_t, f_s, T) = \sum_{n=1}^{N} \sigma(f_t(I_n)/T) \log \left(\frac{\sigma(f_t(I_n)/T)}{\sigma(f_s(I_n)/T)} \right) \tag{5.3}$$

Pruned Neural Networks: He et al. [5] proposed a learning-based compression method. The authors leverage a RL agent, which learns the possible sparsities in each layer and prunes them based on an ℓ_2-norm heuristic. We adapt the RL-agent of AMC-AutoML [5] to support different pruning regularities such as filter-wise (F. Prune), channel-wise (Ch. Prune), kernel-wise (K.

Prune) and weight-wise (W. Prune) pruning (shown in Fig. 5.1). Pruning input channels from a layer also discards corresponding output filters from previous layers. Thus, Ch./F. pruning result in a similar compression ratio and CNN structure. The pruning regularity has a direct impact on the hardware implementation complexity and throughput benefits. In this paper, the pruning rate is set at a constant value of 50% over all experiments and pruning regularities.

Binary Neural Networks: Binarization represents the most aggressive form of quantization, where the network weights W and activations are constrained to ± 1 values. This greatly reduces the memory requirements of CNNs. Binarizing a floating-point CNN reduces its memory footprint by $\times 32$. Rastegari et al. [55] introduced XNOR-Net, where the convolution of an input feature map A^{l-1} and weight tensor W^l is approximated by a combination of XNOR operations and *popcounts* $\oplus$, followed by a multiplication with a scaling factor α, such that $\texttt{Conv}(A^{l-1}, W) \approx (\texttt{sign}(A^{l-1}) \oplus B) \cdot \alpha$ (shown in Fig. 5.1). BNNs typically suffer from accuracy degradation. To mitigate this problem, Lin et al. [8] proposed a scheme for ABC-Net. The authors approximated the convolution by using a linear combination of multiple binary bases for weights and activations, shrinking the accuracy gap to full-precision counterparts. In this chapter, we implement ABC-Net and XNOR-Net binarization techniques, to evaluate the effect of adversarial attacks on accurate BNNs.

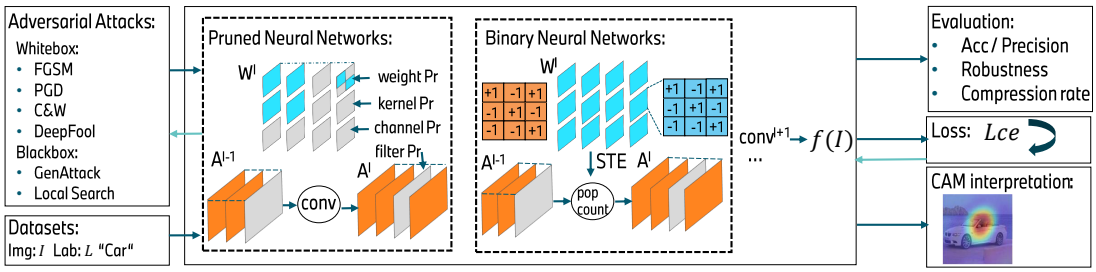


Figure 5.1: Experimental setup for breaking binary (C) and efficient (A) and (B) CNNs attacked with white-box (FGSM, PGD and C&W) and black-box (LocalSearch and GenAttack) adversarial attacks. Evaluated by using loss/accuracy levels, stress-strain graphs, box-plots and class activation mapping (CAM).

We evaluate robustness of CNNs which are trained and evaluated on CIFAR-10 [42] or ImageNet [1] datasets. The 50K train and 10K test images (32×32 pixels) of CIFAR-10 are used to train and evaluate compressed variants of ResNet20/56 [39, 9, 5, 55, 8] respectively. The ImageNet dataset consists of $\sim$1.28M train and 50K validation images (256×256 pixels). Compressed variants of ResNet18/50 are trained and evaluated for ImageNet experiments. If not otherwise mentioned, all hyper-parameters specifying the training and the attacks were adopted from the reference implementation. The robustness evaluation covers various white-box (FGSM, PGD, C&W, DeepFool) and black-box (LocalSearch, GenAttack) attacks on the CIFAR-10-trained ResNet20/56 compressed variants, as well as ImageNet-trained CNNs.

Tab. 5.1 summarizes the compressed CNNs and their full-precision counterparts analyzed in this section. It shows that the different compressed variants differ drastically in their memory demand and their compute complexity. Deep learning inference accelerators such as the NVIDIA-

T4 GPU [158] or Xilinx FPGAs with DSP48 blocks support SIMD-based bit-wise operations [81]. In particular, a single DSP48 block can perform two 16-bit fixed-point multiplications or 48 XNOR operations at once. The normalized compute complexity (NCC) is defined as the optimal utilization of MAC and XNOR operations in one compute unit. The DSP48 block serves as a reference implementation to compute NCC in Tab. 5.1.

Dataset	Model	Acc. [%]	Memory demand [MB]	NCC [10^6]
CIFAR-10	**ResNet20** [39]	92.46 %	1.07	41
	KD-KL [157]	93.25 %	1.07	41
	Ch.Prune [5]	89.76 %	0.70	19
	K.Prune [5]	90.73 %	0.61	20
	W.Prune [5]	91.98 %	0.59	20
	XNOR [55]	82.71 %	0.04	1.3
	ABC(1×1) [8]	83.42 %	0.04	1.3
	ABC(3×3) [8]	88.94 %	0.12	8.0
	ABC(5×5) [8]	90.64 %	0.20	21.3
	ResNet56 [39]	93.88 %	3.40	125
	KD-KL [157]	94.24 %	3.40	125
	Ch.Prune [5]	92.86 %	2.50	62
	K.Prune [5]	93.04 %	2.19	63
	W.Prune [5]	93.54 %	2.02	62
	XNOR [55]	83.24 %	0.11	3.0
	ABC(1×1) [8]	86.29 %	0.11	3.0
	ABC(3×3) [8]	92.48 %	0.33	24
	ABC(5×5) [8]	92.82 %	0.55	66
ImageNet	**ResNet50** [39]	75.43 %	102.01	10216
	ResNet18 [39]	69.00 %	46.72	1814
	ResNet18-Ch.Prune [5]	67.62 %	34.52	884
	ResNet18-XNOR [55]	49.10 %	4.14	173
	ResNet18-ABC(1×1) [8]	51.07 %	3.48	153
	ResNet18-ABC(3×3) [8]	59.83 %	6.28	417

Table 5.1: Accuracy Top1 [%], Memory demand [MB] and the NCC of compressed CNNs and their full-precision counterparts.

5.3.2 Robustness Evaluation on CIFAR-10 dataset

PGD-Evaluation: Considering PGD attack as the *ultimate* first-order attack (Sec. 2.5), this section experimentally explores the structure of the loss surfaces and the corresponding accuracy deterioration of the proposed efficient CNNs, while exposing the models to PGD adversaries, similar to those proposed by Madry et al. [18]. Investigating the resulting structural behavior, especially the loss level to which the PGD attack is converging to and the speed of deterioration of accuracy, helps in understanding the adversarial robustness of the underlying models with respect to a defined PGD threat model τ_{PGD}=$\{\ \epsilon, \alpha, i\ \}$.

All models are pretrained on CIFAR-10 dataset without any adversarial examples, to distinguish the influence of varying compression techniques on adversarial robustness. Subsequently, each model is exposed to PGD attacks from τ_{PGD}=$\{\epsilon$=2, α=0.5, i=1000$\}$. Following the method of Carlini et al. [69], i was increased to verify convergence, ensuring local-maxima, representing potentially worst-case adversarial examples for the underlying model with respect to the applied threat model τ_{PGD}. However, results are only shown up to i=100, since τ_{PGD} showed

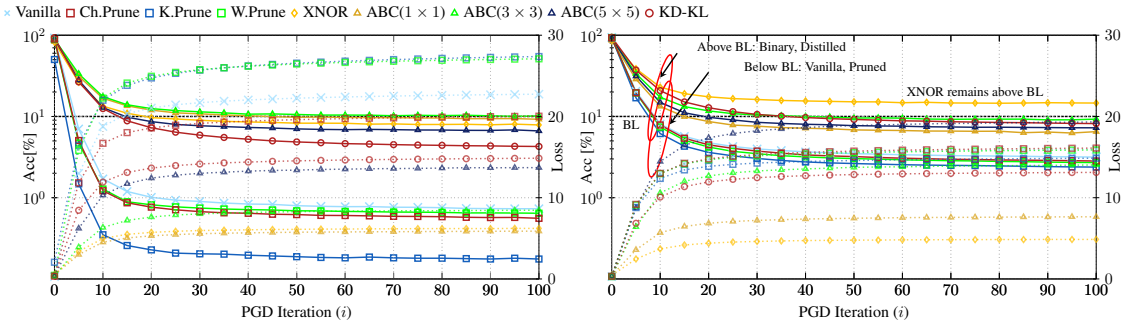


Figure 5.2: PGD attack accuracy (solid) and loss (dashed) over several iterations for compressed variants of ResNet20 (left) and ResNet56 (right) averaged over five reruns of PGD attack. Additionally, the horizontal breaking line (BL - dashed black) visualizes the deterioration of model accuracy below random guessing ($\leq 10\%$) for CIFAR-10. Visual markings are added to categorize models above and below the BL at i=10.

convergence for all investigated models in this range. The loss value and the corresponding accuracy of the models to the adversary were tracked every 5^{th}-iteration. In the following, the adversarial robustness of a model against PGD attacks is evaluated by (1) the overall loss level the PGD attack is converging to and as a consequence the resulting accuracy (2) the number of iterations a model can sustain until it breaks. We can consider a CNN model broken, if its accuracy indicates that the classification is random (10% for CIFAR-10 dataset), represented by model accuracy graphs dropping below the breaking line. Fig. 5.2 shows the mean over five reruns of PGD attack for all models to exploit random initialization, which ensures random exploration of the underlying non-concave maximization problem as described in Sec. 2.5.

Consistently, all investigated pruning techniques harm adversarial robustness against PGD attack with respect to its vanilla versions of ResNet20/56, when considering (1) the loss and accuracy after a converged attack and (2) the speed of breaking. Vanilla and pruned versions of ResNet20 break within five iterations, whereas the respective ResNet56 versions break within ten iterations. KD shows greater resilience to the PGD attack since (1) its accuracy after the converged attack is higher compared to both the ResNet20/56 vanilla variants and (2) breaking at a higher number of iterations. KD-KL breaks at i=15 for its ResNet20 variant and at i=40 at its ResNet56 variant. Binarization can improve the robustness against the defined PGD attack, materializing in (1) the higher loss and accuracy after a converged attack and (2) the greater resilience for a longer period of PGD iterations. XNOR-Net and ABC(5×5) break at i=20, while ABC(3×3) and ABC(1×1) break at around i=60 for their ResNet20 variants. For the ResNet56 variants, ABC(1×1) and ABC(5×5) break at i=20, whereas ABC(3×3) sustains up to i=40. The ResNet56 variant of XNOR-Net outperforms all other models in (1) accuracy after converged attack (∼14%) and (2) being the only model that never breaks throuhgout this experiment (see Fig. 5.2 right).

Investigations on different threat models: In Fig. 5.3, we present box-plots from data collected over a range of experiments. For each attack, we sweep over the respective strength and iterations mentioned in Tab. 5.2. The exact definition of strength and iteration for each attack can be recalled from Sec. 2.5. The data includes both models, ResNet20 and ResNet56.

Attack	Strength ϵ	Iterations i
FGSM	2, 4, 8, 16	N/A
PGD	0.1, 0.5, 1.0, 2.0	2, 3, 4, 5
CW	0.01, 0.1, 1.0, 5.0, 10.0	1,10, 20, 50
DeepFool	N/A	1, 5, 10, 20
Local Search	8, 16, 32	50, 100, 150, 200
GenAttack	8, 12	50,100, 150, 200 $popsize$=6, 16

Table 5.2: Various strength and iteration combinations tested for ResNet20 and ResNet56 variants (vanilla, pruned, binary, and distilled). Strength and iteration definitions for each attack are explained in Sec. 2.5

Fast Gradient Sign Method: For FGSM attacks, the results show that the KD-KL variant is more resilient compared to other compression techniques, as its mean attacked accuracy is higher compared to other variants. During the training, the distillation is performed using higher temperature (T = 30). The attack perturbations are generated using cross-entropy loss with T = 1, resulting in saturated gradients and therefore weakening the attack. We also observe a boost in robustness, when the baseline CNN is the larger ResNet56 model. Interestingly, the binarized ABC models also exhibit similar robustness for both ResNet20 and ResNet56 variants (see third quartile FGSM attacked accuracy of ABC-Nets in Fig. 5.3).

Projected Gradient Descent: For PGD, we raise the attack intensity by increasing the strength ϵ or attack iterations i. We observe that the mean accuracy after attack is higher for XNOR and distilled CNN variants compared to vanilla and pruned CNN models.

Carlini & Wagner: For the C&W method, we observe that the attack impacts all the compressed variants. We use the target class as deer for CIFAR-10 dataset to fool the CNN model. We also observe that the mean accuracy after attack is lower than other attacks making it the strongest threat model.

DeepFool: Similar to the C&W attack, DeepFool renders most of the CNN compressed variants. One exception is the XNOR-Net [55], which can sustain accuracy after attack with high amount of attack intensity. It is worth noting that the other binary CNNs like ABC-Net do not perform as well as XNOR with this threat model.

LocalSearch: We observe the mean accuracy after attack for all the variants of BNNs is higher for local search based black box threat model. The target class we consider for the attack is deer in CIFAR-10 dataset. The data points are based on different strengths ϵ = { 8, 16, 32 } and iterations i = { 50, 100, 150, 200 }.

GenAttack: For GenAttack, we consider the number of generations i as { 50, 100, 150, 200 }, different attack amplitudes ϵ as $\{8, 12\}$ and various population sizes N as $\{6, 16\}$. In Fig. 5.3, a clear difference between the robustness of BNNs and other variants is observed. We can classify BNNs as strong against blackbox based GenAttack.

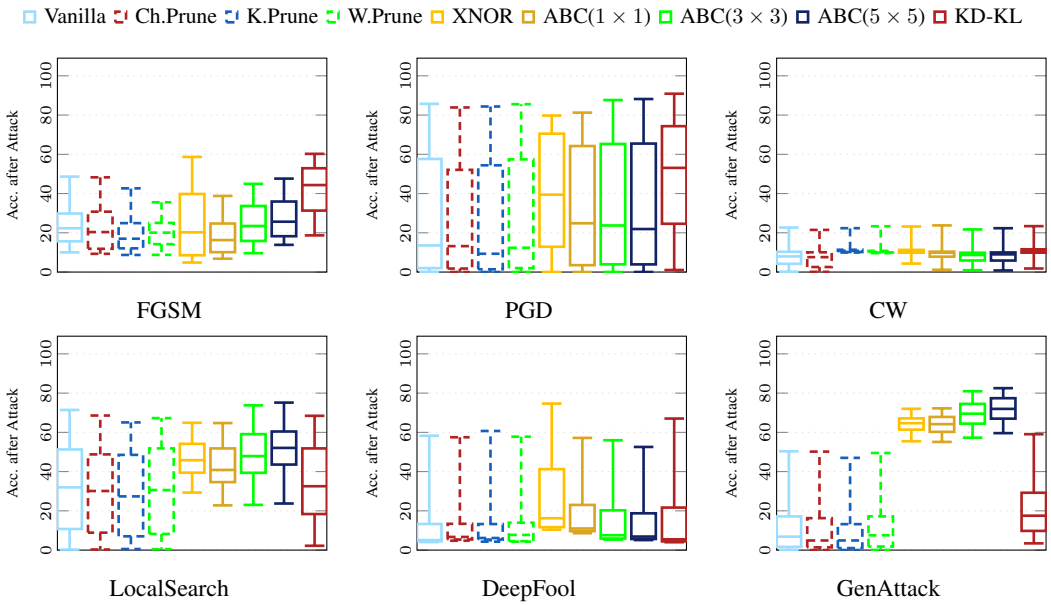


Figure 5.3: Box-plots for attacks on compressed variants of ResNet20 and ResNet56.

5.3.3 Robustness Evaluation on ImageNet dataset

For the robustness evaluation on the ImageNet dataset [1], we use pre-trained ResNet50 and ResNet18 models, and compressed variants of ResNet18. We observe a higher attack search time for ImageNet compared to the CIFAR-10 dataset due to the larger image sizes and model complexity. Therefore, we limit our analysis to two white-box attacks (FGSM and PGD), and one black-box attack (GenAttack). We consider compressed variants such as Ch-Prune, XNOR, ABC(1×1) and ABC(3×3) specified in Tab. 5.3- 5.5.

Fast Gradient Sign Method: In Tab. 5.3, we report the natural accuracy and attacked accuracy for different strengths ($\epsilon = \{2, 4, 8, 16\}$). ResNet50 achieves the highest natural accuracy and attacked accuracy for different strengths compared to other models. Among the compressed variants the channel pruned and ABC(3x3) models portray slightly higher robustness at different strengths.

FGSM	**Nat.Acc**	$\epsilon = 2$	$\epsilon = 4$	$\epsilon = 8$	$\epsilon = 16$
ResNet50 [39]	75.43 %	22.18	16.24	12.08	7.46
ResNet18 [39]	69.00 %	12.82	8.16	5.19	2.95
ResNet18-Ch.Prune [5]	67.62 %	11.18	6.64	3.99	2.34
ResNet18-XNOR [55]	49.10 %	7.57	4.54	2.19	0.93
ResNet18-ABC(1×1) [8]	51.07 %	9.11	4.65	2.30	1.13
ResNet18-ABC(3×3) [8]	59.83 %	11.33	5.73	2.65	1.43

Table 5.3: Accuracy (Top1) [%] of CNNs after FGSM adversarial attacks for ImageNet.

Projected Gradient Decent: In Tab. 5.4, we report the attacked accuracy for two strengths ($\epsilon = 0.1$, $\epsilon = 0.5$). The attacked accuracy decreases for all the models as we increase the number of iterations i. We observe 9.16% higher attacked accuracy for binarized ResNet18 using ABC(3×3) compared to the ResNet50 model at $i = 5$ and $\epsilon = 0.1$. Robustness at higher attack strength $\epsilon = 0.5$ degrades the prediction accuracy for all the compressed variants.

PGD	ϵ	$i = 2$	$i = 3$	$i = 4$	$i = 5$
ResNet50 [39] (75.43 %)	0.1	25.77	16.07	9.83	5.91
	0.5	3.35	0.94	0.43	0.27
ResNet18 [39] (69.00 %)	0.1	17.86	10.32	5.58	3.11
	0.5	1.33	0.17	0.04	0.01
ResNet18-Ch.Prune [5] (67.62 %)	0.1	17.02	10.23	5.92	3.50
	0.5	1.40	0.27	0.06	0.02
ResNet18-XNOR [55] (49.10 %)	0.1	13.16	11.46	10.06	8.84
	0.5	5.67	3.07	1.57	0.78
ResNet18-ABC(1×1) [8] (51.91)	0.1	18.35	16.22	14.20	12.37
	0.5	7.60	3.64	1.75	0.82
ResNet18-ABC(3×3) [8] (59.83)	0.1	23.90	20.81	17.80	15.07
	0.5	8.31	3.70	1.59	0.66

Table 5.4: Accuracy [%] of CNNs after PGD adversarial attacks for ImageNet.

GenAttack: We set an adaptive mutation rate ρ and mutation range α for GenAttack based on the dataset configuration and set the population size to 6, as in [78]. In Tab. 5.5, we report overall attacked accuracy and accuracy w.r.t. the fooled target class at several iterations during the attack search ($i = \{200, 400, 600, 800, 1000\}$). We also analyze the robustness for two attack strengths ($\epsilon = 8, 12$). Similar to previous observations, ABC models portray higher robustness with respect to their unattacked accuracy, when compared to other compressed variants and the vanilla ResNet50 and ResNet18 models.

GenAttack	ϵ	$i = 200$ **OA/TA**	$i = 400$ **OA/TA**	$i = 600$ **OA/TA**	$i = 800$ **OA/TA**	$i = 1000$ **OA/TA**
ResNet50[39] (75.43 %)	8.0	21.29/12.80	11.64/34.46	6.87/51.94	4.67/64.08	3.06/72.82
	12.0	13.16/17.45	5.67/41.19	3.55/56.65	2.40/67.29	1.60/74.58
ResNet18[39] (69.00 %)	8.0	16.41/14.52	8.11/41.83	4.35/62.58	2.36/75.62	1.34/83.29
	12.0	10.24/22.44	5.13/50.74	2.70/68.85	1.58/80.21	1.04/86.62
ResNet18-Ch.Prune[5] (67.62 %)	8.0	12.34/12.82	6.05/39.02	3.17/60.46	2.00/74.46	1.22/82.79
	12.0	7.33/20.25	3.29/49.44	1.84/68.97	1.08/80.11	0.88/86.80
ResNet18-XNOR[55] (49.10 %)	8.0	13.06/0.64	12.86/0.72	12.64/0.84	12.68/0.86	12.68/0.94
	12.0	11.56/0.78	11.14/0.92	11.14/1.04	11.04/1.16	10.82/1.22
ResNet18-ABC(1×1)[8] (51.07 %)	8.0	17.59/1.48	17.67/1.62	17.37/1.76	17.23/1.88	16.89/1.98
	12.0	15.83/1.90	15.40/2.08	15.20/2.26	15.02/2.34	14.86/2.52
ResNet18-ABC(3×3)[8] (59.83 %)	8.0	26.00/0.68	25.02/0.82	25.26/0.92	25.46/0.98	25.58/0.96
	12.0	22.50/0.74	22.04/0.94	22.36/1.02	21.75/1.08	21.90/1.14

OA/TA = Accuracy to original label / Accuracy to target label.

Table 5.5: Accuracy (Top1) [%] of CNNs after GenAttack adversarial attacks for ImageNet. The Population size for the experiments = 6

5.3.4 Class Activation Maps

We use CAM [147] to determine the region of interest (RoI) for the prediction class using clean and attacked images. The output feature maps of the last convolutional layer and the weight tensor of the fully-connected layer is considered as the input to the CAM. The CAM highlights regions of the image that influence the CNN's prediction to a specific class. Similar to heat-maps, red regions indicate those with the highest contribution, while blue indicates the ones with the least. We applied CAM on various compressed variants of ResNet20 and ResNet56, trained on CIFAR-10, which are attacked by DeepFool (Tab. 5.6). As mentioned in Sec. 2.5, DeepFool attempts to find the adversarial perturbation which leads the CNN to the closest decision boundary. Once a perturbation is found, it is reinforced to push the prediction beyond that boundary. Through the CAM visualizations in this section, we attempt to capture this behaviour over the attack iterations.

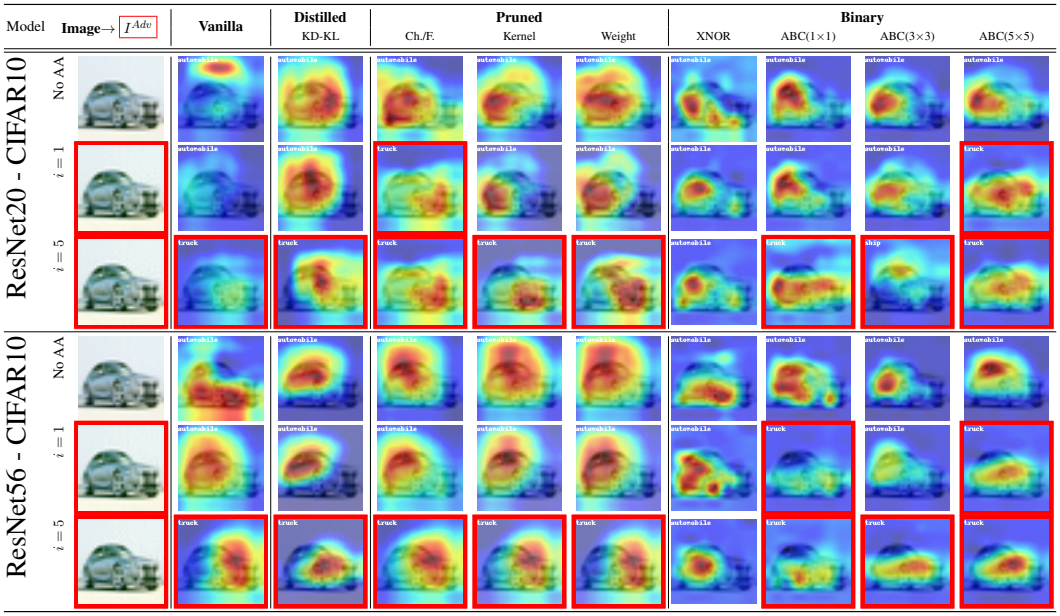


Table 5.6: CAM for ResNet20/56 and its compressed variants performed on non-attacked and DeepFool attacked images on the automobile image from CIFAR-10 dataset.

All the compression techniques produce no mis-classification in the automobile example using the unattacked raw image in Tab. 5.6. Three interpretations can be made from the heat maps. We support our interpretation with quantitative analysis by measuring the third quartile value of the heat map intensity across all the pixels. Observing the CAM output of ResNet56's vanilla and channel-pruned variants for the unattacked input image, the RoI has large focused interest regions. For an intensity range of (0,255) blue→red, the third quartile value of the heat map intensity across all pixels is 184 and 162 for vanilla and channel-pruned respectively, indicating a large RoI. Second, the intensity of the interest regions decreases, after the attack is applied for one iteration. The third quartile value decreases (171, 152) indicating the lower interest regions. Third, after the attack is applied for five iterations, the focus on the attacked region (bonet) is reinforced to fool towards the nearest class ($truck$). The third quartile value further decreases (135, 121). Under

DeepFool attacks, ResNet56 is more robust compared to ResNet20 which can be illustrated by the more distinct RoIs in the heat maps. The BNN variants have a small RoI compared to their vanilla model for unattacked images. The third quartile value for ResNet56-XNOR is 98 indicating this aspect. As the inherent RoI for BNNs are small and concentrated, it could reduce the chances of finding and perturbing the smaller set of critical pixels by the attack model.

5.3.5 Discussion

The robustness of distilled models can be attributed to their soft label training, which can be more informative than sheer, hard labels. The student is ideally able to learn both the correct classification *and* the distribution of closeness among other classes. Furthermore, the student is distilled using a high temperature factor T, causing the magnitude of the predicted class to be T times more confident than when trained on hard labels [75]. Thus, white box attacks like FGSM, PGD and DeepFool would require strong adversarial perturbation for fooling the final prediction to its nearest class. However, the C&W attack is able to fool the distilled model, even at higher temperatures as the attack is not focused on the cross-entropy loss directly.

The training scheme for BNNs is not as simple as vanilla or pruned models. It requires a straight-through-estimator, making the white-box attacks challenging compared to other variants. Introducing multiple scaling factors in case of ABC-Net eases the approximation to its full-precision model. Thus, XNOR-Nets appear to be more resilient against white-box attacks (Fig. 5.3). Moreover, the PGD loss levels in Fig. 5.2 demonstrate the robustness of XNOR-Net through lower loss convergence values and breaking speed. The discretization of weights and activations also makes BNNs stronger against black-box attacks. The CAM results support the robustness for BNNs as they inherently possess smaller and concentrated RoI, reducing the chances of finding and perturbing the critical set of pixels. The BNN robustness is also observed for the ImageNet dataset when attacked with PGD and GenAttack (Tab. 5.4, Tab. 5.5). In Sec. 5.4, we also analyze robustness of uniform, mixed precision CNNs and further realize defensive in-train quantization method.

Pruning is the process of eliminating unused and/or redundant parameters. Here, balancing the compression rate and the accuracy is a key factor. Due to the reduced learning ability, pruned models are not automatically more robust than their full-precision counterpart. This would call for an extra objective function for improving the robustness. Existing works have shown that it is possible to remove more model parameters when pruning is applied in an unstructured manner [6]. A similar behavior can be expected if the robustness is included in the pruning and fine-tuning process. In Sec. 5.4, we realize defensive post-train, in-train pruning methods and highlight its robustness against white box attacks such as PGD and C&W.

5.4 Defensive Pruning and Quantization

In Sec. 5.3, we investigate the adversarial robustness of various compressed CNN variants. We observe that the pruned models are vulnerable to the attacks among all of them. Furthermore, we had also investigated the robustness of multi-bit CNNs and observed superiority in most of

the white box and black box attacks. In this section, we aim to achieve *adversarially robust HW-friendly CNNs* by formulating a defensive compression scheme.

5.4.1 Defensive Compression

In this section, we target two objectives: 1) Compressing a model to reduce the computational effort of a neural network, and 2) increasing the robustness against an adversary manipulating input data. Both can be effectively achieved by formulating a joint optimization problem as shown in Fig 5.4.

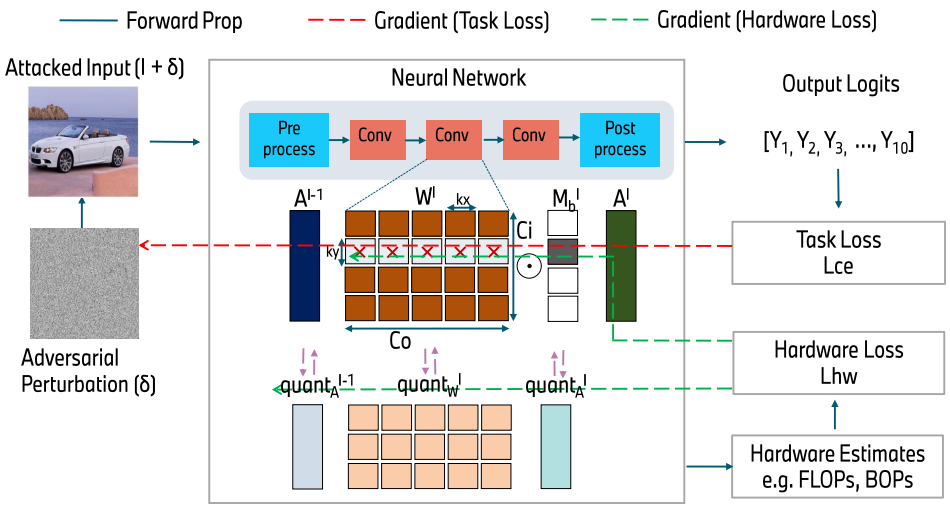


Figure 5.4: Depiction of defensive compression method using in-train pruning and quantization

We modify the in-train pruning approach specified in Sec. 4.4 and in-train quantization approach in Sec. 4.5 to achieve a balanced trade-off between natural accuracy Acc_{nat} (calculated from the ground-truth labels Y for the corresponding images I), adversarial robustness Acc_{adv}, and model complexity $\text{sum}(M_b)$ during the training process. This is achieved without introducing separate (post-training) phases for pruning and fine-tuning.

To obtain robust pruned CNNs, we adopt the concept of adversarial training [18] but incorporate pruning edges in the network using a binary mask $M_b \in \{0, 1\}$. These masks are derived from a trainable continuous mask M, that is, weights W are canceled out if the corresponding dimension of the mask is 0 and left unchanged if it is set to 1. Attacks against a neural networks are described as finding a minimal perturbation δ of an image I (forming the the adversarial example $I_{\text{adv}} = I + \delta$) that changes the outcome of a given model represented by the prediction function $f(\cdot)$ [70]. For adversarial training, we make use of this generation principle, while maximizing the loss $\mathcal{L}$ for a given perturbation budget ϵ in Eq. 5.4.

$$\min_{W,M} \mathop{\mathbb{E}}_{(I,Y)\sim\mathcal{D}} \left[\max_{|\delta|\leq\epsilon} \mathcal{L}\left(f(I+\delta, W \odot M_b), Y\right) \right]. \tag{5.4}$$

The outer minimization problem in Eq.5.4 involves a set of randomly sampled images from dataset $\mathcal{D}$, where the expected loss $\mathbb{E}$ on the random samples is minimized through an adversarial training scheme, such as FastAT [17]. Exposing a model to adversarial images I_{adv} results in the adversarial accuracy Acc_{adv}, representing the measure of adversarial robustness of the underlying model. In principle, one may use different methods for generating adversarial examples for training, such as FGSM [70], PGD [18] and CW [75]. Wong et al. [17], however, show that using FGSM in combination with random initialization is particularly effective. With this, the cost of training, measured in GPU hours, with one iteration of FGSM is significantly lower than with other variants like PGD-based adversarial training [18]. We integrate the in-train update operations of the pruning mask M_b in the FastAT procedure as shown in Alg.1.

Algorithm 1: Joint selection of pruning masks and adversarial training.

Require : Training samples $\mathcal{D}$, perturbation strength ϵ, step size α

1 Initialize θ, $M_b \leftarrow 1$
2 **for** $Epoch = 1, \dots N_{epochs}$ **do**
3 **for** *Batch* $B \subset \mathcal{D}$ **do**
4 Initialize perturbation $\delta \leftarrow random\_uniform(-\epsilon, +\epsilon)$
5 Sample a batch of K examples $\{(I^{(1)}, Y^{(1)}), \cdots, (I^{(K)}, Y^{(K)})\}$ from data distribution.
6 Use FGSM attack to generate perturbations on batch K to update δ
7 $\delta \leftarrow \delta + \alpha \cdot sign(\nabla_\delta \mathcal{L}(f(I + \delta, W \odot M_b), Y)$
8 $\delta \leftarrow \max(\min(\delta, \epsilon), -\epsilon)$
9 $I_{\text{adv}} \leftarrow I + \delta$
10 Update weights W and pruning masks M using SGD for adversarial images:
11 $W \leftarrow W - \eta \cdot \nabla_W \mathcal{L}(f(I_{\text{adv}}, W \odot M_b), Y)$
12 $M \leftarrow M - \eta \cdot \nabla_M \mathcal{L}(f(I_{\text{adv}}, W \odot M_b), Y)$
13 **end**
14 **if** $E_{\textit{Prune, Start}} \leq Epoch \leq E_{\textit{Prune, End}}$ **then**
15 **if** $Epoch \ mod \ Prune\_step = 0$ **then**
16 $M \leftarrow M - \eta \cdot \nabla_M \mathcal{L}_{\text{Prune}}(M, \psi_{\text{base}})$
17 **end**
18 $M_b \leftarrow \text{round}(0.5 \cdot \tanh(\text{M}) + 0.5)$
19 **end**
20 **end**

During each training step, we generate a uniform random initialization for the adversarial perturbation as shown in line 4, followed by performing a step into the ascent gradient direction (line 7) is scaled by the step size α. We update the weights and pruning masks of the neural network jointly in lines 11 and 12 for clean and adversarial images with learning rate η. During these update steps the importance scores for masks M_b get accumulated. Lines 15 and 16 zero out prune masks based on a hardware loss $\mathcal{L}_{\text{Prune}}$ (sec:loss). As shown in line 14, we start and freeze the optimization of prune masks at the epoch corresponding to $E_{\text{Prune, Start}}$ and $E_{\text{Prune, End}}$ respectively.

We also integrate the QAT such as PACT [7] and in-train mixed precision scheme with FastAT [17] based defense method. To obtain mixed precision CNNs, we learn the number of unique values $|U| \in \{1, 256\}$ required to represent weights and activations for each layer. We detail the quantization for weights and activations as W_q and A_q in Eq. 5.5 and Eq. 5.6, respectively. We clip the activations A between the range $[0, +c]$ due to the non-linear activation function (ReLU) and approximate them as $A_q \in \{0,\ 1,\ 2, ...,\ (2^b - 1)\}$, similar to [7].

$$\mathtt{A_q} = Round(\mathtt{Clip}(A, 0, c) \cdot \frac{(2^b - 1)}{c}) \cdot \frac{c}{(2^b - 1)} \tag{5.5}$$

We clip the weight values using a $tanh()$ function and limit the range between [-1, 1], similar to the work in DoReFa-Net [50] as shown in Eq. 5.7. We approximate the continuous domain of weights W into the discrete values $W_q \in \{\ -(2^{b-1}-1), ..., -2,\ -1, 0,\ +1,\ +2, ...,\ (2^{b-1}-1)\}$.

$$W_q = 2\big(Round(\mathtt{Clip_{tanh}}(W) \cdot (2^{b-1} - 1)) \cdot \frac{1}{(2^{b-1} - 1)}\big) - 1 \tag{5.6}$$

$$\mathtt{Clip_{tanh}(x)} = \frac{tanh(x)}{2max(tanh(x))} + \frac{1}{2} \tag{5.7}$$

Our in-train quantization approach aims to achieve a balanced trade-off between natural accuracy Acc_{nat} (calculated from the ground-truth labels Y for the corresponding images I), adversarial robustness Acc_{adv}, and model complexity φ during the training process, rather than introducing separate post-training phases for finding the quantization strategy. Attacks against a neural network are described as finding a minimal perturbation δ of an image I (forming the the adversarial example $I_{\text{adv}} = I + \delta$) with label Y that changes the outcome of a given model represented by the prediction function $f(\cdot)$ [70]. For adversarial training, we make use of this generation principle, while maximizing the loss $\mathcal{L}$ for a given perturbation budget ϵ as shown in Eq. 5.8. We integrate the in-train update operations of the unique values U for quantized weights W_q and activations A_q in the FastAT procedure as shown in Alg.2.

$$\min_{W} \mathop{\mathbb{E}}_{(I,Y)\sim\mathcal{D}} \left[\max_{|\delta|\leq\epsilon} \mathcal{L}\left(f(I + \delta, W_q), Y\right) \right]. \tag{5.8}$$

Different from Alg. 1, we formulate a defensive quantization method in Alg. 2. We update the weights and unique values $|U|$ of the neural network jointly in line 11 for clean and adversarial images, with learning rate η. During these update steps, the importance scores for trainable unique values for each layer $|U|$ get accumulated. Line 13 reduces the number of unique values based on a hardware loss $\mathcal{L}_{\text{HW}}$ and cross entropy loss $\mathcal{L}_{\text{ce}}$. As shown in line 12, we start and freeze the quantization strategy at the epoch corresponding to $E_{\text{Quant, Start}}$ and $E_{\text{Quant, End}}$ respectively.

5.4.2 Experiments - Robust Pruning

In this section, we demonstrate our proposed in-train pruning and quantization's ability to achieve compressed models, balancing the trade-off between natural accuracy and adversarial robustness.

Algorithm 2: Joint learning of quantization strategy and adversarial training.

Require : Training samples $\mathcal{D}$, perturbation strength ϵ, step size α

Initialize θ, $|U| \leftarrow 256$
for $Epoch = 1, \dots N_{epochs}$ **do**
  for *Batch* $B \subset \mathcal{D}$ **do**
    Initialize perturbation $\delta \leftarrow random\_uniform(-\epsilon, +\epsilon)$
    Sample a batch of K examples $\{(I^{(1)}, Y^{(1)}), \cdots, (I^{(K)}, Y^{(K)})\}$ from data distribution.
    Use FGSM attack to generate perturbations on batch K to update δ
    $\delta \leftarrow \delta + \alpha \cdot sign(\nabla_\delta \mathcal{L}(f(I + \delta, Qunat(W)), Y)$
    $\delta \leftarrow \max(\min(\delta, \epsilon), -\epsilon)$
    $I_{\text{adv}} \leftarrow I + \delta$
    Update weights W and unique values $|U|$ using SGD for adversarial images:
    $W \leftarrow W - \eta \cdot \nabla_W \mathcal{L}(f(I_{\text{adv}}, Qunat(W)), Y)$
    if $E_{\textit{Prune, Start}} \leq Epoch \leq E_{\textit{Prune, End}}$ **then**
      $|U| \leftarrow |U| - \eta_{|U|} \cdot \nabla_{|U|} \mathcal{L}(f(I_{\text{adv}}, Qunat(W)), Y)$
    end
  end
end

Baseline Training As a baseline for adversarial training, we implement FastAT [17] (see Tab. 5.7). For FastAT on the CIFAR-10 dataset, we use random FGSM with strength $\epsilon = 8/255$, step size $\alpha = 10/255$ to generate adversarial perturbations during the training process. We train for 300 epochs and set the initial learning rate to 0.1 and scale it down by a factor of 10 every 80 epochs. For evaluating robustness of the pruned models, the PGD attack is performed with $\epsilon = 8/255$ and $\alpha = 2/255$ for 20 iterations.

AMC-based Robust Pruning For the purpose of comparison with post-train pruning approach, we implement the state-of-the-art RL-based pruning scheme AMC [5]. We find pruning configurations tgenerating a trade-off between natural accuracy and adversarial accuracy. We constrain the number of operations to the target specified in Tab. 5.7 and Tab. 5.8 and allow the RL-agent to search for 500 episodes to obtain the pruning strategy. We adversarially fine-tune the resulting network with a cyclic learning rate of 0.1 for 30 epochs.

Improved Robustness with In-Train Pruning We augment our pruning approach with FastAT [17]-based adversarial training and start zeroing the prune masks at $E_{\text{Prune, Start}}$ = 20 and freeze the masks at the $E_{\text{Prune, End}}$ = 240. We use an initial learning rate of 0.1 and decrease it by a factor of 10 at the 80^{th} and 160^{th} epoch. We use the same attack strength as baseline training.

In Tab. 5.7, we make a comparison between the RL-based post-train pruning approach and the proposed in-train pruning method. Across all experiments, we observe an improvement in natural accuracy, while maintaining similar adversarial robustness. For a target reduction ψ^*=0.3 on

ResNet20, we obtain an improvement of 5.91 pp in natural accuracy. For ResNet56 and ψ^*=0.3, we obtain an improvement of 8.65 pp in natural accuracy and with similar adversarial robustness.

Model	Operations Reduction	FastAT + RL Prune		FastAT + In-train Prune	
		Acc [%]	PGD-Acc [%]	Acc [%]	PGD-Acc [%]
ResNet20	1.0	81.52	40.65	81.52	40.65
	0.70	78.89	40.39	80.63	39.27
	0.50	77.11	39.65	80.32	40.14
	0.30	66.97	33.89	72.88	34.33
ResNet56	1.0	84.03	38.45	84.03	38.45
	0.70	82.78	42.47	84.52	36.91
	0.50	81.88	41.78	84.56	36.78
	0.30	74.75	36.95	83.40	36.89

Table 5.7: Comparison between post-train RL-based robust pruning and the proposed in-train robust pruning for various operation constraints.

In Fig. 5.5, we plot the training behaviour to compare the in-train pruning approach with (blue) and without (gray) adversarial robustness, for $\psi^* = 0.3$. We observe noisy improvement in natural accuracy behaviour for the in-train robust pruning (blue). The sudden fluctuation in accuracy at 15K and 30K iterations indicates the change in training behaviour due to the step learning rate policy. During these iterations, we observe large changes in the HW loss, indicating a phase of exploration in the binary prune masks M_b ($0 \leftrightarrow 1$). The changes in the pruning masks result in noisy accuracy improvement but eventually stabilize within 5K training iterations. We freeze the changes in pruning masks at the 45K iteration as the pruning constraint ψ^* is satisfied ($\mathcal{L}_{HW} = 0$).

We also verify the robustness of our in-train pruning scheme with stronger adversarial attacks such as Carlini-Wagner (C&W) [75] as shown in Fig. 5.6. C&W is an iterative attack guided by an optimizer such as Adam, which produces strong adversarial examples by simultaneously minimizing perturbation distance and manipulating the predictions based on a target class. Different loss functions can be applied in C&W attacks. In our experiments, the most efficient l_2-bounded loss is used for the evaluation. We run the attack for 100 iterations and set the C&W constant c=100, which is responsible for controlling the trade-off between the attacked image similarity and the success rate of the target class. We do not perform a binary search on the c value as suggested in the paper, since our focus is not on minimizing adversarial distance. Instead, we use a high value of c to ensure that the models are subjected to the strongest attack for reasonable image perturbations. In Fig. 5.6, we observe that the vanilla model trained without adversarial perturbations breaks very early at the 10^{th} iteration. However, robust models withstand the attack for more iterations (≥ 20) with adversarial accuracy at least greater than 20%. Our pruned models obtain even higher adversarial accuracy than the unpruned RobustAT network after 30 iterations. We also observe higher adversarial accuracy starting from the 20^{th} iteration for our

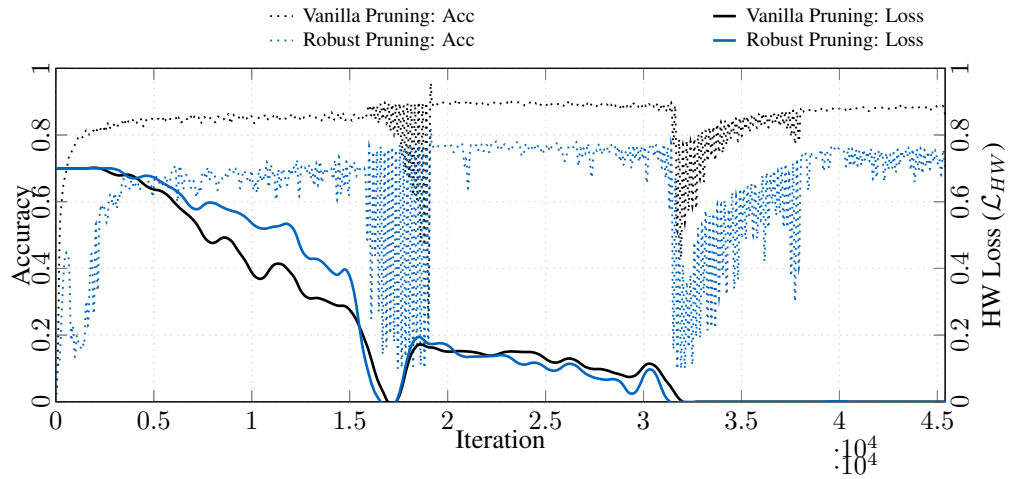


Figure 5.5: Comparison of the proposed in-train pruning scheme for operation constraint $\psi^* = 0.3$ with (blue) and without (gray) the consideration of adversarial robustness.

in-train pruned model with target constraint $\psi^* = 0.3$. This indicates the generalization capability of the in-train pruning approach as the compression rate increases.

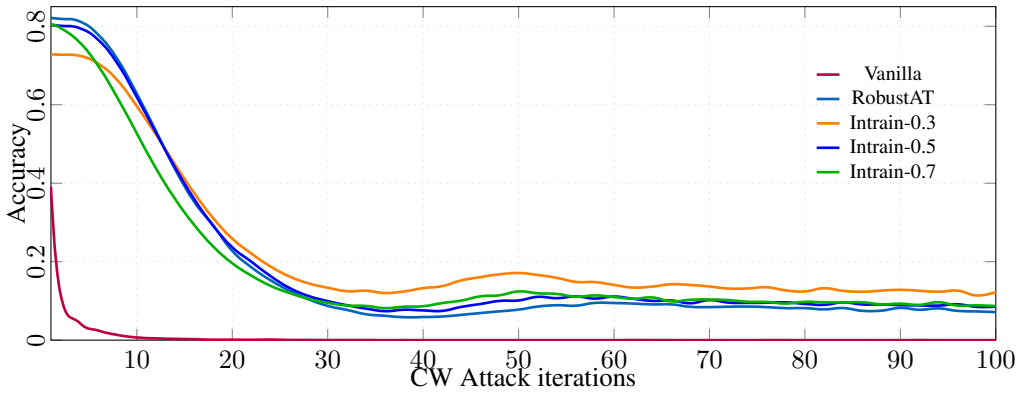


Figure 5.6: Comparison of adversarial robustness for different CNN models using C&W attack for ResNet20.

In Tab. 5.8, we analyze the proposed in-train pruning approach with different pruning regularities in the context of adversarial robustness. Additionally, we compare our results with the post-train RL-based pruning scheme. The RL-agent proposed in the original work of AMC [5] is only suited for channel-wise pruning. We adapted the RL-agent to also perform pruning for different regularities. We observe that irregular weight pruning gives the best trade-off between natural and adversarial accuracy. These observations also align with the robust pruning works in literature [149, 154]. The effectiveness of the in-train pruning scheme compared to the RL-based pruning scheme becomes more evident as the pruning is structured (weight-wise $\rightarrow$ channel-wise). Compared to RL-based weight pruning, we observe a slight degradation in natural accuracy (-0.65

pp) for the in-train pruning scheme on ResNet20. However, the proposed method produces 5.91 pp, 0.44 pp better natural and adversarial accuracy respectively for channel pruning. The same trend also applies to ResNet56. As channel pruning is more advantageous on general-purpose accelerators such as GPUs, this strengthens the motivation for the proposed in-train pruning scheme.

Model	**Pruning Regularity**	**FastAT + RL Prune**		**FastAT + In-train Prune**	
		Acc [%]	**PGD-Acc [%]**	**Acc [%]**	**PGD-Acc [%]**
ResNet20	1.0	81.52	40.65	81.52	40.65
	weight	79.08	40.35	78.43	38.59
	kernel	75.79	38.63	77.92	38.64
	channel	66.97	33.89	72.88	34.33
ResNet56	1.0	84.03	38.45	84.03	38.45
	weight	83.94	41.04	83.21	38.64
	kernel	81.68	40.82	83.31	38.68
	channel	74.75	36.95	83.40	36.89

Table 5.8: Robust pruning for various pruning regularities with target operation constraint ψ^* = 0.3.

In Tab. 5.9, we analyze the scalability of the robust in-train pruning approach for larger and more complex datasets such as ImageNet. We train ResNet18 and ResNet50 models using FastAT [17] to obtain robust baseline models. We consider random FGSM with strength ϵ = 2/255, step size α = 2.5/255 for generating adversarial perturbations during training. The models are trained for 100 epochs, the learning rate is initially 0.1 and scaled down by 10 every 30 epochs. We report adversarial accuracy using the PGD attack with ϵ = 2/255 and α = 0.5/255 for 20 iterations. Tab. 5.9 presents the natural and adversarial accuracies for operations constraint of 30%, 50% and 70% on ResNet18 models. For 30% operations reduction, ResNet18 achieves 1.19 pp better adversarial accuracy compared to the baseline with 1.84 pp degradation in natural accuracy, whereas for 50% operations reduction, ResNet18 achieves 1.76 pp better natural accuracy with less than 1% degradation in adversarial accuracy. However, for 70% operations constraints, both ResNet18 and ResNet50 suffer a degradation of (3.27 pp, 5.34 pp) and (9.21 pp, 7.56 pp) in natural and adversarial accuracies respectively.

We compare the proposed in-train pruning approach to the robust pruning works in literature. In Tab. 5.10, we report the results of RobustADMM [154], Hydra [149] and ATMC [156]. RobustADMM, Hydra and ATMC use different baseline models, PGD evaluation parameters and adversarial training schemes. RobustADMM considers an over-parameterized ResNet as a baseline model and prunes it for various parameter constraints. We report their channel pruning results which achieve a model size of 0.04×10^6 (mentioned as $w = 1$ in [154]) and 0.17×10^6 (mentioned as $w = 2$ in [154]). Differently, our approach uses the smaller ResNet20 as a baseline model and achieves 6.21 pp and 6.31 pp better natural accuracy while maintaining similar adversarial robustness for model sizes with 0.04×10^6 and 0.16×10^6, respectively. The results from our approach dominate in terms of natural as well as adversarial accuracy for the same pruning constraints due to dynamic sparsity ratios across layers and heuristic-free pruning.

Model	Acc [%]	Adv. Acc [%]	Ops Reduction Target	Ops Reduction Original	Model Size $\times 10^6$
ResNet18-AT	47.22	28.29	1.0	-	11.68
ResNet50-AT	58.35	37.33	1.0	-	25.60
ResNet18-Prune	45.38	29.48	0.7	0.69	10.60
ResNet18-Prune	48.98	27.34	0.5	0.49	10.02
ResNet18-Prune	43.95	19.08	0.3	0.29	8.43
ResNet50-Prune	53.01	29.77	0.3	0.29	14.66

Table 5.9: In-train pruning for various operation constraints for ImageNet dataset

Compared to the work in Hydra [149], we achieve a significant improvement for channel pruning configurations. Different from RobustADMM, Hydra performs a PGD attack for 50 iterations to measure adversarial robustness. Compared to a 50% constrained channel-pruned VGG-16 model, we achieve 69.08% model reduction and 29.64 pp improvement in natural accuracy, while maintaining a similar level of adversarial robustness. The work in ATMC [156] evaluates robustness of compressed ResNet-34 with the PGD attack for 7 iterations. For the comparison, we use the weight pruned configuration of ATMC-32bit model with same attack hyper-parameters and obtain 6.63%, 14.43% higher robustness for $\epsilon = 4/255, 8/255$ respectively with similar model size. Different from [154, 149, 156], our pruning method does not require a pre-trained model.

5.4.3 Experiments - Robust Quantization

We demonstrate our proposed mixed-precision approach's ability to achieve compressed models with a balanced trade-off between natural accuracy and adversarial robustness. As a baseline for adversarial training, we implement FastAT [17] with uniform quantization. We augment our trainable quantization parameters in the FastAT defense method and report the natural accuracy and PGD robustness [18] for our mixed precision strategies in Tab. 5.11. The PGD attack, referred as the "ultimate" first-order adversary [18], generates perturbations using iterative multi-step optimization method. By considering random uniform initialization, arbitrary starting points on the corresponding loss surface are ensured, thus resulting in worst-case adversaries for the given image with respect to an underlying CNN model. For PGD evaluation, we use a strength of $8/255$, step size $2/255$ for 20 iterations. Existing work shows that low-precision models exhibit higher adversarial robustness due to the discrete nature of the quantization operations [153]. Thus, we observe an increase in adversarial robustness as the bit-width is reduced for the uniform PACT quantization. Our in-train quantization approach improves the trade-off between the three objectives, namely prediction accuracy, adversarial robustness, and BOPs reduction. The achieved robustness is increased by $0.9\,\mathrm{pp}$ and $1.4\,\mathrm{pp}$, while reducing the number of BOPs by $1.6\times$ and $1.9\times$, compared to adversarially trained uniform 4-bit ResNet20 and ResNet56, respectively.

Work	Baseline Model	Pre-trained Model	Pruning Regularity	PGD Attack $\epsilon, \alpha, iter$	Model Size $\times 10^6$	Acc [%]	Adv. Acc [%]
FastAT Wong et al. [17]	ResNet20	✗	no prune	$8, 2, 10$	0.27	82.06	40.97
				$8, 2, 50$	0.27	82.06	40.52
RobustADMM Ye et al. [154]	ResNet18	✓	channel	$8, 2, 10$	0.04	64.52	38.01
			channel	$8, 2, 10$	0.17	73.36	43.17
In-train Prune (Ours)	ResNet20	✗	channel	$8, 2, 10$	0.04	**70.73**	**39.31**
			channel	$8, 2, 10$	0.16	**79.67**	**43.22**
ATMC Gui et al. [156]	ResNet34	✓	weight	$4, 0.7, 7$	0.11	84.00	62.00
			weight	$8, 1.4, 7$	0.11	84.00	39.00
In-train Prune (Ours)	ResNet56	✗	weight	$4, 0.7, 7$	0.13	82.68	**68.63**
			weight	$8, 1.4, 7$	0.13	82.68	**53.43**
Hydra Sehwag et al. [149]	VGG16	✓	weight	$8, 2, 50$	0.76	78.90	48.70
			weight	$8, 2, 50$	0.15	73.20	41.70
			channel	$8, 2, 50$	7.65	52.90	38.00
In-train Prune (Ours)	VGG16	✗	channel	$8, 2, 50$	**5.51**	**82.54**	**38.36**
			channel	$8, 2, 50$	**0.76**	73.40	30.20

Table 5.10: Comparing the in-train pruning scheme with SoTA on CIFAR-10 dataset.

Model/ Dataset	Method	Bitwidth W_{bit}	Bitwidth A_{bit}	BOPs (M)	Top-1 (%)	PGD-20 (%)
ResNet20 CIFAR-10	PACT [7]	4	4	666	81.9 ± 0.04	40.6 ± 0.27
	PACT [7]	2	2	189	76.0 ± 0.06	41.5 ± 0.32
	Ours (high BOPs)	4.6	3.7	691	82.5 ± 0.50	41.3 ± 0.35
	Ours (low BOPs)	3.5	2.9	**427**	**81.7± 0.08**	**41.5± 0.31**
ResNet56 CIFAR-10	PACT [7]	4	4	2029	85.3 ± 0.25	41.5 ± 0.72
	PACT [7]	2	2	529	82.3 ± 0.58	47.3 ± 2.28
	Ours (high BOPs)	4.4	3.6	2045	85.2 ± 0.70	42.5 ± 0.67
	Ours (low BOPs)	2.9	2.7	**1049**	**84.7 ± 0.91**	**42.9± 0.21**

Table 5.11: Adversarial Robustness of uniformly quantized and mixed precision CNNs.

5.4.4 Discussion

In this chapter, we formulate compression schemes to realize pruned and quantized CNN models, which balance the trade-off between *natural accuracy* and *adversarial robustness*. In Sec. 5.3, we have observed that the pruning based compression variants are the most vulnerable against adversarial attacks. Therefore, we formulate a defensive pruning method by introducing learnable prune masks in Fast adversarial training [17]. The proposed in-train robust pruning approach

demands lower number of GPU-hours than post-train methods such as RL. Post-Train pruning approaches demand iterative fine-tuning as well as retraining phases. These phases consume more GPU-hours in the case of adversarial defense methods. We also observe that our method leads to better a trade-off between natural accuracy and robustness (Tab. 5.7) for severe pruning constraints (70% reduction in the number of operations). Furthermore, we also found the in-train method to produce better trade-off than RL-agent for the channel pruning regularity (Tab. 5.8). The channel pruning configurations as discussed in Sec. 2.3 can be easily deploy-able in general purpose HW-platforms. We further analyzed the robustness of pruning configurations against stronger attacks such as C&W in Fig. 5.6.

We further propose a defensive mixed precision approach by integrating in-train quantization discussed in Sec. 4.5 and FastAT [70]. We observe improved robustness as we reduce the bit-width but with an impact in natural accuracy. These observations align with BNN robustness in Sec. 5.3. We further determine efficient bit-width assignments to determine the balance between *natural accuracy*, *bit operations* and *adversarial robustness*.

5.5 Conclusion

In this chapter, we provided a comprehensive analysis on recent white-box and black-box adversarial attacks against state-of-the-art vanilla, distilled, pruned and binary neural networks. We demonstrated that the robustness of CNNs not only depends on the adversarial attack but also on the compression technique at hand. Conclusions were made on robustness by analyzing PGD loss/accuracy levels, box-plots and CNN heat maps with CAM. The following three conclusions can be made from the analysis: First, knowledge distillation, i.e. by minimizing the KL divergence between a teacher and a student, inherently make the model more robust to various adversarial attacks. Second, on the tested black-box attacks, BNNs are more robust compared to other compressed neural networks. Finally, binary and efficient DNNs break differently on various adversarial attacks. From the presented data, we show that knowledge about the expected adversarial attack or the used compression technique can help the designer or the attacker generate more robust applications or stronger attacks respectively.

We further investigate a joint formulation of the learning and compression objectives allowing us to incorporate advantages from adversarial training and increase the robustness of the compressed network. Thus, this method finds a trade-off between task-specific accuracy, adversarial accuracy and compression rate. Compared to state-of the-art robust pruning approaches, our method is found to improve natural accuracy while maintaining same level of adversarial robustness for similar or higher compression rates. We infer that this improvement in performance is due to the ability to learn dynamic sparsity ratios across layers and heuristic-free pruning. We further integrate adversarial training into mixed precision approach to obtain robust quantized models. We, thereby obtain better tradeoff between prediction accuracy, robustness, and the reduction in number of BOPs.

6 Conclusion and Future Work

6.1 Conclusion

The deployment procedure of CNNs on resource constrained HW accelerators is challenging due to limited compute/memory budgets and higher GPU-hours for optimization. Furthermore, CNNs must be robust against slightest variations of input in the form of adversarial attacks to improve safety and security of CV application. In this thesis, we realize HW-aware robust CNNs to improve the trade-off between the three objectives, namely *task-specific accuracy*, *execution metrics* and *adversarial robustness* with *lower GPU-hours*. This would make the inference of various CV applications more accurate, fast and secure.

We improve the *HW-awareness* of compression methods to minimize the execution metrics of CNNs. We specifically provide HW-metrics as a feedback to compression methods/search algorithms to determine efficient pruning and quantization configurations. Using the proposed approach, we model a spatial array based CNN accelerator to obtain HW-estimates and derive benefits for compressed CNNs used for image classification and semantic segmentation. We also derive latency estimates from the GPU based inference and provide it to RL-based pruning scheme to obtain efficient LiDAR-based 3D object detection.

We realize *Fast Compression* methods to reduce the GPU hours required to determine pruning and quantization configurations. We identify that the iterative fine-tuning phase demands high amount of GPU-hours. We formulate a RL-agent based pruning technique which learns the amount of fine-tuning epochs required to judge a pruning configuration through continuous reward formulation. We further reduce the number of GPU hours by incorporating the pruning in the underlying optimization function of the task specific training process. We thereby break through the barrier between training and pruning, and save the computational effort for additional post-train compression. We also realize mixed precision models by introducing learnable bit-widths directly during the training process. We formulate a differentiable HW estimator through Guassian regression to directly determine efficient bit-width assignments with respect to execution metrics such as latency. We determine mixed precision configurations for different variants of bit-serial accelerator.

We analyze the *adversarial robustness* of different compression variants by performing white-box and black-box attacks against state-of-the-art vanilla, distilled, pruned and binary neural networks. We demonstrated that the robustness of CNNs not only depends on the adversarial attack but also on the compression technique at hand. We determine that the complex training schemes for BNNs leads to robustness against several adversarial attacks. We also identify that various pruning configurations are extremely vulnerable against adversarial attacks. Our joint formulation of the learning and pruning objectives allow us to additionally incorporate recent advantages from adversarial training and increase the robustness of the pruned neural networks. Compared to state-of the-art robust pruning approaches, our method is found to

improve natural accuracy while maintaining same level of adversarial robustness for similar or higher compression rates. We infer that this improvement in performance is due to the ability to learn dynamic sparsity ratios across layers and heuristic-free pruning. We similarly realize robust mixed precision CNNs to improve the trade-off between task-specific accuracy, number of BOPs and adversarial robustness.

6.2 Future Work

We identify potential extension to the current work for improving each of the addressed objectives as following:

Enhanced HW awareness: Pruning techniques have different levels of regularities. We can derive higher compression rates using irregular weight pruning compared to channel pruning for maintaining minimal degradation of prediction accuracy. In this thesis, we schedule and map only channel pruning configurations to realize HW-aware pruned CNNs. However, we can also investigate specialized scheduling schemes [63] to efficiently process irregular sparse convolutional workloads to further reduce the latency. The 3D object detection in this thesis is limited to LiDAR-based sensor modality. Recently, there have been developments in CNN architectures [159, 160] which improve the detection quality by extracting features from both camera and LiDAR sensors. Investigating compression opportunities in these networks would further provide us additional pareto dominant solutions.

Flexible Compression: We determine mixed precision CNNs by learning efficient bit-width assignments during the task-specific training process. However, our approach requires retraining when the change in bit-width assignment is desired. Inorder to flexibly switch the bit-width configuration during inference, mixed precision supernets [161] with robust quantizers need to be realized. This flexibility also enables to formulate quantization-aware NAS approaches. Most QAT or post train quantization methods require sensitive or proprietary training datasets to realize low precision CNNs. Formulation of zero-shot quantization methods such as [162] is needed to realize training data free mixed precision CNNs.

Robustness aware NAS: In this thesis, we derive HW-aware robust CNNs through pruning and mixed precision quantization. By formulating NAS approach with adversarial robustness [163] as one of its search objectives, we can further increase the compression rates. Moreover, the generalization of adversarial robustness towards common input corruptions [164, 165] is an interesting research direction to investigate.

Bibliography

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Conference on Computer Vision and Pattern Recognition*, 2009.

[2] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[3] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition*, 2012.

[4] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision*, 2017.

[5] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *European Conference on Computer Vision*, 2018.

[6] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations*, 2016.

[7] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: parameterized clipping activation for quantized neural networks. abs/1805.06085, 2018.

[8] Xiaofan Lin, Cong Zhao, and W. Pan. Towards accurate binary convolutional neural network. In *Conference on Neural Information Processing Systems*, 2017.

[9] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[10] NVIDIA Corporation. Nvidia turing gpu architecture, 2015.

[11] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture*, 2016.

[12] Dong Wang, Ke Xu, and Diankun Jiang. PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In *International Conference on Field Programmable Technology*, 2017.

[13] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[14] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. APQ: Joint Search for Network Architecture, Pruning and Quantization Policy. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[15] Nael Fasfous, Manoj Rohit Vemparala, Alexander Frickenstein, Emanuele Valpreda, Driton Salihu, Nguyen Anh Vu Doan, Christian Unger, Naveen Shankar Nagaraja, Maurizio Martina, and Walter Stechele. HW-FlowQ: A Multi-Abstraction Level HW-CNN Co-Design Quantization Methodology. *ACM Trans. Embed. Comput. Syst.*, 2021.

[16] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, D. Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2014.

[17] Eric Wong, Leslie Rice, and J. Zico Kolter. Fast is better than free: Revisiting adversarial training. In *International Conference on Learning Representations*, 2020.

[18] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations*, 2018.

[19] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, 2015.

[20] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yang Yang. Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[21] Qiangui Huang, Shaohua Kevin Zhou, Suya You, and Ulrich Neumann. Learning to prune filters in convolutional neural networks. *IEEE Winter Conference on Applications of Computer Vision*, 2018.

[22] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matthew Uyttendaele, and Niraj K. Jha. ChamNet: Towards Efficient Network Design Through Platform-Aware Model Adaptation. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[23] Pu Zhao, Wei Niu, Geng Yuan, Yuxuan Cai, Hsin-Hsuan Sung, Wujie Wen, Sijia Liu, Xipeng Shen, Bin Ren, Yanzhi Wang, and Xue Lin. Achieving real-time lidar 3d object detection on a mobile device. *CoRR*, abs/2012.13801, 2020.

[24] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[25] Manoj Rohit Vemparala, Fasfous Nael, Alexander Frickenstein, Emanuele Valpreda, Manfredi Camelleri, Qing Zhao, Christian Unger, Naveen Shankar Nagaraja, Maurizio Martina, and Walter Stechele. HW-Flow: A Multi-Abstraction Level HW-CNN Co-Design Prining Methodology. *In Leibniz Transaction of Embedded Systems.*, 2022.

[26] Manoj Rohit Vemparala, Anmol Singh, Ahmed Mzid, Nael Fasfous, Alexander Frickenstein, Florian Mirus, Hans Joerg Voegel, Naveen Shankar Nagaraja, Walter Stechele. Pruning CNNs for LiDAR-based Perception in Resource Constrained Environments. In *IEEE Intelligent Vehicles Symposium Workshops, 3D-DLAD*, 2021.

[27] Manoj Rohit Vemparala, Nael Fasfous, Alexander Frickenstein, Mhd Ali Moraly, Aquib Jamal, Lukas Frickenstein, Christian Unger, Naveen Shankar Nagaraja, Walter Stechele. L2PF - Learning to Prune Faster. In *International Conference on Computer Vision Image Processing*, 2020.

[28] Manoj Rohit Vemparala, Nael Fasfous, Alexander Frickenstein, Sreetama Sarkar, Qi Zhao, Sabine Kuhn, Lukas Frickenstein, Anmol Singh, Christian Unger, Naveen Nagaraja, Christian Wressnegger, Walter Stechele. Adversarial robust model compression using in-train pruning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2021.

[29] Manoj Rohit Vemparala, Nael Fasfous, Lukas Frickenstein, Alexander Frickenstein, Anmol Singh, Driton Salihu, Christian Unger, Naveen-Shankar Nagaraja, Walter Stechele. Hardware-aware mixed-precision neural networks using in-train quantization. In *British Machine Vision Conference*, 2021.

[30] Manoj Rohit Vemparala, Alexander Frickenstein, Nael Fasfous, Lukas Frickenstein, Qi Zhao, Sabine Franziska Kuhn, Daniel Ehrhardt, Yuankai Wu, Christian Unger, Naveen Shankar Nagaraja, Walter Stechele. Breakingbed - breaking binary and efficient deep neural networks by adversarial attacks. In *Intelligent Systems Conference*, 2021.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems*, 2012.

[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[33] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, 2019.

[34] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[35] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. In *The European Conference on Computer Vision*, 2018.

[36] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, 2010.

[37] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 1999.

[38] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[39] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016.

[40] Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *Asian Conference on Pattern Recognition*, 2015.

[41] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Conference on Computer Vision and Pattern Recognition*, 2015.

[42] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. 2009. University of Toronto.

[43] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014.

[44] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE international conference on computer vision*, 2015.

[45] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, 2016.

[46] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[47] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[48] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, 2016.

[49] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.

[50] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *ArXiv*, abs/1606.06160, 2016.

[51] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *ArXiv*, abs/1603.01025, 2016.

[52] Sebastian Vogel, Mengyu Liang, Andre Guntoro, Walter Stechele, and Gerd Ascheid. Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base. In *International Conference on Computer-Aided Design*, 2018.

[53] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *ArXiv*, abs/1602.02830, 2016.

[54] Alexander Frickenstein, Manoj Rohit Vemparala, Jakob Mayr, Naveen Shankar Nagaraja, Christian Unger, Federico Tombari, and Walter Stechele. Binary DAD-Net: Binarized Driveable Area Detection Network for Autonomous Driving. *IEEE International Conference on Robotics and Automation*, 2020.

[55] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Proceedings of the European Conference on Computer Vision*, 2016.

[56] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European Conference on Computer Vision*, 2018.

[57] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. Structured binary neural networks for accurate image classification and semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[58] Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. Latent weights do not exist: Rethinking binarized neural network optimization. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[59] Zhongnan Qu, Zimu Zhou, Yun Cheng, and Lothar Thiele. Adaptive loss-aware quantization for multi-bit networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[60] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie. Performance evaluation and optimization of hbm-enabled gpu for data-intensive applications. In *Design, Automation Test in Europe Conference Exhibition*, 2017.

[61] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[62] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture*, 2016.

[63] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[64] Arria 10 FPGA Development Kit Overview. `https://www.intel.com/content/www/us/en/programmable/documentation/iga1437675412911.html`.

[65] Intel, Corporation. *Arria 10 Native Fixed Point DSP IP Core User Guide*, 3 2017.

[66] Intel, Corporation. *Arria 10 Native Floating-Point DSP Intel FPGA Guide*, 6 2017.

[67] Xilinx, Inc. *ZCU102 Evaluation Board*, 6 2019. v1.6.

[68] Xilinx, Inc. *UltraScale Architecture DSP Slice*, 5 2019. v1.8.

[69] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian J. Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness. *CoRR*, abs/1902.06705, 2019.

[70] Ian j. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*, 2015.

[71] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial Machine Learning at Scale. 2016.

[72] Ali Shafahi, Mahyar Najibi, Mohammad Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. Adversarial training for free! In *Advances in Neural Information Processing Systems*. 2019.

[73] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2015.

[74] Rey Reza Wiyatno, Anqi Xu, Ousmane Dia, and Archy de Berker. Adversarial examples in modern machine learning: A review, Nov 2019.

[75] Nicholas Carlini and David A. Wagner. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy*, May 2017.

[76] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *IEEE Symposium on Security and Privacy*, 2016.

[77] Naveed Akhtar and Ajmal S. Mian. Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey. *IEEE Access*, 2018.

[78] Moustafa Alzantot, Yash Sharma, Supriyo Chakraborty, Huan Zhang, Cho-Jui Hsieh, and Mani B. Srivastava. GenAttack: Practical Black-Box Attacks with Gradient-Free Optimization. In *ACM Genetic and Evolutionary Computation Conference (GECCO)*, 2019.

[79] Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple Black-Box Adversarial Attacks on Deep Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2017.

[80] Alexander Frickenstein, Christian Unger, and Walter Stechele. Resource-Aware Multicriterial Optimization of DNNs for Low-Cost Embedded Applications. In *CRV*, Kingston, Ontario, Canada, 2019.

[81] Nael Fasfous, Manoj-Rohit Vemparala, Alexander Frickenstein, and Walter Stechele. OrthrusPE: Runtime Reconfigurable Processing Elements for Binary Neural Networks. In *Design, Automation Test in EuropeConference Exhibition*, 2020.

[82] Alexander Frickenstein, Manoj-Rohit Vemparala, Christian Unger, Fatih Ayar, and Walter Stechele. DSC: Dense-Sparse Convolution for Vectorized Inference of Convolutional Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[83] Nael Fasfous, Manoj-Rohit Vemparala, Alexander Frickenstein, Mohamed Badawy, Felix Hundhausen, Julian Höfer, Christian Unger Naveen-Shankar Nagaraja, Hans-Jörg Vögel, Jürgen Becker, Tamim Asfour, and Walter Stechele. Binary-LoRAX: Low-power and Runtime Adaptable XNOR Classifier for Semi-Autonomous Grasping with Prosthetic Hands. In *International Conference on Robotics and Automation*, 2021.

[84] Nael Fasfous, Manoj-Rohit Vemparala, Alexander Frickenstein, Lukas Frickenstein, Mohamed Badawy, and Walter Stechele. BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices. In *IPDPS-RAW*, 2021.

[85] Manoj-Rohit Vemparala, Alexander Frickenstein, and Walter Stechele. An Efficient FPGA Accelerator Design for Optimized CNNs Using OpenCL. In *Architecture of Computing Systems*, 2019.

[86] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry

Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[87] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019.

[88] Tensorrt: Nvidia. 2018.

[89] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems*, 2018.

[90] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. 2018.

[91] R.-J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

[92] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 2002.

[93] Pierpaolo Mori, Manoj Rohit Vemparala, Nael Fasfous, Saptarshi Mitra, Sreetama Sarkar, Alexander Frickenstein, Lukas Frickenstein, Domenik Helms, Naveen Shankar Nagaraja, Walter Stechele, Claudio Passerone. Accelerating and pruning cnns for semantic segmentation on fpga. In *Design Automation Conference*, 2022.

[94] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*. Morgan Kaufmann Publishers Inc., 1990.

[95] Babak Hassibi, David G. Stork, Gregory Wolff, and Takahiro Watanabe. Optimal Brain Surgeon: Extensions and Performance Comparisons. In *Advances in Neural Information Processing Systems*, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[96] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, 2015.

[97] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, 2016.

[98] Alexander Frickenstein, Manoj-Rohit Vemparala, Nael Fasfous, Laura Hauenschild, Naveen-Shankar Nagaraja, Christian Unger, and Walter Stechele. Alf: Autoencoder-based low-rank filter-sharing for efficient convolutional neural networks. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, 2020.

[99] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

[100] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, and Hartwig Sze, Vivienneand Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *The European Conference on Computer Vision*. Springer International Publishing, 2018.

[101] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. HW-NAS-bench: Hardware-aware neural architecture search benchmark. In *International Conference on Learning Representations*, 2021.

[102] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020.

[103] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Péter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[104] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *International Symposium on Computer Architecture*, 2018.

[105] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2019.

[106] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[107] R. Venkatesan, Y. Shao, Miaorong Wang, Jason Clemons, S. Dai, M. Fojtik, Ben Keller, Alicia Klinefelter, N. Pinckney, Priyanka Raina, Y. Zhang, B. Zimmer, W. Dally, J. Emer, Stephen W. Keckler, and B. Khailany. Magnet: A modular accelerator generator for neural networks. In *International Conference on Computer-Aided Design*, 2019.

[108] T. Yang, Y. Chen, and V. Sze. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017.

[109] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration Systems*, 2017.

[110] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *IEEE/ACM International Conference on Computer-Aided Design*, 2016.

[111] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[112] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016.

[113] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. In *arXiv preprint arXiv:1904.07850*, 2019.

[114] Bin Yang, Wenjie Luo, and Raquel Urtasun. PIXOR: real-time 3d object detection from point clouds. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[115] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2018.

[116] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven L. Waslander. Joint 3d proposal generation and object detection from view aggregation. In *International Conference on Intelligent Robots and Systems*, 2018.

[117] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017.

[118] Charles R. Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J. Guibas. Frustum pointnets for 3d object detection from RGB-D data. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[119] Ming Liang, Bin Yang, Shenlong Wang, and Raquel Urtasun. Deep continuous fusion for multi-sensor 3d object detection. In *Proceedings of the European Conference on Computer Vision*, 2018.

[120] Yan Yan, Yuxing Mao, and Bo Li. SECOND: Sparsely Embedded Convolutional Detection. *Sensors*, 2018.

[121] M. Simon, S. Milz, Karl Amende, and H. Groß. Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds. In *Proceedings of the European Conference on Computer Vision Workshops*, 2018.

[122] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017.

[123] Yangyan Li, Rui Bu, Mingchao Sun, Wei Wu, Xinhan Di, and Baoquan Chen. Pointcnn: Convolution on x-transformed points. In *Advances in Neural Information Processing Systems*, 2018.

[124] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph CNN for learning on point clouds. *ACM Trans. Graph.*, 2019.

[125] Martin Engelcke, Dushyant Rao, Dominic Zeng Wang, Chi Hay Tong, and Ingmar Posner. Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks. In *In International Conference on Robotics and Automation*, 2017.

[126] Bo Li. 3d fully convolutional network for vehicle detection in point cloud. In *International Conference on Intelligent Robots and Systems*. IEEE, 2017.

[127] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *In IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2017.

[128] Weijing Shi and Ragunathan Rajkumar. Point-gnn: Graph neural network for 3d object detection in a point cloud. *CoRR*, abs/2003.01251, 2020.

[129] Yilun Chen, Shu Liu, Xiaoyong Shen, and Jiaya Jia. Fast point R-CNN. *CoRR*, abs/1908.02990, 2019.

[130] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[131] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Sjalander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *Field Programmable Logic and Applications*, 2018.

[132] Ahmed T. Elthakeb, Prannoy Pilligundla, Fatemehsadat Mireshghallah, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. ReleQ : A reinforcement learning approach for automatic deep quantization of neural networks. *IEEE Micro*, 2020.

[133] Qian Lou, Feng Guo, Minje Kim, Lantao Liu, and Lei Jiang. AutoQ: Automated Kernel-Wise Neural Network Quantization. In *International Conference on Learning Representations*, 2020.

[134] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *European Conference on Computer Vision*, 2018.

[135] Tianyun Zhang, Kaiqi Zhang, Shaokai Ye, Jiayu Li, Jian Tang, Wujie Wen, Xue Lin, Makan Fardad, and Yanzhi Wang. ADAM-ADMM: A unified, systematic framework of structured weight pruning for dnns. *CoRR*, abs/1807.11091, 2018.

[136] D. Mocanu, Elena Mocanu, P. Stone, P. Nguyen, M. Gibescu, and A. Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 2018.

[137] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *ArXiv*, abs/1907.04840, 2019.

[138] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *Proceedings of Machine Learning and Systems*. 2020.

[139] Ahmed T. Elthakeb, Prannoy Pilligundla, Fatemehsadat Mireshghallah, Tarek Elgindi, Charles-Alban Deledalle, and Hadi Esmaeilzadeh. WaveQ: Gradient-based deep quantization of neural networks through sinusoidal adaptive regularization. *ArXiv*, abs/2003.00146, 2020.

[140] B. Wu, Y. Wang, P. Zhang, Yuandong Tian, Péter Vajda, and K. Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *ArXiv*, abs/1812.00090, 2018.

[141] Jing Liu, Bohan Zhuang, Peng Chen, Yong Guo, Chunhua Shen, Jianfei Cai, and Mingkui Tan. ABS: Automatic Bit Sharing for Model Compression. *ArXiv*, abs/2101.04935.

[142] Stefan Uhlich, Lukas Mauch, Fabien Cardinaux, Kazuki Yoshiyama, Javier Alonso Garcia, Stephen Tiedemann, Thomas Kemp, and Akira Nakamura. Mixed Precision DNNs: All you need is a good parametrization. In *International Conference on Learning Representations*, 2020.

[143] Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[144] R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. Neural Information Processing Systems, 1999.

[145] S. Mohamed, M. Rosca, M. Figurnov, and A. Mnih. Monte carlo gradient estimation in machine learning. *CoRR*, abs/1906.10652, 2019.

[146] H.-V. Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, 2016.

[147] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition 2016*.

[148] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016.

[149] Vikash Sehwag, Shiqi Wang, Prateek Mittal, and Suman Jana. Hydra: Pruning adversarially robust neural networks. *Conference on Neural Information Processing Systems*, 2020.

[150] Yoshua Bengio, N. Léonard, and Aaron C. Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *ArXiv*, abs/1308.3432, 2013.

[151] Micah Goldblum, Liam Fowl, S. Feizi, and T. Goldstein. Adversarially robust distillation. In *Association for the Advancement of Artificial Intelligence*, 2020.

[152] Nicolas Papernot and P. McDaniel. Extending defensive distillation. *ArXiv*, abs/1705.05264, 2017.

[153] Angus Galloway, Graham W. Taylor, and Medhat Moussa. Attacking Binarized Neural Networks. In *International Conference on Learning Representations*, 2018.

[154] Shaokai Ye, Kaidi Xu, Sijia Liu, Hao Cheng, Jan-Henrik Lambrechts, Huan Zhang, Aojun Zhou, Kaisheng Ma, Yanzhi Wang, and Xue Lin. Adversarial robustness vs. model compression, or both? In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019.

[155] Souvik Kundu, Mahdi Nazemi, P. Beerel, and M. Pedram. DNR: A tunable robust pruning framework through dynamic network rewiring of dnns. *2021 26th Asia and South Pacific Design Automation Conference*, 2021.

[156] Shupeng Gui, Haotao Wang, Haichuan Yang, Chen Yu, Zhangyang Wang, and Ji Liu. Model compression with adversarial robustness: A unified optimization framework. In *Conference on Neural Information Processing Systems*, 2019.

[157] Yonglong Tian, Dilip Krishnan, and Phillip Isola. Contrastive representation distillation. In *International Conference on Learning Representations*, 2020.

[158] Nvidia. Nvidia turing gpu architecture. In *https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, year=2017*.

[159] Sourabh Vora, Alex H. Lang, Bassam Helou, and Oscar Beijbom. Pointpainting: Sequential fusion for 3d object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[160] Tianwei Yin, Xingyi Zhou, and Philipp Krähenbühl. Multimodal virtual point 3d detection. *Advances in Neural Information Processing Systems*, 2021.

[161] Haoping Bai, Meng Cao, Ping Huang, and Jiulong Shan. Batchquant: Quantized-for-all architecture search with robust quantizer. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[162] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[163] Minghao Guo, Yuzhe Yang, Rui Xu, Ziwei Liu, and Dahua Lin. When NAS Meets Robustness: In Search of Robust Architectures Against Adversarial Attacks. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[164] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Edoardo Debenedetti, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. Robustbench: a standardized adversarial robustness benchmark. *arXiv preprint arXiv:2010.09670*, 2020.

[165] Klim Kireev, Maksym Andriushchenko, and Nicolas Flammarion. On the effectiveness of adversarial training against common corruptions. *ArXiv*, abs/2103.02325, 2021.