LAB

Learnable Activation Binarizer for Binary Neural Networks

Sieger Falkena

Master Thesis

TU Delft

LAB

Learnable Activation Binarizer for Binary Neural Networks

by

Sieger Falkena

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday March 24, 2022 at 14:00.

Student number:	4293681		
Project duration:	March, 2021 – March, 2022		
Thesis committee:	dr. ir. H. Jamali-Rad,	Shell[1] and TU Delft,	Daily Supervisor
	dr. ir. J. van Gemert,	TU Delft,	Advisor
	dr. ir. G. Iosifidis,	TU Delft,	External committee member

[1] Shell Global Solutions International B.V. Amsterdam

An electronic version of this thesis is available at `http://repository.tudelft.nl/` best viewed in colour.

TU Delft

Preface

This report presents the work done for my Master thesis project "LAB: Learnable Activation Binarizer for Binary Neural Networks", and describes my work during my internship at Shell Global Solutions International B.V. Amsterdam. During both my internship and thesis, I have been working on combining two of my greatest interests: Embedded Systems and Computer Vision, to aid the development of energy-efficient deep learning models.

I would like to start by expressing my deepest appreciation for dr. Jamali-Rad for not only being my daily supervisor, but being my mentor, truly caring about my professional and personal path. I cannot thank you enough for the time (and coffee) you have spend with me. I would like to thank dr. J.C. van Gemert for his sharp and critical ideas, making my research strong in its fundament. Lastly I would like to thank dr. G. Iosifidis for his interest in the project and for being part of the Exam Committee.

Finally, I would like to thank my girlfriend, family and friends for their support during my Masters.

Sieger Falkena
Hichtum, March 2022

Nomenclature

AI	Artificial Intelligence	1
ASIC	Application-Specific Integrated Circuit	26
BNN	Binary Neural Networks	1
CNN	Convolutional Neural Network	1
CPU	Central Processing Unit	19
DLA	Deep Learning Accelerator	26
FP	Floating Point	1
FPGA	Field-Programmable Gate Array	26
HPC	High Performance Computing	1
MAC	Multiply-Accumulate	14
ML	Machine Learning	14
MLP	Multi Layer Perceptron	14
NAS	Neural Architecture Search	21
NPU	Neural Processing Unit	27
ONNX	Open Neural Network Exchange	22
QAT	Quantization Aware Training	28
STE	Straight Through Estimator	20
TAO	Train, Adapt, Optimize	28
TLT	Transfer Learning Toolkit	28
TPU	Tensor Processing Unit	26
TRT	TensorRT	28

Contents

1 Introduction

1.1. Context of research

Many people think that terms like "Artificial Intelligence" (AI) are very new. However, the first time that the term AI was mentioned already dates back to the fifties, where it was introduced on the Dartmouth Summer Research Project on Artificial Intelligence [35]. In the early stages, the technology had difficulties evolving due to limited computing power available and this lead to the first so called "AI winter". With technology evolving with Moore's law, interest in AI grew again, but to a lack of funding, the second "AI winter" was inevitable during the nineties. It was not before the year 2010 that the technology started to evolve again. As computing hardware matured, the sector of AI technology, together with terms like "Deep Learning", "Big data" and "Computer Vision" has been growing exponentially.

The first neural networks were rather small, but as computing hardware kept maturing, more and more sophisticated networks have been designed. This has evolved to a point where networks are so complex, that most of the state-of-the-art neural networks cannot be trained on normal consumer hardware, but need to be trained on High Performance Computing (HPC) solutions or cloud-based solutions. This instantiates the need for energy-efficient deep learning solutions, which is twofold:

- Energy-efficient models decrease the costs associated with training and inference. These costs apply to both the carbon footprint that these models have, as the electricity (and with that money) it costs to train models on cloud-computing solutions. To give context to this statement, Patterson et. al [36] showed that in order to train a model such as GPT-3 (a language model), $1,287 MWh$ of energy is needed, which produces 552.1 metric tons of CO_2e when trained on Nvidia V100's.

- Not all deep learning devices can rely on a connection with a computation server. In many cases, training a model on such a server is not a big issue, but running inference is preferred to happen on-device. For safety-critical systems for example, there is a need for real-time decision making. Therefore, it is not an option to rely on an external server which goes paired with undesired latencies and possible reliability issues. A second example are devices which can physically not connect to the internet due to their use in remote areas. Most of these so called "edge devices" are embedded devices which are battery powered with relatively little processing power, indicating the need for energy-efficient deep learning algorithms.

Although the research in the field of deep learning mainly has been focusing on achieving high accuracies, the need for energy-efficient deep learning slowly trickles trough as well. Several research fields have emerged to make models more efficient, which will be introduced in later chapters. One of these fields are Binary Neural Networks (BNN), which apply extreme 1-bit quantization of weights and activations to a Convolutional Neural Network (CNN). One can understand that by quantizing the values of a CNN this aggressively, a great portion of information is lost. Therefore, there is still a gap between the accuracy of Floating Point (FP) networks and their binary counterpart. Improving these BNNs to close the accuracy gap is the main point of this thesis. Therefore in the next section, we come to the problem statement.

1.2. Problem statement

In this section, the problem-to-solve is introduced, together with the main questions that need to be answered. The main objective of this research is:

To design and implement an activation binarization function that replaces the traditionally used `sign(.)` function used in Binary Neural Networks, so that more information is preserved in the network.

In order to reach this objective, the following tasks need to be solved:

- Find the relevant state-of-the-art regarding binarization functions for both weights and activations.
- Design and implement the new binarization method, such that the additional latency of the module with respect to the network is kept within limits.
- Ensure the new binarization method is modular, so it can be plugged into existing networks.
- Design and implement a state-of-the-art architecture around the new binarization method and test its performance on a large scale dataset, so that it can be compared against other state-of-the-art.

1.3. Internship

The thesis work was conducted as part of an internship at Shell Global Solutions International B.V. Amsterdam, The Netherlands.

> "Shell's target is to become a net-zero emissions energy business by 2050, in step with society's progress in achieving the goal of the UN Paris Agreement on climate change ... Becoming a net-zero emissions business means offering customers more low-carbon products, from renewable electricity, to charging for electric vehicles and hydrogen." [1]

AI can help to achieve this:

> "Digital and AI can also reduce emissions in the way we move around. Shell is working with customers and partners in the shipping industry to help accelerate decarbonisation towards a net-zero emissions future for shipping." [2]

1.4. Contributions

- The basics of convolutional neural networks and binary neural networks are introduced.
- In our paper, we introduce the *uniqueness bottleneck* in binary neural networks imposed by the traditionally used binarization function.
- We present LAB: a universal activation binarization function that can learn a per-channel binarization kernel. We further show this module can readily be plugged into existing BNN architectures.
- We bring deep learning model compression into practice and bring an object detection model as well as an instance segmentation model to a resource-constrained edge device.

1.5. Thesis outline

The rest of this thesis report is structured as follows: This thesis starts by presenting the scientific paper in **chapter 2**. This paper is a informative summary of the main thoughts of this thesis and describes the main experiments conducted. After the paper, we start with a wide angle of view by providing background information about Deep Learning in **chapter 3**. After these fundamentals are clear, the topic is narrowed down to Binary Neural Networks in **chapter 4**. **Chapter 5** provides additional materials for the novel `LAB` as introduced in the paper. The foremost chapter: **chapter 6**, describes the need for new technologies from a practical viewpoint and shows a real implementation of applying AI at the edge on actual hardware. Finally, this thesis is is closed by a conclusion together with recommendations for future research in **chapter 7**.

[1] https://www.shell.com/energy-and-innovation/the-energy-future/our-climate-target.html, accessed 11 march 2022
[2] https://www.shell.com/energy-and-innovation/digitalisation/news-room/can-digitalisation-and-ai-accelerate-the-energy-transition.html, accessed 11 march 2022

2

Scientific Paper

LAB: Learnable Activation Binarizer for Binary Neural Networks

Sieger Falkena
Delft University of Technology
Delft, The Netherlands
s.t.falkena@student.tudelft.nl

Hadi Jamali-Rad
Delft University of Technology
Delft, The Netherlands
h.jamalirad@tudelft.nl

Jan van Gemert
Delft University of Technology
Delft, The Netherlands
j.c.vangemert@tudelft.nl

Abstract

Binary Neural Networks (BNNs) are receiving an upsurge of attention for bringing power-hungry deep learning towards edge devices. The traditional wisdom in this space is to employ `sign(.)` *for binarizing featuremaps. We argue and illustrate that* `sign(.)` *is a* uniqueness bottleneck*, limiting information propagation throughout the network. To alleviate this, we propose to dispense* `sign(.)`*, replacing it with a learnable activation binarizer (*`LAB`*), allowing the network to learn a fine-grained binarization kernel per layer - as opposed to global thresholding.* `LAB` *is a novel universal module that can seamlessly be integrated into existing architectures. To confirm this, we plug it into four seminal BNNs and show a considerable performance boost at the cost of tolerable increase in delay and complexity. Finally, we build an end-to-end BNN (coined as* `LAB-BNN`*) around* `LAB`*, and demonstrate that it achieves competitive performance on par with the state-of-the-art on ImageNet*[1].

1. Introduction

Convolutional Neural Networks (CNNs) dominate the current state-of-the-art computer vision tasks. With evolving research, models gained increasingly higher accuracy, but in parallel they have grown in size and complexity. This imposes a significant burden for deploying deep learning models on resource-constrained edge devices. Recent studies explore model compression techniques to reduce model size and latency, such as pruning [25], quantization [34], knowledge distillation [13], neural architecture search [10] and low rank approximation [36]. The most extreme form of quantization is realized by binarization, resulting in binary weights and activations $\{-1, +1\}$. Networks utilizing this form of quantization are known as Binary Neural Networks (BNNs) and promise a bright future for energy-efficient deep learning. By quantizing weights and activations aggressively, one can theoretically achieve a memory reduction of $32\times$ and a computational speedup of $58\times$ on typical CPUs [30].

The current consensus in literature is to use `sign(.)` as a mapping from the full-precision to binary values. However, this imposes three widely-known issues: (i) the representational power of `sign(.)` with respect to the floating point counterpart decreases from 2^{32} to only 2 information levels [26]; (ii) the derivative of `sign(.)` is a Dirac Delta returning a zero gradient almost everywhere [7]; (iii) `sign(.)` imposes a global threshold, treating all values in its input the same way. In this work, we identify and debate about a fourth problem which we refer to as *uniqueness bottleneck*. Approaching the problem from both qualitative and quantitative angles, we demonstrate that using `sign(.)` further limits the representational capacity of the network. Interestingly, several studies state that `sign(.)` is a sub-optimal binarization operation and that it is not straightforward to find a new binarization function [6, 32]. This is exactly why we embark on this challenge in this paper.

Multiple remedies have been proposed to cope with the issues of `sign(.)`, including the introduction of scaling factors [30], gradient approximation [28] and pre-binarization distribution shaping [18]. Amongst these directions, we believe that the pre-binarization reshaping shows the most potential to alleviate this information bottleneck of BNNs. In contrast to the existing studies, we argue that shaping the pre-binarization distribution is a means an the end, and not the end in itself. To address the issues enumer-

[1] Codebase in the supplementary will be made publicly available upon acceptance.

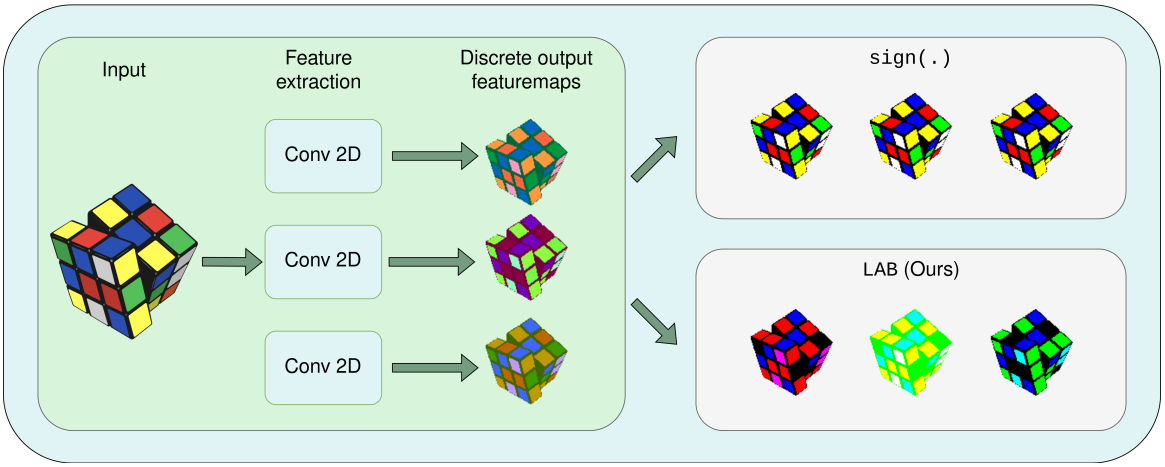


Figure 1: In contrast to `sign(.)`, binarization using `LAB` does not map some of the discrete output featuremaps into the same binary featuremap, but learns to distinguish important features during binarization, improving information propagation capacity. The figure is best viewed in colour.

ated, we design a *learnable* activation binarization function (`LAB`) to automate the mapping from the full-precision featuremaps to the binary counterparts, so that the representational capacity of the network (compared to full-precision) is least impacted. This is shown schematically in Fig. 1 (and elaborated in Section 3), where application of a global `sign(.)` threshold on the diverse spectrum of colors in the discrete featuremaps results in identical outputs (acting like a diversity bottleneck), whereas `LAB` can potentially avoid such loss of information.

Our contributions can be summarized as follows:

- To the best of our knowledge, for the first time, we identify the *uniqueness bottleneck* imposed by the `sign(.)` operation. We demonstrate that `sign(.)` limits the representational capacity of binary featuremaps.
- To address this bottleneck, we introduce a novel learnable activation binarization: `LAB`. We show that `LAB` is a universal module that can readily be plugged into any existing BNN architecture, and improve its performance. Our experimentation on four seminal BNN baselines corroborates this claim.
- We build an end-to-end network around `LAB` (coined as `LAB-BNN`) and demonstrate that it offers competitive performance (64.2% Top-1 validation accuracy) on par with the state-of-the-art in this space on ImageNet.

2. Related Work

2.1. Binary Neural Networks (BNNs)

Current BNNs binarize the full precision weights and activations by applying `sign(.)` on them:

$$x_b = \texttt{sign}\,(x_r) = \begin{cases} +1, \text{ if } x_r > 0 \\ -1, \text{ if } x_r \leq 0 \end{cases}, \qquad (1)$$

where x_b and x_r denote the binary and real (full precision) values, respectively. Naively applying these quantizations to a CNN yields low accuracy and to close the gap between BNN implementations and their real-valued counterparts, several research directions have arisen: minimization of quantization error [7, 30], loss function improvement [9, 16, 24], gradient approximation [21, 24, 28], different network architecture designs [5, 8, 17, 24, 26, 39], training strategies [1, 23, 27, 31] and binary inference engines [3, 12, 35, 37] are a few seminal directions that have been explored in BNN literature. Apart from these main directions, only a few studies investigate new methodologies for binarizing weights and activations, which will be elucidated next.

Weight binarization. A novel approach for weight binarization is presented in [14] where first a BNN is trained and both full precision and binary weights are employed as noisy supervisors for learning a mapping towards the final binary weights. As this mapping is learnable, it can exploit the relationships between weights. SiMaN [20] and RBNN [21] both propose a new binarization method based on so-called angle alignment between the full-precision and binary weights.

Activation binarization. One way to approach activation binarization is through classic computer vision techniques, such as dithering. This technique can binarize an image in a way that shifts quantization error towards higher frequencies. As the human visual system is more receptive to lower frequencies, the binarized image is perceived as having a low quantization error, and thus, carrying more information. A realization of this idea is called DitherNN but it only reports mild improvement [2]. Most activation binarization approaches focus on shaping the pre-binarization distribution, from which a higher entropy can be achieved after binarization [28]. For instance, an extra regularization term is proposed in [9] to explicitly shape the pre-binarization distribution so that it counteracts degeneration, saturation, and gradient mismatch problems. It is argued in [18] that BNNs benefit from an unbalanced pre-binarization distribution. ReActNet [24] argues that BNNs benefit from learning a similar activation distribution as their full-precision counterparts. On a related note, Si-BNN [32] approaches the activation binarization problem from a somewhat different angle and introduces sparsity in the activation binarization process.

Even though these studies show promising performance results for BNNs, we argue that changing the pre-binarization distribution is still a form of adaptive global thresholding, and thus a sub-optimal approach. Therefore, the output featuremap does not fully reflect on, and adapt to the local information of the input featuremap.

3. Problem Formulation

In this section, we reflect on the fundamental limitation a `sign(.)` binarization function inflicts on a BNN. We examine the capability of the network to cope with the single threshold value of `sign(.)`, which we refer to as *global* thresholding - given that it is applied similarly on every input location. In practice, as the input featuremap to `sign(.)` is the output of a previous (convolutional) layer, the kernel of the convolution is learned such that parts of the output featuremap get pushed above or pulled beneath the global threshold value. The batch normalization layer can further guide this process by effectively shifting the threshold value. Although this combination is essential for the learning process of BNNs, we argue that there still is a limitation in its efficacy.

Fig. 2 explains what we perceive as the bottleneck in information propagation of BNNs. Here, $\mathbf{A}$ denotes the discrete input featuremap to the binary convolution. Assume $\mathbf{W}$'s represent the set of all unique weight kernels one can imagine. Given a kernel size k, the number of input channels C, and a single output channel per kernel, the total number of unique $\mathbf{W}_i$'s, $\forall i \in [n]$ will then be $n = 2^{k^2 \times C}$. The output featuremaps $\mathbf{D}_1$ to $\mathbf{D}_n$ are discrete finite-alphabet tensors. Note that n in this case is smaller than the theoretical maximum number of unique activations $N = (k^2 \times C)^{H \times W}$, given a specific input $\mathbf{A}$. Impacted by the activation design of the previous layers, proper design of $\mathbf{A}$ could potentially minimize the gap between n and N. Applying `sign(.)` on $\mathbf{D}_1$ to $\mathbf{D}_n$ maps them to their binary counterparts $\mathbf{B}_1$ to $\mathbf{B}_n$. In theory, it is possible to have n unique binary featuremaps $\mathbf{B}_i$'s, $\forall i \in [n]$, even though in practice, the bottleneck of `sign(.)` maps several of $\mathbf{D}_i$'s into the same binary map, thereby further reducing the total number of unique binary featuremaps $\mathbf{B}_i$'s beneath even n. We dub the aforementioned issue as the *uniqueness bottleneck*. We argue that due to this bottleneck, the network does not utilize its full potential and the representational capacity of the network is going to be impacted.

Table 1: η for 512 unique $\mathbf{W}$'s after applying `sign(.)` function in Bi-Real Net. Later layers show a lower ratio, indicating that the uniqueness bottleneck is more prominently present.

Layer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
η	0.964	0.994	0.996	0.998	0.998	0.986	0.991	0.994
Layer	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
η	0.994	0.927	0.943	0.951	0.959	0.747	0.781	0.803

To demonstrate that this hypothesis is valid, we design a toy experiment that uses single-layer binary input featuremaps $\mathbf{A}$ (in this case with $C = 1$) extracted from the binarized featuremaps (using `sign(.)`) in a trained Bi-RealNet-18 [26]. Kernel size k is set to 3, which makes up for a total of $2^{k^2} = 512$ unique kernels. Following the steps sketched in Fig. 2, we take $\mathbf{A}$ as the starting point, convolve it with every possible kernel $\mathbf{W}_i$ and binarize the output activations $\mathbf{D}_i$'s with the `sign(.)` function. We then count the number of unique binary featuremaps $\mathbf{B}_i$'s, and average over all the channels (per different layers) of 20 different input images. We denote the ratio of counted unique featuremaps (n_c), and theoretical total number of unique featuremaps (n_t) as the uniqueness ratio $\eta = n_c/n_t$. The results are shown in Table 1. We can see that going deeper with convolutions, leading to smaller featuremaps for layers 9 to 16, the uniqueness ratio decreases and the bottleneck becomes more evident. In the next section, we propose a learnable activation binarizer (`LAB`) as a remedy for this bottleneck.

4. The Proposed Method: `LAB`

One possible approach towards addressing the bottleneck of `sign(.)` binarization is to find a mapping from full-precision activation values to corresponding binary values, in such a way that the embedded spatial information from the input featuremap is preserved. To do so, we pro-

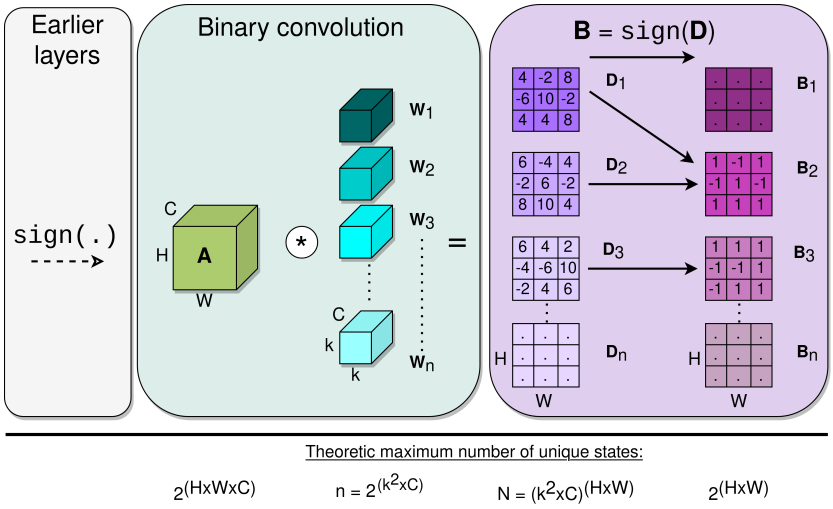


Figure 2: The *uniqueness bottleneck*. Activation $\mathbf{A}$ is convolved with all unique kernels $\mathbf{W}_i$'s. The finite-alphabet featuremaps $\mathbf{D}$ are binarized by the `sign(.)`, which creates the bottleneck of multiple $\mathbf{D}$'s mapping to the same binary featuremap $\mathbf{B}$. The equations (*at the bottom*) indicate the theoretical maximum number of unique combinations a tensor can take.

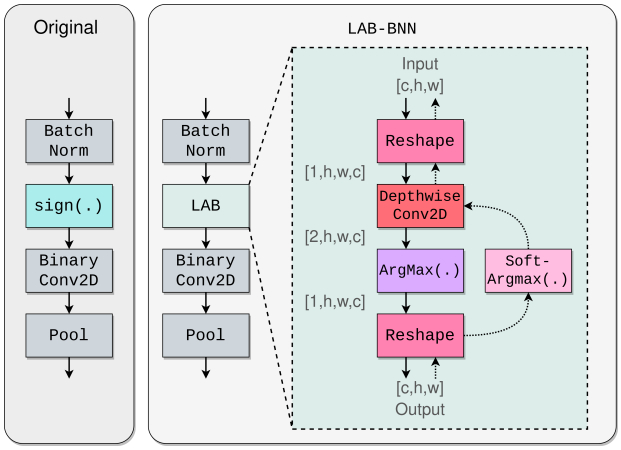


Figure 3: Overview of `LAB` and how it can be used similar to `sign(.)`. Tuples $[\{1,2\},H,W,C]$ indicate the shape of the tensors. The depthwise convolution together with `ArgMax(.)` form the core components of `LAB`. The `Soft-Argmax(.)` is used in the backward pass for differentiability.

pose to forge a different path in contrast to the current wisdom of activation distribution shaping [9,18,24]. More concretely, we propose a novel learnable activation binarizer (`LAB`) to *learn* a binarization kernel per layer, as shown in Fig. 3. The figure demonstrates `LAB` as a building block of a standard BNN. Zooming into the `LAB` unit, as we need to apply channel-wise binarization like `sign(.)`, we first reshape the input for per-channel operations. To capture local spatial information per channel, we apply a 3×3 depthwise convolution with a channel multiplier of 2. The core idea behind this channel doubling is to construct a *miniature segmentation layer* within the `LAB` unit to classify the input as -1 or $+1$. This classification is done through an `ArgMax(.)` across both channels, reducing the featuremap back to a single output channel which is finally reshaped back to its original size. As the `ArgMax(.)` is non-differentiable, we apply the `Soft-ArgMax(.)` for the backward pass [11]. Given that we are dealing with only two classes, the `Soft-ArgMax(.)` in (2) simplifies to a single entry of the `SoftMax(.)` with an extra temperature controlling parameter β which controls the "hardness" of the `ArgMax(.)` approximation:

$$\text{Soft-ArgMax}(x) = \sum_{i=0}^{1} \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}} i = \frac{e^{\beta x_1}}{\sum_j e^{\beta x_j}}. \quad (2)$$

5. Experimental Evaluation

In this section, we first reflect on the experimental setup. Next, we demonstrate the uniqueness bottleneck and how employing `LAB` can help alleviate the problem. We then show that `LAB` is beyond a one-off remedy but more of a universal module that can straightforwardly be plugged into existing baselines. Finally, we propose an end-to-end model architecture (`LAB-BNN`) revolving around `LAB` offering competitive performance against state-of-the-art.

5.1. Experimental Setup

To be able to compare against a variety of existing state-of-the-art baselines - also to allow for a wider adoption by the community - we implement LAB both in PyTorch and TensorFlow. For the comparison with the state-of-the-art baselines, we utilize the TensorFlow-based Larq framework [3].

Network structure. To show the versatility of LAB, we plug it into four different architectures. First, XNOR-Net [30], is chosen to examine the improved information flow on an AlexNet-based architecture [19] with no skip connections. Then, Bi-RealNet [26] and ReActNet [24] are used for their ResNet [15] and MobileNet [17] backbones, respectively. Finally, QuickNet [3], an improved version of Bi-RealNet, is used to assess whether LAB can make an impact on a top performing state-of-the-art network.

Proposed end-to-end network: LAB−BNN. Going beyond the proposed module LAB, we also design an end-to-end network based upon the architecture of Bi-RealNet-18[2], which we further enhance by combining PRelu in [7,24,27,31] and STEM modules proposed by QuickNet [3].

Hyperparameters. All experiments are conducted using 4 NVIDIA GeForce GTX 1080 Ti GPUs and follow standard settings in Larq [3], unless otherwise mentioned. To reproduce nominal reported performance, we used a batch size of 128 and a learning rate of $1e^{-4}$ for the re-training of XNORNet. For Bi-RealNet a batch size of 256 and learning rate $2.5e^{-3}$ was used. For Quicknet we used a batch size of 512 and a learning rate of $2.5e^{-3}$. Lastly, we re-trained ReActNet-A from scratch with a batch size of 128 and learning rate of $2.5e^{-3}$. LAB−BNN uses a batch size of 256 and learning rate of $2.5e^{-3}$ and is trained from scratch for 300 epochs. We chose to train from scratch as recent studies [4] suggest that multi-stage training is not needed for high accuracy models, and that it simplifies the training process significantly. The temperature controlling parameter β is made a learnable parameter, initialized with the value 1.0.

Real-time inference on the edge. The research in BNNs is focussed on bringing deep learning to *resource-constrained edge* devices. Recent studies report the computational complexity of their models using theoretical metrics such as floating-point operations (FLOPs) [24,27] multiply-accumulate (MACs) [4] or arithmetic computation effort (ACE) [38]. In coherence with [3, 29] we argue that latency is the best metric to compare model performances. In order to benchmark model latency, we use a resource-constrained edge computing device, Nvidia Jetson Xavier NX[3] development kit, which is an ARM-based board for development of embedded AI systems. Although this device has a built-in GPU, for the benchmarking exercise we only use the CPU. Thus, these benchmarks can be reproduced on commodity ARM64 devices. To convert models from TensorFlow to Jetson-compatible models, we use the Larq Compute Engine Converter [3], which outputs a TensorFlow Lite (TFLite [22]) model. This model can now be evaluated using a Larq benchmark tool [3] adapted from the TFLite benchmark[4]. For all the benchmarking experiments, the power mode of the Jetson is set to 15W, we use the single thread mode, and report values averaged over 50 runs.

5.2. Uniqueness Bottleneck: Qualitative and Quantitative Analysis

In section 3, we have shown that sign(.) can introduce a bottleneck in reaching the theoretical maximum number of unique binary states, which we argued would limit the capacity of BNNs. Here, we adopt a small-scale experiment (followed by large-scale end-to-end experiments in the next subsection) to qualitatively and quantitatively demonstrate that LAB alleviates this bottleneck to some extent. To this aim, we compare the binary featuremaps of a trained Bi-RealNet-18 with a trained Bi-RealNet-18+LAB (where sign(.) is replaced with LAB). The results are shown in Fig. 4 and Table 2. The figure depicts the selected featuremaps from layer 1 and 6 (out of 18 layers). As can be seen, more structural information is preserved in layer 1 of LAB compared to sign(.), even though the features become too abstract for human understanding as we go to the deeper layer 6. Besides this qualitative demonstration, we further quantify the impact of LAB on the number of unique features maps learned by both networks through two (dis)similarity measures: structural similarity loss (SSIM) [33] and a custom-designed metric we call Euclidean norm dissimilarity (ENDSIM), given by:

$$ENDSIM(\mathbf{A}_i, \mathbf{A}_j) = \sqrt{\left(\frac{1}{HW}\sum_{w,h}|\mathbf{A}_i - \mathbf{A}_j|\right)^2 + \left(\frac{1}{HW}\sum_{w,h}|\mathbf{A}_i + \mathbf{A}_j|\right)^2}, \quad h \in [H], w \in [W], i \neq j \in [C] \tag{3}$$

where W and H denote the width and height of the input images in pixels and $\mathbf{A}_i$ and $\mathbf{A}_j$, are different single channel featuremaps being compared. The metric has two terms in which the first term measures discrepancy, and the second one ensures fully inverted featuremaps are not penalized. The latter is because inverted featuremap layers along the channels can occur but do not indicate structural differences. Both measures are computed and averaged over all possible combinations of featuremap layer pairs across all the 16 convolutional layers of 10 randomly selected images of ImageNet passed through both networks. The results are

[2]We started off with ReActNet-A, but we could not reproduce results in TensorFlow.

[3]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/, accessed 3 March 2022

[4]https://www.tensorflow.org/lite/performance/measurement, accessed 3 March 2022

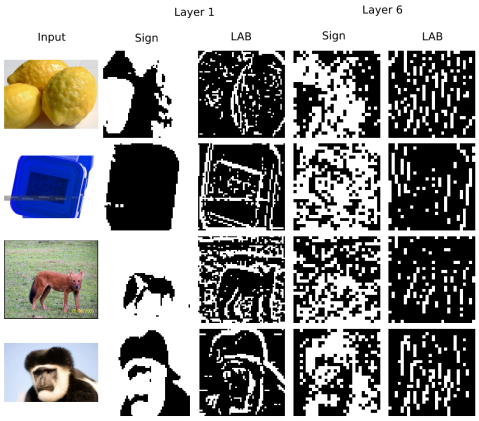


Figure 4: Qualitative comparison between `sign(.)` and `LAB` on two layers of Bi-RealNet-18. `LAB` illustrates higher amount of texture (especially in Layer 1), which indicates allowing more information to pass through.

Table 2: Dissimilarity comparison between pre- and post-binarization of `sign(.)` vs. `LAB` for SSIM and ENDSIM (3) applied to extracted featuremaps from the trained BiRealNet-18 networks (with and without `LAB`). The direction of the arrow indicates higher dissimilarity. The SSIM values are multiplied by $\times 10^3$.

	Layer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SSIM↓ ($\times 10^{-3}$)	**Sign**	94.8	21.3	14.5	13.1	11.9	21.6	16.6	13.5
	LAB	1.5	9.0	4.2	5.4	8.9	5.1	6.6	6.4
ENDSIM↑	**Sign**	1.1	0.93	0.85	0.83	0.8	0.86	0.84	0.8
	LAB	0.92	0.96	1.1	1.3	1.4	1.4	1.5	1.5
	Layer	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
SSIM↓ ($\times 10^{-3}$)	**Sign**	12.2	25.8	19.7	17.6	14.6	29.3	22.4	18.0
	LAB	8.5	4.2	2.9	2.4	8.8	5.6	8.2	7.1
ENDSIM↑	**Sign**	0.76	0.92	0.88	0.85	0.81	0.98	0.93	0.89
	LAB	1.6	1.4	1.4	1.4	1.7	1.4	1.4	1.4

summarized in Table 2, and as can be seen, except for the first layer, `LAB` shows higher dissimilarity (a lower SSIM and a higher ENDSIM) indicating more uniqueness along the channels.

5.3. LAB: A Universal Module

The proposed method `LAB` can readily be applied to any BNN, with no architectural adjustments. In this subsection, we demonstrate this by replacing the `sign(.)` with `LAB` in four seminal BNN architectures and evaluate the impact on the downstream classification task of ImageNet. The results are illustrated in Table 3. As can be seen, at the cost of negligible increase in model size and tolerable increase in delay (max of 2x in [ms]), the boosted architectures XNORNet, Bi-RealNet, QuickNet, and ReActNet offer **3.3%**, **7.9%**, **3.8%** and **1.7%** improved Top-1 accuracies, respectively. The performance boost is slightly less pronounced for Top-5 accuracies. Note that the number of

Table 3: Results of applying `LAB` on the corresponding baselines on ImageNet.

Network	**Backbone**	**Epochs**	**Method**	**Top-1** [%]	**Top-5** [%]	**Model Size** [MB]	**Latency** [ms]
XNOR-Net [30]	AlexNet [19]	60	Sign	44.0	68.1	23.9	50.8
			LAB	46.5	70.3	24.4	55.1
BiRealNet [26]	ResNet-18 [15]	150	Sign	54.4	77.6	4.18	72.5
			LAB	59.1	81.2	4.65	100.6
QuickNet [3]	ResNet-18 [15]	120	Sign	58.7	81.2	4.35	58.1
			LAB	62.5	84.0	4.85	82.4
ReActNet [24]	MobileNetV1 [17]	75	Sign	62.4	83.4	7.74	108.1
			LAB	64.1	84.8	8.69	210.9

required training epochs to reach the nominal performance is dependent upon the architecture itself. The key message from this experiment is that irrespective of the backbone and architecture design, `LAB` makes a considerable impact on all architectures.

5.4. Comparison Against State-of-the-Art

Now that the impact of `LAB` on four seminal BNN architectures is clarified, let us further investigate the efficacy of the proposed end-to-end network (`LAB-BNN`) with `LAB` sitting at its core. To do so, we compare the performance of `LAB-BNN` against the state-of-the-art baselines focused on activation binarization. For the sake of a fair comparison, here we focus on reported Top-1 and 5 validation accuracies on ImageNet with a ResNet-18 backbone. The outcome is summarized in Table 4. As can be seen, both Top-1 and 5 results are competitive with the state-of-the-art and come short of the full precision equivalent network by only about 5%. Notably, ReActNet reports a better performance (that we could not reproduce) in part owing to adopting a multi-stage training strategy, whereas `LAB-BNN` is trained from scratch.

Aside from accuracy, the time and computational complexity of a BNN is just as important. Even though most existing baselines overlook the importance of these metrics and sometimes do not even report them. Following [24] and to solidify our story, in Table 4 we report the total binary and floating-point operations (OP = BOP/64 + FLOP). Here, we achieve a lower total computational complexity (in FLOPs) compared to other baselines including the original Bi-RealNet. To further break this down, in Table 5 we set Bi-RealNet as our reference and state the added (+) complexity per operation listed in the second column. Even though the mechanics of `LAB` adds slight complexity, we compensate for this with a more efficient implementation of `STEM` module [3] leading to an overall decrease of 73.16×10^6 in total FLOPs.

Post-binarization distribution. To illustrate that the post-binarization distribution is not a telling factor about the performance of the network (in contrast to some of the conclusions drawn in [9, 18, 24]), Fig. 5 shows the

Table 4: Comparison against reported state-of-the-art on ImageNet. A dash indicates no value was reported by the original authors.

Network	Method	W/A	Top-1 [%]	Top-5 [%]	BOPs ($\times 10^9$)	FLOPs ($\times 10^8$)	OPs ($\times 10^8$)
ResNet-18	Full-precision	32/32	69.6	89.2	0	18.1	18.1
	Bi-RealNet [26]	1/1	56.4	79.5	1.68	1.39	1.63
	BNN-UAD [18]	1/1	57.2	80.2	-	-	-
	IR-Net [28]	1/1	58.1	80.0	-	-	1.63
	SI-BNN [32]	1/1	59.7	81.8	-	-	-
	SiMaN [20]	1/1	60.1	82.3	-	-	-
	QuickNet [3]	1/1	63.3	84.6	-	-	-
	LAB−BNN	1/1	64.2	85.0	1.68	0.66	0.92
	ReActNet [24]	1/1	65.9	-	-	-	-

Table 5: FLOPs comparison: LAB−BNN vs. Bi-RealNet.

Component	Operation	FLOPs ($\times 10^6$)
LAB (ours)	Depthwise Conv2D	+30.3
	BiasAdd	+1.68
	Multiply	+1.68
	ArgMax	+0.84
	Equal	+0.84
PRelu [31]	Mul	+0.57
	Neg	+0.76
STEM [3]	Conv2D	-109.83
Total		**-73.16**

post-binarization distribution of $+1$'s and -1's for original Bi-RealNet-18 and Bi-RealNet-18+LAB averaged over all channels of 1000 input images. As can be seen, while the distribution of binary values remains almost the same across both networks, the performance of the two (reported in Subsection 5.3) is apart by 7.9%.

Going deep or going LAB? As we discussed, plugging LAB into the standard Bi-RealNet (our base to build LAB−BNN) adds additional complexity to the network, and one could argue that adding this tiny learnable kernel of LAB per layer could virtually help make the network deeper. However, as a counter-argument, we also compare LAB−BNN against a Bi-RealNet with a deeper backbone. More specifically, in the current construct, our method adds a depthwise convolution for each binary layer in Bi-RealNet-18. This is (roughly) equivalent to addition of 16 extra convolution layers, yielding a total of $18 + 16 = 34$ layers. According to [26] Bi-RealNet-34 and Bi-RealNet-50 respectively achieve 62.2% and 62.6% validation accuracies on ImagNet which are both still below the accuracy offered by the proposed LAB−BNN.

6. Ablation Studies

In this section, we further inspect the effect of different components of LAB−BNN. In what follows, we use the same settings listed in Section 5.1, except for the number of epochs shortened to 30. We start off with the original

Table 6: Ablation study for LAB−BNN, trained for 30 epochs on ImageNet.

Method	Top-1 [%]	Top-5 [%]	Model Size [MB]	Latency [ms]
A: BiRealNet [Base]	45.0	69.7	4.18	70.3
B: Base+PRelu	52.4	76.1	4.19	70.8
C: Base+PRelu+LAB	58.1	80.9	4.65	99.3
D: Base+PRelu+LAB+STEM [LAB−BNN]	58.2	80.8	4.62	80.0

Bi-RealNet-18 (model **A**) and update the components progressively towards LAB−BNN (model **D**). We first upgrade model **A** by incorporating PRelu activation [31] resulting in **B**. Model **B** is then extended by replacing sign(.) by the proposed LAB module leading to model **C**. Lastly, we replace the initial convolution in Bi-RealNet with the STEM layer of QuickNet [3], which forms LAB−BNN (model **D**). The results are illustrated in Table 6. As can be seen, plugging LAB into model **B** results in an improvement of about 6% on the Top-1 validation accuracy (and roughly 5% for Top-5), at an increase of 0.46MB in model size and 28.5ms in latency. To make up for the added latency, we apply the STEM layers as proposed by QuickNet [3] which results in decreasing the latency with respect to model **A** down to 9.7ms.

To provide further insights into the distribution of per-operation latencies, in Fig. 6 we profile the network delays for models **A** to **D**[5]. In other words, this is a fine-grained visual breakdown of the total latencies reported in Table 6. The depthwise convolution together with the ArgMax(.) operation in LAB are the main culprits behind the added latency. Additionally, it is clear that the STEM layer indeed helps to alleviate the overall latency as discussed in Table 6.

6.1. Where to apply LAB.

So far, when applying LAB, all instances of sign(.) in every binary layer of the network were replaced with LAB. For Bi-RealNet-18, this means all of the four residual blocks, in all four convolutions per block. As we saw in Table 1, the uniqueness bottleneck is mainly visible in later layers. Therefore, in Table 7 we assess which combination of blocks (out of 16 possible combinations) is most impactful to apply LAB. In doing so, we apply LAB on all layers per selected block. The networks are trained for 30 epochs and we report Top-1 and Top-5 validation accuracies, model size and and latencies. We conclude from the table that omitting LAB (using sign(.) everywhere) leads to the worst results, and see that as we apply LAB in more blocks progressively, accuracy increases. Interestingly, the model with LAB applied to the last 3 blocks (second row) has an exceptional accuracy-latency tradeoff.

[5]These interactive charts will also be available on our GitHub page upon acceptance.

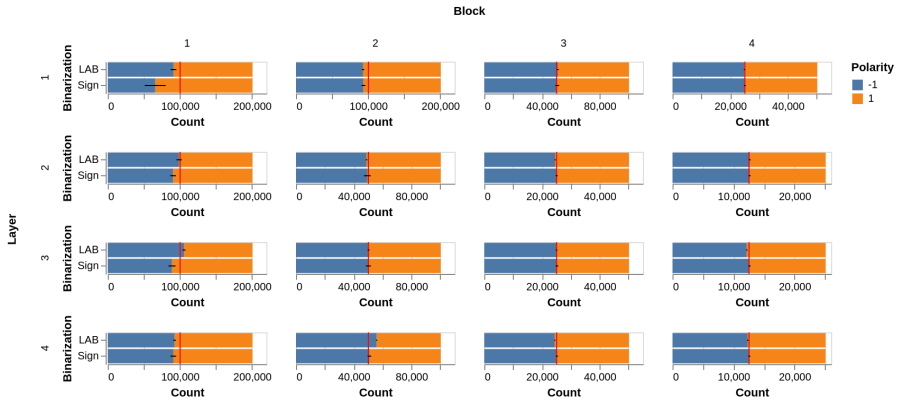


Figure 5: Post-binarization distribution of original Bi-RealNet-18 vs LAB−BNN averaged over all channels of 1000 input images. Rows indicate the blocks in the ResNet structure, columns indicate layer number within each block.

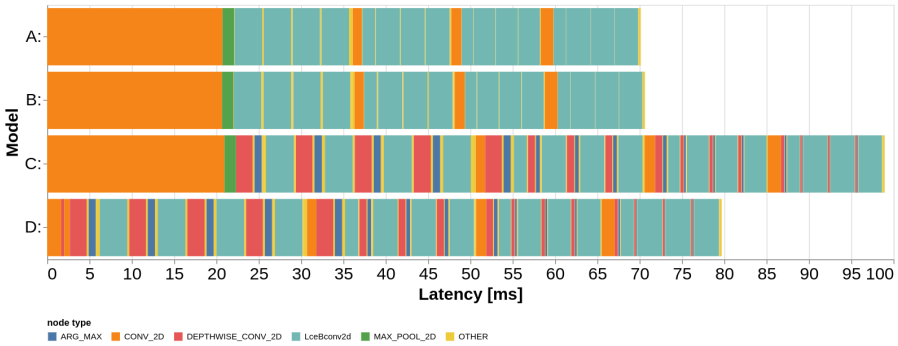


Figure 6: Breakdown of per-operator latencies for ablation study of LAB−BNN. Operators contributing to the latency minimally have been marked as "other".

Table 7: Study on applying LAB to different blocks of LAB−BNN. A checkmark indicates that LAB was used for all layers in that block.

Block 1	Block 2	Block 3	Block 4	Top-1 [%]	Top-5 [%]	Model Size [MB]	Latency [ms]
☑	☑	☑	☑	59.1	81.3	4.68	83.7
☐	☑	☑	☑	58.7	81.0	4.66	68.0
☑	☐	☑	☑	58.4	80.6	4.64	75.3
☑	☑	☑	☐	58.4	80.5	4.33	79.1
☑	☑	☐	☑	58.1	80.4	4.61	77.1
☐	☑	☑	☐	58.0	80.2	4.31	63.9
☐	☑	☐	☑	57.9	80.1	4.58	64.9
☐	☐	☑	☑	57.8	80.0	4.62	60.6
☑	☐	☑	☐	57.8	79.9	4.29	72.9
☐	☐	☑	☐	56.7	78.9	4.27	58.8
☑	☐	☐	☑	56.7	79.3	4.57	71.1
☑	☑	☐	☐	56.4	79.1	4.26	75.0
☐	☐	☐	☑	55.9	78.6	4.54	41.4
☐	☑	☐	☐	55.7	78.1	4.23	47.6
☑	☐	☐	☐	54.9	77.3	4.22	59.9
☐	☐	☐	☐	53.2	76.1	4.20	54.6

7. Concluding Remarks

We have shown that the commonly adopted binarization operation `sign(.)` imposes a *uniqueness bottleneck* on BNNs, making it a sub-optimal choice for binarization. As a remedy, we introduce learnable activation binarizer (LAB), a novel binarization function that allows BNNs to learn a flexible binarization kernel per layer. We have demonstrated that LAB can readily be plugged into existing baseline BNNs boosting their performance regardless of their architecture design. Beyond that, we have also built a new end-to-end network (LAB−BNN) based upon LAB that offers competitive performance on par with the state-of-the-art. For future work, we will investigate applying learnable binarization to weights in addition to activations. We also plan to further extend our experimentation especially around LAB−BNN to push the performance boundaries and advance the state-of-the-art.

References

[1] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D Lane, and Yarin Gal. An empirical study of binary neural networks' optimisation. In *International conference on learning representations*, 2018. 2

[2] Kota Ando, Kodai Ueyoshi, Yuka Oba, Kazutoshi Hirose, Ryota Uematsu, Takumi Kudo, Masayuki Ikebe, Tetsuya Asai, Shinya Takamaeda-Yamazaki, and Masato Motomura. Dither nn: An accurate neural network with dithering for low bit-precision hardware. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 6–13. IEEE, 2018. 3

[3] Tom Bannink, Adam Hillier, Lukas Geiger, Tim de Bruin, Leon Overweel, Jelmer Neeven, and Koen Helwegen. Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks. *Proceedings of Machine Learning and Systems*, 3:680–695, 2021. 2, 5, 6, 7

[4] Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. Back to simplicity: How to train accurate bnns from scratch? *arXiv preprint arXiv:1906.08637*, 2019. 5

[5] Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. Bats: Binary architecture search. In *European Conference on Computer Vision*, pages 309–325. Springer, 2020. 2

[6] Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. High-capacity expert binary networks. In *International Conference on Learning Representations*, 2020. 1

[7] Adrian Bulat, Georgios Tzimiropoulos, Jean Kossaifi, and Maja Pantic. Improved training of binary networks for human pose estimation and image recognition. *arXiv preprint arXiv:1904.05868*, 2019. 1, 2, 5

[8] Tianlong Chen, Zhenyu Zhang, Xu Ouyang, Zechun Liu, Zhiqiang Shen, and Zhangyang Wang. " bnn-bn=?": Training binary neural networks without batch normalization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4619–4629, 2021. 2

[9] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11408–11417, 2019. 2, 3, 4, 6

[10] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019. 1

[11] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2016. 4

[12] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. Riptide: Fast end-to-end binarized neural networks. *Proceedings of Machine Learning and Systems*, 2:379–389, 2020. 2

[13] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021. 1

[14] Kai Han, Yunhe Wang, Yixing Xu, Chunjing Xu, Enhua Wu, and Chang Xu. Training binary neural networks through learning with noisy supervision. In *International Conference on Machine Learning*, pages 4017–4026. PMLR, 2020. 2

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 5, 6

[16] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*, 2016. 2

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 2, 5, 6

[18] Hyungjun Kim, Jihoon Park, Changhun Lee, and Jae-Joon Kim. Improving accuracy of binary neural networks using unbalanced activation distribution. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7862–7871, 2021. 1, 3, 4, 6, 7

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012. 5, 6

[20] Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Fei Chao, Mingliang Xu, Chia-Wen Lin, and Ling Shao. Siman: Sign-to-magnitude network binarization. *arXiv preprint arXiv:2102.07981*, 2021. 2, 7

[21] Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Yan Wang, Yongjian Wu, Feiyue Huang, and Chia-Wen Lin. Rotated binary neural network. *Advances in neural information processing systems*, 33:7474–7485, 2020. 2

[22] TensorFlow Lite. Deploy machine learning models on mobile and iot devices, 2019. 5

[23] Zechun Liu, Zhiqiang Shen, Shichao Li, Koen Helwegen, Dong Huang, and Kwang-Ting Cheng. How do adam and training strategies help bnns optimization. In *International Conference on Machine Learning*, pages 6936–6946. PMLR, 2021. 2

[24] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. In *European Conference on Computer Vision*, pages 143–159. Springer, 2020. 2, 3, 4, 5, 6, 7

[25] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. 1

[26] 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 (ECCV)*, pages 722–737, 2018. 1, 2, 3, 5, 6, 7

[27] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. In *International Conference on Learning Representations*, 2019. 2, 5

[28] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2250–2259, 2020. 1, 2, 3, 7

[29] Haotong Qin, Xiangguo Zhang, Ruihao Gong, Yifu Ding, Yi Xu, et al. Distribution-sensitive information retention for accurate binary neural network. *arXiv preprint arXiv:2109.12338*, 2021. 5

[30] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016. 1, 2, 5, 6

[31] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Thirty-First AAAI conference on artificial intelligence*, 2017. 2, 5, 7

[32] Peisong Wang, Xiangyu He, Gang Li, Tianli Zhao, and Jian Cheng. Sparsity-inducing binarized neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12192–12199, 2020. 1, 3, 7

[33] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. 5

[34] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. Deep neural network compression with single and multiple level quantization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018. 1

[35] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. Bmxnet: An open-source binary neural network implementation based on mxnet. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1209–1212, 2017. 2

[36] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7370–7379, 2017. 1

[37] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, and Tao Mei. dabnn: A super fast inference framework for binary neural networks on arm devices. In *Proceedings of the 27th ACM international conference on multimedia*, pages 2272–2275, 2019. 2

[38] Yichi Zhang, Zhiru Zhang, and Lukasz Lew. Pokebnn: A binary pursuit of lightweight accuracy. *arXiv preprint arXiv:2112.00133*, 2021. 5

[39] Baozhou Zhu, Zaid Al-Ars, and H Peter Hofstee. Nasb: Neural architecture search for binary convolutional neural networks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020. 2

3

Background information

In order to understand the relevant research in BNNs, it is essential to first understand the basics of deep learning and to introduce terms that will be used later, and then to gradually narrow our scope down to BNNs. Therefore, this chapter starts by introducing deep learning (section 3.1) and by explaining the essentials of Multi Layer Perceptrons and Convolutional Neural Networks. Then, existing techniques to make neural networks more energy-efficient will be discussed in section 3.2.

3.1. Deep Learning

As illustrated in Fig. 3.1, Deep Learning is a subset of Machine Learning (ML) which is in its turn a subset of AI again. To start off wide, an AI program is a piece of software which can reason, react and adapt to its inputs. ML is more specific and specifies itself by the fact that it is about algorithms which take in data and learn to make informed decisions. Deep learning is even more specific by accomplishing this task by using multilayered neural networks. The task of deep learning is to learn to extract features from data which can be used in various tasks such as, but not limited to: computer vision [24], natural language processing [20], cybersecurity [33] and medical image analysis [28]. This work limits itself to computer vision related tasks and in special image classification. This task presents an image to the network, and this network should learn to classify the main object in the image into a one of the possible output classes. In the next section (section 3.1.1), the Multi Layer Perceptron (MLP) will be introduced which helps basic understanding of neural networks, whereafter in section 3.1.2, the basics of a Convolutional Neural Network is introduced.

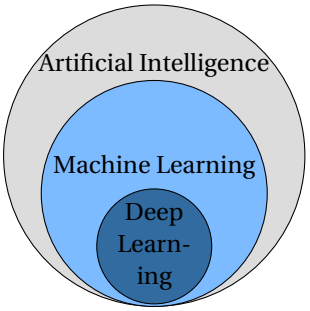


Figure 3.1: Artificial Intelligence subsets.

3.1.1. Multi Layer Perceptron

The simplest neural network is a Multi Layer Perceptron (MLP), which as the name suggests, consists of multiple layers, as can be seen in Fig. 3.3. A layer of a MLP consist of multiple neurons (the nodes in the figure). The operations happening inside a neuron can be found in Fig. 3.2. The term "neuron" comes from the fact that its working is inspired by the neurons in the human brain, which can "fire" when presented with relevant information. All of the edges in a MLP have a weight and the output of a neuron is given by $\sigma(\sum_i x_i \cdot w_i)$, which is a Multiply-Accumulate (MAC) operation between the inputs x_i and weights w_i, followed by an activation function $\sigma(x)$. The activation function is a function that brings non-linearity into the network. More about activation functions and a summary of common activation functions will be discussed in section 3.1.2. A MLP is part of a family of networks called feed-forward networks, due to the fact that every neuron feeds it calculated output value forward to the next layer, until the final output value is computed. By just doing a forward calculation, the network calculates an output, but it does not learn anything yet. A MLP learns by updating the values of it parameters, which are for example its weights and/or biases, by using backpropagation. Backpropagation starts by calculating the loss between the calculated output value and the ground truth value. This loss function is often specific to the tasks that needs to be solved. Then, this loss will be propagated back

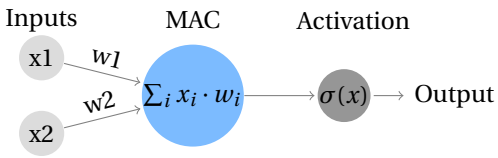


Figure 3.2: A neuron with two inputs.

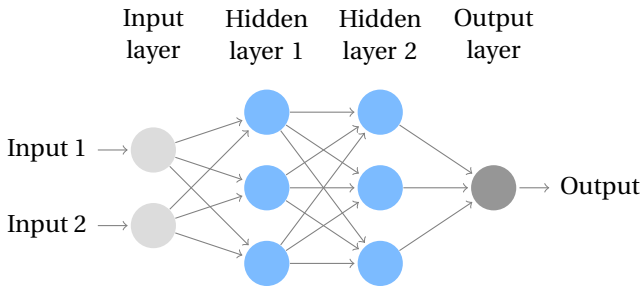


Figure 3.3: A Multi-Layer Perceptron (MLP) with two hidden layers.

trough the network by calculating the gradient of the parameters with respect to the loss. Finally, the weights are updated by the optimizer, and the network will continue with the next forward step until the training of a network has converged, or reached its number of iterations. The training of a neural network often consists of 3 phases, which all have their own dataset associated with it. It starts with the training phase: here, the network is fed with annotated training samples, which contain the actual data, accompanied by the ground truth value. Then, in the validation phase, the network can be tested for data generalization. As the network should learn to generalize well to unseen data samples, the network is again fed with data samples and their ground truth values, but none of the validation samples should be in the training samples. Last is the test phase, in which the network is evaluated after the model has finished training (and validation). This is often to test against competing models.

3.1.2. Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) is very similar to the MLP that we have seen in the previous section. Both are neural networks with learnable parameters. However, a CNN has the difference that it focuses on spatial data and does the computations more efficient than a MLP. The basic form of a CNN with commonly used components can be seen in Fig. 3.4. A first thing that can be seen from this figure is the fact that a CNN is structured in two main components: A part that extract features from the input image and a part that does the classification. First, the feature extractor typically consists of convolutional layers with activation functions, pooling layers and other layers in between. These components will be discussed in the following sections together with commonly used architectures, so that at the end of this section there will be a good understanding of used terms. Second, the classification part is responsible for mapping the output from the feature extractor to the classes that should be learned. The alert reader will have noticed that the fully connected layer in the classification part looks similar to the hidden layers of the MLP discussed earlier.

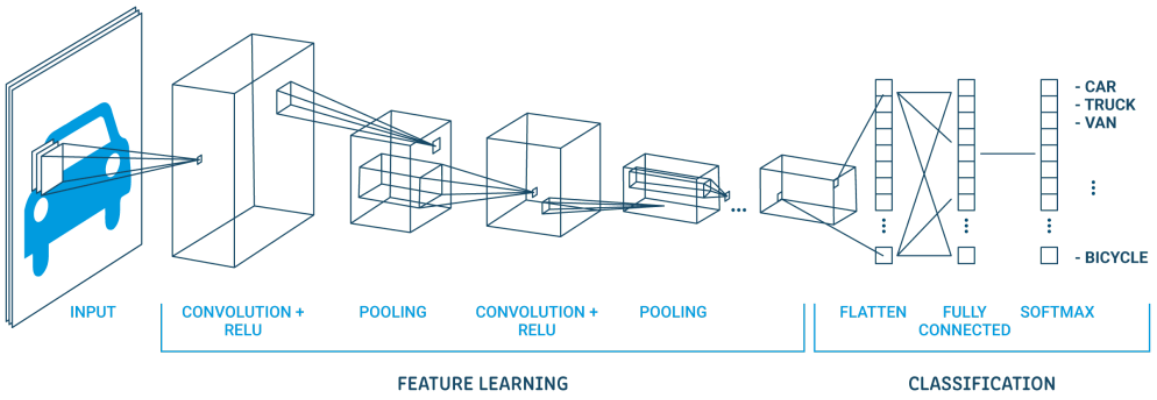


Figure 3.4: A visualization of a CNN architecture with commonly used layers.

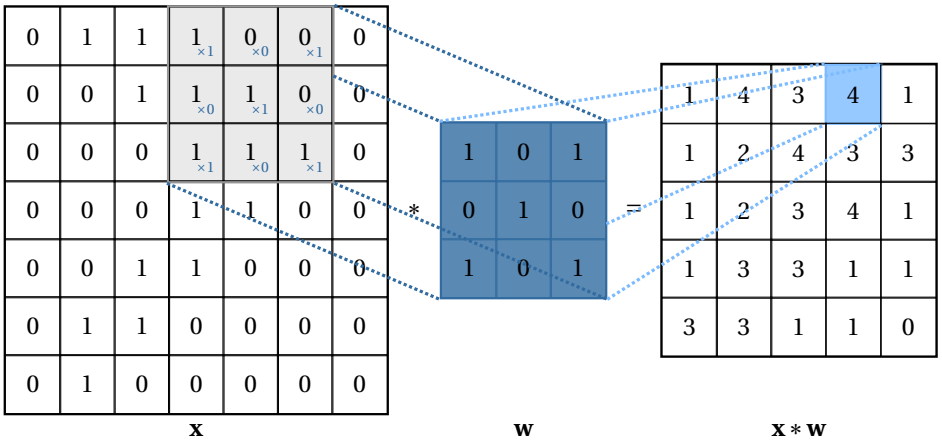


Figure 3.5: Calculation of a 2D convolution.

Convolution layers

The convolutional layers are, as the name suggests, the most important building blocks of a CNN and are, also not surprisingly, based on the convolution operator. The 2D convolution operation for a single image works by sliding a kernel (or else-called a filter or weights) over the layers' input. This kernel has size $k \times k$, where k is called the kernel size which is often chosen to be 3. In each step of the sliding window operation, a dot product between the kernel and the input (often called featuremaps) is calculated, which will be the output pixel at that specific location. The equation for a single input channel, single output channel convolution is given in 3.1,

$$y_{m,n} = (x * w)_{i,j} = \sum_{i=0}^{k} \sum_{j=0}^{k} w_{i,j} \cdot x_{m-i,n-j}, \tag{3.1}$$

and an example of the steps of a 2D convolution can be seen in Fig. 3.5 [1]. The values used for both weights and activations in the figure are just 0 and 1 for easy visualization, but can have any value. One can notice from Fig. 3.5 that the size of the output image shrinks after the convolution operation. This can be counteracted, by padding the input image before convolution. For featuremaps with more channels, the kernel will have the same amount of channels and the convolution outputs from each channel get summed to yield the output value (as can be seen in the convolution+ReLU step in Fig. 3.4). The number of output channels is determined by the number of filters used where the output of the convolution will have as many channels as there are filters.

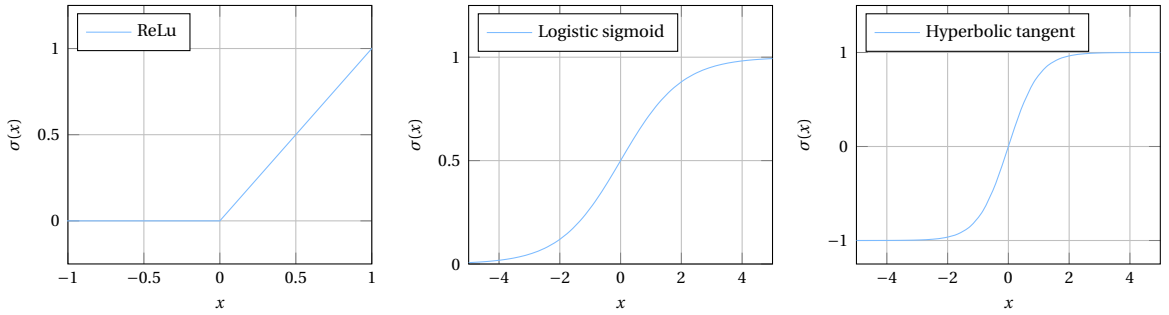
Activation Functions

Simply combining multiple convolutional layers will yield sub-optimal results, as the network will only be able to learn a linear model. To insert non-linearity into the network, the output of the convolution is passed through an activation function. Some examples of commonly used activation function are illustrated in Fig. 3.6. The first is the `ReLu` function, which is a piece-wise linear function. Alternatives to the `ReLu` are the `Leaky-ReLu`, which does not zero out all negative inputs, but gives the negative part a slight slope. When this slope is made learnable, the function is called `PReLu`. Other activations such as the sigmoid and the hyperbolic tangent are also shown in the figure, where the sigmoid can be used for mapping a continuous range into a probabilistic range from 0 to 1. The hyperbolic tangent is of interest as it is used as an approximation function for the `sign(.)` (more about this in chapter 4).

Pooling layers

The pooling layer (as can be seen in Fig. 3.4) is a layer that can be used to reduce the spatial dimension of the featuremap. This ensures that computations will get lighter, but more importantly, it ensures that the network will become more translation invariant, due to features being summarized along their spatial dimension. Two popular implementations of pooling are called max-pooling (taking the maximum pixel location of a region) or average-pooling (taking the average of the region), where pooling is applied to a 2×2 region.

[1] Figure adapted from https://tex.stackexchange.com/questions/522118/visualizing-matrix-convolution, accessed March 22, 2022



(a) ReLU activation function. (b) Sigmoid activation function. (c) Hyperbolic tangent activation function.

Figure 3.6: Common activation functions.

Common CNN architectures

Many BNNs architectures are based on popular network CNN architectures. Therefore for completeness, we briefly introduce those CNN architectures here. **AlexNet [24]** This eight layer network was one of the front runners of CNN architectures on the ImageNet dataset. It was the first network to utilize the `ReLu` activation function and to use multi-GPU training. **ResNet [18]** ResNet has become famous for its use of skip connections (else-called residual connections), which were introduced to counteract the vanishing/exploding gradient problem that was occuring in deeper networks. ResNet is available in different depth configurations, where for this work, we mostly use the ResNet-18 structure. **MobileNet [23]** MobileNet has been introduced with model efficiency in mind. It makes use of depthwise separable convolutions, which splits the normal convolution into a depthwise convolution followed by a pointwise convolution. This significantly reduces the number of parameters required in the network.

3.2. Network compression

As one can imagine, the convolution operation in equation 3.1 will be a very heavy computation when this convolution will have multiple input and output channels. When the network architecture is rather deep (with many layers), the network will have a large latency. This is both a bottleneck for network training time, as well as inference delay. To alleviate this, multiple network compression techniques have been introduced in recent years, which will be discussed below:

3.2.1. Quantization

The first network compression technique is quantization. Quantization works by reducing the bit depth of the parameters in the network. In standard CNNs, all parameters are represented by 32-bit floating point numbers. With millions of parameters, this takes up the main part of the memory usage of the network. Therefore, if there would be a way to express the network with a lower bit depth, significant compression could be achieved. Evidently, by reducing the bit depth of the parameters, the representational capacity of the network drops, and with this, the accuracy of the network reduces. The active research field therefore focuses on reducing bit width, while also retaining the model accuracy. So far, research has been able to reduce the bit width to 4 bits, while keeping or surpassing the original accuracy [9, 45]. Quantization is applied in two ways: the first applies quantization after training (post-training quantization), where the second applies it during training which is called Quantization Aware Training (QAT). Most deep learning frameworks, such as Pytorch, only support 16-bit floating point (FP16) quantization and INT8 quantization. FP16 (16-bit) quantization can often be applied lossless, so without any drop in accuracy. INT8 (8-bit) quantization however, often needs quantization aware training, or retraining on a calibration set, to keep the initial accuracy of the model.

An extreme form of quantization is quantizing the parameters to only 1 bit. These networks, called Binary Neural Networks (BNN) are the main topic of interest of this work and will be explained in more detail in the next chapter. In short, by binarizing weights and activations in a network, one can leverage bitwise operations, with which a memory reduction of 32x and a computation speedup of 58x on CPU can be achieved [38].

3.2.2. Pruning

The main assumption for pruning is the fact that CNNs are heavily overparameterized and contain many redundant parameters [39]. Pruning works by finding the unnecessary parameters and removing them from the network. Interesting here is how to define which parts of the network are unnecessary. An overview of commonly used pruning algorithms can be found in the survey by Liu et al. [29]. Which percentage of the network can be pruned away is highly dependent on the network architecture, but often around a third of the network parameters can be pruned without a drop in accuracy [25].

3.2.3. Knowledge distillation

Knowledge distillation is based around the same assumption as pruning: a trained model consists of many redundant parameters. For knowledge distillation, a large model or an ensemble of methods, called "the teacher" helps a smaller model, called "the student" to learn by transferring the knowledge of the high capacity network to a network with lower capacity [21]. The student model is trained to mimic the outputs of the teacher model. An overview of common algorithms can be found in the survey by Gou et al. [17].

3.2.4. Neural architecture search

Neural architecture search is a compression technique that aims to automatically find a good network architecture by applying a specific search strategy in a predefined search space. The search space is defined by the operations that are allowed to form a valid network. Defining the search space is a consideration between adding prior knowledge to shrink the search space and giving the algorithm the freedom to explore novel architectures. Then to select an architecture, a search algorithm needs to be constructed. To determine whether an architecture could be a proper candidate, it's performance needs to be estimated. The most straight forward way would be to train and validate the model like any other model, however, this is very resource intense. That is why there are works that focus on finding other performance evaluation criteria. More information about neural architecture search can be found in the survey by Elsken et al. [15].

3.2.5. Low-rank factorization

Other than being overparameterized, CNNs suffer from redundancy between parameters of different layers. Low-rank factorization tries to decompose the tensors used in the network into a product of tensors of lower rank [12], so that it reduces the memory footprint and therefore yields a computational speedup in the network. Low-rank factorization can improve improve the speed-up to 30–50% compared to the full-rank matrix representation [10].

4

Binary Neural Networks

4.1. Introduction in BNNs

Binary Neural Networks are networks that are quantized to their most extreme form, where the weights and activations are quantized to only one bit. If the binarization is only applied to the weights, it is referred to as "binary-weight networks". Quantization to only one bit means that there are only two possible values to take, and in most works these values are taken as $\{-1, +1\}$. The first work that published about binary-weight networks is by Courbariaux et al. in 2015 [11] where it was shown that by binarizing the weights, the MAC operations could be replaced with simple accumulate operations, yielding a speedup of around 3 times. One year later, Rastegari et al. showed that by also binarizing the activations it was possible to replace the MAC operations by XNOR and popcount operations and achieve memory savings of 32× and a theoretical speedup of ~ 58× on CPU [38]. The calculation of such a binary convolution is illustrated in Fig. 4.1 and a practical example of the XNOR and popcount operations of this convolution are shown in Fig. 4.2. Fig. 4.1 is similar to the earlier explained notion of 2D convolution in Fig. 3.5, with the only difference being that the weights and activations are binary and so the representative range of the output y are all odd integers $y \in [-9, +9]$ instead of $y \in [-\infty, +\infty]$ for the FP convolution. Note that for this specific example only odd integers can be yielded because of an odd channel number of the featuremap. If this channel number is even, the outputs also will be even. The fact that the output only has limited representative power is one of the reasons why early BNNs had a great accuracy gap with respect to their FP counterpart.

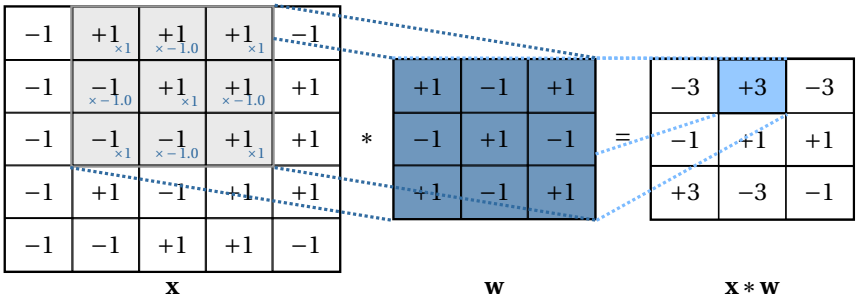


Figure 4.1: Calculation of a binary convolution.

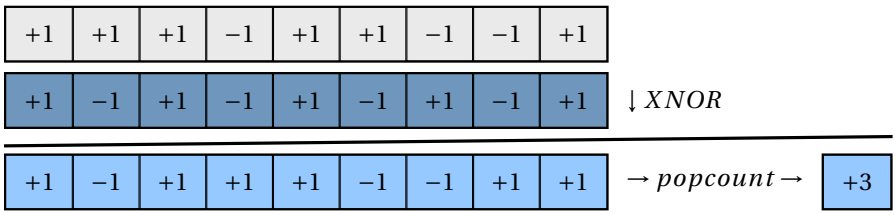


Figure 4.2: XNOR and popcount operations.

4.2. Training a BNN

So far, it has been mentioned that a BNN uses a binarization function to get binary values for weights and activations. In the forward pass of a BNN, the binarization function that is commonly used is the `sign(.)`, given by equation 4.1. Here, x_b refers to the binary value and x_r to the real or FP value.

$$x_b = \texttt{sign}(x_r) = \begin{cases} +1, \text{ if } x_r > 0 \\ -1, \text{ if } x_r \leq 0 \end{cases} \tag{4.1}$$

However, this `sign(.)` has a gradient which is equal to the Dirac delta function, which is zero except at the origin. Using this gradient would trivially break the backpropagation of a BNN. Therefore, in early works, the Straight Through Estimator(STE) [3] was used as an estimation of the gradient of the `sign(.)` [11]. The STE forms the identity function in the backward pass, e.g. the gradient from later layers is just passed trough to earlier layers. Later in section 4.3.2 different kinds of gradient estimation functions will be discussed. The backpropagation step of `sign(.)` updates the FP values of the weights and activations instead of the binary parameters. In this way, it is possible to aggregate parameter updates from multiple iterations. There could be a possibility that parameter upgrades keep aggregating without causing a flip in sign, causing parameters to explode in value. To counteract this, Binaryconnect clips the value of the FP parameter to an absolute value of 1 [11].

Layer order. Constructing a BNN is a little different with respect to the full-precision model architectures. Rastegari et al. stated that where typical CNNs use a layer structure of `Binary convolution` → `Batch normalization` → `Activation` → `Pooling`, this does not make sense to follow for BNNs [38]. To clarify, here the Batch normalization layer normalizes the input batch by its mean and variance and the activation layer includes the binarization operation. As (max)pooling applied to binary values will yield an almost uniform feature map, the pooling is shifted to directly after the convolution layer. Therefore, the typical layer structure of BNNs is in the form `Batch normalization` → `Activation` → `Binary convolution` → `Pooling`. Apart from the order of the layers, the first and last layer of a BNN are commonly kept as a full-precision convolution [1, 6, 26, 31, 38]. XNOR-Net found that binarizing the first layer was not giving considerable speedup, although it did hurt the performance of the model [38]. Tang et al. argue that binarization of the last layer fails due to the extremely large variational range after the binary convolution [41]. This is because the amount of channels in the last layer is high and the channels influence the output range of the binary convolution. When this large variational range is fed into the softmax, the authors argue that the softmax is getting in its saturation region.

4.3. Fields of research

As mentioned earlier, the first naive BNN implementations in 2015 suffered from a great accuracy drop with respect to their full-precision counterparts. Since then, several subfields of research have evolved to counteract this drop. These subfields can be categorized in 6 groups which will be discussed below.

4.3.1. Quantization error minimization

This category of research was probably one of the first to gain improvement in BNNs. The idea is based around the fact that by simply binarizing the FP values, there is a great quantization error with respect to those values. Therefore, XNOR-Net introduced a scaling factor, so that binary weights and activations are rescaled and quantization error is minimized [38]. The scaling factors can either be calculated from the absolute mean of the FP values [38], they can be learned from the data like XNOR-Net++ did [5], or computed in a data-driven manner like Real2Bin [34]. Another method to minimize quantization error is to use multiple binary bases that are used together to approximate the FP values. ABC-Net [27] and PA-net [47] use a linear combination of $3 \sim 5$ binary bases and Tang et al. use a linear combination, combined with a scaling factor learned from residue approximation error from earlier layers [41].

4.3.2. Gradient Error Reduction

As already mentioned in section 4.2, the gradient of the `sign(.)` needs an approximation to be able to train. Next to the earlier mentioned STE, different kind of approximations were designed. Bi-Real Net introduces ApproxSign, which is a second order polynomial in the forward pass and so a piecewise linear function in the backward pass [31]. IR-Net uses the gradient of a parametrized `tanh` (as described in section 3.1.2) function for the backward approximation which they call the Error Decay Estimator [37]. During training the shape of the function gets sharper, so the function gets progressively closer to the `sign(.)` function. Lin et al. propose a viewpoint similar to IR-Net in their work of RBNN, by utilizing a training-aware approximation function for replacing the `sign(.)`[26]. The function used is different, but the idea is similar in the fact that the function progressively approximates the `sign(.)`. Last, FDA-BNN presents a frequency domain approximation of the `sign(.)`, where the low-frequency components are the main contributors to the direction of the gradient [42].

4.3.3. Network Architecture

The network architectures that have been used for BNNs have greatly been inspired by full-precision architectures modified for use with binary parameters. XNOR-Net [38] is based on the Alex-Net structure [24]. Bi-Real Net has taken inspiration from ResNet [18] by introducing identity shortcuts to the binary network [31]. This is possible at low cost, due to the fact that after the binary convolution, the bitcount operation generates discrete-alphabet values anyway, so a simple `add` operation in that step will have low extra latency. The identity shortcuts do contribute a lot in propagating full-precision information to later layers of the network. ReActNet has taken the architecture of MobileNet V1 [23] as the basis for their binary network, as they believe that it make sense to start from a compact network structure [32]. Following Bi-Real Net [31], they adopted the identity shortcuts in their model as well. Other than converting well-known architectures to their binary counterpart, Group-Net describes how to learn the decomposition of full-precision layers into multiple binary groups [49] and BENN leverage ensembles for BNNs [48].

Apart from finding suitable architectures for BNNs in a classical way, several works have applied Neural Architecture Search (NAS) for binary networks, such as BATS [7] and NASB [46]. Lastly, Diffenderfer et al. came up with the Multi-Prize Lottery Ticket hypothesis, in which the task is to find sparse BNN subnetworks in an overparametrized network by applying pruning an quantizing cooperatively [13].

A little different from above mentioned works, which mainly focuses on the complete network design, several works have investigated to add or remove parts of the network: BNN-BN investigates the influence of removing the batch normalization layer for BNNs [8]. Initially, no activation function was used in BNNs, as the belief was that the binarization function also acts as a activation function. However, the use of the extra `PReLu` activation function on the discrete-alphabet outputs was found to be beneficial [41].

4.3.4. Loss Function Design

In order to accelerate the training convergence and to improve the accuracy of the network, several works focus on adding an extra loss term to the already existing loss function. Hou et al. use loss-aware binarization, by incorporating the effect of binarization in the loss [22]. The optimization is solved using the proximal Newton method. IR-Net use the so called Libra Parameter Binarization to not only minimize the quantization error but simultaneously minimize the information loss by reshaping the activation distribution before binarization [37]. Ding et al. also target activation reshaping and introduce a distribution loss consisting of 3 separate terms [14]. ReActNet adds a distribution loss term, so that the binary network is stimulated to learn a similar distribution as it's full-precision counterpart by transfer learning [32].

4.3.5. Training strategies

The last anchor having influence on the accuracy of BNNs is the training strategy and optimizer choice. Real-to-Bin [34] introduced a two-step training approach in which the network gets binarized progressively: in the first step, a student network with binary activations and FP weights is trained, having a full-precision network as a teacher with a similar architecture. Then in the second step, the student network is fully binary and the network trained in the first step will be used as the teacher network. However, multi-stage training is questioned by Bethge et al., which argue that BNNs can achieve state of the art results by training the network from scratch [4].

4.3.6. Binary inference engines

Of course, the research in BNNs is promising, with high theoretical speedup, but it would not make sense if results were only on paper. Common deep learning platforms like Pytorch [1] and Tensorflow [2] do not contain any of the low level operations to make use of the speedup that can be provided by BNNs. That is why several inference engines emerged: programs that take a trained model in one of the aforementioned platform formats and convert it to an optimized model that makes use of XNOR and bitcount operations.

BMXNet is the first BNN library which is based on MXNet and contains the implementation of networks with XNOR operations [43]. daBNN [44] is an inference framework focusing on implementations on ARM devices. It reports that its inference of Bi-Real Net is about 6× faster than its implementation in BMXNet. daBNN supports Open Neural Network Exchange (ONNX), making it compatible with Pytorch and Tensorflow. Riptide [16] is a BNN framework that is based on Tensorflow and TVM which is a deep learning compiler framework for generating machine code for various hardware backends. Riptide extends this by supporting binary operations. Larq Compute Engine is an inference engine based on Tensorflow Lite and supports various binarization operations and state-of-the-art BNN architectures [2].

[1] www.pytorch.org, accessed March 22, 2022
[2] www.tensorflow.org, accessed March 22, 2022

5

LAB: additional materials

As the paper presented in chapter 2 is extensive in terms of experiments and explanation, this chapter will focus on providing more visual comparison between the traditional used `sign(.)` and our newly proposed `LAB`. The training curves with Top-1 and Top-5 training and validation accuracy of `LAB-BNN` on ImageNet are shown in Fig. 5.1. An insight into multiple channels of the first layer of `sign(.)` vs. `LAB` is shown in Fig. 5.2(a)-5.2(e) and an extension of Fig. 4 from the paper for more layers and more images is shown in Fig. 5.3. A scatterplot of the accuracy vs. latency trade off for Table 7 in the paper is found in Fig. 5.4. Furthermore, the code for the `LAB` function has been added in appendix A.

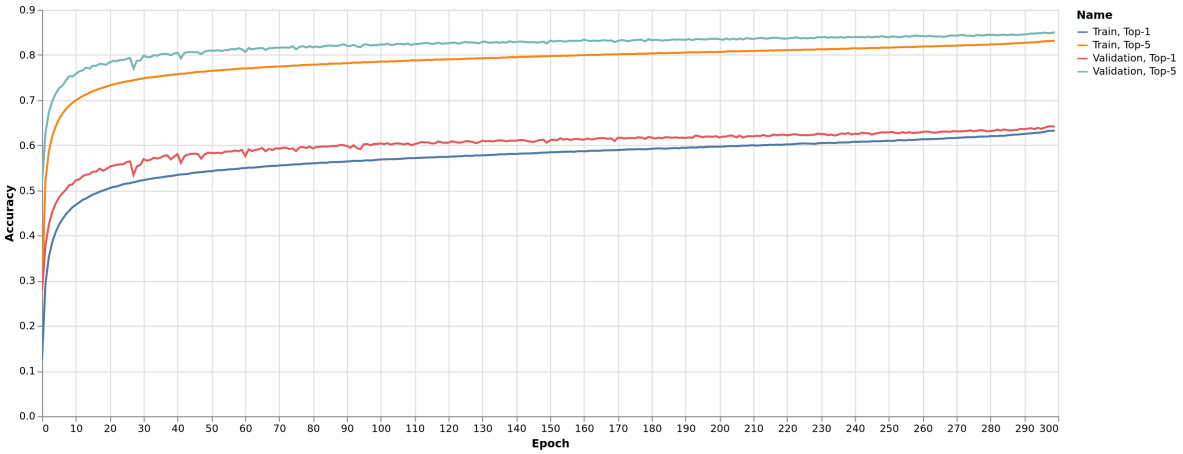


Figure 5.1: Top-1 and Top-5 validation accuracy for training and validation of `LAB-BNN` on ImageNet.

(a) Input image from ImageNet.

(b) Input to `sign(.)`.

(c) Output of `sign(.)`.

(d) Input to `LAB`.

(e) Output of `LAB`.

Figure 5.2: Comparison of in- and output activations on 9\64 channels of the first layer of `sign(.)` vs. `LAB` on a sample of ImageNet.

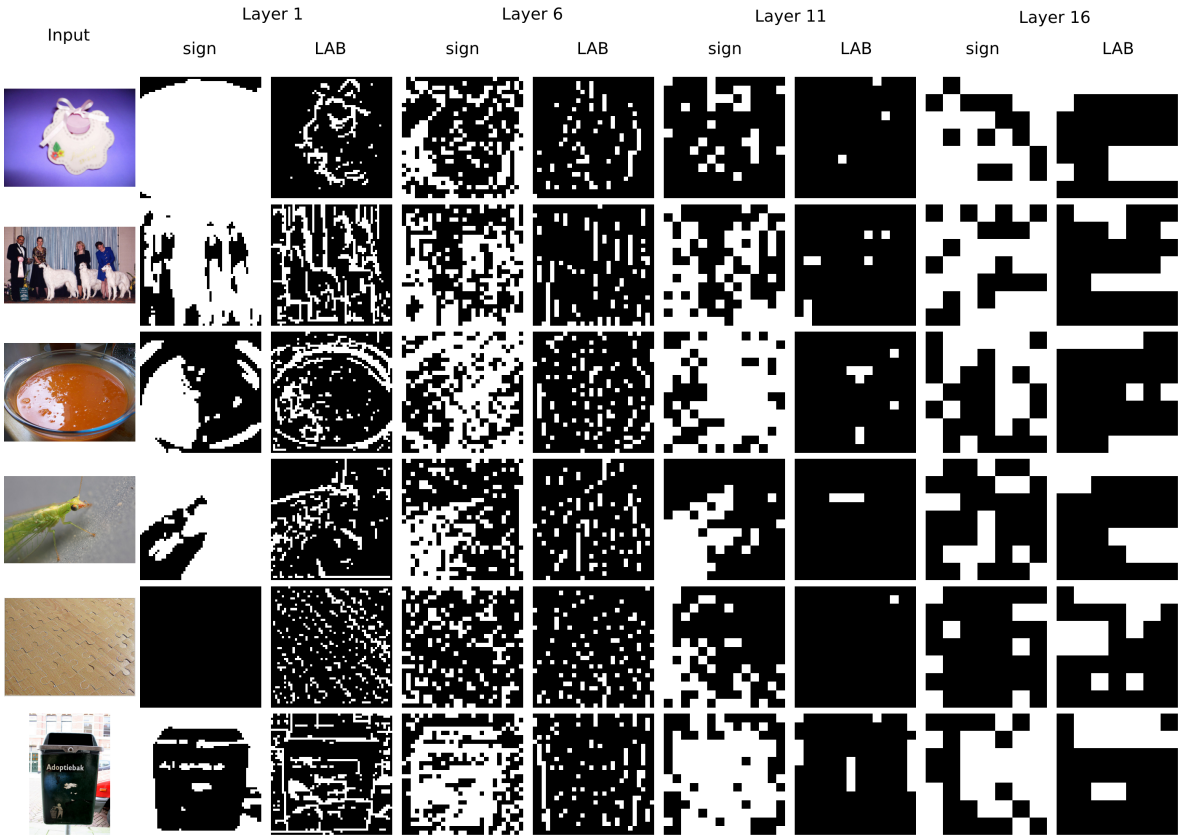


Figure 5.3: Comparison of input and output of `sign(.)` vs. `LAB`on 4 different layers on samples of ImageNet.

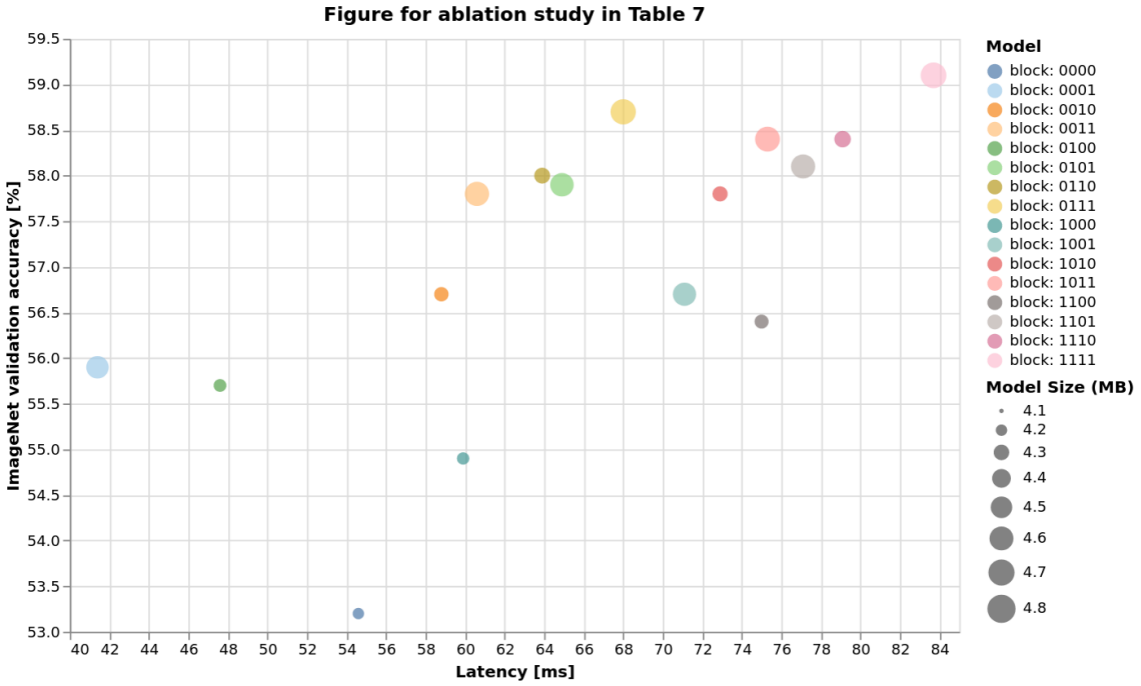


Figure 5.4: Visualization of Table. 7 in the paper: accuracy vs. latency trade off.

6

Applying AI at the edge

6.1. Motivation

Nowadays within a technical company a numerous amount of tasks incorporate the techniques of deep learning, computer vision, natural language processing and more. Similar as what has been argued before, deploying these models to resource-constrained edge devices is a challenging task. For this reason, there is a desire to develop models that can be applied onto battery operated devices like drones or devices with poor network connectivity operating at remote areas as is often the case. To make this goal concrete, it was desired to run both a object detection and instance segmentation model in real-time on resource constrained hardware. The following chapter is a report of this challenge. Although it is not the main topic of this thesis, it is a relatable project for practically bringing deep learning to edge devices. This chapter will start off by explaining the choice of hardware, whereafter the two paths explored for the software pipeline are discussed. After the experimental setup, quantitative and qualitative results of the project will be shown.

6.2. Hardware components

A logical first step is to determine which hardware to use for the project. The group of consumer edge devices can be split in two main categories: dedicated inference hardware, specially designed for deep learning inference, and mobile devices, such as phones and tablets. Good to address is that both of these categories are only useful for applying inference; model training still needs to be done on a HPC. In this section, the different types of available hardware will be discussed, whereafter the choice of hardware for the project will be justified. Note that the focus for hardware is only on consumer-ready hardware. We are aware of the possibilities for Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) implementations, but those do not fit the scope of the project.

6.2.1. Dedicated inference hardware

For dedicated inference hardware available for consumers, 3 manufacturers dominate the market: Nvidia has a range of devices called the Jetson family [1], which are System-On-Modules with a dedicated GPU. Some of the devices of the series offer hardware INT8 support and make use of Nvidia's Tensorcores with Deep Learning Accelerator (DLA) [2]. The hardware comes with the option to make use of Nvidia's deep learning software stack.

Intel has presented the Intel Neural Compute Stick [3]. This is a usb device that can be plugged into host and edge devices and deliver neural network acceleration. Although this is no standalone solution, it is a good way to transform small form factor devices (like Raspberry Pi) into a deep learning inference platform.

Last, Google has produced a family of devices for deep learning inference which is branded as Coral [4]. The family has a range of devices such as an USB accelerator, System-On-Module, M.2 accelerators and development kits. The devices are centered around the Coral Tensor Processing Unit (TPU). Similar to Jetson, Coral provides a software library with numerous deep learning models.

[1] https://developer.nvidia.com/buy-jetson, accessed March 22, 2022
[2] https://nvidia.github.io/Torch-TensorRT/tutorials/using_dla.html, accessed March 22, 2022
[3] https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html, accessed March 22, 2022
[4] https://coral.ai/, accessed March 22, 2022

6.2.2. Mobile devices

Also on our mobile devices like phones and tablets, the research has not been standing still. Recent devices are often equipped with GPU's and special devices for acceleration of DNNs. Often, these devices are called Neural Processing Units (NPU) [5]. For developing models that run on mobile devices, most of the major existing platforms have solutions available. (Pytorch Mobile, TensorFlow Lite, Keras, ONNX)

6.2.3. Chosen Setup

As the chosen setup, it was decided to use two devices for the project: The Nvidia Jetson Xavier NX (shown in Fig. 6.1) and the Nvidia Jetson Xavier AGX. Both are quite similar in architecture, but the AGX is a bit more powerful. There are a couple of reasons why the Jetson family has been chosen: First, there was a demand for a stand-alone device, with options for hardware interfacing to other systems, so that in the future, the device can be used for robotics projects. For testing, a developer kit was desired. This limited the choice between the Jetson family developer kits and the Coral Devboard. Both are quite similar in terms of ports, as both are equipped with HDMI, USB, MIPI-CSI and a series of GPIOS's. The second and main reason to go for the Jetson family is that the Jetson family is more powerful, being a great first step into deploying AI at the edge.

Figure 6.1: Nvidia Jetson Xavier NX with MIPI-CSI camera.

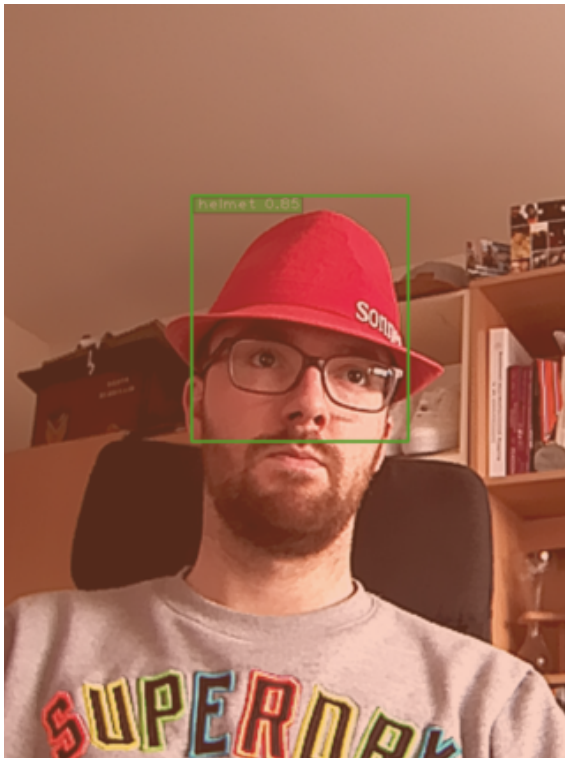


Figure 6.2: Real time hardhat detection on the author.

6.3. Software pipeline

To employ a deep learning model on an edge-device, often a standard sequence of steps need to be followed: First, a pretrained model is taken and retrained on task-specific data. Then, the model will be compressed, which might need additional retraining, whereafter it can be converted for the inference hardware. To achieve the aforementioned pipeline, two different paths have been explored, one being Nvidia TLT and the other being using OpenMMLab. Both will be discussed below, together with their (dis)advantages.

[5] https://developer.arm.com/documentation/102420/0200/Neural-processing-unit-introduction/Description-of-the-neural-processing-unit, accessed March 22, 2022

6.3.1. Nvidia TLT

Note: At the time of writing Transfer Learning Toolkit (TLT) is being rebranded to Train, Adapt, Optimize (TAO), but it is still referenced here as TLT

Retraining. TLT is Nvidia's platform for (re)training deep learning models and deploying them on all devices with an Nvidia GPU. It is capable of doing most of the steps in the pipeline. The pipeline is illustrated in Fig. 6.3 and is used as a guide for the steps in this subsection. Nvidia has a repository where checkpoints of pretrained models are housed, called NGC. Models can be divided into two types of tasks: general models for classification, detection and segmentation, and purpose built models for specific use cases like PeopleNet, TrafficCamNet, etc. In the project, only the general models have been considered. Pre-trained models are added to the repository constantly, but currently the support for Instance Segmentation and Semantic segmentation is limited, which leads to one of the main reasons to explore other options than TLT as well. However, most of the object detection models are developed well. To train a model with TLT, one needs to supply all parameters in configuration files (spec files), which are json-like files.

Model compression. For model compression, TLT supports two types of compression: pruning and quantization. Pruning is applied after model retraining and can be done with a certain threshold controlling pruning granularity. The model size decreases after the pruning step and so does the accuracy. Therefore, it is advised to apply retraining after the pruning step to recover lost accuracy. In practice, it was found that the pruning threshold is a delicate parameter and should be tuned. Not all models support pruning at the time of writing, but support for new models might be added in new updates. For the other compression technique, quantization, two flavours are available: FP16 and INT8 quantization. FP16 can be applied post-training, as this can be applied nearly lossless. INT8 quantization however requires the model to be retrained to compensate for the accuracy loss. TLT supports Quantization Aware Training (QAT) as an extra training step, but folds the actual quantization into the export step. The benefit of using pruning and quantization in TLT is the fact that it is rather straightforward to get it working. The downside of this abstraction however is that for example it is not known which pruning algorithm is used underneath the hood.

Exporting. In the exporting step of TLT, the trained model will be prepared for inference on the target device. TLT has two ways of doing this: The first is to run the export step onto the target and create a TensorRT (TRT) engine, optimized for the target. More common however, is to export the model into an intermediate format, called `encrypted tlt` (.etlt), which can then be moved to the target device. Later, on the target device the tool `tlt-converter` can be used to convert the .etlt file into a TRT engine (.engine). As an option during this export step, the model can be quantized to an INT8 model as well. For post-training quantization, this step needs a folder of calibration images. If the model was trained using QAT, a calibration cache can be supplied during exporting, which holds information about the quantization operation. Exporting in TLT uses TensorRT in the backend. TensorRT is Nvidia's workhorse for GPU optimization and is a versatile tool. In the part of OpenMMLab, TensorRT will be discussed in more detail.

Inference. In order to inference the model on target, Nvidia has a program called Deepstream. It has functionalities to get data from various data sources such as live video, video files, image folders and more. It applies preprocessing, inference and does postprocessing of the results. With deepstream it is possible to apply models in a cascaded fashion, so that the output of one model can directly be used in another model. Deepstream makes use of Gstreamer and TensorRT in the backend. Deepstream uses separate configuration files where the settings of the aforementioned steps can be tuned. Alternatively to using the standard Deepstream program, it is possible to write custom apps in C/C++ and do custom pre/post processing with custom code. As Deepstream uses TensorRT in its backend, exported models can also be perfectly run with TensorRT outside of Deepstream.

6.3.2. OpenMMLab

OpenMMLab is an open source community which contributes to the deployment of deep learning models. OpenMMLab's repositories like MMdetection and MMsegmentation house a variety of models and tools and are built around Pytorch.

Retraining. The availability of models is the strong point of the OpenMMLab repositories. The models are updated very frequently (monthly) and most of the state-of-the-art models are included in the repository. MMdetection builds upon another repository of OpenMMLab, called MMCV. MMCV is the computer vision toolbox of OpenMMLab, and is the fundament of MMdetection and MMsegmentation. With most architectures, there are various amount of pretrained models available with many different backbones. If wanted, each model can be configured differently very easy. MMdetection works with configuration files in which different parts of the model can be swapped easily. Most of the models have been pretrained on COCO-dataset.

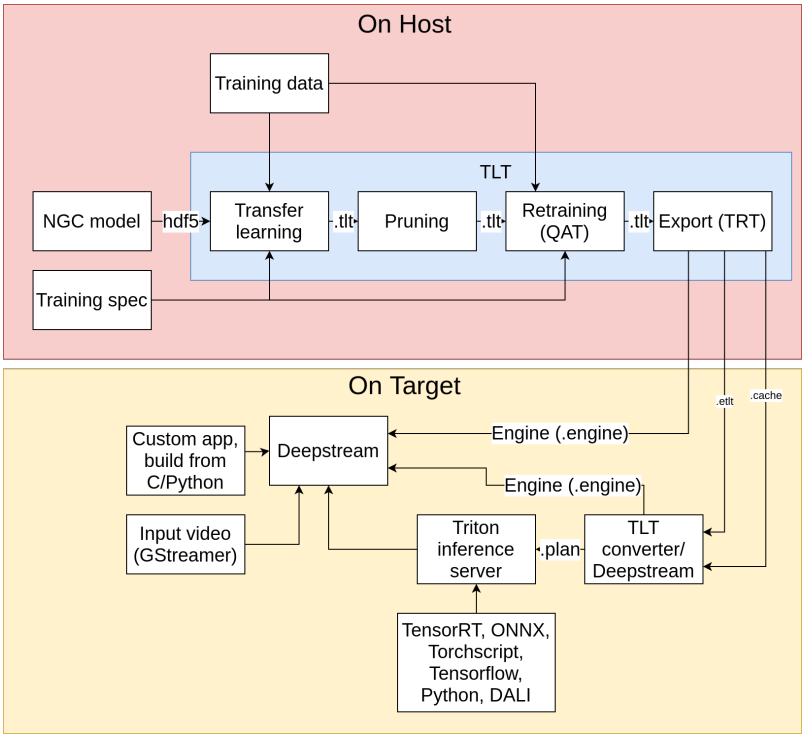


Figure 6.3: Nvidia workflow for deploying AI at the edge using TLT and Deepstream.

Model compression. MMdetection natively does not support any model compression. However, because it is based on Pytorch, in theory all of the compression techniques from Pytorch and industry can be used on the models. At the time of writing, this has not been performed yet. There has been a little research on possibly interesting compression techniques. Pytorch offers QAT, which can be achieved by performing fake quantization (no speedup, only a change of values) in training, so that the model can learn around the quantization. Additionally, pytorch also offers post-training quantization and also offers options for pruning the model.

Exporting. For exporting the model for use on the Jetson device, MMCV, and so also MMdetection have support for TensorRT integration. For this, during installation MMCV needs to be built with the TensorRT plugin. The workflow of exporting a MMdetection model to a TensorRT engine is as follows: First, the trained MMdetection model is converted to an ONNX model. Because this requires custom ONNX operators, MMCV also needs to be built with the custom ONNXRuntime plugin. Once the ONNX model has been generated, the ONNX model can be converted to a TensorRT engine. MMCV has a wrapper around the TensorRT python package. MMdetection currently only supports FP16 mode exporting. This has been found to work well. The TensorRT python package also supports INT8 exporting in a similar way as for the TLT exporting. An attempt has been made to export the model in INT8 precision. Although the exporting of the model succeeded, the model did not seem to reduce in size significantly, and the model output was incorrect. Due to shifting focus, these issues have not been further explored. A hypothesis for these problems might be that the quantization support of some operators was causing issues.

Inference After a TensorRT engine has been exported, the model could be loaded into Deepstream just like TLT. However, because the exploration of MMdetection has initially been chosen because of its freedom in developing models, there was a desire to explore a different option for inferencing. For exporting of the model, the python TensorRT package has already been introduced, which can also be used for applying inference. The benefit of this is that there is full control over the inferencing pipeline, as it is more low-level customizable than Deepstream. To do inference in python, one needs to write custom methods for applying pre- and post-processing. For custom user applications this is beneficial, because it will become very easy to connect the in- and output of the model to other building blocks for easy system integration.

6.4. Experimental Setup

As stated before, it was desired to run both object detection and instance segmentation tasks. To make this more concrete, two domain specific tasks are evaluated:

Object detection model for hardhats: Hardhats are an essential safety equipment for people working in construction and plant operations where the risk of falling objects is quite high. Real time surveillance and monitoring can help ensure compliance and prevent safety incidents. It is not always feasible to set up such a system in remote areas. Portable solutions such as edge based hardhat detectors can solve this problem. Therefore, for the object detection task, a Single Shot Detector (SSD) [30] model was trained on images hardhats. The lightweight mobilenet_v2 [40] architecture was used as a backbone for the model. 5000 images of construction and field operation workers with hardhats were collected from MakeML open source datasets [6]. TLT 3.0 (version 02/02/2020) was used to train, prune and quantize the model to int8 precision.

Instance segmentation model: For instance segmentation a Mask R-CNN [19] model was trained on domain-specific data. Two different backbones were used - resnet50 and resnet18 to train the models. TLT 3.0 (version 02/02/2020) was used to train and export the model.

6.5. Results

The exported models from TLT are converted to TensorRT engines on Jetson Xavier NX and Jetson Xavier AGX, using `tlt-converter` method. Inferences are tested on video streams using NVIDIA's deepstream framework. The models get exported in FP32, FP16 and INT8 precision and both their accuracy and inference speed is recorded. The result of object detection is listed in Table 6.1 and the result of the instance segmentation is listed in Table 6.2. Where available, the results are compared to the benchmarks provided by Nvidia. Both tables include the maximum and average inference FPS given an input stream. For object detection, it was not possible to collect average precision values for FP16 and INT8. In the table of instance segmentation, it can be seen that the accuracies do not drop during quantization. From the tables it can be seen that object detection is able to easily run real-time, but instance segmentation is still rather costly. The compressed INT8 models run around 1.75× as fast for object detection, where for instance segmentation it provides a speedup of around 3.75×. Qualitative results on test data are visualized in Fig. 6.4 and Fig. 6.5. Furthermore, a snapshot of live hardhat detection on the author is shown in Fig. 6.2.

Figure 6.4: Qualitative results for hardhat detection.

[6] https://makeml.app/datasets/hard-hat-workers, accessed March 22, 2022

Figure 6.5: Qualitative results for hardhat detection.

Table 6.1: Results for object detection on SSD with mobilenet_v2 backbone. Results include the maximum and average FPS achieved and the average precision.

	Jetson Xavier NX (FPS)		Jetson Xavier AGX (FPS)		Average Precision
Model	maximum	average	maximum	average	50 % IOU
FP32	150.31	150.52	231.11	229.54	0.38
FP16	223.19	219.17	353.75	353.48	n/a
INT8	265.20	264.36	390.96	390.73	n/a

Table 6.2: Results for instance segmentation on MaskRCNN with ResNet50 backbone. Results include the maximum and average FPS achieved and the average precision.

	Jetson Xavier NX (FPS)			Jetson Xavier AGX (FPS)			Average Precision 50 % IOU	
Model	maximum	average	benchmark	maximum	average	benchmark	detection	segmentation
FP32	1.29	1.27	n/a	-	-	n/a	0.404	0.342
FP16	3.8	3.7	n/a	6.51	6.36	n/a	0.405	0.344
INT8	4.86	4.78	5.4	7.93	7.36	9.2	0.405	0.336

7

Conclusion and recommendations

This work describes the effort of deploying deep learning to resource-constrained edge devices. The contributions are twofold: First, LAB-BNN has been presented, which is a novel work that identifies a bottleneck in BNN design and proposes a completely novel binarization function to alleviate this bottleneck. Second, efforts have been mentioned to compress existing state-of-the-art models and bring them physically to an edge device. It has been shown that this forms an interesting use case for technical companies.

Future work in BNNs. Possible future work for LAB-BNN has been described in the paper. As has been mentioned earlier in this thesis, currently the only way to benefit from BNNs is during inferencing. However, to reflect on section 1.1, the training of a model is what currently is taking the most energy and time. Therefore, a good direction of future work for the open-source community could be to merge the benefit of BNNs into the training stage as well, and provide energy-efficient training to common deep learning training platforms.

A

Code for LAB

```
@utils.register_alias("conv_binarizer_depthwise")
@utils.register_keras_custom_object
class LAB(_BaseQuantizer):
    r"""Custom learnable activation binarizer (LAB)
    """
    precision = 1

    def __init__(self, beta=None, name="convbin_depthwise", **kwargs):
        super().__init__(name=name+str(tf.keras.backend.get_uid(name)), **kwargs)
        if beta:
            self.soft_argmax_beta = beta
        else:
            uid = str(tf.keras.backend.get_uid("soft_argmax_beta"))
            self.soft_argmax_beta = tf.Variable(1.0, name="soft_argmax_beta"+uid)


    def build(self, input_shape):
        self.conv = layers.QuantDepthwiseConv2D(kernel_size=3,
                                                strides=1,
                                                padding='same',
                                                depth_multiplier=2,
                                                depthwise_quantizer=NoOp(precision=1))
        self.n, self.h, self.w, self.c = input_shape


    def call(self, inputs):
        @tf.custom_gradient
        def soft_argmax(x):
            out_no_grad = tf.argmax(x, axis=3)
            out_no_grad = tf.where(out_no_grad==0, tf.constant([-1.0]), tf.constant([1.0]))

            @tf.function
            def argmax_soft(x):
                out = tf.nn.softmax(x, axis=3)[:,:,:,1,:]
                return tf.math.subtract(tf.math.multiply(out,2),1)

            def grad(dy):
                gradient = tf.gradients(argmax_soft(x), x)[0]
                gradient = gradient * tf.expand_dims(dy, axis=3)
                return gradient
            return out_no_grad, grad


        x = self.conv(inputs)
        x = tf.reshape(x, [-1, self.h, self.w, 2, self.c])
        x = x * self.soft_argmax_beta
        outputs = soft_argmax(x)

        return super().call(outputs)

    def get_config(self):
        return {**super().get_config(), "soft_argmax_beta": self.soft_argmax_beta.numpy()}
```

Bibliography

[1] Alexander G Anderson and Cory P Berg. The high-dimensional geometry of binary neural networks. *arXiv preprint arXiv:1705.07199*, 2017.

[2] Tom Bannink, Adam Hillier, Lukas Geiger, Tim de Bruin, Leon Overweel, Jelmer Neeven, and Koen Helwegen. Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks. *Proceedings of Machine Learning and Systems*, 3:680–695, 2021.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. Back to simplicity: How to train accurate bnns from scratch? *arXiv preprint arXiv:1906.08637*, 2019.

[5] Adrian Bulat and Georgios Tzimiropoulos. Xnor-net++: Improved binary neural networks. *arXiv preprint arXiv:1909.13863*, 2019.

[6] Adrian Bulat, Georgios Tzimiropoulos, Jean Kossaifi, and Maja Pantic. Improved training of binary networks for human pose estimation and image recognition. *arXiv preprint arXiv:1904.05868*, 2019.

[7] Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. Bats: Binary architecture search. In *European Conference on Computer Vision*, pages 309–325. Springer, 2020.

[8] Tianlong Chen, Zhenyu Zhang, Xu Ouyang, Zechun Liu, Zhiqiang Shen, and Zhangyang Wang. " bnn-bn=?": Training binary neural networks without batch normalization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4619–4629, 2021.

[9] Hsu-Hsun Chin, Ren-Song Tsay, and Hsin-I Wu. A high-performance adaptive quantization approach for edge cnn applications. *arXiv preprint arXiv:2107.08382*, 2021.

[10] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53(7):5113–5155, 2020.

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

[12] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.

[13] James Diffenderfer and Bhavya Kailkhura. Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning a randomly weighted network. *arXiv preprint arXiv:2103.09377*, 2021.

[14] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11408–11417, 2019.

[15] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[16] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. Riptide: Fast end-to-end binarized neural networks. *Proceedings of Machine Learning and Systems*, 2:379–389, 2020.

[17] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[20] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[21] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015.

[22] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*, 2016.

[23] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[25] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[26] Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Yan Wang, Yongjian Wu, Feiyue Huang, and Chia-Wen Lin. Rotated binary neural network. *Advances in neural information processing systems*, 33:7474–7485, 2020.

[27] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. *Advances in neural information processing systems*, 30, 2017.

[28] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[29] Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. Pruning algorithms to accelerate convolutional neural networks for edge applications: A survey. *arXiv preprint arXiv:2005.04275*, 2020.

[30] 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*, pages 21–37. Springer, 2016.

[31] 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 (ECCV)*, pages 722–737, 2018.

[32] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. In *European Conference on Computer Vision*, pages 143–159. Springer, 2020.

[33] Samaneh Mahdavifar and Ali A Ghorbani. Application of deep learning to cybersecurity: A survey. *Neurocomputing*, 347:149–176, 2019.

[34] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. In *International Conference on Learning Representations*, 2019.

[35] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12, 2006.

[36] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[37] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2250–2259, 2020.

[38] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[39] Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[41] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Thirty-First AAAI conference on artificial intelligence*, 2017.

[42] Yixing Xu, Kai Han, Chang Xu, Yehui Tang, Chunjing Xu, and Yunhe Wang. Learning frequency domain approximation for binary neural networks. *Advances in Neural Information Processing Systems*, 34, 2021.

[43] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. Bmxnet: An open-source binary neural network implementation based on mxnet. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1209–1212, 2017.

[44] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, and Tao Mei. dabnn: A super fast inference framework for binary neural networks on arm devices. In *Proceedings of the 27th ACM international conference on multimedia*, pages 2272–2275, 2019.

[45] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[46] Baozhou Zhu, Zaid Al-Ars, and H Peter Hofstee. Nasb: Neural architecture search for binary convolutional neural networks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[47] Baozhou Zhu, Zaid Al-Ars, and Wei Pan. Towards lossless binary convolutional neural networks using piecewise approximation. *arXiv preprint arXiv:2008.03520*, 2020.

[48] Shilin Zhu, Xin Dong, and Hao Su. Binary ensemble neural network: More bits per network or more networks per bit? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4923–4932, 2019.

[49] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. Structured binary neural networks for accurate image classification and semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 413–422, 2019.