ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Binary Neural Networks At The Edge

Presentata da
Dott.Ing. Lorenzo Vorabbi

Supervisore
Prof. Davide Maltoni
Co-supervisore
Ph.D. Stefano Santi

Dottorato di Ricerca in
Computer Science and Engineering
Ciclo XXXVI

Coordinatore Dottorato
Prof. Ilaria Bartolini

Settore Concorsuale
09/H1

Settore Scientifico Disciplinare
ING-INF/05
SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

ESAME FINALE ANNO 2024

Abstract

Binary Neural Networks at the Edge

Lorenzo Vorabbi

In the last decade, Artificial Intelligence demonstrated to achieve great advancements in many areas such as (but not limited to) Computer Vision, Natural Language Processing, Reinforcement Learning, and Autonomous Driving. Nowadays, AI systems potentially overcome the intelligence observed in human beings as the ambition of developing AI models with cognitive, learning, and problem-solving abilities is the driving force in AI research and development. Achieving this goal would imply the creation of highly sophisticated AI systems capable of generalizing knowledge, learning from diverse data sources, and exhibiting a level of adaptability and creativity comparable to human intelligence. Most of the performance improvements obtained by AI systems have been reached by increasing the complexity and memory footprint of the models, leading to solutions whose deployment is most of the time prevented on resource-constrained hardware. Even though many works in literature addressed the problem of compressing and reducing the computational overhead of a neural network, a lot of effort is still required to successfully deploy a complex AI solution on tiny/embedded devices.

Additionally, current AI solutions achieve poor performances at adapting incrementally to new data. This phenomenon, named *catastrophic forgetting*, happens naturally in Deep Learning architectures when a classical training algorithm, such as backpropagation, is applied. By learning through a sequence of *experiences* incrementally, where the model cannot access old training data, the model knowledge falls resulting in the forgetting of past samples.

During my Industrial PhD activity, pursued at University of Bologna and Datalogic, I investigated the optimization of neural network inference and continual learning on edge devices, because they are central to guarantee good performances of the edge and tiny devices produced by Datalogic. AI technology could considerably empower the capabilities of such devices but many hardware and processing constraints have to be satisfied. Many challenges persist to deploy models on embedded systems that can solve complex tasks and the main activity of this work was aimed to simplify and reduce the computational flow and the memory bottlenecks, while preserving the original performance accuracy of the model. This research focused on the exploitation of extremely low-bit-width models, adopting the Binary Neural Networks, which use only 1-bit to represent weights and activations. Binary networks are naturally suitable to be accommodated on low-power devices as they replace the expensive multiply-and-accumulate operation with bit-wise arithmetic, which is more efficient.

Furthermore, we proposed novel techniques to allow the incremental learning of a binary neural network directly on-device, as this is the standard scenario of many real-world applications. On-device learning remains a formidable challenge, especially when dealing with devices that have limited computational capabilities and models that use low bit-width representation like binary neural networks. This thesis will focus on the applicative aspects that embrace continual learning approaches and binary neural networks to allow the deployment of practical deep learning applications in real-world scenarios.

Outline

Chapter 1 introduces the concepts and motivations of this research activity, outlining the requirements and constraints of real-world applications addressed by Datalogic devices. In particular, it points out the pros and cons of a deep learning approach introducing some of the state-of-the-art techniques adopted to compact and compress a deep learning model.

Chapter 2 describes the Binary Neural Networks by considering several aspects that have driven the advancements and improvements of 1-bit networks. This chapter represents the background knowledge of methods and approaches used for binary networks adopted in the next chapters.

Chapter 3 describes our contributions to the improvement and simplification of data flow of binary networks. In particular, it introduces a technique that reduces the bit-width of binary convolution outputs to 8-bit, removing floating-point computation. Additionally, this chapter describes a method to binarize the input layer of a binary network more efficiently than state-of-the-art solutions.

Chapter 4 considers the on-device learning (using a continual learning approach) use case applied to binary neural networks. The literature lacks of contributions that consider the combination of these topics and this chapter details the solutions we proposed to continuously train binary networks adopting *ad-hoc* quantization schemes.

Chapter 5 discusses some real-world applications that can be improved by adopting the techniques presented in this thesis. In particular, it shows the advantages achievable by adopting binary neural networks for the detection and localization of two-dimensional codes, which is a processing-intensive task with a challenging latency constraint. Additionally, it reports also two continual learning scenarios that need to continuously adapt a binary model on-device.

Eventually, conclusions and future challenges are discussed in Chapter 6.

Acknowledgements

First of all, I would like to thank Datalogic, the CEO Valentina Volta, the Chairman Ing. Romano Volta, and the executives who gave me the fantastic opportunity to pursue a PhD. Within this academic journey, I got to explore in depth the machine learning and deep learning fields that could significantly contribute to the improvement of Datalogic products.

A special thanks goes to my advisor prof. Davide Maltoni for his guidance and support during my PhD, spent during the Covid pandemic period, which complicated human relationships.

I would like to take this opportunity to thank my co-advisor Stefano Santi, that is much more than this. He is a friend, colleague, manager, mentor, and professor who remarkably contributed to the overall development of my technical skills, being a guide and a landmark for my activity within Datalogic. Without his support, patience, and driving path, I would not be able to reach the planned goals. He taught me *how to* solve problems and shape the mindset for applied research tasks and I will be forever grateful to him. He is one of the best scientists that I have ever met.

A special thanks goes to my colleagues at SmartCity-BioLab Guido Borghi and Lorenzo Pellegrini who supported me. In particular, for the Friday discussions held with Guido covering both research stuff and other general topics. I cannot miss my current manager Maurizio, who supported my research activity.

Eventually, I would like to dedicate this work to my family: Simona, Giulio, Giovanni, and Licia (from the oldest to the youngest). The caring grandparents (Cristina, Rodolfo, Daniela, and Riccardo) supported us by taking care of our kids every time we needed it and to my brothers Matteo (who undertook this path before me) and Giacomo. Particularly, without Simona I could not afford the PhD opportunity. I remember well how many night hours (after midnight) I spent doing research, while she was taking care of our little children, feeding them, and being close to them during their *active* night hours. Especially at the beginning of the PhD, she encouraged and supported me when things were hard and the desired goals were too far to be considered feasible. I cannot forget my kids, I hope that the effort and difficulty requested by this research activity could be an inspiration for them later in life.

Contents

Abstract iii

List of Figures ix

List of Tables xiii

1 Introduction 1
- 1.1 Deep Learning (DL) . . . 1
  - 1.1.1 Convolutional Neural Networks (CNNs) . . . 3
  - 1.1.2 Training versus Inference . . . 3
  - 1.1.3 Optimization techniques to reduce computation . . . 4
    - Quantization . . . 5
    - Pruning . . . 8
- 1.2 Datalogic Use Cases . . . 9
  - 1.2.1 Open Issues in DNNs . . . 10
  - 1.2.2 Contributions of this thesis . . . 12

2 Binary Neural Networks (BNNs) 15
- 2.1 Forward Propagation . . . 16
- 2.2 Backward Propagation . . . 18
- 2.3 Binary Neural Network Optimization . . . 20
  - 2.3.1 Quantization Error Minimization . . . 20
    - Scaling Factor . . . 20
    - Quantization Function . . . 21
    - Activations/weights distribution . . . 23
  - 2.3.2 Loss Function Improvement . . . 23
  - 2.3.3 Gradient Approximation . . . 24
  - 2.3.4 Network Topology Structure . . . 25
  - 2.3.5 Tricks and strategy for training Binary Neural Networks . . . 28

3 Improving BNN Inference 31
- 3.1 Optimizing data-flow in Binary Neural Networks . . . 31
  - 3.1.1 Data-Flow Optimizations . . . 32
    - Two-stage Clipping . . . 32
    - Batch Normalization Optimization . . . 35
    - Binary Direct Convolution optimization on ARM . . . 38
  - 3.1.2 Experimental Results . . . 39
    - Efficiency Analysis . . . 39
    - Accuracy Analysis . . . 40
  - 3.1.3 Final Remarks on Data-Flow Optimization . . . 41
- 3.2 Input Layer Binarization with bit-plane Encoding . . . 45
  - 3.2.1 Method . . . 46

3.2.2 Implementation Details . . . 49
3.2.3 Experimental Results . . . 53
3.2.4 Final Remarks on Input Binarization using bit-plane Encoding 54

4 Continual Learning with Binary Neural Networks 55
4.1 Binary Neural Networks and CWR* . . . 55
4.1.1 Continual Learning . . . 56
4.1.2 Method . . . 57
Gradients Computation . . . 57
Quantization Strategy . . . 59
4.1.3 Experiments . . . 60
4.1.4 Results and Remarks . . . 63
4.2 Enabling On-device Continual Learning with Binary Neural Networks and Latent Replay . . . 67
4.2.1 Method . . . 68
Continual Learning with Latent Replays . . . 68
Quantization of activations and weights . . . 69
Quantized Backpropagation . . . 70
4.2.2 Experiments . . . 72
Accuracy comparison . . . 72
Reducing Storage in Latent Replay . . . 76
Splitting q_b in q_b^{bin} and $q_b^{non-bin}$ . . . 76
Efficiency Evaluation . . . 77
4.2.3 Final Remarks . . . 77

5 Preliminary Results on Datalogic Use Cases 81
5.1 Datamatrix Detection . . . 81
5.2 Hazard Symbols Localization and Classification . . . 84
5.3 Produce Classification . . . 86

6 Conclusions and Future Challenges 91
6.1 Efficiency of Binary Neural Networks . . . 91
6.2 Continual Learning at the Edge . . . 92

Bibliography 95

List of Figures

1.1 Example of feature extraction using deep neural networks. The feature maps computed by each convolutional layer contain features of increased complexity as data goes deeper into the network. 3
1.2 Uniform quantization function. Real values of r are mapped into low-precision quantized values (marked with red dots) $Q(r)$. The distance between quantized red values is constant in uniform quantization. The green plot represents the quantization function that converts values from floating-point to integer representation (the corresponding red dots). 6
1.3 On the left it is illustrated an example of symmetric quantization (using a restricted range of $[-127, +127]$) that preserves zero representation. On the right an example of asymmetric quantization. 6
1.4 Flow chart of the steps required to perform Quantization-Aware Training (on the left) and Post-Training Quantization (on the right). 7
1.5 Comparison of the steps required in simulated quantization (Left) and integer-only quantization (Right). 8
1.6 Example of devices produced by Datalogic. 10

2.1 BNN vs. CNN. 16
2.2 Internal steps of a 32-bit CNN neuron compared to a naive BNN cell. . 16
2.3 Naive BNN forward propagation compared to 32-bit CNN. Popcount operation refers to Equation 2.6. 17
2.4 Backward pass approximation as suggested in [58]. 19
2.5 BNN optimization solutions. 21
2.6 Shapes of forward and backward passes. 26
2.7 Shapes of forward and backward passes. 27
2.8 Representative BNN block structures. 28

3.1 a) Standard BNN blocks are used in [113] and [88]. b) BNN block with output convolution clipping used during training. c) Optimized BNN block adopted during inference. Popcount operation is performed using saturation arithmetic to keep the data width to 8 bits at inference time. BN is replaced by a comparison in case a, while in b BN is 8-bit quantized. 33
3.2 Example of output distributions after binary convolution. a, refers to a VGG style network while b to a ResNet architecture. Green shows the distribution before the BN layer and red afterward. 34
3.3 **a)**: 8-bit symmetric quantization procedure that reserves fractional/integer bits based on the range of input 32-bit floating point values. **b)**: implementation of the BN layer with 8-bit quantization using an internal 16-bit representation to preserve accuracy. 36

3.4 The 7×7 input image with 3 different channels (denoted by color) is convolved with two separate kernels to obtain a 5×5 output with two output channels. To better exploit the SIMD 128-bit registers a different memory layout for kernel is devised: $[out_{channels}, H_{filter}, W_{filter}, in_{channels}]$. 38
3.5 The $3 \times 3 \times 128$ input patch is convolved (XNOR + popcount) with one kernel through the Extract sign bit, XNOR, and then popcount operations. Popcount is performed using *vcnt*, summing in pairs the *vcnt* output, and the last step uses the *addv* operation. TL (top left), TM (top middle), TR (top right), and ML (middle left) indicate the position of elements inside the 3×3 patch. 39
3.6 Latency evaluation of our method compared to DaBNN and LCE on Raspberry Pi 3B (a) and 4B (b) devices. 40
3.7 Training loss and testing accuracy curves for VGG11 and VGGSmall on CIFAR10 of the first and second training stages. 42
3.8 Training loss and testing accuracy curves for VGG11 and VGGSmall on SVHN of the first and second training stages. 43
3.9 Training loss and testing accuracy curves for ResNet-18 on CIFAR10 of the first and second training stages. 44
3.10 Training loss and testing accuracy curves for ResNet-18 on SVHN of the first and second training stages. 44
3.11 *(a)* Standard scenario of BNNs where the first convolutional layer is not binarized; weights and inputs are used in 8-bit/floating-point representation. *(b)* Typical approach of the works that binarized the first layer F_1 incrementing the number of input channels; in this case, the input expansion is actually an additional layer. *(c)* Our approach, where depth-wise convolutions are applied to input bit-planes and the resulting maps can replace the F_1 layer, producing a more compact model. 46
3.12 a Example of bit plane representation for a 3×3 8-bit image. b Image representation in bit planes. Each column refers to a bit index extracted from the image; for representation purposes, bit 1 is converted to 255 while bit 0 remains 0. In this example, all bit planes refer to channel G of RGB images. 48
3.13 Binarization process of input layer. a shows the rearrangement phase that extracts, for each bit position of the encoded pixel a bit plane. b shows the binary depth-convolution block applied to each bit plane; the depth multiplier (N) is a hyperparameter and it is dataset and model dependent. 50
3.14 Binarization process of input layer. a shows how to weigh differently feature maps extracted from different bit planes; maps related to the most significant bits receive a higher multiplication factor. In b, the feature maps related to the same 8-bit input channel are fused together through an addition. 51

4.1 Double quantization scheme that uses a different quantization level for weights/activations used in forward and backward pass. 59
4.2 Quantization scheme adopted using q bits for weights and activations. 60
4.3 Accumulation of gradient quantization errors (Mean Absolute Error in percentage using a logarithmic scale) between quantized and floating-point versions for each experience. During the first experience the gradient computation is always executed in floating-point. 61

4.4 CORe50 accuracy results using different quantization methods. 64
4.5 CIFAR10 accuracy results using different quantization methods. 65
4.6 CIFAR100 accuracy results using different quantization methods. . . . 66
4.7 Continual Learning with latent replay memory. When using a BNN the activations stored in the replay memory can be quantized to 1-bit. . 67
4.8 Quantization scheme that uses a different number of bitwidth for forward (q_f) and backward (q_b) pass. Usually, trainable non-binary layers are Batch Normalization [60], Addition and Concatenation layers. . 69
4.12 LR memory requirement using different quantization levels and corresponding test set accuracy on CIFAR10 (a) and CORe50 (b). We considered 15, 20 and 30 elements for each class inside LR; for case (a) we adopted Reactnet-18 model while in (b) we used Quicknet. 79
4.13 q_b memory requirement using different quantization bitwidths for backward layer on CORe50 (a) and CIFAR10 (b). 80

5.1 Examples of 1D and 2D codes. Figure 5.1a represents a *linear* barcode, where the red segment indicates the bars and spaces relative to the letter *D*. Figure 5.1b shows an example of datamatrix symbology. The red elle represents the finder pattern of Datamatrix. 81
5.2 Examples of Datamatrix codes used to train our proprietary BNN model. 83
5.3 Examples of Hazmat classes supported by our proprietary BNN model used for localization and classification tasks. 84
5.4 Example of a highly flammable Hazard symbol. 85
5.5 Accuracy comparison of our custom BNN based on the method BNN+LR+CWR* (Section 4.2.2) with a Yolov5N 8-bit quantized model. 86
5.6 Examples of a weight-scale used for produce recognition. 87
5.7 Examples of fruits and vegetables used to train our proprietary BNN model used for the classification task. 88
5.8 Accuracy comparison of our custom BNN based on the method BNN+LR+CWR* (Section 4.2.2) with a Yolov5N 8-bit quantized model. 89

List of Tables

2.1 BNN XNOR operations 17

3.1 Accuracy comparison (top1) of our method with SOTA on CIFAR10 and SVHN. 41
3.2 Accuracy comparison of our method with SOTA on ImageNet. 41
3.3 Comparison of the first layer MACs required by our method with respect to the state-of-the-art solutions. Input data has a shape $H \times W \times C\,(32 \times 32 \times 3)$ and a precision of M bits; in this example, the first convolutional layer has $F_1 = (128)$ filters with size $F \times F\,(3)$. The expansion channels is $K = 32$ for methods [31, 159]. The depthwise multiplier of our method can be chosen as $N_1 = \left\lfloor \frac{F_1}{C} \right\rfloor = 42$. We conducted our experiments using also a lower value, $N_2 = 32$ instead of N_1 and only 4 bits of input pixels. P represents the number of bit planes extracted by step 3.13a. 49
3.4 Top1 accuracy (%) results of test set on CIFAR10. In the first part we report the result of the first test scenario (standard conditions); in the second half, the results achieved in the second scenario (reducing the MACs of binarization of input layer). 53
3.5 Top1 accuracy (%) results of test set on SVHN. 53
3.6 Top1 accuracy (%) results of test set on CIFAR100. 54

4.1 The table represents a comparison of memory usage (# parameters) for different BNN models. With B we report the number of binary weights that can be updated during back-propagation; with NB the number of non-binary weights. The choice of latent replay (LR) level is discussed in Section 4.2.2. It is worth noting that the largest part of memory weights is used by binary weights. 70
4.2 Efficiency comparison of our method implemented on two different embedded boards, *i.e.* Raspberry Pi 3B and 4B, using Mobilenetv2 and Quicknet model. As shown, our solution achieves up to 2.2× speedup on the same platform. 77

5.1 Acuracy and speed-up comparison of our custom BNN model with a SOTA model (Yolov5 Nano) on our proprietary Datamatrix dataset measured on a Raspberry Pi 3B. 82

Chapter 1

Introduction

The success of Deep Neural Networks (DNNs) in various high-impact applications over the past decade has been remarkable. These applications include object classification [20], speech recognition [43], computer vision [72], natural language processing [135], self-supervised learning [12], reinforcement learning [99], robotics [79], autonomous driving [27], games [126], and sustainable artificial intelligence [94].

However, the increasing size of state-of-the-art neural networks has led to significant challenges in terms of storage and computation requirements. For example, GPT-3, a leading model for natural language processing, contains a massive 175 billion parameters, resulting in a huge training cost. Training such models is only feasible through extensive parallelization, requiring thousands of GPU units. This presents significant financial and environmental implications, as the training procedure of GPT-3 alone costs approximately 12 million for a single instance of the model. Even pioneer neural network architectures ([72, 50]) pose great challenges due to their large parameter counts, making them impractical for deployment on resource-constrained platforms such as embedded devices, mobile phones, or small-scale robotic platforms.

To address these challenges, several studies focused on techniques for reducing the size, inference cost, and training cost of large-scale DNNs without sacrificing performance. These techniques include quantization [146, 113], knowledge distillation [55, 106], neural architecture search [87] (NAS), low-rank compression, and weight pruning [48], among others. These methods aim to make DNNs more practical and efficient for deployment on various platforms.

Furthermore, each application field requires a certain level of adaptability over time that neural networks have to fulfill because the real world is complex and changes constantly. The ability of a neural network model to adapt to new data collected over time is usually referred to as *incremental learning* or *continual learning* (CL). The main obstacle that limits the continuous adaptability of the models even nowadays is a phenomenon known as *catastrophic forgetting*. Catastrophic forgetting occurs when a model is trained incrementally and leads to poor performance accuracy over samples seen in the past.

1.1 Deep Learning (DL)

The success of neural networks took over 50 years to materialize because deep networks required three foundational technologies to become viable: large sets of training data, effective techniques to learn parameters, and high-speed hardware to make the duration of training feasible. Before ImageNet [119], there simply were not large

enough public datasets to support deep networks. However, the large size of ImageNet introduced other problems, one of which was how to compute optimal parameters given so many training images. Traditional solvers that directly compute parameters based on the entire dataset were too computationally inefficient to scale to ImageNet proportions. Fortunately, by adopting the Stochastic Gradient Descent (SGD) technique, it is possible to iteratively adapt network parameters to minimize the loss function. SGD operates on small batches of random samples at a time and computes updates of a network's parameters via gradient calculation that reduces error on that batch. When run on a large dataset for many epochs, the parameters slowly improve and the loss descends to a local minimum. Unfortunately, training a network using SGD often requires hundreds of epochs and, when each epoch consists of processing a million images, the prospect of training and adjusting a deep network becomes prohibitive, and would likely be completely infeasible on a CPU. Since 2007, Nvidia released CUDA, allowing the training of the neural networks directly on GPUs, that massively parallelize computation and provide a huge speed-up of the training time. Since AlexNet [72], deep neural networks (DNNs) have proliferated quickly and today dominate most machine learning applications across various domains such as computer vision, natural language processing, and signal processing.

Deep learning is usually referred to as neural networks having more than three layers, i.e., more than one hidden layer. Nowadays, DNNs are built around the concept of constructing a hierarchy of hundreds, or even thousands of layers. The deep structure can learn high-level features and generalize better than shallower neural networks. The layers placed at the early stages of the model can be interpreted as low-level feature extractors, such as lines and edges. In subsequent layers, these features are combined into higher-level features until the last layer which aggregates all previous information to predict the final output of the network (e.g. classification of an object in the scene). Some of the key aspects that contributed to the popularity and widespread adoption of deep learning are listed below:

- **High-dimensional learning**. In contrast to classical techniques that often rely on manually engineered features, hard to generalize even across similar tasks, Deep Learning can capture and learn representations that better generalize. DL can learn directly from high-dimensional space of data without any human intervention or intermediate step of dimension reduction. This feature is crucial in real-world applications, such as computer vision tasks.

- **End-to-end training**. The primary goal of DL is to minimize a loss function representing the distance between the network output and the expected result. This is implemented through an iterative procedure that uses the gradient information to descend towards a loss function minimum. This process, where the gradient of the loss function is passed back across all layers of the network is referred to as *backpropagation*. Many variations of this technique have been proposed but the core idea remains substantially unaltered.

- **Data eager**. A key concept in DL is that the performance of a model is closely connected to the quantity of data available for training. Adding more training data is often more beneficial than investing time in designing new learning techniques. The availability of data (that covers adequately the input variations) usually constitutes a key factor for the achievement of good performance accuracy.

1.1.1 Convolutional Neural Networks (CNNs)

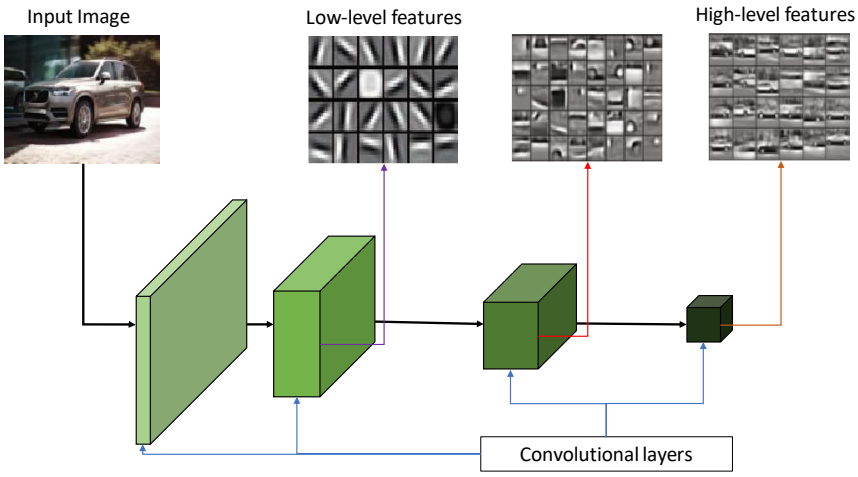


FIGURE 1.1: Example of feature extraction using deep neural networks. The feature maps computed by each convolutional layer contain features of increased complexity as data goes deeper into the network.

Convolutional neural networks emerged from studying the brain's visual cortex, and they have been used in image recognition since the eighties. The widespread adoption of CNNs in computer vision tasks is intrinsically connected to visual processing, as the concept of applying a "sliding window" fits many application scenarios, such as digits recognition [75]. In the last years, the increase in computational power and the amount of training data available significantly pushed the research on CNNs [38, 39, 118, 51], which are nowadays a fundamental building block in a visual recognition system. CNNs are a special type of DNNs that are composed of multiple convolutional layers. In such networks, each layer produces a higher level of abstraction of its input data, called *feature map*. The feature maps extracted by each convolutional layer go from low to high level as data go deeper into the network, as shown in Figure 1.1. Compared to more traditional multi-layer perceptrons, CNNs extract useful information by sliding a kernel of weights (learned at training time and shared for all input neurons) on the input data. In a convolutional layer, the neurons are not connected to every single input neuron, but only to those that belong to the *receptive field*, i.e. are local w.r.t. the considered neuron. The number of kernel parameters is related only to the kernel and input/output channel sizes, reducing the total number of weights. In 2012, Krizhevsky et al. [71] published a ground-breaking work, CNN-based, by outperforming conventional methods on a large-scale dataset, such as Imagenet. The authors showed that a convolutional neural network was able to learn useful representations on such high-dimensional data by adopting a simple training technique. Convolutional layers are particularly suited for computer vision applications as they exploit local information to extract useful information. Nowadays, CNNs are widely employed in a multitude of applications and thousands of papers have been written on this topic.

1.1.2 Training versus Inference

The distinction between training and inference in the context of neural networks is crucial as they require different computational needs. Specifically, training involves refining the network's learnable parameters through an iterative process on a dataset (often large), taking several hours to multiple days to achieve the desired accuracy.

This phase is resource-intensive and the hardware used usually involves the usage of expensive GPUs (or TPUs) that require high power consumption. Most of the time, to train very huge and deep models, training is performed in the cloud or powerful local servers.

In contrast, once trained, the network is used for inference, which can happen both in the cloud or at the edge. In real-world applications, it is often desirable to execute DNN inference directly at the edge near the sensors. Edge processing allows to maintain privacy of data avoiding the costs related to the cloud. The focus on optimizing inference, as opposed to training, is justified by several factors. While training is a one-time cost, inference scales with the usage of the application. As a result, the optimization of inference speed and model size is of paramount importance, especially given the widespread use of trained models in various applications.

The focus on inference optimization is driven by the need for efficient and scalable deployment of models in real-world scenarios. The techniques presented in this thesis are primarily addressed to improve the speed and reduce the model size during inference. An efficient inference pass opens the door to accommodate CNNs on tiny embedded devices and also enables the chance to apply an incremental learning approach directly on devices. On-device learning represents a fundamental milestone for DNNs as it allows to automatically adapt a model to new data, directly at the edge, which would foster significantly the DNNs spread. Unfortunately, training at the edge imposes severe limitations and constraints that must be addressed to reach a good compromise between hardware resource utilization and processing load, as reported in Chapter 4.

1.1.3 Optimization techniques to reduce computation

As introduced in previous sections, DNNs can reach human-level capabilities but at the cost of significant computational complexity. To reach these superior performances, in recent years, DNNs stacked thousands of layers resulting in longer and complex training procedures. As a result, such deep models are difficult to deploy on embedded platforms that cannot provide enough memory and computation resources. Luckily, as reported by Sze et al. [129], deep neural networks are over-parameterized, i.e. redundant neurons are present, and by removing the neurons that do not contribute to an improvement in the accuracy of results it is possible to reduce model complexity. To achieve efficient and accurate DNNs it is necessary to rethink the design, training, and deployments of DNNs. A large part of the literature addressed previous limitations by improving the efficiency of DNNs (lower latency, memory footprint, and energy consumption) while preserving accuracy and generalization. Network optimization techniques can be substantially divided into several groups: 1) designing efficient architectures, 2) designing architectures considering hardware constraints, 3) knowledge distillation, 4) network quantization, and 5) network pruning.

For the relevance of the topics covered in this thesis, in the next paragraphs, we cover quantization and pruning strategies, respectively.

Quantization

Quantization is the process of mapping discrete values belonging to a large input set into output values belonging to a smaller countable set. In DNNs, quantization is a technique used to reduce the size of a model by converting its high-precision floating-point weights to lower-precision representations, such as 16-bit or 8-bit floating-point or fixed-point formats. This conversion leads to improvements in model size, inference speed, and resource utilization without compromising accuracy significantly. Quantization is particularly beneficial for reducing memory bandwidth requirements and enhancing cache utilization during model deployment. Additionally, quantization can be employed to simplify and optimize the computational pipeline of large-scale models, that are anyway too big to fit embedded devices. For instance, large language models (LLMs) can achieve excellent performance on various tasks [12, 157] but their huge memory footprint and computational overhead increase the deployment cost and energy consumption. By quantizing both weights and activations using low-bit integers [153], it is possible to reduce GPU memory requirements and speed up the processing-intensive operations.

Quantization approaches vary based on the model, requiring prior knowledge and extensive fine-tuning for successful implementation. Challenges and trade-offs arise in terms of accuracy and model size, especially with low-precision integer formats like 4-bit fixed-point, which can have a limited dynamic range and lead to accuracy loss during conversion from higher-precision floating-point representations.

Assuming to have a neural network with L layers, whose learnable parameters (denoted as θ) and hidden activations (denoted as h_i for layer i) stored in floating point precision, the goal of quantization is to reduce the precision of both θ weights and h_i intermediate activations to low-precision, with minimal loss of accuracy.

A common quantization function [62, 103] $Q(\cdot)$ used in DNNs is expressed as follows:

$$Q(r) = Int\left(\frac{r}{S}\right) - Z \tag{1.1}$$

where r is the real-valued input, S is the real-valued scaling factor, Z is the integer zero point, and $Int(\cdot)$ approximates a real value to an integer value with a rounding operation. Equation 1.1 is known as *uniform quantization* because the quantized values are equally spaced, as depicted in Figure 1.2. The inversion of Equation 1.1, known as *dequantization*, can be used to recover r real values from $Q(r)$ quantized values, as reported below:

$$\widetilde{r} = S(Q(r) + Z) \tag{1.2}$$

Recovered $\widetilde{r}$ values do not match exactly the original r values due to the rounding operation.

The scaling factor S uniformly divides the range of real values, as reported below:

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{1.3}$$

where $[\alpha, \beta]$ denotes the range of real values and b is the quantization bitwidth. In order to compute the scaling factor S, the range $[\alpha, \beta]$ must be determined through a process named *calibration*. When $\alpha \neq -\beta$ the Equation 1.3 is referred to as *asymmetric quantization*, as it changes the zero representation between real and quantized values (Figure 1.3). In asymmetric quantization, a typical choice for the parameters is $\alpha = r_{min}$, $\beta = r_{max}$. By choosing $\alpha = -\beta$, Equation 1.3 is referred to as *symmetric*

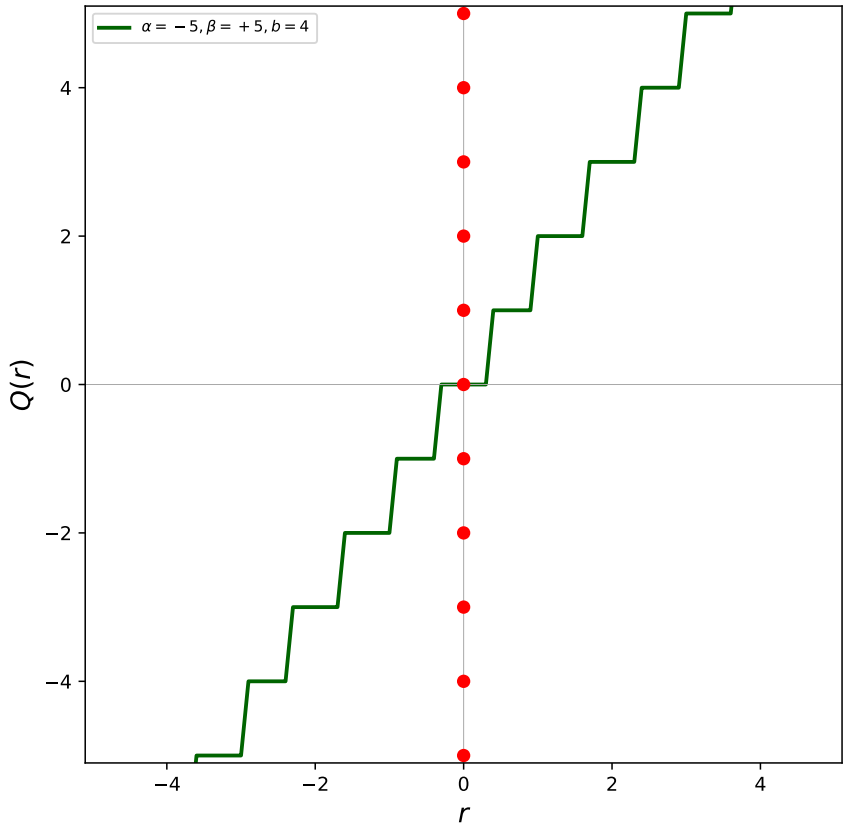


FIGURE 1.2: Uniform quantization function. Real values of r are mapped into low-precision quantized values (marked with red dots) $Q(r)$. The distance between quantized red values is constant in uniform quantization. The green plot represents the quantization function that converts values from floating-point to integer representation (the corresponding red dots).

quantization, as it preserves the zero point representation (Figure 1.3). In symmetric quantization, a common choice is to set $-\alpha = \beta = \max(|r_{max}|, |r_{min}|)$. Asymmetric quantization better exploits the quantized range than symmetric quantization when adopted to quantize unbalanced data, such as outputs of the ReLU activation function. On the other side, symmetric quantization simplifies the computation as it sets $Z = 0$, resulting in:

$$Q(r) = Int\left(\frac{r}{S}\right) \tag{1.4}$$

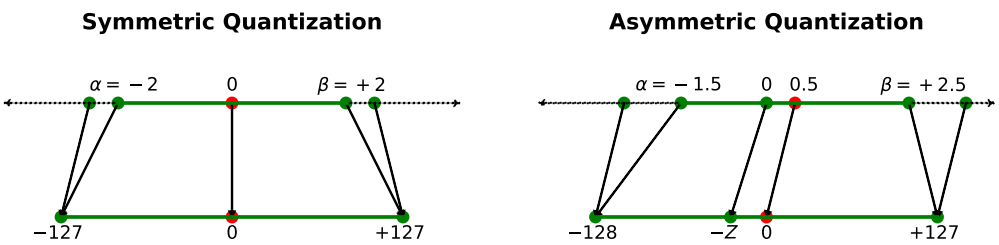


FIGURE 1.3: On the left it is illustrated an example of symmetric quantization (using a restricted range of $[-127, +127]$) that preserves zero representation. On the right an example of asymmetric quantization.

The determination of the clipping range $[\alpha, \beta]$ is typically executed statically both for weights and activations as the dynamic determination of the range for the activations is too computationally expensive. The range for weights can be statically determined at the end of the training by freezing the parameters before inference. For activations, the range is calculated by feeding the network with a series of calibration samples and computing the resulting range for each activation that needs to be quantized.

The quantization of weights and activations inevitably introduces approximations that could degrade model accuracy. It is therefore necessary to slightly change quantized values. This can be achieved by retraining the model, through a process called Quantization-Aware Training (QAT), or performed without re-training, referred to as Post-Training Quantization (PTQ). Both processes are illustrated in Figure 1.4.

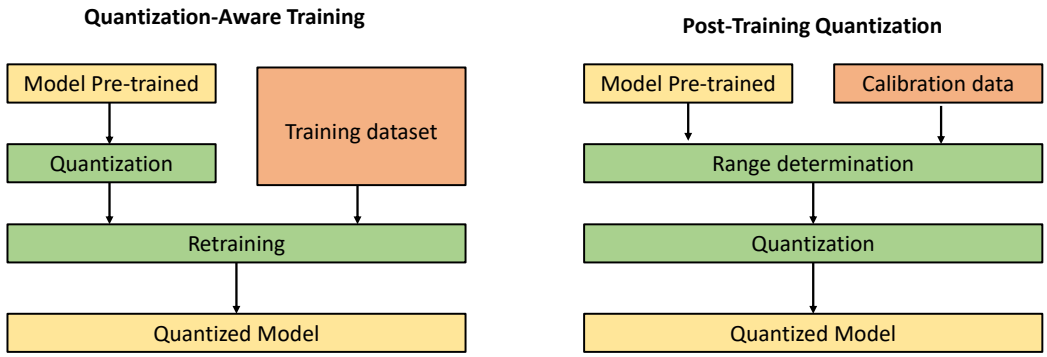


FIGURE 1.4: Flow chart of the steps required to perform Quantization-Aware Training (on the left) and Post-Training Quantization (on the right).

Quantization-aware training is a technique that simulates the quantization process during model training, helping to prepare the model for quantization. In QAT, forward and backward passes are executed on floating-point values which have been approximated using Equations 1.1 and 1.2. The execution of backward passes using floating-point precision is essential as quantized gradients could lead to zero gradient updates with high error. In QAT the Equation 1.1 is not differentiable and even worse, its gradient is zero almost everywhere. A common approach used to address this issue is to approximate the gradient of Equation 1.1 with the Straight Through Estimator [9] (STE), which substantially ignores the quantization operation by approximating gradient with the identity function. STE plays a central role also in Binary Neural Networks and it is further discussed in Section 2.2. The main drawback of QAT is the computation effort required to retrain the model. This process could require hundreds of epochs to recover floating-point accuracy.

A valuable alternative to QAT is represented by Post-Training Quantization, which quantizes weights and activations without any post-training phase [6, 17, 22, 33, 34, 35, 36, 52, 59, 77, 80, 97, 104, 125]. Unlike QAT, which necessitates a large portion of the training dataset, PTQ can be applied to use cases where the availability of data is limited or unbalanced. However, PTQ, as it uses a limited set of training data, yields lower accuracy compared to QAT.

The deployment of a quantized model can happen using directly the *integer-only quantization* or by *simulating quantization*. In the first case, a real quantized graph is deployed where all the operations are executed using integer arithmetic (e.g. 8-bit) and it can provide memory-saving and processing speed-up. Instead, the simulated

quantization generates a fake quantized graph where model weights are stored using integer precision, but the computationally expensive convolutions and matrix multiplications are performed using floating point precision. So, fake quantized graphs cannot provide good efficiency as they require to quantize/dequantize activations before each floating-point operation. Additionally, the execution of the graph requires more memory as all the intermediate activations have to be stored using floating-point precision. Nevertheless, fake quantization can be employed in scenarios that are bandwidth-bound rather than compute-bound. The comparison between integer-only and fake quantization graphs is depicted in Figure 1.5 (more detailed discussions on this topic can be found in [103, 37]).

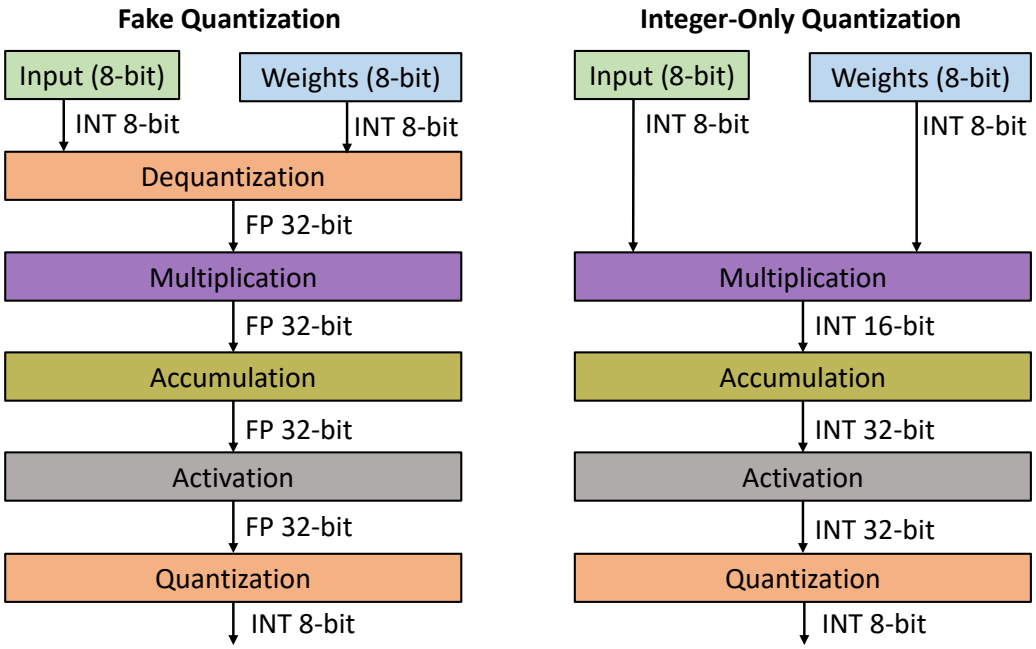


FIGURE 1.5: Comparison of the steps required in simulated quantization (Left) and integer-only quantization (Right).

Pruning

Network pruning involves the removal of redundant parameters or neurons that do not significantly contribute to the accuracy of results. Research on network pruning can be broadly categorized into sensitivity calculation and penalty-term methods [116] and recent works have shown continued interest and improvements in both categories or a combination of them. New pruning techniques have also been developed, and modern approaches can be classified based on various aspects, including whether pruning is structured or unstructured (depending on whether the pruned network is symmetric or not), neuron and connection pruning (based on the pruned element type), and static versus dynamic pruning. In the latter, the distinction between static and dynamic pruning lies in when the pruning steps are performed, offline for static pruning, while dynamic pruning occurs during runtime.

Magnitude-based pruning, introduced by Han et al. [48], involves iteratively removing weights below a certain threshold, fine-tuning the network, and repeating the process until accuracy begins to degrade. The method was applied to various networks, including LeNet [75] on MNIST, as well as AlexNet [72] and VGG16 [127] on ImageNet. The results showed that the number of weights could be significantly

reduced without sacrificing accuracy. The study emphasized the benefits of iterative pruning, showing that earlier layers are more sensitive to pruning, and iterative pruning is more effective than one-shot pruning. Pruning a weight means setting it to zero generating a sparse weight representation that can lead both to memory reduction and processing speed-up. Han et al. achieved an impressive $40\times$ compression rate reduction on AlexNet and VGG16. Additionally, Guo et al. [45] noted a limitation of magnitude pruning, where important weights might be prematurely removed. To address this, they proposed Dynamic Network Surgery (Dyn Surg), maintaining a mask indicating which weights to remove or retain in each training cycle.

In the realm of pruning, methods that operate at the granularity of filters and channels offer advantages over those dealing with sparse-weight matrices, as the former can leverage existing optimizations in many toolkits without requiring specialized libraries or hardware for sparse matrices. Approaches to filter and channel pruning generally fall into three categories:

- **Data-dependent channel pruning methods**: These methods operate on the premise that when presented with different inputs, output channels (or feature maps) should exhibit variations since they are designed to detect discriminative features.
- **Data-independent pruning methods**: These techniques utilize properties of filters and output channels, such as the proportion of zeros present, to determine which filters and channels should be pruned. These methods do not rely on specific data but focus on inherent characteristics.
- **Optimization-based channel approximation pruning methods**: These methods employ optimization techniques to reconstruct filters that approximate the output feature maps. The goal is to recreate filters that capture the essential information conveyed by the original ones, allowing for more informed pruning decisions.

Compared to quantization, pruning requires in general optimized libraries to take advantage of the sparse weight matrices and a specific hardware support [63, 19, 111] to be effective, as tensor processing unit or general purpose SIMD do not offer dedicated instructions. Mixing quantization with pruning can guarantee good speed-up improvements ($3-4\times$), as shown by Han et al. [48].

1.2 Datalogic Use Cases

Datalogic is a company that produces and sells devices used to automatically capture and identify data, operating in many market segments, such as retail, health care, transportation, logistics, and manufacturing. A representative set of Datalogic products is reported in Figure 1.6. Some of them (*Powerscan*, *Hand-Scanner*, *QuickScan*, *Rida* and *Memor*) are battery-powered, therefore requiring efficient processing to increase battery duration (adopting a low-power embedded CPU). Instead, other devices (such as *AV series*) employing more powerful CPUs, are cable-powered as they are used to analyze a huge amount of data with real-time constraints. All the devices shown in Figure 1.6 are equipped with an input sensor used to extract information from the scene: 1D/2D CMOS or CCD sensors but also 3D cameras. In general, the requirements (specific for each application/product) of depth-of-field (DOF) and field-of-view (FOV) force the adoption of high-resolution

FIGURE 1.6: Example of devices produced by Datalogic.

input sensors that are processing-hungry. The data-processing step usually represents the most critical part of Datalogic devices that must be efficient and optimized to reduce latency. By leveraging AI technology it would be possible to increase the performance of the devices but accurate considerations must be made. AI cannot slow down the actual processing even though could increase the accuracy of the results. Therefore, the scope of this work is focused primarily on deploying efficient deep neural networks on Datalogic devices. On such devices, which are usually limited in terms of memory, power (many devices are battery-powered), and computing capabilities, the deployment of neural network models can be challenging. Indeed, considering the barcode decoding scenario, even the low-end devices usually have to process images, having a resolution of 1280×1024, at a frame rate varying from 30 to 60 frames per second. In a simple use case scenario, with a frame rate of 30 fps, the time budget available to analyze each video stream image is around 30 milliseconds. Considering an off-the-shelf shallow model, such as MobileNet [57, 121, 70] or EfficientNet [130, 131], the inference time of its 8-bit quantized version (using technique introduced in section 1.1.3) on a Raspberry Pi 3B is around 300[1] milliseconds for an input image resolution of 224×224, as reported in [40]. This means that a huge speed-up is required to equip Datalogic devices with AI technology without adopting additional dedicated hardware such as a Tensor Processing Unit (TPU), ASIC, or FPGA that would increase the overall cost and power of the system. In Chapter 5, we present some real-world applications running on edge devices showing some preliminary details and solutions using the techniques reported in Chapters 3 and 4.

1.2.1 Open Issues in DNNs

Despite the aforementioned advantages of DNNs and CNNs, some other negative aspects persist, by preventing the deployments of such models on Datalogic's devices (Figure 1.6). In particular, the most important investigated within this thesis are reported below:

[1]Single thread computation using TF Lite with XNN-Pack.

- **Computationally inefficiency**. Deep Learning is computationally very demanding, requiring consistent computational power and time to train deep architectures. Even though many powerful GPUs (or specialized hardware such as TPU) are available for the training of large models, the deployment of such models on edge or embedded devices is still challenging because of a lack of resources. Neural networks are usually trained using 32-bit floating-point computation which is more inefficient compared to fixed-point computation. Moreover, many embedded CPUs are not equipped with floating-point units (FPUs) resulting in prohibitive forward pass timings. Some optimization techniques, such as quantization, have been adopted to speed up the computation but the processing load is still not acceptable for many industrial-embedded applications. In Chapter 3, some optimization techniques that allow to speed up Binary Neural Networks are introduced.
- **Memory demanding**. Deep Learning relies on a deep sequence of layers used to extract features to be propagated along the model. The memory necessary to execute the training and the inference is often too high to accommodate these models on embedded devices. The reduced volatile memory (RAM) available on such tiny systems usually does not fit the memory constraints, limiting or even preventing the usage of neural networks.
- **Poor availability of frameworks for on-device training**. Many inference engines, used to execute forward pass of neural networks, are available on the market, such as Tensorflow Lite, OpenVINO, ONNX Runtime and STMicroelectronics X-Cube-AI or NanoEdgeAI. They can handle many kinds of layers supporting both floating-point and fixed-point computation. Unfortunately, they completely miss the necessary implementations to execute efficient on-device training. This limitation prevents the deployment of models that can continuously learn on-device.
- **Lack of adaptability**. Despite the great improvements achieved in recent years, DNNs still reach poor performances when applied to the target application that inevitably differs from the training data. To face the continual learning challenges offered by real-world use cases, a DNN should be able to distill or eventually learn the core skills or concepts necessary to reach a goal. Ideally, neural networks should be able to incrementally improve when unknown data or problems are encountered. Unfortunately, Deep Learning and the iterative learning procedure (gradient-based) are not sufficient to achieve this level of adaptability and DNNs usually incur in the *catastrophic forgetting* issue. This research field, applied to efficient and quantized models, has been only marginally addressed in the literature. In Chapter 4, some continual learning solutions that work in combination with binary neural network models are presented.

To address previous DNN limitations, many strategies have been proposed but unfortunately, when a low-power device is used to accommodate a quite computationally demanding task with challenging processing constraints, state-of-the-art techniques are not sufficient to guarantee good performances, and further optimizations are requested.

1.2.2 Contributions of this thesis

Binary Neural Networks (Chapter 2) demonstrated to be a valid solution to deploy models on tiny low-power devices as they do not require the multiply-and-accumulate operation, relying exclusively on bitwise and popcount arithmetic. Additionally, BNNs heavily reduce the memory footprint of the models as they need to use only 1-bit for weights and activations. Therefore, BNNs represent a valid alternative to off-the-shelf models such as MobileNet, to be incorporated within Datalogic devices as they reach good speed-up preserving an accuracy comparable to non-binary models.

Pushing further research into binary neural networks is a fundamental element in empowering Datalogic devices with AI technology. Therefore, to facilitate the adoption of deep learning-based solutions within Datalogic products, extending them beyond the current predominant use of traditional and hand-crafted computer vision algorithms, the next points require to be addressed:

- *Efficiency in BNN data flow*. Despite the great advancements proposed in the literature, only a few works proposed solutions benchmarked on real embedded devices without using dedicated hardware such as FPGA or TPU. Most of the solutions proposed rely on floating-point computation to execute non-binary layers, therefore mixing quantized (binary) with floating-point computation. By improving BNN data flow it is possible to further reduce inference time and remove the constraint of using a floating-point unit, not available on many tiny devices (Section 3.1).
- *Fully-binary model*. Almost all the works on BNNs do not binarize the input layer of the model as the binarization of the first layer usually introduces a high accuracy gap compared to the equivalent model with the input layer in floating-point. The binarization of the first layer offers the opportunity to completely remove floating-point calculus allowing a simpler deployment on FPGA or ASIC devices, as the floating-point unit usually dominates the amount of silicon chip used (Section 3.2).

Furthermore, real-world applications require a certain level of adaptability to the constantly changing environment. The ability of the systems to adapt their behaviors represents a crucial feature for the widespread adoption of such devices. Many works in literature addressed the incremental or continual learning task but only a few of them focused on low bit-width efficient models. Putting effort into making a model computationally efficient is important as much as letting it adapt to new data acquired. The continual learning solution, to be employed in real-world applications, should investigate the following points, all related to the so-called *on-device learning* (as reported in Chapter 4):

- *Operate with limited memory available*. Training a model requires much more memory compared to the model memory footprint (weights) [83]. To avoid the catastrophic forgetting issue, a useful technique adopted is to replay past samples when learning new data. This approach requires additional memory to store past samples, further increasing the total memory demand. On embedded devices such memory amount is often challenging and ad-hoc optimization is requested to fit the constraints.
- *Learn with a truly-quantized model*. Network training is executed with floating-point precision to guarantee enough precision in gradients computation and

update. Unfortunately, models deployed on embedded devices are usually quantized (or binarized) to be more efficient. Improving model adaptability by learning with quantized weights and activation represents a great challenge that requires investigations to balance the trade-offs.

- *Handle unknown scenarios*. A system deployed on real devices should have the capability to handle unexpected scenarios and tasks. If a system is restricted to a specific setting then it is challenging for the system to learn and act autonomously and effectively.

Chapter 2

Binary Neural Networks (BNNs)

Binary Neural Networks (BNNs) are a specific type of neural network where the activations and weights in all hidden layers, usually except for the input and output layers, are represented with 1-bit values. BNNs can be seen as a highly compressed variant of NN (including Convolutional Neural Networks, CNNs), as they share the same structure but differ in the precision of their activations and weights. BNNs specifically focus on the technique of binarization, which involves converting 32-bit activations and weights into 1-bit values. This binarization process is used both to reduce the storage requirements of the model and also to minimize matrix computation costs by utilizing XNOR and popcount operations. Research by Rastegari et al. [113] has demonstrated that BNNs could achieve memory savings 32 times greater and perform convolution operations 58$\times$ faster than their 32-bit CNN counterparts. In traditional CNNs, a significant portion of computational costs is attributed to matrix multiplication within the convolution operation. The basic convolution operation, excluding bias, can be mathematically expressed as:

$$Z = I \circledast W, \tag{2.1}$$

where I and W represent the activations and weights, respectively, while Z is the output. This multiplication involves a significant number of floating-point operations, including both multiplication and addition, which can increase latency during neural network inference. To address this issue, Courbariaux et al. [23] proposed the first vanilla BNN architecture, reported in the next section.

An artificial neural network requires two fundamental processes: forward propagation and backward propagation. Forward propagation involves the information flow from the input layer (located on the left side) to the output layer (located on the right side) as depicted in Figure 2.1. This process is also referred to as model inference, where the network makes predictions based on the given input. On the other hand, backward propagation entails the movement of information from the output layer (right) back to the input layer (left), as shown in Figure 2.1. This process is responsible for fine-tuning the model's weights through a technique known as gradient descent.

Sections 2.1 and 2.2 specifically delve into the functioning of BNN during forward propagation and backward propagation, respectively.

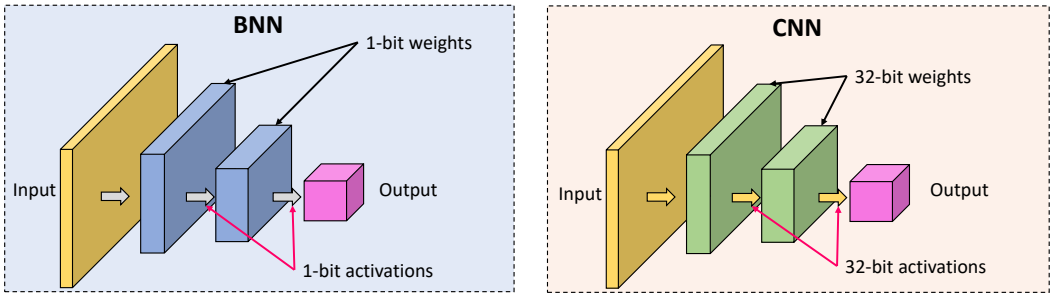


FIGURE 2.1: BNN vs. CNN.

2.1 Forward Propagation

The neural cell serves as a fundamental computational unit in the forward propagation of a neural network. In contrast to the 32-bit Convolutional Neural Network (CNN), the neural cell in the Binary Neural Network (BNN) incorporates binarization step for the input activations (I) and weights (W) prior to the convolution operation. This binarization step aims to represent the floating-point values of the activations and weights using a single bit. Figure 2.2 illustrates the disparity in computation steps within a neural cell along the forward path between the naive BNN and the 32-bit CNN. The sign function (Equation 2.2) is widely adopted to binarize both weights and activations:

$$sign\,(x) = \begin{cases} +1 & if\ x \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2.2}$$

After binarization, the binary weights (Equation 2.3) and activations (Equation 2.4) can be expressed respectively as:

$$B_W = sign\,(W) \tag{2.3}$$

$$B_I = sign\,(I)\,. \tag{2.4}$$

By using Equations 2.3 and 2.4 we can express the binary convolution operation as:

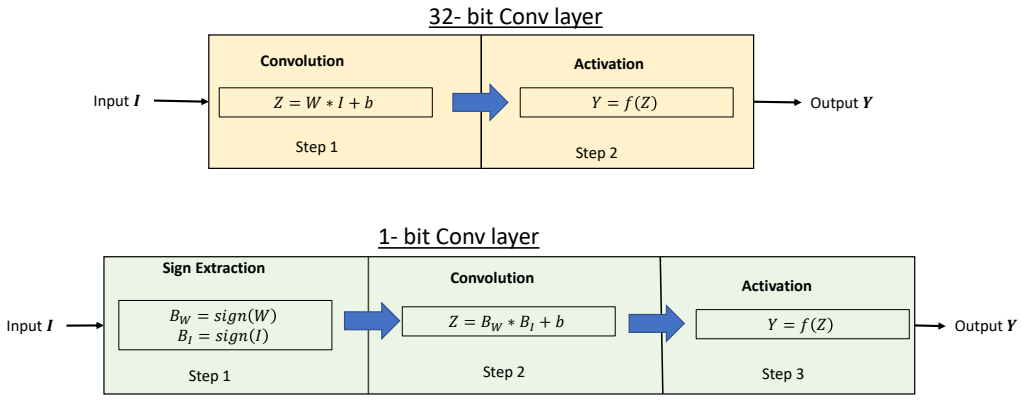


FIGURE 2.2: Internal steps of a 32-bit CNN neuron compared to a naive BNN cell.

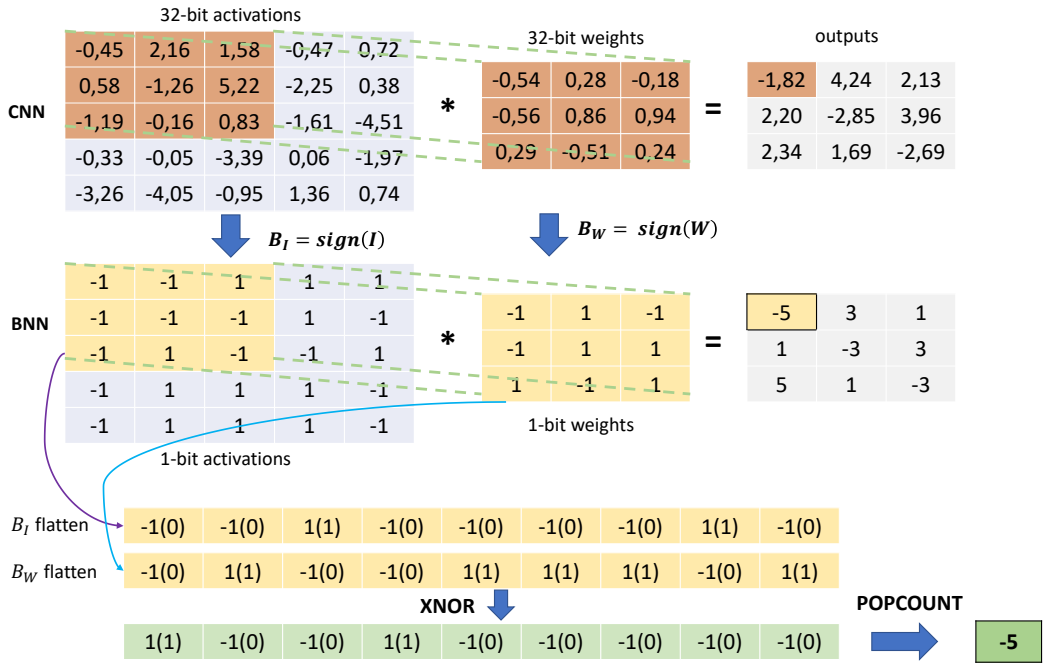


FIGURE 2.3: Naive BNN forward propagation compared to 32-bit CNN. Popcount operation refers to Equation 2.6.

Binary Activations B_I	Binary Weights B_W	XNOR Results %
$-1\,(0)$	$-1\,(0)$	$+1\,(1)$
$-1\,(0)$	$+1\,(1)$	$-1\,(0)$
$+1\,(1)$	$-1\,(0)$	$-1\,(0)$
$+1\,(1)$	$+1\,(1)$	$+1\,(1)$

TABLE 2.1: BNN XNOR operations

$$B_Z = B_I \circledast B_W = sign\,(I) \circledast sign\,(W)\,. \tag{2.5}$$

Because B_I and B_W possible values are $-1, +1$, the output of the binary multiplication corresponds to the result of XNOR, as shown in Table 2.1. This result allows us to replace the expensive matrix multiplication of Equation 2.5, with a lightweight bitwise XNOR and popcount operation. The complete formula that uses XNOR and popcount to compute the binary convolution is expressed in Equation 2.6

$$B_Z = popcount\,(XNOR\,(B_W, B_I)) * 2 - N_{size}, \tag{2.6}$$

popcount returns the number of bits set to 1 and is already available on all modern CPUs (ARM, Intel). N_{size} is the input size of the XNOR operation; in Figure 2.3 is equal to 9. Figure 2.3 shows a comparison of the convolution operation between a standard CNN and a BNN.

2.2 Backward Propagation

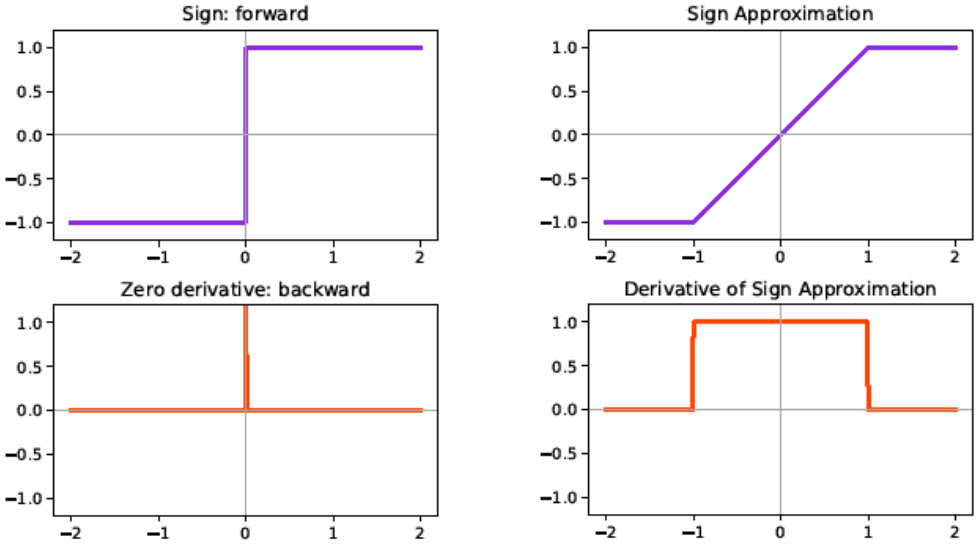
The binarization function 2.2 is not differentiable, and even worse, is zero almost everywhere. So, the traditional gradient descent method based on a backward propagation algorithm would not work for learning the binary weights. To overcome this challenge, Courbariaux et al. [23] utilize the technique known as the straight-through estimator (STE) [133, 9] to learn binary weights during backward propagation, as reported below:

$$STE\,(x) = \begin{cases} 1, & if\ x \geq -1\ and\ x \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.7}$$

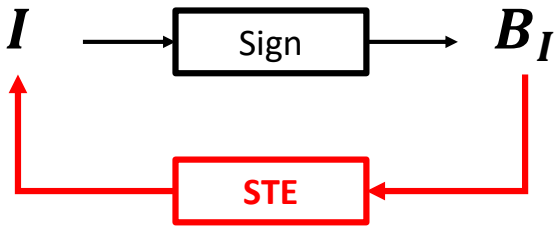
Using Equation 2.7, the gradient of the sign function is ignored and the incoming gradient is passed through as if the derivative of the sign function was the Identity function. the formula 2.7 also clips the gradient values when the input tensor exceeds the range $-1, +1$, as suggested in [58].

Figure 2.4 provides an illustration of the process of learning binarized weights in a Binary Neural Network. During BNN training, each layer's real weights are retained and updated using STE. Upon completion of training, binarized weights are saved, and the real weights are discarded.

In spite of the faster inference speed and smaller weight sizes exhibited by naive Binary Neural Network [23], its accuracy performance is considerably lower compared to full-precision Convolutional Neural Networks in the initial stages. This discrepancy can be attributed to the significant loss of information resulting from parameter binarization, which includes binary activations and binary weights. To mitigate this concern, several optimization approaches have been introduced in recent years. In the subsequent section, we categorize and examine these methods.



$$Sign(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases} \qquad Approx(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x \leq +1 \\ +1, & x > +1 \end{cases}$$



$$\boldsymbol{STE(x)} = \frac{\partial Approx(x)}{\partial x} = \begin{cases} 1, & -1 \leq x \leq +1 \\ 0, & otherwise \end{cases}$$

FIGURE 2.4: Backward pass approximation as suggested in [58].

2.3 Binary Neural Network Optimization

To provide an up-to-date overview of the latest advancements in Binary Neural Network models, multiple optimization and improvement techniques have been proposed. These enhanced solutions can be categorized into five distinct categories, as depicted in Figure 2.5: (1) quantization error minimization, (2) improvement of the loss function, (3) gradient approximation, (4) network topology structure, and (5) training strategy and tricks.

2.3.1 Quantization Error Minimization

Scaling Factor

To mitigate the information loss that occurs during the binarization of 32-bit values to 1-bit values by means of the sign function, Rastegari et al. [113] proposed the use the channel-wise scaling factors for both activations and weights in their XNOR-Net approach. As a result, Equations 2.4 and 2.3 are modified to incorporate these scaling factors, represented by α and β. The revised equations are as follows:

$$B_I = \alpha * sign\left(I\right) \tag{2.8}$$

$$B_W = \beta * sign\left(W\right), \tag{2.9}$$

where α and β are:

$$\alpha = \frac{1}{n} \left\| I \right\|_{L_1}, \tag{2.10}$$

$$\beta = \frac{1}{m} \left\| W \right\|_{L_1}, \tag{2.11}$$

n and m are the number of elements in I and W, respectively. Therefore Equation 2.1 can be re-written for binary operations as:

$$B_Z = B_I \circledast B_W = \left(\alpha * sign\left(I\right)\right) \circledast \left(\beta * sign\left(W\right)\right) = \left(\alpha * \beta\right) * \left(sign\left(I\right) \circledast sign\left(W\right)\right). \tag{2.12}$$

In addition to the approach proposed by Rastegari et al., Bulat et al. [15] proposed an improvement to XNOR-Net, named XNOR-Net++, where the contribution of activations and weights is merged into a single scaling factor as reported in Equation 2.12:

$$B_Z = \left(B_I \circledast B_W\right) \odot \Gamma. \tag{2.13}$$

The authors proposed to construct Γ in four different methods but the one that achieves the best accuracy is the following:

$$\Gamma = \alpha \otimes \beta \otimes \gamma,\ \alpha \in \mathbb{R}^{B_Z}, \beta \in \mathbb{R}^{I_{out}}, \gamma \in \mathbb{R}^{W_{out}}, \tag{2.14}$$

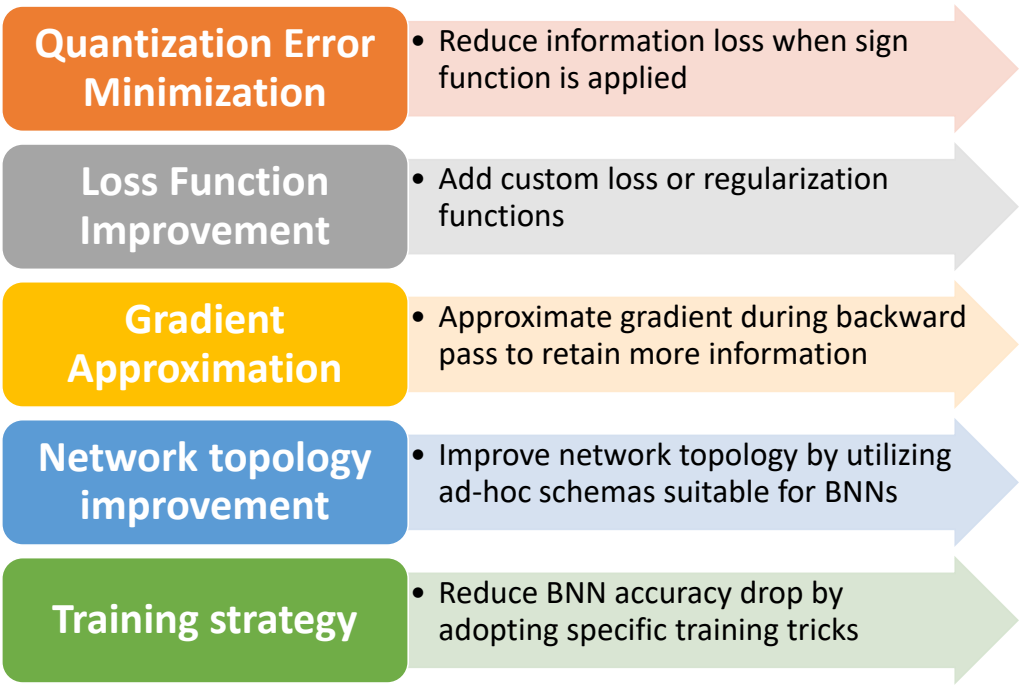


FIGURE 2.5: BNN optimization solutions.

where $\otimes$ represents the outer product and α, β and γ are learnable parameters. Zhao et al. [160] introduced a novel method called DA-BNN (Data-Adaptive Binary Neural Network). This method is designed to generate an adaptive amplitude based on spatial and channel attention to better approximate real-values output features using 1-bit convolutions.

Quantization Function

In addition to using sign functions for activation and weight binarization, several works have introduced alternative methods to quantize parameters belonging to the values $\{-1; +1\}$. These methods include DoReFa-Net [161], UniQ [108], Quantization-Networks [151], and DSQ [41], which proposed k-bit methods for parameter quantization, including binarization. These 1-bit methods offer a different approach to binarizing parameters compared to the use of sign functions. SI-BNN [142] introduced an approach that suggests binarizing activations to the range of $[0; +1]$ and binarizing weights to the range of $[-1; +1]$ as a means to alleviate information loss. Liu et al. [90] proposed ReActNet, which employs the RSign as a binarization function. It incorporates channel-wise learnable thresholds, expressed below:

$$B_I^j = rsign\left(I^j\right) = \begin{cases} +1, & if\ I^j > \alpha^j \\ -1, & if\ I^j \leq \alpha^j, \end{cases} \tag{2.15}$$

where I^j is a real-valued activation on the jth channel and α^j is a per-channel learnable threshold. ReActNet replaces the binary convolution of Equation 2.5 with the following expression:

$$B_Z = (rsign\,(I) \circledast sign\,(W)) \odot \alpha, \tag{2.16}$$

where α represents a channel-wise scaling factor of weights. The derivative $\frac{\partial rsign\left(I^j\right)}{\partial \alpha^j}$ can be computed through the chain rule as:

$$\frac{\partial rsign\left(I^j\right)}{\partial \alpha^j} = -1. \tag{2.17}$$

Instead, the derivative $\frac{\partial rsign\left(I^j\right)}{\partial I^j}$ is approximated considering a piece-wise polynomial function as:

$$react\left(a\right) = \begin{cases} -1 & if\ I < -1 \\ 2I + I^2 & if\ -1 \leq I < 0 \\ 2I - I^2 & if\ 0 \leq I < 1 \\ 1 & \text{otherwise,} \end{cases} \tag{2.18}$$

$$\frac{\partial react\left(I\right)}{\partial I} = \begin{cases} 2 + 2I & if\ -1 \leq I \leq 0 \\ 2 - 2I & if\ 0 \leq I < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

Xu et al. [150] proposed the Rectified Clamp Unit (ReCU), which is a weights standardization method that aims to address the inherent trade-off between minimizing quantization error and maximizing information entropy in BNN. As reported in their work, ReCU is defined as follows:

$$recu\left(W\right) = \max\left(\min\left(W, Q_{(\tau)}\right), Q_{(1-\tau)}\right), \tag{2.20}$$

where $Q_{(\tau)}$ and $Q_{(1-\tau)}$ represent the τ quantile and $1-\tau$ quantile of W, respectively. After applying balancing, weight standardization, and the Rectified Clamp Unit (ReCU) to the weight parameter W, the generalized probability density function (PDF) of W can be expressed as follows:

$$f\left(W\right) = \begin{cases} \frac{1}{2b}\exp\left(\frac{-|W|}{b}\right) & if\ |W| < Q\left(\tau\right) \\ 1 - \tau & if\ |W| = Q\left(\tau\right) \\ 0 & \text{otherwise,} \end{cases} \tag{2.21}$$

where b is computed using the following:

$$b = mean\left(|W|\right), \tag{2.22}$$

$mean\left(|\cdot|\right)$ returns the mean of the absolute values w.r.t. the inputs. The gradient of weights is computed by means of the chain rule and employing the STE approximation for sign extraction 2.7. The derivative of the ReCU activation function with respect to the input I is given by the gradient of the piecewise polynomial function, as reported below:

$$\frac{\partial \mathcal{L}}{\partial sign\left(I\right)} = \frac{\partial \mathcal{L}}{\partial sign\left(I\right)} \cdot \frac{\partial sign\left(I\right)}{\partial I} \approx \frac{\partial \mathcal{L}}{\partial sign\left(I\right)} \cdot \frac{\partial F\left(I\right)}{\partial I}, \tag{2.23}$$

where

$$\frac{\partial F(I)}{\partial I} = \begin{cases} 2+2I & if \ -1 \leq I < 0 \\ 2-2I & if \ 0 \leq I < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.24}$$

Lin et al. [85] proposed SiMaN, an angle alignment objective, known as sign-to-magnitude, to constrain weight binarization to the values of $[0; +1]$. Tu et al. [134] introduced AdaBin, an optimal binary set of weights and activations for each layer.

Activations/weights distribution

In contrast to directly optimizing the binarization process in the convolution layer, several approaches directly focus on optimizing and reshaping activations and weights distribution before the binarization step, such as Qin et al., [112], Shen et al., [123], Yang et al., [152], Lin et al., [84], Kim et al., [67], and Li et al., [81]. Additionally, Hou et al., [56] proposed a proximal Newton algorithm that directly minimizes the loss w.r.t. the binarized weights. Li et al. [82] proposed a recursive binary quantization technique to mitigate information loss. Wang et al. [145] suggested utilizing a reinforcement learning model to explore channel-wise interactions by applying channel-wise priors on the intermediate feature maps using the interacted bitcount function. Han et al. [47] introduced a training approach where the binarization function predicts the binarized weights through supervision noise learning. He et al. [53] constructed a proxy matrix to reduce weights quantization error by preventing binary regularizations directly on the latent floating-point values.

2.3.2 Loss Function Improvement

In order to mitigate the accuracy drop observed in BNNs compared to real-valued networks, several methodologies have been proposed. Essentially, all proposed solutions try to regulate the activation distribution, expressed as:

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + \lambda \mathcal{L}_{DL}, \tag{2.25}$$

where $\mathcal{L}_{\mathcal{CE}}$ represents the standard cross-entropy loss for deep neural network training, $\mathcal{L}_{\mathcal{DL}}$ is the distribution loss for acquiring the appropriate binarization, and λ serves to balance the impact of the two types of losses. By incorporating this supplementary loss, the trained neural network can effectively overcome the aforementioned challenges as reported below. For instance, Tang et al. [132] introduced the How-to-Train approach, where the usual L_2 regularization function (which encourages the weights to be closed to zero) is replaced by ad-hoc regularization term that forces the weights to be bipolar. Darabi et al. [24] developed the BNN-RBNT technique by introducing a new regularization function that encourages weights to be closed to binary values adding trainable scaling factors to the regularization term. Ding et al. [29] introduced BNN-DL which proposes to use the distribution loss to regularize the activations. Additionally, Xu and Cheung [149] put forth the CCNN approach by adding a L_2 regularization term to weights scaling factors. Moreover, Gu et al. [44] proposed BONN adopting multiple bayesian losses to simultaneously optimize the network in both continuous and discrete spaces. Lin et al. [84] introduced RBNN which considers the angle between the full-precision and binarized weight vectors within the loss function. Shang et al. [122] presented LCR, proposing to keep the Lipschitz continuity as a regularization term, improving the model robustness.

2.3.3 Gradient Approximation

The sign function's (Equation 2.2) derivative output is zero, which causes the weights to remain unchanged during back-propagation. One way to approximate sign gradients is by utilizing STE (Equation 2.7). However, STE fails to update weights near the boundaries of -1 and +1, affecting the back-propagation's updating ability. To address this issue, several methodologies have been proposed. For example, Sakr et al. [120] introduced GB-Net, which employs true gradient-based learning with parametrized clipping functions (PCF) and scaled binary activation functions (SBAF) to train BNN. The SBAF is expressed as

$$\text{SBAF}\,(x) = \alpha \times \mathbb{1}_{x>0}, \tag{2.26}$$

where α represents a scaling parameter and $\mathbb{1}_{x>0}$ denotes the indicator function. Instead, the PCF is expressed as:

$$\text{PCF}\,(x) = \min\left(\max\left(\frac{x}{m} + \frac{\alpha}{2}, 0\right), \alpha\right), \tag{2.27}$$

where m is the slope parameter. Similarly, Darabi et al. [24] proposed BNN-RBNT, which utilizes a backward approximation based on the sigmoid function. The proposed binarization function is expressed as follows:

$$\text{SS}_\beta\,(x) = 2\sigma\,(\beta x)\,[1 + \beta x\,\{1 - \sigma\,(\beta x)\}], \tag{2.28}$$

where $\sigma\,(x)$ is the sigmoid function and β regulates the rate at which the activation function approaches the values -1 and $+1$ as the input increases.

Liu et al. [88] presented Bi-Real Net, which uses a polynomial steps function to approximate the forward sign function as follows:

$$bireal\,(I) = \begin{cases} -1 & if\ I < -1 \\ 2I + I^2 & if\ -1 \leq I < 0 \\ 2I - I^2 & if\ 0 \leq I < 1 \\ 1 & \text{otherwise} \end{cases} \tag{2.29}$$

$$\frac{\partial bireal\,(I)}{\partial I} = \begin{cases} 2 + 2I & if\ -1 \leq I \leq 0 \\ 2 - 2I & if\ 0 \leq I < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.30}$$

The forward propagation of Bi-Real Net is the same as Equation 2.5. Xu and Cheung [149] introduced a derivation estimator to approximate their binarization function in CCNN. In particular, they adopted the sign 2.2 and the unit step (reported below) function for weight and activation binarization respectively.

$$H\,(x) = \begin{cases} 1 & if\ x \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.31}$$

The authors of [149] proposed as the derivative of the sign function a piece-wise linear function whose range is in $[-0.5; +0.5]$ as follows:

$$\frac{\mathrm{d}sign\,(x)}{\mathrm{d}x} \approx \begin{cases} 4 - 8\,|x| & if\ -0.5 \leq x \leq +0.5 \\ 0 & \text{otherwise.} \end{cases} \tag{2.32}$$

The derivative of H(x) is obtained through a long-tailed higher-order approximation function with a range in $[-1;+1]$. The piecewise function proposed is the following:

$$\frac{\mathrm{d}H(x)}{\mathrm{d}x} \approx \begin{cases} 2-4|x| & if \ -0.4 \leq x \leq +0.4 \\ 0.4 & if \ 0.4 < |x| \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.33}$$

Qin et al. [112] and Lin et al. [84] independently designed a dynamic gradient estimator that adjusts the gradient approximation during the training process in IR-Net and RBNN, respectively. Specifically, IR-Net proposed a binarization function defined as follows:

$$\begin{gathered} g(x) = k * tanh(kx) \\ k = \max\left(\tfrac{1}{t}, 1\right), \ T_{min} = 10^{-1}, \ T_{max} = 10^{2} \\ t = T_{min} * 10^{\frac{i}{N} * \log\left(\frac{T_{max}}{T_{min}}\right)}, \end{gathered} \tag{2.34}$$

where i is the current epoch on training and N is the total number of training epochs. Wang et al. [142] developed SI-BNN, which utilizes a gradient estimator with two trainable parameters on top of STE. Kim et al. [66] quantitatively analyzed differentiable approximation functions and proposed using the gradient of the smoothed loss function to estimate the gradient in BinaryDuo. Xu et al. [148] introduced FDA, which utilizes a combination of sine functions in the Fourier frequency domain to estimate the gradient of sign functions. Figures 2.6 and 2.7 provide a summary of BNN techniques that propose gradient approximation methodologies.

2.3.4 Network Topology Structure

Binarization involves converting activations and weights to the set $\{-1;+1\}$. This process effectively regularizes the data, leading to an unexpected change in the data distribution post-binarization. Adjusting the network structure offers a promising solution for accommodating these distribution changes. A simple reordering of the layers in the network can enhance the performance of the binary neural network. Alizadeh et at. [2] noted that most binarization studies have repositioned the pooling layer. Placing the pooling layer immediately after the convolutional layer helps prevent information loss resulting from max pooling after binarization. Experiments have demonstrated a significant improvement in accuracy due to this reordering. In addition to the pooling layer, the placement of the batch normalization layer also significantly impacts the stability of the training in binary neural networks. Wang et al. [143] and Cai et al. [18] inserted a batch normalization layer before all quantization operations to rectify the data. This transformation ensures that the quantized input adheres to a stable distribution, often close to Gaussian, thus maintaining reasonable mean and variance values and leading to a smoother training process.

In contrast with previous approaches, Liu et al. [88] proposed to directly modify the network structure instead of adding new layers. Bi-Real Net connects the full-precision feature maps across layers to the subsequent network, effectively adjusting the data distribution through structural transformation. Mishra et al. [98] devised Wide Reduced-Precision Networks (WRPN), which increase the number of filters in each layer, thereby reforming the data distributions.

Zhu et al. [163] proposed the Binary Ensemble Neural Network (BENN) that leverages the ensemble method to fit the underlying data distributions. Liu et al. [86]

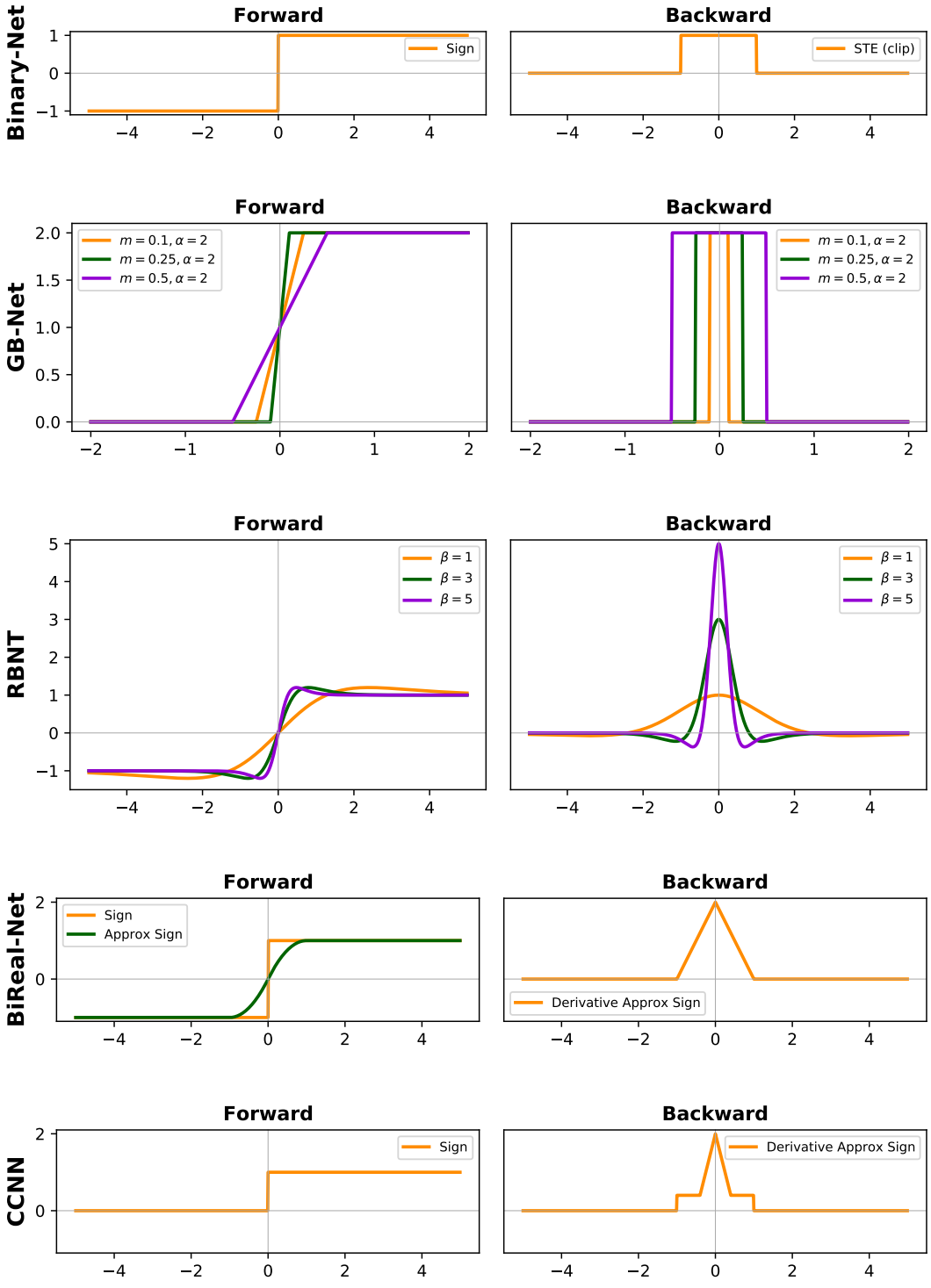


FIGURE 2.6: Shapes of forward and backward passes.

proposed circulant filters (CiFs) and a circulant binary convolution (CBConv) to enhance the capacity of binarized convolutional features, with circulant backpropagation (CBP) proposed to train the structures. Additionally, Shen et al. [123] appended a gated residual to compensate for information loss during the forward process.

Bulat et al. [13], Kim et al. [65], Zhu et al. [162], and Bulat et al. [14] used designed

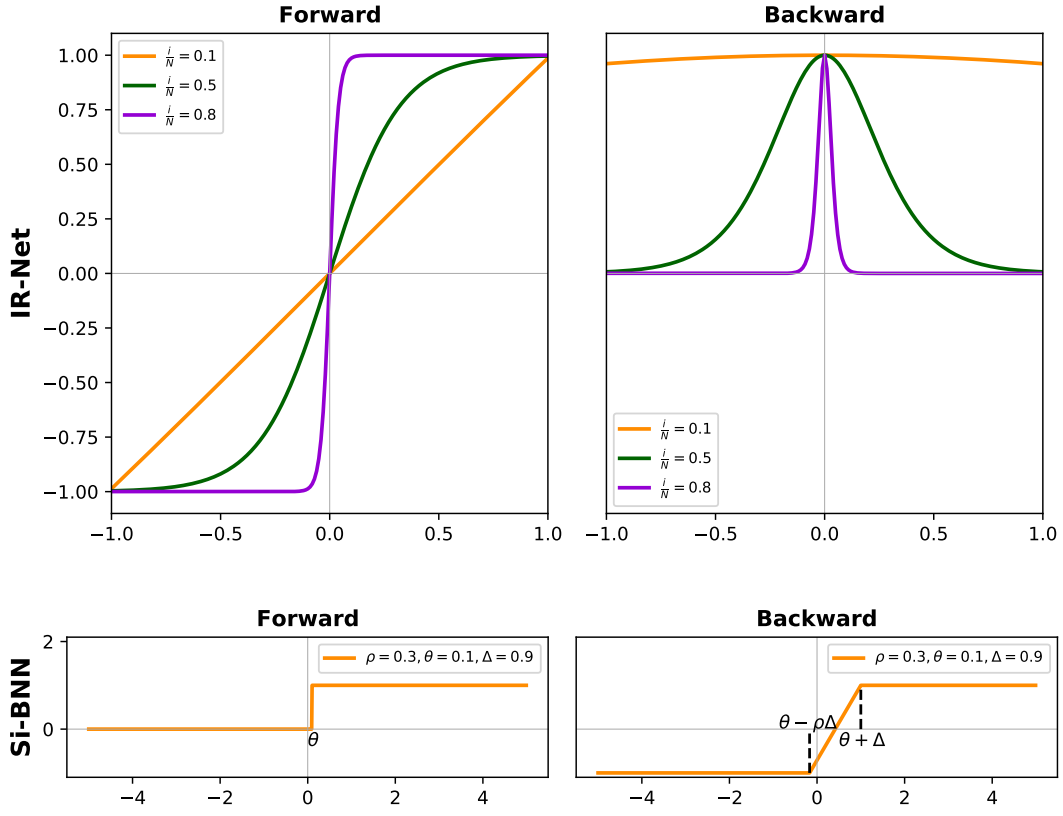


FIGURE 2.7: Shapes of forward and backward passes.

NAS methods to search for BNNs architectures to compare accuracy performance with other BNNs with similar model sizes that are binarized from classic networks such as ResNet.

Inspired by MobileNet-v1, Phan et al. proposed MoBiNet-Mid [110] and Binarized MobileNet [109], new BNN architectures with high accuracy performance, fewer operations, and lighter model sizes. Bethge et al. [10](MeliusNet) and Liu et al. [90](ReActNet) designed new architectures that can beat the accuracy rate of full-precision lightweight MobileNet with fewer OPs computation cost.

Chen et al. [21] replaced the batch normalization (BatchNorm) with scaling factor, and ReActNet without BatchNorm still has a competitive classification top-1 accuracy on the ImageNet dataset. Zhang et al. [159] extended ReActNet's topology by re-balancing the blocks of networks and designing a two 1-bit activation scheme to improve feature learning, with a competitive top-1 prediction result on the ImageNet dataset compared to full precision MobileNet-v2. Eventually, Redfern et al. [115] designed a customized structure for ImageNet classification with a lower model size compared to MeliusNet and ReActNet.

Zhang et al. [158] suggested establishing a connection between the unquantized input activations and the output of the Conv and BN combination with a shortcut. Unlike ReActNet [90], the authors recommended incorporating the Dynamic PReLU (DPReLU) after the residual addition, followed by multiplying the output by the result of a Sqeeze-and-Excitation (SE) block, 4-bits quantized.

Shi et al [124] introduced a replaceable convolution module called RepConv. This module enhances feature maps by replicating the input or output along the channel dimension by β times, without incurring additional costs on the number of parameters and convolutional computation. Falkena et al. [32] proposed to substitute the $sign\,(\cdot)$ binarization function, which is an information bottleneck, with a learnable activation binarizer (LAB) that enables the network to learn a fine-grained binarization kernel per layer. Lee et al. [76] proposed a BNN design named INSTAnce-aware threshold BNN (INSTA-BNN), which dynamically controls the quantization threshold in an input-dependent or instance-aware manner. The main representative set of BNN structures is reported in Figure 2.8.

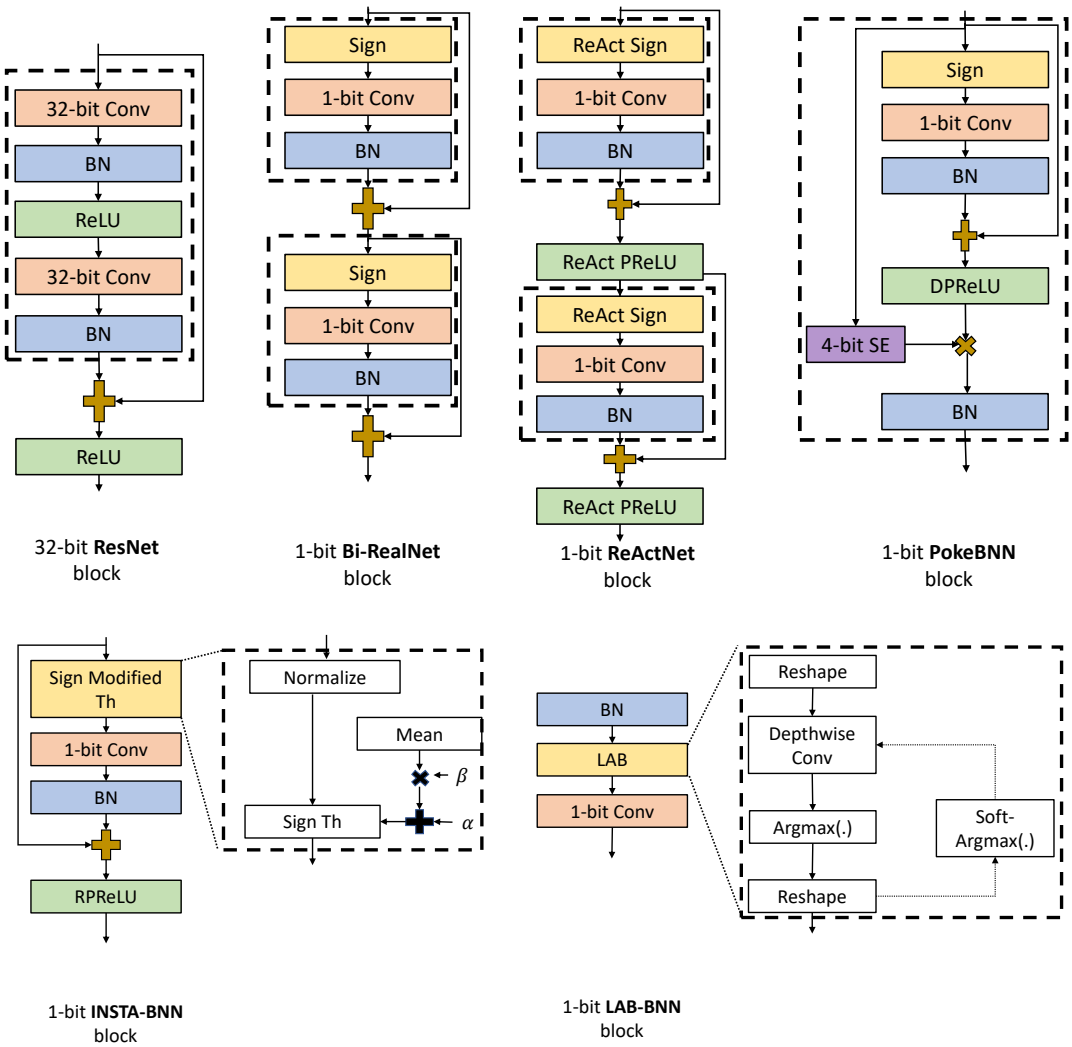


FIGURE 2.8: Representative BNN block structures.

2.3.5 Tricks and strategy for training Binary Neural Networks

There are several different training schemes and techniques that can impact the final accuracy of Binary Neural Networks (BNNs). For instance, Dong et al. [30] proposed a stochastic quantization algorithm that gradually trains and quantizes BNN to compensate for quantization error.

Liu et al. [88] used trainable parameters based on pre-trained real value networks and replaced the ReLU activation function with the Hard Tanh function. Tang et al. [132] replaced ReLU activation with the PReLU function and explored the influence of learning rate on the accuracy of the final trained BNN.

Xu et al. [147] introduced filter pruning for BNN. Diffenderfer et al. [28] designed a scheme to learn highly accurate BNNs simply by pruning and randomly quantizing weighted full precision CNN, inspired by the Lottery Ticket Hypothesis.

Wang et al. [145] trained simultaneously a reinforcement graph model and a BNN to alleviate binarization inconsistency. Martinez et al. [95] designed a two-step training strategy that applies the transfer teaching method by learning from a real value pretrain network.

Alizadeh et al. [2] explored the impact of pooling, optimizer, and learning rate initialization for training BNN. Helwegen et al. [54] and Pham et al. [108] separately proposed new optimizers for BNN training. Liu et al. [89] investigated and designed a new training scheme based on Adam optimizer to improve Real-to-Bin [95] and ReActNet's [90] performance. Kim et al. [66] proposed a two-stage training scheme to decouple a ternary-activation network into a two-binary-activation BNN network. Ajanthan et al. [1] applied mirror descent to optimize BNN's optimizer.

Laydevant et al. [73] and Wang et al. [141] explored BNN training directly on the edge, using respectively Equilibrium Propagation and low-memory/energy training schemes. Livochka et al. [91] proposed a transfer training and initialization scheme for BNN using the stochastic relaxation approach to improve accuracy on the small-scale CIFAR-10 [71] dataset. Wang et al. [144] proposed a new method to further compress and accelerate BNN in FPGA (more detailed solutions on this can be found in [78, 128]) based on the observation of the binary kernels in BNN. These different training strategies and techniques contribute to the advancement and improvement of Binary Neural Networks, each offering unique approaches to address the challenges and limitations of BNNs.

Chapter 3

Improving BNN Inference

In this chapter, we describe our contributions to the optimization of the inference forward pass for a binary neural network. Specifically, the chapter introduces a new method (section 3.1) to reduce the inference time of a BNN by removing most of the floating-point computation. Later, in section 3.2, we present our solution used to fully-binarize a neural network by binarizing also the input layer. The method presented in Section 3.1 has been presented at *TinyML EMEA Innovation Forum 2022*, while the research detailed in Section 3.2 has been published in [137].

3.1 Optimizing data-flow in Binary Neural Networks

The binarization approach, introduced in Chapter 2, has shown that a CNN model can be quantized to 1-bit thus achieving a remarkable speedup compared to the full precision network. Unfortunately, as reported in Section 2.3.3, the aggressive quantization can make BNNs less accurate than their full-precision counterparts. Many techniques have been proposed to fill the gap between 1-bit and 32-bit networks. To prevent the generation of feature maps of lower quality and capacity, produced by 1-bit layers, a combination of binary and floating-point layers is usually adopted. Unfortunately, each time a binary layer is connected to a floating-point one, the efficiency of the pipeline is compromised by input/output layer data type conversion. In addition, the internal parallelism of a binary layer depends on the encoding of the accumulator, which is often maintained at 32 bits to prevent overflow. To overcome the previous issue, we propose several optimizations that allow training a BNN with an inter-layer data width of 8 bits. Moreover, even if 8-bit quantization of weights and activations of a neural network is a well-known topic, as reported in [62], in BNNs, 8-bit quantization is not widespread and the Batch Normalization (BN) layer is usually executed in floating point arithmetic using off-the-shelf inference engines [155, 8]. In contrast, we show that the quantization of the BN layer and the reduced width size of the accumulator inside the binary operator can lead to a substantial speed-up of the 1-bit layer (binary operation + BN).

Most prior work on BNNs emphasizes overall network accuracy; in contrast, we aim to preserve initial accuracy while improving efficiency. Besides many efforts to develop more efficient and accurate architectures, a few works have provided benchmarks on real devices such as ARM processors. Based on the analysis provided in [8], the fastest inference engines for binary neural networks, with proven benchmarks (Section 4 of [8]), are LCE and DaBNN. Our contributions, (graphically highlighted in Figure 3.1a and 3.1b) reported in this section, can be summarized as follows:

- A novel training scheme is proposed to improve the data flow in the BNN pipeline (Section 3.1.1); specifically, we introduce a clipping block to shrink the data width from 32 to 8 bits while simultaneously reducing the internal accumulator size.
- we provide (Section 3.1.1) an optimization of the Batch Normalization layer when executed after a binary operation that decreases latency and simplifies deployment.
- we optimize the Binary Direct Convolution method for ARM instruction sets (Section 3.1.1).

To prove the effectiveness of the proposed optimizations, in Section 3.1.2 we provide experimental evaluations that show the speed-up relative to state-of-the-art BNN engines like LCE [8] and DaBNN [155].

3.1.1 Data-Flow Optimizations

As illustrated in Figs. 3.1a and 3.1b (a), the most commonly used BNN architectures (e.g., VGG and ResNet) have four essential blocks in each convolution/fully-connected (CONV/FC) layer: sign (binarization), XNOR, popcount and Batch Normalization (BN). Since the weights, inputs, and outputs are all binary, the traditional multiply-and-accumulate operation is replaced by XNOR and bit counting (i.e., popcount). XNOR and popcount are usually fused to improve efficiency. The use of Batch Normalization after each binarized layer is very important in BNNs because it makes the optimization landscape significantly smoother; this smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training. Figures 3.1a and 3.1b (b and c) point out our proposed BNN optimizations during training and inference. Before discussing them in detail, we show the data-flow bottlenecks that affect existing solutions and then describe how to reduce them.

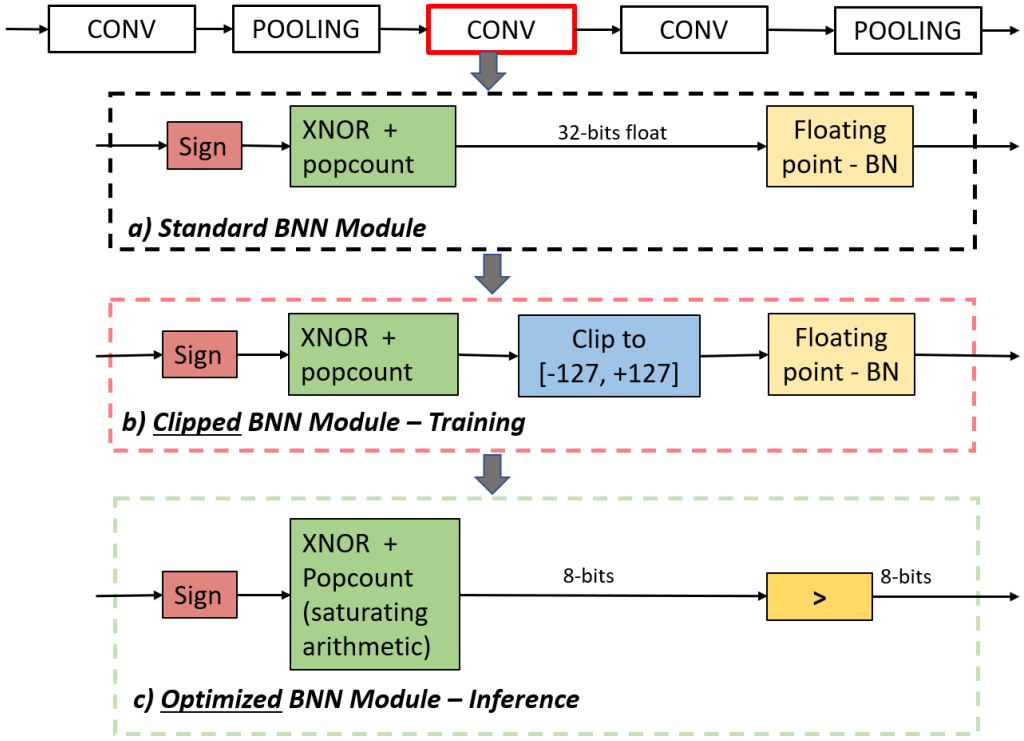
In Figures 3.2a and 3.2b we report an example of binary convolutional layer outputs for a VGG and a ResNet model. The ranges of activation values after popcount (green histograms) exceed the interval $[-128; +127]$ [1], so adopting an 8-bit encoding would lead to overflow. To prevent such a data loss, most of the existing BNN frameworks (including [8, 155]) encode such data in a 32-bit floating point. On the other hand, the ranges of values after BN (red histograms in Figure 3.2) are more limited.

We propose to accumulate the popcount output with 8-bit integers (using saturation arithmetic) through a two-stage training procedure, which is designed to preserve model accuracy. In the next subsection, we show how to apply this technique to VGG and ResNet models.

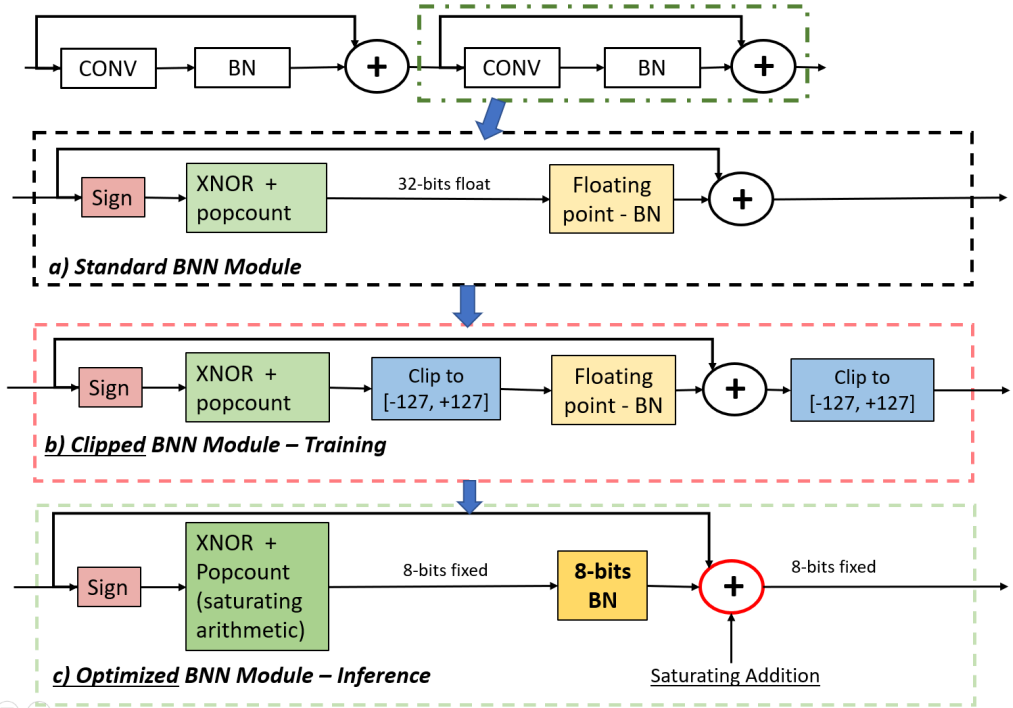
Two-stage Clipping

Our training procedure selectively executes or skips a clipping operation at each binary layer (row *b* of Figs. 3.1a and 3.1b, blue blocks). A two-stage training method is introduced to avoid accuracy loss when clipping is enabled: during the first warm-up stage, the model is trained without any range constraints, while in the second stage (details are reported in Algorithm 1) the network is trained with the clipping

[1] We consider the symmetric quantization interval $[-127; +127]$ because this choice enables a substantial optimization opportunity, as reported in Appendix B of [62].

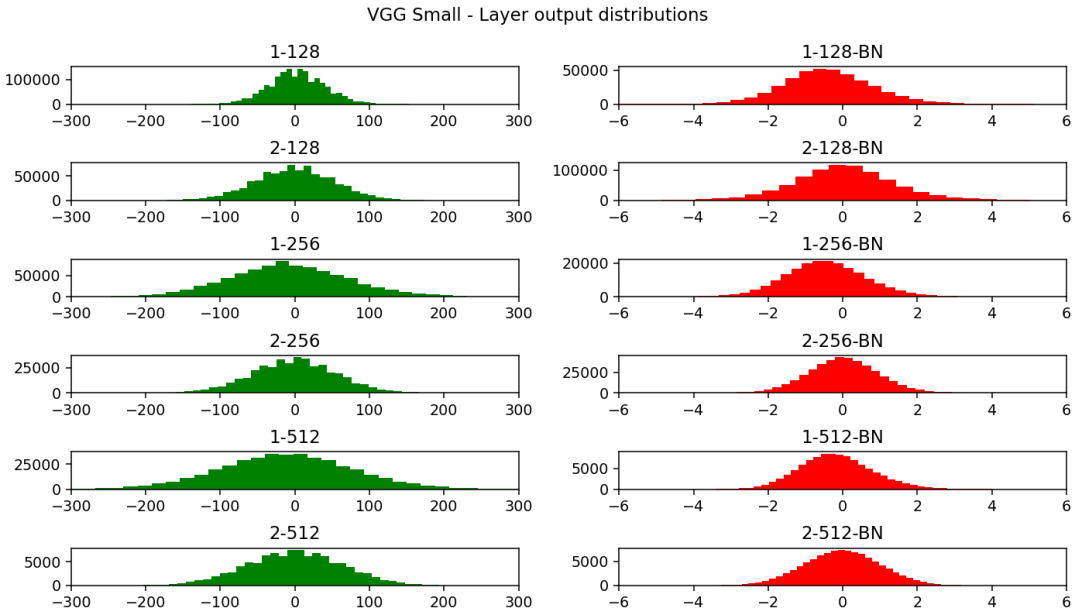


(A) VGG style block.

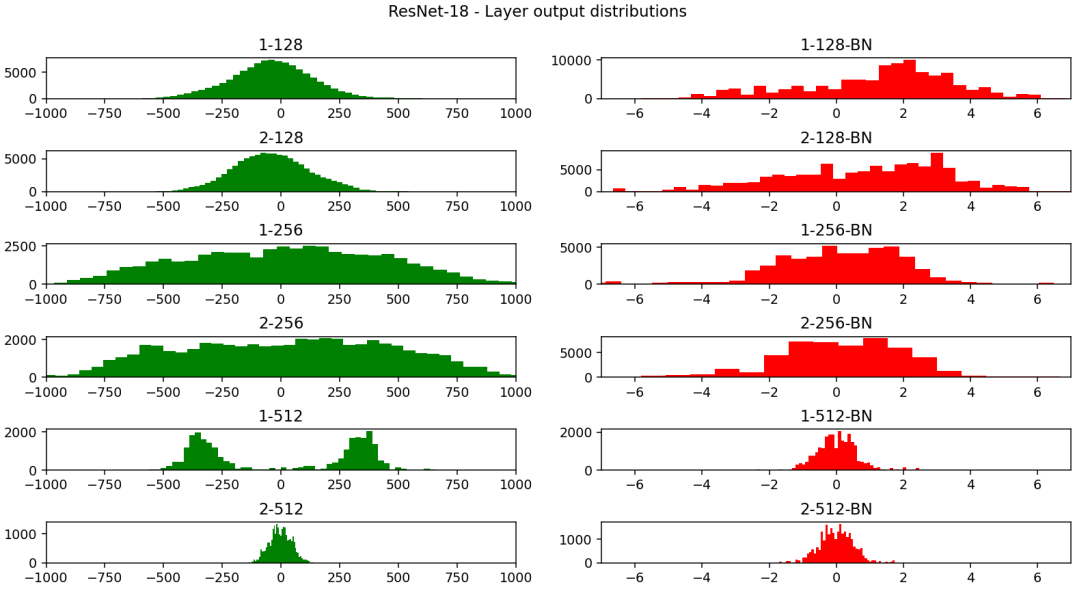


(B) ResNet style block.

FIGURE 3.1: a) Standard BNN blocks are used in [113] and [88]. b) BNN block with output convolution clipping used during training. c) Optimized BNN block adopted during inference. Popcount operation is performed using saturation arithmetic to keep the data width to 8 bits at inference time. BN is replaced by a comparison in case a, while in b BN is 8-bit quantized.



(A) VGG Small output layers



(B) ResNet output layers

FIGURE 3.2: Example of output distributions after binary convolution. a, refers to a VGG style network while b to a ResNet architecture. Green shows the distribution before the BN layer and red afterward.

Algorithm 1 Second stage training procedure for BNNs

Input: The full-precision weights W; the input training dataset
Output: BNN model with convolution output clipped
Initialize network weights W
repeat
{Forward Propagation}
for $l = 1$ **to** L **do**
Binarize floating point weights: $W^l_{bin} = sign\left(W^l\right)$
Binarize floating point features of previous layer: $F^{l-1}_{bin} = sign\left(F^{l-1}\right)$
Compute binary convolutions features: $F^l_{out} = F^{l-1}_{bin} * W^l_{bin}$
Clip F^l_{out} values to interval $[-127; +127]$ with: $F^l_{out\,clipped} = max\left(min\left(127, F^l_{out}\right), -127\right)$
Perform Batch Normalization: $BN\left(F^l_{out\,clipped}\right) = \gamma^l \frac{F^l_{out\,clipped} - \mu^l}{\sigma^l} + \beta^l$
end for
{Backward Propagation}
for $l = 1$ **to** L **do**
Compute gradients based on the binarization weights W^l_{bin}, clipped convolutions $F^l_{out\,clipped}$ and batch normalization output $BN\left(F^l_{out\,clipped}\right)$
Update full-precision weights W^l
end for
until Convergence

block enabled. Based on the high accuracy reached at the end of the first training stage, in the second training stage the model better tolerates clipping 8-bit quantization; we experimentally found that this approach preserves the accuracy of a model that does not contain clipping.

Batch Normalization Optimization

The BN layers after the clipping are also optimized/8-bit quantized to further increase the data flow of the inference pipeline. The Batch Normalization layer scales and shifts the output of the CONV/FC layer as follows:

$$BN\left(F^l_{out}\right) = \gamma \frac{F^l_{out} - \mu}{\sigma} + \beta \tag{3.1}$$

where $\gamma,\ \mu,\ \sigma\ and\ \beta$ are learned parameters and F^l_{out} is the output feature of layer l that is the input of BN function.

The BN optimization depends on the network model: VGG or ResNet. In both cases, we show that it is possible to keep the inter-layer data type to 8-bit with appropriate changes to the binary layer structure.

- **VGG style block**. When the BN layer is inserted in a pipeline similar to Figure 3.1a, where the following block is still binary, the BN operation can be simplified by replacing multiplication and division in Equation 3.1 with a simple comparison with a threshold τ. The simplification of Equation 3.1 leads to:

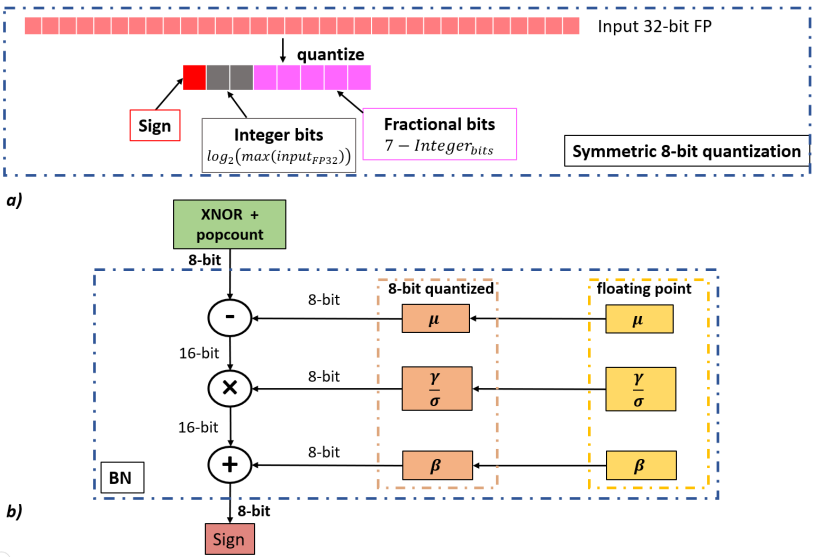


FIGURE 3.3: **a)**: 8-bit symmetric quantization procedure that reserves fractional/integer bits based on the range of input 32-bit floating point values. **b)**: implementation of the BN layer with 8-bit quantization using an internal 16-bit representation to preserve accuracy.

$$sign\left(BN\left(F^l_{out}\right)\right) = \begin{cases} +1 & if \;\; BN\left(F^l_{out}\right) \geq 0 \\ -1 & otherwise \end{cases}$$

$$\gamma \frac{F^l_{out} - \mu}{\sigma} + \beta \geq 0 \Rightarrow \tau \doteq \mu - \beta \frac{\sigma}{\gamma} \tag{3.2}$$

$$sign\left(BN\left(F^l_{out}\right)\right) = \left\{ \begin{matrix} +1 \; if \; F^l_{out} \geq \tau \; else \; -1 \;\; \left(when \; \frac{\gamma}{\sigma} \geq 0\right) \\ -1 \; if \; F^l_{out} \leq \tau \; else \; +1 \;\; \left(when \; \frac{\gamma}{\sigma} < 0\right) \end{matrix} \right\}$$

The threshold τ of Equation 3.2 can be computed offline and easily quantized to 8 bits in order to exploit the output features of layer l. Therefore, when multiple BNN modules are stacked, Batch Normalization can be replaced by a threshold comparison according to Equation 3.2. Even if BN can be replaced with a threshold comparison, 8-bit data flow is still important because it allows to accumulate the binary xnor and popcount operations directly on 8-bit using saturation arithmetic instead of the standard 32-bit.

- **ResNet style block**. When a BNN block is placed in a ResNet style pipeline, followed by an addition operator, Figure 3.1b, the BN layer can be executed with both scaling and bias factors to 8 bits. As reported in Figure 3.3, the internal data representation of a quantized BN layer is expanded to 16-bit to preserve accuracy during quantization but the input/output data type still remains within 8 bits. The iterative quantization procedure we adopted is symmetric and keeps unaltered the zero point representation, as reported in Algorithm 2. The procedure iterates over the BN floating-point layers and, for each one: computes the quantization scale, quantizes, freezes the weights, and retrains the remaining layers.

Algorithm 2 Procedure to quantize the BN floating point layers in a BNN model where convolution output is clipped.

Input: The full-precision weights W; the input training dataset
Output: BNN model with BN float layers replaced by 8-bit quantized version
for $l = 1$ **to** L **do**
  if l is BN floating point **then**
    Compute range of features F^l_{out} as: $Range^l = \left[min\left(F^l_{out}\right); max\left(F^l_{out}\right)\right]$ {l^N is the number of layer variables (4 for BN)}
    for $h = 1$ **to** l^N **do**
      Compute range of variable w^l_h as: $Range_{w^l_h} = \left[min\left(w^l_h\right); max\left(w^l_h\right)\right]$ {1 bit is reserved for sign}
      Compute bits used for range as: $RangeBits_{w^l_h} = clip\left(\left\lceil log_2\left(max\left(abs\left(Range_{w^l_h}[0]\right), \left(Range_{w^l_h}[1]\right)\right)\right)\right\rceil, 0, 15\right)$
      Compute number of bits used for fractional part as: $FracBits_{w^l_h} = 15 - RangeBits_{w^l_h}$
    end for
    Select the Integer part (range) for all N weights as: $RangeBits_{w^l} = max\left(RangeBits_{w^l_h}\right)$
    Select the Fractional part for all weights as: $FracBits_{w^l} = 15 - RangeBits_{w^l}$
    for $h = 1$ **to** l^N **do**
      Add quantization noise to floating point weights w^l_h as: $w^l_{q_h} = \frac{1}{FracBits_{w^l}} round\left(2^{FracBits_{w^l}} * w^l_h\right)$
      Replace w^l_h with $w^l_{q_h}$
    end for
    Freeze w^l weights and retrain the model {Export the quantized weights of layer l for deployment}
    for $h = 1$ **to** l^N **do**
      $w^l_{quantized_h} = round\left(2^{FracBits_{w^l}} * w^l_{q_h}\right)$
    end for
  end if
end for

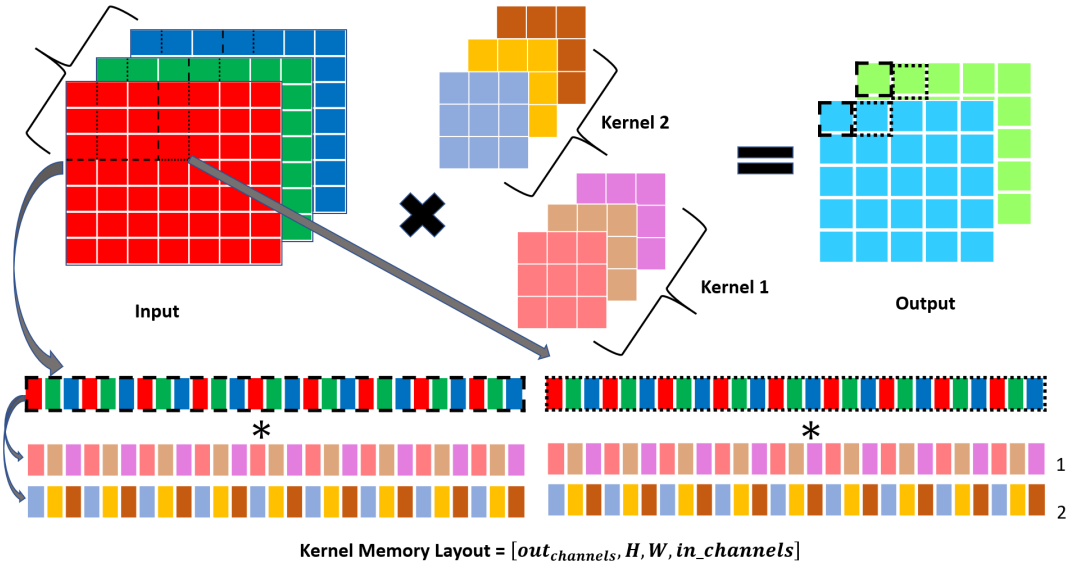


FIGURE 3.4: The 7×7 input image with 3 different channels (denoted by color) is convolved with two separate kernels to obtain a 5×5 output with two output channels. To better exploit the SIMD 128-bit registers a different memory layout for kernel is devised: $\left[out_{channels},\ H_{filter},\ W_{filter},\ in_{channels}\right]$.

Binary Direct Convolution optimization on ARM

The GEneral Matrix Multiplication (GEMM) is a widely adopted method to efficiently implement convolutions. However, as reported in [156], the GEMM approach increases the memory footprint of the model, making a model's porting to an embedded device more difficult. Furthermore, GEMM routines are not always optimized for convolutions on ARM devices, in particular ARMv8 (64-bit ARM architecture) and its relevant operations such as *vcount* and *addv*.

vcount takes an N-byte vector as input and outputs an N-byte vector containing the number of 1*s* present in each input byte. *addv* takes an N-byte vector as input and outputs the sum of the N bytes as one single value.

Inspired by [156] and [155] we propose a hybrid direct binary convolution (see Figure 3.4) that uses both the *addv* instruction and the common *add* operations. The binary convolution is usually composed of three different steps: binarization/bit-packing, padding, and convolution. [155], executes these steps in a sequential way. In contrast, we devise a more cache-friendly approach that collapses the previous steps in one operation executed with tiling. We also devise a different kernel memory layout that better fits ARMv8 SIMD processing instructions, as illustrated in Figure 3.4.

The implementation details of our binary convolution are reported in Figure 3.5. The operation *Extract sign bit* executes the binarization, bit-packing, and padding. Then, the (bit-wise) XNOR output is consumed by the popcount operation (*vcnt 8-bit wise*, *add* and *addv*). On the ARM architecture, the latter can be implemented with *vcount* and a sequence of additions (*addv* instructions). We decided to implement several pair-wise additions and only a final *addv* instruction (which is more expensive). The entire convolution process does not provide intermediate outputs but instead processes the input data as a whole. It is worth noting that the clipping operation can be obtained for free on ARM devices by exploiting its saturation arithmetic; all the

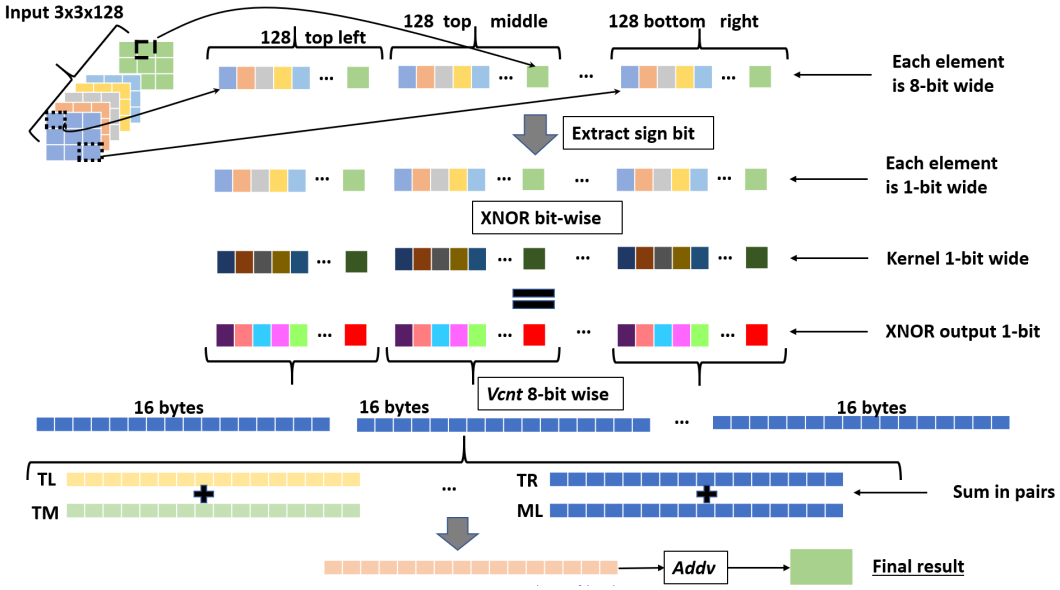


FIGURE 3.5: The $3 \times 3 \times 128$ input patch is convolved (XNOR + popcount) with one kernel through the Extract sign bit, XNOR, and then popcount operations. Popcount is performed using *vcnt*, summing in pairs the *vcnt* output, and the last step uses the *addv* operation. TL (top left), TM (top middle), TR (top right), and ML (middle left) indicate the position of elements inside the 3×3 patch.

addition operations (*add* and *addv*) can be limited to the fixed range $[-127; +127]$ by simply adding the postfix q to the operations and executing max to avoid -128 value.

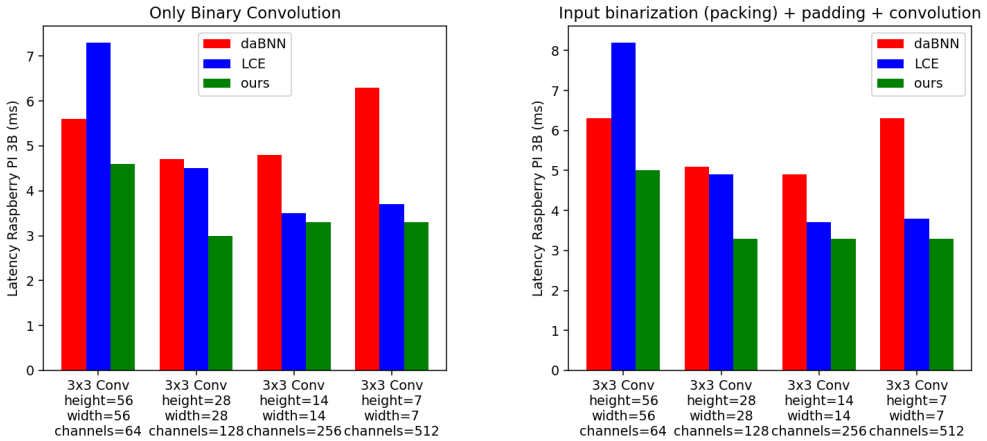
3.1.2 Experimental Results

In this section, we first evaluate the efficiency result of our approach compared to the state-of-art BNN frameworks such as LCE and DaBNN; the comparison is carried out on real hardware devices like Raspberry Pi Model 3B and 4B with 64-bit OS. Then, we present various accuracy benchmarks of the proposed two-stage training procedure focusing on CIFAR-10, SVHN, and ImageNet, and on two different architectures: VGG and Resnet-18.

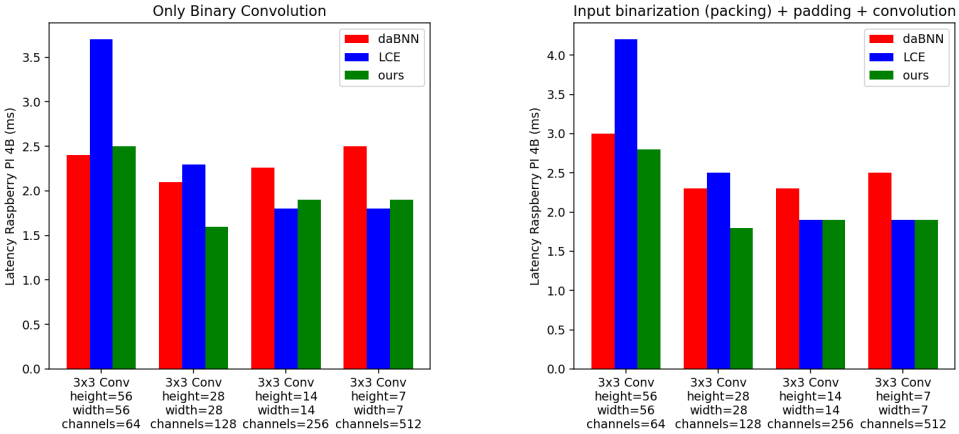
Efficiency Analysis

To validate the efficiency of our method we focused on the convolution macro-block (to extend the results reported in [8] also to Raspberry Pi 3) of Figure 3.1 and compared the efficiency of the proposed approach with LCE and DaBNN, which, to the best of our knowledge, are the fastest inference engines for binary neural networks.

Our assessment was performed on ARMv8 platforms, Raspberry Pi 3B and 4B. We implemented, differently from our predecessors, the convolution operation using ARM NEON *intrinsics* instead of inline assembly. Intrisics allow to produce code easier to maintain and automatically fits both ARMv7 and ARMv8 platforms without losing appreciable performance compared to pure assembly code. In Figure 3.6 we compare implementations on targets Rpi 3B and 4B. Our solution shows a clear



(A) Raspberry Pi 3 benchmark.



(B) Raspberry Pi 4 benchmark.

FIGURE 3.6: Latency evaluation of our method compared to DaBNN and LCE on Raspberry Pi 3B (a) and 4B (b) devices.

performance improvement for single binarized convolutions for all kernels and, including all the optimizations introduced in previous sections, accelerates binary convolution up to 1.91 and 2.73× compared to LCE and DaBNN with an average improvement of 1.46 and 1.61× respectively.

Accuracy Analysis

We evaluated two VGG-style networks (VGG-11 and VGG-Small) and a ResNet-18 for CIFAR-10 and SVHN. VGG-11 [147] and VGG-Small [154] are both high-capacity networks for classification. Pre-trained binary models (BinaryResNetE18 and BinaryDenseNet28) were adopted to evaluate the accuracy on ImageNet.

Results on CIFAR10 and SVHN. For CIFAR10 the RGB images are scaled to the interval $[-1.0; +1.0]$ and the following data augmentation was used: zero padding of 4 pixels for each side, a random 32×32 crop and a random horizontal flip. No augmentation is used at test time. The models have been trained for 140 epochs.
On SVHN the input images are scaled to the interval $[-1.0; +1.0]$ and the following data augmentation procedure is used: random rotation (± 8 degrees), zoom

Method	Topology	Bit-width	CIFAR10 %	SVHN %
BNN [23]	VGGSmall [154]	32 FP	93.8	96.5
Main/Subs. Net.	VGG11 [147]	32 FP	83.8	-
ResNet-18 [112]	ResNet-18	32 FP	93.0	97.3
BNN	VGGSmall	1-bit	89.9	96.5
XNOR-Net [113]	VGGSmall	1-bit	82.0	96.5
Bop [54]	VGGSmall	1-bit	91.3	-
BNN-DL [29]	VGGSmall	1-bit	89.9	97.2
IR-Net [41]	VGGSmall	1-bit	90.4	-
Main/Subs. Net.	VGG11	1-bit	82.0	-
Bi-Real Net [88]	ResNet-18	1-bit	89.3	94.7
ReActNet [90]	ResNet-18	1-bit	91.5	95.7
ours	VGGSmall	1-bit	88.8	96.1
ours	VGG11	1-bit	**83.7**	95.5
ours	ResNet-18	1-bit	90.3	95.3

TABLE 3.1: Accuracy comparison (top1) of our method with SOTA on CIFAR10 and SVHN.

Method	Topology	Bit-width	top1 %	top5 %
XNOR-Net [113]	ResNet-18	1-bit	51.2	73.2
Bi-Real Net [88]	ResNet-18	1-bit	56.4	79.5
BinaryResNetE18 [11]	ResNet-18	1-bit	58.1	80.6
BinaryDenseNet28 [11]	DenseNet-28	1-bit	60.7	82.4
ours	ResNet-18	1-bit	58.1	80.6
ours	DenseNet-28	1-bit	60.7	82.4

TABLE 3.2: Accuracy comparison of our method with SOTA on ImageNet.

([0.95, 1.05]), random shift ([0; 10]) and random shear ([0; 0.15]). The models have been trained for 70 epochs.

All the networks have been trained using the same training procedure without adopting additional distillation losses to further improve the accuracy of BNN models.

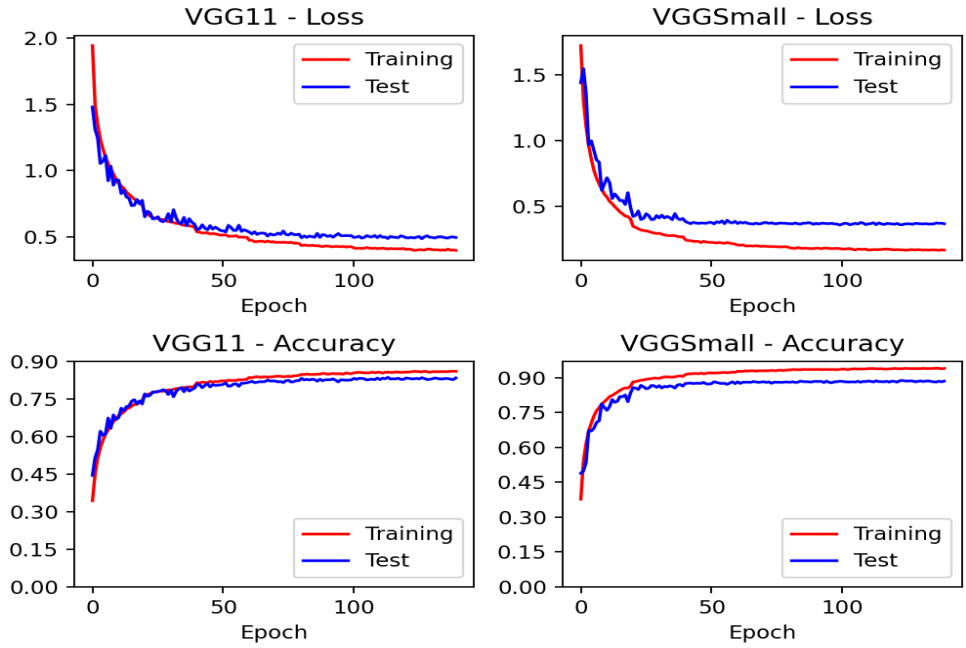
The accuracy achieved by the models is reported in Table 3.1 showing that the clip operation does not substantially affect the overall accuracy and the two-stage clipping allows to preserve the original accuracy. Figures 3.7, 3.8, 3.9 and 3.10 show the training and validation curves on CIFAR10 and SVHN; we can note that a limited number of epochs is necessary during the second training stage to recover accuracy.

Results on ImageNet. Tests were performed by using pre-trained binary versions of ResNet18 and DenseNet28 [11] taken from *zoo literature of Plumerai* [2]. Each BNN module (refer to Figure 3.1) has been modified according to Figure 3.1b. Residual blocks seem to be more robust to clipping compared to VGG style blocks (Results are in Table 3.2).

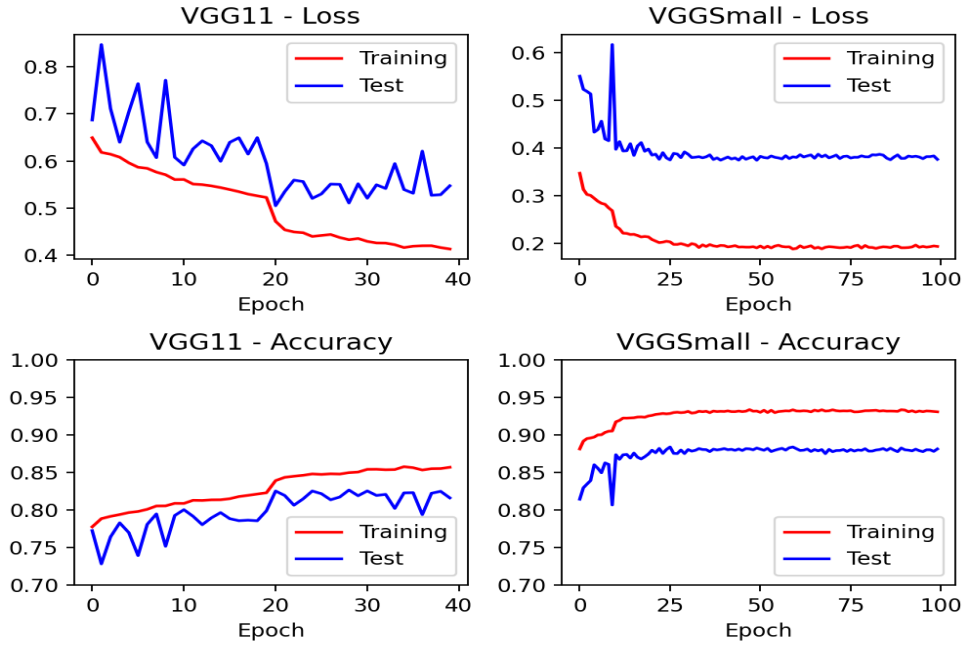
3.1.3 Final Remarks on Data-Flow Optimization

In Section 3.1, it has been shown several optimizations in the BNN data flow that achieve an overall speed-up of **1.91** and **2.73**× compared to state-of-the-art BNNs

[2] https://docs.larq.dev/zoo/api/literature/

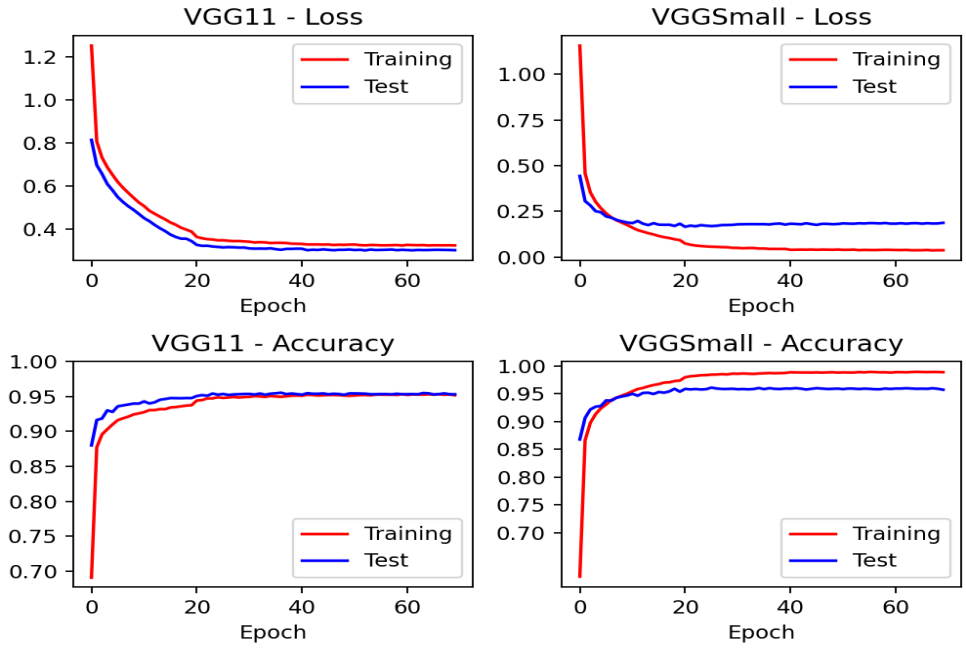


(A) CIFAR10 first training stage curves.

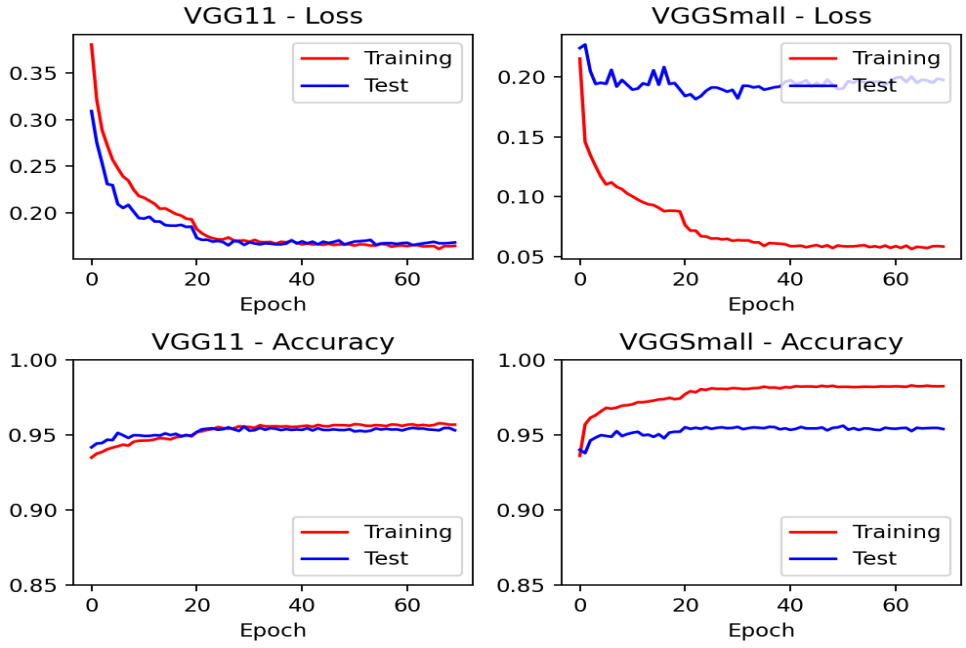


(B) CIFAR10 second training stage curves.

FIGURE 3.7: Training loss and testing accuracy curves for VGG11 and VGGSmall on CIFAR10 of the first and second training stages.

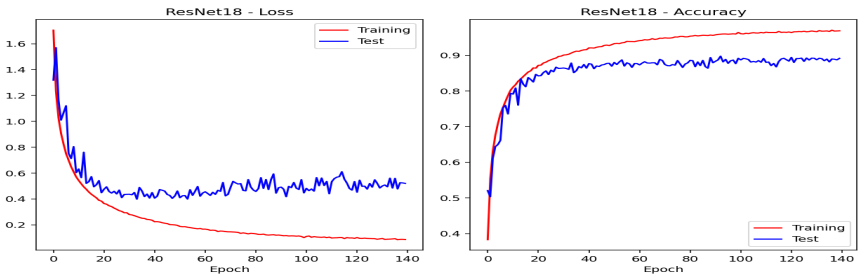


(A) SVHN first training stage curves.

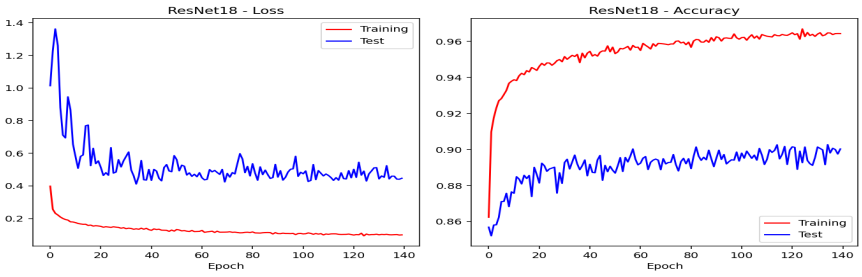


(B) SVHN second training stage curves.

FIGURE 3.8: Training loss and testing accuracy curves for VGG11 and VGGSmall on SVHN of the first and second training stages.

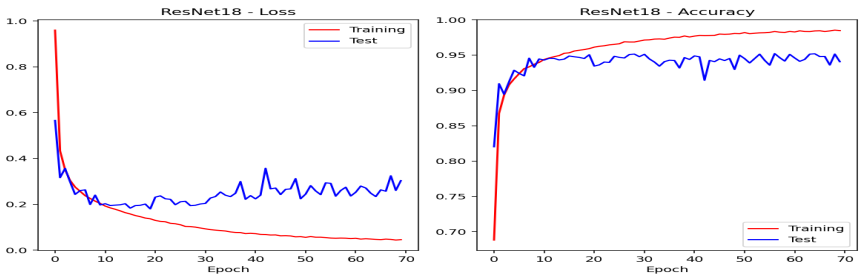


(A) CIFAR10 first training stage curves for ResNet-18.

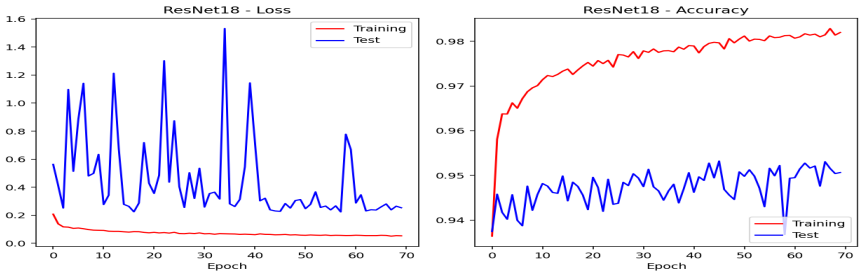


(B) CIFAR10 second training stage curves for ResNet-18.

FIGURE 3.9: Training loss and testing accuracy curves for ResNet-18 on CIFAR10 of the first and second training stages.



(A) SVHN first training stage curves for ResNet-18.



(B) SVHN second training stage curves for ResNet-18.

FIGURE 3.10: Training loss and testing accuracy curves for ResNet-18 on SVHN of the first and second training stages.

frameworks, LCE and daBNN, without any accuracy loss for at least one full-precision model. Specifically, we introduced a clipping block that decreases the data width from 32 bits to 8. Furthermore, we reduced the internal accumulator size of a binary layer, usually 32-bit, to prevent data overflow without losing accuracy. Additionally, we provided an optimization of the Batch Normalization layer that both reduces latency and simplifies deployment. Finally, we presented an optimized implementation of the Binary Direct Convolution for ARM NEON instruction sets.

3.2 Input Layer Binarization with bit-plane Encoding

Previous section 3.1 introduced some optimizations that can be used to speed up the inference time of a BNN model. In this section, we present a solution used to fully binarize a model. In fact, most of the BNNs do not fully exploit the benefits of 1-bit quantization, since they exclude from binarization the first and last layers that normally work with fixed-point numbers. In general, the number of parameters and the computational effort of the first layer is relatively low compared to intermediate deep convolutional layers employed in VGG [127] or ResNet [50] models since input data has typically fewer channels (e.g. color images have three channels). This usually leads to deploy the first layer of BNN models using floating-point or quantizing it using 8-bit; the consequence is that two different types of multipliers (8-bit for the first layer, binary for the remaining), with different bit widths, are needed to execute the computations leading to a solution which increases the power consumption (8-bit multiplier requires more power than xnor) and consumes more hardware resources (e.g. an FPGA design) than xnor gates. Conversely, the challenge of binarizing both weights and activations in the input layer is due to the small number of input channels [4]. Therefore, almost all the works addressing the binarization of the first layer tried to increase the number of input channels to enrich data representation. For instance, FBNA [4] proposes a two-step optimization scheme that consists of binarization and pruning; during the binarization phase, the number of input channels is increased by a factor 256$\times$ and then, during pruning, the lowest bits of input data are dropped away. The constraint of FBNA is that the encoded vector must be a power of two. BIL [31] attempts to directly unpack the 8-bit fixed-point input data, called *DBID*, and add an additional binary pointwise convolutional layer between the unpacked input data and the first layer to increase the number of channels, dubbed as *BIL*. The authors of FracBNN [159] propose to use thermometer encoding to transform a pixel to a thermometer vector (expanding each input channel to 32 binary channels) that then is transformed to the $\{-1, +1\}$ bipolar representation.

In contrast with previous works where the number of input channels has been increased, our method directly uses the fixed-point representation of a pixel. The results show that the proposed technique is competitive both in terms of efficiency and accuracy. Our contributions to the binarization of the input layer can be summarized as follows:

- we propose a general approach to binarize the first layer of a CNN using the native 8-bit fixed-point inputs. We rearrange the 8-bit input data into 8 bit planes, each bit plane is consumed by a binary depth-wise convolutional layer which gives more importance (using a multiplier, actually a shift operation) to the most significant bit planes. Finally, all feature maps are fused together through an addition operator. The entire process, depicted in Figures 3.13 and

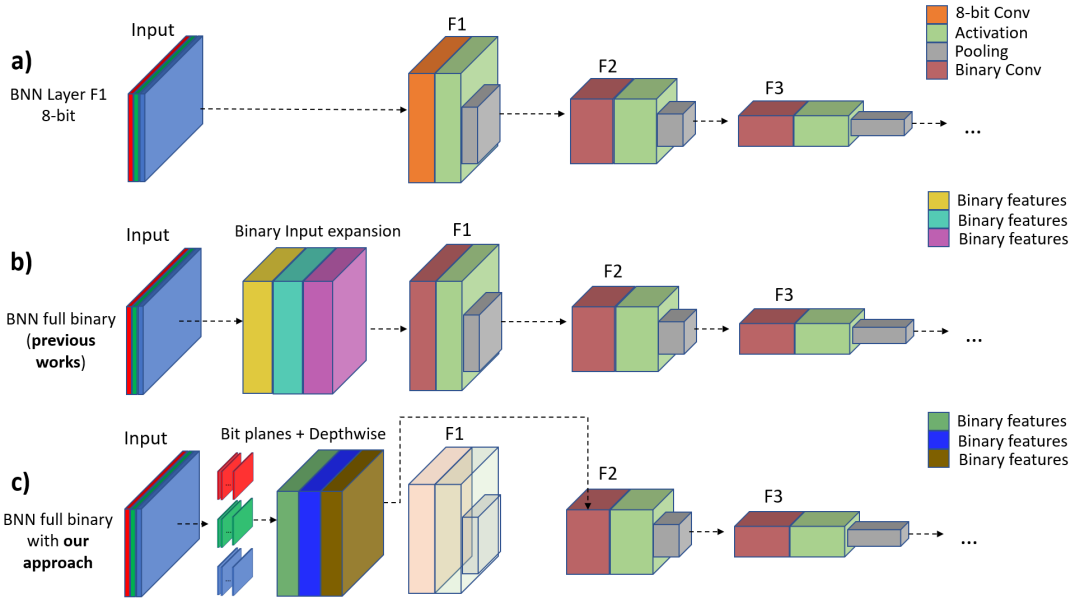


FIGURE 3.11: *(a)* Standard scenario of BNNs where the first convolutional layer is not binarized; weights and inputs are used in 8-bit/floating-point representation. *(b)* Typical approach of the works that binarized the first layer F_1 incrementing the number of input channels; in this case, the input expansion is actually an additional layer. *(c)* Our approach, where depth-wise convolutions are applied to input bit-planes and the resulting maps can replace the F_1 layer, producing a more compact model.

3.14, does not rely on floating-point computation, resulting in more suitable to be deployed on ASIC or FPGA systems.

- we show that the feature maps resulting from our bit-plane manipulations allow to skip the original[3] F_1 first network layer (see Figure 3.11c) with a minimal accuracy loss, leading to a model which uses less BMACs.
- we evaluate our concept on three classification datasets (SVHN, CIFAR10, and CIFAR100 [71]) showing that our solution outperforms all previous methods introduced to binarize the input layer.

3.2.1 Method

A common CNN model employed for computer vision problems works with RGB input images; it takes an input volume with three channels ($H \times W \times C$, where C is the number of channels) and extracts the features using convolutional blocks. To increase the receptive field of the network, a sequence of *pooling* operations is used. Each input pixel p is usually a fixed-point integer with 8 bit precision, namely $p = \sum_{m=0}^{7} x_m \cdot 2^m$.

In BNNs, typically the first layer (usually a convolutional one) is not binarized, all the input pixels are processed using 8-bit weights, producing F_1 output 8-bit feature maps (Figure 3.11a). The previous works in literature that addressed the problem to generate F_1 binary feature maps, adopted different techniques to increase the number of input channels C (generating a more sparse representation) in order to use binary weights and inputs for layer F_1; usually a good tradeoff between accuracy and increment of first layer MACs is to wide the number of channels C by 32× [31,

[3] Before the addition of our depth-wise convolutions.

159]. This process is depicted in Figure 3.11b, where the increment of input channels leads to a bigger model footprint; in fact, a linear increment of the number of input channels, linearly increases also the kernel parameters of a 2D convolutional layer.

The intuition behind our approach is that by extracting a different bit plane for each bit position, the semantic spatial information is preserved for most of the high index bits (4 to 7), as shown in Figure 3.12b. Lower bit indexes ($0-3$) contain less correlated spatial information of image pixels and, depending on the dataset, they can be selectively omitted to further reduce the computational effort. The overall diagram of our method is reported in Figures 3.13 and 3.14 and it is composed of the following steps:

1. **Bit Rearrangement:** An input image $\mathcal{I}$ (W, H, C, where C is the number of channels), having M bits for each pixel (usually 8), is rearranged into bit planes (as shown in Figure 3.13a); each 8-bit input channel is decomposed into eight 1-bit planes. A bit plane x is a 1-bit map containing only the bit of index x for all pixels (see Figure 3.12a). The bit-plane image bp corresponding to channel c can be indicated as $\mathcal{I}(c, bp)$.

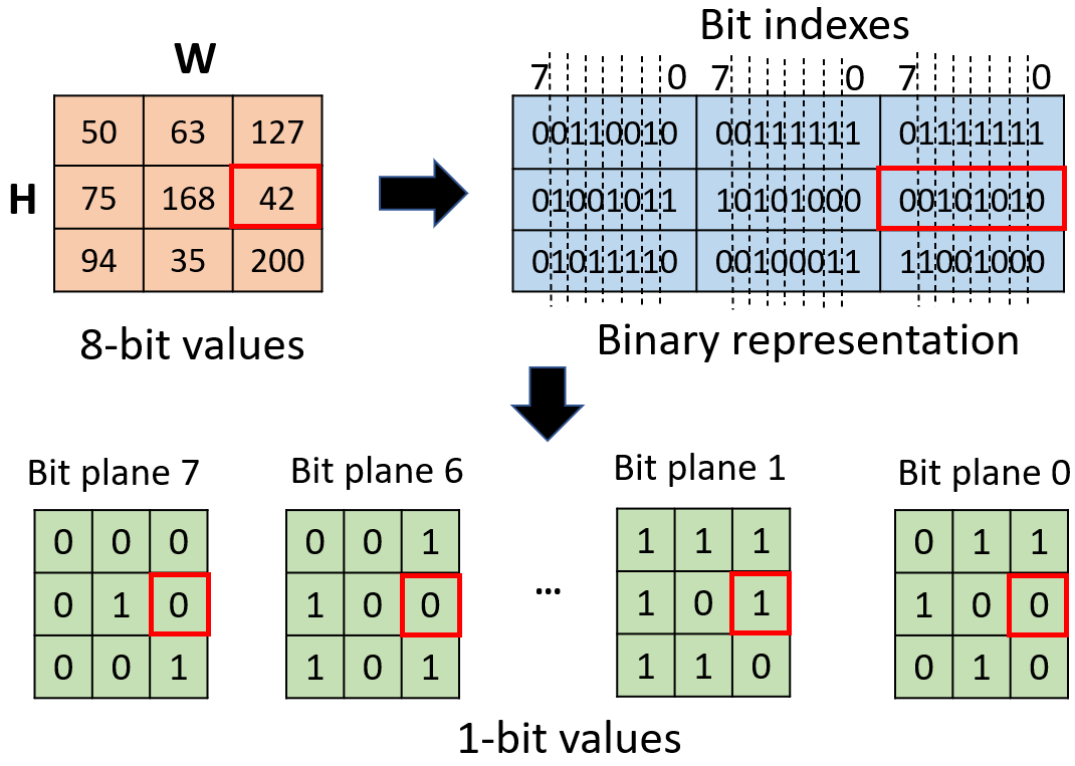
2. **Feature Extraction:** Each binary bit-plane is consumed by a binary depthwise convolution layer that generates **N** feature maps for each bit plane, as reported in Figure 3.13b. The output of feature extraction ($\mathcal{FE}$) step can be formulated as:

$$\mathcal{FE}(c, bp) = \gamma(c, bp) \frac{(\mathcal{I}(c, bp) * W(c, bp) + b(c, bp)) - \mu(c, bp)}{\sigma(c, bp)} + \beta(c, bp) \tag{3.3}$$

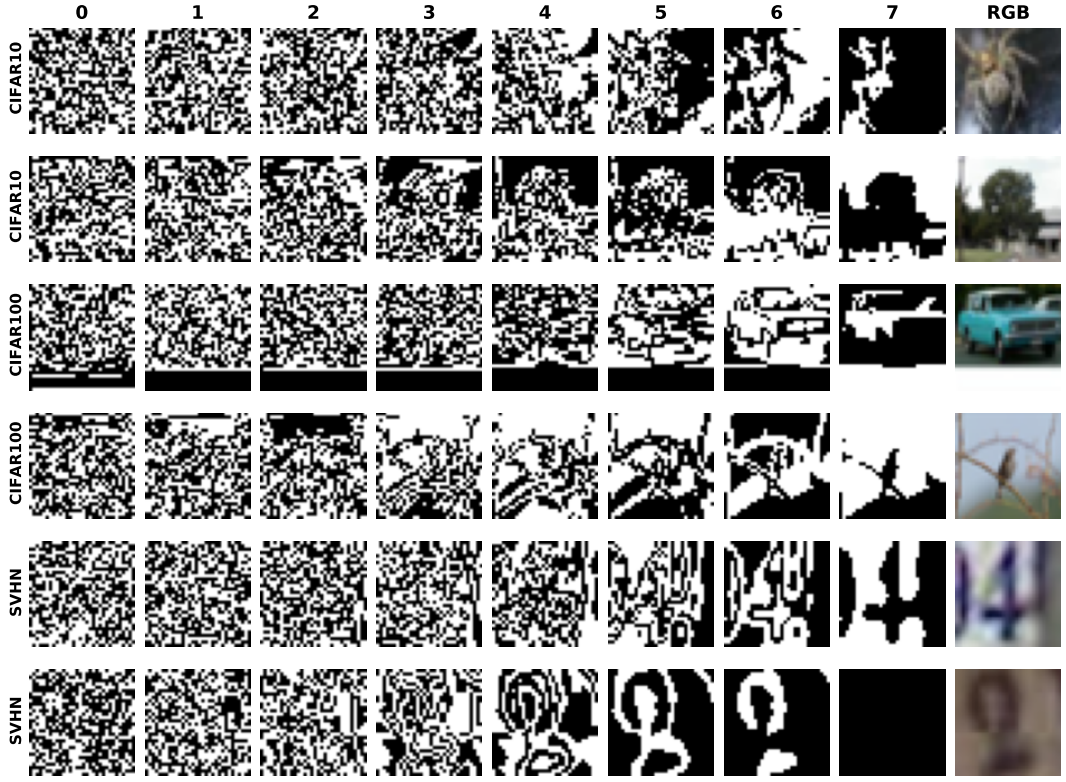
   where $*$ is the convolution operator, $W(c, bp)$ and $b(c, bp)$ represent the weights of the depth-wise convolution while γ, μ, σ and β are the Batch Normalization (BN) [60] parameters; Eq. 3.3 refers to a single feature map of depth-wise convolution, which is dependent on channel c and bit plane bp. In Eq. 3.3 the non-linear activation function can be omitted because binarization of activation and weights already introduce non-linearity. The use of Batch Normalization after each binary layer plays a key role in BNNs because it promotes a smoother optimization process allowing a stable behavior of the gradients. BN layer is usually executed in floating-point precision when mixed with binary layers, but the authors of [139] proved that it can be executed in 8-bit fixed point without accuracy loss.

3. **Features Re-Weight:** Following the intuition based on Figure 3.12b, where high index bit planes preserve the spatial information of the image, this stage re-weights the feature maps based on the bit plane index. Higher bit planes are multiplied by higher scalar values. In order to simplify this stage, the multiplication can be replaced by a shift operation. The N feature maps of each bit plane are shifted by the same quantity (namely a power of two).

4. **Features Fusion:** The re-weighted feature maps, corresponding to a different 8-bit input channel, are summed to combine the information encoded by different bit indexes and can be expressed as:



(A) Bit Planes example



(B) Bit Planes on CIFAR10, CIFAR100 and SVHN

FIGURE 3.12: a Example of bit plane representation for a 3×3 8-bit image. b Image representation in bit planes. Each column refers to a bit index extracted from the image; for representation purposes, bit 1 is converted to 255 while bit 0 remains 0. In this example, all bit planes refer to channel G of RGB images.

Method	Type	# MACs	# weights	$\frac{MACs\ method}{MACs\ Baseline}$ [1]	Speedup[2]
Baseline	8-bit	$HWCF^2F_1$	CF^2F_1	1×	1×
DBID[31]	1-bit	$HWCMF^2F_1$	$CMK + F^2F_1K$	8×	1.12×
BIL[31]	1-bit	$HWK\left(CM + F^2F_1\right)$	CKF^2F_1	10.8×	0.81×
Thermometer[159]	1-bit	$HWCKF^2F_1$	CF^2N_1M	32×	0.27×
ours $(P = 8, N_1)$	1-bit	$HWCPF^2N_1$	CPF^2N_1	2.6×	**3.42**×
ours $(P = 4, N_1)$	1-bit	$HWCPF^2N_1$	CMF^2N_1	1.3×	**6.84**×
ours $(P = 4, N_2)$	1-bit	$HWCPF^2N_2$	CPF^2N_2	1×	**9**×

[1] A lower ratio means a higher reduction of MACs.
[2] According to [8] (Figure 2), the worst case speedup of binary convolution compared to 8-bit is 9×.

TABLE 3.3: Comparison of the first layer MACs required by our method with respect to the state-of-the-art solutions. Input data has a shape $H \times W \times C$ $(32 \times 32 \times 3)$ and a precision of M bits; in this example, the first convolutional layer has $F_1 = (128)$ filters with size $F \times F$ (3). The expansion channels is $K = 32$ for methods [31, 159]. The depthwise multiplier of our method can be chosen as $N_1 = \left\lfloor \frac{F_1}{C} \right\rfloor = 42$. We conducted our experiments using also a lower value, $N_2 = 32$ instead of N_1 and only 4 bits of input pixels. P represents the number of bit planes extracted by step 3.13a.

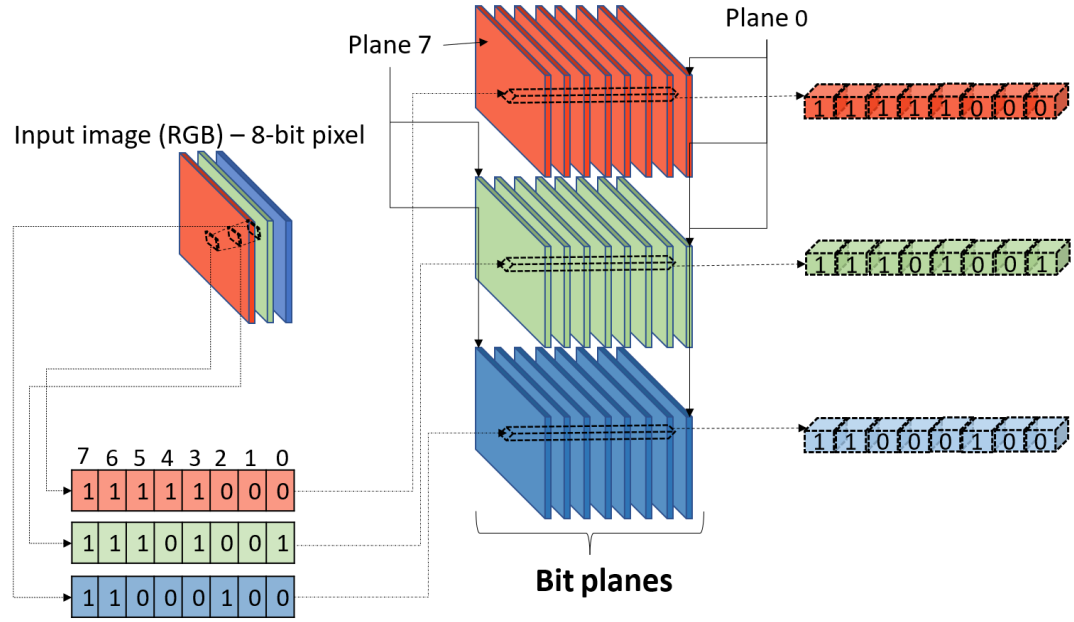
$$\mathcal{FWF}\left(c, bp\right) = \sum_{i=0}^{M} \mathcal{FE}\left(c, bp\right) \cdot 2^i \tag{3.4}$$

In Eq. 3.4 the multiplication by 2^i represents the re-weight of feature maps that can be implemented with a shift operation. The sum instead can be implemented by accumulating features over a 32-bit register; if the subsequent layer extracts *sign* from inputs, then the 32-bit output maps can be reduced to 1-bit saving memory overhead. The N feature maps corresponding to a different channel are concatenated to create a volume of $N \times 3$ maps that is used to feed the network, Figure 3.11c. Such volume of N maps can replace the first layer of the CNN with almost no accuracy loss, as shown in Section 3.2.3, thus reducing the complexity of the overall model. F_1 requires a topology change of the first network layer when its weights and inputs are binarized and the expansion of depth-wise convolutions (Figure 3.13b) can be set up in order to keep a number of feature maps equivalent to the layer F_1.

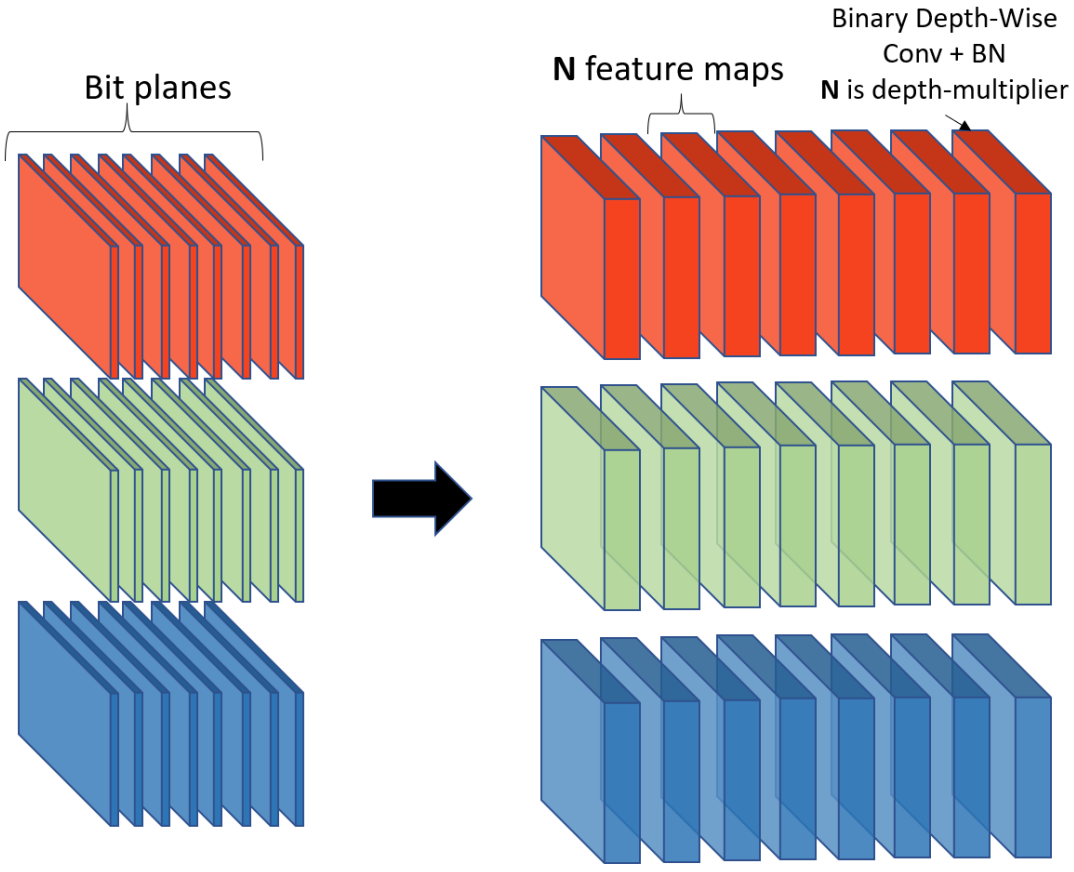
Table 3.3 reports the MACs of different approaches used to binarize the input layer of a CNN, reporting the theoretical speedup of the methods; our solution is clearly competitive, in terms of MACs, with respect to the baseline (input not binarized) and other existing approaches.

3.2.2 Implementation Details

We evaluate our method on three classification datasets: CIFAR10, CIFAR100, and SVHN with different BNN architectures. For each model architecture, we tested different state-of-the-art binarization techniques of the input layer; input binarization does not modify the other layers of the network, which remain unaltered. For each dataset, we conducted our experiments with the same training procedure (same number of epochs, optimizer, learning rate scheduling, loss function) for all topologies without adding distillation losses or special regularization to the overall loss

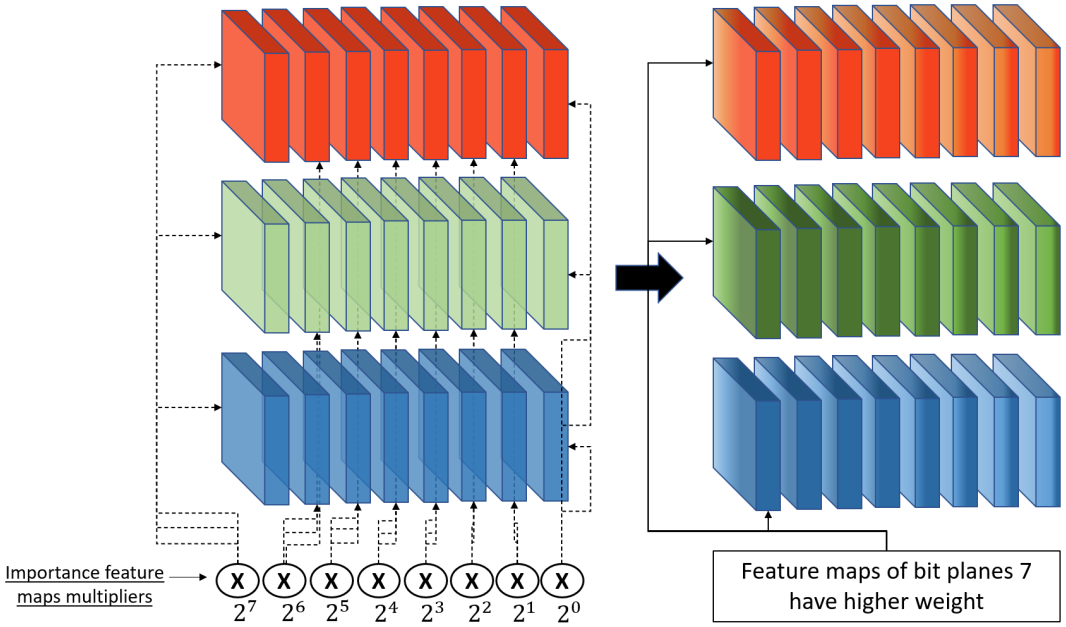


(A) Bit Rearrangement

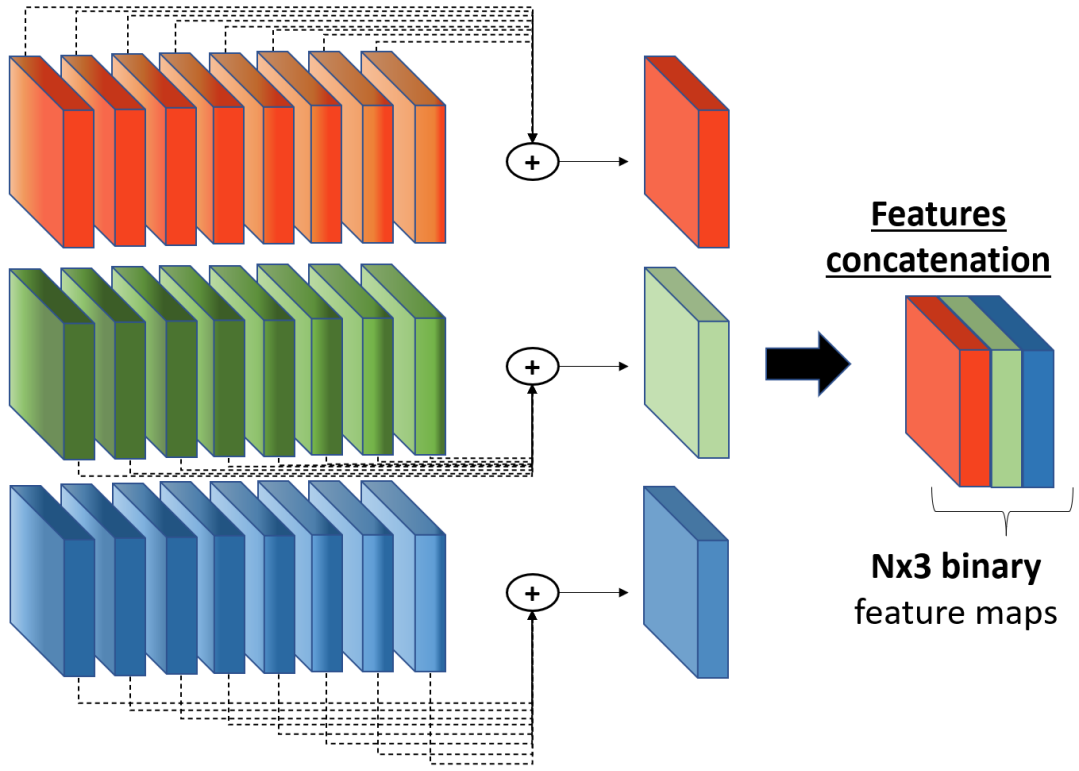


(B) Features Extraction

FIGURE 3.13: Binarization process of input layer. a shows the rearrangement phase that extracts, for each bit position of the encoded pixel a bit plane. b shows the binary depth-convolution block applied to each bit plane; the depth multiplier (N) is a hyperparameter and it is dataset and model dependent.



(A) Features Re-Weight



(B) Features Fusion

FIGURE 3.14: Binarization process of input layer. a shows how to weigh differently feature maps extracted from different bit planes; maps related to the most significant bits receive a higher multiplication factor. In b, the feature maps related to the same 8-bit input channel are fused together through an addition.

function. The binarization of weights and activations always happens at training time using an approximation of the gradient (Equation 2.7 or derived solutions that are model dependent) for *sign* function. The augmentation procedure for all datasets is performed with floating-point arithmetic but, before feeding data to the network, the input image is quantized using 8-bit fixed precision.
We adopted specific input data rescaling and augmentation procedure for each dataset tested, as reported below:

CIFAR10 and CIFAR100 The RGB images are scaled to the interval $[-1.0\,;+1.0]$ and the following data augmentation was used: zero padding of 4 pixels for each size, a random 32×32 crop and a random horizontal flip. No augmentation is used at test time. The models have been trained for 140 epochs.

SVHN The RGB input images are scaled to the interval $[-1.0\,;+1.0]$ and the following data augmentation procedure is used: random rotation (± 8 degrees), zoom ($[0.95, 1.05]$), random shift ($[0; 10]$) and random shear ($[0; 0.15]$). The models have been trained for 70 epochs.

We evaluated the following networks:

VGG-Small[154] Network structure is the following: $2 \times (128 - C3) + MP2 + 2 \times (256 - C3) + MP2 + 2 \times (512 - C3) + MP2 + FC1024 + FC1024 + Softmax$[4]. The VGG-Small model adopted uses the straight-through-estimator (STE) to approximate the gradient on non-differentiable layers[58, 9].

VGG-11[147] Network structure is the following: $64 - C3 + MP2 + 128 - C3 + MP2 + 2 \times (256 - C3) + MP2 + 2 \times (512 - C3) + MP2 + 2 \times (512 - C3) + MP2 + Softmax$. Even VGG-11 uses the STE estimator for binarization operation during back-propagation.

BiRealNet[88] It is a modified version of classical ResNet that proposes to preserve the real activations before the sign function to increase the representational capability of the 1-bit CNN, through a simple shortcut. Bi-RealNet adopts a tight approximation to the derivative of the non-differentiable sign function with respect to activation and a magnitude-aware gradient to update weight parameters. We used two instances of the network, an *18-layer* and a *34-layer* Bi-Real net [5].

ReactNet[90] To further compress compact networks, this model constructs a baseline based on MobileNetV1 [57] and add a shortcut to bypass every 1-bit convolutional layer that has the same number of input and output channels. The 3×3 depth-wise and the 1×1 point-wise convolutional blocks of MobileNet are replaced by the 3×3 and 1×1 vanilla convolutions in parallel with shortcuts in React Net[6]. As for Bi-Real Net, we tested two different versions of React Net: a *18-layer* and a *34-layer*.

	VGG-Small	VGG-11	BiReal-18	BiReal-34	React-18	React-34	
DBID[31]$(P=8)$	84.5	77.6	81.8	85.0	86.0	86.7	
BIL[31]$(P=8)$	82.0	78.9	81.3	82.2	84.0	83.5	
Therm[159]$(K=32)$	84.2	80.1	85.2	85.3	86.6	86.6	1*st*
ours$(P=8,N_1)$	**85.9**	79.1	**87.7**	**88.5**	**89.9**	**90.2**	
baseline	89.2	84.7	89.1	89.3	90.6	90.6	
DBID$(P=4)$	83.6	76.8	74.9	83.7	83.7	85.3	
BIL$(P=4)$	80.9	82.4	80.8	82.3	82.7	83.4	
Therm$(K=16)$	83.8	79.6	84.7	85.9	86.5	86.8	2*nd*
ours$(P=4,N_2)$	**85.0**	78.3	**86.9**	**87.7**	**88.5**	**89.0**	
baseline	88.3	83.7	87.4	88.3	88.8	89.1	

TABLE 3.4: Top1 accuracy (%) results of test set on CIFAR10. In the first part we report the result of the first test scenario (standard conditions); in the second half, the results achieved in the second scenario (reducing the MACs of binarization of input layer).

	VGG-Small	VGG-11	BiReal-18	BiReal-34	React-18	React-34	
DBID[31]$(P=8)$	94.5	92.2	94.3	95.1	94.9	95.1	
BIL[31]$(P=8)$	93.5	92.1	94.3	93.4	94.1	94.7	
Therm[159]$(K=32)$	89.7	88.9	89.2	89.8	89.8	90.2	1*st*
ours$(P=8,N_1)$	**94.8**	**93.4**	94.3	95.0	**95.1**	**95.7**	
baseline	95.7	95.5	94.3	95.1	95.5	95.9	
DBID$(P=4)$	94.3	92.1	94.3	94.7	94.8	95.0	
BIL$(P=4)$	93.4	92.1	94.4	93.5	93.8	94.5	
Therm$(K=16)$	89.5	88.6	89.8	89.7	89.8	90.1	2*nd*
ours$(P=4,N_2)$	**94.8**	**93.3**	94.3	**95.0**	**95.1**	**95.7**	
baseline	95.6	95.0	94.4	95.1	95.5	96.0	

TABLE 3.5: Top1 accuracy (%) results of test set on SVHN.

3.2.3 Experimental Results

The validation of our solution has been accomplished through two different test scenarios; in the first one, we compared the accuracy (measured on test set) of our binarization method w.r.t. the state-of-the-arts input layer binarization approaches, keeping unaltered the structure of the network except for the input data binarization layer (first half of Tables 3.4, 3.5 and 3.6). In this first scenario, all the 8-bit planes are exploited, layer F1 (Figure 3.11) is executed and our proposed solution is able to reach a better accuracy compared to other input binarization methods, closing the accuracy gap with the baseline.

In the second scenario, to further reduce the MACs of our solution, we propose an optimization of our method that uses only the 4 most significant bits and reduces

[4] $m \times (n - CK)$ stands for m consecutive convolutional layers, each one with n output channels and K kernel size. $MP2$ is the max pooling layer with subsample 2 while FCx is a fully-connected layer having x neurons. $Softmax$ represents the last dense classification layer using softmax as activation.

[5] Refer to the following `https://github.com/liuzechun/Bi-Real-net` repository for all the details.

[6] Refer to the following `https://github.com/liuzechun/ReActNet` repository for all the details.

	VGG-Small	VGG-11	BiReal-18	BiReal-34	React-18	React-34	
DBID[31]$(P=8)$	53.6	43.1	51.8	58.5	56.3	58.0	
BIL[31]$(P=8)$	50.0	42.9	52.7	56.0	55.4	55.5	
Therm[159]$(K=32)$	53.0	43.5	57.2	57.1	57.4	57.9	1*st*
ours$(P=8, N_1)$	**56.5**	**46.0**	**58.7**	**60.6**	**61.7**	**62.9**	
baseline	60.6	52.3	63.4	65.0	64.9	65.3	
DBID$(P=4)$	52.3	41.8	50.5	56.5	55.2	56.7	
BIL$(P=4)$	49.5	42.0	52.1	54.5	52.1	53.6	
Therm$(K=16)$	52.1	42.6	56.7	54.5	56.8	58.6	2*nd*
ours$(P=4, N_2)$	**54.8**	**44.5**	**57.7**	**59.6**	**60.2**	**62.0**	
baseline	60.3	50.3	60.0	61.7	62.0	63.4	

TABLE 3.6: Top1 accuracy (%) results of test set on CIFAR100.

the depth-wise multiplier from N_1 to N_2 (the reduction to 4 bits is based on Figure 3.12b, that shows how the bit planes corresponding to less significant bits convey less information). In the second half of tables 3.4, 3.5 and 3.6, we report the results of the optimized version compared with other solutions properly modified in order to compute an equivalent number of channels [7]. As reported, our solution is able to preserve the baseline accuracy using fewer input bits while the other methods get a consistent accuracy drop when reducing input bits and binary channels.
Differently from other works, our solution re-weights the feature extracted by bit-planes giving more importance to the features corresponding to the most significant bit-planes; this stage contributes to scale down the footprint of our binarization approach simplifying the deployment on resource-constrained devices (low-power embedded CPUs). Furthermore, the accuracy of our method is higher than *thermometer encoding*[159], which preserves the feature similarity after binarizing the input layer, as pointed out by Anderson et al. [4].

3.2.4 Final Remarks on Input Binarization using bit-plane Encoding

In Section 3.2, it has been introduced a novel input layer binarization method that reaches higher accuracy when compared to state-of-the-art solutions reducing the gap to the baseline on average by **2.2** percentage points. Our solution is able to preserve model accuracy when only 4 bits of input pixels are used in the input binarization layer, proving to be more resource-constrained and device-friendly than existing ones.

[7] For *DBID*, *thermometer* and *baseline* methods, we reduced to 32 the number of output channels of layer *F*1; for *BIL* and *ours*, we skipped the layer *F*1 because the convolution operation is already exploited within the input layer binarization process. For *DBID*, *BIL*, and *ours* we used only the 4 most significant bits of input data. For *thermometer* we applied also a reduced expansion factor of $K = 16$.

Chapter 4

Continual Learning with Binary Neural Networks

In this chapter, we describe our solutions proposed to continually update a BNN directly on-device in the context of Continual Learning (CL) targeting different application tasks such as classification, regression, and segmentation. Specifically, the chapter introduces a new method to continually update a binary model on-device by proposing an *ad-hoc* quantization scheme (section 4.1) for the classification head. Later, in section 4.2, we introduce a solution that, exploiting a latent replay memory (RM) to store past sample activations, retrains a small portion of convolutional layers to increase model accuracy (compared to the approach of section 4.1). The solution described in Section 4.1 has been published in [138] while the method detailed in Section 4.2 has been published in [140] and awarded with the *Best Industrial Paper* acknowledgement.

4.1 Binary Neural Networks and CWR*

In recent times, the integration of Artificial Intelligence into the Internet of Things (IoT) paradigm [100, 3], enabling the provision of intelligent systems capable of learning even within embedded or tiny devices, has garnered significant attention in the literature. This trend has been facilitated by various factors, including the evolution of microchips, which have led to the availability of cost-effective chips in many everyday objects. Additionally, the exploration of new learning paradigms, such as Continual Learning (CL) [105, 96], has contributed to the development of techniques for training neural networks continuously, on small data portions (denoted as *experiences*) at a time, mitigating the issue of catastrophic forgetting [69]. In this manner, a neural network, in contrast to the traditional machine learning paradigm, does not learn from a single large dataset accessible entirely during the training phase but rather from small data portions accessible gradually over time. This limited amount of data needed by the training procedure effectively simplifies the adoption of a CL training implementation on embedded devices.

Despite the keen interest of the scientific community, numerous challenges still persist, rendering the utilization of deep learning models on devices particularly demanding. These challenges are primarily associated with the high computational resources required by deep neural networks, even though based on CL strategies. Indeed, embedded devices often have limited available memory, preventing the storage of a vast amount of data. Furthermore, a powerful GPU is usually absent due to cost, space constraints, and energy consumption. These competing needs have

given rise in the last few years to a specific branch of machine learning and deep learning called TinyML [5], focused on shrinking and compressing neural network models with respect to the target device characteristics. One of the most extreme TinyML approaches, is Binary Neural Networks (chapter 2), where a single bit is used to encode weights and activations. However, almost no literature work addresses the problem of training (or tuning) such models on-device, a task which is still more complex than inference because:

- quantization is known to affect back propagation and weights update
- popular inference engines (e.g. Tensorflow Lite, pytorch mobile, ecc.) do not support model training

This work proposes on-device learning of BNN to enable continual learning of a pre-trained model. We start from CWR* [93], a simple but effective continual learning approach that limits weight updates to the output head, and designs an ad-hoc quantization approach that preserves most of the accuracy with respect to a floating point implementation. We prove that several state of the art BNN models can be used in conjunction with our approach to achieve good performance on classical continual learning dataset/benchmarks such as CORe50 [92], CIFAR10 [71] and CIFAR100 [71].

4.1.1 Continual Learning

The classical deep learning approach is to train a model on a large dataset split into several batches and then freeze it before deployment on edge devices; this does not allow adapting the model to a changing environment where new classes (NC scenario) or new items/variation of known classes (NI scenario) can appear over time. Collecting new data and periodically retraining a model from scratch is not efficient and sometime not possible because of privacy, so the CL approach is to adapt an existing model by using only new data. Unfortunately, this is prone to forgetting old knowledge, and specific techniques are necessary to balance the model stability and plasticity[26].
In this work we focus on Single Object Recognition task addressing the two CL scenarios of NI and NC; in both cases, the learning phase of the model is usually split in *experiences*, each one containing different training samples belonging or not to known classes (this depends on the CL scenario).
Few works in the literature addressed the on-device learning task proposing solutions to primarily reduce the memory requirement of the learning algorithm: Ren et al. [117] brought the transfer learning task on tiny devices by adding a trainable layer on top of a frozen inference model. Cai et al. [16] proposed to freeze the model weights and retrain only the biases reducing the memory storage during forward pass. Lin et al. [83] introduced a sparse update technique to skip the gradient computation of less important layers and sub-tensors. QLR-CL [114] relies on low-bitwidth quantization (8-bit) to speed up the execution of the network up to the latent layer and at the same time reduce the memory requirement of the latent replay vectors from the 32-bit floating point to 8-bit. In addition, backpropagation is performed with floating-point precision. In [101, 102], Nadalini et al. introduced a framework to execute on-device learning on tiny devices using floating-point (32 and 16 bits) computation.
The continual learning approach employed in this work, CWR*, mantains two sets of weights for the output classification layer: *cw* are the consolidated weights used

during inference while tw are the temporary weights that are iteratively updated during back-propagation. cw are initialized to 0 before the first batch and then updated according to Algorithm 3 (for more details see [93]), while tw are reset to 0 before each training mini-batch. CWR*, for each already encountered class (of current training batch), reloads the consolidated weights cw at the beginning of each training batch and, during the consolidation step, adopts a weighted sum based on the number of the training samples encountered in the past batches and those of current batch. The consolidation step has a negligible overhead and can be quantized adopting the same quantization scheme used for CWR* weights. In CWR*, during the first training experience (supposed to be executed offline) all the layers of the model are trained but from the second experience, only the weights of the output classification layer are adjusted during the back-prop stage, to simulate a real case scenario (lines 7 - 10 of Algorithm 3).

Algorithm 3 CWR* pseudocode: $\overline{\Theta}$ are the class-shared parameters. Both tw and cw refer to the same layer index k of the model and are quantized according to the scheme reported in section 4.1.2. Quantization of CWR* is fundamental to deploying our solution on devices such as FPGA that could hardly accommodate a floating-point implementation.

1: $cw_k = 0$ {k is the index of the classification layer}
2: $past = 0$ {number of samples for each class i encountered}
3: init $\overline{\Theta}$ random or from pre-trained model
{B_j is the mini-batch of index j}
4: **for all** training batch B_j **do**
5: $tw_k[i] = \begin{cases} cw_k[i], & \text{if class } i \text{ in } B_j \\ 0, & \text{otherwise} \end{cases}$
6: train the model with SGD;
7: **if** $B_j = B_1$ **then**
{$\overline{\Theta}$ is trained offline during first experience}
8: learn both $\overline{\Theta}$ and tw_k
9: **else**
10: learn tw_k while keeping $\overline{\Theta}$ fixed
11: **end if**
{consolidation step}
12: **for all** class i in B_j **do**
13: $wpast_i = \sqrt{\frac{past_i}{cur_i}}$, where cur_i is the number of patterns of class i in B_j
14: $cw_k[i] = \frac{cw_k[i] \cdot wpast_i + (tw_k[i] - avg(tw_k))}{wpast_i + 1}$
15: $past_i = past_i + cur_i$
16: **end for**
17: test the model by using $\overline{\Theta}$ and cw_k
18: **end for**

4.1.2 Method

Gradients Computation

In this section we make explicit the weights update formula of the classification layer; without loss of generality, a neural network $M(\cdot)$ is composed by a sequence of k layers represented as:

$$M(\cdot) = f_{w_k}\left(f_{w_{k-1}}\left(\cdots f_{w_2}\left(f_{w_1}(\cdot)\right)\right)\right) \tag{4.1}$$

where w_i represents the weights of the i^{th} layer. In CWR* the temporary weights tw_k (lines 8 and 10 of Algorithm 3) are updated according to Equations 4.9 and 4.10, whose quantization is discussed in the next section. Denoting with a_i and a_{i+1} [1] the input and output activations of the i^{th} layer respectively, with $\mathcal{L}$ the loss function, the backpropagation process consists in the computation of two different sets of gradients: $\frac{\partial \mathcal{L}}{\partial a_i}$ and $\frac{\partial \mathcal{L}}{\partial w_i}$.

In CWR* the on-device backpropagation algorithm is limited to the last layer which can be considered a linear layer (with a non-linear activation function) with the following forward formula:

$$a_{k+1} = f_k\left(o_{k+1}\right), \;\; o_{k+1} = a_k W_k + b_k \tag{4.2}$$

where a_{k+1} represents the output of the neural network.

Considering a classification task (with M classes) with an unitary batch size, the *Cross-Entropy* loss function is formulated as:

$$\mathcal{H}\left(y,\, a_{k+1}\right) = -\sum_{i=0}^{M-1} y^i log\left(a_{k+1}^i\right) \tag{4.3}$$

where y^i represents the element of a one-hot encoded vector of ground truth and a_{k+1}^i is the i^{th} output activation sample. Using the softmax as activation for the last layer, reported below:

$$a_{k+1}\left(o_{k+1}^t\right) = \frac{e^{o_{k+1}^t}}{\sum_{j=1}^{M} e^{o_{k+1}^j}} \tag{4.4}$$

, the gradient formulas for the last classification layer can be expressed using the chain rule:

$$\frac{\partial \mathcal{H}}{\partial W_k} = \frac{\partial \mathcal{H}}{\partial a_{k+1}} \frac{\partial a_{k+1}}{\partial o_{k+1}} \frac{\partial o_{k+1}}{\partial W_k} \tag{4.5}$$

$$\frac{\partial \mathcal{H}}{\partial b_k} = \frac{\partial \mathcal{H}}{\partial a_{k+1}} \frac{\partial a_{k+1}}{\partial o_{k+1}} \frac{\partial o_{k+1}}{\partial b_k} \tag{4.6}$$

The final expression for Eq. 4.5 using the Eq. 4.3 as loss function and 4.4 as non-linear $f_k\left(\cdot\right)$ is a well-known result, that can be easily derived:

$$\frac{\partial \mathcal{H}}{\partial W_k} = \left(a_{k+1} - y\right) a_k \tag{4.7}$$

$$\frac{\partial \mathcal{H}}{\partial b_k} = \left(a_{k+1} - y\right) \tag{4.8}$$

Using a stochastic gradient descent optimizer with learning rate η, the weights update equation is:

[1] Note that the output a_{i+1} of level i corresponds to the input of level $i+1$

$$W_k^{i+1} = W_k^i - \eta \left(a_{k+1} - y\right) a_k \tag{4.9}$$

$$b_k^{i+1} = b_k^i - \eta \left(a_{k+1} - y\right) \tag{4.10}$$

Therefore in CWR* the temporary weights tw_k (lines 8 and 10 of Alg. 3) are updated according to Equations 4.9 and 4.10, whose quantization is discussed in the next section.

Quantization Strategy

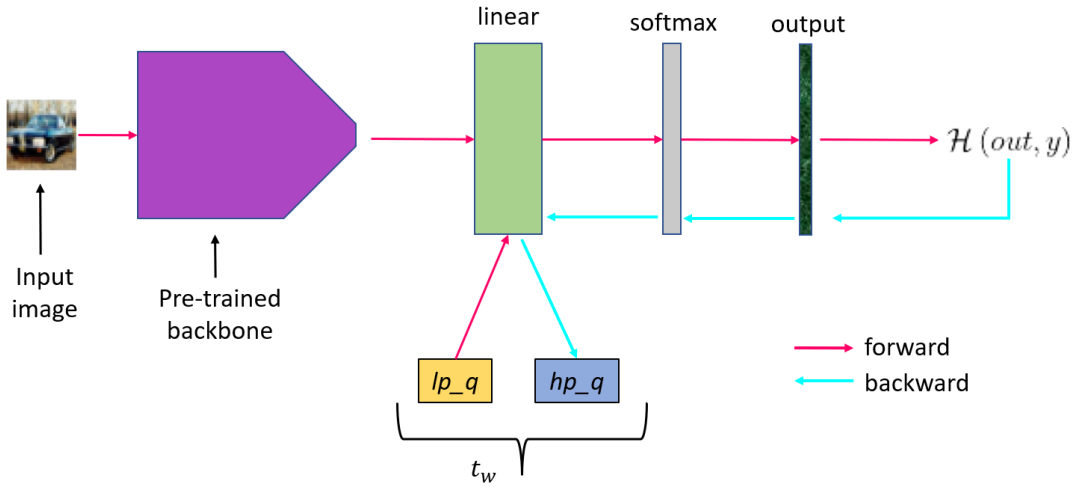


FIGURE 4.1: Double quantization scheme that uses a different quantization level for weights/activations used in forward and backward pass.

Our approach considers two different quantizations: the former uses 1-bit (also called binarization) to represent weights and activations employed by the pre-trained backbone; the latter is used in the last classification layer, to quantize both forward and backward operations. This solution both reduces the latency and simplifies the adaptation of the model on new item/classes encountered.
In particular, for the last layer quantization we followed the scheme proposed in [62] and implemented in GEMMLOWP library [61]. The quantized output of a 32-bit floating point linear layer, reported in Eq. 4.2, can be represented as:

$$\overline{o_{k+1}^{int\_q}} = cast\_to\_int\_q\lfloor s_k^{int\_32} \left(\overline{W_k^{int\_q}} a_k^{int\_q} + \overline{b_k^{int\_q}} \right) \rceil \tag{4.11}$$

The quantization Eq. 4.11 depends on the number of the quantization q bits used $(8, 16, 32)$, $\bar{\cdot}$ represents the quantized version of a tensor and s^{int_32} is the fixed-point scaling factor having 32-bit precision, as shown in Figure 4.2. Similarly to previous works [58, 98], we used the straight-through estimator (STE, Equation 2.7) approach to approximate differentiation through discrete variables; STE represents a simple and hardware-friendly method to deal with the computation of the derivative of discrete variables that are zero almost everywhere.

Based on the results reported in [46, 25, 7], the quantization of the gradients in Equations 4.7 and 4.8 represents the main cause of accuracy degradation during training and therefore we propose to use two separate versions of layer weights W_k, one with

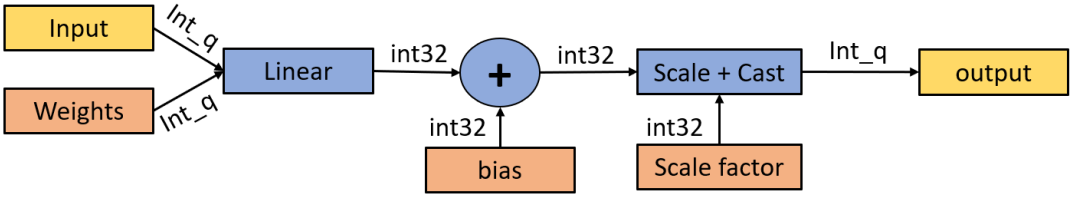


FIGURE 4.2: Quantization scheme adopted using q bits for weights and activations.

low-precision (lp_q) and another with higher precision (hp_q). As shown in Figure 4.1, the idea is to use the lp_q version of the weights for the computations that have strict timing deadlines (forward pass), while the hp_q version is adopted during the weight update step (Equations 4.9 and 4.10), which has typically more relaxed timing constraints (it can be executed also as a background process). Every time a new high-precision copy of weights is computed, a lower version is derived from it and stored.

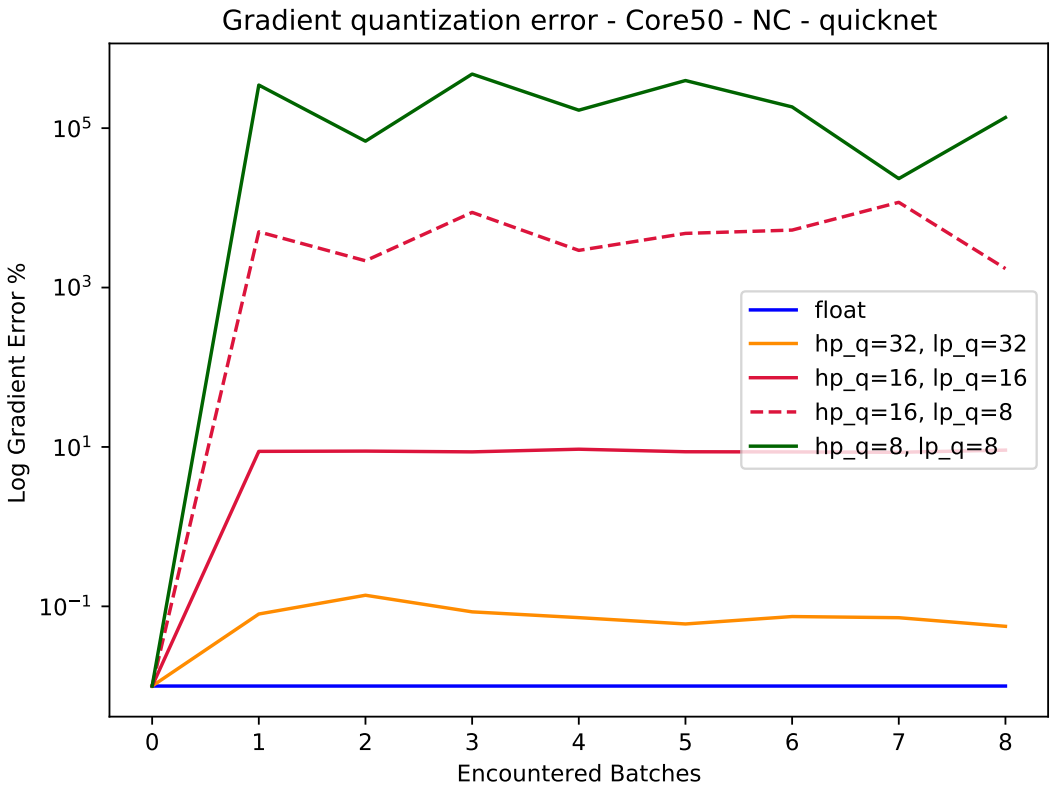
Gradient quantization inevitably introduces an approximation error that can affect the accuracy of the model; to check the amount of approximation for different quantization levels, for each mini-batch, we compute the Mean Absolute Error (MAE, in percentage) between the floating point gradient and the quantized one for the weight tensor of the CWR* layer (for the dataset CORe50 [92]). The MAE is then accumulated for all training mini-batches of each experience, as shown in Figure 4.3, where a logarithmic scale is applied. In order to evaluate only the quantization error introduced, both floating-point and quantized gradients are computed starting from the same W_k^i weights (Eq. 4.9). The plot curves of Figures 4.3a and 4.3b refer respectively to the *quicknet* [8] and *realtobinary* [95] models; it is evident that the quantization error introduced using the lp_q with 8 bits is much larger compared to higher quantization schemes (16/32 bits or floating point) whose gap w.r.t. the floating point implementation is quite low, as pointed out in section 4.1.4.

4.1.3 Experiments

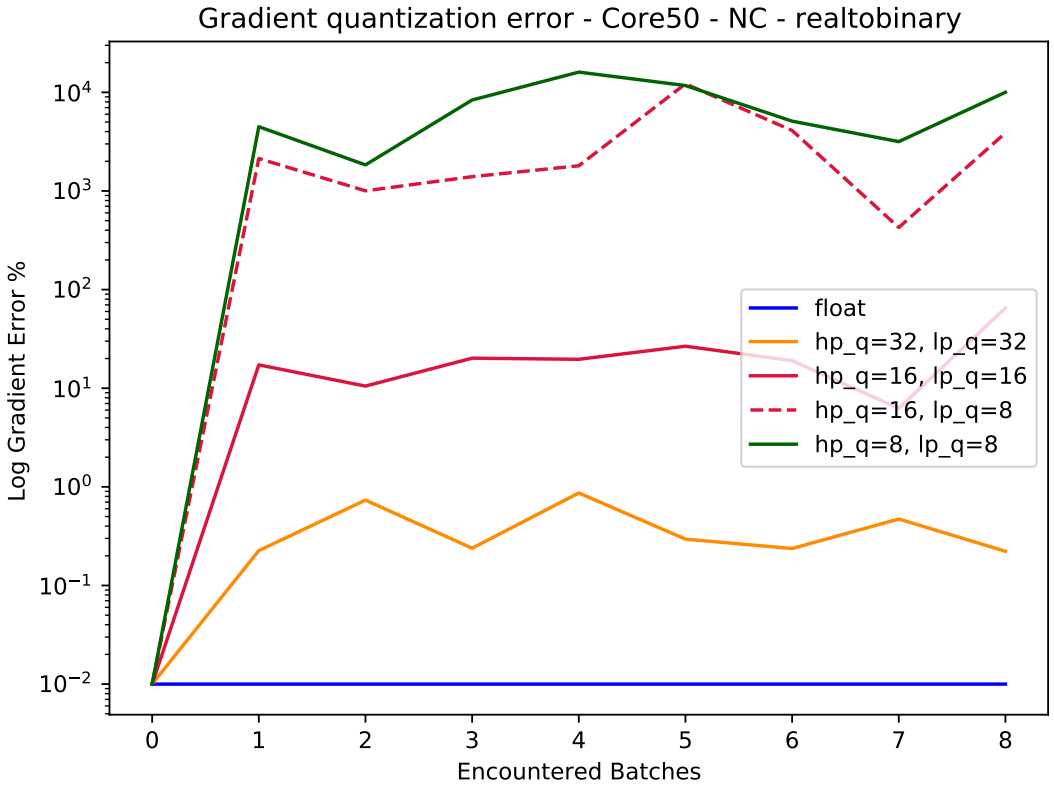
We evaluate the proposed approach on three classification datasets: CORe50, CIFAR10 and CIFAR100 with different BNN architectures. The BNN models employed for CORe50 have been pre-trained on ImageNet [119] and taken from Larq repository[2]; instead, the models used for CIFAR10 and CIFAR100 have been pre-trained on Tiny Imagenet[3]. For each dataset, we conducted several tests using a different number of quantization bits with the same training procedure. Our work is targeting a model that could continuously learn and therefore we limited the number of epochs to 10 for the first experience and to 5 for the remaining. The results of Eq. 4.7 and 4.9 require the adoption of the Cross Entropy as loss function and the Stochastic Gradient Descent (SGD) as optimizer; the choice of SGD is encouraged as it requires a simple computation with a limited overhead compared to the Adam [68] optimizer. The binarization of weights and activations always happens at training time using an approximation of the gradient (STE introduced in Section 2.2 or derived solutions that are model dependent) for *sign* function.

Hereafter we provide some details on the BNN models employed, the augmentation pipeline and the CL protocols adopted for each dataset:

[2] https://docs.larq.dev/zoo/api/sota/
[3] http://cs231n.stanford.edu/tiny-imagenet-200.zip



(A) *quicknet*



(B) *realtobinary*

FIGURE 4.3: Accumulation of gradient quantization errors (Mean Absolute Error in percentage using a logarithmic scale) between quantized and floating-point versions for each experience. During the first experience the gradient computation is always executed in floating-point.

CORe50 [92] It is a dataset specifically designed for Continuous Object Recognition containing a collection of 50 domestic objects belonging to 10 categories. The dataset has been collected in 11 distinct sessions (8 indoor and 3 outdoor) characterized by different backgrounds and lighting. For the continuous learning scenarios (NI, NC) we use the same test set composed of sessions #3, #7 and #10. The remaining 8 sessions are split in batches and provided sequentially during training obtaining 9 experiences for NC scenario and 8 for NI. No augmentation procedure has been implemented since the dataset already contains enough variability in terms of rotations, flips and brightness variation. The input RGB image is standardized and rescaled to the size of $128 \times 128 \times 3$.

CIFAR10 and CIFAR100 [71] Due to the lower number of classes, the NC scenario for CIFAR10 contains 5 experiences (adding 2 classes for each experience) while 10 are used for CIFAR100. For both datasets the NI scenario is composed by 10 experiences. Similar to CORe50, the test set does not change over the experiences. The RGB images are scaled to the interval $[-1.0\,;+1.0]$ and the following data augmentation was used: zero padding of 4 pixels for each size, a random 32×32 crop and a random horizontal flip. No augmentation is used at test time.

On CORe50 dataset, we evaluated the three binary models reported below:

Realtobinary [95] This network proposes a real-to-binary attention matching mechanism that aims to match spatial attention maps computed at the output of the binary and real-valued convolutions. In addition, the authors proposed to use the real-valued activations of the binary network before the binarization of the next layer to compute scaling factors, used to rescale the activations produced after the application of the binary convolution.

Quicknet and QuicknetLarge[8] This network follows the previous works [88, 11, 95] proposing a sequence of blocks, each one with a different number of binary 3×3 convolutions and residual connections over each layer. Transition blocks between each residual section halve the spatial resolution and increase the filter count. QuicknetLarge employs more blocks and feature maps to increase accuracy.

For CIFAR10 and CIFAR100 datasets, whose input resolution is 32×32, we evaluated the following networks (pre-trained on Tiny Imagenet):

BiRealNet[88] It is a modified version of classical ResNet that proposes to preserve the real activations before the sign function to increase the representational capability of the 1-bit CNN, through a simple shortcut. Bi-RealNet adopts a tight approximation to the derivative of the non-differentiable sign function with respect to activation and a magnitude-aware gradient to update weight parameters. We used the instance of the network that uses *18-layers*[4].

ReactNet[90] To further compress compact networks, this model constructs a baseline based on MobileNetV1 [57] and add shortcut to bypass every 1-bit convolutional layer that has the same number of input and output channels. The 3×3 depth-wise and the 1×1 point-wise convolutional blocks of MobileNet are replaced by the 3×3 and 1×1 vanilla convolutions in parallel

[4]Refer to the following `https://github.com/liuzechun/Bi-Real-net` repository for all the details.

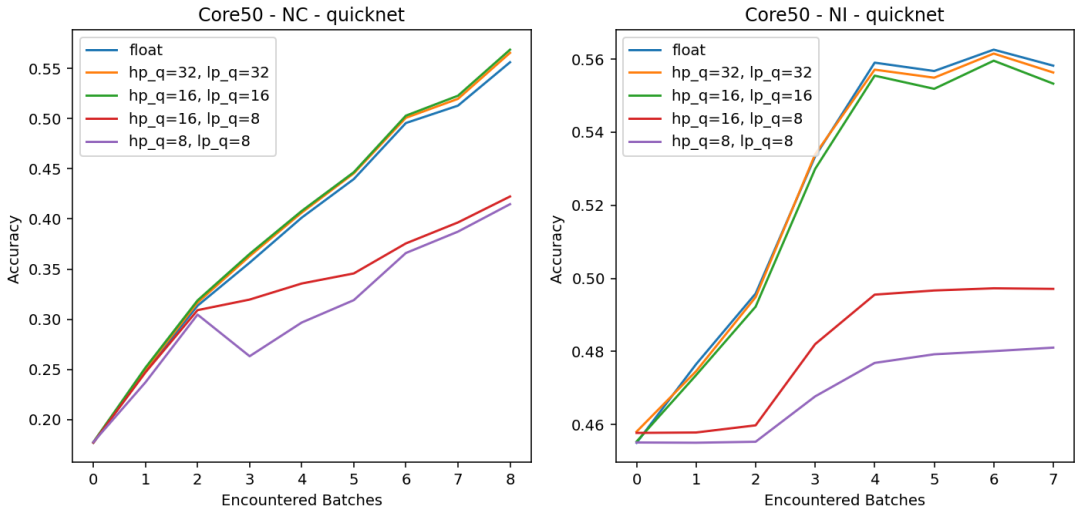
with shortcuts in React Net[5]. As for Bi-Real Net, we tested the version of React Net that uses *18-layers*.

Our tests were performed on both the NI and NC scenario (discussed in Section 4.1.1). Figures 4.4, 4.5 and 4.6 summarize the experimental results. On CORe50 dataset (Figure 4.4) NC scenario, the quantization scheme lp_8 gets a consistent accuracy drop over the experiences showing a limited learning capability; instead, the quantizations with lp_16 and lp_32 reach the same accuracy level of the floating point model. A similar situation can be observed in the NI scenario with the exception of the QuicknetLarge model where the lower quantization schemes are not able to increase the accuracy of the first experience. For datasets CIFAR10 and CIFAR100 (Figure 4.5 and 4.6) we find similar results for the NI scenario, where the 8-bit quantization scheme limits the learning capability of the model during the experiences. Instead, in the NC scenario, both Bi-Realnet and Reactnet models with lp_8 quantization, are able to reach an accuracy result close to the floating-point model. From our analysis, it appears that the 8-bit quantization of the gradients limits noticeably the learning ability of a binary model when employed in a continual learning scenario for CWR* method. To reach an accuracy comparable to a floating point implementation we devise the adoption of at least 16 bits both for lp and hp; it is worth noting that the computational effort of 16 bits is anyway limited in CWR* because the quantization is confined to the last classification layer.

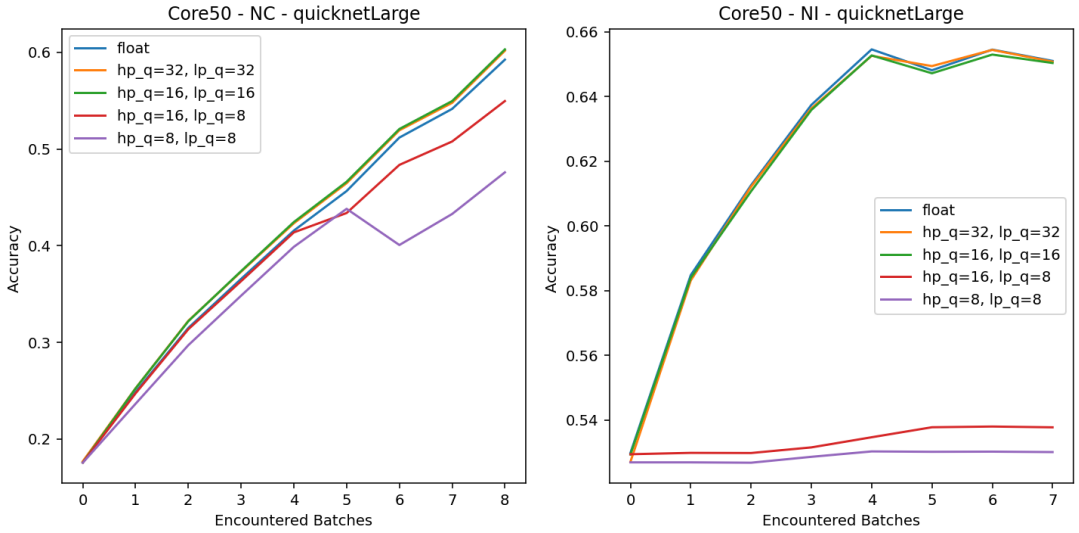
4.1.4 Results and Remarks

On-device training (or adaptation) can play an essential role in the IoT, enabling the large adoption of deep learning solutions. In this section, it has been evaluated the implementation of CWR* on edge-devices, relying on binary neural networks as backbone and proposing an ad-hoc quantization scheme. By using an 8-bit quantization bit width, the learning capability of the model degrades too much, while 16 bits represents a good compromise.

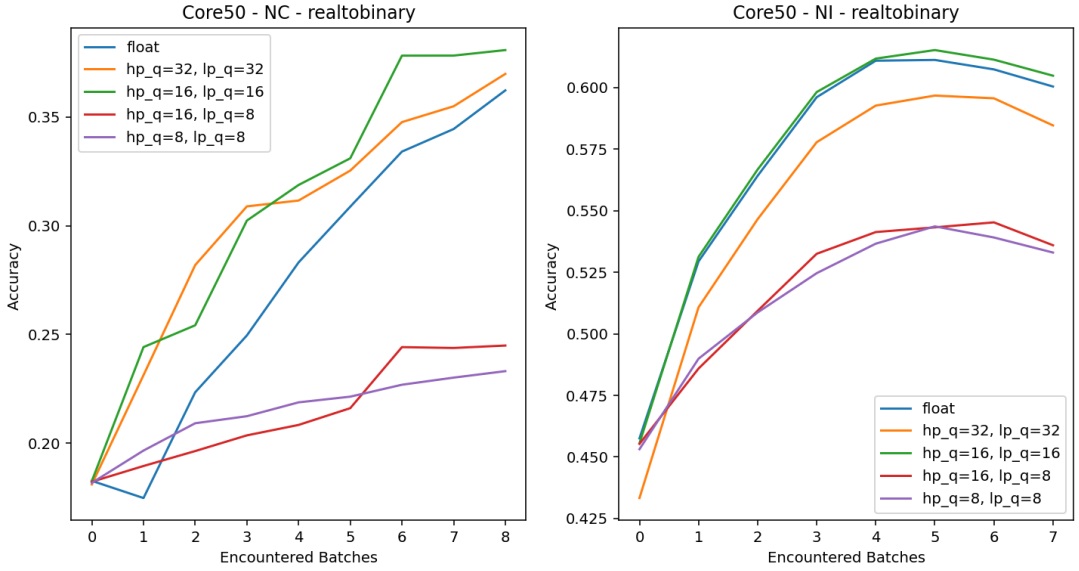
[5]Refer to the following `https://github.com/liuzechun/ReActNet` repository for all the details.



(A) Quicknet

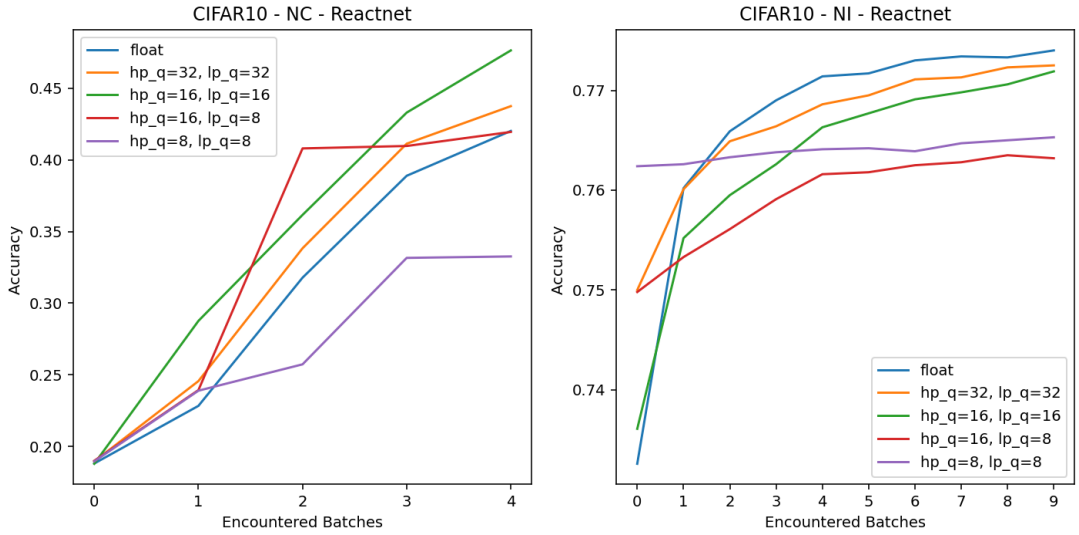


(B) QuicknetLarge

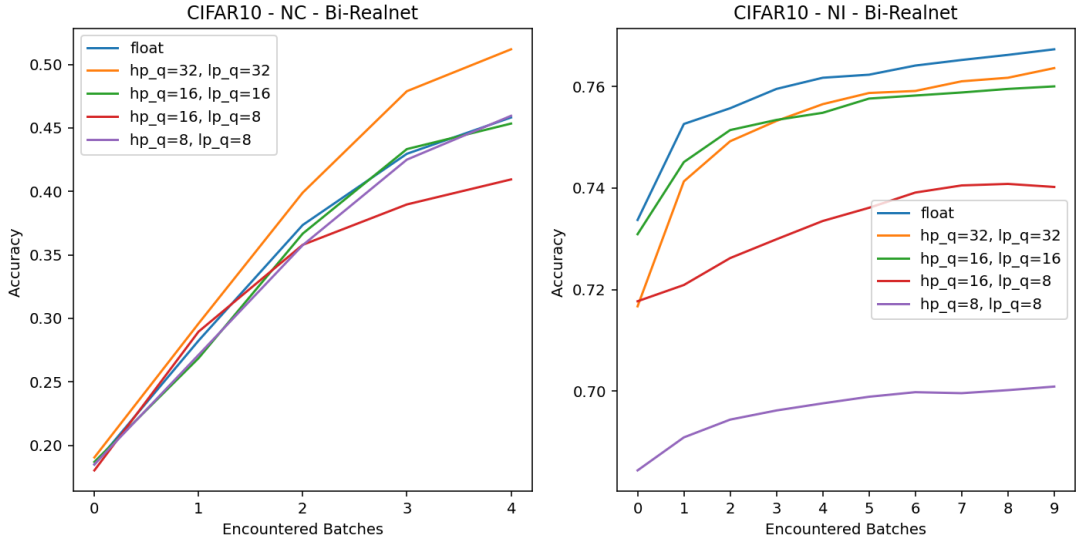


(C) Realtobinary

FIGURE 4.4: CORe50 accuracy results using different quantization methods.

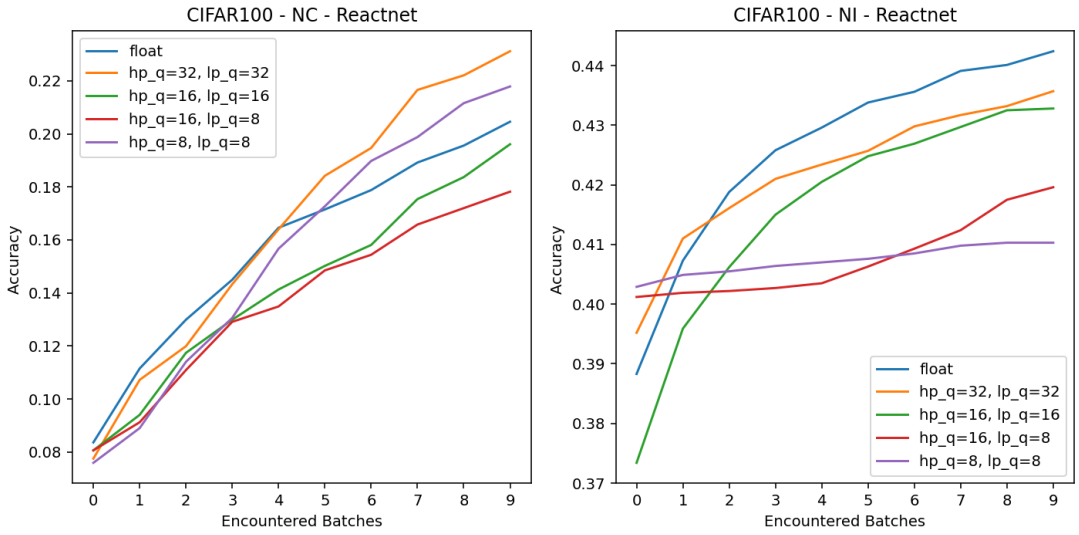


(A) Reactnet

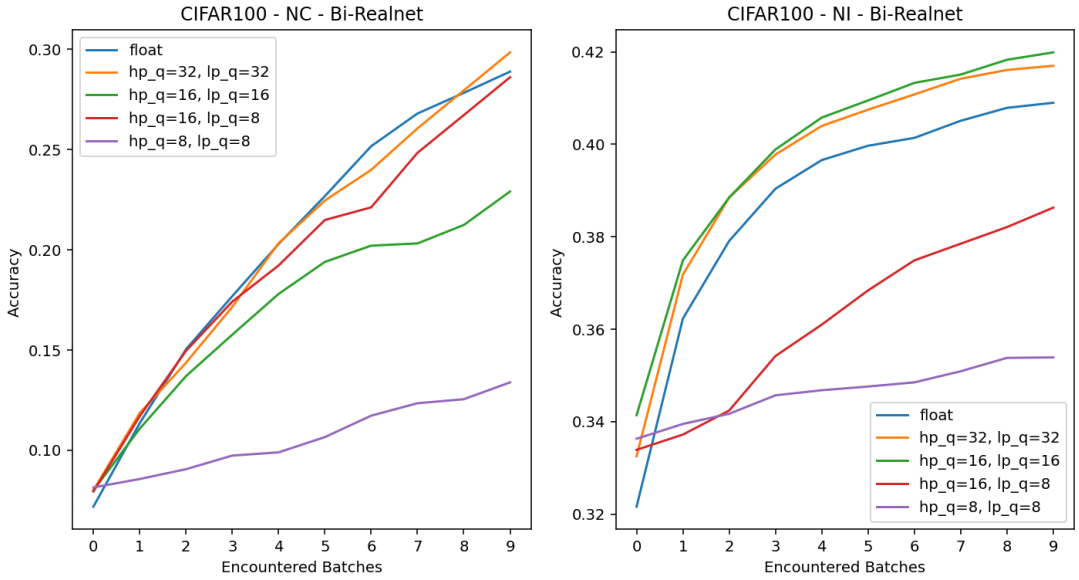


(B) Bi-Realnet

FIGURE 4.5: CIFAR10 accuracy results using different quantization methods.



(A) Reactnet



(B) Bi-Realnet

FIGURE 4.6: CIFAR100 accuracy results using different quantization methods.

4.2 Enabling On-device Continual Learning with Binary Neural Networks and Latent Replay

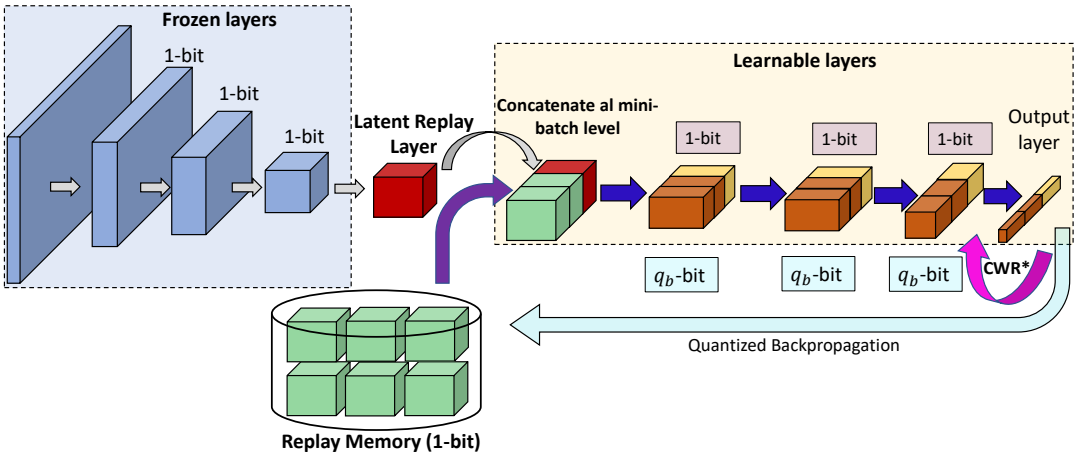


FIGURE 4.7: Continual Learning with latent replay memory. When using a BNN the activations stored in the replay memory can be quantized to 1-bit.

The previous section 4.1 explored the possibility of training a BNN model on-device by freezing the binary backbone and allowing the adaptation of only the last classification layer, where forgetting is mitigated by CWR* [93, 42]. Unfortunately, the reported results are interesting but the final accuracy is significantly lower w.r.t. a system where all the layers can be tuned. In this section, we present a solution that enables on-device training while maintaining competitive performance. Specifically, our approach leverages binary latent replay (LR) activations and an improved quantization scheme that reduces the number of bits required for gradient computation.

Pellegrini et al. [107] showed that a good accuracy/efficiency tradeoff in CL can be achieved by continuously training only some convolutional layers (typically from 3 to 5), placed before the classification head. Replaying part of old data (stored in a replay memory or buffer), interleaved with new samples, was proved to be an effective approach to mitigate catastrophic forgetting [69]. If past samples are stored as intermediate activations (instead of raw data), the replay technique takes the name *latent replay* [107] (see Figure 4.7). Latent replay is particularly interesting when combined with BNN (as proposed in this section) since the latent activations can be quantized to 1-bit, leading to a remarkable storage saving. Unfreezing some intermediate layers requires to back-propagate gradients along the model to update weights; on the edge, the implementation of this process, usually referred to as *on-device* learning, requires an efficient and lightweight back-propagation implementation, which is not yet available in the most popular training frameworks. The reduction of bitwidths during backward pass, made possible by a fixed point (many low-power CPUs are not equipped with floating-point unit) implementation, can speedup the learning phase but the tradeoffs between accuracy loss and efficiency need to be evaluated with attention.

We propose a solution to combine the Continual Learning paradigm with training on the edge using BNNs. Specifically, through the introduction of a back-propagation and input binarization algorithm, we demonstrate how it is possible to continuously tune a CNN model (including classification head and convolutional layers) with low memory requirements and high efficiency. Our work represents a step beyond the

classical quantization approach of BNNs published in the literature, where binarization is typically considered only during forward pass and a binary model is trained using latent floating-point weights [54]. Some works showed [16, 83] good improvements in reducing both the memory demand and the computational effort to enable training on the edge, but they did not focus on the Continual Learning (CL) scenario, which we primarily address. We conducted experiments with multiple BNN models, evaluating the advantages offered by the proposed methodology in comparison to the method outlined in section 4.1, where only the classification head is tuned.

The main contributions of this section can be summarized as follows:

1. **Reduced Replay Memory Requirement**: our replay memory stores intermediate activations quantized to 1-bit allowing a relevant storage saving. We investigate the trade-offs required to maintain model accuracy while simultaneously reducing memory consumption.
2. **Improved Model Accuracy**: by enabling the continual adaptation of intermediate convolutional layers (besides the final classification head) our BNN-based model significantly outperforms the closest previous solution [138].
3. **Quantization of Backpropagation for Non-Binary layers**: we introduce a quantization approach for the back-propagation step in non-binary layers, enabling the preservation of accuracy while eliminating floating-point operations.
4. **Optimized Binary Weight Quantization**: we present an optimized quantization strategy tailored for binary weights, leading to a remarkable $8\times$ reduction in memory requirements. Binary layers are typically trained by storing latent floating-point representations of weights that are subsequently binarized during inference. Replicating this schema on-device would result in an unacceptable increase of memory usage and computational overhead.
5. **Optimized Back-Propagation Framework**: we implemented a comprehensive back-propagation framework capable of supporting various quantization levels both inference and back-propagation stages.

In the next sections we describe the latent replay mechanism (section 4.2.1) providing an estimation of the memory saved when applied to a binary layer, the quantization approach (section 4.2.1) used for both forward and backward passes and a comprehensive experimental evaluation (section 4.2.2), focusing on the accuracy comparison w.r.t. the CWR* algorithm (section 4.2.2), the reduction of the storage needed by the replay memory (section 4.2.2) and the efficiency in the backpropagation algorithm (section 4.2.2).

4.2.1 Method

In this section, we detail our contributions to efficiently deploy CL methods using Latent Replay and BNNs. In particular, the CWR* approach (briefly summarized in Algorithm 3) is used to correct class-bias in the classification head.

Continual Learning with Latent Replays

In Figure 4.7 we illustrate the CL process with Latent Replay. When new data becomes available, they are fed to the neural network that during the forward pass produces their latent activations, which represent the feature maps corresponding to a specific intermediate layer. We denote this layer as l (where $l \in [0, L)$), with L

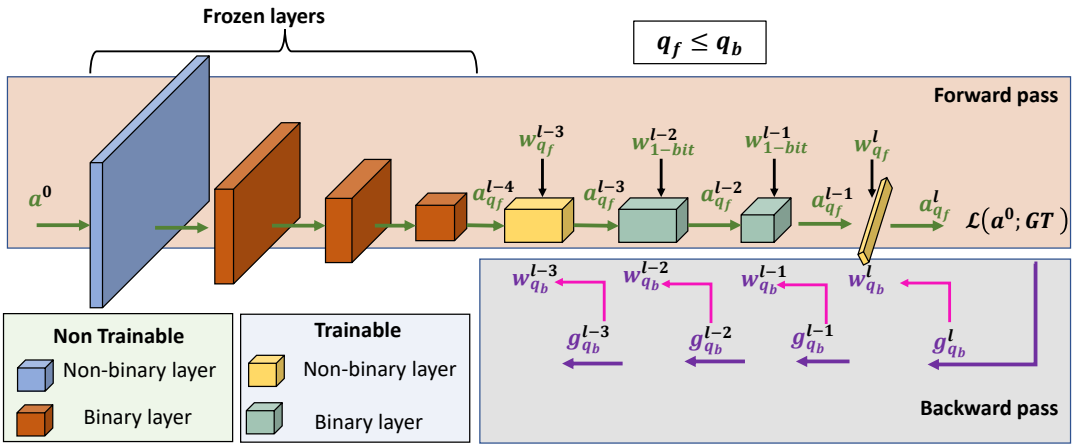


FIGURE 4.8: Quantization scheme that uses a different number of bitwidth for forward (q_f) and backward (q_b) pass. Usually, trainable non-binary layers are Batch Normalization [60], Addition and Concatenation layers.

representing the total number of layers within the model. Activations of new data are joined (at minibatch level) with the replay activations (previously stored) and forward/backward passes on the remaining layers, specifically those with index from $l+1$ to $L-1$. To elucidate further, if B_N denotes the minibatch size of the newly acquired latent activations, a subset of replay vectors (B_R) is extracted from the replay memory and merged, thus forming a minibatch of total size $B_T = B_N + B_R$. In contrast, the layers with an index less than l are maintained in a frozen state and are not included in the learning process. After the conclusion of each training experience, the replay memory is updated by including samples from the last experience and using class-balanced reservoir sampling [136], which ensures a double balancing: (i) in terms of samples per classes, (ii) in terms of samples from experience (see Algorithm 4).

Quantization of activations and weights

Quantization techniques have gained widespread adoption to diminish the data size associated with model parameters and the activations of layers. Employing quantization strategies enables the reduction of data bitwidth from the conventional 32-bit floating-point representation to a lower bit-precision format, typically 8 bits or even less, while typically incurring a negligible loss in accuracy during the forward pass of the model. For the quantization of non-binary layers that need to be trained on-device, we adopted the same approach proposed in section 4.1.2.

By representing the dynamic range of the activations at the i-th layer of the network as $[a^i_{min}, a^i_{max}]$, we can define the quantized activations a_q as:

$$a^i_q = cast\_to\_q\lfloor \frac{a^i}{S^i_a} \rceil, \quad S^i_a = \frac{a^i_{max} - a^i_{min}}{2^q - 1} \tag{4.12}$$

where q denotes the number of quantization bits used $(8, 16, 32)$, a^i represents the full-precision activations and a^i_{max}, a^i_{min} are determinated through calibration on the training dataset. Weight quantization can be accomplished using an equation analogous to equation 4.12. However, as recommended in section 4.1, we utilize two separate sets of quantization bits for both the forward and backward passes. For binary layers, during the forward pass, binarization is executed using equation 2.2,

Algorithm 4 Procedure used to populate the replay memory (RM). RM is initially pre-populated using training samples of the first experience. The reservoir sampling is used on a class basis to maintain the balance among different classes. This approach prevents a skewed representation of classes within RM.

Input: N = max number of samples per class
Input: C = max number of classes
$RM_{size} = C \cdot N$ {$C \cdot N$ is the max size of RM populated during the first experience.}

for all on-device experience **do**
  T is the number of classes
  M_t = samples of class t
  RM_t = samples of class t already in RM

  for $t = 0$ **to** $T - 1$ **do**
    $B_t = RM_t \cup M_t$ {# is the cardinality operator}
    RM_t^{new} = apply Reservoir sampling to extract $\#RM_t$ samples from B_t
    remove not selected RM_t samples
    update RM with RM_t^{new}
  end for
end for

	# total weights	LR shape	# B = binary weights after LR	# NB = non-binary weights after LR	$\frac{B}{B+NB}$
BiReal-18	11.2M	(4, 4, 512)	7.0M	19K	99.7%
BiReal-18	11.2M	(8, 8, 256)	10.1M	28K	99.7%
React-18	11.1M	(4, 4, 512)	7.0M	18K	99.7%
React-18	11.1M	(8, 8, 256)	8.3M	24K	99.7%
VGG-Small	4.6M	(8, 8, 512)	2.3M	86K	96.4%
QuickNet	12.7M	(7, 7, 256)	9.5M	36K	99.6%
QuickNet	12.7M	(14, 14, 128)	11.9M	43K	99.6%
QuickNetLarge	22.8M	(7, 7, 256)	14.2M	40K	99.7%
QuickNetLarge	22.8M	(14, 14, 128)	21.3M	56K	99.7%

TABLE 4.1: The table represents a comparison of memory usage (# parameters) for different BNN models. With B we report the number of binary weights that can be updated during back-propagation; with NB the number of non-binary weights. The choice of latent replay (LR) level is discussed in Section 4.2.2. It is worth noting that the largest part of memory weights is used by binary weights.

as proposed in [23]. In backward pass, STE (Equation 2.7) computes the derivative of sign as if the binary operation was a linear function. This approximation has been further improved by other works [88, 90] and in general it is model dependent.

Quantized Backpropagation

Drawing upon the findings presented in the works of Gupta et al. [46], Das et al. [25], and Banner et al. [7], it is evident that the quantization of gradients stands out as the primary contributor to accuracy degradation during the training process. Therefore, we advocate for a quantization scheme akin to that introduced in [138]. In this scheme, we employ two distinct sets of quantization bits for the forward and backward passes.

The back-propagation algorithm operates in an iterative manner to calculate the gradients of the loss function (denoted as $\mathcal{L}$) with respect to the input a^{l-1} for the layer l:

$$g^l = \frac{\partial \mathcal{L}}{\partial a^l} \tag{4.13}$$

starting from the last layer. Every layer in the network is tasked with computing two sets of gradients to execute the iterative update process. The first set corresponds to the layer activation gradient w.r.t. the inputs, which serves the purpose of propagating gradients backward to the previous layer. Considering a linear layer, where $a^l = W^l \cdot a^{l-1}$ and $\frac{\partial a^l}{\partial a^{l-1}} = W^l$, the gradients can be computed as follows:

$$g^{l-1} = \frac{\partial \mathcal{L}}{\partial a^l} \cdot \frac{\partial a^l}{\partial a^{l-1}} = W^l g^l \tag{4.14}$$

The other set is used to update the weights of layer index l:

$$g_w^l = \frac{\partial \mathcal{L}}{\partial a^l} \cdot \frac{\partial a^l}{\partial W^l} = a^{l-1} g^l \tag{4.15}$$

Based on Eq. 4.14 and 4.15, the backward pass requires approximately twice Multiply-And-Accumulate (MAC) operations compared to the forward pass and therefore the gradient quantization becomes essential to efficiently train neural network models on-device. The quantization of weights and gradients (Eq. 4.14 and 4.15) is implemented through Eq. 4.12 and can be visually summarized in Figure 4.8; as shown in [7, 138], backward pass usually needs higher bitwidth to preserve the directionality of the weight tensor and, based on that, we propose to use lower bitwidth during forward pass (Figure 4.8, q_f bits, green path) to minimize latency and more bits for the backward pass to be more accurate in gradient representation (Figure 4.8, q_b bits, purple path). Considering the constrained memory resources available on embedded devices, accurately estimating the memory requirements of the learning algorithm becomes imperative. We can categorize memory into two distinct types: the memory utilized by the CL method (*e.g.* the replay memory) and the memory necessary to store intermediate tensors during the forward pass, which are subsequently used in the backpropagation, along with the model weights. In this context, we will focus mainly on the latter aspect, particularly for binary layers where q_f is fixed at 1-bit while q_b can vary depending on the desired level of accuracy. In Table 4.1, we present an assessment of the memory usage for representing binary weights of trainable layers on-device. It is worth noting that binary weights, as indicated in the fifth column of the same table, constitute a substantial portion of the total model parameters. Consequently, reducing q_b to 1-bit offers significant memory savings in comparison to a more conventional approach where q_b is set to 16 bits. The reduction in memory usage exhibits an almost linear relationship with the number of bits utilized. We distinguish q_b between binary and non-binary layers to apply different quantization bitwidts, as elaborated in Section 4.2.2, which demonstrates that it is feasible to maintain accuracy while significantly reducing q_b for binary layers. Denoting q_b^{bin} and $q_b^{non-bin}$ as the quantization settings for binary and non-binary layers, respectively, in Section 4.2.2 we illustrate that setting q_b^{bin} to 1-bit results in minimal accuracy loss compared to higher quantization bitwidths.

4.2.2 Experiments

We evaluate our methods on three classification datasets: CORe50[92], CIFAR10 [71] and CIFAR100[71] with different BNN architectures. The BNN models employed for CORe50 have been pre-trained on ImageNet through the Larq repository[6]; differently, the models used for CIFAR10 and CIFAR100 have been pre-trained on Tiny-ImageNet[74]. For each dataset, we conducted several tests using a different number of quantization bits (both for forward and backward passes) with the same training procedure. In addition to the work of section 4.1, in these experiments we kept different bitwidths for binary and non-binary layers because, as reported in Table 4.1, memory of trainable binary weights is predominant.

The details about the BNN models characteristics, the dataset benchmarked and related CL protocols can be found at section 4.1.3. On CORe50 dataset, we evaluated the Quicknet and QuicknetLarge models while on CIFAR10 and CIFAR100, whose input resolution is 32×32, we evaluated Bi-Realnet and ReActnet. In this study we did not include the *Realtobinary* [95] model, as it achieved notably lower accuracy levels that were not aligned with our research objectives and goals.

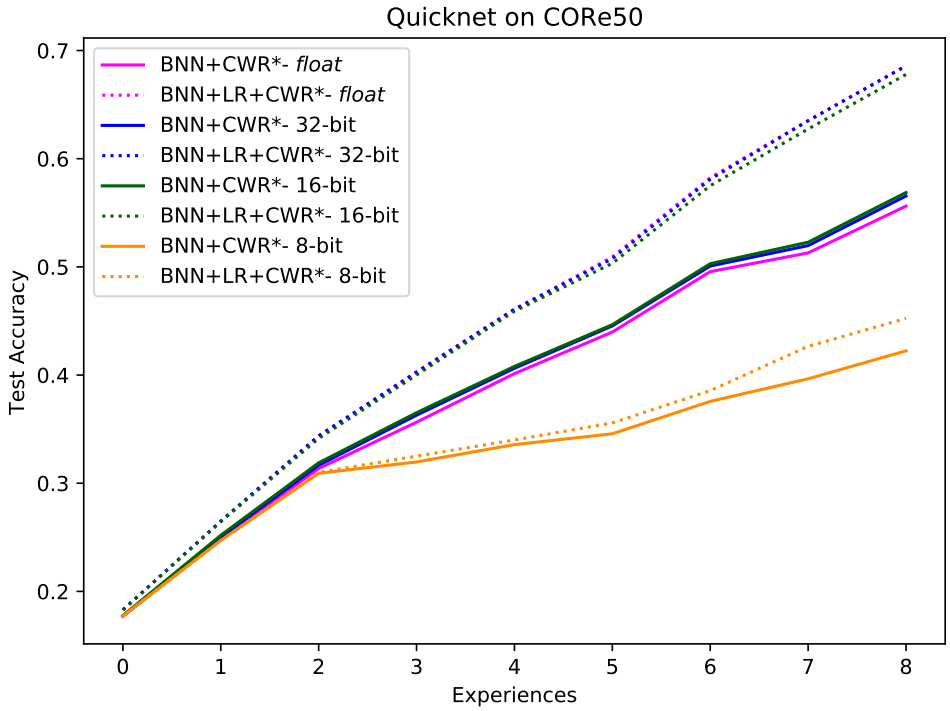
In our experimental setup, we discovered that reducing the number of epochs in each learning experience had minimal impact on model accuracy. Consequently, we empirically set the number of epochs to 5, thus constraining the training time on-device platform. Across all classification tasks, we utilized the Cross Entropy loss function in conjunction with Stochastic Gradient Descent (SGD) as the optimizer. The former was chosen due to its simplicity in derivative computation when combined with the Softmax activation function. The latter was preferred for its computational efficiency, offering lower overhead compared to more complex algorithms like Adam [68]. In our experiments, the ratio of B_N to the batch size of the latent activations sampled from the replay memory is set at $1/4$. Both weight and activation binarization were performed during training, including both the first training experience and on-device stages. This choice requires the implementation of a quantized backward pass technique for all the non-differentiable functions, specifically the binarization functions (using Eq. 4.14 and 4.12). To assess model accuracy during on-device training, we developed the quantized backward steps for all layers employed by the previously described models.

Our experiments primarily concentrated on the NC scenario. As highlighted in [107], the adoption of a latent replay memory did not significantly enhance model accuracy in the NI context. Moreover, the NC scenario more closely resembles real-world applications where the model's recognition capability must be expanded to accommodate new, previously unknown classes.

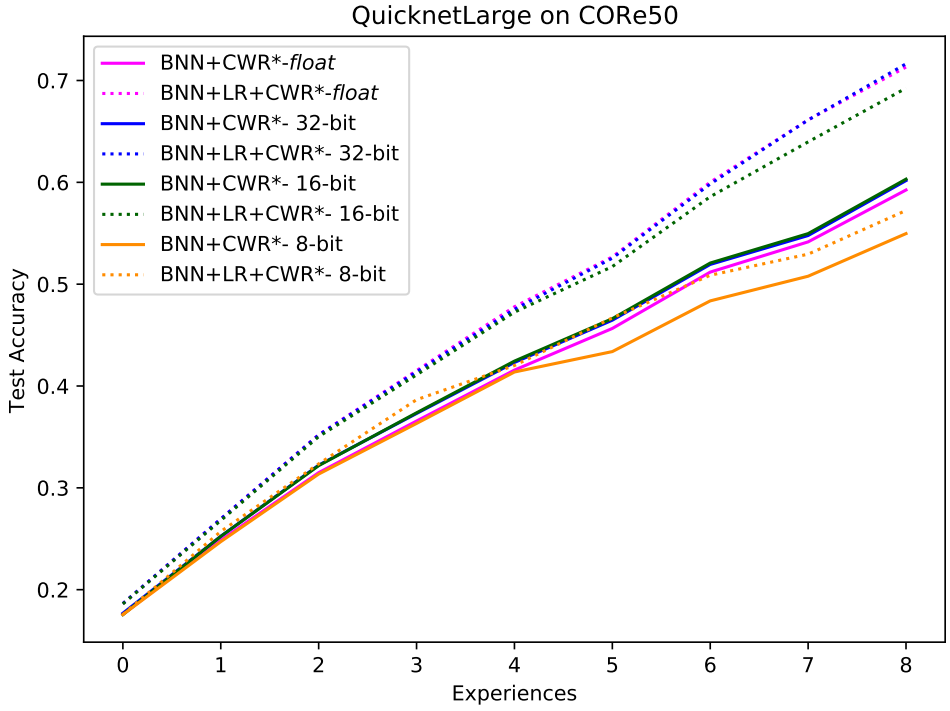
Accuracy comparison

To assess the accuracy of our solution, we initiated our evaluation by comparing it with prior work, specifically BNN+CWR* [138](section 4.1.3), where only the final classification layer is trained on-device, without employing a replay memory. We conducted a series of tests with varying quantization bitwidths for both forward and backward passes. In Figure 4.9a, 4.9b, 4.10a, 4.10b, 4.11a and 4.11b we

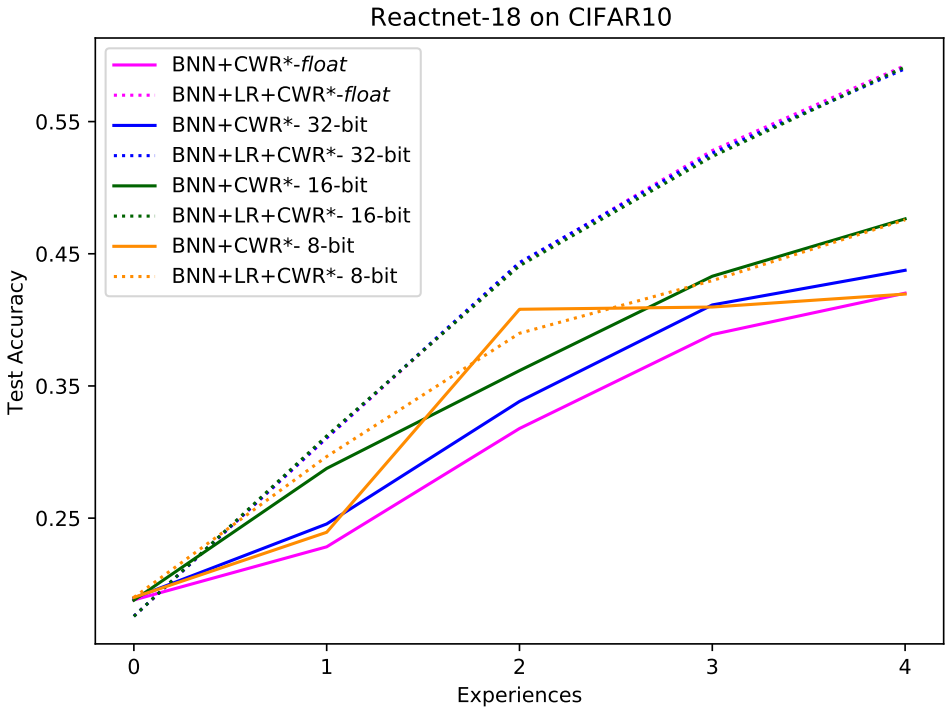
[6] https://docs.larq.dev/zoo/api/sota/



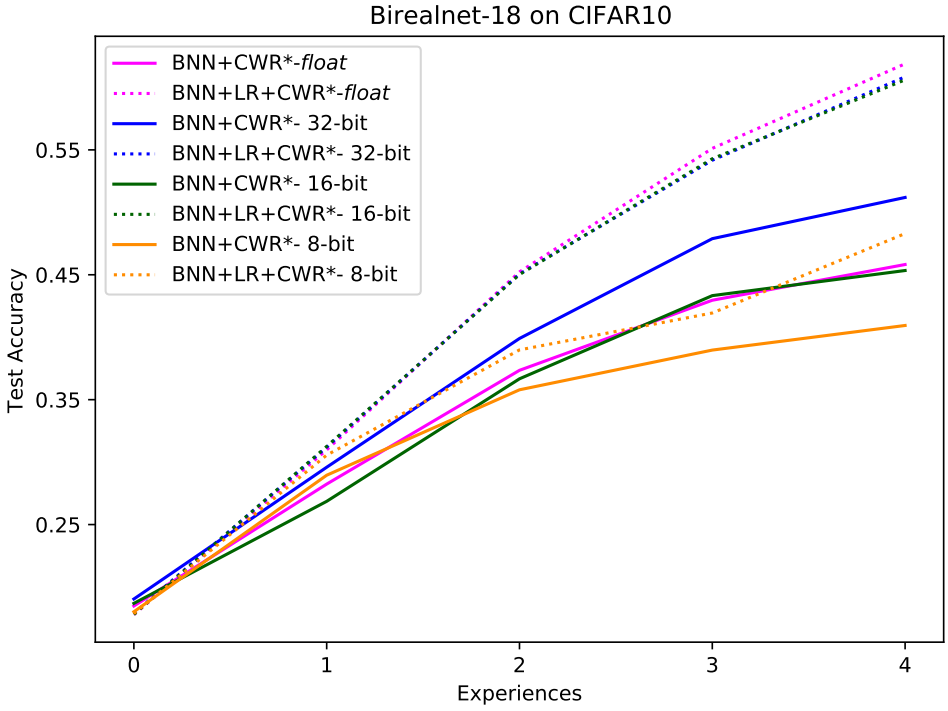
(A) Accuracy comparison of our solution (BNN+LR+CWR*) with previous work BNN+CWR* [138] on CORe50 using *quick* model.



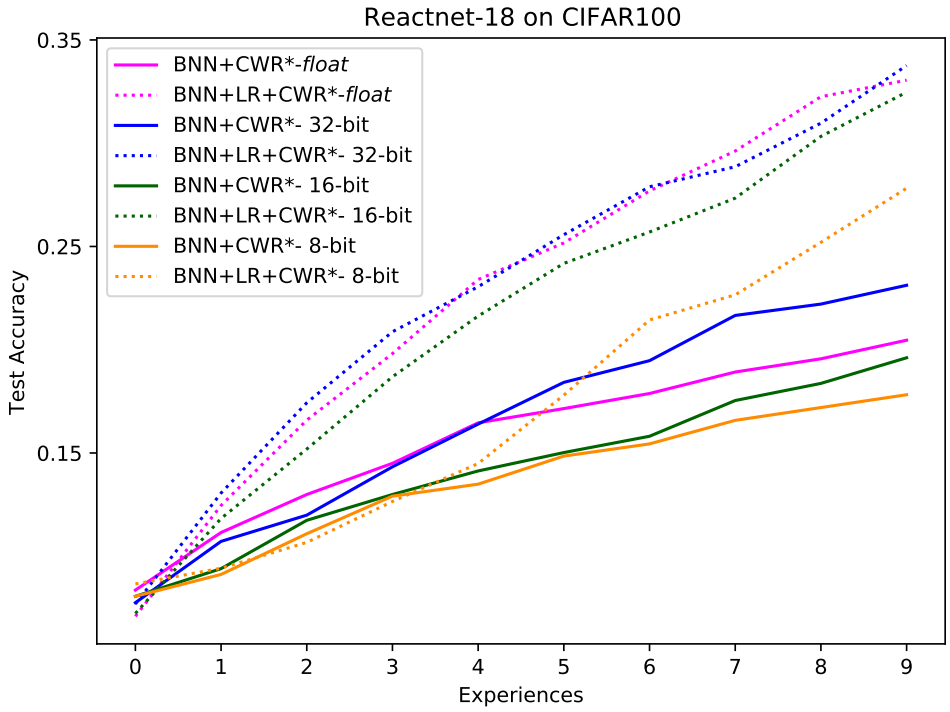
(B) Accuracy comparison of our solution (BNN+LR+CWR*) with previous work BNN+CWR* [138] on CORe50 using *QuickNetLarge* model.



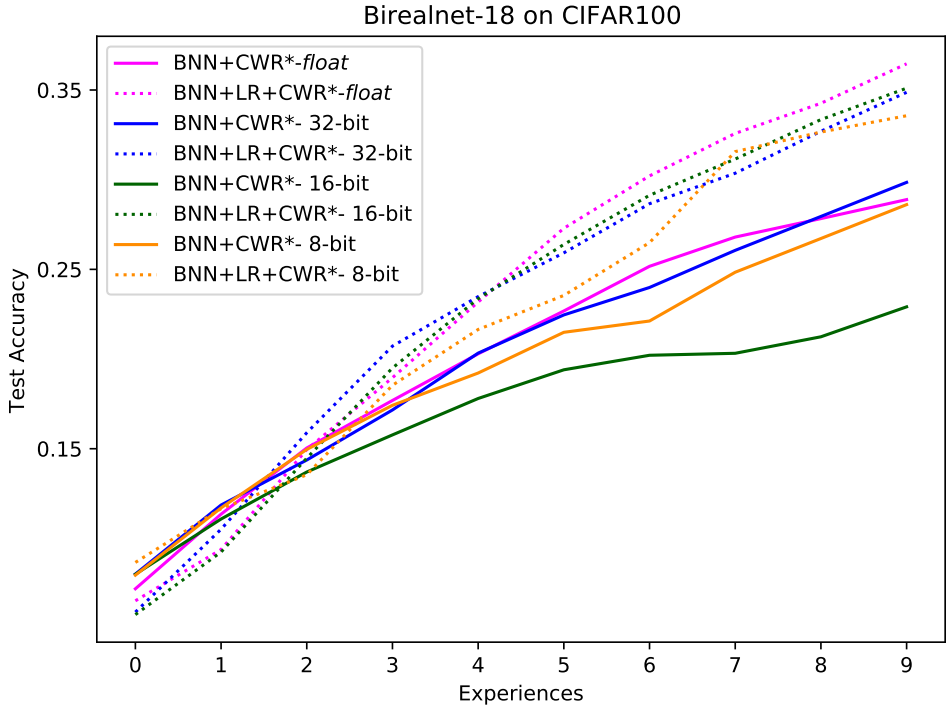
(A) Accuracy comparison of our solution (BNN+LR+CWR*) with previous work BNN+CWR* [138] on CIFAR10 using *Reactnet* model.



(B) Accuracy comparison of our solution (BNN+LR+CWR*) with previous work BNN+CWR* [138] on CIFAR10 using *Birealnet* model.



(A) Accuracy comparison of our solution (BNN+LR+CWR*) with previous work BNN+CWR* [138] on CIFAR100 using *Reactnet* model.



(B) Accuracy comparison of our solution (BNN+LR+CWR*) with previous work BNN+CWR* [138] on CIFAR100 using *Birealnet* model.

present accuracy comparisons between BNN+CWR* with the current method, denoted as **BNN+LR+CWR***, across different datasets: CORe50, CIFAR10 and CIFAR100. Each figure illustrates the performance improvement of the new method for all quantization settings tested, encompassing floating-point arithmetic, 32-bit, 16-bit and 8-bit quantized representations. It is noteworthy that, in this assessment, we applied the same quantization bitwidths (q_b) for both binary (q_b^{bin}) and non-binary ($q_b^{non-bin}$) layers during the backward pass, as BNN+CWR* does not distinguish these cases. The results consistently demonstrate that our BNN+LR+CWR* approach outperforms previous results, not only when using floating-point arithmetic but also for quantized implementations. This underscores the superior performance achieved by BNN+LR+CWR*. In our solution, we observed that employing $q_b = 8$ in BNN+LR+CWR* leads to a notable drop in accuracy compared to higher quantization bitwidth settings, aligning with the outcomes obtained by BNN+CWR*. This reaffirms the importance of using higher bitwidth representations during the backward pass to preserve model accuracy. Furthermore, the results highlight the chance to replace the standard floating-point backpropagation with a quantized version as the final accuracy is substantially comparable when enough bits are employed (16 or 32). This possibility would drastically reduce the constraints of deploying a similar solution on devices where the floating-point computation is expensive (such as FPGA) or not possible (low-power microcontrollers). For the experiments, we utilized $LR_{size} = 1500$ for CORe50, $LR_{size} = 300$ for CIFAR10 and $LR_{size} = 3000$ for CIFAR100 as our replay memory sizes.

Reducing Storage in Latent Replay

The storage requirements of the latent replay memory are closely interlinked with the bitwidths utilized to represent latent activations. As the bitwidths increase, so does the memory footprint of LR. In our approach we capitalize on the 1-bit activations inherent to BNNs to significantly mitigate the need for high-memory storage while maintaining a minimal accuracy gap, as depicted in Figure 4.12. Our experiments demonstrate that BNN models can attain a minimal accuracy gap on both CIFAR10 and CORe50 datasets, even when adopting 1-bit latent activations for LR. This translates to a huge memory reduction of 32× when compared to using floating-point latent activations. In our analysis, we considered various sizes for the LR memory, with 15, 20 and 30 elements allocated for each class. Importantly, we observed that the number of past samples in LR had a relatively minor impact on model accuracy, with the accuracy loss being within 1%. Utilizing 1-bit latent activations for LR opens the possibility to scale up applications to accommodate thousands of classes, as illustrated in Figure 4.12, thanks to the substantial reduction in memory constraints achieved.

Splitting q_b in q_b^{bin} and $q_b^{non-bin}$

As highlighted in Table 4.1, the memory footprint of BNN weights is predominantly occupied by trainable binary weights, encompassing nearly 100% of the memory. Conventionally, a binary layer is trained using latent floating-point weights [54]. However, if we were to replicate this approach on the device, it would result in a substantial increase in memory storage requirements during backpropagation stage, as it would require setting $q_b^{bin} = 32 - bit$. The quantization methodology proposed in Section 4.2.1 offers a potential solution to mitigate this constraint by reducing q_b to 8 bits. However, as depicted in Figure 4.13a and 4.13b, such a reduction in bitwidths

Model	Raspberry 3B	Raspberry 4B	Binary	Quantization q_f	Quantization q_b	Forward (ms)	Backward (ms)	Speedup
Mobilenetv2 [57]	✓			8-bit	float	340	134	1.0×
Quicknet [8]	✓		✓	1-bit	16-bit	**160**	**55**	**2.2**×
Mobilenetv2 [57]		✓		8-bit	float	225	90	1.0×
Quicknet [8]		✓	✓	1-bit	16-bit	**105**	**38**	**2.2**×

TABLE 4.2: Efficiency comparison of our method implemented on two different embedded boards, *i.e.* Raspberry Pi 3B and 4B, using Mobilenetv2 and Quicknet model. As shown, our solution achieves up to 2.2× speedup on the same platform.

would lead to a noticeable accuracy drop in the model. To address this challenge, we evaluated the impact of distinct quantization levels for binary weights (q_b^{bin}) and non-binary weights ($q_b^{non-bin}$). Specifically, we experimented with representing q_b^{bin} using both 4 bits and 1 bit. Our findings, as shown in Figure 4.13, indicate that 4-bit representation for binary layers does not introduce a substantial accuracy loss. Moreover, employing a 1-bit representation of weights during the back-propagation stage is feasible, as binary weights remain frozen during on-device learning. In this scenario, the model still effectively preserves accuracy. This latest result carries significant implications for on-device learning, as it reduces the computational burden by requiring backward steps only for non-binary layers, primarily those employing $q_b^{non-bin} = 16 - \text{bits}$, as observed in our experiments.

Efficiency Evaluation

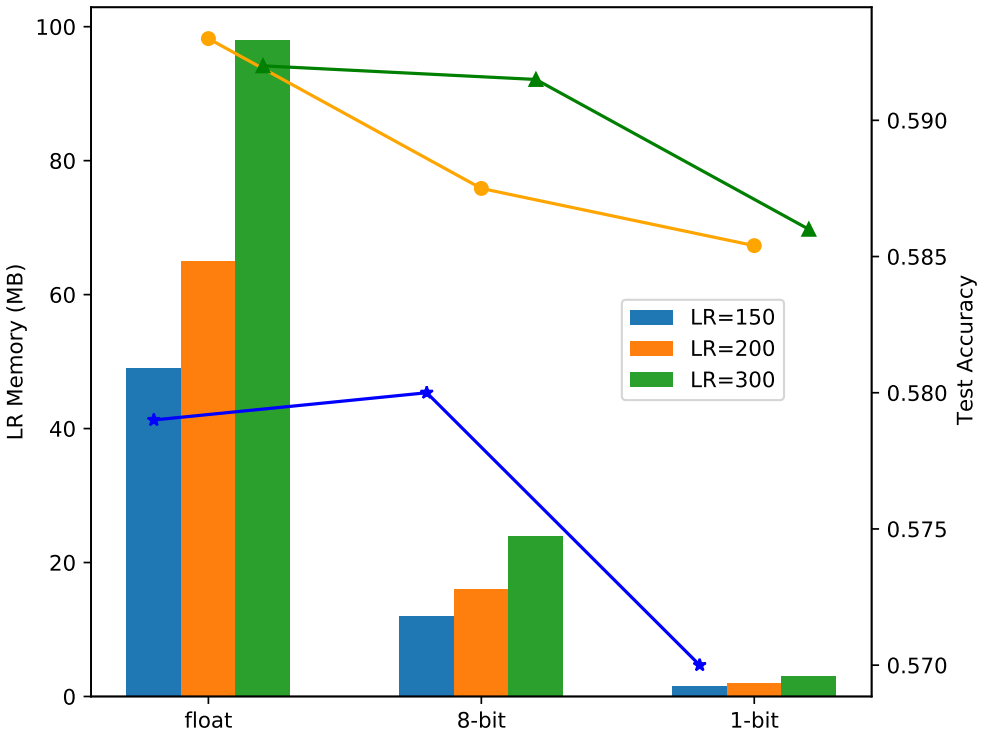
To demonstrate the applicability of our approach on real-world embedded boards, we provide an estimation analysis of the on-device performance. For this evaluation, we select two popular boards commonly used in the IoT paradigm, both based on the single-thread ARMv8 platform: Raspberry Pi 3B and Raspberry Pi 4B. Based on the efficiency analysis reported in [8, 107], we report in Table 4.2 the inference and backward timings of our BNN+LR+CWR* method compared to a non-binary solution (using a Mobilenetv2) [107]: the results obtained adopting Mobilenetv2 rely on floating-point precision for layers from LR up to the classification head. The frozen backbone is quantized using 8-bit (latent activations are stored with 8-bit precision) and executed with Tensorflow-Lite. Instead, BNN+LR+CWR* employs Quicknet model with the following quantization setting: $q_f = 8, q_b^{bin} = 8, q_b^{non-bin} = 16$; the framework used to execute binary inference is LCE [8]. The image input size considered is 224×224 and the batch size is 1. Our empirical evaluation for backward pass shows that our BNN+LR+CWR* can achieve a minimum speedup of 2.2× compared to a non-binary solution. In our evaluation we consider the worst-case scenario for backward step by setting $q_b^{bin} = 8$; instead, by setting $q_b^{bin} = 1$, the speedup reported in the last column of Table 4.2 should improve significantly.

4.2.3 Final Remarks

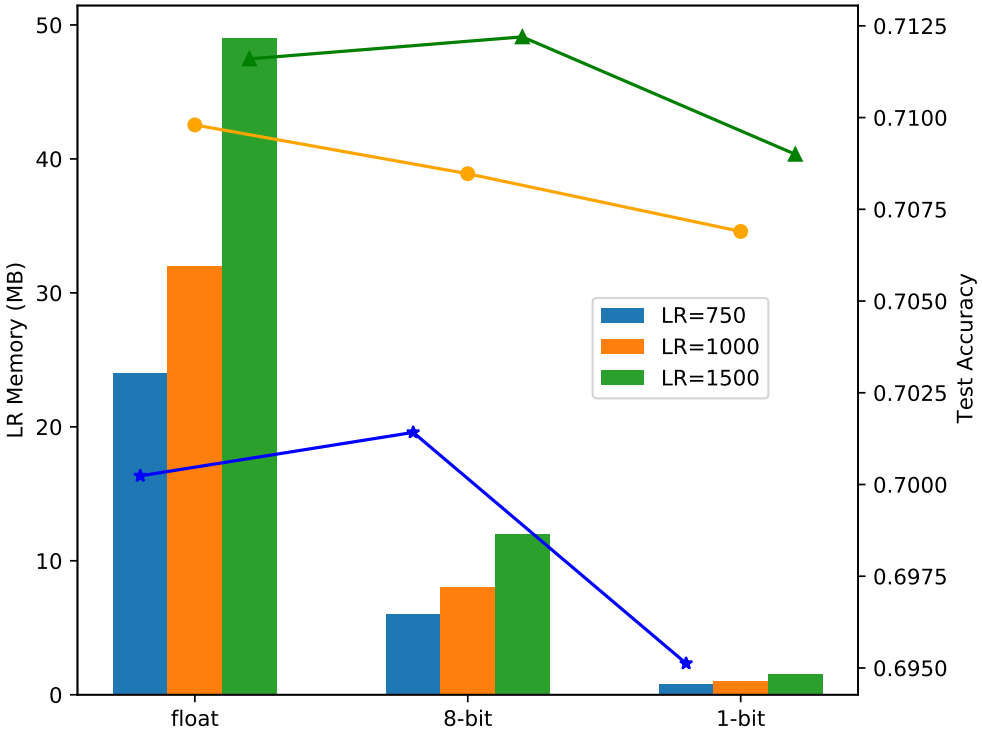
On-device training holds great potential in the realm of the IoT, as it can facilitate the widespread adoption of deep learning solutions. In this study, our primary focus was the implementation of Binary Neural Networks (BNNs) in combination with Continual Learning algorithms, an approach not yet fully investigated in the literature. In particular, we propose the use of the CWR* method with the support of a replay memory, implementing several customized quantization schemes

tailored to alleviate memory constraints and computational bottlenecks during the back-propagation stage. Summarizing, experimental achievements of this section include the following:

- **Reduced memory usage**: we significantly reduced the memory storage required for replay memory by employing 1-bit latent activations, as opposed to the state-of-the-art approach that employs 8-bit precision. A limited storage requirement is a key element in addressing on-device training, especially with embedded systems with a limited storage capability.
- **Improved model accuracy**: we improve the accuracy obtained across different binarized backbones and the BNN+CWR* approach. Specifically, we reduce the gap in performance that commonly affects BNNs by introducing a latent replay approach as a safeguard against catastrophic forgetting.
- **Efficiency in backpropagation**: we minimize the computational effort related to the backpropagation of the latent replay through a proper quantization scheme. In this manner, we combine the good performance of the model with limited computation requirements for the learning phase. This achievement, in combination with reduced memory usage, paves the way for future on-device and real-world training of learning systems.

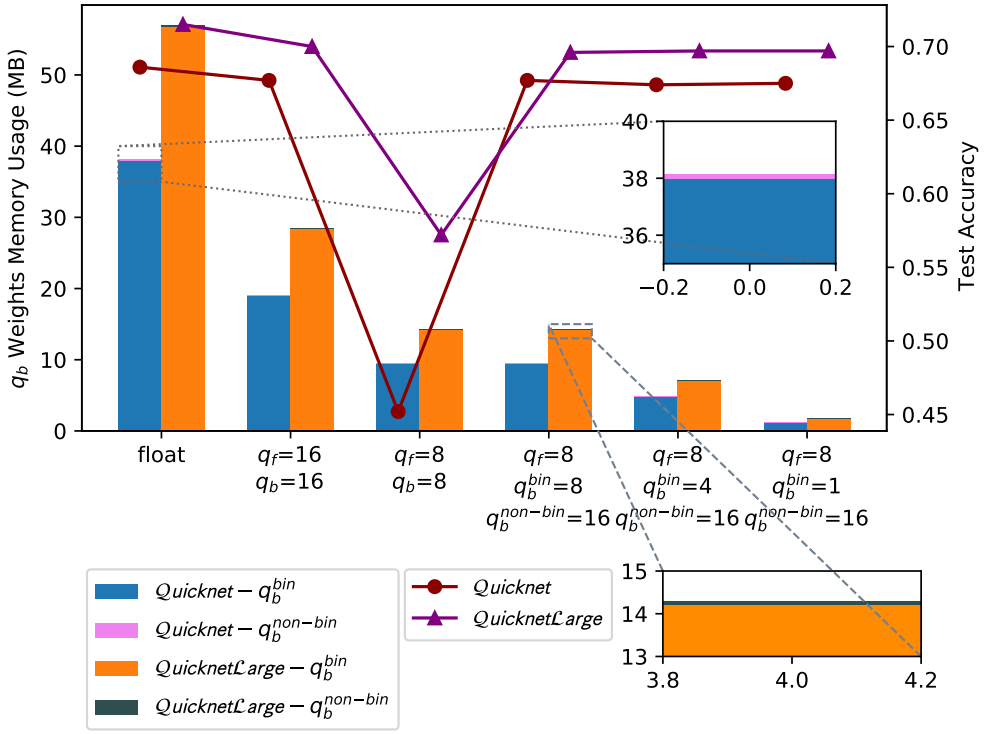


(A) Reactnet-18 on CIFAR10.

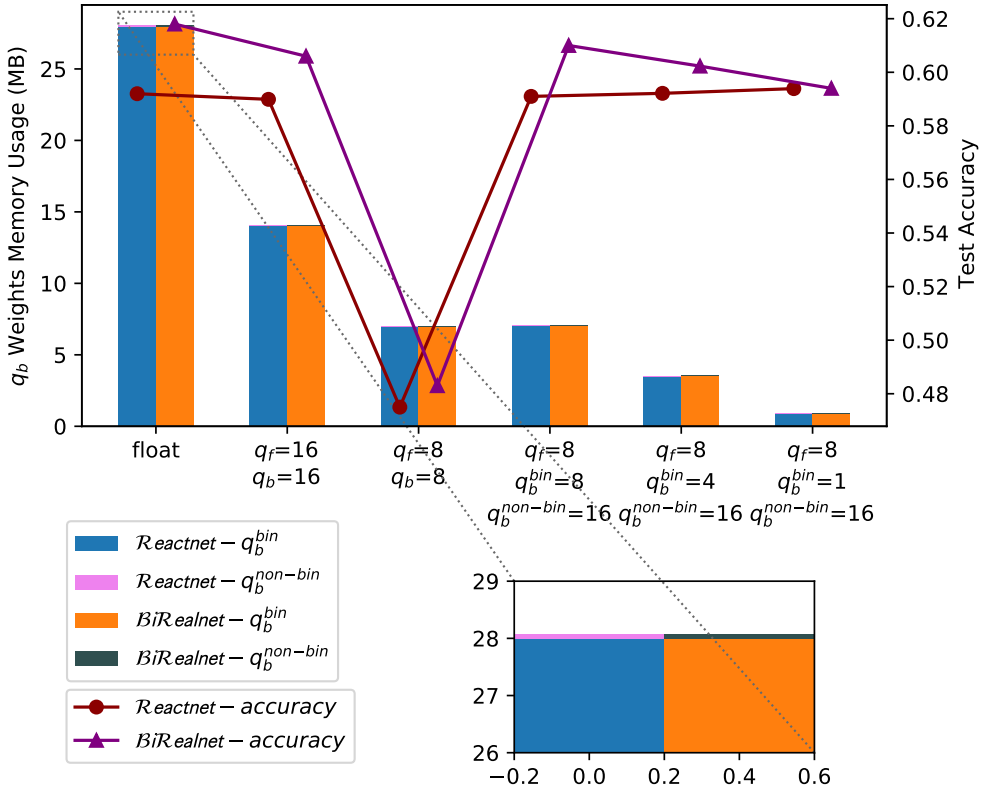


(B) Quicknet on CORe50.

FIGURE 4.12: LR memory requirement using different quantization levels and corresponding test set accuracy on CIFAR10 (a) and CORe50 (b). We considered 15, 20 and 30 elements for each class inside LR; for case (a) we adopted Reactnet-18 model while in (b) we used Quicknet.



(A) Quicknet and QuicknetLarge on CORe50.



(B) Reactnet-18 on CIFAR10.

FIGURE 4.13: q_b memory requirement using different quantization bitwidths for backward layer on CORe50 (a) and CIFAR10 (b).

Chapter 5

Preliminary Results on Datalogic Use Cases

This chapter describes some company use cases reporting preliminary results achieved by adopting binary neural networks. BNNs remarkably increase the efficiency of the products by reducing latency and memory, and the continual learning approaches reported in Chapter 4 would allow the adaptation of Datalogic devices to a continuously changing environment. Specifically, in Section 5.1 we present the localization task of a two-dimensional code, named Datamatrix, which is typically a time-consuming operation as a high-resolution input image has to be processed on a low-power CPU. In Sections 5.2 and 5.3 we introduce a couple of product cases where efficiency is mandatory, thus adopting BNNs, but the necessity to adapt the model to new data pushes the development of on-device learning solutions for low-bit widths models.

5.1 Datamatrix Detection

(A) Example of Code128

(B) Example of Datamatrix

FIGURE 5.1: Examples of 1D and 2D codes. Figure 5.1a represents a *linear* barcode, where the red segment indicates the bars and spaces relative to the letter *D*. Figure 5.1b shows an example of datamatrix symbology. The red elle represents the finder pattern of Datamatrix.

The core business of Datalogic is the automatic identification of items marked with a code that can be represented as a sequence of vertical bars (sometimes named as *linear* barcode) or as a two-dimensional grid. As shown in Figure 5.1, in the 1D barcode (Figure 5.1a) the information is encoded in the width of black bars and white

Model	Quantization	Accuracy (%)	Speed-up
Yolov5N	8-bit	95.3	1.0×
Custom BNN (**ours**)	1-bit	93.5	**4.1**×

TABLE 5.1: Acuracy and speed-up comparison of our custom BNN model with a SOTA model (Yolov5 Nano) on our proprietary Datamatrix dataset measured on a Raspberry Pi 3B.

spaces. Each combination of bars and spaces determines a specific codeword which is mapped to a number or letter (symbology dependent). The height of the bars guarantees a high information redundancy of the code and the number of bars is correlated to the total encoded characters. Instead, the 2D code (Figure 5.1b), exploiting both horizontal and vertical directions to embed data, results in higher encoding capabilities. Unfortunately, the analysis of a two-dimensional grid is more complex than sampling multiple lines across the vertical bars, leading to a more sophisticated and time-consuming decoding pipeline. Datamatrix, characterized by a *finder pattern* that is *L*-shaped, is the 2D symbology that requires the highest computational workload (shown in Figure 5.1b). Such a finder pattern is not very discriminative as the L shape is quite common within an image acquired in the scenarios addressed by Datalogic devices. Therefore, the localization process (the detection and validation of a valid L-pattern) of the Datamatrix is used to evaluate many patterns candidates for each image considered, substantially increasing the complexity and the computational workload for a low-power CPU that has to process high-resolution images, such as 1280×1024.

The usage of deep neural networks can certainly facilitate and improve the localization of Datamatrix code reaching state of the art results of accuracy, but the latency constraint of 30fps is hard to satisfy even using a shallow 8-bit quantized model, such as MobileNetv2 [121]. BNNs represent a valid alternative to consistently speed up the localization process. For this evaluation, we used a private dataset of Datamatrix samples having various sizes, scales, and resolutions. In Figure 5.2 we report a subset of the Datamatrix samples captured using different contrast levels, rotations, and textures. The network takes as input a sub-sampled version of the original image and produces as output, a maps containing all the codes detected and other parameters estimated, similar to the approach proposed in Yolov5 [64].

The BNN architecture employed is a custom and shallower version of *Quicknet* [8] and *ReactNet* [90], properly modified to fit the latency constraint using the methods proposed in Chapter 3. In Table 5.1, we report the accuracy and the estimated efficiency reached by our custom BNN architecture compared to the Yolov5 Nano 8-bit quantized. The preliminary results obtained by our BNN model achieve a remarkable speed-up of **4.1**× measured on a Raspberry Pi 3B with a minimal accuracy loss of 1.8%, allowing the successful deployment of the BNN on many Datalogic devices. For this evaluation, we employed a proprietary optimized inference engine that supports both binary and non-binary layers.

FIGURE 5.2: Examples of Datamatrix codes used to train our proprietary BNN model.

5.2 Hazard Symbols Localization and Classification

FIGURE 5.3: Examples of Hazmat classes supported by our proprietary BNN model used for localization and classification tasks.

Hazardous materials symbols, also known as warning symbols or safety symbols, are visual indicators designed to convey information about potential hazards associated with specific materials, locations, or objects. These symbols play a crucial role in ensuring the safety of individuals, particularly in industrial, laboratory, and public environments. Here are some common types of hazard symbols and their meanings:

1. **Chemical Hazard Symbols**:
   - **Explosive**: Indicates materials or substances that can explode.
   - **Flammable**: Indicates materials that can catch fire easily.
   - **Oxidizing**: Indicates substances that provide oxygen and may intensify fires.
2. **Biological Hazard Symbols**: Warns of the presence of dangerous biological substances.

3. **Radioactive Hazard Symbols**: Indicates the presence of radioactive materials emitting ionizing radiation.
4. **Toxic Hazard Symbols**: Indicates substances that are toxic and can cause harm.

These symbols are often standardized and regulated by national and international organizations but a certain level of variability for each Hazard symbol class is allowed. For instance, as shown in Figure 5.3, to specify that an object is flammable (Figure 5.4), many choices are available. The standard is not restrictive as for the barcode case and usually, every courier handles a customization for each Hazard class symbol. In the Transportation and Logistic (T&L) market, the detection and classification of all the Hazard symbols present on each parcel (to guarantee a high level of safety) is one of the requirements that the devices used for traceability must satisfy.

FIGURE 5.4: Example of a highly flammable Hazard symbol.

The variation of Hazard symbols, even within the same class, requires an automatic solution that has to easily handle new customizations. This scenario represents a good fit for a continual learning approach, specifically the New Classes. Practically, in this scenario, a deep neural network model is pre-trained on the initial subset of Hazard symbols. Before deploying the model, if a customer needs to recognize new symbols, the new Hazards are collected (this process can be performed automatically by using synthetic generation) into the experience 1. Next, the pre-trained model is trained on the newly collected set of symbols using the approach reported in Section 4.2. A subset of the symbols used to pre-train the model is used as initial replay memory content. Whenever a new set of Hazards has to be added (a new experience to execute), the previous process is repeated. The continuous learning phase of the model does not impose strict constraints of latency as it is executed only once for a new set of symbols but the inference of such a model on an embedded device has to satisfy severe latency limitations. The adoption of a Binary Neural Network can certainly reduce the processing load compared to an 8-bit quantized model but *ad-hoc* quantization schemes need to be employed for a BNN (as reported in Section 4.2).

The localization and classification of Hazard symbols can be accomplished using many types of CNNs. In this evaluation, we considered the Yolov5 Nano 8-bit quantized model, for the task of Section 5.1. The BNN architecture employed has a similar structure to the one presented in Section 5.1. The Hazard dataset used contains a collection of 60 symbols belonging to 25 categories. The same test set is used for all experiences. The samples are split into batches and provided sequentially during training obtaining 5 experiences. The first one with 40 symbols, and the remaining

with 5 symbols each. For the BNN model, we used the quantization scheme proposed in Section 4.2.1 setting $q_f = 8$, $q_b^{bin} = 1$ and $q_b^{non-bin} = 16$ as they ensure the best trade-offs between accuracy and memory demand. Both the BNN and Yolov5N models use a replay memory that stores 30 samples for each symbol class. In Figure 5.5 we present an accuracy comparison of our custom BNN model, denoted as BNN+LR+CWR*, with a standard Yolov5N solution. The results demonstrate that our binary model can reach a comparable accuracy to the 8-bit model while reducing the memory demand of replay memory by 8×.

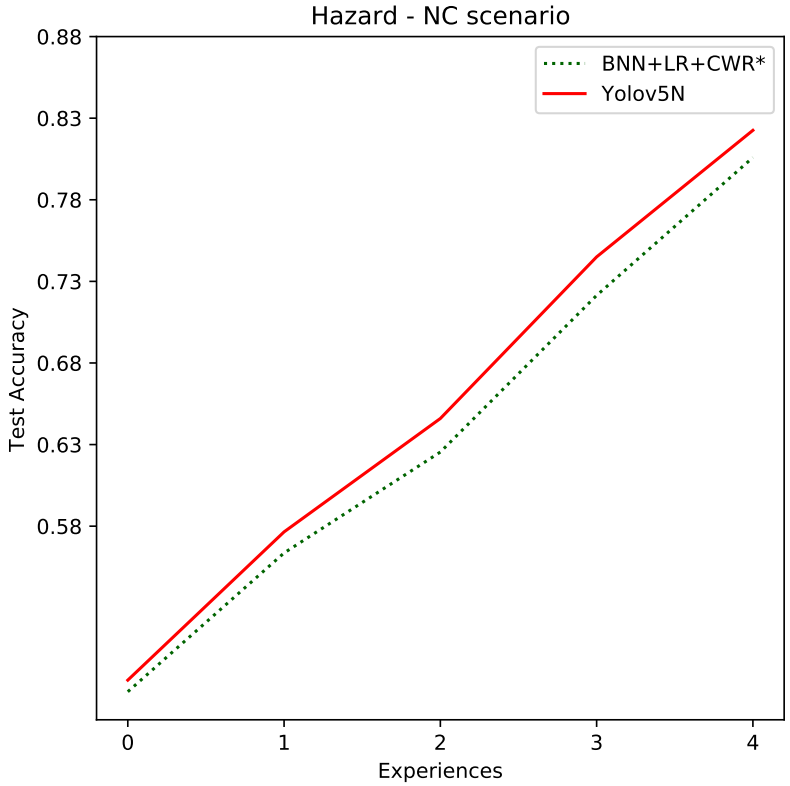


FIGURE 5.5: Accuracy comparison of our custom BNN based on the method BNN+LR+CWR* (Section 4.2.2) with a Yolov5N 8-bit quantized model.

5.3 Produce Classification

In retail applications there is an increasing request of smart solutions assisting the customers at the self checkout or the cashiers on the attended checkout lanes. A typical use case considered is reported in Figure 5.6, where a customer has to weigh fruit, and the system, AI-powered, assists the user by displaying the most likely produce candidates by analyzing the scene through a camera. When the system fails a classification, the user can modify the choice and this interaction can be exploited to continuously adjust the algorithm on new and fresh data.

This scenario presents many challenges as fruits and vegetables' appearance can significantly change over time. This task, similar to that of Section 5.2, can be addressed using a continual learning approach. Differently from the previous section, this application should be able to manage both items belonging to new classes (NC scenario) and new instances of already known classes (NI scenario). In both scenarios, the model is pre-trained on an initial dataset of samples, and then, when deployed

FIGURE 5.6: Examples of a weight-scale used for produce recognition.

on the field, the system can monitor the classification accuracies and trigger new training experiences when the performance accuracies degrade too much. The system can exploit the user feedback as ground truth information to be used during next experiences. Instead, when the classification is correct but the confidence is not robust, it is possible to add the associated images to the experience set to improve the classification score. When enough images have been collected, a continual learning experience is scheduled. In Figure 5.7 we report a subset of produce items used in this evaluation.

Similar to the previous section, this task requires a fast inference time but it can tolerate higher latencies during the on-device back-propagation phase. Therefore, we adopted the same BNN model of Section 5.2 comparing it with the Yolov5 Nano model (8-bit quantized). The produce dataset used contains a collection of 50 different items, split into 6 experiences for the NC scenario. The first one with 40 items, and the remaining with 2. For the BNN model, we used the same quantization settings of the previous section ($q_f = 8$, $q_b^{bin} = 1$ and $q_b^{non-bin} = 16$) and the replay memory stores 30 samples for each class. In Figure 5.8 we present an accuracy comparison of the models considered, and these preliminary results show a minimal accuracy gap between binary and non-binary models.

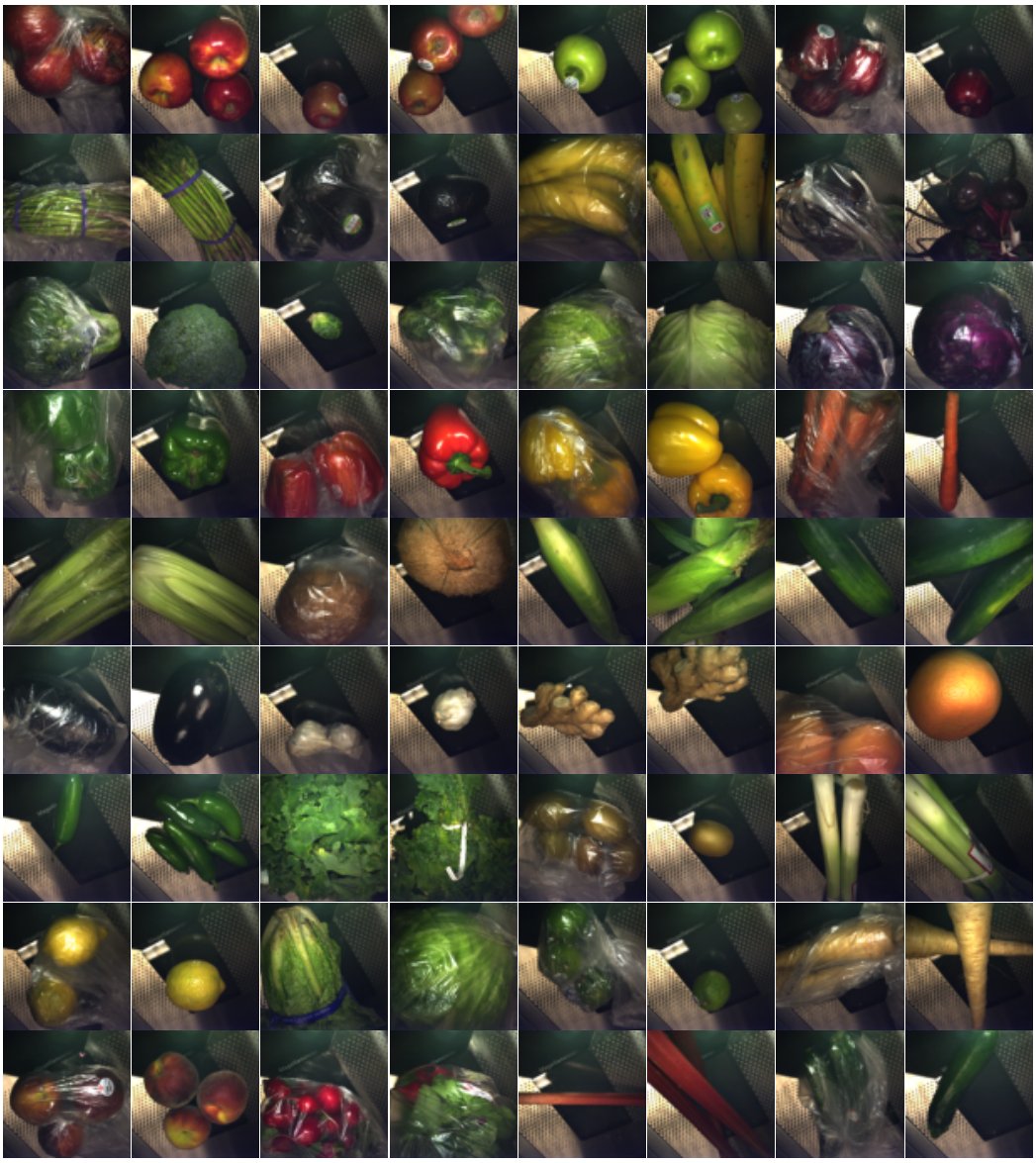


FIGURE 5.7: Examples of fruits and vegetables used to train our proprietary BNN model used for the classification task.

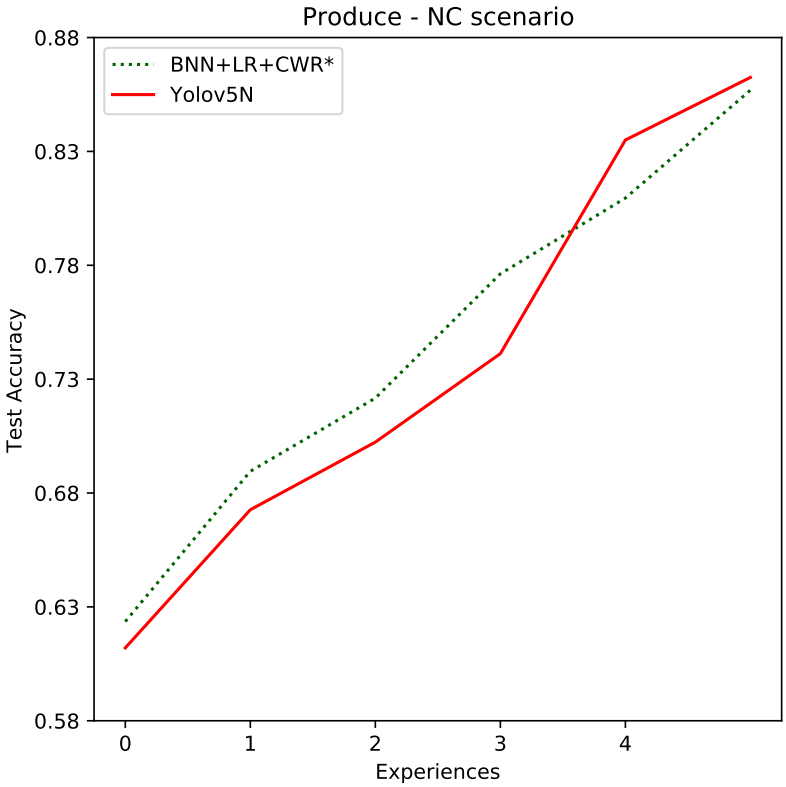


FIGURE 5.8: Accuracy comparison of our custom BNN based on the method BNN+LR+CWR* (Section 4.2.2) with a Yolov5N 8-bit quantized model.

Chapter 6

Conclusions and Future Challenges

This chapter summarizes the contributions presented in this thesis. For each contribution discussed, we explore the potential future research directions.

6.1 Efficiency of Binary Neural Networks

The techniques presented in Chapter 2 revise several model architectures, training tricks, and loss function designs aimed at improving the accuracy of binary neural networks. Even though plenty of literature work is available on this topic, only a few works addressed the deployment of BNNs on real hardware devices. When running on low-power embedded CPUs, it is fundamental the inference engine adopted to guarantee a fast forward pass of the network. The fastest inference engines available on the market for BNNs are daBNN [155] and LCE [8]. While the former is a custom implementation tailored to the layers used by Bi-Realnet [88], the latter integrates optimized implementations for binary layers directly within Tensorflow Lite. The necessity to further improve BNN model efficiency pushed the research of this thesis. In Chapter 3 we detailed a technique to speed up the forward pass of a BNN. In particular, it was shown in section 3.1.1, that data-flow constraints (section 3.1.1) of executing pipeline can be injected directly within the training procedure of the network, effectively restricting output values of binary convolutions into the 8-bit representation range. Additionally, to exploit previous achievement, in section 3.1.1 we proposed an optimization of the batch normalization layer when applied to the output of a binary convolution. In section 3.1.1 we proposed our optimized implementation of binary convolution for ARM NEON architectures. Overall, our solutions achieved a speed-up of **1.91** and **2.73**× compared to LCE and daBNN.

The BNN models covered in Chapter 2 and 3 typically exclude the first and last layers from binarization as it introduces a remarkable accuracy gap. This solution usually leads to deploying models where intermediate layers process 1-bit data while input and last layers are executed in floating-point or 8-bit fixed-point arithmetic. Such a choice usually increases power consumption and uses more hardware resources when deployed on FPGA systems. To address such issues, in section 3.2 we proposed a method to binarize the input layer of a BNN that reaches higher accuracy when compared to state-of-the-art solutions, reducing the gap to the floating-point baseline on average by 2.2 percentage points. Our method has proved to be more resource-constrained and hardware-friendly than existing solutions, as reported in Table 3.3.

Research directions As reported in Chapters 2 and 3, BNNs showed an impressive accuracy improvement in the last years, substantially filling up the gap with

corresponding floating-point models. One of the main challenges in BNN research is to further improve model efficiency, by reducing computational bottlenecks. Real-world applications, especially in the industrial environment, require a high accuracy level that is hard to reach using a fully binarized model. We argue that binary neural network strategies should address this limitation to create an optimized binary end-to-end pipeline. To validate our hypothesis, in Section 5.1, we report a company use case where the usage of a BNN represents a promising solution to speed up a processing-intensive application such as the Datamatrix detection and validation allowing it to reach an accuracy comparable to a floating-point model. Additionally, even though a multitude of binary architectures has been proposed, there is not yet a standard methodology on how to build a binary model, based on the task to solve. This aspect is tightly connected to the training procedure which is often elaborated as it is usually split into two training stages (during the first one only weights are binarized, in the second one also the activations) and it takes more epochs, compared to floating-point models, to converge. Much more could be done to simplify and speed up the long and sophisticated training procedure of BNNs allowing the adoption of this technology to a broader set of applications.

6.2 Continual Learning at the Edge

Chapter 4 faces the challenges of on-device learning using a binary neural network. Indeed, practical and real-world applications, like those presented in section 1.2, require optimized and fast processing pipelines, as those offered by BNNs, but at the same time they have to adapt to new circumstances and environments. The huge variety of everyday data is hard to capture with a single offline training, hence it is more convenient to explore the chance to continually adapt a model when new data is encountered, without incurring catastrophic forgetting. In literature, many solutions have been proposed to address the continual learning use case but very few of them considered the usage of very low bit-width models, such as binary networks. From the Datalogic company perspective, the necessity to investigate the on-device learning within the context of BNNs is essential to guarantee a high level of adaptability of the products.

In section 4.1, we presented our method to continually adapt a BNN model using the CWR* approach, which is an efficient technique like DSLDA [49], to handle the training phase. We explored the possibility of employing a frozen BNN backbone with a classification head that could adapt to new classes or new samples of known classes, by using an *ad-hoc* quantization scheme for back-propagation. We evaluated the trade-offs over many benchmarks adopting multiple BNN architectures. We discovered that the backward pass is more sensitive to low bit-width quantization as the small gradient updates require enough precision to avoid the accuracy gap w.r.t. the floating-point model. In section 4.2 we investigated the possibility of training a BNN model on-device by adopting a replay memory, used to store old samples. The replay memory mitigates the model forgetting of past samples but it also increases the overall memory footprint (in addition to the one used for back-propagation). In our solution, we do not store the raw input data but the activations corresponding to an intermediate layer, denoted as *latent replay layer*, to minimize memory usage. By combining this technique with BNNs it is possible to consistently decrease the RM space if the activations of past samples are saved using only 1-bit. This opportunity facilitates the deployment of continual learning solutions on edge devices, such

as those produced by Datalogic. The employment of RM in addition to the quantization scheme proposed to train binary and non-binary layers achieved superior performances compared to the results of section 4.1.

Research directions The achievements reported in Chapter 4 showed promising results to effectively train BNNs on-device. One of the main challenges to facilitating on-device learning is the development of efficient and lightweight training frameworks that can support quantized back-propagation. Indeed, to satisfy the constraints of efficiency and portability, the training framework cannot rely on floating-point computation as it is not available on all hardware platforms and it is more power-demanding compared to fixed-point arithmetic. We argue that the availability of efficient on-device training frameworks would boost consistently the diffusion of continual learning applications on edge devices. The preliminary results on real-world applications, reported in sections 5.2 and 5.3, verify our assumptions and enable the adaptation of binary models to new data directly on-device, preserving the efficiency of such models, the privacy of data, and reducing the costs. Additionally, to improve the accuracy achieved by continual learning approaches, it is fundamental the ability to constantly train a model using unlabeled data. Even if this is still an open field, the real-world applications on edge devices could benefit from a large stream of data acquired by sensors and the combination of supervised and unsupervised approaches should push this research direction.

Bibliography

[1] Thalaiyasingam Ajanthan et al. "Mirror descent view for neural network quantization". In: *International conference on artificial intelligence and statistics*. PMLR. 2021, pp. 2809–2817.

[2] Milad Alizadeh et al. "A systematic study of binary neural networks' optimisation". In: *International Conference on Learning Representations*. Vol. 63. 2019, p. 81.

[3] Fatima Alshehri and Ghulam Muhammad. "A comprehensive survey of the Internet of Things (IoT) and AI-based smart healthcare". In: *IEEE Access* 9 (2020), pp. 3660–3678.

[4] Alexander G Anderson and Cory P Berg. "The high-dimensional geometry of binary neural networks". In: *arXiv preprint arXiv:1705.07199* (2017).

[5] Colby R Banbury et al. "Benchmarking tinyml systems: Challenges and direction". In: *arXiv preprint arXiv:2003.04821* (2020).

[6] Ron Banner, Yury Nahshan, and Daniel Soudry. "Post training 4-bit quantization of convolutional networks for rapid-deployment". In: *Advances in Neural Information Processing Systems* 32 (2019).

[7] Ron Banner et al. "Scalable methods for 8-bit training of neural networks". In: *Advances in neural information processing systems* 31 (2018).

[8] Tom Bannink et al. "Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 680–695.

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

[10] J Bethge et al. "MeliusNet: Can binary neural networks achieve mobilenet-level accuracy? arXiv 2020". In: *arXiv preprint arXiv:2001.05936* ().

[11] Joseph Bethge et al. "Back to simplicity: How to train accurate bnns from scratch?" In: *arXiv preprint arXiv:1906.08637* (2019).

[12] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

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

[14] Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. "High-capacity expert binary networks". In: *arXiv preprint arXiv:2010.03558* (2020).

[15] Adrian Bulat and Georgios Tzimiropoulos. "Xnor-net++: Improved binary neural networks". In: *arXiv preprint arXiv:1909.13863* (2019).

[16] Han Cai et al. “Tinytl: Reduce memory, not parameters for efficient on-device learning”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 11285–11297.

[17] Yaohui Cai et al. “Zeroq: A novel zero shot quantization framework”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 13169–13178.

[18] Zhaowei Cai et al. “Deep learning with low precision by half-wave gaussian quantization”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5918–5926.

[19] Xuepeng Chang et al. “A mixed-pruning based framework for embedded convolutional neural network acceleration”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.4 (2021), pp. 1706–1715.

[20] Liang-Chieh Chen et al. “Rethinking atrous convolution for semantic image segmentation”. In: *arXiv preprint arXiv:1706.05587* (2017).

[21] Tianlong Chen et al. “" BNN-BN=?": Training Binary Neural Networks Without Batch Normalization”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 4619–4629.

[22] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Training deep neural networks with low precision multiplications”. In: *arXiv preprint arXiv:1412.7024* (2014).

[23] Matthieu Courbariaux et al. “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1”. In: *arXiv preprint arXiv:1602.02830* (2016).

[24] Sajad Darabi et al. “Regularized binary network training”. In: *arXiv preprint arXiv:1812.11800* (2018).

[25] Dipankar Das et al. “Mixed precision training of convolutional neural networks using integer operations”. In: *arXiv preprint arXiv:1802.00930* (2018).

[26] Matthias De Lange et al. “A continual learning survey: Defying forgetting in classification tasks”. In: *IEEE transactions on pattern analysis and machine intelligence* 44.7 (2021), pp. 3366–3385.

[27] Jonathan DeCastro et al. “Counterexample-guided safety contracts for autonomous driving”. In: *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer. 2020, pp. 939–955.

[28] James Diffenderfer and Bhavya Kailkhura. “Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning a randomly weighted network”. In: *arXiv preprint arXiv:2103.09377* (2021).

[29] Ruizhou Ding et al. “Regularizing activation distribution for training binarized deep networks”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 11408–11417.

[30] Yinpeng Dong et al. “Stochastic quantization for learning accurate low-bit deep neural networks”. In: *International Journal of Computer Vision* 127 (2019), pp. 1629–1642.

[31] Robert Dürichen et al. “Binary Input Layer: Training of CNN models with binary input data”. In: *arXiv preprint arXiv:1812.03410* (2018).

[32] Sieger Falkena, Hadi Jamali-Rad, and Jan van Gemert. "LAB: Learnable Activation Binarizer for Binary Neural Networks". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 6425–6434.

[33] Jun Fang et al. "Near-lossless post-training quantization of deep neural networks via a piecewise linear approximation". In: *arXiv preprint arXiv:2002.00104* 10 (2020), pp. 978–3.

[34] Jun Fang et al. "Post-training piecewise linear quantization for deep neural networks". In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*. Springer. 2020, pp. 69–86.

[35] Sahaj Garg et al. "Confounding tradeoffs for neural network quantization". In: *arXiv preprint arXiv:2102.06366* (2021).

[36] Sahaj Garg et al. "Dynamic precision analog computing for neural networks". In: *IEEE Journal of Selected Topics in Quantum Electronics* 29.2: Optical Computing (2022), pp. 1–12.

[37] Amir Gholami et al. "A survey of quantization methods for efficient neural network inference". In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[38] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[39] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[40] Wojciech Glegoła, Aleksandra Karpus, and Adam Przybyłek. "MobileNet family tailored for Raspberry Pi". In: *Procedia Computer Science* 192 (2021), pp. 2249–2258.

[41] Ruihao Gong et al. "Differentiable soft quantization: Bridging full-precision and low-bit neural networks". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 4852–4861.

[42] Gabriele Graffieti, Guido Borghi, and Davide Maltoni. "Continual learning in real-life applications". In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 6195–6202.

[43] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee. 2013, pp. 6645–6649.

[44] Jiaxin Gu et al. "Bayesian optimized 1-bit cnns". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 4909–4917.

[45] Yiwen Guo, Anbang Yao, and Yurong Chen. "Dynamic network surgery for efficient dnns". In: *Advances in neural information processing systems* 29 (2016).

[46] Suyog Gupta et al. "Deep learning with limited numerical precision". In: *International conference on machine learning*. PMLR. 2015, pp. 1737–1746.

[47] Kai Han et al. "Training binary neural networks through learning with noisy supervision". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 4017–4026.

[48] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[49] Tyler L Hayes and Christopher Kanan. "Lifelong machine learning with deep streaming linear discriminant analysis". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. 2020, pp. 220–221.

[50] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[51] Kaiming He et al. "Mask r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[52] Xiangyu He and Jian Cheng. "Learning compression from limited unlabeled data". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 752–769.

[53] Xiangyu He et al. "Proxybnn: Learning binarized neural networks via proxy matrices". In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*. Springer. 2020, pp. 223–241.

[54] Koen Helwegen et al. "Latent weights do not exist: Rethinking binarized neural network optimization". In: *Advances in neural information processing systems* 32 (2019).

[55] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* (2015).

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

[57] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[58] Itay Hubara et al. "Binarized neural networks". In: *Advances in neural information processing systems* 29 (2016).

[59] Itay Hubara et al. "Improving post training neural quantization: Layer-wise calibration and integer programming". In: *arXiv preprint arXiv:2006.10518* (2020).

[60] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[61] Benoit Jacob, P Warden, and ME Guney. "gemmlowp: a small self-contained low-precision GEMM library.(2017)". In: *URL https://github. com/google/gemmlowp* (2017).

[62] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[63] Taehee Jeong et al. "Neural network pruning and hardware acceleration". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2020, pp. 440–445.

[64] Glenn Jocher et al. "ultralytics/yolov5: v5. 0-YOLOv5-P6 1280 models, AWS, Supervise. ly and YouTube integrations". In: *Zenodo* (2021).

[65] Dahyun Kim, Kunal Pratap Singh, and Jonghyun Choi. "Learning architectures for binary networks". In: *European conference on computer vision*. Springer. 2020, pp. 575–591.

[66] Hyungjun Kim et al. "Binaryduo: Reducing gradient mismatch in binary activation network by coupling binary activations". In: *arXiv preprint arXiv:2002.06517* (2020).

[67] Hyungjun Kim et al. "Improving accuracy of binary neural networks using unbalanced activation distribution". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 7862–7871.

[68] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[69] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.

[70] Brett Koonce and Brett Koonce. "MobileNetV3". In: *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization* (2021), pp. 125–144.

[71] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[73] Jérémie Laydevant et al. "Training dynamical binary neural networks with equilibrium propagation". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 4640–4649.

[74] Ya Le and Xuan Yang. "Tiny imagenet visual recognition challenge". In: *CS 231N* 7.7 (2015), p. 3.

[75] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[76] Changhun Lee et al. "INSTA-BNN: Binary neural network with instance-aware threshold". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 17325–17334.

[77] Jun Haeng Lee et al. "Quantization for rapid deployment of deep neural networks". In: *arXiv preprint arXiv:1810.05488* (2018).

[78] Guy GF Lemieux et al. "TinBiNN: Tiny binarized neural network overlay in about 5,000 4-LUTs and 5mw". In: *arXiv preprint arXiv:1903.06630* (2019).

[79] Sergey Levine et al. "End-to-end training of deep visuomotor policies". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1334–1373.

[80] Yuhang Li et al. "Brecq: Pushing the limit of post-training quantization by block reconstruction". In: *arXiv preprint arXiv:2102.05426* (2021).

[81] Yunqiang Li, Silvia-Laura Pintea, and Jan C van Gemert. "Equal bits: Enforcing equally distributed binary network weights". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 36. 2. 2022, pp. 1491–1499.

[82] Zefan Li et al. "Performance guaranteed network acceleration via high-order residual quantization". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2584–2592.

[83] Ji Lin et al. "On-device training under 256kb memory". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 22941–22954.

[84] Mingbao Lin et al. “Rotated binary neural network”. In: *Advances in neural information processing systems* 33 (2020), pp. 7474–7485.

[85] Mingbao Lin et al. “Siman: Sign-to-magnitude network binarization”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.5 (2022), pp. 6277–6288.

[86] Chunlei Liu et al. “Circulant binary convolutional networks: Enhancing the performance of 1-bit dcnns with circulant back propagation”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 2691–2699.

[87] Hanxiao Liu, Karen Simonyan, and Yiming Yang. “Darts: Differentiable architecture search”. In: *arXiv preprint arXiv:1806.09055* (2018).

[88] Zechun Liu et al. “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)*. 2018, pp. 722–737.

[89] Zechun Liu et al. “How do adam and training strategies help bnns optimization”. In: *International conference on machine learning*. PMLR. 2021, pp. 6936–6946.

[90] Zechun Liu et al. “Reactnet: Towards precise binary neural network with generalized activation functions”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIV 16*. Springer. 2020, pp. 143–159.

[91] Anastasiia Livochka and Alexander Shekhovtsov. “Initialization and transfer learning of stochastic binary networks from real-valued ones”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 4660–4668.

[92] Vincenzo Lomonaco and Davide Maltoni. “Core50: a new dataset and benchmark for continuous object recognition”. In: *Conference on Robot Learning*. PMLR. 2017, pp. 17–26.

[93] Vincenzo Lomonaco, Davide Maltoni, and Lorenzo Pellegrini. “Rehearsal-Free Continual Learning over Small Non-IID Batches.” In: *CVPR Workshops*. Vol. 1. 2. 2020, p. 3.

[94] Björn Lütjens, Lucas Liebenwein, and Katharina Kramer. “Machine learning-based estimation of forest carbon stocks to increase transparency of forest preservation efforts”. In: *arXiv preprint arXiv:1912.07850* (2019).

[95] Brais Martinez et al. “Training binary neural networks with real-to-binary convolutions”. In: *arXiv preprint arXiv:2003.11535* (2020).

[96] Marc Masana et al. “Class-incremental learning: survey and performance evaluation on image classification”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.5 (2022), pp. 5513–5533.

[97] Eldad Meller et al. “Same, same but different: Recovering neural network quantization error through weight factorization”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4486–4495.

[98] Asit Mishra et al. “WRPN: Wide reduced-precision networks”. In: *arXiv preprint arXiv:1709.01134* (2017).

[99] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[100] Esraa Mohamed. "The relation of artificial intelligence with internet of things: A survey". In: *Journal of Cybersecurity and Information Management* 1.1 (2020), pp. 30–24.

[101] Davide Nadalini et al. "PULP-TrainLib: Enabling on-device training for RISC-V multi-core MCUs through performance-driven Autotuning". In: *International Conference on Embedded Computer Systems*. Springer. 2022, pp. 200–216.

[102] Davide Nadalini et al. "Reduced Precision Floating-Point Optimization for Deep Neural Network On-Device Learning on MicroControllers". In: *arXiv preprint arXiv:2305.19167* (2023).

[103] Markus Nagel et al. "A white paper on neural network quantization". In: *arXiv preprint arXiv:2106.08295* (2021).

[104] Markus Nagel et al. "Data-free quantization through weight equalization and bias correction". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1325–1334.

[105] German I Parisi et al. "Continual lifelong learning with neural networks: A review". In: *Neural networks* 113 (2019).

[106] Wonpyo Park et al. "Relational knowledge distillation". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 3967–3976.

[107] Lorenzo Pellegrini et al. "Latent replay for real-time continual learning". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 10203–10209.

[108] Phuoc Pham, Jacob A Abraham, and Jaeyong Chung. "Training multi-bit quantized and binarized networks with a learnable symmetric quantizer". In: *IEEE Access* 9 (2021), pp. 47194–47203.

[109] Hai Phan et al. "Binarizing mobilenet via evolution-based searching". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 13420–13429.

[110] Hai Phan et al. "Mobinet: A mobile binary network for image classification". In: *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 2020, pp. 3453–3462.

[111] Jef Plochaet and Toon Goedemé. "Hardware-Aware Pruning for FPGA Deep Learning Accelerators". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 4481–4489.

[112] Haotong Qin et al. "Forward and backward information retention for accurate binary neural networks". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 2250–2259.

[113] Mohammad Rastegari et al. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[114] Leonardo Ravaglia et al. "A tinyml platform for on-device continual learning with quantized latent replays". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (2021), pp. 789–802.

[115] Arthur J Redfern, Lijun Zhu, and Molly K Newquist. "BCNN: A binary CNN with all matrix Ops Quantized to 1 bit precision". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 4604–4612.

[116] Russell Reed. "Pruning algorithms-a survey". In: *IEEE transactions on Neural Networks* 4.5 (1993), pp. 740–747.

[117] Haoyu Ren, Darko Anicic, and Thomas A Runkler. "Tinyol: Tinyml with online-learning on microcontrollers". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[118] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems* 28 (2015).

[119] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[120] Charbel Sakr et al. "True gradient-based training of deep binary activated neural networks via continuous binarization". In: *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2018, pp. 2346–2350.

[121] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[122] Yuzhang Shang et al. "Lipschitz continuity retained binary neural network". In: *European conference on computer vision*. Springer. 2022, pp. 603–619.

[123] Mingzhu Shen et al. "Balanced binary neural networks with gated residual". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 4197–4201.

[124] Xulong Shi et al. "RepBNN: towards a precise Binary Neural Network with Enhanced Feature Map via Repeating". In: *arXiv preprint arXiv:2207.09049* (2022).

[125] Gil Shomron et al. "Post-training sparsity-aware quantization". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 17737–17748.

[126] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[127] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[128] Yuanxin Su et al. "Binary Neural Networks in FPGAs: Architectures, Tool Flows and Hardware Comparisons". In: *Sensors* 23.22 (2023), p. 9254.

[129] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[130] Mingxing Tan and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.

[131] Mingxing Tan and Quoc Le. "Efficientnetv2: Smaller models and faster training". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 10096–10106.

[132] Wei Tang, Gang Hua, and Liang Wang. "How to train a compact binary neural network with high accuracy?" In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 31. 1. 2017.

[133] Tijmen Tieleman and G Hinton. "Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning". In: *Technical report* (2017).

[134] Zhijun Tu et al. "Adabin: Improving binary neural networks with adaptive binary sets". In: *European conference on computer vision*. Springer. 2022, pp. 379–395.

[135] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[136] Jeffrey S Vitter. "Random sampling with a reservoir". In: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pp. 37–57.

[137] Lorenzo Vorabbi, Davide Maltoni, and Stefano Santi. "Input Layer Binarization with Bit-Plane Encoding". In: *Artificial Neural Networks and Machine Learning – ICANN 2023*. Ed. by Lazaros Iliadis et al. Cham: Springer Nature Switzerland, 2023, pp. 395–406. ISBN: 978-3-031-44198-1.

[138] Lorenzo Vorabbi, Davide Maltoni, and Stefano Santi. "On-Device Learning with Binary Neural Networks". In: *Image Analysis and Processing - ICIAP 2023 Workshops*. Ed. by Gian Luca Foresti, Andrea Fusiello, and Edwin Hancock. Cham: Springer Nature Switzerland, 2024, pp. 39–50. ISBN: 978-3-031-51023-6.

[139] Lorenzo Vorabbi, Davide Maltoni, and Stefano Santi. "Optimizing data-flow in Binary Neural Networks". In: *arXiv preprint arXiv:2304.00952* (2023).

[140] Lorenzo Vorabbi. et al. "Enabling On-Device Continual Learning with Binary Neural Networks and Latent Replay". In: *Proceedings of the 19th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 2: VISAPP*. INSTICC. SciTePress, 2024, pp. 25–36. ISBN: 978-989-758-679-8.

[141] Erwei Wang et al. "Enabling binary neural network training on the edge". In: *Proceedings of the 5th international workshop on embedded and mobile deep learning*. 2021, pp. 37–38.

[142] Peisong Wang et al. "Sparsity-inducing binarized neural networks". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 07. 2020, pp. 12192–12199.

[143] Peisong Wang et al. "Two-step quantization for low-bit neural networks". In: *Proceedings of the IEEE Conference on computer vision and pattern recognition*. 2018, pp. 4376–4384.

[144] Yikai Wang et al. "Sub-bit neural networks: Learning to compress and accelerate binary neural networks". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 5360–5369.

[145] Ziwei Wang et al. "Learning channel-wise interactions for binary convolutional neural networks". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 568–577.

[146] Jiaxiang Wu et al. "Quantized convolutional neural networks for mobile devices". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4820–4828.

[147] Yinghao Xu et al. "A main/subsidiary network framework for simplifying binary neural networks". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 7154–7162.

[148] Yixing Xu et al. "Learning frequency domain approximation for binary neural networks". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 25553–25565.

[149] Zhe Xu and Ray CC Cheung. "Accurate and compact convolutional neural networks with trained binarization". In: *arXiv preprint arXiv:1909.11366* (2019).

[150] Zihan Xu et al. "Recu: Reviving the dead weights in binary neural networks". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 5198–5208.

[151] Jiwei Yang et al. "Quantization networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 7308–7316.

[152] Zhaohui Yang et al. "Searching for low-bit weights in quantized neural networks". In: *Advances in neural information processing systems* 33 (2020), pp. 4091–4102.

[153] Zhewei Yao et al. "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 27168–27183.

[154] Dongqing Zhang et al. "Lq-nets: Learned quantization for highly accurate and compact deep neural networks". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 365–382.

[155] Jianhao Zhang et al. "dabnn: A super fast inference framework for binary neural networks on arm devices". In: *Proceedings of the 27th ACM international conference on multimedia*. 2019, pp. 2272–2275.

[156] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. "High performance zero-memory overhead direct convolutions". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 5776–5785.

[157] Susan Zhang et al. "Opt: Open pre-trained transformer language models". In: *arXiv preprint arXiv:2205.01068* (2022).

[158] Yichi Zhang, Zhiru Zhang, and Lukasz Lew. "Pokebnn: A binary pursuit of lightweight accuracy". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12475–12485.

[159] Yichi Zhang et al. "FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2021, pp. 171–182.

[160] Junhe Zhao et al. "Data-adaptive binary neural networks for efficient object detection and recognition". In: *Pattern Recognition Letters* 153 (2022), pp. 239–245.

[161] Shuchang Zhou et al. "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients". In: *arXiv preprint arXiv:1606.06160* (2016).

[162] 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)*. IEEE. 2020, pp. 1–8.

[163] 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*. 2019, pp. 4923–4932.