Enabling Binary Neural Network Training on the Edge

ERWEI WANG, Imperial College London, United Kingdom
JAMES J. DAVIS, Imperial College London, United Kingdom
DANIELE MORO, Google, United States
PIOTR ZIELINSKI, Google, United States
JIA JIE LIM, iSize, United Kingdom
CLAUDIONOR COELHO, Advantest, United States
SATRAJIT CHATTERJEE,, United States
PETER Y. K. CHEUNG, Imperial College London, United Kingdom
GEORGE A. CONSTANTINIDES, Imperial College London, United Kingdom

The ever-growing computational demands of increasingly complex machine learning models frequently necessitate the use of powerful cloud-based infrastructure for their training. Binary neural networks are known to be promising candidates for on-device inference due to their extreme compute and memory savings over higher-precision alternatives. However, their existing training methods require the concurrent storage of high-precision activations for all layers, generally making learning on memory-constrained devices infeasible. In this article, we demonstrate that the backward propagation operations needed for binary neural network training are strongly robust to quantization, thereby making on-the-edge learning with modern models a practical proposition. We introduce a low-cost binary neural network training strategy exhibiting sizable memory footprint reductions while inducing little to no accuracy loss *vs* Courbariaux & Bengio's standard approach. These decreases are primarily enabled through the retention of activations exclusively in binary format. Against the latter algorithm, our drop-in replacement sees memory requirement reductions of 3–5×, while reaching similar test accuracy (±2 pp) in comparable time, across a range of small-scale models trained to classify popular datasets. We also demonstrate from-scratch ImageNet training of binarized ResNet-18, achieving a 3.78× memory reduction. Our work is open-source, and includes the Raspberry Pi-targeted prototype we used to verify our modeled memory decreases and capture the associated energy drops. Such savings will allow for unnecessary cloud offloading to be avoided, reducing latency, increasing energy efficiency, and safeguarding end-user privacy.

CCS Concepts: • **Computing methodologies → Machine learning**; • **Computer systems organization → Embedded systems**.

Additional Key Words and Phrases: Deep neural network, binary neural network, training, edge devices, embedded systems, memory reduction.

1 INTRODUCTION

Although binary neural networks (BNNs) feature weights and activations with just single-bit precision, many models are able to reach accuracy indistinguishable from that of their higher-precision counterparts [13, 43]. Since BNNs are functionally complete, their limited precision does not impose an upper bound on achievable accuracy [12]. BNNs represent the ideal class of neural network for edge inference, particularly for custom hardware implementation, due to their

Authors' addresses: Erwei Wang, Imperial College London, London, Exhibition Road, SW7 2BX, United Kingdom, erwei.wang@amd.com; James J. Davis, Imperial College London, London, Exhibition Road, SW7 2BX, United Kingdom, james.davis@imperial.ac.uk; Daniele Moro, Google, Mountain View, 2015 Stierlin Court, CA, 94043, United States, danielemoro@google.com; Piotr Zielinski, Google, Mountain View, 2015 Stierlin Court, CA, 94043, United States, zielinski@google.com; Jia Jie Lim, iSize, London, 107 Cheapside, EC2V 6DN, United Kingdom, jj.lim@isize.co; Claudionor Coelho, Advantest, San Jose, 3061 Zanker Rd, CA, 95134, United States, claudionor.coelho@alumni.stanford.edu; Satrajit Chatterjee, Palo Alto, CA, United States, satrajit@gmail.com; Peter Y. K. Cheung, Imperial College London, London, Exhibition Road, SW7 2BX, United Kingdom, p.cheung@imperial.ac.uk; George A. Constantinides, Imperial College London, London, Exhibition Road, SW7 2BX, United Kingdom, g.constantinides@imperial.ac.uk.

Table 1
Applied approximations used in low-cost neural network training works. ✘ signifies approximation-free (`float32`) variables.

	Weights		Weight gradients	Activations		Activation gradients	Batch norm.
	Forward	Backward		Forward	Backward		
[51]	int6[1]	int6	int6	int6	int6	int6	✘
[20]	✘	✘	✘	✘	Recomputed[2]	✘	✘
[17]	float16	float16	float16	float16	float16	float16	✘
[19]	✘	✘	✘	int	int	✘	✘
[38]	✘	✘	✘	E5M2	E5M2	✘	✘
[3]	✘	✘	bool	✘	✘	✘	✘
[48]	✘	✘	✘	✘	✘	✘	ℓ_1
This work	bool	float16	bool	bool	bool	float16	BNN-specific

[1] Arbitrary precision was supported, but significant accuracy degradation was observed below 6 bits.
[2] Activations were not retained between forward and backward propagation in order to save memory.

use of XNOR for multiplication: a fast and cheap operation to perform. Their compact weights also suit systems with limited memory and increases opportunities for caching, providing further potential performance boosts. FINN, the seminal BNN implementation for field-programmable gate arrays, reached the highest CIFAR-10, and SVHN classification rates to date at the time of its publication [40].

Despite featuring binary forward propagation, existing BNN training approaches perform backward propagation using high-precision floating-point data types—typically `float32`—often making training infeasible on memory-constrained devices. The high-precision activations used between forward and backward propagation commonly constitute the largest proportion of the total memory footprint of a training run [7, 37]. Our understanding of standard BNN training algorithms led us to the following realization: high-precision activations should not be used since we are only concerned with weights and activations' *signs*. In this article, we present a low-memory BNN training scheme based on this intuition featuring binary activations only, facilitated through batch normalization modification.

By increasing the viability of learning on the edge, this work will reduce the domain mismatch between training and inference—particularly in conjunction with federated learning [6, 31]—and ensure privacy for sensitive applications [1]. Via the aggressive memory footprint reductions they facilitate, our proposals will enable models to be trained without the network access reliance, latency and energy overheads or data divulgence inherent to cloud offloading. Herein, we make the following contributions.

- We conduct a variable representation and lifetime analysis of Courbariaux & Bengio's standard BNN training process [13]. We use this to identify opportunities for memory savings through approximation.
- Via our proposed BNN-specific forward and backward batch normalization operations, we implement a neural network training regime featuring all-binary activations. This significantly reduces the greatest constituent of a given training run's total memory footprint.

- We present a successful combination of binary activations and binary weight gradients during neural network training. This aggregation allows for further reductions in memory footprint.
- We systematically evaluate the impact of each of our approximations, and provide a detailed characterization of our scheme's memory requirements *vs* accuracy.
- Against the standard approach, we report memory reductions of up to 5.44×, with little to no accuracy or convergence rate degradation, when training BNNs to classify MNIST, CIFAR-10, and SVHN. No hyperparameter tuning is required. We also show that the batch size used can be increased by ∼10× while remaining within a given memory envelope, and even demonstrate the efficacy of ImageNet training as a hard target.
- We provide an open-source release of our Keras-based training software, memory modeling tool, and Raspberry Pi-targeted prototype for the community to use and build upon[1]. Our memory breakdown analysis represents a clear road map to further, future reductions.

2 RELATED WORK

The authors of all published works on BNN inference acceleration to date made use of high-precision floating-point data types during training [13, 14, 16, 21, 27–29, 39, 41, 42]. There is precedent, however, for the use of quantization when training non-binary networks, as we show in Table 1 via side-by-side comparison of the approximation approaches taken in those works along with those detailed in this article.

The effects of quantizing the gradients of models with high-precision data, either fixed or floating point, have been studied extensively. Zhou *et al.* [51] and Wu *et al.* [47] trained networks with fixed-point weights and activations using fixed-point gradients, reporting no accuracy loss for AlexNet classifying ImageNet with gradients wider than five bits. Wen *et al.* [45] and Bernstein *et al.* [3] focused solely on aggressive weight gradient quantization, aiming to reduce communication costs for distributed learning. Weight gradients were losslessly quantized into ternary and binary formats, respectively, with forward propagation and activation gradients kept at high precision. Tatsumi et al. identified redundancy in the rounding implementations of IEEE-754 standard, such as the IEEE-754 conversion for rounding, subnormal, and not-a-number and infinity encodings, at MAC outputs [38]. The authors also presented empirical evidences showing the feasibility of training DNNs using low-precision floating point formats such as `E5M1` and `E5M2` which use five bits for exponent and one and two bits for mantissa, respectively. In this work, we make the novel observation that BNNs are more robust to approximation during training than higher-precision networks. We thus propose a data representation scheme more aggressive than all of the aforementioned works combined, delivering large memory savings with near-lossless performance.

An intuitive method to lower the memory footprint of training is to simply reduce the batch size. However, doing so generally leads to increased total training time due to reduced memory reuse [37]. The method we propose in this article does not conflict with batch size tuning, and further allows the use of large batches while remaining within the memory limits of edge devices.

Gradient checkpointing—the recomputation of activations during backward propagation—has been proposed as a method to reduce the memory consumption of training [10, 20]. Such methods introduce additional forward passes, however, and so increase each run's duration and energy cost. Graham [19] and Chakrabarti & Moseley [8] saved memory during training by buffering activations in low-precision formats, achieving comparable accuracy to all-`float32` baselines. Wu *et al.* [48] and Hoffer *et al.* [23] reported reduced computational costs via ℓ_1 batch normalization. Finally, Helwegen *et al.* [22] asserted that the use of both trainable weights and momenta is superfluous in

[1]https://github.com/awai54st/Enabling-Binary-Neural-Network-Training-on-the-Edge

BNN optimizers, proposing a weightless BNN-specific optimizer, Bop, able to reach the same level of accuracy as Adam. We took inspiration from these works in locating sources of redundancy present in standard BNN training schemes, and propose BNN-specific modifications to ℓ_1 batch normalization allowing for activation quantization all the way to binary, thus saving memory without increasing latency. Yayla *et al.* [49] further developed methods to compress the momentum values uniquely introduced in Bop, and obtained memory savings in BNN training without incurring significant loss in accuracy. Our method aims to identify common bottlenecks for BNN training, irrespective of the optimizer choice, and is therefore orthogonal and complementary to techniques such as Yayla *et al.*'s.

Recent efforts have shown that, in some circumstances, batch normalization can be completely removed from BNN training. Chen *et al.* replaced the trainable scaling factors and biases within standard ℓ_2 batch normalization with hand-tuned values, thereby approximating these functions via trial and error [11]. Our method follows a conventional training approach; no manual, offline steps are required. Jiang *et al.* proposed the use of batch normalization-free BNNs for super-resolution imaging [24]. The information loss incurred from the removal of batch normalization in this case is recovered by expanding the receptive fields of convolution operations using parallel sets of binary dilated convolutions. While Jiang *et al.* demonstrated promising results for super-resolution imaging, we assume a generic deep learning setting rather than focusing on a specific application domain. We further present an open-source Raspberry Pi-based prototype to corroborate our memory reduction estimates, making our work closer to real application deployment than both of the aforementioned publications.

The authors of works including Bi-Real Net [29], ResNetE-18 [4], and ReActNet [28] discovered that the accuracy of BNNs can be significantly increased via the addition of high-precision skip connections. Many further enhanced BNN performance via improvements to gradient approximation and weight initialization [4, 15, 28–30]. Optimizations such as these are intended to increase accuracy: a goal orthogonal to ours of efficiently deploying BNNs on edge-scale devices. Nevertheless, we incorporated all of them into our work in order to reach competitive accuracy.

For works such as ReActNet [28], BN-Free [11], BN-Free ISR [24], and Real-to-Binary [30], it was found that knowledge distillation—the employment of a high-precision network as a "teacher" running alongside a BNN—can greatly improve the performance of the latter's training. This method is, however, outside our scope; the teacher would dominate overall memory requirements and thereby make savings with regards to the BNN insignificant.

3 STANDARD TRAINING FLOW

For simplicity of exposition, we assume the use of a multi-layer perceptron (MLP), although the presence of convolutional layers would not change any of the principles that follow. We use ∂ symbol to represent a gradient with respect to the neural network cost function C, such that ∂x denotes gradient $\partial C/\partial x$. Let $\boldsymbol{W}_l$ and $\boldsymbol{X}_l$ denote matrices of weights and activations, respectively, in the network's l^{th} layer, with $\partial \boldsymbol{W}_l$ and $\partial \boldsymbol{X}_l$ being their gradients. For $\boldsymbol{W}_l$, rows and columns span input and output channels, respectively, while for $\boldsymbol{X}_l$ they span a batch's feature maps and their channels. Henceforth, we use $\hat{\bullet}$ to denote binary encoding.

Fig. 1 shows the training graph of a fully connected binary layer. A detailed description of the standard BNN training procedure introduced by Courbariaux & Bengio [13] for each batch of B training samples, which we henceforth refer to as a *step*, is provided in Algorithm 1. Therein, "$\odot$" signifies element-wise multiplication. For brevity, we omit some of the intricacies of the baseline implementation—lack of first-layer quantization, use of a final softmax layer, and the inclusion of weight gradient cancelation [13]—as these standard BNN practices are not impacted by our work. We initialize weights as outlined by Glorot & Bengio [18].

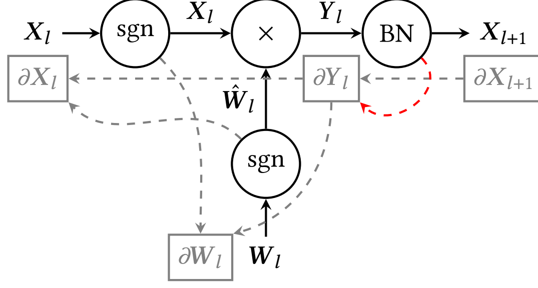

Fig. 1. Standard BNN training graph for fully connected layer l. "sgn", "×", and "BN" are sign, matrix multiplication, and batch normalization operations. Forward propagation dependencies are shown with solid lines; those for backward passes are dashed. High-precision activations must be retained due to the red dependency.

Algorithm 1 Standard BNN training step.

1: **for** $l \leftarrow \{1, \cdots, L-1\}$ **do** ▷ Forward prop.
2: $\hat{\boldsymbol{X}}_l \leftarrow \mathrm{sgn}(\boldsymbol{X}_l)$
3: $\hat{\boldsymbol{W}}_l \leftarrow \mathrm{sgn}(\boldsymbol{W}_l)$
4: $Y_l \leftarrow \hat{\boldsymbol{X}}_l \hat{\boldsymbol{W}}_l$
5: **for** $m \leftarrow \{1, \cdots, M_l\}$ **do** ▷ Batch norm.
6: $\psi_l^{(m)} \leftarrow \sigma\left(\boldsymbol{y}_l^{(m)}\right)$
7: $\boldsymbol{x}_{l+1}^{(m)} \leftarrow \frac{\boldsymbol{y}_l^{(m)} - \mu\left(\boldsymbol{y}_l^{(m)}\right)}{\psi_l^{(m)}} + \beta_l^{(m)}$
8:
9: **for** $l \leftarrow \{L-1, \cdots, 1\}$ **do** ▷ Backward prop.
10: **for** $m \leftarrow \{1, \cdots, M_l\}$ **do** ▷ Batch norm.
11: $\boldsymbol{v} \leftarrow \frac{1}{\psi_l^{(m)}} \partial \boldsymbol{x}_{l+1}^{(m)}$
12: $\partial \boldsymbol{y}_l^{(m)} \leftarrow \boldsymbol{v} - \mu(\boldsymbol{v}) - \mu\left(\boldsymbol{v} \odot \boxed{\boldsymbol{x}_{l+1}^{(m)}}\right) \boxed{\boldsymbol{x}_{l+1}^{(m)}}$
13: $\partial \beta_l^{(m)} \leftarrow \sum \partial \boldsymbol{x}_{l+1}^{(m)}$
14: $\partial \boldsymbol{X}_l \leftarrow \partial Y_l \hat{\boldsymbol{W}}_l^{\mathrm{T}}$
15: $\partial \boldsymbol{W}_l \leftarrow \hat{\boldsymbol{X}}_l^{\mathrm{T}} \partial Y_l$
16:
17: **for** $l \leftarrow \{1, \cdots, L-1\}$ **do** ▷ Weight update
18: $\boldsymbol{W}_l \leftarrow \mathrm{Optimize}(\boldsymbol{W}_l, \partial \boldsymbol{W}_l, \eta)$
19: $\boldsymbol{\beta}_l \leftarrow \mathrm{Optimize}\left(\boldsymbol{\beta}_l, \partial \boldsymbol{\beta}_l, \eta\right)$
20: $\eta \leftarrow \mathrm{LearningRateSchedule}(\eta)$

Algorithm 2 Proposed BNN training step.

1: **for** $l \leftarrow \{1, \cdots, L-1\}$ **do** ▷ Forward prop.
2: $\hat{\boldsymbol{X}}_l \leftarrow \mathrm{sgn}(\boldsymbol{X}_l)$
3: $\hat{\boldsymbol{W}}_l \leftarrow \mathrm{sgn}(\boldsymbol{W}_l)$
4: $Y_l \leftarrow \hat{\boldsymbol{X}}_l \hat{\boldsymbol{W}}_l$
5: **for** $m \leftarrow \{1, \cdots, M_l\}$ **do** ▷ Batch norm.
6: $\psi_l^{(m)} \leftarrow \left\| \boldsymbol{y}_l^{(m)} - \mu\left(\boldsymbol{y}_l^{(m)}\right) \right\|_1 / B$
7: $\boldsymbol{x}_{l+1}^{(m)} \leftarrow \frac{\boldsymbol{y}_l^{(m)} - \mu\left(\boldsymbol{y}_l^{(m)}\right)}{\psi_l^{(m)}} + \beta_l^{(m)}$
8: $\omega_{l+1}^{(m)} \leftarrow \left\| \boldsymbol{x}_{l+1}^{(m)} \right\|_1 / B$
9: **for** $l \leftarrow \{L-1, \cdots, 1\}$ **do** ▷ Backward prop.
10: **for** $m \leftarrow \{1, \cdots, M_l\}$ **do** ▷ Batch norm.
11: $\boldsymbol{v} \leftarrow \frac{1}{\psi_l^{(m)}} \partial \boldsymbol{x}_{l+1}^{(m)}$
12: $\partial \boldsymbol{y}_l^{(m)} \leftarrow \boldsymbol{v} - \mu(\boldsymbol{v}) - \mu\left(\boldsymbol{v} \odot \boxed{\hat{\boldsymbol{x}}_{l+1}^{(m)} \omega_{l+1}^{(m)}}\right) \boxed{\hat{\boldsymbol{x}}_{l+1}^{(m)}}$
13: $\partial \beta_l^{(m)} \leftarrow \sum \partial \boldsymbol{x}_{l+1}^{(m)}$
14: $\partial \boldsymbol{X}_l \leftarrow \partial Y_l \hat{\boldsymbol{W}}_l^{\mathrm{T}}$
15: $\partial \boldsymbol{W}_l \leftarrow \hat{\boldsymbol{X}}_l^{\mathrm{T}} \partial Y_l$
16: $\partial \hat{\boldsymbol{W}}_l \leftarrow \mathrm{sgn}(\partial \boldsymbol{W}_l)$
17: **for** $l \leftarrow \{1, \cdots, L-1\}$ **do** ▷ Weight update
18: $\boldsymbol{W}_l \leftarrow \mathrm{Optimize}\left(\boldsymbol{W}_l, \partial \hat{\boldsymbol{W}}_l / \sqrt{M_{l-1}}, \eta\right)$
19: $\boldsymbol{\beta}_l \leftarrow \mathrm{Optimize}\left(\boldsymbol{\beta}_l, \partial \boldsymbol{\beta}_l, \eta\right)$
20: $\eta \leftarrow \mathrm{LearningRateSchedule}(\eta)$

$\hat{\bullet}$ denotes binary encoding. Our refinements are shown in red. Dashed boxes highlight Algorithm 2's lack of high-precision activations.

Table 2
Exemplary memory-related properties of variables used during CIFAR-10 training of BinaryNet with Adam and a batch size of 100.

Variable	Per-layer lifetime[1]	Standard training			Proposed training		
		Data type	Modeled memory (MiB)	%	Data type	Modeled memory (MiB)	Δ (× ↓)
$\boldsymbol{X}$	✘	f32	111.33	21.71	bool	3.48	32.00
$\partial\boldsymbol{X}$, $\boldsymbol{Y}^2$	✔	f32	50.00	9.75	f16	25.00	2.00
$\mu(\boldsymbol{y}_l)$, $\sigma(\boldsymbol{y}_l)$	✘	f32	0.03	0.00	f16	0.02	2.00
$\partial\boldsymbol{Y}$	✔	f32	50.00	9.75	f16	25.00	2.00
$\boldsymbol{W}$	✘	f32	53.49	10.43	f16	26.74	2.00
$\partial\boldsymbol{W}$	✘	f32	53.49	10.43	bool	1.67	32.00
$\boldsymbol{\beta}$, $\partial\boldsymbol{\beta}$	✘	f32	0.03	0.00	f16	0.02	2.00
Momenta	✘	f32	106.98	20.86	f16	53.49	2.00
Pooling masks	✘	f32	87.46	17.06	bool	2.73	32.00
Total			512.81	100.00		138.15	3.71

[1] ✔ indicates that a variable does not need to be retained between forward, backward or update phases.
[2] $\partial\boldsymbol{X}$ and $\boldsymbol{Y}$ can share memory since they are equally sized and have non-overlapping lifetimes.

Many authors have established that BNNs require batch normalization in order to avoid gradient explosion [2, 33, 35], and our early experiments confirmed this to indeed be the case. We thus apply it as standard. Matrix products $\boldsymbol{Y}_l$ are channel-wise batch-normalized across each layer's M_l output channels (lines 5–7) to form the subsequent layer's inputs, $\boldsymbol{X}_{l+1}$. $\boldsymbol{\beta}$ constitutes the batch normalization biases. Layer-wise moving means $\mu(\boldsymbol{y}_l)$ and standard deviations $\sigma(\boldsymbol{y}_l)$ are retained for use during backward propagation and inference. We forgo trainable scaling factors; these are irrelevant to BNNs since their activations are binarized thereafter (line 2).

As emphasized in both Fig. 1 and Algorithm 1 (line 12), *high-precision* storage of the entire network's activations is required. Addressment of this forms our key contribution.

4 VARIABLE ANALYSIS

In order to quantify the potential gains from approximation, we conducted a variable representation and lifetime analysis of Algorithm 1 following the approach taken by Sohoni *et al.* [37]. Table 2 lists the properties of all variables in Algorithm 1, with each variable's contribution to the total footprint shown for a representative example. Variables are divided into two classes: those that must remain in memory between computational phases (forward propagation, backward propagation, and weight update), and those that need not. This is of pertinence since, for those in the latter category, only the largest layer's contribution counts towards the total memory occupancy. For example, $\partial\boldsymbol{X}_l$ is read during the backward propagation of layer $l-1$ only, thus $\partial\boldsymbol{X}_{l-1}$ can safely overwrite $\partial\boldsymbol{X}_l$ for efficiency. Additionally, $\boldsymbol{Y}$ and $\partial\boldsymbol{X}$ are shown together since they are equally sized and only need to reside in memory during the forward and backward pass for each layer, respectively.

5 LOW-COST BNN TRAINING

As shown in Table 2, all variables within the standard BNN training flow use `float32` representation. In the subsections that follow, we detail the application of aggressive approximation specifically

tailored to BNN training. Further to this, and in line with the observation by many authors that `float16` can be used for ImageNet training without inducing accuracy loss [17, 32, 44], we also switch all remaining variables to this format. Our final training procedure is captured in Algorithm 2, with modifications from Algorithm 1 in red and the corresponding data representations used shown in Table 2.

5.1 Batch Normalization Approximation

Analysis of the backward pass of Algorithm 1 reveals conflicting requirements for the precision of $\boldsymbol{X}$. When computing weight gradients $\partial \boldsymbol{W}$ (line 15), only binary activations $\hat{\boldsymbol{X}}$ are needed. For the batch normalization training (lines 10–13), however, high-precision $\boldsymbol{X}$ is used. The latter occurrences are highlighted with dashed boxes. Per Table 2, the storage of $\boldsymbol{X}$ between forward and backward propagation constitutes the single largest portion of the algorithm's total memory. If we are able to use $\hat{\boldsymbol{X}}$ in place of $\boldsymbol{X}$ for these operations, there will be no need for this high-precision activation retention, significantly reducing memory footprint as a result. We achieve this as follows.

Step 1: ℓ_1 Normalization. Standard batch normalization sees channel-wise ℓ_2 normalization performed on each layer's centralized activations. Wu *et al*, however, shows that the less-costly ℓ_1 normalization is approximately equivalent to the original ℓ_2 normalization, by proving that ℓ_1 normalization is approximately equivalent to the original ℓ_2 normalization multiplied with a fixed scaling factor equal to $\sqrt{\pi/2}$ [48]. We argue that this observation is especially true for BNNs, in which batch normalization is immediately followed by binarization, thus canceling the effects of any scaling factor.

Replacement of batch normalization's backward propagation operation with our ℓ_1 norm-based version sees lines 11–12 of Algorithm 1 swapped with (1), where B is the batch size. Not only does our use of ℓ_1 batch normalization transform one occurrence of $\boldsymbol{x}_{l+1}^{(m)}$ into its binary form, it also beneficially eliminates all squares and square roots.

$$\begin{aligned} \boldsymbol{v} &\leftarrow \frac{1}{\left\|\boldsymbol{y}_l^{(m)} - \mu\left(\boldsymbol{y}_l^{(m)}\right)\right\|_1 / B} \partial \boldsymbol{x}_{l+1}^{(m)} \\ \partial \boldsymbol{y}_l^{(m)} &\leftarrow \boldsymbol{v} - \mu(\boldsymbol{v}) - \mu\left(\boldsymbol{v} \odot \boldsymbol{x}_{l+1}^{(m)}\right) \hat{\boldsymbol{x}}_{l+1}^{(m)} \end{aligned} \tag{1}$$

Our derivation of this function is as follows. Let

$$\boldsymbol{a} = \boldsymbol{y}_l - \mu(\boldsymbol{y}_l)$$

and

$$\boldsymbol{v} = \frac{1}{\frac{\|\boldsymbol{a}\|}{B}} \frac{\partial C}{\partial \boldsymbol{x}_{l+1}},$$

so that our forward function in line 7 becomes

$$\boldsymbol{x}_{l+1}^{(m)} \leftarrow \frac{a^{(m)}}{\psi_l^{(m)}} + \beta_l^{(m)}.$$

We compute the expression for gradient $\partial C/\partial \boldsymbol{y}_l$ by first computing $\partial C/\partial \boldsymbol{a}$, which can be derived with chain rule,

$$\frac{\partial C}{\partial a^{(m)}} = \frac{\partial C}{\partial \boldsymbol{x}_{l+1}^{(m)}} \cdot \frac{\partial \boldsymbol{x}_{l+1}^{(m)}}{\partial a^{(m)}} + \frac{\partial C}{\partial \psi_l^{(m)}} \cdot \frac{\partial \psi_l^{(m)}}{\partial a^{(m)}}$$
$$= \frac{\partial C}{\partial \boldsymbol{x}_{l+1}^{(m)}} \cdot \frac{\partial \boldsymbol{x}_{l+1}^{(m)}}{\partial a^{(m)}} + \frac{\partial C}{\partial \boldsymbol{x}_{l+1}^{(m)}} \cdot \frac{\partial \boldsymbol{x}_{l+1}^{(m)}}{\partial \psi_l^{(m)}} \cdot \frac{\partial \psi_l^{(m)}}{\partial a^{(m)}}.$$

By evaluating each component in the above equation, we have

$$\frac{\partial C}{\partial \boldsymbol{a}} = \boldsymbol{v} - \frac{\boldsymbol{a}}{|\boldsymbol{a}|} \times \frac{1}{\left(\frac{\|\boldsymbol{a}\|}{B}\right)^2} \times \mu\left(\boldsymbol{a} \odot \frac{\partial C}{\partial \boldsymbol{x}_{l+1}}\right)$$

and thus

$$\frac{\partial C}{\partial \boldsymbol{y}_l} = \frac{\partial C}{\partial \boldsymbol{a}} - \mu\left(\frac{\partial C}{\partial \boldsymbol{a}}\right)$$
$$= (\boldsymbol{v} - \mu(\boldsymbol{v})) -$$
$$\left(\frac{\boldsymbol{x}_{l+1}}{\frac{\|\boldsymbol{a}\|}{B} \odot |\boldsymbol{a}|} - \mu\left(\frac{\boldsymbol{x}_{l+1}}{\frac{\|\boldsymbol{a}\|}{B} \odot |\boldsymbol{a}|}\right)\right)\left(\mu\left(\boldsymbol{a} \odot \frac{\partial C}{\partial \boldsymbol{x}_{l+1}}\right)\right).$$

Since the output of batch normalization, $\boldsymbol{x}_{l+1}$, is expected to have a mean value of zero across samples in a batch, *i.e.*,

$$\mu(\boldsymbol{x}_{l+1}) \approx \boldsymbol{0},$$

we have

$$\frac{\partial C}{\partial \boldsymbol{y}_l} \approx (\boldsymbol{v} - \mu(\boldsymbol{v})) - \mu(\boldsymbol{v} \odot \boldsymbol{x}_{l+1})\hat{\boldsymbol{x}}_{l+1}.$$

Step 2: BNN-Specific Approximation. We further replace the remaining $\boldsymbol{x}_{l+1}^{(m)}$ term in (1) with the product of its signs and mean magnitude— $\hat{\boldsymbol{x}}_{l+1}^{(m)}\omega_{l+1}^{(m)}$—where $\omega_{l+1}^{(m)}$ is precomputed (line 8).

Our complete batch normalization training functions are shown on lines 10–13 of Algorithm 2. As again highlighted within dashed boxes, these only require the storage of binary $\hat{\boldsymbol{X}}$ along with layer- and channel-wise mean magnitudes. With elements of $\boldsymbol{X}$ now binarized, we reduce its memory cost by 32× and also save energy thanks to the corresponding memory traffic reduction.

5.2 Weight Gradient Quantization

In common with other BNN training approaches, we employ “straight-through estimation” (STE) to facilitate gradient propagation in the presence of discretization in forward functions. STE approximates the gradient of a discontinuity by disregarding the derivative of the discretizer itself. As shown in Table 2, `float32` gradients were typically used with STE in the past. Intuitively, BNNs should be particularly robust to weight gradient quantization since their weights only constitute signs. On line 16 of Algorithm 2, therefore, we binarize and store post-STE weight gradients, $\partial\hat{\boldsymbol{W}}$, for weight update. During that phase, we attenuate the gradients by $\sqrt{N_l}$, where N_l is layer l’s fan-in, to reduce the learning rate and prevent premature weight clipping as advised by Sari *et al.* [35] (line 18). Since fully connected layers are used as an example in Algorithm 2, $N_l = M_{l-1}$ in this instance.

Table 2 shows that, with binarization, the portion of our exemplary training run’s memory consumption attributable to weight gradients dropped from 53.49 to just 1.67 MiB, leaving the

scarce resources available for more quantization-sensitive variables such as $\boldsymbol{W}$ and momenta. Energy consumption will also decrease due to the associated reduction in memory traffic.

6 EVALUATION

6.1 Keras Emulation

We built a GPU-based implementation emulating our BNN training method using Keras and TensorFlow, and experimented with the small-scale MNIST, CIFAR-10, and SVHN datasets, as well as large-scale ImageNet, using a range of network models. By emulating our algorithm on GPU, we can leverage the many powerful ML training softwares developed around it, and obtain large batches of experimental results in a short period of time. Our emulation environment is built on a Nvidia GeForce RTX 3090 GPU cluster with Red Hat Linux 9 operating system. Our baseline for comparison was the standard BNN training method introduced by Courbariaux & Bengio [13], and we followed those authors' practice of reporting the highest test accuracy achieved in each run. Note that we did not tune hyperparameters, thus it is likely that higher accuracy than we report is achievable.

6.1.1 Small-Scale Datasets. For MNIST we evaluated using a five-layer MLP—henceforth simply denoted "MLP"—with 256 neurons per hidden layer, and CNV [40] and BinaryNet [13] for both CIFAR-10 and SVHN. We used three popular BNN optimizers: Adam [26], stochastic gradient descent (SGD) with momentum, and Bop [22]. While all three function reliably with our training scheme, we used Adam by default due to its stability. We used the development-based learning rate scheduling approach proposed by Wilson *et al.* [46] with an initial learning rate η of 0.001 for all optimizers except for SGD with momentum, for which we used 0.1. We used batch size $B = 100$ for all except for Bop, for which we used $B = 50$ as recommended by Helwegen *et al.* [22]. MNIST and CIFAR-10 were trained for 1000 epochs; SVHN for 200.

Our choice of quantization targets primarily rested on the intuition that BNNs should be more robust to approximation in backward propagation than their higher-precision counterparts. To illustrate that this is indeed the case, we applied our method to both BNNs and `float32` networks, with identical topologies and hyperparameters. Results of those experiments are shown in Table 3, in which significantly higher accuracy degradation was observed for the non-binary networks, as expected.

While our proposed BNN training method does exhibit limited accuracy degradation, as can be seen for three cases in Table 4, this comes in return for a geomean modeled memory saving of 3.67×. It is also interesting to note that the reduction achievable for a given dataset depends on the model used. This observation is largely orthogonal to our work: by applying our approach to the training of a more appropriately chosen model, one can obtain the advantages of both optimized network selection and training.

In order to explore the impacts of the various facets of our scheme, we applied them sequentially while training BinaryNet to classify CIFAR-10 with multiple optimizers. As shown in Table 5, choices of data type, optimizer, and batch normalization implementation lead to tradeoffs against performance and memory costs. Major savings are attributable to the use of `float16` variables and through the high-precision activation elimination our ℓ_1 norm-based batch normalization facilitates.

Fig. 2 shows the modeled memory footprint savings from our proposed BNN training method for different optimizers and batch sizes, again for BinaryNet with the CIFAR-10 dataset. Across all of these, we achieved a geomean reduction of 4.81×. Also observable from Fig. 2 is that, for all optimizers, movement from the standard to our proposed BNN training allows the batch size used to increase by around 10×, facilitating faster completion, without a material memory increase. Fig. 2 finally shows that test accuracy does not drop significantly due to our approximations. With Adam

Table 3
Test accuracy of non-binary networks and BNNs using the standard and our proposed training approaches with Adam and a batch size of 100. Results for our training approach applied to the former are included for reference only; we do not advocate for its use with non-binary networks.

Model	Dataset	Top-1 test accuracy						
		Standard training			Reference training		**Proposed training**	
		NN (%)[1]	BNN (%)	Δ (pp)	NN (%)[1]	Δ (pp)[2]	**BNN (%)**	**Δ (pp)[3]**
MLP [40]	MNIST	98.22	98.24	0.02	89.98	−8.24	**96.90**	**−1.34**
CNV [40]	CIFAR-10	86.37	82.67	−3.70	69.88	−16.49	**83.08**	**0.41**
CNV	SVHN	97.30	96.37	−0.93	79.44	−17.86	**94.28**	**−2.09**
BinaryNet [13]	CIFAR-10	88.20	88.74	1.61	76.56	−11.64	**89.09**	**0.35**
BinaryNet	SVHN	96.54	97.40	0.86	85.71	−10.83	**95.93**	**−1.47**

[1] Non-binary neural network.
[2] Baseline: non-binary network with standard training.
[3] Baseline: BNN with standard training.

Table 4
Test accuracy and memory footprint of the standard and our proposed training schemes using Adam and a batch size of 100.

Model (Dataset)	Top-1 test accuracy			Modeled memory		
	Std. (%)	**Prop. (%)**	**Δ (pp)**	Std. (MiB)	**Prop. (MiB)**	**Δ (× ↓)**
MLP (MNIST)	98.24	**96.90**	**−1.34**	7.40	**2.65**	**2.78**
CNV (CIFAR-10)	82.67	**83.08**	**0.41**	134.05	**32.16**	**4.17**
CNV (SVHN)	96.37	**94.28**	**−2.09**	134.05	**32.16**	**4.17**
BinaryNet (CIFAR-10)	88.74	**89.09**	**0.35**	512.81	**138.15**	**3.71**
BinaryNet (SVHN)	97.40	**95.93**	**−1.47**	512.81	**138.15**	**3.71**

and Bop, accuracy was near-identical, while with SGD we actually saw modest improvements. Unlike Adam or Bop, the standard SGD optimizer is unable to adapt its learning rate during gradient descent, thus scaling in batch size means scaling in learning rate. This leads to the decline in accuracy we see in Fig. 2(b), where increasing the batch size leads to undesirable learning rates. Our method, on the other hand, binarizes the weight gradients, which effectively normalizes the learning rate from the effects of batch size scaling.

While not of concern with regards to memory consumption, decreases in convergence rate are undesirable due to their elongation of training times and, consequently, reduction of energy efficiency. In order to ensure that our algorithmic modifications do not cause material convergence rate degradation, we inspected the validation accuracy curves obtained during our training runs.

Table 5
Impacts of moving from the standard to our proposed data representations with BinaryNet and CIFAR-10 and a batch size of 100.

Optimizer	Data type		Batch normalization	Top-1 test accuracy		Modeled memory	
	∂W	∂Y		%	Δ (pp)[1]	MiB	Δ (× ↓)[1]
Adam	float32	float32	ℓ_2	88.74	–	512.81	–
	float16	float16	ℓ_2	88.71	−0.03	256.41	2.00
	bool	float16	ℓ_2	87.93	−0.81	231.33	2.22
	bool	float16	ℓ_1	89.69	0.95	231.33	2.22
	bool	float16	Proposed	89.09	0.35	138.15	3.71
SGD with momentum	float32	float32	ℓ_2	88.52	–	459.32	–
	float16	float16	ℓ_2	88.54	0.02	229.66	2.00
	bool	float16	ℓ_2	87.35	−1.17	204.58	2.25
	bool	float16	ℓ_1	89.09	0.57	204.58	2.25
	bool	float16	Proposed	88.10	−0.42	109.20	4.21
Bop	float32	float32	ℓ_2	91.38	–	405.83	–
	float16	float16	ℓ_2	91.36	−0.02	202.92	2.00
	bool	float16	ℓ_2	90.54	−0.84	177.84	2.28
	bool	float16	ℓ_1	91.27	−0.11	177.84	2.28
	bool	float16	Proposed	91.48	0.10	82.45	4.92

[1] Baseline: float32 $\partial \boldsymbol{W}$ and $\partial \boldsymbol{X}$ with standard (ℓ_2) batch normalization.

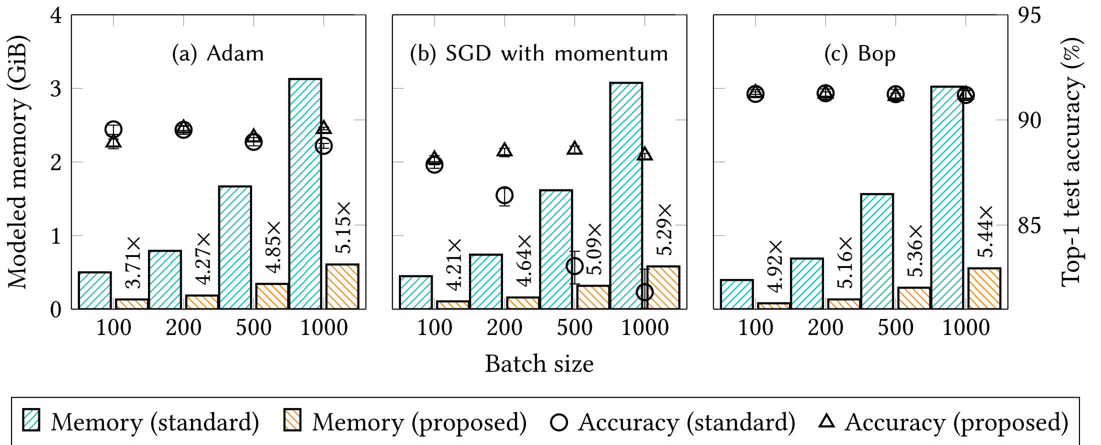

Fig. 2. Batch size *vs* training memory footprint and achieved test accuracy for BinaryNet with CIFAR-10. Annotations show memory reductions for the proposed training approach. Each test accuracy point marks the mean of five independent training runs, with an error bar indicating its distribution.

Figs. 3 and 4 exemplify these for the experiments whose results were reported in Table 4 and Fig. 2, respectively. No discernible change in convergence rate can be seen in any of the plots, thus we can be confident that our proposals will not negatively impact training times.

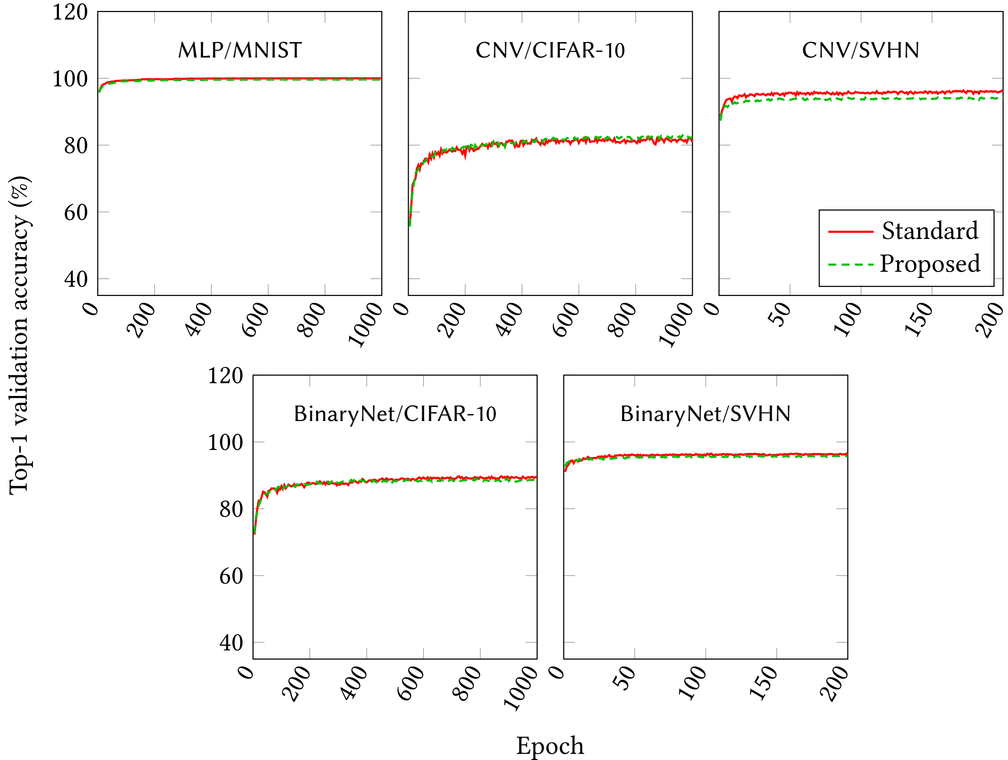

Fig. 3. Comparison in achieved validation accuracy curves between the standard and our proposed training schemes with multiple combinations in models and datasets, using Adam and a batch size of 100. These plots correspond to results which are reported in Table 4.

For the results presented thus far, we made use of off-the-shelf network models. As confirmed by Zhang *et al.*, a network possess perfect expressivity once its number of parameters matches the number of data points used for its training [50]. Consequently, most practical networks are overparameterized. While the impact of overparametrization on network generalization is an active research field [9] and outside the scope of this work, we sought to investigate whether overparametrization was the source of robustness to gradient approximation that we observed of BNNs. To do this, we performed neural architecture search (NAS) for the MNIST, CIFAR-10 and SVHN datasets, comparing the impact of removing network redundancy on both the standard and our training approaches. We adopted Shen *et al.*'s approach to BNN NAS, applying it to the MLP and BinaryNet models as starting points [36]. Following their proposals, we set accuracy-to-parameter weight factor λ to 0.1 for MLP with MNIST and 0.01 for BinaryNet with CIFAR-10 and SVHN. As shown in Table 6, we achieved sizeable parameter reductions for all of these and, most importantly, observed no difference in accuracy degradation for the two training approaches. These experiments therefore suggest that the reduction of network complexity impacts both methods equally, and that the performance of ours is not reliant on overparameterization.

6.1.2 ImageNet. We also trained ResNetE-18 [4] and Bi-Real-18 [29]—mixed-precision models with most convolutional layers binarized—to classify ImageNet. These models are representative of a broad class of ImageNet-capable networks, thus similar results should be achievable for others with

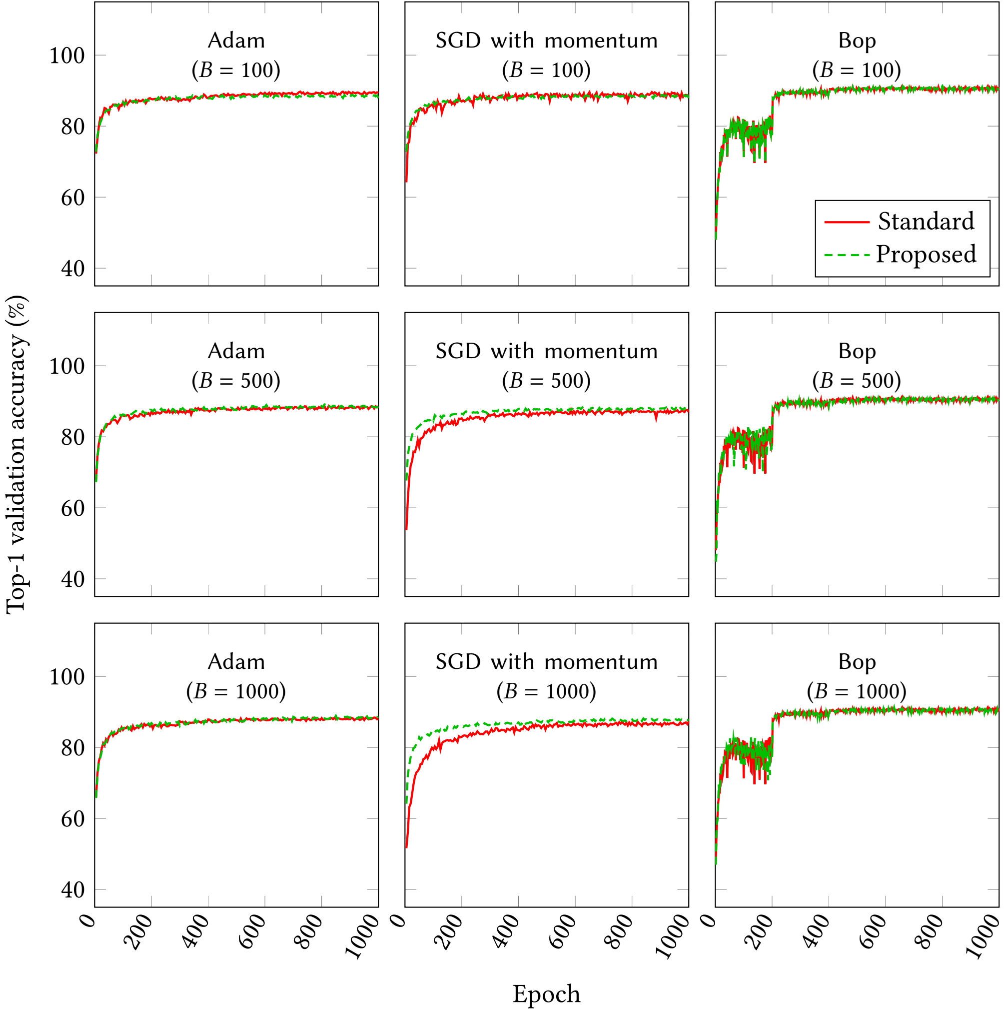

Fig. 4. Comparison in achieved validation accuracy curves between the standard and our proposed training schemes with multiple combinations in optimizers and batch sizes (B), using BinaryNet model and CIFAR-10 dataset. These plots correspond to results which are reported in Fig. 2.

Table 6
Model complexity and test accuracy impacts of NAS under the standard and proposed training schemes.

Model	Dataset	Parameters (M)			Top-1 test accuracy					
		Pre	Post		Standard training			**Proposed training**		
		#	#	Δ (× ⇃)	Pre (%)	Post (%)	Δ (pp)	**Pre (%)**	**Post (%)**	Δ **(pp)**
MLP	MNIST	0.40	0.16	2.52	98.24	97.58	−0.66	**96.90**	**96.35**	**−0.55**
BinaryNet	CIFAR-10	14.02	3.62	3.87	88.74	87.14	−1.60	**89.09**	**87.17**	**−1.92**
BinaryNet	SVHN	14.02	3.77	3.72	97.40	97.26	−0.14	**95.93**	**95.38**	**−0.55**

Table 7
Test accuracy and memory footprint of the standard and proposed schemes for ImageNet training with Adam and a batch size of 4096.

Approximations	ResNetE-18				Bi-Real-18			
	Top-1 test acc.		Modeled memory		Top-1 test acc.		Modeled memory	
	%	Δ (pp)[1]	GiB	Δ (× ↓)[1]	%	Δ (pp)[1]	GiB	Δ (× ↓)[1]
None	58.77	–	70.11	–	56.71	–	70.11	–
All-`bfloat16`	58.85	0.08	35.45	1.98	56.72	0.01	35.45	1.98
`bool` $\partial \boldsymbol{W}$ only	57.59	−1.28	70.07	1.00	55.69	−1.02	70.07	1.00
ℓ_1 batch norm. only	58.34	−0.43	70.11	1.00	56.08	−0.63	70.11	1.00
Prop. batch norm. only	58.23	−0.54	47.86	1.46	55.59	−1.12	47.86	1.46
Proposed	**57.04**	**−1.73**	**18.54**	**3.78**	**54.45**	**−2.26**	**18.54**	**3.78**

[1] Baseline: approximation-free training.

which they share architectural features. Finding development-based learning rate scheduling to not work well with ResNetE-18, we resorted to the fixed decay schedule described by Bethge *et al.* [4]. η began at 0.016 and decayed by a factor of 10 at epochs 70, 90, and 110. We trained for 120 epochs with $B = 4096$. For Bi-Real-18, we trained for 80 epochs with $B = 512$ and a cosine-decaying learning rate starting from $\eta = 0.001$. Both models were optimized using Adam.

We show the performance of these benchmarks when applying each of our proposed approximations in turn, as well their assemblage, in Table 7. Since the Tensor Processing Units we used here natively support `bfloat16` rather than `float16`, we switched to the former for these experiments. Where `bfloat16` variables were used, these were employed across all layers; the remaining approximations were applied only to binary layers. While these savings are smaller than those for our small-scale experiments, we note that the first convolutional layer of both ResNetE-18 and Bi-Real-18 is the largest and is non-binary, thus its activation storage dwarfs that of the remaining layers. We also remark that, while ~2 pp accuracy drops may not be acceptable for some application deployments, sizable memory reductions are otherwise achievable. The effects of binarized $\partial \boldsymbol{W}$ are insignificant since ImageNet's large images result in proportionally small weight memory occupancy.

We acknowledge that dataset storage requirements likely render ImageNet training on edge platforms infeasible, and that network fine-tuning is a task more commonly deployed on devices of such scale. However, given that the accuracy changes and resource savings we report for more challenging, from-scratch training are favorable and reasonably consistent across a wide range of use-cases, we have confidence that positive results are readily achievable for fine-tuning as well. Nevertheless, our ImageNet proof of concept confirms the efficacy of large-scale neural network training on the edge.

In common with our small-scale experiments, our proposals did not lead to noticeable convergence rate changes *vs* the standard BNN training algorithm. This is evident from Fig. 5, which contains the validation accuracy curves obtained for the experiments whose results were reported in Table 7.

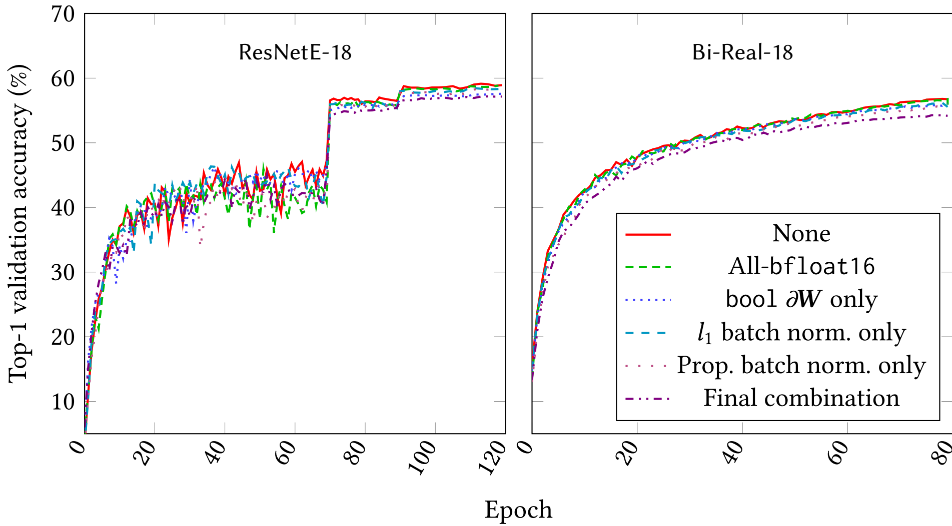

Fig. 5. Achieved validation accuracy over time for the experiments whose results are reported in Table 7.

6.2 Embedded Platform Prototypes

To more concretely demonstrate the benefits of our proposed training method, we also wrote software targeting an embedded-scale computing platform. We chose to use a Raspberry Pi 3B+, a popular single-board computer with hardware representative of current mobile and other edge devices, for this purpose. The platform features a four-core, 64-bit Arm Cortex-A53 CPU clocked at 1.4 GHz and 1 GiB of LPDDR2 RAM. We used the PyPI `memory_profiler` module and Valgrind to monitor the memory occupancy of Keras- and C++-based implementations, respectively. Energy consumption was logged with a standard USB power meter connected to the Raspbberry Pi's external power supply [34].

6.2.1 Naïve C++ Implementation. While existing training frameworks, including TensorFlow and PyTorch, allow for some data format customization, they lack support for direct control of variable storage. Moreover, when in training mode, they tend to reserve hundreds of MiBs of memory regardless of the model size, making their use infeasible on edge devices. TensorFlow-lite delivers low-memory inference, but it does not support training. Therefore, while these existing frameworks are useful for accuracy evaluation, implementations of our approach that realize its promised memory advantage must be built from scratch. Our first prototypes were direct implementations of Algorithms 1 and 2 in C++. We also trained using Keras, where possible within the Raspberry Pi's memory limit, for comparison.

Measurements of the peak memory use of our naïve C++ prototypes prove the validity of our memory model. As reflected in Fig. 6, two effects cause the model to produce underestimates. There is a constant, ~5% memory increase across all experiment pairs. This is attributable to process overheads, which we left unmodeled. There is also a second, batch size-correlated overhead due to activation copying between layers. This is significantly more pronounced for the standard algorithm due to its use of `float32`—rather than `bool`—activations. While we did not model these copies since they are not strictly necessary, their avoidance would have unbeneficially complicated our software.

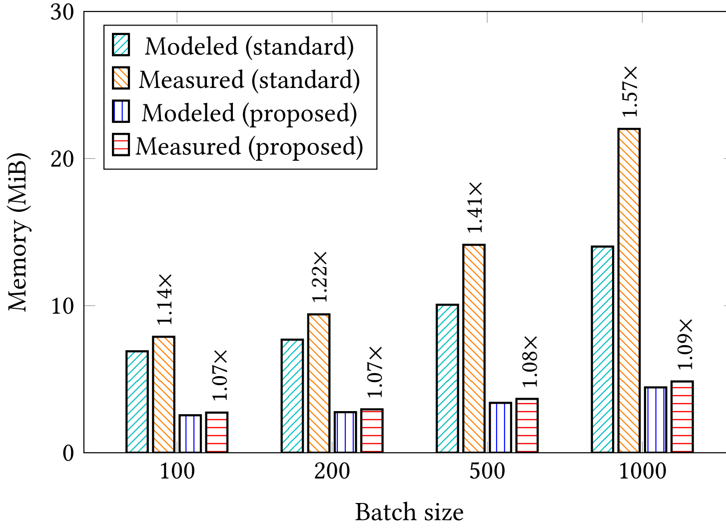

Fig. 6. Batch size *vs* memory footprint for our naïve C++ prototypes training MLP to classify MNIST with Adam. Annotations mark the ratio between measured and modeled memory pairs.

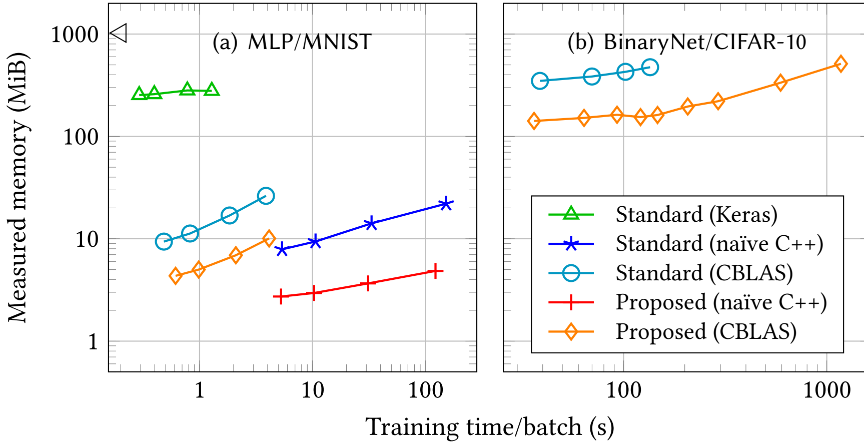

Fig. 7. Measured peak memory consumption *vs* training time (a)–(b) per batch for implementations training MLP/MNIST and BinaryNet/CIFAR-10. Each data point represents a distinct batch size. BinaryNet/CIFAR-10 training with Keras was not possible due to the Raspberry Pi's memory limit (◁).

Figs. 7(a) and 7(b) show the measured memory footprint *vs* training time for the naïve (standard and proposed) and Keras implementations across a range of batch sizes. For MLP trained to classify MNIST, our naïve implementation saw memory requirements reduce by 2.90–4.54× *vs* the standard approach, with no impact on speed. While use of Keras led to much shorter training times, this came at the cost of superproportional memory increases: two orders of magnitude higher than the demands of the proposed approach. Keras-based training of BinaryNet is not possible due to the platform's 1 GiB memory limit. Keras' training backend uses methods which buffer additional

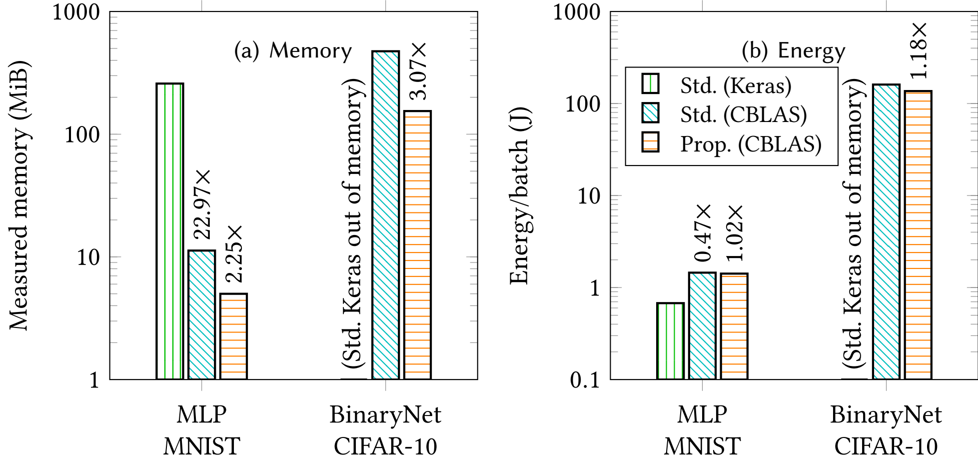

Fig. 8. Measured peak memory consumption (a) and energy consumption per batch (b) for implementations training MLP/MNIST and BinaryNet/CIFAR-10. Batch sizes of 200 and 40 were chosen for MLP and BinaryNet, respectively. BinaryNet/CIFAR-10 training with Keras was not possible due to the Raspberry Pi's memory limit. Annotations show decreases *vs* the bar to the left. The energy savings in (b) were less significant than memory savings in (a), since the memory traffic-associated energy reductions are partially offset by the costs of `bool`-packing (and -unpacking) operations.

copies of data to optimize for training speed and, as far as we know, the option is not exposed for parametrization [25].

6.2.2 CBLAS Acceleration. In a bid to close our training time gap with Keras, we optimized our prototypes using the CBLAS library, trading off memory for speed [5]. As shown in Fig. 7(a), this reimplementation led to reductions in training times of an order of magnitude with MLP, making our optimized implementations reach similar speed to Keras. While the CBLAS-accelerated proposed algorithm requires 1.59–2.08× more memory than its naïve counterpart, this comes in return for speedups of 8.60–29.76× while remaining 2.16–2.61× more memory-efficient than the standard approach with acceleration. Our approach with CBLAS bettered Keras' memory requirements by 27.66–58.34× while experiencing slowdowns of 2.10–3.22×. Experiments with BinaryNet and CIFAR-10 showed similar trends, with the accelerated standard implementation failing to run with a batch size over 40. Note that, due to operating system overheads, it was not possible for the running training program to occupy all of the platform's memory. In our CBLAS implementation, the additional data format conversions between floating point and boolean were efficiently accelerated with ARM's single-cycle VCVT instructions. ARM also features native support for fp16 format with VFPv3 architecture in more advanced devices, which would further advance our memory savings.

Energy Efficiency. In addition to memory savings, our use of low-precision activations and gradients also reduces memory traffic, leading to reduced energy consumption. Fig. 8 shows the measured memory footprint and energy consumption per epoch for both MLP with MNIST and BinaryNet with CIFAR-10. For the batch sizes we tested, the CBLAS-accelerated implementation of our proposed training method surpasses the equally optimized standard approach in terms of energy efficiency by 1.02× and 1.18× for those respective network-dataset pairs. We remark that these savings shown in Fig. 8(b) are not as significant when compared against the huge memory reductions shown in Fig. 8(a), since data movement cost only accounts for a portion of the overall

energy cost, and the memory traffic-associated energy reductions are partially offset by the costs of `bool`-packing (and -unpacking) operations at output (and input) to *every* non-`float32` GEMM kernel. Due to lack of an assembly level-optimized bit-packing operation in CBLAS library, we opt to implement in our prototypes a C++-based function which revisits all input and output data to the GEMM kernels, leading to extra data movements. This overhead can be reduced by customizing the CBLAS GEMM implementation to perform bit packing (and unpacking) on the fly.

7 CONCLUSION

In this article, we introduced a neural network training scheme tailored specifically to BNNs. Moving first to 16-bit floating-point representation, we selectively and opportunistically approximated beyond this based on careful analysis of the standard training algorithm presented by Courbariaux & Bengio [13]. With a comprehensive evaluation conducted across multiple models, datasets, optimizers, and batch sizes, we showed the generality of our approach and reported significant memory reductions *vs* the prior art, challenging the notion that the resource constraints of edge platforms present insurmountable barriers to on-device learning. We validated the veracity of our claimed savings with Raspberry Pi-targeted prototypes, whose source code we have made openly available for use and further development. In the future, we will explore the potential of our training approximations in the custom hardware domain, within which we expect there to be vast energy-saving opportunity via use of tailor-made arithmetic operators.

8 ACKNOWLEDGMENTS

The authors are grateful for the support of the United Kingdom EPSRC (grant numbers EP/P010040/1 and EP/S030069/1). They also wish to thank Sergey Ioffe and Michele Covell for their helpful suggestions.

For the purpose of open access, the authors will apply a Creative Commons Attribution (CC BY) license to any accepted version of this manuscript.

REFERENCES

[1] Naman Agarwal, Ananda Theertha Suresh, Felix Yu, Sanjiv Kumar, and H. Brendan McMahan. 2018. CpSGD: Communication-Efficient and Differentially-Private Distributed SGD. In *International Conference on Neural Information Processing Systems.*

[2] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. 2018. An Empirical study of Binary Neural Networks' Optimisation. In *International Conference on Learning Representations.*

[3] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. SignSGD: Compressed Optimisation for Non-Convex Problems. In *International Conference on Machine Learning.*

[4] Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. 2019. Back to Simplicity: How to Train Accurate BNNs from Scratch? *arXiv preprint arXiv:1906.08637* (2019).

[5] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[6] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H. Brendan McMahan, Timon van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards Federated Learning at Scale: System Design. In *Conference on Machine Learning and Systems.*

[7] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. Tiny Transfer Learning: Towards Memory-Efficient On-Device Learning. In *IEEE Conference on Computer Vision and Pattern Recognition.*

[8] Ayan Chakrabarti and Benjamin Moseley. 2019. Backprop with Approximate Activations for Memory-efficient Network Training. In *Advances in Neural Information Processing Systems.*

[9] Satrajit Chatterjee and Piotr Zielinski. 2022. On the Generalization Mystery in Deep Learning. *arXiv preprint arXiv:2203.10036* (2022).

[10] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).

[11] Tianlong Chen, Zhenyu Zhang, Xu Ouyang, Zechun Liu, Zhiqiang Shen, and Zhangyang Wang. 2021. "BNN - BN = ?": Training Binary Neural Networks Without Batch Normalization. In *IEEE Conference on Computer Vision and Pattern Recognition.*

[12] George A. Constantinides. 2019. Rethinking Arithmetic for Deep Neural Networks. *Philosophical Transactions of the Royal Society A* 378, 2166 (2019).

[13] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv preprint arXiv:1602.02830* (2016).

[14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. In *Conference on Neural Information Processing Systems.*

[15] Sajad Darabi, Mouloud Belbahri, Matthieu Courbariaux, and Vahid Partovi Nia. 2018. *BNN+: Improved Binary Network Training.* https://openreview.net/pdf?id=SJfHg2A5tQ

[16] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual Binarized Neural Network. In *IEEE International Symposium on Field-Programmable Custom Computing Machines.*

[17] Boris Ginsburg, Sergei Nikolaev, and Paulius Micikevicius. 2017. Training of Deep Networks with Half-precision Float. In *Nvidia GPU Technology Conference.*

[18] Xavier Glorot and Yoshua Bengio. 2010. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics.*

[19] Benjamin Graham. 2017. Low-precision Batch-normalized Activations. *arXiv preprint arXiv:1702.08231* (2017).

[20] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient Backpropagation Through Time. In *Advances in Neural Information Processing Systems.*

[21] Xiangyu He, Zitao Mo, Ke Cheng, Weixiang Xu, Qinghao Hu, Peisong Wang, Qingshan Liu, and Jian Cheng. 2020. ProxyBNN: Learning Binarized Neural Networks via Proxy Matrices. In *European Conference on Computer Vision.*

[22] Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. 2019. Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization. In *Advances in Neural Information Processing Systems.*

[23] Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. 2018. Norm Matters: Efficient and Accurate Normalization Schemes in Deep Networks. In *Advances in Neural Information Processing Systems.*

[24] Xinrui Jiang, Nannan Wang, Jingwei Xin, Keyu Li, Xi Yang, and Xinbo Gao. 2021. Training Binary Neural Network without Batch Normalization for Image Super-Resolution. In *AAAI Conference on Artificial Intelligence.*

[25] Keras. [n. d.]. *memory leak in tf.keras.Model.predict.* https://github.com/tensorflow/tensorflow/issues/44711

[26] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations.*

[27] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards Accurate Binary Convolutional Neural Network. In *Conference on Neural Information Processing Systems*.

[28] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. 2020. ReActNet: Towards Precise Binary Neural Network with Generalized Activation Functions. In *European Conference on Computer Vision*.

[29] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. 2018. Bi-Real Net: Enhancing the Performance of 1-bit CNNs With Improved Representational Capability and Advanced Training Algorithm. In *European Conference on Computer Vision*.

[30] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. 2020. Training Binary Neural Networks with Real-to-binary Convolutions. In *International Conference on Learning Representations*.

[31] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *International Conference on Artificial Intelligence and Statistics*.

[32] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*.

[33] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary Neural Networks: A Survey. *Pattern Recognition* 105 (2020).

[34] reichelt. [n. d.]. *RPI USB METER2*. https://www.reichelt.com/de/en/raspberry-pi-amp-voltmeter-2-way-usb-rpi-usb-meter2-p223623.html?r=1

[35] Eyyüb Sari, Mouloud Belbahri, and Vahid P. Nia. 2019. How Does Batch Normalization Help Binary Training. *arXiv preprint arXiv:1909.09139* (2019).

[36] Mingzhu Shen, Kai Han, Chunjing Xu, and Yunhe Wang. 2019. Searching for Accurate Binary Neural Architectures. In *International Conference on Computer Vision Workshops*.

[37] Nimit S. Sohoni, Christopher R. Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-memory Neural Network Training: A Technical Report. *arXiv preprint arXiv:1904.10631* (2019).

[38] Mariko Tatsumi, Silviu-Ioan Filip, Caroline White, Olivier Sentieys, and Guy Lemieux. 2022. Mixing Low-Precision Formats in Multiply-Accumulate Units for DNN Training. In *2022 International Conference on Field-Programmable Technology (ICFPT)*.

[39] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. 2020. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In *International Conference on Field-Programmable Logic and Applications*.

[40] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip H. W. Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[41] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. 2019. LUTNet: Rethinking Inference in FPGA Soft Logic. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*.

[42] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. 2020. LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference. *IEEE Trans. Comput.* 69, 12 (2020).

[43] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. 2019. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *Comput. Surveys* 52, 2 (2019).

[44] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Advances in Neural Information Processing Systems*.

[45] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *Advances in Neural Information Processing Systems*.

[46] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. 2017. The Marginal Value of Adaptive Gradient Methods in Machine Learning. In *Advances in Neural Information Processing Systems*.

[47] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and Inference with Integers in Deep Neural Networks. In *International Conference on Learning Representations*.

[48] Shuang Wu, Guoqi Li, Lei Deng, Liu Liu, Dong Wu, Yuan Xie, and Luping Shi. 2018. $L1$-Norm Batch Normalization for Efficient Training of Deep Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 30, 7 (2018).

[49] Mikail Yayla and Jian-Jia Chen. 2022. Memory-efficient training of binarized neural networks on the edge. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*.

[50] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. Understanding Deep Learning (Still) Requires Rethinking Generalization. *Commun. ACM* 64, 3 (2021).

[51] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv preprint arXiv:1606.06160* (2016).