To Fold or Not to Fold: a Necessary and Sufficient Condition on Batch-Normalization Layers Folding

Edouard Yvinec[1,2*], **Arnaud Dapogny**[2], **Kevin Bailly**[1,2]
[1]Datakalab, 114 Boulevard Malesherbes, 75017 Paris, France
[2]Sorbonne Université, CNRS, Institut des Systèmes Intelligents et de Robotique, ISIR, Paris, France
{ey@datakalab.com}

Abstract

Batch-Normalization (BN) layers have become fundamental components in the evermore complex deep neural network architectures. Such models require acceleration processes for deployment on edge devices. However, BN layers add computation bottlenecks due to the sequential operation processing: thus, a key, yet often overlooked component of the acceleration process is BN layers folding. In this paper, we demonstrate that the current BN folding approaches are suboptimal in terms of how many layers can be removed. We therefore provide a necessary and sufficient condition for BN folding and a corresponding optimal algorithm. The proposed approach systematically outperforms existing baselines and allows to dramatically reduce the inference time of deep neural networks.

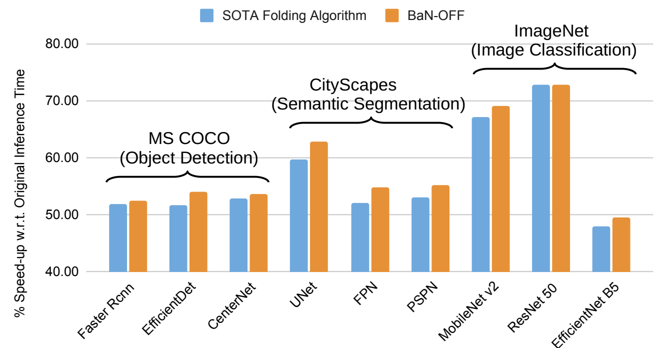

Figure 1: Comparison between the state-of-the-art BN folding algorithm and the proposed BaN-OFF method on several architectures and tasks. Both methods significantly improve the inference speed at virtually no cost. The proposed method systematically matches or outperforms the already existing method on all configurations.

1 Introduction

Deep Neural Networks (DNNs) are ubiquitous in various subdomains of computer vision, e.g. in Image Classification [He *et al.*, 2016], Object Detection [He *et al.*, 2017] or Semantic Segmentation [Chen and others, 2017]. Such models evolved from narrow architectures [LeCun and others, 1989], to deeper and more complex architectures [Tan and Le, 2019]. The main concern in the early developments of very deep neural networks was the efficiency of the learning process [Krizhevsky *et al.*, 2012] with batch-normalization (BN) layers [Ioffe and Szegedy, 2015] being a prime example of solution to this challenge. BN layers center and reduce the features which solves the internal covariate shift problem arising from the change of distribution of the layer inputs. It was so effective that it became almost mandatory in modern architectures.

However, very deep neural network deployment is still limited due to their heavy computational requirements. To tackle this problem, many solutions for DNN acceleration have been developed including, but not limited to, pruning [Frankle and Carbin, 2018; Lin and others, 2020; Yvinec *et al.*, 2021] and quantization [Zhao and others, 2019; Meller *et al.*, 2019; Finkelstein *et al.*, 2019]. Pruning consists in removing a fraction of the mathematical operations performed within each layers while quantization consists in reducing the temporal cost of scalar operations by using lower bit-wise representations. However most of these methods leverage the dependencies across consecutive convolutional and fully-connected layers. Those dependencies are affected by BN layers. This is the first motivation for BN folding. It consists in removing the BN layers from the graph by updating the remaining parameters to keep the predictive function unchanged. It facilitates the use of most pruning and quantization protocols.

The second motivation is straightforward: the lower the number of layers, the faster the inference, all other factors being equal, which is the case with BN folding [Jacob and others, 2018]. Although BN operations adds very few parameters to the network, the induced computational cost is important because of the sequentiality of the operations. The folding process removes this bottleneck and without changing the predictive function but it is still often overlooked in the literature.

For those two reasons, BN folding is usually presented in a naive way where BN layers are folded if and only if they are immediately connected to a convolutional layer or a fully-connected layer. Although, this condition is sufficient for a BN layer to be foldable, it is far from necessary meaning that

*Contact Author

it can't be optimal. In this study, we determine a necessary and sufficient condition for a BN layer to be foldable which translates into an algorithm for optimal batch-normalization layers folding, dubbed **BaN-OFF**. Our tests are two-folds: first, we validate the systematic improvement over the standard folding protocol. Second, we demonstrate the importance as a compression step in more complex pipeline. Our contributions are:

- From a theoretical standpoint: a novel necessary and sufficient condition on the feasibility of batch normalization layers folding in any neural network architecture.
- From a practical standpoint: an algorithm which implements the proposed optimal condition and outperforms the existing folding methods at virtually no cost in preprocessing time.
- Outstanding inference acceleration results on several tasks and architectures and systematic improvement from the standard folding algorithm, see Figure 1.

In this work, we focused on BN as defined by [Ioffe and Szegedy, 2015], however, all our results extend to any normalization layers that implements static affine transformations such as Group Normalization [Wu and He, 2018] or Switchable Normalization [Luo and others, 2019].

2 Related Work

Batch-normalization, introduced by Ioffe *et al.* in [2015], had and continues to have a tremendous impact in deep learning. Prior to it, architectures such as VGG [Simonyan and Zisserman, 2014] achieved remarkable results on ImageNet [Deng *et al.*, 2009] but were difficult to train. However as discussed in [Martin and Mahoney, 2019] where the authors drew a comparison between the standard VGG and VGG-BN where additional BN layers are used. It appeared that such layers drastically improve the performance while simplifying the training process. Concurrently, these layers have been used in almost all novel architectures ([Ren and others, 2015; He *et al.*, 2016; Szegedy and others, 2016; Lin and others, 2017; Zhao and others, 2017; Sandler *et al.*, 2018; Zhou and others, 2019; Tan and Le, 2019; Tan *et al.*, 2020]) to the point that they are now considered an unavoidable part of the standard convolutional and fully-connected layers.

Inference Acceleration. There are two main approaches to DNN inference acceleration: pruning and quantization. Pruning consists in removing elements of the graph defined by the DNN [Renda *et al.*, 2020]. Quantization consists in mapping the weight values from continuous values to evenly-spread integer values using lower bit-wise representations (e.g. int8 or int4) [Krishnamoorthi, 2018]. Both approaches are often combined or even intrinsically leverage BN folding.

Batch-normalization Folding. BN folding consists in removing such BN layers from the network's graph by updating the appropriate adjacent layers. This operation is performed if and only if the structural changes don't affect the predictive function associated to the network. The naive approach [Jacob and others, 2018] is widely documented and applied in DNN acceleration. However it is not optimal as it doesn't fold every foldable BN layers. To solve this issue, we propose BaN-OFF, a necessary and sufficient condition for such layers to be folded as well as its corresponding algorithm.

Batch-normalization Folding and other Accelerations. In [Yvinec *et al.*, 2021], authors propose to prune similar neurons based on their weight representations. To preserve the predictive function they update the consecutive layers and to do so they perform BN folding. Other pruning mechanisms also leverage BN folding such as [Srinivas and Babu, 2015; Liu and others, 2017; Ye and others, 2018]. [Nagel *et al.*, 2019] aim at balancing the weights, for quantization, across the entire network post BN folding and before performing quantization. Similarly, in [Zhao and others, 2019], the method folds the BN layers before splitting outlier neurons in order to compactify the distribution of the weight values thus reducing the quantization error.

Our novel algorithm allows to fold more BN layers than existing baselines, significantly reducing the inference time of deep neural networks. In addition, BAN-OFF allows to process more layers using existing pruning or quantization methods.

3 Methodology

Let f be a feed forward neural network with L layers $(f_l)_{l\in\{1,\dots,L\}}$. In this work, we define any layer as one of the following operations:

1. expressive layers: a layer f_l is expressive if and only if it can be expressed as the mathematical operation $f_l : x \mapsto W_l x + b_l$.
2. non-affine layers: a layer f_l is non-affine if and only if it corresponds to a mathematical operation that doesn't satisfy $f(ax+b) = \tilde{a}f(x) + \tilde{b}$.
3. BN layers [Ioffe and Szegedy, 2015]: a layer f_l of BN is defined, in the general case, by four parameters γ_l, β_l, μ_l and σ_l such that $f : x \mapsto \gamma_l \frac{x-\mu_l}{\sigma_l+\epsilon} + \beta_l$ with ϵ a small constant positive scalar (usually $\epsilon = 10^{-3}$).
4. other layers.

For instance, convolutional and fully-connected layers without their activation functions are the most common examples of expressive layers. Non-affine layers correspond and are limited to the activation functions in most deep neural network architectures. Finally, pooling layers, concatenations, additions and flattening are examples of other layers commonly found in said architectures.

3.1 Batch-Normalization Folding Limits

Simplest Case. let's assume f is sequential with the following patterns: if f_l is an expressive layer then f_{l+1} is a BN layer and f_{l+2} is a non-affine layer, *i.e.* $f_{l+2}(f_{l+1}(f_l(x))) = \sigma\left(\text{BN}(W_l x + b_l)\right)$ where σ is an activation function and BN is a BN operation, see Figure 2 a). Under those assumptions all the BN layers can be folded using the naive BN-folding algorithm as follows: for each BN layer $l+1$ we remove the layer in the graph associated to the network and update the

weights of f_l with

$$\begin{cases} W_l \leftarrow \gamma_{l+1} \frac{W_l}{\sigma_{l+1}+\epsilon} \\ b_l \leftarrow \gamma_{l+1} \frac{b_l - \mu_{l+1}}{\sigma_{l+1}+\epsilon} + \beta_{l+1} \end{cases} \quad (1)$$

Then the resulting neural network is mathematically identical to the original network while requiring less operations to be computed.

Sequential Model. Let's assume f is sequential in the most general sense, *i.e.* $f : x \mapsto f_L(\ldots f_1(x))$, see Figure 2 b). The difference with the previous case is the possible presence of other layers between the BN layers and the expressive ones as well as the the possibility to have the activation before the BN. The naive BN-folding algorithm consists in folding the BN layers in the previous or next expressive layer as long as their are no intermediate non-affine layers. To fold a BN layer backward (*i.e.* the expressive layer to update has a lower index) we use equation 1. On the other hand, to fold a BN layer forward (*i.e.* the expressive layer to update has a higher index $l + k$) we use

$$\begin{cases} W_{l+k} \leftarrow \gamma_{l+1} \frac{W_{l+k}}{\sigma_{l+1}+\epsilon} \\ b_{l+k} \leftarrow -\gamma_{l+1} \frac{\mu_{l+1}}{\sigma_{l+1}+\epsilon} + \beta_{l+1} W_{l+k} + b_{l+k} \end{cases} \quad (2)$$

Under the sequential assumption, the algorithm folds all the BN layers that can be folded. Therefore this BN-folding algorithm is optimal in this specific case.

General DAG. Let's assume the graph of f is a directed acyclic graph (DAG), see Figure 2 c). Compared to a sequential model this corresponds to adding skip connections and possibly multiple leaves to the graph. In this case, the naive BN-folding algorithm consists in only considering sequential portions of the graph and limiting the resolution to the previous case. However a number of BN layers may be involved in non-sequential patterns as illustrated in Figure 2 c). Such BN layers can be folded by the proposed method. As illustrated, this is done by folding the BN layer in the input expressive layers while updating adequately the the consecutive expressive layer (the one below the BN node in Figure 2 c). For this reason the naive BN-folding algorithm is not optimal. Now, we will define a necessary and sufficient condition for BN folding.

3.2 Theoretical Folding Condition

Every feed forward neural network architecture can be expressed as a DAG. Let f be a DAG with n nodes. We recall that, by definition, a layer f_l is affine, if and only if $f_l(ax + b) = \tilde{a} f_l(x) + \tilde{b}$. The intuition is to isolate BN layers from non-affine layers. To do so, we define the connected components C of a node N (corresponding to a BN layer) recursively. We start with the set $C = \{N\}$ and we add all the adjacent, affine nodes (*i.e.* nodes not associated with non-affine layers). For each new, not-expressive, element of C we repeat the process until there are no new elements. The graph defined by C is decomposed in the layers entering node N and the layers leaving node N. We note the first part C_{in} and the second C_{out}. For a simple example (Figure 2 c), if we consider the VGG BN case and a BN layer f_l, we have

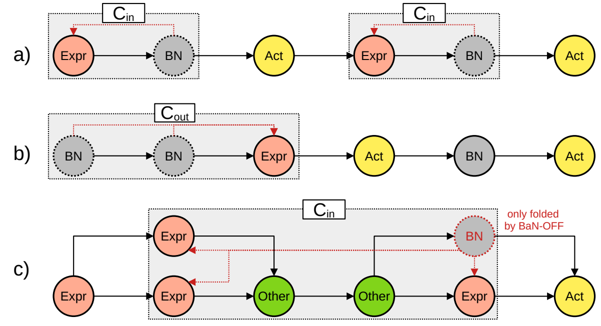

Figure 2: BN Folding of different neural networks archetypes. First a) a VGG with BN layers. Second an example of a general sequential model. Finally c) a network (DAG) with skip connections. Red nodes correspond to expressive layers, Yellow nodes to activation functions, grey nodes to BN layers and green nodes to other layers. The dark connections correspond to the graph associated to the network and dashed, red connections to the BN folding.

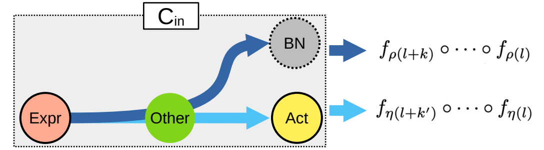

Figure 3: Two paths in the same C_{in} sub-graph with different leaves $f_{\rho(l+k)}$ and $f_{\eta(l+k')}$. In such instance, the activation layer $f_{\rho(l+k)}$ receives the same processed features as the BN layer $f_{\eta(l+k')}$. Thus the folding can't be performed without altering the predictive function.

$C_{\text{in}} = \{f_{l-1}, f_l\}$ with the input expressive layer f_{l-1} and $C_{\text{out}} = \{f_l, f_{l+1}\}$ with f_{l+1} the activation layer. More examples of such connected components are illustrated in Figure 2. Then, for a given BN layer, the following theorem gives us our necessary and sufficient condition on the possibility to fold the layer.

Theorem 1. *Let f be a DAG associated to a neural network. A BN layer of node N can be folded if and only if at least one of C_{in} or C_{out} satisfies:*

1. *the set is not limited to $\{N\}$*
2. *all the leaves (aside N) of the set are expressive layers*

Proof. Here we will prove the necessity of the condition by assuming that either one of the condition is not satisfied by both C_{in} and C_{out}. First, let's assume C_{in} and C_{out} are restricted to $\{N\}$, then it implies that the BN layer is "surrounded" by non-affine layers and thus can't be folded (second grey node of Figure 2 b). Second, let's assume that both C_{in} and C_{out} have at least one leaf that is not associated to an expressive layer aside from N itself. Under this assumption, in C, there exists a path $f_{\rho(l+k)} \circ \cdots \circ f_{\rho(l)}$ of length k and a path $f_{\eta(l+k')} \circ \cdots \circ f_{\eta(l)}$ of length k' (see Figure 3) such that

1. $f_{\eta(l+k')}$ corresponds to node N and $f_{\eta(l+k')} \neq f_{\rho(l+k)}$
2. $f_{\eta(l)} = f_{\rho(l)}$

By definition of C all the consecutive layers of $f_{\rho(l+k)}$ are non-affine. Then if we fold N, the layer $f_{\eta(l)}$ won't compute

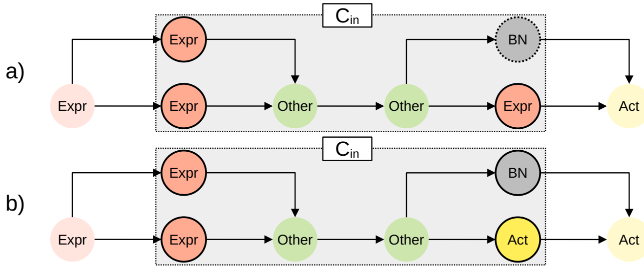

Figure 4: Critical examples of BN that can be folded a) and can't be folded b). The leaves of the sub-graphs C_{in} are highlighted to show when the second condition of theorem 1 is satisfied.

the same operations and by definition, the path $f_{\rho(l)} \circ \cdots \circ f_{\rho(l+k)}$ can't be updated to negate this effect. thus the node N can't be folded. Such scenario is illustrated in Figure 4 b). This demonstrates the necessity of the proposed condition. □

Following this result, we can see in Figure 2 c) a BN layer that can be folded but not with the naive BN-folding algorithm. To conclude our proof we will demonstrate that the condition is sufficient.

3.3 Proposed Practical Algorithm

Instead of an analytic proof we propose a constructive proof *via* an algorithm which performs the BN folding as long as the condition from theorem 1 is satisfied. Let's assume C_{in} satisfies the condition, then we separate the leaves in two categories: the ones with a path to node N, noted I and the others O. The nodes in I are updated using equation 1 and the others negate the previous update with the following update:

$$\begin{cases} W \leftarrow \frac{W}{\gamma}(\sigma + \epsilon) \\ b \leftarrow b + \frac{W\mu}{\sigma+\epsilon}\gamma - \beta W \end{cases} \tag{3}$$

If C_{in} doesn't satisfy the second condition from theorem 1, we use equation 2 to update I and the following for O:

$$\begin{cases} W \leftarrow \frac{W}{\gamma}(\sigma + \epsilon) \\ b \leftarrow \frac{b}{\gamma}(\sigma + \epsilon) + \frac{\mu}{\sigma+\epsilon}\gamma - \beta \end{cases} \tag{4}$$

Now we simply need to verify that any path in the graph associated to f computes the same outputs before and after folding. As we only update C, it is enough to do the verification for any path in C. We limit the verification to the case study of Figure 4 a). Before BN folding, the graph corresponds to the mathematical function f. We note W_i and b_i the weights and biases of each expressive node of the graph. We note σ the operation performed by the non-affine node and g_1 and g_2 the operations from the other nodes (green nodes). As such, we have

$$\begin{cases} G_1 : x \mapsto g_1(W_2(W_1 x + b_1) + b_2, W_3(W_1 x + b_1) + b_3) \\ G_2 : x \mapsto g_2(G_1(x)) \\ f : x \mapsto \sigma(BN(G_2(x))), \sigma(W_4 G_2(x) + b_4) \end{cases} \tag{5}$$

where G_1 and G_2 correspond to the transformation of the inputs by the first green node and second green node, respectively. In this example, the BN layer satisfies the conditions of theorem 1. We fold the BN layer in the layers of set I, *i.e.* the second and third expressive layers, using equation 1. We note their weights and biases $\tilde{W}_i$ and $\tilde{b}_i$. Finally, we adapt the layer of O, *i.e.* the fourth expressive layer, to its new processed inputs using equation 3 and note $\bar{W}_4$ and $\bar{b}_4$. The folded network will compute:

$$\begin{cases} \tilde{G}_1 : x \mapsto g_1(\tilde{W}_2(W_1 x + b_1) + \tilde{b}_2, \tilde{W}_3(W_1 x + b_1) + \tilde{b}_3) \\ \tilde{G}_2 : x \mapsto g_2(\tilde{G}_1(x)) = BN(G_2(x)) \\ \tilde{f} : x \mapsto \sigma(\tilde{G}_2(x)), \sigma(\bar{W}_4 \tilde{G}_2(x) + \bar{b}_4) \\ \qquad = \sigma(BN(G_2(x))), \sigma(W_4 BN^{-1}(BN(G_2(x))) + b_4) \\ \qquad = f(x) \end{cases} \tag{6}$$

where the folded operator $\tilde{G}_2$ is equal to $BN(G_2)$ before folding. Because we negated the folding in W_4 and b_4 the predictive function remains unchanged.

Consequently, the algorithm based on the proposed necessary and sufficient condition from theorem 1 is optimal, *i.e.* a BN layer can be folded if and only if

1. the set is not limited to $\{N\}$
2. all the leaves (aside N) of the set are expressive layers

We now validate the empirical, cost-free, benefits resulting from BaN-OFF.

4 Experiments

4.1 Toy

We implemented the toy architectures described in Figure 2. Table 1 lists our results on an Intel CPU m3. The inference speed-up is measured as the following ratio $\frac{T_{\text{old}} - T_{\text{new}}}{T_{\text{old}}} \in [0; 100\%]$ where T_{new} is the inference time of the network after BN folding and T_{old} is the original inference time, *i.e.* the higher the better.

As expected the two algorithms, naive and BaN-OFF, share the same performance on the two first architectures as they perform the exact same transformations of the network. On the other hand, for the last configuration only the proposed method performed BN folding and resulted in inference speed increase. We also observe that the reduction of inference time is not correlated with the proportion of removed parameters as BN layers don't require many parameters but may put sequential computations of the architecture on hold during the forward pass. Furthermore, the predictions were only changed by up to 10^{-6} in L^1 norm on vector outputs of 1000 elements. This is due to numerical approximations. Overall, this shows the importance of BN folding in neural networks acceleration as it doesn't change the predictive function and provide a significant boost in terms of speed.

4.2 Main Results

Cifar10. Table 2 summarizes our results on ResNets [He *et al.*, 2016] trained on Cifar10 [Krizhevsky *et al.*, 2009]. For both algorithms, the folded networks share identical predictions with the original models. We conducted our first experi-

Model	Naive	BaN-OFF	% removed parameters
Figure 2 a)	**63.42**	**63.42**	0.488
Figure 2 b)	**51.44**	**51.44**	0.488
Figure 2 c)	0	**47.81**	0.314

Table 1: Comparison between the naive and proposed BN-folding algorithm in terms of increased inference speed on an Intel CPU m3. We also report the proportion of removed parameters.

Model	Naive	BaN-OFF	% removed parameters
ResNet 20	30.64	**35.04**	0.61
ResNet 56	43.19	**50.97**	0.63
ResNet 110	49.21	**54.36**	0.63
ResNet 164	34.35	**39.20**	0.63

Table 2: Comparison between the naive and proposed BN-folding algorithm, on ResNets trained on Cifar10, in terms of percentage of relative reduced inference time on an Intel CPU m3. We also report the percentage of removed parameters.

Model	Naive	BaN-OFF	% removed params
MobileNet	24.35	**29.01**	0.97
ResNet 50	**6.48**	**6.48**	0.61
ResNet 101	**10.34**	**10.34**	0.63
ResNet 152	**21.31**	**21.31**	0.63
EfficientNet B0	17.10	**18.27**	0.79
EfficientNet B1	33.88	**35.95**	0.80
EfficientNet B5	19.93	**21.38**	0.57

Table 3: Comparison between the naive and proposed BN-folding algorithm, on models trained on ImageNet, in terms of percentage of relative reduced inference time on an Intel CPU m3. We also report the percentage of removed parameters.

ments on a small CPU as it corresponds to the most challenging use-case. On a small device, sequential operations, e.g. BN layers, are not bottleneck compared to large expressive layers. Furthermore, using a small device induces instability on the results for both algorithms. Standard deviation values (over the percentage of relative reduced inference time per frame) range from 3.97 to 13.65. However the proposed method systematically outperforms naive BN-folding.

Furthermore, we confirm the previous observation on the decorrelation between the proportion of removed parameters and the improvement in inference speed. We also observe that the inference boost can't be naively deduced from simple architecture properties such as depth and wideness.

ImageNet. Table 3 contains our results on ResNets [He *et al.*, 2016], MobileNet v2 [Sandler *et al.*, 2018] and EfficientNets [Tan and Le, 2019] trained on ImageNet [Deng *et al.*, 2009]. Folded networks, with the naive and the proposed method, the resulting networks share identical predictions with the original model. Similarly to Cifar10, the results were obtained on an Intel m3 CPU. Standard deviations values range between 1.14 for the least accelerated models and 12.68 for the most accelerated models. As expected, the proposed method systematically outperforms the naive approach.

Model	CPU m3	CPU i9-9900K	RTX 3090
ResNet 20	35.04	38.85	22.03
ResNet 56	50.97	49.37	47.33
ResNet 110	54.36	51.20	40.86
ResNet 164	39.20	48.16	56.94
MobileNet	29.01	27.76	69.14
ResNet 50	6.48	18.54	72.83
ResNet 101	10.34	16.77	47.89
ResNet 152	21.31	17.82	40.97
EfficientNet B0	18.27	20.74	52.72
EfficientNet B1	35.95	20.51	43.92
EfficientNet B5	21.38	11.80	49.60

Table 4: Comparison of relative runtime reduction between CPU and GPU devices, on different hardware and different architectures trained on Cifar10 and ImageNet.

In terms of proportion of removed parameters, the results are comparable to our results on Cifar10 which demonstrates that, even for more challenging tasks it matters more to remove a few meaningful parameters (e.g. BN layers) than a lot of parameters. This is based on the results obtained with pruning methods such as [Frankle and Carbin, 2018; Lin and others, 2020]. We also note that the correlation between depth and acceleration due to BN-folding on ResNets may be a coincidence.

4.3 Stability over Hardware

Central Processing Unit (CPU) comparison. Our results, in Table 4, show the stability of the inference boost across different CPUs. We also note that the maximum standard deviation on the results drops from 13.65 to 2.06 on the larger CPU. These results show the difficulties to work on very small devices and the importance of efficient acceleration.

Graphical Processing Unit (GPU) comparison. Table 4 presents the comparison between CPU and GPU devices response to BN folding. In contrast with CPU devices, GPU are very sensitive to sequential computations and very efficient on parallel operations. This property translates as significantly larger inference speed-up on GPU when compared to CPU. As expected from a larger device, the standard deviations on the results represent less than 5.04% of the average value which shows the stability of the inference speed-up across different runs.

Furthermore, such results motivate the use of acceleration frameworks comprising BN folding for efficient inference on cloud servers using large GPUs.

4.4 Other Acceleration Methods

BN usually constitutes a complementary step in pruning and quantization. However it is not systematically applied. We put the emphasis on the efficiency of BN-folding for DNN acceleration by comparing the cost in terms of accuracy to reach similar acceleration with pruning and quantization methods.

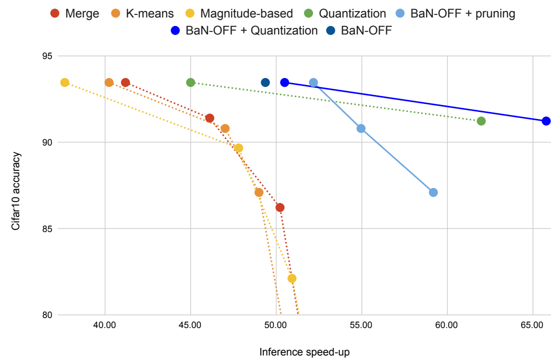

Figure 5: Comparison of different acceleration method on Cifar10 ResNet 56. Pruning and Quantization alone come at a cost in accuracy and need complex methodologies. Meanwhile, BaN-OFF comes at no cost and offer great improvements in runtime at no cost.

Model	Naive	BaN-OFF	improvement
Faster rcnn (mob)	51.78	**52.40**	+0.63
Mask rcnn (res)	**55.37**	**55.37**	+0.00
EfficientDet D0	51.72	**53.93**	+2.21
EfficientDet D7	44.36	**45.73**	+1.37
CenterNet	52.86	**53.55**	+0.69

Table 5: Object detection results on MS COCO. We report the percentage of relative reduced inference time of both methods for Faster RCNN (MobileNet backbone), Mask RCNN (ResNet backbone), EfficientDet D0, D7 and CenterNet with a stacked hourglass.

Pruning. Figure 5 shows the comparison of the influence of pruning (merge step from [Yvinec *et al.*, 2021]) alone and BN folding on a ResNet 56 trained on Cifar10. For the sake of comparison, we implemented three classical pruning methods: Merge (which consists in merging similar neurons [Yvinec *et al.*, 2021]), K-means (which consists in clustering neurons by their weight values) and magnitude-based unstructured pruning (which consists in removing weight values based on their absolute value). These methods allows inference speed-ups up to 50.23% but often significantly degrade the model accuracy. In comparison, the proposed BAN-OFF method allows similar runtime reduction without altering the predictive function. This suggests that, while the community focused on removing a lot of parameters, removing a few parameters with BN folding saves a lot of computations. Furthermore, BN folding and such pruning techniques can be combined for improved runtime reduction (up to 59.2%).

Quantization. We considered the naive quantization operator introduced in [Krishnamoorthi, 2018]. Such methods are usually more effective than pruning in terms of inference speed-up, which is confirmed in Figure 5. Similarly to the previous results, BN folding and quantization can be used in conjunction for improved runtime reduction ($50.5 - 65.8\%$).

Model	run-T	BaN-OFF	improvement
U-Net (Res34)	57.89	**59.95**	+2.06
U-Net (effB0)	59.77	**62.86**	+3.09
FPN (Res34)	52.13	**54.80**	+2.67
PSPN (Res34)	53.01	**55.14**	+2.14

Table 6: Results of the proposed method on object detection methods trained on CityScapes. We report the percentage of relative reduced inference time of the two methods for U-Net with ResNet 34 and Efficient B0, FPN and PSPN with ResNet 34 backbone.

4.5 Other Data Applications

MS COCO. Table 5 summarizes our results on the detection task of MS COCO [Lin and others, 2014] on an RTX 3090. We considered widely used architectures such as Faster RCNN [Ren and others, 2015], Mask RCNN [He *et al.*, 2017], EfficientDet [Tan *et al.*, 2020] and CenterNet [Zhou and others, 2019]. Similarly to ImageNet, the folded models and the original models share the same predictions and the same accuracy. We see results very comparable to the results from table 4 (column RTX 3090) which is understandable as the detection algorithm are very similar, in the sense of the presence of BN layers, to their corresponding backbones.

CityScapes. From our results, listed in Table 6, on the semantic segmentation task of CityScapes [Cordts and others, 2016], we observe similar results to object detection. On both FPN [Lin and others, 2017] and PSPN [Zhao and others, 2017] models, the proposed method achieves remarkable results with 55% reduction of the inference time on an RTX 3090. Similar results are obtained on U-Net [Ronneberger *et al.*, 2015] with an EfficientNet B0 backbone with great stability across runs (standard deviation below 2 points). Finally, on U-Net, we reach our most remarkable result of almost 63% reduction of latency with a standard deviation below 1 point.

5 Conclusion

In this work, we highlighted and addressed the limitations of existing batch normalization (BN) layer folding approaches. We argued that BN folding is often overlooked in the deep neural network acceleration literature, as it allows drastic runtime reduction at inference time, especially compared to other acceleration methods, e.g. pruning or quantization.

From a theoretical point of view, we proposed a novel necessary and sufficient condition for BN folding working for any architecture. We also derived a practical algorithm, dubbed BAN-OFF, for optimal BN folding. BAN-OFF allows to significantly reduce the inference time of any deep neural network, regardless of its task (classification, object detection or semantic segmentation), architecture or the hardware of which it shall run on, at virtually no cost. The source code is available in gitlab and as a pip package (pip install tensorflow-batchnorm-folding).

Acknowledgments

This work has been supported by the french National Association for Research and Technology (ANRT) and the company Datakalab (CIFRE convention C20/1396).

References

[Chen and others, 2017] Liang-Chieh Chen et al. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *TPAMI*, 2017.

[Cordts and others, 2016] Marius Cordts et al. The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016.

[Deng *et al.*, 2009] J. Deng, W. Dong, et al. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[Finkelstein *et al.*, 2019] Alexander Finkelstein, Uri Almog, and Mark Grobman. Fighting quantization bias with bias. *ECV workshop CVPR*, 2019.

[Frankle and Carbin, 2018] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *ICLR*, 2018.

[He *et al.*, 2016] Kaiming He, Xiangyu Zhang, et al. Deep residual learning for image recognition. In *CVPR*, 2016.

[He *et al.*, 2017] Kaiming He, Georgia Gkioxari, et al. Mask r-cnn. In *ICCV*, 2017.

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*, 2015.

[Jacob and others, 2018] Benoit Jacob et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2018.

[Krishnamoorthi, 2018] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv*, 2018.

[Krizhevsky *et al.*, 2009] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[Krizhevsky *et al.*, 2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *NeurIPS*, 25, 2012.

[LeCun and others, 1989] Yann LeCun et al. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4), 1989.

[Lin and others, 2014] Tsung-Yi Lin et al. Microsoft coco: Common objects in context. In *ECCV*. Springer, 2014.

[Lin and others, 2017] Tsung-Yi Lin et al. Feature pyramid networks for object detection. In *CVPR*, 2017.

[Lin and others, 2020] Mingbao Lin et al. Hrank: Filter pruning using high-rank feature map. In *CVPR*, 2020.

[Liu and others, 2017] Zhuang Liu et al. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017.

[Luo and others, 2019] Ping Luo et al. Switchable normalization for learning-to-normalize deep representation. *TPAMI*, 2019.

[Martin and Mahoney, 2019] Charles H Martin and Michael W Mahoney. Traditional and heavy-tailed self regularization in neural network models. *ICML*, 2019.

[Meller *et al.*, 2019] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different: Recovering neural network quantization error through weight factorization. In *ICML*, 2019.

[Nagel *et al.*, 2019] Markus Nagel, Mart van Baalen, et al. Data-free quantization through weight equalization and bias correction. In *ICCV*, 2019.

[Ren and others, 2015] Shaoqing Ren et al. Faster r-cnn: Towards real-time object detection with region proposal networks. *NeurIPS*, 28, 2015.

[Renda *et al.*, 2020] Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing rewinding and fine-tuning in neural network pruning. *ICLR*, 2020.

[Ronneberger *et al.*, 2015] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*. Springer, 2015.

[Sandler *et al.*, 2018] Mark Sandler, Andrew Howard, et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[Simonyan and Zisserman, 2014] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *BMVC*, 2014.

[Srinivas and Babu, 2015] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *BMVC*, 2015.

[Szegedy and others, 2016] Christian Szegedy et al. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[Tan and Le, 2019] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *ICML*, 2019.

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

[Wu and He, 2018] Yuxin Wu and Kaiming He. Group normalization. In *ECCV*, 2018.

[Ye and others, 2018] Jianbo Ye et al. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. *ICLR*, 2018.

[Yvinec *et al.*, 2021] Edouard Yvinec, Arnaud Dapogny, Matthieu Cord, and Kevin Bailly. Red: Looking for redundancies for data-free structured compression of deep neural networks. *NeurIPS*, 2021.

[Zhao and others, 2017] Hengshuang Zhao et al. Pyramid scene parsing network. In *CVPR*, 2017.

[Zhao and others, 2019] Ritchie Zhao et al. Improving neural network quantization without retraining using outlier channel splitting. In *ICML*, 2019.

[Zhou and others, 2019] Xingyi Zhou et al. Objects as points. *ICCV*, 2019.