In-Place Activated BatchNorm for Memory-Optimized Training of DNNs

Samuel Rota Bulò, Lorenzo Porzi, Peter Kontschieder
Mapillary Research
research@mapillary.com

Abstract

*In this work we present In-Place Activated Batch Normalization (*InPlace-ABN*) – a novel approach to drastically reduce the training memory footprint of modern deep neural networks in a computationally efficient way. Our solution substitutes the conventionally used succession of BatchNorm + Activation layers with a single plugin layer, hence avoiding invasive framework surgery while providing straightforward applicability for existing deep learning frameworks. We obtain memory savings of up to 50% by dropping intermediate results and by recovering required information during the backward pass through the inversion of stored forward results, with only minor increase (*0.8-2%*) in computation time. Also, we demonstrate how frequently used checkpointing approaches can be made computationally as efficient as* InPlace-ABN. *In our experiments on image classification, we demonstrate on-par results on ImageNet-1k with state-of-the-art approaches. On the memory-demanding task of semantic segmentation, we report competitive results for COCO-Stuff and set new state-of-the-art results for Cityscapes and Mapillary Vistas. Code can be found at* `https://github.com/mapillary/inplace_abn`.

1. Introduction

High-performance computer vision recognition models typically take advantage of deep network backbones, generating rich feature representations for target applications to operate on. For example, top-ranked architectures used in the 2017 LSUN or MS COCO segmentation/detection challenges are predominantly based on ResNet/ResNeXt [9, 30] models comprising >100 layers.

Obviously, depth/width of networks strongly correlate with GPU memory requirements and at given hardware memory limitations, trade-offs have to be made to balance feature extractor performance vs. application-specific parameters like network output resolution or training data size. A particularly memory-demanding task is semantic segmentation, where one has to compromise significantly on the number of training crops per minibatch and their spatial resolution. In fact, many recent works based on modern backbone networks have to set the training batch size to no more than *a single* crop per GPU [2, 28], which is partially also due to suboptimal memory management in some deep learning frameworks. In this work, we focus on increasing the memory efficiency of the training process of modern network architectures in order to further leverage performance of deep neural networks in tasks like image classification and semantic segmentation.

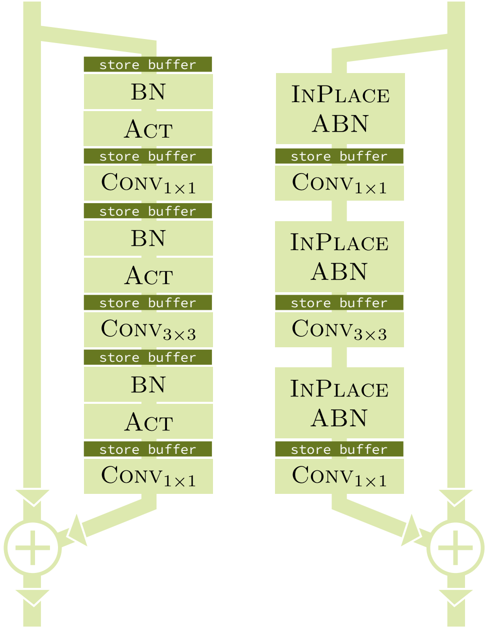


Figure 1. Example of residual block with identity mapping [10]. Left: Implementation with standard BN and in-place activation layers, which requires storing 6 buffers for the backward pass. Right: Implementation with our proposed InPlace-ABN layer, which requires storing only 3 buffers. Our solution avoids storing the buffers that are typically kept for the backward pass through BN and exhibits a lower computational overhead compared to state-of-the-art memory-reduction methods.

We introduce a novel and unified layer that replaces the commonly used succession of batch normalization (BN) and nonlinear activation layers (Act), which are integral with modern deep learning architectures like ResNet [9], ResNeXt [30], Inception-ResNet [26], WideResNet [32], Squeeze-and-Excitation Networks [11], DenseNet [12], *etc*. Our solution is coined InPlace-ABN and proposes to merge batch normalization and activation layers in order to enable in-place computation, using only a single memory buffer for storing the results (see illustration in Figure 1). During the backward pass, we can efficiently recover all required quantities from this buffer by inverting the forward pass computations. Our approach yields a theoretical memory reduction of up to 50%, and our experiments on semantic segmentation show additional data throughput of up to $+75\%$ during training, when compared to prevailing sequential execution of BN+Act. Our memory gains are obtained without introducing noticeable computational overhead, *i.e*. side-by-side runtime comparisons show only between +0.8-2% increase in computation time.

As additional contribution, we review the *checkpointing* memory management strategy [4] and propose a computationally optimized application of this idea in the context of BN layers. This optimization allows us to drop recomputation of certain quantities needed during the backward pass, eventually leading to reduced computation times as per our InPlace-ABN. However, independent of the proposed optimized application of [4], conventional checkpointing in general suffers from higher implementation complexity (with the necessity to invasively manipulate the computation graph), while our main InPlace-ABN contribution can be easily implemented as self-contained, standard *plug-in layer* and therefore simply integrated in any modern deep learning framework.

Our experimental evaluations demonstrate on-par performance with state-of-the-art models trained for image classification on ImageNet [25] (in directly comparable memory settings), and significantly improved results for the memory-critical application of semantic segmentation.

To summarize, we provide the following contributions:

- Introduction of a novel, self-contained InPlace-ABN layer that enables joint, in-place computation of BN+Act, approximately halvening the memory requirements during training of modern deep learning models.
- A computationally more efficient application of the *checkpointing* memory management strategy in the context of BN layers, inspired by optimizations used for InPlace-ABN.
- Experimental evaluations for i) image classification on ImageNet-1k showing approximately on-par performance with state-of-the-art models and ii) semantic segmentation on COCO-Stuff, Cityscapes and Mapillary Vistas, considerably benefiting from the additional available memory and generating new high-scores on the challenging Cityscapes and Mapillary Vistas datasets.

2. Related Work

The topic of optimizing memory management in deep learning frameworks is typically addressed at different levels. Efficient deep learning frameworks like TensorFlow, MxNet or PyTorch follow distinct memory allocation strategies. Among them is *checkpointing* [4, 19], which provides additional memory at the cost of runtime via storing activation buffers as so-called *checkpoints*, from where required quantities can be re-computed during the backward pass. The paper in [4] describes how to recursively apply such a variant on sub-graphs between checkpoints. In [8] this is further optimized with dynamic programming, where a storage policy is determined that minimizes the computational costs for re-computation at a fixed memory budget.

Virtually all deep learning frameworks based on NVIDIA hardware exploit low-level functionality libraries CUDA and cuDNN[1], providing GPU-accelerated and performance-optimized primitives and basic functionalities. Another line of research has focused on training CNNs with reduced precision and therefore smaller memory-footprint datatypes. Such works include (partially) binarized weights/activations/gradients [6, 13, 14], which however typically lead to degraded overall performance. With mixed precision training [20], this issue seems to be overcome and we plan to exploit this as complementary technique in future work, freeing up even more memory for training deep networks without sacrificing runtime.

In [7] the authors modify ResNet in a way to contain reversible residual blocks, *i.e*. residual blocks whose activations can be reconstructed backwards. Backpropagation through reversible blocks can be performed without having stored intermediate activations during the forward pass, which allows to save memory. However, the cost to pay is twofold. First, one has to recompute each residual function during the backward pass, thus having the same overhead as checkpointing [19]. Second, the network design is limited to using blocks with certain restrictions, *i.e*. reversible blocks cannot be generated for bottlenecks where information is supposed to be discarded.

Finally, we stress that only training time memory-efficiency is targeted here while test-time optimization as done *e.g*. in NVIDIAs TensorRT [2] is beyond our scope.

3. In-Place Activated Batch Normalization

Here, we describe our contribution to avoid the storage of a buffer that is typically needed for the gradient computation during the backward pass through the batch normalization layer. As opposed to existing approaches we also show that our solution minimizes the computational overhead we have to trade for saving additional memory.

[1] https://developer.nvidia.com

[2] https://developer.nvidia.com/tensorrt

3.1. Batch Normalization Review

Batch Normalization has been introduced in [15] as an effective tool to reduce internal covariate shift in deep networks and accelerate the training process. Ever since, BN plays a key role in most modern deep learning architectures.

The key idea consists in having a normalization layer that applies an axis-aligned whitening of the input distribution, followed by a scale-and-shift operation aiming at preserving the network's representation capacity. The whitening operation exploits statistics computed on a minibatch level only. The by-product of this approximation is an additional regularizing effect for the training process.

In details, we can fix a particular unit x in the network and let $x_\mathcal{B} = \{x_1, \dots, x_m\}$ be the set of values x takes from a minibatch $\mathcal{B}$ of m training examples. The batch normalization operation applied to x_i first performs a whitening of the activation using statistics computed from the minibatch:

$$\hat{x}_i = \mathrm{BN}(x_i) = \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}\,. \tag{1}$$

Here $\epsilon > 0$ is a small constant that is introduced to prevent numerical issues, and $\mu_\mathcal{B}$ and $\sigma^2_\mathcal{B}$ are the empirical mean and variance of the activation unit x, respectively, computed with respect to the minibatch $\mathcal{B}$, *i.e.*

$$\mu_\mathcal{B} = \frac{1}{m}\sum_{j=1}^{m} x_j\,, \qquad \sigma^2_\mathcal{B} = \frac{1}{m}\sum_{j=1}^{m}(x_j - \mu_\mathcal{B})^2\,.$$

The whitened activations $\hat{x}_i$ are then scaled and shifted by learnable parameters γ and β, obtaining

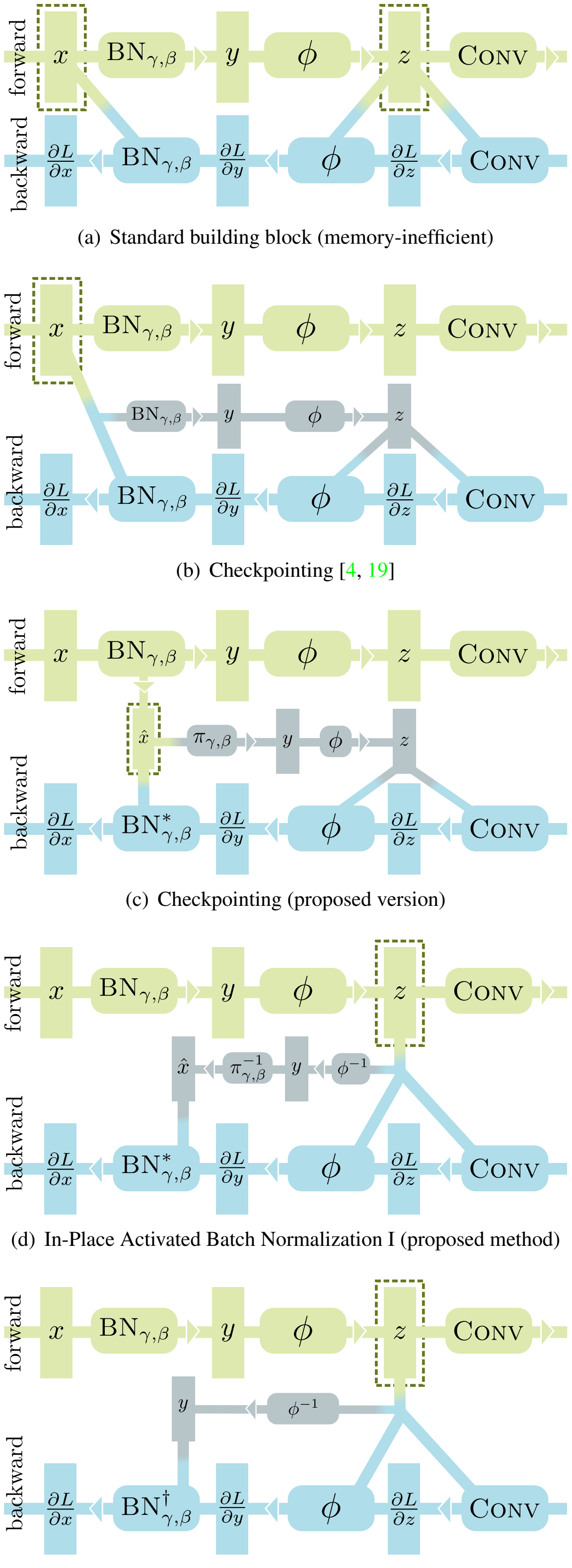
$$y_i = \mathrm{BN}_{\gamma,\beta}(x_i) = \gamma\hat{x}_i + \beta\,.$$

The BN transformation described above can in principle be applied to any activation in the network and is typically adopted with channel-specific (γ, β)-parameters. Using BN renders training resilient to the scale of parameters, thus enabling the use of higher learning rates.

At test time, the BN statistics are fixed to $\mu_\mathcal{T}$ and $\sigma_\mathcal{T}$, estimated from the entire training set $\mathcal{T}$. These statistics are typically updated at training time with a running mean over the respective minibatch statistics, but could also be recomputed before starting the testing phase. Also, the computation of networks trained with batch normalization can be sped up by absorbing BN parameters into the preceding Conv layer, by performing a simple update of the convolution weights and biases. This is possible because at test-time BN becomes a linear operation.

3.2. Memory Optimization Strategies

Here we sketch our proposed memory optimization strategies after introducing both, the standard (memory-inefficient) use of batch normalization and the state-of-the-art coined *checkpointing* [4, 19].


forward
backward
x
$\mathrm{BN}_{\gamma,\beta}$
y
ϕ
z
Conv
$\frac{\partial L}{\partial x}$
$\frac{\partial L}{\partial y}$
$\frac{\partial L}{\partial z}$
$\hat{x}$
$\pi_{\gamma,\beta}$
$\mathrm{BN}^*_{\gamma,\beta}$
$\pi^{-1}_{\gamma,\beta}$
ϕ^{-1}
$\mathrm{BN}^\dagger_{\gamma,\beta}$

(a) Standard building block (memory-inefficient)

(b) Checkpointing [4, 19]

(c) Checkpointing (proposed version)

(d) In-Place Activated Batch Normalization I (proposed method)

(e) In-Place Activated Batch Normalization II (proposed method)

Figure 2. Comparison of standard BN, state-of-the-art checkpointing from [4, 19] and our proposed methods. See § 3.2 for a detailed description.

In Figure 2, we provide diagrams showing the forward and backward passes of a typical building block BN+ACT+CONV[3] as usually implemented in modern deep architectures. The activation function (*e.g.* RELU) is denoted by ϕ. Computations occurring during the forward pass are shown in green and involve the entire minibatch $\mathcal{B}$ (we omit the subscript $\mathcal{B}$). Computations happening during the backward pass are shown in cyan and gray. The gray part aims at better highlighting the additional computation that has been introduced to compensate for the memory savings. Rectangles are in general volatile buffers holding intermediate results, except for rectangles surrounded by a dashed frame, which represent buffers that need to be stored for the backward pass and thus significantly impact the training memory footprint. *E.g.*, in Figure 2(a) x and z will be stored for the backward pass, while in Figure 2(b) only x is stored. For the sake of presentation clarity, we have omitted two additional buffers holding $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ for the BN backward phase. Nevertheless, these buffers represent in general a small fraction of the total allocated memory. Moreover, we have also omitted the gradients with respect to the model parameters (*i.e.* γ, β and CONV weights).

Standard. In Figure 2(a) we present the standard implementation of the reference building block, as used in all deep learning frameworks. During the forward pass both, the input x to BN and the output of the activation function ϕ need to be stored for the backward pass. Variable x is used during the backward pass through $\text{BN}_{\gamma,\beta}$ to compute both the gradient w.r.t. its input and γ, *i.e.* $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial \gamma}$ where L denotes the loss, while z is required for the backward pass through the activation ϕ as well as potential subsequent operations like *e.g.* the convolution shown in the figure.

Checkpointing [4, 19]. This technique allows to trade computation for memory when training neural networks, applicable in a very broad setting. In Figure 2(b), we limit its application to the building block under consideration like in [22]. In contrast to the standard implementation, which occupies two buffers for the backward pass of the shown building block, checkpointing requires only a single buffer. The trick consists in storing only x and recomputing z during the backward pass by repeating the forward operations starting from x (see gray-colored operations). Clearly, the computational overhead to be paid comprises both, recomputation of the BN and activation layers. It is worth observing that recomputing $\text{BN}_{\gamma,\beta}$ (gray) during the backward phase can reuse values for $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ available from the forward pass and fuse together the normalization and subsequent affine transformation into a single scale-and-shift operation. Accordingly, the cost of the second forward pass over $\text{BN}_{\gamma,\beta}$ becomes less expensive (see also [22]).

The three approaches that follow are all contributions of this work. The first represents a variation of checkpointing, which allows us to save additional computations in the context of BN. *The second and third are our main contributions, providing strategies that yield the same memory savings and even lower computational costs compared to the proposed, optimized checkpointing, but are both self-contained and thus much easier to integrate in existing deep learning frameworks.*

Checkpointing (proposed version). Direct application of the checkpointing technique in the sketched building block, which is adopted also in [22], is not computationally optimal since additional operations could be saved by storing $\hat{x}$, *i.e.* the normalized value of x as per Eq. (1), instead of x. Indeed, as we will see in the next subsection, the backward pass through BN requires recomputing $\hat{x}$ if not already stored. For this reason, we propose in Figure 2(c) an alternative implementation that is computationally *more efficient* by retaining $\hat{x}$ from the forward pass through the BN layer. From $\hat{x}$ we can recover z during the backward pass by applying the scale-and-shift operation $\pi_{\gamma,\beta}(\hat{x}) = \gamma\hat{x} + \beta$, followed by the activation function ϕ (see gray-colored operations). In this way, the computation of z becomes slightly more efficient than the one shown in Figure 2(b), for we save the fusion operation. Finally, an additional saving of the normalization step derives from using the stored $\hat{x}$ in the backward implementation of BN rather than recomputing it from x. To distinguish the efficient backward implementation of BN from the standard one we write $\text{BN}^*_{\gamma,\beta}$ in place of $\text{BN}_{\gamma,\beta}$ (cyan-colored, see additionally § 3.3).

In-Place Activated Batch Normalization I. A limitation of the memory-reduction strategy described above is that the last layer, namely CONV in the example, depends on non-local quantities like x (or $\hat{x}$) for the computation of the gradient. This makes the implementation of the approach within standard frameworks somewhat cumbersome, because the backward pass of any layer that follows ϕ, which relies on the existence of z, has to somehow trigger its recomputation. To render the implementation of the proposed memory savings *easier* and *self-contained*, we suggest an alternative strategy shown in Figure 2(d), which relies on having only z as the saved buffer during the forward pass, thus operating an in-place computation through the BN layer (therefrom the paper's title). By doing so, any layer that follows the activation ϕ would have the information for the gradient computation locally available. Having stored z, we need to recompute $\hat{x}$ *backwards*, for it will be needed in the backward pass through the BN layer.[4] However, this operation is only possible if the activation func-

[3] Having the convolution at the end of the block is not strictly necessary, but supports comprehension.

[4] This solution can technically still be considered as a form of checkpointing, but instead of recovering information forwards as in [4, 19], we recover it backwards, thus bearing a similarity to reversible nets [7].

tion is invertible. Even though this requirement does not hold for ReLU, *i.e.* one of the most dominantly used activation functions, we show in § 4.1 that an invertible function like Leaky ReLU [18] with a small slope works well as a surrogate of ReLU without compromising on the model quality. We also need to invert the scale-and-shift operation $\pi_{\gamma,\beta}$, which is in general possible if $\gamma \neq 0$.

In-Place Activated Batch Normalization II. The complexity of the computation of $\hat{x} = \pi^{-1}_{\gamma,\beta}(y) = \frac{y-\beta}{\gamma}$ used in the backward pass of InPlace-ABN I can be further reduced by rewriting the gradients $\frac{\partial L}{\partial \gamma}$ and $\frac{\partial L}{\partial x}$ directly as functions of y instead of $\hat{x}$. The explicit inversion of $\pi_{\gamma,\beta}$ to recover $\hat{x}$ applies m scale-and-shift operations (per feature channel). If the partial derivatives are however based on y directly, the resulting modified gradients (derivations given in [24]) show that the same computation can be absorbed into the gradient $\frac{\partial L}{\partial x_i}$ at $\mathcal{O}(1)$ cost (per feature channel). In Figure 2(e) we show the diagram of this optimization, where we denote as $\text{BN}^{\dagger}_{\gamma,\beta}$ the implementation of the backward pass as a function of y.

3.3. Technical Details

The key components of our method are the computation of the inverse of both the activation function (InPlace-ABN I & II) and $\pi_{\gamma,\beta}$ (InPlace-ABN I), and the implementation of a backward pass through the batch normalization layer that depends on y, *i.e.* the output of the forward pass through the same layer.

Invertible activation function. Many activation functions are actually invertible and can be computed in-place (*e.g.* sigmoid, hyperbolic tangent, Leaky ReLU, and others), but the probably most commonly used one, namely ReLU, is not invertible. However, we can replace it with Leaky ReLU and a small slope without impacting the quality of the trained models [31]. Leaky ReLU and its inverse share the same computational cost, *i.e.* an elementwise sign check and scaling operation. Hence, the overhead deriving from the recomputation of ϕ in the backward pass of the previously shown, checkpointing-based approaches and its inverse ϕ^{-1} employed in the backward pass of our method are equivalent. To give further evidence of the interchangeability of ReLU and Leaky ReLU with slope $a = 0.01$, we have successfully retrained well-known models like ResNeXt and WideResNet on ImageNet using Leaky ReLU (see § 4.1 and [24]).

InPlace-ABN I: Backward pass through BN. The gradient $\frac{\partial L}{\partial x} = \{\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_m}\}$, which is obtained from the backward pass through the BN layer, can be written as a function of $\hat{x} = \{\hat{x}_1, \ldots, \hat{x}_m\}$ and $\frac{\partial L}{\partial y} = \{\frac{\partial L}{\partial y_1}, \ldots, \frac{\partial L}{\partial y_m}\}$ as

$$\frac{\partial L}{\partial x_i} = \left\{ \frac{\partial L}{\partial y_i} - \frac{1}{m}\frac{\partial L}{\partial \gamma}\hat{x}_i - \frac{1}{m}\frac{\partial L}{\partial \beta} \right\} \frac{\gamma}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \,,$$

where the gradients of the BN parameters are given by

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}\hat{x}_i \,, \quad \frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \,.$$

The expression above differs from what is found in the original BN paper [15], but the refactoring was already used in the Caffe [16] framework. It is implemented by $\text{BN}^{*}_{\gamma,\beta}$ in the proposed solutions in Figures 2(c) and 2(d) and does not depend on $\mu_{\mathcal{B}}$. Hence, we store during the forward pass only $\sigma_{\mathcal{B}}$ (this dependency was omitted from the diagrams). Instead, $\text{BN}_{\gamma,\beta}$ in Figures 2(a) and 2(b), which depends on x, requires the additional recomputation of $\hat{x}$ from x via Eq. (1). Hence, it also requires storing $\mu_{\mathcal{B}}$. Our solution is hence memory-wise more efficient than the state-of-the-art from Figure 2(b).

Inversion of $\boldsymbol{\pi_{\gamma,\beta}}$. In the configuration of InPlace-ABN I, the inversion of $\pi_{\gamma,\beta}$ becomes critical if $\gamma = 0$ since $\pi^{-1}_{\gamma,\beta}(y) = \frac{y-\beta}{\gamma}$. While we never encountered such a case in practice, one can protect against it by preventing γ from getting less than a given tolerance. We can even avoid this problem by simply not considering γ a learnable parameter and by fixing it to 1, in case the activation function is scale covariant (*e.g.* all ReLU-like activations) and when a Conv layer follows. Indeed, it is easy to show that the network retains the exact same capacity in that case, for γ can be absorbed into the subsequent Conv layer.

InPlace-ABN II: Backward pass through BN. We obtain additional memory savings for our solution illustrated in Figure 2(e) and as outlined in § 3.2. The gradient $\frac{\partial L}{\partial x}$ when written as a function of y instead of $\hat{x}$ becomes

$$\frac{\partial L}{\partial x_i} = \left[\frac{\partial L}{\partial y_i} - \frac{1}{\gamma m}\frac{\partial L}{\partial \gamma} y_i - \frac{1}{m}\left(\frac{\partial L}{\partial \beta} + \frac{\beta}{\gamma}\frac{\partial L}{\partial \gamma} \right) \right] \frac{\gamma}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \,.$$

For the gradients of the BN parameters, $\frac{\partial L}{\partial \beta}$ remains as above but we get

$$\frac{\partial L}{\partial \gamma} = \frac{1}{\gamma}\left[\sum_{j=1}^{m} \frac{\partial L}{\partial y_j} y_j - \beta \frac{\partial L}{\partial \beta} \right]$$

and we write $\text{BN}^{\dagger}_{\gamma,\beta}$ for the actual backward implementation in Figure 2(e). Detailed derivations are provided in [24].

In summary, both of our optimized main contributions are memory-wise more efficient than the state-of-the-art solution in Figure 2(b) and InPlace-ABN II is computationally even more efficient than the proposed, optimized checkpointing from Figure 2(c).

3.4. Implementation Details

We have implemented the proposed InPlace-ABN I layer in PyTorch, by simply creating a new layer that fuses

Algorithm 1 InPlace-ABN Forward

```
Require: $x, \gamma, \beta$
1: $y, \sigma_\mathcal{B} \leftarrow \mathrm{BN}_{\gamma,\beta}(x)$
2: $z \leftarrow \phi(y)$
3: save for backward $z$, $\sigma_\mathcal{B}$
4: return $z$
```

Algorithm 2 InPlace-ABN Backward

```
Require: $\frac{\partial L}{\partial z}, \gamma, \beta$
1: $z, \sigma_\mathcal{B} \leftarrow$ saved tensors during forward
2: $\frac{\partial L}{\partial y} \leftarrow \phi_{\text{backward}}(z, \frac{\partial L}{\partial z})$
3: $y \leftarrow \phi^{-1}(z)$
4: if InPlace-ABN I (see Fig. 2(d)) then
5:     $\hat{x} \leftarrow \pi^{-1}_{\gamma,\beta}(y)$
6:     $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial \gamma}, \frac{\partial L}{\partial \beta} \leftarrow \mathrm{BN}^*_{\gamma,\beta}(\hat{x}, \frac{\partial L}{\partial y}, \sigma_\mathcal{B})$
7: else if InPlace-ABN II (see Fig. 2(e)) then
8:     $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial \gamma}, \frac{\partial L}{\partial \beta} \leftarrow \mathrm{BN}^\dagger_{\gamma,\beta}(y, \frac{\partial L}{\partial y}, \sigma_\mathcal{B})$
9: return $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial \gamma}, \frac{\partial L}{\partial \beta}$
```

batch normalization with an (invertible) activation function. In this way we can deal with the computation of $\hat{x}$ from z internally in the layer, thus keeping the implementation self-contained. We have released code at https://github.com/mapillary/inplace_abn for easy plug-in replacement of the block BN+Act in modern architectures. The forward and backward implementations are also given as pseudocode in Algorithm 1 and 2. In the forward pass, in line 3, we explicitly indicate the buffers that are stored and needed for the backward pass. Any other buffer can be overwritten with in-place computations, *e.g.* x, y and z can point to the same memory location. In the backward pass, we recover the stored buffers in line 1 and, again, every computation can be done in-place if the buffer is not needed anymore (*e.g.* $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial y}$, $\frac{\partial L}{\partial z}$ can share the same memory location as well as $\hat{x}$, y and z).

4. Experiments

We assess the effectiveness of our proposed, memory efficient InPlace-ABN layer for the tasks of image classification and semantic segmentation in § 4.1 and 4.2, respectively. Additionally, we provide timing analyses in § 4.3. Experiments were run and timed on machines comprising four NVIDIA Titan Xp cards (with 12GB of RAM each). Where not otherwise noted, the activation function used in all experiments is Leaky ReLU with slope $a = 0.01$.

4.1. Image Classification

We have trained several residual-unit-based models on ImageNet-1k [25] to demonstrate the effectiveness of InPlace-ABN for the task of image classification. In particular, we focus our attention on two aspects: i) whether using an invertible activation function (*i.e.* Leaky ReLU in our experiments) impacts on the classification performance of the models, and ii) how the memory savings obtained with our method can be exploited to improve classification accuracy. We have trained `ResNeXt-101`/`ResNeXt-152` [30] models (using cardinality 64, SGD with Nesterov updates, initial learning rate 0.1, weight decay 10^{-4} and momentum 0.9, 90 epochs in total and reducing the learning rate every 30 epochs by a factor of 10). We additionally trained `DenseNet-264` [12] and `WideResNet-38` [29] with the same hyperparameters, except for the latter using a linear learning rate decay from 0.1 to 10^{-6}. For all models, we proportionally scale input images so that their smallest side equals 256 pixels, before randomly taking 224×224 crops. RGB images are per-channel mean and variance normalized and color augmentation is applied as described in [30].

Discussion of results. As a baseline, we train `ResNeXt-101` with standard Batch Normalization and the maximum batch size that fits in GPU memory, *i.e.* 256 images per batch. Then we consider two different scenarios: i) using the extra memory to fit more images per training batch while fixing the network architecture, or ii) fixing the batch size while training a larger network. For option i) we double the batch size to 512 (`ResNeXt-101`, InPlace-ABN, 512 in Table 1), while for option ii) we train `ResNeXt-152` and `WideResNet-38`. Note that neither `ResNeXt-152` nor `WideResNet-38` would fit in memory when using 256 images per training batch and when using standard BN. As it is clear from the table, both i) and ii) result in a noticeable performance increase. Interestingly, training `ResNeXt-101` with an increased batch size results in similar accuracy to the deeper (and computationally more expensive) `ResNeXt-152` model. As an additional reference, we train `ResNeXt-101` with synchronized Batch Normalization (InPlace-ABN$^{\text{sync}}$), which can be seen as a "virtual" increase of batch size applied to the computation of BN statistics. In this case we only observe small accuracy improvements when compared to the baseline model. Finally, we also report results for `DenseNet-264` [12], which was trained with a batch size of 256 that otherwise also would not fit in GPU memory. All models can be downloaded from our github page.

4.2. Semantic Segmentation

The goal of semantic segmentation is to assign categorical labels to each pixel in an image. State-of-the-art segmentations are typically obtained by combining classification models pretrained on ImageNet (typically referred to as *body*) with segmentation-specific *head* architectures and jointly fine-tuning them on suitable, (densely) annotated training data like Cityscapes [5], COCO-Stuff [1], ADE20K [34] or Mapillary Vistas [21].

Network		224^2 center		224^2 10-crops		320^2 center	
	batch size	top-1	top-5	top-1	top-5	top-1	top-5
ResNeXt-101, STD-BN	256	77.04	93.50	78.72	94.47	77.92	94.28
ResNeXt-101, INPLACE-ABN	512	78.08	93.79	79.52	94.66	79.38	94.67
ResNeXt-152, INPLACE-ABN	256	78.28	94.04	79.73	94.82	79.56	94.67
WideResNet-38, INPLACE-ABN	256	79.72	94.78	81.03	95.43	80.69	95.27
DenseNet-264, INPLACE-ABN	256	78.57	94.17	79.72	94.93	79.49	94.89
ResNeXt-101, INPLACE-ABN$^{\text{sync}}$	256	77.70	93.78	79.18	94.60	78.98	94.56

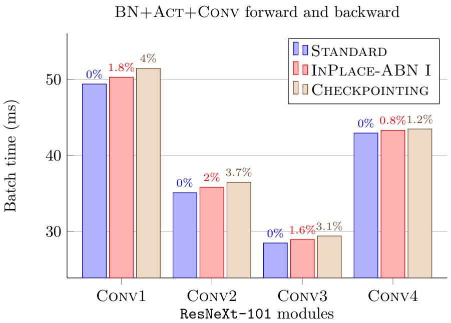


Table 1. (Left) Imagenet validation set results using different architectures and training batch sizes. (Right) Computation time required for a forward and backward pass through basic BN+ACT+CONV blocks from ResNeXt-101, using different BN strategies.

Datasets used for Evaluation. We report results on Cityscapes [5], COCO-Stuff [1] and Mapillary Vistas [21]. Cityscapes shows street-level images captured in central Europe and comprises a total of 5k densely annotated images (19 object categories + 1 *void* class, all images sized 2048×1024), split into 2975/500/1525 images for training, validation and test, respectively. We use the additional 20k training images with so-called *coarse* annotations only for the model evaluated on test data (Table 2, bottom), otherwise we learn only from the high-quality (fine) annotations in the *training* set and test on the corresponding *validation* set. We also provide results on COCO-Stuff, which holds *stuff*-class annotations for the well-known MS COCO dataset [17]. This dataset comprises 55k COCO images (with 40k for training, 5k for validation, 5k for test-dev and 5k as challenge test set) with annotations for 91 stuff classes and 1 *void* class. Finally, we report results on Mapillary Vistas (research edition), a novel and large-scale street-level image dataset comprising 25k densely annotation images (65 object categories + 1 *void* class, images have varying aspect ratios and sizes up to 22 Megapixel), split into 18k/2k/5k images for training, validation and test, respectively. Here, we only use the training set when testing on validation data, and use both training and validation when evaluating on test data.

Segmentation approach. We chose to adopt the recently introduced DeepLabV3 [3] segmentation approach as head, and evaluate its performance with body networks from § 4.1. DeepLabV3 is exploiting atrous (dilated) convolutions in a cascaded way for capturing contextual information, together with crop-level features encoding global context (close in spirit to PSPNet's [33] global feature). We follow the parameter choices suggested in [3], assembling the head as 4 parallel CONV blocks with 256 output channels each and dilation rates $(1, 12, 24, 36)$ (with x8 downsampled crop sizes from the body) and kernel sizes $(1^2, 3^2, 3^2, 3^2)$, respectively. The global 1×1 features are computed in a channel-specific way and CONVed into 256 additional channels. Each output block is followed by BatchNorm before all 1280 features are stacked and reduced by another CONV+BN+ACT block (into 256 features) and finally CONVed to the number of target classes. We exploit our proposed INPLACE-ABN strategy also in the head architecture. Finally, we apply bilinear upsampling to the logits to obtain the original input crop resolution before computing the loss using an *online bootstrapping* strategy as described in [23, 28] (setting $p = 1.0$ and $m = 25\%$). We did not apply hybrid dilated convolutions [27] nor added an auxiliary loss as proposed in [33]. Training data is sampled in a uniform way unless otherwise stated (by shuffling the database in each epoch) and all Cityscapes experiments are run for 360 epochs using an initial learning rate of 2.5×10^{-3} and polynomial learning rate decay $(1 - \frac{iter}{max\_iter})^{0.9}$, following [3]. COCO-Stuff experiments were trained only for 30 epochs, which however approximately matches the number of iterations on Cityscapes due to the considerably larger dataset size. For optimization, we use stochastic gradient descent with momentum 0.9 and weight decay 10^{-4}. For training data augmentation, we apply random horizontal flipping (with prob. 0.5) and random scaling selected from 0.7 - 2.0 before cropping the actual patches.

Discussion of Results. In Table 2 (top), we provide results (all scores are averaged Jaccard numbers) on validation data for Cityscapes and COCO-Stuff under different BN layer configurations. We distinguish between standard BN layers [15] (coined STD-BN) and our proposed variants using in-place, activated BN (INPLACE-ABN) as well as its gradient-synchronized version INPLACE-ABN$^{\text{sync}}$. All experiments are based on LEAKY RELU activations. Trainings were conducted in a way to maximize GPU memory utilization by *i)* fixing the training crop size and therefore pushing the amount of crops per minibatch to the limit (denoted as FIXED CROP) or *ii)* fixing the number of crops per minibatch and maximizing the training crop resolutions (FIXED BATCH). Experiments are conducted for ResNeXt-101 and WideResNet-38 network bodies, where the latter seems preferable for segmentation tasks.

BatchNorm	`ResNeXt-101`				`WideResNet-38`			
	Cityscapes		COCO-Stuff		Cityscapes		COCO-Stuff	
Std-BN + Leaky ReLU	16×512^2	74.42	16×480^2	20.30	20×512^2	75.82	20×496^2	22.44
InPlace-ABN, fixed crop	28×512^2 **[+75%]**	75.80	24×480^2 **[+50%]**	22.63	28×512^2 **[+40%]**	77.75	28×496^2 **[+40%]**	22.96
InPlace-ABN, fixed batch	16×672^2 **[+72%]**	77.04	16×600^2 **[+56%]**	23.35	20×640^2 **[+56%]**	**78.31**	20×576^2 **[+35%]**	24.10
InPlace-ABN^{sync}, fixed batch	16×672^2 **[+72%]**	**77.58**	16×600^2 **[+56%]**	**24.91**	20×640^2 **[+56%]**	78.06	20×576^2 **[+35%]**	**25.11**
Cityscapes **val** (single model & scale)	12×872^2	79.16	Cityscapes **val** (single model & scale) + Class-Uniform sampling				12×872^2	**79.40**
Cityscapes **test** (single Vistas pre-trained model, 5 scales + horizontal flipping, fine + coarse label data) + Class-Uniform sampling							12×872^2	**82.03**
Mapillary Vistas **val** (single model & scale, no horizontal flipping) + Class-Uniform sampling							12×776^2	**53.12**
Mapillary Vistas **test** (single model & scale, no horizontal flipping) + Class-Uniform sampling							12×776^2	**53.37**

Table 2. (Top) Validation results (single scale test) for segmentation experiments on Cityscapes and COCO-Stuff, using `ResNeXt-101` and `WideResNet-38` network bodies and different batch compilations (see text). (Bottom) Validation and test data results using `WideResNet-38`+InPlace-ABN^{sync} models for Cityscapes and Vistas with tuned hyperparameters. All result numbers in [%].

Both body networks were solely trained on ImageNet-1k. All results at the top of Table 2 derive from single-scale testing without horizontal image flipping. In general, results improve when applying more training data (in terms of both, #training crops per minibatch and input crop resolutions). The increase of data (w.r.t. pixels/minibatch) we can put in GPU memory, relative to the baseline (first row) is reported in square brackets. We observe that higher input resolution is in general even more beneficial than adding more crops to the batch.

Results for tuned `WideResNet-38` InPlace-ABN^{sync} models using even larger input crops are shown on the bottom of Table 2 for both, validation and test data. The combination of using InPlace-ABN^{sync} with fewer, but larger crops and an alternative minibatch compilation strategy coined Class-Uniform sampling yields new high scores for both, Cityscapes (model is Vistas pre-trained, used also coarsely labeled data for training and multi-scale $[1.0, 1.25, 1.5, 1.75, 2.0]$ + horizontally flipped testing) and Mapillary Vistas (initial learning rate of 3.5×10^{-3} and trained for 90 epochs, single scale inference) test datasets. For Class-Uniform sampling, we compiled the minibatches per epoch in a way to show all classes uniformly instead of randomly perturbing the dataset (thus following an oversampling strategy for underrepresented categories).

4.3. Timing analyses

Besides the discussed memory improvements and their impact on computer vision applications, we also provide actual runtime comparisons and analyses for the InPlace-ABN I setting shown in 2(d), as this is the implementation we released on github. Isolating a single BN+Act+Conv block, we evaluate the computational times required for a forward and backward pass over it (Figure right to Table 1). We compare the conventional approach of serially executing layers and storing intermediate results (Standard), our proposed InPlace-ABN I and the Checkpointing approach. In order to obtain fair timing comparisons, we re-implemented the checkpointing idea in PyTorch. The results are obtained by running all operations over a batch comprising 32-images and setting the meta-parameters (number of feature channels, spatial dimensions) to those encountered in the four modules of `ResNeXt-101`, denoted as Conv1-Conv4. The actual runtimes were averaged over 200 iterations.

We observe consistent speed advantages in favor of our method when comparing against Checkpointing, with the actual percentage difference depending on block's meta-parameters. As we can see, InPlace-ABN induces computation time increase between $0.8-2\%$ over Standard while Checkpointing is almost doubling our overheads.

5. Conclusions

In this work we have presented InPlace-ABN, which is a novel, computationally efficient fusion of batch normalization and activation layers, targeting memory-optimization for modern deep neural networks during training time. We reconstruct necessary quantities for the backward pass by inverting the forward computation from the storage buffer, and manage to free up almost 50% of the memory needed in conventional BN+Act implementations at little additional computational costs. In contrast to state-of-the-art checkpointing attempts, our method is reconstructing discarded buffers *backwards* during the backward pass, thus allowing us to encapsulate BN+Act as self-contained layer, which is easy to implement and deploy in virtually all modern deep learning frameworks. We have validated our approach with experiments for image classification on ImageNet-1k and semantic segmentation on Cityscapes, COCO-Stuff and Mapillary Vistas. Our obtained networks have performed consistently and considerably better when trained with larger batch sizes (or training crop sizes), leading to new high-scores on the challenging Cityscapes and Mapillary Vistas datasets.

Acknowledgements. We acknowledge financial support from project *DIGIMAP*, funded under grant #860375 by the Austrian Research Promotion Agency (FFG).

References

[1] H. Caesar, J. R. R. Uijlings, and V. Ferrari. COCO-Stuff: Thing and stuff classes in context. *CoRR*, abs/1612.03716, 2016. 6, 7

[2] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *CoRR*, abs/1606.00915, 2016. 1

[3] L. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation. *CoRR*, abs/1706.05587, 2017. 7

[4] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. 2, 3, 4

[5] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The Cityscapes dataset for semantic urban scene understanding. In *(CVPR)*, 2016. 6, 7

[6] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *(NIPS)*. 2015. 2

[7] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. In *(NIPS)*, December 2017. 2, 4

[8] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. In *(NIPS)*, 2016. 2

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 1

[10] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016. 1

[11] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017. 2

[12] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *(CVPR)*, July 2017. 2, 6

[13] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *(NIPS)*. 2016. 2

[14] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016. 2

[15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. 3, 5, 7

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 5

[17] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common objects in context. *CoRR*, abs/1405.0312, 2014. 7

[18] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. 5

[19] J. Martens and I. Sutskever. *Training Deep and Recurrent Networks with Hessian-Free Optimization*, pages 479–535. Springer Berlin Heidelberg, 2012. 2, 3, 4

[20] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017. 2

[21] G. Neuhold, T. Ollmann, S. Rota Bulò, and P. Kontschieder. The mapillary vistas dataset for semantic understanding of street scenes. In *(ICCV)*, October 2017. 6, 7

[22] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, and K. Q. Weinberger. Memory-efficient implementation of densenets. *CoRR*, abs/1707.06990, 2017. 4

[23] S. Rota Bulò, G. Neuhold, and P. Kontschieder. Loss max-pooling for semantic image segmentation. In *(CVPR)*, July 2017. 7

[24] S. Rota Bulò, L. Porzi, and P. Kontschieder. In-place activated batchnorm for memory-optimized training of DNNs. *CoRR*, abs/1712.02616, December 2017. 5

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *(IJCV)*, 2015. 2, 6

[26] C. Szegedy, S. Ioffe, and V. Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. 2

[27] P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou, and G. W. Cottrell. Understanding convolution for semantic segmentation. *CoRR*, abs/1702.08502, 2017. 7

[28] Z. Wu, C. Shen, and A. van den Hengel. High-performance semantic segmentation using very deep fully convolutional networks. *CoRR*, abs/1604.04339, 2016. 1, 7

[29] Z. Wu, C. Shen, and A. van den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *CoRR*, abs/1611.10080, 2016. 6

[30] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016. 1, 2, 6

[31] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015. 5

[32] S. Zagoruyko and N. Komodakis. Wide residual networks. In *(BMVC)*, 2016. 2

[33] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia. Pyramid scene parsing network. *CoRR*, abs/1612.01105, 2016. 7

[34] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba. Semantic understanding of scenes through the ADE20K dataset. *CoRR*, abs/1608.05442, 2016. 6