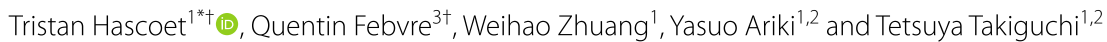
EURASIP Journal on Image
and Video Processing

RESEARCH **Open Access**

Reversible designs for extreme memory cost reduction of CNN training


Tristan Hascoet[1*†], Quentin Febvre[3†], Weihao Zhuang[1], Yasuo Ariki[1,2] and Tetsuya Takiguchi[1,2]

†Tristan Hascoet and Quentin Febvre contributed equally

*Correspondence: tristan@people.kobe-u.ac.jp

[1] Kobe University, 1-1 Rokkodaicho, Nada Ward, Kobe 657-0013, Japan
[2] Association for Advanced Science and Technology, 1-1 Rokkodaicho, Nada Ward, Kobe 657-0013, Japan
[3] IMT Atlantique, 655 Avenue du Technopôle, Plouzané 29280, France

Abstract

Training Convolutional Neural Networks (CNN) is a resource-intensive task that requires specialized hardware for efficient computation. One of the most limiting bottlenecks of CNN training is the memory cost associated with storing the activation values of hidden layers. These values are needed for the computation of the weights' gradient during the backward pass of the backpropagation algorithm. Recently, reversible architectures have been proposed to reduce the memory cost of training large CNN by reconstructing the input activation values of hidden layers from their output during the backward pass, circumventing the need to accumulate these activations in memory during the forward pass. In this paper, we push this idea to the extreme and analyze reversible network designs yielding minimal training memory footprint. We investigate the propagation of numerical errors in long chains of invertible operations and analyze their effect on training. We introduce the notion of pixel-wise memory cost to characterize the memory footprint of model training, and propose a new model architecture able to efficiently train arbitrarily deep neural networks with a minimum memory cost of 352 bytes per input pixel. This new kind of architecture enables training large neural networks on very limited memory, opening the door for neural network training on embedded devices or non-specialized hardware. For instance, we demonstrate training of our model to 93.3% accuracy on the CIFAR10 dataset within 67 minutes on a low-end Nvidia GTX750 GPU with only 1GB of memory.

Keywords: CNN, Reversibility, Memory optimization, Numerical analysis

1 Introduction

Over the last few years, Convolutional Neural Networks (CNN) have enabled unprecedented progress on a wide array of computer vision tasks. One disadvantage of these approaches is their resource consumption: training deep models within a reasonable amount of time requires special Graphical Processing Units (GPU) with numerous cores and large memory capacity. Given the practical importance of these models, a lot of research effort has been directed towards algorithmic and hardware innovations to improve their resource efficiency such as low-precision arithmetic [1], network pruning for inference [2], or efficient stochastic optimization algorithms [3].

Springer Open

© The Author(s) 2023. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

In this paper, we focus on a particular aspect of resource efficiency: optimizing the memory cost of training CNNs. We envision several potential benefits from the ability to train large neural networks within limited memory:

Democratization of deep learning research: Training large CNNs requires special GPUs with large memory capacity. Typical desktop GPUs memory capacity is too small for training large CNNs. As a result, getting into deep learning research comes with the barrier cost of either buying specialized hardware or renting live instances from cloud service providers. Reducing the memory cost of deep model training would allow training deep networks on standard graphic cards without the need for specialized hardware, effectively removing this barrier cost. In this paper, we demonstrate efficient training of a CNN on the CIFAR10 dataset (93.3% accuracy within 67 min) on an Nvidia GTX750 with only 1 GB of memory.

On-device training: With mobile applications, a lot of attention has been given to optimize inference on edge devices with limited computation resources. Training state-of-the-art CNN on embedded devices, however, has still received little attention. Efficient on-device training is a challenging task for the underlying power efficiency, computation and memory optimization challenges it involves. As such, CNN training has thus far been relegated to large cloud servers, and trained CNNs are typically deployed to embedded device fleets over the network. On-device training would allow bypassing these server–client interactions over the network. We can think of several potential applications of on-device training, including:

- Life-long learning: Autonomous systems deployed in evolving environments like drones, robots or sensor networks might benefit from continuous life-long learning to adapt to their changing environment. On-device training would enable such application without the expensive communication burden of having edge devices continuously sending their data to remote servers over the network. It would also provide resilience to network failures in critical application scenarios.
- In privacy-critical applications such as biometric mobile phone authentication, users might not want to have their data sent over the network. On-device training would allow fine-tuning recognition models on local data without sending sensitive data over the network.

In this work, we propose an architecture with minimal training memory cost requirements which enables training within the tight memory constraints of embedded devices.

Research in optimization: Recent works on stochastic optimization algorithms have highlighted the benefits of large batch training [4, 5]. For example, in Imagenet, linear speed-ups in training have been observed with increasing batch sizes up to tens of thousands of samples [5]. Optimizing the memory cost of CNN training may allow further research on the optimization trade-offs of large batch training. For small datasets like MNIST or CIFAR10, we are able to process the full dataset in 14 and 18 GB of memory, respectively. Although large batch training on such small dataset is very computationally inefficient with current stochastic optimization algorithms [5], the ability to process the full dataset in one pass allows to easily train CNNs on the true gradient of the error. Memory optimization techniques have the potential to facilitate research

on optimization techniques outside the realm of Stochastic Gradient Descent to be investigated.

In this paper, we build on recent works on reversible networks [6, 7] and ask the question: how far can we reduce CNN training memory cost using reversible designs with minimal impact on the accuracy and computational cost? To do so, we take as a starting point the Resnet-18 architecture and analyze its training memory requirements. We then analyze the memory cost reduction of invertible designs successively introduced in the RevNet and iRevNet architectures. We identify the memory bottleneck of such architectures, which leads us to introduce a layer-wise invertible architecture. However, we observe that layer-wise invertible networks accumulate numerical errors across their layers, which leads to numerical instabilities impacting model accuracy. We characterize the accumulation of numerical errors within long chains of revertible operations and investigate their effect on model accuracy. To mitigate the impact of these numerical errors on the model accuracy, we propose both a reparameterization of invertible layers and a hybrid architecture combining the benefits of layer-wise and residual-block-wise reversibility to stabilize training.

Our main result is to present a new architecture that allows to efficiently train a CNN with the minimal memory cost of 352 bytes per pixel. We demonstrate the efficiency of our method by efficiently training a model to 93.3% accuracy on the CIFAR10 dataset within 67 minutes on a low-end Nvidia GTX750 with only 1 GB of VRAM.

2 Related work

2.1 Reversibility

Reversible network designs have been proposed for various purposes including generative modeling, visualization, solving inverse problems, or theoretical analysis of hidden representations.

Flow-based generative models use analytically invertible transformations to compute the change of variable formula. Invertibility is either achieved through channel partitioning schemes (NICE [8] Real-NVP [9]), weight matrix factorization (GLOW [10]) or constraining layer architectures to easily invertible unitary operations (Normalization flows [11])

Neural ODEs [12] take a drastically different take on invertibility: They leverage the analogy between residual networks and the Euler method to define continuous hidden state systems. The conceptual shift from a finite set of discrete transformations to a continuous regime gives them invertibility for free. The computational efficiency of this approach, however, remains to be demonstrated.

The RevNet model [6] was inspired by the Real-NVP generative model. They adapt the idea of channel partitioning and propose an efficient architecture for discriminative learning. The iRevNet [7] model builds on the RevNet architecture: they propose to replace the irreversible max-pooling operation with an invertible operation that reshapes the hidden activation states so as to compensate the loss of spatial resolution by an increase in the channel dimension. By preserving the volume of activations, their pooling operation allows for exact reconstruction of the inverse. In their original work, the authors focus on the analysis of the representations learned by invertible models rather than resource efficiency. From a resource optimization point of view, one

downside of their method is that the proposed invertible pooling scheme drastically increases the number of channels in upper layers. As the size of the convolution kernel weights grows quadratically in the number of channels, the memory cost associated with storing the model weights becomes a major memory bottleneck. We address this issue in our proposed architecture. In [13], the authors use these reversible architectures to study undesirable invariances in feature space.

In [14], the authors propose a unified architecture performing well on both generative and discriminative tasks. They enforce invertibility by regularizing the weights of residual blocks so as to guarantee the existence of an inverse operation. However, the computation of the inverse operation is performed with power iteration methods which are not optimal from a computational perspective.

Finally, [15] propose to reconstruct the input activations of normalization and activation layers using their inverse function during the backward pass. We propose a similar method for layer-wise invertible networks. However, as their model does not invert convolution layers, it does not feature long chains of invertible operations so that they do not need to account for numerical instabilities. Instead, our proposed model features long chains of invertible operations so that we need to characterize numerical errors in order to stabilize training.

2.2 Resource efficiency

Research into resource optimization of CNNs covers a wide array of techniques, most of which are orthogonal to our work. We briefly present some of these works.

On the architectural side, Squeezenet [16] was first proposed as an efficient neural architecture reducing the number of model parameters while maintaining high classification accuracy. MobileNet [17] uses depth-wise separable convolutions to further reduce the computational cost of inference for embedded device applications.

Network pruning [2] is a set of techniques developed to decrease the model weight size and computational complexity. Network pruning works by removing the network weights that contribute the least to the model output. Pruning deep models has been shown to drastically reduce the memory cost and computational cost of inference without significantly hurting model accuracy. Although pruning has been concerned with optimization of the resource inference, the recently proposed lottery ticket hypothesis [18] has shown that specifically pruned networks could be trained from scratch to high accuracy. This may be an interesting and complementary line of work to investigate in the future to reduce training memory costs.

Low precision arithmetic has been proposed as a mean to reduce both memory consumption and computation time of deep learning models. Mixed precision training [19] combines FP16 with FP32 operations to avoid numerical instabilities due to either overflow or underflow. For inference, integer quantization [1, 20] has been shown to drastically improve the computation and memory efficiency and has been successfully deployed on both edge devices and data centers. Integrating mixed-precision training to our proposed architecture would allow us to further reduce training memory costs.

Accumulating the weights' gradients over multiple batches is used to increase the effective batch size during the training with constant memory requirements. Although

this method allows for training on arbitrary large batch sizes, it does not reduce the memory requirements for training on a single batch.

Most related to our work, gradient checkpointing was introduced as a mean to reduce the memory cost of deep neural network training. Gradient checkpointing, first introduced in [21], trades off memory for computational complexity by storing only a subset of the activations during the forward pass. During the backward pass, missing activations are recomputed from the stored activations as needed by the backpropagation algorithm. Follow-up work [22] has since built on the original gradient checkpointing algorithm to improve this memory/computation trade-off. However, reversible models like RevNet have been shown to offer better computational complexity than gradient checkpointing, at the cost of constraining the model architecture to invertible residual blocks.

3 Preliminaries

In this section, we analyze the memory footprint of training architectures with different reversibility patterns. We start by introducing some notations and briefly review the backpropagation algorithm in order to characterize the training memory consumption of deep neural networks. In our analysis, we use a Resnet-18 as a reference baseline and analyze its training memory footprint. We then gradually augment the baseline architecture with reversible designs and analyze their impact on computation and memory consumption.

3.1 Backpropagation and notations

Let us consider a model F made of N sequential layers trained to minimize the error e defined by a loss function $\mathcal{L}$ for an input x and ground-truth label $\bar{y}$:

$$F : x \rightarrow y, \tag{1a}$$

$$y = f_N \circ \ldots \circ f_2 \circ f_1(x), \tag{1b}$$

$$e = \mathcal{L}(y, \bar{y}). \tag{1c}$$

During the forward pass, each layer f_i takes as input the activations z_{i-1} from the previous layer and outputs activation features $z_i = f_i(z_{i-1})$, with $z_0 = x$ and $z_N = y$ being the input and output of the network, respectively.

During the backward pass, the gradient of the loss with respect to the hidden activations are propagated backward through the layers of the networks using the chain rule as:

$$\frac{\delta \mathcal{L}}{\delta z_{i-1}} = \frac{\delta \mathcal{L}}{\delta z_i} \times \frac{\delta z_i}{\delta z_{i-1}}. \tag{2}$$

Before propagating the loss gradient with respect to its input to the previous layer, each parameterized layer computes the gradient of the loss with respect to its parameters. In vanilla SGD, for a given learning rate η, the weight gradients are subsequently used to update the weight values as:

$$\frac{\delta \mathcal{L}}{\delta \theta_i} = \frac{\delta \mathcal{L}}{\delta z_i} \times \frac{\delta z_i}{\delta \theta_i}, \tag{3a}$$

$$\theta_i \leftarrow \theta_i - \eta \times \frac{\delta \mathcal{L}}{\delta \theta_i}. \tag{3b}$$

However, the analytical form of the weight gradients are functions of the layer's input activations z_{i-1}. In convolution layers, for instance, the weight gradients can be computed as the convolution of the input activation by the output's gradient:

$$\frac{\delta \mathcal{L}}{\delta \theta_i} = z_{i-1} \star \frac{\delta \mathcal{L}}{\delta z_i}. \tag{4}$$

Hence, computing the derivative of the loss with respect to each layer's parameters θ_i requires knowledge of the input activation values z_{i-1}. In the standard backpropagation algorithm, hidden layers activations are stored in memory upon computation during the forward pass. Activations accumulate in live memory buffers until used for the weight gradients computation in the backward pass. Once the weight gradients computed in the backward pass, the hidden activation buffers can be freed from live memory. However, the accumulation of activation values stored within each parameterized layer along the forward pass creates a major bottleneck in GPU memory.

The idea behind reversible designs is to constrain the network architecture to feature invertible transformations. Doing so, activations z_i in lower layers can be recomputed through inverse operations from the activations $z_{j>i}$ of higher layers. In such architectures, activation do not need to be kept in memory during the forward pass as they can be recomputed from higher layer activations during the backward pass, effectively freeing up the GPU live memory.

3.2 Memory footprint

We denote the memory footprint of training a neural network as a value $\mathcal{M}$ in bytes. Given an input x and ground-truth label $\bar{y}$, the memory footprint represents the peak memory consumption during an iteration of training including the forward and backward pass. We divide the total training memory footprint $\mathcal{M}$ into several memory cost factors: the cost M_θ of storing the model weights, the hidden activations M_z, and the hidden activations' gradients M_g:

$$\mathcal{M} = M_\theta + M_z + M_g. \tag{5}$$

We choose not to include the cost of storing the gradients of the weights in our analysis since their accumulation has more to do with the implementation details of current differentiable frameworks than with algorithmic necessity. In the following subsections, we detail the memory footprint of existing architectures with different reversibility patterns. To help us formalize these memory costs, we further introduce the following notations: let $n(x)$ denote the number of elements in a tensor x, i.e., if x is an $h \times w$ matrix, then $n(x) = h \times w$. Let bpe be the memory cost in bytes per elements of a given precision so that the actual memory cost for storing an $h \times w$ matrix is $n(x) \times bpe$. For instance,

FP32 tensors have a memory cost per element $bpe = 4$. We use bs to denote the batch size, and c_i to denote the number of channels at layer i.

It should be noted that the memory cost of the activations and the gradients are proportional to the size of the input image batch: training a CNN on twice larger input image batch sizes or twice higher resolution requires twice more memory. Thus, these costs, for a given architecture, are better characterized in bytes per input pixels, which we denote M_z' and M_g', respectively, and are defined by:

$$M_z' = \frac{M_z}{bs \times h \times w}, \tag{6a}$$

$$M_g' = \frac{M_g}{bs \times h \times w}. \tag{6b}$$

The memory cost of the weights, on the other hand, is independent of the input size and thus reported in bytes.

3.3 Vanilla ResNet

The architecture of a vanilla ResNet-18 is shown in Fig. 1. Vanilla ResNets do not use reversible computations so that the input activations of all parameterized layers need to

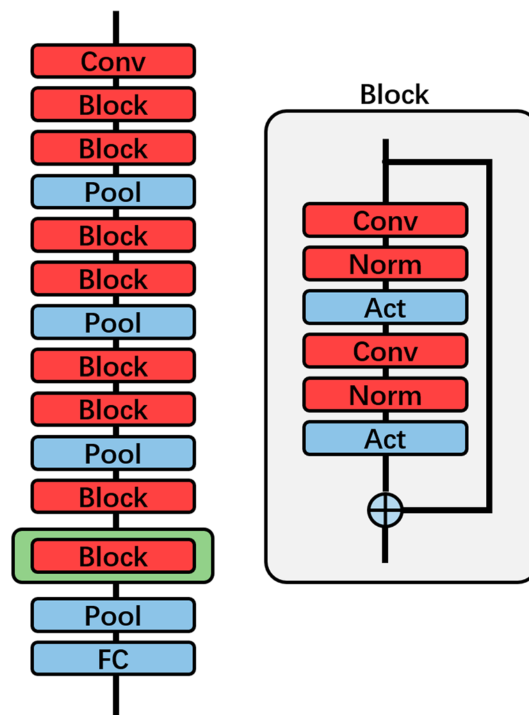


Fig. 1 Illustration of the ResNet-18 architecture and its memory requirements. Modules contributing to the peak memory consumption are shown in red. These modules contribute to the memory cost by storing their input in memory. The green annotation represents the extra memory cost of storing the gradient in memory. The peak memory consumption happens in the backward pass through the last convolution so that this layer is annotated with an additional gradient memory cost. At this step of the computation, all lower parameterized layers have stored their input in memory, which constitutes the memory bottleneck

be accumulated in memory during the forward pass for the computation of the weight gradients to be done in the backward pass.

Hence the peak memory footprint of training a vanilla ResNet happens at the beginning of the backward pass when the top layer's activation gradients need to be stored in memory in addition to the full stack of hidden activation values.

Let us denote by $P \subset N$ the subset of parameterized layers of a network F (i.e., convolutions and batch normalization layers, excluding activation functions and pooling layers). The memory cost associated with storing the hidden activation values is given by:

$$M_z = \sum_{i \in P} n(z_i) \times bpe \tag{7a}$$

$$= \sum_{i \in P} bs \times c_i \times h_i \times w_i \times bpe, \tag{7b}$$

where h_i and w_i represent the spatial dimensions of the activation values at layer i. h_i and w_i are determined by the input image size $h \times w$ and the pooling factor p_i of layer i, so we can factor out both the spatial dimensions and the batch size from this equation, yielding the memory cost per input pixel:

$$M_z = \sum_{i \in P} bs \times h \times w \times p_i \times c_i \times bpe \tag{8a}$$

$$= bs \times h \times w \times \sum_{i \in P} p_i \times c_i \times bpe, \tag{8b}$$

$$M'_z = \sum_{i \in P} p_i \times c_i \times bpe. \tag{8c}$$

The memory footprint of the weights is given by:

$$M_\theta = \sum_{i \in P} n(\theta_i) \times bpe. \tag{9}$$

The memory footprint of the gradients correspond to the size of the gradient buffers at the time of peak memory usage. In a vanilla ResNet18 model, this peak memory usage happens during the backward pass through the last convolution of the network. Hence, the memory footprint of the gradients correspond to the memory cost of storing the gradients with respect to either the input or the output of this layer.

$$M_g = max(n(g_{N-1}), n(g_N)) \times bpe \tag{10a}$$

$$= h \times w \times bs \times p_i \times max(c_{N-1}, c_N) \times bpe, \tag{10b}$$

$$M'_g = p_i \times max(c_{N-1}, c_N) \times bpe. \tag{10c}$$

Figure 1 illustrates the peak memory consumption of a ResNet-like architecture. For a ResNet parameterized following Table 1, the peak memory consumption can then be computed as:

$$\mathcal{M} = M_\theta + M_z + M_g \tag{11a}$$

$$= M_\theta + (M'_z + M'_g) \times (h \times w \times bs) \tag{11b}$$

$$= 12.5 * 10^6 + 1928 \times (h \times w \times bs). \tag{11c}$$

(11d)

For example, a training iteration over a typical batch of 32 images of resolution 240×240 requires 12.5 MB of memory to store the model weights and 3.8 GB of memory to store the hidden layers activations and gradients for a total of $\mathcal{M} = 3.81$ GB of VRAM. The memory cost of the hidden activations is thus the main memory bottleneck of training a ResNet as the cost associated with the model weights is negligible in comparison.

3.4 RevNet

The RevNet architecture introduces reversible blocks as drop-in replacements of the residual blocks of the ResNet architecture. Reversible blocks have analytical inverses that allow for the computation of both their input and hidden activation values from the value of their output activations. Two factors create memory bottlenecks in training RevNet architectures, which we refer to as the local and global bottlenecks.

First, the RevNet architecture features non-volume preserving max-pooling layers, for which the inverse cannot be computed. As these layers do not have analytical inverses, their input must be stored in memory during the forward pass for the reconstruction of lower layer's activations to be computed during the backward pass. We refer to the memory cost associated with storing these activations as the global bottleneck, since these activations need to be accumulated during the forward pass through the full architecture.

The local memory bottleneck has to do with the synchronization of the reversible block computations: while activations values are computed by a forward pass through the reversible block modules, gradients computations flow backward through these modules so that the activations and gradient computations cannot be performed simultaneously. Figure 2 illustrates the process of backpropagating through a reversible block: first, the input activation values of the parameterized hidden layers within the reversible blocks are recomputed from the output. Once the full set of activation have been computed and stored in GPU memory, the backpropagation of the gradients through the reversible block can begin. We refer to the accumulation of the hidden activation values within the reversible block as the local memory bottleneck.

For a typical parameterization of a RevNet, as summarized in Table 1, the local bottleneck of lower layers actually outweighs the global memory bottleneck introduced by non-reversible pooling layers. Indeed, as the spatial resolution decreases with pooling operations, the cost associated with storing the input activations of higher layers becomes negligible

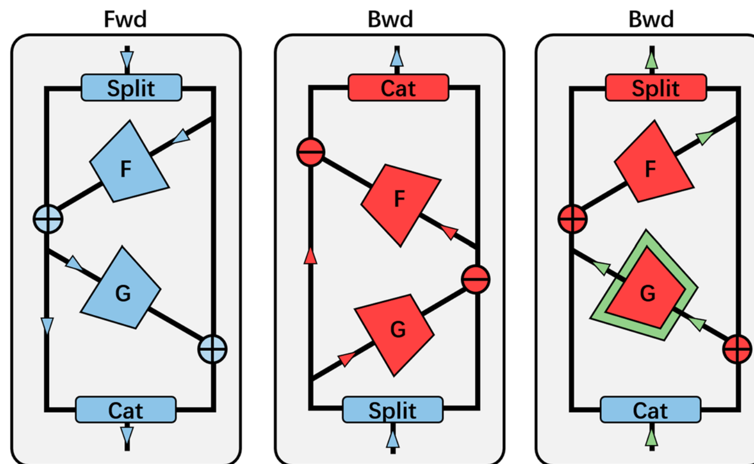


Fig. 2 Illustration of the backpropagation process through a reversible block. In the forward pass (left), activations are propagated forward from top to bottom. The activations are not kept in live memory as they are to be recomputed in the backward pass so no memory bottleneck occurs. The backward pass is made of two phases: first the hidden and input activations are recomputed from the output through an additional forward pass through both modules (middle). Once the activations recomputed, the activations gradient are propagated backward through both modules of the reversible blocks (right). Because the activation and gradient computations flow in opposite directions through both modules, both computations cannot be efficiently overlapped, which results in the local memory bottleneck of storing all hidden activations within the reversible block before the gradient backpropagation step

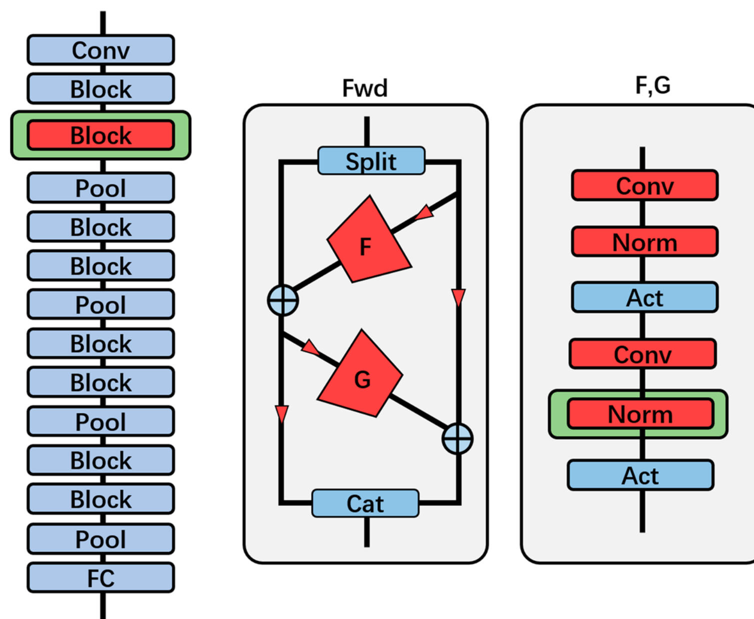


Fig. 3 Illustration of the Revnet architecture and its memory consumption. Modules contributing to the peak memory consumption are shown in red. The peak memory consumption happens during the backward pass through the first reversible block. At this step of the computations, all hidden activations within the reversible block are stored in memory simultaneously

compared to the cost of storing activation values in lower layers. Hence, surprisingly, the peak memory consumption of the RevNet architecture, as illustrated in Fig. 3, happens in the backward pass through the first reversible block, in which the local memory bottleneck

is maximum. For the architecture described in Table 1, the peak memory consumption can be computed as:

$$\mathcal{M} = M_\theta + M_z + M_g \tag{12a}$$

$$= (M_\theta + (M'_z + M'_g) \times (h \times w \times bs) \tag{12b}$$

$$= 12.7 \times 10^6 + 640 \times (h \times w \times bs). \tag{12c}$$

Following our previous example, a RevNet architecture closely mimicking the ResNet-18 architecture requires $\mathcal{M} = 1.19$ GB of VRAM for a training iteration over batch of 32 images of resolution 240×240.

Finally, the memory savings allowed by the reversible block come with the additional computational cost of computing the hidden activations during the backward pass. As noted in the original paper, this computational cost is equivalent to performing one additional forward pass.

3.5 iRevNet

The iRevNet model builds on the RevNet architecture: they replace the irreversible max-pooling operation with an invertible operation that reshapes the hidden activation states so as to compensate for the loss of spatial resolution by an increase in the channel dimension. As such, the iRevNet architecture is fully invertible, which alleviates the global memory bottleneck of the RevNet architecture.

This pooling operation works by stacking the neighboring elements of the pooling regions along the channel dimension, i.e., for a 2D pooling operation with 2×2 pooling window, the number of output channels is four times the number of input channels. Unfortunately, the size of a volume-preserving convolution kernel grows quadratically in the number of input channels:

$$M(\theta) = c_{in} \times c_{out} \times k_h \times k_w \tag{13a}$$

$$= c^2 \times k_h \times k_w. \tag{13b}$$

Consider an iRevNet network with initial channel size 32. After three levels of 2×2 pooling, the effective channel size becomes $32 \times 4^3 = 2048$. A typical 3×3 convolution layer kernel for higher layers of such network would have $n(\theta) = 2048^2 \times 3 \times 3 = 37M$ parameters. At this point, the memory cost of the network weights M_θ becomes an additional memory bottleneck.

Furthermore, the iRevNet architecture does not address the local memory bottleneck of the reversible blocks. Figure 4 illustrates such architecture. For an initial channel size of 32, as summarized in Table 1, the peak memory consumption is given by:

$$\mathcal{M} = M_\theta + M_z + M_g \tag{14a}$$

$$= M_\theta + (M'_z + M'_g) \times (h \times w \times bs) \tag{14b}$$

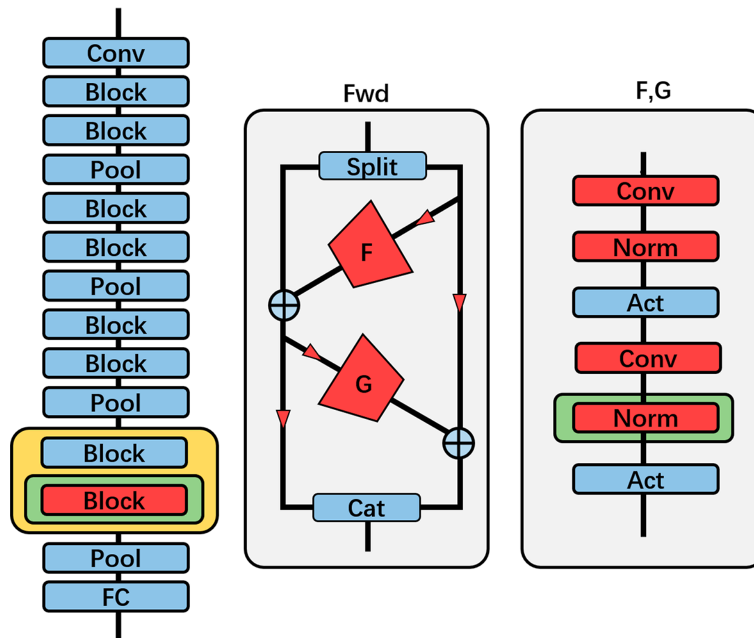


Fig. 4 Illustration of the i-Revnet architecture and its memory consumption. The peak memory consumption happens during the backward pass through the top reversible block. In addition to this local memory bottleneck, the cost of storing the top layers weights (in orange) becomes a new memory bottleneck as the weight kernel size grows quadratically in the number of channels

$$= 171 \times 10^6 + 640 \times (h \times w \times bs). \tag{14c}$$

Training such an architecture for an iteration over batches of 32 images of resolution 240×240 would require $\mathcal{M} = 1.35$GB of VRAM. In the next section, we introduce both layer-wise reversibility, and a variant on this pooling operations to address the local memory bottleneck of reversible blocks, and the weight memory bottleneck, respectively.

4 Method

RevNet and iRevNet architectures implement reversible transformations at the level of residual blocks. As we have seen in the previous section, the design of these reversible blocks creates a local memory bottleneck as all hidden activations within a reversible block need to be computed before the gradients are backpropagated through the block. In order to circumvent this local bottleneck, we introduce layer-wise invertible operations in section 4.2. However, these invertible operations introduce numerical errors, which we characterize in the following subsections. In section 5, we will show that these numerical errors lead to instabilities that degrade the model accuracy. Hence, in "Hybrid architecture", we propose a hybrid model combining layer-wise and residual block-wise reversible operations to stabilize training while resolving the local memory bottleneck at the cost of a small additional computational cost. Section 4.1 starts by motivating the need for, and the methodology of, our numerical error analysis.

4.1 Numerical error analysis

Invertible networks are defined as the composition of invertible operations. During the backward pass, each operation is supposed to reconstruct its input x given the value of its output y using its inverse function:

$$y = f(x), \tag{15a}$$

$$x = f^{-1}(y). \tag{15b}$$

In reality, however, the output of the network is an approximation of its true analytical value due to floating point numbers' precision $\hat{y} = y + \epsilon_y$. Hence, the noisy input $\hat{x}$ reconstructed by the inverse operation contains a noise ϵ_x due to the noise ϵ_y in the output, and the error propagates through the successive inverse computations.

$$\hat{x} = f^{-1}(y + \epsilon_y), \tag{16a}$$

$$\hat{x} = (x + \epsilon_x), \tag{16b}$$

$$\epsilon_x = x - f^{-1}(y + \epsilon_y). \tag{16c}$$

The operation f may either refer to an individual layer, as is the case for the layer-wise invertible architecture we propose in this paper, or at the level of residual blocks as for the reversible blocks proposed in RevNet or iRevNet.

For each operation, we can compute the signal-to-noise ratio (SNR) of its output and input, respectively:

$$snr_o = \frac{|y|^2}{|\epsilon^y|^2}, \tag{17a}$$

$$snr_i = \frac{|x|^2}{|\epsilon^x|^2}. \tag{17b}$$

We are interested in characterizing the factor α of reduction of the SNR through the inverse reconstruction:

$$\alpha = \frac{snr_i}{snr_o}. \tag{18}$$

Indeed, given a layer i in a network, its input z_i will be reconstructed from the noisy network output $\hat{y}$ by the composition of its upstream layers. Hence, the noise ϵ_i in the reconstructed and noisy input $\hat{z}_i$ can be computed as:

$$\hat{z}_i = z_i + \epsilon_i, \tag{19a}$$

$$\hat{z}_i = f_i^{-1} \circ f_{i+1}^{-1} \circ \ldots \circ f_N^{-1}(\hat{y}), \tag{19b}$$

$$|\epsilon_i|^2 = \frac{|\epsilon_y|^2 \times |z_i|^2}{|y|^2} \times \prod_i^N \alpha_j. \tag{19c}$$

As z_i is used in the computation of layer i's weights' gradients according to Eq. 4, accumulated errors yield noisy gradients which prevent the network from converging as the SNR reaches certain levels. Hence, it is important to characterize the factor α for the different invertible layers proposed below.

4.2 Layer-wise invertibility

In this section, we present invertible layers that act as drop-in replacement for convolution, batch normalization, pooling and non-linearity layers. We then characterize the numerical instabilities arising from the invertible batch normalization and non-linearities.

4.2.1 Invertible batch normalization

As batch normalization is not a bijective operation, it does not admit an analytical inverse. However, the inverse reconstruction of a batch normalization layer can be realized with minimal memory cost. Given first- and second-order moment parameters β and γ, the forward f and inverse f^{-1} operation of an invertible batch normalization layer can be computed as follows:

$$y = f(x) = \gamma \times \frac{x - \hat{x}}{\sqrt{\dot{x}} + \epsilon} + \beta, \tag{20a}$$

$$x = f^{-1}(y, \hat{x}, \dot{x}) = (\sqrt{\dot{x}} + \epsilon) \times \frac{y - \beta}{\gamma} + \hat{x}, \tag{20b}$$

where $\hat{x}$ and $\dot{x}$ represent the mean and variance of x, respectively. Hence, the input activation x can be recovered from y through f^{-1} at the minimal memory cost of storing the input activation statistics $\hat{x}$ and $\dot{x}$.

The formula for the SNR reduction factor of the batch normalization is given below:

$$\alpha = \frac{\sum_i (\hat{x}_i^2 + \dot{x}_i)}{\sum_i (\gamma_i^2 + \beta_i^2)} \times \frac{c}{\sum_i \frac{\sqrt{\dot{x}_i} + \epsilon}{\gamma_i}}, \tag{21}$$

in which c represents the number of channels. The full proof of this formula is given in the Appendix. The only assumption made by this proof is that both the input x and output noise ϵ^y are identically distributed across all channels, which we have found to hold true in practice.

In essence, numerical instabilities in the inverse computation of the batch normalization layer arise from the fact that the signal across different channels i and j are amplified by different factors γ_i and γ_j. While the signal amplification in the forward and inverse path cancel out each other ($x = f^{-1}(f(x))$), the noise only gets amplified in the backward pass, which degrades the reconstructed signal.

We verify the validity of equation (22ac) by empirically evaluating the different α ratio yielded by a toy parameterization of the batch normalization using only two channels with parameters and $\gamma = [1, \rho]$. This toy parameterization has been used by the proof in the Appendix. The factor ρ there represents the imbalance in the multiplicative factor between both channels. Figure 5 shows the expected evolution of α through our toy layer for different values of the factor ρ. and find it to closely match the theoretical results we derived.

Finally, we propose the following modification, introducing the hyperparameter ϵ_i, to the invertible batch normalization layer:

$$y = f(x) = |\gamma + \epsilon_i| \times \frac{x - \hat{x}}{\sqrt{\dot{x}} + \epsilon} + \beta, \tag{22a}$$

$$x = f^{-1}(y) = (\sqrt{\dot{x}} + \epsilon) \times \frac{y - \beta}{|\gamma + \epsilon_i|} + \hat{x}. \tag{22b}$$

The introduction of the ϵ_i hyperparameter serves two purposes: first, it stabilizes the numerical errors described above by lower bounding the smallest γ parameters. Second, it prevents numerical instabilities that would otherwise arise from the inverse computation as γ parameters tend towards zero.

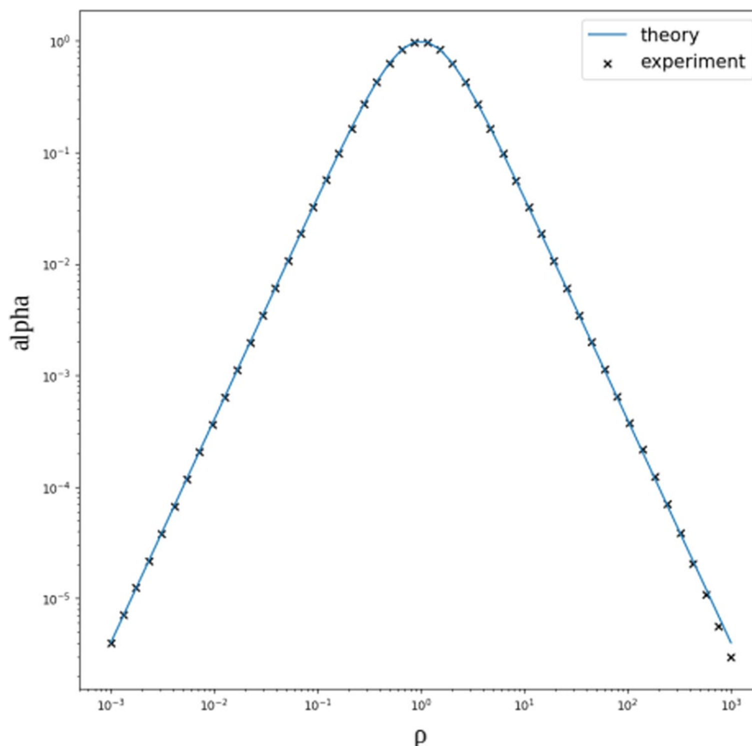


Fig. 5 Illustration of the numerical errors arising from batch normalization layers. Comparison of the theoretical and empirical evolution of the α ratio for different ρ values in our toy example. Empirical values were computed for a Gaussian input signal with zero mean and standard deviation 1 and a white Gaussian noise of standard deviation 10^{-5}

4.2.2 Invertible activation function

A good invertible activation function must be bijective (to guarantee the existence of an inverse function) and non-saturating (for numerical stability). For these properties, we focus our attention on Leaky ReLUs whose forward f and inverse f^{-1} computations are defined, for a negative slope parameter n, as follows:

$$y = f(x) = \begin{cases} x, & \text{if } x > 0 \\ x/n, & \text{otherwise} \end{cases} \tag{23a}$$

$$x = f^{-1}(y) = \begin{cases} y, & \text{if } y > 0 \\ y \times n, & \text{otherwise} \end{cases}. \tag{23b}$$

As derived in the Appendix, and following a similar proof to the batch normalization, we find the below formula for the SNR reduction factor:

$$\alpha = \frac{4}{(1 + \frac{1}{n^2}) \times (1 + n^2)}. \tag{24}$$

Hence numerical errors can be controlled by setting the value of the negative slope n. As n tends towards 1, α converges to 1, yielding minimum signal degradation. However, as n tends towards 1, the network tends toward a linear behavior, which hurts the model expressivity. Figure 6 shows the evolution of the SNR degradation α for different negative

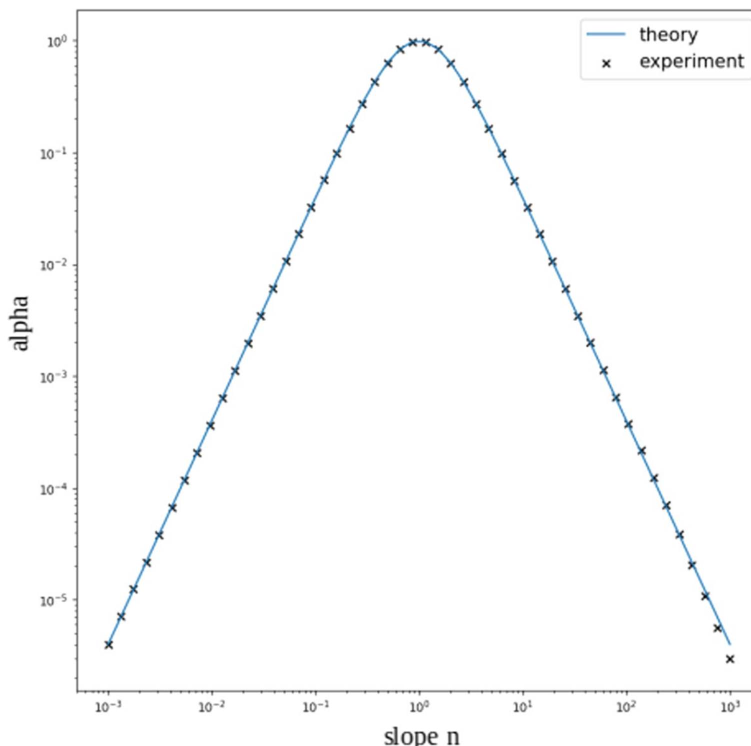


Fig. 6 Illustration of the numerical errors arising from invertible activation layers. Comparison of the theoretical and empirical evolution of the α ratio for different negative slopes n. Empirical values were computed for a Gaussian input signal with zero mean and standard deviation 1 and a white Gaussian noise of standard deviation 10^{-5}

slopes n; and, in section 5, we investigate the impact of the negative slope parameter on the model accuracy.

It should be noted that this equation only holds for the regime $|y|^2 \gg |\epsilon_y|^2$. When the noise reaches an amplitude similar to or greater than the activation signal, this equation no longer holds. However, in this regime, the signal-to-noise ratio becomes too low for training to converge, as numerical errors prevent any useful weight update. We have thus left the problem of characterizing this regime open.

4.2.3 Invertible convolutions

Invertible convolution layers can be defined in several ways. The inverse operation of a convolution is often referred to as deconvolution, and is defined for a subspace of the kernel weight space.

However, deconvolutions are computationally expensive and prone to numerical errors. Instead, we choose to implement invertible convolutions using the channel partitioning scheme of the reversible block for its simplicity, numerical stability and computational efficiency. Hence, invertible convolutions, in our architecture, can be seen as minimal reversible blocks in which both modules consist of a single convolution. Gomez et al. [6] found the numerical errors introduced by reversible blocks to have no impact on the model accuracy. Similarly, we found reversible blocks extremely stable yielding negligible numerical errors compared to the invertible Batch Normalization $\alpha_{Rev} \ll \alpha_{BN}$ and Leaky ReLU layers $\alpha_{Rev} \ll \alpha_{LReLU}$.

4.2.4 Pooling

In [7], the authors propose an invertible pooling operation that operates by stacking the neighboring elements of the pooling regions along the channel dimension. As noted in section 3.5, the increase in channel size at each pooling level induces a quadratic increase in the number of parameters of upstream convolution, which creates a new memory bottleneck.

To circumvent this quadratic increase in the memory cost of the weight, we propose a new pooling layer that stacks the elements of neighboring pooling regions along the batch size instead of the channel size. We refer to both kind of pooling as channel pooling $\mathcal{P}_c$ and batch pooling $\mathcal{P}_b$, respectively, depending on the dimension along which activation features are stacked. Given a 2×2 pooling region and an input activation tensor x of dimensions $bs \times c \times h \times w$, where bs refers to the batch size, c to the number of channels and $h \times w$ to the spatial resolution, the reshaping operation performed by both pooling layers can be formalized as follows:

$$\mathcal{P}_c : x \rightarrow y \tag{25a}$$

$$: \mathbb{R}^{bs \times c \times h \times w} \rightarrow \mathbb{R}^{bs \times 4c \times \frac{h}{2} \times \frac{w}{2}} \tag{25b}$$

$$\mathcal{P}_b : x \rightarrow y \tag{25c}$$

$$: \mathbb{R}^{bs \times c \times h \times w} \rightarrow \mathbb{R}^{4bs \times c \times \frac{h}{2} \times \frac{w}{2}}. \tag{25d}$$

Channel pooling gives us a way to perform volume-preserving pooling operations while increasing the number of channels at a given layer of the architecture, while batch pooling gives us a way to perform volume-preserving pooling operations while keeping the number of channel constant. By alternating between channel and batch pooling, we can control the number of channels at each pooling level of the model's architecture.

As this pooling operation only performs a reshaping between input and output, it does not induce any numerical error: $\alpha_{Pool} = 1$.

4.3 Layer-wise invertible architecture

Putting together the above building blocks, Fig. 7 illustrates a layer-wise invertible architecture. The peak memory usage for a training iteration of this architecture, as parameterized in Table 1, can be computed as follows:

$$\mathcal{M} = M_\theta + M_z + M_g \tag{26a}$$

$$= M_\theta + (M'_z + M'_g) \times (h \times w \times bs) \tag{26b}$$

$$= 29.6 \times 10^6 + 320 \times (h \times w \times bs). \tag{26c}$$

Training an iteration over a typical batch of 32 images with resolution 240×240 would require $\mathcal{M} = 590$MB of VRAM. Similar to the RevNet architecture, the reconstruction of the hidden activations by inverse transformations during the backward pass comes with an additional computational cost similar to a forward pass.

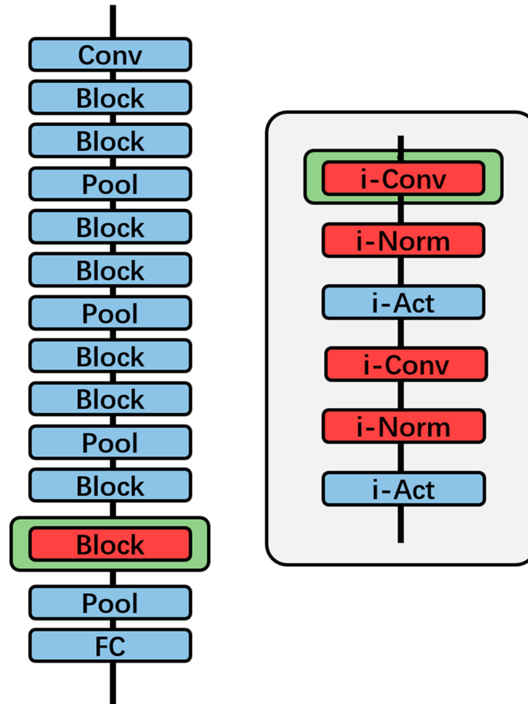


Fig. 7 Illustration of a layer-wise invertible architecture and its memory consumption

As analyzed in the previous section, the numerical errors in this architecture are dominated by Batch Normalization and Leaky ReLU layers. Following equation 19, the numerical error associated with the activations at a given layer i in this architecture can thus be approximated by:

$$\epsilon_i|^2 = \frac{|\epsilon_y|^2 \times |z_i|^2}{|y|^2} \times \prod_i^N (\alpha_{LReLU} \times \alpha_{BN}), \tag{27}$$

in which N represents the number of Batch Normalization and Leaky ReLU layers between the layer i and the output.

4.4 Hybrid architecture

In section 5.1, we saw that layer-wise activation and normalization layers degrade the signal-to-noise ratio of the reconstructed activations. In "Impact of numerical stability", we will quantify the accumulation of numerical errors through long chains of layer-wise invertible operations and show that numerical errors negatively impact model accuracy.

To prevent these numerical instabilities, we introduce a hybrid architecture, illustrated in Fig. 8, combining reversible residual blocks with layer-wise invertible functions. Conceptually, the role of the residual-level reversible block is to reconstruct the input activation of residual blocks with minimal errors, while the role of the layer-wise invertible layers is to efficiently recompute the hidden activations within the reversible residual blocks at the same time as the gradient propagates to circumvent the local memory bottleneck of the reversible module.

The backward pass through these hybrid reversible blocks is illustrated in Fig. 9 and proceeds as follows: first, the input x is computed from the output y through

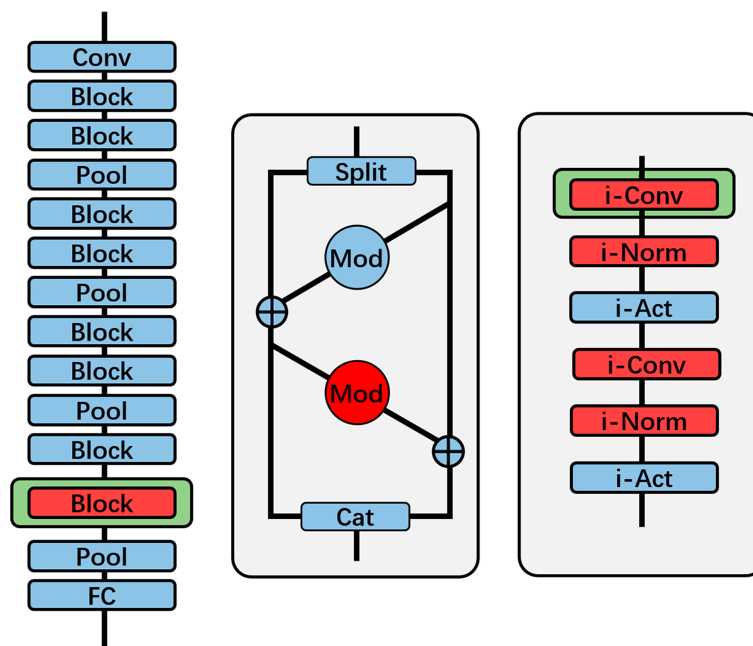


Fig. 8 Illustration of a hybrid architecture and its peak memory consumption

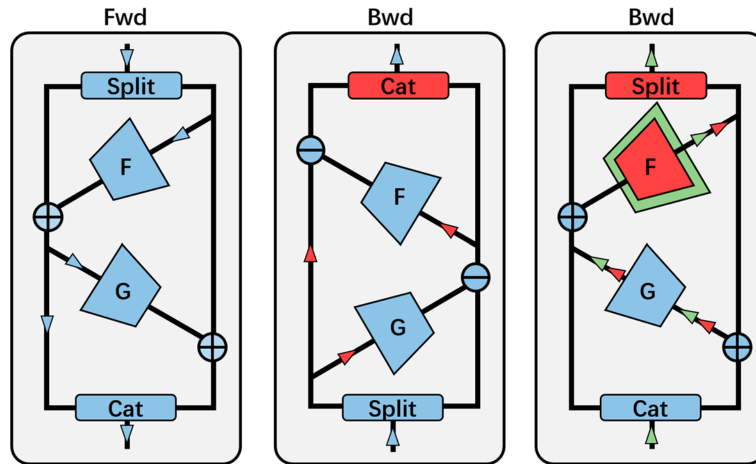


Fig. 9 Illustration of the backpropagation process through a reversible block of our proposed hybrid architecture. In the forward pass (left), activations are propagated forward from top to bottom. The activations are not kept in live memory as they are to be recomputed in the backward pass so that no memory bottleneck occurs. The backward pass is made of two phases: first the input activations are recomputed from the output using the reversible block analytical inverse (middle). This step allows to reconstruct the input activations with minimal reconstruction error. During this step, hidden activations are not kept in live memory so as to avoid the local memory bottleneck of the reversible block. Once the input activation recomputed, the gradients are propagated backward through both modules of the reversible blocks (right). During this second phase, hidden activations are recomputed backward through each module using the layer-wise inverse operations, yielding minimal memory footprint

the analytical inverse of the reversible block. These computations are made without storing the hidden activation values of the sub-modules. Second, the gradient of the activations are propagated backward through the reversible of the block modules. As each layer within these modules is invertible, the hidden activation values are computed using the layer-wise inverse along the gradient.

The analytical inverse of the residual-level reversible blocks is used to propagate hidden activations with minimal reconstruction error to the lower modules, while layer-wise inversion allows us to alleviate the local bottleneck of the reversible block by computing the hidden activation values together with the backward flow of the gradients. As layer-wise inverses are only used for hidden feature computations within the scope of the reversible block, and reversible blocks are made of relatively short chains of operations, numerical errors do not accumulate up to a damaging degree.

The peak memory consumption of our proposed architecture, as illustrated in Fig. 8 and parameterized in Table 1, can be computed as:

$$\mathcal{M} = M_\theta + M_z + M_g \tag{28a}$$

$$= M_\theta + (M_z' + M_g') \times (h \times w \times bs) \tag{28b}$$

$$= 14.8 \times 10^6 + 352 \times (h \times w \times bs). \tag{28c}$$

Training an iteration over batch of 32 images of resolution 240×240 would require $\mathcal{M} = 648$MB of VRAM.

It should be noted, however, that this architecture adds an extra computational cost as both the reversible block inverse and layer-wise inverse need to be computed. Hence, instead of one additional forward pass, as in the RevNet and layer-wise architectures, our hybrid architecture comes with a computational cost equivalent to performing two additional forward passes during the backward pass.

Following equation 19, the numerical error associated with the activations at a given layer i in this architecture are given by:

$$\epsilon_i|^2 = \frac{|\epsilon_y|^2 \times |z_i|^2}{|y|^2} \times \prod_i^K \alpha_{Rev}, \tag{29}$$

in which K represents the number of reversible blocks between the layer i and the output.

Comparing this equation to equation 27, the stability of this architecture is due to the following two factors: first the number of reversible blocks K is typically two to three times smaller than the number of layers N as a reversible block is typically made of several convolutions: $K < N$. Second, and most importantly, the SNR reduction factor is much smaller in reversible blocks than in Batch Normalization $\alpha_{Rev} \ll \alpha_{BN}$ and Leaky ReLU layers $\alpha_{Rev} \ll \alpha_{LReLU}$.

5 Results and discussion

This section is organized in two parts: first, we start by analyzing numerical errors arising in both the layer-wise invertible and hybrid architectures. Second, we compare our hybrid architecture to existing models.

All our experiments use the CIFAR10 dataset as a benchmark. The CIFAR10 dataset is complex enough to require efficient architectures to reach high accuracy, yet small enough to enable us to rapidly iterate over different architectural designs.

Unless stated otherwise, all models were trained for 50 epochs of stochastic gradient descent with cyclical learning rate and momentum [23] with minimal image augmentation. The code used to produce the results is available at the link given in the “Availability of data and materials" section.

5.1 Impact of numerical stability

The idea of layer-wise invertibility is attractive for it maximally reduces the memory footprint of CNN training by bypassing the local bottleneck of architectures based on reversible blocks (i.e., RevNet or iRevNet). Unfortunately, we will show in this section that deep architectures based only on layer-wise invertibility cannot be successfully trained due to numerical errors preventing the model from converging to high quality solutions. Instead, layer-wise invertibility can be combined with reversible block-level invertibility to get the best of both world: reversible blocks allow for long chain of reconstruction without numerical errors reaching critical values, while layer-wise invertibility is used within reversible blocks to bypass the local memory bottleneck.

Figure 10 shows the inverse reconstruction error for each layer of both architectures, in order to visualize these phenomenon. This figure suggests that layer-wise invertible architecture cannot scale with depth as numerical errors accumulate with depth. On

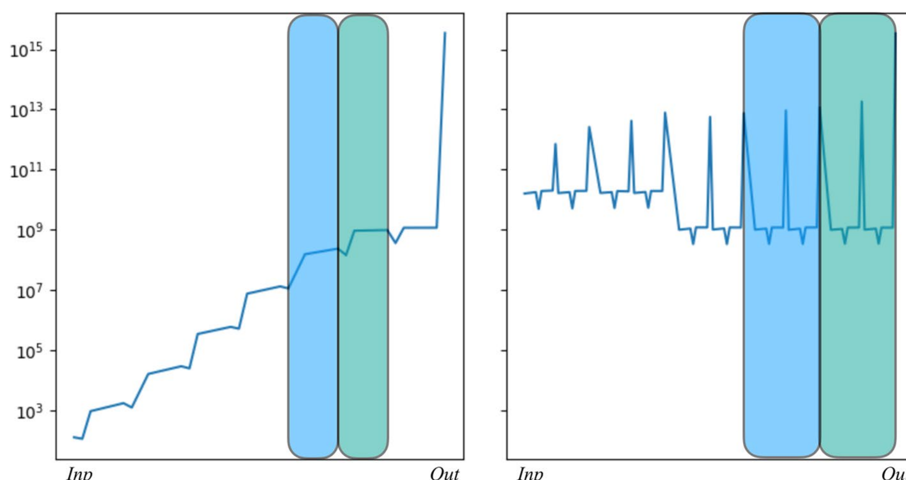


Fig. 10 Evolution of the SNR through the layers of a (left) layer-wise invertible model and (right) hybrid architecture model. The lower the SNR is, the more important numerical errors of the inverse reconstructions are. The x axis corresponds to layer indices of the model: right-most values represent the top layer of the model, in which the least noise is observed. Left-most values represent input layers in which maximum levels of noise accumulate. (Left): color boxes illustrate the span of two consecutive convolutional blocks (convolution–normalization–activation layers). The SNR gets continuously degraded throughout each block of the network, resulting in numerical instabilities. (Right): color boxes illustrate consecutive reversible blocks. Within reversible blocks, the SNR quickly degrades due to the numerical errors introduced by invertible layers. However, the signal propagated to the input of each reversible block is recomputed using the reversible block inverse, which is much more stable. Hence, we can see a sharp decline of the SNR within the reversible blocks, but the SNR almost raises back to its original level at the input of each reversible block

the other hand, in the case of the hybrid architecture, one can see that numerical errors accumulate within reversible blocks, but that the long-term trend of the SNR is stable due to the stable inverse operation of the reversible blocks.

Figure 11 quantifies the degradation of the inverse reconstruction for the two models. We found the two most impacting parameters to be the depth N of the network and the negative slope n of the activation function, so we show the evolution of the reconstruction errors when varying both parameters.

Finally, we investigate the impact of numerical errors on the accuracy. In order to isolate the impact of the numerical errors, we compare the accuracy reached by the same architectures with and without inverse reconstruction of the hidden layers activations. Without reconstruction, the hidden activation values are stored in memory along the forward pass, and the gradient updates are computed from the true, noiseless activation values. With inverse reconstructions, activation values are recovered by inverse operators during the backward pass. Hence, the only difference between both settings is the noise introduced by the inverse reconstructions. In Fig. 12, we show evolution of the accuracy with increasing depth.

In the case of the layer-wise invertible architecture: For small depths (or high negative slopes), in which the numerical errors are minimum, both models yield similar accuracy. However, as the numerical errors grow, the accuracy of the model goes down, while the accuracy of the ideal baseline keeps increasing, which can be seen with both depth and negative slopes. This loss in accuracy is the direct result of numerical errors, which prevent the model from converging to higher accuracies.

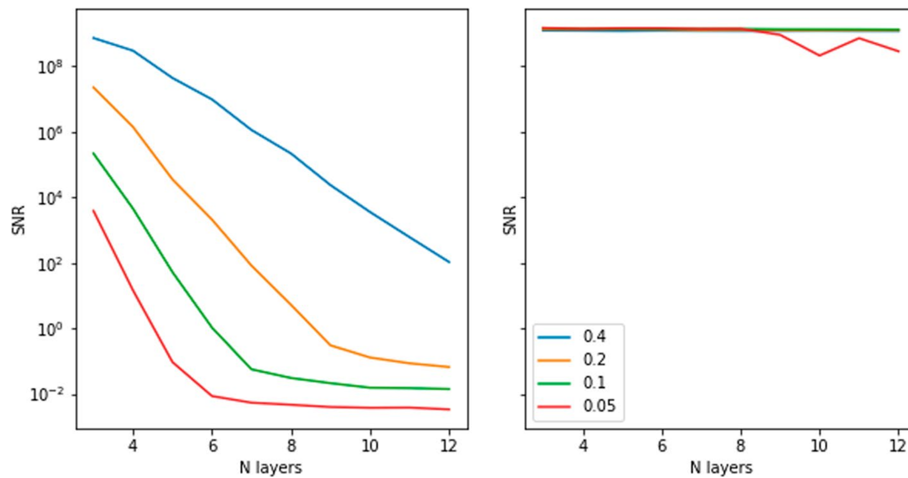


Fig. 11 Illustration of the impact of depth (in number of layers N) and negative slope n on the numerical errors of (left) the layer-wise invertible architecture and (right) the hybrid architecture. Both figures show the evolution of the SNR at the input layer of the network for increasing depth N on the x axis, and with different negative slopes n in different colors. (Left): the SNR decreases with depth until it reaches an SNR value of 1. At this point, the noise is of the same scale as the signal, and no learning can happen. It is impressive that with only four layers of depth, a negative slope of $n = 0.005$ reaches a SNR of 1. With such parameterization, even the most shallow models are not capable of learning. (Right) The hybrid architecture successfully stabilizes the numerical error propagation

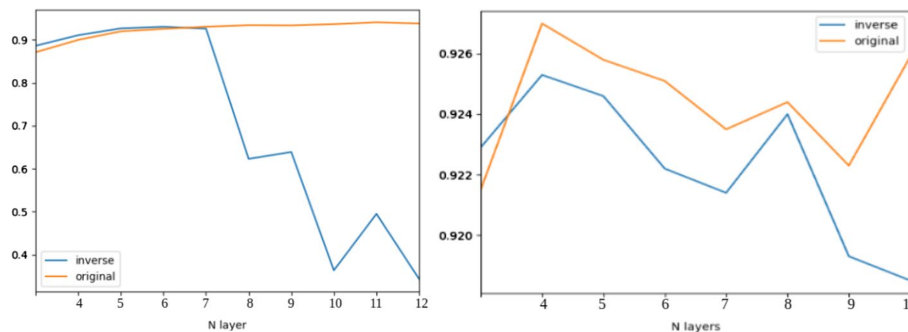


Fig. 12 Impact of the numerical errors on the accuracy of (left) layer-wise invertible models and (right) hybrid architecture model. (Left): evolution of the accuracy with depth for a negative slope $n = 0.2$ with and without inverse reconstructions. Without reconstruction, the model accuracy benefits from depth. With inverse reconstructions, the model similarly benefits from depth as the number of layers grow from 3 to 7. For $N > 7$, however, the accuracy sharply decreases toward lower values due to numerical errors. (Right): our proposed hybrid architecture greatly stabilizes the numerical errors, which results in smaller effects of the depth and negative slope on accuracy

Table 1 Summary of architectures with different levels of reversibility

Model	Accuracy	#Params	Channels	Pooling	M_θ	$M'_z + M'_g$	$\mathcal{M}$
Resnet	94.7%	3.1M	$32-64-128-256$	Max Pooling	12.5M	1928	1.01G
RevNet	94.5%	3.1M	$40-80-256-320$	Max Pooling	12.7M	640	348M
i-RevNet	93.8%	42.8M	$32-128-512-2048$	$\mathcal{P}_c-\mathcal{P}_c-\mathcal{P}_c$	171M	640	500M
RevNeXt (ours)	93.3%	3.7M	$32-128-512-512$	$[\mathcal{P}_c, \mathcal{P}_c, \mathcal{P}_b]$	14.8M	352	200M

In the case of the hybrid architecture, the negative impacts of numerical errors observed in the layer-wise architecture are gone, confirming that the numerical stability brought by the hybrid architecture effectively stabilizes training.

5.2 Model comparison

Table 1 compares architectures with different patterns of reversibility. We called our model RevNeXt as a reference to both prior works and the eXtreme memory reduction of the RevNet architecture aimed by our work. The exact parameterization of our proposed RevNeXt is given together with other architectures in Table 1.

To allow for a fair comparison, we have tweaked each architecture to keep the number of parameters as close as possible, with the notable exception of the i-RevNet architecture. The i-Revnet pooling scheme enforces a quadratic growth of its parameters with each level of pooling. In order to keep the number of parameters of the i-RevNet close to the other baselines, we would have to drastically reduce the number of channels of lower layers, which we found yield poor performance. Furthermore, it should be noted that the i-RevNet architecture we present slightly differs from the original i-Revnet model as our implementation uses RevNet-like reversible modules with one module per channel split for similarity with the other architecture we evaluate instead of the single module used in the original architecture.

Our model drastically cuts the memory cost of training, which comes at the cost of both a small degradation in accuracy, and additional computations. The additional computation requirements remain manageable though: Our hybrid architecture requires the computational equivalent of two additional forward passes within each backward pass.

As an illustration of applications enabled by our model, Table 2, we compare the time of training our proposed architecture to 93.3% on a high-end Nvidia GTX 1080Ti and a low-end Nvidia GTX750. The GTX750 only has 1GB of VRAM, which results in roughly 400MB of available memory after the initialization of various frameworks. Training a vanilla ResNet with large batch sizes on such limited memory resources is impractical, while our architecture allows for efficient training.

6 Conclusion

Convolutional Neural Networks form the backbone of modern computer vision systems. However, the accuracy of these models comes at the cost of resource-intensive training and inference procedures. While tremendous efforts have been put into the optimization of the inference step on resource-limited device, relatively little work have focused on algorithmic solutions for limited resource training. In this paper, we have presented an architecture able to yield high accuracy classifications within very tight memory constraints. We highlighted several potential applications of memory-efficient training procedures, such as on-device

Table 2 Training statistics on different hardware

GPU	Accuracy	Time
GTX750	93.3%	67 min
GTX 1080Ti	93.3%	35 min

training, and illustrated the efficiency of our approach by training a CNN to 93.3% accuracy on a low-end GPU with only 1GB of memory.

Appendix

Proof of batch normalization results

To illustrate the mechanism through which the batch normalization inverse operation reduces the SNR, let us consider a toy layer with only two channels and parameters $\beta = [0, 0]$ and $\gamma = [1, \rho]$. For simplicity, let us consider an input signal x independently and identically distributed across both channels with zero mean and standard deviation 1 so that, in the forward pass, we have:

$$y = [y_0, y_1] \tag{30a}$$

$$= [x_0, x_1 \times \rho], \tag{30b}$$

$$|y|^2 = |x_0|^2 + |x_1|^2 \times \rho^2 \tag{30c}$$

$$= \frac{1}{2} \times |x|^2 + \frac{1}{2} \times |x|^2 \times \rho^2 \tag{30d}$$

$$= \frac{|x|^2}{2} \times (1 + \rho^2), \tag{30e}$$

in which we used the assumption that x is independently and identically distributed across both channels to factorize $|x_0|^2 = |x_1|^2 = \frac{1}{2} \times |x|^2$ in Eq. (17ad).

During the backward pass, the noisy estimate $\tilde{y} = y + \epsilon^y$ is fed back as input to the inverse operation. Similarly, let us suppose a noise ϵ^y identically distributed across both channels so that we have:

$$\tilde{y} = [\tilde{y}_0, \tilde{y}_1] \tag{31a}$$

$$= [x_0 + \epsilon^y_0, x_1 \times \rho + \epsilon^y_1], \tag{31b}$$

$$\tilde{x} = [\tilde{y}_0, \frac{\tilde{y}_1}{\rho}] \tag{31c}$$

$$= [x_0 + \epsilon^y_0, x_1 + \frac{\epsilon^y_1}{\rho}], \tag{31d}$$

$$\epsilon^x = \tilde{x} - x \tag{31e}$$

$$=[\epsilon_0^y, \frac{\epsilon_1^y}{\rho}] \tag{31f}$$

$$|\epsilon^x|^2 =|\epsilon_0^y|^2 + \frac{|\epsilon_1^y|^2}{\rho^2}, \tag{31g}$$

$$=\frac{1}{2} \times |\epsilon^y|^2 + \frac{1}{2} \times \frac{|\epsilon^y|^2}{\rho^2} \tag{31h}$$

$$=\frac{|\epsilon^y|^2}{2} \times (1 + \frac{1}{\rho^2}). \tag{31i}$$

Using the above formulation, the SNR reduction factor α can be expressed as:

$$\alpha =\frac{snr_i}{snr_o} \tag{32a}$$

$$=\frac{|x|^2}{|\epsilon^x|^2} \times \frac{|\epsilon^y|^2}{|y|^2} \tag{32b}$$

$$=\frac{4}{(1 + \frac{1}{\rho^2}) \times (1 + \rho^2)}. \tag{32c}$$

In essence, numerical instabilities in the inverse computation of the batch normalization layer arise from the fact that the signal across different channels i and j are amplified by different factors γ_i and γ_j. While the signal amplification in the forward and inverse path cancel out each other ($x = f^{-1}(f(x))$), the noise only gets amplified in the backward pass.

In the above demonstration, we have used a toy parameterization of the invertible batch normalization layer to illustrate the mechanism behind the SNR degradation. For arbitrarily parameterized batch normalization layers, the SNR degradation factor becomes:

$$\alpha =\frac{snr_i}{snr_o} \tag{33a}$$

$$=\frac{|x|^2}{|\epsilon^x|^2} \times \frac{|\epsilon^y|^2}{|y|^2} \tag{33b}$$

$$=\frac{|x|^2}{|y|^2} \times \frac{|\epsilon^y|^2}{|\epsilon^x|^2}. \tag{33c}$$

Assuming a noise ϵ^y, equally distributed across all channels, the noise ratio can be computed as follows:

$$\tilde{y}_i = \gamma_i \times \frac{x_i - \hat{x}_i}{\sqrt{\dot{x}_i} + \epsilon} + \beta_i + \epsilon_i^y, \tag{34a}$$

$$\tilde{x}_i = (\sqrt{\dot{x}_i} + \epsilon) \times \frac{\tilde{y}_i - \beta_i}{\gamma_i} + \hat{x}_i \tag{34b}$$

$$= x_i + \frac{\sqrt{\dot{x}_i} + \epsilon}{\gamma_i} \times \epsilon_i^y, \tag{34c}$$

$$\epsilon_i^x = \tilde{x}_i - x_i \tag{34d}$$

$$= \frac{\sqrt{\dot{x}_i} + \epsilon}{\gamma_i} \times \epsilon_i^y, \tag{34e}$$

$$\frac{|\epsilon^y|^2}{|\epsilon^x|^2} = \frac{|\epsilon^y|^2}{\frac{|\epsilon^y|^2}{c} \times \sum_i \frac{\dot{x}_i^2}{\gamma_i^2}} \tag{34f}$$

$$= \frac{c}{\sum_i \frac{\sqrt{\dot{x}_i} + \epsilon}{\gamma_i}}. \tag{34g}$$

Assuming input x following a Gaussian distribution with channel-wise mean $\hat{x}_i$ and variance $\dot{x}_i$, the SNR reduction factor α becomes:

$$\frac{|x|^2}{|y|^2} = \frac{\sum_i |x_i|^2}{\sum_i |y_i|^2} \tag{35a}$$

$$= \frac{\sum_i (\hat{x}_i^2 + \dot{x}_i)}{\sum_i (\gamma_i^2 + \beta_i^2)}, \tag{35b}$$

$$\alpha = \frac{|x|^2}{|y|^2} \times \frac{|\epsilon^y|^2}{|\epsilon^x|^2} \tag{35c}$$

$$= \frac{\sum_i (\hat{x}_i^2 + \dot{x}_i)}{\sum_i (\gamma_i^2 + \beta_i^2)} \times \frac{c}{\sum_i \frac{\sqrt{\dot{x}_i} + \epsilon}{\gamma_i}}. \tag{35d}$$

Proof of activation function results

The analysis of the numerical errors yielded by the invertible Leaky ReLU follows a similar reasoning as the toy batch normalization example with an additional subtlety: Similar to the toy batch normalization example, we can think of the leaky ReLU as artificially splitting the input x across two different channels, one channel leaving the output unchanged and one channel that divides the input by a factor n during the forward pass and multiplies its output by a factor n during the backward pass.

However, these artificial channels are defined by the sign of the input and output during the forward and backward pass, respectively. Hence, we need to consider the cases in which the noise flips the sign of the output activations, which leads to different behaviors of the invertible Leaky ReLU across four cases:

$$y = \begin{cases} y_{nn} \text{ if } \hat{y} < 0 \quad \text{and } y < 0 \\ y_{np} \text{ if } \hat{y} >= 0 \text{ and } y < 0 \\ y_{pp} \text{ if } \hat{y} >= 0 \text{ and } y >= 0 \\ y_{pn} \text{ if } \hat{y} < 0 \quad \text{and } y >= 1 \end{cases}, \tag{36a}$$

where the index np, for instance, represents negative activations whose reconstructions have become positive due to the added noise. The signal-to-noise ratio of the input and outputs can be expressed, respectively, as:

In the case where $y \gg \epsilon_y$, the probability of sign flips (y_{np}, y_{pn}) is negligible, so that the output signal y is evenly split along y_{pp} and y_{nn}. In this regime, the degradation of the SNR obeys a formula similar to the toy batch normalization example:

$$y = [y_{pp}, y_{nn}] \tag{37a}$$

$$= [x_{pp}, \frac{x_{nn}}{n}], \tag{37b}$$

$$|y|^2 = \frac{1}{2} \times |x|^2 + \frac{1}{2} \times \frac{|x|^2}{n^2} \tag{37c}$$

$$= \frac{|x|^2}{2} \times (1 + \frac{1}{n^2}). \tag{37d}$$

$$\tilde{y} = [\tilde{y}_{pp}, \tilde{y}_{nn}] \tag{38a}$$

$$= [x_{pp} + \epsilon^y_{pp}, \frac{x_{nn}}{n} + \epsilon^y_{nn}], \tag{38b}$$

$$\tilde{x} = [\tilde{y}_{pp}, \tilde{y}_{nn} \times n] \tag{38c}$$

$$= [x_{pp} + \epsilon^y_{pp}, x_{nn} + \epsilon^y_{nn} \times n], \tag{38d}$$

$$\epsilon^x = \tilde{x} - x \tag{38e}$$

$$= [\epsilon^y_{pp}, \epsilon^y_{nn} \times n], \tag{38f}$$

$$|\epsilon^x|^2 = \frac{1}{2} \times |\epsilon^y|^2 + \frac{1}{2} \times |\epsilon^y|^2 \times n^2 \tag{38g}$$

$$=\frac{|\epsilon^y|^2}{2} \times (1+n^2). \tag{38h}$$

Using the above formulation, the signal-to-noise ratio reduction factor α can be expressed as:

$$\alpha = \frac{snr_i}{snr_o} \tag{39a}$$

$$=\frac{|x|^2}{|\epsilon^x|^2} \times \frac{|\epsilon^y|^2}{|y|^2} \tag{39b}$$

$$=\frac{4}{(1+\frac{1}{n^2}) \times (1+n^2)}. \tag{39c}$$

When the noise reaches an amplitude similar to or greater than the activation signal, the effects of sign flips complicate the equation. However, in this regime, the signal-to-noise ratio becomes too low for training to converge, as numerical errors prevent any useful weight update, so we leave the problem of characterizing this regime open.

Abbreviations

CNN	Convolutional Neural Network
SNR	Signal-to-noise ratio
GPU	Graphical Processing Unit
ResNet	Residual network
RevNet	Reversible network

Acknowledgements
At the time of this writing, the only contributors to the content of this work are the authors. We would be happy to thank reviewers for useful feedback.

Author contributions
T.H and Q.F equally contributed to the investigation and implementation presented in this work. T.T and Y.A provided advice on the research, and W.Z provided figures and modifications as suggested by the reviewers. All authors have read and approved the final version of the manuscript.

Funding
This work was supported by a scholarship MEXT from the Japanese Ministry of Education, Culture, Sports, Science, and Technology. A part of this study is subsidized by JSPS Grant-in-Aid for Scientific Research and Research granted JP 17K00236. A part of this study is subsidized by JSPS Grant-in-Aid for Scientific Research and Research granted JP 19K24344 and JP 20K19823. This work was supported in part by PRESTO, JST (Grant No. JPMJPR15D2).

Availability of data and materials
The data used in this work are publicly available online. The code used for the experiments is available on GitHub at the following address: https://github.com/TristHas/reversibility_paper

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 14 May 2019 Accepted: 29 November 2022
Published online: 05 January 2023

References

1. B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2704–2713 (2018)

2. P. Molchanov, S. Tyree, T. Karras, T. Aila, J. Kautz, Pruning convolutional neural networks for resource efficient inference. arXiv preprint arXiv:1611.06440 (2016)
3. D.P. Kingma, J. Ba, Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
4. C.J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, G.E. Dahl, Measuring the effects of data parallelism on neural network training. arXiv preprint arXiv:1811.03600 (2018)
5. S. McCandlish, J. Kaplan, D. Amodei, O. Dota Team, An empirical model of large-batch training. arXiv preprint arXiv:1812.06162 (2018)
6. A.N. Gomez, M. Ren, R. Urtasun, R.B. Grosse, The reversible residual network: backpropagation without storing activations. In: *Advances in Neural Information Processing Systems*, pp. 2214–2224 (2017)
7. J.-H. Jacobsen, A. Smeulders, E. Oyallon, i-revnet: Deep invertible networks. arXiv preprint arXiv:1802.07088 (2018)
8. L. Dinh, D. Krueger, Y. Bengio, Nice: Non-linear independent components estimation. arXiv preprint arXiv:1410.8516 (2014)
9. L. Dinh, J. Sohl-Dickstein, S. Bengio, Density estimation using real nvp. arXiv preprint arXiv:1605.08803 (2016)
10. D.P. Kingma, P. Dhariwal, Glow: Generative flow with invertible 1x1 convolutions. In: Advances in Neural Information Processing Systems, pp. 10215–10224 (2018)
11. D.J. Rezende, S. Mohamed, Variational inference with normalizing flows. arXiv preprint arXiv:1505.05770 (2015)
12. T.Q. Chen, Y. Rubanova, J. Bettencourt, D.K. Duvenaud, Neural ordinary differential equations. In: Advances in Neural Information Processing Systems, pp. 6571–6583 (2018)
13. J.-H. Jacobsen, J. Behrmann, R. Zemel, M. Bethge, Excessive invariance causes adversarial vulnerability. arXiv preprint arXiv:1811.00401 (2018)
14. J. Behrmann, D., Duvenaud, J.-H. Jacobsen, Invertible residual networks. arXiv preprint arXiv:1811.00995 (2018)
15. S. Rota Bulò, L. Porzi, P. Kontschieder, In-place activated batchnorm for memory-optimized training of dnns. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5639–5647 (2018)
16. F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. arXiv preprint arXiv:1602.07360 (2016)
17. A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
18. J. Frankle, M. Carbin, The lottery ticket hypothesis: Finding sparse, trainable neural networks. arXiv preprint arXiv:1803.03635 (2018)
19. P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2017)
20. S. Wu, G. Li, F. Chen, L. Shi, Training and inference with integers in deep neural networks. arXiv preprint arXiv:1802.04680 (2018)
21. J. Martens, I. Sutskever, Training deep and recurrent networks with hessian-free optimization. In: *Neural Networks: Tricks of the Trade*, pp. 479–535. Springer, (2012)
22. T. Chen, B. Xu, C. Zhang, C. Guestrin, Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016)
23. L.N. Smith, N. Topin, Super-convergence: Very fast training of neural networks using large learning rates. arXiv preprint arXiv:1708.07120 (2017)

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com