RESEARCH

Reversible designs for extreme memory cost reduction of CNN training

Tristan Hascoet[????], Quentin Febvre[??*], Yasuo Ariki[??,??] and Tetsuya Takiguchi[??,??]

Abstract

Training Convolutional Neural Networks (CNN) is a resource intensive task that requires specialized hardware for efficient computation. One of the most limiting bottleneck of CNN training is the memory cost associated with storing the activation values of hidden layers 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 analyse their affect 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 XXX 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 device or non-specialized hardware. For instance, we demonstrate training of our model to 94% accuracy on the CIFAR10 dataset within XXX minutes on a low-end Nvidia GTX750 GPU with only 1GB of memory.

Keywords: CNN; Reversibility; Memory optimization; Numerical analysis

1 Introduction

Convolutional Neural Networks (CNN) have enabled unprecedent progress on a wide array of computer vision task over the last few years. 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 [], network pruning for inference [], or efficient stochastic optimization algorithms [].

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 CNN 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 to train deep nets 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 (XXX% accuracy within XXX minutes) on an Nvidia GTX750 with only 1GB 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 delegated to large cloud servers, and trained CNN are typically deployed to embedded device fleets over the network. On-device training would

*Correspondence: XXX
??Quentin's job, xxx, XXX XXX, XXX
Full list of author information is available at the end of the article

allow to bypass 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 environment 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 authentification, 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 []. For example, in Imagenet, linear speed-ups in training have been observed with increasing batch sizes up to tens of thousands of samples []. Optimizing the memory cost of CNN training may allow further research on the optimization tradeoffs of large batch training. For small datasets like MNIST or CIFAR10, we are able to process the full dataset in XXX memory. Although large batch training on such small dataset is very computationally inefficient with current stochastic optimization algorithms [], 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 [] 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 analyse 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 architectures. However, we observe that layer-wise invertible networks accumulate numerical errors accross 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 affect 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 combiningthe 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 minimal memory cost of XXXbytes per pixel. Using a large batch size of XXX, we are able to reach XXX% accuracy on the CIFAR10 dataset within XXX minutes using only XXX of memory.

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 paritioning schemes (XXX [] Real-NVP []), weight matrix factorization (GLOW []) or constraining layer architectures to easily invertible unitary operations (Normalization flows [])

Neural ODEs take a drastically different take on invertibility: They leverage the analogy between residual networks and the Euler method to define continuous hidden states 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 [] 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 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 learned representations rather than resource efficiency. From a resource optimization point of view, one downside of their method is that the proposed invertible pooling increases the number of channels. As the size of the convolution kernel weights grows quadratacally 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 XXX, the authors use these reversible architectures to study undesirable invariances in feature space.

In XXX, the authors propose a unified architecture performing well on both generative and discriminative tasks. They enforce inversibility 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 perfored with power iteration methods which is not optimal from a computational perspective.

Finally, XXX 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 cover 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 was first proposed as an efficient neural architecture reducing the number of model parameters while maintining high classification accuracy. MobileNet use depth-wise separable convolutions to further reduce the computational cost of inference for embedded device applications.

Network pruning is a set of techniques [] developped 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 computation time of inference without significantly hurting model accuracy. Although pruning has been concerned with optimization of the resource inference, the recently proposed lotery tickect hypothesis [] has shown than 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 [] combines float16 with float32 operations to avoid numerical instabilities due to either overflow or underflow. For inference, integer quantization [] has been shown to drastically improve the computation and memory efficiency and has been successfully deployed on both edge devices and data centers.

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 [], 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. Several works [] have 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 desgins and analyse their impact on computation and memory consumption.

3.1 Backpropagation & 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 $\bar{y}$:

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

$$F : x \rightarrow f_N \circ ... \circ f_2 \circ f_1(x) \tag{1b}$$

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

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:

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

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

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 function with respect to each layer's parameters requires knowledge of the input activation values z_i. In the standard backpropagation algorithm, hidden layers activations are stored 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, the hidden activation buffers can be freed from live memory. However, the accumulation of activation values stored within each parameterized layer along the depth of the network during the forward pass creates a major bottleneck in GPU memory, which constrains both input and model sizes.

The idea behind reversible designs is to constrain the network architecture to feaure 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 groundtruth 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 of storing the model weights, the hidden activations, and the gradients:

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

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 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. For float32 tensors, $bpe = 4$. We use bs to denote the batch size, and c_i to denote the number of channels at layer i.

3.3 Vanilla ResNet

The architecture of a vanilla ResNet-18 is shown in Figure 1. Vanilla ResNet do not use reversible computations so that the input activations of all parameterized layers need to 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 $P \subset N$ be the subset of parameterized layers of the F (i.e.; convolutions and batch normalization layers, excluding activation functions and pooling layers). Then the memory cost associated with storing the hidden activation values is:

$$M_z = \sum_{i \in P} n(z_i) \times bpe \tag{6a}$$

$$M_z = \sum_{i \in P} bs \times c_i \times h_i \times w_i \times bpe \tag{6b}$$

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 a memory cost per input pixel M'_z:

$$M_z = \times \sum_{i \in P} bs \times h \times w \times p_i \times c_i \times bpe \tag{7a}$$

$$M_z = bs \times h \times w \times \sum_{i \in P} p_i \times c_i \times bpe \tag{7b}$$

$$M'_z = M_z / (bs \times h \times w) \tag{7c}$$

$$M'_z = \sum_{i \in P} p_i \times c_i \times bpe \tag{7d}$$

The memory footprint of the weights is given by:

$$M_\theta = \sum_{i \in P} n(\theta_i) \times bpe \tag{8a}$$

$$M_\theta = \sum_{i \in P} c_i \times c_{i-1} \times k^2 \times bpe \tag{8b}$$

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 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, which also depends on the input pixel size:

$$M_g = max(n(g_{i-1}), n(g_i)) \times bpe \tag{9a}$$

$$M_g = h \times w \times bs \times p_i \times max(c_{i-1}, c_i) \tag{9b}$$

$$M'_g = p_i \times max(c_{i-1}, c_i) \tag{9c}$$

Figure XXX illustrates the peak memory consumption of the ResNet-18 architecture. This peak memory consumption can then be summarized as:

$$\mathcal{M} = M_\theta + M_z + M_g \tag{10a}$$

$$\mathcal{M} = (M_\theta + (M'_z + M'_g) \times (h \times w \times bs)) \times bpe \tag{10b}$$

For example, a training iteration of batch of 512 images of resolution 32×32 requires $\mathcal{M} = XXX$GB of VRAM.

3.4 RevNet

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

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

The local memory bottleneck has to do with the synchronization of the reversible block computations: Because the flow of hidden activation reconstruction does not follow the flow of the gradient computations, the gradient computation can not be performed simultaneously to the gradient computation, which would allow for efficient memory buffer deallocation. Figure XXX 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.

The peak memory consumption in the RevNet architecture, as illustrated in Figure XXX, happens in the backward pass through the last reversible block in which the memory cost of storing the full set of non-preserving layers input activations adds to the local memory cost of storing the hidden activations of the reversible block.

The peak memory consumption can be computed as:

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

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

$$M'_z = M_{local} + M_{global} \tag{11c}$$

Following our previous example, a RevNet architecture closely mimicking the ResNet-18 architecture would requires $\mathcal{M} = XXX$GB of VRAM for a training iteration over batch of 512 images of resolution 32×32.

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:

$$M(\theta) = c_{in} \times c_{out} \times k_h \times k_w \tag{12a}$$
$$M(\theta) = c^2 \times k_h \times k_w \tag{12b}$$

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 $n(\theta) = 2048 \times 3 \times 3 \times 2048 = 37M$ parameters. At this point, the memory cost of the network weights M_θ may outweight the activation memory cost and becomes a new bottleneck. However, these pooling layers introduces a new memory bottleneck through the weight memory cost. Furthermore, the iRevNet architecture does not address the local memory bottleneck of the reversible blocks.

Figure XXX illustrates such architecture, for which the peak memory consumption is given by:

$$M(\theta) = c_{in} \times c_{out} \times k_h \times k_w \tag{13a}$$
$$M(\theta) = c^2 \times k_h \times k_w \tag{13b}$$

For an input channel size of 32, training an iteration over batch of 512 images of resolution 32×32 would requires $\mathcal{M} = XXX$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 create a local memory bottleneck as all hidden activations within a reversible block need to be computed before the gradients are back-propagated through the block. In order to circumvent this local bottleneck, we introduce layer-wise invertible operations. However, these invertible operations introduce numerical error, which we characterize in the following subsection. In Section XXX, we will show that these numerical errors lead to instabilitites that degrade the model accuracy. Hence, in section XXX, 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.

4.1 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.

4.1.1 Invertible batch normalization

As batch normalization is not a bijective operation, it does not a have 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:

$$y = f(x) = \gamma \times \frac{x - \hat{x}}{\sqrt{\dot{x}} + \epsilon} + \beta \tag{14a}$$
$$x = f^{-1}(y, \hat{x}, \dot{x}) = (\sqrt{\dot{x}} + \epsilon) \times \frac{y - \beta}{\gamma} + \hat{x} \tag{14b}$$

Where $\hat{x}$ and $\dot{x}$ represent the mean and standard deviation 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 $\dot{x}$.

Let us consider the accumulation of numerical errors arising from the inverse computation of an invertible batch normalization layer. During the backward pass, the invertible batch norm layer is supposed to compute its input $x = f^{-1}(y)$ from the ouput y. In reality, however, the output propagated back to the layer by upstream invertible layers is a noisy estimate $\hat{y} = y + \epsilon_y$ of the true output due to the numerical errors introduced by upstream layers. Let us define the signal to noise ratio (SNR) of the input and output signal as follows:

$$snr_o = \frac{|y|^2}{|\epsilon_y|^2} \tag{15a}$$
$$snr_i = \frac{|x|^2}{|\epsilon_x|^2} \tag{15b}$$

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

$$\alpha = \frac{snr_i}{snr_o} \tag{16}$$

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 independantly and indentically distributed across both channels with zero mean and standard deviation 1 so that, in the forward pass, we have:

$$y = [y_0, y_1] \tag{17a}$$
$$y = [x_0, x_1 \times \rho] \tag{17b}$$
$$|y|^2 = \frac{1}{2} \times |x|^2 + \frac{1}{2} \times |x|^2 \times \rho^2 \tag{17c}$$
$$|y|^2 = |x|^2 \times \frac{1+\rho^2}{2} \tag{17d}$$

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

$$\hat{y} = [x_0 + \epsilon_{y0}, x_1 \times \rho + \epsilon_{y1}] \tag{18a}$$
$$\hat{x} = [\hat{y}, \hat{y}/\rho] \tag{18b}$$
$$\hat{x} = [x_0 + \epsilon_{y0}, x_1 + \epsilon_{y1} \times \rho] \tag{18c}$$
$$\epsilon_x = \hat{x} - x \tag{18d}$$
$$\epsilon_x = [\epsilon_{y0}, \epsilon_{y1} \times \rho] \tag{18e}$$
$$|\epsilon_x|^2 = [\epsilon_{y0}|, \epsilon_{y1} \times \rho] \tag{18f}$$

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

$$\alpha = \frac{snr_i}{snr_o} \tag{19a}$$
$$\alpha = \frac{|x|^2}{|\epsilon_x|^2} \times \frac{|\epsilon_y|^2}{|y|^2} \tag{19b}$$
$$\alpha = \frac{1}{\frac{1}{\rho^2} \times (1 \times \rho^2} \tag{19c}$$

In essence, numerical instabilities in the inverse computation of the batch normalization layer arise from the fact that the signal accross 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 used a toy parameterization of the invertible batch normalization layer to ease the explanation. For arbitrarily parameterized batch normalization layers, this formula becomes:

$$ok \tag{20}$$

The full proof of this formula is given in the appendix and follows the same reasoning as for the toy parametrization. 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 found to hold true in practice. Figure 1 compares the theoretical and empirically observed α ratio for randomly parameterized layers given random Gaussian input activation and output noise, and shows a perfect match.

The more relative difference exists in channel parameters γ, the worse the signal to noise ratio gets degraded by the inverse formulation. We propose the following modification, introducing the hyperparameter ϵ_i to the invertible batch normalization layer:

$$y = f(x) = |\gamma + \epsilon_i| \times \frac{x - \hat{x}}{\sqrt{\dot{x} + \epsilon}} + \beta \tag{21a}$$
$$x = f^{-1}(y) = (\sqrt{\dot{x}} + \epsilon) \times \frac{y - \beta}{|\gamma + \epsilon_i|} + \hat{x} \tag{21b}$$

The introduction of the ϵ_i hyper parameter 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.

4.1.2 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 follow:

$$y = f(x) = \begin{cases} x, & \text{if } x > 0 \\ x/n, & \text{otherwise} \end{cases} \tag{22a}$$
$$x = f^{-1}(y) = \begin{cases} y, & \text{if } y > 0 \\ y \times n, & \text{otherwise} \end{cases} \tag{22b}$$

The analysis of the numerical errors yielded by the invertible Leaky ReLU follows a similar reasoning as the toy batch normalization example with a few additional subtelties: 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 multiplying the input by a factor $1/n$ during the forward pass and multiplying the output by a factor n during the inverse 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 behaviour of the invertible Leaky ReLU across four cases:

$$y = \begin{cases} y_{nn} \text{ if } \hat{y} < 0 & \text{and } y < 0 \\ y_{np} \text{ if } \hat{y} >= 0 & \text{and } y < 0 \\ y_{pp} \text{ if } \hat{y} >= 0 & \text{and } y >= 0 \\ y_{pn} \text{ if } \hat{y} < 0 & \text{and } y >= 1 \end{cases} \tag{23a}$$

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 $y >> \epsilon_y$, the effects of sign are negligible, yielding a formula similar to our toy batch normalization examples:

In this regime, numerical errors can be controlled by setting a negative slope n closer to 1 As n tends towards 1, however, the network tends toward a linear behaviour. In Section XXX, we investigate the impact of the negative slope parameter on the model accuracy.

When the noise reaches an amplitude similar 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 as numerical errors prevent any useful weight update. We characterize this regime in the appendix, and derive the *alpha* formula for uniform noise and input distributions.

4.1.3 Invertible convolutions

Invertible convolution layers could be defined in different ways. For instance, convolution layers can be made invertible by constraining their kernel and the inverse of such operation is often referred to as deconvolution. However, computing the inverse of a convolution can be computationally expensive and is prawn to numerical errors.

Instead, we choose to implement invertible convolutions using the reversible block design for its simplicity, numerical stability and computational efficiency. Hence, an invertible convolution in our architecture refers to a minimal reversible block in which module F and G consist of a single convolution respectively.

XXX *et al.*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 and Leaky ReLU layers.

4.1.4 Pooling

In XXX *et al.*, the authors propose an inversible pooling operation that works by stacking the neighboring elements of the pooling regions along the channel dimension. 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:

$$M(\theta) = c_{in} \times c_{out} \times k_h \times k_w \tag{24a}$$
$$M(\theta) = c^2 \times k_h \times k_w \tag{24b}$$

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 $n(\theta) = 2048 \times 3 \times 3 \times 2048 = 37M$ parameters. At this point, the memory cost of the network weights M_θ may outweight the activation memory cost and becomes a new 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:

$$\mathcal{P}_c : x \rightarrow y \tag{25a}$$
$$\mathcal{P}_c : \mathbb{R}^{bs \times c \times h \times w} \rightarrow \mathbb{R}^{bs \times 4c \times \frac{h}{2} \times \frac{w}{2}} \tag{25b}$$
$$\mathcal{P}_b : x \rightarrow y \tag{25c}$$
$$\mathcal{P}_b : \mathbb{R}^{bs \times c \times h \times w} \rightarrow \mathbb{R}^{4bs \times c \times \frac{h}{2} \times \frac{w}{2}} \tag{25d}$$

Batch pooling gives us a way to perform volume-preserving pooling operations without the cost of a quadratic increase in the kernel weights of subsequent layer. By alterning between channel and batch pooling, we can control the number of channels at each pooling level of the model's architecture.

4.1.5 Layer-wise invertible architecture

XXX

4.2 Hybrid architecture

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

To prevent these numerical instabilities, we introduce an hybrid architecture that leverages both residual block-level and layer-wise invertible modules. This hybrid architecture, illustrated in Figure XXX, combines residual reversible 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.

In more detail, the backward pass through a reversible block of this architecture proceeds as follows: First, given the reversible block output y_1 and y_2, we recompute the input x_1 and x_2 using the analytical inverse of the reversible block, without storing the intermediate activation values of the F and G module. Second, we backpropagate the gradient of the activations through module G and F, reconstructing the hidden activation values on-the-fly using the layer-wise inverse operation.

The analytical inverse of the residual level reversible blocks allows us to propagate hidden activations with minimal reconstruction error to the ower modules, while layer-wise inversion allows us to alleviate the local bottleneck of the reversible block by computing the hidden activation values on the fly. Layer-wise inverse are only used for hidden feature computations within the scope of the reversible block so that numerical errors do not accumulate up to a damaging degree as reversible blocks are made of relatively short chains of operations.

5 Experiments and results

5.1 Impact of Numerical stability

In this section, we quantify the accumulation of numerical errors in layer-wise invertible architectures and analyse their impact on the accuracy. We start by evaluating a small layer-wise invertible LeNet-like architecture on the MNIST dataset. We show the impact of the hyperparameters on the numerical errors and investigate the impact of numerical errors on the accuracy.

We then investigate a larger layer-wise invertible vgg-like model on the CIFAR10 dataset and show that the accumulation of numerical errors prevent from converging. We demonstrate that our hybrid model, resnet-like model stabilize training.

5.1.1 MNIST experiments

Figure XXX shows the architecture. Figure XXX shows the SNR at the input of each layer for different values of negative slopes.

To isolate the impact of numerical errorrs, we use the same architecture with and without reconstructions. The difference in accuracy between both versions show the impact of numerical errors. Figure XXX compares the accuracy of several architecture on MNIST.

5.1.2 CIFAR experiments

Figure XXX shows the architecture.

Figure XXX shows the SNR.

Figure XXX shows the ResNet-like architecture.

5.2 Model comparison

Feature table showing number of params, FLOPS, Memory and accuracy

6 Conclusion

Abreviation

CNN: Convolutional Neural Network

Availability of data and material

All the code and data used in this work are publicly available at XXX

Competing interests

The authors declare that they have no competing interests.

Author's contributions

T.H and Q.F equally contributed to the investigation and implementation presented in this work. T.T and Y.A jointly supervised this project.

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. This work was supported in part by PRESTO, JST (Grant No. JPMJPR15D2).

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 feedbacks.

FIGURE LEGEND