PETRA: Parallel End-to-end Training with Reversible Architectures

Anonymous Author(s)
Affiliation
Address
email

Abstract

Reversible architectures have been shown to be capable of performing on par with their non-reversible architectures, being applied in deep learning for memory savings and generative modeling. In this work, we show how reversible architectures can solve challenges in parallelizing deep model training. We introduce PETRA, a novel alternative to backpropagation for parallelizing gradient computations. PETRA facilitates effective model parallelism by enabling stages (i.e., a set of layers) to compute independently on different devices, while only needing to communicate activations and gradients between each other. By decoupling the forward and backward passes and keeping a single updated version of the parameters, the need for weight stashing is also removed. We develop a custom autograd-like training framework for PETRA, and we demonstrate its effectiveness on CIFAR-10, ImageNet32, and ImageNet, achieving competitive accuracies comparable to backpropagation using ResNet-18, ResNet-34, and ResNet-50 models.

1 Introduction

First-order methods using stochastic gradients computed via backpropagation on mini-batches are the de-facto standard for computing parameter updates in Deep Neural Networks [25]. As datasets and models continue to grow [1] there is an urgent need for memory-efficient and scalable parallelization of deep learning training across multiple workers. Data parallelism via mini-batches [25] has been widely adopted in deep learning frameworks [26]. This approach computes gradients across model replicas distributed among workers, yet it requires frequent synchronization to aggregate gradients, leading to high communication costs, as well as substantial memory redundancy. Furthermore, with the increasing size and scale of models exceeding that of the growth of on-device memory, the forward and backward passes now often exceed a single device's memory capacity [35]. To further address these issues, methods have attempted to mitigate this memory overhead and to parallelize the sequential backpropagation steps themselves across devices, while computing exact gradients. Techniques like optimizer sharding [34], tensor parallelism [36], activation checkpointing [6], or pipelining [15], have been deployed individually or combined, leading for instance to the development of 3D parallelism [37], a popular methodology which improves the efficiency of the backpropagation implementation. On the other hand, the fundamental inefficiency underlying the parallelization of backpropagation has not been addressed by these methods.

However, the use of exact gradient restricts algorithmic choices and parallel implementations, as highlighted by [20]. For instance, backpropagation is *backward locked*: the inputs of each layer must be propagated through the network and preserved until an error signal is retropropagated to the layer of origin. This requirement enforces a synchronous dependency among subsequent layers and requires them to systematically store intermediary activations, potentially impeding overall resource efficiency as workers must wait for each other to continue their computations and release memory used for activations. To unlock the potential of backpropagation, inexact backpropagation procedures

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

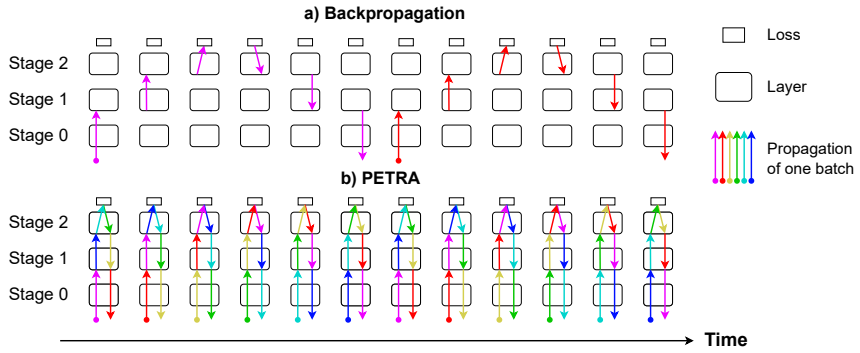

Figure 1: **Comparison of PETRA with standard backpropagation.** This approach splits the stages of a model and decouples their forward and backward passes, resulting in a sixfold increase in parallelization speed in this example.

have been proposed. These procedures are generally conceptualized within the context of model parallelism, where a neural network is split into stages that can process their activations in parallel, potentially on multiple devices. For example, some methods use outdated parameters or activations, such as double-buffered pipelining [14] or delayed gradient approaches [44]. However, these methods introduce significant memory overhead due to the use of ad hoc buffers for activations, parameters, or both. Following an opposite direction, local learning methods [33, 4], which estimate inexact gradients via a local auxiliary neural network, pave the way to parallel gradient computations but often lead to unrecoverable performance drops [11]. This underscores the need for a robust alternative to backpropagation, with limited memory overhead.

In this work, we introduce PETRA (Parallel End-to-End Training with Reversible Architectures), a novel method designed to parallelize gradient computations within reversible architectures with minimal computational overhead. Reversible architectures are an ideal candidate for this task, as they can significantly reduce memory overhead during standard backpropagation with limited communication costs. Furthermore, reversibility is a minor requirement, as many studies have demonstrated that standard architectures can be adapted into reversible ones without any performance drops [12, 19, 29, 22]. By allowing parameters to evolve in parallel and by computing an approximate inversion during backward, we propose an effective alternative to backpropagation which allows high model parallelism with a constant communication overhead and **no additional buffers**. In fact, for a constant increase in communication overhead, PETRA achieves a linear speedup compared to standard backpropagation with respect to the number J of stages the network is split into. We illustrate our approach in Fig. 1, by contrasting the evolution of PETRA with a standard backpropagation pass.

Contributions. Our contributions are as follows: **(1)** We introduce PETRA, a streamlined approach for parallelizing the training of reversible architectures. This method leverages a delayed, approximate inversion of activations during the backward pass, allowing for enhanced computational efficiency. **(2)** Our technique significantly reduces memory overhead by minimizing the necessity to store extensive computational graphs. **(3)** It enables the parallelization of forward and backward pass computations across multiple devices, effectively distributing the workload and reducing training time. **(4)** We validate the efficacy of PETRA through rigorous testing on benchmark datasets such as CIFAR-10, ImageNet-32, and ImageNet, where it demonstrates robust performance with minimal impact on accuracy. **(5)** Additionally, we provide a flexible reimplementation of the autograd system in PyTorch, specifically tailored for our experimental setup, which we make available to the research community.

2 Related work

Reversible architectures. Reversible DNNs are composed of layers that are invertible, meaning that the input of a layer can be computed from its output. This approach allows to avoid the need to store intermediary activations during the forward pass by reconstructing them progressively during the backward pass [12], at the cost of an extra computation per layer. Invertible networks further improve this method by removing dimensionality reduction steps such as downsamplings, making

the networks fully invertible [18]. Reversibility is not restricted to a type of architecture or tasks and has been extensively used for generative models [9], for ResNets [12], and Transformers [29]. However, as far as we know, reversible architectures have never been used to enhance parallelization capabilities.

Alternatives to backpropagation. Multiple alternatives to backpropagation have been proposed previously to improve over its computational efficiency. For instance, DNI [20] is the first to mention the backpropagation inefficiency and its inherent synchronization locks. However, they address those locks with a method non-competitive with simple baselines. Local (or greedy) learning [33, 3] propose to use layerwise losses to decouple the training of layers, allowing them to train in parallel [5]. Local learning in videos [28] notably uses the similarity between successive temporal features to remove buffer memory. However, the difference in training dynamics between local training and backpropagation still limits such approaches [11, 38].

Pipeline parallelism. Pipelining encompasses a range of model parallel techniques that divide the components of a network into stages that compute in parallel, while avoiding idle workers. Initially popularized by [15], a batch of data is divided into micro-batches that are processed independently at each stage. Although more efficient pipelining schedules have been proposed [10], notably to mitigate the peak memory overhead, keeping an exact batch gradient computation requires leaving a bubble of idle workers. By alternating one forward and one backward pass for each worker, PipeDream [31] can allow to get rid of idleness bubbles, but at the expense of introducing staleness in the gradients used. [32] mitigates this staleness to only one optimization step by accumulating gradients, thus also reducing the parameter memory overhead to only two versions of the parameters. Nevertheless, these approaches still suffer from a quadratic activation memory overhead with regard to the number of stages, as micro-batch activations pile up in buffers, especially for early layers. Some implementations propose to limit this overhead by combining activation checkpointing [6] with pipelining [21, 27], although the memory overhead still scales with the number of stages.

Delayed gradient. By allowing stale gradients in the update process, these previous methods provide the context for our approach. Delayed gradient optimization methods are model parallel techniques that aim to decouple and process layers in parallel during backpropagation. In these approaches, delays occur stage-wise: the backward pass may be computed with outdated parameters or activations compared to the forward pass. For instance, [16] proposes a feature replay approach, where a forward pass first stores intermediary activations, which are then "replayed" to compute the backward pass in parallel. This method still requires heavy synchronization between layers, yielding a lock on computations. In [42] and [43], stale gradients are computed from older parameter versions differing from the parameters used during the update. This staleness can be mitigated: [43] 'shrinks' the gradient by the delay value, but more advanced techniques also exist [41, 23]. Still, these methods are limited like previous pipelining methods by their memory overhead as the computational graph is fully stored. A first step to reduce this, as proposed in Diversely Stale Parameters (DSP) [40], PipeMare [41] and [23], is to keep a single set of parameters and approximate the gradients computed during the backward pass with the updated parameters, which differ from the ones used in the forward pass. This requires, like in activation checkpointing, an additional reconstruction of the computational graph. Furthermore, the quadratic activation memory overhead still limits the scalability of these methods for a large number of stages.

3 Method

3.1 Standard backpropagation

We consider a DNN composed of J stages (e.g., a layer or a set of layers). An input x_0 is propagated through the network, recursively defined by

$$x_j \triangleq F_j(x_{j-1}, \theta_j)\,, \tag{1}$$

where F_j is the j-th stage parameterized by θ_j. The backpropagation algorithm is the ubiquitous algorithm to compute parameter gradients. First, an input is propagated through the network with a forward pass, while storing its intermediate activations. A scalar loss $\mathcal{L}$ is then deduced from the corresponding output x_J. Parameter gradients are then computed during the backward pass by taking

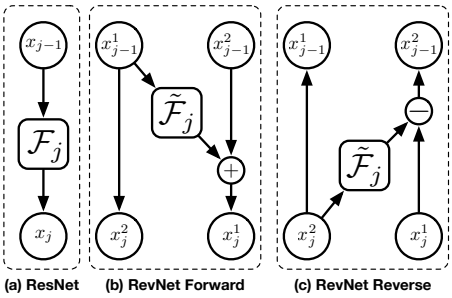

Figure 2: **Differences between the residual block of a ResNet and its reversible counterpart. (a)** Forward of a residual block. **(b)** Forward and **(c)** Reverse forward of a reversible residual block. For reversible blocks, similarly to [12], the input x_j is doubled in size and split equally into $\{x_j^1, x_j^2\}$ along the channel dimension. The function $\mathcal{F}_j$ includes a skip-connection while $\tilde{\mathcal{F}}_j$ does not.

advantage of the chain rule: starting from the last stage with $\delta_J = \nabla_{x_J}\mathcal{L}$, the gradients with regard to the activations are given by

$$\delta_j \triangleq \nabla_{x_{j-1}}\mathcal{L} = \partial_x F_j(x_{j-1}, \theta_j)^{\mathrm{T}}\delta_{j+1}\,, \tag{2}$$

and the gradients with regard to the parameters are defined as

$$\Delta_j \triangleq \nabla_{\theta_j}\mathcal{L} = \partial_\theta F_j(x_{j-1}, \theta_j)^{\mathrm{T}}\delta_{j+1}\,. \tag{3}$$

Note that these computations follow a synchronous and sequential order. The parameters θ_j can then be updated given their gradient estimate Δ_j, using any optimizer.

3.2 Reversible architectures

We focus on the reversible neural networks presented in [12], although our method is not dependent on this architecture. In practice, only a few stages which do not preserve feature dimensionality are not reversible and correspond to the downsampling blocks in the ResNet. Fig. 2 highlights how reversible residual blocks F_j differ from their standard counterpart. The input is split into two equal-size inputs, along the channel dimension, that are propagated forward according to Fig. 2b using an ad-hoc operator $\tilde{\mathcal{F}}_j$. It can be reconstructed by reverse propagating the output according to Fig. 2c, by subtracting the output of $\tilde{\mathcal{F}}_j$ rather than adding it like in the previous forward.

Reversible stages. In order to compute the exact gradients during the backpropagation phase, each reversible stage needs to retrieve its output from the stage above. We note F_j^{-1} the reverse stage function, which reconstructs the input from the output. We recursively apply the reconstruction to the final activation x_J, such that

$$\begin{bmatrix} x_{j-1} \\ \delta_j \end{bmatrix} = \begin{bmatrix} F_j^{-1}(x_j, \theta_j) \\ \partial_x F_j(F_j^{-1}(x_j, \theta_j), \theta_j)^{\mathrm{T}}\delta_{j+1} \end{bmatrix}\,. \tag{4}$$

Note that reconstructing the input in our procedure is computationally equivalent to recomputing the activations in activation checkpointing, meaning it is equivalent to a single forward pass. Thus, this augmented backward procedure is equivalent to one regular forward call and backward call. However, one should observe that since the input x_{j-1} must be sent to the reversible stages, this doubles the cost of backward communications.

Non-reversible stages. In practice, a reversible architecture includes layers that reduce dimensionality for computational efficiency, which thus correspond to non-invertible functions. For those very few stages, we employ a buffer mechanism to store activations and, like activation checkpointing, we recompute the computational graph with a forward pass during the backward pass. Note that this would not be the case for invertible (i.e., bijective) architectures [18], which use an invertible downsampling.

Table 1: **Comparisons with other methods in an ideal setting for one stage.** We compare several methods to compute a gradient estimate in a model parallel setting. Here, J is the total number of stages while j is the stage index. For the sake of simplicity, we assume that a backward pass requires approximately 2 times more FLOPs than a forward pass. *Full Graph* indicates that it is required to store the full computational graph of a local forward pass. With a limited increase in communication volume and FLOPs, PETRA requires the least storage of all methods while being linearly faster than backpropagation. We assume that the forward and backward passes can be executed in parallel for PETRA or delayed gradients, making the backward pass responsible for most of the computation time in parallelizable approaches.

Methods	Storage Activations	Storage Params.	Comm. Volume	FLOPs	Mean time per batch
Backpropagation	Full Graph (FG)	**1**	**1**	$\mathbf{3}J$	$3J$
Reversible backprop. [12]	**0**	**1**	4	$4J$	$4J$
Delayed gradients [42]	$2(J-j)\times$ FG	$\frac{2(J-j)}{k}$	**1**	$\mathbf{3}J$	**2**
+ Checkpointing [40]	$2(J-j)$	**1**	**1**	$4J$	3
PETRA (ours)	**0**	**1**	4	$4J$	3

3.3 A parallelizable approach: PETRA

As with any model parallel training technique, PETRA requires to partition the network architecture into stages F_j that are distributed across distinct devices. Each device j needs only to communicate with its neighboring devices $j-1$ and $j+1$. The pseudo-code in Alg. 1 details the operations performed by each device, and the whole algorithm execution can be summarized as follows. The first device sequentially accesses mini-batches, initiating the data propagation process. When receiving its input x_{j-1}^t from the previous stage, each stage processes it in forward mode and passes it to the next stage, until the final stage is reached. The final stage evaluates the loss and computes the gradients with regard to its input and parameters, thus initiating the backward process, which is performed in parallel of the forward process. In it, each stage processes the input and its associated gradient from the next stage. This means first reconstructing the computational graph, either while reconstructing the input $\tilde{x}_{j-1}^t$ for reversible stages or with a forward pass as in activation checkpointing otherwise. Then, the parameter gradient approximation Δ_j^{t+1} and the input gradient are computed before passing the latter to the previous stage. For intermediary reversible stages, this translates into the following equations, where t corresponds to the current time step of the training,

$$\begin{cases} x_j^{t+1} = F_j(x_{j-1}^t, \theta_j^t) \\ \tilde{x}_{j-1}^{t+1} = F_j^{-1}(\tilde{x}_j^t, \theta_j^t) \\ \delta_j^{t+1} = \partial_x F_j(\tilde{x}_{j-1}^{t+1}, \theta_j^t)^{\mathrm{T}} \delta_{j+1}^t \\ \Delta_j^{t+1} = \partial_\theta F_j(\tilde{x}_{j-1}^{t+1}, \theta_j^t)^{\mathrm{T}} \delta_{j+1}^t \\ \theta_j^{t+1} = \text{Optimizer}_j^t(\theta_j^t, \Delta_j^{t+1}) \,. \end{cases} \tag{5}$$

Note that this complete set of equations effectively decouples communications, computations, and parameter updates between independent devices. Indeed, reversible stages are able to operate without maintaining any state between the forward and corresponding backward phase by simply avoiding weight stashing, similarly to [40], and by reversing the output into the input during the backward phase, removing the need for an input buffer. As parameters are updated between the forward and backward phases, the reversible stage produces an approximate input reconstruction, thus evaluating gradients with an approximate set of inputs and parameters during the backward phase. We illustrate in Fig. 3 the mechanism of PETRA compared to standard delayed gradient approaches that rely on additional buffers [44, 42].

Complexity analysis. We now discuss the benefits of our method, which are summarized in Tab. 1. In this discussion, we assume a homogeneous setting in which almost identical stages are distributed across J devices uniformly. First, we consider the backpropagation setting, assuming a model

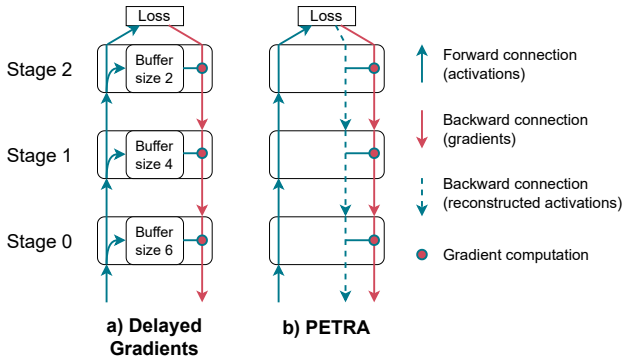

Figure 3: **Comparison of our PETRA method to a standard Delayed Gradient method [42].** By avoiding weight stashing and reversing the output into the input during the backward phase, we are able to fully decouple the forward and backward phases in all reversible stages, with no memory overhead, compared to standard delayed gradient approaches.

parallelism strategy: a standard backpropagation pass requires storing locally both the parameters and the computational graph and due to the update lock of backpropagation [20], requires synchronization between subsequent layers which impede the speed of computations. Standard Delayed Gradients strategies as implemented in [44, 42] allow to unlock this barrier, but they require buffers for storing both the computational graph and parameters which can become impractical when using large models. In [40], an activation checkpointing strategy removes the need for storing parameters, yet it requires a small computational overhead of 33% (assuming a backward pass is approximatively two times slower than a forward pass, see Fig. 6 of [17] and [30]). To avoid storing activations, we rely on reversible architectures [12] which increases the amount of forward communications by a factor of 2 and backward communication by a factor of 4 – activations sizes double and one has to pass both activations and gradients at the same time during backward. None of the aforementioned methods scale with the depth J: PETRA combines all the advantages of the previous methods, allowing an efficient parallelization, while leading to a limited overhead in computations and communications.

4 Numerical experiments

4.1 Classification accuracy

We now describe our experimental setup on CIFAR-10 [24], ImageNet-32 [7], and ImageNet [8].

Experimental setup. All our experiments use a standard SGD optimizer with a Nesterov momentum factor of 0.9. We train all models for 300 epochs on CIFAR-10 and 90 epochs on ImageNet32 and ImageNet. We apply standard data augmentation, including horizontal flip, random cropping, and standard normalization but we do not follow the more involved training settings of [39], which potentially leads to higher accuracy. We perform a warm-up of 5 epochs where the learning rate linearly increases from 0 to 0.1, following [13]. Then, the learning rate is decayed by a factor of 0.1 at epochs 30, 60, and 80 for ImageNet32 and ImageNet – it is decayed at epochs 150 and 225 for CIFAR-10. We use a weight decay of 5e-4 for CIFAR-10 and 1e-4 for ImageNet32 and ImageNet. As suggested in [13], we do not apply weight decay on the batch norm learnable parameters and biases of affine and convolutional layers. For our standard backpropagation experiments, we follow the standard practice and use a batch size of 128 on ImageNet32 and CIFAR-10, and 256 on ImageNet32. However, we made a few adaptations to train our models with PETRA. As suggested by [42, 43], we employ an accumulation factor k and a batch size of 64, which allows to reduce the effective staleness during training: in this case, k batches of data must be successively processed before updating the parameters of a stage (see Alg. 1). Such gradient accumulation however also increases the effective batch size, and we apply the training recipe used in [13] to adjust the learning rate; note that we use the average of the accumulated gradients instead of the sum. The base learning rate is thus given by the formula $\texttt{lr} = 0.1\frac{64k}{256}$, with k the accumulation factor.

Algorithm 1 Worker perspective for training in parallel with PETRA, on a stage j, assuming initialized parameters θ_j and time step t, as well as an accumulation factor $k > 1$.

In parallel *on the j-th stage, $1 \leq j < J$, perform:*
  Forward Communications and Computations:
    If $j = 1$ **then**
      $x_0 \leftarrow$ **Read**$_{\text{dataset}}$
    Else
      $x_{j-1} \leftarrow$ **Wait and Receive** $_{\text{from } j-1}$
    If stage j is not reversible :
      Buffer$_j \leftarrow x_j$
    $x_j \leftarrow F_j(x_{j-1}, \theta_j)$
    Send $_{\text{to } j+1}(x_j)$
  Backward Communications and Computations:
    $(\tilde{x}_j, \delta_{j+1}) \leftarrow$ **Wait and Receive** $_{\text{from } j+1}$
    If stage j is reversible:
      $\tilde{x}_{j-1} \leftarrow F_j^{-1}(\tilde{x}_j, \theta_j)$ *and keep computational graph in memory*
    Else :
      $\tilde{x}_{j-1} \leftarrow$ **Buffer**$_j$
      $x_j \leftarrow F_j(\tilde{x}_{j-1}, \theta_j)$ *to recompute the computational graph*
    $\delta_j \leftarrow \partial_x F_j(\tilde{x}_{j-1}, \theta_j)^T \delta_{j+1}$
    $\Delta_j \leftarrow \Delta_j + \frac{1}{k} \partial_\theta F_j(\tilde{x}_{j-1}, \theta_j)^T \delta_{j+1}$
    If $t \bmod k = 0$ **then**:
      Update parameters θ_j with Δ_j
      $\Delta_j \leftarrow 0$
    $t \leftarrow t + 1$
    Send $_{\text{to } j-1}(x_j, \delta_j)$

In parallel *on the final stage J, perform:*
    $x_{J-1} \leftarrow$ **Wait and Receive** $_{\text{from } J-1}$
    $\mathcal{L} \leftarrow F_J(x_{J-1}, \theta_J)$
    $\delta_J \leftarrow \nabla_{x_J} \mathcal{L}$
    $\Delta_J \leftarrow \Delta_J + \frac{1}{k} \nabla_{\theta_J} \mathcal{L}$
    If $t \bmod k = 0$ **then**:
      Update parameters θ_J with Δ_J
      $\Delta_J \leftarrow 0$
    $t \leftarrow t + 1$
    Send $_{\text{to } J-1}(x_{J-1}, \delta_J)$

Model adaptations. For designing our RevNet architectures, we adopt a methodology similar to [12]: the number of channels in each stage is multiplied by 2 to account for the second data stream according to Fig. 2. However, as the stage function $\tilde{\mathcal{F}}_j$ operates only on one of the two streams, the number of parameters stays almost the same between a residual block and its revertible counterpart. Consequently, the DNNs are split to preserve each residual block, resulting in 10 stages for RevNet18, and 18 stages for RevNet34 and RevNet50; thus varying the level of staleness between configurations. On CIFAR-10, the input layer uses 3x3 convolutions instead of 7x7 convolutions and does not perform max-pooling. The running statistics of batch normalization layers are updated when recomputing the activations during the backward pass and are then used during model evaluation – the running statistics are not updated during the forward pass.

Performance comparison. Tab. 2 reports our numerical accuracy on several vision datasets, comparing a backpropagation performance from an official PyTorch implementation of ResNets (the numbers can be found as v1 of `https://pytorch.org/hub/pytorch_vision_resnet/`), for our own implementation of ResNets and RevNets in our custom computational framework, and our proposed method, PETRA. For PETRA, we report the best classification accuracy after the last learning rate drop, using the best value (picked on the training set) of accumulation steps within $\{1, 2, 4, 8, 16, 32\}$. Our CIFAR-10 accuracies are averaged over 3 runs, with a variance smaller than

Table 2: **Classification accuracies using our PETRA method with RevNets, compared to standard backpropagation on ResNets and RevNets** on CIFAR-10, ImageNet32, and ImageNet. Our method delivers competitive results with backpropagation, even on ImageNet.

Method	Model	Param. count	CIFAR-10	ImNet32	ImNet
Backprop	ResNet18 (PyTorch)	11.7M	-	-	69.8
Backprop	ResNet18 (Ours)	11.7M	95.0	54.0	70.8
Backprop	**Rev**Net18 (Ours)	12.2M	94.9	54.6	70.8
PETRA	**Rev**Net18 (Ours)	12.2M	94.9	54.6	71.0
Backprop	ResNet34 (PyTorch)	21.8M	-	-	73.3
Backprop	ResNet34 (Ours)	21.8M	95.5	56.5	74.0
Backprop	**Rev**Net34 (Ours)	22.3M	95.3	56.4	73.2
PETRA	**Rev**Net34 (Ours)	22.3M	94.8	56.1	73.5
Backprop	ResNet50 (PyTorch)	25.6M	-	-	76.1
Backprop	ResNet50 (Ours)	25.6M	94.8	58.8	75.6
Backprop	**Rev**Net50 (Ours)	30.4M	95.2	59.7	75.4
PETRA	**Rev**Net50 (Ours)	30.4M	94.5	59.6	74.8

0.1. We observe that while our reversible models have about the same parameter count, they all perform in the same range of accuracy as their non-reversible counterparts. Only the RevNet-50 leads to a small drop in accuracy on ImageNet of about 0.6%: using different downsampling layers removes this gap at the expense of a substantial increase in the parameter count (30.4M to 50M). However, we decided not to include this result for the sake of comparison with respect to the original ResNets.

Impact of the accumulation k. We test the impact of the accumulation on a RevNet-18 trained via PETRA for various values of accumulations with k spanning $\{1, 2, 4, 8, 16, 32\}$ on the ImageNet dataset. Fig. 4 indicates that our method can benefit from large accumulation factors, with the well-known trade-off of large batches mentioned in [13]. Increasing the accumulation factor reduces the effective staleness during training, and closes the performance gap with standard backpropagation with perfect matching for $k = 32$. This confirms that this large-batch training recipe derived for synchronous data parallelism is also particularly suited for our model parallel approach.

4.2 Technical details

A note on the implementation. We shortly describe our implementation details. We base our method on PyTorch [2], although we require significant modifications to the Autograd framework in order to manage delayed first-order quantities consistently with PETRA. We rely heavily on the *Vector Jacobian Product* of PyTorch to compute gradients during the backward pass of each stage, but other backends could be used. The backward pass for reversible stages only necessitates a reconstruction step and a backward step – a naive implementation would use a reconstruction step, followed by a forward and a backward step. This is because we only need the output gradient as well as the computational graph of $\tilde{\mathcal{F}}_j$ to compute the input and parameter gradients at line 12 and 13 of Alg. 1, which can be obtained during the input reconstruction phase. For non-reversible stages, we reconstruct the computational graph with a forward pass on the input retrieved from the buffer during the backward pass. Our models can run on a single A100, 80GB.

Memory benefits and training time. To better understand the advantage of our method compared to other delayed gradient approaches [14, 40, 23], we emphasize the practical memory savings associated with different methods in Tab. 3. We estimate the memory needed in gigabytes, as the sum of the model size, the input buffer size, and the parameter buffer size, while excluding the input buffer size of the first stage, which corresponds to retrievable dataset inputs. We do not include the effect of gradient accumulation since it depends on the value of k and only affects the length of the parameter buffer, which is small in our case, i.e., we use $k = 1$. Note that the batch size also affects the memory savings, and we set it to 64 for consistency with Tab. 2. Storing both inputs and parameters into a buffer corresponds to the PipeDream approach [14]. Only storing inputs into

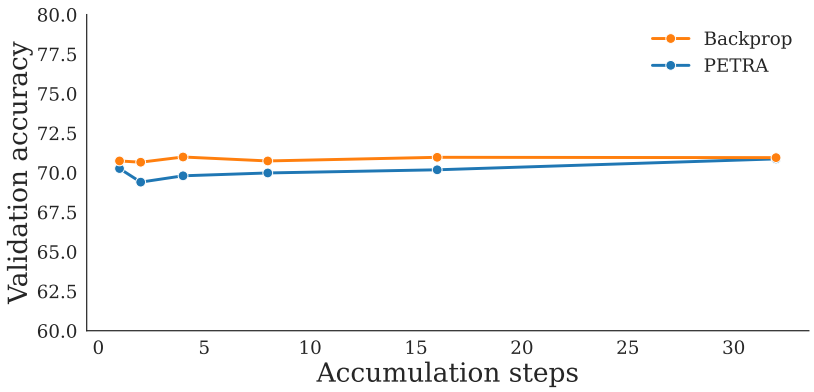

Figure 4: **Validation accuracy of PETRA and backpropagation for a various number of accumulation steps**, for a RevNet18 trained on ImageNet with $k \in \{1, 2, 4, 8, 16, 32\}$. The validation accuracies are averaged over the last 10 epochs. As the number of accumulation steps increases, the effective staleness in PETRA decreases, closing the gap with standard backpropagation.

Table 3: **Memory savings for RevNet50 on ImageNet with our method for different configurations.** We indicate the use of memory buffers for inputs or parameters. The savings are computed with respect to the first configuration, where inputs and buffers are stored. Our method achieves 54.3% memory reduction over the base configuration of Delayed Gradients.

Buffer		Memory (GB)	Saving (%)
Input	Params.		
√	√	44.5	0.0
√	×	43.6	2.0
×	√	21.2	52.3
×	×	**20.3**	**54.3**

buffers would correspond to the approach in [40, 23]. The third and fourth lines are only applicable to reversible architectures as they do not store the input into buffers. As can be seen, the input buffer has the biggest impact on the total memory needed, being responsible for 52.3% of the memory footprint. Dropping the parameter buffer in PETRA pushes the memory savings further to 54.3% for a RevNet50 on ImageNet. Note that non-reversible stages account for the majority of total memory use, meaning that savings would be much higher for fully invertible architectures.

5 Conclusion

In this work, we introduce PETRA, a novel model parallel training technique for reversible architectures which is a novel promising alternative to backpropagation. It achieves a significant parallelization with a limited overhead compared to standard backpropagation or other competitive alternatives to end-to-end training, like delayed gradients approaches. Our method has the potential to achieve linear speedup compared to standard backpropagation and allows reversible layers to operate without any parameter or activation buffers, effectively decoupling the forward and backward phases. Despite using an approximate delayed gradient estimate, our method delivers competitive performances compared to standard backpropagation on standard computer vision datasets.

In future work, we aim to implement and optimize PETRA for Large Language Models (LLMs), with a first baseline being Reformers [22], invertible transformers that have been shown to scale. This will validate PETRA's effectiveness and robustness, solidifying its potential as a cutting-edge training technique.

References

[1] I. M. Alabdulmohsin, B. Neyshabur, and X. Zhai. Revisiting neural scaling laws in language and vision. *Advances in Neural Information Processing Systems*, 35:22300–22312, 2022.

[2] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[3] E. Belilovsky, M. Eickenberg, and E. Oyallon. Greedy layerwise learning can scale to imagenet. In *International conference on machine learning*, pages 583–593. PMLR, 2019.

[4] E. Belilovsky, M. Eickenberg, and E. Oyallon. Decoupled greedy learning of cnns. In *International Conference on Machine Learning*, pages 736–745. PMLR, 2020.

[5] E. Belilovsky, L. Leconte, L. Caccia, M. Eickenberg, and E. Oyallon. Decoupled greedy learning of cnns for synchronous and asynchronous distributed learning. *arXiv preprint arXiv:2106.06401*, 2021.

[6] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost, 2016.

[7] P. Chrabaszcz, I. Loshchilov, and F. Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets, 2017.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] L. Dinh, D. Krueger, and Y. Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[10] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[11] L. Fournier, S. Rivaud, E. Belilovsky, M. Eickenberg, and E. Oyallon. Can forward gradient match backpropagation? In *Fortieth International Conference on Machine Learning*, 2023.

[12] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. *Advances in neural information processing systems*, 30, 2017.

[13] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[14] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[15] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[16] Z. Huo, B. Gu, and H. Huang. Training neural networks using features replay. *Advances in Neural Information Processing Systems*, 31, 2018.

[17] Z. Huo, B. Gu, H. Huang, et al. Decoupled parallel backpropagation with convergence guarantee. In *International Conference on Machine Learning*, pages 2098–2106. PMLR, 2018.

[18] J.-H. Jacobsen, A. Smeulders, and E. Oyallon. i-revnet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

[19] J.-H. Jacobsen, A. W. M. Smeulders, and E. Oyallon. i-revnet: Deep invertible networks. *ArXiv*, abs/1802.07088, 2018.

[20] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *International conference on machine learning*, pages 1627–1635. PMLR, 2017.

[21] C. Kim, H. Lee, M. Jeong, W. Baek, B. Yoon, I. Kim, S. Lim, and S. Kim. torchgpipe: On-the-fly pipeline parallelism for training giant models, 2020.

[22] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[23] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Koster. Pipelined backpropagation at scale: training large models without batches. *Proceedings of Machine Learning and Systems*, 3:479–501, 2021.

[24] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[25] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[26] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[27] Y. Liu, S. Li, J. Fang, Y. Shao, B. Yao, and Y. You. Colossal-auto: Unified automation of parallelization and activation checkpoint for large-scale models, 2023.

[28] M. Malinowski, D. Vytiniotis, G. Swirszcz, V. Patraucean, and J. Carreira. Gradient forward-propagation for large-scale temporal video modelling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9249–9259, 2021.

[29] K. Mangalam, H. Fan, Y. Li, C.-Y. Wu, B. Xiong, C. Feichtenhofer, and J. Malik. Reversible vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10830–10840, 2022.

[30] E. Mizutani and S. Dreyfus. On complexity analysis of supervised mlp-learning for algorithmic comparisons. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, volume 1, pages 347–352 vol.1, 2001.

[31] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[32] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[33] A. Nøkland and L. H. Eidnes. Training neural networks with local error signals. In *International conference on machine learning*, pages 4839–4850. PMLR, 2019.

[34] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[35] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[36] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[37] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[38] Y. Wang, Z. Ni, S. Song, L. Yang, and G. Huang. Revisiting locally supervised learning: an alternative to end-to-end training. *arXiv preprint arXiv:2101.10832*, 2021.

[39] R. Wightman, H. Touvron, and H. Jégou. Resnet strikes back: An improved training procedure in timm, 2021.

[40] A. Xu, Z. Huo, and H. Huang. On the acceleration of deep learning model parallelism with staleness. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2085–2094, 2019.

[41] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

[42] H. Zhuang, Z. Lin, and K.-A. Toh. Accumulated decoupled learning: Mitigating gradient staleness in inter-layer model parallelization. *arXiv preprint arXiv:2012.03747*, 2020.

[43] H. Zhuang, Y. Wang, Q. Liu, and Z. Lin. Fully decoupled neural network learning using delayed gradients. *IEEE transactions on neural networks and learning systems*, 33(10):6013–6020, 2021.

[44] H. Zhuang, Z. Weng, F. Luo, T. Kar-Ann, H. Li, and Z. Lin. Accumulated decoupled learning with gradient staleness mitigation for convolutional neural networks. In *International Conference on Machine Learning*, pages 12935–12944. PMLR, 2021.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Our novel method is presented in Sec. 3 as described. We provide numerical accuracy and ablations in Sec. 4.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss it along with our experimental results in Sec. 4.2.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: We do not provide theoretical results in this paper.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The experimental results are detailed in the article, and can be reproduced with the provided code.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The data is openly accessible and properly referenced in the text. The code is included in the supplementary material.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: The hyperparameters are detailed in the article and the code.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report accuracy averaged over 3 runs on CIFAR10 which have low variance, and we follow to the standard practice on ImageNet and ImageNet32.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

Justification: Details about resources and executions are presented Sec 4.2.

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Our research conforms to the Code of Ethics.

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: Our article is concerned with allowing general deep learning optimization, and does not have particular societal impacts.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: This paper does not present such risks.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The datasets used are cited, and we credit the author of the code used for our implementation.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The code will be provided with a README file describing how to run experiments.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: No crowdsourcing or human subjects.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: No potential risks or study participants.