Moonwalk: Inverse-Forward Differentiation

Dmitrii Krylov
University of California, Irvine

Armin Karamzade
University of California, Irvine

Roy Fox
University of California, Irvine

Abstract

Backpropagation's main limitation is its need to store intermediate activations (residuals) during the forward pass, which restricts the depth of trainable networks. This raises a fundamental question: can we avoid storing these activations? We address this by revisiting the structure of gradient computation. Backpropagation computes gradients through a sequence of vector–Jacobian products, an operation that is generally irreversible. The lost information lies in the cokernel of each layer's Jacobian. We define submersive networks—networks whose layer Jacobians have trivial cokernels—in which gradients can be reconstructed exactly in a forward sweep without storing activations. For non-submersive layers, we introduce fragmental gradient checkpointing, which records only the minimal subset of residuals necessary to restore the cotangents erased by the Jacobian. Central to our approach is a novel operator, the vector–inverse-Jacobian product (vijp), which inverts gradient flow outside the cokernel. Our mixed-mode algorithm first computes input gradients with a memory-efficient backward pass, then reconstructs parameter gradients in a forward sweep that does not need to store activations. We implement this method, called Moonwalk, and show that it matches backpropagation's runtime while training networks more than twice as deep under the same memory budget.

1 INTRODUCTION

Training deep neural networks via reverse-mode automatic differentiation, commonly known as backpropagation (Backprop), requires storing *residuals* during the forward pass—intermediate values sufficient to evaluate the vector–Jacobian products (*vjp*) used in the backward pass. This incurs a memory cost that grows with the network depth and often limits model capacity. Existing techniques such as reversible layers [Gomez et al., 2017] and activation checkpointing [Martens and Sutskever, 2012] modify Backprop to reduce its memory footprint, but either impose significant architectural constraints or increase computation time. In contrast, forward-mode differentiation [Williams and Zipser, 1989] does not impose any constraints on model architecture, and only takes a single forward pass, but suffers from prohibitively high computational cost.

In this paper, we show that for a broad class of *submersive layers*, i.e. those whose Jacobians are surjective, true gradients can be computed in forward mode with significantly lower memory. Our core insight is that once the loss gradient with respect to the network input (the input *cotangent*) is known, all parameter gradients can be recovered through a sequence of right-inverse Jacobian products in a single forward sweep. This yields the **Moonwalk** method, which in some architectures can match Backprop's runtime while using less memory. Moonwalk has two variants, **mixed-mode** and **pure-forward**, that pre-compute the input cotangent in reverse-mode and, respectively, forward-mode.

Traditional backpropagation computes gradients in two distinct phases. In the first phase, it performs a forward pass to compute and store the residuals of all layers. These stored residuals are then used in the second phase to compute cotangents (loss gradients) in reverse topological order, from the loss variable back to the first layer, using the gradient chain rule. For large networks, the memory footprint scales with the number of residuals [Novikov et al., 2023], which can result in significant memory overhead, limiting the ability to scale neural networks.

Moonwalk first computes the input cotangent, either in reverse-mode (in mixed-mode Moonwalk) or in forward-mode (in pure-forward Moonwalk). Because this phase only computes the input cotangent and not parameter gradients, it can, in some architectures, incur signifi-

Proceedings of the 29[th] International Conference on Artificial Intelligence and Statistics (AISTATS) 2026, Tangier, Morocco. PMLR: Volume 300.

cantly less memory overhead than full automatic differentiation. Moonwalk then performs a forward sweep using our custom vector-inverse-Jacobian product (*vijp*) operator to recover parameter gradients. It offers the same computational complexity as Backprop in architectures where *vijp* is as efficient as *vjp*, but with a reduced memory footprint since parameter gradients are computed without storing full residuals.

In summary, this work contributes:

- A novel algebraic identity (Eq. 7) enabling true forward-mode gradient computation in submersive networks, given a pre-computed input cotangent.
- Two variants of the Moonwalk method, pure-forward and mixed-mode, both offering reduced memory footprints with a computational trade-off.
- An efficient implementation of the required *vijp* operators, including derived conditions for submersive convolutions and fully parallelizable forms.
- Fragmental gradient checkpointing, a memory-efficient scheme for forward-mode reconstruction of complete cotangents from partially stored ones.
- Empirical results on submersive convolutional layers using efficient *vijp* operators, demonstrating up to $\times 2$ memory reduction with comparable computation time, enabling the training of networks more than $\times 2$ deeper or with larger batch sizes than backpropagation.

2 RELATED WORK

Reducing automatic differentiation memory. Prior work has explored various strategies to reduce the memory footprint of training deep networks. A widely used approach is *activation checkpointing* [Martens and Sutskever, 2012, Chen et al., 2016, Gruslys et al., 2016, Kumar et al., 2019, Jain et al., 2019, Zhao et al., 2023, Gusak et al., 2025, Liu et al., 2025, Chen et al., 2025, Zeng et al., 2025, Xu et al., 2025, Korthikanti et al., 2022], which reduces memory usage in a network with L layers by a factor of $O(\sqrt{L})$. This is achieved by storing only $O(\sqrt{L})$ activations (layer outputs) during the forward pass and running a second forward pass inside the backward loop to rematerialize intermediate values. While effective, this technique increases compute time and still requires full residuals in the inner-loop forward pass.

Invertible architectures. Another line of work leverages invertible layers that allow activations to be recomputed exactly during the backward pass [Gomez et al., 2017, MacKay et al., 2018, Mangalam et al., 2022, Cai et al., 2023, Gal et al., 2025]. Reversible backpropagation that avoids storing activations altogether has been applied to memory-efficient training [Gomez et al., 2017], improved representations [Jacobsen et al., 2018], and generative models [Kingma and Dhariwal, 2018, Dinh et al., 2014, Rezende and Mohamed, 2015]. For example, Bulo et al. [2018] replaced ReLU and batch normalization layers with invertible alternatives, reducing memory usage by up to 50%. Synthetic gradients have also been explored to decouple gradient computation [Jaderberg et al., 2017]. However, these methods are restricted to architectures where layers are exactly invertible. In contrast, Moonwalk applies to the larger class of *submersive networks*, whose layer Jacobians are everywhere surjective but not necessarily invertible.

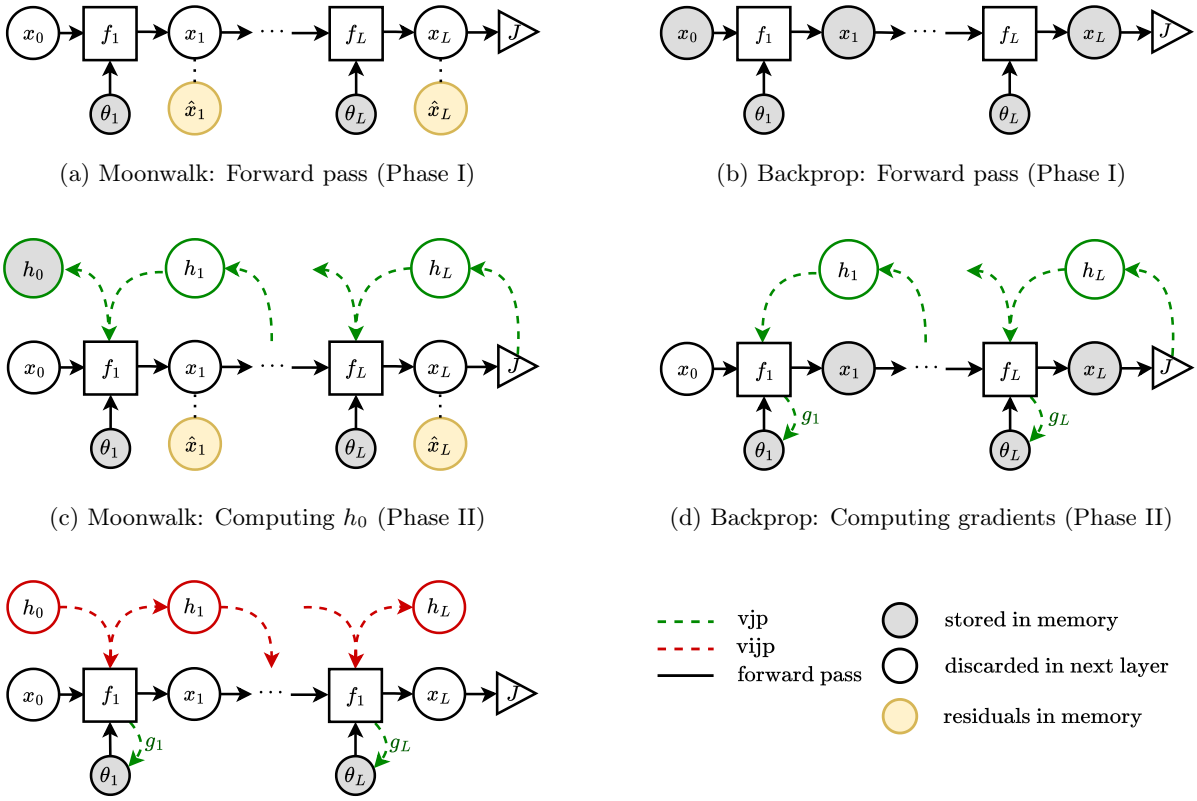
Forward-mode differentiation. Forward-mode automatic differentiation has been proposed as an alternative to reverse-mode, particularly in recurrent networks (as in RTRL [Williams and Zipser, 1989]). More recently, projection-based variants have emerged, where directional derivatives are used to approximate true gradients [Silver et al., 2021, Baydin et al., 2022]. While these approaches reduce memory costs, they introduce gradient noise due to stochastic tangent vectors or imprecise surrogate networks [Ren et al., 2022, Fournier et al., 2023], leading to subpar optimization performance. In contrast, Moonwalk avoids both full Jacobian materialization and projection noise by computing exact gradients using an efficient forward-mode operator, the vector–inverse-Jacobian product (*vijp*).

Vector-Inverse-Jacobian product (vijp). While the operator itself is not typically named explicitly, closely related inverse-Jacobian computations appear in several adjacent lines of work. In particular, inverse accumulation–mode automatic differentiation formulates gradient propagation via inverse-Jacobian applications rather than standard vector–Jacobian products, providing a principled alternative to reverse-mode AD [Pearlmutter and Siskind, 2026]. Similarly, implicit differentiation methods such as Deep Equilibrium (DEQ) [Bai et al., 2019] models rely on solving linear systems involving Jacobians, where inverse-Jacobian products are central to computing gradients without unrolling the forward computation. In contrast to these approaches, which typically rely on iterative or approximate solvers, our *vijp* operator enables exact and efficient forward recovery of cotangents in submersive networks, forming the basis of the Moonwalk algorithm.

3 BACKGROUND

3.1 Notation

We consider a neural network $f_\theta : \mathcal{X} \to \mathcal{Y}$ composed of L sequential layers with parameters $\theta = \{\theta_i\}_{i=1}^{L}$, where $\theta_i \in \mathbb{R}^{d_i}$ denotes the parameters of layer i. Let $x_0 \in \mathcal{X}$



(a) Moonwalk: Forward pass (Phase I)

(b) Backprop: Forward pass (Phase I)

(c) Moonwalk: Computing h_0 (Phase II)

(d) Backprop: Computing gradients (Phase II)

(e) Moonwalk: Final gradients (Phase III)

Legend

Figure 1: The computation flow in Moonwalk (left column) and Backprop (right column). Backprop typically stores all activations from Phase I to Phase II, to facilitate the computation of both input cotangents and parameter gradients via *vjp* operators. In contrast, Moonwalk stores only the subset $\hat{x}_i$ of residuals that is needed to compute input cotangents.

denote the input to the network. The output of layer $i = 1, \ldots, L$ is given by

$$x_i = f_i(x_{i-1}; \theta_i) \in \mathbb{R}^{n_i},$$

where n_i is the dimensionality of the layer's output. Let $J_\theta(x_0) = J(f_\theta(x_0))$ be a scalar-valued loss function. Our goal is to compute the gradient $\nabla_\theta J_\theta$ for use in gradient-based optimization.

In convolutional layers, we distinguish between abstract vector notation (as above) and structured tensor notation. Specifically, we represent the input to a convolutional layer as a tensor $x \in \mathbb{R}^{\mathbf{n} \times m}$, where $\mathbf{n} \in \mathbb{N}^d$ denotes the spatial shape and m the number of input channels. Bold symbols $\mathbf{i}, \mathbf{k}, \mathbf{p}, \mathbf{s}$ denote multi-dimensional spatial values (respectively, element coordinates, kernel sizes, padding, and stride), while c, c' index input and output channels, respectively. Outputs of convolutional layers are similarly denoted $x' \in \mathbb{R}^{\mathbf{n}' \times m'}$.

Throughout, we refer to the Jacobian–vector product, vector–Jacobian product, and vector–inverse-Jacobian product as *jvp*, *vjp*, and *vijp*, respectively:

$$\mathrm{jvp}(f, x, u) = (\partial f / \partial x)\, u, \tag{1}$$

$$\mathrm{vjp}(f, x, v) = v\, (\partial f / \partial x)\,, \tag{2}$$

$$\mathrm{vijp}(f, x, v) = v\, (\partial f / \partial x)^+\,, \tag{3}$$

and similarly with respect to the parameters θ in place of the input x, where above u is a tangent column vector, v is a cotangent row vector, and $(\cdot)^+$ denotes any right-inverse of the Jacobian matrix. While *jvp* and *vjp* are standard primitives in autodiff frameworks such as JAX [Bradbury et al., 2018], *vijp* is a custom operator introduced in this work (see Appendix 2).

3.2 Preliminaries

The parameter gradient of layer $i = 1, \ldots, L$ can be written using the chain rule as

$$\frac{\partial J}{\partial \theta_i} = \frac{\partial J}{\partial x_L} \left(\prod_{j=L}^{i+1} \frac{\partial x_j}{\partial x_{j-1}} \right) \frac{\partial x_i}{\partial \theta_i}. \tag{4}$$

While Backprop computes the product in (4) from left to right, forward-mode differentiation is an alternative approach that computes it from right to left. The suffix of the product can be computed during the forward execution of the function, such that, unlike in Backprop, no residuals need to be stored. On the other hand, while the prefixes are vectors of dimension n_j that can be computed using *vjp*, the suffixes are matrices of dimensions $n_j \times d_i$. To avoid storing these matrices in memory, they are commonly computed column-by-column using *jvp*, however this method's time complexity is significantly larger than Backprop's (see Table 1).

Definition 1 (Submersion)**.** *A differentiable function $f : \mathcal{X} \to \mathcal{Y}$ is a* submersion *if its differential $\mathrm{d}f(x)$ is surjective for all $x \in \mathcal{X}$.*

This general definition from differential topology implies, in the special case of real vector spaces, that the Jacobian ${}^{\partial f}/_{\partial x} \in \mathbb{R}^{n' \times n}$ is right-invertible, which requires $n' \leq n$ and full row rank for all x. We call a neural network *submersive* if each of its layers is a submersion with respect to its input, for all valid parameters. Note that all invertible networks are submersive, but the converse is not true. We also emphasize that the surjectivity of a Jacobian implies that the layer has a trivial cokernel. In a case when a Jacobian is not surjective, in order to compute *vijp*, we need to store additional information that is required to reconstruct the cotangent (see Section 5.1).

4 MOONWALK

4.1 Overview

Moonwalk, in its main variant that mixes the forward and reverse modes, executes gradient computation in three phases (Fig. 1):

Phase I: Memory-efficient forward pass. In the first phase, Moonwalk performs a forward pass while storing only a subset of residuals, specifically those required to compute the loss gradients (cotangents) with respect to the input of each layer.

For commonly used layer types, such as dense and convolutional layers, this subset of stored values is substantially smaller than the complete set of residuals required to also compute gradients with respect to the parameters. For non-submersive layers, we additionally store minimal information required to reconstruct cotangents (Section 5.1). In practice, this phase requires about $2 - 3\times$ less memory than the first phase of Backprop.

Phase II: Input cotangent recovery. The second phase is very similar to the second phase of Backprop and does not add any memory overhead beyond what was stored in Phase I. Cotangent vectors with respect to the input are computed by propagating the loss gradients backwards through the network, similar to traditional backpropagation, but retracing only the computation path from the loss back to the input and ignoring gradients with respect to parameters.

Phase III: Parameter gradients via Forward. We sequentially traverse the network in the forward direction, computing each layer's activation together with its **output cotangent** and the gradients with respect to the layer's **parameters**.

The output cotangent is obtained from the layer's **input cotangent** using our custom operator, the **vector–inverse-Jacobian product** (*vijp*), which efficiently applies the right-inverse of the layer's Jacobian without materializing the full matrix. When the Jacobian has a non-trivial cokernel and lacks a right-inverse, we reconstruct the output cotangent using stored information from **Phase I** (see Section 5.1). Finally, the gradient with respect to the layer's **parameters** is computed via a standard vector–Jacobian product (*vjp*).

Pure-forward Moonwalk differs from the above mixed-mode variant by replacing Phases I and II with forward-mode computation of the input cotangent, avoiding any residual storage at the cost of increased computation.

Regardless of how the input cotangent is obtained, Moonwalk needs nothing more for its Phase III in submersive networks, making the memory overhead of this phase constant in the network depth. Mixed-mode Moonwalk can also be applied to non-submersive layers through **gradient checkpointing** in Phase II, where only the cotangents required for *vjp* with respect to the parameters are checkpointed, avoiding full residual storage. This strategy can be further refined into **fragmental cotangent checkpointing**, a memory-efficient variant that stores minimal cotangent fragments, enabling parallel reconstruction in Phase III (Section 5.1).

Because submersive networks are a much larger superset of invertible networks, Moonwalk is usable in some architectures where inversion-based methods are not, such as networks that reduce the dimensionality of their input or have non-injective activation functions.

Moonwalk can also be combined with checkpointing, also known as rematerialization, to reduce the effective network depth in Phase II.

4.2 The Moonwalk Identity

In order to benefit from the memory advantage of forward-mode gradient computation, while keeping the time complexity similar to that of Backprop and avoiding the introduction of noisy gradients through projection, we first restrict our attention to the class of submersive networks, in which the Jacobian of each layer with respect to its input is guaranteed to be right-invertible (see Definition 1). Then we can rewrite layer i's parameter gradient of the loss, for $i = 1, \dots, L$, as

$$g_i \coloneqq \frac{\partial J}{\partial \theta_i} = \frac{\partial J}{\partial x_i}\frac{\partial x_i}{\partial \theta_i} = \frac{\partial J}{\partial x_i}\frac{\partial x_i}{\partial x_0}\left(\frac{\partial x_i}{\partial x_0}\right)^+ \frac{\partial x_i}{\partial \theta_i} \tag{5}$$

$$= \frac{\partial J}{\partial x_0}\left(\prod_{j=i}^{1}\frac{\partial x_j}{\partial x_{j-1}}\right)^+ \frac{\partial x_i}{\partial \theta_i} \tag{6}$$

$$= \frac{\partial J}{\partial x_0}\prod_{j=1}^{i}\left(\frac{\partial x_j}{\partial x_{j-1}}\right)^+ \frac{\partial x_i}{\partial \theta_i}. \tag{7}$$

Given the input cotangent $\frac{\partial J}{\partial x_0}$, we can recurse on prefixes of (7) to compute each layer's parameter gradient in forward-mode. To this end, denote the input cotangent of layer $i = 1, \dots, L$ by

$$h_i \coloneqq \frac{\partial J}{\partial x_i} = \frac{\partial J}{\partial x_0}\prod_{j=1}^{i}\left(\frac{\partial x_j}{\partial x_{j-1}}\right)^+, \tag{8}$$

and the network's input cotangent by $h_0 \coloneqq \frac{\partial J}{\partial x_0}$. This formulation allows us to compute each cotangent h_i and parameter gradient g_i, as we evaluate $f_i(x_{i-1}; \theta_i)$ in a forward pass, from only the layer input cotangent h_{i-1} and the local Jacobians, without any residuals:

$$h_i = h_{i-1}\left(\frac{\partial x_i}{\partial x_{i-1}}\right)^+ = \mathrm{vijp}(f_i, x_{i-1}, h_{i-1}), \tag{9}$$

and from (7) we have

$$g_i = h_i \frac{\partial x_i}{\partial \theta_i} = \mathrm{vjp}(f_i, \theta_i, h_i). \tag{10}$$

Assuming that we have the input cotangent h_0, we can construct the parameter gradient for each layer on-the-fly by the two operators in (9) and (10) and store only $h_i \in \mathbb{R}^{n_i}$ temporarily for the next layer's computation. Despite the general existence of multiple right-inverses, the result is unique in submersive networks because each input cotangent is in the row-space of its layer's Jacobian. The complete procedure is given in Algorithm 1 and illustrated in Fig. 1e.

In the remainder of this section we describe the computation of h_0 in either reverse- or forward-mode, discuss the trade-offs of each variant, and introduce techniques offering additional improvements.

Algorithm 1 Moonwalk

for each gradient step with input x_0 **do**
  Compute $h_0 \leftarrow \frac{\partial J}{\partial x_0}$
  for $i = 1, \dots, L$ **do**
    $x_i \leftarrow f_i(x_{i-1}; \theta_i)$
    $h_i \leftarrow \mathrm{vijp}(f_i, x_{i-1}, h_{i-1})$
    $g_i \leftarrow \mathrm{vjp}(f_i, \theta_i, h_i)$
    Apply g_i to θ_i
  end for
end for

4.3 Mixed-Mode Moonwalk

A key component of Moonwalk is the computation of the input cotangent h_0. In the mixed-mode variant, we obtain h_0 using reverse-mode differentiation. At first glance, this may appear to negate the memory benefits of our approach. However, the crucial observation is that computing h_0 only requires traversing the portion of the computation graph that contributes to the input cotangent, rather than the full set of parameter-dependent paths.

In particular, for many layers—such as convolutions—the input cotangent depends only on the layer parameters (e.g., kernel weights) and the output cotangent, but not on the full set of stored activations required to compute parameter gradients. As a result, reverse-mode computation of h_0 can be carried out without storing the residuals needed for parameter updates. This decoupling allows Moonwalk to retain the efficiency of reverse-mode for computing h_0, while avoiding its primary memory bottleneck.

Once h_0 is obtained, all parameter gradients are recovered in a single forward sweep using the *vijp* operator, eliminating the need to store intermediate activations entirely in submersive networks. This hybrid strategy combines the strengths of reverse- and forward-mode differentiation: reverse-mode efficiently computes a single cotangent, while forward-mode reconstructs all remaining gradients with reduced memory overhead.

From a systems perspective, this design also contrasts with standard backpropagation, where parameter gradients are typically accumulated across the entire network before any updates are applied. In Moonwalk, parameter gradients are computed sequentially during the forward sweep and need not be stored simultaneously, further reducing peak memory usage. This distinction is particularly impactful in architectures with large parameter counts, where gradient storage can dominate memory consumption.

Finally, we note that alternative choices for the reconstruction "seed" can be used in place of h_0, such as the

first-layer parameter gradient g_0 or its output cotangent h_1, leading to modest efficiency gains in some architectures. These variants highlight the flexibility of the mixed-mode formulation and promise further opportunities for optimizing memory–compute trade-offs.

4.4 Pure-Forward Moonwalk

An alternative way to obtain the input cotangent h_0 is to compute it entirely in forward-mode. In this setting, each component $h_{0,j}$ is obtained by propagating a standard basis vector e_j through the network using Jacobian–vector products (*jvp*). Concretely, this corresponds to the recursion

$$\frac{\partial x_i}{\partial x_0} e_j = \mathrm{jvp}(f_i, x_{i-1}, \tfrac{\partial x_{i-1}}{\partial x_0} e_j),$$

evaluated independently for each input dimension $h_{0,j}$.

This approach eliminates the need for reverse-mode entirely and requires no storage of intermediate activations. Instead, only the components of h_0 are retained, making the method maximally memory-efficient. As in the mixed-mode variant, once h_0 is computed, all parameter gradients are recovered in a single forward sweep via the *vijp* operator.

However, this benefit comes at a computational cost: constructing h_0 requires one forward-mode pass per input dimension. As a result, the method scales linearly with the input size and can become prohibitively expensive in high-dimensional settings. Consequently, pure-forward Moonwalk is most suitable when the input dimension is small or when memory constraints dominate compute considerations. In such regimes, it provides a simple and fully forward-mode alternative that highlights the conceptual generality of the Moonwalk framework.

4.5 Residual Impact

A central advantage of Moonwalk is its reduced reliance on storing intermediate residuals. In contrast to Backprop, which retains all activations required to compute vector–Jacobian products, Moonwalk stores only a subset sufficient for backpropagating and reconstructing input/output cotangents. As illustrated in Fig. 1a, this corresponds to storing compressed representations $\hat{x}_i$ rather than full activations.

This reduction can be substantial in certain common architectures. For example, in a sequence of convolutional layers with LeakyReLU activations, Backprop must store the full input to each convolution to compute parameter gradients. In contrast, Moonwalk only requires storing the sign of each activation to evaluate the LeakyReLU *vjp*, which dramatically reduces memory usage. In practice, this representation can be up to 16–32× smaller than storing full-precision activations.

More generally, this shift from activation storage to compact structural summaries enables significant memory savings over Backprop, particularly in networks with large feature maps or high-resolution inputs. As a result, Moonwalk can scale to deeper architectures or larger batch sizes under the same memory budget.

5 SUBMERSIVE CONVOLUTIONAL LAYERS

Convolutional layers are structured linear operators that can use relatively few parameters to transform large inputs while respecting (typically translational) symmetries. As such, they are ideal for the performance gains offered by Moonwalk, both because Backprop residuals are large and because the *vijp* operator can leverage the convolutional structure. The following lemma provides sufficient conditions for a convolutional layer to be submersive.

Lemma 1 (Sufficient Conditions for Submersive Convolutional Layers)**.** *A channel-wise convolution with input $x \in \mathbb{R}^{\mathbf{n} \times m}$, kernel $w \in \mathbb{R}^{\mathbf{k} \times m \times m'}$, output $x' \in \mathbb{R}^{\mathbf{n}' \times m'}$, padding $\mathbf{p}$, and stride $\mathbf{s}$, defined by*

$$x'_{\mathbf{i}',c'} = \sum_{\mathbf{j},c} w_{\mathbf{j},c,c'} \cdot x_{\mathbf{s}\mathbf{i}'+\mathbf{j}-\mathbf{p},c}, \tag{11}$$

is submersive—i.e., its input–output Jacobian is right-invertible—if the following conditions hold:

(i) ***Spatial bounds:*** $\mathbf{k} > \mathbf{p}$, $\mathbf{s} > \mathbf{p}$, *and* $\mathbf{n} > \mathbf{s}(\mathbf{n}' - 1)$*;*

(ii) ***Channel-wise triangularity:*** $w_{\mathbf{p},c,c'} = 0$ *for all $c < c'$ (implying $m' \leq m$); and*

(iii) ***Diagonal support:*** $w_{\mathbf{p},c',c'} \neq 0$ *for all $c' \leq m'$.*

Proof outline. We provide a constructive proof by uniquely recovering the output cotangent of the convolutional layer from its input cotangent. This construction will then also serve as an efficient implementation of the *vijp* operator for submersive convolutional layers.

In reverse-mode, the input cotangent h can be computed from the output cotangent h' as:

$$\begin{aligned} h &= \mathrm{TransposeConv}(h', w, \mathbf{n}, \mathbf{p}, \mathbf{s}) \\ &= \mathrm{Conv}(\bar{h}', w^\top, \mathbf{k} - \mathbf{p} - \mathbf{1}, \mathbf{1}) \qquad (12) \\ h_{\mathbf{i},c} &= \sum_{\mathbf{j},c'} w_{\mathbf{j},c,c'} \cdot \bar{h}'_{\mathbf{i}+\mathbf{p}-\mathbf{j},c'}, \qquad (13) \end{aligned}$$

where $\bar{h}'$ is obtained from h' by dilating it by $\mathbf{s}$ (i.e., $\bar{h}'_{\mathbf{si}',c'} = h'_{\mathbf{i}',c'}$ and zero elsewhere), and padding the end of each spatial dimension with $\left\{\frac{\mathbf{n}+2\mathbf{p}-\mathbf{k}}{\mathbf{s}}\right\}\mathbf{s}$ zeros, where $\{\cdot\}$ denotes the fractional part. Kernel transposition $w^\top$ is a reversal of kernel elements along spatial dimensions and a transposition of channel dimensions.

When the convolution is submersive, the *vjp* operator defined in (13) can be uniquely inverted to recover h' from h. The sufficient conditions in the lemma are aimed at simplifying this *vijp* operator as Gaussian elimination with fixed indices. Specifically, we prove in Appendix 9 that each $h'_{\mathbf{i}',c'}$ is the leading element on the right-hand side of (13) for $h_{\mathbf{si}',c'}$, with coefficient $w_{\mathbf{p},c',c'} \neq 0$. It follows that

$$w_{\mathbf{p},c',c'} h'_{\mathbf{i}',c'} = h_{\mathbf{si}',c'} - \text{TransposeConv}(h', \tilde{w}, \mathbf{n}, \mathbf{p}, \mathbf{s})_{\mathbf{si}',c'}, \quad (14)$$

where $\tilde{w}$ is identical to w except that $\tilde{w}_{\mathbf{p},c',c'} = 0$ for all c'. Because all nonzero h' terms needed to compute $h'_{\mathbf{i}',c'}$ have indices smaller than $(\mathbf{i}', c')$, we can parallelize the computation over all entries with the same total index sum $t = \sum \mathbf{i}' + c'$. Pseudocode for a highly efficient parallel implementation is provided in Appendix 2. This implementation enables the simultaneous computation of all spatial cotangent values, offering significant parallelization benefits.

5.1 Fragmental Gradient Checkpointing

To increase the flexibility of supported architectures, we aim to incorporate non-submersive layers. The challenge, however, is that the output cotangent of such a layer may not be uniquely recoverable from its input cotangent, while storing the full output cotangent can be memory-intensive. To address this, we propose fragmental checkpointing: instead of retaining the entire output cotangent, we selectively store a minimal subset of its elements that is sufficient to reconstruct the remaining elements through recursive elimination.

As an example, in a channel-last 1D convolution with kernel weights $\mathbf{w} \in \mathbb{R}^{k \times m \times m'}$, $p = 1$, and $s = 1$, an output cotangent element $h'_{i',c'}$ can be computed as

$$h'_{i',c'} = h_{i'-1,c'} - \sum_{c > c'} w_{0,c,c'} \cdot h'_{i',c'} - \sum_{\substack{j \geq 1 \\ c \geq c'}} w_{j,c,c'} \cdot h'_{i'-j,c'} \quad (15)$$

assuming that

- $w_{0,c,c'} = 0$ for all $c < c'$; and
- $w_{0,c',c'} = 1$.

Recovering all channels $h'_{i',\cdot}$ at index i' therefore requires, in addition to the input cotangent elements $h_{i'-1,\cdot}$, the previous $k-1$ output elements $h'_{i'-j,\cdot}$ (in all channels). The full derivation is in Appendix 10.

More generally, fragmental checkpointing can be applied in blocks. For block size B, storing just the first $k-1$ entries per block allows reconstruction of the rest of the block. For example, given a residual tensor shaped 1024×64 (time steps $\times$ channels), with $k = 3$ and $B = 4$, only half of each block (2 elements) needs to be stored, reducing memory from 1024×64 to 512×64. Increasing the block size to $B = 16$ with the same k reduces the memory cost to 1/8 of full checkpointing. Additional implementation details are in Appendix 3.

Table 1 summarizes the order of growth of time and memory in the methods we compare, and the next section evaluates them empirically.

6 EXPERIMENTS

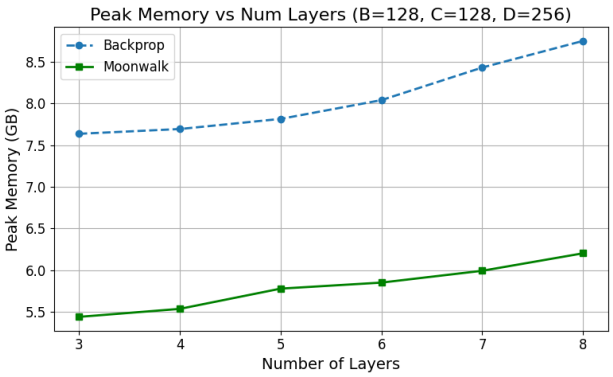
We empirically evaluate the memory and computational performance of Moonwalk against Backprop on deep residual convolutional networks. Our setup is essentially identical to the original Resnet He et al. [2015] on Imagenet Deng et al. [2009]. Our experiments are designed to isolate the memory–computation tradeoffs introduced by Moonwalk in both fully submersive networks and under fragmental checkpointing. We also showcase that our implemented convolutional *vijp* operator does not introduce a computational overhead while leading to significant memory savings.

6.1 Experimental Setup

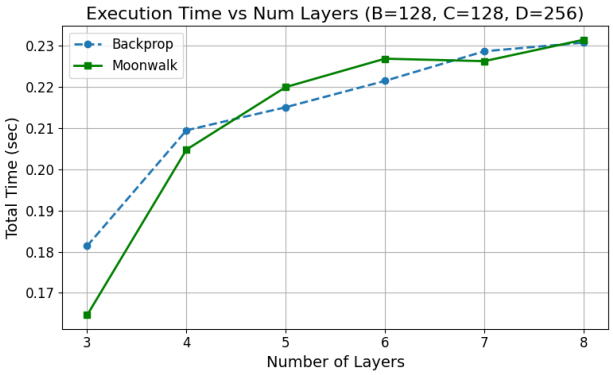
All experiments are conducted on a single NVIDIA RTX 3090 GPU (24 GB) using JAX with `jaxlib` version 0.4.30 and CUDA 12.6. We measure peak GPU memory usage using `jax.device.memory_stats()`, and we report total wall-clock execution time for a single forward and backward pass on one batch (batch size 128), after the initial JIT compilation. We compare Moonwalk against Backprop and Backprop with rematerialization.

6.2 Fully Parallel Submersive 2D CNN

We begin with a fully submersive 2D convolutional architecture. We simulate a typical setup of the most common residual CNNs architectures trained on Imagenet Deng et al. [2009]. An input of size $256 \times 256 \times 3$ is first upsampled to $256 \times 256 \times 128$ channels. Each subsequent layer applies a $3 \times 3 \times 128 \times 128$ convolution with stride 2 and padding 1, allowing a fully parallelizable *vijp* operator and halving the spatial resolution.



(a) 2D - GPU memory vs. # of layers.



(b) 2D - Total time vs. # of layers.

Figure 2: Comparison of Backprop and Moonwalk as network depth increases in a 2D CNN. (a) Moonwalk reduces peak memory by about 30% compared to Backprop. (b) Both methods incur similar runtimes, demonstrating that Moonwalk's memory savings come at no extra compute cost.

All activation functions are LeakyReLU, and the final layer performs max pooling and projects the feature map to a scalar.

This configuration enables efficient memory usage, as illustrated in Fig. 2a. Specifically, Moonwalk reduces the memory footprint from 9.5 GB to 6.6 GB in the 8-block configuration, while maintaining comparable computational performance as illustrated in Fig. 2b. This corresponds to a 30% reduction in memory without sacrificing throughput. These savings are primarily due to the fully parallelizable *vijp* operator, which computes all intermediate values simultaneously without sequential dependency.

6.3 Fragmental Checkpointing

To extend Moonwalk to non-submersive layers, we introduce *fragmental gradient checkpointing*, which leverages the structure of convolutional operators.

For implementation simplicity, we illustrate this approach on 1D convolutional networks, but note that it is equally applicable to higher spatial dimensions. In non-submersive layers, the Jacobian has a non-trivial cokernel, preventing exact recovery of cotangents through *vijp* alone. To address this, we store a minimal subset of cotangent fragments during the backward pass, which are later used to reconstruct the missing gradient components during the forward sweep.

Our experimental setup mirrors the 2D case, but preserves spatial resolution at each layer by setting the stride and padding to 1. The 2048×3 input is upsampled to 2048×256, and kept at this shape across layers. While this configuration breaks submersivity, fragmental checkpointing enables exact cotangent recovery.

This approach provides a flexible memory–compute trade-off. With block size 4—where half of the cotangent components are stored and the rest recomputed in parallel—memory usage is reduced from 14 GB to 7 GB (Fig. 3a), a 50% reduction. Increasing the block size further decreases memory usage, at the cost of recomputing a larger number of unstored elements, leading to higher runtime (Fig. 3b).

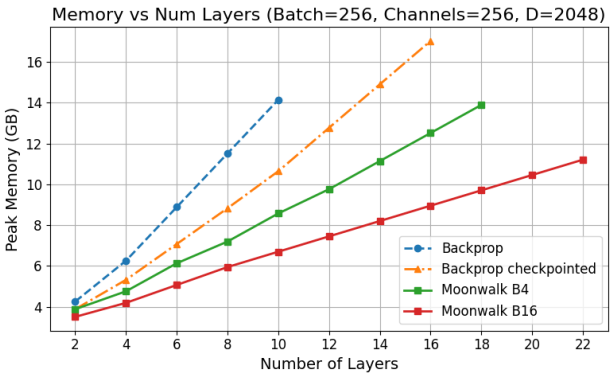
Overall, fragmental checkpointing enables Moonwalk to scale to deeper networks under tighter memory budgets. In our experiments, standard backpropagation fails beyond 10 layers and checkpointed backpropagation reaches up to 16 layers, whereas Moonwalk with block size 16 successfully trains networks up to 22 layers.

6.4 Constrained Convolutions

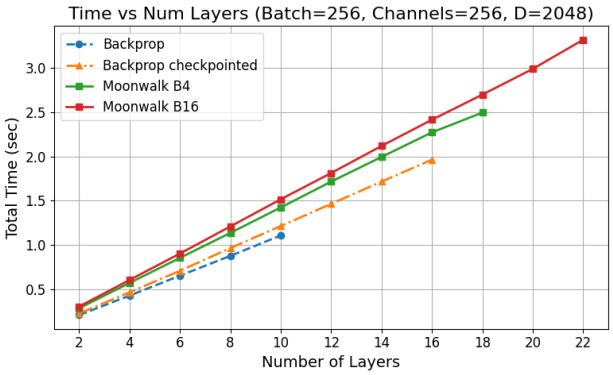
To enable efficient computation of the *vijp*, we parametrize convolutional layers according to the submersivity conditions in Lemma 1. Concretely, this imposes a structured constraint on the convolutional weights—e.g., an upper-triangular channel-wise form—that guarantees the existence of efficient right-inverse Jacobian operations.

A natural concern is whether such constraints limit model expressivity. To evaluate this, we compare constrained and unconstrained models on a 10-layer 2D residual network, where all convolutions in the constrained variant follow the submersive parameterization. Despite the imposed structure, both models achieve comparable performance, reaching approximately 90% test accuracy (Fig. 4).

These results suggest that the proposed constraints do not significantly degrade representational power in practice, while enabling the efficient *vijp* computations required by Moonwalk. This highlights a favorable trade-off between architectural structure and computational efficiency, and supports the viability of submer-



(a) 1D Fragmental - GPU memory vs. # of layers.



(b) 1D Fragmental - Total time vs. # of layers.

Figure 3: Evaluation of Moonwalk with fragmental checkpointing on a 1D CNN. (a) At fixed block size B=4, Moonwalk reduces memory usage by up to 50% compared to Backprop. (b) Varying the block size reveals a trade-off: bigger blocks require more recomputation and increase runtime.

sive parameterizations in realistic settings.

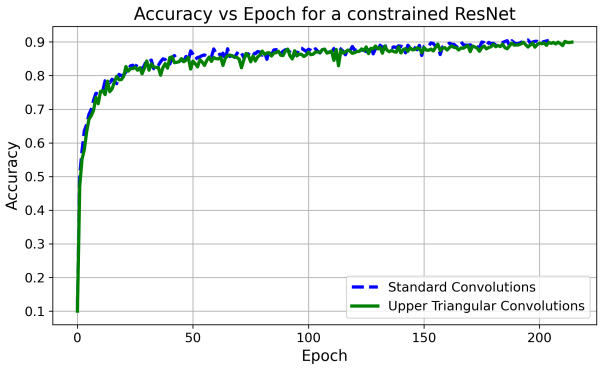


Figure 4: Accuracy comparison between upper-triangular convolutions (green) and standard convolutions (blue). Both setups converge to ∼90% accuracy.

7 LIMITATIONS

The primary limitation of our approach is the need to implement the *vijp* operator manually for each layer class, which is more involved than relying on the automatically generated *vjp* operators provided by modern AD frameworks. In cases where constructing an efficient *vijp* is difficult, this overhead can be mitigated through partial checkpointing of cotangents.

Despite this requirement, we find that for many practical architectures—particularly convolutional networks—efficient *vijp* implementations are attainable, yielding favorable memory–compute trade-offs while remaining compatible with standard model designs.

8 CONCLUSION AND FUTURE WORK

We introduced **Moonwalk**, a memory-efficient gradient computation framework with two variants—pure-forward and mixed-mode—that, in some architectures, achieves up to $\times 2$ lower memory usage with comparable runtime to backpropagation. By eliminating the need to store full residuals, Moonwalk addresses a central limitation of reverse-mode AD, making it well-suited for memory-constrained settings.

We derived sufficient conditions for convolutional layers to be submersive and used this structure to design an efficient, parallelizable *vijp* operator with no additional computational overhead. These results demonstrate that exact forward-mode gradient computation can be practical in dense architectures when paired with appropriate structural assumptions.

While our analysis focuses on submersive layers, Moonwalk naturally extends to general architectures via partial cotangent storage. Our fragmental gradient checkpointing further reduces memory by enabling selective storage and parallel reconstruction of gradients, highlighting a broader design space that blends forward- and reverse-mode differentiation.

Future work includes extending Moonwalk to attention-based architectures and integrating it more tightly into modern deep learning frameworks.

Acknowledgements

Authors Krylov and Karamzade were supported by Hasso Plattner Foundation Fellowships.

References

Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. Deep equilibrium models, 2019. URL `https://arxiv.`

org/abs/1909.01377.

Atılım Güneş Baydin, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without backpropagation, 2022.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Samuel Rota Bulo, Lorenzo Porzi, and Peter Kontschieder. In-place activated batchnorm for memory-optimized training of dnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5639–5647, 2018.

Yuxuan Cai, Yizhuang Zhou, Qi Han, Jianjian Sun, Xiangwen Kong, Jun Li, and Xiangyu Zhang. Reversible column networks, 2023. URL https://arxiv.org/abs/2212.11696.

Ping Chen, Wenjie Zhang, Shuibing He, Weijian Chen, Siling Yang, Kexin Huang, Yanlong Yin, Xuan Zhan, Yingjie Gu, Zhuwei Peng, Yi Zheng, Zhefeng Wang, and Gang Chen. Optimizing large model training through overlapped activation recomputation, 2025. URL https://arxiv.org/abs/2406.08756.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848. URL https://ieeexplore.ieee.org/document/5206848.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Louis Fournier, Stéphane Rivaud, Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Can forward gradient match backpropagation?, 2023.

Eshed Gal, Moshe Eliasof, Javier Turek, Uri Ascher, Eran Treister, and Eldad Haber. Reversing large language models for efficient training and fine-tuning, 2025. URL https://arxiv.org/abs/2512.02056.

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *Advances in neural information processing systems*, 29, 2016.

Julia Gusak, Xunyi Zhao, Théotime Le Hellard, Zhe Li, Lionel Eyraud-Dubois, and Olivier Beaumont. HiRemate: Hierarchical approach for efficient rematerialization of neural networks. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 21418–21443. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/gusak25a.html.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Jörn-Henrik Jacobsen, Arnold WM Smeulders, and Edouard Oyallon. i-revnet: Deep invertible networks. In *International Conference on Learning Representations*, 2018.

Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *International conference on machine learning*, pages 1627–1635. PMLR, 2017.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *CoRR*, abs/1910.02653, 2019. URL http://arxiv.org/abs/1910.02653.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Advances in neural information processing systems*, 31, 2018.

Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022. URL https://arxiv.org/abs/2205.05198.

Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient rematerialization for deep networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.

neurips.cc/paper_files/paper/2019/file/ffe10334251de1dc98339d99ae4743ba-Paper.pdf.

Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. Mario: Near zero-cost activation checkpointing in pipeline parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25, page 197–211, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714436. doi: 10.1145/3710848.3710878. URL https://doi.org/10.1145/3710848.3710878.

Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger B Grosse. Reversible recurrent neural networks. *Advances in Neural Information Processing Systems*, 31, 2018.

Karttikeya Mangalam, Haoqi Fan, Yanghao Li, Chao-Yuan Wu, Bo Xiong, Christoph Feichtenhofer, and Jitendra Malik. Reversible vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10830–10840, 2022.

James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade: Second Edition*, pages 479–535. Springer, 2012.

Georgii Sergeevich Novikov, Daniel Bershatsky, Julia Gusak, Alex Shonenkov, Denis Valerievich Dimitrov, and Ivan Oseledets. Few-bit backward: Quantized gradients of activation functions for memory footprint reduction. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26363–26381. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/novikov23a.html.

Barak A. Pearlmutter and Jeffrey Mark Siskind. *Automatic Differentiation: Inverse Accumulation Mode*, page 1–12. Society for Industrial and Applied Mathematics, January 2026. ISBN 9781611979039. doi: 10.1137/1.9781611979039.1. URL http://dx.doi.org/10.1137/1.9781611979039.1.

Mengye Ren, Simon Kornblith, Renjie Liao, and Geoffrey Hinton. Scaling forward gradient with local losses. In *The Eleventh International Conference on Learning Representations*, 2022.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

David Silver, Anirudh Goyal, Ivo Danihelka, Matteo Hessel, and Hado van Hasselt. Learning by directional gradient descent. In *International Conference on Learning Representations*, 2021.

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989. doi: 10.1162/neco.1989.1.2.270.

Xingzi Xu, Amir Tavanaei, Kavosh Asadi, and Karim Bouyarmane. Activation sharding for scalable training of large models. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=kQCuMcEneq.

Zihao Zeng, Chubo Liu, Xin He, Juan Hu, Yong Jiang, Fei Huang, Kenli Li, and Wei Yang Bryan Lim. Autohete: An automatic and efficient heterogeneous training system for llms, 2025. URL https://arxiv.org/abs/2503.01890.

Xunyi Zhao, Théotime Le Hellard, Lionel Eyraud, Julia Gusak, and Olivier Beaumont. Rockmate: an efficient, fast, automatic and generic tool for re-materialization in pytorch, 2023. URL https://arxiv.org/abs/2307.01236.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [Yes]
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes]
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Yes / in supplemental material]

2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. [Yes]
   (b) Complete proofs of all theoretical results. [Yes]
   (c) Clear explanations of any assumptions. [Yes]

3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [Yes in supplemental]
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Yes]
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [Not Applicable]
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator If your work uses existing assets. [Not Applicable]
   (b) The license information of the assets, if applicable. [Not Applicable]
   (c) New assets either in the supplemental material or as a URL, if applicable. [Not Applicable]
   (d) Information about consent from data providers/curators. [Not Applicable]
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [Not Applicable]

5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. [Not Applicable]
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [Not Applicable]
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [Not Applicable]

Appendix

9 Submersive Convolutional Layers Proof

Lemma 1 (Submersion Conditions for Convolutional Layers)**.** *A channel-wise convolution with input* $x \in \mathbb{R}^{\mathbf{n}\times m}$*, kernel* $w \in \mathbb{R}^{\mathbf{k}\times m\times m'}$*, output* $x' \in \mathbb{R}^{\mathbf{n}'\times m'}$*, padding* $\mathbf{p}$*, and stride* $\mathbf{s}$*, defined by*

$$x'_{\mathbf{i}',c'} = \sum_{\mathbf{j},c} w_{\mathbf{j},c,c'} \cdot x_{\mathbf{s}\mathbf{i}'+\mathbf{j}-\mathbf{p},c}, \tag{16}$$

is submersive—i.e., its input–output Jacobian is right-invertible—if the following conditions hold:

(i) ***Spatial bounds:*** $\mathbf{k} > \mathbf{p}$*,* $\mathbf{s} > \mathbf{p}$*, and* $\mathbf{n} > \mathbf{s}(\mathbf{n}'-1)$*;*

(ii) ***Channel-wise triangularity:*** $w_{\mathbf{p},c,c'} = 0$ *for all* $c < c'$ *(implying* $m' \le m$*); and*

(iii) ***Diagonal support:*** $w_{\mathbf{p},c',c'} \neq 0$ *for all* $c' \le m'$*.*

Proof. As outlined in the main text, reverse-mode computes the input cotangent h from the output cotangent h' as

$$\begin{aligned} h &= \mathrm{TransposeConv}(h', w, \mathbf{n}, \mathbf{p}, \mathbf{s}) \\ &= \mathrm{Conv}(\bar{h}', w^\top, \mathbf{k}-\mathbf{p}-\mathbf{1}, \mathbf{1}) \qquad (17) \\ h_{\mathbf{i},c} &= \sum_{\mathbf{j},c'} w_{\mathbf{j},c,c'} \cdot \bar{h}'_{\mathbf{i}+\mathbf{p}-\mathbf{j},c'}, \qquad (18) \end{aligned}$$

where $\bar{h}'$ is obtained from h' by dilating it by $\mathbf{s}$ (i.e., $\bar{h}'_{\mathbf{s}\mathbf{i}',c'} = h'_{\mathbf{i}',c'}$ and zero elsewhere), and padding the end of each spatial dimension with $\left\{\frac{\mathbf{n}+2\mathbf{p}-\mathbf{k}}{\mathbf{s}}\right\}\mathbf{s}$ zeros, where $\{\cdot\}$ denotes the fractional part. The transposed kernel $w^\top$ is obtained by reversing the kernel elements along spatial dimensions and transposing the channel dimensions.

The conditions in the lemma were selected to facilitate simple and fast Gaussian elimination. The ability to always recover h' from $h = h'\,(\partial f/\partial x)$ then also serves to prove that the Jacobian is right-invertible, as required.

The elimination is by lexicographic order of the index $\mathbf{i}', c'$ of h', with each $h'_{\mathbf{i}',c'}$ recovered by the equation for $h_{\mathbf{s}\mathbf{i}',\mathbf{c}'}$. It is therefore enough to show that $h'_{\mathbf{i}',c'}$ is the lexicographically largest element on the right-hand side of (18) for $h_{\mathbf{s}\mathbf{i}',\mathbf{c}'}$ with non-zero coefficient.

First, for any $0 \le \mathbf{i}' < \mathbf{n}'$, the index $\mathbf{s}\mathbf{i}'$ falls within the bounds of the input size $\mathbf{n}$ due to the assumption $\mathbf{n} > \mathbf{s}(\mathbf{n}'-1)$. Furthermore, note that the coefficient of $h'_{\mathbf{i}',c'} = \bar{h}'_{\mathbf{s}\mathbf{i}',c'}$ on the right-hand side of $h_{\mathbf{s}\mathbf{i}',\mathbf{c}'}$ is $w_{\mathbf{p},c',c'}$, which is within the bounds of the kernel due to $\mathbf{k} > \mathbf{p}$ and non-zero by assumption (iii).

Next, any $\bar{h}$ with any spatial index larger than $\mathbf{s}\mathbf{i}'$ is either 0 due to the dilation structure or larger by at least $\mathbf{s}$, making its corresponding w coefficient smaller than the aforementioned $\mathbf{p}$ by at least $\mathbf{s}$. However, that index falls outside the kernel matrix due to $\mathbf{s} > \mathbf{p}$.

Finally, any $\bar{h}_{\mathbf{s}\mathbf{i}',c''}$ with the same spacial index but a channel index c'' larger than c' has coefficient $w_{\mathbf{p},c',c''} = 0$ by assumption (ii). □

10 Fragmental Gradient Checkpointing

Consider a *channel-wise* 1-D convolution with stride $s = 1$, padding $p = 1$, and kernel size $k = 3$. For its reverse pass we have, for every spatial index i and output channel c',

$$h_{i,c'} \;=\; \sum_{j=0}^{k-1} \sum_{c=0}^{m-1} w_{j,c',c}\, h'_{i-j+1,c}. \tag{19}$$

Because $0 \le j \le 2$, the *largest* cotangent index that appears on the right is $h'_{i+1,c}$ (from $j = 0$), followed by $h'_{i,c}$ and $h'_{i-1,c}$, we have

$$h_{i,c'} \;=\; \sum_{c} \big(w_{0,c',c}\, h'_{i+1,c} + w_{1,c',c}\, h'_{i,c} + w_{2,c',c}\, h'_{i-1,c}\big).$$

Assuming $w_{0,c',c} = 0$ for $c > c'$ and $w_{0,c',c'} = 1$ for $0 \le c', c < m$, we can solve for the *future* cotangent $h'_{i+1,c'}$ directly as

$$\begin{aligned} h'_{i+1,c'} = h_{i,c'} &- \sum_{c<c'} w_{0,c',c}\, h'_{i+1,c} \\ &- \sum_{j=1}^{2} \sum_{c} w_{j,c',c}\, h'_{i-j+1,c}. \end{aligned} \tag{20}$$

Equation (20) shows that $h'_{i+1,c'}$ depends only on the just-computed entries $h'_{i+1,c}$ with $c < c'$, and the two earlier spatial slices $h'_{i,c'}$ and $h'_{i-1,c'}$. Hence a rolling window of three spatial slices suffices, and the update can be executed block-wise exactly as in Algorithm 3.

The above derivation generalizes to arbitrary kernel size k. In that case, the second summation in (20) extends over $j = 1, \ldots, k-1$, and we must retain at least the previous $k-1$ cotangent slices in memory.

Unlike the submersive case, the fragmental checkpointing strategy does not require the spatial stride to exceed the padding (i.e., it does not impose $s \succ p$). However, it does require retaining at least one future cotangent value h' in memory at each step to enable sequential reconstruction.

Algorithm 2 Fully Parallel VIJP Computation for 2D Convolution

Require: Gradient tensor $g \in \mathbb{R}^{H' \times W' \times C_{\text{out}}}$, kernel $\mathbf{w} \in \mathbb{R}^{k \times k \times C_{\text{out}} \times C_{\text{in}}}$, stride s, padding p

Initialize $h[i, j, c] \leftarrow 0$ for all (i, j, c)
for each output position (i, j) **in parallel do**
  for $c = 0$ to $C_{\text{in}} - 1$ **do**
    $v \leftarrow g[si, sj, 0] \,/\, \mathbf{w}[p, p, 0, c]$
    for $c' = 1$ to $C_{\text{out}} - 1$ **do**
      $v \leftarrow g[si, sj, c']$
      for $c'' = 0$ to $c' - 1$ **do**
        $v \leftarrow v - \mathbf{w}[p, p, c'', c] \cdot h[i, j, c'']$
      end for
      $h[i, j, c'] \leftarrow v$ ▷ diagonal is normalized to 1
    end for
  end for
end for
return h

Algorithm 3 Fragmental Parallel VIJP Computation for 1D Convolution

Require: Gradient tensor $g \in \mathbb{R}^{N \times C_{\text{out}}}$, kernel $\mathbf{w} \in \mathbb{R}^{k \times C_{\text{in}} \times C_{\text{out}}}$, block size B, number of blocks M, stored cotangents $h_{\text{init}} \in \mathbb{R}^{M(k-1) \times C_{\text{in}}}$

for each block $b = 0$ to $M - 1$ **in parallel do**
  for $i = k - 1$ to $B - 1$ **do**
    for $c = 0$ to $C_{\text{in}} - 1$ **do**
      $v \leftarrow g^{(b)}[i - 1, c]$
      for $j = 1$ to $k - 1$ **do**
        $v \leftarrow v - \sum_{c'=0}^{C_{\text{in}}-1} \mathbf{w}[j, c', c] \cdot h[i - j, c']$
      end for
      $h[i, c] \leftarrow v$
    end for
  end for
end for
return h

11 Complexity Analysis

While an estimate of the exact time and memory consumption of different methods for computing the gradients can greatly depend on the choice of the network's architecture and the detailed implementation, in this section we will provide an asymptotic analysis, in terms of the main architectural parameters, of the time and memory complexities of our methods and compare them with related previous works (Table 1). We omit all methods' linear dependence in time and memory on the mini-batch size. We analyze the computational complexity of the following methods:

1. **Backprop:** Throughout the forward pass (or during a checkpointed block's forward recomputation), all residuals are cached, and subsequently, during a backward pass, gradients for each layer are computed using *vjp*.
2. **Forward:** During the forward pass, complete Jacobians for each layer are computed using *jvp*. In practice, a separate forward pass is used for each column, to reuse memory for the same time complexity.
3. **ProjForward:** In projected forward [Baydin et al., 2022], parameter gradients projected in a random or predicted direction are obtained using *jvp* concurrently with the forward pass.
4. **RevBackprop:** In invertible networks [Gomez et al., 2017], no residuals need to be stored during the forward pass. In a subsequent backward pass, the output of each layer is used to compute its input via the inverse function, as well as its parameter gradient via *vjp*.
5. **Moonwalk:** The input cotangent is computed using Backprop, to reduce computation time at the expense of some memory impact (Section 4.3).
6. **Pure-Moonwalk:** Initially, the input cotangent is computed using Forward. Then parameter gradients are obtained using *vijp* and *vjp* in a second forward pass (Section 4.4).

We evaluate time based on the standard cost of matrix multiplication, i.e. the product of their two outer dimensions and shared inner dimension, without considering optimization tricks, sparse layers, or other network structures. To evaluate memory, we define $M_{x,i}$ to be the required memory to store the necessary information to compute $\partial x_i / \partial x_{i-1}$, and $M_{\theta,i}$ the added memory to also compute $\partial x_i / \partial \theta_i$. For simplicity, we assume that these values, as well as n_i and d_i, scale similarly across layers, and omit the layer index. We refer to memory consumption as the extra amount of memory needed to compute gradients without reflecting the memory to store the parameters or gradients themselves after computation.

Memory complexity: For Backprop, we have to store residuals required for both input and parameter gradients for every layer, which results in $O(M_x L + M_\theta L)$ memory complexity. For Moonwalk, we only need to store M_x for every layer in the first phase, in order to compute the input cotangent h_0, and then we can reuse M_θ in the second phase after computing each parameter gradient, for a total complexity of $O(M_x L + M_\theta)$. All other methods can discard activations after each layer, for a complexity of $O(M_x + M_\theta)$.

Memory complexity with checkpointing: In the case of Backprop with checkpointing, we will have

Table 1: Asymptotic complexity and key characteristics of Moonwalk, its pure variant, and four existing methods, analyzed in Section 11. **Forward:** only operates in forward-mode; **Submersive:** applicable to non-invertible submersive networks.

Method	Time	Memory	High-variance	Forward	Submersive
Backprop	$O(n^2L + ndL)$	$O\left(M_xL + M_\theta L\right)$	✗	✗	✓
Backprop + checkpoint	$O(n^2L + ndL)$	$O(\sqrt{n(M_x + M_\theta)L})$	✗	✗	✓
Forward-mode	$O(n^2dL^2)$	$O(M_x + M_\theta)$	✗	✓	✓
ProjForward	$O(n^2L + ndL)$	$O(M_x + M_\theta)$	✓	✓	✓
RevBackprop	$O(n^2L + ndL)$	$O(M_x + M_\theta)$	✗	✗	✗
Pure-Moonwalk	$O(n^3L + ndL)$	$O(M_x + M_\theta)$	✗	✓	✓
Moonwalk	$O(n^2L + ndL)$	$O(M_xL + M_\theta)$	✗	✗	✓
Monwalk + checkpoint	$O(n^2L + ndL)$	$O(\sqrt{nM_xL} + M_\theta)$	✗	✗	✓

additional memory of $O(cn)$, where $c \leq L$ is the number of checkpoints. Then, during backward, we must reconstruct each block of $O(L/c)$ layers and store residuals in $O((M_x + M_\theta)L/c)$ memory. The best trade-off, obtained at $c = O(\sqrt{(M_x + M_\theta)L/n})$ if feasible, is $O(\sqrt{n(M_x + M_\theta)L})$ memory. We can similarly apply checkpointing to the first phase of Moonwalk, which has no need to store M_θ when reconstructing from a checkpoint, for overall memory of $O(\sqrt{nM_xL} + M_\theta)$. In that case, we still prefer Moonwalk when $M_\theta \gg M_x$, although to a lesser extent than without checkpointing: in the extreme case that layers are so complex that we should checkpoint each one, $nL = O(M_x + M_\theta)$ and both Backprop and Moonwalk require $O(M_x + M_\theta)$ memory. On the other hand, any non-negligible M_θ ensures the benefit of Moonwalk when $L = \omega(M_\theta/n)$.

Time complexity for Backprop and RevBackprop: Backprop computation consists of computing two vector–Jacobian products in each layer i, $\text{vjp}(f_i, x_{i-1}, h_i)$ and $\text{vjp}(f_i, \theta_i, h_i)$, which accounts for per-layer time complexity of $O(n^2)$ and $O(nd)$, respectively, and for a total of $O(n^2L + ndL)$ time. RevBackprop additionally needs to evaluate the inverse function $f_i^{-1}(x_i)$, which does not impact the overall complexity in the terms we consider.

Time complexity for Forward-mode and ProjForward: In Forward-mode, each single parameter $\theta_{j,\ell}$ in layer j, of the total dL parameters, generates a pass to compute its gradient, in which we compute $\text{jvp}(f_i, x_{i-1}, \partial x_{i-1}/\partial \theta_{j,\ell})$ with complexity $O(n^2)$ in each layer $i = j + 1, \ldots, L$, for a total of $O(n^2dL^2)$ time. ProjForward with tangent $u = \{u_i\}_{i=1,\ldots,L}$ is similar to Forward-mode with just a single pass instead of dL passes, but additionally accumulating $\text{jvp}(f_i, \theta_i, u_i)$ in each layer i, for a total of $O(n^2L + ndL)$ time, which coincides with the time complexity of Backprop.

Time complexity for Moonwalk and Pure-Moonwalk: The first phase of Pure-Moonwalk computes $\text{jvp}(f_i, x_{i-1}, \frac{\partial x_{i-1}}{\partial x_0} e_\ell)$ in each layer i for each input element ℓ, for a total time complexity of $O(n^3L)$. The second phase computes $\text{vijp}(f_i, x_{i-1}, h_{i-1})$ and $\text{vjp}(f_i, \theta_i, h_i)$ in each layer for $O(n^2L + ndL)$ more, and a total of $O(n^3L + ndL)$ time. Moonwalk replaces the first phase with Backprop for just the input gradient, incurring time complexity $O(n^2L)$. Together with the same second phase as in Moonwalk, this totals $O(n^2L + ndL)$ time complexity.