Moonwalk: Inverse-Forward Differentiation

Dmitrii Krylov, Armin Karamzade, Roy Fox
University of California, Irvine
AISTATS 2026

The problem: Standard backprop stores residuals during the forward pass, impacting memory

Solution: Keep only input-gradient residuals; ignore residuals required for parameters

Result: A new method for gradient computation with up to 2× memory reduction and the same compute.

Moonwalk: Overview

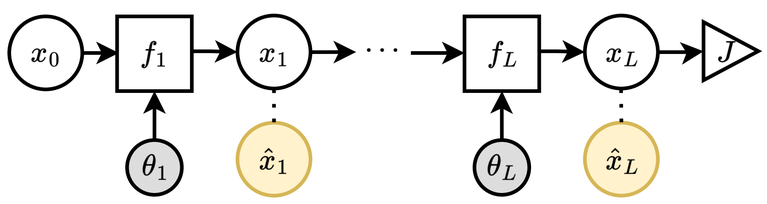
Moonwalk computes gradients in three phases:

Phase I (Forward pass) – Compute and store input-gradient residuals (highlighted in yellow), similar to backpropagation, but ignore parameter-gradient residuals.

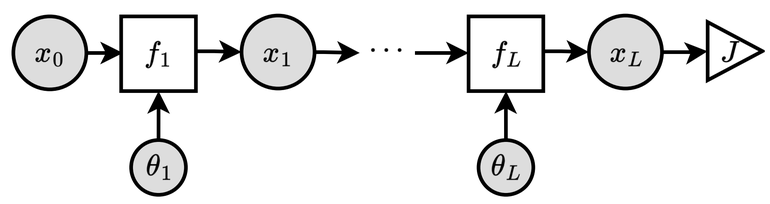
Phase II (Reverse pass) – Starting from the loss, propagate gradients with respect to only the inputs of each layer, yielding the gradient of the loss with respect to the network's input.

Phase III (Inverse forward pass) – Compute gradients for each layer, starting from the first, by multiplying each input gradient by each layer's Jacobian right-inverse.

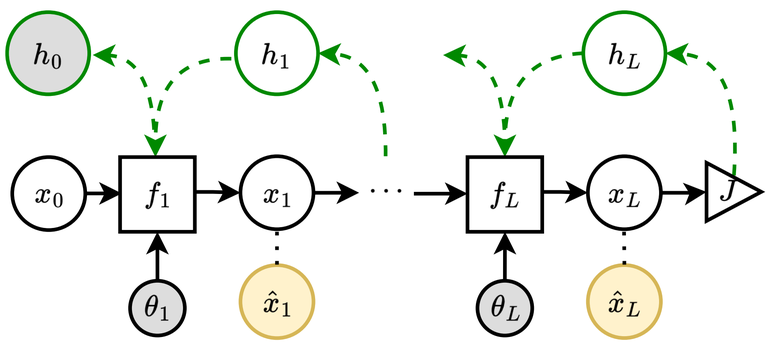
Moonwalk vs Backprop: A side by side comparison



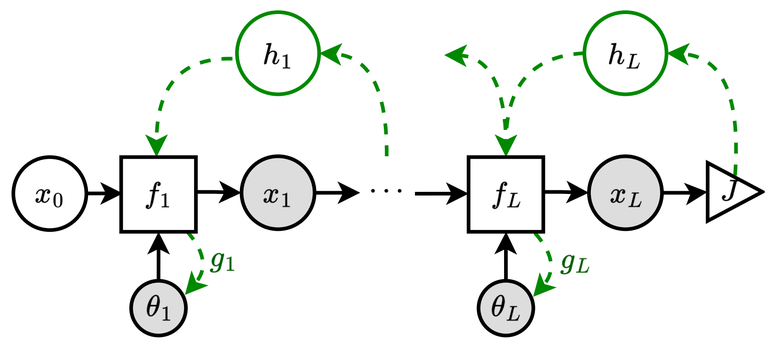
Moonwalk: Forward pass (Phase I)



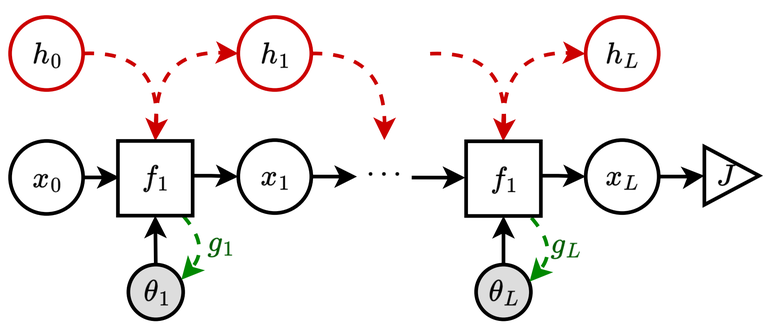
Backprop: Forward pass (Phase I)



Moonwalk: Computing h_0 (Phase II)



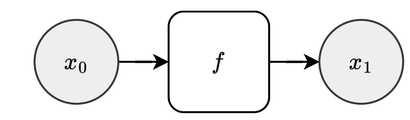
Backprop: Computing gradients (Phase II)



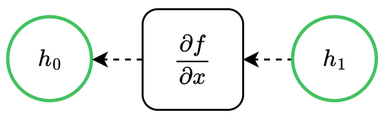
Moonwalk: Final gradients (Phase III)

VIJP Operator

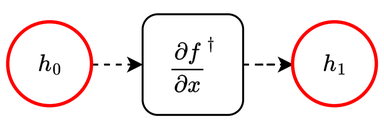
In Phase III, gradients must be reconstructed by multiplying each layer input gradient with the Jacobian right-inverse. This operation is memory-intensive; instead, we parameterize it using a single VIJP call.



Forward function: propagates the input to the loss function



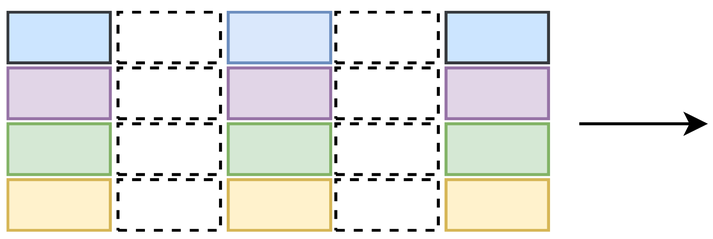
Vector-Jacobian Product: propagates gradients from the loss function to the input



Vector-Inverse-Jacobian Product: propagates gradients from the input to the loss function

Fragmental Gradient Checkpointing

Two improvements: (1) For layers whose Jacobian isn't always right-invertable, we store additional residuals, called *fragments*. (2) Since reconstruction can be slow, we further optimize it via block-wise reconstruction, which can be parallelized.



Example of parallel block-wise channel reconstruction for convolutional layers

Results: Convolutional Networks

We report results on 1D and 2D convolutional networks with both fragmental and non-fragmental gradients. Key results include memory reduction from 9.5 GB to 6.6 GB, allowing scaling from 10 to 22 layers of 1D convolutions using fragmental gradients.