PNODE: A MEMORY-EFFICIENT NEURAL ODE FRAMEWORK BASED ON HIGH-LEVEL ADJOINT DIFFERENTIATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We present a memory-efficient neural ODE framework PNODE based on high-level adjoint algorithmic differentiation. It is implemented using PyTorch and PETSc, one of the most commonly used portable, scalable scientific computing libraries. By leveraging discrete adjoint time integrators and advanced checkpointing strategies tailored for these integrators, PNODE can provide a balance between memory and computational costs while computing the gradients consistently and accurately. We demonstrate the performance through numerical experiments on image classification, continuous normalizing flow, and time series regression. We show that PNODE achieves the highest memory efficiency when compared with other reverse-accurate methods. We also show PNODE enables the application of implicit time integration methods that are desired for stiff dynamical systems.

1 INTRODUCTION

A residual network can be seen as a forward Euler discretization of a continuous time-dependent ordinary differential equation (ODE), as discussed in Lu et al. (2018); Haber & Ruthotto (2017); Ruthotto & Haber (2020). In the limit of smaller time steps, a new family of deep neural network models called neural ordinary differential equations (neural ODEs) was introduced in Chen et al. (2018). Compared with traditional discrete-time models, this family is advantageous in a wide range of applications, for example, modeling of invertible normalizing flows Grathwohl et al. (2019) and continuous time series with irregular observational data Rubanova et al. (2019). This continuous model also makes neural ODEs particularly attractive for learning and modeling the nonlinear dynamics of complex systems. They have been successfully incorporated into many data-driven models Rubanova et al. (2019); Greydanus Google Brain et al. (2019); Ayed et al. (2019) and adapted to a variety of differential equations including hybrid systems Dupont et al. (2019) and stochastic differential equations Jia & Benson (2019).

A main challenge in training neural ODEs is the balance between *stability*, *accuracy*, and *memory efficiency*. A naive implementation is to backpropagate the ODE solvers directly, resulting in a large redundant computational graph. To overcome this memory limitation, Chen et al. Chen et al. (2018) proposed to use a continuous adjoint method in place of backpropagation and invert the forward trajectory, which requires no storage of previous states and allows for training with constant memory. However, as pointed out in Gholaminejad et al. (2019); Onken & Ruthotto (2020); Zhuang et al. (2020), the continuous adjoint method may lead to inaccuracy in the gradient and instability during training. A framework named ANODE, based on a discrete adjoint method with checkpointing strategy, was proposed in Gholaminejad et al. (2019) and extended to ANODEV2 in Zhang et al. (2019b). A generalization of ANODE and ANODEV2 has been implemented in a Julia library Rackauckas et al. (2020). ANODE compresses the memory cost by saving only the initial states and recomputing the forward pass. The adaptive checkpoint adjoint method proposed in Zhuang et al. (2020) is similar to Gholaminejad et al. (2019) but uses a slightly different checkpointing strategy.

In this paper we begin by introducing and analyzing the two different adjoint methods that are common techniques for solving optimal control problems. Then we show that the discrete adjoints derived from a time integration method can be cast as a high-level abstraction of automatic differentiation and can produce exact gradients if the Jacobian is exact. Based on the discrete adjoint method with checkpointing, we propose PNODE to address the challenges in neural ODEs. The main contributions of this paper are:

1. We propose a framework that minimizes the depth of the computation graph for backpropagation, leading to significant savings in memory cost for neural ODEs. Our code is available online at https://github.com/pnode-dev/pnode.
2. High-level adjoint calculation with backpropagation through a minimal computation graph allows more flexibility in the design of neural ODEs. We show our framework enables implicit time integration and opens up possibilities to incorporate other integration methods.
3. We show that PNODE outperforms existing approaches in memory efficiency on a variety of tasks including image classification and continuous normalizing flow. We demonstrate the application of implicit integration methods with a time series regression example.

2 Preliminaries

2.1 Neural ODEs as an optimal control problem

Neural ODEs are a new class of models that consider the continuum limit of neural networks where the input-output mapping is realized by solving a system of parameterized ODEs

$$\frac{d\boldsymbol{u}}{dt} = f(\boldsymbol{u}, \boldsymbol{\theta}, t) \quad \boldsymbol{u}(t_0) = \boldsymbol{u}_0, \quad t \in [t_0, t_F], \tag{1}$$

where $\boldsymbol{u} \in \mathbb{R}^N$ is the state, $\boldsymbol{\theta} \in \mathbb{R}^{N_p}$ are the weights, and $f : \mathbb{R}^N \times \mathbb{R}^{N_p} \times \mathbb{R} \rightarrow \mathbb{R}^N$ is the vector field approximated by a neural network. Essentially, the input-output mapping is learned through a data-driven approach. During training, $\boldsymbol{u}^{(1)}, \boldsymbol{u}^{(2)}, \ldots, \boldsymbol{u}^{(S)}$ are fed as initial states, and the output should match the solutions of (1) at t_F.

Training neural ODEs can be viewed as solving an optimal control problem

$$\min_{\theta, \boldsymbol{u}} \left\{ \mathcal{L} := \phi(\boldsymbol{u}(t_F)) + \int_{t_0}^{t_F} q(\boldsymbol{u}(t), t) dt \right\}, \tag{2}$$

subject to the dynamical constraint (1).

This formulation generalizes the loss functional that depends on the final solution in the vanilla neural ODEs to additional functionals that depend on the entire trajectory in the time (or depth) domain. The integral term may come as part of the loss function or as a regularization term, for example, Tikhonov regularization, typically used for solving ill-posed problems. Finlay et al. (2020) showed that regularization techniques can be used to significantly accelerate the training of neural ODEs. Their approach can also be captured by the formula (2).

Notations For convenience, we summarize the notions used across the paper:

- N_t: the number of time steps in time integration.
- N_s: the number of stages in the time integration method.
- N_f: the number of layers in the neural network that approximates the vector field f in (1).
- N_b: the number of ODE blocks (one instance of (1) corresponds to one block).
- N_c: the number of maximum allowed checkpoints (one checkpoint can store one state).

2.2 Discrete adjoint vs continuous adjoint

To evaluate the objective $\mathcal{L}$ in (2), both the ODEs (1) and the integral need to be discretized in time. In order to calculate the gradient of $\mathcal{L}$ for training, adjoint methods (i.e., backpropagation in machine learning) can be used. Two distinct ways of deriving the adjoints exist: continuous adjoint and discrete adjoint.

Continuous adjoint The continuous adjoint sensitivity equation is

$$\frac{d\widetilde{\boldsymbol{\lambda}}}{dt} = -\frac{\partial f}{\partial \boldsymbol{u}} \widetilde{\boldsymbol{\lambda}} - \frac{\partial q}{\partial \boldsymbol{u}}, \tag{3}$$

$$\widetilde{\boldsymbol{\lambda}}(t_F) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{u}(t_F)}, \tag{4}$$

Table 1: Comparison between different adjoint approaches for forward Euler.

forward propagation	$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + hf(\boldsymbol{u}_n, \boldsymbol{\theta}, t_n)$
continuous adjoint	$\widetilde{\boldsymbol{\lambda}}_n = \widetilde{\boldsymbol{\lambda}}_{n+1} + h \left(\frac{\partial f(\boldsymbol{u}_{n+1}, \boldsymbol{\theta}, t_{n+1})}{\partial \boldsymbol{u}} \right)^T \widetilde{\boldsymbol{\lambda}}_{n+1}$
discrete adjoint	$\boldsymbol{\lambda}_n = \boldsymbol{\lambda}_{n+1} + h \left(\frac{\partial f(\boldsymbol{u}_n, \boldsymbol{\theta}, t_n)}{\partial \boldsymbol{u}} \right)^T \boldsymbol{\lambda}_{n+1}$

where $\widetilde{\boldsymbol{\lambda}} \in \mathbb{R}^N$ is an adjoint variable. Its solution gives the gradient of $\mathcal{L}$ with respect to the initial state $\boldsymbol{u}_0$; that is, $\widetilde{\boldsymbol{\lambda}}(t_0) = \frac{d\mathcal{L}}{d\boldsymbol{u}_0}$. The gradient of $\mathcal{L}$ with respect to the weights $\boldsymbol{\theta}$ is

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} + \int_{t_0}^{t_F} \left(\widetilde{\boldsymbol{\lambda}}^T \frac{\partial f}{\partial \boldsymbol{\theta}} + \frac{\partial q}{\partial \boldsymbol{\theta}} \right) dt. \tag{5}$$

The infinite-dimensional sensitivity equations and the integrals also need to be discretized in time in order to obtain a finite-dimensional approximation to the derivatives. In the practical implementation, one must integrate (1) forward in time and save the intermediate states. Then the sensitivity equations are solved backward in time, during which the saved states are restored to evaluate the Jacobians. These two steps are analogous to forward pass and backpropagation in machine learning nomenclature. Theoretically, different time integration methods and different time step sizes can be chosen for solving the forward equations and the sensitivity equations.

Discrete adjoint An alternative approach is to derive the adjoint for the discretized version of the infinite-dimensional equations (1) directly. Assume the discretization is formally represented by a time-stepping operator $\boldsymbol{\mathcal{N}}$ that propagates the solution from one time step to another:

$$\boldsymbol{u}_{n+1} = \boldsymbol{\mathcal{N}}(\boldsymbol{u}_n). \tag{6}$$

The adjoint sensitivities are propagated with

$$\begin{aligned} \boldsymbol{\lambda}_n &= \left(\frac{d\boldsymbol{\mathcal{N}}}{d\boldsymbol{u}}(\boldsymbol{u}_n) \right)^T \boldsymbol{\lambda}_{n+1}, \quad n = N-1, \ldots, 0, \\ \boldsymbol{\lambda}_N &= \left(\frac{d\psi}{d\boldsymbol{u}}(\boldsymbol{u}_N) \right)^T. \end{aligned} \tag{7}$$

Similarly, $\boldsymbol{\lambda}_0$ gives the sensitivity of $\mathcal{L}$ to the initial state $\boldsymbol{u}_0$. Parametric sensitivities $\frac{d\mathcal{L}}{d\boldsymbol{\theta}}$ can be derived by augmenting the original ODEs with $\frac{d\boldsymbol{\theta}}{dt} = 0$. See the appendix for more details.

2.3 Which adjoint method is better for neural ODEs?

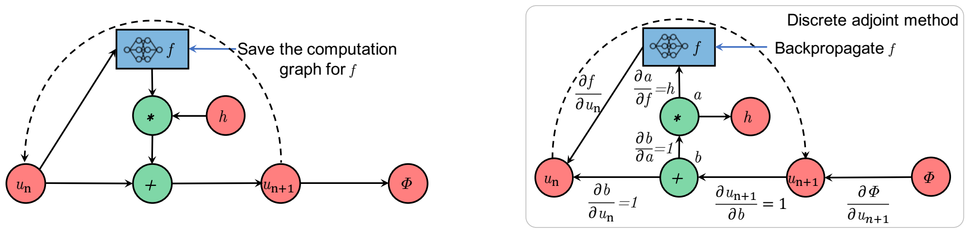
While both approaches can generate the gradients for the loss function, these gradients are typically not the same even if the same time integration method and step sizes are used. They are asymptotically equivalent as the time-step size approaches zero. As a concrete example, we compare the continuous adjoint equation after time discretization using the forward Euler method with the discrete adjoint for the same method. As shown in Table 1, the Jacobian is evaluated at $\boldsymbol{u}_{n+1}$ and $\boldsymbol{u}_n$, respectively, causing a discrepancy that can be quantified by the following proposition. To show the influence on the gradient, we analyze the discrepancy in the adjoint variable between the two methods after one backward time step.

Proposition 1 *Assuming $\widetilde{\boldsymbol{\lambda}}_{n+1} = \boldsymbol{\lambda}_{n+1}$, the local discrepancy between the continuous adjoint sensitivity and the discrete adjoint sensitivity is*

$$\|\widetilde{\boldsymbol{\lambda}}_n - \boldsymbol{\lambda}_n\| = h^2 \|\mathcal{H}\left(\boldsymbol{u}_n + \epsilon(\boldsymbol{u}_{n+1} - \boldsymbol{u}_n), \boldsymbol{\theta}, t_n\right) f(\boldsymbol{u}_n, \boldsymbol{\theta}, t_n)\| \|\boldsymbol{\lambda}_{n+1}\|, \tag{8}$$

where $\mathcal{H}$ is the Hessian of f and ϵ is a constant in $(0, 1)$.

This indicates that $\widetilde{\boldsymbol{\lambda}}_{n+1} = \boldsymbol{\lambda}_n$ for linear functions since the Hessian is zero and for nonlinear functions, such as deep neural networks, $\widetilde{\boldsymbol{\lambda}}_{n+1}$ approaches $\boldsymbol{\lambda}_n$ quadratically as $h \to 0$. Nevertheless, the accumulated discrepancy may clearly lead to inconsistency in the gradient calculated with the continuous adjoint approach. Similar results also hold for other multistage time integration methods such as Runge–Kutta methods.



Save the computation graph for f
f
$*$
h
$+$
u_n
u_{n+1}
Φ
Discrete adjoint method
Backpropagate f
$\frac{\partial f}{\partial u_n}$
$\frac{\partial a}{\partial f}=h$
a
$\frac{\partial b}{\partial a}=1$
b
$\frac{\partial b}{\partial u_n}=1$
$\frac{\partial u_{n+1}}{\partial b}=1$
$\frac{\partial \Phi}{\partial u_{n+1}}$

(a) Forward pass (b) Reverse pass

Figure 1: Illustration of the discrete adjoint method using forward Euler. For simplicity, the state variable is assumed to be scalar, and the function f is assumed to be a scalar function.

3 METHOD

3.1 HIGH-LEVEL ABSTRACTION OF AUTOMATIC DIFFERENTIATION

Despite the nuances in different approaches for gradient computation in neural ODEs, they can be interpreted fundamentally as a unified method that utilizes automatic differentiation (AD) (Nocedal & Wright, 2006) at different abstraction levels. Specifically, the reverse (adjoint) mode of AD can compute the gradient of a scalar function efficiently, at a cost independent of the number of parameters. In order to compute the gradient, the function needs to be evaluated first, often called a forward pass that generates output from input. Then a list of operations during the evaluation is recorded and forms a computational graph of the function. To compute the derivatives, one applies the chain rule of differentiation to each node in the graph in a backward order.

This procedure can be performed in different ways based on the definition of a *primitive* operation in the list. In the continuous adjoint approach adopted in vanilla neural ODE, the primitive operation is the ODE itself; in the discrete adjoint approach in Section 2.2, the primitive operation is a propagation equation for a particular time-stepping algorithm, thus yielding a high-level representation of the computation flow.

3.2 ADJOINT METHODS REDUCE THE DEPTH OF COMPUTATIONAL GRAPH.

In a naive neural ODE approach, one can use an AD tool to backpropagate the entire forward solve, thus requiring $\mathcal{O}(N_t N_s N_f)$ memory to record every operation in the graph. The memory cost for backpropagation increases with the size of the neural network and the number of time steps, making this approach infeasible for large problems and long-time time integration.

In contrast, high-level AD methods, such as the discrete adjoint method in Section 2.2, compute the gradient by composing the derivatives for the *primitive* operations. Therefore, one only needs to backpropagate the *primitive* operations instead of the entire ODE solver. If the primitive operation is a time step, then one can backpropagate one time step at a time, and the memory cost for backpropagation will be independent of the number of time steps.

3.3 PNODE MINIMIZES THE DEPTH OF THE COMPUTATIONAL GRAPH AND BALANCES THE COSTS OF RECOMPUTATION AND CHECKPOINTING

Based on discrete adjoint method, we propose PNODE that checkpoints the solutions at selective time steps in the forward pass, with the stage values if multistage time integration methods such as Runge-Kutta methods are used. Essentially, we take the *function evaluation* of f as the *primitive* operation in the high-level discrete adjoint method. In the reverse pass, the checkpoints are restored and used to compute the transposed Jacobian-vector product $\left(\frac{\partial f}{\partial \boldsymbol{u}}\right)^T v$, where the vector v is determined by the time integration algorithm. The transposed Jacobian-vector product is obtained by backpropagating f with a constant memory cost $\mathcal{O}(N_f)$. To illustrate our approach, we show the workflow for forward Euler in Figure 1. By assembling all the edges along each path from the output to the input according, we obtain the gradient of the output function to the input $\mathbf{u}_0$. Nota that the high-level discrete adjoint algorithm (Zhang et al., 2019a) for a time integration method can be derived and implemented manually, and it can be reused in different applications.

The memory cost thus consists of two parts for backpropogation and checkpointing, respectively. The cost for backpropogation is $\mathcal{O}(N_f)$, which is minimal for backpropagating the computational graph of f. The cost for checkpointing is $\mathcal{O}(N_c)$. With limited memory budget ($N_c < N_t$), the states that are not checkpointed in the forward pass can be *recomputed* from a nearby checkpoint. See Figure 1 in Appendix for illustration. We employ optimal checkpointing schedules with minimal recomputations developed by (Zhang & Constantinescu, 2021) to achieve the best efficiency.

Proposition 2 *(Zhang & Constantinescu, 2021) Given N_c allowed checkpoints in memory, the minimal number of extra forward steps (recomputations) needed for the adjoint computation of N_t time steps is*

$$\tilde{p}(N_t, N_c) = (t-1)\,N_t - \binom{N_c+t}{t-1} + 1, \tag{9}$$

where t is the unique integer satisfying $\binom{N_c+t-1}{t-1} < N_t \leq \binom{N_c+t}{t}$.

We remark that this checkpointing strategy allows us to balance between the recomputation cost and the memory cost for checkpointing. In an ideal case where memory is sufficient for saving all stages at each time step, no recomputation is needed. Then the computational cost of PNODE becomes $\mathcal{O}(N_t N_s)$.

3.4 PNODE ENABLES IMPLICIT TIME INTEGRATION

The application of implicit time integration is highly desired for stiff dynamical systems where explicit methods may fail due to stability constraints. However, backpropagating through the implicit solver is difficult because the complexity of the nonlinear/linear solve required at each time step and the large amount of memory needed for the resulting computational graph. By taking the function (f) evaluation as the *primitive* operation in high-level AD, PNODE excludes the nonlinear/linear solvers from the computational graph for backpropagation. Instead, it solves an adjoint equation (a transposed linear system) to propagate the gradients. See Appendix C.3 for examples. We solve both the linear system in the forward pass and the transposed system in the reverse pass with a matrix-free iterative method for efficiency. The action of the matrix or its transpose is computed efficiently by backpropagating f. Again, only the computational graph for f needs to be created in our approach.

3.5 COMPARISON WITH OTHER METHODS

NODE cont: the original implementation by Chen et al. (2018) with continuous adjoint The vanilla neural ODE (Chen et al., 2018) avoids recording everything by solving the ODE backward in time to obtain the intermediate solutions needed when solving the continuous adjoint equation. Therefore, it requires a constant memory cost $\mathcal{O}(N_f)$ to backpropagate f.

NODE naive: a variant of the original implementation by Chen et al. (2018) This is a naive method that backpropagates the ODE solvers and has the deepest computational graph, but it has the minimum computational cost $\mathcal{O}(N_t)$ since no recomputation is needed in the backward solve.

ANODE: a framework with discrete adjoint and checkpointing method proposed by Gholaminejad et al. (2019) The checkpointing method for ANODE (Gholaminejad et al., 2019) has the same memory cost $\mathcal{O}(N_t)$ as NODE naive theoretically when a single ODE block is considered. For multiple ODE blocks, ANODE saves only the initial states for each block and recomputes the forward pass before the backpropagation for each block; consequently, the total memory cost for checkpointing is $\mathcal{O}(N_t) + \mathcal{O}(N_b)$ where N_b is the number of ODE blocks.

ACA: the adaptive checkpoint adjoint (ACA) method proposed by Zhuang et al. (2020) To achieve reverse accuracy, ACA checkpoints the state at each time step and recomputes the forward steps locally during the backward pass, thus consuming $\mathcal{O}(N_t)$ memory for checkpointing. To save memory, ACA deletes redundant computation graphs for rejected time steps when searching for optimal step size. Its memory cost for backpropagation is $\mathcal{O}(N_t N_s N_f)$.

A summary comparing PNODE with existing representative implementations of neural ODEs is given in Table 2. For simplicity, we assume the number of reverse time steps in the continous adjoint

Table 2: Comparison between different implementations of neural ODEs for one ODE block. The computational complexity is measured in terms of the number of function evaluations (f).

	NODE cont	NODE naive	ANODE	ACA	PNODE(Ours)
Computational complexity	$2N_tN_s$	N_tN_s	$2N_tN_s$	$2N_tN_s$	$2N_tN_s$
Backpropagation memory	N_f	$N_tN_sN_f$	$N_tN_sN_f$	$N_tN_sN_f$	N_f
Checkpointing memory	-	-	N_t	N_t	N_c
Reverse-accuracy	×	✓	✓	✓	✓
Implicit time stepping support	×	×	×	×	✓

method is also N_t and do not consider rejected time steps. Note that the rejected time steps have no influence on the computational cost and the memory cost of PNODE because the adjoint calculation in the reverse pass involves only accepted time steps (Zhang & Constantinescu, 2021).

3.6 Implementation

Drawing on state-of-the-art adjoint-capable ODE solvers (`TSAdjoint` in `PETSc`), we implemented PNODE by interfacing `PETSc` to `PyTorch` and utilizing its discrete adjoint solvers with optimal checkpointing. As a key step, we implemented a data conversion mechanism in `petsc4py`, the Python bindings for `PETSc`, based on the DLPack standard.[1] This enables in-place conversion between `PETSc` vectors and `PyTorch` tensors for both CPUs and GPUs and thus allows `PETSc` and `PyTorch` to access the same data seamlessly. Although PNODE can be implemented with any differentiable ODE solvers such as `FATODE` (Zhang & Sandu, 2014) and `DiffEqSensitivity.jl` (Rackauckas & Nie, 2017), `PETSc` has several favourable features that other tools may lack:

Rich set of numerical integrators As a widely used time-stepping library, `PETSc` offers a large collection of time integration algorithms for solving ODEs, differential algebraic equations, and hybrid dynamical systems (Abhyankar et al., 2018; Zhang et al., 2017). It includes explicit, implicit methods, and implicit-explicit methods, multirate methods with various stability properties, and adaptive time-stepping support. The discrete adjoint approach proposed in Section 2.2 has been implemented for some time integrators and can be easily expanded to others (Zhang et al., 2019a).

Discrete adjoint solvers with matrix-free Jacobian When using the adjoint solver, we compute the transposed Jacobian-vector product through AutoGrad in `PyTorch` and supply it as a callback function to the solver, instead of building the Jacobian matrix and performing matrix-vector product, which are expensive tasks especially for dense matrices. In addition, combining low-level AD with the high-level discrete adjoint solver guarantees reverse accuracy, as explained in Section 3.1.

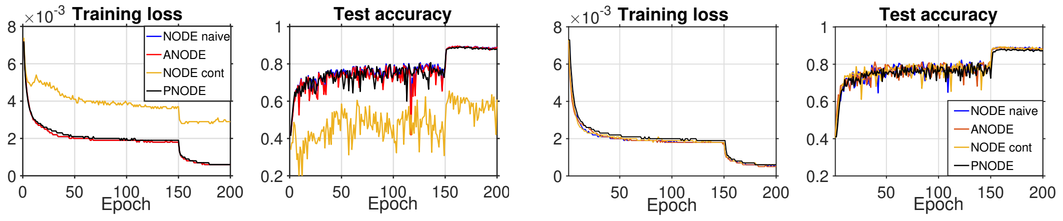
Optimal checkpointing for multistage methods Checkpointing is critical for balancing memory consumption and computation cost. In addition to the classic `Revolve` algorithm (Griewank & Walther, 2000), `PETSc` supports optimal algorithms (Zhang & Constantinescu, 2021) that are extended from `Revolve` and tailored specifically for multistage methods. `PETSc` also support sophisticated checkpointing algorithms (Stumm & Walther, 2010) for hierarchical storage systems.

HPC-friendly linear algebra kernels `PETSc` has fully fledged GPU support for efficient training of neural ODEs, including multiprecision support (half, single, double, float128) and extensive parallel computing that leverages CUDA-aware MPI (Mills et al., 2020).

4 Experimental results

In this section we test the performance of our proposed framework PNODE and compare it with the aforementioned four implementations of neural ODEs on diverse benchmark tasks. All the experiments are conducted on an NVIDIA Tesla V100 GPU with a memory capacity of 32 GB. When testing PNODE, we use $N_c = N_t$ for best speed. Note that this corresponds to the worst-case memory cost for PNODE.

[1]https://github.com/dmlc/dlpack



(a) Euler (b) RK4

Figure 2: Training and testing performance of SqueezeNext on CIFAR10 using Euler and fourth-order Runge–Kutta (RK4) schemes with one time step. The inaccuracy in the gradient calculated via continuous adjoint causes a significant gap in performance between discrete adjoint and continuous adjoint when using the Euler scheme, while the performance is consistent when using RK4.

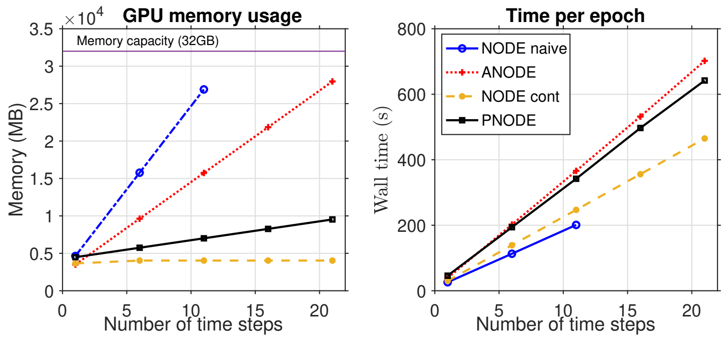


Figure 3: GPU memory and wall-clock time usage of different implementations as functions of the number of time steps (N_t) using RK4 scheme. PNODE is the only discrete adjoint method that does not exceed the memory capacity with $N_t = 12$. Both NODE naive and ANODE fail for $N_t \geq 12$.

4.1 Replacing residual blocks in image classification

Chen et al. Chen et al. (2018) showed that replacing residual blocks with ODE blocks yields good accuracy and parameter efficiency on the MNIST dataset from Lecun et al. (1998). Here we perform the experiment on a more complex dataset CIFAR-10 from Krizhevsky (2012) using a SqueezeNext network introduced by Gholami et al. (2018), where every nontransition block is replaced with a neural ODE block. There are 4 ODE blocks of different dimensions with $199,800$ trainable parameters in total. Unfortunately, we were not able to run the ACA (and MALI) code on this test using the same time integration methods. Thus ACA is not included for comparison here.

Accuracy We train the model using the same architecture with one time step but different schemes, and we display the results in Figure 2. The ReLU activations in this model result in irreversible dynamics and inaccurate gradient calculation through continuous adjoint, and the adaptive Runge–Kutta 4(5) method (dopri5) using the continuous adjoint method leads to divergent training, as also observed in Gholaminejad et al. (2019). With the low-accuracy forward Euler scheme, our implementation and other reverse-accurate methods converge with lower loss and higher accuracy compared with the continuous adjoint implementation.

Memory/time efficiency We perform an empirical comparison on the same task while fixing $L = 4$ given the SqueezeNext architecture and letting N_t vary. The statistics of GPU memory and time usage are displayed in Figure 3. Among all reverse-accurate methods, PNODE has the slowest growth when N_t increases when the number of time steps increases. Its memory cost is approximately 18.5% of NODE and 28.8% of ANODE. The training speed is comparable to but slightly better than ANODE, which agrees with the theoretical result shown in Table 2.

4.2 Continuous normalizing flow for density estimation

For continuous normalizing flow (CNF), a continuous dynamics from a base contribution to the observable data is constructed through solving an initial value problem for a neural ODE. The model is naturally reversible through integrating backward in time. We consider the CNF approach for density estimation, where the loss function is the negative log-likelihood (lower is better).

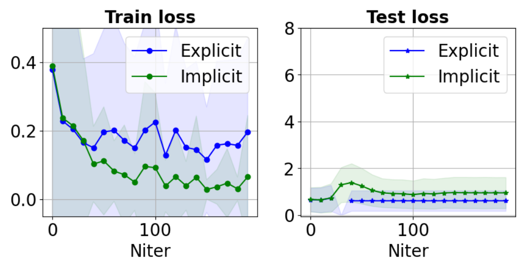
We follow the architecture and hyperparameters chosen by Grathwohl et al. (2019) and compare the results using fixed and adaptive step sizes on three public datasets with different dimensions: POWER, MINIBOONE, and BSDS300 (see Papamakarios et al. (2017) for a detailed description). The statistics are shown in Table 3. All implementations yield comparable test loss. As expected, PNODE has the best memory efficiency among the reverse-accurate methods and has better time efficiency than ANODE has. We note that when training models using adaptive step size solver, the number of function evaluations in the forward pass (NFE-F) and backward pass (NFE-B) are not fixed ahead of time and may increase drastically. As a result, they take significantly more time than do schemes with fixed step size. On the BSDS300 dataset, experiments using an adaptive step size solver were unable to be finished for neural ODEs using discrete adjoint methods: the GPU memory usage grows significantly and exceeds the memory capacity within few epochs, and NODE cont failed to converge after 14 days of training and was terminated prematurely.

Table 3: Performance statistics on POWER, MINIBOONE, and BSDS300. *Note that for BSDS300, the batchsize is $1,000$ for discrete adjoint methods (to fit into the memory capacity) and $10,000$ for the continuous adjoint method.

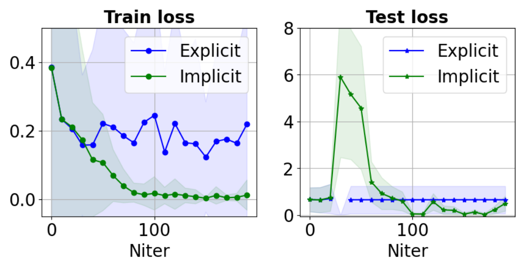
	Scheme	NFE-F	NFE-B	Time per epoch (s)	GPU Mem (MB)	Test loss
		POWER, 5 flow steps				
NODE naive	dopri5, $N_t = 10$	300	0	5.10E2	13,375	-0.15
ANODE		300	300	3.29E2	3,681	-0.34
PNODE		305	300	2.72E2	**1,802**	-0.42
NODE naive	dopri5, adaptive	556(160-568)	0	3.39E3	-	0.04
NODE cont		501(166-604)	502(183-579)	4.76E2	1,549	-0.34
ACA		528(105-532)	497(145-509)	7.18E2	2,301	-0.29
		MINIBOONE, 1 flow step				
NODE naive	dopri5, $N_t = 4$	24	0	2.02	2,159	10.45
ANODE		24	24	3.21	2,295	10.54
PNODE		25	24	3.11	**1,701**	10.91
NODE naive	dopri5, adaptive	114(32-140)	0	2.66E1	-	10.53
NODE cont		110(32-122)	87(33-93)	1.68E1	1,582	10.44
ACA		104(21-105)	85(29-85)	9.48	2,423	10.15
		BSDS300, 2 flow steps				
NODE naive	dopri5, $N_t = 20$	240	0	2.63E3	24,314	-128.68
ANODE		240	240	2.78E3	12,022	-166.57
PNODE		242	240	2.32E3	**2,063**	-143.95
NODE naive	dopri5, adaptive	125(58-190)	0	1.99E3	-	18.88
NODE cont		408(70-604)	325(66-516)	2.71E3	*9,572	-148.87
ACA		177(42-210)	166(58-184)	4.41E2	26,553	40.40

4.3 Time series regression

Implicit solvers are known to have absolute stability with arbitrary step size, while the stability of explicit solvers typically requires the step size to be smaller than a certain value, which is hard to determine without prior knowledge of the dynamics. In this example we demonstrate the potential of implicit time integration methods in time series modeling through the following two-dimensional toy example:



(a) First-order methods: Euler (explicit) and backward Euler (implicit)



(b) Second-order methods: midpoint (explicit) and Crank–Nicolson (implicit)

Figure 4: Train and test loss with explicit/implicit ODE solvers as functions of number of iterations.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = A \begin{bmatrix} x_1^3 \\ x_2^3 \end{bmatrix}, \quad A = \begin{bmatrix} -0.10 & 2.00 \\ -2.00 & -0.10 \end{bmatrix}.$$

The goal is to fit the matrix A by minimizing the sum of the absolute values of residuals. The time ranges from 0 to 16, and we have a batch size of 20. For the training model, we use a single linear layer with 2D input and output applied to the elementwise cubic of x, so the true underlying model can be represented exactly. The reference solution and samples for training are generated from the high-accuracy adaptive dopri5 solver.

Here we show results with fixed step size ($h = 0.1$) using four different solvers: Euler (first order, explicit), backward Euler (first order, implicit), midpoint (second order, explicit), and Crank–Nicolson (second order, implicit). The training and test loss as functions of iteration are shown in Figure 4. Models with implicit solvers outperform explicit solvers with the same order by reaching a smaller training loss. Moreover, training with explicit solvers caused the ODE solutions to explode because of instability, reflected by the discontinuities of curves in the test loss. In contrast, there is no stability issue for training with implicit solvers.

In addition, the matrices learned by neural ODE with implicit solvers are closer to the ground truth, as shown below.

$$A^{Euler} = \begin{bmatrix} -0.23 & 1.24 \\ -1.21 & -0.22 \end{bmatrix}, \quad A^{BEuler} = \begin{bmatrix} 0.11 & 1.98 \\ -2.00 & 0.10 \end{bmatrix}$$
$$A^{Midpoint} = \begin{bmatrix} -0.05 & 1.16 \\ -1.14 & -0.03 \end{bmatrix}, \quad A^{CN} = \begin{bmatrix} -0.11 & 2.00 \\ -1.99 & -0.12 \end{bmatrix}$$

5 Conclusion

In this work we propose PNODE, a framework for neural ODEs based on high-level discrete adjoint methods with checkpointing. We show that PNODE can minimize the depth of the computation graph for backpropagation while still generating an accurate gradient. We successfully reduce the memory cost of neural ODEs to $\mathcal{O}(N_c) + \mathcal{O}(N_f)$, and $\mathcal{O}(N_t N_s) + \mathcal{O}(N_f)$ in the worse case when checkpointing all intermediate states (including ODE solutions and stage vectors). With extensive numerical experiments we demonstrate that PNODE has the best memory efficiency among the existing neural ODEs with accurate gradients. Furthermore, we demonstrate that PNODE enables the application of implicit integration methods, which offers a potential solution for stabilizing the training of stiff dynamical systems. We have made PNODE freely available. We believe it will be useful for a broad range of AI applications, especially for scientific machine learning tasks such as the discovery of unknown physics.

REFERENCES

Shrirang Abhyankar, Jed Brown, Emil M Constantinescu, Debojyoti Ghosh, Barry F Smith, and Hong Zhang. PETSc/TS: a modern scalable ODE/DAE solver library. *arXiv preprint arXiv:1806.01437*, 2018.

Ibrahim Ayed, Emmanuel de Bézenac, Arthur Pajot, Julien Brajard, and Patrick Gallinari. Learning dynamical systems from partial observations, 2019.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural ODEs. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Chris Finlay, Joern-Henrik Jacobsen, Levon Nurbekyan, and Adam Oberman. How to train your neural ODE: the world of Jacobian and kinetic regularization. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 3154–3164. PMLR, 13–18 Jul 2020.

Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. SqueezeNext: Hardware-aware neural network design. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

Amir Gholaminejad, Kurt Keutzer, and George Biros. ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 730–736. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/103.

Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, and David Duvenaud. Scalable reversible generative models with free-form continuous dynamics. In *International Conference on Learning Representations*, 2019.

Sam Greydanus Google Brain, Misko Dzamba PetCube, and Jason Yosinski. Hamiltonian Neural Networks. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, March 2000. ISSN 0098-3500. doi: 10.1145/347837.347846.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 12 2017. doi: 10.1088/1361-6420/aa9a90.

Junteng Jia and Austin R Benson. Neural jump stochastic differential equations. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 3276–3285. PMLR, 10–15 Jul 2018.

Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, Alp Dener, Matthew Knepley, Scott E. Kruger, Hannah Morgan, Todd Munson, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Junchao Zhang. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Computing (to appear)*, 2020.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

Derek Onken and Lars Ruthotto. Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows. *arXiv e-preprints*, 2020.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. doi: 10.5334/jors.151. URL `https://app.dimensions.ai/details/publication/pub.1085583166andhttp://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/`. Exported from https://app.dimensions.ai on 2019/05/05.

Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Ramadhan. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

Yulia Rubanova, Ricky T. Q. Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, 62(3):352–364, 2020. doi: 10.1007/s10851-019-00903-1.

Philipp Stumm and Andrea Walther. New algorithms for optimal online checkpointing. *SIAM Journal on Scientific Computing*, 32(2):836–854, 2010. ISSN 1064-8275. doi: 10.1137/080742439.

Hong Zhang and Emil M. Constantinescu. Revolve-based adjoint checkpointing for multistage time integration. In *International Conference on Computational Science*, (online), in main track, 2021.

Hong Zhang and Adrian Sandu. FATODE: a library for forward, adjoint, and tangent linear integration of ODEs. *SIAM Journal on Scientific Computing*, 36(5):C504–C523, 2014. ISSN 1064-8275. doi: 10.1137/130912335.

Hong Zhang, Shrirang Abhyankar, Emil Constantinescu, and Mihai Anitescu. Discrete adjoint sensitivity analysis of hybrid dynamical systems with switching. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(5):1247–1259, 2017. ISSN 15498328. doi: 10.1109/TCSI.2017.2651683.

Hong Zhang, Emil M. Constantinescu, and Barry F. Smith. PETSc TSAdjoint: a discrete adjoint ODE solver for first-order and second-order sensitivity analysis. *arXiv e-preprints*, 2019a.

Tianjun Zhang, Zhewei Yao, Amir Gholami, Joseph E Gonzalez, Kurt Keutzer, Michael W Mahoney, and George Biros. ANODEV2: A coupled neural ODE framework. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019b.

Juntang Zhuang, Nicha Dvornek, Xiaoxiao Li, Sekhar Tatikonda, Xenophon Papademetris, and James Duncan. Adaptive checkpoint adjoint method for gradient estimation in neural ODE. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 11639–11649, 2020.