ABSTRACT

Neural ordinary differential equations (neural ODEs) have emerged as a novel network architecture that bridges dynamical systems and deep learning. However, the gradient obtained with the continuous adjoint method in the vanilla neural ODE is not reverse-accurate. Other approaches suffer either from excessive memory requirement due to deep computational graphs or from limited choices for the time integration scheme, hampering their application to large-scale complex dynamical systems. To achieve accurate gradients without compromising memory efficiency and flexibility, we present a new neural ODE framework, PNODE, based on high-level discrete adjoint algorithmic differentiation. 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 provide an open-source implementation based on PyTorch and PETSc, one of the most commonly used portable, scalable scientific computing libraries. We demonstrate the performance through extensive numerical experiments on image classification and continuous normalizing flow problems. We show that PNODE achieves the highest memory efficiency when compared with other reverse-accurate methods. On the image classification problems, PNODE is up to two times faster than the vanilla neural ODE and up to 2.3 times faster than the best existing reverse-accurate method. We also show that PNODE enables the use of the implicit time integration methods that are needed for stiff dynamical systems.

Keywords Neural ODEs, deep learning, adjoint differentiation, checkpointing, implicit methods

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 [1–3]. In the limit of smaller time steps, a new family of deep neural network models called neural ODEs was introduced in [4]. Compared with traditional discrete-time models, this family is advantageous in a wide range of applications, such as modeling invertible normalizing flows [5] and continuous time series with irregular observational data [6]. This continuous model also makes neural ODEs particularly attractive for learning and modeling the nonlinear dynamics of complex systems. Neural ODEs have been successfully incorporated into many data-driven models [6–8] and adapted to a variety of differential equations including hybrid systems [9] and stochastic differential equations [10].

The success of neural ODEs consolidates the connection between deep learning and dynamical systems, allowing well-established dynamical system theory to be applied to deep learning. However, training neural ODEs efficiently is still challenging. A major part of the challenge is computing the gradient for the ODE layer, while achieving a balance

* This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under contract DE-AC02-06CH11357 at Argonne National Laboratory. (Corresponding author: Hong Zhang.)
between stability, accuracy, and memory efficiency. A naive approach for the gradient calculation is to backpropagate the ODE layer directly, resulting in a large redundant computational graph. To overcome this memory limitation, Chen et al. [4] proposed using a continuous adjoint method in place of backpropagation and reverse the forward trajectory, which requires no storage of previous states and allows for training with constant memory. As pointed out in [11–13], however, the continuous adjoint method may lead to inaccurate gradients and instability during training. Therefore, reverse-accurate methods based on discrete adjoint methods and backpropagation have been developed in [11, 13–15], and certain simple checkpointing strategies are used to reduce the memory overhead. The MALI method in [15] can avoid the use of checkpointing, but it is restricted to a symplectic time integrator that is typically designed for Hamiltonian systems.

Another limitation of existing frameworks is that they support only explicit time integration methods such as Runge–Kutta methods. This leads to tremendous difficulties in dealing with stiff dynamical systems, partly because of the stability constraints of explicit schemes. Implicit schemes are unconditionally stable; however, they require the solution of nonlinear/linear systems. Directly backpropagating through implicit schemes is not computationally efficient and can be infeasible because of the complexity in the procedure, which is usually iterative [16].

Therefore, we propose PNODE to overcome the limitations of existing neural ODEs. By utilizing a high-level discrete adjoint method together with checkpointing, PNODE achieves reverse accuracy and memory efficiency in the gradient calculations and, at the same time, allows for flexibility in training strategies. The main contributions of this work are as follows:

1. A framework that minimizes the depth of the computational graph for backpropagation, leading to significant savings in memory for neural ODEs. Our code is available online at https://github.com/caidao22/pnode.
2. High-level discrete adjoint calculation with backpropagation through a minimal computational graph that allows more flexibility in the design of neural ODEs. We show that our framework enables the use of implicit time integration schemes and opens up the possibility of incorporating other integration methods.
3. Demonstration that PNODE outperforms existing methods in memory efficiency on diverse tasks including for image classification and continuous normalizing flow problems.
4. Successful application of PNODE to learning stiff dynamical systems with implicit methods, which has been recognized to be very challenging.

The rest of this article is organized as follows. In Section 2 we introduce the basics of neural ODEs and discuss the two adjoint methods that are commonly used techniques for computing gradients when solving optimal control problems. In Section 3 we present the key techniques in our discrete adjoint-based framework, explain their advantages over existing frameworks, and describe the implementation. In Section 4 we demonstrate the performance of our approach and showcase its success in learning stiff dynamics from data. In Section 5 we summarize our conclusions.

2 Preliminaries

2.1 Neural ODEs as an optimal control problem

Neural ODEs are a 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:

$$\frac{du}{dt} = f(u, \theta, t) \quad u(t_0) = u_0, \quad t \in [t_0, t_F],$$

(1)

where \( u \in \mathbb{R}^N \) is the state, \( \theta \in \mathbb{R}^{N_p} \) are the weights, and \( f : \mathbb{R}^N \times \mathbb{R}^{N_p} \times \mathbb{R} \to \mathbb{R}^N \) is the vector field approximated by a neural network. The input-output mapping is learned through a data-driven approach. During training, \( u^{(1)}, u^{(2)}, \ldots, u^{(S)} \) are input 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,

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

subject to the dynamical constraint (1).

(2)

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, such as Tikhonov regularization, which is typically used for solving...
ill-posed problems. Finlay et al. [17] 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).

**Notation** For convenience, we summarize the notation used in the article:

- \( N_t \): number of time steps in time integration
- \( N_B \): number of time steps in a backward pass
- \( N_s \): number of stages in the time integration method
- \( N_l \): number of layers in the neural network that approximates the vector field \( f \) in (1).
- \( N_b \): number of ODE blocks (one instance of (1) corresponds to one block).
- \( N_c \): number of maximum allowed checkpoints

### 2.2 Discrete adjoint vs continuous adjoint

In order to evaluate the objective \( L \) in (2), both the ODEs (1) and the integral need to be discretized in time. In order to calculate the gradient of \( L \) 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

\[
\frac{d\tilde{\lambda}}{dt} = -\left( \frac{\partial f}{\partial u} \right)^T \tilde{\lambda} - \frac{\partial q}{\partial u}, \tag{3}
\]

\[
\tilde{\lambda}(t_F) = \frac{\partial L}{\partial u(t_F)}, \tag{4}
\]

where \( \tilde{\lambda} \in \mathbb{R}^N \) is an adjoint variable. Its solution gives the gradient of \( L \) with respect to the initial state \( u_0 \). That is, \( \tilde{\lambda}(t_0) = \frac{dL}{du_0} \). The gradient of \( L \) with respect to the weights \( \theta \) is

\[
\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} + \int_{t_0}^{t_F} \left( \tilde{\lambda}^T \frac{\partial f}{\partial \theta} + \frac{\partial q}{\partial \theta} \right) dt. \tag{5}
\]

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 a 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 the 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 \( \mathcal{N} \) that propagates the ODE solution from one time step to another:

\[
u_{n+1} = \mathcal{N}(u_n, \theta). \tag{6}
\]

The adjoint sensitivities are propagated by

\[
\lambda_n = \left( \frac{\partial \mathcal{N}(u_n)}{\partial u} \right)^T \lambda_{n+1},
\]

\[
\mu_n = \left( \frac{\partial \mathcal{N}(u_n)}{\partial \theta} \right)^T \lambda_n + \mu_{n+1}, \quad n = N-1, \ldots, 0,
\]

with the terminal condition

\[
\lambda_N = \left( \frac{\partial L}{\partial u_N} \right)^T, \quad \mu_N = \left( \frac{\partial L}{\partial \theta} \right)^T. \tag{8}
\]

The solutions \( \lambda_0 \) and \( \mu_0 \) represent the sensitivity of \( L \) to the initial state \( u_0 \) and to the parameters \( \theta \), respectively. See [18, 19] for more details.
2.3 Choice of adjoint method 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 \( u_{n+1} \) and \( u_n \), respectively, causing a discrepancy that can be bounded by Proposition 1.

| Table 1: Comparison of different adjoint approaches for forward Euler. |
|---------------------------------------------------------------|
| **forward propagation** | \( u_{n+1} = u_n + hf(u_n, \theta, t_n) \) |
| **continuous adjoint**  | \( \tilde{\lambda}_n = \tilde{\lambda}_{n+1} + h \left( \frac{\partial f(u_{n+1}, \theta, t_{n+1})}{\partial u} \right)^T \tilde{\lambda}_{n+1} \) |
| **discrete adjoint**    | \( \lambda_n = \lambda_{n+1} + h \left( \frac{\partial f(u_n, \theta, t_n)}{\partial u} \right)^T \lambda_{n+1} \) |

**Proposition 1** Assuming \( \tilde{\lambda}_{n+1} = \lambda_{n+1} \), the local discrepancy between the continuous adjoint sensitivity and the discrete adjoint sensitivity is

\[
\| \tilde{\lambda}_n - \lambda_n \| \leq h^2 \| H \left( u_n + \epsilon (u_{n+1} - u_n), \theta, t_n \right) f(u_n, \theta, t_n) \| \| \lambda_{n+1} \|,
\]

where \( H \) is the Hessian of \( f \) and \( \epsilon \) is a constant in \((0, 1)\).

This proposition can be easily proved by using Taylor’s expansion and the triangle inequality for the norm. One can see that \( \tilde{\lambda}_{n+1} = \lambda_n \) for linear functions since the Hessian is zero and for nonlinear functions, such as deep neural networks, \( \lambda_{n+1} \) approaches \( \lambda_n \) 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. Many studies [20–22] have suggested that the discrete adjoint approach produces accurate gradients that match the machine precision, making it more favorable for gradient-based optimization algorithms.

3 Method

In this section we propose a new neural ODE framework, PNODE, based on high-level discrete adjoint methods.

3.1 High-level abstraction of automatic differentiation

Despite the nuances in the different approaches for the gradient computations in neural ODEs, they can be interpreted fundamentally as a unified method that utilizes automatic differentiation (AD) [23] at different abstraction levels. Specifically, the reverse (adjoint) mode of AD can compute the gradient of a scalar differentiable 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 the input. During the evaluation, a list of operations is recorded, which forms a computational graph of the function. To compute the derivatives, one applies the chain rule of differentiation to each node in the computational graph in a backward order.

This procedure can be performed in different ways based on the definition of a primitive operation (a node in the graph). In the continuous adjoint approach adopted in vanilla neural ODEs, the primitive operation is the ODE itself; in the discrete adjoint approach in Section 2.2, the primitive operation is the function evaluation of \( f \) in (1), while the propagation equation (6) for a particular time-stepping algorithm consists of a combination of primitive operations, thus yielding a high-level representation of the computational graph. Differentiation at this level of granularity has several advantages. It can reduce the computation and memory cost. Moreover, function calls to external libraries such as PyTorch can be wrapped into primitives; therefore, PyTorch’s AD engine needs to create only shallow computational graphs within the primitives.
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 $O(N_t N_s N_l)$ 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, however, making this approach infeasible for large problems and long-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 needs to backpropagate only 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 the discrete adjoint method, PNODE checkpoints the solutions at select 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 $(\frac{\partial f}{\partial u})^T$ or the transposed Jacobian-vector product $(\frac{\partial f}{\partial u})^T v$, 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 $O(N_l)$. Note that the high-level discrete adjoint formula (7) can be derived for any time integration method and implemented manually, and it can be reused in different applications. The algorithm for training PNODE is summarized in Algorithm 6.

Algorithm 1 PNODE to achieve accuracy and memory efficiency with high-level adjoint differentiation.

**Forward**
- Solve (1) with a time-stepping algorithm (6)
- Checkpoint the states and stage values for all or selective time steps
- Compute the loss $\mathcal{L}$

**Backward**
- Initialize the adjoint variables $\lambda$ and $\mu$ with (8)
- for $n := N - 1$ to 0 do
  - Restore from the closest checkpoint, and recompute the forward pass to time $t_{n+1}$
  - Compute $\lambda_n$ and $\mu_n$ with (7)
- end for
- Output the gradient $\frac{d\mathcal{L}}{d\theta} = \mu_0$

The checkpointing process in PNODE provides a trade-off between storage and computational overhead. The maximum space needed for checkpointing is $(N_t - 1) \times \text{size of a checkpoint}$, where a checkpoint consists of the state vector and $N_s$ stage vectors for one time step. In the ideal case where memory is sufficient for saving all stages, no time steps are recomputed in the backward pass. With a limited memory budget ($N_c < N_t - 1$), the states that are not checkpointed in the forward pass can be recomputed from a nearby checkpoint. To minimize the number of recomputations, we use the binomial strategy presented in [24, 25]. The algorithm is an extension of the classic Revolve algorithm [26]. Its optimality for multistage time integration methods is proved in [24] and summarized below. We refer readers to [24, 25] for illustrations of the checkpointing procedure.

Proposition 2 [24, 25] 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

$$\tilde{p}(N_t, N_c) = (t - 1) N_t - \left(\frac{N_c + t}{t - 1}\right) + 1,$$

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

The total memory cost of PNODE thus consists of two parts: backpropagation for $f$ and checkpointing for the adjoint calculation. Backpropagation requires $O(N_t)$ memory, while the memory cost of checkpointing is at most $O((N_t-1)(N_c+1))$.  

5
We solve both the linear systems in the forward pass and the transposed systems in the reverse pass with a matrix-free λ with respect to the adjoint variable.

The checkpointing method for ANODE: a framework with discrete adjoint and checkpointing method [11] we assume that the number of reverse time steps in the continuous adjoint method is also A summary comparing PNODE with representative implementations of neural ODEs is given in Table 2. For simplicity, the computation cost of PNODE can be split into three parts: the forward computation, the reverse computation, and the recomputation overhead.

The application of implicit time integration is highly desired for stiff dynamical systems where explicit methods may fail because of stability constraints. Backpropagating through the implicit solver is difficult, however, because of 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. For example, the discrete adjoint of the backward Euler method in a simplified form (with no integral term in loss function (2))

\[ M u_{n+1} = M u_n + h_n \theta f(u_{n+1}) \]

is

\[ M^T \lambda_s = \lambda_{n+1} + h f_u^T(u_{n+1}) \lambda_s, \]
\[ \lambda_n = \lambda_{n+1} + h f_u^T(u_{n+1}) \lambda_s, \]
\[ \mu_n = \mu_{n+1} + h f_u^T(u_{n+1}) \lambda_s, \]

where \( M \) is the mass matrix. At each reverse step, (12) requires solving the transposed linear system \( M^T - h f_u^T(u_{n+1}) \) with respect to the adjoint variable \( \lambda_s \).

We solve both the linear systems in the forward pass and the transposed systems in the reverse pass with a matrix-free iterative method for efficiency. The action of the matrix or its transpose is computed by using PyTorch functions to backpropagate \( f \). Again, only the computational graph for \( f \) needs to be created in our approach.

### 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 because of stability constraints. Backpropagating through the implicit solver is difficult, however, because of 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. For example, the discrete adjoint of the backward Euler method in a simplified form (with no integral term in loss function (2))

\[ M u_{n+1} = M u_n + h_n \theta f(u_{n+1}) \]

is

\[ M^T \lambda_s = \lambda_{n+1} + h f_u^T(u_{n+1}) \lambda_s, \]
\[ \lambda_n = \lambda_{n+1} + h f_u^T(u_{n+1}) \lambda_s, \]
\[ \mu_n = \mu_{n+1} + h f_u^T(u_{n+1}) \lambda_s, \]

where \( M \) is the mass matrix. At each reverse step, (12) requires solving the transposed linear system \( M^T - h f_u^T(u_{n+1}) \) with respect to the adjoint variable \( \lambda_s \).

We solve both the linear systems in the forward pass and the transposed systems in the reverse pass with a matrix-free iterative method for efficiency. The action of the matrix or its transpose is computed by using PyTorch functions to backpropagate \( f \). Again, only the computational graph for \( f \) needs to be created in our approach.

### 3.5 Comparison with existing methods

A summary comparing PNODE with representative implementations of neural ODEs is given in Table 2. For simplicity, we assume that the number of reverse time steps in the continuous adjoint 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 [24].

**NODE cont:** the original implementation in [4] with continuous adjoint The vanilla neural ODE [4] 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 \( O(N_t) \) to backpropagate \( f \). The backward ODE solve requires a recomputation cost of \( O(N_t N_s) \).

**NODE naive:** a variant of the original implementation in [4] This is a naive method that backpropagates the ODE solvers and has the deepest computational graph, but it has no recomputational overhead.

**ANODE:** a framework with discrete adjoint and checkpointing method [11] The checkpointing method for ANODE has the same memory cost \( O(N_t) \) as NODE naive 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 memory cost for checkpointing is \( O(N_b) \), where \( N_b \) is the number of ODE blocks, and the backpropagation requires \( O(N_t/N_b N_s N_t) \) memory. Each ODE block needs to be recomputed in the backward
pass, so the total recomputation cost is $O(N_t N_s)$. A generalization of ANODE has been implemented in a Julia library [27].

ACA: the adaptive checkpoint adjoint (ACA) method [13] ACA is similar to ANODE but uses a slightly different checkpointing strategy. ACA checkpoints the state at each time step, thus consuming $O(N_t)$ memory for checkpointing. To save memory, ACA deletes redundant computational graphs for rejected time steps when searching for the optimal step size. Backpropagation is applied to each time step, so it requires $O(N_t N_l)$ memory. In the backward pass, ACA first performs an additional forward pass to save the checkpoints and then recomputes each time step to generate the local computational graph, resulting in a total recomputation cost of $O(2N_t N_s)$.

3.6 Implementation

Drawing on state-of-the-art adjoint-capable ODE solvers (TSAdjoint in PETSc [28]), 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. 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 solver such as FATODE [19] and DiffEqSensitivity.jl [29], PETSc has several favorable features that other tools 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 [22, 30]. It includes explicit and implicit methods, implicit-explicit methods, multirate methods with various stability properties, and adaptive time-stepping. The discrete adjoint approach proposed in Section 2.2 has been implemented for some time integrators and can be easily expanded to others [18].

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 products, 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 In addition to the offline binomial checkpointing algorithms [24–26], PETSc supports more sophisticated checkpointing algorithms such as online algorithms [31] for hierarchical storage systems.

HPC-friendly linear algebra kernels PETSc has full-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 [32].

4 Experimental Results

In this section we test the performance of PNODE and compare it with existing neural ODE methods on two distinct benchmark tasks: image classification and continuous normalizing flows (CNF). We then demonstrate the application of PNODE to learning stiff dynamics. All the experiments are conducted on an NVIDIA Tesla V100 GPU with a memory capacity of 32 GB. When testing PNODE, unless specified otherwise, we assume $N_c \geq N_t$ and checkpoint all the intermediate solutions and stage values for the best speed. Note that this approach leads to no recomputation, thus giving the best speed but also the worst-case memory cost for PNODE.

Existing frameworks may support only a subset of the time integration schemes used in our experiments. For a comprehensive and fair evaluation and comparison, we have added the same time integration schemes in all frameworks if they were not originally implemented. In the benchmark tests, we focused on fixed step schemes instead of adaptive step schemes because error estimation and adaptive strategies can be vastly different across different frameworks, making a fair comparison difficult, especially when investigating memory consumption, which is sensitive to the number of time steps. The results for fixed step schemes can reflect the fundamental differences among the neural ODE methods and indicate the expected performance of these methods for adaptive step schemes.
Figure 1: Training and testing performance of SqueezeNext on CIFAR10 using various 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.
4.1 Image classification

Chen et al. [4] showed that replacing residual blocks with ODE blocks yields good accuracy and parameter efficiency on the MNIST dataset from [33]. However, the effect of the accuracy of the gradients cannot be determined easily with this simple dataset. Here we experiment on the more complex CIFAR-10 dataset from [34] using a SqueezeNext network introduced by [35], where every nontransition block is replaced with a neural ODE block. We use 4 ODE blocks of different dimensions with 199,800 trainable parameters in total.

Accuracy We train the same neural ODE structure using one time step while varying the time integration method. The results are displayed in Figure 1. The ReLU activation in this model results in irreversible dynamics and inaccurate gradient calculation for the vanilla neural ODE [4], leading to divergent training with the Euler method and the RK4 method and to suboptimal accuracy. PNODE and other neural ODE methods converge to higher accuracy due to the reverse accuracy guaranteed by the discrete adjoint method and automatic differentiation. With the low-accuracy methods (Euler and Midpoint), PNODE achieves the highest accuracy among all the methods tested.

Memory/time efficiency We perform a systematic comparison of the efficiency and memory cost for different methods by varying \( N_t \). For PNODE, we include an additional variant (denoted by PNODE2) that saves only the ODE solution at each time step. This leads to \( N_t - 1 \) recomputations in the reverse pass. Thus PNODE2 has almost the same memory cost as ACA. As shown in Figure 2, PNODE significantly outperforms all the other methods in terms of per-epoch training time. When using Dopri5, PNODE is three times faster than ANODE, 2.3 times faster than ACA, and two times faster than the vanilla neural ODE. Similar speedups can be observed for other time integration methods. Among all reverse-accurate methods, PNODE has the lowest memory growth as the number of time steps increases. For example, using Dopri5 with 11 time steps, the memory consumption of PNODE is approximately 71% less than NODE naive and 55% less than ANODE. Compared with PNODE, PNODE2 reduces the memory cost by up to 42% with a slight increase in training time. PNODE2 has a similar memory cost but much faster training speed than ACA, which agrees with the theoretical analysis shown in Table 2. Note that the CUDA runtime allocates \( \sim 0.4 \) GB memory for PNODE, and it is included in all the PNODE results presented in this paper. This overhead is inevitable for loading any library that contains CUDA kernels.

4.2 Continuous normalizing flow for density estimation

We select three datasets—POWER, MINIBOONE, and BSDS3000—that are respectively 6-, 43-, and 63-dimensional tabular datasets commonly used in CNF [36]. The FFJORD [5] approach is used to transform a multivariate Gaussian distribution to the target distributions for all three datasets.

We adopt the tuned architecture and hyperparameters (e.g., learning rates, number of hidden layers, and number of flow steps) from [5]. We vary the number of steps for different time integration methods only for stability considerations. On all three datasets, all the methods that converge yield comparable testing losses, but all the reverse-accurate methods converge much faster than NODE cont because of more accurate gradient estimates. On BSDS300, NODE cont failed to converge after 14 days of training and was terminated prematurely. Similar observations were reported in [12].

The performance statistics including the number of function evaluations per iteration, the training time per iteration, and the maximum GPU memory usage are shown in Tables 3–7. NFE-F and NFE-B are the number of function evaluations in the forward pass and backward pass, respectively. NFE-B for PNODE and NODE cont reflects the cost of the transposed Jacobian-vector products, while NFE-B for other methods reflects the cost for recomputing the time steps in the backward pass. The observed results agree with our theoretical analysis in Table 2.

Excluding the 0.4 GB constant overhead, PNODE has the lowest memory consumption among all reverse-accurate methods, and it consistently outperforms ACA and ANODE in terms of training time for all three datasets. When using the Dopri5 scheme that is commonly used in neural ODEs, PNODE is 28.7% faster and consumes 68.2% less memory than does ACA for BSDS300. The advantage becomes more evident for high-order schemes that have more stages. The reason is that the computational graph for automatic differentiation grows deeper as the number of stages increases.

NODE naive and ANODE run out of GPU memory for BSDS300. But when memory is sufficient, NODE naive is the fastest, as expected. For MINIBOONE, the difference in memory consumption is marginal for all methods because of a number of factors such as small batch size (1000), small number of steps (4), and small neural network.

\(^{2}\text{https://github.com/dmlc/dlpack}\)
Figure 2: GPU memory usage and time per epoch of different implementations as functions of the number of time steps ($N_t$) for various schemes.
Table 3: Performance statistics for the Euler scheme. Missing values are due to out-of-memory errors.

| Dataset  | Framework | Integration method | NFE-F | NFE-B | Time per iteration (s) | GPU Mem (GB) |
|----------|-----------|--------------------|-------|-------|------------------------|--------------|
| POWER    | NODE naive| Euler, \( N_t=50 \) | 250   | 0     | 0.760                  | 11.648       |
|          | NODE cont |                    | 250   | 250   | 1.101                  | 1.680        |
|          | ANODE     |                    | 250   | 250   | 1.297                  | 3.624        |
|          | ACA       |                    | 250   | 505   | 1.382                  | 1.735        |
|          | PNODE     |                    | 250   | 250   | **1.117**              | **2.104**    |
| MINIBOONE| NODE naive| Euler, \( N_t=20 \) | 20    | 0     | 0.071                  | 2.481        |
|          | NODE cont |                    | 20    | 20    | 0.094                  | 1.716        |
|          | ANODE     |                    | 20    | 20    | 0.108                  | 2.500        |
|          | ACA       |                    | 20    | 41    | 0.102                  | 1.762        |
|          | PNODE     |                    | 20    | 20    | **0.099**              | **2.085**    |
| BSDS300  | NODE naive|                    | –     | –     | –                      | –            |
|          | NODE cont |                    | 200   | 200   | 13.409                 | 3.826        |
|          | ANODE     |                    | –     | –     | –                      | –            |
|          | ACA       |                    | 200   | 402   | 17.191                 | 5.564        |
|          | PNODE     |                    | 200   | 200   | **13.541**             | **4.920**    |

Table 4: Performance statistics for the Midpoint scheme.

| Dataset  | Framework | Integration method | NFE-F | NFE-B | Time per iteration (s) | GPU Mem (GB) |
|----------|-----------|--------------------|-------|-------|------------------------|--------------|
| POWER    | NODE naive| Midpoint, \( N_t=40 \) | 400   | 0     | 1.368                  | 17.659       |
|          | NODE cont |                    | 400   | 400   | 1.780                  | 1.680        |
|          | ANODE     |                    | 400   | 400   | 1.975                  | 4.822        |
|          | ACA       |                    | 400   | 805   | 2.109                  | 1.819        |
|          | PNODE     |                    | 400   | 400   | **1.883**              | **2.152**    |
| MINIBOONE| NODE naive| Midpoint, \( N_t=16 \) | 32    | 0     | 0.104                  | 2.991        |
|          | NODE cont |                    | 32    | 32    | 0.134                  | 1.737        |
|          | ANODE     |                    | 32    | 32    | 0.167                  | 3.012        |
|          | ACA       |                    | 32    | 65    | 0.145                  | 1.846        |
|          | PNODE     |                    | 32    | 32    | **0.137**              | **2.087**    |
| BSDS300  | NODE naive| Midpoint, \( N_t=80 \) | 320   | 320   | 21.404                 | 3.826        |
|          | NODE cont |                    | 320   | 320   | 24.977                 | 7.525        |
|          | ANODE     |                    | 320   | 642   | 24.977                 | 7.525        |
|          | ACA       |                    | 320   | 320   | **21.651**             | **5.388**    |

4.3 Learning stiff dynamics

Stiff dynamical systems are characterized by widely separated time scales, which pose computational difficulties for explicit time integration methods. Several researchers [37–39] have recognized that learning stiff dynamics from time-series data is also challenging for data-driven approaches such as neural ODEs. The reasons are twofold. First, the computational difficulties in solving stiff ODEs remain in neural ODEs. Second, stiffness could lead to ill-conditioned gradients in classical neural ODEs. Kim et al. [38] proposed applying scaling to the differential equations and loss functions to mitigate stiffness. In this section we apply feature scaling to the input data and then use an implicit method for training the neural ODEs. We demonstrate the success of our approach with a well-known stiff chemical reaction system and illuminate the benefits of using implicit solvers, which are equipped with discrete adjoint capabilities and efficient nonlinear and linear solvers, uniquely enabled through PNODE.
Table 5: Performance statistics for the Bosh3 scheme.

| Dataset  | Framework | Integration method | NFE-F | NFE-B | Time per iteration (s) | GPU Mem (GB) |
|----------|-----------|--------------------|-------|-------|------------------------|--------------|
| POWER    | NODE naive | Bosh3, $N_t=30$ | 465   | 0     | 1.532                  | 20.282       |
|          | NODE cont  | Bosh3, $N_t=30$ | 465   | 465   | 3.058                  | 6.555        |
|          | ANODE      |                    | 450   | 450   | 2.593                  | 10.106       |
|          | ACA        |                    | 450   | 905   | 2.632                  | 6.771        |
|          | PNODE      |                    | 455   | 450   | **2.084**              | **2.217**    |
| MINIBOONE| NODE naive | Bosh3, $N_t=12$ | 32    | 0     | 0.104                  | 2.991        |
|          | NODE cont  | Bosh3, $N_t=12$ | 39    | 39    | 0.164                  | 1.737        |
|          | ANODE      |                    | 36    | 36    | 0.187                  | 3.182        |
|          | ACA        |                    | 36    | 73    | 0.219                  | 1.930        |
|          | PNODE      |                    | 37    | 36    | **0.153**              | **2.091**    |
| BSDS300  | NODE naive | Bosh3, $N_t=60$ | –     | –     | –                      | –            |
|          | NODE cont  | Bosh3, $N_t=60$ | –     | –     | –                      | –            |
|          | ANODE      |                    | –     | –     | –                      | –            |
|          | ACA        |                    | 360   | 722   | 31.001                 | 9.821        |
|          | PNODE      |                    | 362   | 360   | **24.439**             | **6.145**    |

Table 6: Performance statistics for the RK4 scheme.

| Dataset  | Framework | Integration method | NFE-F | NFE-B | Time per iteration (s) | GPU Mem (GB) |
|----------|-----------|--------------------|-------|-------|------------------------|--------------|
| POWER    | NODE naive | RK4, $N_t=20$ | 400   | 0     | 1.615                  | 22.529       |
|          | NODE cont  | RK4, $N_t=20$ | 400   | 400   | 2.291                  | 6.553        |
|          | ANODE      |                    | 400   | 400   | 2.345                  | 9.697        |
|          | ACA        |                    | 400   | 805   | 2.365                  | 6.834        |
|          | PNODE      |                    | 400   | 400   | **1.765**              | **2.156**    |
| MINIBOONE| NODE naive | RK4, $N_t=8$  | 32    | 0     | 0.110                  | 2.991        |
|          | NODE cont  | RK4, $N_t=8$  | 32    | 32    | 0.131                  | 1.737        |
|          | ANODE      |                    | 32    | 32    | 0.168                  | 3.012        |
|          | ACA        |                    | 32    | 65    | 0.200                  | 2.016        |
|          | PNODE      |                    | 32    | 32    | **0.135**              | **2.089**    |
| BSDS300  | NODE naive | RK4, $N_t=40$ | –     | –     | –                      | –            |
|          | NODE cont  | RK4, $N_t=40$ | –     | –     | –                      | –            |
|          | ANODE      |                    | –     | –     | –                      | –            |
|          | ACA        |                    | 320   | 642   | 27.035                 | 12.053       |
|          | PNODE      |                    | 320   | 320   | **21.681**             | **5.422**    |

The system we consider is governed by Robertson’s equations:

\[
\begin{align*}
\frac{du_1}{dt} &= -k_1 u_1 + k_3 u_2 u_3, \\
\frac{du_2}{dt} &= k_1 u_1 - k_2 u_2^2 - k_3 u_2 u_3, \\
\frac{du_3}{dt} &= k_2 u_2^2,
\end{align*}
\]

where $u_1$, $u_2$, $u_3$ are the concentrations of three species and $k_1 = 0.04$, $k_2 = 3 \times 10^7$, $k_3 = 10^4$ are reaction rate constants. We train a neural ODE in form 1 with the right-hand side function approximated by a neural network. The loss function is defined as the discrepancy between the observed data and the prediction by the ML model

\[
\mathcal{L} = \text{MAE}(u^{ob}(t), u^{pred}(t)),
\]

where MAE is the mean absolute error. The neural network has five hidden layers with an activation function of GELU as used in [38].
Table 7: Performance statistics for the Dopri5 scheme.

| Dataset   | Framework   | Integration method | NFE-F | NFE-B | Time per iteration (s) | GPU Mem (GB) |
|-----------|-------------|--------------------|-------|-------|------------------------|--------------|
| POWER     | NODE naive  | Dopri5, N_t=10     | 300   | 0     | 0.976                  | 13.653       |
|           | NODE cont   | 300 300            | 300   | 300   | 1.357                  | 1.687        |
| ANODE     | D           | 300 300            | 300   | 300   | 1.524                  | 4.035        |
| ACA       | D           | 300 605            | 300   | 605   | 1.816                  | 2.131        |
| PNODE     |             | 305 300            | 305   | 300   | **1.329**              | **2.150**    |

| MINIBOONE | NODE naive  | Dopri5, N_t=10     | 24    | 0     | 0.074                  | 2.653        |
|           | NODE cont   | 24 24              | 24    | 24    | 0.120                  | 1.758        |
| ANODE     | D           | 24 24              | 24    | 24    | 0.131                  | 2.672        |
| ACA       | D           | 24 49              | 24    | 49    | 0.143                  | 2.184        |
| PNODE     |             | 25 24              | 25    | 24    | **0.105**              | **2.089**    |

| BSDS300   | NODE naive  | Dopri5, N_t=10     | 240   | 240   | 16.175                 | 3.983        |
|           | NODE cont   | 240 240            | 240   | 240   | 21.003                 | 16.667       |
| ANODE     | D           | 240 482            | 240   | 482   | **16.314**             | **5.304**    |

The training data are generated by solving (13) with the initial conditions of \([u_1, u_2, u_3] = [1, 0, 0]\) over a time span of \([10^{-5}, 100]\) and sampling 40 data points equally spaced in a logarithm scale. We train the neural ODE for 10,000 epochs on a GPU using the AdamW optimizer with an initial learning rate of 0.005.

4.3.1 Data preprocessing

Since the three species have widely varying concentrations (5 orders of magnitude), their contributions to the loss function are not proportionate, thus leading to slow convergence in training. Figure 3(c) shows the results learned by using the raw data after 10,000 epochs’ training. The predictions for the first and third species match the ground truth well; however, the second species shows a noticeable discrepancy. To remedy this issue, we apply the standard min-max normalization

\[ u' = \frac{u - \min(u)}{\max(u) - \min(u)} \]  

(15)
to scale the input data to the range \([0, 1]\).

4.3.2 Implicit versus explicit

PNODE enables us to use an implicit method to integrate the forward model and use its discrete adjoint to compute the gradients for training. Here, we demonstrate the advantage of implicit methods by comparing them with the adaptive explicit methods that are widely used in existing neural ODE frameworks because of their efficiency and ease of implementation. For implicit methods, we use the Crank–Nicolson (CN) scheme. The nonlinear systems arising at each time step are solved with a Newton method, and the linear systems are solved with a matrix-free GMRES method [40]. For explicit methods we use the adaptive Dopri5 method with tolerances \(abstol = reltol = 10^{-6}\). As shown in Figure 4, PNODE with CN can learn the dynamics perfectly, while Dopri5 fails to match the ground truth. This result is expected because of the limitations of explicit methods in handling stiff systems. Figure 4 shows that when using Dopri5, the gradient explodes, preventing the convergence of the training loss.

4.3.3 Computation cost

The training costs for Dopri5 and CN are given in Table 8. We can see that training with CN is slightly faster than training with Dopri5 for the Robertson’s equations. Note that function evaluations are required not only for the time integration but also for Autograd to generate the Jacobian-vector product and the transposed Jacobian-vector product, which are needed in the adjoint calculation. In principle, explicit methods have lower per-step cost than implicit methods have because they do not require solving linear systems. In our case, however, they require a large number of steps according to Table 8. Initially, the training model is not stiff, requiring relatively few time steps when integrating the neural ODE. As the training model approaches the ground truth, however, the stiffness increases, causing the number of time steps to increase as well.
Figure 3: Ground truth of Robertson’s equations and the predictions of the neural ODE models with different training methods.

Figure 4: Training results on Robertson’s equations with the Crank–Nicolson method (left) and with the Dopri5 method (right). Training loss is plotted on the top, and gradient norm is plotted on the bottom. The gradients for Dopri5 explode after about 1,500 epochs.
Table 8: Computation cost comparison between Dopri5 and CN.

| Integration method | Average NFE-F | Average NFE-B | Average time per iteration (s) |
|--------------------|---------------|---------------|-------------------------------|
| CN                 | 505           | 136           | 0.709                         |
| Dopri5             | 805           | 805           | 0.778                         |

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 the discrete adjoints derived from a time integration scheme can be cast as a high-level abstraction of automatic differentiation and produce gradients exact to machine precision. Adopting this approach, one can avoid backpropagating through an ODE solver or a time integration procedure and can minimize the depth of the computational graph for backpropagation, while guaranteeing reverse accuracy. With high-level adjoint differentiation and the checkpointing technique, we successfully reduce the memory cost of neural ODEs to $O(N_t N_s) + O(N_l)$ when checkpointing all intermediate states (including ODE solutions and stage vectors) and $O(N_l) + O(N_l)$ when checkpointing solutions only, while a naive automatic differentiation approach requires a memory cost of $O(N_t N_s N_l)$. Extensive numerical experiments on image classification and continuous normalizing flow problems show that PNODE achieves the best memory efficiency and training speed among the existing neural ODEs that are reverse-accurate. Furthermore, our high-level adjoint method not only allows a balance between memory and computational costs but also offers more flexibility in the solver design than traditional neural ODEs have. We demonstrate that PNODE enables the application of implicit integration methods, which offers more possibilities for stabilizing the training of stiff dynamical systems. We have made PNODE freely available and believe that accelerated memory-efficient neural ODEs will benefit a broad range of artificial intelligence applications, especially for scientific machine learning tasks such as the discovery of unknown physics.

References

[1] Y. Lu, A. Zhong, Q. Li, and B. Dong, “Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations,” in Proceedings of the 35th International Conference on Machine Learning, vol. 80. PMLR, 10–15 Jul 2018, pp. 3276–3285.

[2] E. Haber and L. Ruthotto, “Stable architectures for deep neural networks,” Inverse Problems, vol. 34, no. 1, p. 014004, 12 2017.

[3] L. Ruthotto and E. Haber, “Deep neural networks motivated by partial differential equations,” Journal of Mathematical Imaging and Vision, vol. 62, no. 3, pp. 352–364, 2020.

[4] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, “Neural ordinary differential equations,” in Advances in Neural Information Processing Systems, vol. 31. Curran Associates, Inc., 2018.

[5] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, and D. Duvenaud, “Scalable reversible generative models with free-form continuous dynamics,” in International Conference on Learning Representations, 2019.

[6] Y. Rubanova, R. T. Q. Chen, and D. K. Duvenaud, “Latent ordinary differential equations for irregularly-sampled time series,” in Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., 2019.

[7] S. Greydanus Google Brain, M. Dzamba PetCube, and J. Yosinski, “Hamiltonian neural networks,” in Advances in Neural Information Processing Systems, vol. 32, 2019.

[8] I. Ayed, E. de Bézenac, A. Pajot, J. Brajard, and P. Gallinari, “Learning dynamical systems from partial observations,” arXiv preprint arXiv:1902.11136, 2019.

[9] E. Dupont, A. Doucet, and Y. W. Teh, “Augmented neural ODEs,” in Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., 2019.

[10] J. Jia and A. R. Benson, “Neural jump stochastic differential equations,” in Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., 2019.

[11] A. Gholaminejad, K. Keutzer, and G. Biros, “ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs,” in Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 730–736.

[12] D. Onken and L. Ruthotto, “Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows,” arXiv preprint arXiv:2005.13420, 2020.
[13] J. Zhuang, N. Dvornek, X. Li, S. Tatikonda, X. Papademetris, and J. Duncan, “Adaptive checkpoint adjoint method for gradient estimation in neural ODE,” in Proceedings of the 37th International Conference on Machine Learning, vol. 119, 2020, pp. 11 639–11 649.

[14] T. Zhang, Z. Yao, A. Gholami, J. E. Gonzalez, K. Keutzer, M. W. Mahoney, and G. Biros, “ANODEV2: A coupled neural ODE framework,” in Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., 2019.

[15] J. Zhuang, N. C. Dvornek, S. Tatikond, and J. S. Duncan, “MALI: A memory efficient and reverse accurate integrator for neural ODEs,” in International Conference on Learning Representations, 2021.

[16] T. Beck, “Automatic differentiation of iterative processes,” Journal of Computational and Applied Mathematics, vol. 50, pp. 109–119, 1994.

[17] C. Finlay, J.-H. Jacobsen, L. Nurbekyan, and A. Oberman, “How to train your neural ODE: the world of Jacobian and kinetic regularization,” in Proceedings of the 37th International Conference on Machine Learning, vol. 119. PMLR, 13–18 July 2020, pp. 3154–3164.

[18] H. Zhang, E. M. Constantinescu, and B. F. Smith, “PETSc TSAdjoint: a discrete adjoint ODE solver for first-order and second-order sensitivity analysis,” SIAM Journal on Scientific Computing, vol. 44, no. 1, pp. C1–C24, 2022.

[19] H. Zhang and A. Sandu, “FATODE: a library for forward, adjoint, and tangent linear integration of ODEs,” SIAM Journal on Scientific Computing, vol. 36, no. 5, pp. C504–C523, 2014.

[20] E. J. Nielsen and W. L. Kleb, “Efficient construction of discrete adjoint operators on unstructured grids using complex variables,” AIAA Journal, vol. 44, no. 4, pp. 827–836, 2006.

[21] E. J. Nielsen and B. Diskin, “Discrete adjoint-based design for unsteady turbulent flows on dynamic overset unstructured grids,” AIAA Journal, vol. 51, no. 6, pp. 1355–1373, 2013.

[22] H. Zhang, S. Abhyankar, E. Constantinescu, and M. Anitescu, “Discrete adjoint sensitivity analysis of hybrid dynamical systems with switching,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 64, no. 5, pp. 1247–1259, 2017.

[23] J. Nocedal and S. J. Wright, Numerical Optimization, 2nd ed. New York, NY, USA: Springer, 2006.

[24] H. Zhang and E. Constantinescu, “Revolve-based adjoint checkpointing for multistage time integration,” in International Conference on Computational Science, (online), in main track, 2021.

[25] H. Zhang and E. Constantinescu, “Optimal checkpointing for adjoint multistage time-stepping schemes,” arXiv preprint arXiv:2106.13879, 2022.

[26] A. Griewank and A. Walther, “Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation,” ACM Transaction on Mathematical Software, vol. 26, no. 1, p. 19–45, Mar. 2000.

[27] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and A. Ramadhan, “Universal differential equations for scientific machine learning,” arXiv preprint arXiv:2001.04385, 2020.

[28] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, “PETSc users manual,” Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.14, 2020.

[29] C. Rackauckas and Q. Nie, “DifferentialEquations.jl — a performant and feature-rich ecosystem for solving differential equations in Julia,” The Journal of Open Research Software, vol. 5, no. 1, 2017.

[30] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang, “PETSc/TS: a modern scalable ODE/DAE solver library,” arXiv preprint arXiv:1806.01437, 2018.

[31] P. Stumm and A. Walther, “New algorithms for optimal online checkpointing,” SIAM Journal on Scientific Computing, vol. 32, no. 2, pp. 836–854, 2010.

[32] R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and J. Zhang, “Toward performance-portable PETSc for GPU-based exascale systems,” Parallel Computing, vol. 108, p. 102831, 2021.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

[34] A. Krizhevsky, “Learning multiple layers of features from tiny images,” University of Toronto, 05 2012.
[35] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, “SqueezeNext: Hardware-aware neural network design,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, June 2018.

[36] G. Papamakarios, T. Pavlakou, and I. Murray, “Masked autoregressive flow for density estimation,” in Advances in Neural Information Processing Systems, vol. 30, 2017.

[37] A. Ghosh, H. Behl, E. Dupont, P. Torr, and V. Namboodiri, “STEER: Simple temporal regularization for neural ODE,” in Advances in Neural Information Processing Systems, vol. 33. Curran Associates, Inc., 2020, pp. 14831–14843.

[38] S. Kim, W. Ji, S. Deng, Y. Ma, and C. Rackauckas, “Stiff neural ordinary differential equations,” Chaos: An Interdisciplinary Journal of Nonlinear Science, vol. 31, no. 9, p. 093122, 2021.

[39] S. Liang, Z. Huang, and H. Zhang, “Stiffness-aware neural network for learning Hamiltonian systems,” in International Conference on Learning Representations, 2022.

[40] Y. Saad and M. H. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” SIAM Journal on Scientific and Statistical Computing, vol. 7, no. 3, pp. 856–869, 1986.