MIXED PRECISION TRAINING OF NEURAL ODES*

ELENA CELLEDONI†, BRYNJULF OWREN*‡, LARS RUTHOTTO §, AND NICOLE TIANJIAO YANG¶

Abstract. Exploiting low-precision computations has become a standard strategy in deep learning to address the growing computational costs imposed by ever larger models and datasets. However, naïvely performing all computations in low precision can lead to roundoff errors and instabilities. Therefore, mixed precision training schemes usually store the weights in high precision and use low-precision computations only for whitelisted operations. Despite their success, these principles are currently not reliable for training continuous-time architectures such as neural ordinary differential equations (Neural ODEs). This paper presents a mixed precision training framework for neural ODEs consisting of explicit ODE solvers and a custom backpropagation scheme and shows their effectiveness in a range of learning tasks. Our scheme uses low-precision computations for evaluating the velocity, parameterized by the neural network, and for storing intermediate states, while stability is provided by a custom dynamic adjoint scaling and by accumulating the solution and gradients in higher precision. These contributions address two key challenges in training neural ODE: the computational cost of repeated network evaluations and the growth of memory requirements with the number of time steps or layers. Along with the paper we publish our extendable, open-source PyTorch package rampde whose syntax resembles that of leading packages to provide a drop-in replacement in existing codes. We demonstrate the reliability and effectiveness of our scheme using challenging test cases and on neural ODE applications in image classification and generative models, achieving approximately 50% memory reduction and up to 2× speedup while maintaining accuracy comparable to single-precision training.

Key words. Mixed precision computing, deep learning, neural ODEs

MSC codes. 68T07, 65L06, 65G50

1. Introduction.

Mixed precision training (MPT) has become a standard practice in deep learning to mitigate the growth of computational costs associated with increasing models and data sizes. It is increasingly used in various learning tasks, including natural language processing, computer vision [14], and it has also made inroads into scientific machine learning [11].

Mixed precision training aims to reduce the computational costs of training by evaluating as many compute-intensive parts of the deep network in low-precision arithmetic as possible, while using as few high-precision operators as necessary to preserve training stability and match the accuracy of networks trained using high-precision arithmetic. The two most significant aspects in which MPT lowers computational costs are enabling the training of larger models and larger batch sizes by storing hidden activations and features in low precision and reducing computational time when models are sized to take advantage of tensor compute units available in modern hardware accelerators such as graphical processing units (GPUs).

In addition to the growing size of models and data, the widespread use of MPT

*Submitted to the editors DATE.

Funding: LR's and NTY's work was supported by the Office of Naval Research award N00014-24-1-2221/ P00003 and also in part by NSF award DMS 2038118. The project was supported by the Horizon Europe, MSCA-SE project 101131557 (REMODEL).

†NTNU – Norwegian University of Science and Technology, Trondheim, Norway (elena.celledoni@ntnu.no, https://www.ntnu.edu/employees/elena.celledoni)

‡(brynjulf.owren@ntnu.no, https://www.ntnu.edu/employees/brynjulf.owren)

§Emory University, Atlanta, GA, USA (lruthotto@emory.edu, https://www.math.emory.edu/~lruthot/)

¶University of Tennessee, Knoxville, TN, USA (nicole.yang@utk.edu, https://nicoletyang.github.io)

in deep learning can be explained by several characteristics of the problem [13]. First, training is inherently stochastic due to the presence of noise in the data and updates in stochastic gradient methods. Second, deep networks are composed of a small subset of linear algebra routines, often involving dense arrays, which are ideal targets for developing specialized hardware that reduces computation times and energy consumption. Third, the optimization problem is only a surrogate for the actual goal of learning and therefore does not need to be solved with high accuracy.

Most MPT algorithms today maintain the network weights in high precision to prevent cancellation errors during optimization steps, but quantize the weights to low precision to accelerate the forward and backward passes. During network evaluations, whitelisted, compute-intensive operators are evaluated in low precision, high precision is used for delicate operations such as accumulation and some nonlinearities, and the loss function is scaled before backpropagation to maximize the range and avoid overflow and underflow [20]. This strategy balances computational efficiency with numerical stability: compute-intensive operations benefit from hardware acceleration while sensitive accumulations maintain accuracy through higher precision. Developing MPT algorithms is also simplified by the integration of these algorithms into modern deep learning software, such as PyTorch, JAX, and Keras.

Despite these advances, existing MPT techniques can be unreliable for training neural ordinary differential equations (Neural ODEs), which are continuous-time models in which the forward propagation involves the numerical solution of initial value problems whose dynamics are parameterized by a neural network [5, 10, 4]. Due to the nonlinearity of neural networks, Neural ODEs typically rely on explicit time integrators, which can necessitate many steps and thereby exacerbate computational costs and memory requirements. Calculating the gradient of the loss function with respect to the network weights requires solving a continuous or discrete adjoint equation. We demonstrate theoretically and experimentally that the straightforward application of MPT can lead to the accumulation of roundoff errors during time integration, resulting in growing errors in the solution and gradients as the number of time steps increases. The main contributions of this paper are:

1. **Mixed precision ODE solvers and backpropagation**: We design and analyze mixed precision explicit solvers for neural ODEs with a custom backpropagation scheme that stores weights, states, and adjoints in high precision while evaluating networks in low precision, achieving approximately 50% memory reduction and up to 2× speedup in our largest example.
2. **Dynamic adjoint scaling**: We develop an adaptive scaling heuristic that maximizes the range during backpropagation, prevents underflow in float16 where standard techniques can fail without additional hyperparameter tuning.
3. **Theoretical analysis**: We show that roundoff errors remain in the order of the unit roundoff of the low precision and do not grow uncontrollably with the number of time steps.
4. **Implementation**: We provide the open-source PyTorch package `rampde` (robust automatic mixed precision for differential equations) that serves as a drop-in replacement for existing neural ODE implementations at
   https://github.com/EmoryMLIP/rampde

Our paper is structured as follows. In section 2, we provide some background on mixed precision algorithms in scientific computing and deep learning. In section 3, we describe our mixed precision training framework for neural ODEs. In section 4 we perform a roundoff error analysis of our forward and backward schemes to show that errors are dominated by roundoff errors in the low-precision network evaluation

and nearly independent of the number of integration steps. In section 5, we describe our open-source implementation `rampde`. In section 6, we demonstrate reliability and broad applicability to neural ODE training problems arising in image classification and generative modeling. In section 7, we discuss our findings and outline the potential for future improvements.

2. Background. Let us introduce our main notation, briefly review the mixed precision training (MPT) framework for deep learning, and provide background for our contribution by discussing recent works on MPT in scientific machine learning.

Notation. Without loss of generality, we describe an unsupervised training of a neural network $F : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^m$ with n input features, m output features, and p parameters using samples $\mathbf{x}$ from the data distribution π_{data} by minimizing the regularized loss function

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim\pi_{\text{data}}}\left[\ell(F(\mathbf{x},\boldsymbol{\theta}))\right] + \frac{\alpha}{2}\|\boldsymbol{\theta}\|^2.$$

Here, the choice of loss function $\ell : \mathbb{R}^m \to \mathbb{R}$ depends on the task, and we use a Tikhonov regularizer (in this context known as weight decay) controlled by the parameter $\alpha \geq 0$. The design of the neural network also varies depending on the context. When training the network using stochastic approximation schemes such as stochastic gradient descent (SGD), the objective function and its gradient are approximated using a batch of independent and identically distributed (i.i.d.) samples $\mathcal{B} = \{\mathbf{x}_1, \ldots, \mathbf{x}_b\}$

$$\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|}\sum_{x_i\in\mathcal{B}} \ell(F(\mathbf{x}_i,\boldsymbol{\theta})) + \frac{\alpha}{2}\|\boldsymbol{\theta}\|^2.$$

Mixed Precision Training (MPT). Let us briefly review the principles of MPT proposed in [20] that are widely adopted in deep learning, discuss their implementation in modern machine learning software, and their use in scientific applications. These principles establish the foundation for our neural ODE-specific adaptations, particularly in handling the unique challenges of time integration. The goal of MPT is to combine low- and high-precision arithmetic to reduce the costs (in terms of memory, energy, and computational time) of network training while achieving similar precision compared to pure high-precision training.

Following standard practice of deep learning today, we use 16-bit floating point formats (either `float16` or `bfloat16`) as low precision and `float32` as high precision. The main difference between the low-precision formats is the allocation of bits between the mantissa and exponent. The `float16` or IEEE-754 half precision has one sign, five exponent bits, and ten mantissa bits, while `bfloat16`, developed by Google [15], has one sign, eight exponent bits, and seven mantissa bits. The `bfloat16` format is appealing because its range coincides with that of `float32`, which usually alleviates the need to modify the network architecture and adjust hyperparameters and also simplifies conversion. In comparison, `float16`, uses more bits for the mantissa, resulting in higher precision but a smaller range.

We denote by $Q_{16} : \mathbb{R}^n \to \mathcal{F}_{16}^n$ and $Q_{32} : \mathbb{R}^n \to \mathcal{F}_{32}^n$ the quantization operators that map the elements of an input vector or tensor to the set of representable numbers in the 16-bit and 32-bit floating point representations, respectively.

For ease of presentation, consider a simple stochastic gradient descent (SGD) scheme for training the network, and note that MPT can be similarly used for other stochastic approximation methods. Given a master copy of the weights $\boldsymbol{\theta}_k$ in high

precision, the kth iteration of SGD in MPT performs the update

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \gamma_k Q_{32}(\nabla(S_k \cdot \mathcal{L}_{\mathcal{B}_k}(Q_{16}(\boldsymbol{\theta}_k))))/S_k, \tag{2.1}$$

where $\mathcal{B}_k$ is a randomly sampled batch from the training set, $\gamma_k > 0$ is the step size (or learning rate), and S_k is a loss scaling factor that aims to maximize the range of the low-precision floating point system and avoid underflow and overflow during derivative computations. In MPT, the input features are also converted to match the precision of the weights. The sequence of learning rates is chosen by the user and the loss scale S_k is updated heuristically during training. Choosing S_k effectively is crucial since too small values can lead to underflow, which means that small entries in the gradient are lost, and too large values can lead to overflow, which means that updates need to be discarded. The most widely used heuristics therefore halve S_k when gradients overflow and double it when no overflow occurred in a user-defined number of the last few consecutive steps.

Since a single scaling factor cannot keep the scales of all hidden layers in a meaningful range for deep and complex network architectures, adaptive loss scaling schemes use layer-dependent scaling factors [30]. The proposed heuristic for computing the scaling factors automatically during training reduced the time to convergence and improved accuracy compared to standard MPT techniques in examples from image classification and object detection.

Modern deep learning software packages simplify the use and experimentation with mixed precision algorithms by providing automatic mixed precision (AMP) contexts. These AMP contexts whitelist compute-intensive operations that are usually safe in low precision (e.g., dense matrix-matrix operations, convolutions, etc.) and blacklist more delicate operations (e.g., norms, accumulations, nonlinearities such as softmax, etc.) When evaluating the neural network and its gradients in an AMP context, the inputs of the whitelisted operations are automatically converted and performed with that precision. Although this adds convenience, it can complicate the interpretation of mixed precision schemes such as (2.1) since only parts of weights and operations may be performed in low precision.

Another practical aspect of MPT on modern GPU hardware is that some linear algebra operations leverage mixed precision algorithms internally. One prominent example is the introduction of block fused multiply-adds (FMAs) in tensor compute units, which is described and analyzed in detail in [1]. Block FMAs add accuracy for operations on 16-bit tensors, as accumulation is done in single precision intermediates that are then rounded to half precision [15]. With appropriately sized matrix dimensions, observed speed-ups of tensor compute units can be up to 8 times with a reduced loss of accuracy compared to pure 16-bit operations [1, 20].

Mixed precision training has also been used in scientific machine learning (SciML) tasks such as solving partial differential equations (PDEs) with Physics-Informed Neural Networks (PINNs) and Deep Operator Networks (DeepONet). As expected, pure low-precision training can fail in SciML tasks, but the use of the MPT framework can lead to effective training at reduced costs when hyperparameters and loss functions are chosen adequately [11].

Neural ODEs. Instead of defining the neural network as the composition of a finite number of layers, Neural ODEs are continuous-time models that define $F(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{y}(T)$ as the terminal state of the initial value problem

$$\frac{d}{dt}\mathbf{y}(t) = f(t, \mathbf{y}(t), \boldsymbol{\theta}(t)), \quad \text{for } t \in (0, T], \quad \text{and} \quad \mathbf{y}(0) = \mathbf{x}. \tag{2.2}$$

Here, T is the terminal time, $f : [0,T] \times \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^n$ is a neural network with potentially time-dependent weights $\boldsymbol{\theta} : [0,T] \to \mathbb{R}^p$. Although first instances of continuous-time recurrent neural networks can be dated back at least to the 1980s (see historical notes in [19]) and ODE-based architectures have also been used for time series regression in the 1990s (see, for example, [25]), their recent interpretation of continuous residual networks has contributed to their reemergence; see [5, 10]. The term neural ODE was coined in [4], and their use has spread to other deep learning tasks such as generative modeling [9, 21, 6], mean field games [27], and optimal control [22]. A widely used implementation in the PyTorch framework is the `torchdiffeq` project [3]. Generalizations to rough paths and stochastic dynamics also provide the basis of Neural SDEs [16] and score-based generative models [28].

A key distinguishing factor in neural ODE implementations is the order of differentiation and discretization. Optimize-then-discretize algorithms such as [4, 9], derive the gradient of the objective function in continuous time, and use adaptive time integrators for the forward and adjoint process to approximate the update in each step. Although this simplifies their use, the implicit function theorem requires that both ODEs be solved relatively accurately to approximate the continuous gradient with sufficient accuracy [7, 23]. Discretize-then-optimize first discretize the problem in time and then compute the gradient of the discrete problem analytically or through automatic differentiation. The main advantage is that the accuracy of the time integrator can be chosen arbitrarily without compromising the accuracy of the (discrete) gradient. In general, and in particular in machine learning tasks using a lower accuracy for the ODE solver is attractive and can lead to significant speedups when the data is noisy and the ODE is not derived from first principles. In an MPT framework it is not clear that the ODE can be solved as accurately as required in the optimize-then-discretize approach (i.e., the adaptive solver may fail because required step sizes are rounded to zero or accuracy is limited by low-precision evaluations of f) and therefore, we limit the discussion to discretize-then-optimize methods.

Mixed precision implementations of additive Runge-Kutta (RK) schemes have been proposed in [8] and their performance has been analyzed in [2]. These works aim to speed up implicit steps of the solvers using low precision (here single precision), while using high precision (here double or quadruple) for explicit steps to maintain overall accuracy. Performance evaluations on IBM Power9 and Intel chips show speedups and reductions in energy costs for specific solver parameters and software kernels. Those schemes could be attractive for neural ODEs in an optimize-then-discretize approach; however, implicit steps are rarely used in deep learning due to the nonlinearity introduced by the neural network, which often has limited exploitable structure for the nonlinear solver. As noted above, following the ODE trajectory with high accuracy is usually less important in a learning context than in physical models, as long as the neural ODE leads to effective learning. In order to apply these schemes in a discretize-then-optimize approach, one could extend these schemes by deriving a mixed precision implementation of the differentiated time integrator following the general principles we propose in this work for simpler time integrators.

3. Mixed Precision Training of Neural ODEs. We describe our mixed precision scheme for optimization problems governed by a neural ODE such as (2.2), the custom backpropagation scheme to compute gradients of the loss with respect to time, input feature and weights, and our dynamic adjoint scaling to maximize the range of the low precision.

Let $\mathbf{t} \in \mathbb{R}^{N+1}$ with $0 = \mathbf{t}_0 < \mathbf{t}_1 < \cdots < \mathbf{t}_N = T$ be a partition of the time interval

Algorithm 3.1 Mixed Precision Forward Pass

Require: time steps $0 \le \mathbf{t}_0 < \mathbf{t}_1 < \cdots < \mathbf{t}_N = T$, initial state $\mathbf{y}_0 = \mathbf{x}$, weights $\boldsymbol{\theta}$, increment function $\Phi(f, \mathbf{y}, t, h, \boldsymbol{\theta})$
Ensure: state trajectory $\{Q_{16}(\mathbf{y}_i)\}_{i=0}^N$ in low precision
1: $\mathbf{y} \leftarrow Q_{32}(\mathbf{x})$ {state in high precision}
2: **for** $i = 0$ **to** $N-1$ **do**
3: $\quad h_i \leftarrow \mathbf{t}_{i+1} - \mathbf{t}_i$
4: $\quad d\mathbf{y} \leftarrow \Phi(f, Q_{16}(\mathbf{y}), \mathbf{t}_i, h_i, Q_{16}(\boldsymbol{\theta}))$ {evaluate increment in low precision}
5: $\quad \mathbf{y} \leftarrow \mathbf{y} + h_i \cdot Q_{32}(d\mathbf{y})$ {accumulate in high precision}
6: $\quad \mathbf{y}_{i+1} \leftarrow Q_{16}(\mathbf{y})$ {store state in low precision}
7: **end for**
8: **return** $\{\mathbf{y}_i\}_{i=0}^N$

and let $h_i = \mathbf{t}_{i+1} - \mathbf{t}_i$ be the size of the ith step where $i = 0, \ldots, N-1$. For ease of presentation, we assume a batch-size of one example. Given the initial state $\mathbf{y}_0 = \mathbf{y}(0) = \mathbf{x}$, we are interested in approximating the ODE solution $\mathbf{y}_i \approx \mathbf{y}(\mathbf{t}_i)$ with an explicit single-step method of the form

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \Phi(f, \mathbf{y}_i, \mathbf{t}_i, h_i, \boldsymbol{\theta}),$$

where different choices of the increment function, Φ, lead to different time stepping schemes. For example, $\Phi(f, \mathbf{y}_i, \mathbf{t}_i, h_i, \boldsymbol{\theta}) = f(\mathbf{t}_i, \mathbf{y}_i, \boldsymbol{\theta}(\mathbf{t}_i))$ leads to the forward Euler method.

To limit roundoff errors during time integration, we store the current state and compute the accumulation in high-precision, which gives the mixed precision step

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i Q_{32}(\Phi(f, Q_{16}(\mathbf{y}_i), \mathbf{t}_i, h_i, Q_{16}(\boldsymbol{\theta}))).$$

Similar to (2.1), the computationally expensive part of evaluating the neural network is performed in low precision. To save memory, our method stores and returns the intermediate states $\{\mathbf{y}_i(\mathbf{t}, \mathbf{x}, \boldsymbol{\theta})\}_{i=0}^N$ in low precision; see Algorithm 3.1 for a summary.

Using the output from the forward propagation, we approximate an objective functional of the form

$$\mathcal{L}(t, \mathbf{x}, \boldsymbol{\theta}) = \int_0^T R(t, \mathbf{y}(t), \boldsymbol{\theta}(t))dt + C(\mathbf{y}(T))$$

with running cost R and terminal cost C. We use a quadrature, leading to the discrete objective function

$$L(\{\mathbf{y}_i(\mathbf{t}, \mathbf{x}, \boldsymbol{\theta})\}_{i=0}^N, \boldsymbol{\theta}, \mathbf{t}) = \sum_{i=0}^N w_i R(\mathbf{t}_i, \mathbf{y}_i, \boldsymbol{\theta}(\mathbf{t}_i)) + C(\mathbf{y}_N). \tag{3.1}$$

In the backward pass, we differentiate the objective function with respect to the intermediate states of the ODE, the neural network weights, and the time steps by reversing the time integration. We initialize $\mathbf{a} = Q_{32}\left(\frac{\partial C}{\partial \mathbf{y}_N}\right)$, $\mathbf{g} = Q_{32}\left(\frac{\partial L}{\partial \boldsymbol{\theta}}\right) \in \mathbb{R}^p$, and $\mathbf{t}' = Q_{32}(\frac{\partial L}{\partial \mathbf{t}}) \in \mathbb{R}^{N+1}$.

Our dynamic adjoint scaling scheme, which can be seen as an adaptation of the layer-wise scaling in [30] to the ODE setting, scales the adjoint variable so that the range of the low-precision system is maximally exploited. To minimize the risk of

underflow, we initialize the adjoint scale to $S_N = 2^{\lfloor \log_2(-u_L \|\mathbf{a}\|) \rfloor}$ to ensure that $\|S_N \mathbf{a}\| \approx \frac{1}{u_L}$. Here, u_L is the unit roundoff of the low precision, $\|\cdot\|$ is the infinity norm of a vector. For the `float16` format it is $u_L := 2^{-11}$ and for `bfloat16` it is $u_L = 2^{-8}$. To avoid additional errors from rounding the mantissa, we round the loss scale to a power of two so that the scaling only affects the exponent.

For $i = N-1, N-2, \ldots, 0$, we use automatic differentiation to compute the vector-Jacobian products (vjp) with $\mathbf{J}_\Phi = \left[\left(\frac{\partial \Phi}{\partial \mathbf{y}}\right)^\top, \left(\frac{\partial \Phi}{\partial \mathbf{t}}\right)^\top, \left(\frac{\partial \Phi}{\partial h}\right)^\top, \left(\frac{\partial \Phi}{\partial \boldsymbol{\theta}}\right)^\top \right]$ in low precision to compute the updates

$$[d\mathbf{a}^\top, d\mathbf{t}^\top, dh^\top, d\boldsymbol{\theta}^\top] = Q_{16}(S_i \cdot \mathbf{a})^\top \mathbf{J}_\Phi(f, \mathbf{y}_i, \mathbf{t}_i, h_i, \boldsymbol{\theta}). \tag{3.2}$$

Since all these derivatives are along the direction $\mathbf{a}$, the computations are combined into one call of the automatic differentiation and the computational graph is built only once by recomputing the increment function using the ith state in low precision.

If any of the steps, $d\mathbf{a}$, $d\mathbf{t}$, $dh, d\boldsymbol{\theta}$ contains any overflow or not-a-number (NaN), we halve the scale factor setting $S_i \leftarrow S_i/2$ and repeat (3.2) until all quantities are finite. These computations re-use the computational graph and do not require additional recomputation of the increment function. When dynamic scaling is not used, our implementation either performs no overflow checking for optimal performance (float32, bfloat16) or returns infinity gradients for compatibility with PyTorch's GradScaler (float16 without dynamic scaling).

After successful computation of the updates, we accumulate the derivative information in high precision by setting

$$\mathbf{t}_i' = \mathbf{t}_i' + (h_i/S_i) Q_{32}(d\mathbf{t}_i - dh_i) - Q_{32}\left(\Phi(f, \mathbf{y}_i, \mathbf{t}_i, h_i, \boldsymbol{\theta})^\top \mathbf{a}\right) \tag{3.3}$$

$$\mathbf{t}_{i+1}' = \mathbf{t}_{i+1}' + (h_i/S_i) dh_i + Q_{32}\left(\Phi(f, \mathbf{y}_i, \mathbf{t}_i, h_i, \boldsymbol{\theta})^\top \mathbf{a}\right) \tag{3.4}$$

$$\mathbf{a} = \mathbf{a} + w_i Q_{32}\left(\frac{\partial R}{\partial \mathbf{y}}\right) - (h_i/S_i) Q_{32}(d\mathbf{a}) \tag{3.5}$$

$$\mathbf{g} = \mathbf{g} + (h_i/S_i) Q_{32}(d\mathbf{g}). \tag{3.6}$$

If no re-scaling was needed in the Jacobian computation, all these quantities are finite, and the adjoint variable is small (i.e., $\|\mathbf{a}\| u_L \le \frac{1}{2}$) we double the scale factor for the next time step. This completes the step, and we continue with the $(i-1)$th step.

This process yields the total derivatives of the loss, $\frac{dL}{d\mathbf{x}} = \mathbf{a}$, $\frac{dL}{d\boldsymbol{\theta}} = \mathbf{g}$, $\frac{dL}{d\mathbf{t}} = \mathbf{t}'$, and is summarized in Algorithm 3.2.

Example 3.1. To assess the effectiveness of our custom backpropagation and dynamic adjoint scaling, we consider the following test problem

$$y'(t) = -\lambda(t) y(t), \quad \text{where} \quad \lambda(t) = \boldsymbol{\theta}_1 t^2 + \boldsymbol{\theta}_2 t + \boldsymbol{\theta}_3, \quad \text{and } y(0) = x \tag{3.7}$$

whose analytical solution is

$$y(t) = x \cdot \exp\left(-\left(\frac{\boldsymbol{\theta}_1}{3} t^3 + \frac{\boldsymbol{\theta}_2}{2} t^2 + \boldsymbol{\theta}_3 t\right)\right). \tag{3.8}$$

We select the parameters of this problem such that $|y(t)|$ and $|y'(t)|$ span nearly the entire range of the normal numbers in `float16`, which is $[2^{-14}, 65504]$. We choose the terminal time $T = 2.65$, the weights $\boldsymbol{\theta} = (8, -11, 2^{-16})$, and the initial state $x = \frac{65504}{180}$. We use a 4th-order Runge-Kutta method with 400 equidistant time steps. As can be seen by the solid lines in Figure 1, the analytical solution and its derivative vary by

Algorithm 3.2 Mixed Precision Backward Pass with Dynamic Adjoint Scaling

Require: state trajectory $\{\mathbf{y}_i\}_{i=0}^N$, loss function $L(\{\mathbf{y}_i(\mathbf{t},\mathbf{x},\boldsymbol{\theta})\}_{i=0}^N,\boldsymbol{\theta},\mathbf{t})$, parameters $\boldsymbol{\theta}\in\mathbb{R}^p$, maximum number of attempts $k_{\max}$
Ensure: total derivatives $\frac{dL}{d\mathbf{x}}$, $\frac{dL}{d\boldsymbol{\theta}}$, $\frac{dL}{d\mathbf{t}}$

$\mathbf{a}\leftarrow Q_{32}\left(\frac{\partial L}{\partial \mathbf{y}_N}\right), \mathbf{g}=Q_{32}\left(\frac{\partial L}{\partial \boldsymbol{\theta}}\right), \mathbf{t}'=Q_{32}\left(\frac{\partial L}{\partial \mathbf{t}}\right)$ {partials in high precision}
$S_N\leftarrow 2^{\lfloor \log_2(-u_{\mathrm{L}}\|\mathbf{a}\|)\rfloor}$ {initialize scaling factor}
$\boldsymbol{\theta}\leftarrow Q_{16}(\boldsymbol{\theta})$ {weights in low precision}
for $i=N-1$ **to** 0 **do**
  $h_i\leftarrow \mathbf{t}_{i+1}-\mathbf{t}_i$
  $d\mathbf{y}\leftarrow \Phi(f,\mathbf{y}_i,\mathbf{t}_i,h_i,\boldsymbol{\theta})$ {re-compute with low precision}
  $S_i\leftarrow S_{i+1}$
  for $k=1,\dots,k_{\max}$ **do**
    $d\mathbf{a}^\top, d\mathbf{t}_i, dh_i, d\boldsymbol{\theta}^\top\leftarrow Q_{16}(S_i\cdot\mathbf{a})^\top\mathbf{J}_\Phi(f,\mathbf{y}_i,\mathbf{t}_i,h_i,\boldsymbol{\theta})$
    if $d\mathbf{a}, d\boldsymbol{\theta}, d\mathbf{t}_i, dh$ all finite **then**
      break for loop and continue
    else
      $S_i\leftarrow S_i/2$
    end if
  end for
  $\mathbf{t}'_i\leftarrow \mathbf{t}'_i+(h_i/S_i)Q_{32}(d\mathbf{t}_i-dh_i)-Q_{32}(\Phi(f,\mathbf{y}_i,\mathbf{t}_i,h_i,\boldsymbol{\theta})^\top\mathbf{a}_{i+1})$ {high-precision}
  $\mathbf{t}'_{i+1}\leftarrow \mathbf{t}'_{i+1}+(h_i/S_i)dh_i+Q_{32}(\Phi(f,\mathbf{y}_i,\mathbf{t}_i,h_i,\boldsymbol{\theta})^\top\mathbf{a}_{i+1})$ {high-precision update}
  $\mathbf{a}\leftarrow \mathbf{a}+w_iQ_{32}\left(\frac{\partial R}{\partial \mathbf{y}}\right)-(h_i/S_i)Q_{32}(d\mathbf{a})$ {high-precision update}
  $\mathbf{g}\leftarrow \mathbf{g}+(h_i/S_i)Q_{32}(d\mathbf{g})$ {high-precision update}
  if $k=1$ (no scaling was needed) **and** $\|\mathbf{a}\|\le\frac{1}{2u_{\mathrm{L}}}$ **then**
    $S_i\leftarrow 2S_i$
  end if
end for
return $\frac{dL}{d\mathbf{x}}=\mathbf{a}$, $\frac{dL}{d\boldsymbol{\theta}}=\mathbf{g}$, $\frac{dL}{d\mathbf{t}}=\mathbf{t}'$

orders of magnitude and remain within the normal range of float16. The low-precision approximation (indicated by a dashed red line) fits the analytical solution closely.

In Table 1, we report the relative errors between the true quantities and the numerical approximations of the terminal state and the derivatives of the loss function $L(\{y_i\}_{i=1}^N,\boldsymbol{\theta},\mathbf{t})=\frac{1}{2}\|y_N\|_2^2$ with respect to the initial state and the components of $\boldsymbol{\theta}$ for the precision of float32 and float16 and with and without scaling. As expected, scaling does not affect the accuracy of high-precision computations but is crucial in float16. Here, derivative computations underflow without scaling, whereas our dynamic scaling heuristic yields gradient approximations nearly as accurate as the forward solve. It is important to note that simply scaling the loss function to avoid underflow would cause overflow at intermediate time steps in this example.

4. Roundoff Error Analysis. We show that, for sufficiently regular ODEs, the relative errors of the forward solution, adjoint variables, and weight gradients between the exact time integrator and our mixed precision scheme are of the order of the low precision unit roundoff and do not grow uncontrolled as the number of time steps, N, increases.

For brevity, we limit our analysis to the error introduced by finite-precision arithmetic but note that our result can be used to bound the overall error between the

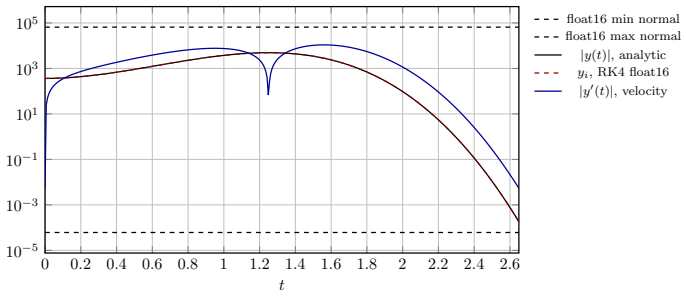

Fig. 1: Log-scale plot of $|y(t)|$ (black solid line) and $|y'(t)|$ (blue solid line) used in Theorem 3.1 and the numerical approximation of the solution using RK4 with 400 equidistant time steps in float16. The horizontal dashed lines indicate the range of the normal numbers in float16. In this set up, the ODE almost exhausts the range of float16, complicating the derivative computation.

dtype	**scaling**	RE $y(T)$	RE ∂_{y_0}	RE ∂_{θ_1}	RE ∂_{θ_2}	RE ∂_{θ_3}
float32	none	7.01e-5	1.40e-4	1.25e-4	1.30e-4	1.34e-4
float32	dynamic	7.01e-5	1.60e-4	1.28e-4	1.35e-4	1.45e-4
float16	none	3.67e-3	1.99e+3	1.00e+0	1.00e+0	1.00e+0
float16	dynamic	3.67e-3	5.89e-3	6.05e-3	5.96e-3	5.88e-3
bfloat16	none	3.65e-2	4.50e-2	5.24e-2	4.96e-2	4.73e-2
bfloat16	dynamic	3.65e-2	4.49e-2	5.24e-2	4.95e-2	4.73e-2

Table 1: Comparing the relative errors (RE) of numerical approximations of the terminal state and the derivatives of $L(\{y_i\}_{i=1}^N, \boldsymbol{\theta}, \mathbf{t}) = \frac{1}{2}\|y_N\|_2^2$ with respect to the initial state and weights for different precisions and scaling strategies in Theorem 3.1. The table demonstrates the necessity and effectiveness of dynamic adjoint scaling in float16, which leads to gradient approximations with accuracy comparable to the forward solve. As expected, the gradient scaling has little effect on float32 and bfloat16 as they have the same range.

numerical and true solution. To this end, we note that the total error comprises the discretization error and the mixed precision error. If the latter can be bounded independently of the step size, this means that our mixed precision scheme is expected to show the same order of convergence until the discretization error reaches that level.

In our analysis, we consider a fixed $\boldsymbol{\theta}$ and equidistant time grid with step size h to simplify our notation; that is, we write the update mapping as $\Phi = \Phi(\mathbf{y})$ suppressing the dependence on f, h and $\boldsymbol{\theta}$. Unless otherwise noted, products and divisions of two vectors in this section are to be understood component-wise. We denote exact quantities (computed in infinite precision) without decoration (for example, $\Phi, \mathbf{y}_i, \mathbf{a}_i$) and rounded quantities (computed in finite precision) with a tilde (for example, $\tilde{\Phi}, \tilde{\mathbf{y}}_i, \tilde{\mathbf{a}}_i$), following the convention of Higham [12]. We denote error vectors as $\mathbf{e}_\mathbf{x} = \mathbf{x} - \tilde{\mathbf{x}}$ (exact minus rounded) and, for vectors $\mathbf{x}$ with non-zero components, relative error vectors as $\boldsymbol{\delta}_\mathbf{x} = \frac{\mathbf{e}_\mathbf{x}}{\mathbf{x}}$. The vectors $\mathbf{r}_L$ and $\mathbf{r}_H$ with $\|\mathbf{r}_L\| \leq u_\mathrm{L}$ and $\|\mathbf{r}_H\| \leq u_\mathrm{H}$ denote the relative quantization errors and u_L and u_H are the unit roundoffs of the low and high preci-

sion, respectively. We assume round-to-nearest mode, the IEEE 754 default rounding mode. As common, we measure all errors in the infinity norm of a vector and to ease notation, we drop the subscript on norms.

The quantization errors satisfy

$$Q_{16}(\mathbf{x}) = \tilde{\mathbf{x}} = (\mathbf{1} + \mathbf{r}_{16}) \cdot \mathbf{x}, \quad \text{for} \quad \|\mathbf{r}_{16}\| \leq u_{\mathrm{L}},$$

where $\mathbf{1}$ is a vector of all ones and $\tilde{\mathbf{x}}$ denotes the rounded value. We further assume that $Q_{32}(\mathbf{x})$ introduces no error when converting from low to high precision, as low-precision values are exactly representable in high precision. However, rounding the result of an addition in high precision introduces an error

$$Q_{32}(\mathbf{x} + \mathbf{y}) = (\mathbf{1} + \mathbf{r}_{32}) \cdot (\mathbf{x} + \mathbf{y}) \quad \text{with} \quad \|\mathbf{r}_{32}\| \leq u_{\mathrm{H}}.$$

To simplify our notation, we ignore the running costs as they can be added to the state vector.

We denote with $D\Phi$ and ∇C the Jacobian of Φ and gradient of C in exact arithmetic, and with $D\tilde{\Phi}(\mathbf{y})$ and $\nabla \tilde{C}$ their finite precision counterparts obtained using automatic differentiation and finite precision arithmetic.

Assumption 4.1.

1. **Smoothness and boundedness**: The increment function Φ is bounded and has bounded first and second derivatives, and we denote with L_Φ and $L_{D\Phi}$ constants such that

$$\|D\Phi(\mathbf{y})\| \leq L_\Phi, \quad \|D^2\Phi(\mathbf{y})\| \leq L_{D\Phi}, \quad \forall \mathbf{y} \in \mathbb{R}^d, \quad i = 1, \ldots, N, \tag{4.1}$$

and similarly for its finite precision implementation we assume there is a constant $L_{\tilde{\Phi}}$ such that

$$\|D\tilde{\Phi}(\mathbf{y})\| \leq L_{\tilde{\Phi}}, \quad \text{for all } \mathbf{y} \in \mathbb{R}^d, \quad i = 1, \ldots, N. \tag{4.2}$$

This constant depends on the number and nature of floating point operations performed in each evaluation.
The cost function C is differentiable and has bounded derivatives, with Lipschitz constant $L_{\nabla C}$, for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$,

$$\|\nabla C(\mathbf{x}) - \nabla C(\mathbf{y})\| \leq L_{\nabla C} \|\mathbf{x} - \mathbf{y}\|. \tag{4.3}$$

2. **Behavior along the states sequences**: The components of the states $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$ are bounded above by $M_{\mathbf{y}}$. If components are zero or near-zero, one could add a small shift $m_{\mathbf{y}}$ to avoid division by zero when deriving relative errors. We also assume there are constants $G_{\mathbf{y}}$ and $G_{\mathbf{a}}$ such that

$$G_{\mathbf{y}} := \max_i \left\| \frac{\tilde{\Phi}_i(\mathbf{y}_i)}{\tilde{\mathbf{y}}_{i+1}} \right\|, \qquad G_{\mathbf{a}} := \max_i \left\| \frac{d\tilde{\mathbf{a}}_i}{\tilde{\mathbf{a}}_{i-1}} \right\|.$$

And there are constants $\mu_{\mathbf{y}}$ and $\mu_{\mathbf{a}}$ such that

$$\mu_{\mathbf{y}} := \max_{0 \leq i \leq N-1} \|\tilde{\mathbf{y}}_i\| \, \|\tilde{\mathbf{y}}_{i+1}^{-1}\|, \qquad \mu_{\mathbf{a}} := \max_{0 \leq i \leq N-1} \|\tilde{\mathbf{a}}_{i+1}\| \, \|\tilde{\mathbf{a}}_i^{-1}\|.$$

There is a constant $m_{\nabla C}$ such that $m_{\nabla C} := \min_i \|\nabla C(\mathbf{y}_i)\|$.

3. **Relative error bounds in terms of the precision**: There exist constants M_Φ, $M_{D\Phi}$, and $M_{\nabla C}$ and the following bounds of the relative error of low-precision evaluations. Along $\{\mathbf{y}_i\}_{i=0}^N$ we have

$$\boldsymbol{\delta}_{\Phi(\mathbf{y}_i)} := \frac{\Phi(\mathbf{y}_i) - \tilde{\Phi}(\mathbf{y}_i)}{\tilde{\Phi}(\mathbf{y}_i)}, \qquad \|\boldsymbol{\delta}_{\Phi(\mathbf{y}_i)}\| \le M_\Phi u_{\mathrm{L}}, \tag{4.4}$$

$$\|D\Phi(\mathbf{y}_i)^\top - D\tilde{\Phi}(\mathbf{y}_i)^\top\| \le M_{D\Phi} u_{\mathrm{L}} \|D\tilde{\Phi}(\mathbf{y}_i)^\top\|, \tag{4.5}$$

$$\boldsymbol{\delta}_{\nabla C} := \frac{\nabla C(\mathbf{y}_i) - \nabla \tilde{C}(\mathbf{y}_i)}{\nabla \tilde{C}(\mathbf{y}_i)}, \qquad \|\boldsymbol{\delta}_{\nabla C}\| \le M_{\nabla C} u_{\mathrm{L}}. \tag{4.6}$$

The goal of this section is to prove the following error estimates.

THEOREM 4.2 (Main result). *Under Theorem* 4.1 *and with* $\|\boldsymbol{\delta}_{\mathbf{y}_0}\| = \mathcal{O}(u_{\mathrm{L}}) + \mathcal{O}(u_{\mathrm{H}}/h)$*, our mixed precision time integrator and custom backward satisfy:*

1. *accuracy of the state:* $\|\boldsymbol{\delta}_{\mathbf{y}_i}\| = \mathcal{O}(u_{\mathrm{L}}) + \mathcal{O}(u_{\mathrm{H}}/h)$*;*
2. *accuracy of the adjoint:* $\|\boldsymbol{\delta}_{\mathbf{a}_i}\| = \mathcal{O}(u_{\mathrm{L}}) + \mathcal{O}(u_{\mathrm{H}}/h)$*;*
3. *accuracy of the gradient:* $\|\boldsymbol{\delta}_{\mathbf{g}}\| = \mathcal{O}(u_{\mathrm{L}}) + \mathcal{O}(u_{\mathrm{H}}/h)$*.*

The theorem has some practical implications for choosing a step size. For $h \gg \sqrt{u_{\mathrm{H}}}$ we expect the $\mathcal{O}(u_{\mathrm{L}})$ term to dominate and the mixed precision scheme to work well. For $h \ll \sqrt{u_{\mathrm{H}}}$, the accumulation error of $\mathcal{O}(u_{\mathrm{H}}/h)$ can dominate the error, limiting the performance of mixed precision training. In between, the performance depends on the constants.

LEMMA 4.3 (Forward error). *Under Theorem* 4.1*, for all* $i = 1, \dots, N$*, the relative forward propagation error* $\boldsymbol{\delta}_{\mathbf{y}_i} = \frac{\mathbf{y}_i - \tilde{\mathbf{y}}_i}{\tilde{\mathbf{y}}_i}$ *satisfies*

$$\|\boldsymbol{\delta}_{\mathbf{y}_i}\| \le \|\boldsymbol{\delta}_{\mathbf{y}_0}\| \exp(\mathbf{t}_i K_1) + (\exp(\mathbf{t}_i K_1) - 1)\left(K_2 u_{\mathrm{L}} + K_3 \frac{u_{\mathrm{H}}}{h}\right), \tag{4.7}$$

for constants K_1, K_2, K_3 *independent of* h*.*

Proof. The ith step of the exact and mixed precision scheme are, respectively,

$$\begin{aligned} \mathbf{y}_{i+1} &= \mathbf{y}_i + h\Phi(\mathbf{y}_i) \\ (\mathbf{1} + \mathbf{r}_H^{i+1})\tilde{\mathbf{y}}_{i+1} &= (\tilde{\mathbf{y}}_i + h\tilde{\Phi}(\tilde{\mathbf{y}}_i)) \end{aligned}$$

Subtracting the two gives the error

$$\mathbf{y}_{i+1} - \tilde{\mathbf{y}}_{i+1} = (\mathbf{y}_i - \tilde{\mathbf{y}}_i) + h\left(\Phi(\mathbf{y}_i) - \tilde{\Phi}(\tilde{\mathbf{y}}_i)\right) + \mathbf{r}_H^{i+1}\tilde{\mathbf{y}}_{i+1}.$$

Using Taylor's theorem, the error in the step computation can be expressed as

$$\begin{aligned} \Phi(\mathbf{y}_i) - \tilde{\Phi}(\tilde{\mathbf{y}}_i) &= \Phi(\mathbf{y}_i) - \Phi(\tilde{\mathbf{y}}_i) + \Phi(\tilde{\mathbf{y}}_i) - \tilde{\Phi}(\tilde{\mathbf{y}}_i) \\ &= D\Phi(\tilde{\mathbf{x}}_i)\mathbf{e}_{\mathbf{y}_i} + \mathbf{e}_{\Phi(\mathbf{y}_i)} \end{aligned}$$

where $D\Phi$ is the Jacobian of the increment function Φ, the jth component of $\tilde{\mathbf{x}}_i$ is between the corresponding components of $\tilde{\mathbf{y}}_i$ and $\mathbf{y}_i$, and $\mathbf{e}_{\Phi(\mathbf{y}_i)} = \Phi(\tilde{\mathbf{y}}_i) - \tilde{\Phi}(\tilde{\mathbf{y}}_i)$. Substituting this into (1) gives

$$\mathbf{e}_{\mathbf{y}_{i+1}} = (\mathbf{I} + hD\Phi(\tilde{\mathbf{x}}_i))\mathbf{e}_{\mathbf{y}_i} + h\mathbf{e}_{\Phi(\mathbf{y}_i)} + \mathbf{r}_H^{i+1}\tilde{\mathbf{y}}_{i+1},$$

where $\mathbf{I}$ denotes the identity matrix. At this point, we can already see that the error due to the low-precision computation of the increment function is scaled by h while

the roundoff error from adding the increment is not, which can cause it to dominate the error propagation as the total number of time steps, N, increases. To get a relative error, we divide component-wise by $\tilde{\mathbf{y}}_{i+1}$ and simplify

$$\boldsymbol{\delta}_{\mathbf{y}_{i+1}} = \tilde{\mathbf{y}}_{i+1}{}^{-1}(\mathbf{I} + hD\Phi(\tilde{\mathbf{x}}_i))\tilde{\mathbf{y}}_i\boldsymbol{\delta}_{\mathbf{y}_i} + h\frac{\mathbf{e}_{\Phi(\mathbf{y}_i)}}{\tilde{\mathbf{y}}_{i+1}} + \mathbf{r}_H^{i+1},$$

and obtain the bound

$$\|\boldsymbol{\delta}_{\mathbf{y}_{i+1}}\| \leq \left\|\tilde{\mathbf{y}}_{i+1}{}^{-1}\left(\mathbf{I} + hD\Phi(\tilde{\mathbf{x}}_i)\right)\tilde{\mathbf{y}}_i\right\| \|\boldsymbol{\delta}_{\mathbf{y}_i}\| + h\left\|\frac{\mathbf{e}_{\Phi(\mathbf{y}_i)}}{\tilde{\mathbf{y}}_{i+1}}\right\| + \|\mathbf{r}_H^{i+1}\|.$$

To obtain a step-dependent bound on the first term, we use (4.1) and obtain

$$\left\|\tilde{\mathbf{y}}_{i+1}{}^{-1}\left(\mathbf{I} + hD\Phi(\mathbf{y}_i)\right)\tilde{\mathbf{y}}_i\right\| \leq 1 + h(G_{\mathbf{y}} + L_\Phi\mu_{\mathbf{y}}) + u_{\mathrm{H}},$$

where we have used that

$$\tilde{\mathbf{y}}_{i+1}{}^{-1}\tilde{\mathbf{y}}_i = \tilde{\mathbf{y}}_{i+1}{}^{-1}((\mathbf{1} - \mathbf{r}_H^{i+1})\tilde{\mathbf{y}}_{i+1} - h\tilde{\Phi}(\tilde{\mathbf{y}}_i)) = \mathbf{1} - \mathbf{r}_H^{i+1} - h\tilde{\mathbf{y}}_{i+1}{}^{-1}\tilde{\Phi}(\tilde{\mathbf{y}}_i),$$

and therefore $\|\tilde{\mathbf{y}}_{i+1}{}^{-1}\tilde{\mathbf{y}}_i\| \leq 1 + hG_{\mathbf{y}} + u_{\mathrm{H}}$ upon discarding higher order terms in h.

To obtain a step-independent bound on the second term, we use (4.4) and obtain

$$\left\|\frac{\mathbf{e}_{\Phi(\mathbf{y}_i)}}{\tilde{\mathbf{y}}_{i+1}}\right\| = \left\|\frac{\tilde{\Phi}(\tilde{\mathbf{y}}_i)}{\tilde{\mathbf{y}}_{i+1}}\boldsymbol{\delta}_{\Phi(\mathbf{y}_i)}\right\| \leq \left\|\frac{\tilde{\Phi}(\tilde{\mathbf{y}}_i)}{\tilde{\mathbf{y}}_{i+1}}\right\| \|\boldsymbol{\delta}_{\Phi(\mathbf{y}_i)}\| \leq G_{\mathbf{y}}M_\Phi u_{\mathrm{L}}.$$

This gives the recursive error bound

$$\|\boldsymbol{\delta}_{\mathbf{y}_{i+1}}\| \leq (1 + h\alpha)\|\boldsymbol{\delta}_{\mathbf{y}_i}\| + (h\beta u_{\mathrm{L}} + u_{\mathrm{H}}),$$

with $\alpha = \mu_{\mathbf{y}}L_\Phi + G_{\mathbf{y}}$ and $\beta = M_\Phi G_{\mathbf{y}}$ and applying the linear recursion formula [29, Lemma 7.2.2.2], we get finally

$$\|\boldsymbol{\delta}_{\mathbf{y}_i}\| \leq \exp(\mathbf{t}_i\alpha)\|\boldsymbol{\delta}_{\mathbf{y}_0}\| + \frac{\exp(\mathbf{t}_i\alpha) - 1}{\alpha}\left(\beta u_{\mathrm{L}} + \frac{u_{\mathrm{H}}}{h}\right). \qquad \square$$

LEMMA 4.4 (Adjoint error). *Under Theorem* 4.1*, the relative adjoint error* $\boldsymbol{\delta}_{\mathbf{a}_i} = \frac{\mathbf{a}_i - \tilde{\mathbf{a}}_i}{\tilde{\mathbf{a}}_i}$ *satisfies*

$$\|\boldsymbol{\delta}_{\mathbf{a}_i}\| \leq \exp((T - \mathbf{t}_i)K_4)\|\boldsymbol{\delta}_{\mathbf{a}_N}\| + K_5\|\delta_{\mathbf{y}_0}\| + \left(K_6u_{\mathrm{L}} + K_7\frac{u_{\mathrm{H}}}{h}\right),$$

for constants K_4, K_5, K_6, K_7 *independent on* h.

Proof. The relative error in the terminal adjoint state is equal to the relative error of the low-precision evaluation of ∇C, that is, $\boldsymbol{\delta}_{\mathbf{a}_N} = \boldsymbol{\delta}_{\nabla C}$. For the $i < N$th step, subtracting the low-precision adjoint step, $\tilde{\mathbf{a}}_{i-1}$ from exact counterpart, $\mathbf{a}_{i-1}$, we get the error propagation

$$\mathbf{e}_{\mathbf{a}_{i-1}} = \mathbf{e}_{\mathbf{a}_i} - h\mathbf{e}_{d\mathbf{a}_i} + \mathbf{r}_H^{i-1}\tilde{\mathbf{a}}_{i-1},$$

where the error in the increment is given by

$$\begin{aligned}\mathbf{e}_{d\mathbf{a}_i} &= d\mathbf{a}_i - d\tilde{\mathbf{a}}_i = D\Phi(\mathbf{y}_{i-1})^\top\mathbf{a}_i - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1})^\top\tilde{\mathbf{a}}_i \\ &= D\Phi(\mathbf{y}_{i-1})^\top\mathbf{e}_{\mathbf{a}_i} + (D\Phi(\mathbf{y}_{i-1}) - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1}))^\top\tilde{\mathbf{a}}_i.\end{aligned}$$

This gives the error propagation

$$\mathbf{e}_{\mathbf{a}_{i-1}} = (\mathbf{I} - hD\Phi(\mathbf{y}_{i-1})^\top)\mathbf{e}_{\mathbf{a}_i} - h\left(D\Phi(\mathbf{y}_{i-1}) - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1})\right)^\top \tilde{\mathbf{a}}_i + \mathbf{r}_H^{i-1}\tilde{\mathbf{a}}_{i-1}.$$

By dividing component-wise by $\tilde{\mathbf{a}}_{i-1}$ and simplifying

$$\boldsymbol{\delta}_{\mathbf{a}_{i-1}} = \tilde{\mathbf{a}}_{i-1}{}^{-1}(\mathbf{I} - hD\Phi(\mathbf{y}_{i-1})^\top)\tilde{\mathbf{a}}_i \cdot \boldsymbol{\delta}_{\mathbf{a}_i} - h\tilde{\mathbf{a}}_{i-1}{}^{-1}\left(D\Phi(\mathbf{y}_{i-1}) - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1})\right)^\top \tilde{\mathbf{a}}_i + \mathbf{r}_H^{i-1}.$$

Using the bounds on the Jacobian, we see that

$$\|\tilde{\mathbf{a}}_{i-1}{}^{-1}(\mathbf{I} - hD\Phi(\mathbf{y}_{i-1})^\top)\tilde{\mathbf{a}}_i\| \leq 1 + h(G_{\mathbf{a}} + \mu_{\mathbf{a}}L_\Phi) + u_{\mathrm{H}}$$

where we have used

$$\tilde{\mathbf{a}}_{i-1}{}^{-1}\tilde{\mathbf{a}}_i = \tilde{\mathbf{a}}_{i-1}{}^{-1}((\mathbf{1} - \mathbf{r}_H^{i-1})\tilde{\mathbf{a}}_{i-1} + hd\mathbf{a}_i) = \mathbf{1} - \mathbf{r}_H^{i-1} + h\frac{d\mathbf{a}_i}{\tilde{\mathbf{a}}_{i-1}}, \qquad \|\tilde{\mathbf{a}}_{i-1}{}^{-1}\tilde{\mathbf{a}}_i\| \leq 1 + hG_{\mathbf{a}} + u_{\mathrm{H}}.$$

Taking the norm of both sides and using the triangle inequality, and dropping terms $\mathcal{O}(hu_{\mathrm{H}})$, we obtain

$$\|\boldsymbol{\delta}_{\mathbf{a}_{i-1}}\| \leq (1 + h(\mu_{\mathbf{a}}L_\Phi + G_{\mathbf{a}}))\|\boldsymbol{\delta}_{\mathbf{a}_i}\| + h\mu_{\mathbf{a}}\left\|D\Phi(\mathbf{y}_{i-1})^\top - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1})^\top\right\| + u_{\mathrm{H}}.$$

To obtain a step-independent bound on the second term, we compute

$$\begin{aligned}
\|D\Phi(\mathbf{y}_{i-1})^\top - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1})^\top\| &= \|(D\Phi(\mathbf{y}_{i-1}) - D\Phi(\tilde{\mathbf{y}}_{i-1}))^\top + (D\Phi(\tilde{\mathbf{y}}_{i-1}) - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1}))^\top\| \\
&\leq \|(D\Phi(\tilde{\mathbf{y}}_{i-1}) - D\Phi(\mathbf{y}_{i-1}))^\top\| + \|(D\Phi(\tilde{\mathbf{y}}_{i-1}) - D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1}))^\top\| \\
&\leq L_{D\Phi}\|\boldsymbol{\delta}_{\mathbf{y}_{i-1}}\tilde{\mathbf{y}}_{i-1}\| + u_{\mathrm{L}}M_{D\Phi}\|D\tilde{\Phi}(\tilde{\mathbf{y}}_{i-1})^\top\| \\
&\leq L_{D\Phi}M_{\mathbf{y}}\|\boldsymbol{\delta}_{\mathbf{y}_{i-1}}\| + u_{\mathrm{L}}M_{D\Phi}L_{\tilde{\Phi}},
\end{aligned}$$

where we used the Lipschitz continuity of $D\Phi$, (4.5), the upper bound of $\{\|\mathbf{y}_i\|\}_{i=0}^N$ and the boundedness of $D\tilde{\Phi}$ (4.2). Dropping higher order terms, we obtain the recursive error bound

$$\|\boldsymbol{\delta}_{\mathbf{a}_{i-1}}\| \leq (1 + h\gamma)\|\boldsymbol{\delta}_{\mathbf{a}_i}\| + h\theta\|\delta_{\mathbf{y}_{i-1}}\| + (h\sigma u_{\mathrm{L}} + u_{\mathrm{H}}),$$

with $\gamma = (\mu_{\mathbf{a}}L_\Phi + G_{\mathbf{a}})$, $\theta = \mu_{\mathbf{a}}L_{D\Phi}M_{\mathbf{y}}$, $\sigma = \mu_{\mathbf{a}}M_{D\Phi}L_{\tilde{\Phi}}$. Using (4.7), we can bound the contributions of the forward error as

$$\|\boldsymbol{\delta}_{\mathbf{y}_i}\| \leq \|\boldsymbol{\delta}_{\mathbf{y}_0}\|\exp(\mathbf{t}_N K_1) + \tilde{K}_2 u_{\mathrm{L}} + \tilde{K}_3\frac{u_{\mathrm{H}}}{h}, \quad \forall i,$$

where $\tilde{K}_2 = (\exp(\mathbf{t}_N K_1) - 1)K_2$ and $\tilde{K}_3 = (\exp(\mathbf{t}_N K_1) - 1)K_3$. This allows us to apply [29, Lemma 7.2.2.2] to

$$\|\boldsymbol{\delta}_{\mathbf{a}_{i-1}}\| \leq (1 + h\gamma)\|\boldsymbol{\delta}_{\mathbf{a}_i}\| + h\left(\theta\|\boldsymbol{\delta}_{\mathbf{y}_0}\|\exp(\mathbf{t}_N K_1) + (\theta\tilde{K}_2 + \sigma)u_{\mathrm{L}} + (\theta\tilde{K}_3 + 1)\frac{u_{\mathrm{H}}}{h}\right),$$

which gives the final estimate

$$\begin{aligned}
\|\boldsymbol{\delta}_{\mathbf{a}_i}\| \leq{}& \exp((\mathbf{t}_N - \mathbf{t}_i)\gamma)\|\boldsymbol{\delta}_{\mathbf{a}_N}\| \\
&+ \frac{\exp((\mathbf{t}_N - \mathbf{t}_i)\gamma) - 1}{\gamma}\left(\frac{\theta\|\boldsymbol{\delta}_{\mathbf{y}_0}\|}{\exp(\mathbf{t}_N K_1)} + (\theta\tilde{K}_2 + \sigma)u_{\mathrm{L}} + (\theta\tilde{K}_3 + 1)\frac{u_{\mathrm{H}}}{h}\right). \qquad \square
\end{aligned}$$

LEMMA 4.5 (Gradient error). *Under Theorem* 4.1*, the relative error of* $\mathbf{g} = \frac{dL}{d\boldsymbol{\theta}}$ *satisfies*

$$\|\boldsymbol{\delta}_{\mathbf{g}}\| = \mathcal{O}(u_{\mathrm{L}}) + \mathcal{O}(u_{\mathrm{H}}/h).$$

Proof. The gradient accumulations are

$$\mathbf{g} = h \sum_{i=0}^{N-1} Q_{32}\left(\frac{\partial \Phi}{\partial \boldsymbol{\theta}}(Q_{16}(\mathbf{y}_i))^{\top} Q_{16}(\mathbf{a}_i)\right).$$

The sum contains $N = T/h$ terms, so we must make sure it does not grow unboundedly. Each term in the sum has an error of $h\mathcal{O}(u_{\mathrm{L}})$ from the low-precision computations, and the high-precision summation adds $\mathcal{O}(u_{\mathrm{H}})$ for each of the N terms. Since N and h cancel in the u_{L} term (giving $Nh\mathcal{O}(u_{\mathrm{L}}) = T\mathcal{O}(u_{\mathrm{L}})$), while the high-precision errors accumulate as $N\mathcal{O}(u_{\mathrm{H}}) = \frac{T}{h}\mathcal{O}(u_{\mathrm{H}})$, the result follows. ∎

We are finally ready to prove our main result Theorem 4.2.

Proof of Theorem 4.2. The first item follows from Lemma 4.3. The second item follows from Lemma 4.4, noting that using (4.3) and (4.6) we have

$$\begin{aligned}
\|\boldsymbol{\delta}_{\mathbf{a}_N}\| &\leq \left\| \frac{\nabla C(\mathbf{y}_N) - \nabla \tilde{C}(\tilde{\mathbf{y}}_N)}{\nabla \tilde{C}(\tilde{\mathbf{y}}_N)} \right\| \\
&\leq \left\| \frac{\nabla C(\mathbf{y}_N) - \nabla C(\tilde{\mathbf{y}}_N)}{\nabla \tilde{C}(\tilde{\mathbf{y}}_N)} \right\| + \left\| \frac{\nabla C(\tilde{\mathbf{y}}_N) - \nabla \tilde{C}(\tilde{\mathbf{y}}_N)}{\nabla \tilde{C}(\tilde{\mathbf{y}}_N)} \right\| \\
&\leq \frac{L_{\nabla C}}{m_{\nabla C}} M_{\mathbf{y}} \|\delta_{\mathbf{y}_N}\| + M_{\nabla C} u_{\mathrm{L}},
\end{aligned}$$

and using Lemma 4.3 for bounding $\|\boldsymbol{\delta}_{\mathbf{y}_N}\|$. The third item follows from Lemma 4.5. We note the recurrent theme that the low-precision errors enter through function evaluations multiplied by h and thus accumulate as $\mathcal{O}(u_{\mathrm{L}})$, independent of h while the high-precision errors from the state, adjoint, and gradient updates accumulate as $\mathcal{O}(u_{\mathrm{H}}/h)$. ∎

5. Implementation and Software. Our implementation of the mixed precision forward propagation Algorithm 3.1 and customized backward propagation with dynamic scaling Algorithm 3.2 for neural ODEs in the PyTorch package `rampde`. The package is designed to train neural ODEs using mixed precision as effectively as in single precision while reducing memory usage and time to solution. We use PyTorch's built-in autocast functionality and hardware support to accelerate computations particularly on modern GPUs equipped with tensor compute units.

To allow users to experiment with `rampde` without excessive code changes, our software is designed to closely mirror that of `torchdiffeq` [3], the perhaps most commonly used library for neural ODE training. In most cases, using our package only requires changing an import and adding an autocast context. A minimal example reads as follows:

```
#from torchdiffeq import odeint
from rampde import odeint
...
with autocast(device_type='cuda', dtype=dtype_low):
    y = odeint(func, y0, timesteps)
    loss = L(y)
```

```
loss.backward()
```

A key difference to `torchdiffeq` is that our package is limited to discretize-then-optimize schemes using fixed time steps.

The autocast context in the above example controls the mixed precision computation including casting variables and accelerating whitelisted operations; see the documentation of PyTorch's built-in autocast functionality [24]. Our current implementation is limited to CUDA-enabled GPUs, where we support both float16 and bfloat16 arithmetic. Users can also experiment with bfloat16 on some CPUs. When mixed precision is not supported (e.g., on Apple Silicon/MPS devices), the code automatically falls back to single-precision arithmetic.

The `rampde` package can be installed via pip or conda. The source code is hosted on Github where we also provide detailed instructions, minimal examples, and list all required dependencies with explicit version constraints. We publish the code under the permissive MIT license.

The key component of our package is the mixed precision forward propagation in Algorithm 3.1 and the custom backpropagation with dynamic adjoint scaling in Algorithm 3.2. Our package is designed modularly to allow extending the available forward solvers (currently, forward Euler and RK4) and scaling (currently, no scaling for testing and dynamic adjoint scaling as default).

To achieve optimal performance across different precision formats and scaling requirements, we provide three backward pass implementations that share the same forward propagation. The unscaled variant eliminates overflow checking and scaling infrastructure for minimal overhead and is the default for float32 and bfloat16, where the wider range makes overflow unlikely. The unscaled-safe variant adds exception handling for overflow protection and is the default for float 16 without dynamic scaling. It provides compatibility with PyTorch's standard GradScaler when overflow protection is needed without dynamic scaling overhead. The dynamic variant implements the full scaling algorithm from Algorithm 3.2 with automatic scale adjustment for most robust float16 training. The appropriate solver variant is automatically selected based on the precision format and loss scaler type, removing the burden of manual selection from users while ensuring optimal performance for each configuration.

Although the package is primarily designed to support the development of mixed precision algorithms for neural ODE, our implementation is optimized to be deployed on recent GPUs with tensor compute units and we tested the code on NVIDIA RTX A6000 GPUs. On our hardware, we find that the performance gains can be significant when the problem sizes are adequate for tensor cores and sufficiently large.

To verify the correctness of our code, we include a suite of unit tests. Using a linear ODE system, we test the approximation of solution and gradients. Using a nonlinear ODE, we test the correctness of our manual backward propagation. Using Theorem 3.1, we test the effectiveness of our adjoint scaling. We also implement a regression test using a linear ODE to ensure that future updates to the code maintain some speed-ups for sufficiently large systems on our hardware.

One limitation of our package is that our custom backward pass casts all neural network weights and inputs to the specified low precision rather than using autocast. This implementation was needed because of the limited interoperability of PyTorch's autocast and autograd features. This means that all operations, including blacklisted ones in autocast, are performed in low-precision. To avoid possible discrepancies between forward and backward pass, users need to add manual casting operations around blacklisted operations in their neural network model. We closely monitor release notes of future version of PyTorch and will update our code when the support

for automatic mixed precision is sufficiently improved.

6. Numerical Experiments. In this section, we demonstrate the effectiveness of our mixed precision training framework for neural ODEs across three learning tasks of increasing computational complexity. All experiments were conducted on NVIDIA RTX A6000 GPUs, which provide hardware acceleration for both float16 and bfloat16 arithmetic through tensor compute units. We implemented our algorithms in PyTorch and provide the open-source `rampde` package for reproducibility. We compare our implementation against the widely-used `torchdiffeq` package [3] across different floating-point precisions and experiment with different scaling strategies.

6.1. Continuous Normalizing Flows. Continuous Normalizing Flows (CNFs) are a class of generative models that compute a transformation between a complex target distribution, π_{data}, represented by samples and a simple base distribution, π_1, usually a standard normal through a continuous-time flow [4, 9]. Using the change of variables formula, it can be seen that the log-likelihood of a data point $\mathbf{x} \sim \pi_{\text{data}}$ under the flow is given by

$$\log \pi_{\text{data}}(\mathbf{x}) = \log \pi_1(\mathbf{z}(1)) - \ell(1), \tag{6.1}$$

where $\mathbf{z}(1)$ and $\ell(1)$ are the terminal state of the neural ODE system with joint state $\mathbf{y}(t) = [\mathbf{z}(t), \ell(t)]^\top$,

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}(t), \boldsymbol{\theta}) = \begin{bmatrix} v(t, \mathbf{z}(t), \boldsymbol{\theta}) \\ -\mathrm{tr}\left(\frac{\partial v}{\partial \mathbf{z}}\right) \end{bmatrix}, \quad t \in (0, 1], \quad \mathbf{y}(0) = \begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix}.$$

Here, $\mathbf{z}(t) \in \mathbb{R}^d$ tracks the trajectories of the samples, $\ell(t) \in \mathbb{R}$ computes the volume change, and the velocity field $v : [0, 1] \times \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^d$ is parameterized by a neural network. To train the network weights, we minimize the expected negative log-likelihood over π_{data}. To generate new samples, we reverse the flow with terminal state obtained from the standard normal distribution. Assuming backward stability and sufficiently accurate time integration, the neural ODE enables tractable computations of the inverse of the flow and likelihood estimation. The high accuracy requirements in the ODE solver make CNF an interesting testbed problem for mixed precision training.

We evaluate our mixed precision framework on three standard 2D benchmark datasets (2-spirals [17], 8-Gaussians, and checkerboard). To enable a direct comparison with `torchdiffeq`, we modify the implementation of the two-dimensional toy problems used in [9]. We use the discretize-then-optimize approach and integrate the augmented ODE using RK4 with 128 fixed timesteps. We use the Adam optimizer and train for 2000 iterations with batch size 1024 and learning rate 0.01 without weight decay. We evaluate the performance using validation negative log-likelihood and assess the visual quality of generated samples. Following [9], we parameterize the vector field using a hypernetwork, that is,

$$v(t, \mathbf{z}, \boldsymbol{\theta}) = \frac{1}{w} \sum_{j=1}^{w} \mathbf{U}_j(t) \cdot \tanh(\mathbf{W}_j(t)\mathbf{z} + \mathbf{b}_j(t)),$$

where $\{\mathbf{W}_j(t), \mathbf{b}_j(t), \mathbf{U}_j(t)\}_{j=1}^{w}$ are computed using three fully connected layers with hidden dimension 32, generating $w = 128$ sets of time-dependent weights for the velocity field with weights $\boldsymbol{\theta}$. The most significant modification we made to the code is replacing the stochastic trace estimation with a direct implementation of $\mathrm{tr}(\partial v / \partial \mathbf{z})$,

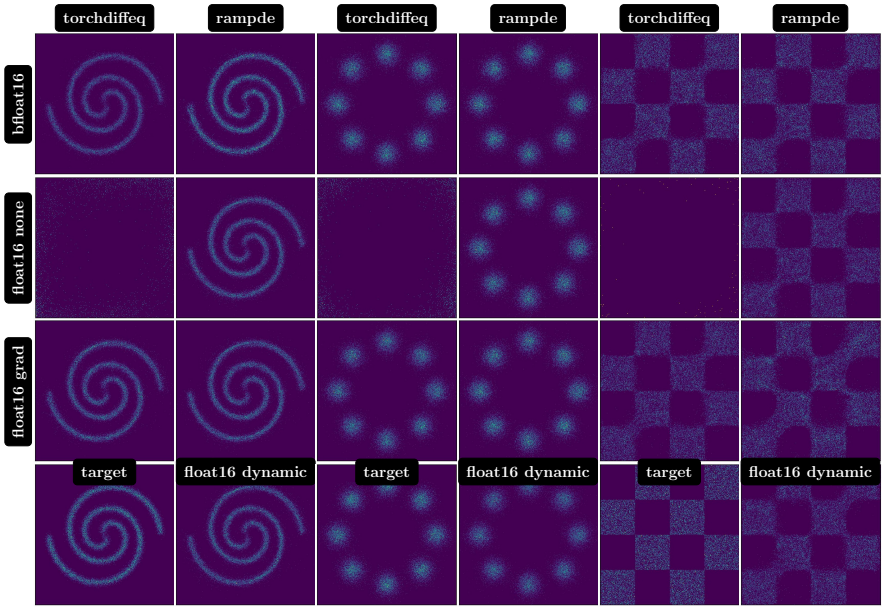

Fig. 2: Comparison of continuous normalizing flow (CNF) sample quality in subsection 6.1 across precision formats and ODE solvers on 2D datasets also used in [9]. The columns contain the datasets: 2-spirals (left pair), 8-Gaussians (center pair), and checkerboard (right pair) datasets. For each dataset, there are two columns, comparing torchdiffeq (left) and rampde (right). Rows demonstrate different precision and scaling configurations. The bottom row shows ground truth targets with float16 dynamic scaling, which is only available in rampde. Across these tests, the rampde solver exhibits visually comparable sample quality even without loss scaling, which is important for float16 training using torchdiffeq for all the datasets in this example.

because the automatic differentiation did not reliably compute derivatives in the precision set by autocast in our experiments.

Figure 2 visualizes the quality of generated samples across different precision formats and scaling strategies. Both `torchdiffeq` and `rampde` produce visually indistinguishable samples to single precision when proper scaling is applied, demonstrating that mixed precision training leads to performance similar to that of single precision. The quantitative results in Table 2 confirm this observation, with validation losses remaining consistent across precisions when scaling is enabled. One indication that scaling is essential can be seen in the `torchdiffeq` results at float16 where the training fails to reduce the validation loss. The losses of all models start at approximately 4 for the 2-spirals dataset but for `torchdiffeq` without gradient scaling, the final loss is around 9.0, compared to 2.7 for the successful experiments. We observe similar results for `torchdiffeq` without scaling on the other datasets. Both gradient scaling and dynamic adjoint scaling effectively stabilize the float16 training.

The two ODE solver implementations differ in their computational cost due to their different memory-compute tradeoffs. While `torchdiffeq` stores all intermediate states and computational graphs during the forward pass, `rampde` saves only the states and recomputes the ODE dynamics during backpropagation, which lowers memory

Table 2: Mixed precision training results for Continuous Normalizing Flows (CNFs) on the 2D datasets 2spirals [17], 8gaussians, and checkerboard. We test torchdiffeq and rampde with network evaluations in float32, bfloat16, and float16. For the latter, we test gradient scaling (Grad), no scaling (None), and dynamic scaling (Dyn). Both solvers achieve comparable results to single precision in bfloat16 and float16 with proper scaling. For float16 without scaling, torchdiffeq performs 2000 iterations without NaN exceptions but the training loss increases. Since this is a small-scale tests, we observe no speedup and only slight memory savings compared to single precision. As expected, rampde uses significantly less memory at the cost of more expensive backpropagation.

	float32		bfloat16		float16				
	torchdiffeq	rampde	torchdiffeq	rampde	torchdiffeq		rampde		
Metric					Grad	None	Grad	None	Dyn
2spirals									
Val Loss	2.671	2.670	2.667	2.674	2.666	9.065	2.679	2.664	2.673
Avg Fwd Time (s)	0.41	0.27	0.47	0.32	0.48	0.49	0.33	0.33	0.33
Avg Bwd Time (s)	0.60	0.88	0.68	1.03	0.70	0.70	1.16	1.15	1.68
Total Time (s)	2026.2	2312.0	2304.1	2694.3	2362.9	2377.7	2980.1	2961.7	4009.8
Max Memory	1.3GB	35.3MB	919.5MB	29.6MB	919.5MB	919.5MB	29.7MB	29.7MB	29.5MB
8gaussians									
Val Loss	2.805	2.882	2.888	2.859	2.831	9.448	2.840	2.828	2.879
Avg Fwd Time (s)	0.42	0.28	0.48	0.33	0.49	0.49	0.32	0.33	0.33
Avg Bwd Time (s)	0.61	0.89	0.69	1.04	0.72	0.71	1.15	1.16	1.71
Total Time (s)	2047.5	2329.2	2341.1	2747.4	2407.5	2389.6	2949.1	2985.9	4082.5
Max Memory	1.3GB	35.3MB	919.5MB	29.6MB	919.5MB	919.5MB	29.7MB	29.7MB	29.5MB
checkerboard									
Val Loss	3.565	3.592	3.565	3.560	3.549	12.366	3.604	3.566	3.573
Avg Fwd Time (s)	0.41	0.27	0.48	0.33	0.48	0.49	0.33	0.33	0.33
Avg Bwd Time (s)	0.61	0.89	0.69	1.05	0.71	0.71	1.15	1.17	1.70
Total Time (s)	2038.6	2328.9	2336.2	2755.6	2384.2	2399.6	2958.0	3005.6	4068.0
Max Memory	1.3GB	35.3MB	919.5MB	29.6MB	919.5MB	919.5MB	29.7MB	29.7MB	29.5MB

usage at the cost of additional computation. This results in a large reduction of memory usage (1.3GB for `torchdiffeq` to 35MB for `rampde` in float32) while increasing the backward pass time by approximately 50% for `rampde`. For these small-scale 2D problems, reducing the precision did not yield wall-clock speedups. This is probably due to the small problem size, which leads to poor utilization of the hardware's tensor cores and the overhead of precision conversions.

Next, we illustrate the behavior of roundoff errors in both approaches to empirically support our theoretical analysis in section 4. To this end, we use a network trained using `rampde` in float32 using the 2 spirals dataset and evaluate the relative errors of state and gradients to double precision as the number of time steps increases. In Figure 3, we show the relative errors for float16 with proper scaling, that is, gradient scaling for `torchdiffeq` and dynamic adjoint scaling for `rampde`. It can be seen that both approaches accurately approximate the terminal state and the initial adjoint, which is the derivative for the initial value. For `torchdiffeq` we notice a slight positive trend in the errors of the weight gradients in this example. As expected from our theory, the error in the weight gradients is comparable for the different step sizes in `rampde`.

6.2. Optimal Transport Flows. We compare the impact of mixed precision training on a CNF example with optimal transport regularization on the BSDS300 dataset [18] following the experimental setup in our previous work [21]. The BSDS300

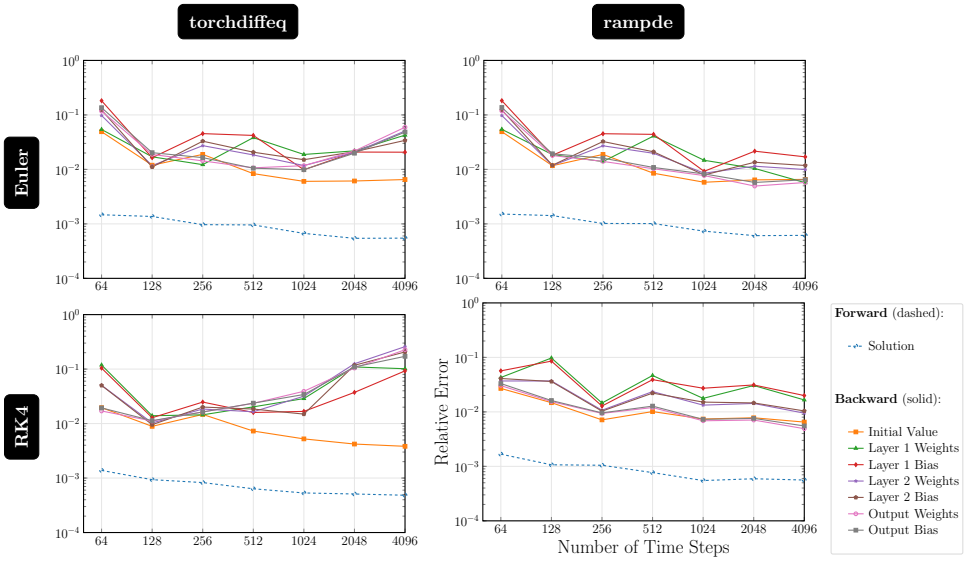

Fig. 3: Roundoff error experiment for the 2D Continuous Normalizing Flow training using the 2spirals dataset in subsection 6.1. We compare the accuracy of forward (dashed) and backward propagation (solid) in float 16 for torchdiffeq (left) and rampde (right) for forward Euler (top) and RK4 (bottom) as the number of time steps increases. This is done by the relative error of each quantity specified in the legend of the figure to the float64 computation. The accuracy of the solution and gradients for the initial value is consistent for torchdiffeq, while the gradients of the neural network weights become slightly less accurate as N grows. Consistent with our analysis in section 4, the relative errors for rampde remain stable.

dataset is a 63-dimensional feature representation derived from natural image patches in the original Berkeley segmentation dataset. The dataset size makes it an attractive target for mixed precision training because of the large memory and computational costs and because, when adding the time to the ODE state as done in OT-Flow, the network is evaluated with 64-dimensional inputs, which increases tensor core utilization.

Unlike standard CNFs, OT Flows adds transport costs to regularize the training objective and uses the underlying optimality condition to express the optimal velocity as the negative gradient of the value function. The training objective reads

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim\pi_{\text{data}}}\left[\alpha_1\left(\frac{1}{2}\|\mathbf{z}(1)\|^2 - \ell(1)\right) + c_{\text{L}}(1) + \alpha_2 c_{\text{HJB}}(1)\right] \tag{6.2}$$

where $\mathbf{z}(1), \ell(1), c_{\text{L}}(1), c_{\text{HJB}}(1)$ track the trajectory of the sample, the volume change, the transport costs, and the violation of the Hamilton-Jacobi-Bellman (HJB) equation, respectively. They are the terminal states of the augmented neural ODE with

states $\mathbf{y}(t) = [\mathbf{z}(t), \ell(t), c_L(t), c_{\mathrm{HJB}}(t)]^\top$ and dynamics

$$\frac{d\mathbf{y}}{dt} = f_{\mathrm{OT}}(t, \mathbf{y}(t), \boldsymbol{\theta}) = \begin{bmatrix} -\nabla V(t, \mathbf{y}(t), \boldsymbol{\theta}) \\ \Delta V(t, \mathbf{y}(t), \boldsymbol{\theta}) \\ \frac{1}{2}\|\nabla V(t, \mathbf{y}(t), \boldsymbol{\theta})\|^2 \\ \left|\frac{\partial V}{\partial t}(t, \mathbf{y}(t), \boldsymbol{\theta}) + \frac{1}{2}\|\nabla V(t, \mathbf{y}(t), \boldsymbol{\theta})\|^2\right| \end{bmatrix}, \quad \mathbf{y}(0) = \begin{bmatrix} \mathbf{x} \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The OT-Flow approach parameterizes the value function $V : [0,1] \times \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}$ and computes its gradient and Laplacian analytically. For this experiment, we use a wider version of the architecture as in [21], which consists of an opening layer with 64 inputs (concatenating $\mathbf{z}(t)$ and t) and 1024 outputs, followed by one residual layer and one affine layer that maps the 1024 outputs to a scaler. The activation function is the antiderivative of the tanh function, which we compute in single precision. As in [21], we use $\alpha_1 = 800, \alpha_2 = 2000$. Sample quality is evaluated using the Maximum Mean Discrepancy (MMD) metric in addition to the individual loss components. The ODE is integrated using RK4 with 16 equidistant time steps during both training and 30 during evaluation. We train for up to 10,000 iterations with batch size 512, employing the same stopping criteria as in the original OT-Flow experiments to enable early stopping when the validation loss stops improving.

Table 3 presents the results on the BSDS300 dataset using a set-up similar to that of the CNF experiment. Our baseline in this experiment is TensorFloat-32 (tfloat32), which is the default float32 mode on modern NVIDIA GPUs (Ampere and newer). Unlike pure float32, tfloat32 internally uses mixed precision within tensor cores. Specifically, it rounds float32 inputs to 10-bit mantissas while maintaining the 8-bit exponent range, effectively performing computations in a reduced precision before accumulating in float32. This hardware-level mixed precision provides speed benefits while maintaining the programming interface of float32. In this experiment, float16 training fails to converge without gradient scaling due to NaN in the gradients in the beginning of training. This indicates that the limited range of float16 requires careful scaling or hyperparameter tuning to prevent gradient underflow. With appropriate scaling strategies, all precision configurations achieve comparable validation metrics: negative log-likelihoods range from -159 to -167, transport costs remain bounded between 2.8 and 3.3, and HJB penalties stay close to 1.0, indicating that the mixed precision schemes perform comparably to single precision. The Maximum Mean Discrepancy (MMD) values, which measure sample quality, are comparable ($\in [0.02, 0.05]$) in all successful configurations, indicating a comparable match between the data and the generated distributions across the precisions.

Compared to the small-scale CNF experiment, the memory savings of the mixed precision training for each scheme are more pronounced (almost 50% savings for `torchdiffeq` and 25% savings for `rampde`). As before, `rampde` significantly reduces the memory footprint in each configuration. We observe a modest speedup in the float16 and bfloat16 configurations compared to tfloat32, especially in the average time of the forward passes. As expected, `rampde`'s backward pass takes longer than in `torchdiffeq` due to the recomputation of the dynamic. The overall training time varies slightly due to the early stopping.

This experiment demonstrates that mixed precision training can be attractive for high-dimensional optimal transport problems, with the choice of precision and solver depending on the specific computational constraints. For memory-limited scenarios typical in large-scale generative modeling, `rampde` with bfloat16 offers an excellent balance, providing 10× memory reduction with only modest speed penalties. When

Table 3: Mixed precision training results for optimal transport flows (OT-Flow) on the 63-dimensional BSDS300 feature dataset. We test torchdiffeq and rampde across tfloat32, bfloat16, and float16 precisions and for the latter use different scaling strategies (Grad=gradient scaling, None=no scaling, Dyn=dynamic adjoint scaling). Both solvers fail to converge in float16 without gradient scaling (indicated by "—"). With proper scaling, all configutations achieve similar validation metrics but vary in computational costs. By avoiding re-computations in the backward pass, torchdiffeq generally is faster but requires more memory. In contrast, rampde achieves significant memory savings at some additional costs during the backward pass. Both schemes see slight speedups in 16-bit precisions compared to tfloat32.

	tfloat32		bfloat16		float16				
	torchdiffeq	rampde	torchdiffeq	rampde	torchdiffeq		rampde		
Metric					Grad	None	Grad	None	Dyn
Val Loss	-128767.7	-124379.1	-129675.6	-128714.9	-131398.6	—	-131309.3	—	-129021.0
Val L	3.001	3.292	2.986	2.871	3.041	—	3.059	—	3.114
Val NLL	-163.7	-159.0	-164.8	-163.4	-167.0	—	-166.9	—	-164.0
Val HJB	1.079	1.416	1.061	1.000	1.093	—	1.098	—	1.086
Val MMD	0.0332	0.0237	0.0285	0.0469	0.0285	—	0.0328	—	0.0276
Avg Fwd Time (s)	0.246	0.195	0.211	0.163	0.212	—	0.161	—	0.162
Avg Bwd Time (s)	0.374	0.613	0.310	0.515	0.310	—	0.537	—	0.780
Avg Time per Step (s)	0.620	0.808	0.521	0.677	0.521	—	0.698	—	0.942
Total Time (s)	6016.0	5898.5	5160.7	6569.3	5005.2	—	6701.3	—	9140.8
Total Iterations	9,700	7,300	9,900	9,700	9,600	—	9,600	—	9,700
Max Memory (GB)	18.9	2.0	11.1	1.5	11.1	—	1.5	—	1.5

training speed is paramount and memory is available, `torchdiffeq` with bfloat16 delivers the fastest convergence.

6.3. STL-10 Image Classification. To evaluate the potential of our mixed precision scheme to reduce memory consumption and improve training time on a large-scale learning problem, we apply it to train a Neural ODE-based convolutional neural network to classify the STL-10 dataset. Rather than improving the state-of-the-art accuracy, we focus on the computational aspects of this problem and the comparison of our algorithm to `torchdiffeq` in different precisions.

The STL-10 dataset contains 12,000 natural images that are divided into 10 classes and is derived from the larger ImageNet dataset. The dataset is divided into 5,000 training images and 7,000 test images. Each image has three color channels for RGB and has 96×96 pixels.

In order to generalize effectively from only 500 labeled training images per class, we use data augmentation during training. We upscale the images to 128×128 to use the tensor cores efficiently, and we use a random resize crop with a uniform scale sampled from $[0.5, 1.0]$ and aspect ratio from $[0.75, 1.33]$. We then apply a random horizontal flip (probability 0.5) and apply a color jitter to manipulate the brightness, contrast, saturation, and hue with respective maximum adjustments of 0.4, 0.4, 0.4, and 0.1. To encourage the model to focus more on shapes, texture, and edges rather than relying on color information, we convert the image to grayscale with 0.1 probability. Finally, we normalize the image to zero mean and unit variance using the per-channel mean and standard deviation computed from the training images. For the validation and test images, we only perform center cropping to 128×128, tensor conversion, and normalization.

We use a version of the parabolic CNN architecture proposed in [26] with about 12.6M trainable weights. This architecture provides a particularly challenging test

case for mixed precision training because its stable dynamics with antisymmetric structure lead to varying scales throughout the network. Additionally, unlike more widely used architectures, hyperparameter choice is less well understood and it is critical to avoid precision-specific hyperparameter optimization. The architecture consists of several blocks, three of which are defined as neural ODEs. The first block is an opening layer that maps the RGB images to 128 channels with a convolution 3×3 followed by instance normalization and ReLU activation. The output of the opening layer is used as the initial state of the first neural ODE block governed by the dynamics

$$\frac{d}{dt}\mathbf{y}(t) = -\mathbf{W}(t)^\top \sigma\left(\mathcal{N}(\mathbf{W}(t)\mathbf{y}(t) + \mathbf{b}(t))\right), \quad t \in (0, 1]. \tag{6.3}$$

Here, $\mathbf{b}$ is a bias, $\mathbf{W}$ is a convolution operator with 128 input and output channels and 3×3 stencils, $\mathcal{N}$ denotes the instance normalization, and σ is the ReLU activation. We discretize the neural ODE with four equidistant RK4 steps and model the weights and biases as piecewise constants on each time interval. The terminal state of the neural ODE block is then fed into a connecting block that consists of a 3×3 convolution that doubles the number of channels, instance normalization, ReLU activation, and an average pooling layer to halve the number of pixels in each direction. This is followed by another neural ODE block of the form (6.3) but with twice the number of channels, followed by another connector, and a final neural ODE block with 512 channels and images of size 32×32. Finally, the feature channels are global averaged to reduce sensitivity to translations before the final fully connected layer. The loss function is a softmax cross-entropy loss.

We investigate whether evaluating the forward pass and backward pass in 16-bit precisions using PyTorch's autocast context can lead to similar performance to single precision training at reduced computational cost. For `rampde`, this means that the three neural ODE blocks are evaluated with our custom mixed precision handling, while the opening and connecting blocks use the default whitelisted and blacklisted of operators. We use the same strategy to evaluate the model when testing our rampde package and the baseline, torchdiffeq.

We compare both implementations with tfloat32, bfloat16, and float16 floating-point precisions and for the latter we experiment with gradient scaling, dynamic scaling (rampde only) and no scaling. Each configuration is trained with SGD (momentum 0.9, weight decay 5×10^{-4}) for 160 epochs, batch size 16, initial learning rate 0.05, and a cosine annealing schedule. All experiments use a fixed random seed of 25 for data splitting and model initialization to ensure reproducibility. We calculate the loss and accuracy on the training and validation sets and plot the training loss versus wall-clock time in Figure 4. After the training is completed, we compute the accuracy and loss on the test set. We summarize all key performance metrics and additional statistics (training time, peak memory) in Table 4.

Although we used the same random seed for the data split and initialization, we observed that training is not fully deterministic, which is likely due to non-deterministic low-level hardware behavior.

At float16, torchdiffeq fails early on in the training as the loss is beyond the representable range (at iteration 9 without gradient scaling and iteration 31 with scaling). This demonstrates the need for careful combination of the precisions or more flexible dynamic scaling schemes when training neural ODEs in float16. In contrast, our rampde implementation successfully trains in float16 even without scaling, though dynamic scaling provides the best results.

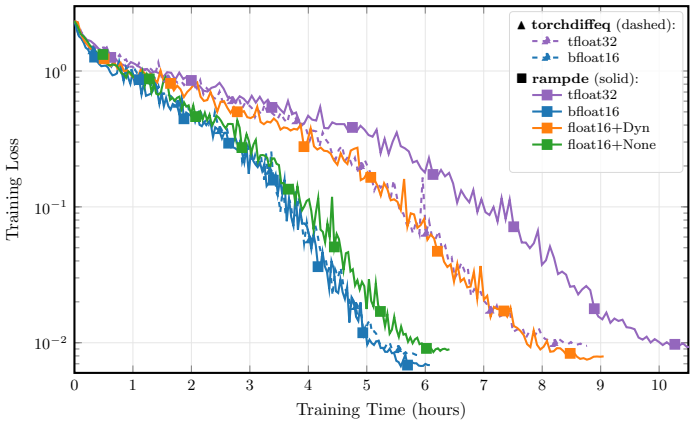

Fig. 4: Training loss convergence for the STL-10 experiment in subsection 6.3 comparing torchdiffeq (triangular markers, dashed lines) and rampde (square markers, solid lines) in different precisions. The plot shows training loss (log scale) versus wall-clock time in hours, demonstrating that mixed precision variants can achieve comparable final loss values more quickly. For clarity, we omit rampde in float16 with gradient scaling as its curve is similar to that without scaling. We also omit torchdiffeq at float16 as it failed with NaN after a few training steps even with gradient scaling activated.

Table 4: STL10 image classification performance for torchdiffeq and rampde. Our rampde implementation achieves similar accuracy in 16-bit than in single precision while reducing the memory footprint and training time by almost a factor of two. With float16 precision it works well with gradient scaling (Grad) but achieves slightly better test accuracy with dynamic scaling (Dyn) albeit at some additional cost during backward. Torchdiffeq fails in float16 without loss scaling due to a NaN.

	float32		tfloat32		bfloat16		float16			
	torchdiffeq	rampde	torchdiffeq	rampde	torchdiffeq	rampde	torchdiffeq	rampde		
Metric							Grad	Dyn	Grad	None
STL10										
Val Acc	0.762	0.797	0.742	0.780	0.735	0.805	0.110	0.783	0.784	0.775
Val Loss	0.878	0.752	0.934	0.765	0.914	0.667	2.326	0.790	0.759	0.824
Test Acc	0.770	0.800	0.759	0.789	0.741	0.810	0.110	0.792	0.788	0.775
Test Loss	0.872	0.748	0.908	0.768	0.905	0.663	2.326	0.755	0.765	0.836
Avg Fwd Time (s)	0.29	0.28	0.28	0.27	0.19	0.15	1.16	0.15	0.15	0.15
Avg Bwd Time (s)	0.55	0.77	0.51	0.73	0.34	0.40	1.09	0.67	0.43	0.43
Max Memory	21.5GB	4.5GB	21.5GB	4.3GB	6.9GB	2.2GB	6.9GB	2.4GB	2.3GB	2.3GB

This experiment indicates that MPT can achieve memory and runtime reduction for large-scale problems while maintaining competitive accuracy. Comparing the memory usage across implementations, rampde reduces peak memory from 21.5GB to 4.3GB in single precision, approximately a 5× reduction. For runtime performance, the fastest configuration overall is torchdiffeq with bfloat16 (0.19s forward, 0.34s backward per iteration). The fastest rampde configuration (bfloat16: 0.15s forward, 0.40s backward) achieves comparable forward pass speed and only 43% overhead in the backward pass, while using 3× less memory (2.2GB vs 6.9GB). Within

rampde, mixed precision variants provide approximately 1.8× speedup compared to tfloat32 (0.27s vs 0.15s forward pass). Most importantly, all configurations maintain test accuracy above 77% when proper scaling is used, validating that neural ODEs can be effectively trained in mixed precision without sacrificing model performance or requiring precision-specific hyperparameter tuning.

7. Discussion. We developed and analyzed mixed precision training (MPT) techniques for Neural ODEs that take advantage of low-precision support in modern hardware and improve scalability to larger problem sizes. Neural ODEs present unique challenges for existing MPT frameworks due to the potential accumulation of roundoff errors over many time steps. Following the intuition from roundoff error analysis and existing MPT guidelines, we accumulate in high precision while computing expensive neural network evaluations in low precision. We carefully analyze roundoff errors, provide an efficient implementation, and thoroughly validate our approach. Our results demonstrate that mixed precision neural ODEs can achieve accuracy comparable to single-precision training while offering significant memory reduction and, for adequately sized problems, also 3reduced training times.

Our MPT approach has three main pillars. First, we designed explicit ODE solvers that maintain states and accumulate in high precision (float32) while evaluating the neural network in low precision (float16 or bfloat16) and implemented a custom backward propagation scheme that uses a similar principle for differentiation. Second, we developed a dynamic adjoint scaling scheme that adapts layer-wise scaling ideas to the continuous-time setting and automatically maintains the range during backpropagation. Third, our roundoff error analysis proves that this intuitive approach is theoretically sound in that the relative errors remain $\mathcal{O}(u_{\mathrm{L}})$ and do not accumulate uncontrollably with the number of time steps for reasonable step sizes. The CNF experiments empirically confirm that roundoff errors of our scheme behave as predicted by our theoretical analysis while that of our benchmark method, torchdiffeq, shows slight increase of errors for smaller step sizes.

Our experiments demonstrate the potential of mixed precision training for neural ODEs. In the STL-10 image classification task, we achieved almost a 2× speedup and 2× memory reduction using float16 and bfloat16 precisions compared to tfloat32, while maintaining over 76% test accuracy across all properly scaled configurations. The OT-Flow experiments on 63-dimensional data showed modest speedups with float16 and bfloat16 compared to tfloat32, validating the approach for moderately high-dimensional generative modeling tasks.

Based on our theoretical and empirical findings, we offer the following recommendations for using mixed precision neural ODEs. We emphasize that MPT is most beneficial for problems with moderate accuracy requirements where data is inherently noisy and that are sufficiently large such that memory and the computational time of tensor-core friendly operations are limiting factors. In such situations, `rampde` with bfloat16 provides a good accuracy-memory tradeoff and may be sufficient. If more accuracy is required, `rampde` with gradient or dynamic scaling is worth considering and in our experiments did not require additional hyperparameter tuning that is otherwise common in float16. Finally, MPT can be attractive even if training times are higher when deployment requires low-precision inference, as MPT directly optimizes quantized neural ODEs and can avoid post-training quantization errors.

Our approach has several limitations that suggest directions for future research. Performance gains are hardware-dependent, with best results on GPUs with tensor core support. Our current implementation focuses on explicit solvers, which are stan-

dard in deep learning applications where the nonlinearity of neural networks makes implicit methods less attractive. Several research directions appear promising. Extending to 8-bit quantization would provide extreme compression, though our theoretical analysis suggests this may be challenging with $\mathcal{O}(u_{\mathrm{L}}) \approx 2^{-3}$ or $\mathcal{O}(u_{\mathrm{L}}) \approx 2^{-4}$ depending on the format. Adapting the framework to neural stochastic differential equations (neural SDEs) would broaden applicability to score-based generative models and stochastic control problems. Applying these techniques to optimal control problems and PDE-constrained optimization, where neural ODEs show increasing promise, could enable new applications.

Acknowledgements. We thank Deepanshu Verma for his help in investigating different mixed precision training frameworks and many useful discussions. In accordance with SIAM's editorial policy on AI usage, we describe our use of AI tools in as much detail as possible: We used generative search and ChatGPT's DeepResearch to assist in discovering related literature in the machine learning domain. We used Claude Code with Sonnet 4 and Opus 4 as a coding assistant to: (1) refactor and optimize our prototype implementations to a publishable state enabling reproducibility; (2) expand unit test coverage; (3) help document the code; (4) develop reproducible scripts for generating all figures and tables presented in this paper. These scripts are available in our repository. Additionally, we used LLMs to polish the manuscript text for grammar, clarity, and consistency of mathematical notation. We take full responsibility for the accuracy and integrity of all content in this paper.

REFERENCES

[1] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, *Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores*, SIAM Journal on Scientific Computing, 42 (2020), pp. C124–C141, https://doi.org/10.1137/19M1289546, https://epubs.siam.org/doi/10.1137/19M1289546 (accessed 2025-05-28).

[2] B. Burnett, S. Gottlieb, Z. J. Grant, and A. Heryudono, *Performance evaluation of mixed-precision Runge–Kutta methods*, arXiv, (2021), https://doi.org/10.48550/arxiv.2107.03357, https://arxiv.org/abs/2107.03357.

[3] R. T. Q. Chen, *torchdiffeq*, 2018, https://github.com/rtqichen/torchdiffeq.

[4] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural Ordinary Differential Equations*, Dec. 2019, https://doi.org/10.48550/arXiv.1806.07366, http://arxiv.org/abs/1806.07366 (accessed 2025-05-28). arXiv:1806.07366 [cs].

[5] W. E, *A Proposal on Machine Learning via Dynamical Systems*, Communications in Mathematics and Statistics, 5 (2017), pp. 1–11, https://doi.org/10.1007/s40304-017-0103-z, http://link.springer.com/10.1007/s40304-017-0103-z (accessed 2025-05-28).

[6] 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 of Proceedings of Machine Learning Research, PMLR, 11 2020, pp. 3154 – 3164.

[7] A. Gholami, K. Keutzer, and G. Biros, *ANODE: Unconditionally Accurate Memory-Efficient Gradients for Neural ODEs*, arXiv.org, cs.LG (2019), arXiv.org, https://arxiv.org/abs/1902.10298.

[8] Z. J. Grant, *Perturbed Runge–Kutta Methods for Mixed Precision Applications*, Journal of Scientific Computing, 92 (2022), p. 6, https://doi.org/10.1007/s10915-022-01801-2, https://link.springer.com/10.1007/s10915-022-01801-2 (accessed 2025-05-28).

[9] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, *FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models*, in International Conference on Learning Representations, 10 2019.

[10] E. Haber and L. Ruthotto, *Stable architectures for deep neural networks*, Inverse Problems, 34 (2018), p. 014004, https://doi.org/10.1088/1361-6420/aa9a90, http://arxiv.org/abs/1705.03341 (accessed 2025-05-28). arXiv:1705.03341 [cs].

[11] J. Hayford, J. Goldman-Wetzler, E. Wang, and L. Lu, *Speeding up and reducing memory usage for scientific machine learning via mixed precision*, Jan. 2024, https:

//doi.org/10.48550/arXiv.2401.16645, http://arxiv.org/abs/2401.16645 (accessed 2025-05-28). arXiv:2401.16645 [cs].

[12] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, PA, 2nd ed., 2002.

[13] N. J. Higham and T. Mary, *Mixed precision algorithms in numerical linear algebra*, Acta Numerica, 31 (2022), pp. 347–414, https://doi.org/10.1017/S0962492922000022, https://www.cambridge.org/core/product/identifier/S0962492922000022/type/journal_article (accessed 2025-05-28).

[14] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu, *Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes*, arXiv, (2018), https://doi.org/10.48550/arxiv.1807.11205, https://arxiv.org/abs/1807.11205.

[15] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, *A Study of BFLOAT16 for Deep Learning Training*, June 2019, https://doi.org/10.48550/arXiv.1905.12322, http://arxiv.org/abs/1905.12322 (accessed 2025-05-28). arXiv:1905.12322 [cs].

[16] P. Kidger, *On Neural Differential Equations*, Feb. 2022, https://doi.org/10.48550/arXiv.2202.02435, http://arxiv.org/abs/2202.02435 (accessed 2025-05-28). arXiv:2202.02435 [cs].

[17] K. J. Lang and M. J. Witbrock, *Learning to tell two spirals apart*, in Proceedings of the 1988 Connectionist Models Summer School, San Mateo, CA, 1989, Morgan Kaufmann, pp. 52–59.

[18] D. Martin, C. Fowlkes, D. Tal, and J. Malik, *A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics*, in Proceedings Eighth IEEE International Conference on Computer Vision (ICCV 2001), vol. 2, Vancouver, Canada, 2001, IEEE, pp. 416–423.

[19] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama, *Dissecting Neural ODEs*, Jan. 2021, https://doi.org/10.48550/arXiv.2002.08071, http://arxiv.org/abs/2002.08071 (accessed 2025-05-28). arXiv:2002.08071 [cs].

[20] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, *Mixed Precision Training*, Feb. 2018, https://doi.org/10.48550/arXiv.1710.03740, http://arxiv.org/abs/1710.03740 (accessed 2025-05-28). arXiv:1710.03740 [cs].

[21] D. Onken, S. W. Fung, X. Li, and L. Ruthotto, *OT-Flow: Fast and Accurate Continuous Normalizing Flows via Optimal Transport*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, AAAI Press, 2021, pp. 9223–9232, arXiv.org.

[22] D. Onken, L. Nurbekyan, X. Li, S. W. Fung, S. Osher, and L. Ruthotto, *A Neural Network Approach for High-Dimensional Optimal Control*, arXiv, (2021), https://arxiv.org/abs/2104.03270.

[23] D. Onken and L. Ruthotto, *Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and Continuous Normalizing Flows*, arXiv.org, cs.LG (2020), arXiv.org.

[24] PyTorch Team, *Automatic Mixed Precision package - torch.amp*, 2024, https://pytorch.org/docs/stable/amp.html. Accessed August 2025.

[25] R. Rico-Martínez, K. Krischer, and I. G. Kevrekidis, *Discrete- vs. Continuous-time Nonlinear Signal Processing of Cu Electrodissolution Data*, Chemical Engineering Communications, 118 (1992), pp. 25 – 48, https://doi.org/10.1080/00986449208936084.

[26] L. Ruthotto and E. Haber, *Deep Neural Networks Motivated by Partial Differential Equations*, Journal of Mathematical Imaging and Vision, (2020).

[27] L. Ruthotto, S. J. Osher, W. Li, L. Nurbekyan, and S. W. Fung, *A machine learning framework for solving high-dimensional mean field game and mean field control problems*, Proceedings of the National Academy of Sciences, 117 (2020), pp. 9183 – 9193, https://doi.org/10.1073/pnas.1922204117/-/dcsupplemental. Supported by NSF DMS 1751636.

[28] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, *Score-Based Generative Modeling through Stochastic Differential Equations*, arXiv, (2020), https://arxiv.org/abs/2011.13456.

[29] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Texts in Applied Mathematics, Springer, New York, NY, 3 ed., 2002, https://doi.org/10.1007/978-0-387-21738-3, https://doi.org/10.1007/978-0-387-21738-3.

[30] R. Zhao, B. Vogel, and T. Ahmed, *Adaptive Loss Scaling for Mixed Precision Training*, Oct. 2019, https://doi.org/10.48550/arXiv.1910.12385, http://arxiv.org/abs/1910.12385 (accessed 2025-05-28). arXiv:1910.12385 [cs].