"Hey, that's not an ODE": Faster ODE Adjoints with 12 Lines of Code

Patrick Kidger
University of Oxford;
The Alan Turing Institute
kidger@maths.ox.ac.uk

Ricky T. Q. Chen
University of Toronto;
Vector Institute
rtqichen@cs.toronto.edu

Terry Lyons
University of Oxford;
The Alan Turing Institute
tlyons@maths.ox.ac.uk

Abstract

Neural differential equations may be trained by backpropagating gradients via the adjoint method, which is another differential equation typically solved using an adaptive-step-size numerical differential equation solver. A proposed step is accepted if its error, *relative to some norm*, is sufficiently small; else it is rejected, the step is shrunk, and the process is repeated. Here, we demonstrate that the particular structure of the adjoint equations makes the usual choices of norm (such as L^2) unnecessarily stringent. By replacing it with a more appropriate (semi)norm, fewer steps are unnecessarily rejected and the backpropagation is made faster. This requires only minor code modifications. Experiments on a wide range of tasks—including time series, generative modeling, and physical control—demonstrate a median improvement of 40% fewer function evaluations.

1 Introduction

The general approach of neural ordinary differential equations (E, 2017; Chen et al., 2018) is to use ODEs as a learnable component of a differentiable framework. Typically the goal is to approximate a map $x \mapsto y$ by learning functions $\ell_1(\,\cdot\,,\phi)$, $\ell_2(\,\cdot\,,\psi)$ and $f(\,\cdot\,,\,\cdot\,,\theta)$, which are composed such that

$$z(\tau) = \ell_1(x,\phi), \qquad z(t) = z(\tau) + \int_\tau^t f(s,z(s),\theta)\,\mathrm{d}s \quad \text{and} \quad y \approx \ell_2(z(T),\psi). \tag{1}$$

The variables ϕ, θ, ψ denote learnable parameters and the ODE is solved over the interval $[\tau, T]$. We include the (often linear) maps $\ell_1(\,\cdot\,,\phi)$, $\ell_2(\,\cdot\,,\psi)$ for generality, as in many contexts they are important for the expressiveness of the model (Dupont et al., 2019; Zhang et al., 2020), though our contributions will be focused around the ODE component and will not depend on these maps.

1.1 Adjoint equations

The integral in equation (1) may be backpropagated through either by backpropagating through the internal operations of a numerical solver, or by solving the backwards-in-time *adjoint equations* with respect to some (scalar) loss L.

$$\begin{aligned}
a_z(T) &= \frac{\mathrm{d}L}{\mathrm{d}z(T)}, & a_z(t) &= a_z(T) - \int_T^t a_z(s)\cdot\frac{\partial f}{\partial z}(s,z(s),\theta)\,\mathrm{d}s & \text{and} \quad \frac{\mathrm{d}L}{\mathrm{d}z(\tau)} &= a_z(\tau),\\
a_\theta(T) &= 0, & a_\theta(t) &= a_\theta(T) - \int_T^t a_z(s)\cdot\frac{\partial f}{\partial \theta}(s,z(s),\theta)\,\mathrm{d}s & \text{and} \quad \frac{\mathrm{d}L}{\mathrm{d}\theta} &= a_\theta(\tau),
\end{aligned} \tag{2}$$

As the integrands require $z(s)$, the adjoint equations are typically augmented with equation (1) run backwards in time. Then the joint system $a(t) = [a_z(t), a_\theta(t), z(t)]$ is solved backwards-in-time.

Third Workshop on Machine Learning and the Physical Sciences (NeurIPS 2020), Vancouver, Canada.

2 Method

2.1 Numerical solvers

The forward and backward passes of equations (1) and (2) are both solved with a numerical differential equation solver. Our interest here is in **adaptive-step-size solvers**. Indeed the default choice for solving many equations is the adaptive-step-size Runge–Kutta 5(4) scheme of Dormand and Prince (1980), for example as implemented by `dopri5` in the `torchdiffeq` package or `ode45` in MATLAB.

The part of interest to us is the accept/reject scheme. Suppose for some fixed t the solver has computed some estimate $\widehat{z}(t) \approx z(t)$, and it now seeks to take a step $\Delta > 0$ to compute $\widehat{z}(t+\Delta) \approx z(t+\Delta)$. A step is made, and some candidate $\widehat{z}_{\text{candidate}}(t+\Delta)$ is generated. The solver additionally produces $z_{\text{err}} \in \mathbb{R}^d$ representing an estimate of the numerical error made in each channel during that step.

Given some prespecified absolute tolerance $ATOL$ (for example 10^{-9}), relative tolerance $RTOL$ (for example 10^{-6}), and (semi)norm $\|\cdot\| : \mathbb{R}^d \to [0, \infty)$ (for example $\|z\| = \sqrt{\frac{1}{d}\sum_{i=1}^{d} z_i^2}$ the RMS norm), then an estimate of the size of the equation is given by

$$SCALE = ATOL + RTOL \cdot \max(\widehat{z}(t), \widehat{z}_{\text{candidate}}(t+\Delta)) \in \mathbb{R}^d, \tag{3}$$

where the maximum is taken *channel-wise*, and the error ratio

$$r = \left\| \frac{z_{\text{err}}}{SCALE} \right\| \in \mathbb{R} \tag{4}$$

is then computed. If $r \leq 1$ then the error is deemed acceptable, the step is accepted and we take $\widehat{z}(t+\Delta) = \widehat{z}_{\text{candidate}}(t+\Delta)$. If $r > 1$ then the error is deemed too large, the candidate $\widehat{z}_{\text{candidate}}(t+\Delta)$ is rejected, and the procedure is repeated with a smaller Δ.

2.2 Adjoint seminorms

Not an ODE The key observation is that a_θ does not appear anywhere in the vector fields of equation (2). This means that (conditioned on knowing z and a_z), the integral corresponding to a_θ is just an integral—not an ODE. As such, it is arguably inappropriate to solve it with an ODE solver, which makes the implicit assumption that small errors now may propagate to create large errors later.

Accept/reject This is made manifest in the accept/reject step of equation (4). Typical choices of norm $\|\cdot\|$, such as L^2, will usually weight each channel equally. But we have just established that to solve the adjoint equations accurately, it is far more important that z and a_z be accurate than it is that a_θ be accurate.

Seminorms Thus, when solving the adjoint equations equation (2), we propose to use a $\|\cdot\|$ that scales down the effect in those channels corresponding to a_θ. In practice, in our experiments, we scale $\|\cdot\|$ all the way down by applying zero weight to the offending channels, so that $\|\cdot\|$ is in fact a seminorm. This means that the integration steps are chosen solely for the computation of a_z and z, and the values of a_θ are computed just by integrating with respect to those steps.

Code This is a simple change requiring few lines of code. The additional 12 lines are marked.

```
import torchdiffeq

def rms_norm(tensor):                                                    #
    return tensor.pow(2).mean().sqrt()                                   #
                                                                         #
def make_norm(state):                                                    #
    state_size = state.numel()                                           #
    def norm(aug_state):                                                 #
        y = aug_state[1:1 + state_size]                                  #
        adj_y = aug_state[1 + state_size:1 + 2 * state_size]             #
        return max(rms_norm(y), rms_norm(adj_y))                         #
    return norm                                                          #
                                                                         #
torchdiffeq.odeint_adjoint(func=..., y0=..., t=...,
                           adjoint_options=dict(norm=make_norm(y0)))     #
```

Table 1: Results for Neural CDEs. Mean ± standard deviation over five repeats.

	Default norm		Seminorm	
RTOL, ATOL	Accuracy (%)	Bwd. NFE (10^6)	Accuracy (%)	Bwd. NFE (10^6)
$10^{-3}, 10^{-6}$	92.6±0.4	14.36±1.09	92.5±0.5	**8.67±1.60**
$10^{-4}, 10^{-7}$	92.8±0.4	30.67±2.48	92.5±0.5	**12.75±2.00**
$10^{-5}, 10^{-8}$	92.4±0.7	77.95±4.47	92.9±0.4	**29.39±0.80**

Does this reduce the accuracy of parameter gradients? One obvious concern is that we are typically ultimately interested in the parameter gradients a_θ, in order to train a model; with respect to this our approach seems counter-intuitive.

However, we verify empirically that models still train without a reduction in performance. We explain this by noting that the z, a_z channels truly are ODEs, so that small errors now do propagate to create larger errors later. Thus these are likely the dominant source of error overall.

3 Experiments

We compare our proposed technique against conventionally-trained neural differential equations, across a range of tasks drawn from the main applications of neural differential equations. In every case, the differential equation solver used is the Dormand–Prince 5(4) solver "`dopri5`". The default norm is a mixed L^∞/L^2 norm used in `torchdiffeq`. The code for these experiments can be found at `https://github.com/patrick-kidger/FasterNeuralDiffEq/`.

Neural Controlled Differential Equations Consider the Neural Controlled Differential Equation (Neural CDE) model of Kidger et al. (2020). To recap, given some (potentially irregularly sampled) time series $\mathbf{x} = ((t_0, x_0), \ldots, (t_n, x_n))$, with each $t_i \in \mathbb{R}$ the timestamp of the observation $x_i \in \mathbb{R}^v$, let $X \colon [t_0, t_n] \to \mathbb{R}^{1+v}$ be an interpolation such that $X(t_i) = (t_i, x_i)$. For example X could be a natural cubic spline. Then take $f(t, z, \theta) = g(z, \theta)\frac{\mathrm{d}X}{\mathrm{d}t}(t)$ in a Neural ODE model, so that changes in $\mathbf{x}$ provoke changes in the vector field, and the model incorporates the incoming information $\mathbf{x}$. This may be thought of as a continuous-time RNN.

We apply a Neural CDE to the Speech Commands dataset (Warden, 2020). This is a dataset of one-second audio recordings of spoken words such as 'left', 'right' and so on. We take 34975 time series corresponding to 10 words, to produce a balanced classification problem. The initial map ℓ_1 (of equation (1)) is taken to be linear on (t_0, x_0). The terminal map ℓ_2 is taken to be linear on $z(t_n)$. We investigate how the effect changes for varying tolerances by varying the pair $(RTOL, ATOL)$ over $(10^{-3}, 10^{-6})$, $(10^{-4}, 10^{-7})$, and $(10^{-5}, 10^{-8})$. For each such pair we run five repeated experiments.

See Table 1 for results on number of function evaluations. The accuracy of the model is unaffected by our proposed change, whilst the backward pass uses 40%–62% fewer steps, depending on tolerance. Figure 1 shows how accuracy and function evaluations change during training. We see that accuracy quickly gets close to its maximum value during training, with only incremental improvements for most of the training procedure. Additionally, we see that the number of function evaluations is much lower for the seminorm throughout training.

Continuous Normalising Flows Continuous Normalising Flows (CNF) (Chen et al., 2018) are a class of generative models that define a probability distribution as the transformation of a simpler distribution by following the vector field parameterized by a Neural ODE. Let $p(z_0)$ be an arbitrary base distribution that we can efficiently sample from, and compute its density. Then let $z(t)$ be the solution of the initial value problem

$$z(0) \sim p(z_0), \qquad \frac{dz(t)}{zt} = f(t, z(t), \theta), \qquad \frac{d \log p(z(t))}{dt} = -\mathrm{tr}\left(\frac{\partial f}{\partial z}(t, z(t), \theta)\right),$$

for which the change in log probability density is also tracked, as the sample is transformed through the vector field (Chen et al., 2018).

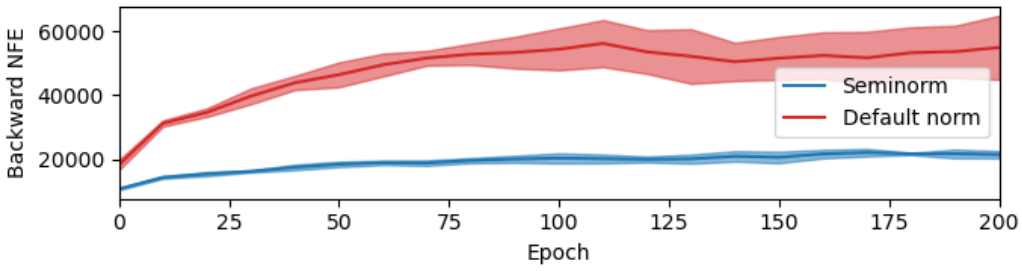

Figure 1: Mean $\pm$ standard deviation of the number of function evaluations (NFE) for the backward pass of Neural CDEs, over the course of training.

Table 2: Results for Continuous Normalising Flows. Mean $\pm$ standard deviation over three repeats.

		Default norm		Seminorm	
		Test bits/dim	Bwd. NFE (10^6)	Test bits/dim	Bwd. NFE (10^6)
CIFAR-10	($d_h = 64$)	3.3492$\pm$0.0059	41.65$\pm$1.97	3.3388$\pm$0.0082	**38.93$\pm$1.32**
MNIST	($d_h = 32$)	0.9968$\pm$0.0020	41.50$\pm$2.35	0.9942$\pm$0.0013	**37.03$\pm$0.96**
MNIST	($d_h = 64$)	0.9637$\pm$0.0008	44.24$\pm$1.78	0.9601$\pm$0.0025	**37.85$\pm$0.94**
MNIST	($d_h = 128$)	0.9504$\pm$0.0036	48.60$\pm$2.30	0.9501$\pm$0.0031	**41.84$\pm$1.92**

In Table 2 we show the final test performance and the total number of function evaluations (NFEs) used in the adjoint method over 100 epochs. We see substantially fewer NFEs in experiments on both MNIST and CIFAR-10. Next, we investigate changing model size, by varying the complexity of the vector field f, which is a CNN with d_h hidden channels. We find that using the seminorm, the backward NFE does not increase as much as when using the default norm.

Hamiltonian dynamics in reinforcement learning and control Finally we consider the problem of learning Hamiltonian dynamics, using the Symplectic ODE-Net model of Zhong et al. (2020). This involves training a neural network-parameterized Hamiltonian system within a model-based control setting. We consider the fully-actuated double pendulum ("acrobot") problem. Training data involves small oscillations under constant forcing.

We find that the model successfully learns the dynamics. Across five repeats, the baseline (default norm) model achieves a test L^2 loss of $(1.247 \pm 0.520) \times 10^{-4}$ whilst the proposed (seminorm) model achieves a a test L^2 loss of $(2.995 \pm 2.190) \times 10^{-4}$.

However, the default norm required $(4.645 \pm 0.001) \times 10^5$ function evaluations to train (for the adjoint equation, accumulated over all epochs), whilst the seminorm required only $(2.655 \pm 0.001) \times 10^5$ function evaluations. Our proposed change thus uses 43% fewer function evaluations on the backward pass.

Finally, we verify that the end goal of controlling the system is achievable. The double pendulum is controlled from the full-down to the full-upright position, using the seminorm-trained model. See Figure 2.

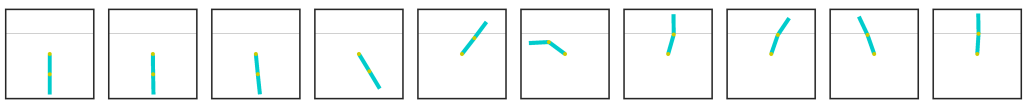

Figure 2: Frames from controlling the fully-actuated double pendulum to the full-upright position.

4 Conclusion

We introduce a method for speeding up training neural differential equations. The method is simple to implement and offers substantial speed-ups with no observed downsides.

Broader Impact

The use of differential equations a modelling paradigm within science is centuries old. As such we expect the primary consequence of this work to be a positive, with applications to modelling in the sciences. No specific negative ethical consequences are anticipated.

Acknowledgments

PK was supported by the EPSRC grant EP/L015811/1. PK and TL were supported by the Alan Turing Institute under the EPSRC grant EP/N510129/1.

References

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural Ordinary Differential Equations. In *Advances in Neural Information Processing Systems 31*, pages 6571–6583. Curran Associates, Inc., 2018.

J. R. Dormand and P. J. Prince. A family of embedded Runge–Kutta formulae. *J. Comp. Appl. Math*, 6:19–26, 1980.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes. In *Advances in Neural Information Processing Systems 32*, pages 3140–3150. Curran Associates, Inc., 2019.

Weinan E. A Proposal on Machine Learning via Dynamical Systems. *Commun. Math. Stat.*, 5(1): 1–11, 2017.

Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural Controlled Differential Equations for Irregular Time Series. *arXiv:2005.08926*, 2020.

Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv:1804.03209*, 2020.

Han Zhang, Xi Gao, Jacob Unterman, and Tom Arodz. Approximation capabilities of neural odes and invertible residual networks. *International Conference on Machine Learning*, 2020.

Yaofeng Desmond Zhong, Biswadip Dey, and Amit Chakraborty. Symplectic ode-net: Learning hamiltonian dynamics with control. In *International Conference on Learning Representations*, 2020.