NeuroDiffEq: A Python package for solving differential equations with neural networks

Feiyu Chen[1], **David Sondak**[1], **Pavlos Protopapas**[1], and **Marios Mattheakis**[1]

1 Institute for Applied Computational Science, Harvard University, Cambridge, MA, United States

DOI: 10.21105/joss.01931

Software

- Review
- Repository
- Archive

Editor: Vincent Knight

Reviewers:

- @dawbarton
- @marcharper

Submitted: 01 November 2019
Published: 03 December 2019

Summary

Differential equations emerge in various scientific and engineering domains for modeling physical phenomena. Most differential equations of practical interest are analytically intractable. Traditionally, differential equations are solved by numerical methods. Sophisticated algorithms exist to integrate differential equations in time and space. Time integration techniques continue to be an active area of research and include backward difference formulas and Runge-Kutta methods (Conde, Gottlieb, Grant, & Shadid, 2017). Common spatial discretization approaches include the finite difference method (FDM), finite volume method (FVM), and finite element method (FEM) as well as spectral methods such as the Fourier-spectral method. These classical methods have been studied in detail and much is known about their convergence properties. Moreover, highly optimized codes exist for solving differential equations of practical interest with these techniques (Seefeldt et al., 2017)(Smith & Abeysinghe, 2017). While these methods are efficient and well-studied, their expressibility is limited by their function representation.

Artificial neural networks (ANN) are a framework of machine learning algorithms that use a collection of connected units to learn function mappings. The most basic form of ANNs, multilayer perceptrons, have been proven to be universal function approximators (Hornik, Stinchcombe, & White, 1989). This suggests the possibility of using ANNs to solve differential equations. Previous studies have demonstrated that ANNs have the potential to solve ordinary differential equations (ODEs) and partial differential equations (PDEs) with certain initial/boundary conditions (Lagaris, Likas, & Fotiadis, 1998). These methods show nice properties including (1) continuous and differentiable solutions, (2) good interpolation properties, (3) less memory-intensive. By less memory-intensive we mean that only the weights of the neural network have to be stored. The solution can then be recovered anywhere in the solution domain because a trained neural network is a closed form solution. Given the interest in developing neural networks for solving differential equations, it would be extremely beneficial to have an easy-to-use software package that allows researchers to quickly set up and solve problems.

`NeuroDiffEq` is a Python package built with `PyTorch` (Paszke et al., 2017) that uses ANNs to solve ordinary and partial differential equations (ODEs and PDEs). During the release of `NeuroDiffEq` we discovered that two other groups had almost simultaneously released their own software packages for solving differential equations with neural networks: `DeepXDE` (Lu, Meng, Mao, & Karniadakis, 2019) and `PyDEns` (Koryagin, Khudorozkov, & Tsimfer, 2019). Both `DeepXDE` and `PyDEns` are built on top of `TensorFlow` (Abadi et al., 2015). `DeepXDE` has an emphasis on the wide variety of problems it can solve. It supports mixing different boundary conditions and solving on domains with complex geometries. `PyDEns` is less flexible in the range of solvable problems but provides a more user-friendly API. This trade-off is partially determined by the way these two packages implement the solver, which will be discussed later.

`NeuroDiffEq` is designed to encourage the user to focus more on the problem domain (What is the differential equation we need to solve? What are the initial/boundary conditions?) and at the same time allow them to dig into solution domain (What ANN architecture and loss function should be used? What are the training hyperparameters?) when they want to. `NeuroDiffEq` can solve a variety of canonical PDEs including the heat equation and Poisson equation in a Cartesian domain with up to two spatial dimensions. We are actively working on extending `NeuroDiffEq` to support three spatial dimensions. `NeuroDiffEq` can also solve arbitrary systems of nonlinear ordinary differential equations. Currently, `NeuroDiffEq` is being used in a variety of research projects including to study the convergence properties of ANNs for solving differential equations as well as solving the equations in the field of general relativity (Schwarzchild and Kerr black holes).

Methods

The key idea of solving differential equations with ANNs is to reformulate the problem as an optimization problem in which we minimize the residual of the differential equations. In a very general sense, a differential equation can be expressed as

$$\mathcal{L}u - f = 0$$

where $\mathcal{L}$ is the differential operator, $u\left(x,t\right)$ is the solution that we wish to find, and f is a known forcing function. We denote the output of the neural network as $u_N\left(x,t;p\right)$ where the parameter vector p is a vector containing the weights and biases of the neural network. We will drop the arguments of the neural network solution in order to be concise. If u_N is a solution to the differential equation, then the residual

$$\mathcal{R}\left(u_N\right) = \mathcal{L}u_N - f$$

will be identically zero. One way to incorporate this into the training process of a neural network is to use the residual as the loss function. In general, the L^2 loss of the residual is used. This is the convention that `NeuroDiffEq` follows, although we note that other loss functions could be conceived. Solving the differential equation is re-cast as the following optimization problem:

$$\min_{p}\left(\mathcal{L}u_N - f\right)^2.$$

It is necessary to inform the neural network about any boundary and initial conditions since it has no way of enforcing these *a priori*. There are two primary ways to satisfy the boundary and initial conditions. First, one can impose the initial/boundary conditions in the loss function. For example, given an initial condition $u\left(x,t_0\right) = u_0\left(x\right)$, the loss function can be modified to:

$$\min_{p}\left[\left(\mathcal{L}u_N - f\right)^2 + \lambda\left(u_N\left(x,t_0\right) - u_0\left(x\right)\right)^2\right]$$

where the second term penalizes solutions that don't satisfy the initial condition. Larger values of the regularization parameter λ result in stricter satisfaction of the initial condition while sacrificing solution accuracy. However, this approach does not lead to *exact* satisfaction of the initial and boundary conditions.

Another option is to transform the u_N in a way such that the initial/boundary conditions are satisfied by construction. Given an initial condition $u_0\left(x\right)$ the neural network can be transformed according to:

$$\widetilde{u}\left(x,t\right) = u_0\left(x\right) + \left(1 - e^{-\left(t-t_0\right)}\right)u_N\left(x,t\right)$$

so that when $t = t_0$, $\widetilde{u}$ will always be u_0. Accordingly, the objective function becomes

$$\min_{p}\left(\mathcal{L}\widetilde{u} - f\right)^2.$$

This approach is similar to the trial function approach (Lagaris et al., 1998), but with a different form of the trial function. Modifying the neural network to account for boundary conditions can also be done. In general, the transformed solution will have the form:

$$\widetilde{u}\left(x,t\right) = A\left(x,t;x_{\text{boundary}},t_0\right) u_N\left(x,t\right)$$

where $A\left(x,t;x_{\text{boundary}},t_0\right)$ must be designed so that $\widetilde{u}\left(x,t\right)$ has the correct boundary conditions. This can be very challenging for complicated domains.

Both of these two methods have their advantages. The first way is simpler to implement and can be more easily extended to high-dimensional PDEs and PDEs formulated on complicated domains. The second way assures that the initial/boundary conditions are exactly satisfied. Considering that differential equations can be sensitive to initial/boundary conditions, this is expected to play an important role. Another advantage of the second method is that fixing these conditions can reduce the effort required during the training of the ANN (McFall & Mahan, 2009). `DeepXDE` uses the first way to impose initial/boundary conditions. `PyDEns` uses a variation of the second approach to impose initial/boundary conditions. `NeuroDiffEq`, the software described herein, employs the second approach.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Retrieved from http://tensorflow.org/

Conde, S., Gottlieb, S., Grant, Z. J., & Shadid, J. N. (2017). Implicit and implicit–explicit strong stability preserving runge–kutta methods with high linear order. *Journal of Scientific Computing*, *73*(2-3), 667–690. doi:10.1063/1.4992752

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, *2*, 359–366. doi:10.1016/0893-6080(89)90020-8

Koryagin, A., Khudorozkov, R., & Tsimfer, S. (2019). PyDEns: A python framework for solving differential equations with neural networks. *arXiv preprint arXiv:1909.11544*.

Lagaris, I. E., Likas, A., & Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, *9*(5), 987–1000. doi:10.1109/72.712178

Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2019). DeepXDE: A deep learning library for solving differential equations. *arXiv preprint arXiv:1907.04502*.

McFall, K. S., & Mahan, J. R. (2009). Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions. *IEEE Transactions on Neural Networks*, *20*(8), 1221–1233. doi:10.1109/TNN.2009.2020735

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., et al. (2017). Automatic differentiation in PyTorch. In *NIPS autodiff workshop*.

Seefeldt, B., Sondak, D., Hensinger, D. M., Phipps, E. T., Foucar, J. G., Pawlowski, R. P., Cyr, E. C., et al. (2017). *Drekar v. 2.0*. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Smith, C. W., & Abeysinghe, E. (2017). The phasta science gateway: Web-based execution of adaptive computational fluid dynamics simulations. In *Proceedings of the practice and experience in advanced research computing 2017 on sustainability, success and impact* (p. 70). ACM. doi:10.1145/3093338.3104151