Neural Ordinary Differential Equations
A Re-Implementation

Manas Tanneeru [1]

Abstract

In this paper, we are exploring the re-implementation of Neural Ordinary Differential Equations (Chen et al., 2019). This paper proposed Neural ODEs as an alternative to Residual Networks for supervised learning. Since neural networks are made up of stacked layers, we can re-frame a neural network as an Ordinary Differential Equation of some undefined depth. We can then use any existing ODE solvers to approximate. This showcases that ODE networks approximates a function better than a Recurrent Network, and this is shown in the paper.

1. Introduction

In this paper, we delve into various methods of training neural networks by focusing on the re-implementation of Neural Ordinary Differential Equations (Chen et al., 2019). The original paper presented a novel approach to constructing neural network architectures using Ordinary Differential Equations (ODEs), enabling more flexible and efficient processing of time-series data. This paper aims to outline the re-implementation process and reproduce the key findings and results derived from the original paper.

The primary motivation for investigating ODEs as an alternative to traditional residual networks stems from the limitations associated with conventional techniques. Traditional methods necessitate a fixed number of layers and a predetermined computational budget for processing each input. These constraints can result in bottlenecks when handling large and complex datasets that demand a high degree of flexibility in the model architecture.

To circumvent these bottlenecks, network architectures based on Ordinary Differential Equations can be employed, offering continuous depth and more efficient training for time-series data. Rather than explicitly defining the behavior of each layer, we can establish a continuous-time dynamic on hidden states governed by an ODE. Various existing ODE solvers can then be used to solve this ODE, generating a flow of hidden states employed for making predictions. By conceptualizing the layers of a neural network as an ODE, we can learn the optimal depth and computational budget for each input during the training process.

This paper will demonstrate the outcomes of the conducted experiments and provide a detailed explanation of the Neural Ordinary Differential Equations implementation (Chen et al., 2019), emphasizing its advantages as a superior method of training.

The code for this project is available on GitHub at https://github.com/mtanneer/Neural-ODEs.

By exploring and comparing various methods of training neural networks, this paper contributes to the ongoing pursuit of more effective, flexible, and efficient models capable of tackling increasingly complex and large-scale datasets, ultimately advancing the field of machine learning and its practical applications.

2. Related work

Presently, most neural networks are trained as Residual Networks. In such networks, the neural network constructs complex transformations by composing a series of transformations applied to a hidden layer.

$$x_{k+1} = x_k + f(x_k) \tag{1}$$

An ordinary differential equation (ODE) establishes a relationship between an independent variable, a dependent variable, and one or more differential coefficients of the dependent variable with respect to the independent variable (Ince, 3). When revisiting ODE theory, one technique for solving an initial value problem (IVP) is employing a solving method like *Euler's Method*.

$$y_{n+1} = y_n + f(t_n, y_n) \tag{2}$$

This method iteratively zeroes in on the solution of an ODE, based on a provided initial value. As observed, Equation (1) and Equation (2) are quite similar, leading to the conclusion that ResNets are essentially ODE solutions.

[1]School of Electrical and Computer Engineering, Purdue University.

This fundamental concept of Neural ODEs posits that a chain of residual blocks in a neural network is essentially a solution of the ODE using the *Euler's Method*.

However, a neural network is a differentiable function, allowing for training using gradient-based optimization techniques. In the current implementation, we treat the ODE Solver as a black-box. Consequently, we employ a specific method called the *Adjoint Sensitivity Method*, which enables us to compute gradients of a scalar-valued loss with respect to all inputs of any ODE solver without backpropagating through the solver's steps. The *Adjoint Sensitivity Method* is elaborated upon in detail in Section 3.1.

By exploring the connection between ResNets and ODEs, we can gain a deeper understanding of the underlying mathematics behind these neural networks, potentially leading to innovative approaches in the development of more efficient and effective models for various applications.

3. Method

3.1. Mathematical Basis

Adjoint Sensitivity Method

The *adjoint sensitivity method* is an efficient technique employed to compute the sensitivity of a function with respect to its parameters (Allaire). By solving a second augmented ODE backwards in time, this method calculates gradients. Additionally, it is applicable to any ODE solver.

In essence, the adjoint system describes the derivative states at each point of the process in reverse, along with the original dynamical system that characterizes the process. To elucidate this concept mathematically, we first need to establish the relevant notations and variables.

Imagine a neural network with continuous depth, which is governed by an Ordinary Differential Equation (ODE) of the form:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta), \tag{3}$$

where $\mathbf{h}(t)$ represents the hidden states at time t, f is a function that maps the hidden states and time to their derivatives, and θ denotes the parameters of the function f.

The primary goal is to determine the gradient of the loss function L with respect to the parameters θ. The loss function can be expressed as:

$$L = g(\mathbf{h}(T), T, \theta), \tag{4}$$

where g is a function that computes the loss at the final time T.

The Adjoint Sensitivity method introduces the concept of the adjoint state $\mathbf{a}(t)$, which is defined as the gradient of the loss function with respect to the hidden states, i.e., $\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$. By applying the chain rule to compute the gradient of the loss function concerning the parameters θ, we derive the following expression:

$$\frac{\partial L}{\partial \theta} = \int_0^T \mathbf{a}(t)^T \frac{\partial f(\mathbf{h}(t), t, \theta)}{\partial \theta} dt. \tag{5}$$

To calculate the adjoint state $\mathbf{a}(t)$, we must determine its dynamics. By differentiating the adjoint state with respect to time and applying the chain rule, we obtain the subsequent ODE:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{h}(t), t, \theta)}{\partial \mathbf{h}(t)} \tag{6}$$

This ODE can be solved backward in time, starting from the final time T with the initial condition $\mathbf{a}(T) = \frac{\partial L}{\partial \mathbf{h}(T)}$.

Through the adjoint sensitivity method, we can efficiently compute the gradients required for training the neural network without explicitly backpropagating through the ODE solver's steps. This method provides an elegant and computationally efficient approach to training continuous-depth neural networks while preserving the advantages offered by the ODE formulation.

Adjoint Sensitivity Method in Neural ODEs

In Neural Ordinary Differential Equations (Chen et al., 2019), the authors propose using a continuous time representation of a neural network. This continuous time representation is formulated as an ODE, which is then solved using an ODE solver. The parameters of the ODE solver are optimized using the *adjoint sensitivity method*.

More specifically, let $f_\theta(x)$ be the continuous-time dynamics of the Neural ODE, where θ denotes the parameters of the ODE. Then, the ODE solver computes the solution x_T at time T given the initial value x_0. The goal is to optimize the parameters θ of the ODE solver such that the output of the Neural ODE at time T, denoted by $h_\theta(x_0)$, best matches the target output y. This optimization problem can be written as follows:

$$\min_\theta \quad \mathcal{L}(h_\theta(x_0), y) \tag{7}$$

where $\mathcal{L}$ is a loss function that measures the discrepancy between the output of the Neural ODE and the target output.

To optimize the parameters θ, the authors use backpropagation through the ODE solver. The gradients of the loss function with respect to the parameters of the ODE solver can be computed using the adjoint sensitivity method. Specifically,

the authors compute the adjoint state a_t, which represents the gradient of the loss function with respect to the state x_t at time t:

$$a_t = -\frac{\partial \mathcal{L}}{\partial x_t} \tag{8}$$

The adjoint state is then used to compute the gradient of the loss function with respect to the parameters θ:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_0^T \frac{\partial f_\theta(x_t)}{\partial \theta} a_t dt \tag{9}$$

This gradient can be efficiently computed using the *adjoint sensitivity method*, which involves solving the adjoint ODE backwards in time. Once the gradients are computed, they can be used to update the parameters of the ODE solver using standard optimization techniques. As a brief summary of this process, Figure 2 from (Chen et al., 2019, Figure 2) showcases how the adjoint method solves the ODE backwards. This is attached to this paper as Figure 1.

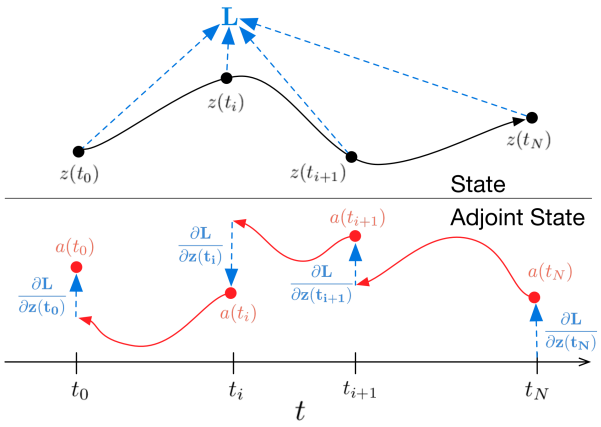

Figure 1. Reverse-Mode differentiation of an ODE.(Chen et al., 2019, Figure 2)

Overall, the *adjoint sensitivity method* is a key component of the optimization process in Neural Ordinary Differential Equations, allowing for efficient computation of gradients and optimization of the ODE solver parameters.

3.2. Implementation

In this paper, a re-implementation of Neural Ordinary Differential Equations (Chen et al., 2019) has been carried out using `Python3` and trained for at least 50 epochs. This specific implementation introduces an innovative neural network architecture known as *ODENet*, which is founded upon ordinary differential equations. The performance of this novel architecture is compared against the conventional *ResNET* architecture, employing the well-established MNIST dataset as a benchmark for comparison.

The machine learning framework employed in this re-implementation is *Pytorch*, which is built upon the "Torch" library. This powerful framework facilitates efficient handling and training with the MNIST dataset. Both the *ODENet* and *ResNet* models are constructed using PyTorch, ensuring seamless integration between the components. Central to the ODENet model are the ODEBlock and ODEfunc classes, which encapsulate the continuous dynamics of the ordinary differential equations within the network layers. The key aspect of this paper's implementation lies in leveraging *torchdiffeq*, which offers an ODE solver capable of utilizing augmented dynamics in the adjoint sensitivity method to compute the *forward* pass of the ODEBlock.

Furthermore, the implementation consists of numerous utility functions that aid in data preprocessing, learning rate scheduling, and model evaluation. For example, the `mnist_loaders` function plays a crucial role in loading the MNIST dataset, while the `lr_decay` function adjusts the learning rate according to a predetermined schedule, thus encouraging improved convergence during the training process. The `accuracy` function measures the model's performance on a given dataset. Moreover, the `train_and_evaluate` function not only trains both the ODENet and ResNet models but also assesses their performance on the MNIST dataset. Stochastic Gradient Descent (SGD) with momentum is employed as the optimization algorithm during the training process, further enhancing the models' performance.

By elaborating on the various components and functions involved in this re-implementation, it is evident that the ODENet architecture offers a promising alternative to traditional neural network architectures. The use of ordinary differential equations in the construction of neural networks paves the way for more flexible and efficient model designs, which can potentially lead to superior performance in various machine learning tasks.

3.3. Technical Difficulties

One of the hardest parts of this re-implementation was Pytorch. Just as revolutionary as it is, for a beginner like me, it gave me a lot of trouble. But considering how popular it is, it was easy to get questions answered and find quicker and more efficient ways to do things.

Another technical difficulty I faced was understanding `data_loader`. But I learned that it is a very useful tool (pyt)

4. Experiment

For testing, the MNIST dataset of handwritten images was used. (Paszke et al., 2019). Below, in Figure 2, is a subset of all the symbols present in the dataset that is loaded.

Figure 2. Subset of MNIST dataset values

As depicted in Figure 2, the images within the dataset are the primary reason for employing Torch in this implementation. The `train_and_evaluate` function serves as the crux of the testing and experimentation process. The MNIST dataset was chosen due to its diverse set of values, and because the authors in (Chen et al., 2019) also utilize it. The objective was to replicate and verify their results. As a result, the implementation is heavily inspired by Chen's implementation of the `torchdiffeq` module (Chen, 2021).

The core of the training process occurs when the `train_and_evaluate` function is invoked with the inputs for both *ODENet* and *ResNet*. In each of these calls, the model, either *ODENet* or *ResNet*, is trained with a specific split of MNIST data. The training data is divided into pre-train and train-eval data, enabling the model to be trained with the pre-train data and have its accuracy verified with the train-eval data. The model is ultimately tested against the testing data, which has been withheld until this point. The model then performs classification, and its accuracy is determined by evaluating the testing accuracy. The term "model" here refers exclusively to the network being trained. The key distinction between the two is the architecture of these networks, which differs for *ODENet* and *ResNet*. The network architecture is responsible for defining the capabilities of these models. Following this, the results of both models are compared. The output from executing the script is displayed below.

```
ODENET----------------------------------
Epoch 0000 | Train Acc 0.0992 | Test Acc 0.1032
Epoch 0001 | Train Acc 0.9732 | Test Acc 0.9777
. . .
Epoch 0048 | Train Acc 0.9986 | Test Acc 0.9946
Epoch 0049 | Train Acc 0.9990 | Test Acc 0.9958

RESNET----------------------------------
Epoch 0000 | Train Acc 0.0807 | Test Acc 0.0820
Epoch 0001 | Train Acc 0.9723 | Test Acc 0.9749
. . .
Epoch 0048 | Train Acc 0.9993 | Test Acc 0.9966
Epoch 0049 | Train Acc 0.9990 | Test Acc 0.9952
```

However, this output is challenging to visualize. Consequently, the training and testing accuracies for each epoch are computed and plotted. The legends in each of the figures, Figure 3 and Figure 4, indicate which of the lines correspond to *ODENet* and *ResNet*.

Figure 3 showcases the results of training and verifying the accuracy of both networks against the MNIST dataset. The call to `train_and_evaluate` before the training loop ensures that both models receive the same loaders, granting them access to the same data.

In a similar vein, Figure 4 illustrates the testing accuracy for each epoch. The testing accuracy serves as a more precise metric for assessing the model's accuracy. As this data is not available during training, it enables the calculation of an accurate metric.

By examining the model metrics, it is evident that the maximum testing accuracy for both *ODENet* and *ResNet* are quite close. However, it is clear that, albeit by a small margin, the *ODENet* is a superior network architecture.

A key factor to be noted is that both these plot are plotted after getting rid of the outliers. This is generally the output from the first epoch.

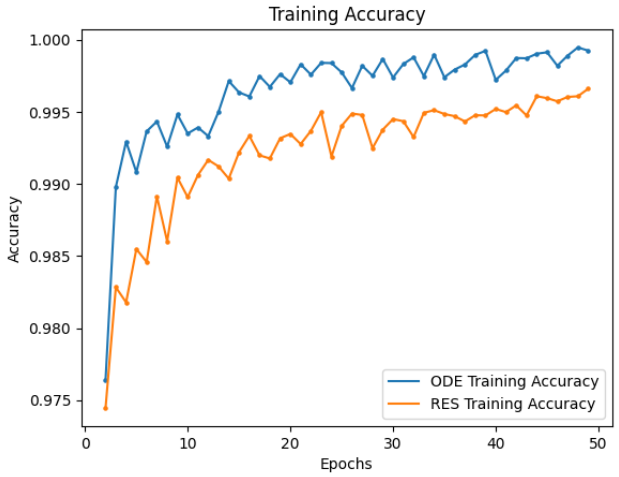

Figure 3. Training Accuracy by Epoch

An additional crucial observation that can be drawn from the testing accuracy in Figure 4 is that, for both *ODENet* and *ResNet*, there does not appear to be a distinct trend as the number of trained epochs increases. It is worth noting that although it is not always the case, the model often becomes more fitted as more epochs are executed. This outcome leaves some questions open for further investigation.

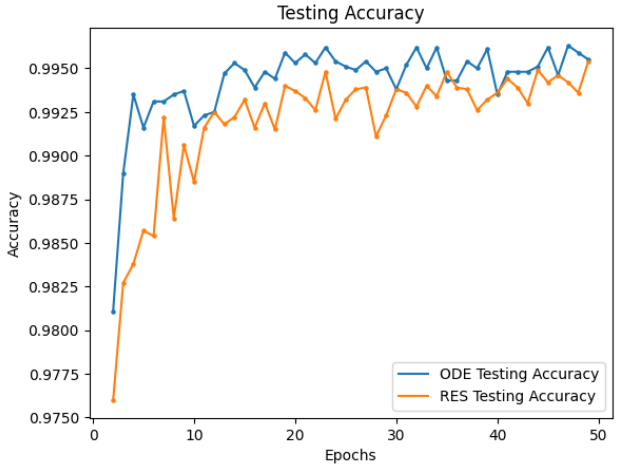

Figure 4. Testing Accuracy by Epoch

One of the primary reasons for conducting the experiment with both *ODENet* and *ResNet* is the necessity for a quantifiable method to determine which model is superior. The authors of the original paper (Chen et al., 2019) assert that ODENet is a better architecture, and the findings of this experiment corroborate their claim. In addition, they argue that the memory cost for ODENet is merely $\mathcal{O}(1)$, although this aspect has not been verified within the scope of this experiment.

Moreover, it is essential to explore the potential advantages and limitations of both models in a broader context, beyond the scope of the current experiment. This may involve investigating different use cases, considering other datasets, or examining additional performance metrics. Furthermore, the computational complexity and memory cost of both models should be thoroughly analyzed to provide a more comprehensive comparison. By delving deeper into these aspects, researchers can gain a better understanding of the practical applications and potential drawbacks of each model, ultimately leading to more informed choices when selecting a network architecture for various tasks.

5. Conclusion

In conclusion, this paper presented the reimplementation of the Neural Ordinary Differential Equations (NODE) framework, originally introduced by Ricky T. Q. Chen et al. (Chen et al., 2019) The primary goal was to reproduce the key findings and results from the original paper, showcasing the advantages of using ODE-based neural networks for flexible and efficient processing of time-series data.

The main motivation for exploring ODEs as an alternative to traditional residual networks stemmed from the limitations of the latter. Conventional methods necessitate a fixed number of layers and a fixed computational budget for processing each input, leading to a bottleneck when handling large and complex datasets. These datasets demand a high degree of flexibility in the model architecture, which traditional methods struggle to provide.

To circumvent this bottleneck, we employed Ordinary Differential Equations-based network architecture, which allows for continuous depth and more efficient training on time-series data. Instead of explicitly specifying the behavior of each layer, we defined a continuous-time dynamics on hidden states, governed by an ODE. This ODE could be solved by various existing ODE solvers, generating a flow of hidden states used for making predictions. By treating the layers of a neural network as an ODE, we learned the optimal depth and computational budget for each input during training.

A critical aspect of our reimplementation was the use of the Adjoint Sensitivity method to efficiently compute gradients required for training the ODE-based neural network. This technique solved a second augmented ODE backwards in time, facilitating the calculation of gradients while minimizing memory overhead. As a result, the NODE framework could leverage existing ODE solvers to accurately and adaptively compute gradients.

Our experiments and results demonstrated the effectiveness of the NODE framework as compared to the ResNet. For instance, we observed improved performance in accuracy when using ODENet. Furthermore, the NODE framework provided a more memory-efficient approach to training neural networks compared to traditional methods (Chen et al., 2019), a claim I have not been able to verify, mostly due to the limited access to hardware.

Additionally the process of training ODENet is very straightforward and similar to training a ResNet. This means that realistically any ResNet can be replaced with and ODENet, and achieve optimal performance. The ability to treat the training module, in this case an ODE solver, as black box allows for a high level abstraction, that makes it easier for people to use. They can use it without the knowledge of the internal layers.

It is essential to acknowledge the limitations of the NODE framework. While it excels in specific applications, it may not always outperform traditional methods in every scenario. If the data is regularly spaced and timed, a normal ResNET will perform better because of sampling. ResNets are also an established architecture, which means that they are well studied and have multiple resources. However, the NODE framework serves as a valuable alternative approach that can be considered when dealing with complex time-series data and adaptive computation requirements.

In scenarios involving regularly spaced time-series data and well-established architectures, a traditional ResNet may perform better than the NODE framework. Regularly sampled time-series data aligns with ResNet's design, as it is optimized to handle fixed sampling rates. Additionally, when using proven architectures such as ResNet-50, ResNet-101, or ResNet-152, these networks have been extensively fine-tuned and tested for various tasks, leading to robust performance. In such cases, the complexity and adaptability offered by the NODE framework might not outweigh the benefits of an optimized ResNet, making the latter a more suitable choice for handling consistent time-series data with established architectures.

In summary, the re-implementation of the Neural Ordinary Differential Equations framework highlighted its potential as a more efficient and flexible method for training neural networks. Our work confirmed the key findings from the original paper and showcased the advantages of using ODE-based architectures in various applications. We believe that the NODE framework holds great promise for future research and development in the field of deep learning, opening up new avenues for exploration and innovation.

6. Acknowledgements

I would like to express my deepest gratitude to the course staff for providing me with this incredible opportunity and for their support throughout this semester. The paper Neural Ordinary Differential Equations (Chen et al., 2019) has been the cornerstone of all the research and work presented in this paper, and I am immensely appreciative of the insights it has provided.

The original implementation by Ricky Chen (Chen, 2021) has been an great resource, serving as the foundation upon which all the experiments and outcomes discussed in this paper have been built.

Furthermore, the comprehensive paper by G. Allaire (Allaire) has enhanced my understanding of the *Adjoint Sensitivity Method*, allowing me to effectively implement and explain this concept in its usage in Neural Ordinary Differential Equations.

In addition to the course staff and the authors of the aforementioned papers, I would like to acknowledge my peers who are not in this class, who have provided invaluable feedback amd suggestions throughout the course of the semester. Their input has been crucial in refining my ideas and shaping the overall direction of the work presented herein.

Finally, I would like the acknowledge the refinement that was provided by GPT3.5 for the some text written by me, to make sure that I did not make any grammatical or punctuation errors.

Once again, I am grateful to everyone who has been a part of this experience, and I look forward to continuing my journey in the exciting world of machine learning research.

References

Datasets & DataLoaders ; PyTorch Tutorials 2.0.0+cu117 documentation — pytorch.org.

Allaire, G. A review of adjoint methods for sensitivity analysis, uncertainty quantification and optimization in numerical codes.

Chen, R. T. Q. torchdiffeq, June 2021. URL https://github.com/rtqichen/torchdiffeq.

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. Neural ordinary differential equations, 2019.

Ince, E. *Ordinary Differential Equations*. Dover Books on Mathematics. ISBN 9780486603490.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.