Neural ODEs

Jacob Wiberg

May 7, 2025

1 Introduction

Many of the most useful tools created with AI and ML are created using Neural Networks, computational structures modeled after our own brains. Neural Networks are a fundamental concept in modern computing, and one of the most powerful tools we have. Traditional Neural Networks consist of layers of discrete interconnected nodes, or "neurons," that process data by adjusting the strength of their connections based on input patterns [5]. By learning from examples, Neural Networks can recognize complex relationships in data, making them especially powerful for tasks such as image recognition, natural language processing, and predictive modeling. They are a key technology behind deep learning, which has enabled significant breakthroughs in artificial intelligence.

2 Traditional Neural Networks

The traditional Neural Network refers to a type of Neural Network called a feedforward Neural Network. In these structures, data flows through discrete layers in one direction. It starts as input, is processed through several hidden layers, and becomes output. These Neural Networks can be represented as a series of equations [2]:

$$h_1(x) = f_1(x), h_2 = f_2(h_1), \cdots, h_n = f_n(h_{n-1}) \tag{1}$$

in which each hidden layer is represented by a function h_n. Each layer takes the previous layer (or the input for layer 1) as input and outputs a transformed input, which is one step closer to the final output. When each layer is passed, the network has a final prediction, which will be compared against the target prediction using a loss function. This result is then back-propagated through the network so that the parameters (weights and biases of each neuron) can be adjusted to minimize loss (error).

Traditional Neural Networks, once trained, are fast and easy to deploy on standard hardware, easy to implement, very versatile, and highly expressive when given enough layers and neurons. With sufficient compute, traditional Neural Networks can approximate any continuous function. This means that as long as the relation between inputs and outputs is continuous, a Neural Network (in principle) can model it no matter the complexity [6]. This is why these tools are so powerful. They are able to model the underlying function of any activity, even if we ourselves do not know it. Take, for example, the self-driving car. We humans would find it nearly impossible to write down a mathematical function that would allow a machine to predict every decision needed to drive a car, but with enough computing power and training, a Neural Network can create this function.

3 Limitations of Traditional Feedforward Neural Networks

While traditional Neural Networks are extremely powerful, they do have their limitations. The Universal Approximation Theory, the theorem that states Neural Networks can approximate any continuous function, guarantees the possibility, not the efficiency of a model [6]. There are a few factors that make traditional Neural Networks slower than others. Having a discrete layer depth means that the model can't quickly adjust its computation depth depending on the complexity of the input [2]. This means that the model will use the same amount of compute to generate output for a very simple and a very complex problem/relationship. This discrete layer depth also means that Traditional

Neural Networks are not suited for modeling physical systems governed by continuous time dynamics because, as the name implies, continuous time dynamics don't operate based on discrete layers of time, but rather a smooth curve [2].

There are also significant memory inefficiencies in these Neural Networks, especially if they are deep Neural Networks. In a feedforward (traditional) Neural Network, the output of each layer l is [3]:

$$a^{(l)} = \sigma(z^{(l)}) \quad \text{where} \quad z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \tag{2}$$

where $a^{(l)}$ is the activation vector, $W^{(l)}$ is the weight matrix, and $b^{(l)}$ is the bias for layer l. During back propagation, to compute the gradient of the loss L with respect to the weights $W^{(l)}$, the chain rule is applied [3]:

$$\frac{\partial L}{\partial W^{(L)}} = \delta^{(l)} \cdot (a^{(L-1)})^\top \tag{3}$$

where $\delta^{(l)} = \frac{\partial L}{\partial z^{(l)}}$ is the error term at layer l and $a^{(l-1)}$ is the activation from the previous layer. This multiplication computes how the loss changes with each weight. As you can see, the gradient depends on both the current error and the previous layer's activation, meaning back propagation needs to store all intermediate activations for gradient computation. This can lead to extremely high memory usage in complex models [2].

4 Neural ODEs

To solve these problems, the programmers of Neural Networks needed a method of reformulating the transformation from input to output as a continuous-time function. This is where Ordinary Differential Equations come in. Neural Ordinary Differential Equations (Neural ODEs) were introduced by Ricky T. Q. Chen, along with Yulia Rubanova, Jesse Bettencourt, and David Duvenaud [2]. The concept was first presented in their 2018 paper titled: "Neural Ordinary Differential Equations" [2]. This paper proposed a new family of deep learning models where the transformation of hidden states is defined by a continuous-time differential equation parameterized by a Neural Network. Instead of stacking discrete layers, Neural ODEs model the hidden state dynamics continuously, solving the ODE with a black-box ODE solver, a general-purpose numerical algorithm that can solve ODEs without requiring the user to know or modify the internal workings of the solver. The equation they came up with is:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta) \tag{4}$$

in which $\mathbf{h}(t)$ is the hidden state at time t, and θ are the parameters of the Neural Network f. The ODE is solved from an initial value $\mathbf{h}(t_0)$ over a time interval $[t_0, t_1]$, giving the integral that is computed by the black box ODE solver:

$$\mathbf{h}(t_1) = \mathbf{h}(t_0) + \int_{t_0}^{t_1} f(\mathbf{h}(t), t, \theta) dt \tag{5}$$

With these equations, Neural Networks can define the entire trajectory of the hidden states using a continuous differential equation, rather than a series of calculations [2]. They go from having a discrete number of layers upon which problems can be solved to having a dynamic function that can adapt to inputs of varying complexity. The model can now change its "depth" of computation for more or less complex inputs. Also, now that the function is continuous, the Neural Network is much better equipped to model systems based on continuous time, like physical simulations, dynamical systems, and irregularly sampled time-series.

Neural Ordinary Differential equations also make Neural Networks much more memory efficient because the Neural Network no longer needs to store each activation, but rather can compute the gradient of the loss function for a state $L(h(t))$ with respect to the parameters θ using the adjoint sensitivity method [2]. This is a method for finding the gradient by defining an adjoint state $a(t)$ and solving a new ODE backward in time. Rather than backpropagating through each layer/time step, the network solves the following adjoint ODE backward from $t = T$ to $t = 0$ [1]:

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

and computes the parameters by solving:

$$\frac{dL}{d\theta} = -\int_T^0 a(t)^\top \frac{\partial f(h(t), t, \theta)}{\partial \theta}\, dt \tag{7}$$

With this method, activations no longer need to be stored to optimize the loss function, as they are now computed on the fly, making the program much less memory-intensive.

Lastly, using ODEs to train Neural Networks allows them to be invertible and reversible. This means that not only can the Neural Network construct output from input, but also reconstruct input from a given output. This property is essential in models like normalizing flows, where you want to map simple distributions to complex ones and compute exact likelihoods of a data point for training a model [4]. Simply, Neural ODEs not only learn how to warp simple distributions of data into complex ones, but also can determine how likely an output is based on an input.

5 Conclusion

Neural Ordinary Differential Equations really show us how useful ODEs can be for solving real-world problems. They show that even though the math behind differential equations can be difficult to wrap your head around at first, the applications they allow for can make extremely large and complex problems (like Neural Networks and deep learning) more simple and efficient. Getting a better understanding of how these equations can be used in actual technology that we may encounter in the future of our engineering careers allows us to gain a deeper understanding of what they are and why we are learning about them now.

References

[1] Y. Cao, S. Li, L. Petzold, and R. Serban, "Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and Its Numerical Solution," *SIAM Journal on Scientific Computing*, vol. 24, pp. 1076–1089, Jan. 2003.

[2] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural Ordinary Differential Equations,"

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models," Oct. 2018. arXiv:1810.01367 [cs].

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[6] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[1] [2] [3] [4] [5] [6]