Artificial Neural Networks for Solving ODEs and PDEs

Jun Hao (Simon) Hu

August 2, 2018

Abstract

Neural networks (NNs) are the holy grail of modern computation. They have seen their fair share of applications across numerous scientific disciplines. In this work, I present a method for solving initial and boundary-value problems using artificial neural networks (ANNs). I will mainly study its applications to both ordinary and partial differential equations up to the second order. Additionally, I will study error properties of this numerical method. Several examples and comparisons against other numerical methods commonly used will be presented.

1 Introduction.

Numerical methods for solving differential equations are crucial to many problem in physics and engineering. The task of solving differential equations numerically, however, can be a computationally difficult challenge. In part, this is due to the curse of dimensionality and sometimes properties of the function space that the solution is an element of. Traditionally, numerical methods such as finite differences, element, and volume are adequate for most applications. Unfortunately, since these numerical methods are based on a particular collocation of the domain and as such, only output the solution to the PDE at the collocation points. This is acceptable in real-world applications, but not in theoretical applications where we want to extrapolate properties of solutions to the PDE. To mollify this issue, we employ neural networks, which can produce analytic solutions. It is well know that a multilayer perceptron, with one hidden layer, has excellent function approximation properties. As such, it is no surprise that we would want to employ these types of neural networks for solving differential equations, once we cast the PDE problem as a function approximation problem. By employing a feed-forward neural network architecture to solve differential equations, we obtain an analytic approximation to the solution which can have the required topological properties. These topological properties are desired when analyzing PDEs.

This numerical method exploits the ability of neural networks with a feed-forward architecture to approximate functions up to an arbitrary accuracy. These approximations have a closed form and have some degree of regularity, which makes these approximations desired for practical applications. The idea is to first guess a form of the solution, called the trial solution, and then train a neural network to learn the network parameters, which in this case are the weights and biases associated with each node. These network parameters are learned by minimizing an appropriate cost function. This trial solution is constructed with the initial and boundary conditions in mind.

Though the study of existence and uniqueness for PDEs is a rich and interesting field of study, we will not concern ourselves with these issues. As a caveat, I will assume that the differential equation problem has a solution, and this solution does not need to be unique. Furthermore, I will only consider differential equations up to, and including, the second order. This method can potentially work for higher-order differential equations, but that is an area that will not be touched in this paper.

1.1 Structure of Paper.

In section 2, I present the proposed numerical method. In section 3, 4, and 5, I focus on classes of linear ODEs, systems of ODEs, and PDEs respectively, that can be tackled using the proposed method. Examples will be presented and the results will be compared against numerical solutions obtained via other methods. In section 6, I discuss the possibility of using this method for non-linear PDE. In section 7, I discuss further directions this research topic can take.

2 Proposed Numerical Method.

We begin with a remark regarding the notation used in this paper.

Remark 2.1. For the remainder of the paper,

1. $\Omega \subset \mathbb{R}^n$ is an open set,
2. $\partial\Omega$ denotes the boundary of Ω,
3. $\bar{\Omega} = \Omega \cup \partial\Omega$ denotes the closure of Ω,
4. we will use the shorthand f_i to denote $f(x_i)$, where f is any arbitrary function
5. D and D^2 denote the gradient and Hessian operators, respectively.

2.1 Formulation of the Problem.

Consider the problem of finding a function $\psi = \psi(x)$, $\psi : \bar{\Omega} \to \mathbb{R}$ that, for some specified $f = f(x)$, $f : \Omega \to \mathbb{R}$, satisfies

$$G(x, \psi(x), D\psi(x), D^2\psi(x)) = f(x),\ x \in \Omega \tag{2.1}$$

subject to appropriate initial and boundary conditions. This is not a particularly easy problem to solve, especially if the left-hand side is complicated and/or non-linear. Additionally, the requirement that Eq. (2.1) hold *for all* $x \in \Omega$ creates a computational burden. As this proposed method is a numerical method at its core, the problem must be treated as a numerical one. Thus, we relax the condition by saying that, while Eq. (2.1) may not hold for all $x \in \Omega$, can it hold for a finite collect of x? To relieve the burden, we perform a discretization of the domain Ω and its boundary $\partial\Omega$. Let $U \subset \Omega$ be the discretized domain and ∂U be its boundary. With this particular discretization, the problem reduces to finding a function ψ that satisfies

$$G(x_i, \psi_i, D\psi_i, D^2\psi_i) = f(x_i),\ \forall\, x_i \in U \tag{2.2}$$

subject to a discretized version of the initial and boundary conditions.

2.2 Trial Function.

We will construct a function $\psi_{\text{trial}} = \psi_{\text{trial}}(x, p)$, $\psi_{\text{trial}} : \bar{\Omega} \to \mathbb{R}$, where p is the vector of parameters, that solves (2.2). For our cases, the function ψ_{trial} contains the weights and biases associated with our network, which will be learned by the network using some sort of gradient descent algorithm. In light of the trial solution, (2.2) becomes an approximation problem! We have an initial guess of our solution and that initial guess is fine-tuned so that the left-hand side approximates the right-hand side.

Let us suppose that the trial function has the form

$$\psi_{\text{trial}}(x, p) = \xi(x) + \gamma(x)N(x, p) \tag{2.3}$$

where $N(x, p)$ is the output of the feed-forward neural network. The function $\xi(x)$ is chosen so that the boundary and initial conditions are satisfied, and $\gamma(x)$ is chosen so that it does not contribute to the boundary and initial conditions. Since we used collocation methods to reduce the problem to a numerical problem, it is often true that $\gamma(x)$ and $\xi(x)$ are constructed using discretized boundary conditions.

Our hope is that, with the learned parameters in hand, the function ψ_{trial} smoothly approximates ψ to some degree of accuracy. To measure the effectiveness of our approximation, we need to decide on an appropriate cost function.

2.3 Cost Function.

Choosing a good cost function is an art in and of itself, but it is usually clear from the problem at hand what the cost function should be, since it has a physical interpretation. For example, when performing statistical inference, an underlying metric is specified, so the cost function should be chosen so as to optimize with respect to this metric. A good choice of a cost function can make the difference between an effective and ineffective model. For simplicity, when discussing linear ODEs and systems of ODEs, we will use the standard squared-error cost function. That is, the loss function $J = J(p)$ is given by

$$J(p) := \sum_{x_i \in U} [G(x_i, \psi_i, D\psi_i, D^2\psi_i) - f_i]^2. \tag{2.4}$$

Training the network is equivalent to finding p^* such that

$$p^* := \arg\min_p [J(p)].$$

With this p^* in hand, the optimal trial solution is therefore given by $\psi_{\text{trial}}(x, p^*)$, which we treat as the solution to (2.1).

The reader should note that (2.4) is not always the most optimal choice of cost function. As we will see later with PDEs, a better choice of cost function should optimize with respect to the norm of the function space in which the solution lies in. With PDEs, the choices of cost functions are much richer as there is a lot of theory, that we will not discuss in detail, behind solutions to PDEs. To reiterate, the choice of cost function requires a lot of knowledge about the underlying theory of differential equations. Our choice is based on simplicity.

2.4 Neural Network Architecture.

Consider a feed-forward neural network architecture with one hidden layer, an input layer, and an output layer, as shown in the figure below.

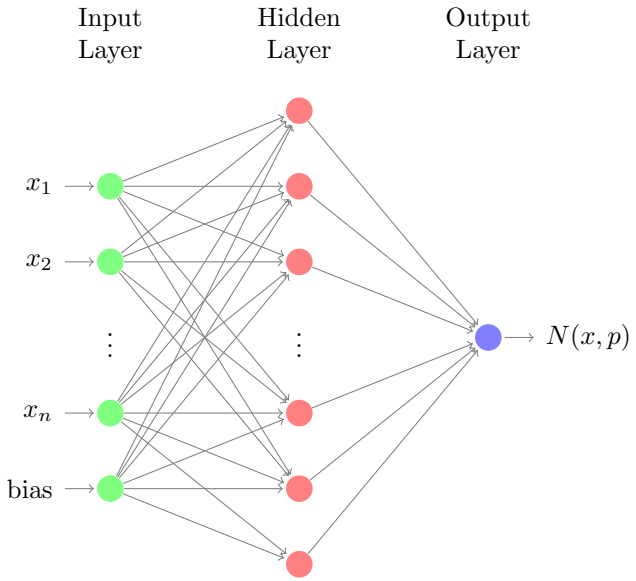

Figure 1: Schematic of a feed-forward neural network with one hidden layer.

In this neural network structure, we have an input layer with $n + 1$ nodes, a hidden layer with m nodes, and an output layer with 1 node. The output of this neural network is the function $N(x, p)$, which appears

in our assumed form of the trial solution. We will use the ReLU activation function, however, wherever appropriate we will also use the leaky ReLU activation function to rectify issues arising from dead neurons. The ReLU activation function is chosen due to its effectiveness at deterring the vanishing gradient effect.

3 Ordinary Differential Equations.

4 Systems of Ordinary Differential Equations.

5 Linear Partial Differential Equations.

5.1 Monte Carlo Method for Efficient Computation of Gradient.

6 Non-Linear Partial Differential Equations.

7 References.