Artificial Neural Networks for Solving ODEs and PDEs

Jun Hao (Simon) Hu

August 1, 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.

I will assume that the reader has some experience dealing with differential equations and neural networks. The reader should not be too concerned if they do

In the proceeding section, I will develop the framework for the proposed numerical method. The general problem to be solved will be presented. It is also in this section where I introduce error properties associated with this method, as well as discuss opportunities and dangers. Sections 3, 4, and 5 focus on classes of linear ODEs, systems of ODEs and PDEs, respectively, that can be tackled using the proposed method. Section 6 discusses the possibility of using this method for non-linear PDE. In sections 3, 4, 5, 6, examples will be presented.

2 Formulation of the Method.

For the remainder of this paper, we will work in some open set $\Omega \subset \mathbb{R}^n$. Consider the problem of finding a function $\psi = \psi(x), \psi : \Omega \to \mathbb{R}$ that satisfies the general differential equation

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

subject to appropriate initial and boundary conditions. In the above, D is the gradient, D^2 is the Hessian operator, and $f = f(x)$ is a known function.

The above condition that the differential equation hold for *all* $x \in \Omega$ produces a computational burden. To placate this issue, we relax the condition and say that the differential equation holds for a finite number of $x \in \Omega$. We perform some sort of discretization of the domain Ω and its boundary $\partial\Omega$. Let $U \subset \Omega$ be the discretized domain and ∂U be its boundary. Thus, the above problem reduces to finding ψ that solves the differential equation

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

subject to the initial and boundary conditions, as before.

Remark 2.1. For notational simplicity, we will denote $\psi_i = \psi(x_i)$. In a similar manner, we use ϕ_i and f_i to denote $\phi(x_i, p), f(x_i)$, respectively.

Our goal is to construct a function $\phi = \phi(x, p), \phi : \Omega \to \mathbb{R}$, where p is the vector of parameters, that solves (2.2). The function ϕ contains parameters (the weights and biases), which can be learned by the neural network. Our hope is that, with the learned parameters in hand, the function ϕ approximates ψ to some degree.

Next, we discuss our choice of a loss function. This is an art in and of itself, but it is usually clear from the problem at hand what the loss function should be, since it has a physical interpretation. For example, when performing statistical inference a metric is specified, so your loss function should be chosen so as to optimize with respect to that metric. A good choice of a loss function can make the difference between an efficient and inefficient model. For the sake of simplicity, we will use the standard squared-error index. Thus, the loss function, J is given by

$$J(p) := \sum_{x_i \in U} (G(x_i, \phi_i, D\phi_i, D^2\phi_i) - f_i)^2. \tag{2.3}$$

Thus, our goal is to train the network to learn the p^* such that

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

For the feed-forward architecture, the parameters that must be learned are the weights and the biases.

3 References.