IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier none

Neural Networks for Solving First-order Ordinary Differential Equations

DIOGO MEALHA[1]
[1]University of Aveiro, Physics Department (e-mail: mealha@ua.pt)

This project was done for a Fundamentals in Artificial Intelligence class

ABSTRACT Neural networks have emerged as powerful tools for solving differential equations, offering a versatile approach to approximating solutions with high accuracy. In this paper, we present a comprehensive tutorial on how one can use neural networks for solving first-order ordinary differential equations. We begin by defining the problem statement, aiming to find a function $y(x)$ that satisfies an initial value problem. Our approach involves implementing a simple neural network architecture comprising three parameters: w, β, and v, while employing an activation function to introduce non-linearity. We unveil the mathematics behind this implementation, simplifying the complexities for practical application. Furthermore, we demonstrate and discuss the results obtained in three different scenarios with varying complexity, through our approach, shedding light on the efficacy and potential of neural networks in solving ordinary differential equations.

INDEX TERMS Neural Networks, Ordinary Differential Equations, Gradient Descent.

I. INTRODUCTION

ORDINARY differential equations (ODEs) constitute a fundamental framework for mathematical modeling across scientific and engineering disciplines. These equations describe the relationship between a dependent variable and its rate of change with respect to an independent variable. The solutions of ODEs offer invaluable insights into the temporal evolution of diverse systems.

Neural networks, as stated in the Universal Approximation Theorem, can be used to approximate any function with an arbitrary level of accuracy. Therefore using them for solving this type of equations is possible.

II. MOTIVATION

This paper aims to present an intuitive and efficient approach for utilizing neural networks to solve first-order ODEs. We focus on developing a clear and accessible tutorial that guides readers through the implementation process. Complementing this paper there is also a jupyter notebook written in python.

III. PROBLEM

The problem we are trying to solve is as follows:

$$\frac{dy}{dx} = f(x, y), \quad y(a) = A \tag{1}$$

Where $f(x, y)$ is a function we have beforehand, that is used later to tune our neural network (nn). Therefore, we are looking for a solution $y(x)$ that satisfies an initial value problem. In this paper, for the sake of simplicity, we will consider $a = 0$.

IV. NEURAL NETWORK

In this paper, we employ a basic nn architecture, with only one hidden layer, as is depicted in figure 1. This nn consists of three parameters: w, representing the first weight; β, denoting the bias term; and v, representing the second weight. The activation function σ, crucial for introducing non-linearity, is utilized to transform the output of the hidden layer. In this paper, we adopt the sigmoid function, defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

V. MATHEMATICS

In this section, we delve into the mathematical definitions of an nn and how we assess its performance.

Let P represent the matrix containing all weights of the nn. We denote the output of this nn, given a single input x, and for a total of q neurons, as $N(x, P)$ or simply N. Mathematically, this relationship is expressed as:

$$N(x, P) = \sum_{i=1}^{q} v_i \sigma(w_i x + \beta_i) \tag{3}$$

A. COST FUNCTION

To assess the performance of our nn and guide the weight updates during the training, we define a cost function. This

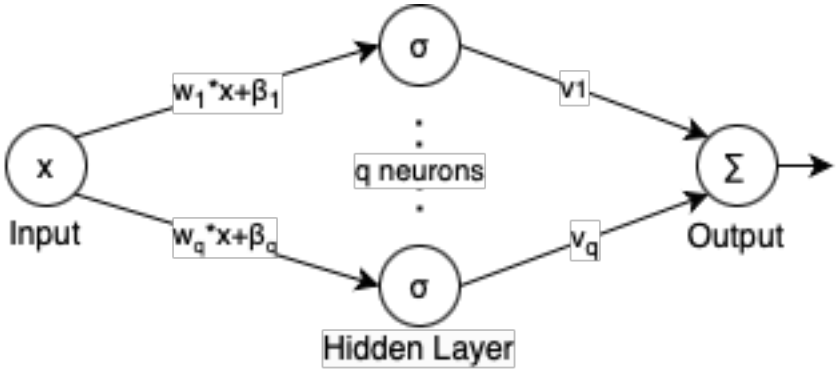

FIGURE 1. Structure of neural network

function is derived from equation 1. By checking that equality on all points of our training data and squaring the difference to ensure non-negativity, we obtain the following function:

$$J(\vec{x}, P) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{dy}{dx} - f(x_i, y)\right)^2 \tag{4}$$

Where $\vec{x}$ is a vector of lenght m, which is the number of training data points.

This cost function has been use in many different papers and has been shown to work.

B. UPDATING WEIGHTS

For our nn to "learn", we need to update its weights. This can be done through various different methods, which will be discussed in the conclusion. In this paper we will use a simple gradient descent algorithm. This process happens through iterations, taking into account the cost function defined in equation 4:

$$P_i = P_i - \alpha\frac{\partial J}{\partial P_i} \tag{5}$$

In which α is the learning rate, controlling the step size during weight updates.

Deriving the cost function can be computationally intensive. In the next section we will present all the the essential formulas required for this computation.

C. TRIAL SOLUTION

The solution for our problem can be written generically like this:

$$y_t(x) = A + (x - a) * N(x, P) \tag{6}$$

Where y_t is called the trial solution.

Equation 6 enforces the initial conditions, guarantying that $y_t(a) = A$, despite the value of N. Since we consider $a = 0$, this equation can be written in a simpler form.

VI. DERIVATIVES

All the derivatives necessary to implement the gradient descent method will be presented here.

With the definition of y_t as shown in equation 6, we can express its derivative as follows:

$$\frac{dy_t}{dx} = N(x, P) + x\frac{\partial N(x, P)}{\partial x} \tag{7}$$

Using this equation, we can extensively write the formula for the derivative of the cost function in respect to a single weight:

$$\frac{\partial J}{\partial P_i} = \frac{2}{m}\sum_{i=1}^{m}\left(N(x_i, P) + x_i\frac{\partial N(x_i, P)}{\partial x}\right) * \left(\frac{\partial N(x_i, P)}{\partial P_i} + x_i\frac{\partial}{\partial P_i}\left(\frac{\partial N(x_i, P)}{\partial x}\right) - \frac{\partial f(x_i, y_t)}{\partial P_i}\right) \tag{8}$$

Looking at equation 8, there are still a few derivatives we need to calculate. The equations presented next will be generalized, and can be used when trying to implement an nn to solve higher order differential equations.

Firstly, the λ derivative of N with respect to x:

$$\frac{\partial^\lambda N}{\partial x^\lambda} = N^{(\lambda)} = \sum_{i=1}^{q} v_i w_i^\lambda \sigma^{(\lambda)}(w_i x + \beta_i) \tag{9}$$

In which $\sigma^{(\lambda)}$ is the λ derivative of the activation function.

The derivative of $N^{(\lambda)}$ in respect to the first set of weights w:

$$\frac{\partial N^{(\lambda)}}{\partial w_i} = \begin{cases} v_i x\sigma^{(1)}(w_i x + \beta_i), & \lambda = 0 \\ v_i(\lambda w_i^{\lambda-1}\sigma^{(\lambda)}(w_i x + \beta_i) + \\ \quad x w_i^\lambda \sigma^{(\lambda+1)}(w_i x + \beta_i)) & , \lambda > 0 \end{cases} \tag{10}$$

The derivative of $N^{(\lambda)}$ in respect to the bias β:

$$\frac{\partial N^{(\lambda)}}{\partial \beta_i} = v_i w_i^\lambda \sigma^{(\lambda+1)}(w_i x + \beta_i) \tag{11}$$

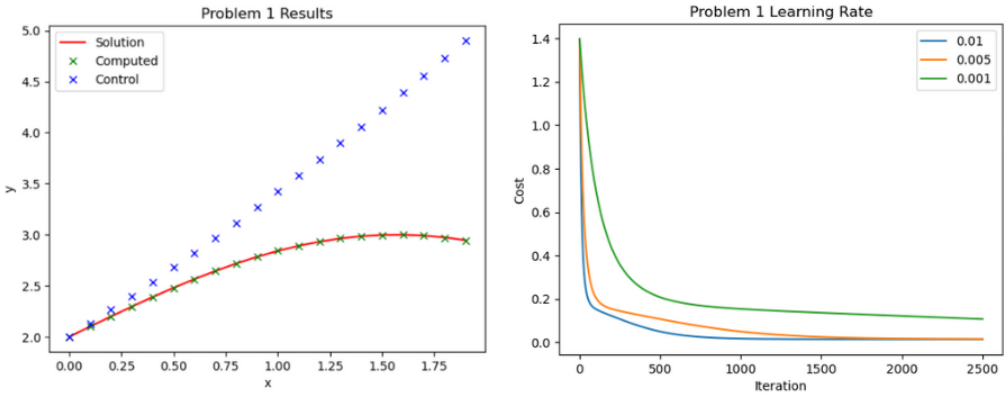

FIGURE 2. Problem 1 charts

And lastly, the derivative of $N^{(\lambda)}$ in respect to the final weight v:

$$\frac{\partial N^{(\lambda)}}{\partial v_i} = w_i^{\lambda}\sigma^{(\lambda)}(w_i x + \beta_i) \tag{12}$$

The final thing we need to calculate is the derivative of our function f in respect to a weight P_i, which can be done using the chain rule:

$$\frac{\partial f(x,y)}{\partial P_i} = \frac{\partial f}{\partial y} * \frac{\partial y}{\partial N} * \frac{\partial N}{P_i} \tag{13}$$

The derivative of f in respect to y needs to be calculated beforehand. If f does not depend on y, then we can simplify this equation since it is equal to 0. The derivative of y in respect to N is simply x and the rest has already been shown.

With these formulas ready we can begin discussing results.

VII. RESULTS

When discussing the results obtained we will look at the cost function variation when training our nn with different learning rates and the comparison between the real solution and the output given by equation 6.

A. PROBLEM 1

The first problem we are trying to solve is the following:

$$\frac{dy}{dx} = cos(x), \;\; y(0) = 2 \tag{14}$$

Which has the real solution:

$$y(x) = sin(x) + 2 \tag{15}$$

Since $f(x, y) = cos(x)$, in this case, the derivative of the cost function gets simpler.

This nn was trained over 70000 iterations and took around 25 seconds to finish with a cost of approximately $7.68e^{-5}$. This value is very close to 0 and the left graph on figure 2 shows it. We can see that the green dots are very close to the real solution. Furthermore, on the right graph we see the tendency of the cost function to converge to 0 despite the learning rate.

B. PROBLEM 2

In case 2, we decided to make f a function of both x and y. This time, the problem we are trying to solve is as follows:

$$\frac{dy}{dx} = \frac{cos(x)}{2y}, \;\; y(0) = 1 \tag{16}$$

The real solution can be calculated and is given by:

$$y(x) = \sqrt{\sin(x) + 1} \tag{17}$$

Admitting that $y > 0$.

In this problem we should not simplify equation 8. Nevertheless, even if we do so, the results are still satisfying.

In figure 3 the graph representing the loss function curve is a little different. For each learning rate we plotted the variation without simplifying equation 8, represented with a line. And another considering equation 13 equal to 0, represented with dots. We can see that the difference is practically null.

In the left graph we see the performance of our nn after it was trained for 70000 iteration, taking around 25 seconds, finishing with a cost of around $5.61e^{-5}$. This value is very similar to cost we got when training another nn for the same number of iterations with the simplification of equation 8, which was $5.83e^{-5}$. This was also done about 1 second faster.

C. PROBLEM 3

In this section we tried solving a complex first-order ODE:

$$\frac{dy}{dx} = y + x, \;\; y(0) = 0 \tag{18}$$

This differential equation is hard for humans to solve. Nevertheless the solution is as follows:

$$y(x) = -1 - x + e^x \tag{19}$$

The results from our nn, without simplifying equation 8 are represented in figure 4. The nn was trained over 100000 iterations, taking around 35 seconds to finish and the final cost was $9.79e^{-5}$.

The loss function curve was a more chaotic in this case, which my be related to the problem itself.

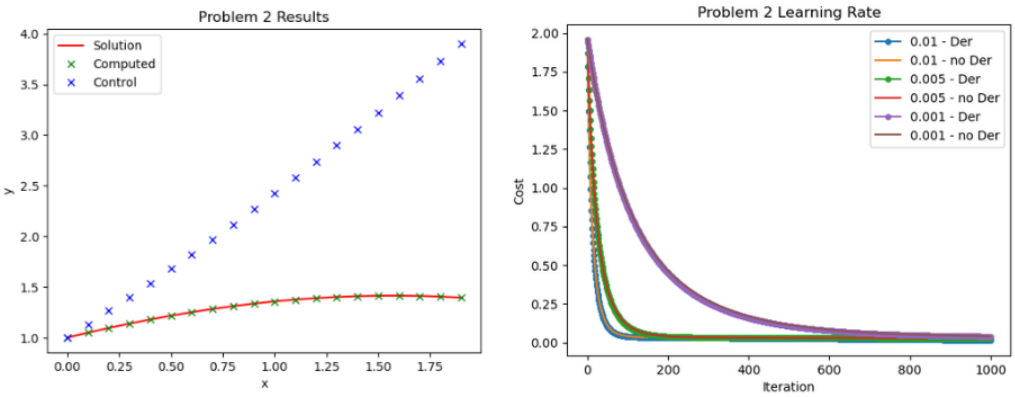

FIGURE 3. Problem 2 charts

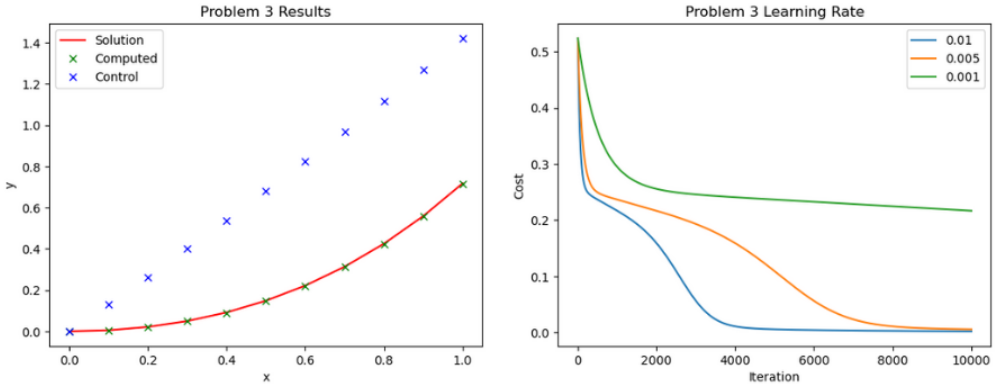

FIGURE 4. Problem 3 charts

VIII. CONCLUSION

One important thing to notice is that this strategy is flawed in the sense that only works for a specific differential equation. One step forward could be a general nn that can use the differential equation and the training data as inputs and returns the desired outputs, making it more general.

A. OPTIMIZATION

In this paper was used the gradient descent for optimization. This method is considered a perfect balance between simplicity and efficiency in the sense that it gets good results while being simple to implement. However, due to the nature of this problem which implicates deriving the output of the nn in respect to the inputs, this might get tricky. An alternative is the Nelder-Mead algorithm, implemented in [2]. Since this method is gradient free it does not require equation 8.

An interesting aspect about this problem is that there is no over-fitting. Since we know exactly the interval of our training data, we know what values we are going to feed our nn, and so we can train it for as long as we need. This is not true when using the Nelder-Mead algorithm.

B. RESULTS

In the figures shown, the results obtained all represent neural networks with a cost in the order of 10^{-5}, which we found to be a good value when displaying results.

Figures 2 and 4 are interesting in the sense that they show how different learning rates might influence the tuning of our nn. Figure 4 also shows that increasing complexity generates less traditional loss function curves, which we believe is related the complexity of function f we are trying to integrate.

The python jupyter notebook can be found here .

REFERENCES

[1] I. E. Lagaris, A. Likas and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," in IEEE Transactions on Neural Networks, vol. 9, no. 5, pp. 987-1000, Sept. 1998, doi: 10.1109/72.712178.

[2] R. Rostamian. Programming projects in C. Society for Industrial and Applied Mathematics, 2014

[3] Simos, T.E., Famelis, I.T. A neural network training algorithm for singular perturbation boundary value problems. Neural Comput Applic 34, 607–615 (2022). https://doi.org/10.1007/s00521-021-06364-1