2022 14th International Conference on Computational Intelligence and Communication Networks (CICN) | 978-1-6654-8771-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CICN56167.2022.10008361

Neural Network for Solving Ordinary Differential Equations

Wengyao Jiang
Department of applied mathematics, Xi'an Jiaotong-liverpool University
Suzhou, China
Wengyao.Jiang20@student.xjtlu.edu.cn

Chen Xuan
Department of applied mathematics, Xi'an Jiaotong-liverpool University
Suzhou, China
Chen.Xuan@xjtlu.edu.cn

***Abstract*—Deep learning and machine learning are immensely prevalent and highly interactive in a myriad of fields, typically neural networks is widely used in mathematics. We outline a technique for employing artificial neural networks (ANN) to solve ordinary differential equations. For better illustration, we present the basic logic and formula of ANN and gradient computation, following with one typical first order differential equation as example. In order to research the flexibility and feasibility of our model, we compare several hyperparameters and different optimizer using control variable method. Finally, our neural networks model is applied into the second order differential equations with innovative modification by analogy. In this article, we illustrate the relatively novel method to solve the ordinary differential equations and examine our model through adjustable parameters, then convert into the second order which shows a wide application range.**

Keywords—*Neural network, Ordinary differential equations, Pytorch visualization, Optimizer*

I. Introduction

There are many existing methods of solving ordinary differential equations. Some of them generate an array-based solution that contains the value of the solution at a chosen set of points [1]. Others turn the original issue into a system of linear equations and depict the answer in analytical form using basis-functions [2]. The majority of the prior research on utilizing neural networks to solve differential equations is limited to the scenario of solving linear systems of algebraic equations that come from the distribution of the domain. The architecture of a Hopfield neural network is mapped onto the solution of a linear system of equations [3]. The Python programming language with the tensorflow package was used to create the neural network [4].

A different strategy for solving ordinary differential equations connected to neural network is based on the fact that some spline types, like linear B-splines, can be created by superimposing piece-wise linear activation functions [5]. By resolving a system of linear or non-linear equations to determine the parameters of the splines, it is possible to arrive at the solution of a differential equation using linear B-splines as basis functions [6]. By substituting the total of the piece-wise linear activation functions that correspond to the hidden units for each spline, a solution form of this kind is immediately mapped onto the architecture of a feed-forward neural network [7-8]. This approach takes into account local basis-functions, in general, necessitates a large number of splines in order to produce precise results. Furthermore, adapting these methods to multidimensional domains is difficult.

Artificial neural network (ANN) was developed to find ODE solutions in a closed continuous analytical form in order to overcome some existing restrictions. In this way, further studies have focused on applying ANN to the solution of generic initial value problems (IVP), boundary value problems (BVP), and partial differential equations (PDE).

In this article, we mainly focus on a general method of solving ordinary differential equations using neural network with parameters containing weights and biases are adjusted to minimize the loss function, which was first used by Lagaris in 1997 [9]. In the first part, we plot several animations together in order to further understand the deep logic and hidden mechanism of neural network training by torch visualization. In the second part, we use different optimization techniques to update the gradients of parameters and compare each contribution in order to better utilize the method. It is reasonable to consider this type of network architecture as a candidate model for treating differential equations.

In the last part, we put the model into the second order differential equations to see if it works. As we know that the second order differential equation systems such as simple harmonic motion and Lotka-Volterra model [10-11], are immensely critical in mathematical and physical field.

II. Description of the Method

A. Basic logic

First, to better illustrate the method, we display the basic type of the first order ordinary differential equation:

$$\frac{du}{dt} = f(t, u), \tag{1}$$

with $t \in [0,1]$ and initial conditions (IC):

$$u(0) = u_0. \tag{2}$$

And the base function connected to the neural network is :

$$U(t,\theta) = u_0 + tNN(t,\theta), \tag{3}$$

where $\mathrm{NN}(t,\theta)$ is a neural network . The solution $U(t,\theta)$ automatically satisfies the initial conditions. The key part is to minimize the loss function when $u(0) = 0$:

$$L(\theta) = \int_0^1 \left[\frac{dU(t,\theta)}{dt} - f(t, U(t,\theta))\right]^2 dt. \quad (4)$$

Given that $\frac{dU(t,\theta)}{dt} = NN(t,\theta) + t\,\frac{dNN(t,\theta)}{dt}$, we move on to the next part to illustrate the procedures of updating the gradients of the neural network parameters.

B. *Gradient computation*

The key part of training process for the neural network may be the effective minimization of the loss function, where the loss value that corresponds to each input vector must decrease to zero. In addition to the network output, which is used in traditional training, the derivatives of the output with respect to any of its inputs are also factored into the computation of this loss number. We must thus compute both the gradient of the network and the gradient of the network derivatives with respect to its inputs in order to compute the gradient of the loss with respect to the network weights and bias.

Given that a multilayer perceptron with n input units, H sigmoid units, and one linear output unit is an example. Subsequently, we express the neural networks as $NN = \sum_{i=1}^{H} v_i \sigma(z_i)$, where $z_i = \sum_{j=1}^{H} w_{ij}x_j + b_i$, w_{ij} is the weight input unit j to the hidden unit i, v_i is the weight from the hidden unit i to the output, u_i is the bias of hidden unit i and $\sigma(z_i)$ is the sigmoid function of the ith layer. We can express the derivative below:

$$\frac{dNN}{dx_j} = \sum_{i=1}^{H} v_i w_{ij} \sigma(z_i), \quad (5)$$

$$\frac{d^k NN}{dx_j{}^k} = \sum_{i=1}^{H} v_i w_{ij}{}^k \sigma(z_i)^{(k)}, \quad (6)$$

which gives a more general formula with the kth derivative.

After defining the derivative of the error with respect to the network parameters, using practically any minimization strategy is simple. For instance, one may employ the conjugate gradient method or the steepest descent. Additionally, if parallel hardware is available, the derivatives of each network with respect to the parameters may be determined simultaneously for a particular point.

In our trial, the optimizer like adam, Gradient Descent (GD) and Stochastic Gradient Descent(SGD) will be introduced and compared later, which all work as the tools of gradient descent.

C. *One specific example*

The targeted equation is $f(t) = e^{-\frac{t}{5}}\cos(t) - \frac{t}{5}$, and the sigmoid activation of each hidden unit is $\sigma(z) = \frac{1}{1+e^{-z}}$.

Before the training, we have already known the analytic solution $uA(t) = e^{-\frac{t}{5}}\sin(t)$. In order to get greater accuracy, we first choose the iteration to be 10000, and get the result below. One thing should be mentioned that all the figures in this part are processed by optimizer adam as default.

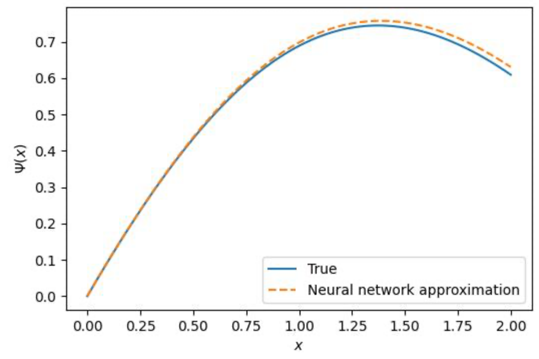

Fig. 1. Iteration=10000.

Next, we modify the code to solve x between 0 and 5. After several experiments, we found batchsize = 4 seems a good choice, while the larger iteration times generate better accuracy, but it takes more time. Maybe iteration = 50000 could show the acceptable accuracy with shorter time. The follow shows the great accuracy when iteration = 80000.

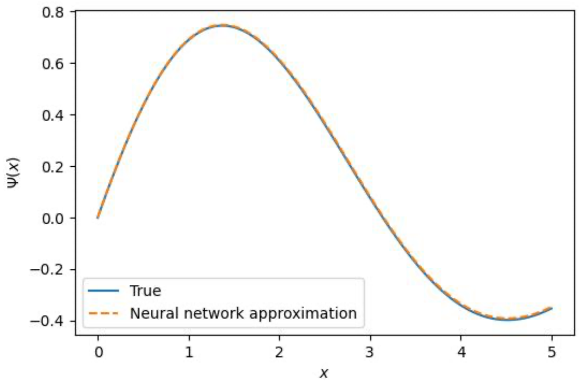

Fig. 2. Iteration=80000 for expanded domain.

After confirming the ability of neural network solving ordinary differential equations by plotting the analytic solution and NN solution, we also look through the whole progress of the parameters and the loss function during the iteration, and get the figure below. In the last picture, the little red dots stand for the number of batchsize points which are transmitted to neural network each iteration.

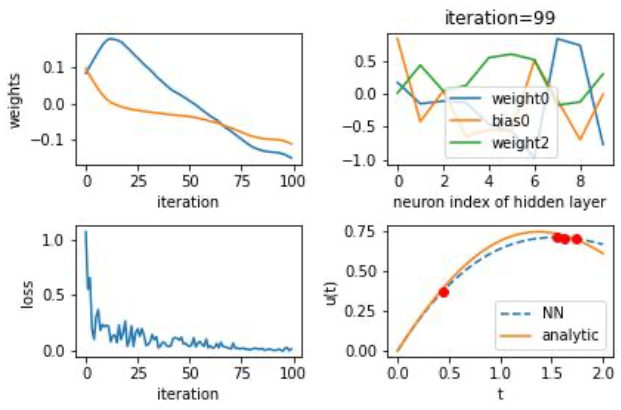
Fig. 3. Iteration=100 of multiple features.

Keep your text and graphic files separate until after the text has been formatted and styled. Do not use hard tabs, and limit use of hard returns to only one return at the end of a paragraph. Do not add any kind of pagination anywhere in the paper. Do not number text heads-the template will do that for you.

III. Adjustable Optimizer and Parameters

A. Gradient Descent (GD), Stochastic Gradient Descent(SGD) and adam

SGD differs from the GD in that GD takes the derivative of the function and calculates every sample in the whole training set, while SGD operates on one or more samples. However, in our model the two optimizer actually function in the similar way obtaining the same result. More distinct differences will be revealed in optimization methods for large-scale machine learning.

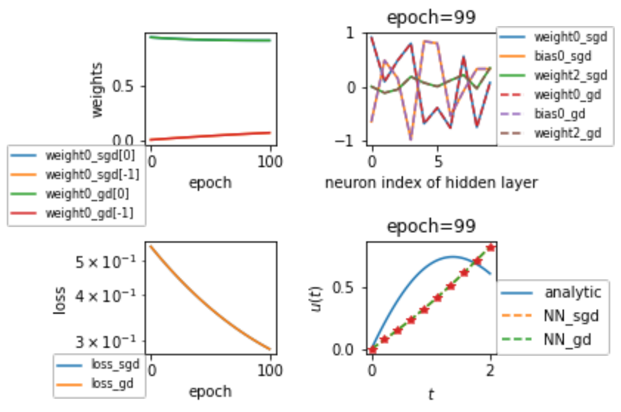
Fig. 4. GD vs SGD.

Adam takes in the benefits of momentum gradient descent method and Adagrad (gradient descent algorithm with adaptive learning rate), which can not only adapt to sparse gradient, that is, difficulties with natural language and computer vision, but also solve the issue of gradient oscillation, showing a high efficiency in quickly getting the good results comparing to SGD.

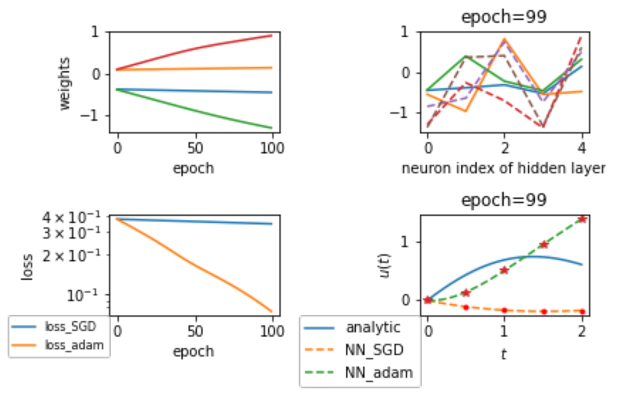
Fig. 5. SGD vs adam.

B. Minibatch

Minibatch divides the data into several batches, and updates the parameters according to the batches. In this way, a group of data in a batch jointly determines the direction of this gradient, and it is not easy to deviate when it falls down, thus reducing the randomness. On the other hand, because the number of samples in a batch is much smaller than the whole data set, the amount of calculation is not very large.

It should be noted that the step of each traditional iteration is different because of the size of mini batch, so it is impossible to draw different mini batches in a picture, so we adopt different batchsize.

The results according to the figures below are: the larger the minibatch size is, the more obvious the decline effect is and the smoother the decline curve is. However, it can be seen that if the number of Epoch on the whole data set is considered, the smaller the minibatch size number is, the faster the convergence will be.

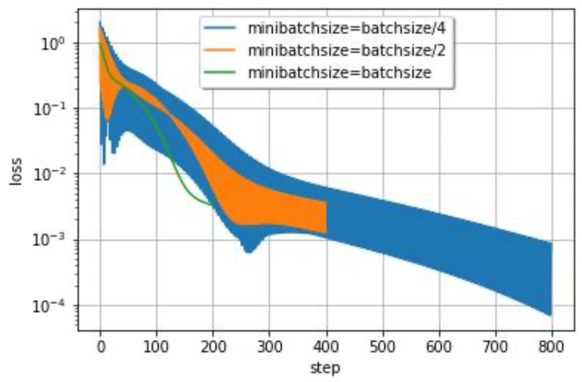

Fig. 6. Different minibatchsize under each step.

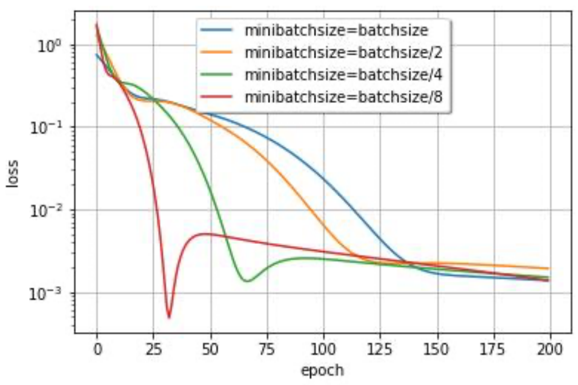

Fig. 7. Different minibatchsize under each epoch.

C. *Adjustable parameters*

There are a host of hyperparameters in the neural networks model which can be modified to get a better convergent results, such as neurons in hidden layer, batchsize which stands for how many points to use per epoch and learning rate. After several trials, different learning rates show a more distinct difference. Additionally, the activation function also plays an effective way in influencing the convergence of the result.

From the figures below, we can conclude that when learning rate equals 0.035 is the most suitable for Adam to converge, and tanh(x) is the most suitable activation function.

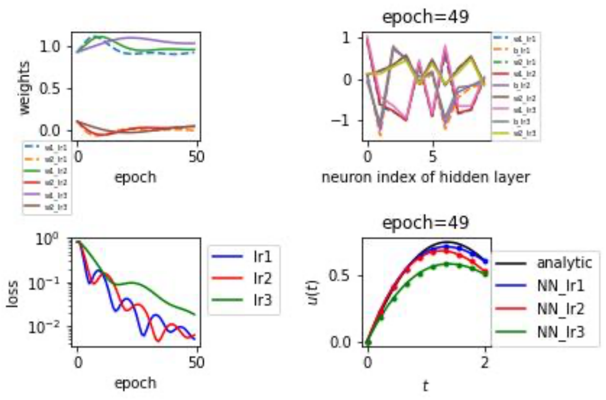

Fig. 7.lr1=0.035, lr2=0.025, lr3=0.01.

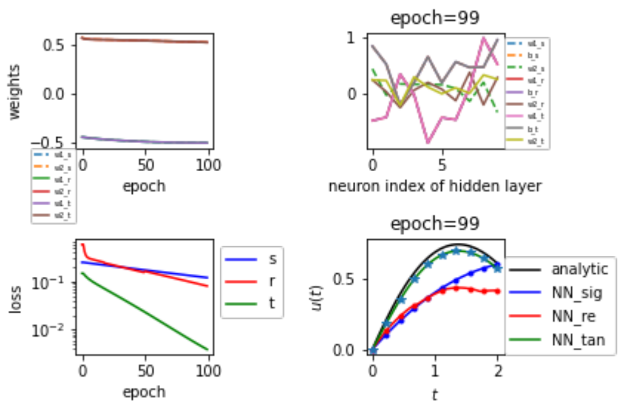

Fig. 8. Section of activation functions.

IV. Innovation of the Second Order Differentiation Equations

A. *Modification of the second order ODEs in the first method*

Firstly, we consider the second order ODE as $\frac{d^2u}{dt^2} + u = 0$, with t∈ [0,1], and the initial conditions are u(0) = u_0 = 1, $u'(0) = 0$.

The key part is that we convert the second order ODE into the first order ODE by setting v(t)=u$'$(t), then we will get:

$$\frac{dv}{dt} = -u(t) = f(t,v),\ v(0) = v_0 = 0; \qquad (7)$$

$$\frac{du}{dt} = v(t) = g(t,u),\ u(0) = u_0 = 1. \qquad (8)$$

We can write them together into matrix form:

$$\overrightarrow{uv}' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \overrightarrow{uv} = f(\overrightarrow{uv}),\ \overrightarrow{uv}\backslash_0 = (u_0\ v_0), \quad (9)$$

and the solutions from our neural networks can be expressed by:

$$\overrightarrow{uv_{NN}} = \overrightarrow{uv}\backslash_0 + tNN(t), \qquad (10)$$

where NN(t) is a neural network. The loss function can be written as:

$$L = [\overrightarrow{uv_{NN}}' - \vec{f}(\overrightarrow{uv_{NN}})]^2. \qquad (11)$$

The analytic solution could be written as:

$$uA=\cos(t),\ vA=-\sin(t). \qquad (12)$$

After setting all the parameters and formula already, we could plot the figure blow, which shows a relatively satisfying convergence of the constructed u(t) and v(t). While the disadvantage is that the given domain is too short to convince the convergence in larger tspan, which could be better improved in the future. When we expand the domain of the ODEs, the convergence becomes worse and worse actually, which could also related to the oscillation of the solutions.

Below, we set the epoch to be 500 in order to get a relatively strong convergence.

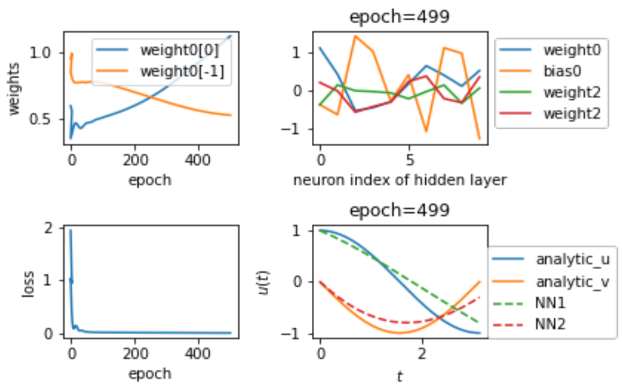

Fig. 9. Convergence of the second order ODE of method1.

B. *Analogy of the second order ODEs in the second method*

If we directly solve the second order differential equation:

$$U_{NN} = u_0 + v_0 t + t^2 NN(t), \quad (13)$$
$$u(0) = u_0 = 1, v(0) = u'(0) = 0, \quad (14)$$

thus, the loss function is:

$$L = [u_{NN}'' - u_{NN}]^2. \quad (15)$$

Then we can plot the result figure as below. We can find that epoch equal 200 is already enough to see the strong convergence comparing to the method 1. However, the domain is still not considerable enough.

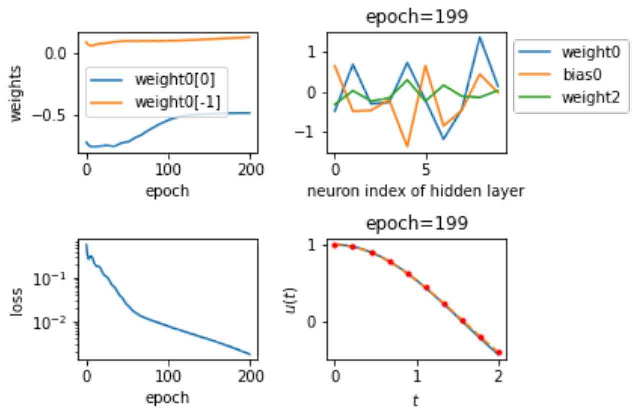

Fig. 10. Convergence of the second order ODE of method 2.

CONCLUSION

In summary, we present a method to solve the ordinary differential equations by using neural networks, which not only gives a relatively high accuracy in solution, but also in a quick speed, showing a great capacity in approximating the solutions of ODEs. In our first order ODEs, ANNs work relatively well considering the small gap between analytic solutions. Additionally, through modifying adjustable parameters, the most convergent situation could be settled. However, this method still has some limitations when solving the second order ODEs, which typically lie in the large range of t, leading to a much larger gap between the analytic solutions. Presumably because the neural works are sensitive to the range of input data. Such situation could be even worse when facing oscillating solutions like cosx or sinx. In the future, we intend to investigate further aspects of neural network, including the number of layers and hidden nodes, and the connection between layers, to determine whether there are any better options for various types of ODEs. We would also pay attention to the optimization component, which includes the effectiveness and stability of various approaches.

REFERENCES

[1] D. Kincaid, and W. Cheney, Numerical Analysis, Brooks/Cole Publishing Company, 1991.

[2] H. Lee, and I. Kang, Neural algorithms for solving differential equations, Journal of Computational Physics, vol. 91, pp. 110-117, 1990.

[3] A.J. Meade Jr, and A.A. Fernadez, The numerical solution of Linear Ordinary Differential Equations by Feedforward Neural networks, Math. Comput. Modelling, vol. 19, no. 12, pp. 1-25, 1994.

[4] M. Abadi, et al, TensorFlow: a system for large-scale machine learning, pp. 265–283, USENIX Association, USA, 2016.

[5] M.T. Hagan, M.B. Menhaj, Training feedforward networks with the Marquardt algorithm, 5(6), pp. 989–993, 1994.

[6] A.J. Meade Jr, and A.A. Fernadez, The numerical solution of Linear Ordinary Differential Equations by Feedforward Neural networks, Math. Comput. Modelling, vol. 20, no. 9, pp. 19-66, 1994.

[7] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, 2(5), pp. 359–366, 1989.

[8] B. Bishnoi, Lagrangian Density Space-Time Deep Neural Network Topology, 2022.

[9] I. E. Lagaris, A. Likas, and D. I. Fotiadis, Artificial Neural Networks for Solving Ordinary and Partial Differential Equations, 1997.

[10] C. Galeriu, An Arduino Investigation of Simple Harmonic Motion, 2014.

[11] L. Wu, S. Liu, and Y. Wang, Grey Lotka–Volterra model and its application,vol.79, no. 9, pp. 1720-1730, 2012.