Machine Learning with Applications 5 (2021) 100058

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

Deep neural network for system of ordinary differential equations: Vectorized algorithm and simulation

Tamirat Temesgen Dufera

Adama Science and Technology University, Adama, Ethiopia

ARTICLE INFO

Keywords:
Deep Neural Network
Algorithm
Systems of ordinary differential equation

ABSTRACT

This paper is aimed at applying deep artificial neural networks for solving system of ordinary differential equations. We developed a vectorized algorithm and implemented using python code. We conducted different experiments for selecting better neural architecture. For the learning of the neural network, we utilized the adaptive moment minimization method. Finally, we compare the method with one of the traditional numerical methods-Runge–Kutta order four. We have shown that, the artificial neural network could provide better accuracy for smaller numbers of grid points.

1. Introduction

Deep neural network (DNN) has obtained great attention for solving engineering problems. System of ordinary differential equations (ODEs) that can model various physical phenomena could utilize the advantages of using the method. Though there are well established traditional numerical methods for solving systems of ODEs, they have their own advantages and disadvantages in-terms of accuracy, stability, convergence, computation time etc. One of the well known method is the fourth order Runge–Kutta method (RK4). It is among the finite difference methods well suited for non-stiff problems.

Artificial neural network (ANN) is an alternative method known to the scientific community since 1940s. The beginning of ANN is often attributed to the research article by McCulloch and Pitts (1943). It was less popular due to the capacity of computational machines. The recent development and progresses in the area is attributed to the exponential improvement in the computing capacity of machines both in storing data and processing speed (Basheer & Hajmeer, 2000). "An artificial neural network is an information-processing system that has certain performance characteristics in common with biological neural networks" (Yadav, Yadav, Kumar, et al., 2015). The network imitates the work of biological human brain (Basheer & Hajmeer, 2000). The structure of the architecture constitutes layers: input, hidden and output. Each layer have neurons or units. The name deep neural network (DNN) is used when the structure has more than one hidden layers (Goodfellow, Bengio, Courville, & Bengio, 2016; Schmidhuber, 2015).

2. Related works

2.1. Deep neural networks

The DNN method has contributed a lot to the progress in artificial intelligence specifically in computer vision, image processing, pattern recognition and Cybersecurity (Dixit & Silakari, 2021; Dong, Wang, & Abbas, 2021; Minaee et al., 2021). The performance is due to features are learned rather than hand-crafted, the deep layers are able to capture more variances (Bruna & Dec, 2018).

Some of the challenging issues related to DNN are, stability, robustness, provability and adversarial perturbation which are discussed in Haber and Ruthotto (2017), Malladi and Sharapov (2018), Szegedy et al. (2013), Zheng and Hong (2018) and Zheng, Song, Leung, and Goodfellow (2016). The optimization problems arising form learning the DNN also need special consideration which are presented in Nouiehed and Razaviyayn (2018) and Yun, Sra, and Jadbabaie (2018).

Moreover, the search and selection of an optimal neural architecture is difficult task (Elsken, Metzen, Hutter, et al., 2019). Some widely implemented deep learning architectures — autoencoder, convolutional network, deep belief network and restricted Boltzmann machine were presented in Liu et al. (2017). A broader survey of advance in convolutional neural network can be found in Gu et al. (2018). More related works and recent advances in application of deep neural networks such as in Cybersecurity, image segmentation, background subtraction and self-supervised image recognition, are presented in Bouwmans, Javed, Sultana, and Jung (2019), Dixit and Silakari (2021), Dong et al. (2021),

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

E-mail address: tamirat.temesgen@astu.edu.et.

https://doi.org/10.1016/j.mlwa.2021.100058
Received 27 April 2021; Received in revised form 1 June 2021; Accepted 2 June 2021
Available online 7 June 2021

Minaee et al. (2021), Ohri and Kumar (2021) and Yi, Shiyu, Xiusheng, and Zhigang (2016).

2.2. Works related to solving differential equations

Nowadays, researchers are applying ANN for solving differential equations. Some of the advantages of using ANN over the traditional numerical methods are: the solutions obtained by ANN are differentiable, and closed analytic form, the method could handle complex differential equations and helps to overcome the repetition of iteration (Chakraverty & Mall, 2017).

Lee and Kang (1990) implemented neural algorithm for solving differential equations. They have used the method for minimization purpose where development of highly parallel algorithms for solving the difference equations required.

Meade and Fernandez (1994) implemented a feedforward neural networks to approximate the solution of linear ODE. They have used the hard limit activation function to construct direct and non-iterative feedforward neural network. The author implemented the method on three layers, input layer, a hidden layer and output layer. Simple first and second order ordinary differential equation were considered for testing the method.

Lagaris, Likas, and Fotiadis (1998) used ANN for solving ordinary and partial differential equations. For solving initial and boundary value problems, they used trail solution satisfying the given conditions. Then, network were trained to satisfy the differential equations. The results were compared with well established numerical method — finite element. The authors obtained accurate and differentiable solution in a closed analytic form.

Partial differential equation with initial and boundary condition were solved using neural network (Aarts & Van Der Veer, 2001). The architecture of the network were, multiple input units, single output unit and single hidden layer feedforward with a linear output layer with no bias. Evolutionary algorithms were implemented for the cost minimization. The authors tested the method on problems from physics and geological process.

For solving ODE using ANN, unsupervised kernel mean square algorithm were used in Sadoghi Yazdi, Pakdaman, and Modaghegh (2011). Trial solution similar to the authors in Lagaris et al. (1998), were implemented to obtain accurate results. Nascimento, Fricke, and Viana (2020) presented the direct implementation of integration of ODE through recurrent neural networks.

Berg and Nyström (2018) implemented deep feedforward ANN to approximate solution of partial differential equations in complex geometries. They solved problems that could not be addressed or difficult by the traditional method. They did comparison between shallow versus deep networks. More recent development and applications of ANN in partial differential equations can be found in Berg and Nyström (2019), Rackauckas et al. (2020), Raissi, Perdikaris, and Karniadakis (2019) and Wang, Huan, and Garikipati (2019).

The application of ANN were also extended to the computation of integral equations. Asady, Hakimzadegan, and Nazarlue (2014), introduced an efficient application of ANN for approximating solution of linear two-dimensional Fredholm integral equation of the second kind. They have found remarkable accuracy and proposed extension to the case of more general integral equations.

For the implementation of ANN, clear and reproducible algorithm with implementation needs great attention. An efficient neural network architecture has to be investigated corresponding to systems of ODE. We need to look at the effects of numbers of hidden layers in the architect as well as the numbers of neurons in the layers on accuracy, speed and performance of the model in general. We need to investigate and propose the best selection of activation function. Addressing the issue of minimization method for the cost function is also crucial.

In this paper, we present a vectorized algorithm for solving systems of ordinary differential equation using DNN. We implement the algorithm in python and perform experimental simulations to look at the effects of different neural architecture on the performance of the model. Moreover, we observe the advantage of using the ANN over the traditional methods. Specifically, we consider the fourth order Runge–Kutta finite difference method.

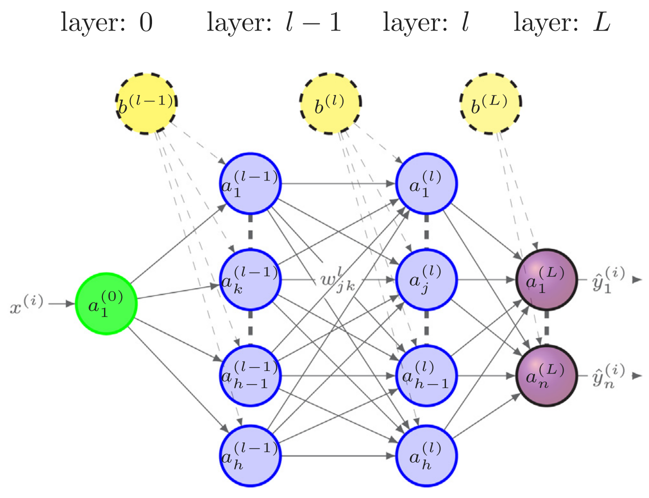

Fig. 1. The schematic diagram of deep ANN.

The paper is structured as follows: first we remind our reader the general formulation of systems of ordinary differential equation. Next we setup the general form of DNN and its application in the area of differential equations. Moreover, we perform different experiment using python code. At the end we implement the algorithm and compare the result with the analytical solution and with numerical solution obtained using Runge–Kutta method.

3. Systems of ordinary differential equation

The general form of a system of n ODEs is given by,

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(t, y_1, y_2, \dots, y_n) \\ \frac{dy_2}{dt} &= f_2(t, y_1, y_2, \dots, y_n) \\ \vdots \quad & \quad \vdots \\ \frac{dy_n}{dt} &= f_n(t, y_1, y_2, \dots, y_n), \end{aligned} \tag{1}$$

defined on $a < t < b$ with given initial values, $y_1(a) = a_1, \dots, y_n(a) = a_n$. The initial value problem (1) can be written in compact way as follows;

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{2}$$

where $\mathbf{y} = \left[y_1, y_2, \dots, y_n\right]^T$ is the unknown having dimension of $n\times 1$, and

$$\mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} f_1(t, y_1, y_2, \dots, y_n) \\ f_2(t, y_1, y_2, \dots, y_n) \\ \vdots \\ f_n(t, y_1, y_2, \dots, y_n) \end{bmatrix}$$

is given vector valued function having dimension of $n \times 1$. The uniqueness and existence of the solutions to the initial value problem is well established theory. To obtain more understanding on existence and uniqueness of solution to the initial value problem (2), one may refer Coddington and Levinson (1955).

4. Deep neural network for system of ODEs

We consider a dense network of L layers indicated in Fig. 1. The network contains one neuron in the input layer corresponding to the independent variable for the system of ODEs. The output layer contains n neurons corresponding to the unknown variables. For training the DNN, we take m sample points from the domain $[a,b]$, and form a matrix $X = [t^{(1)},\dots,t^{(m)}] \in \mathbb{R}^{1\times m}$. Here $t^{(i)} \in [a,b] \subset \mathbb{R}$ is the i^{th} sample point or training example. Moreover, we denote by $\mathbf{N} \in \mathbb{R}^{n\times m}$ the output matrix. For the example, $N_k(t^{(i)},P_k)$ is the output of the k^{th} unknown corresponding to the i^{th} sample point, where P_k stands for the corresponding parameters, the weights and the bias. Following the references, see e.g., Lagaris et al. (1998) and Malek and Shekari Beidokhti (2006), for each $t \in [a,b]$ we set a trial solution given by,

$$\hat{y}_j(t,P_j) = a_j + (t-a)N_j(t,P_j), \quad j = 1,\dots,n. \tag{3}$$

The trial solution in Eq. (3), satisfies the initial conditions. We train the network in such a way that the total cost function given by,

$$J = \sum_{i=1}^{m}\sum_{j=1}^{n}\left(\frac{d\hat{y}_j}{dt} - f_j\right)^2, \tag{4}$$

converges to zero. Here, $f_j = f_j(t^{(i)},\hat{y}_j(t^{(i)},P_j))$. Note that, the learning process or the training is unsupervised as there are no targeted solutions.

4.1. Forward propagation

We label nodes on layer $l-1$ by k and nodes on layer l by j. Then, the following value goes into the j th node of layer l,

$$z_j^l = \sum_{k=1}^{h} w_{jk}^l a_k^{l-1} + b_j^l, \tag{5}$$

where h denotes the number of nodes in layer l. The *matrix* form of Eq. (5) is;

$$\mathbf{z}^l = W^l\mathbf{a}^{l-1} + \mathbf{b}^l, \tag{6}$$

were the matrix W^l containing all the multiplicative parameters, i.e., the weights w_{jk}^l, and $\mathbf{b}^l$ is the bias. The values in (5) will pass to the next hidden layer by an appropriate choice of an activation functions, denoted by σ^l. In this study, we choose the same activation function for all nodes in a layer. Thus, the values for the next layer is expressed by,

$$\mathbf{a}^l = \sigma^l(\mathbf{z}^l) = \sigma^l(W^l\mathbf{a}^{l-1} + \mathbf{b}^l). \tag{7}$$

For the m grid points, $i = 1,2,\dots,m$, we follow the following steps. At the first layer,

$$\mathbf{z}^{1(i)} = W^1\mathbf{x}^{(i)} + \mathbf{b}^1,$$
$$\mathbf{a}^{1(i)} = \sigma^1(\mathbf{z}^{1(i)}),$$

at the second layer,

$$\mathbf{z}^{2(i)} = W^2\mathbf{a}^{1(i)} + \mathbf{b}^2,$$
$$\mathbf{a}^{2(i)} = \sigma^2(\mathbf{z}^{2(i)}),$$

continue this till the output layer,

$$\mathbf{z}^{L(i)} = W^L\mathbf{a}^{L-1(i)} + \mathbf{b}^L,$$
$$\mathbf{a}^{L(i)} = \sigma^L(\mathbf{z}^{L(i)}).$$

Now stacking each examples into a matrix X as column vector, and similarly the $\mathbf{z}^{1(i)}$'s and the $\mathbf{a}^{1(i)}$'s in to the matrix Z^1 and A^1 respectively, we have the following *matrix* form,

$$Z^1 = W^1X + \mathbf{b}^1,$$
$$A^1 = \sigma^L(Z^1),$$

the second layer

$$Z^2 = W^2A^1 + \mathbf{b}^2,$$
$$A^2 = \sigma^2(Z^2),$$

continue till the output layer L,

$$Z^L = W^LA^{L-1} + \mathbf{b}^L,$$
$$A^L = \sigma^L(Z^L).$$

4.2. The vectorized algorithm

Here we describe algorithm of DNN method for solving system of ODE:

1. *Input data*: Take m discrete points from the domain $[a,b]$ and form a vector $X = [t^{(1)},t^{(2)},\dots,t^{(m)}]$ of size $1\times m$.
2. *Define the neural network structure*: Here we determine the number of layers L, input layer having one units, $L-2$ hidden layer having h^l units, for each $1 \le l \le L-1$ and the output layer having n units which is equal to the number of unknown in the system.
3. *Initialize the parameters*, P_j, $j = 1,\dots,n$ and $2 \le l \le L-1$:
   - W^1 has $h^1 \times 1$ dimension,
   - W^l has $h^l \times h^{l-1}$ dimension,
   - W^L has $n \times h^{L-1}$ dimension,
   - $\mathbf{b}^1$ has $h^1 \times 1$ dimension,
   - $\mathbf{b}^l$ has $h^l \times 1$ dimension, and
   - $\mathbf{b}^L$ has $n \times 1$ dimension.
4. Forward propagation:
   - For the input layer start by assigning, $A^0 = X$.
   - For the hidden layers, $1 \le l \le L-1$,

     $$Z^l = W^lA^{l-1} + \mathbf{b}^l,$$
     $$A^l = \sigma^l(Z^l),$$

     where, σ^l is the activation function corresponding to the l^{th} hidden layer.
   - For the output layer,

     $$Z^L = W^LA^{L-1} + \mathbf{b}^L,$$
     $$A^L = \sigma^L(Z^L),$$
     $$\mathbf{N}(X,P_j) = A^L.$$

   - Assign the trial solution using Eq. (3): To arrive at the trial solution of an unknown function, we need to initialize a corresponding sets of parameters.
5. Compute the cost and its gradient, using Eq. (4): Calculate gradients with respect of X and with respect to the learning parameters. Here we implement the automatic differentiation (Baydin, Pearlmutter, Radul, & Siskind, 2018; Bradbury et al., 2018).
6. Update the parameter using the method of gradient decent or any other best optimization method. One of the widely used is the gradient decent. We randomly initialize the parameters and update according to the following rule; for $j = 1,2,$

   $$P_j^{k+1} = P_j^k - \eta\nabla J(P_j^k),$$

   where η is the learning rate and k corresponds to iteration.

Note that, in addition to the simple gradient decent method, currently there are more advanced optimization tools and still is an active research topics (Calin, 2020).

The *moment method* is the modification of gradient decent method designed to avoid getting stuck in a local minimum. The updating rule is as follows;

$$P_j^{k+1} = P_j^k + V_j^{k+1},$$
$$V_j^{k+1} = \mu V_j^k - \eta\nabla J(P^k),$$

where $\eta > 0$ is the learning rate and μ is a coefficient between 0 and 1 called the momentum. Here k indicates iteration and V_j is a new parameter (velocity) initialized from zero corresponding to each unknown.

The *Nesterov accelerated Gradient (NAG)* is obtained by modifying the momentum method and the update rule is given as follow,

$$P_j^{k+1} = P_j^k + V_j^{k+1},$$
$$V^{k+1} = \mu V_j^k - \eta \nabla J(P^k + \mu V_j^k).$$

The main difference from the moment method is that, the argument of the gradient is computed at the correlated value $P^k + \mu V_j^k$ instead of computing it at the current position P_j^k.

The *AdaGrad*, Adaptive Gradient: the update rule have the form;

$$V_j^k = V_j^{k-1} + (\nabla J(P_j^k))^2$$
$$P_j^{k+1} = P_j^k - \frac{\eta}{\sqrt{V_j^k + \epsilon}} \nabla J(P_j^k),$$

where ϵ is small number to avoid division by zero. The method changes the learning rate for the parameters in proportional to the update history. It decays the learning rate.

The *Root Mean Square Propagation*, or RMSProp is family of the gradient decent method having adaptive learning rate, again following Calin (2020), our update rule is as follows;

$$V_j^k = \beta V_j^{k-1} + (1-\beta)(\nabla J(P_j^k))^2$$
$$P_j^{k+1} = P_j^k - \frac{\eta}{\sqrt{V_j^k + \epsilon}} \nabla J(P_j^k),$$

where $\beta \in (0,1)$ is the forgetting factor.

Adam, Adaptive Moment, is also an adaptive learning method which combines AdaGrad and RMSProp methods. In our case the updating rule have the form;

$$M_j^k = \beta_1 M_j^{k-1} + (1-\beta_1)\nabla J(P_j^k),$$
$$V_j^k = \beta_2 V_j^{k-1} + (1-\beta_2)(\nabla J(P_j^k))^2,$$
$$\hat{M}_j^k = \frac{M_k}{1-\beta_1^k}, \qquad \hat{V}_j^k = \frac{V_k}{1-\beta_2^k},$$
$$P_j^{k+1} = P_j^k - \frac{\eta}{\sqrt{\hat{V}_j^k + \epsilon}} \hat{M}_j^k,$$

where $\beta_1, \beta_2 \in [0,1)$, are decay rates for the moment estimates, and we initialize the parameters V_j and M_j to be zero.

5. Implementation and comparison

In this section we implement the algorithm for solving a known nonlinear systems of differential equations. First, we perform simulation for selecting appropriate number of layers and neurons in the layer. Then, we compare with the analytical solution and with a numerical solution obtained using the traditional methods. For this purpose, we consider the following problem found in Lagaris et al. (1998),

$$\frac{dy_1}{dt} = \cos(t) + y_1^2 + y_2 - (1 + t^2 + \sin^2(t)), \tag{8}$$
$$\frac{dy_2}{dt} = 2t - (1+t^2)\sin(t) + y_1 y_2,$$

with $t \in [0,1]$ and $y_1(0) = 0$ and $y_2(0) = 1$. The analytic solutions are $y_1 = \sin(t)$ and $y_2 = 1 + t^2$.

5.1. Experiment on the network

We conducted an experiment on number of neurons in a layer. We looked at the effect of number of neurons on the error function. We took different sizes of neurons in the hidden layer, $h = 4, 17, 60, 150, 200$, and we plotted the cost function versus the number of iterations for the comparison of convergence. In the simulation, we displayed the cost at the end of iterations corresponding to each neuron size and the time it take for the calculation. All other parameters are the same.

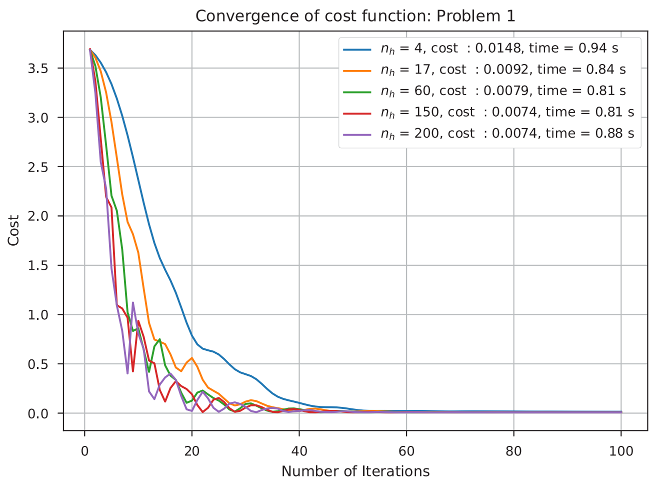

Fig. 2. Convergence of loss functions for system (8).

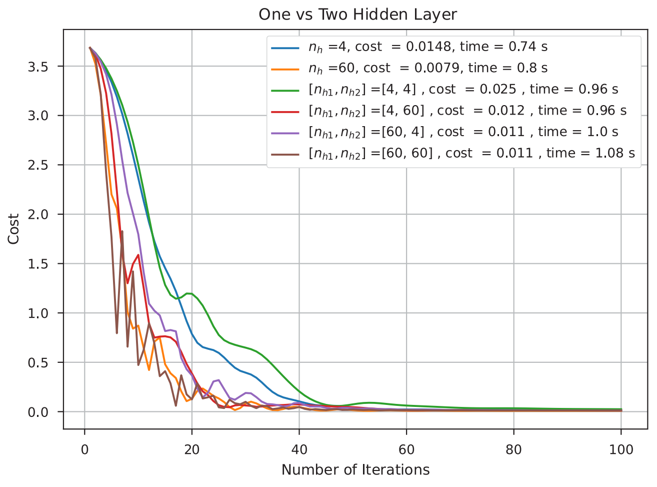

Fig. 3. Convergence of error functions for problem (8), two hidden layers.

From the simulation shown Fig. 2, we observe that, one can obtain the required accuracy even for a single neuron in the hidden layer. However, it needs large number of iterations for smaller number of neurons leading to problem of computational time. Increasing the numbers of neuron has advantage on the performance of the model. However, an arbitrary increase is unnecessary. In this case $h = 60$ has similar accuracy with $h = 200$ with less computational time.

The next experiment is on the number of hidden layers. Raissi et al. (2019), have shown that for Burgers' equation, more hidden layer results in better performance as far as error is concerned. Also, Berg and Nyström (2018), observed the improvement of accuracy of solving diffusion equation. In our case, fixing all parameters and activation functions the same as the previous experiments, we performed a simulation to compare one hidden layer and two hidden layers varying the numbers of neurons.

In Fig. 3, the result shows that, for the system of differential equation (8), adding more hidden layer do not lead to better performance.

5.2. Numerical solutions

Now we use the ANN method for solving the system of differential equations. In line with the above simulations, we selected a single hidden layer with 60 neurons (tuning in plus minus may not have significant effect). For this experiment, $m = 11$, uniform grid points were sampled from the given interval. The solutions using ANN and

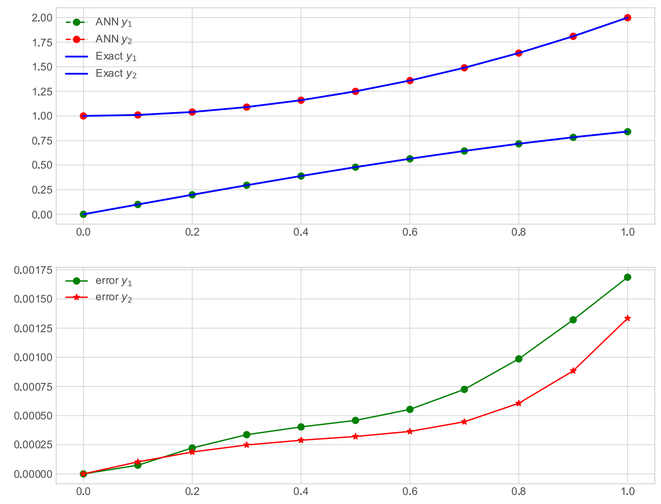

Fig. 4. Comparing the ANN solution of (8), with the exact solution and error plot.

the corresponding analytical solutions are indicated in Fig. 4. The numerical quantities are indicated in Table 1.

Table 2 indicates the error due to the neural network method.

5.3. Advantage of ANN over the RK4 methods

We selected different sizes of uniform grid points, $m = 11, 16, 21$ and 26 from the domain $[0, 4]$, and computed solutions of the system of ODEs using the two methods. The simulation in Fig. 5 shows one of the significant advantage of using neural network method over other traditional method — finite difference. ANN gives better performance for smaller grid pints. Also, observe that, at the end point $t = 4$, the ANN is more accurate than the Runge–Kutta method 3. This show that, the method could be employed for application problem requiring large data points. However, for larger grid points, RK4 is more accurate as expected.

6. Conclusions and outlook

In this paper, we presented a vectorized algorithm for solving systems of ODE using DNN. We conducted different experiment using python code and simulated the result using graphs. We have obtained some insight on the nature of the architecture for the model. We have seen that for some specific problems we can obtain a required accuracy even for a single neuron in the hidden layer. More neuron size provides more accuracy, but more iteration for learning the parameters.

Table 1
ANN and analytical solutions.

t	y_1 ANN	y_1 Analytic	y_2 ANN	y_2 Analytic
0.0	0.000000	0.000000	1.000000	1.00
0.1	0.099759	0.099833	1.009897	1.01
0.2	0.198447	0.198669	1.039812	1.04
0.3	0.295184	0.295520	1.089752	1.09
0.4	0.389015	0.389418	1.159711	1.16
0.5	0.478967	0.479426	1.249679	1.25
0.6	0.564089	0.564642	1.359636	1.36
0.7	0.643493	0.644218	1.489553	1.49
0.8	0.716370	0.717356	1.639394	1.64
0.9	0.782005	0.783327	1.809116	1.81
1.0	0.839784	0.841471	1.998666	2.00

Table 2
ANN Error.

t	error y_1	error y_2
0.0	0.000000	0.000000
0.1	0.000075	0.000103
0.2	0.000223	0.000188
0.3	0.000336	0.000248
0.4	0.000403	0.000289
0.5	0.000459	0.000321
0.6	0.000553	0.000364
0.7	0.000725	0.000447
0.8	0.000987	0.000606
0.9	0.001321	0.000884
1.0	0.001687	0.001334

Table 3
Error at the end point $t = 4$, RK4 and ANN compared for different grid points.

Grid points	ANN error	RK4 error
11	0.982	9.588
16	0.959	4.668
21	1.005	1.902
26	0.990	0.825

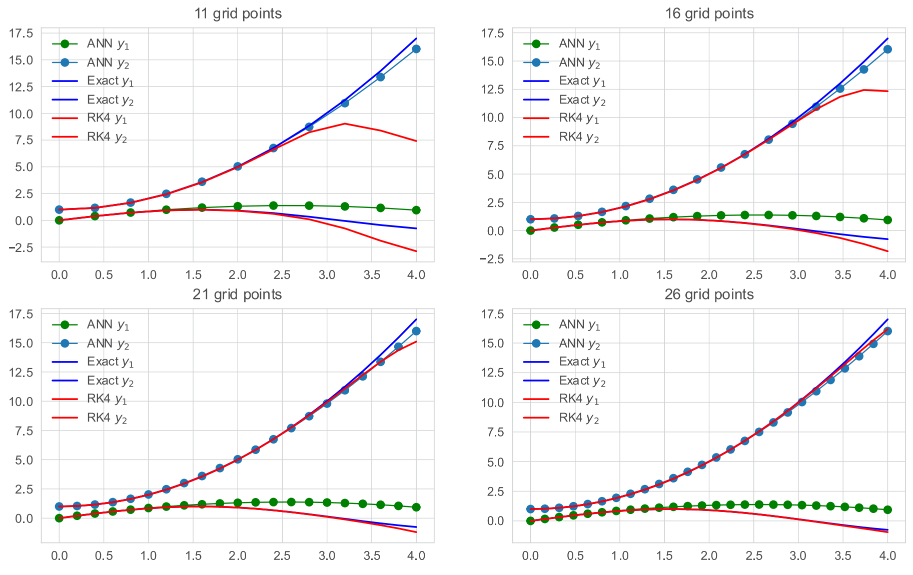

Fig. 5. Comparing the ANN solution of (8), with the exact and RK4 solutions for different grid points.

Moreover, arbitrary increase of neurons is not recommended. Based on the underlying problem one has to set for the best size of neurons.

We compared the ANN method with the well known fourth order Runge–Kutta method. The result showed that, the ANN produced more accurate result for small number of the grid points. Moreover, for larger value of the domain, the ANN method provides better accuracy than RK4 method.

For a future work, further analytical investigation is required to strength the foundation of DNN for solving system of ODE including delay differential equations and stochastic differential equations. These include looking at stability, convergence and robustness of DNN related to solving system of ODEs. In the same way looking at using other architectures such as, recurrent neural networks, constitutional neural network, deep probabilistic neural network, general adversarial networks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aarts, L. P., & Van Der Veer, P. (2001). Neural network method for solving partial differential equations. *Neural Processing Letters*, *14*(3), 261–271.

Asady, B., Hakimzadegan, F., & Nazarlue, R. (2014). Utilizing artificial neural network approach for solving two-dimensional integral equations. *Mathematical Sciences*, *8*(1), 117.

Basheer, I., & Hajmeer, M. (2000). Artificial neural networks: Fundamentals, computing, design, and application. *Journal of Microbiological Methods*, *43*(1), 3–31. http://dx.doi.org/10.1016/S0167-7012(00)00201-3, URL: https://www.sciencedirect.com/science/article/pii/S0167701200002013, Neural Computting in Micrbiology.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, *18*.

Berg, J., & Nyström, K. (2018). A unified deep artificial neural network approach to partial differential equations in complex geometries. *Neurocomputing*, *317*, 28–41. http://dx.doi.org/10.1016/j.neucom.2018.06.056.

Berg, J., & Nyström, K. (2019). Data-driven discovery of PDEs in complex datasets. *Journal of Computational Physics*, *384*, 239–252. http://dx.doi.org/10.1016/j.jcp.2019.01.036.

Bouwmans, T., Javed, S., Sultana, M., & Jung, S. K. (2019). Deep neural network concepts for background subtraction: A systematic review and comparative evaluation. *Neural Networks*, *117*, 8–66.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., et al. (2018). JAX: Composable transformations of Python+NumPy programs. URL: http://github.com/google/jax.

Bruna, J., & Dec, L. (2018). *Mathematics of deep learning*. NYU: Courant Institute of Mathematical Science.

Calin, O. (2020). *Deep learning architectures*. Springer.

Chakraverty, S., & Mall, S. (2017). *Artificial neural networks for engineers and scientists: Solving ordinary differential equations*. CRC Press.

Coddington, E. A., & Levinson, N. (1955). *Theory of ordinary differential equations*. Tata McGraw-Hill Education.

Dixit, P., & Silakari, S. (2021). Deep learning algorithms for cybersecurity applications: A technological and status review. *Computer Science Review*, *39*, Article 100317.

Dong, S., Wang, P., & Abbas, K. (2021). A survey on deep learning and its applications. *Computer Science Review*, *40*, Article 100379.

Elsken, T., Metzen, J. H., Hutter, F., et al. (2019). Neural architecture search: A survey. *Journal of Machine Learning Research*, *20*(55), 1–21.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning (vol. 1)*. MIT Press Cambridge.

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., et al. (2018). Recent advances in convolutional neural networks. *Pattern Recognition*, *77*, 354–377.

Haber, E., & Ruthotto, L. (2017). Stable architectures for deep neural networks. *Inverse Problems*, *34*(1), Article 014004.

Lagaris, I. E., Likas, A., & Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, *9*(5), 987–1000.

Lee, H., & Kang, I. S. (1990). Neural algorithm for solving differential equations. *Journal of Computational Physics*, *91*, 110–131.

Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F. E. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing*, *234*, 11–26.

Malek, A., & Shekari Beidokhti, R. (2006). Numerical solution for high order differential equations using a hybrid neural network—Optimization method. *Applied Mathematics and Computation*, *183*(1), 260–271. http://dx.doi.org/10.1016/j.amc.2006.05.068.

Malladi, S., & Sharapov, I. (2018). FastNorm: Improving numerical stability of deep network training with efficient normalization.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*, 115–133.

Meade, A., & Fernandez, A. (1994). The numerical solution of linear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling*, *19*, 1–25. http://dx.doi.org/10.1016/0895-7177(94)90095-7.

Minaee, S., Boykov, Y. Y., Porikli, F., Plaza, A. J., Kehtarnavaz, N., & Terzopoulos, D. (2021). Image segmentation using deep learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Nascimento, R. G., Fricke, K., & Viana, F. A. (2020). A tutorial on solving ordinary differential equations using Python and hybrid physics-informed neural network. *Engineering Applications of Artificial Intelligence*, *96*, Article 103996. http://dx.doi.org/10.1016/j.engappai.2020.103996.

Nouiehed, M., & Razaviyayn, M. (2018). Learning deep models: Critical points and local openness. arxiv preprint arxiv:1803.02968.

Ohri, K., & Kumar, M. (2021). Review on self-supervised image recognition using deep neural networks. *Knowledge-Based Systems*, Article 107090.

Rackauckas, C., Ma, Y., Martensen, J., Warner, C., Zubov, K., Supekar, R., et al. (2020). Universal differential equations for scientific machine learning. arxiv preprint arxiv:2001.04385.

Raissi, M., Perdikaris, P., & Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. http://dx.doi.org/10.1016/j.jcp.2018.10.045.

Sadoghi Yazdi, H., Pakdaman, M., & Modaghegh, H. (2011). Unsupervised kernel least mean square algorithm for solving ordinary differential equations. *Neurocomputing*, *74*(12), 2062–2071. http://dx.doi.org/10.1016/j.neucom.2010.12.026.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117. http://dx.doi.org/10.1016/j.neunet.2014.09.003, URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., et al. (2013). Intriguing properties of neural networks. arxiv preprint arxiv:1312.6199.

Wang, Z., Huan, X., & Garikipati, K. (2019). Variational system identification of the partial differential equations governing the physics of pattern-formation: Inference under varying fidelity and noise. *Computer Methods in Applied Mechanics and Engineering*, *356*, 44–74. http://dx.doi.org/10.1016/j.cma.2019.07.007.

Yadav, N., Yadav, A., Kumar, M., et al. (2015). *An introduction to neural network methods for differential equations*. Springer.

Yi, H., Shiyu, S., Xiusheng, D., & Zhigang, C. (2016). A study on deep neural networks framework. In *2016 IEEE advanced information management, communicates, electronic and automation control conference* (pp. 1519–1522). IEEE.

Yun, C., Sra, S., & Jadbabaie, A. (2018). A critical view of global optimality in deep learning. arxiv preprint arxiv:1802.03487.

Zheng, Z., & Hong, P. (2018). Robust detection of adversarial attacks by modeling the intrinsic properties of deep neural networks. In *Proceedings of the 32nd international conference on neural information processing systems* (pp. 7924–7933).

Zheng, S., Song, Y., Leung, T., & Goodfellow, I. (2016). Improving the robustness of deep neural networks via stability training. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4480–4488).