ELKHA : *Jurnal Teknik Elektro, Vol. 13 No.2, October 2021, pp. 134 - 140*
ISSN: 1858-1463 (print), 2580-6807 (online)

Comparison and Analysis of Neural Solver Methods for Differential Equations in Physical Systems

Fabio M Sim[1)], Eka Budiarto[2*)], and Rusman Rusyadi[3)]

[1, 3)] Department of Mechanical Engineering – Mechatronics Concentration, Swiss German University, Indonesia
[2)] Master of Information Technology, Swiss German University, Indonesia
Corresponding Email: [*)] eka.budiarto@sgu.ac.id

Abstract – **Differential equations are ubiquitous in many fields of study, yet not all equations, whether ordinary or partial, can be solved analytically. Traditional numerical methods such as time-stepping schemes have been devised to approximate these solutions. With the advent of modern deep learning, neural networks have become a viable alternative to traditional numerical methods. By reformulating the problem as an optimisation task, neural networks can be trained in a semi-supervised learning fashion to approximate nonlinear solutions. In this paper, neural solvers are implemented in TensorFlow for a variety of differential equations, namely: linear and nonlinear ordinary differential equations of the first and second order; Poisson's equation, the heat equation, and the inviscid Burgers' equation. Different methods, such as the naive and ansatz formulations, are contrasted, and their overall performance is analysed. Experimental data is also used to validate the neural solutions on test cases, specifically: the spring-mass system and Gauss's law for electric fields. The errors of the neural solvers against exact solutions are investigated and found to surpass traditional schemes in certain cases. Although neural solvers will not replace the computational speed offered by traditional schemes in the near future, they remain a feasible, easy-to-implement substitute when all else fails.**

Keywords: Differential Equations, Deep Learning, Neural Networks, Numerical Methods.

I. INTRODUCTION

Differential equations (DEs) describe a physical system as a mathematical model. DEs (both ordinary and partial) are vital in many fields including, but not limited to, biology, economics, physics, chemistry and engineering. For the simpler linear equations, established analytical solutions often exist and are well-defined. However, for more complicated DEs, exact solutions cannot be expressed in elementary functions or may not even exist. As a result, numerical methods are frequently utilised to approximate these solutions, up to a certain amount of error.

With the advent of deep learning, neural networks (NNs) have resurfaced onto the field of computer simulation. Having evolved from the simple perceptron model [1] to the multilayer perceptron [2] and finally to the modern variants known today such as convolutional and recurrent networks [3], NNs are capable of solving a variety of machine learning tasks. Specifically, they have exhibited superb results in classification and regression problems, surpassing even human-level performance on particular image classification datasets [4]. In addition to computer vision, NNs have also advanced the field of natural language processing, in problems such as language understanding and machine translation [5], [6]. NNs possess these capabilities owing to their versatility as black-box function approximators. Given sufficient data, NNs can be trained to model complex and non-trivial input-output relationships.

Via the formulation of an initial value problem (IVP) or a boundary value problem (BVP) as a semi-supervised optimisation problem [7]–[10], the vast repertoire of techniques from NNs can be applied in order to approach the solution of an ordinary differential equation (ODE) or partial differential equation (PDE). Operating as universal function approximators [11], artificial NNs are capable of modelling continuous functions to a substantial extent with appropriate activation functions—the only limiting factor being the complexity of the network architecture itself.

One such approach is the work of Dissanayake and Phan-Thien [7], henceforth referred to as the naive method, wherein the DE and initial or boundary conditions are rewritten as a convex loss function parametrised by the network weights. The objective of the optimisation problem is thus to minimise this loss function with respect to the weights. Lagaris et al. [8] improves upon the naive method by introducing an ansatz for the solution. Constructing such an ansatz allows the form of the solution to be readily constrained, eliminating the need for the condition loss functions. This removal focuses the total loss function, accelerating convergence and preventing suboptimal trivial solutions. The ansatz method, however, requires knowledge of a suitable ansatz beforehand.

Further refinements include [9], in which it is suggested that the network be first pretrained on the condition loss alone in order to roughly gauge the range of the solution. Another study [10] proposed the Deep Galerkin Method (DGM), which tries to incorporate the stochasticity of conventional NN training, i.e., stochastic gradient descent (SGD) [12]. Samples of the input domain are randomly

Manuscript received 06-09-2021; revised 05-10-2021; accepted 06-10-2021

generated at every iteration to be fed into the network, conserving memory on what would otherwise be consumed by high-dimensional grids or discretisations.

Several applications of the aforementioned methods include utilising NNs in cosmological phase transitions and quantum field theory [13]; applying NNs to solve the Navier-Stokes equations with various boundary conditions [14]; and solving Poisson's equation in two and three dimensions for electric potential using convolutional NNs [15].

This work will focus on the implementation of deep neural network solvers for various types of differential equations including linear, semilinear, quasilinear, nonlinear ODEs and PDEs of predominantly the first and second orders, using TensorFlow [16], [17]. In addition, various architectures for the network will be analysed and compared to traditional numerical schemes in terms of error. The robustness of the training will also be tested by varying parameters of the NN, such as layers and the input domain. Finally, the resulting network outputs for certain simulatable problems will be validated using experimental data.

The structure of this paper is organised as follows: the introduction highlights the significance of the DE problem and provides a brief look into the evolution of NNs as well as how they have been used to solve this problem. The second section delineates the problem definition in depth and shows how one can reproduce the experimental methods alongside the implementation details and cases. In the third section, the results of the experiments are thoroughly discussed and analysed. Lastly, the conclusion summarises this research and puts forward several directions in which future work can be carried out.

II. Methodology

Consider the following IVP, consisting of an explicit first order ODE and an initial condition; and the function f on the right-hand side of the ODE is known.

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y),\ y(x_0) = y_0 \tag{1}$$

The unknown solution $y(x)$ can be approximated as a neural network $y^*(x, \mathbf{W})$, wherein $\mathbf{W}$ denotes the network's weights. In particular, the type of networks employed here are feedforward, also known as dense, NNs, consisting of a sequence of layers where the outputs of one layer are fed as the inputs of the next layer. Each dense layer is made up of a number of neurons, and each neuron is connected to all neurons of the previous layer. Mathematically, given an input x, a dense layer a_i is modelled as the transformation $a_i(x) = \sigma_i(\mathrm{W}_i x + \vec{b}_i)$. Each layer is parametrised by its kernel matrix W_i and bias vector $\vec{b}_i$, which together are known collectively as the weights $\mathbf{W}$. The activation function $\sigma_i(\cdot)$ serves to imbue nonlinearities in the network.

For example, a dense neural network with two hidden layers and one output layer can be written as the composition $y^*(x, \mathbf{W}) = a_3\left(a_2\left(a_1(x)\right)\right)$. When used for function approximation, hidden activations are typically sigmoid or hyperbolic tangent functions, whereas the output activation is not applied, i.e., linear activation. Under this formulation, errors are then introduced in both the ODE and the condition, known respectively as the equation loss (ϵ_1) and the condition loss (ϵ_2) as shown in equation (2).

$$\frac{\partial y^*}{\partial x} = f(x, y^*) + \epsilon_1,\ y^*(x_0, \mathbf{W}) = y_0 + \epsilon_2 \tag{2}$$

The sum of the squares of these losses provides a suitable objective function which the network can be trained to minimise, i.e., $\mathcal{L} = \epsilon_1^2 + \epsilon_2^2$. Such loss functions, which include both losses, are characteristic of the naive method. By a similar derivation, these loss functions can be extended to higher-order problems, PDEs, and systems of DEs. The training dataset then consists of all points inside the discretised input domain on which a solution is desired. Using one of the many variants of SGD, such as the Adam optimiser [18], the weights can be iteratively updated until convergence.

On the other hand, ansatz or trial functions can also be constructed to approximate the solutions, as shown in [8]. For instance, the following ansatz may be used to solve equation (1). Note that the initial condition is automatically satisfied regardless of the state of the network weights. Similar trial functions can be contrived for higher order ODEs, PDEs, and systems of DEs as well.

$$y_{\text{ansatz}} = (x - x_0)y^*(x, \mathbf{W}) + y_0 \tag{3}$$

As for the dataset used in the experiments to train the NNs, due to the relatively low dimensionality of the problems considered, an input batch generator similar to [10] is deemed unnecessary since the dataset shall fit entirely in memory. To construct this training dataset, the concerned domain is discretised into a grid of equidistant points containing the initial or boundary conditions. Supposing that the IVP in equation (1) is to be solved in the real interval given by $[x_0, x_0 + L]$, where L denotes the length of the domain, then the training dataset X is described as the following set of N points.

$$X = \{x_i \in [x_0, x_0 + L] \mid x_i = x_0 + \frac{L}{N-1} i\} \tag{4a}$$

Alternatively, the training dataset X can also be described explicitly as the sequence of equidistant points $\{x_0, x_1, x_2, \dots, x_{N-1}\}$:

$$X = \{x_0, x_0 + \frac{L}{N-1}, x_0 + \frac{2L}{N-1}, \dots, x_0 + L\} \tag{4b}$$

The training algorithms are implemented in TensorFlow, an open-source platform for machine learning. Key features of the library which facilitate the programming of the algorithms include: an object-oriented paradigm for building the models, support for hardware acceleration, a diverse accoutrement of

mathematical functions, and most importantly, the ability to compute arbitrary derivatives using auto-differentiation. Unlike typical loss functions found in supervised learning, the loss functions implemented to train neural solvers can depend on not only the network outputs, but also the inputs themselves and derivatives with respect to those inputs. Therefore, being able to compute and backpropagate through these peculiar loss functions is paramount.

Neural solvers will be implemented to solve the following cases: first and second order ODEs with varying degrees of linearity; elliptic (Poisson's equation), parabolic (heat equation), and hyperbolic (the inviscid Burgers' equation) PDEs, shown in equations (5, 6, 7) along with their respective boundary or initial conditions.

Poisson's equation with Dirichlet boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \sin(\pi x)\sin(\pi y) \tag{5}$$

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0$$

The heat equation with Dirichlet boundary conditions:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{6}$$

$$u(t, 0) = u(t, 1) = 0$$
$$u(0, x) = \sin(\pi x)$$

The inviscid Burgers' equation with the initial condition (both positive and negative initial conditions will be attempted):

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{7}$$

$$u(0, x) = \pm\sin(2\pi x)$$

To analyse their overall performance, the error across the domain is compared to that of conventional numerical schemes, e.g., forward Euler and Runge-Kutta schemes for ODEs, and the finite-difference method (FDM) for PDEs.

As for experimental validation, the neural solver method will be applied to the simulation of two problems and juxtaposed with their respective experimental measurements. The first experiment involves the classical spring-mass system, modelled by the linear constant-coefficient second order ODE. Letting y represent the vertical displacement, t the time, y_0 the initial displacement, and v_0 the initial velocity, the model is given in equation (8).

$$m\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} + b\frac{\mathrm{d}y}{\mathrm{d}t} + ky = 0, y(0) = y_0, y'(0) = v_0 \tag{8}$$

The constants which define the characteristic solution are the mass m, the friction b, and the spring constant k. The second experiment, performed by Moradi and Marvasti [19], is governed by Gauss's law of electrostatics for the electric field $\mathbf{E}$ within a domain containing zero charge, $\nabla \cdot \mathbf{E} = 0$. Substituting with the electric potential instead, $\mathbf{E} = \nabla u$, the law boils down to Laplace's equation in two dimensions with Dirichlet boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{9}$$

$$u(x, y_0) = u_1,\ u(x_1, y) = u_2$$
$$u(x, y_1) = u_3,\ u(x_0, y) = u_4$$

III. Results and Discussion

For ODEs, three dense NNs, each with two hidden layers with hyperbolic tangent activations, are trained according to the naive loss for a total of 1000 epochs each using the Adam optimiser. The hidden sizes are denoted within the brackets in the legend. The minimum error achieved against the exact solution is recorded and compared to that of traditional schemes such as the forward Euler and fourth order Runge-Kutta methods for IVPs; and the FDM for BVPs, over increasingly larger grid sizes. The mean squared error is used as the common metric. In most cases, the neural method managed to achieve similar, if not smaller, errors as shown in the following figures.

Note that a fresh network is created for each combination of grid size and network architecture in order to ensure that the initial weights are randomly initialised and so that the training process is fair.

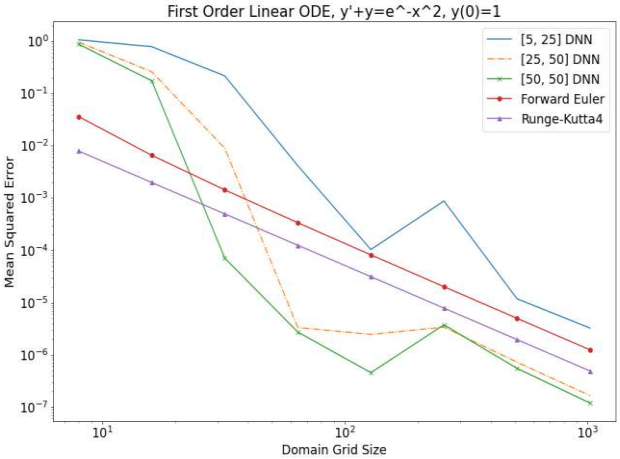

Figure 1. Error comparisons against the exact solution for neural solvers of varying sizes, the forward Euler and fourth order Runge-Kutta schemes for the first order linear ODE. The number within the brackets in the legend denotes the number of neurons in each hidden layer of the network.

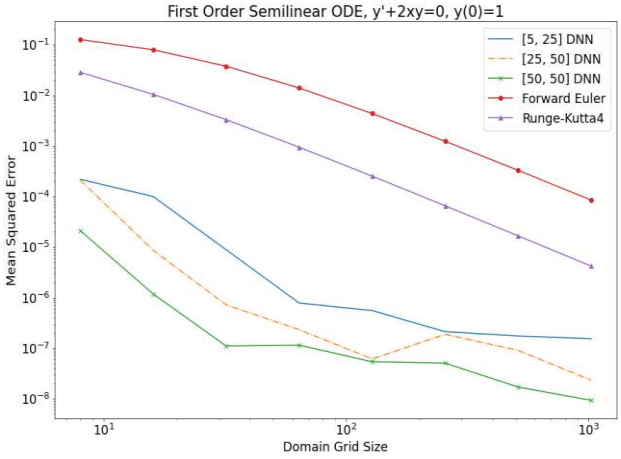

Figure 2. Error comparisons against the exact solution for neural solvers of varying sizes, the forward Euler and fourth order Runge-Kutta schemes for the first order semilinear ODE.

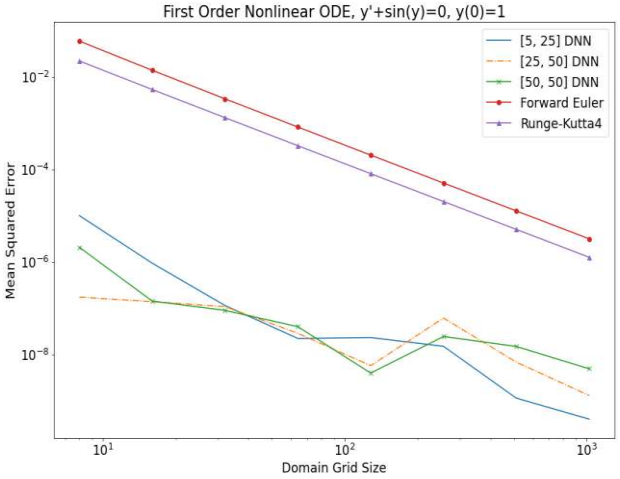

Figure 3. Error comparisons against the exact solution for neural solvers of varying sizes, the forward Euler and fourth order Runge-Kutta schemes for the first order nonlinear ODE.

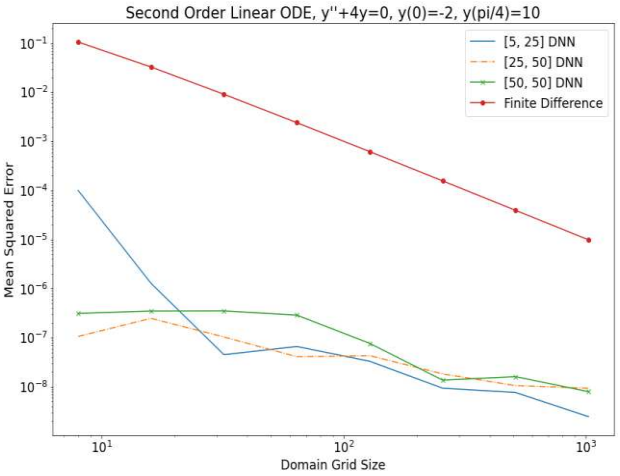

Figure 4. Error comparisons against the exact solution for neural solvers of varying sizes and the FDM for the second order linear ODE.

In certain cases, such as in Figures 2 and 3, the overall errors with respect to the exact solutions for the neural method across all tested grid sizes were smaller than that of forward Euler and Runge-Kutta schemes by roughly a couple orders of magnitude, though this is not always the case, as shown in Figure 1, where the neural method only manages to outperform the traditional schemes on finer grids. Ultimately, their performance varies on a case-by-case basis according to the problem. As for the BVP in Figure 4, the neural solvers achieved a much smaller magnitude of error than the FDM.

The superior performance of the neural solvers can be attributed to their powerful approximation capabilities, in the sense that, unlike traditional time-stepping schemes which deterministically yield a discrete set of points, the neural solvers can be iteratively trained for multiple epochs. On top of that, because of the choice of the hyperbolic tangent activation, the network output is constrained to be continuous and smooth, which further improves the quality of the neural solution. Generally, increasing the size of the network causes the error to decrease due to the greater modelling capacity of the network, though this difference is not always clear-cut.

Similar results were observed for PDEs over many grid sizes, as shown in Figures 5 and 6, which respectively correspond to the test equations of Poisson's equation (see equation (5) and the heat equation (see equation (6)). The mean squared errors are again computed with respect to the exact solutions, and the mean squared error is again used as the common metric for comparison.

Due to the greater difficulty of solving PDEs as compared to ODEs, i.e., the higher number of independent variables (dimensions), smaller NNs with shallower and less wide layers are unable to correctly solve PDEs as a consequence of their limited modelling capacities. Thus, a sufficiently deep NN with three hidden layers of fifty neurons each is chosen here. This choice of architecture proves to possess enough flexibility to approximate the more difficult PDEs.

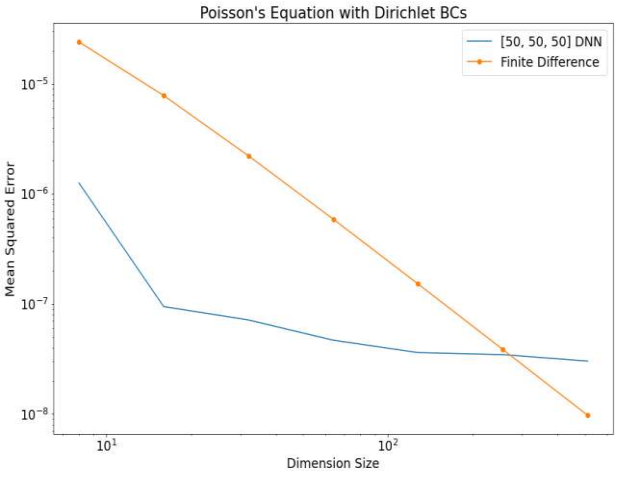

Figure 5. Error comparisons against the exact solution for neural solvers and the FDM for Poisson's equation (see equation (5)).

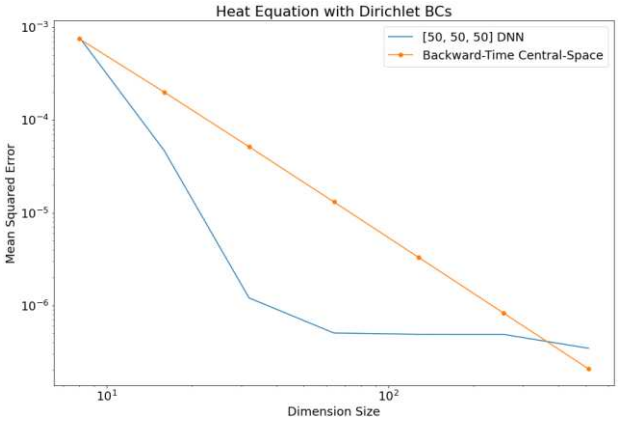

Figure 6. Error comparisons against the exact solution for neural solvers and the Backward-Time Central-Space (BTCS) method for the heat equation (see equation (6)).

Although the neural solvers demonstrate smaller errors over the majority of grid sizes, they experience diminishing returns after which traditional schemes overtake them. It can be inferred that the neural method fares better on sparse grids for PDEs. In fact, they were able to converge on grids where explicit time-marching schemes would have diverged. This is evident in the case of the heat equation, where an equidistant two-dimensional grid was used. If the explicit Forward-Time Central-Space (FTCS) method had been used instead of the implicit BTCS, the Courant-Friedrichs-Lewy (CFL) condition [20] would have been violated, and the numerical instability would have caused the scheme to diverge. On the contrary, the neural method fares well for a grid with such a large Courant number, behaving very stably as though it were implicit.

The numerical stability demonstrated by neural solvers when solving PDEs can be traced back to the very nature of NN training. In contrast to explicit time-marching schemes which compute future values of the solution based solely on known present values, neural solvers are granted access to the entire temporal domain during the training process. It is precisely due to this process that neural solvers are capable of exhibiting the numerical stability which implicit schemes often achieve through an additional computation step.

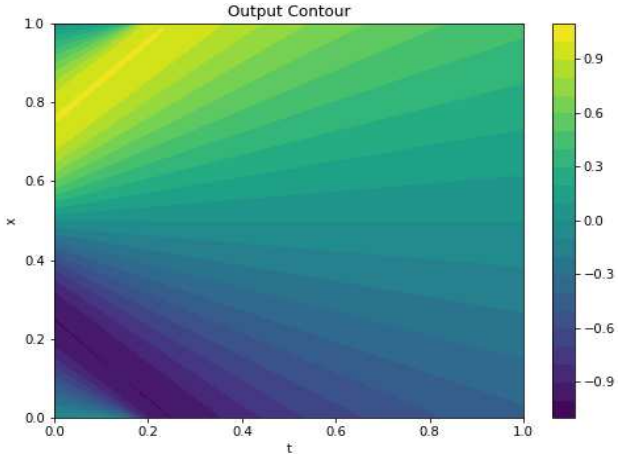

Figure 7. Inviscid Burgers' equation (see equation (7)) results (convergent).

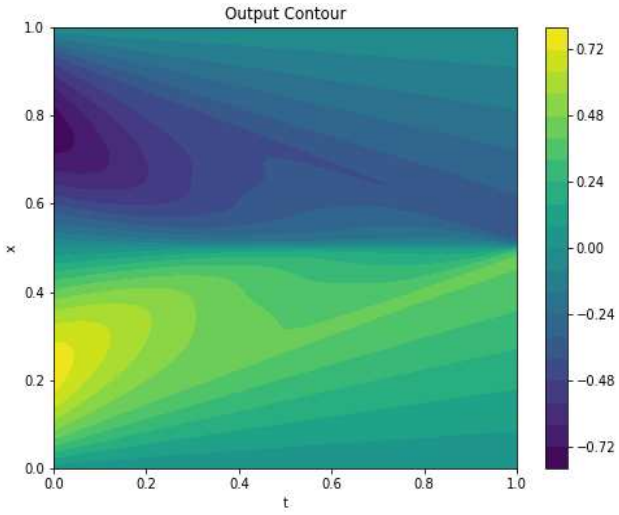

Figure 8. Inviscid Burgers' equation (see equation (7)) results (divergent).

Convergence of a solution to the inviscid Burgers' equation (see equation (7)), on the other hand, is solely dependent upon the absence of any shock waves, i.e., discontinuities where the PDE is not well-defined. Unlike finite-volume schemes, wherein volume integrals are conserved throughout the domain, the neural method relies on the bona-fide PDE itself for stability. In this case, due to the formation of shock waves, the equation becomes ill-defined and the method destabilises. The collapse of the training algorithm occurs not only in the vicinity of the shock wave but also percolates everywhere else in the domain, as in Figure 8. Nonetheless, initial conditions that do not lead to shock wave formation can converge successfully (Figure 7).

For experimental validation, the classical spring-mass system was implemented, and measurement data was obtained for comparison with the solutions to its mathematical model in equation (8). It can be observed in Figure 9 that the neural method closely follows the exact solution and differs from the measurement results by a small margin of error (see Figure 10).

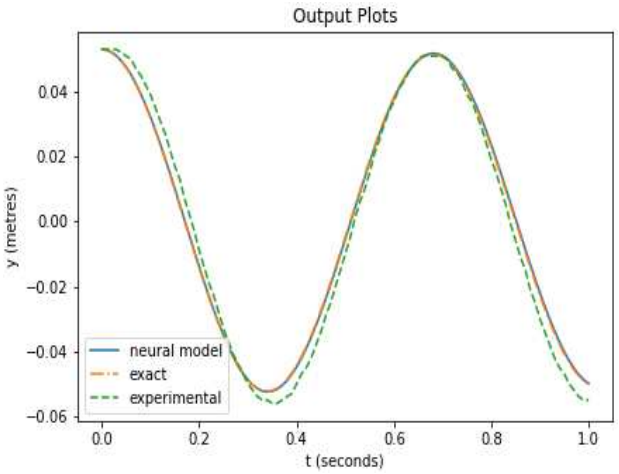

Figure 9. Spring-mass system (see equation (8)) results: solutions.

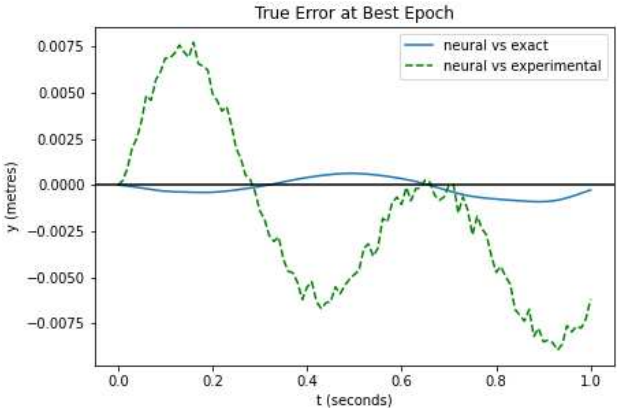

Figure 10. Spring-mass system (see equation (8)) results: errors.

In addition, the neural method is also applied to the experiment performed in [19], wherein the left and top sides of the domain are held at ground potential while the right and bottom sides are connected to a potential of a hundred millivolts. In terms of equation (9), these conditions correspond to $u(x,0) = u(0.3,y) = 0.1$ and $u(x,0.2) = u(0,y) = 0$. Due to measurement noise, the contours of the experimental plot shown in Figure 11 appear slightly jagged.

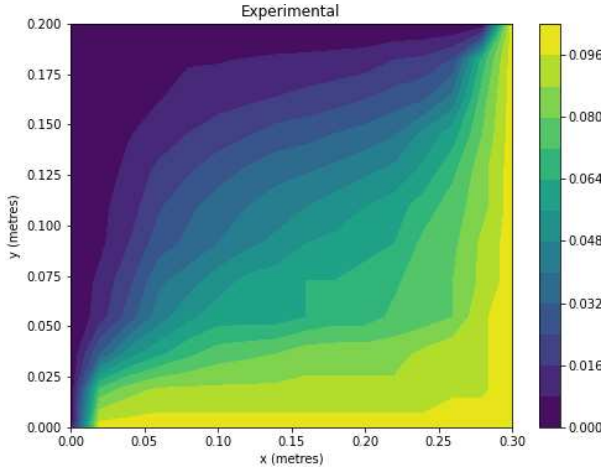

Figure 11. Gauss's law (see equation (9)) results: measurement data from [19].

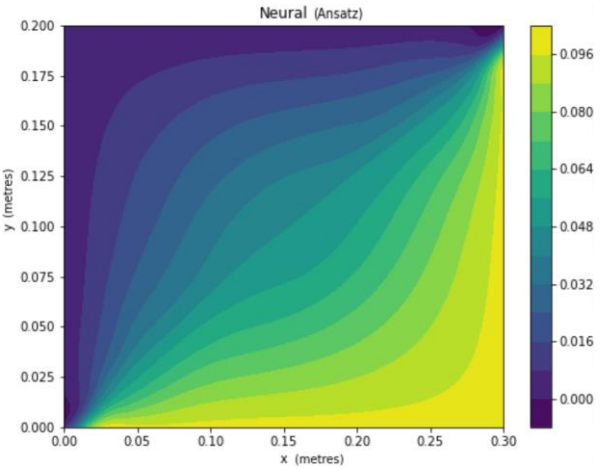

Figure 12. Gauss's law (see equation (9)) results: ansatz solution.

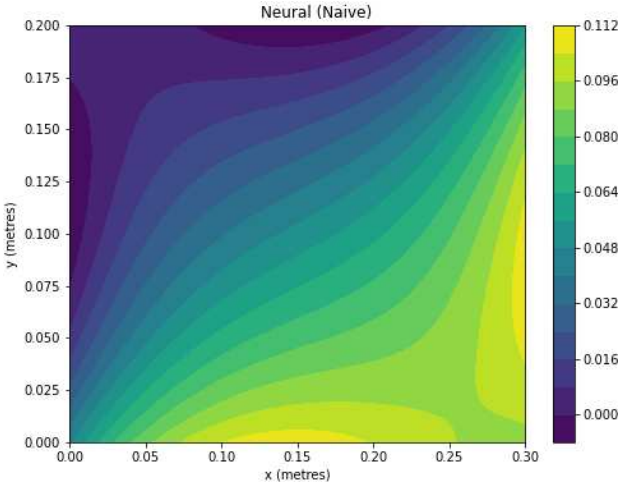

Figure 13. Gauss's law (see equation (9)) results: naive solution.

As observed in Figure 12 and 13, the networks manage to capture the gradation of the potential field smoothly. The corner discontinuities as well as the border potentials, however, are only prominent in the ansatz solution of Figure 12, since the boundary conditions are already satisfied by construction.

IV. Conclusion

In summary, the neural method for solving differential equations was implemented in TensorFlow for a large collection of cases, namely ordinary differential equations with varying degrees of linearity, and partial differential equations, exhibiting excellent convergence and smaller true errors by a couple orders of magnitude compared to traditional schemes such as Runge-Kutta and finite-difference. As for numerical stability, the neural method has been shown to be unaffected by constraints such as the Courant-Friedrichs-Lewy condition and is in fact capable of solving partial differential equations on extremely coarse grids. Finally, the neural method is validated with experimental measurements.

Future work related to this topic may incorporate: the implementation of lattice networks to solve differential equations in a more constrained manner; the investigation of the precise convergence criteria of loss functions that depend on not only the network output, but also its inputs and derivatives; and further development on techniques to accelerate convergence.

Acknowledgements

The authors of this paper would like to thank Swiss German University for providing the equipment to conduct the spring-mass experiment, and for funding the publication costs.

References

[1] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain,"

Psychol. Rev., vol. 65, no. 6, pp. 386–408, Nov. 1958, doi: 10.1037/h0042519.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, doi: 10.1038/323533a0.

[3] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/nature14539.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034, doi: 10.1109/ICCV.2015.123.

[5] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, Jun. 2017, vol. 2017-Decem, pp. 5999–6009, Accessed: Jan. 28, 2021. [Online]. Available: https://arxiv.org/abs/1706.03762v5.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *NAACL HLT 2019 - 2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf.*, vol. 1, pp. 4171–4186, Oct. 2018, Accessed: Jan. 28, 2021. [Online]. Available: http://arxiv.org/abs/1810.04805.

[7] M. W. M. G. Dissanayake and N. Phan-Thien, "Neural-network-based approximations for solving partial differential equations," *Commun. Numer. Methods Eng.*, vol. 10, no. 3, pp. 195–201, Mar. 1994, doi: 10.1002/cnm.1640100303.

[8] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE Trans. Neural Networks*, vol. 9, no. 5, pp. 987–1000, 1998, doi: 10.1109/72.712178.

[9] J. Berg and K. Nyström, "A unified deep artificial neural network approach to partial differential equations in complex geometries," *Neurocomputing*, vol. 317, pp. 28–41, Nov. 2018, doi: 10.1016/j.neucom.2018.06.056.

[10] J. Sirignano and K. Spiliopoulos, "DGM: A deep learning algorithm for solving partial differential equations," *J. Comput. Phys.*, vol. 375, pp. 1339–1364, Dec. 2018, doi: 10.1016/j.jcp.2018.08.029.

[11] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control. Signals, Syst.*, vol. 2, no. 4, pp. 303–314, Dec. 1989, doi: 10.1007/BF02551274.

[12] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv*, Sep. 2016, Accessed: Jan. 28, 2021. [Online]. Available: http://arxiv.org/abs/1609.04747.

[13] M. L. Piscopo, M. Spannowsky, and P. Waite, "Solving differential equations with neural networks: Applications to the calculation of cosmological phase transitions," *Phys. Rev. D*, vol. 100, no. 1, Jul. 2019, doi: 10.1103/PhysRevD.100.016002.

[14] M. Baymani, S. Effati, H. Niazmand, and A. Kerayechian, "Artificial neural network method for solving the Navier–Stokes equations," *Neural Comput. Appl.*, vol. 26, no. 4, pp. 765–773, May 2015, doi: 10.1007/s00521-014-1762-2.

[15] W. Tang *et al.*, "Study on a Poisson's equation solver based on deep learning technique," in *2017 IEEE Electrical Design of Advanced Packaging and Systems Symposium, EDAPS 2017*, Jan. 2018, vol. 2018-Janua, pp. 1–3, doi: 10.1109/EDAPS.2017.8277017.

[16] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *CoRR*, vol. abs/1603.0, Mar. 2016, Accessed: Jan. 28, 2021. [Online]. Available: http://arxiv.org/abs/1603.04467.

[17] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, May 2016, pp. 265–283, Accessed: Jan. 28, 2021. [Online]. Available: http://arxiv.org/abs/1605.08695.

[18] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," Dec. 2015, Accessed: Jan. 28, 2021. [Online]. Available: https://arxiv.org/abs/1412.6980v9.

[19] G. Moradi and M. Marvasti, "Experimental Solution to the Laplace Equation, a Tutorial Approach," *IJARCCE*, vol. 5, no. 9, pp. 278–284, Sep. 2016, doi: 10.17148/ijarcce.2016.5960.

[20] R. Courant, K. Friedrichs, and H. Lewy, "Über die partiellen Differenzengleichungen der mathematischen Physik," *Math. Ann.*, vol. 100, no. 1, pp. 32–74, Dec. 1928, doi: 10.1007/BF01448839.