# Finding Multiple Solutions of ODEs with Neural Networks

**Marco Di Giovanni[1], David Sondak[2], Pavlos Protopapas[2], Marco Brambilla[1]**
[1]Politecnico di Milano. Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB). Milan, Italy
[2]Institute for Applied Computational Science, Harvard University, Cambridge, MA, United States

CEUR-WS.org/Vol-2587/article_7.pdf

## Abstract

Applications of neural networks to numerical problems have gained increasing interest. Among different tasks, finding solutions of ordinary differential equations (ODEs) is one of the most intriguing problems, as there may be advantages over well established and robust classical approaches. In this paper, we propose an algorithm to find all possible solutions of an ordinary differential equation that has multiple solutions, using artificial neural networks. The key idea is the introduction of a new loss term that we call the *interaction loss*. The interaction loss prevents different solutions families from coinciding. We carried out experiments with two nonlinear differential equations, all admitting more than one solution, to show the effectiveness of our algorithm, and we performed a sensitivity analysis to investigate the impact of different hyper-parameters.

## Introduction

A wide variety of phenomena are governed by mathematical laws that can be represented using ordinary differential equations, ranging from the field of physics to chemistry to finance.

In the field of physics, examples include Navier-Stokes equation, Schrodinger equation and many others. However, it is not always possible to solve those problems analytically, and sometimes numerical approaches are required since these equations are too complicated or the number of equations and variables is too big.

Numerical approaches to solve differential equations have been widely studied and improved almost since the first differential equations were written down. Classical approaches for spatial discretization of partial differential equations (PDEs) include finite differences, finite elements, finite volumes, and spectral methods. Classical methods for discretizing ordinary differential equations (ODEs) include the Runge-Kutta (RK) methods and the backward difference formulas (BDF). Also artificial neural networks (ANNs) have been applied to this scope. Lagaris, Likas, and Fotiadis(1998) listed a set of theoretical advantages of the ANN approach, suggesting that even if the results are not quite as good as the classical approaches, there are good reasons to continue the research in this direction. In fact, the solution obtained is differentiable with a closed analytic form and this approach is easily parallelizable and general since it can be applied to ODEs, systems of ODE, and PDEs.

Copyright © 2020, for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CCBY 4.0).

In this paper we design an algorithm in order to apply this technique to solve a different problem: finding all possible solutions of an ordinary differential equation that permits non-unique solutions. The vast majority of analytical an numerical techniques have been developed and applied to differential equations emitting unique solutions. However, there are practical scientific problems that are modeled by nonlinear differential equations that do not result in unique solutions. The Bratu equation has been used to model combustion phenomena as well as temperatures in the sun's core. The Bratu equation does not have a known analytical solution in more than one spatial dimension. Moreover, classical numerical methods will not be able to find all solution families. (Karkowski 2013).

After describing the background concepts and the proposed algorithm, we list a set of simple problems where the solutions are known in order to test the algorithm. The results are presented including sensitivity analysis of the hyper-parameters that need to be tuned.

## Related Work

Algorithms to solve differential equations using neural networks were originally developed by Lagaris, Likas, and Fotiadis(1998). The authors proposed a novel approach to tackle the problem of numerically solving differential equations using ANNs, listing some advantages of their algorithm with respect to classical ones. Testing it with ordinary differential equations (ODEs), partial differential equations (PDEs) and systems of ODEs, they checked their generalization abilities, obtaining surprisingly good results. Finally, they extended it to problems with arbitrarily shaped domains, in more than one dimension (Lagaris, Likas, and Papageorgiou(1998)) and they applied it to quantum mechanics (Lagaris, Likas, and Fotiadis(1997)). An interesting survey is presented by Kumar and Yadav(2011). The authors collect a large number of papers about solving dif-

ferential equations, focusing on both multi-layer perceptrons and radial basis functions techniques, published between 2001 and 2011. A similar approach, called the Deep Galerkin Method (DGM), was proposed by Sirignano and Spiliopoulos(2018). The authors present an algorithm similar to Galerkin methods, but with neural networks instead of basis functions. Raissi, Perdikaris, and Karniadakis(2019) formulate Physics-informed neural networks, a framework developed to solve two main classes of problems: data-driven solution of PDEs, also inspired by Lagaris, Likas, and Fotiadis(1998), and a data-driven discovery of PDEs, encoding in the neural networks the physical laws that govern a data-set. In another work, Baymani, Kerayechian, and Effati(2010) compare the performance of neural networks with the errors using Aman-Kerayechian method, for obtaining the solutions of Stokes Equation.

As stated before, one of the theoretical advantages of using artificial neural networks to solve ODEs, with respect to classical methods, is that the latter are affected to the curse of dimensionality, while the NN approach scales better with the number of dimensions. E, Han, and Jentzen(2017) analyze this propriety, reformulating the PDEs using backward stochastic differential equations, and applying it to solve the nonlinear Black-Scholes equation, the Hamilton-Jacobi-Bellman equation, and the Allen-Cahn equation.

It is also essential that neural networks incorporate known physical laws. Mattheakis et al.(2019) embed physical symmetries into the structure of the neural network. The symplectic neural network obtained is tested with a system of energy-conserving differential equations and outperforms an unsupervised, non-symplectic neural network.

There is also a new trend investigating dynamical systems, introducing new families of neural networks, characterized by continuous-depth, whose output is computed using black-box differential equation solvers. Their application is different from this work, since they are mainly used for classification tasks (He et al. 2016; Chen et al. 2018; Zhu, Chang, and Fu 2018).

## Background

As described by Lagaris, Likas, and Fotiadis(1998), fully connected feed forward neural networks can be used to compute solutions of differential equations with some advantages and disadvantages with respect to classical approaches. In this section, we briefly expose the background concepts.

### Differential equations

We write a general ODE as,

$$F(t, u(t), u'(t), u''(t), ...) = 0 \quad (1)$$

where $t$ is the independent variable and $u(t)$ is the solution of the differential equation. We restrict the scope of the present paper to differential equations of only a single variable and interpret $t$ as time. Note that $u'$ represents the first derivative and $u''$ represents the second derivative. We remark that the differential equation considered herein (1) need not to be separable with respect to $u'(t)$, as required in classical Runge-Kutta methods. This is one of the main advantages of using neural networks to solve differential equations with respect to classical approaches. In order to determine $u(t)$ from (1), one must specify either initial or boundary values. Boundary value problems (BVPs) impose the values of the function at the boundaries of the interval $t \in [t_0, t_f]$, e,g. $u_0 = u(t_0)$ and $u_f = u(t_f)$. Initial value problems (IVPs), on the other hand, impose the value of the function and its derivatives at $t_0$, e.g. $u_0 = u(t_0)$, $v_0 = u'(t_0)$. Both of these approaches can be implemented using neural networks with very little modification to the loss. For example, switching between an IVP and a BVP does not require any modification to the network architecture. One only need adjust the loss function to incorporate the appropriate initial or boundary conditions. Classical methods often require different algorithms for IVPs or BVPs and therefore may require additional code infrastructure. The situation may become more severe when considering PDEs. The neural network architecture will now take in multiple inputs (e.g. space and time) and the loss function will incorprate the appropriate initial and boundary conditions. However, traditional PDE software codes may use multiple algorithms to solve the entire PDE (e.g. time-integration in time and finite elements in space (Seefeldt et al. 2017; Alnæs et al. 2015; Burns et al. 2016).

### Solving ODEs with Neural networks

Following the methodology explained in Lagaris, Likas, and Fotiadis(1998), we use a fully connected feed forward neural network to solve ODEs.

The key idea of this approach is that the neural network itself represents the solution of the differential equation. Its parameters are learnt in order to obtain a function that minimizes a loss, forcing the differential equation to be solved and its boundary values (or initial conditions) to be satisfied. Firstly, the architecture of the neural network has to be selected, setting the number of layers, the number of units per layer and the activation functions. Fine-tuning these parameters and testing more complex architectures can be done to better approximate the loss functions. However, for the purpose of this work, we fix the architecture of the neural network selecting the one that we believe is suitable enough for the differential equations selected, by manually inspecting the validation loss. Too small architectures could be unable to model the target function, while bigger architecture could be harder to train. For ODEs, the neural network only has a single input: the independent variable $t$ at several discrete points $t_i$. The output of the neural network represents the value of the function that we want to learn. Since we are only focusing on scalar equations in the present work, the number of units in the output layer should also be one. Automatic differentiation (Corliss et al. 2002) is used to compute exact derivatives with respect not only to the parameters of the network, but also to the inputs to the neural network (here $t$). The training process of the neural network is accomplished by tuning the parameters of the neural network to minimize a loss function. In the current problem, the loss function is simply the differential operator acting on the predicted func-

tion. That is,

$$\mathcal{L}_{ODE} = \sum_i F(t_i, u(t_i), u'(t_i), u''(t_i), ...)^2. \quad (2)$$

We note that this method is related to the Galerkin Least-Squares method where the residual is evaluated at discrete points (similar to a collocation method). A function that guarantees $F(\cdot) = 0$ everywhere is said to satisfy the differential equation. In practice, the best case is that the neural network will find a function $u^*(t)$ that minimizes (2) by obtaining values for the residual close to zero.

To train the neural network, we pick a set of $n_{train}$ points belonging to the domain in which we are interested $[t_0, t_f]$. In this work we consider equally spaced points, although this is not required. We perform a forward pass of the network which provides the function approximation and calculate the derivatives as required by the differential operator through automatic differentiation. The forward pass is completed by computing the loss via (2). The weights and biases of the network are updated with the gradient descent algorithm via application of the backpropagation algorithms. This process is repeated until convergence or until the maximum number of epochs is reached. The output of the process is the function that obtains the lowest loss calculated using an evaluation set, picking $n_{val}$ points equally spaced in the same domain.

A critical consideration is to realize the specified initial and/or boundary conditions. To encode boundary values or initial conditions, we use the method described by Sirignano and Spiliopoulos(2018). We add a term in the loss function evaluating the squared distance between the actual value of the function at its boundaries and the imposed values. In the case of BVPs this additional loss function is,

$$\mathcal{L}_{BV} = (u(t_0) - u_0)^2 + (u(t_f) - u_f)^2. \quad (3)$$

In the case of IVPs, the new loss function is,

$$\mathcal{L}_{IC} = \sum_{n=0}^{n_{ord}-1} \left( u^{(n)}(t_0) - v_0^{(n)} \right)^2 \quad (4)$$

where $n_{ord}$ is the order of the differential operator, $u^{(n)}$ is the $n^{\text{th}}$ derivative of the function $u$ and $v_0^{(n)}$ is the initial value corresponding to the $n^{\text{th}}$ derivative. For example, for a second order IVP, the loss would contain $v_0^{(0)}$ and $v_0^{(1)}$.

The total loss is,

$$\mathcal{L} = \mathcal{L}_{ODE} + \mathcal{L}_{\mathcal{I}} \quad (5)$$

where $\mathcal{L}_{\mathcal{I}}$ is given by (3) or (4) depending on the context of the problem.

In some applications, it may be necessary to introduce a penalty parameter before $\mathcal{L}_{\mathcal{I}}$ to mitigate different scalings between $\mathcal{L}_{ODE}$ and $\mathcal{L}_{\mathcal{I}}$. In the present work, we fix the penalty parameter to unity and note that this did not interfere with the current results. It may be necessary to tune this penalty parameter for more complicated problems.

## Equation library

The proposed approach is tested on two nonlinear differential equations. Each of these equations is known to admit multiple families of solutions.

**Clairaut equation** The general Clairaut equation is

$$u(t) = tu'(t) + f(u'(t)), \quad u(0) = u_0 \quad (6)$$

where $f$ is a known nonlinear function Ince(1956). There are two families of solutions to (6). The first is a line of the form,

$$u(t) = Ct + f(C) \quad (7)$$

where is determined by the initial conditions. The second family of solutions can be represented in parametric form as,

$$\begin{cases} t(z) = -f'(z) \\ u(z) = -zf'(z) + f(z). \end{cases} \quad (8)$$

In this work, we will consider two forms for the function $f$. The first form is,

$$f(u'(t)) = \frac{1}{u'(t)} \quad (9)$$

which can be written in the form of (1) as,

$$F(t, u(t), u'(t)) = tu'(t)^2 - u(t)u'(t) + 1 = 0 \quad (10)$$

With $u(1) = 2$, the two separate solutions are,

$$u_1(t) = 2\sqrt{t} \quad (11a)$$

$$u_2(t) = t + 1 \quad (11b)$$

For the second problem we select,

$$f(u'(t)) = u'(t)^3 \quad (12)$$

which can be written as,

$$F(t, u(t), u'(t)) = tu'(t) - u(t) + u'(t)^3 = 0. \quad (13)$$

This problem has three distict solutions. With $u(-3) = 2$, the three solutions are,

$$u_1(t) = 2\left(-\frac{t}{3}\right)^{\frac{3}{2}} \quad (14a)$$

$$u_2(t) = 2t + 8 \quad (14b)$$

$$u_3(t) = -t - 1 \quad (14c)$$

**1D Bratu problem** The one-dimensional Bratu problem is

$$u''(t) + Ce^{u(t)} = 0, \quad u(0) = u(1) = 0. \quad (15)$$

This ODE has two solutions if $C < C_{\text{crit}}$ and only one solution if $C \geq C_{\text{crit}}$, where $C_{\text{crit}} \approx 3.51$. We select $C = 1$ in the regime where two functions solve the problem (Bratu 1914). The exact analytical solution is

$$u(t) = 2 \ln \frac{\cosh(\alpha)}{\cosh(\alpha(1 - 2t))} \quad (16)$$

where $\alpha$ satisfies

$$\cosh \alpha = \frac{4}{\sqrt{2C}} \alpha. \quad (17)$$

For $C = 1$, equation (17) has two solutions given by $\alpha_1 \approx 0.379$ and $\alpha_2 \approx 2.73$, which ultimately results in two solutions for (16).

## Methodology

In this section, we describe an algorithm that generalizes the method described in section *"Solving ODEs with Neural networks"* to find all families of solutions to a given differential equation.

To find the whole set of solutions of an ODE, we represent each of the $N$ solutions with a different neural network $u_l(t)$, $l = 1, ..., N$. Each function $u_l(t)$ is modelled through a neural network with the same architecture, the same number of layers and the same number of units, but weights are not shared.

The training is done simultaneously, minimizing a loss function,

$$\mathcal{L} = \sum_{j}^{N} \ell_j \tag{18}$$

where

$$\ell_j = \mathcal{L}_{_{ODE_j}} + \mathcal{L}_{_{I_j}} \tag{19}$$

as in equation 5. Since the neural networks are not "interacting", they will learn one of the possible solutions, but there is no requirement that the learnt solutions will be different. To overcome this shortcoming, we define an interaction function $g\left(u_1, u_2\right)$, detecting if two predicted functions are close. Given two functions $u_1(t)$ and $u_2(t)$, the value of $g(u_1, u_2)$ needs to be high if they are similar, while, if they are different, its value should be low. We propose an interaction function of the form,

$$g_p(u_1, u_2) = \sum_{i}^{n_{train}} |u_1(t_i) - u_2(t_i)|^{-p} \tag{20}$$

where $p$ is a hyperparameter that controls the separation between the two functions. We propose a new loss function that incorporates this interaction via,

$$\mathcal{L}_{\text{tot}} = \mathcal{L} + \lambda \sum_{i \neq j} g_p(u_i, u_j) \tag{21}$$

where $\mathcal{L}$ is given by (18) and $\lambda$ is a hyperparameter setting the importance of the interaction.

Training the neural network with loss defined in (21) could lead to bad results near the boundaries or initial points. This is because the different solution families must be close to each other in those regions, but (21) insists that they should be different. To overcome this problem, we use the modified loss in equation (21) for a training interval $[n_i,\ n_f]$ where $n_i > 0$ and $n_f < n_{\text{tot}}$.

To improve the performance of the algorithm, the interaction term can be scaled by a factor dependent on the distance of the point with respect to the boundaries. For example, if we are dealing with an IVP the scaling factor will be $\dfrac{t - t_0}{t_f - t_0}$, so that the interaction is low near $t_0$, while it is maximum near $t_f$. For BVPs, the procedure is similar, with low values at the borders and high values in the middle of the interval.

The basic training process is as follows. First, $N$ neural networks are randomly initialized. The first part of the training proceeds for $n_i$ epochs using the loss in (18). The networks could converge to the same minima or to different ones. After $n_i$ epochs, the new loss in (21) replaces the loss in (18) and the networks begin to interact through the interaction term. The networks gradually try to converge to solutions of the differential equation while moving away from each other. After $n_f$ total epochs, the interaction term is removed and the original loss in (18) is used to help each network converge to the real minima. If the second phase of the training separated the functions enough to be near different minima, then they will converge to different solutions. A variant to this approach is to fix one or more of the neural networks during the second part of the training, so that the effect of the interaction is to move the remaining solutions away from the others. For example, if there are two distinct solutions, one will be fixed near a real minimum, while the other will be pushed away. If the number of solutions of the differential equation is lower than the number of neural networks, at least two of them will eventually converge to the same solution.

To explore better the space of solutions, we can gradually increase the strength of the interaction $\lambda$. During the second part of the training, the differential equation is gradually neglected and the functions will tend to become more distinct.

```
Data: λ_0, λ_M, loss_M, D_m, k
N ← 1, λ ← λ_0, NN_old ← None;
while λ < λ_M do
    NN ← createNN(N);
    loss, D ← train(NN);
    if ∃ i |loss_i > loss_M then
        return NN_old;
    end
    if ∃ i, j |D_ij < D_m then
        λ ← λ * k;
    end
    else
        NN_old ← NN;
        N ← N + 1;
    end
end
return NN_old;
```

**Algorithm 1:** Complete algorithm

Algorithm 1 depicts the method in detail. We initialize the number of neural networks to 1 and the interaction parameter to $\lambda_0$. We train the network and compute the loss and the pairwise distances. The pairwise distances $D_{ij}$ are calculated using (22).

$$D_{ij} = \frac{1}{n_{train}} \sum_{l}^{n_{train}} |u_i(t_l) - u_j(t_l)| \tag{22}$$

Then, we check that every final loss is lower than a threshold $loss_M$, in order to understand if the algorithm converged or not. If not, it means that the training process must be improved (for example using a better optimization technique, increasing the number of epochs or widening the architecture).

If the final losses are all lower than a threshold, then we check the pairwise distances, to understand if the solutions

obtained are the same or not. If there is at least one couple of solutions that has a distance lower than a fixed threshold $D_m$, then it means that those functions converged to the same minimum, thus we increase the strength of interaction by a factor $k$ and perform again the training. Otherwise, it means that we were able to find at least $N$ solutions. We increase the number of networks by 1 and rerun the algorithm in order to check if there are more than $N$ possible solutions. This procedure allows us to calculate all the solutions without knowing their number a priori. To speed up the training, we can use the neural networks learned as initialization of the new ones, so that only one network has to be learned from scratch each time we increase $N$.

## Results and discussion

In this section the results are presented, followed by a sensitivity analysis of the hyper-parameters selected.

### Neural network architecture

All results were obtained using the same network architecture. The network consists of two fully-connected layers with 8 units per layer and sigmoid activations functions. The architecture also uses a skip connection, which we observed helps with learning linear functions. The network is trained using the Adam optimizer with learning rate $10^{-2}$, $\beta_1 = 0.99$ and $\beta_2 = 0.999$

### Interaction function analysis

We analyzed different shapes of interaction functions. The results suggest that the interaction function in equation (20) represents fairly well our idea of interaction and can be easily tuned. In figure 1, the shape of the function changing $p$ is shown, while in figure 2, the interaction functions are shown for different levels of strength $\lambda$. While $\lambda$ increases during the training, we fix the value of $p = 5$.

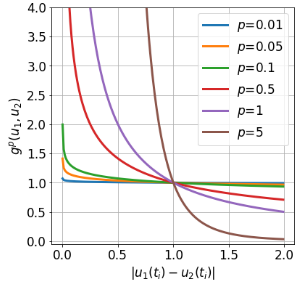


Figure 1: Interaction function at different $p$ values at $\lambda = 1$

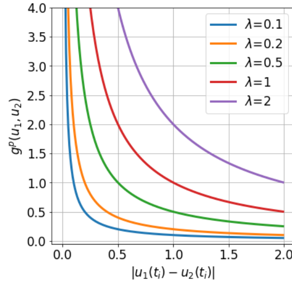


Figure 2: Interaction function at different $\lambda$ values at $p = 1$

### Quantitative results

The proposed approach is able to successfully find the distinct solutions corresponding to the analytical ones (equations (11), (14), and (16)), up to a threshold of $10^{-5}$, calculated as mean squared error for 10000 test points equally spaced in the domains. In figures 3, 4, and 5, the solutions of the three problems are plotted. The real solutions and the ones obtained with the new algorithm are in very good agreement, their difference is not visible.

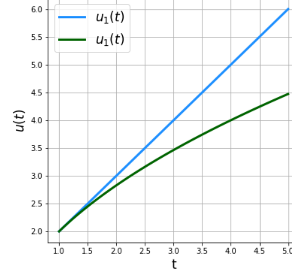


Figure 3: Solutions of Clairaut equation, first problem (equation 11)

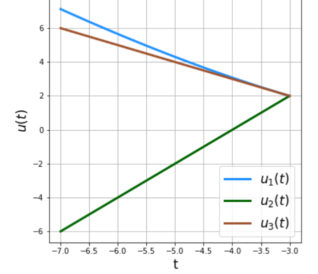


Figure 4: Solutions of Clairaut equation, second problem (equation 14)

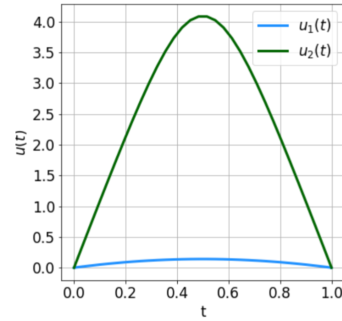


Figure 5: Solutions of Bratu problem (equation 16)

### Training phases

Figures 6 and 7 show how the solutions evolve in different training phases on the Clairaut equation, when the loss function changes from equation (18) to equation (21). Figure 6 shows the functions learned after $n_i = 1000$ epochs. They have not already converged, but we note that they are converging to the same minima. Figure 7 shows the functions learned in $n_f = 1000$ epochs after the first training phase. The functions learned are not satisfying the differential equation yet, but are distinct enough to converge to different minima that represents the two solutions of the differential equation. The proposed algorithm doesn't guarantee that the solutions will be far enough to reach different minima, if they exist, but increases the chances of not converging to the same one. After the third training phase, the solutions converged (see figure 3).

In figure 8, we plot an example of loss function from the Clairaut problem (problem 1) . The blue line represents the loss in equation (18), while the green one is the total

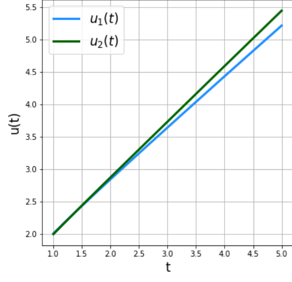


Figure 6: Two solutions of Clairaut equation (problem 1), after the first training phase

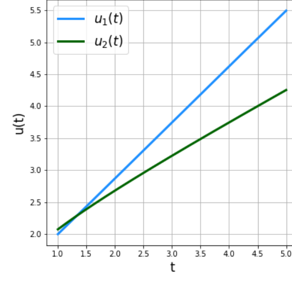


Figure 7: Two solutions of Clairaut equation (problem 1), after the second training phase

loss (equation (21)). We notice that they coincide before $n_i = 1000$ and after $n_i + n_f = 2000$, while they are different elsewhere where the algorithm tries to minimize the loss with the interaction term included.

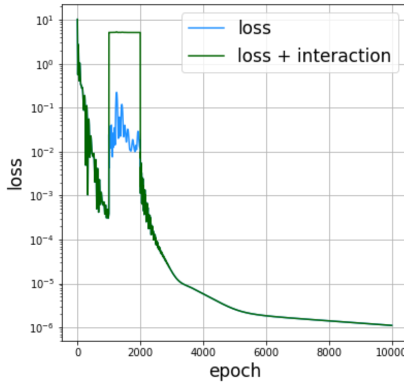


Figure 8: Validation loss during training

## Selection of Hyperparameters

The algorithm described above (algorithm 1) requires as inputs a set of hyper-parameters, that we describe and analyze in this section:

- $\lambda_0$ is the initial strength of the interaction. This is not a sensitive parameter. If not properly tuned, it will only slow down the algorithm without any severe impact on the accuracy of the solutions. The only caution needed is to select it small enough to make the algorithm converge, otherwise the second training phase could move the solutions too far from the minima not to be able to properly converge back to the real solutions. We set this value to $10^{-1}$;
- $k$ is the strength increase and is usually set to 2 in order to double the strength of the interaction at every iteration;
- $\lambda_M$ is the maximum strength to try, usually set as a stopping condition. We should be careful to set it not too small, since the second training phase should separate the solutions enough to make them converge to different minima, but not too high in order to not spend to much time in the training phase (the higher $\lambda_M$, the higher the number of iterations of the loop). We set this value to 10;
- $loss_M$ is the maximum value of the loss so that we know the algorithm converged (the functions learned satisfy the differential equation). We set it to $10^{-4}$;
- $D_m$ is the most important hyper parameter to set since it defines if two learned functions are the same or not. If this value is too low, then two functions will be considered different even if they are actually the same, while if it is set too high then two real solutions too close to each other will be viewed as the same leading to an error. This parameter is analyzed in more detail in the following section.

**Analysis of $D_m$** In figure 9, we show how a wrong selection of $D_m$ leads to wrong results. If we set $D_m = 10^{-2}$, a value too low, the algorithm finds more solutions than the real ones. All the functions shown in the figure have loss less than $loss_M$ and are therefore considered solutions of the differential equation. However, they are distant more than $D_m$ from each other so they are not considered the same solution.

If we pick $D_m$ too high, the distinct solutions could be considered as the same and the algorithm will not be able to detect both of them. For example, since for Clairaut equation (problem 1), the average distance of the two solutions is $D^* \approx 0.61$ in the range $[1, 5]$, setting $D_m \geq D^*$ will lead to wrong results.

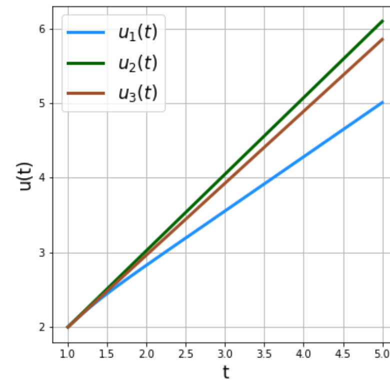


Figure 9: Example of $D_m$ value set too low for Clairaut equation (problem 1)

## Conclusion

In this paper, we designed a novel algorithm to find all possible solutions of a differential equation with non-unique solutions using neural networks. We tested the accuracy applying it to three simple nonlinear differential equations that we

know admit more than one possible solution. The method successfully finds the solutions with high accuracy. We tested different interaction functions and hyper-parameters and we verified the robustness of the approach. We expect this algorithm to work also for more challenging ODEs, if an accurate selection of hyper-parameters is performed. Future work will involve not only application and analysis of the proposed algorithm to higher dimension ODEs and systems of ODEs, but also the exploitation of more appropriate architectures such as dynamical systems like ResNets (He et al. 2016).

## References


Alnæs, M.; Blechta, J.; Hake, J.; Johansson, A.; Kehlet, B.; Logg, A.; Richardson, C.; Ring, J.; Rognes, M. E.; and Wells, G. N. 2015. The fenics project version 1.5. *Archive of Numerical Software* 3(100).

Baymani, M.; Kerayechian, A.; and Effati, S. 2010. Artificial neural networks approach for solving stokes problem. *Applied Mathematics* 1:288–292.

Bratu, G. 1914. Sur les équations intégrales non linéaires. *Bulletin de la Société Mathématique de France* 42:113–142.

Burns, K. J.; Vasil, G. M.; Oishi, J. S.; Lecoanet, D.; and Brown, B. 2016. Dedalus: Flexible framework for spectrally solving differential equations. *Astrophysics Source Code Library*.

Chen, T. Q.; Rubanova, Y.; Bettencourt, J.; and Duvenaud, D. K. 2018. Neural ordinary differential equations. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc. 6571–6583.

Corliss, G.; Faure, C.; Griewank, A.; Hascoet, L.; and Naumann, U. 2002. *Automatic differentiation of algorithms: from simulation to optimization*, volume 1. Springer Science & Business Media.

E, W.; Han, J.; and Jentzen, A. 2017. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in Mathematics and Statistics* 5(4):349–380.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ince, E. 1956. *Ordinary Differential Equations*. Dover Books on Mathematics. Dover Publications.

Karkowski, J. 2013. Numerical experiments with the bratu equation in one, two and three dimensions. *Computational and Applied Mathematics* 32(2):231–244.

Kumar, M., and Yadav, N. 2011. Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: A survey. *Computers and Mathematics with Applications* 62(10):3796 – 3811.

Lagaris, I.; Likas, A.; and Fotiadis, D. 1997. Artificial neural network methods in quantum mechanics. *Computer Physics Communications* 104(1):1 – 14.

Lagaris, I. E.; Likas, A.; and Fotiadis, D. I. 1998. Artificial neural networks for solving ordinary and partial differential equations. *Trans. Neur. Netw.* 9(5):987–1000.

Lagaris, I. E.; Likas, A.; and Papageorgiou, D. G. 1998. Neural network methods for boundary value problems defined in arbitrarily shaped domains. *CoRR* cs.NE/9812003.

Mattheakis, M.; Protopapas, P.; Sondak, D.; Giovanni, M. D.; and Kaxiras, E. 2019. Physical symmetries embedded in neural networks.

Raissi, M.; Perdikaris, P.; and 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.

Seefeldt, B.; Sondak, D.; Hensinger, D. M.; Phipps, E. T.; Foucar, J. G.; Pawlowski, R. P.; Cyr, E. C.; Shadid, J. N.; Smith, T. M.; Weber, P. D.; et al. 2017. Drekar v. 2.0. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Sirignano, J., and Spiliopoulos, K. 2018. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics* 375:1339 – 1364.

Zhu, M.; Chang, B.; and Fu, C. 2018. Convolutional neural networks combined with runge-kutta methods. *arXiv preprint arXiv:1802.08831*.