One-Shot Transfer Learning for Nonlinear ODEs

Anonymous Author(s)
Affiliation
Address
email

Abstract

We introduce a generalizable approach that combines perturbation method and one-shot transfer learning to solve nonlinear ODEs with a single polynomial term, using Physics-Informed Neural Networks (PINNs). Our method transforms non-linear ODEs into linear ODE systems, trains a PINN across varied conditions, and offers a closed-form solution for new instances within the same non-linear ODE class. We demonstrate the effectiveness of this approach on the Duffing equation and suggest its applicability to similarly structured PDEs and ODE systems.

1 Introduction

Differential equations are crucial in scientific modeling, traditionally solved by methods such as Runge-Kutta and finite element analysis. Recently, Physics-Informed Neural Networks (PINNs) have shown promise in solving ODEs and PDEs by leveraging neural network capabilities (see (6), (8), (10), (15)and reference within). However, computational cost remains a barrier, as PINNs cannot be generalized across different instances of similar equation types (9), (3), a workaround is to train for multiple instances of same equation (4). To address this, we propose a novel hybrid approach that combines the perturbation method with one-shot transfer learning on PINNs (17) to efficiently and accurately solve non-linear ODEs of same type.

Related Work Introduced in 1998, neural networks for solving differential equations paved the way for today's Physics-Informed Neural Networks (PINNs) (10), NeuroDiffEq and DeepXDE (13; 2) are two popular software programs that employ PINNs. PINNs have gained popularity and find applications in many domains today, a review of current state-of-the-art applications of PINNs can be found in (6), (11) and (8). PINNs have shown tremendous success in solving complex problems whose analytical solutions don't exist (1), however they perform poorly with generalizing solutions. Work related to adding physical constraints in NN structure (14), bounding errors on PINNs (12), characterizing and mitigating failure modes (9) and improving uncertainty quantification on Bayesian PINNs (5) has been important in increasing reliability of PINNs. PINNs handle interpolation problems very well, however face trouble with extrapolation. Transfer learning (TL) methods may be a potential solution, a study of effectiveness of TL can be found in this work (3) and an application can be found here (16). In (16; 18), a multi-headed neural network was used to learn the "latent space" of a class of differential equations. Researchers have developed a transfer learning approach to solve linear differential equations in "one-shot" (17). Extending this to non-linear equations is not possible without modifications since non-linear equations have a non-quadratic loss function which cannot be optimized analytically in one-shot. The head weights have to be learned through an iterative process, such as gradient descent. Our study fills this gap in the literature by extending one-shot transfer learning to non-linear ODEs using perturbation method.

Submitted to the DLDE-III Workshop in the 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

2 Methodology

We are interested in solving non-linear ODEs with a single polynomial non-linear term of the following form:

$$Dx + \epsilon x^q = f(t) \tag{1}$$

where D is a differential operator of the form $D = \sum_{j=0}^{m} g_j \frac{d^j}{dt^j}$ and the RHS is a time-dependent forcing function. Note that we define $g_0 \frac{d^0}{dt^0} x = g_0 x$. The equation is also subject to a boundary conditions: $x(t=0) = x^*$ and $\frac{d^j}{dt^j} x(t=0) = x^{(j)*}$ for $j = 1, 2, ..., m-1$. Our framework employs perturbation and one-shot transfer learning to solve a specific class of non-linear ODEs. We plan to extend this to handle systems of ODEs and PDEs.

Perturbation Method

As mentioned above, non-linear ordinary differential equations (ODEs) do not possess analytical solutions to their loss functions with respect to the weights of the linear output layer, which is necessary for one-shot transfer learning.

In order to remove the non-linearity in the equation, we approximate the non-linear term ϵx^q as a composition of functions (7). Assume $x = \sum_{i=0}^{\infty} \epsilon^i x_i$, where x_i are unknown functions of t. We approximate x with only p terms as: $x \approx \sum_{i=0}^{p} \epsilon^i x_i$. It is important to note that this truncated expansion of x is only meaningful when the magnitude of ϵ is less than 1. Furthermore, the p-term approximation is more precise when the magnitude of ϵ is smaller. Fortunately, in most cases, we can adjust the equation by scaling to reduce the magnitude of ϵ. When we substitute the p-term approximation into 1 and expand using multinomial theorem , we obtain:

$$\sum_{i=0}^{p} \epsilon^i D x_i + \epsilon \left(\sum_{k_0+k_1+...+k_p=q} \frac{q!}{k_1!k_2!...k_p!} \epsilon^{\sum_{i=0}^{p} i k_i} \prod_{i=0}^{p} x_i^{k_i} \right) = f \tag{2}$$

The LHS of equation (3) is a polynomial of ϵ. Since it holds for all values of ϵ, the 0^{th} order term of LHS should be equal to the RHS and the coefficients of all higher-order terms of ϵ should all be 0. Therefore, for the 0^{th} order, we obtain: $Dx_0 = f$, and more generally for the j^{th} order, where $1 \leq j \leq p$, we have:

$$Dx_j = - \sum_{\substack{k_0+k_1+...+k_p=q \\ \sum_{i=0}^{p} i k_i = j-1}} \frac{q!}{k_1!k_2!...k_p!} \prod_{i=0}^{p} x_i^{k_i} := f_j \tag{3}$$

where f_j is the forcing function for j^{th} ODE. The first few terms for this expansion look like:

$$Dx_0 = f \quad Dx_1 = -x_0^2 \quad Dx_2 = -2x_0x_1 \quad ... \tag{4}$$

The forcing function f_j depends only on previously solved x_i's. Therefore, (1) is reduced to a series of $p+1$ linear ODEs of the same form: $Dx_j = f_j$ that can be solved iteratively. There are a variety of ways to ensure that the initial boundary conditions are met. The main concept is to fix all $p+1$ boundary conditions to be the same, so that the total solution's, x, boundary condition is satisfied; that is, for all $k = 0, 1, ...p$, $x_k(t=0) = x^* / \sum_{i=0}^{p} \epsilon^i$ and $\frac{d^j}{dt^j} x_k(t=0) = x^{(j)*} / \sum_{i=0}^{p} \epsilon^i$.

Multi-head Fully Connected Neural Network

As established earlier, solving the non-linear ODE is equivalent to solving a sequence of linear ODEs in the form $Dx = f$. To minimize computational complexity, we transform higher-order differential equations into first-order equations by introducing $m-1$ additional dependent variables for an m^{th} order differential equation. Let $\boldsymbol{u} = [x, x^{(1)}, x^{(2)}, \ldots, x^{(m-1)}]^T$ be a function mapping from $\mathbb{R}$ to $\mathbb{R}^m$, where $x^{(1)} = \dot{x}$ and $x^{(i)} = \dot{x}^{(i-1)}$ for $i = 2, \ldots, m-1$. The equation $Dx = f$ is then reduced to a first-order linear ODE system (see Appendix C for details).

$$\begin{cases} \dot{x} - x^{(1)} = 0 \\ \dot{x}^{(i-1)} - x^{(i)} = 0, i = 2, 3, 4, ..., m-1 \\ g_0 x + \sum_{i=1}^{m-1} g_i x^{(i)} + g_m \dot{x}^{(m-1)} = f \end{cases} \tag{5}$$

Equation 5 is equivalent to: $B\dot{\boldsymbol{u}} + A\boldsymbol{u} = F_j$ with boundary conditions $\boldsymbol{u}(t=0) = \boldsymbol{u}^* \in \mathbb{R}^m$, where $\dot{\boldsymbol{u}} = [\dot{x}, \dot{x}^{(1)}, \dot{x}^{(2)}, ..., \dot{x}^{(m-1)}]^T$ and $F_j = [0, 0, ..., f_j]^T$. A detailed description of the matrices A and B can be found in Appendix A.

We create a fully connected neural network with K heads in two parts to approximate the K functions $\{\boldsymbol{u}_k\}_{k=1}^K$. The first part connects a 1D input to hidden layers, with the last layer having dimension mh. The activations of the last hidden layer are reshaped into a matrix $H \in \mathbb{R}^{m \times h}$, which reflects the hidden state of the ODE class and is then passed to the second part of the network. H connects to K heads, each associated with a linear ODE system. The output of each head is $\hat{\boldsymbol{u}}_k = HW_k \in \mathbb{R}^m$. A diagram of the general structure of the network can be found in Appendix B. The loss function for the k^{th} head in the network is defined over a sampled data set T as :

$$L_k = \frac{1}{mN} \sum_{t \in T} ||B_k \dot{\hat{\boldsymbol{u}}}_k(t) + A_k \hat{\boldsymbol{u}}_k(t) - F_k(t)||_2^2 + \frac{1}{m} ||\hat{\boldsymbol{u}}_k(0) - \boldsymbol{u}_k^*||_2^2 \tag{6}$$

where $\boldsymbol{u}_k^*$ is the boundary condition of the k^{th} ODE. The total loss of the network is defined as: $L_{total} = \frac{1}{K} \sum_{k=1}^K L_k$ The purpose of training this neural network is to learn the latent space for one class of linear ODEs. Ideally, the larger K is, the better the learning of latent space and hence a generalization to a wider range of parameters.

One-Shot Transfer Learning
After training, we freeze the weights in the hidden layers. When encountering a new ODE of the same class, we only use one head, and the weights in this head can be calculated analytically in one shot. Suppose W is the time-independent network parameter in the last layer. The network now becomes: $\hat{\boldsymbol{u}}(t) = H(t)W$. We get the loss for this single-head neural network by substituting $\hat{\boldsymbol{u}}_k = \hat{\boldsymbol{u}}(t) = H(t)W$ into Eq. 6:

$$L = \frac{1}{mN} \sum_{t \in T}^{n} ||B\dot{H}_t W + AH_t W - F(t)||_2^2 + \frac{1}{m} ||H_0 W - \boldsymbol{u}^*||_2^2 \tag{7}$$

where H_t is the hidden state of the network evaluated at t and H_0 is the hidden state at the boundary. Differentiating L from W and setting $\frac{dL}{dW} = 0$, we obtain (details omitted):

$$W = M^{-1} \left(H_0^T \boldsymbol{u}^* + \frac{1}{N} \sum_{t \in T} B\dot{H}_t F(t) + \frac{1}{N} \sum_{t \in T} H_t^T A^T F(t) \right) \tag{8}$$

$$M = \frac{1}{N} \sum_{t \in T} (\dot{H}_t^T B^T B \dot{H}_t + \dot{H}_t^T B^T A H_t + H_t^T A^T B \dot{H}_t + H_t^T A^T A H_t) + H_0^T H_0 \tag{9}$$

For a fixed Duffing equation, the matrices A and B are fixed for all its $p+1$ reduced ODE systems. Thus, M only needs to be computed and inverted once. We only need to update the forcing function F in 8 which iteratively depends on previous solutions. By reusing the first part of the neural network and using only one head, we optimally and iteratively compute the head parameters for each ODE system to solve them.

3 Result

We applied our proposed methodology to the 1D Duffing equation. The Duffing equation we are interested in 10 is a second order non-linear ODE with five parameters: $\delta, \alpha, \beta, \gamma, \omega$ and one boundary condition $x(0) = x^*$. All higher-order boudnary conditions are set to 0.

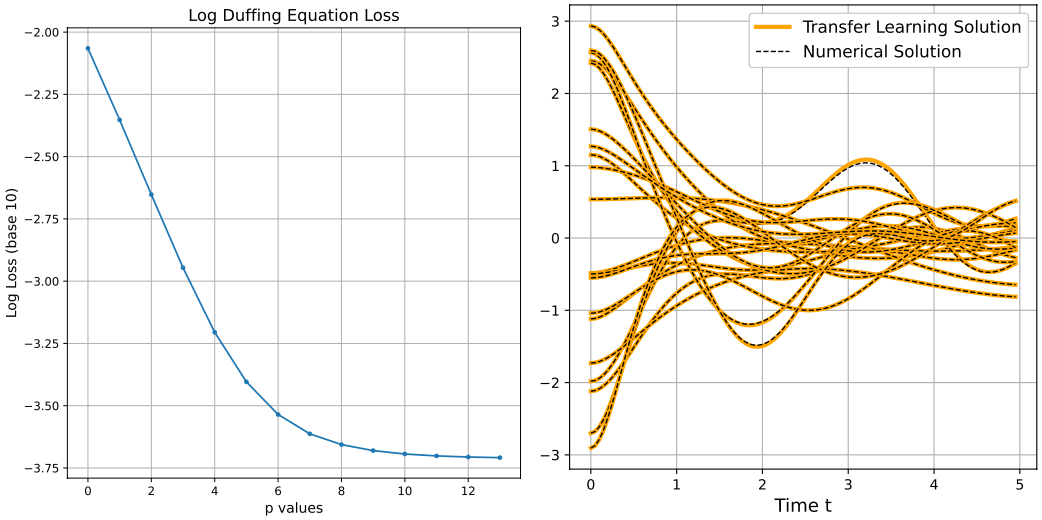
$$\frac{d^2x}{dt^2} + \delta \frac{dx}{dt} + \alpha x + \beta x^3 = \gamma cos(\omega t) \tag{10}$$

Using our framework, we first utilized the perturbation method and introduced new variables to reduce the Duffing equation to a series of $p+1$ first-order linear ODE systems of the form: $\dot{u}_i +$

$Au_i = F_i$. We then built a network described in Section 2.2 with 10 heads. Each head represents a unique parameter setting. The specific details of the network structure can be found in Appendix B. The 10 parameter sets are uniformly randomly generated in the following range:

$$\gamma \in (0.5, 3), \omega \in (0.5, 3), \alpha \in (0.5, 4.5), \delta \in (0.5, 4.5), \boldsymbol{u}_1^* \in (-3, 3) \tag{11}$$

and $\boldsymbol{u}_2^* = 0$. After training (details in Appendix B), the network can accurately solve the 10 systems, and it acquires a significant understanding of the latent space of the not-linear ODE. We then test our method on an unseen Duffing equation. We measure the performance of the TL solution by computing the ODE loss of the Duffing equation. We used 14 different values of p to solve and approximate the solution. As shown in Figure 1(a), as the p value increases, the Duffing ODE loss decreases to around $10^{-3.75}$. The elbow shape can be used to figure out how many terms should be included in the perturbation expansion p.



(a) Log Duffing Equation Loss vs. p Values

(b) Numerical and TL Solutions of 20 Randon In-distribution Duffing Equation

Figure 1

We also test our method by comparing the transfer learning solutions with numerical solutions (explicit Runge-Kutta method of order 8) on 20 randomly generated Duffing equations in the same parameter range 11 (we fix $\beta = 0.5$ and $p = 12$). Each Duffing equation can be solved in seconds. (Details in Appendix B) As shown in Figure 2(b), the 20 transfer learning solutions align almost perfectly with the numerical solutions, indicating that our methodology is very effective on equations in the same parameter range 11.

4 Conclusion

We introduced a framework using perturbation and one-shot transfer learning on PINNs to efficiently solve non-linear ODEs. We reduced non-linear ODEs to linear ODEs, trained a neural network with k heads to handle them, and derived a formula for network weights. This approach allows us to solve various non-linear ODEs of the same form with a single trained network. Future work aims to extend this methodology to various non-linear ODE and PDE systems.

Our work should be considered as a starting point for this methodology. Future work is needed to extend the framework to non-linear ODE and PDE systems with various non-linearity forms.

References

[1] CHANTADA, A. T., LANDAU, S. J., PROTOPAPAS, P., SCÓCCOLA, C. G., AND GARRAFFO, C. Cosmology-informed neural networks to solve the background dynamics of the universe. *Phys. Rev. D 107* (Mar 2023), 063523.

[2] CHEN, FEIYU, E. A. Neurodiffeq: A python package for solving differential equations with neural networks. *Journal of Open Source Software* (2020).

[3] FESSER, L., QIU, R., AND D'AMICO-WONG, L. Understanding and mitigating extrapolation failures in physics-informed neural networks, 2023.

[4] FLAMANT, C., PROTOPAPAS, P., AND SONDAK, D. Solving differential equations using neural network solution bundles, 2020.

[5] GRAF, O., FLORES, P., PROTOPAPAS, P., AND PICHARA, K. Error-aware b-pinns: Improving uncertainty quantification in bayesian physics-informed neural networks, 2022.

[6] HAO, Z., LIU, S., ZHANG, Y., YING, C., FENG, Y., SU, H., AND ZHU, J. Physics-informed machine learning: A survey on problems, methods and applications, 2023.

[7] J. KEVORKIAN, J. D. C. *Perturbation Methods in Applied Mathematics*. Springer New York, NY, 2010.

[8] KARNIADAKIS, G. E., KEVREKIDIS, I. G., LU, L., PERDIKARIS, P., WANG, S., AND YANG, L. Physics-informed machine learning. *Nature Reviews Physics* (2021).

[9] KRISHNAPRIYAN, A., GHOLAMI, A., ZHE, S., KIRBY, R., AND MAHONEY, M. W. Characterizing possible failure modes in physics-informed neural networks. In *Advances in Neural Information Processing Systems* (2021), M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, Curran Associates, Inc., pp. 26548–26560.

[10] LAGARIS, I., LIKAS, A., AND FOTIADIS, D. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks 9*, 5 (1998), 987–1000.

[11] LAWAL, Z. K., YASSIN, H., LAI, D. T. C., AND CHE IDRIS, A. Physics-informed neural network (pinn) evolution and beyond: A systematic literature review and bibliometric analysis. *Big Data and Cognitive Computing 6*, 4 (2022).

[12] LIU, S., HUANG, X., AND PROTOPAPAS, P. Residual-based error bound for physics-informed neural networks, 2023.

[13] LU, LU, E. A. Deepxde: A deep learning library for solving differential equations. *CoRR* (2019). `https://arxiv.org/abs/1907.04502`.

[14] MATTHEAKIS, M., PROTOPAPAS, P., SONDAK, D., GIOVANNI, M. D., AND KAXIRAS, E. Physical symmetries embedded in neural networks, 2020.

[15] NASCIMENTO, R. G., FRICKE, K., AND VIANA, F. A. A tutorial on solving ordinary differential equations using python and hybrid physics-informed neural network. *Engineering Applications of Artificial Intelligence 96* (2020), 103996.

[16] PELLEGRIN, R., BULLWINKEL, B., MATTHEAKIS, M., AND PROTOPAPAS, P. Transfer learning with physics-informed neural networks for efficient simulation of branched flows. *arXiv preprint arXiv:2211.00214* (2022).

[17] PROTOPAPAS, P., E. A. One-shot transfer learning of physics-informed neural networks. *CoRR* (2021). `https://arxiv.org/abs/2110.11286`.

[18] ZOU, Z., AND KARNIADAKIS, G. E. L-hydra: Multi-head physics-informed neural networks, 2023.

Supplementary Material

Appendix A

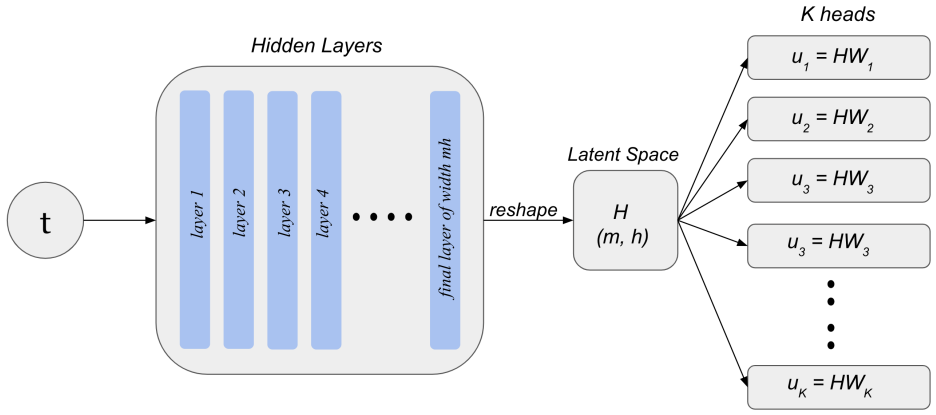
The matrix B 12 is an m by m diagonal matrix in which the first m-1 diagonals are all 1 and the last diagonal is g_m. Matrix A 12 is an m by m matrix whose second upper diagonal entries are all -1 and last row is $[g_0, g_1, ..., g_{m-1}]$.

$$B = \begin{bmatrix} I_{m-1\times m-1} & \vec{0} \\ \vec{0}^T & g_m \end{bmatrix}, A = \begin{bmatrix} 0 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & -1 & \cdots & \cdots & 0 \\ 0 & 0 & 0 & -1 & \cdots & \vdots \\ \vdots & \vdots & \vdots & 0 & \ddots & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 \\ g_0 & g_1 & g_2 & \cdots & \cdots & g_{m-1} \end{bmatrix} \tag{12}$$

Appendix B

Here we show the diagram of the general multi-head PINN structure. In our real implementation to solve Duffing Equation, the network has 4 layers of hidden layers with width 256, 256, 256, 512 respectively. Hidden layers are all connected by $tanh$ activation functions. The activations of the last hidden layer is reshaped into a matrix $H \in \mathbb{R}^{2\times 256}$. The matrix H is connected to 10 heads of dimension 2 by a linear transform.

Figure 2: General Multihead PINN structure



To train the network, we used Adam optimizer for 5000 iterations with an initial learning rate of 2×10^{-4}. Note that we applied an exponential decay to the learning rate: the learning rate is multiplied by a factor of 0.96 each 100 iterations. 200 random data points are uniformly generated in the domain $(0, 5)$ in each iteration as a sample set to compute the ODE loss. After 5000 iterations, the total loss (the sum of ODE loss and boundary loss) is reduced below 10^{-4}.

We ran our code on google colab using an Intel Xeon CPU with 2 vCPUs (virtual CPUs) and 51GB of RAM. Solving an unseen Duffing equations generally takes $0.5p + 1$ seconds, where $p + 1$ is the number of linear ODE systems we reduce to using perturbation method and the additional 1 s is the time to compute and invert the matrix M.

Appendix C

To reduce the ODE $Dx = f$ to first order, we introduce $m-1$ time-dependent variables: $\{x^{(i)}\}_{i=1}^{m-1}$ to form a function $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^m$. $\boldsymbol{u} = [x, x^{(1)}, x^{(2)}, ..., x^{(m-1)}]^T$. $Dx = f$ is equivalent to:

$$g_0 x + \sum_{i=1}^{m} g_i \frac{d^i}{dt^i} x = f \tag{13}$$

The introduced variables are defined as: $x^{(1)} = \dot{x}$ and $x^{(i)} = \dot{x}^{(i-1)}$ for $i = 2, 3, ..., m-1$. Expand these relations iteratively, we obtain $\frac{d^i}{dt^i} x = x^{(i)}$ for $i = 1, 2, ..., m-1$, plug into 13, we obtain:

$$g_0 x + \sum_{i=1}^{m-1} g_i x^{(i)} + g_m \dot{x}^{(m-1)} = f \tag{14}$$

The definitions of these $m-1$ variables introduces another $m-1$ constraints, together with 14, we recover the linear ODE system in 5.