Generalized One-Shot Transfer Learning of Linear Ordinary and Partial Differential Equations

Harmit Raval, Pavlos Protopapas
Harvard University
{hraval, pprotopapas}@g.harvard.edu

Abstract

We present a generalizable methodology to perform "one-shot" transfer learning on systems of linear ordinary and partial differential equations using physics-informed neural networks (PINNs). PINNS have attracted researchers as an avenue through which both data and studied physical constraints can be leveraged in learning solutions to differential equations. Despite their benefits, PINNs are currently limited by the computational costs needed to train such networks on different but related tasks. Transfer learning addresses this drawback. In this work, we present a generalizable methodology to perform "one-shot" transfer learning on linear systems of equations. First, we describe a process to train PINNs on equations with varying conditions across multiple "heads". Second, we show how this multi-headed training process can be used to yield a latent space representation of a particular differential equation form. Third, we derive closed-form formulas, which represent generalized network weights that minimize the loss function. Finally, we demonstrate how the learned latent representation and derived network weights can be utilized to instantaneously transfer learn solutions to equations, demonstrating the ability to quickly solve many systems of equations in a variety of environments.

1 Introduction

Given the wide-ranging significance of solving differential equations, a great deal of research has been done in developing numerical methods to solve them. While these traditional numerical methods have been proven to perform well and yield stable solutions with a high degree of fidelity, neural networks offer a much more attractive alternative [3]. Specifically, PINNs are neural networks that can use both the data and the studied physical laws to learn the solutions of equations. PINNs offer various benefits over traditional numerical methods by eliminating the need for a numerical integrator [8], generating continuous and differentiable solutions [8], improving accuracy in high dimensions [5] [6], easily incorporating data [4], and maintaining small memory footprints [3].

Despite the numerous benefits PINNs offer, one limitation is the computational expense required to train networks on different but closely related tasks [2]. Fortunately, this drawback can be addressed via transfer learning. In fact, researchers have recently shown that a PINN pre-trained on a family of differential equations can be efficiently reused to solve new equations in one-shot [3]. Our work generalizes this idea of "one-shot" transfer learning to any system of linear ODEs and PDEs.

2 Related Work

The use of neural networks to solve differential equations was first introduced in 1998, when researchers analytically computed the partial derivatives of a neural network output with respect to inputs [8]. These researchers demonstrated that a neural network can represent a differential equation when the network architecture is known and when given access to the equation solution

DLDE-III Workshop in the 37th Conference on Neural Information Processing Systems (NeurIPS 2023).

and derivatives. This work enabled the development and wide-spread use of PINNs which rely on backpropagation as opposed to analytical computation of partial derivatives[10] [7]. Currently, PINNs have been used in a variety of different applications, including the solution of high-dimensional PDEs [11].

Furthering the use of PINNs in solving differential equations, various software packages have been created that use neural networks and backpropagation to approximate solutions such as NeuroDiffEq and DeepXDE [1] [9]. Until now, one area of work that has remained largely uninvestigated in solving differential equations has been transfer learning. Recently, researchers have developed an approach to solve differential equations in "one-shot" [3]. Specifically, their work presents a framework for transfer learning PINNs that enables one-shot inference for certain ODEs and PDEs, yielding highly accurate solutions instantaneously without retraining an entire network.

3 Methodology

3.1 Training Procedure

We first discuss a network training approach that generalizes to any number of equations and is flexible enough to work for both ODEs and PDEs. The goal of the training procedure is to learn a latent space representation, H, of equations. At a high level, the training loss function minimizes the residuals of the differential equations and satisfies the initial and/or Dirichlet boundary conditions. For ODEs, we formulate the loss as the following convex function:

$$L_{ode} = \frac{1}{n}\sum_{t}\left((\dot{u} + Au - f)^2\right) + \frac{1}{n}\sum_{t}\left(u(t=0) - u_0\right)^2 \tag{1}$$

Similarly, for PDEs, the convex loss is constructed as shown below:

$$L_{pde} = \frac{1}{n}\sum_{x\in\text{Batch}}(((D_u G^T) \circ I)\overrightarrow{1} + Au - f)^2 + \frac{1}{n}\sum_{x\in\text{Batch}}\left(u(x) - u_0(x)\right)^2 \tag{2}$$

In Equations 1 and 2, u is the system solution, A represents the coefficient matrix for u, f is the forcing function, and n is the batch size. Additionally, in Equation 2, D_u is the partial derivative matrix, G is the coefficient matrix for D_u, and "$\circ$" represents the Hadamard product. In each loss function, the first summation term represents the differential equation that needs to be satisfied and the second summation term represents the initial or Dirichlet boundary condition. In order to learn the latent space H, we leverage a multi-headed training approach.

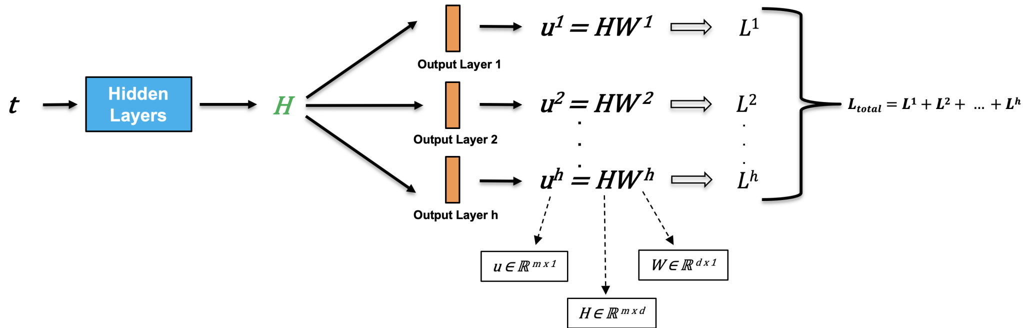

Figure 1: Diagram of multi-headed training procedure for ODEs. For each "head", the same input t is fed through the hidden layers (blue). Each "head" corresponds to its own set of training conditions for which a solution u^i is found after the input flows through the corresponding output layer (orange).

In the multi-headed training approach, the network accepts an input time t for an ODE (or multiple inputs in the case of PDEs) and proceeds to flow the input through the hidden layers. After the final hidden layer's activation function is applied, H is outputted. H has a shape of $m \times d$, where m

represents the number of equations and d is the number of nodes in the last hidden layer. The output from H is sent through h different output layers (different "heads"). Each "head" contains its own set of weights W^i as each corresponds to a different set of equation coefficients and initial or boundary conditions during training. Hence, there are h network outputs u^i, each of which can be computed by multiplying H by the corresponding W^i. All paths through the network lead to their own loss functions L^i, which are summed together to form an overall L_{total}. L_{total} is what is minimized while training. Figure 1 illustrates this training process.

The key observation to note in Figure 1 is that all h "heads" share the same H. In other words, by using varying training conditions across all the "heads" and a single H, the diversity of the training conditions will be encoded in this one matrix. Using a larger H yields a richer latent space encoding.

3.2 Computing General Network Weights

In order to transfer learn solutions u to ODEs, we need two components: H and W_0. In Section 3.1, we illustrated how to recover a generalized H. Now we discuss how to find a set of generalized network weights W_0, different from the W learned during training, in order to compute u. To derive W_0 for ODEs, we start with the loss function in Equation 1 and substitute in $u = HW_0$:

$$L_{ode} = \frac{1}{n}\sum_t \left(\dot{H}W_0 + AHW_0 - f\right)^2 + \frac{1}{n}\sum_t \left(H_0 W_0 - u_0\right)^2$$

When we expand the quadratic terms, take the loss gradient with respect to these weights, and solve for W_0, we arrive at the following closed-form representation of W_0 for ODEs:

$$\begin{gathered} W_0 = M^{-1}\left(H_0^T u_0 + \frac{1}{n}\sum_t \dot{H} f + \frac{1}{n}\sum_t H^T A^T f\right) \\ \text{where } M = \left[\frac{1}{n}\sum_t (\dot{H}^T\dot{H} + \dot{H}^T AH + H^T A^T \dot{H} + H^T A^T AH) + H_0^T H_0\right] \end{gathered} \tag{3}$$

For PDEs, we follow an analogous process with Equation 2 and find the representation below for W_0:

$$\begin{gathered} W_0 = M^{-1}\left(\frac{1}{n}\sum_{x\in \text{Batch}} S^T f + \frac{1}{n}\sum_{x\in \text{Batch}} H^T A^T f + \frac{1}{n}\sum_{x\in \text{Batch}} H_0^T u_0\right) \\ \text{where } M = \left[\frac{1}{n}\sum_{x\in \text{Batch}} (S^T AH + H^T A^T S + S^T S + H^T A^T AH + H_0^T H_0)\right] \end{gathered} \tag{4}$$

In Equation 4, S is a matrix such that $S_{ij} = \sum_k g_{ik}\frac{\partial H_{ij}}{\partial x_k}$ where g_{ik} refers to each element of the coefficient matrix G and $\frac{\partial H_{ij}}{\partial x_k}$ is the partial derivative of each element of the latent space H with respect to each of the k independent variables.

3.2.1 Performing One-Shot Transfer Learning

In order to perform "one-shot" transfer learning, the network training needs only be done once, after which H must be saved. Until a new equation form is to be solved, the values of u_0, A, G, and f can freely be changed to new values, unseen during training. After choosing new values for these parameters, Equation 3 or 4 can be used to solve for W_0. With H and W_0, the desired differential equation solutions can be computed in one-shot by following $u = HW_0$. It is important to keep in mind that the further the chosen values deviate from the training regime, the more error we will accumulate in the transfer-learned solution [1]. Finally, the main computational cost in finding W_0 is inverting M, a $d \times d$ matrix. Generally, M is not large, and therefore, this is usually more efficient than retraining a whole network. If A is constant, M^{-1} can be pre-computed, further reducing computational demands in subsequent transfer learning tasks.

[1] The error accumulation occurs because the representation of H is not perfect after a finite training period.

4 Results

We include transfer learning results for two of the systems that we solved with our “one-shot" transfer learning approach. The problems of interest are a time-dependent linear ODE and a system of coupled linear PDEs, shown in Equations 5 and 6, respectively. In both equations, all “c", “v", and “f" values are parameters that will vary per “head" during training and can be transfer learned.

$$\begin{cases} \dfrac{du}{dt} + ctu = f \\ \quad u(0) = v \end{cases} \tag{5}$$

$$\begin{cases} c_1 \dfrac{\partial u_1}{\partial x_1} + c_2 \dfrac{\partial u_1}{\partial x_2} = f_1, \quad c_3 \dfrac{\partial u_2}{\partial x_1} + c_4 \dfrac{\partial u_2}{\partial x_2} = f_2 \\ \quad u_1(x_1, 0) = c_5 x_1, \quad u_2(x_1, 0) = c_6 x_1 \end{cases} \tag{6}$$

Figures 2 and 3 illustrate the comparison between a reference solution and our transfer learned solution for each system. Significantly, for both examples, each right subplot illustrates that the equation residuals are smaller than 10^{-2}, demonstrating an accurate reconstruction of the reference solutions. The network training parameters' details and exact training conditions used per “head" for both problems can be found in the supplementary material. It should be noted that for the ODE, the reference solution was computed using `scipy.integrate.solve_ivp` [12] and for the PDE, the reference solution was computed analytically.

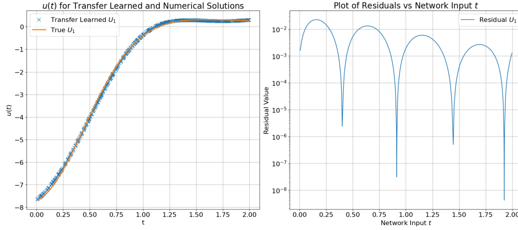

Figure 2: Transfer learning results for Equation 5 after changing u_0 to -10.7 and A to $5.0t$. The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

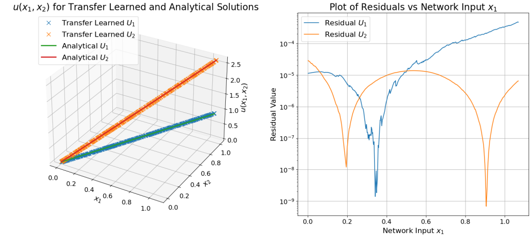

Figure 3: Transfer learning results for Equation 6 after changing G to $\begin{bmatrix} 3.2 & 5.1 \\ 1.4 & 7.3 \end{bmatrix}$. The left subplot shows the transfer learned solutions (blue/orange) versus the true solutions (green/red) and the right subplot shows the network equation residuals.

5 Conclusion

In this work, we have shown how to apply “one-shot" transfer learning to systems of linear ordinary and partial differential equations. Namely, we have described a process to train PINNs on differential equations with varying conditions across multiple “heads". We showed how this thorough training procedure yields a latent space representation, H, for the equations of interest. From here, we derived closed-form formulas to calculate the generalized network weights, W_0, for both linear ODEs and PDEs. Both H and W_0 were used to rapidly find the equation solutions u since $u = HW_0$. With a generalizable training and transfer learning methodology, our work provides the ability to quickly solve many systems of differential equations. Specifically, by being able to adjust parameters such as initial or boundary conditions u_0 and coefficient matrix A, we can construct solutions to equations that exhibit different ranges of values, end behaviors, and curvatures. The power of our methodology lies in the fact that after training a network only once, solving equations with new coefficients and initial conditions simply requires a recalculation of the weights W_0 - no additional fine-tuning is performed. Future work may apply this network training and transfer learning approach to larger scale problems and develop an analogous technique to solve nonlinear systems in order to examine problems that exhibit interesting behaviors.

6 Supplementary Material

6.1 ODE Training Details

Below we include detailed information about the parameter values used in training the network for the ODE example shown in Section 4. Additionally, we specify the A, u_0, and f values used for each “head" during the training procedure.

TRAINING PARAMETER	VALUE
NUMBER OF ITERATIONS	10000
NUMBER OF GRID POINTS	512
GRID SPACE	$[0, 2]$
NUMBER OF HIDDEN LAYERS	3
NODES PER HIDDEN LAYER	$[128, 128, 256]$
OPTIMIZER	SGD (MOMENTUM=0.9)
LEARNING RATE	0.001
ACTIVATION	TANH

Table 1: Network parameters used for training non-constant coefficient ODE shown in Section 4.

HEAD	A	u_0	f
1	$1.0t$	0.12	2.0
2	$1.25t$	0.87	2.0
3	$1.5t$	0.34	2.0
4	$2.0t$	0.75	2.0

Table 2: Head conditions used for training non-constant coefficient ODE shown in Section 4.

6.2 PDE Training Details

Below, we include detailed information about the parameter values used in training the network for the PDE example shown in Section 4. Additionally, we specify the A, G, u_0, and f values used for each “head" during the training procedure.

TRAINING PARAMETER	VALUE
NUMBER OF ITERATIONS	40000
NUMBER OF GRID POINTS	1024
GRID SPACE	$[0, 1] \times [0, 1]$
NUMBER OF HIDDEN LAYERS	5
NODES PER HIDDEN LAYER	$[256, 256, 256, 256, 512]$
OPTIMIZER	ADAM (BETAS=$(0.9, 0.999)$)
LEARNING RATE	0.001
ACTIVATION	SILU

Table 3: Network parameters used for training coupled PDE shown in Section 4.

Head	A	G	u_0	f
1	$\begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 1.0 & 4.0 \\ 1.0 & 9.0 \end{bmatrix}$	$\begin{bmatrix} 2.0x_1 \\ 3.0x_1 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$
2	$\begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 2.3 & 3.0 \\ 4.1 & 5.7 \end{bmatrix}$	$\begin{bmatrix} 3.1x_1 \\ 4.2x_1 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$
3	$\begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 1.2 & 4.3 \\ 1.5 & 6.3 \end{bmatrix}$	$\begin{bmatrix} 0.3x_1 \\ 0.11x_1 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$
4	$\begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 1.1 & 3.4 \\ 2.5 & 8.3 \end{bmatrix}$	$\begin{bmatrix} 1.2x_1 \\ 5.3x_1 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$

Table 4: Head conditions used for training coupled PDE shown in Section 4.

References

[1] Feiyu Chen, David Sondak, Pavlos Protopapas, Marios Mattheakis, Shuheng Liu, Devansh Agarwal, and Marco Di Giovanni. Neurodiffeq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46):1931, 2020.

[2] Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maizar Raissi, and Francesco Piccialli. Scientific machine learning through physics-informed neural networks: Where we are and what's next. *CoRR*, abs/2201.05624, 2022.

[3] Shaan Desai, Marios Mattheakis, Hayden Joy, Pavlos Protopapas, and Stephen J. Roberts. One-shot transfer learning of physics-informed neural networks. *CoRR*, abs/2110.11286, 2021.

[4] Franck Djeumou, Cyrus Neary, Eric Goubault, Sylvie Putot, and Ufuk Topcu. Neural networks with physics-informed architectures and constraints for dynamical systems modeling. In Roya Firoozi, Negar Mehr, Esen Yel, Rika Antonova, Jeannette Bohg, Mac Schwager, and Mykel Kochenderfer, editors, *Proceedings of The 4th Annual Learning for Dynamics and Control Conference*, volume 168 of *Proceedings of Machine Learning Research*, pages 263–277. PMLR, 23–24 Jun 2022.

[5] Philipp Grohs, Fabian Hornung, Arnulf Jentzen, and Philippe von Wurstemberger. A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of black-scholes partial differential equations, 2018.

[6] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, aug 2018.

[7] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6), 5 2021.

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

[9] Lu Lu, Xuhui Meng, Zhiping Mao, and George E. Karniadakis. Deepxde: A deep learning library for solving differential equations. *CoRR*, abs/1907.04502, 2019.

[10] M. Raissi, P. Perdikaris, and G.E. Karniadakis. 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, 2019.

[11] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, dec 2018.

[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.