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

A DISSERTATION PRESENTED
BY
HARMIT N. RAVAL
TO
THE DEPARTMENT OF APPLIED COMPUTATION

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ENGINEERING
IN THE SUBJECT OF
COMPUTATIONAL SCIENCE AND ENGINEERING

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2023

Thesis advisor: Professor Pavlos Protopapas Harmit N. Raval

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

Abstract

Differential equations are prevalent in many areas of science and engineering and often times, applications require rapidly solving variants of the same equations with differing conditions. There is growing interest in generating approximate solutions to these equations with neural networks, which provide continuous and differentiable solutions. Specifically, physics informed neural networks (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 the benefits that PINNs offer, they are currently limited by the computational costs needed to train such networks on different but related tasks. To address this drawback, we turn to transfer learning differential equation solutions generated from PINNs.

In this thesis, we present a generalizable and robust methodology to perform "one-shot transfer learning" on linear systems of ordinary and partial differential equations. First, we describe a process to train PINNs on vectorized representations of equations with varying conditions, leveraging a multi-headed approach. Second, we show how this multi-headed training process can be used to yield a latent space representation of a particular equation form. Third, we produce a derivation of closed-form formulas which represent a set of generalized network weights, for both ordinary and partial differential equations. Finally, we demonstrate how the learned latent space representation and derived network weights can be used to rapidly transfer learn solutions to various equations. We apply our novel methodology to a suite of linear systems, illustrating no limitation by the order of the equations, the number of equation, or the time-dependence of equation coefficients.

Contents

To my family.

Acknowledgments

First and foremost, I would like to thank Dr. Pavlos Protopapas for his endless guidance and support as I worked on my thesis during the year. His willingness to meet with me to discuss potential solutions to difficult roadblocks, regularly review code, and explain novel concepts on a weekly basis was instrumental in enabling me to complete my work.

In addition to my advisor, I would also like to thank my other committee members, Professor Cengiz Pehlevan and Dr. Fabian Wermelinger, for allocating time to meet with me regarding progress, reading preliminary copies of this thesis document, and attending my defense.

Next, I would like to thank my large support system that has made this master's thesis journey a special one. Coming into Harvard's Science and Engineering Complex everyday was a great privilege as I was always greeted by the company of my peers, Haedo Cho, Chris Gumb, Kevin Howarth, Tale Lokvenec, and Varshini Reddy. Their encouragement and engaging conversations were both stimulating and refreshing.

Finally, of course none of this would be possible without the love and moral support of my parents, Nitant and Bhakti Raval, my sister, Vedi Raval, and my girlfriend, Priyanshi Patel.

Science is a differential equation. Religion is a boundary condition.

Alan Turing

0

Introduction

Over the last decade, deep learning has made considerable strides across a wide array of challenging tasks. However, such progress did not occur overnight. In fact, it took years for researchers to even understand the scope of work the field would require. Naively, in the 1950s, scientists gathered at Dartmouth College to put forth what they dubbed, "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence"[12]. While the researchers did not uncover all the secrets to the field that summer, their work did make it evident that machine learning, and later deep learning, would not be confined to a static set of problems.

Today, the applications of deep learning continue to blossom across a plethora of domains. For instance, neural networks are capable of making a diagnosis for gastrointestinal endoscopy[13], writing well-structured scientific text[1], beating the world champion in Go[15], and even modeling the spread of Covid-19[14]. While these tasks were never envisioned or even thought possible for computers, neural networks have permitted such capabilities and beyond.

In addition to solving exciting real-world problems, deep learning presents itself as an indispensable tool in academic fields ranging from applied mathematics and economics to cosmology and quantum mechanics. Specifically, a great deal of interest has recently arisen in utilizing neural networks to solve differential equations which represent natural phenomena that occur on varying scales in the universe and touch many applications of scientific research. Solving these differential equations both accurately and efficiently is pivotal to many of these areas of work, as such equations model complex rules and relationships in an comprehensible manner.

Given the wide-ranging significance of solving differential equations, an extensive amount of research has been done in developing intricate numerical methods to solve them. While these tradi-

tional numerical methods have been proven to perform well and yield stable solutions with a high degree of fidelity[4], neural networks offer a much more attractive alternative. Namely, physics informed neural networks (PINNs) are neural networks which are able to leverage both data and studied physical constraints to learn solutions to differential equations. PINNs offer various benefits over traditional numerical methods in solving these equations. These benefits include eliminating the need for a numerical integrator[9], generating continuous and differentiable solutions[9], improving accuracy in high dimensions[6][7], easily incorporating data[5], and maintaining small memory footprints[4].

Despite the numerous benefits PINNs offer, one current limitation is the computational expense needed to train networks for different but closely related tasks[2]. Fortunately, this drawback can be addressed via transfer learning, which is the primary motivating theme for this thesis. Transfer learning is the method of training a network on a specific task and then transferring the learned knowledge to a new task. While transfer learning for PINNs has been largely unexplored thus far, researchers have recently shown that a PINN pre-trained on a family of differential equations can be efficiently re-used to solve new differential equations[4]. Their work illustrates that by freezing the hidden layers of a pre-trained PINN, solving new differential equations belonging to the same family reduces to fine-tuning a single linear layer. Furthermore, this work shows that for small systems of linear ordinary and partial differential equations, the fine-tuning of the final linear layer equates to solving a set of equations in a convex space[4]. Hence, the desired optimal weights needed to represent a new differential equation can be computed in "one-shot", requiring no further fine-tuning to perform transfer learning.

In this thesis, we build upon the work presented in *One-Shot Transfer Learning of Physics-Informed Neural Networks*[4] by generalizing this idea of "one-shot transfer learning" to any system of linear ordinary differential equations (ODEs) and partial differential equations (PDEs). In Chapter 1, we discuss solving ODEs and PDEs using PINNs and representing these equations in a manner digestible to networks. In Chapter 2, we introduce the multi-headed architecture used for training our networks and the generalized loss functions needed for ODEs and PDEs. Chapter 3 presents the normal equations representing the latent space of learnt functions needed for transfer learning ODEs and results on a range of ordinary differential equation systems. Similarly, Chapter 4 derives the normal equations of the latent space of learnt functions for PDEs and illustrates results on varying partial differential equation systems. Finally, we end with a discussion of our key takeaways, the limitations of our approach, and opportunities for future work in Chapters 5 and 6.

I visualize a time when we will be to robots what dogs are to humans, and I'm rooting for the machines.

Claude Shannon

1

Differential Equations

Efficiently and accurately solving differential equations is essential for progress in many areas of scientific research. While it may seem straightforward to analytically derive the solutions to simple equations, this approach does not fare well for when an equation or system of equations requires advanced integration. Similarly, traditional numerical solutions used to solve these equations suffer weaknesses in that they require a numerical integrator[9], sometimes suffer from the "curse of dimensionality" in higher dimensions[6 7], and propagate numerical errors[11]. In light of both analytical and numerical drawbacks, neural networks offer a powerful alternative.

1.1 Differential Equation and Problem Overview

Before discussing what neural networks are and how they can solve differential equations, we briefly describe what differential equations are and the types of problems this thesis engages with. In general, a differential equation is an equation or a system of equations that relates one or more unknown functions and their derivatives. Solving these equations entails uncovering the unknown function(s) that satisfy the problem criteria.

Mathematically, differential equations can be solved for arbitrary functions, however in applications the solution functions often represent some physical quantities and the derivatives of the solution functions represent their rates of change. In addition to satisfying the equation itself, the solutions must also satisfy associated initial and/or boundary value conditions. Initial and boundary conditions are added pieces of information which specify the value(s) of the solution functions at particular point(s). At a high level, there are two classes of problems: ordinary and partial differen-

tial equations. Both classes of problems can contain equations with solutions of varying orders. For this thesis, we work on linear differential equations of varying degrees of difficulty.

It is worth emphasizing that differential equations have a very long history and are fully covered across a large number of courses and whole textbooks. The history of solving differential equations can be traced back to the creation of calculus in the 17^{th} century by Isaac Newton and Gottfried Wilhelm Leibniz. The field continued to evolve in the 18^{th} and 19^{th} centuries when mathematicians such as Joseph-Louis Lagrange, Leonhard Euler, and Carl Friedrich Gauss worked on developing methods to solve ordinary differential equations. In the 20^{th} century, numerical methods for solving these equations were developed due to the advent of computers, hence enabling people to solve problems that would be impossible to solve analytically. Today, the study of these equations continues to expand with new applications being discovered frequently. Fully understanding this subject matter would require years of study. The description that follows and our work more generally touches a very small subset of the problems in this field.

Ordinary Differential Equations

Ordinary differential equations can be defined generally in terms of a function F, which is made up of independent variable x, function u, and derivatives of u:

$$F(x, u, u', \ldots, u^{(n-1)}) = u^{(n)} \tag{1.1}$$

In Equation 1.1 above, we note that $u^i = \frac{d^i u}{dx^i}$ represents the i^{th} derivative of the solution $u(x)$

with respect to the independent time variable x. From this general form, we can represent the linear subclass of problems as follows:

$$\hat{D}_n u = f(x);\ \hat{D}_n u = \sum_{i=0}^{n} a_i(x) u^{(i)} \tag{1.2}$$

In Equation 1.2, $f(x)$ is considered a forcing function or forcing term that impacts the homogeneity of the solution and $a_i(x)$ is a coefficient for each derivative in the problem[4]. This equation shows that ordinary differential equations that can be written as a linear combination of the derivatives of u are said to be linear - all other ordinary differential equations are non-linear.

Partial Differential Equations

Similar to ordinary differential equations, partial differential equations can be defined in a generalized form. For simplicity of explanation, we represent a linear second order PDE with two independent variables x_1 and x_2 as follows:

$$(D^{x_1} + D^{x_2} + D^{x_1 x_2} + V(x_1, x_2))u(x_1, x_2) = f(x_1, x_2) \tag{1.3}$$

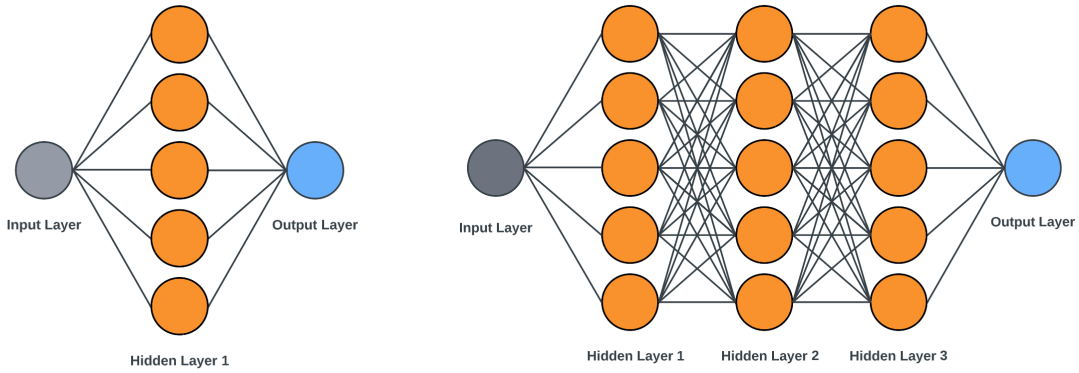
In Equation 1.3, $f(x_1, x_2)$ is a forcing function in terms of independent variables x_1 and x_2 and $V(x_1, x_2)$ is what is commonly referred to as a potential function. Moreover, $D^{x_1} u$, $D^{x_2} u$, and $D^{x_1 x_2} u$ are second-order operators representing the individual and mixed independent variable dimensions. In general, the form of Equation 1.3 can be extended to higher orders and incorporate more independent variables. Regardless of the orders and number of variables, this equation illustrates that

partial differential equations that can be written as a linear combination of the partial derivatives of u are linear.

1.2 Feed Forward Neural Networks

With the previous differential equation overview in mind, we now discuss neural network models and how they can be applied to solve differential equations. We leave a detailed discussion of our specific training procedure, loss function expressions, and network architecture for Chapter 2.

At a high level, neural networks are complex machine learning models which can be used to solve classification or regression problems. Given their immense level of expressiveness, they are well-suited to a variety of different tasks. The use of artifical neural networks to solve differential equations was first proposed by Lagaris et al.[9] and has attracted a considerable amount of interest as of recent.



(a) Network with one hidden layer

(b) Network with three hidden layers

Figure 1.1: Two sample fully-connected neural network architectures. The diagram on the left illustrates a network with one hidden layer with 5 nodes (orange circles) and the diagram on the right illustrates a network with three hidden layers, each with 5 nodes. In both diagrams the gray and blue circles represent the input and output layers of the network, respectively.

To solve differential equations, neural networks are trained on numerical data sampled from a predefined range of values in order to learn a weight matrix W. This matrix represents the solution function(s). Structurally, a neural network is composed of interconnected nodes dispersed across various layers. A network contains an input layer, one or more hidden layers, and an output layer. Mathematically, each hidden layer of a network performs a set of operations on the input data using some finite number of nodes. Namely, these operations include multiplying the input by W, adding a translation vector b, and applying a non-linear activation function σ. Hence, for a single hidden layer network as shown in Figure 1.1a, the input x is transformed into the output u as follows:

$$u = \sigma(Wx + b) \tag{1.4}$$

For networks with more than one hidden layer, as shown in Figure 1.1b, the output is transformed by each hidden layer in succession. All neural networks, regardless of the number of hidden layers, are universal approximators[3]. In other words, a network with any number of layers and nodes can approximately approach or represent any function $f(x)$. The addition of more layers and nodes to a network adds to its expressiveness, yielding more flexibility in accurately representing the underlying function of interest.

To find the optimal representation of an underlying function, neural networks minimize a loss function during training. In the case of ordinary differential equations, the neural network attempts

to minimize a loss of the following form:

$$L(u_s, x) = \sum_{x \in \text{Data}} \left(F(x, u_s, u_s', \ldots, u_s^{(n-1)}) - u_s^{(n)} \right)^2 \tag{1.5}$$

We note that when plugging in the network solution $u_s(x)$ into Equation 1.1, we are working with the differential equation residuals. Hence, in Equation 1.5, the network's goal is to minimize the square of the equation residuals. In other words, the solution $u_s(x)$ is sought such that it makes $L(u_s, x)$ as close to 0 as possible. However, it is important to note that Equation 1.5 does not provide a complete picture of the loss function used for training networks on differential equations. Namely, the equation does not address how initial value or boundary conditions are handled. We address these concerns in Chapter 3.

1.3 Vectorized Equation Representation

While the general, mathematical form of ordinary and partial differential equations was introduced in Section 1.1 earlier, we now turn our attention to how these equations can be written in a form understandable by networks. In other words, we present a vectorized equation representation which can generalize to any number and form of linear equations. We subsequently use this representation and notation for all equations presented in the following chapters.

1.3.1 Ordinary Differential Equations

Consider the following system of coupled first order ordinary differential equations as an example to vectorize:

$$\begin{cases} \dfrac{du_1}{dt} + 3u_2 = 0.1 \\ \dfrac{du_2}{dt} + 2u_1 = 0.3 \end{cases} \tag{1.6}$$

Equation 1.6 is a system of two ordinary differential equations with an independent variable t and two unknown functions $u_1(t)$ and $u_2(t)$. We aim to write this and any other system of linear first order ordinary differential equations in the following form:

$$\dot{u} + Au = f \tag{1.7}$$

In Equation 1.7, $\dot{u}$ is a vector of the derivatives of the solution u with respect to the independent variable t, A represents a square matrix* of coefficients for non-derivative terms, and f is a vector representing the forcing function terms. Given this formulation, we can naturally rewrite Equation 1.6 as follows:

$$\begin{pmatrix} \dot{u_1} \\ \dot{u_2} \end{pmatrix} + \begin{bmatrix} 0 & 3 \\ 2 & 0 \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} - \begin{pmatrix} 0.1 \\ 0.3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{1.8}$$

*The A matrix is always of size $m \times m$ where m is the number of equations in the system of interest. Each row of A corresponds to an equation's non-derivative coefficients.

It is straightforward to multiply out the terms of Equation 1.8 and observe that it indeed recovers the original problem presented in Equation 1.6.

For a less obvious example, consider the following higher order differential equation:

$$\frac{d^2u}{dt^2} + 0.1\frac{du}{dt} + 0.3u = 0.8 \tag{1.9}$$

To represent Equation 1.9 in a form amenable to network training, we must first re-write it such that it is in a form similar to Equation 1.6. In other words, we need to eliminate the higher order derivatives. We accomplish this by introducing the following new variable via substitution: $v = \dot{u}$. Hence, we can write the single equation into a system of equations as follows:

$$\begin{cases} \dfrac{dv}{dt} + 0.1v + 0.3u = 0.8 \\ \dfrac{du}{dt} - v = 0 \end{cases} \tag{1.10}$$

Immediately, it becomes clear that Equation 1.10 can now be written in our vectorized representation, similar to the previous example:

$$\begin{pmatrix} \dot{v} \\ \dot{u} \end{pmatrix} + \begin{bmatrix} 0.3 & 0.1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} u \\ v \end{pmatrix} - \begin{pmatrix} 0.8 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{1.11}$$

The motivation for using this representation will become more evident when we discuss the complete loss function representation in Chapter 3.

1.3.2 Partial Differential Equations

Let us now examine the following system of partial differential equations as an example to vectorize:

$$\begin{cases} \dfrac{\partial u_1}{\partial x_1} + 4\dfrac{\partial u_1}{\partial x_2} = 1 \\ \dfrac{\partial u_2}{\partial x_1} + 9\dfrac{\partial u_2}{\partial x_2} = 3 \end{cases} \tag{1.12}$$

Equation 1.12 is an example of a system of two linear partial differential equations with independent variables x_1 and x_2 and two unknown functions $u_1(x_1, x_2)$ and $u_2(x_1, x_2)$. Our goal is to be able to write this example and any other system of linear partial differential equations in the form shown below:

$$((D_u G^T) \circ I)\overrightarrow{1} + Au = f \tag{1.13}$$

In Equation 1.13 above, D_u is a matrix of the partial derivatives of the solution u with respect to the independent variables x_1 and x_2 (the Jacobian of u) and G represents a matrix of coefficients for the partial derivatives in D_u[†]. Next, I represents the identity matrix, $\overrightarrow{1}$ is a vector of ones, A is a matrix of coefficients for non-derivative terms, and f is a vector representing the forcing function terms. Finally, note that the "$\circ$" operator represents the Hadamard product. Based off of this representation,

[†]Both the D_u and G matrices are always of size $m \times k$ where m is the number of equations in the system of interest and k is the number of independent variables. Each row of D_u and G corresponds to an equation in the system.

we can rewrite Equation 1.12 as follows:

$$\left(\left(\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 9 \end{bmatrix}^T\right) \circ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \tag{1.14}$$

It becomes evident that expanding out the matrices in Equation 1.14 would yield the original system shown in Equation 1.12.

To avoid redundancy, we omit showing how to represent a higher order PDE using the formulation from Equation 1.13. However, it is worth briefly highlighting that the process to write such an equation would be similar to representing a higher order ODE as shown in Equation 1.9. A higher order PDE would first have to be rewritten into a system of PDEs by introducing new variables via substitution. This simple rewriting would then make the PDE into a system which can then be straightforwardly written using the form from Equation 1.13.

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

John von Neumann

2

Training Procedure

A robust training procedure is important in order to successfully perform transfer learning on a variety of differential equations. In other words, we want our network training methodology to be able to generalize to any number of equations and be flexible enough to work for both ordinary and partial differential equations. This approach would ensure that we can handle various systems in a consistent manner.

Below, we describe the formulation of the loss functions for ODEs and PDEs respectively, as well as the exact training set-up used to learn a latent space representation of equations. We conclude with a description of the hardware specifications used to execute the training procedure.

2.1 Loss Function

For both ODEs and PDEs, we make use of custom loss functions in the training procedure. At a high level, the goal is to minimize the residuals of the differential equations and satisfy the initial and/or boundary conditions. We sum both of these components together for each iteration of training and use this loss value to subsequently inform the updates of our weights by leveraging backpropagation and autograd [10].

2.1.1 Ordinary Differential Equations

The loss function for ODEs is composed of two components: the system of differential equations that need to be satisfied and the initial conditions. Hence, we formulate this function as follows:

$$L_{ode} = \frac{1}{n}\sum_{t}\left((\dot{u} + Au - f)^2\right) + \frac{1}{n}\sum_{t}\left(u(t=0) - u_0\right)^2 \tag{2.1}$$

In Equation 2.1 above, the first summation term represents the differential equation that needs to be satisfied. The summation takes place over each of the batch elements t and then the mean is taken over all n such elements. This value is then added to the initial value condition, which is simply the squared difference of the network output at $t = 0$ and the true solution at $t = 0$. It is ultimately this convex function which we want to minimize during training. We will use this loss function, and the one presented below for PDEs, to analytically derive the weights needed for transfer learning in Chapters 3 and 4, respectively.

We note that while the initial value condition was handled by incorporating it into Equation 2.1 via the second summation term, another viable solution would have been to use a reparametrization of the network solution[8]. In other words, we could rewrite the network output in a way such that it satisfies the initial value condition by construction, hence simplifying the loss function. The reparametrization approach would eliminate the need for the second loss function term. Both solutions are equally valid, however in this work we opt for the former approach in order to simplify the transfer learning mathematical analysis that is performed in Chapters 3 and 4.

2.1.2 Partial Differential Equations

Similar to ODEs, the loss function for PDEs can be described in terms of two pieces: the system of differential equations that must be satisfied and the boundary conditions. With this, we formulate the loss function as shown below:

$$L_{pde} = \frac{1}{n} \sum_{x \in \text{Batch}} (((D_u G^T) \circ I) \overrightarrow{1} + Au - f)^2 + \frac{1}{n} \sum_{x \in \text{Batch}} (u(x) - u_0(x))^2 \tag{2.2}$$

In Equation 2.2, the first summation term corresponds to the differential equation that must be satisfied. Similar to Equation 2.1, we take the mean of the individual L_x elements over the batch size. The second term of the loss function corresponds to the boundary conditions, namely Dirichlet boundary conditions. Here, we find the squared difference between the network solution and true solution at the boundary condition x. As mentioned in Section 2.1.1, using the reparametrization technique would eliminate the second loss term, but would significantly complicate the transfer learning.

In this work, we did not deal with PDEs that contained Neumann boundary conditions. However, to incorporate these conditions into the existing framework, one would simply need to add a third term to the existing loss function shown in Equation 2.2. This third term would correspond to the Neumann boundary conditions and a system could then be handled which had only a Dirichlet or Neumann boundary condition or even a combination of both.

2.2 Model Architecture

The network architecture used for training both ordinary and partial differential equations is quite simple in nature. In both cases, we use a PINN with a fixed number of hidden layers and all hidden layers are fully-connected layers. Tables 2.1 and 2.2 show the number of layers for each network architecture and the number of nodes per layer.

ODE Network Architecture	
Layer	*(Input, Output) Nodes*
Input Layer	(1, 128)
Hidden Layer 1	(128, 128)
Hidden Layer 2	(128, 128)
Hidden Layer 3	(128, 256)
Output Layer	(256, m)

Table 2.1: Table illustrating the 3-layered network used to train systems of ODEs. For the output layer, m corresponds to the number of equations in the system.

PDE Network Architecture	
Layer	*(Input, Output) Nodes*
Input Layer	(k, 256)
Hidden Layer 1	(256, 256)
Hidden Layer 2	(256, 256)
Hidden Layer 3	(256, 256)
Hidden Layer 4	(256, 256)
Hidden Layer 5	(256, 512)
Output Layer	(512, m)

Table 2.2: Table illustrating the 5-layered network used to train systems of PDEs. For the input and output layers, k and m represent the number of dependent variables and number of equations in the system, respectively.

For ODEs, we make use of a 3-layered network with 128 nodes in the first two hidden layers and 256 nodes in the third hidden layer. The input layer always contains 1 node for ODEs since there is only one dependent variable and the output layer contains m nodes corresponding to m equations. On the other hand, for PDEs, we make use of a 5-layered network with 256 nodes in the first four hidden layers and 512 nodes in the fifth hidden layer. The input layer contains k nodes corresponding to the number of dependent variables and the output layer, again, is made up of m nodes for m equations.

We note two important characteristics that one might observe about the set-up of these networks. First, it is clear that the partial differential equation network contains more nodes and layers compared to the ordinary differential equation network. This is due to the fact that PDEs are naturally much more expressive than ODEs and hence require a larger network to sufficiently recover the solutions of interest. The exact number of nodes chosen were based on empirical experiments. Secondly, we note that the final hidden layer for both architectures contains double the number of nodes compared to the previous layers in the respective networks. This is necessary because the last hidden layer informs the size of the latent space (called H) which will represent the equations of interest. In general, we want more nodes here to have an accurate and generalized representation. We discuss this latent space H in more detail in Section 2.3 below.

Finally, with regards to the optimizer choice used in training, we make use of stochastic gradient descent (SGD) for our ODE network and ADAM for our PDE network. For activation functions, the ODE network uses *Tanh* activation for all hidden layers and the PDE network uses *SiLU* activation for all hidden layers.

2.3 Training Set-up

To train a network on ordinary or partial differential equations, it is important that we learn a generalized representation of the particular system of equations that we are interested in. In other words, during training, the ultimate goal is for the network to learn what the latent space representation of the equations is. Rather than training a network on one specific set of equation coefficients and

initial/boundary conditions, it should be familiar with a range of values such that it can learn the "general equation form". It is this methodology that we describe in Section 2.3.2, after outlining the structure of a simplified training approach in Section 2.3.1.

2.3.1 Single-headed Approach

Before describing the actual training scheme used in this work, it is valuable to understand a simplified version of the approach, as shown in Figure 2.1. Namely, we observe that the network accepts an input time t for an ODE (or multiple inputs in the case of PDEs) and proceeds to flow through the hidden layers. After the final hidden layer's activation function is applied, we call this output H. As alluded to in the introduction to the section, H represents the latent space representation of the equations of interest. H has a shape of $m \times d$ or the number of equations by the number of nodes in the last hidden layer.

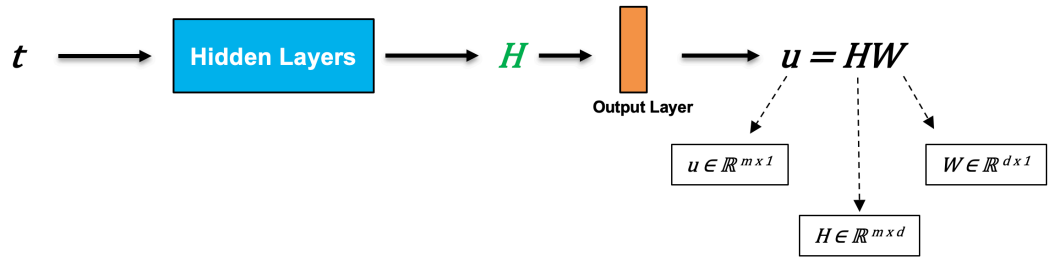


Figure 2.1: Diagram of simplified network training procedure for ODEs with no varying training conditions. For a single "head", an input t is fed through the hidden layers (blue). The output after the last hidden layer's activation function is applied is called the latent space H. Finally, once the input goes through the output layer (orange), the solution u is found. This solution can be reconstructed by multiplying H by the weights of the output layer, W.

In the case of this simplified training procedure, there is only one "head", where a "head" in this context refers to a set of conditions (equation coefficients and initial conditions) that the network is

being trained on. In other words, there is one path through the network. Hence, training a network this way means that H will correspond exactly to the single set of conditions it is being trained on. After the input goes through the output layer, the differential equation solution u is reached. Mathematically, $u = HW$ where W is the weights of the output layer and has a shape of $d \times 1$ or the number of nodes in the last hidden layer by 1.

It will be the goal of Section 2.3.2 to show how we can learn a generalized H that corresponds to a broader set of conditions. Ultimately, we want to save the value of H after training and later compute a set of generalized weights W_0 such that we can construct any differential equation solution u by multiplying a learned H with a derived set of weights W_0.

2.3.2 Multi-headed Approach

With the multi-headed approach, we aim to learn a representation of H which captures the general equation form that is being trained on. Unlike the single-headed method discussed in Section 2.3.1, using multiple "heads" with diverse training conditions results in an H that can be used for transfer learning equations of the same training form but with different coefficients, forcing functions, and initial/boundary conditions.

Figure 2.2 illustrates how the multi-headed training process can enable us to learn this general H. First, an input t (or multiple inputs in the case of PDEs) given to the network will flow through all hidden layers. Similar to Section 2.1, H is what comes out of applying the final hidden layer's activation function. The primary difference in the multi-headed approach is having h output layers as opposed to one. Each output layer contains its own set of weights W^i. Hence, there are h network

outputs u^i, each of which can be computed by multiplying H by the corresponding W^i. All of the h paths through the network leads to their own loss function L^i, which can then be summed together to form an overall L_{total} which is what is minimized while training.

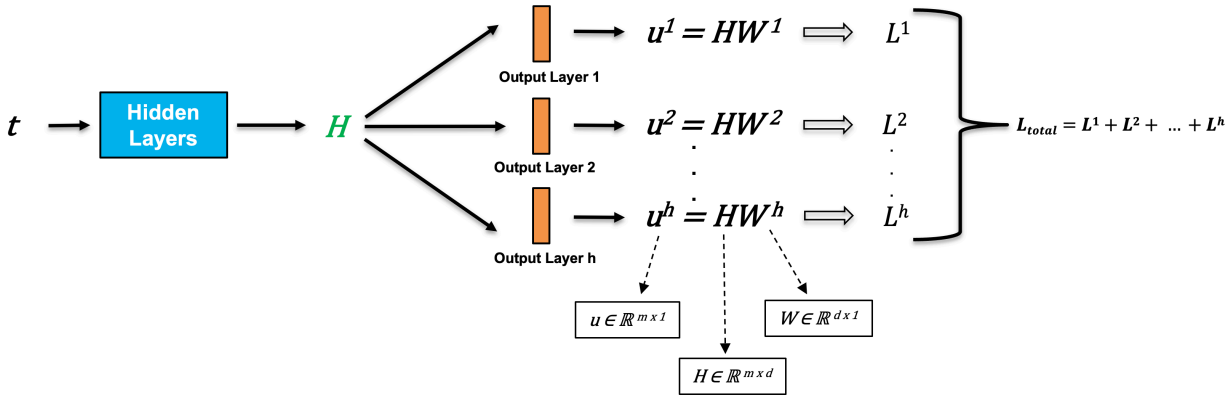


Figure 2.2: Diagram of multi-headed training procedure for ODEs with h "heads". For multiple "heads", the same input t is fed through the hidden layers (blue). The output after the last hidden layer's activation function is applied is called the latent space H. Finally, since there are multiple "heads", a solution u^i is found for each set of conditions after the input goes through each output layer (orange). Each solution u^i can be reconstructed by multiplying H by the corresponding output layer's weights, W^i.

The key observation to note in Figure 2.2 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 all be encoded in this one $m \times d$ matrix. Due to this, H will effectively be the latent space representation of the general equation system being trained on. During training, we empirically observe that the network learns to shift the W^i values to accommodate the different conditions as it converges on an H matrix. To ensure that the network has been sufficiently trained, it can be confirmed that at much later epochs, H stabilizes, while the W^i values are what fluctuate. For an example of training conditions that are used in this thesis, refer to Table A.4 in Appendix

A.1. To generate the results discussed in Chapters 3 and 4, we make use of the multi-headed training approach with four heads.

2.4 Computing Resources

We conclude this chapter with a brief description of the hardware that was used to train all our networks. Namely, an NVIDIA Tesla T4 GPU from Google Colaboratory was leveraged to run all ODE and PDE training sessions. The relevant specification details of the Tesla T4 are included in Table 2.3 below.

GPU Specification	
Hardware Metric	*Numeric Value*
Cores	2560 Cores
Memory Size	16 GB
Memory Type	GDDR6
Bus Width	256 bit
Bandwidth	320.0 GB/s
Base Clock	585 MHz
Boost Clock	1590 MHz

Table 2.3: NVIDIA Tesla T4 GPU Specifications

Given that neural network development pipelines are naturally suited to parallelization, all training procedures, evaluation of model performance, and transfer learning PyTorch code was written in parallel and run on the Tesla T4 GPU.

If I have been able to see further, it was only because I

stood on the shoulders of giants.

Isaac Newton

3

Transfer Learning ODEs

In order to perform transfer learning on ODEs we need to demonstrate how to recover the set of generalized network weights, W_0. These derived, generalized weights and the learned equation latent space, H, will enable us to recover differential equation solutions u. This is because $u = HW$ based on the network architecture shown in Chapter 2.

In this chapter, we discuss each of the components, H and W_0, needed for transfer learning in more detail, derive a closed-form representation of W_0, and present a series of results on various ordinary differential equations. We cover analogous content for partial differential equations in Chapter 4.

3.1 Components for Transfer Learning

As highlighted above, both H and W_0 are needed to perform transfer learning for ODEs (and PDEs). It has already been thoroughly shown in Section 2.3.2 how H can be generated through the multi-headed training process. Hence, we now focus on understanding how W_0 can be derived using H and known information about the loss function. First, it is worth highlighting that W_0 is different from the W values shown in Figure 2.2. Namely, the W values shown in Figure 2.2 correspond to the weights specific to each head's training conditions - they do not generalize to *any* set of equation conditions. In fact, after training a network on a set of heads with varying equation conditions and recovering H, we can simply ignore the specific W values the network learns.

We derive our own set of weights, called W_0, by analytically computing which weights minimize the ODE (or PDE) loss function. These W_0 correspond to the set of generalized weights that solve

any differential equation or system of equations that match the training form and are represented by the latent space H. Ultimately, once we derive W_0, we use these weights in combination with the learned H to compute the solutions to equations by finding $u = HW_0$.

3.2 Derivation of Generalized Equation Weights

The fact that our network loss function, shown in Equation 2.1, is convex enables us to analytically solve for the weights of interest W_0. An analytical derivation provides us with a closed-form representation of W_0, which an approach such as empirically fine tuning the last hidden layer of the network would not yield. To derive W_0, we start with the full ODE loss function as follows:

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

Now, given that we know that $u = HW_0$, we make this substitution into the loss function such that the loss is dependent on H and W_0:

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

Next, we multiply out the quadratic terms inside both the summation and initial condition terms:

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

For notation purposes, let us split the L_{ode} into two pieces, L_t and L_0, where we have the equation

piece $L_t = \frac{1}{n}\sum_t \left(\dot{H}W_0 + AHW_0 - \dot{f}\right)^T \left(\dot{H}W_0 + AHW_0 - \dot{f}\right)$ and the initial condition piece $L_0 = \frac{1}{n}\sum_t \left(H_0W_0 - u_0\right)^T \left(H_0W_0 - u_0\right)$. More simply, this means that the full loss function can be written as follows:

$$L_{ode} = L_t + L_0$$

With this, we can simplify both L_t and L_0 by first distributing the transpose in both the equation and initial condition portions and then foiling out the individual terms. We omit the detailed steps for brevity and instead show the results of the careful simplification for both chunks:

$$L_t = \frac{1}{n}\sum_t (W_0^T\dot{H}^T\dot{H}W_0 + W_0^T\dot{H}^TAHW_0 - W_0^T\dot{H}^Tf + W_0^TH^TA^T\dot{H}W_0 + W_0^TH^TA^TAHW - W_0^TH^TA^Tf - f^T(\dot{H}W_0 + AHW_0 - f))$$

$$L_0 = \frac{1}{n}\sum_t \left(W_0^TH_0^TH_0W_0 - W_0^TH_0u_0 - u_0^TH_0W_0 + u_0^Tu_0\right)$$

Given that we want to find the W_0 that minimize the ODE loss function, we must take the gradient of the loss with respect to these weights and then solve for them:

$$\frac{\partial L_{ode}}{\partial W_0} = \frac{\partial L_t}{\partial W_0} + \frac{\partial L_0}{\partial W_0}$$

For clarity, we omit the full set of steps required in actually taking the gradient and simplifying

terms. The result of these calculations is the following:

$$\frac{\partial L_{ode}}{\partial W_0} = \left[\frac{1}{n}\sum_t (\dot{H}^T\dot{H} + \dot{H}^T AH + H^T A^T\dot{H} + H^T A^T AH) + H_0^T H_0\right] W_0 - H_0^T u_0 - \frac{1}{n}\sum_t \dot{H}f - \frac{1}{n}\sum_t H^T A^T f$$

Here, let us set the variable $M = \left[\frac{1}{n}\sum_t (\dot{H}^T\dot{H} + \dot{H}^T AH + H^T A^T\dot{H} + H^T A^T AH) + H_0^T H_0\right]$ such that we can simplify our notation and set the expression equal to 0 as shown below:

$$MW_0 - H_0^T u_0 - \frac{1}{n}\sum_t \dot{H}f - \frac{1}{n}\sum_t H^T A^T f = 0$$

At this point, we can move all terms except the first to the right-hand side to isolate the W_0 term:

$$MW_0 = H_0^T u_0 + \frac{1}{n}\sum_t \dot{H}f + \frac{1}{n}\sum_t H^T A^T f$$

Now, we multiply both sides of this simplified equation by the inverse of M, which gives us a closed-form representation of W_0:

$$W_0 = M^{-1}\left(H_0^T u_0 + \frac{1}{n}\sum_t \dot{H}f + \frac{1}{n}\sum_t H^T A^T f\right) \tag{3.1}$$

3.3 Performing One-Shot Transfer Learning

In order to perform transfer learning on ODEs, it is first necessary that a multi-headed network be sufficiently trained on varying u_0, A, and f values. After training, a single forward pass must be

completed through the trained model in order to obtain H. After saving H, we can compute the generalized weights W_0 using Equation 3.1 for a chosen u_0, A, and f. Finally, with H and W_0, the desired differential equation solution can be computed in "one-shot" by following $u = HW_0$.

The key observation, and most powerful characteristic of this transfer learning approach is that the network training only needs to be done once, until a new equation form wants to be solved. Assuming the same equation form, we can now change u_0, A, and f freely to new values that have not been trained on. 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 our transfer learned solution.

We conclude this section by highlighting which parts of Equation 3.1 need to be recomputed depending on what type of transfer learning is to be performed. First, if we want to change the initial condition u_0, then we simply need to recompute the $H_0^T u_0$ term. In other words, having precomputed M (dependent on A, H, and H_0), $\frac{1}{n}\sum_t \dot{H}f$, and $\frac{1}{n}\sum_t H^T A^T f$, only one term is recalculated. Next, if we choose to change A, based on Equation 3.1, we need to recompute M (and invert it) and $\frac{1}{n}\sum_t H^T A^T f$ as both of these pieces depend on A. Finally, if we choose to change f, we only need to calculate $\frac{1}{n}\sum_t \dot{H}f$, and $\frac{1}{n}\sum_t H^T A^T f$ given their dependence on f. It is worth pointing out that changing u_0 or f requires recalculations based on a few matrix multiplications and additions, whereas changing A is more expensive as it entails recomputing the $d \times d$ M matrix and inverting it.

3.4 Results

Below, we include training and subsequent transfer learning results on a variety of different ordinary differential equation systems that were solved with the "one-shot transfer learning" approach. In each case, we present the system being solved, multi-headed training outcomes, and then a few examples of transfer learning on different values. The training parameters' details and exact training conditions used per head for all problems can be found in Appendix A.1. It should be noted that in all cases, the true or actual solutions are computed using `scipy.integrate.solve_ivp`[16]. For systems of ODEs, SciPy provides a high-quality numerical solver that leverages an explicit 4^{th} order Runge-Kutta method.

3.4.1 Single ODE

The most straightforward problem to solve is a linear first order equation as shown in Equation 3.2. Here, c, f, and v are not time-dependent coefficients, but their values are varied per head in training.

$$\begin{cases} \dfrac{du}{dt} + cu = f \\ \\ \quad u(0) = v \end{cases} \tag{3.2}$$

Training Outcomes

The results of training a network on Equation 3.2's form can be seen in Figure 3.1. We observe that in 10000 iterations, the loss and MSE values have reached below 10^{-4} and 10^{-7}, respectively. Addi-

tionally, by plotting the network-learned solutions on top of the true solutions for each of the heads, it becomes clear that the network has learned a general representation of Equation 3.2. For all four heads, Figure 3.1 shows that the network and true solutions overlap tightly.

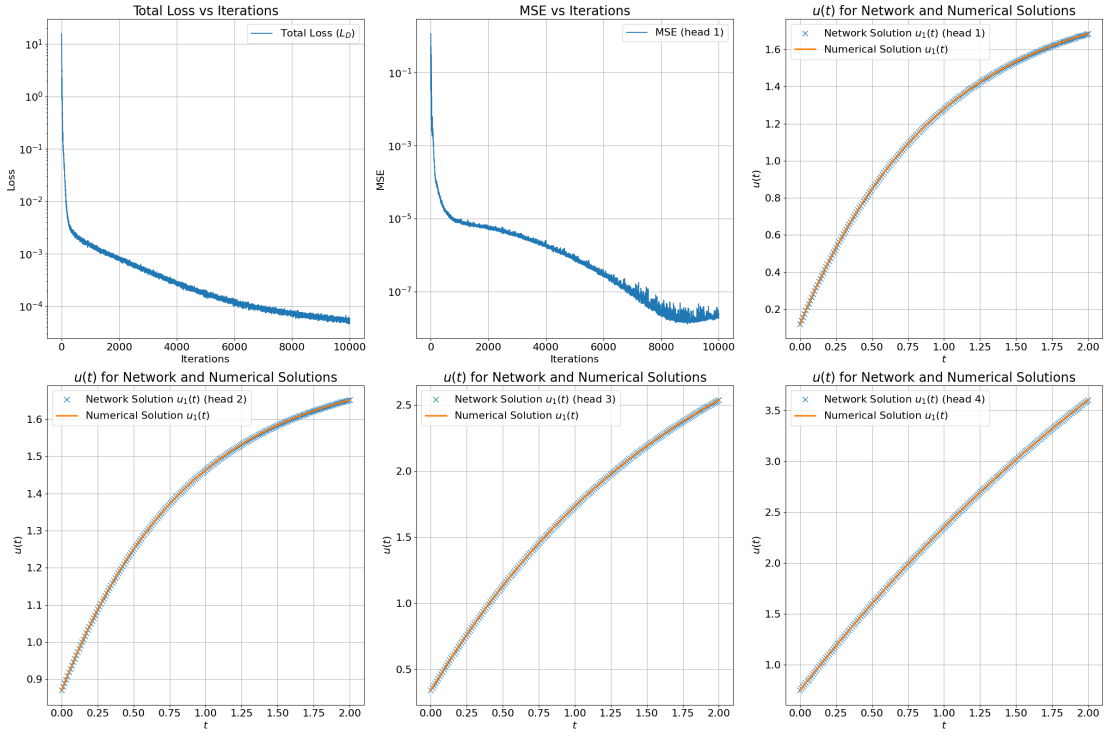


Figure 3.1: Training results for Equation 3.2 after 10000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and numerical solutions). The remaining four plots show the network learned solution (blue) versus the true solution (orange) for each head's equation configurations.

Change Initial Condition u_0

For the first transfer learning instance, we change u_0 and generate a solution to Equation 3.2 as shown in the left-hand side of Figure 3.2. The values of u_0 during training ranged from roughly 0.1

to 0.8 and the value we transfer to is -20.69, well outside of the training region. Despite this, we see that the equation residuals remain smaller than 10^{-4}, as shown in the right-hand side of Figure 3.2.

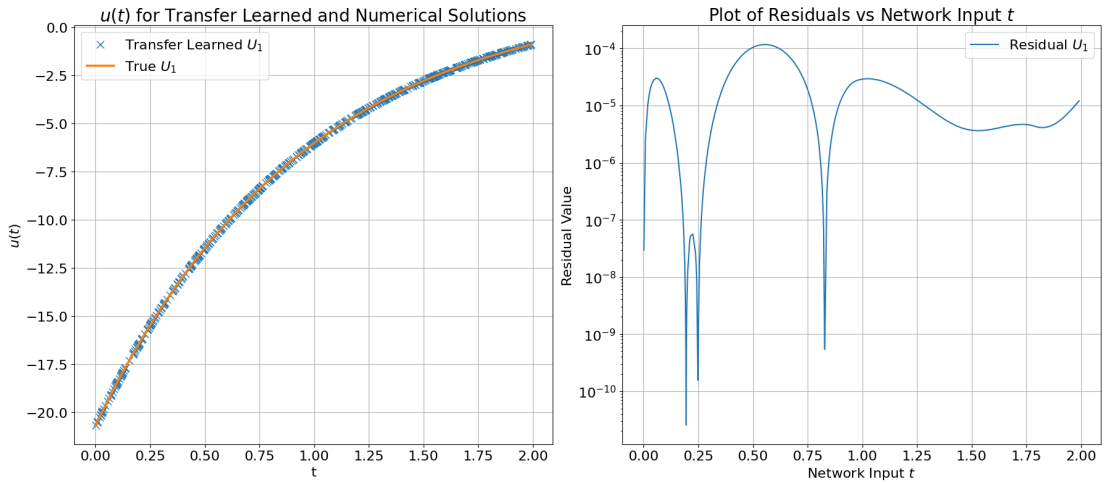


Figure 3.2: Transfer learning results after changing u_0 to -20.69 (see Table A.2 in Appendix A.1 for u_0 training values). The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

Change Forcing Function f

For the second instance of transfer learning, we adjust the forcing function value f in Equation 3.2 from the training value of 2.0 to 10.6. Figure 3.3 confirms that the transfer learned solution nearly fully recovers the true solution as the residual values are almost always smaller than 10^{-6}. It is worth emphasizing that the f value was not varied among the heads during training, but nonetheless we were able to effectively transfer learn to a new value. Naturally, due to the lack of variation in f during training, the equation residuals are not as small as they could be if f was varied among the heads.

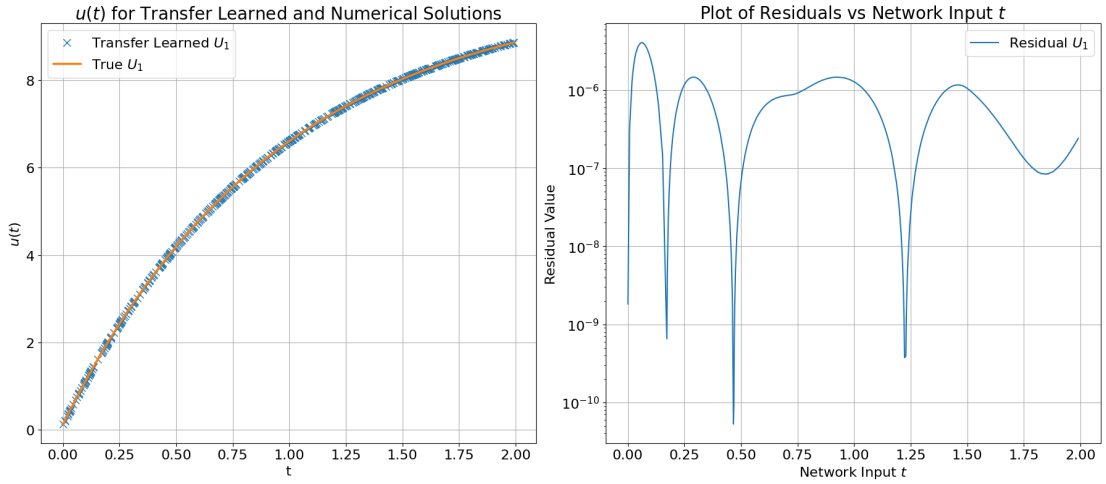


Figure 3.3: Transfer learning results after changing f to 10.6 (see Table A.2 in Appendix A.1 for the f training value). The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

3.4.2 Two Coupled ODE System

We now solve a system of two linear first order equations, as written in Equation 3.3. In this system, $c_1, c_2, c_3, c_4, f_1, f_2, v_1$, and v_2 are not time-dependent, but their values are varied during training.

$$\begin{cases} \dfrac{du_1}{dt} + c_1 u_1 + c_2 u_2 = f_1 \\ \dfrac{du_2}{dt} + c_3 u_1 + c_4 u_2 = f_2 \\ u_1(0) = v_1 \\ u_2(0) = v_2 \end{cases} \tag{3.3}$$

Training Outcomes

Figure 3.4 shows the results of training on Equation 3.3. It is evident that that we have sufficiently trained the network as our loss plateaus below roughly 10^{-4}, the mean squared error reaches slightly below 10^{-7}, and the network-learned solutions and true solutions align very closely across all four heads.

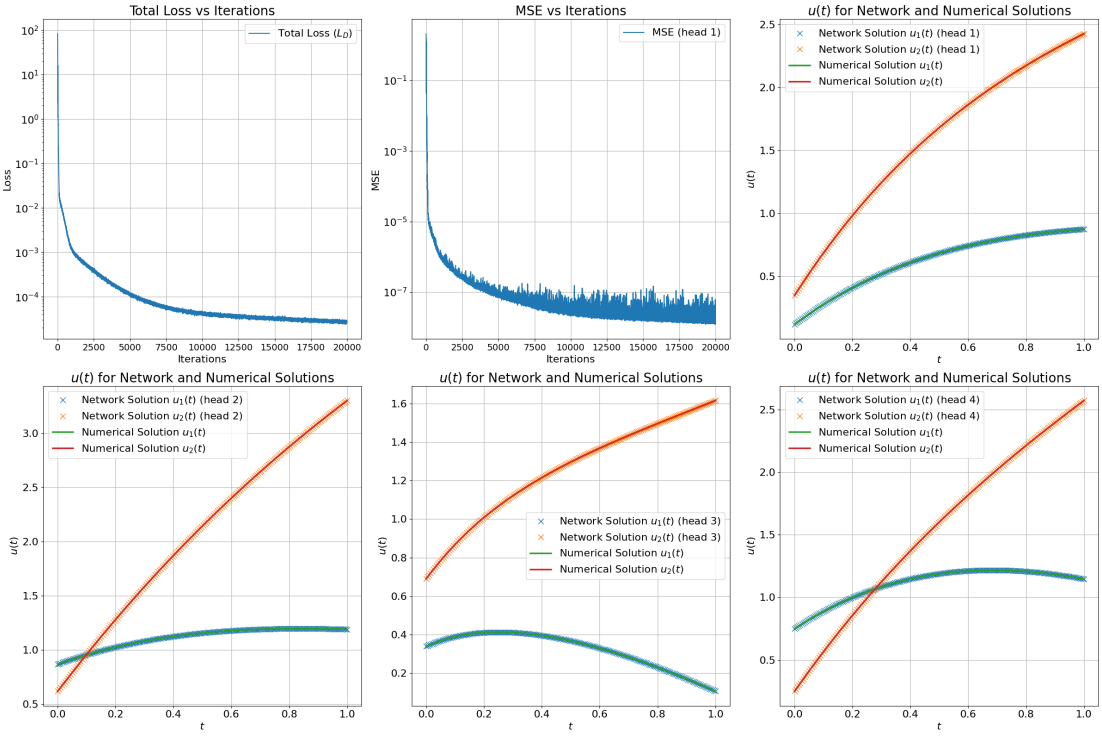


Figure 3.4: Training results for Equation 3.3 after 20000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and numerical solutions). The remaining four plots show the network learned solutions (blue/orange) versus the true solutions (green/red) for each head's equation configurations.

Change Coefficient Matrix A

To perform transfer learning on the system of ODEs shown in Equation 3.3, we first choose to change the coefficient matrix A and plot the transfer learned solution in the left subplot of Figure 3.5. The A matrices used during training were constructed with values ranging from 0 to 2.0, but here we transfer to $A = \begin{bmatrix} 0 & -2.5 \\ 3.1 & 0 \end{bmatrix}$. We again achieve accurate results, as proven by the equation residuals having a median around 10^{-7}. It is particularly interesting to note how using an A matrix with 0s on the diagonal results in a very different curvature in the transfer learning example compared to the training curves arising from the A matrices used in Figure 3.4.

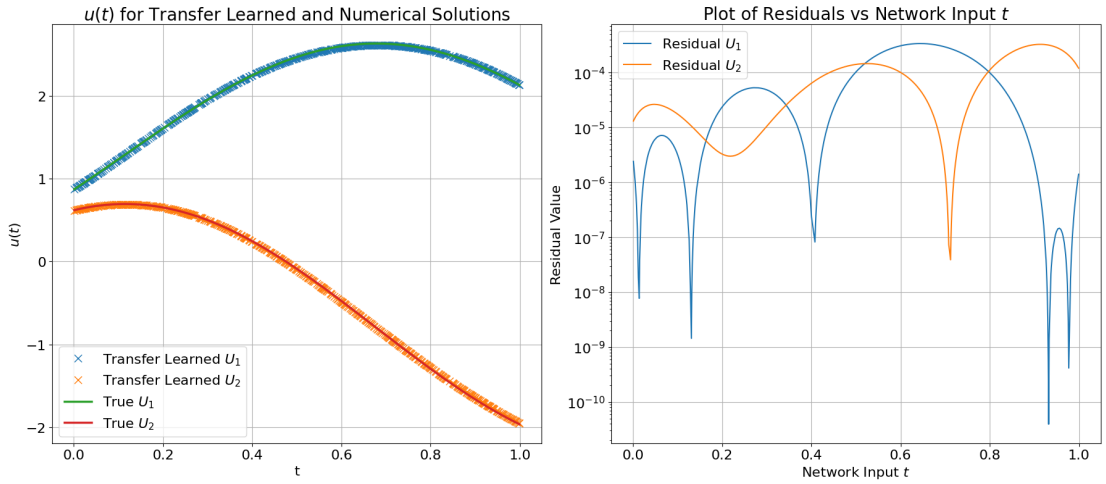


Figure 3.5: Transfer learning results after changing A to $\begin{bmatrix} 0 & -2.5 \\ 3.1 & 0 \end{bmatrix}$ (see Table A.4 in Appendix A.1 for training A values). The left subplot shows the transfer learned solutions (blue/orange) versus the true solutions (green/red) and the right subplot shows the network equation residuals.

Change Initial Condition u_0 and Coefficient Matrix A

In addition to altering A, we can simultaneously change u_0 in our transfer learning approach. Namely, we show the results of changing both parameters in Figure 3.6. Despite the fact that we have considerably transformed the equations (altering all adjustable coefficients except f), Figure 3.6 still yields reasonable results. Naturally, the residuals are larger than previous examples as they increase to a median around 10^{-3}, however this is due to the fact that the transfer learned system is very different compared to the variants used during training.

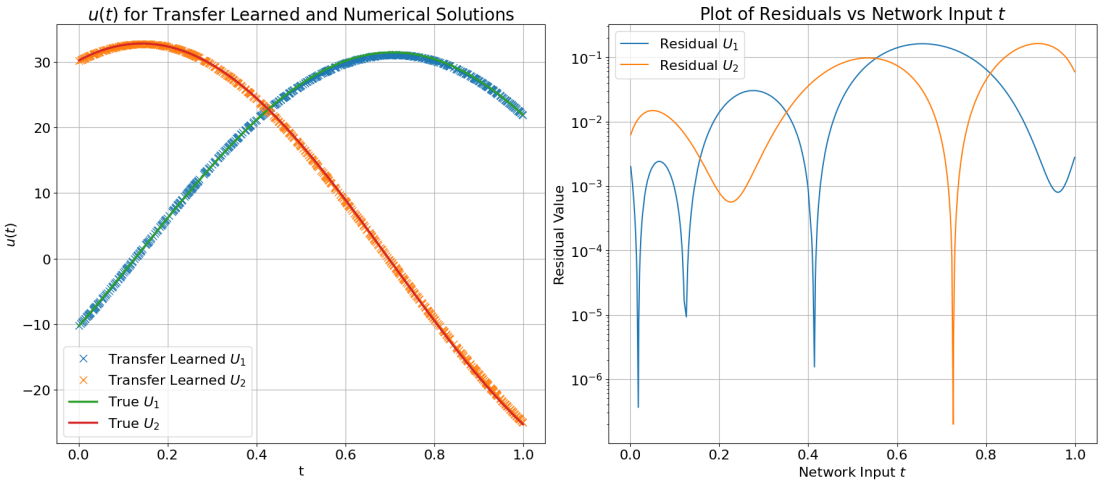


Figure 3.6: Transfer learning results after changing u_0 to $\begin{bmatrix} -10.33 \\ 30.25 \end{bmatrix}$ and A to $\begin{bmatrix} 0 & -2.5 \\ 3.1 & 0 \end{bmatrix}$ (see Table A.4 in Appendix A.1 for training u_0 and A values). The left subplot shows the transfer learned solutions (blue/orange) versus the true solutions (green/red) and the right subplot shows the network equation residuals.

3.4.3 Higher Order ODE

In addition to solving first order ODEs, we can also work with higher order equations. For instance, consider Equation 3.4 below. This is a second order linear differential equation which needs to be rewritten such that it can take the form of Equation 1.7 required for training.

$$\begin{cases} \dfrac{d^2u}{dt^2} + cu = f \\ \quad u(0) = v_1 \\ \left.\dfrac{du}{dt}\right|_{t=0} = v_2 \end{cases} \tag{3.4}$$

In order to reformulate Equation 3.4, we introduce a new variable w and claim that $w = \frac{du}{dt}$. This trick enables us to write our single second order ODE into a system of first order ODEs as follows:

$$\begin{cases} \dfrac{du}{dt} - w = 0 \\ \dfrac{dw}{dt} + cu = f \\ \quad u(0) = v_1 \\ \quad w(0) = v_2 \end{cases} \tag{3.5}$$

Training Outcomes

In Figure 3.7 we observe the training results for Equation 3.5. Again, it is clear the network has converged on an H representation due to the loss stabilizing below 10^{-4}, the mean squared error getting below 10^{-6}, and strong alignment between the network and true solutions in all four training sce-

narios. We note that because Equation 3.4 was rewritten with two variables (u and w), the solution plots in Figure 3.7 contain two sets of curves: one corresponding to the desired solution u for the original problem and the second representing the derivative of the desired solution u.

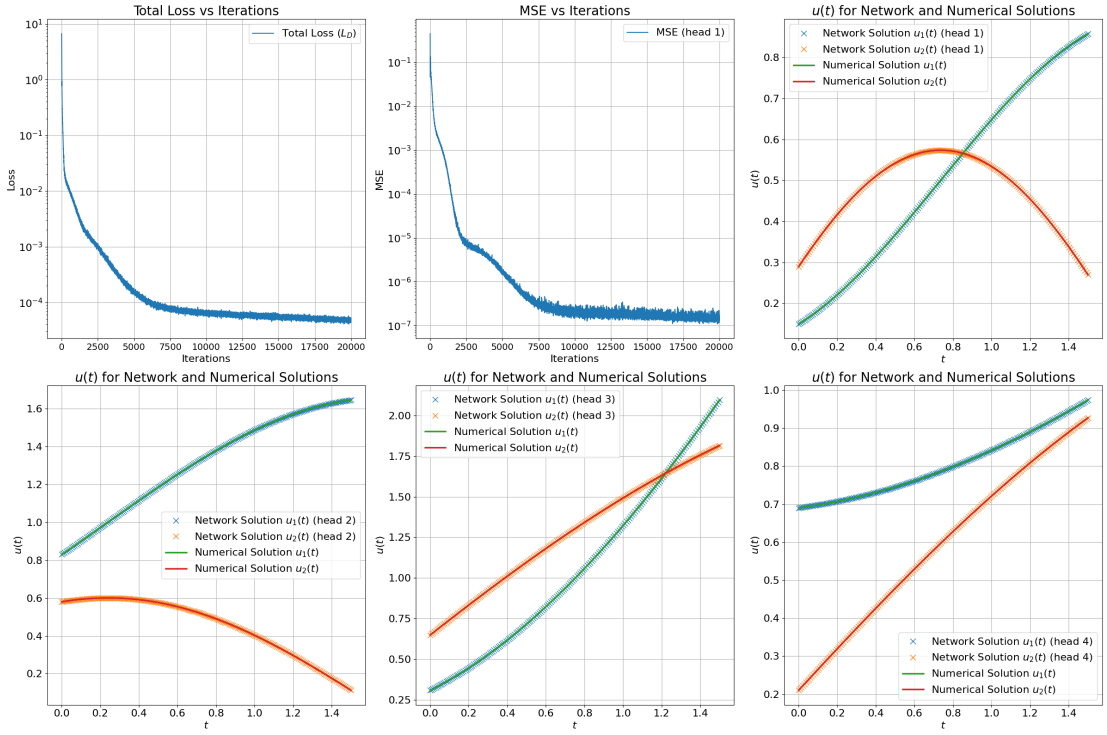


Figure 3.7: Training results for Equation 3.5 after 20000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and numerical solutions). The remaining four plots show the network learned solutions (blue/orange) versus the true solutions (green/red) for each head's equation configurations.

Change Initial Condition u_0 and Coefficient Matrix A

For Equation 3.5, we now perform a case of transfer learning where we change both u_0 and A. Figure 3.8 shows the results of changing both parameters. As mentioned earlier, one set of curves

(blue/green) represents the original solution u and the second set of curves represents the derivative of the solution u (orange/red).

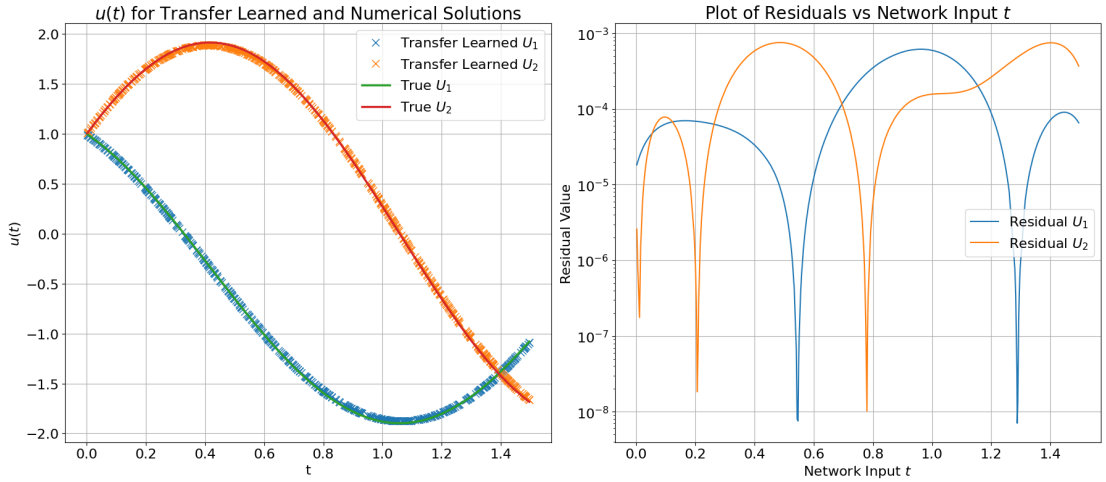


Figure 3.8: Transfer learning results after changing u_0 to $\begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$ and A to $\begin{bmatrix} 0 & 2.0 \\ -3.0 & 0 \end{bmatrix}$ (see Table A.6 in Appendix A.1 for training u_0 and A values). The left subplot shows the transfer learned solutions (blue/orange) versus the true solutions (green/red) and the right subplot shows the network equation residuals.

3.4.4 Large ODE System

We now solve a larger ODE system consisting of six second order ODE's as written in Equation 3.6. In this system, $f_1 \ldots f_{12}$ and $v_1 \ldots v_{12}$ are not time-dependent coefficients, but their values vary among the heads. Given that this system is second-order, similar to the example in Section 3.4.3, it also needs to be rewritten to match the form of Equation 1.7. In this case, we must introduce 6 new variables and the system would evolve to 12 equations. For clarity, we do not write the updated

system in the text here.

$$
\begin{cases}
\dfrac{d^2x_A}{dt^2} + (x_A - x_B) + (x_A - x_C) = f_1 \\
\dfrac{d^2y_A}{dt^2} + (y_A - y_B) + (y_A - y_C) = f_2 \\
\dfrac{d^2x_B}{dt^2} + (x_B - x_A) + (x_B - x_C) = f_3 \\
\dfrac{d^2y_B}{dt^2} + (y_B - y_A) + (y_B - y_C) = f_4 \\
\dfrac{d^2x_C}{dt^2} + (x_C - x_A) + (x_C - x_B) = f_5 \\
\dfrac{d^2y_C}{dt^2} + (y_C - y_A) + (y_C - y_B) = f_6 \\
x_A(0) = v_1, \quad \left.\dfrac{dx_A}{dt}\right|_{t=0} = v_2 \\
x_B(0) = v_3, \quad \left.\dfrac{dx_B}{dt}\right|_{t=0} = v_4 \\
x_C(0) = v_5, \quad \left.\dfrac{dx_C}{dt}\right|_{t=0} = v_6 \\
y_A(0) = v_7, \quad \left.\dfrac{dy_A}{dt}\right|_{t=0} = v_8 \\
y_B(0) = v_9, \quad \left.\dfrac{dy_B}{dt}\right|_{t=0} = v_{10} \\
y_C(0) = v_{11}, \quad \left.\dfrac{dy_C}{dt}\right|_{t=0} = v_{12}
\end{cases}
\tag{3.6}
$$

Training Outcomes

The results of training this larger ODE system can be seen in Figure 3.9. Despite the larger number of equations to represent, the training procedure yields a loss around 10^{-4} and a mean squared error of about 10^{-7} after only 12000 training iterations. Moreover, while slightly more difficult to keep

track of, Figure 3.9 still shows that all network solutions to the system tightly follow the reference solutions.

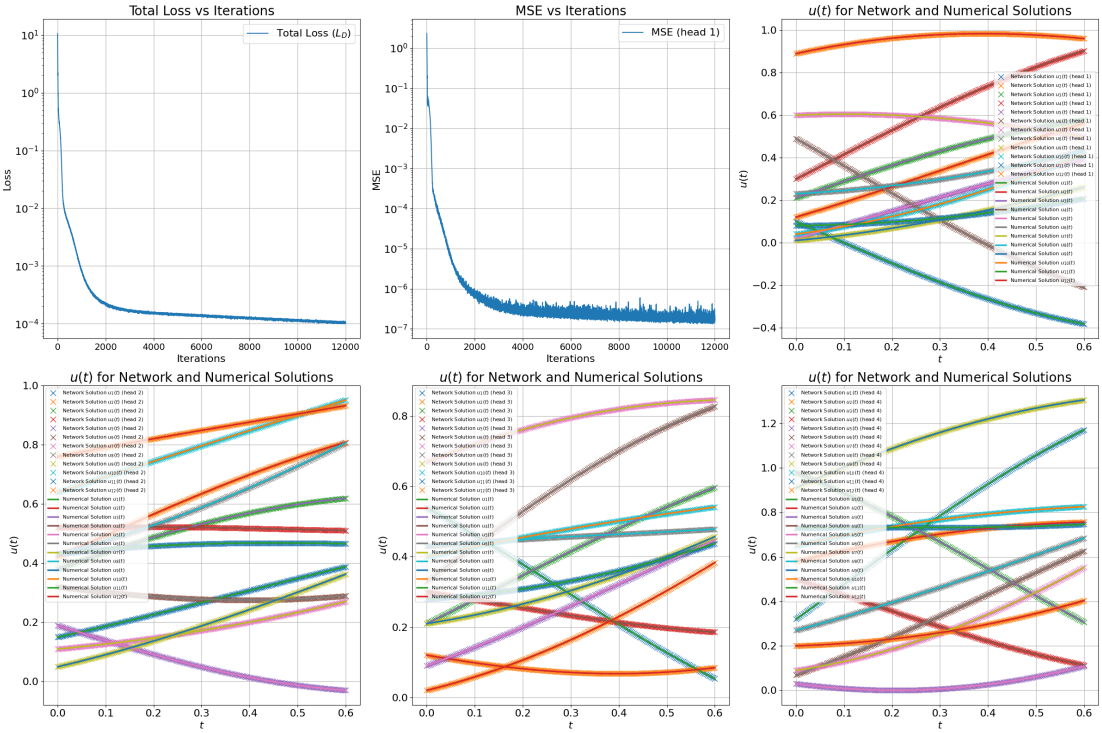


Figure 3.9: Training results for Equation 3.6 after 12000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and numerical solutions). The remaining four plots show the network learned solutions versus the true solutions for each head's equation configurations.

Change Initial Condition u_0

To transfer learn on the system shown in Equation 3.6, we change the u_0 value and plot the generated solution in the left-hand side of Figure 3.10. While there are many curves to represent, our approach scales and recreates the true solution curves with a high degree of accuracy. Specifically, the residuals for all network solutions in the right-hand side of Figure 3.10 are nearly always smaller

than 10^{-5} and have a median around 10^{-8}. This scalability and ability to immediately solve a system of equations on new, unseen sets of conditions are the biggest strengths of our work.

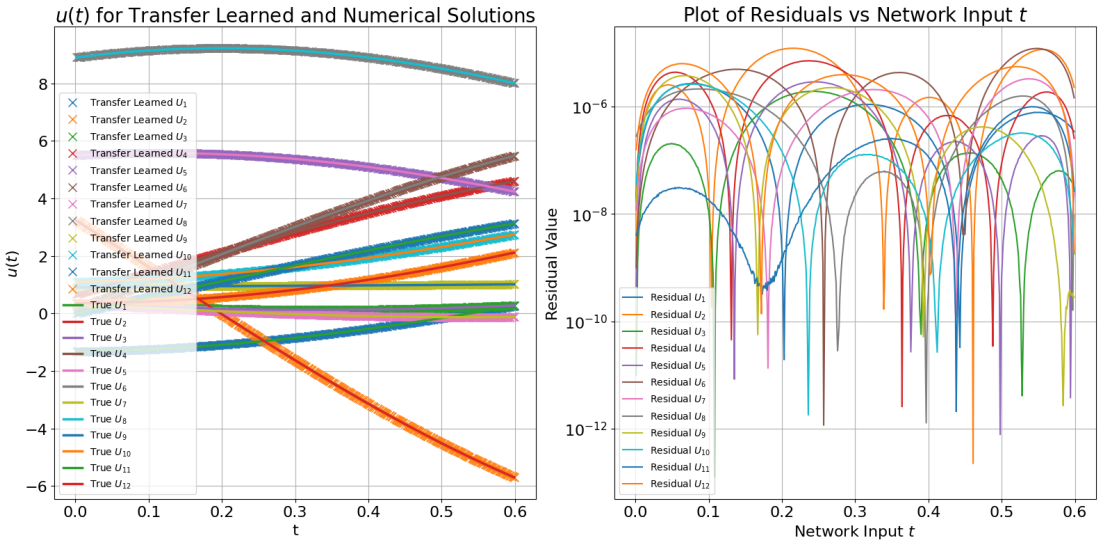


Figure 3.10: Transfer learning results after changing u_0 to $[-1.33, 3.25, 0.35, 0.67, 5.5, 0.12, 0.32, 8.9, 0.90, 1.09, 0.01, 0.35]^T$ (see Table A.8 in Appendix A.1 for training u_0 values). The left subplot shows the transfer learned solutions versus the true solutions and the right subplot shows the network equation residuals.

3.4.5 Non-constant Coefficient ODE

Aside from solving systems of ODEs with constant coefficients, we can also work with problems that have time-dependent coefficients. These problems are especially difficult given that they do not typically yield closed-form solutions. Equation 3.7 is an example of a non-constant coefficient ODE.

In this example, ct is a time-dependent coefficient.

$$\begin{cases} \dfrac{du}{dt} + ctu = f \\ \\ \quad u(0) = v \end{cases} \tag{3.7}$$

Training Outcomes

Training a network on Equation 3.7 yields the results shown in Figure 3.11 below. We observe that the loss achieves values below 10^{-4} and the mean squared error equals about 10^{-7} at the end of 10000 training iterations. All four training heads show close alignment between the true and network-learned solutions, even with the varying time-dependent u coefficient.

Change Coefficient Matrix A

In Figure 3.12, we showcase the results of changing the time-dependent A matrix. During training, the A matrix values used were $1.0t$, $1.25t$, $1.5t$, and $2.0t$, however in the left-hand side of Figure 3.12 we plot the transfer learned and true solution for $A = 5.0t$. It is possible to observe some slight deviations between the transfer learned and true solutions, especially at the tails of the solution space. Specifically, in the intervals for $t = [0, 0.25]$ and $t = [1.75, 2.0]$, these variations exist. In examining the residuals for this case, we note that they are slightly larger than most of the previous examples, however they still all remain smaller than 10^{-4} for the chosen time interval.

The deviations in the transfer learned solutions are to be expected. First, the transfer learned A value is over a factor of 2 larger than the maximum A used in the training region. Additionally, more

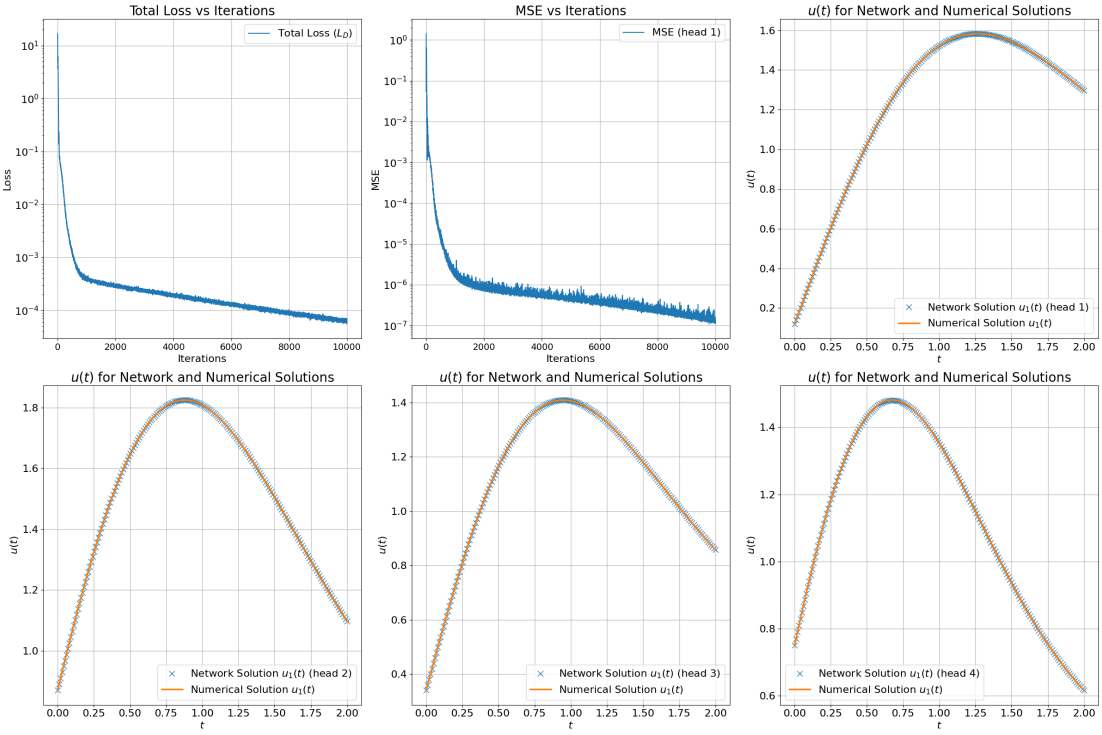


Figure 3.11: Training results for Equation 3.7 after 10000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and numerical solutions). The remaining four plots show the network learned solution (blue) versus the true solution (orange) for each head's equation configurations.

error is expected in the reconstruction of solutions where there is no closed-form solution, as is the case with Equation 3.7. In general, this example is significant in that it illustrates how our "one-shot approach" can be generalized to values well outside the training region and yield a solution quickly, but with the trade-off of accumulating more error than a standard numerical approach like Runge-Kutta.

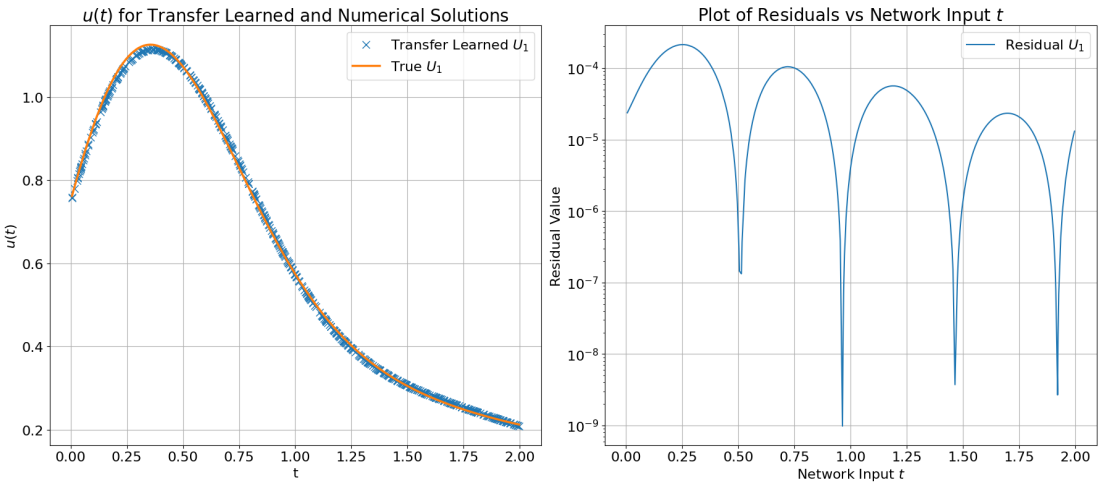


Figure 3.12: Transfer learning results after changing A to $5.0t$ (see Table A.10 in Appendix A.1 for A training values). The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

Change Initial Condition u_0 and Coefficient Matrix A

As a final example, we change both u_0 and A for Equation 3.7. Figure 3.13 includes the results of altering both parameters. It is worth emphasizing that by changing both parameters in this case, the equation we are able to solve has a very different end behavior and range of values it exhibits in the interval of interest. Again, to achieve these generalizations we make a trade-off in the accuracy of our solutions, although the residuals still remain at reasonable levels, with a median around 10^{-4}.

3.4.6 One-Shot Transfer Learning versus Numerical Methods Approach

To measure efficiency, we compare the run-times between the "one-shot transfer learning" approach and the Runge-Kutta method. In Table 3.1, we first examine the times needed between the two approaches when calculating 1000 solutions for different u_0 values. For "one-shot transfer learning",

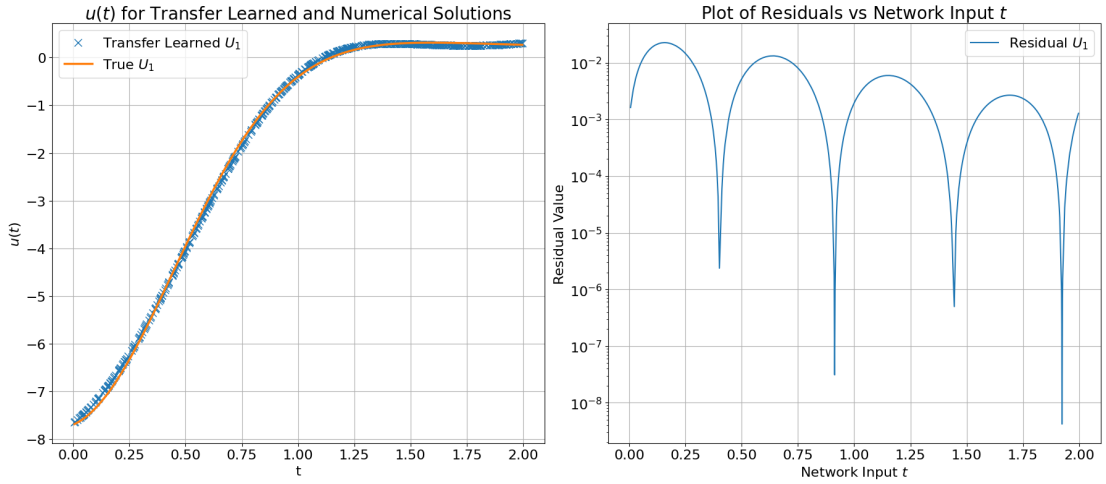


Figure 3.13: Transfer learning results after changing u_0 to -10.70 and A to $5.0t$ (see Table A.10 in Appendix A.1 for u_0 and A training values). The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

the time needed across the four examples is steady at around 0.6 to 0.7 seconds. On the other hand, the calculation time for Runge-Kutta tends around 6.0 to 6.4 seconds, depending on the problem at hand. Namely, "one-shot transfer learning" yields results about 10 times faster compared to the numerical approach, empirically highlighting a key advantage our approach offers. Due to the fact that only the $H_0^T u_0$ term needs to be recalculated in Equation 3.1, our approach yields instantaneous results for each new u_0. In other words, all but one term of Equation 3.1 remain fixed and most importantly, the M matrix is calculated and inverted once.

In addition to measuring the computation times for varying u_0, we also benchmark the run-time for varying A. In Table 3.2 shown below, we observe run-times for the four ODE problems after changing the A matrix 1000 times. For three out of the four problems shown, we see that Runge-Kutta yields results quicker than the transfer learning approach. However, it is worth noting that

ODE Problem	Average Time (sec.) Across 1000 Runs	
	"One-Shot" T.L.	*Runge-Kutta*
Single ODE	0.739	6.085
Two Coupled ODE	0.736	6.457
Higher Order ODE	0.605	6.411
Non-constant Coef. ODE	0.717	6.034

Table 3.1: Time needed to calculate differential equation solutions after changing u_0 1000 times. The table shows the run-times across four ODE problems presented in Section 3.4 for both "one-shot transfer learning" and a 4^{th} order Runge-Kutta implementation from SciPy.

the run-times across both methods are within the same order of magnitude for each example. These results are promising, especially considering that for each new A, our transfer learning approach requires a recalculation and inversion of the M matrix shown in Equation 3.1 and that our code is not nearly as optimized as that of the SciPy implementation of Runge-Kutta.

ODE Problem	Average Time (sec.) Across 1000 Runs	
	"One-Shot" T.L.	*Runge-Kutta*
Single ODE	9.848	4.317
Two Coupled ODE	4.416	4.987
Higher Order ODE	7.387	3.984
Non-constant Coef. ODE	8.439	4.112

Table 3.2: Time needed to calculate differential equation solutions after changing A 1000 times. The table shows the run-times across four ODE problems presented in Section 3.4 for both "one-shot transfer learning" and a 4^{th} order Runge-Kutta implementation from SciPy.

Nearly everything is interesting if you look into it deeply enough.

Richard Feynman

4

Transfer Learning PDEs

Performing transfer learning on PDEs is quite similar to ODEs, with a few key changes. Namely, the loss function we require is much more complex, as was covered in Chapter 2. Additionally, the derivation for the generalized weights W_0 requires a rewriting of a portion of the differential equation loss for ease of calculation. Aside from these adjustments, discussing the steps needed for transfer learning PDEs naturally builds off of the previous chapter.

In this chapter, we will not have an analogous version of Section 3.1, as the high-level components necessary for transfer learning are identical between both ODEs and PDEs. Moreover, in Section 4.3 below, the PDE results do not include a section on timing a numerical method versus our transfer learning approach. This is due to the fact that analytical solutions were used as the true/reference solutions when working with PDEs. For a run-time comparison of the "one-shot transfer learning" approach versus a numerical approach, refer back to Section 3.4.6.

4.1 Derivation of Generalized Equation Weights

Similar to Section 3.2, we can analytically solve for the weights of interest W_0 needed for transfer learning partial differential equations. This is due to the fact that the network loss function for PDEs, shown in Equation 2.2, is convex. We start the derivation for W_0 by beginning with the PDE loss function as shown in Equation 2.2:

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

Now, we know that $u = HW_0$ and this enables us to substitute into the loss function as follows:

$$L_{pde} = \frac{1}{n} \sum_{x \in \text{Batch}} (((D_u G^T) \circ I) \overrightarrow{1} + AHW_0 - f)^2 + \frac{1}{n} \sum_{x \in \text{Batch}} (H_0 W_0 - u_0(x))^2$$

At this point, we can perform a substitution to clean up the $((D_u G^T) \circ I) \overrightarrow{1}$ term. We define a matrix S such that $S_{ij} = \sum_k g_{ik} \frac{\partial H_{ij}}{\partial x_k}$ where g_{ik} refers to each element of the 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. This definition of S is what enables us to find that $((D_u G^T) \circ I) \overrightarrow{1} = SW_0$. Using this equality, our loss function can be simplified:

$$L_{pde} = \frac{1}{n} \sum_{x \in \text{Batch}} (SW_0 + AHW_0 - f)^2 + \frac{1}{n} \sum_{x \in \text{Batch}} (H_0 W_0 - u_0(x))^2$$

Next, we expand the quadratic terms inside both the differential equation and boundary condition terms:

$$L_{pde} = \frac{1}{n} \sum_{x \in \text{Batch}} (SW_0 + AHW_0 - f)^T (SW_0 + AHW_0 - f) + \frac{1}{n} \sum_{x \in \text{Batch}} (H_0 W_0 - u_0(x))^T (H_0 W_0 - u_0(x))$$

To use a more compact notation, we can split L_{pde} into L_x and L_0, corresponding to the differential equation and boundary condition terms, respectively. Hence, $L_x = \frac{1}{n} \sum_{x \in \text{Batch}} (SW_0 + AHW_0 - f)^T (SW_0 + AHW_0 - f)$ and $L_0 = \frac{1}{n} \sum_{x \in \text{Batch}} (H_0 W_0 - u_0(x))^T (H_0 W_0 - u_0(x))$. In

other words, this results in the overall loss function being written as the following:

$$L_{pde} = L_x + L_0$$

Now, we are able to multiply out the quadratic terms in the L_x and L_0 pieces by distributing the transpose and foiling out the individual terms. We omit outlining all the steps for this portion of the simplification, but show the results below:

$$L_x = \frac{1}{n} \sum_{x \in \text{Batch}} (W_0^T S^T S W_0 + W_0^T S^T A H W_0 - W_0^T S^T f + W_0^T H^T A^T S W_0 + W_0^T H^T A^T A H W_0 - W_0^T H^T A^T f - f^T (S W_0 + A H W_0 - f))$$

$$L_0 = \frac{1}{n} \sum_{x \in \text{Batch}} (W_0^T H_0^T H_0 W_0 - W_0^T H_0^T u_0 - u_0^T H_0 W_0 + u_0^T u_0)$$

In order to uncover the W_0 that minimizes our PDE loss function, we must take the gradient of the loss with respect to the generalized weights and solve for W_0:

$$\frac{\partial L_{pde}}{\partial W_0} = \frac{\partial L_x}{\partial W_0} + \frac{\partial L_0}{\partial W_0}$$

Similar to the ODE derivation, we exclude the detailed steps needed in taking the gradient and sim-

plifying individual terms. The result is as follows:

$$\frac{\partial L_{pde}}{\partial W_0} = \left[\frac{1}{n}\sum_{x\in\text{Batch}}\left(S^TAH + H^TA^TS + S^TS + H^TA^TAH + H_0^TH_0\right)\right]W_0 - \frac{1}{n}\sum_{x\in\text{Batch}} S^Tf - \frac{1}{n}\sum_{x\in\text{Batch}} H^TA^Tf - \frac{1}{n}\sum_{x\in\text{Batch}} H_0^Tu_0$$

With this, we can now set the variable $M = \left[\frac{1}{n}\sum_{x\in\text{Batch}}\left(S^TAH + H^TA^TS + S^TS + H^TA^TAH + H_0^TH_0\right)\right]$ in order to reduce our notation and set the expression equal to 0:

$$MW_0 - \frac{1}{n}\sum_{x\in\text{Batch}} S^Tf - \frac{1}{n}\sum_{x\in\text{Batch}} H^TA^Tf - \frac{1}{n}\sum_{x\in\text{Batch}} H_0^Tu_0 = 0$$

To isolate W_0, we can move all terms except for MW_0 over to the right-hand side:

$$MW_0 = \frac{1}{n}\sum_{x\in\text{Batch}} S^Tf + \frac{1}{n}\sum_{x\in\text{Batch}} H^TA^Tf + \frac{1}{n}\sum_{x\in\text{Batch}} H_0^Tu_0$$

Finally, we can multiply both sides of this equation by M^{-1}, and this yields the following closed-form representation of W_0:

$$W_0 = M^{-1}\left(\frac{1}{n}\sum_{x\in\text{Batch}} S^Tf + \frac{1}{n}\sum_{x\in\text{Batch}} H^TA^Tf + \frac{1}{n}\sum_{x\in\text{Batch}} H_0^Tu_0\right) \tag{4.1}$$

4.2 Performing One-Shot Transfer Learning

As mentioned in the introduction to this chapter, performing transfer learning on PDEs naturally follows from ODEs. After a multi-headed network has been trained with varying heads, only a single forward pass must be run on the trained model to acquire the H matrix. We save H for future use and can then proceed to leverage Equation 4.1 to compute W_0 for a particular u_0, A, G, and f. Ultimately, once the values of H and W_0 are found, the solutions to the differential equations of interest can be determined by solving $u = HW_0$.

Again, just like in the ODE case, the reader should observe that the most salient point of this transfer learning methodology is that for a selected equation form, the network training is completed only once. Unless the equation form changes, we are able to alter u_0, A, G, and f to values that have not been in the training set. In theory, while any parameter values can be used for transfer learning (assuming sufficient training), in practice we accumulate more errors in our transfer learned solutions the further we stray from the training space.

Depending on which parameter values want to be changed during transfer learning, different elements of Equation 4.1 need to be recomputed. In the case of changing u_0, all we need to do is recalculate $\frac{1}{n}\sum_{x\in\text{Batch}} H_0^T u_0$, assuming all other terms of the equation have been saved beforehand. Next, if we change the values of the A coefficient matrix, then we need to recompute both M (and invert it) and $\frac{1}{n}\sum_{x\in\text{Batch}} H^T A^T f$. Similarly, changing entries of the G matrix requires a recalculation and inversion of M as the M matrix depends on S which is built from the entries of G (see derivation of W_0 above). Finally, changing f is straightforward as it only requires recalculat-

ing $\frac{1}{n}\sum_{x\in\text{Batch}} S^T f$ and $\frac{1}{n}\sum_{x\in\text{Batch}} H^T A^T f$. As was pointed out in Section 3.3, changing u_0 or f is a much faster computation to carry out as it only requires additional matrix multiplications and additions, but changing A or G takes longer as it requires recomputing and inverting the $d \times d$ M matrix.

4.3 Results

Similar to Section 3.4, below we highlight training and "one-shot transfer learning" results on a set of partial differential equation examples. For each case, we include the equations of interest, the multi-headed training results, and a few situations of transfer learning different parameters. To learn about the details surrounding the training parameters and multi-headed training conditions for each PDE problem, refer to Appendix A.2. Unlike for ODEs, numerical methods were not used to produce reference solutions. Rather, analytical solutions were leveraged due to the availability of such solutions for the problems within the scope of this thesis.

4.3.1 Single PDE

The first partial differential equation problem we discuss is a linear first order equation as shown in Equation 4.2. In the example below, c_1, c_2, c_3, c_4, and f are variables which are not time-dependent.

$$\begin{cases} c_1 \dfrac{\partial u}{\partial x_1} + c_2 \dfrac{\partial u}{\partial x_2} + c_3 u = f \\ \qquad u(x_1, 0) = \sin(c_4 x_1) \end{cases} \tag{4.2}$$

Training Outcomes

In Figure 4.1, we show the training results based on Equation 4.2. We note that after 20000 iterations, the loss reaches below 10^{-6} and the mean squared error is well below 10^{-4}. In addition to these metrics, the network and true solution curves overlap directly on top of each other for all four "heads", exhibiting the appropriate curvatures. Given that Equation 4.2 has two independent variables, x_1 and x_2, we are able to plot the solution curves in three dimensions.

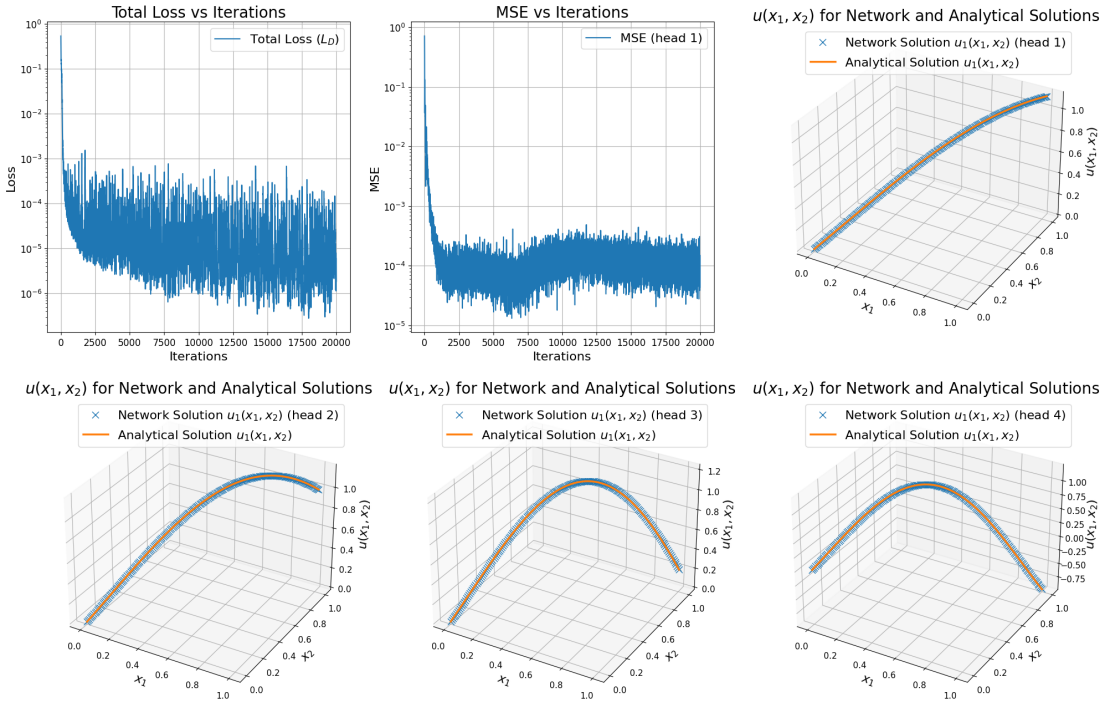


Figure 4.1: Training results for Equation 4.2 after 20000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and analytical solutions). The remaining four plots show the network learned solution (blue) versus the true solution (orange) for each head's equation configurations.

Change Boundary Condition u_0

For the first instance of PDE transfer learning, we change u_0 for Equation 4.2 and show the resulting network and true solution in Figure 4.2's left subplot. In this case, the boundary conditions used during training were $\sin(x_1)$, $\sin(1.5x_1)$, $\sin(2.0x_1)$, and $\sin(2.5x_1)$. While all c_4 coefficients were positive here, the boundary condition used for transfer learning was $\sin(-1.3x_1)$ - namely a negative c_4 value. Despite using a value far from the training region values, the right-hand side of Figure 4.1 illustrates the accuracy of our solution as the residuals are nearly all below 10^{-5}.

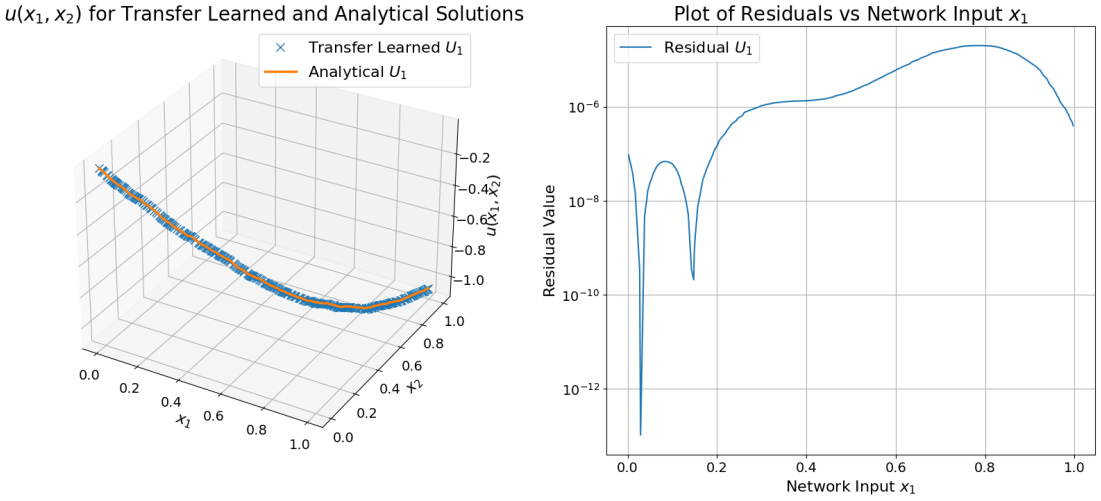


Figure 4.2: Transfer learning results after changing u_0 to $\sin(-1.3x_1)$ (see Table A.12 in Appendix A.2 for training u_0 values). The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

Change Boundary Condition u_0 and Coefficient Matrix A

As a more difficult example, both u_0 and A can be changed when transfer learning. Figure 4.3 shows the results of changing u_0 to $\sin(-1.3x_1)$ and A to 0.38, both of which are values that were not

used during the training process. Again, we observe that the network and analytical solutions align closely, however the residuals in the right subplot of Figure 4.3 are slightly larger than those we saw in the previous case of only adjusting u_0. This is to be expected given that more elements of Equation 4.1 need to be reconstructed when calculating W_0, leading to more error being accumulated.

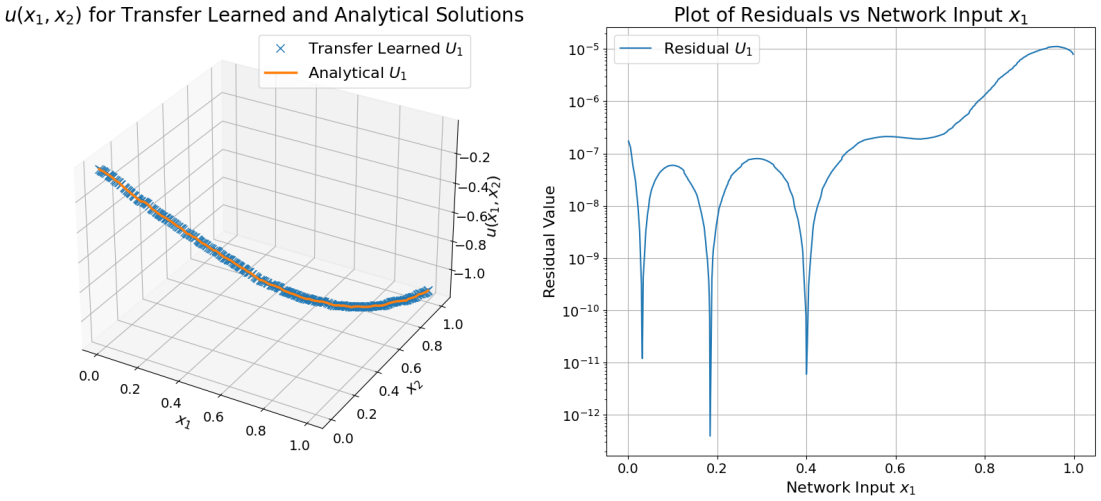


Figure 4.3: Transfer learning results after changing u_0 to $\sin(-1.3x_1)$ and A to 0.38 (see Table A.12 in Appendix A.2 for training u_0 and A values). The left subplot shows the transfer learned solution (blue) versus the true solution (orange) and the right subplot shows the network equation residuals.

4.3.2 Two Coupled PDE System

Next, we discuss the system of linear PDEs shown in Equation 4.3 below. In this problem, $c_1 ... c_6, f_1$, and f_2 are not dependent on time.

$$\begin{cases} c_1 \dfrac{\partial u_1}{\partial x_1} + c_2 \dfrac{\partial u_1}{\partial x_2} = f_1 \\ c_3 \dfrac{\partial u_2}{\partial x_1} + c_4 \dfrac{\partial u_2}{\partial x_2} = f_2 \\ \qquad u_1(x_1, 0) = c_5 x_1 \\ \qquad u_2(x_1, 0) = c_6 x_1 \end{cases} \tag{4.3}$$

Training Outcomes

Figure 4.4 illustrates that our network for Equation 4.3 has been sufficiently trained. Specifically, we observe that the loss and mean squared error are both much smaller than 10^{-4} and the network and analytical solutions for all four training conditions are aligned tightly to one another.

Change Coefficient Matrix G

For Equation 4.3, we include an example of transfer learning by changing G as shown in Figure 4.5 below. The value of G is changed to $\begin{bmatrix} 3.2 & 5.1 \\ 1.4 & 7.3 \end{bmatrix}$, which effectively alters all coefficients in front of Equation 4.3's partial derivative terms. Given that the analytical solution to this system is relatively simple (two linear equations), the curves in Figure 4.5 appear similar to the training plots. How-

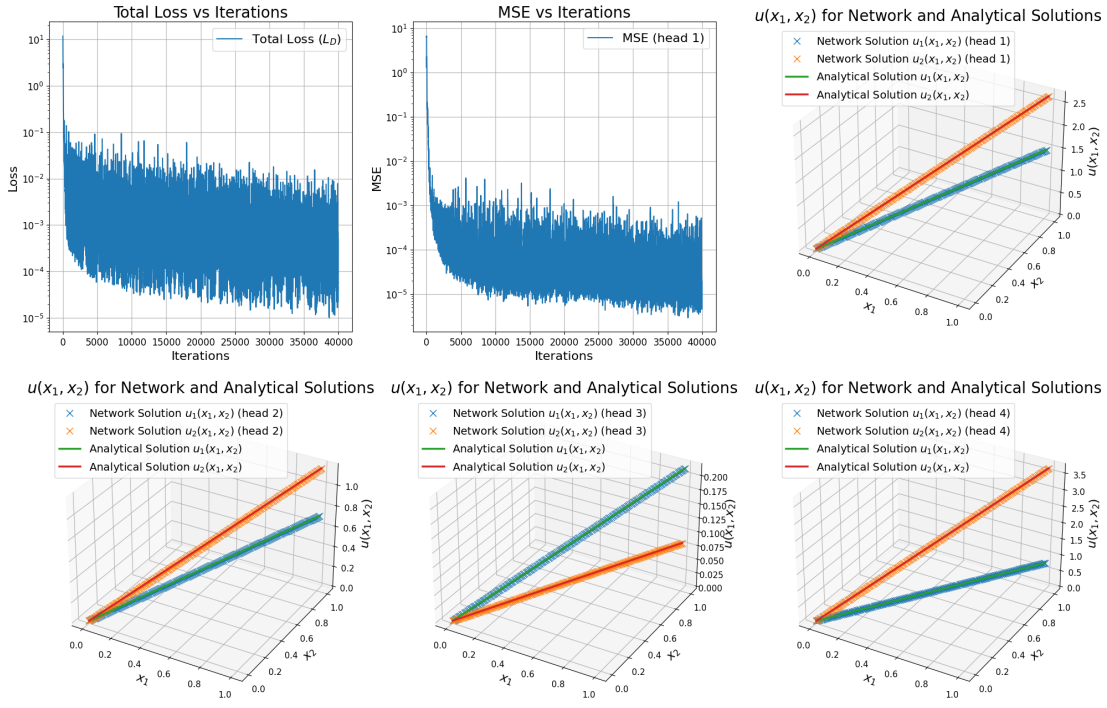


Figure 4.4: Training results for Equation 4.3 after 40000 iterations, using four heads. In the top row, the first plot illustrates the loss vs. iterations (differential equation residuals) and the second plot highlights the mean squared error vs iterations (average of the square of the difference between network and analytical solutions). The remaining four plots show the network learned solutions (blue/orange) versus the true solutions (green/red) for each head's equation configurations.

ever, we do note that the scale of Figure 4.5's left-hand side subplot is different compared to the set of four training conditions. Nonetheless, we observe that the residuals for both curves are nearly always smaller than 10^{-4}, suggestive of a well constructed solution to the equation.

Change Boundary Condition u_0 and Coefficient Matrix G

As a final case, we change the boundary condition u_0 in addition to the coefficient matrix G. In Figure 4.6 below, the left-hand plot shows the resulting curves. In this case, u_0 is changed to $\begin{bmatrix} 2.0x_1 \\ -1.1x_1 \end{bmatrix}$

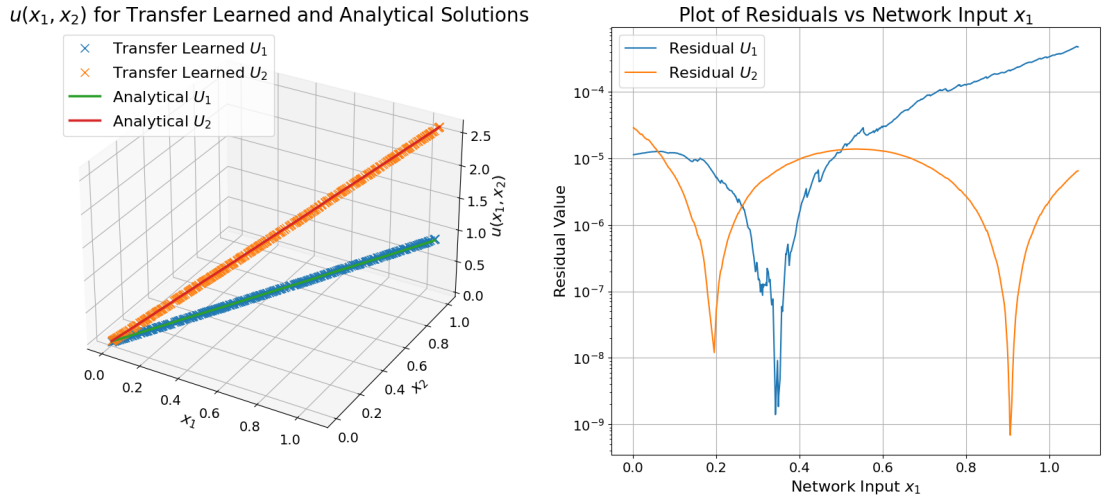


Figure 4.5: Transfer learning results after changing G to $\begin{bmatrix} 3.2 & 5.1 \\ 1.4 & 7.3 \end{bmatrix}$ (see Table A.14 in Appendix A.2 for training G values). The left subplot shows the transfer learned solutions (blue/orange) versus the true solutions (green/red) and the right subplot shows the network equation residuals.

and G is changed to $\begin{bmatrix} 3.2 & 5.1 \\ 1.4 & 7.3 \end{bmatrix}$. It is worth pointing out that the u_0 values used during training contained all positive coefficients, however in this transfer learning example we were able to successfully construct a solution with a negative coefficient in the boundary condition.

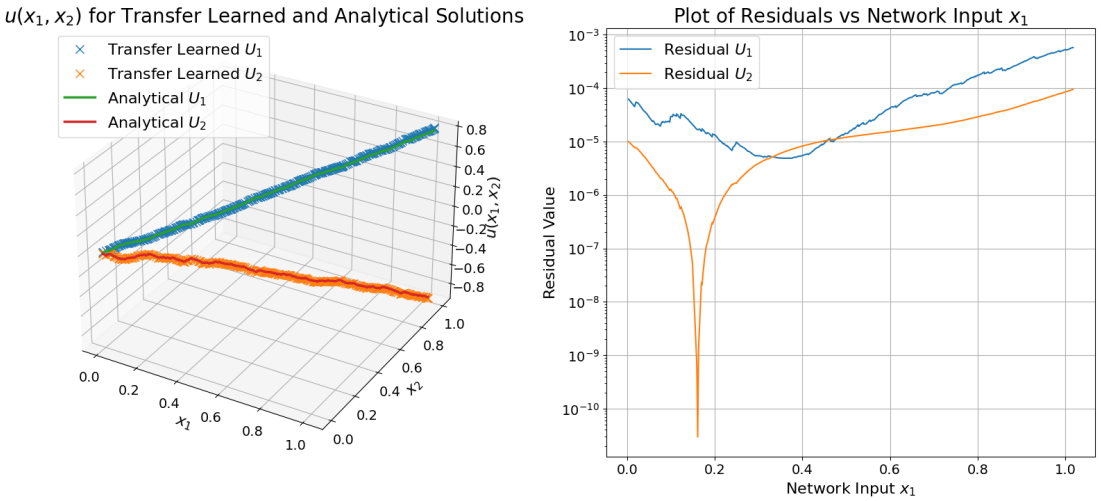


Figure 4.6: Transfer learning results after changing u_0 to $\begin{bmatrix} 2.0x_1 \\ -1.1x_1 \end{bmatrix}$ and G to $\begin{bmatrix} 3.2 & 5.1 \\ 1.4 & 7.3 \end{bmatrix}$ (see Table A.14 in Appendix A.2 for training G values). The left subplot shows the transfer learned solutions (blue/orange) versus the true solutions (green/red) and the right subplot shows the network equation residuals.

It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment.

Carl Friedrich Gauss

5

Discussion

The principal objective of this work was to provide a robust and well-defined method of training linear ordinary and partial differential equations and derive a mathematical procedure to transfer the general, learned representations onto different equations. This thesis serves as a generalization of the work researchers originally presented in *One-Shot Transfer Learning of Physics-Informed Neural Networks*[4]. Namely, the contributors in the original work were able to solve a variety of differential equations and subsequently transfer learn the solutions, however the transfer learning procedure was not mathematically generalizable. In both Chapters 3 and 4 of this thesis, we provided such a closed form, generalizable representation of solving for the weights W_0 needed to transfer learn any linear ODE or PDE system.

With a generalizable training and transfer learning methodology, our work provides the ability to quickly solve many systems of differential equations that yield interesting patterns not observed during training. Specifically, by being able to adjust parameters such as the 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 regardless of the varying behaviors we aim to recreate, all that needs to be done is a recalculation of the weights W_0 - no additional fine-tuning is performed.

This ability to reconstruct solutions in "one-shot" is particularly valuable when dealing with extraordinary large systems of differential equations. Consider a system of ordinary differential equations describing the movement of thousands of particles in two dimensions based on a set of initial conditions describing their starting positions. Naturally, training a neural network on the system of equations needed to represent such a large system would be a time consuming endeavour. In such a

system, applying a numerical method once would likely yield a solution much quicker with a high degree of fidelity. However, in the situation that we want to learn about how different starting positions affect the movement of the particles over time, a numerical approach would be quite costly. In other words, having to resolve the system each time a condition or coefficient changes would be expensive to accomplish numerically. On the other hand, the cost of training a network once is a much cheaper price to pay in the case that we are interested in solving variants of the system multiple times. After the first, each instance of resolving the system can be done nearly instantaneously with our transfer learning approach.

As another practical application of this work, our methodology provides researchers with a potential avenue to reuse training efforts carried out by others. More concretely, if a team trained a network on a system of equations (such as the particle-based example described above) and performed transfer learning on this system, they can save their solved weights W_0. After saving these weights, they could upload them to an online community portal along with details about the system they solved for others to leverage. In this way, other researchers can immediately solve the same set of equations without having to be concerned about the initial training phase. This ability to effortlessly reuse learned training insights and apply them onto other systems does not exist for numerical methods, hence demonstrating the concrete advantage our transfer learning approach would bring for multiple research communities.

While solving differential equations with neural networks brings forth many key advantages, it is worth noting that this a relatively new area of research that does not have the well established theory and vast number of contributions which numerical methods provide. Namely, it is known that

neural networks do not provide a guarantee in terms of training stability or convergence. Hence, for most real-world, critical applications where accuracy of solutions is paramount, our network-based solution should be used in conjunction with numerical approaches.

Once we accept our limits, we go beyond them.

Albert Einstein

6

Conclusion

In this thesis, we have presented a robust methodology to apply "one-shot transfer learning" to systems of linear ordinary and partial differential equations. Namely, we have described a process to train physics informed neural networks on vectorized representations of differential equations with varying conditions across multiple "heads". We then 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. Ultimately, both H and W_0 were used to rapidly find the equation solutions u since $u = HW_0$.

Leveraging the "one-shot transfer learning" approach on a suite of ODE and PDE systems illustrated that we can train networks on different equation coefficients and initial or boundary conditions and then transfer the learned representations to new equations of the same form. We demonstrated that we are not limited by the order of equations, the number of equations, or the time-dependence of equation coefficients. In this regard, our approach proves immensely useful in effectively solving differential equations for a wide variety of cases.

However, despite the strengths of our proposed approach, there are still limitations worth emphasizing. First, the proposed "one-shot" approach discussed in this thesis is only applicable to linear differential equations. In the case of nonlinear problems, one must solve such equations with a different method, namely a perturbation approach - this technique would iteratively solve nonlinear problems by leveraging the W_0 formulation presented in this thesis. Second, we must be aware that "one-shot transfer learning" does not typically yield solutions with 0 error, as is the case for analytical solutions (where they exist). As was highlighted with residual plots in both Chapters 3 and 4, the

further we stray away from the training values of equation coefficients and initial value or boundary conditions, the more error we will accumulate in our solution constructions.

In general, training PINNs on systems of differential equations with multiple "heads" is a computationally expensive endeavour. For the results shown in this thesis, all training procedures used at most 40000 iterations and exactly 4 heads. While the results in this text included transfer learned solutions that had a high degree of accuracy, a more exhaustive training process would be fruitful. In other words, if computational resources permit, training many more heads could ensure that the latent space H captures an even more general representation of a problem's form. In turn, this would provide added flexibility in transfer learning to conditions far from the training regime.

We conclude this thesis by highlighting a few avenues for future work. One opportunity exists in applying the network training and transfer learning methodology to a larger scale, real-world problem. One such problem in current consideration is the simulation of many particles in a box as described in Chapter 5. A second direction for future work exists in enhancing and implementing the current code into a differential equation library. At the time of writing, plans exist to incorporate the generalized "one-shot transfer learning" code into the NeuroDiffEq library. Finally, as alluded to earlier, given that our current work only deals with linear systems, we are limited to a particular subset of problems. Transfer learning nonlinear systems would open up the potential to examine problems that exhibit interesting behaviors such as the non-linear Burgers' and Hamilton equations. To solve such problems, we could build upon our existing work to leverage the perturbation approach described above.

A

Appendix

A.1 ODE Training Details

In the tables that follow below, we include detailed information about the parameter values used in training the networks for each of the ODE examples shown in Chapter 3. Additionally, we specify the A, u_0, and f values used for each head during the training procedures.

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 A.1: Parameters used for training single ODE shown in Section 3.4.1

Head	A	u_0	f
1	1.05	0.12	2.0
2	1.15	0.87	2.0
3	0.55	0.34	2.0
4	0.25	0.75	2.0

Table A.2: Head conditions used for training single ODE shown in Section 3.4.1

Training Parameter	Value
Number of Iterations	20000
Number of Grid Points	512
Grid Space	$[0, 1]$
Number of Hidden Layers	3
Nodes per Hidden Layer	$[128, 128, 256]$
Optimizer	`SGD (momentum=0.9)`
Learning Rate	0.001
Activation	`Tanh`

Table A.3: Parameters used for training coupled ODE system shown in Section 3.4.2

Head	A	u_0	f
1	$\begin{bmatrix} 0.9 & 0.43 \\ 1.1 & 0.78 \end{bmatrix}$	$\begin{bmatrix} 0.12 \\ 0.35 \end{bmatrix}$	$\begin{bmatrix} 2.0 \\ 4.0 \end{bmatrix}$
2	$\begin{bmatrix} 1.0 & 0.27 \\ 0.19 & 0.53 \end{bmatrix}$	$\begin{bmatrix} 0.87 \\ 0.62 \end{bmatrix}$	$\begin{bmatrix} 2.0 \\ 4.0 \end{bmatrix}$
3	$\begin{bmatrix} 0.67 & 1.61 \\ 1.8 & 1.99 \end{bmatrix}$	$\begin{bmatrix} 0.34 \\ 0.69 \end{bmatrix}$	$\begin{bmatrix} 2.0 \\ 4.0 \end{bmatrix}$
4	$\begin{bmatrix} 0.35 & 0.79 \\ 0.75 & 0.55 \end{bmatrix}$	$\begin{bmatrix} 0.75 \\ 0.25 \end{bmatrix}$	$\begin{bmatrix} 2.0 \\ 4.0 \end{bmatrix}$

Table A.4: Head conditions used for training coupled ODE system shown in Section 3.4.2

Training Parameter	Value
Number of Iterations	20000
Number of Grid Points	512
Grid Space	$[0, 1.5]$
Number of Hidden Layers	3
Nodes per Hidden Layer	$[128, 128, 256]$
Optimizer	`SGD (momentum=0.9)`
Learning Rate	0.001
Activation	`Tanh`

Table A.5: Parameters used for training higher order ODE system shown in Section 3.4.3

Head	A	u_0	f
1	$\begin{bmatrix} 0.0 & -1.0 \\ 2.0 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 0.15 \\ 0.29 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$
2	$\begin{bmatrix} 0.0 & -1.2 \\ 1.0 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 0.83 \\ 0.58 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$
3	$\begin{bmatrix} 0.0 & -0.93 \\ 0.21 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 0.31 \\ 0.65 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$
4	$\begin{bmatrix} 0.0 & -0.32 \\ 0.65 & 0.0 \end{bmatrix}$	$\begin{bmatrix} 0.69 \\ 0.21 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$

Table A.6: Head conditions used for training higher order ODE system shown in Section 3.4.3

Training Parameter	Value
Number of Iterations	12000
Number of Grid Points	512
Grid Space	$[0, 0.6]$
Number of Hidden Layers	3
Nodes per Hidden Layer	$[128, 128, 256]$
Optimizer	`SGD (momentum=0.9)`
Learning Rate	0.001
Activation	`Tanh`

Table A.7: Parameters used for training large ODE system shown in Section 3.4.4

Head	A	u_0	f
1	$\begin{bmatrix} 0_{6\times6} & \star_{6\times6} \\ -I_{6\times6} & 0_{6\times6} \end{bmatrix}_{12\times12}$	$\begin{bmatrix} 0.1 & \vdots \\ 0.12 & 0.6 \\ 0.21 & 0.23 \\ 0.3 & 0.01 \\ 0.02 & 0.04 \\ 0.49 & 0.08 \\ \vdots & 0.89 \end{bmatrix}_{12\times1}$	$\begin{bmatrix} 0.1 & \vdots \\ 0.2 & 0.0 \\ 0.15 & 0.0 \\ 0.13 & 0.0 \\ 0.18 & 0.0 \\ 0.25 & 0.0 \\ \vdots & 0.0 \end{bmatrix}_{12\times1}$
2	$\begin{bmatrix} 0_{6\times6} & \star_{6\times6} \\ -I_{6\times6} & 0_{6\times6} \end{bmatrix}_{12\times12}$	$\begin{bmatrix} 0.15 & \vdots \\ 0.42 & 0.11 \\ 0.38 & 0.43 \\ 0.52 & 0.05 \\ 0.19 & 0.64 \\ 0.32 & 0.43 \\ \vdots & 0.76 \end{bmatrix}_{12\times1}$	$\begin{bmatrix} 0.1 & \vdots \\ 0.2 & 0.0 \\ 0.15 & 0.0 \\ 0.13 & 0.0 \\ 0.18 & 0.0 \\ 0.25 & 0.0 \\ \vdots & 0.0 \end{bmatrix}_{12\times1}$
3	$\begin{bmatrix} 0_{6\times6} & \star_{6\times6} \\ -I_{6\times6} & 0_{6\times6} \end{bmatrix}_{12\times12}$	$\begin{bmatrix} 0.54 & \vdots \\ 0.12 & 0.67 \\ 0.21 & 0.43 \\ 0.3 & 0.21 \\ 0.09 & 0.4 \\ 0.33 & 0.28 \\ \vdots & 0.02 \end{bmatrix}_{12\times1}$	$\begin{bmatrix} 0.1 & \vdots \\ 0.2 & 0.0 \\ 0.15 & 0.0 \\ 0.13 & 0.0 \\ 0.18 & 0.0 \\ 0.25 & 0.0 \\ \vdots & 0.0 \end{bmatrix}_{12\times1}$
4	$\begin{bmatrix} 0_{6\times6} & \star_{6\times6} \\ -I_{6\times6} & 0_{6\times6} \end{bmatrix}_{12\times12}$	$\begin{bmatrix} 0.32 & \vdots \\ 0.58 & 0.09 \\ 0.98 & 0.27 \\ 0.5 & 0.91 \\ 0.03 & 0.65 \\ 0.07 & 0.73 \\ \vdots & 0.2 \end{bmatrix}_{12\times1}$	$\begin{bmatrix} 0.1 & \vdots \\ 0.2 & 0.0 \\ 0.15 & 0.0 \\ 0.13 & 0.0 \\ 0.18 & 0.0 \\ 0.25 & 0.0 \\ \vdots & 0.0 \end{bmatrix}_{12\times1}$

Table A.8: Head conditions used for training large ODE system shown in Section 3.4.4. In the "A" column, $\star$ is a 6×6 matrix where the first row is $[2, 0, -1, 0, -1, 0]$ and every subsequent row is a right-circular shift of the previous.

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 A.9: Parameters used for training non-constant coefficient ODE shown in Section 3.4.5

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 A.10: Head conditions used for training non-constant coefficient ODE shown in Section 3.4.5

A.2 PDE Training Details

In the tables that follow below, we include detailed information about the parameter values used in training the networks for each of the PDE examples shown in Chapter 4. Additionally, we specify the A, G, u_0, and f values used for each head during the training procedures.

Training Parameter	Value
Number of Iterations	20000
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 A.11: Parameters used for training single PDE shown in Section 4.3.1

Head	A	G	u_0	f
1	0.25	$\begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}$	$\sin(x_1)$	0.0
2	0.5	$\begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}$	$\sin(1.5x_1)$	0.0
3	0.75	$\begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}$	$\sin(2.0x_1)$	0.0
4	1.0	$\begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}$	$\sin(2.5x_1)$	0.0

Table A.12: Head conditions used for training single PDE shown in Section 4.3.1

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 A.13: Parameters used for training coupled PDE shown in Section 4.3.2

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 A.14: Head conditions used for training coupled PDE shown in Section 4.3.2

References

[1] Anthony, L. & Lashkia, G. (2003). Mover: a machine learning tool to assist in the reading and writing of technical papers. *IEEE Transactions on Professional Communication*, 46(3), 185–193.

[2] Cuomo, S., Cola, V. S. D., Giampaolo, F., Rozza, G., Raissi, M., & Piccialli, F. (2022). Scientific machine learning through physics-informed neural networks: Where we are and what's next. *CoRR*, abs/2201.05624.

[3] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4), 303–314.

[4] Desai, S., Mattheakis, M., Joy, H., Protopapas, P., & Roberts, S. J. (2021). One-shot transfer learning of physics-informed neural networks. *CoRR*, abs/2110.11286.

[5] Djeumou, F., Neary, C., Goubault, E., Putot, S., & Topcu, U. (2022). Neural networks with physics-informed architectures and constraints for dynamical systems modeling. In R. Firoozi, N. Mehr, E. Yel, R. Antonova, J. Bohg, M. Schwager, & M. Kochenderfer (Eds.), *Proceedings of The 4th Annual Learning for Dynamics and Control Conference*, volume 168 of *Proceedings of Machine Learning Research* (pp. 263–277).: PMLR.

[6] Grohs, P., Hornung, F., Jentzen, A., & von Wurstemberger, P. (2018). A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of black-scholes partial differential equations.

[7] Han, J., Jentzen, A., & E, W. (2018). Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34), 8505–8510.

[8] Lagaris, I., Likas, A., & Fotiadis, D. (1997). Artificial neural network methods in quantum mechanics. *Computer Physics Communications*, 104(1), 1–14.

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

[10] Maclaurin, D., Duvenaud, D., & Adams, R. P. (2015). Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*.

[11] Mattheakis, M., Sondak, D., Dogra, A. S., & Protopapas, P. (2022). Hamiltonian neural networks for solving equations of motion. *Physical Review E*, 105(6).

[12] McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (2006). A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, 27(4), 12.

[13] Pang, X., Zhao, Z., & Weng, Y. (2021). The role and impact of deep learning methods in computer-aided diagnosis using gastrointestinal endoscopy. *Diagnostics*, 11(4).

[14] Paticchio, A., Scarlatti, T., Mattheakis, M., Protopapas, P., & Brambilla, M. (2020). Semi-supervised neural networks solve an inverse problem for modeling covid-19 spread. *CoRR*, abs/2010.05074.

[15] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[16] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., & SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17, 261–272.