machine learning & knowledge extraction

MDPI

Article

Deep Theory of Functional Connections: A New Method for Estimating the Solutions of Partial Differential Equations

Carl Leake * and Daniele Mortari

Aerospace Engineering, Texas A&M University, College Station, TX 77843, USA; mortari@tamu.edu
* Correspondence: leakec@tamu.edu

Received: 11 February 2020; Accepted: 9 March 2020; Published: 12 March 2020

Abstract: This article presents a new methodology called Deep Theory of Functional Connections (TFC) that estimates the solutions of partial differential equations (PDEs) by combining neural networks with the TFC. The TFC is used to transform PDEs into unconstrained optimization problems by analytically embedding the PDE's constraints into a "constrained expression" containing a free function. In this research, the free function is chosen to be a neural network, which is used to solve the now unconstrained optimization problem. This optimization problem consists of minimizing a loss function that is chosen to be the square of the residuals of the PDE. The neural network is trained in an unsupervised manner to minimize this loss function. This methodology has two major differences when compared with popular methods used to estimate the solutions of PDEs. First, this methodology does not need to discretize the domain into a grid, rather, this methodology can randomly sample points from the domain during the training phase. Second, after training, this methodology produces an accurate analytical approximation of the solution throughout the entire training domain. Because the methodology produces an analytical solution, it is straightforward to obtain the solution at any point within the domain and to perform further manipulation if needed, such as differentiation. In contrast, other popular methods require extra numerical techniques if the estimated solution is desired at points that do not lie on the discretized grid, or if further manipulation to the estimated solution must be performed.

Keywords: deep learning; neural network; theory of functional connections; partial differential equation

1. Introduction

Partial differential equations (PDEs) are a powerful mathematical tool that is used to model physical phenomena, and their solutions are used to simulate, design, and verify the design of a variety of systems. PDEs are used in multiple fields including environmental science, engineering, finance, medical science, and physics, to name a few. Many methods exist to approximate the solutions of PDEs. The most famous of these methods is the finite element method (FEM) [1–3]. FEM has been incredibly successful in approximating the solution to PDEs in a variety of fields including structures, fluids, and acoustics. However, FEM does have some drawbacks.

FEM discretizes the domain into elements. This works well for low-dimensional cases, but the number of elements grows exponentially with the number of dimensions. Therefore, the discretization becomes prohibitive as the number of dimensions increases. Another issue is that FEM solves the PDE at discrete nodes, but if the solution is needed at locations other than these nodes, an interpolation scheme must be used. Moreover, extra numerical techniques are needed to perform further manipulation of the FEM solution.

Reference [4] explored the use of neural networks to solve PDEs, and showed that the use of neural networks avoids these problems. Rather than discretizing the entire domain into a number of elements that grows exponentially with the dimension, neural networks can sample points randomly from the domain. Moreover, once the neural network is trained, it represents an analytical approximation of the PDE. Consequently, no interpolation scheme is needed when estimating the solution at points that did not appear during training, and further analytical manipulation of the solution can be done with ease. Furthermore, Ref. [4] compared the neural network method with FEM on a set of test points that did not appear during training (i.e., points that were not the nodes in the FEM solution), and showed that the solution obtained by the neural network generalized well to points outside of the training data. In fact, the maximum error on the test set of data was never more than the maximum error on the training set of data. In contrast, the FEM had more error on the test set than on the training set. In one case, the test set had approximately three orders of magnitude more error than the training set. In short, Ref. [4] presents strong evidence that neural networks are useful for solving PDEs.

However, what was presented in Ref. [4] can still be improved. In Ref. [4] the boundary constraints are managed by adding extra terms to the loss function. An alternative method is to encapsulate the boundary conditions, by posing the solution in such a way that the boundary conditions must be satisfied, regardless of the values of the training parameters in the neural network. References [5,6] manage boundary constraints in this way when solving PDEs with neural networks by using a method similar to the Coons' patch [7] to satisfy the boundary constraints exactly.

Exact boundary constraint satisfaction is of interest for a variety of problems, particularly when confidence in the constraint information is high. This is especially important for physics informed problems. Moreover, embedding the boundary conditions in this way means that the neural network needs to sample points from the interior of the domain only, not the domain and the boundary. While the methods presented in [5,6] work well for low-dimensional PDEs with simple boundary constraints, they lack a mechanized framework for generating expressions that embed higher-dimensional or more complex constraints while maintaining a neural network with free-to-choose parameters. For example, the fourth problem in the results section cannot be solved using the solution forms shown in Ref. [5,6]. Luckily, a framework that can embed higher-dimensional or more complex constraints has already been invented: The Theory of Functional Connections (TFC) [8,9]. In Ref. [10], TFC was used to embed constraints into support vector machines, but left embedding constraints into neural networks to future work. This research shows how to embed constraints into neural networks with the TFC, and leverages this technique to numerically estimate the solutions of PDEs. Although the focus of this article is a new technique for numerically estimating the solutions of PDEs, the article's contribution to the machine learning community is farther reaching, as the ability to embed constraints into neural networks has the potential to improve performance when solving any problem that has constraints, not just differential equations, with neural networks.

TFC is a framework that is able to satisfy many types of boundary conditions while maintaining a function that can be freely chosen. This free function can be chosen, for example, to minimize the residual of a differential equation. TFC has already been used to solve ordinary differential equations with initial value constraints, boundary value constraints, relative constraints, integral constraints, and linear combinations of constraints [8,11–13]. Recently, the framework was extended to n-dimensions [9] for constraints on the value and arbitrary order derivative of $(n-1)$-dimensional manifolds. This means the TFC framework can now generate constrained expressions that satisfy the boundary constraints of multidimensional PDEs [14].

2. Theory of Functional Connections

The Theory of Functional Connections (TFC) is a mathematical framework designed to turn constrained problems into unconstrained problems. This is accomplished through the use of constrained expressions, which are functionals that represent the family of all possible functions that satisfy the problem's constraints. This technique is especially useful when solving PDEs, as it

reduces the space of solutions to just those that satisfy the problem's constraints. TFC has two major steps: (1) Embed the boundary conditions of the problem into the constrained expression; (2) solve the now unconstrained optimization problem. The paragraphs that follow will explain these steps in more detail.

The TFC framework is easiest to understand when explained via an example, like a simple harmonic oscillator. Equation (1) gives an example of a simple harmonic oscillator problem.

$$m\,\frac{\mathrm{d}^2 y}{\mathrm{d}x_1^2} + k\,y = 0 \qquad \text{subject to: } \begin{cases} y(0) = y_0 \\ y_{x_1}(0) = y_{x_1}, \end{cases} \tag{1}$$

where the subscript in y_{x_1} denotes a derivative of y with respect to x_1.

Based on the univariate TFC framework presented in Ref. [8], the constrained expression is represented by the functional,

$$f(x_1, g(x_1)) = g(x_1) + \sum_{j=1}^{2} \eta_j\, s_j(x_1),$$

where the $s_j(x_1)$ are a set of mutually linearly independent functions, called support functions, and η_j are coefficient functions that are computed by imposing the constraints. For this example, let's choose the support functions to be the first two monomials, $s_1(x_1) = 1$ and $s_2(x_1) = x_1$. Hence, the constrained expression becomes,

$$f(x_1, g(x_1)) = g(x_1) + \eta_1 + x_1\,\eta_2. \tag{2}$$

The coefficient functions, $\eta_1(x_1)$ and $\eta_2(x_1)$, are solved by substituting the constraints into the constrained expression and solving the resultant set of equations. For the simple harmonic oscillator this yields the set of equations given by Equations (3) and (4).

$$y(0) = g(0) + \eta_1 \tag{3}$$

$$y_{x_1}(0) = g_{x_1}(0) + \eta_2. \tag{4}$$

Solving Equation (3) results in $\eta_1 = y(0) - g(0)$, and solving Equation (4) yields $\eta_2 = y_{x_1}(0) - g_{x_1}(0)$. Substituting η_1 and η_2 into Equation (3) we obtain,

$$f(x_1, g(x_1)) = g(x_1) + y(0) - g(0) + x_1\Big[y_{x_1}(0) - g_{x_1}(0)\Big]$$

which is an expression satisfying the constraints, no matter what the free function, $g(x_1)$, is. In other words, this equation is able to reduce the solution space to just the space of functions satisfying the constraints, because for any function $g(x_1)$, the boundary conditions will always be satisfied exactly. Therefore, using constrained expressions transforms differential equations into unconstrained optimization problems.

This unconstrained optimization problem could be cast in the following way. Let the function to be minimized, $\mathcal{L}$, be equal to the square of the residual of the differential equation,

$$\mathcal{L}(x_1) = \left[m\frac{\mathrm{d}^2 f(x_1, g(x_1))}{\mathrm{d}x_1^2} + k\,f(x_1)\right]^2.$$

This function is to be minimized by varying the function $g(x_1)$. One way to do this is to choose $g(x_1)$ as a linear combination of a set of basis functions, and calculate the coefficients of the linear combination via least-squares or some other optimization technique. Examples of this methodology using Chebyshev orthogonal polynomials to obtain least-squares solutions of linear and nonlinear ordinary differential equations (ODEs) can be found in Refs. [11,12], respectively.

2.1. n-Dimensional Constrained Expressions

The previous example derived the constrained expression by creating and solving a series of simultaneous algebraic equations. This technique works well for constrained expressions in one dimension; however, it can become needlessly complicated when deriving these expression in n dimensions for constraints on the value and arbitrary order derivative of $n-1$ dimensional manifolds [9]. Fortunately, a different, more mechanized formalism exists that is useful for this case. The constrained expression presented earlier consists of two parts; the first part is a function that satisfies the boundary constraints, and the second part projects the free-function, $g(x_1)$, onto the hyper-surface of functions that are equal to zero at the boundaries. Rearranging Equation (2) highlights these two parts,

$$f(x_1, g(x_1)) = \underbrace{y(0) + x_1 y_{x_1}(0)}_{A(x_1)} + \underbrace{g(x_1) - g(0) - x_1 g_{x_1}(0)}_{B(x_1, g(x_1))},$$

where $A(x_1)$ satisfies the boundary constraints and $B(x_1, g(x_1))$ is a functional projecting the free-function onto the hyper-surface of functions that are equal to zero at the boundaries.

The multivariate extension of this form for problems with boundary and derivative constraints in n-dimensions can be written compactly using Equation (5).

$$\begin{aligned} f(\mathbf{x}, g(\mathbf{x})) = & \underbrace{\mathcal{M}_{i_1,i_2,\cdots,i_n}(c(\mathbf{x}))v_{i_1}(x_1)v_{i_2}(x_2)\cdots v_{i_n}(x_n)}_{A(\mathbf{x})} + \\ & + \underbrace{g(\mathbf{x}) - \mathcal{M}_{i_1,i_2,\cdots,i_n}(g(\mathbf{x}))v_{i_1}(x_1)v_{i_2}(x_2)\cdots v_{i_n}(x_n)}_{B(\mathbf{x}, g(\mathbf{x}))} \end{aligned} \tag{5}$$

where $\mathbf{x} = \{x_1,\ x_2,\ \cdots,\ x_n\}^{\mathrm{T}}$ is a vector of the n independent variables, $\mathcal{M}$ is an n-th order tensor containing the boundary conditions $c(\mathbf{x})$, the $v_{i_1}, \cdots, v_{i_n}$ are vectors whose elements are functions of the independent variables, $g(\mathbf{x})$ is the free-function that can be chosen to be any function that is defined at the constraints, and $f(\mathbf{x}, g(\mathbf{x}))$ is the constrained expression. The first term, $A(\mathbf{x})$, analytically satisfies the boundary conditions, and the term, $B(\mathbf{x}, g(\mathbf{x}))$, projects the free-function, $g(\mathbf{x})$, onto the space of functions that vanish at the constraints. A mathematical proof that this form of the constrained expression satisfies the boundary constraints is given in Ref. [9]. The remainder of this section discusses how to construct the n-th order tensor $\mathcal{M}$ and the $\mathbf{v}$ vectors shown in Equation (5).

Before discussing how to build the $\mathcal{M}$ tensor and $\mathbf{v}$ vectors, let's introduce some mathematical notation. Let $k \in [1, n]$ be an index used to denote the k-th dimension. Let ${}^k c_p^d := \left.\dfrac{\partial^d c(\mathbf{x})}{\partial x_k^d}\right|_{x_k = p}$ be the constraint specified by taking the d-th derivative of the constraint function, $c(\mathbf{x})$, evaluated at the $x_k = p$ hyperplane. Further, let ${}^k c_{\mathbf{p}_k}^{\mathbf{d}_k}$ be the vector of ℓ_k constraints defined at the $x_k = \mathbf{p}_k$ hyperplanes with derivative orders of $\mathbf{d}_k$, where $\mathbf{p}_k$ and $\mathbf{d}_k \in \mathbb{R}^{\ell_k}$. In addition, let's define a boundary condition operator ${}^k b_p^d$ that takes the d-th derivative with respect to x_k of a function, and then evaluates that function at the $x_k = p$ hyperplane. Mathematically,

$${}^k b_p^d[f(\mathbf{x})] = \left.\frac{\partial^d f(\mathbf{x})}{\partial x_k^d}\right|_{x_k = p}.$$

This mathematical notation will be used to introduce a step-by-step method for building the $\mathcal{M}$ tensor. This step-by-step process will be be illustrated via a 3-dimensional example that has Dirichlet boundary conditions in x_1 and initial conditions in x_2 and x_3 on the domain $x_1, x_2, x_3 \in [0,1] \times [0,1] \times [0,1]$. The $\mathcal{M}$ tensor is constructed using the following three rules.

1. The element $\mathcal{M}_{111} = 0$.

2. The first order sub-tensor of $\mathcal{M}$ specified by keeping one dimension's index free and setting all other dimension's indices to 1 consists of the value 0 and the boundary conditions for that dimension. Mathematically,

$$\mathcal{M}_{1,\ldots,1,i_k,1,\ldots,1} = \left\{0, {}^{k}c^{\mathbf{d}_k}_{\mathbf{p}_k}\right\}.$$

Using the example boundary conditions,

$$\begin{aligned}
\mathcal{M}_{i_1 11} &= \left[0, c(0,x_2,x_3), c(1,x_2,x_3)\right]^{\mathrm{T}} \\
\mathcal{M}_{1 i_2 1} &= \left[0, c(x_1,0,x_3), c_{x_2}(x_1,0,x_3)\right]^{\mathrm{T}} \\
\mathcal{M}_{11 i_3} &= \left[0, c(x_1,x_2,0), c_{x_3}(x_1,x_2,0)\right]^{\mathrm{T}}.
\end{aligned} \tag{6}$$

3. The remaining elements of the $\mathcal{M}$ tensor are those with at least two indices different than one. These elements are the geometric intersection of the boundary condition elements of the first order tensors given in Equation (6), plus a sign ($+$ or $-$) that is determined by the number of elements being intersected. Mathematically this can be written as,

$$\mathcal{M}_{i_1 i_2 \ldots i_n} = {}^{1}b^{\mathbf{d}^1_{i_1-1}}_{\mathbf{p}^1_{i_1-1}}\left[{}^{2}b^{\mathbf{d}^2_{i_2-1}}_{\mathbf{p}^2_{i_2-1}}\left[\ldots\left[{}^{n}b^{\mathbf{d}^n_{i_n-1}}_{\mathbf{p}^n_{i_n-1}}[c(\mathbf{x})]\right]\ldots\right]\right](-1)^{m+1},$$

where m is the number of indices different than one. Using the example boundary conditions we give three examples:

$$\begin{aligned}
M_{133} &= -c_{x_2x_3}(x_1,0,0) \\
M_{221} &= -c(0,0,x_3) \\
M_{332} &= c_{x_2}(1,0,0)
\end{aligned}$$

A simple procedure also exists for constructing the v_{i_k} vectors. The v_{i_k} vectors have a standard form:

$$v_{i_k} = \left\{1, \quad \sum_{i=1}^{\ell_k}\alpha_{i1}\,h_i(x_k), \quad \sum_{i=1}^{\ell_k}\alpha_{i2}\,h_i(x_k), \quad \ldots, \quad \sum_{i=1}^{\ell_k}\alpha_{i\ell_k}\,h_i(x_k)\right\}^{\mathrm{T}},$$

where $h_i(x_k)$ are ℓ_k linearly independent functions. The simplest set of linearly independent functions, and those most often used in the TFC constrained expressions, are monomials, $h_i(x_k) = x_k^{i-1}$. The $\ell_k \times \ell_k$ coefficients, α_{ij}, can be computed by matrix inversion,

$$\begin{bmatrix}
{}^{k}b^{d_1}_{p_1}[h_1] & {}^{k}b^{d_1}_{p_1}[h_2] & \ldots & {}^{k}b^{d_1}_{p_1}[h_{\ell_k}] \\
{}^{k}b^{d_2}_{p_2}[h_1] & {}^{k}b^{d_2}_{p_2}[h_2] & \ldots & {}^{k}b^{d_2}_{p_2}[h_{\ell_k}] \\
\vdots & \vdots & \ddots & \vdots \\
{}^{k}b^{d_{\ell_k}}_{p_{\ell_k}}[h_1] & {}^{k}b^{d_{\ell_k}}_{p_{\ell_k}}[h_2] & \ldots & {}^{k}b^{d_{\ell_k}}_{p_{\ell_k}}[h_{\ell_k}]
\end{bmatrix}
\begin{bmatrix}
\alpha_{11} & \alpha_{12} & \ldots & \alpha_{1\ell_k} \\
\alpha_{21} & \alpha_{22} & \ldots & \alpha_{2\ell_k} \\
\vdots & \vdots & \ddots & \vdots \\
\alpha_{\ell_k 1} & \alpha_{\ell_k 2} & \ldots & \alpha_{\ell_k\ell_k}
\end{bmatrix}
=
\begin{bmatrix}
1 & 0 & \ldots & 0 \\
0 & 1 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 1
\end{bmatrix}.$$

Using the example boundary conditions, let's derive the v_{i_1} vector using the linearly independent functions $h_1 = 1$ and $h_2 = x_1$.

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \rightarrow \qquad \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

$$v_{i_1} = \left\{1, \quad 1 - x_1, \quad x_1\right\}^{\mathrm{T}}.$$

For more examples and a mathematical proof that these procedures for generating the $\mathcal{M}$ tensor and the $\mathbf{v}$ vectors form a valid constrained expression see Ref. [9].

2.2. Two-Dimensional Example

This subsection will give an in depth example for a two-dimensional TFC case. The example is originally from problem 5 of Ref. [5], and is one of the PDE problems analyzed in this article. The problem is shown in Equation (7).

$$\begin{aligned} &\nabla^2 z(x,y) = e^{-x}(x-2+y^3+6y) \quad \text{subject to:} \\ &\begin{cases} z(x,0) = c(x,0) = xe^{-x} \\ z(0,y) = c(0,y) = y^3 \\ z(x,1) = c(x,1) = e^{-x}(x+1) \\ z(1,y) = c(1,y) = (1+y^3)e^{-1} \end{cases} \\ &\text{where} \quad (x,y) \in [0,1] \times [0,1] \end{aligned} \tag{7}$$

Following the step-by-step procedure given in the previous section we will construct the $\mathcal{M}$ tensor:

1. The first element is $\mathcal{M}_{11} = 0$.
2. The first order sub-tensors of $\mathcal{M}$ are:

$$\mathcal{M}_{i_1 1} = \begin{Bmatrix} 0 & c(0,y) & c(1,y) \end{Bmatrix}$$
$$\mathcal{M}_{1 i_2} = \begin{Bmatrix} 0 & c(x,0) & c(x,1) \end{Bmatrix}$$

3. The remaining elements of $\mathcal{M}$ are the geometric intersection of elements from the first order sub-tensors.

$$\mathcal{M}_{22} = -c(0,0) \qquad \mathcal{M}_{23} = -c(1,0)$$
$$\mathcal{M}_{32} = -c(0,1) \qquad \mathcal{M}_{33} = -c(1,1)$$

Hence, the $\mathcal{M}$ tensor is,

$$\mathcal{M}_{i_1 i_2} = \begin{bmatrix} 0 & c(0,y) & c(1,y) \\ c(x,0) & -c(0,0) & -c(1,0) \\ c(x,1) & -c(0,1) & -c(1,1) \end{bmatrix} = \begin{bmatrix} 0 & y^3 & (1+y^3)e^{-1} \\ xe^{-x} & 0 & -e^{-1} \\ e^{-x}(x+1) & -1 & -2e^{-1} \end{bmatrix}$$

Following the step-by-step procedure given in the previous section we will construct the $\mathbf{v}$ vectors. For v_{i_1}, let's choose the linearly independent functions $h_1 = 1$ and $h_2 = x$.

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

$$v_{i_1} = \begin{Bmatrix} 1, & 1-x, & x \end{Bmatrix}^{\mathrm{T}}.$$

For v_{i_2} let's choose the linearly indpendent functions $h_1 = 1$ and $h_2 = y$.

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

$$v_{i_2} = \begin{Bmatrix} 1, & 1-y, & y \end{Bmatrix}^{\mathrm{T}}.$$

Now, we use the constrained expression form given in Equation (5) to finish building the constrained expression.

$$f(x,y,g(x,y)) = g(x,y) + \frac{xy(y^2-1)}{e} + e^{-x}(x+y) + \\ + (1-x)\Big(g(0,0) + y\Big(g(0,1) + y^2 - g(0,0) - 1\Big)\Big) + (x-1)g(0,y) + \\ + x\Big(yg(1,1) + (1-y)g(1,0)\Big) - xg(1,y) + (y-1)g(x,0) - yg(x,1) \tag{8}$$

Notice, that Equation (8) will always satisfy the boundary conditions of the problem regardless of the value of $g(x,y)$. Thus, the problem has been transformed into an unconstrained optimization problem where the cost function, $\mathcal{L}$, is the square of the residual of the PDE,

$$\mathcal{L}(x,y,g(x,y)) = \Big(\nabla^2 f(x,y,g(x,y)) - e^{-x}(x-2+y^3+6y)\Big)^2.$$

For ODEs, the minimization of the cost function was accomplished by choosing g to be a linear combination of orthogonal polynomials with unknown coefficients, and performing least-squares or some other optimization technique to find the unknown coefficients. For two dimensions, one could make $g(x,y)$ the product of two linear combinations of these orthogonal polynomials, calculate all of the cross-terms, and then solve for the coefficients that multiply all terms and cross-terms using least-squares or non-linear least-squares. However, this will become computationally prohibitive as the dimension increases. Even at two dimensions, the number of basis functions needed, and thus the size of the matrix to invert in the least-squares, becomes large. An alternative solution, and the one explored in this article, is to make the free function, $g(x,y)$, a neural network.

3. PDE Solution Methodology

Similar to the last section, the easiest way to describe the methodology is with an example. The example used throughout this section will be the PDE given in Equation (7).

As mentioned previously, Deep TFC approximates solutions to PDEs by finding the constrained expression for the PDE and choosing a neural network as the free function. For all of the problems analyzed in this article, a simple, fully connected neural network was used. Each layer of a fully connected neural network consists of non-linear activation functions composed with affine transformations of the form $\mathcal{A} = W \cdot \mathbf{x} + \mathbf{b}$, where W is a matrix of the neuron weights, $\mathbf{b}$ is a vector of the neuron biases, and $\mathbf{x}$ is a vector of inputs from the previous layer (or the inputs to the neural network if it is the first layer). Then, each layer is composed to form the entire network. For the fully connected neural networks used in this paper, the last layer is simply a linear output layer. For example, a neural network with three hidden layers that each use the non-linear activation function ϕ can be written mathematically as,

$$\mathcal{N}(\mathbf{x};\theta) = W_4 \cdot \phi\Big(W_3 \cdot \phi\Big(W_2 \cdot \phi\big(W_1 \cdot \mathbf{x} + \mathbf{b}_1\big) + \mathbf{b}_2\Big) + \mathbf{b}_3\Big) + \mathbf{b}_4,$$

where $\mathcal{N}$ is the symbol used for the neural network, $\mathbf{x}$ is the vector of inputs, W_k are the weight matrices, $\mathbf{b}_k$ are the bias vectors, and θ is a symbol that represents all trainable parameters of the neural network; the weights and biases of each layer constitute the trainable parameters. Note that the notation $\mathcal{N}(x,y,\ldots;\theta)$ is also used in this paper for independent variables $x, y, \ldots$ and trainable parameters θ.

Thus, the constrained expression, given originally in Equation (8), now has the form given in Equation (9).

$$\begin{aligned} f(x,y;\theta) = {} & \mathcal{N}(x,y;\theta) + \frac{xy(y^2-1)}{e} + e^{-x}(x+y) + \\ & + (1-x)\Big(\mathcal{N}(0,0;\theta) + y\Big(\mathcal{N}(0,1;\theta) + y^2 - \mathcal{N}(0,0;\theta) - 1\Big)\Big) + (x-1)\mathcal{N}(0,y;\theta) + \\ & + x\Big(y\mathcal{N}(1,1;\theta) + (1-y)\mathcal{N}(1,0;\theta)\Big) - x\mathcal{N}(1,y;\theta) + (y-1)\mathcal{N}(x,0;\theta) - y\mathcal{N}(x,1;\theta) \end{aligned} \tag{9}$$

In order to estimate the solution to the PDE, the parameters of the neural network have to be optimized to minimize the loss function, which is taken to be the square of the residual of the PDE. For this example,

$$\mathcal{L} = \sum_i^N \mathcal{L}_i(x_i, y_i; \theta) \qquad \text{where} \qquad \mathcal{L}_i(x_i, y_i; \theta) = \Big(\nabla^2 f(x_i, y_i; \theta) - e^{-x_i}(x_i - 2 + y_i^3 + 6y_i)\Big)^2.$$

The attentive reader will notice that training the neural network will require, for this example, taking two second order partial derivatives of $f(x,y;\theta)$ to calculate $\mathcal{L}_i$, and then taking gradients of $\mathcal{L}$ with respect to the neural network parameters, θ, in order to train the neural network.

To take these higher order derivatives, TensorFlow's™ gradients function was used [15]. This function uses automatic differentiation [16] to compute these derivatives. However, one must be conscientious when using the gradients function to ensure they get the desired gradients.

When taking the gradient of a vector, y_j, with respect to another vector, x_i, TensorFlow™computes,

$$z_i = \frac{\partial}{\partial x_i}\Big(\sum_{j=1}^N y_j\Big)$$

where z_i is a vector of the same size as x_i. The only example where it is not immediately obvious that this gradient function will give the desired gradient is when computing $\nabla^2 f_i$. The desired output of this calculation is the following vector,

$$z_i = \left\{ \frac{\partial^2 f_1}{\partial x_1^2} + \frac{\partial^2 f_1}{\partial y_1^2}, \quad \cdots, \quad \frac{\partial^2 f_N}{\partial x_N^2} + \frac{\partial^2 f_N}{\partial y_N^2} \right\}^{\mathrm{T}},$$

where z_i has the same size as f_i and (x_i, y_i) is the point used to generate f_i. TensorFlow's™ gradients function will compute the following vector,

$$\tilde{z}_i = \left\{ \frac{\partial^2 \left(\sum_{j=1}^N f_j\right)}{\partial x_1^2} + \frac{\partial^2 \left(\sum_{j=1}^N f_j\right)}{\partial y_1^2}, \quad \cdots, \quad \frac{\partial^2 \left(\sum_{j=1}^N f_j\right)}{\partial x_N^2} + \frac{\partial^2 \left(\sum_{j=1}^N f_j\right)}{\partial y_N^2} \right\}^{\mathrm{T}}.$$

However, because f_i only depends on the point (x_i, y_i) and the derivative operator commutes with the sum operator, TensorFlow's™ gradients function will compute the desired vector (i.e., $\tilde{z}_i = z_i$). Moreover, the size of the output vector will be correct, because the input vectors, x_i and y_i, have the same size as f_i.

Training the Neural Network

Three methods were tested when optimizing the parameters of the neural networks:

1. Adam optimizer [17]: A variant of stochastic gradient descent (SGD) that combines the advantages of two other popular SGD variants: AdaGrad [18] and RMSProp [19].

2. Broyden–Fletcher–Goldfarb–Shanno [20] (BFGS): A quasi-Newton method designed for solving unconstrained, non-linear optimization problems. This method was chosen based on its performance when optimizing neural network parameters to estimate PDE solutions in Ref. [5].
3. Hybrid method: Combines the first two methods by applying them in series.

For all four problems shown in this article, the solution error when using the BFGS optimizer was lower than with the other two methods. Thus, in the following section, the results shown use the BFGS optimizer.

The BFGS optimizer is a local optimizer, and the weights and biases of the neural networks are initialized randomly. Therefore, the solution error when numerically estimating PDEs will be different each time. However, Deep TFC guarantees that the boundary conditions are satisfied, and the loss function is the square of the residual of the PDE. Therefore, the loss function indicates how well Deep TFC is estimating the solution of the PDE at the training points. Moreover, because Deep TFC produces an analytical approximation of the solution, the loss function can be calculated at any point. Therefore, after training, one can calculate the loss function at a set of test points to determine whether the approximate solution generalizes well or has over fit the training points.

Due to the inherit stochasticity of the method, each Deep TFC solution presented in the results section that follows is the solution with the lowest mean absolute error of 10 trials. In other words, for each problem, the Deep TFC methodology was performed 10 times, and the best solution of those 10 trials is presented. Moreover, to show the variability in the Deep TFC method, problem 1 contains a histogram of the maximum solution error on a test set for 100 Monte Carlo trials.

4. Results

This section compares the estimated solution found using Deep TFC with with the analytical solution. Four PDE problems are analyzed. The first is the example PDE given in Equation (7), and the second is the wave equation. The third and fourth PDEs are simple solutions to the incompressible Navier–Stokes equations.

4.1. Problem 1

The first problem analyzed was the PDE given by Equation (7), copied below for the reader's convenience.

$$\nabla^2 z(x,y) = e^{-x}(x-2+y^3+6y) \quad \text{subject to:}$$
$$\begin{cases} z(x,0) = xe^{-x} \\ z(0,y) = y^3 \\ z(x,1) = e^{-x}(x+1) \\ z(1,y) = (1+y^3)e^{-1} \end{cases}$$
$$\text{where} \quad (x,y) \in [0,1] \times [0,1]$$

The known analytical solution for this problem is,

$$z = e^{-x}(x+y^3).$$

The neural network used to estimate the solution to this PDE was a fully connected neural network with 6 hidden layers, 15 neurons per layer, and 1 linear output layer. The non-linear activation function used in the hidden layers was the hyperbolic tangent. Other fully connected neural networks with various sizes and non-linear activation functions were tested, but this combination of size and activation function performed the best in terms of solution error. The biases of the neural network were all initialized as zero, and the weights were initialized using TensorFlow's^{TM} implementation of the Xavier initialization with uniform random initialization [21]. One hundred training points, (x,y), evenly distributed throughout the domain were used to train the neural network.

Figure 1 shows the difference between the analytical solution and the estimated solution using Deep TFC on a grid of 10,000 evenly distributed points. This grid represents the test set. Figure 2 shows a histogram of the maximum solution error on the test set for 100 Monte Carlo trials.

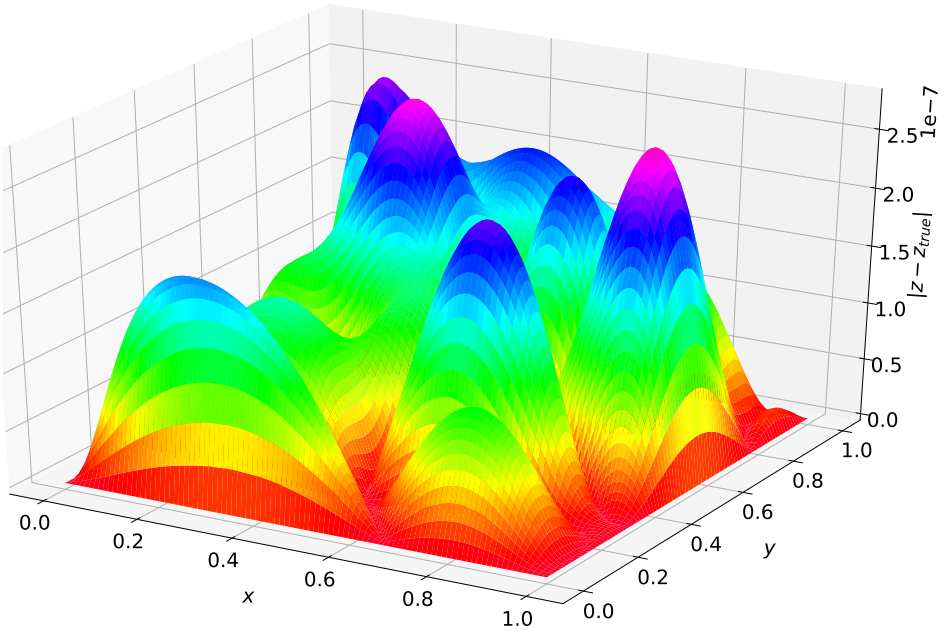

Figure 1. Problem 1 solution error.

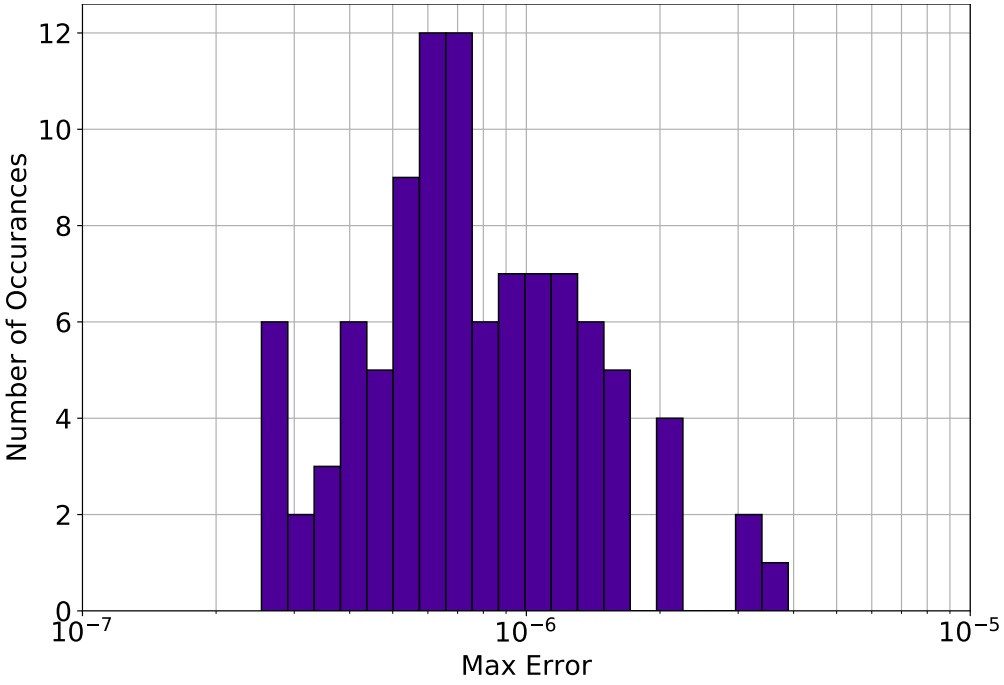

Figure 2. Problem 1 maximum test set solution error from 100 Monte Carlo trials.

The maximum error on the test set shown in Figure 1 was 2.780×10^{-7} and the average error was 8.517×10^{-8}. Figure 2 shows that Deep TFC produces a solution at least as accurate as the solution in Figure 1 approximately 10% of the time. The remaining 90% of the time the solution error will be larger. Moreover, Figure 2 shows that the Deep TFC method is consistent. The maximum solution error in the 100 Monte Carlo tests was 3.891×10^{-6}, approximately an order of magnitude larger than the maximum solution error shown in Figure 1. The maximum error from Figure 1 is relatively

low, six orders of magnitude lower than the solution values, which are on the order of 10^{-1}. Table 1 compares the maximum error on the test and training sets obtained with Deep TFC with the method used in Ref. [5] and FEM. Note, the FEM data was obtained from Table 1 of Ref. [5].

Table 1. Comparison of Deep TFC, Ref. [5], and finite element method (FEM).

Method	Training Set	Test Set
Deep TFC	3×10^{-7}	3×10^{-7}
Ref. [5]	5×10^{-7}	5×10^{-7}
FEM	2×10^{-8}	1.5×10^{-5}

Table 1 shows that Deep TFC is slightly more accurate than the method from Ref. [5]. Moreover, in consonance with the findings from Ref. [5], the FEM solution performs better on the training set than Deep TFC, but worse on the solution set. Note also "that the accuracy of the finite element method decreases as the grid becomes coarser, and that the neural approach considers a mesh of 10×10 points while in the finite element case a 18×18 mesh was employed" [5].

The neural network used in this article is more complicated than the network used in Ref. [5], even though the two solution methods produce similarly accurate solutions. The constrained expression, $f(x, y; \theta)$, created using TFC, which is used as the assumed solution form, is more complex both in the number of terms and the number of times the neural network appears than the assumed solution form in Ref. [5]. For the reader's reference, the assumed solution form for problem 1 from Ref. [5] is shown in Equation (10). Equation (10) was copied from Ref. [5], but the notation used has been transformed to match that of this paper; furthermore, a typo in the assumed solution form from Ref. [5] has been corrected here.

$$\begin{aligned} f(x, y; \theta) = x(1-x)y(1-y)\mathcal{N}(x, y; \theta) + (1-x)y^3 + x(1+y^3)e^{-1} \\ + (1-y)x(e^{-x} - e^{-1}) + y((1+x)e^{-x} - (1-x+2e^{-1})) \end{aligned} \tag{10}$$

To investigate how the assumed solution form affects the accuracy of the estimated solution, a comparison was made between the solution form from Ref. [5] and the solution form created using TFC in this article, while keeping all other variables constant. Furthermore, in this comparison, the neural network architecture used is identical to the neural network architecture given in Ref. [5] for this problem: one hidden layer with 10 neurons that uses a sigmoid non-linear activation function and a linear output layer. Each network was trained using the BFGS optimizer. The training points used were 100 evenly distributed points throughout the domain.

Figure 3 was created using the solution form posed in [5]. The maximum error on the test set was $4.246 \times 10^{-7}$and the average error on the test set was 1.133×10^{-7}. Figure 4 was created using the Deep TFC solution form. The maximum error on the test set was 8.790×10^{-6} and the average error on the test set was 2.797×10^{-6}.

Comparing Figures 3 and 4 shows that the solution form from [5] gives an estimated solution that is approximately an order of magnitude lower in terms of average error and maximum error for this problem. Hence, the more complex TFC solution form requires a more complex neural network to achieve the same accuracy as the simpler solution form from Ref. [5] with a simple neural network. This results in a trade-off. The TFC constrained expressions allow for more complex boundary conditions (i.e., derivatives of arbitrary order) and can be used on n-dimensional domains, but require a more complex neural network. In contrast, the simpler solution form from Ref. [5] can achieve the same level of accuracy with a simpler neural network, but cannot be used for problems with higher order derivatives or n-dimensional domains.

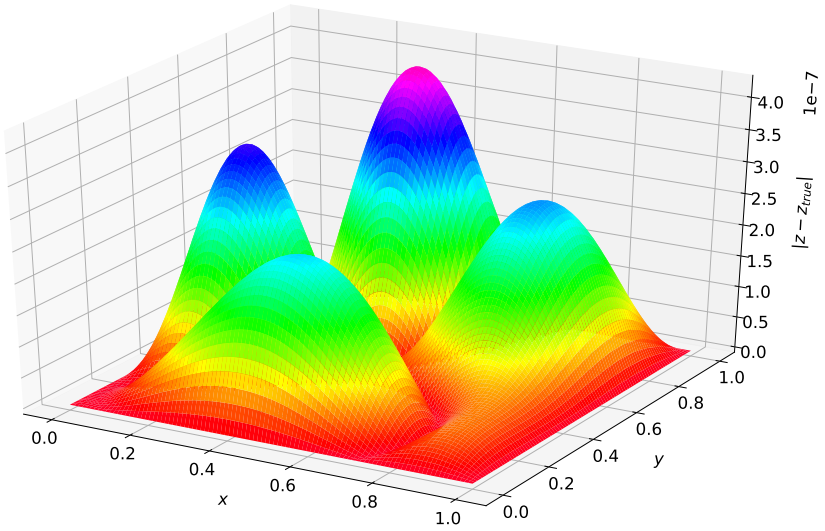

Figure 3. Problem 1 solution error using Ref. [5] solution form.

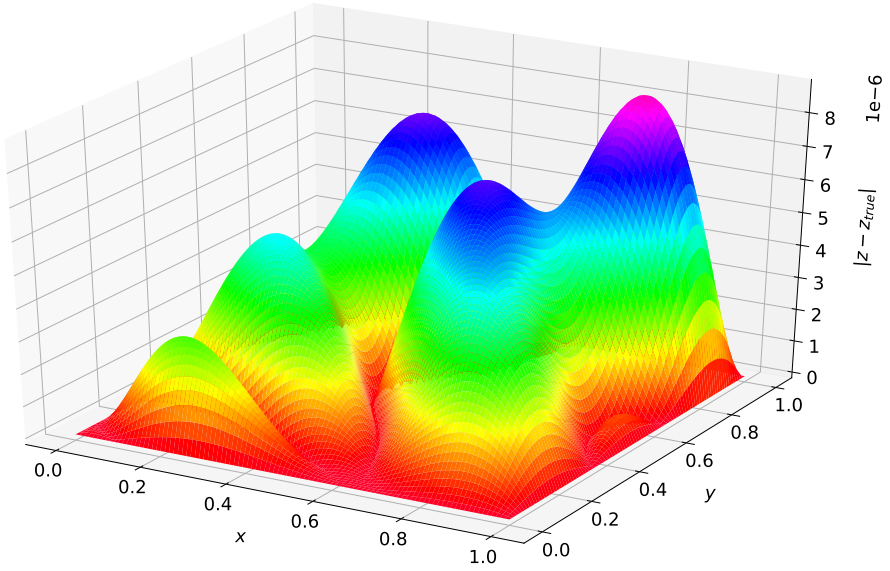

Figure 4. Problem 1 solution error using Deep TFC solution form.

4.2. Problem 2

The second problem analyzed was the wave equation, shown in Equation (11).

$$\frac{\partial^2 u}{\partial t^2}(x,t) = c^2 \frac{\partial^2 u}{\partial x^2}(x,t) \quad \text{subject to:}$$
$$\begin{cases} z(0,t) = 0 \\ z(1,t) = 0 \\ z(x,0) = x(1-x) \\ z_t(x,0) = 0 \end{cases} \tag{11}$$
$$\text{where} \quad (x,t) \in [0,1] \times [0,1]$$

where the constant, $c = 1$. The analytical solution for this problem is,

$$z(x,t) = \sum_{k=0}^{\infty} \frac{8}{(2k+1)^3\pi^3} \sin\Big((2k+1)\pi x\Big) \cos\Big((2k+1)c\pi t\Big).$$

Although the true analytical solution is an infinite series, for the purposes of making numerical comparisons, one can simply truncate this infinite series such that the error incurred by truncation falls below machine level precision. The constrained expression for this problem is shown in Equation (12).

$$\begin{aligned} f(x,t;\theta) = (1-x)\Big[\mathcal{N}(0,0;\theta) - \mathcal{N}(0,t;\theta)\Big] + x\Big[\mathcal{N}(1,0;\theta) - \mathcal{N}(1,t;\theta)\Big] - \mathcal{N}(x,0;\theta) \\ + x(1-x) + \mathcal{N}(x,t;\theta) + t\Big[(1-x)\mathcal{N}_t(0,0;\theta) + x\mathcal{N}_t(1,0;\theta) - \mathcal{N}_t(x,0;\theta)\Big] \end{aligned} \tag{12}$$

The neural network used to estimate the solution to this PDE was a fully connected neural network with three hidden layers and 30 neurons per layer. The non-linear activation function used was the hyperbolic tangent. The biases and weights were initialized using the same method as problem 1. The training points, (x,t), were created by choosing x to be an independent and identically distributed (IID) random variable with uniform distribution in the range $[0,1]$, and t to be an IID random variable with uniform distribution in the range $[0,1]$. The network was trained using the BFGS method and 1000 training points.

Figure 5 shows the difference between the analytical solution and the estimated solution using Deep TFC on a grid of 10,000 evenly distributed points; this grid represents the test set.

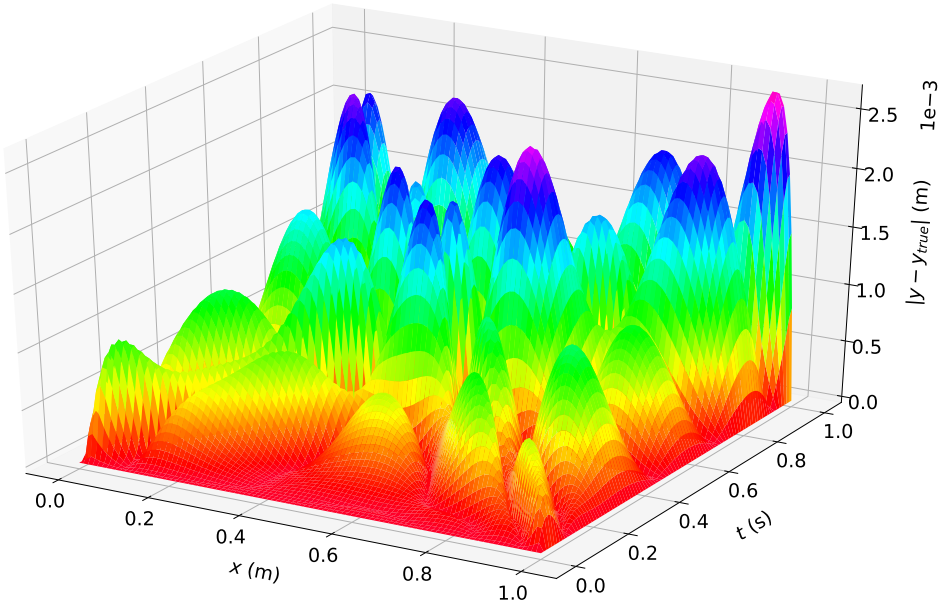

Figure 5. Problem 2 solution error.

The maximum error on the test set was 2.643×10^{-3} m and the average error on the test set was 6.305×10^{-4} m. The error of this solution is larger than in the problem 1, while the solution values are on the same order of magnitude, 10^{-1} m, as in problem 1. The larger relative error in problem 2 is due to the more oscillatory nature of the solution (i.e., the surface of the true solution in problem 2 is more complex than that of problem 1).

4.3. Problem 3

The third problem analyzed was a known solution to the incompressible Navier–Stokes equations, called Poiseuille flow. The problem solves the flow velocity in a two-dimensional pipe in steady-state

with a constant pressure gradient applied in the longitudinal axis. Equation (13) shows the associated equations and boundary conditions.

$$\begin{aligned}
&\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0\\
&\rho\left(\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}\right)=-\frac{\partial P}{\partial x}+\mu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)\\
&\rho\left(\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}\right)=\mu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)\\
&\text{subject to:}\\
&\begin{cases}
u(0,y,t)=u(L,y,t)=u(x,y,0)=\frac{1}{2\mu}\frac{\partial P}{\partial x}\left(y^2-\left(\frac{H}{2}\right)^2\right)\\
u(x,\frac{H}{2},t)=u(x,-\frac{H}{2},t)=0\\
v(0,y,t)=v(L,y,t)=v(x,y,0)=0\\
v(0,\frac{H}{2},t)=v(0,-\frac{H}{2},t)=0
\end{cases}
\end{aligned} \tag{13}$$

where u and v are velocities in the x and y directions respectively, H is the height of the channel, P is the pressure, ρ is the density, and μ is the viscosity. For this problem, the values $H=1$ m, $\rho=1$ kg/m^3, $\mu=1$ Pa·s, and $\frac{\partial P}{\partial x}=-5$ N/m^3 were chosen. The constrained expressions for the u-velocity, $f^u(x,y,t;\theta)$, and v-velocity, $f^v(x,y,t;\theta)$, are shown in Equation (14).

$$\begin{aligned}
f^u(x,y,t;\theta)&=\mathcal{N}(x,y,t;\theta)-\mathcal{N}(x,y,0;\theta)+\frac{L-x}{L}\Big(\mathcal{N}(0,y,0;\theta)-\mathcal{N}(0,y,t;\theta)\Big)\\
&+\frac{x}{L}\Big(\mathcal{N}(L,y,0;\theta)-\mathcal{N}(L,y,t;\theta)\Big)+\frac{P\left(4y^2-H^2\right)}{8\mu}\\
&+\frac{1}{2HL}\Bigg((2y-H)\Big((L-x)\mathcal{N}\Big(0,-\frac{H}{2},0;\theta\Big)+x\mathcal{N}\Big(L,-\frac{H}{2},0;\theta\Big)-L\mathcal{N}\Big(x,-\frac{H}{2},0;\theta\Big)\\
&-(L-x)\mathcal{N}\Big(0,-\frac{H}{2},t;\theta\Big)+L\mathcal{N}\Big(x,-\frac{H}{2},t;\theta\Big)-x\mathcal{N}\Big(L,-\frac{H}{2},t;\theta\Big)\Big)\\
&-(H+2y)\Big((L-x)\mathcal{N}\Big(0,\frac{H}{2},0;\theta\Big)-L\mathcal{N}\Big(x,\frac{H}{2},0;\theta\Big)+x\mathcal{N}\Big(L,\frac{H}{2},0;\theta\Big)\\
&-(L-x)\mathcal{N}\Big(0,\frac{H}{2},t;\theta\Big)-x\mathcal{N}\Big(L,\frac{H}{2},t;\theta\Big)+L\mathcal{N}\Big(x,\frac{H}{2},t;\theta\Big)\Big)\Bigg)\\
f^v(x,y,t;\theta)&=\mathcal{N}(x,y,t;\theta)-\mathcal{N}(x,y,0;\theta)+\frac{L-x}{L}\Big(\mathcal{N}(0,y,0;\theta)-\mathcal{N}(0,y,t;\theta)\Big)\\
&+\frac{x}{L}\Big(\mathcal{N}(L,y,0;\theta)-\mathcal{N}(L,y,t;\theta)\Big)+\frac{1}{2HL}\Bigg((2y-H)\Big((L-x)\mathcal{N}\Big(0,-\frac{H}{2},0;\theta\Big)\\
&+x\mathcal{N}\Big(L,-\frac{H}{2},0;\theta\Big)-L\mathcal{N}\Big(x,-\frac{H}{2},0;\theta\Big)-(L-x)\mathcal{N}\Big(0,-\frac{H}{2},t;\theta\Big)+L\mathcal{N}\Big(x,-\frac{H}{2},t;\theta\Big)\\
&-x\mathcal{N}\Big(L,-\frac{H}{2},t;\theta\Big)\Big)-(H+2y)\Big((L-x)\mathcal{N}\Big(0,\frac{H}{2},0;\theta\Big)-L\mathcal{N}\Big(x,\frac{H}{2},0;\theta\Big)\\
&+x\mathcal{N}\Big(L,\frac{H}{2},0;\theta\Big)-(L-x)\mathcal{N}\Big(0,\frac{H}{2},t;\theta\Big)-x\mathcal{N}\Big(L,\frac{H}{2},t;\theta\Big)+L\mathcal{N}\Big(x,\frac{H}{2},t;\theta\Big)\Big)\Bigg)
\end{aligned} \tag{14}$$

The neural network used to estimate the solution to this PDE was a fully connected neural network with four hidden layers and 30 neurons per layer. The non-linear activation function used was the sigmoid. The biases and weights were initialized using the same method as problem 1. The training points, (x,y,t), were created by sampling x, y, and t IID from a uniform distribution that spanned the range of the associated independent variable. For x, the range was $[0,1]$. For y, the range was $[-\frac{H}{2},\frac{H}{2}]$, and for t, the range was $[0,1]$. The network was trained using the BFGS method on a batch

size of 1000 training points. The loss function used was the sum of the squares of the residuals of the three PDEs in Equation (13).

The maximum error in the u-velocity was 3.308×10^{-7} m per second, the average error in the u-velocity was 9.998×10^{-8} m per second, the maximum error in the v-velocity was 5.575×10^{-7} m per second, and the average error in the v-velocity was 1.542×10^{-7} m per second. Despite the complexity, the maximum error and average error for this problem are six to seven orders of magnitude lower than the solution values. However, the constrained expression for this problem essentially encodes the solution, because the initial flow condition at time zero is the same as the flow condition throughout the spatial domain at any time. Thus, if the neural network outputs a value of zero for all inputs, the problem will be solved exactly. Although the neural network does output a very small value for all inputs, it is interesting to note that none of the layers have weights or biases that are at or near zero.

4.4. Problem 4

The fourth problem is another solution to the Navier–Stokes equations, and is very similar to the third. The only difference is that in this case, the fluid is not in steady state, it starts from rest. Equation (15) shows the associated equations and boundary conditions.

$$\begin{aligned}
&\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \\
&\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = -\frac{\partial P}{\partial x} + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\
&\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) = \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \\
&\text{subject to:} \\
&\begin{cases}
u(0,y,t) = \frac{\partial u}{\partial x}(L,y,t) = u(x,y,0) = 0 \\
u(x,\frac{H}{2},t) = u(x,-\frac{H}{2},t) = 0 \\
v(0,y,t) = \frac{\partial v}{\partial x}(L,y,t) = v(x,y,0) = 0 \\
v(x,\frac{H}{2},t) = v(x,-\frac{H}{2},t) = 0
\end{cases}
\end{aligned} \tag{15}$$

This problem was created to avoid encoding the solution to the problem into the constrained expression, as was the case in the previous problem. The constrained expressions for the u-velocity, $f^u(x,y,t;\theta)$, and v-velocity, $f^v(x,y,t;\theta)$, are shown in Equation (16).

$$\begin{aligned}
f^u(x,y,t;\theta) &= \mathcal{N}(x,y,t;\theta) - \mathcal{N}(x,y,0;\theta) + \mathcal{N}(0,y,0;\theta) - \mathcal{N}(0,y,t;\theta) + x\mathcal{N}_x(L,y,0;\theta) - x\mathcal{N}_x(L,y,t;\theta) \\
&+ \frac{1}{2H}\left((2y-H)\left(\mathcal{N}\left(0,-\frac{H}{2},0;\theta\right) - \mathcal{N}\left(x,-\frac{H}{2},0;\theta\right) + x\mathcal{N}_x\left(L,-\frac{H}{2},0;\theta\right) - \mathcal{N}\left(0,-\frac{H}{2},t;\theta\right)\right.\right. \\
&+ \mathcal{N}\left(x,-\frac{H}{2},t;\theta\right) - x\mathcal{N}_x\left(L,-\frac{H}{2},t;\theta\right)\Big) - (H+2y)\left(\mathcal{N}\left(0,\frac{H}{2},0;\theta\right) - \mathcal{N}\left(x,\frac{H}{2},0;\theta\right)\right. \\
&\left.\left. + x\mathcal{N}_x\left(L,\frac{H}{2},0;\theta\right) - \mathcal{N}\left(0,\frac{H}{2},t;\theta\right) + \mathcal{N}\left(x,\frac{H}{2},t;\theta\right) - x\mathcal{N}_x\left(L,\frac{H}{2},t;\theta\right)\right)\right) \\
f^v(x,y,t;\theta) &= \mathcal{N}(x,y,t;\theta) - \mathcal{N}(x,y,0;\theta) + \mathcal{N}(0,y,0;\theta) - \mathcal{N}(0,y,t;\theta) + x\mathcal{N}_x(L,y,0;\theta) - x\mathcal{N}_x(L,y,t;\theta) \\
&+ \frac{1}{2H}\left((2y-H)\left(\mathcal{N}\left(0,-\frac{H}{2},0;\theta\right) - \mathcal{N}\left(x,-\frac{H}{2},0;\theta\right) + x\mathcal{N}_x\left(L,-\frac{H}{2},0;\theta\right) - \mathcal{N}\left(0,-\frac{H}{2},t;\theta\right)\right.\right. \\
&+ \mathcal{N}\left(x,-\frac{H}{2},t;\theta\right) - x\mathcal{N}_x\left(L,-\frac{H}{2},t;\theta\right)\Big) - (H+2y)\left(\mathcal{N}\left(0,\frac{H}{2},0;\theta\right) - \mathcal{N}\left(x,\frac{H}{2},0;\theta\right)\right. \\
&\left.\left. + x\mathcal{N}_x\left(L,\frac{H}{2},0;\theta\right) - \mathcal{N}\left(0,\frac{H}{2},t;\theta\right) + \mathcal{N}\left(x,\frac{H}{2},t;\theta\right) - x\mathcal{N}_x\left(L,\frac{H}{2},t;\theta\right)\right)\right)
\end{aligned} \tag{16}$$

The neural network used to estimate the solution to this PDE was a fully connected neural network with four hidden layers and 30 neurons per layer. The non-linear activation function used was

the hyperbolic tangent. The biases and weights were initialized using the same method as problem 1. Problem 4 used 2000 training points that were selected the same way as in problem 3, except the new ranges for the independent variables were $[0, 15]$ for x, $[0, 3]$ for t, and $[-\frac{H}{2}, \frac{H}{2}]$ for y.

Figures 6–8 show the u-velocity of the fluid throughout the domain at three different times. Qualitatively, the solution should look as follows. The solution should be symmetric about the line $y = 0$, and the solution should develop spatially and temporally such that after a sufficient amount of time has passed and sufficiently far from the inlet, $x = 0$, the u-velocity will be equal, or very nearly equal, to the steady state u-velocity of problem 3. Qualitatively, the u-velocity field looks correct in Figures 7 and 8, and throughout most of the spatial domain in Figure 6. However, near the left end of Figure 6, the shape of the highest velocity contour does not match that of the other figures. This stems from the fact that none of the training points fell near this location. Other numerical estimations of this PDE were made with the exact same method, but with different sets of random training points, and in those that had training points near this location, the u-velocity matched the qualitative expectation. However, none of those estimated solutions had a quantitative u-velocity with an error as low as the one shown in Figures 6–8. Quantitatively, the u-velocity at $x = 15$ from Figure 8 was compared with the known steady state u-velocity, and had a maximum error of 5.378×10^{-4} m per second and an average error of 3.117×10^{-4} m per second.

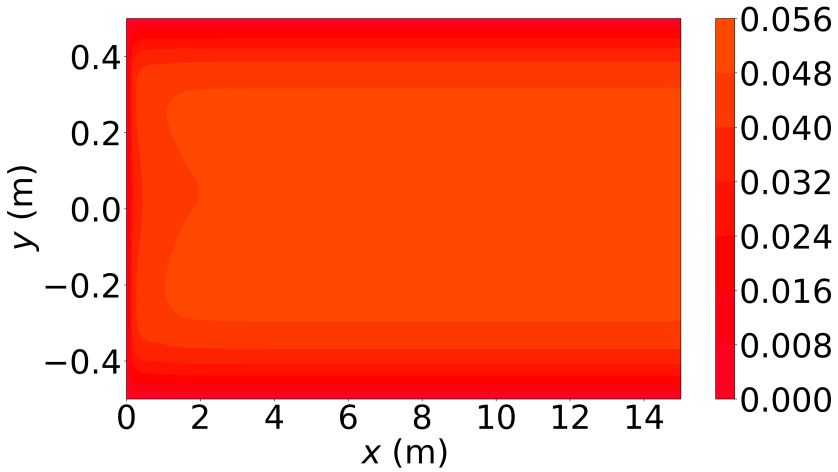

Figure 6. u-velocity in meters per second at 0.01 s.

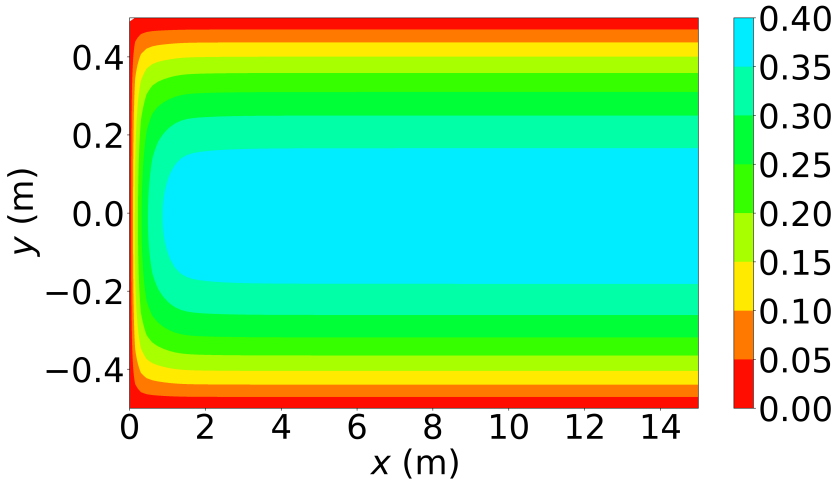

Figure 7. u-velocity in meters per second at 0.1 s.

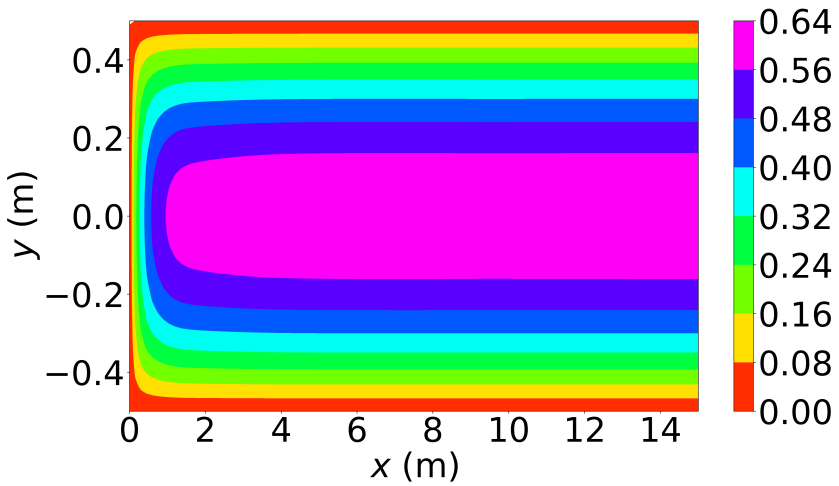

Figure 8. u-velocity in meters per second at 3.0 s.

5. Conclusions

This article demonstrated how to combine neural networks with the Theory of Functional Connections (TFC) into a new methodology, called Deep TFC, that was used to estimate the solutions of PDEs. Results on this methodology applied to four problems were presented that display how accurately relatively simple neural networks can approximate the solutions to some well known PDEs. The difficulty of the PDEs in these problems ranged from linear, two-dimensional PDEs to coupled, non-linear, three-dimensional PDEs. Moreover, while the focus of this article was on numerically estimating the solutions of PDEs, the capability to embed constraints into neural networks has the potential to positively impact performance when solving any problem that has constraints, not just differential equations, with a neural network.

Future work should investigate the performance of different neural network architectures on the estimated solution error. For example, Ref. [4] suggests a neural network architecture where the hidden layers contain element-wise multiplications and sums of sub-layers. The sub-layers are more standard neural network layers like the fully connected layers used in the neural networks of this article. Another architecture to investigate is that of extreme learning machines [22]. This architecture is a single layer neural network where the weights of the linear output layer are the only trainable parameters. Consequently, these architectures can ultimately be trained by linear or non-linear least squares for linear or non-linear PDEs respectively.

Another topic for investigation is reducing the estimated solution error by sampling the training points based on the loss function values for the training points of the previous iteration. For example, one could create batches where half of the new batch consists of half of the points in the previous batch that had the largest loss function value and the other half are randomly sampled from the domain. This should consistently give training points that are in portions of the domain where the estimated solution is farthest from the real solution.

Finally, future work will explore extending the hybrid systems approach presented in Ref. [23] to n-dimensions. Doing so would enable Deep TFC to solve problems that involve discontinuities at interfaces. For example, consider a heat conduction problem that involves two slabs of different thermal conductivities in contact with one another. At the interface condition, the temperature is continuous but the derivative of temperature is not.

Author Contributions: Conceptualization, C.L.; Formal analysis, C.L.; Methodology, C.L.; Software, C.L.; Supervision, D.M.; Writing—original draft, C.L.; Writing—review & editing, C.L. and D.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a NASA Space Technology Research Fellowship, Leake [NSTRF 2019] Grant #: 80NSSC19K1152.

Acknowledgments: The authors would like to acknowledge Jonathan Martinez for valuable advice on neural network architecture and training methods. In addition, the authors would like to acknowledge Robert Furfaro and Enrico Schiassi for suggesting extreme learning machines as a future work topic. Finally, the authors would like to acknowledge Hunter Johnston for providing computational resources.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

FEM	finite element method
IID	Independent and identically distributed
PDE	partial differential equation
TFC	Theory of Functional Connections

References

1. Argyris, J.; Kelsey, S. Energy Theorems and Structural Analysis: A Generalized Discourse with Applications on Energy Principles of Structural Analysis Including the Effects of Temperature and Non-Linear Stress-Strain Relations. *Aircr. Eng. Aerosp. Technol.* **1954**, *26*, 347–356. [CrossRef]
2. Turner, M.J.; Clough, R.W.; Martin, H.C.; Topp, L.J. Stiffness and Deflection Analysis of Complex Structures. *J. Aeronaut. Sci.* **1956**, *23*, 805–823. [CrossRef]
3. Clough, R.W. *The Finite Element Method in Plane Stress Analysis*; American Society of Civil Engineers: Reston, VA, USA, 1960; pp. 345–378.
4. Spiliopoulos, J.S.K. DGM: A deep learning algorithm for solving partial differential equations. *J. Comput. Phys.* **2018**, 1339–1364. [CrossRef]
5. Lagaris, I.E.; Likas, A.; Fotiadis, D.I. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. Neural Netw.* **1998**, *9*, 987–1000. [CrossRef] [PubMed]
6. Yadav, N.; Yadav, A.; Kumar, M. *An Introduction to Neural Network Methods for Differential Equations*; Springer: Dordrecht, The Netherlands, 2015. [CrossRef]
7. Coons, S.A. *Surfaces for Computer-Aided Design of Space Forms*; Technical report; Massachusetts Institute of Technology: Cambridge, MA, USA, 1967.
8. Mortari, D. The Theory of Connections: Connecting Points. *Mathematics* **2017**, *5*, 57. [CrossRef]
9. Mortari, D.; Leake, C. The Multivariate Theory of Connections. *Mathematics* **2019**, *7*, 296. [CrossRef]
10. Leake, C.; Johnston, H.; Smith, L.; Mortari, D. Analytically Embedding Differential Equation Constraints into Least Squares Support Vector Machines Using the Theory of Functional Connections. *Mach. Learn. Knowl. Extr.* **2019**, *1*, 1058–1083. [CrossRef]
11. Mortari, D. Least-squares Solutions of Linear Differential Equations. *Mathematics* **2017**, *5*, 48. [CrossRef]
12. Mortari, D.; Johnston, H.; Smith, L. Least-squares Solutions of Nonlinear Differential Equations. In Proceedings of the 2018 AAS/AIAA Space Flight Mechanics Meeting Conference, Kissimmee, FL, USA, 8–12 January 2018.
13. Johnston, H.; Mortari, D. Linear Differential Equations Subject to Relative, Integral, and Infinite Constraints. In Proceedings of the 2018 AAS/AIAA Astrodynamics Specialist Conference, Snowbird, UT, USA, 19–23 August 2018.
14. Leake, C.; Mortari, D. An Explanation and Implementation of Multivariate Theory of Connections via Examples. In Proceedings of the 2019 AAS/AIAA Astrodynamics Specialist Conference, AAS/AIAA, Portland, MN, USA, 11–15 August 2019.
15. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: tensorflow.org (accessed on 30 January 2020).
16. Baydin, A.G.; Pearlmutter, B.A.; Radul, A.A. Automatic differentiation in machine learning: A survey. *arXiv* **2015**, arXiv:1502.05767.
17. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. *arXiv* **2014**, arXiv:1412.6980.
18. Duchi, J.; Hazan, E.; Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* **2011**, *12*, 2121–2159.

19. Tieleman, T.; Hinton, G. *Lecture 6.5—RMSProp, COURSERA: Neural Networks for Machine Learning*; Technical report; University of Toronto: Toronto, ON, Canada, 2012.
20. Fletcher, R. *Practical Methods of Optimization*, 2nd ed.; Wiley: New York, NY, USA, 1987.
21. Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, Sardinia, Italy, 13–15 May 2010; Teh, Y.W., Titterington, M., Eds.; PMLR: Sardinia, Italy, 2010; Volume 9, pp. 249–256.
22. Huang, G.B.; Zhu, Q.Y.; Siew, C.K. Extreme learning machine: Theory and applications. *Neurocomputing* **2006**, *70*, 489–501. [CrossRef]
23. Johnston, H.; Mortari, D. Least-squares solutions of boundary-value problems in hybrid systems. *arXiv* **2019**, arXiv:1911.04390.