A NEURAL METHOD FOR SYMBOLICALLY SOLVING PARTIAL DIFFERENTIAL EQUATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

We describe a neural-based method for generating exact or approximate solutions to differential equations in the form of mathematical expressions. Unlike other neural methods, our system returns symbolic expressions that can be interpreted directly. Our method uses a neural architecture for learning mathematical expressions to optimize a customizable objective, and is scalable, compact, and easily adaptable for a variety of tasks and configurations. The system has been shown to effectively find exact or approximate symbolic solutions to various differential equations with applications in natural sciences. In this work, we highlight how our method applies to partial differential equations over multiple variables and more complex boundary and initial value conditions.

1 INTRODUCTION

Much of the physics governing the natural world can be written in the language of differential equations. Despite their simple appearances, these equations can be as challenging to solve as they are common. With recent advancements in machine learning, and deep learning in particular, a new tool became available for finding solutions to problems that previously had seemed impenetrable.

The typical approach that neural networks take for solving differential equations is to model the solution using a neural network function itself. This network is trained to fit the differential equation and produces highly accurate approximations to how the solution is supposed to behave. Although this is practically useful, it does not have the advantage of clarity and conciseness that symbolic mathematical expressions provide. Unfortunately, the application of deep learning to produce symbolic solutions is relatively underdeveloped.

In this work, we present a method for generating symbolic solutions to differential equations that takes advantage of the flexibility and strength of deep learning. Our method leverages the power of gradient-based back-propagation training to provide symbolic solutions for differential equations that can be easily and directly used and interpreted. We focus on partial differential equations over multiple variables, as these problems are more common and more complex than differential equations over a single variable. Indeed, ordinary differential equations can be solved as a special case using our proposed method, along with other symbolic mathematical tasks such as integration, function inversion, and symbolic regresson. As an added benefit, when our method fails to obtain an exact solution, including when no elementary solution exists, it will return a symbolic function that approximates the true solution, rather than leave you empty-handed.

In the sections that follow, we outline the core of our framework and the architecture of the multivariate symbolic function learner, or MSFL. We then show several experiments that demonstrate the power of our method on practical examples of PDEs.

2 RELATED WORK

There are several papers that apply neural networks for solving both ordinary and partial differential equations, including those by Meade Jr & Fernandez (1994); Lagaris et al. (1998); Berg & Nyström (2018); Malik et al. (2020). These approaches typically do not provide symbolic solutions to differential equations, but highly accurate approximate functions. There are also interesting works for discovering PDEs, such as by Long et al. (2019), which is a different problem from what we are addressing here.

One excellent work that directly links deep learning with symbolic mathematics is given by Lample & Charton (2019). This paper attempts to train a transformer model to learn the language of symbolic mathematics and "translate" between inputs and outputs of mathematical tasks, such as integration and solving differential equations. Although the results are remarkably impressive, they depend upon an extremely costly training procedure that cannot scale well to new configurations, for example, involving a different set of operations than the ones used in the training set. On a deeper level, the model has faced criticism for issues ranging from the artificiality of its testing to the fact that it "has no understanding of the significance of an integral or a derivative or even a function or a number" (Davis, 2019).

3 Method

There are two components to our symbolic PDE solution system. The multivariate symbolic function learning algorithm is a submodule that produces symbolic mathematical expressions over many variables in order to minimize a given cost function. The equation-solving wrapper is a system that uses the submodule to find solutions to partial differential equations. We will start by describing the wrapper system and how it relates to PDEs, and then outline a possible framework for the symbolic function learner that it uses.

3.1 Equation Solver System

A general form for representing any partial differential equation (PDE) is

$$g\left(x_1, x_2, \ldots, x_d, y, \frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_d}, \frac{\partial^2 y}{\partial x_1^2}, \frac{\partial^2 y}{\partial x_1 \partial x_2}, \ldots\right) = 0. \tag{1}$$

Here, g is a real-valued mathematical expression in terms of the variables $x_1, \ldots, x_d$, a function over these variables y, and any partial derivative of any order of y over any set of the variables.

It should be noted that this format encompasses far more than just PDEs; in fact, integration, function inversion, and many functional equation problems can be expressed in the form of Equation 1.

The solution to Equation 1 is the symbolic function $f(x_1, \ldots, x_d)$ that minimizes the expected error

$$L_1(f) = \mathbb{E}\left[\left|g\left(x_1, \ldots, x_d, f(x_1, \ldots, x_d), \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_d}, \frac{\partial^2 f}{\partial x_1^2}, \frac{\partial^2 f}{\partial x_1 \partial x_2}, \ldots\right)\right|^2\right] \tag{2}$$

where $x_1, \ldots, x_d$ are distributed over the desired domain of f. Note that in most cases, it is sufficient to use discrete approximations for partial derivatives, such as

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\boldsymbol{x} + \boldsymbol{e}^{(i)}\varepsilon/2) - f(\boldsymbol{x} - \boldsymbol{e}^{(i)}\varepsilon/2)}{\varepsilon}$$

$$\frac{\partial^2 f}{\partial x_i^2} \approx \frac{f(\boldsymbol{x} + \boldsymbol{e}^{(i)}\varepsilon) - 2f(\boldsymbol{x}) + f(\boldsymbol{x} - \boldsymbol{e}^{(i)}\varepsilon)}{\varepsilon^2}$$

where $\boldsymbol{e}^{(i)}$ is the unit vector in the ith direction, and ε is some small constant.

PDEs are frequently accompanied by boundary conditions or initial value constraints which solutions must satisfy. Often, these constraints come in the form of an interval, such as $f(x, 0) = c(x)$ for some specified c. We will approximate these constraints by taking N uniformly spaced points $(\boldsymbol{x}_i, y_i)$ along the interval and inserting them into the secondary error function

$$L_2(f) = \sum_i \|f(\boldsymbol{x}_i) - y_i\|^2. \tag{3}$$

Combining these two error functions gives the total error

$$L_{total} = L_1(f) + \lambda L_2(f). \tag{4}$$

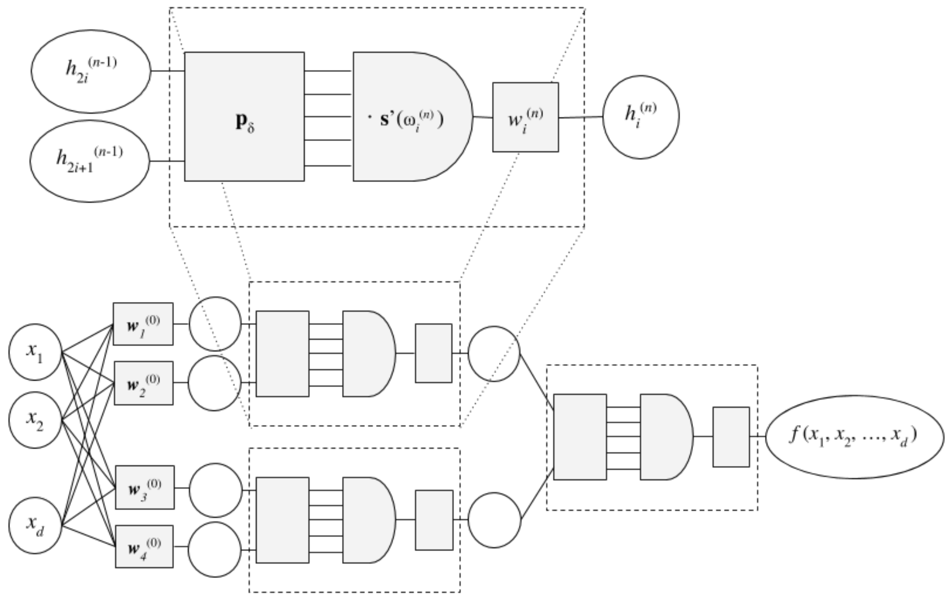

Figure 1: The architecture of the multivariate symbolic function learner (MSFL). **Top:** The neural representation of a single operator (interior) node in the symbolic parse tree. The operator function **p** takes in two child node as input and applies all available operators on their values. The discretized softmax function $\mathbf{s}'$ is a gate that allows exactly one of these operators to pass through, determined by learnable weight ω. This output is then scaled by learnable weight w. (Note: bias scalars b are omitted from the diagram to save space.) **Bottom:** After an initial layer of leaf nodes which combine in a fully connected fashion, a series of operator nodes form the template of a balanced binary parse tree. The weight parameters determine how to interpret this tree as a single, well-formed symbolic mathematical expression f over multiple variables.

The best solution to the PDE is the function f that minimizes L_{total}. To find such an f, we must perform an optimization search using L_{total} as an objective function. While neural networks do this naturally using back-propagation, it is not immediately clear how to perform this optimization over all symbolic functions. In the next section, we outline one way in which this can be done.

3.2 Multivariate Symbolic Function Learner (MSFL)

The multivariate symbolic function learner (or MSFL for short) is an algorithm that generates symbolic mathematical expressions that minimize a given cost function. Any algorithm meeting this requirement will be a suitable fit for the proposed symbolic PDE solving system. In particular, many symbolic regression algorithms would fit the role with minor adjustments, including those by Schmidt & Lipson (2009), Sahoo et al. (2018), and Udrescu & Tegmark (2020). In this section, we describe one model designed for this task.

Recall that every mathematical expression can be represented as a syntactic parse tree, where nodes in the interior represent operators and leaf nodes represent variables or constants. To evaluate the represented expression, each operator takes in the values of its children nodes as input. By introducing an identity operator and defining unary operators to apply on a pre-specified combination of two child nodes, these parse trees can be standardized to be balanced and perfectly binary.

If the structure of a parse tree is taken as a template, then it is possible to produce a mathematical function by identifying what operations and input quantities to use in each node of the tree. This is the essence of the MSFL algorithm. The goal is to learn what each node represents in the expression that minimizes the given cost function. We will show how this can be done in a way that is fully differentiable, and hence caters to deep learning techniques, such as back-propagation.

Let U be a list of allowable unary operators $[u_1, \dots, u_r]$ that map $\mathbb{R}$ to $\mathbb{R}$, and let V be a list of binary operators $[v_{r+1}, \dots, v_k]$ that map $\mathbb{R}^2$ to $\mathbb{R}$, for a total of k allowable operators. We define the "operate" function $\mathbf{p} : \mathbb{R}^2 \to \mathbb{R}^k$ by

$$\mathbf{p}(x_1, x_2) \quad = \quad [u_1(x_1 + x_2), \dots, u_r(x_1 + x_2), v_{r+1}(x_1, x_2), \dots, v_k(x_1, x_2)].$$

For example, if $U = [id, \sin, \exp]$ and $V = [\times]$, then

$$\mathbf{p}(x_1, x_2) = [x_1 + x_2, \sin(x_1 + x_2), e^{x_1+x_2}, x_1 x_2].$$

Note that we use id to refer to the unary identity function that returns its input value unchanged. In fact, it is equivalent to the addition operator because of how unary operators are defined to combine two child node inputs into one by using their sum.

We compute the **p** function in each interior node of the parse tree. In this way, we are applying all possible operators to the node's two children as input. In order to interpret the parse tree as a mathematical expression, one of these operators should be selected to pass on its output value, to the exclusion of all others. This can be done using a trainable gating operation.

The gate can be set up as follows. Let ω be a learnable weight vector in $\mathbb{R}^k$. Denote by **s** the softmax function, i.e.

$$\mathbf{s}(\omega) = \left[\frac{e^{\omega_i}}{\sum_{j=1}^{k} e^{\omega_j}} \right]_{i=1}^{k}.$$

Now $\mathbf{s}(\omega)$ is a nonnegative vector in $\mathbb{R}^k$ with entries summing to 1. The dot product $\mathbf{p}(h_1, h_2) \cdot \mathbf{s}(\omega)$ is then a convex linear combination of the outputs over all operators allowed by **p**, skewed to give most weight to the operator at the index corresponding to the largest entry of ω. The choice of operator that is represented by a given node can thus be learned by updating the learnable weight vector ω during training.

Although the softmax gate places most weight on a single entry of the vector of outputs passing through, it still retains nonzero contributions from all other operators in the output of **p**. This can be corrected by adjusting the output of $\mathbf{s}(\omega)$ to become $\mathbf{s}'(\omega)$, where $\mathbf{s}'$ is a discretized form of softmax that returns a vector with 1 at the entry corresponding to the largest value of ω and 0 at all other entries. One way to compute the discretized softmax is

$$\mathbf{s}'(\omega) = \left[H_1 \left(\frac{\mathbf{s}(\omega)_i}{\max \mathbf{s}(\omega)} \right) \right]_{i=1}^{k}$$

where H_1 is a narrow hump function centred at 1, such as $H_1(x) = e^{-1000(x-1)^2}$. Since the division operator, maximum function, and hump function are all differentiable almost everywhere, the discretized softmax preserves the differentiability of our model that allows deep learning using gradient-based training methods.

The above framework for a single operator node forms the foundational unit for the larger parse tree structure. Let m represent the number of layers in the entire parse tree. The number of layers is a measure of how complex the represented mathematical expression is allowed to be. Trees with more layers form a richer space of mathematical functions, but provide a challenge by expanding the search space significantly.

For $i = 0, \dots, 2^m - 1$, define

$$h_i^{(0)} = \mathbf{w}_i^{(0)} \cdot \mathbf{x} + b_i^{(0)}$$

where $\mathbf{x} = [x_1, \dots, x_d]$ is a vector of the variable in the mathematical expression being constructed, and $\mathbf{w}_i^{(0)} \in \mathbb{R}^d$ and $b_i^{(0)} \in \mathbb{R}$ are learnable parameters. This represents the lowest layer of the tree, consisting of leaf nodes that denote numerical quantities. Each of these quantities is in the form of a learnable linear combination of all possible variables.

Working up the layers of the tree, as $n = 1, \dots, m$, the value of each node in recursively defined as

$$h_i^{(n)} = w_i^{(n)} \left(\mathbf{s}' \left(\omega_i^{(n)} \right) \cdot \mathbf{p}_\delta \left(h_{2i}^{(n-1)}, h_{2i+1}^{(n-1)} \right) \right) + b_i^{(n)}$$

for each $i = 0, \dots, 2^{m-n-1}$. Here, each $\omega_i^{(n)} \in \mathbb{R}^k$ and $w_i^{(n)}, b_i^{(n)} \in \mathbb{R}$ are learnable weights whose values will be learned during the training process.

The value at the root node of the tree, $h_0^{(m)}$, is the value of the mathematical expression represented by the tree when evaluated using the input $\mathbf{x}$. It is by using this value for $\hat{f}(\mathbf{x})$ that the MSFL can be trained using the cost function in Equation 4. Note that $\hat{f}(\mathbf{x}) = h_0^{(m)}$ is obtained as a differentiable function over $\mathbf{x}$ and all learnable weights in the MSFL.

At the end of the training procedure, the MSFL returns the symbolic expression represented by the final state of the parse tree. That is, it interprets each interior tree node as the operator determined by the weight in its ω vector, and applies an affine transformation to each node using the defined by the corresponding w and b values. If the resulting function is not the exact solution to the PDE, it will usually be a close approximation, as it is the result of a training procedure designed to optimize for a low fitting error.

4 Experiments

We test our system on a number of PDE problems and demonstrate the results below.

In each problem, we run our system over a function g in the form of Equation 1. The MSFL algorithm is run using the unary operators $U = [id, \sin, \exp]$ and the binary operator $V = [\times]$. In order to avoid illegal argument errors, we automatically compute the absolute value before entering any value as input to operations defined only on the positive half-plane, such as the square root function. As mentioned in Section 3.2, the nature of the id operator removes the need for an explicit addition operator.

For each task, we run our method 20 times, training a model on 5000 randomly generated points within the domain of f each time. Each run returns the function represented by the parse tree after 6000 iterations. The standard softmax function $\mathbf{s}$ is used for the first 1250 (= 25%) of iterations of each run, and the discrete form $\mathbf{s}'$ is used for the remainder of the training. This allows an introductory exploratory training phase before the model converges to a single expression structure, spending the remaining iterations fine-tuning the values of constants. After 20 runs are complete, we select from the 20 returned functions the one with the lowest validation error, as per Equation 4.

In all of the solution visualizations shown below, the left-most graphs are plots of the learned symbolic functions. The central graphs show the residual error from the learned functions, that is, the value of g evaluated at $\boldsymbol{x}$ as in $L_1(f)$ used in Equation 2. The right graphs demonstrate how the learned functions fit the specified boundary conditions.

4.1 Wave Equation

The motion of wave travelling in one spatial dimension over time can be modelled by a function $u(x, t)$ that satisfies the PDE

$$\frac{\partial^2 u}{\partial t^2}(x, t) = c^2 \frac{\partial^2 u}{\partial x^2}(x, t) \tag{5}$$

for $x \in [0, \pi]$, $t > 0$. This type of motion can be found in particles vibrating around a rest position along a single direction (Speiser & Williams, 2008).

Consider the case where $c = 1$ and we are given the boundary conditions

$$u(0, t) = 0$$
$$u(\pi, t) = 0$$

for $t > 0$, and initial conditions

$$\begin{aligned} u(x, 0) &= 0 \\ \frac{\partial u}{\partial t}(x, 0) &= \sin x \end{aligned}$$

for $x \in [0, \pi]$.

This system has the exact solution $u(x,t) = \sin x \sin t$. Our method produced the symbolic solution

$$\hat{u}(x,t) = 1.0002 \sin(1.000x) \sin(0.9998t).$$

This result is represented visually in Figure 2.

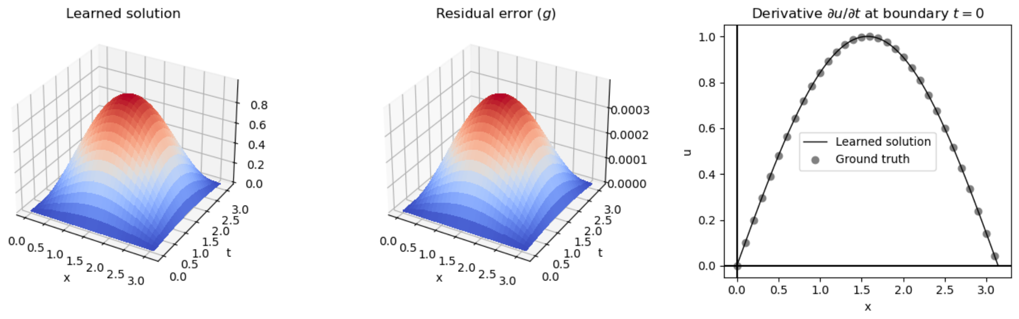

Figure 2: The learned solution to the wave equation (5), with residual error and boundary values.

4.2 Heat Equation

The diffusion of heat through a medium along a single spatial dimension over time can be modelled by a function $u(x,t)$ that satisfies the PDE

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{6}$$

for $x \in [0, \pi]$, $t > 0$. For example, this equation might be used to model how an uneven initial distribution of heat disperses and levels out along a rod of uniform composition.

Consider the case given by the boundary conditions

$$u(0,t) = 0$$
$$u(\pi,t) = 0$$

for $t > 0$, and initial conditions

$$u(x,0) = \sin x$$

for $x \in [0, \pi]$.

This system has the exact solution $u(x,t) = e^{-t} \sin x$. Our method produced the symbolic solution

$$\hat{u}(x,t) = (1.005e^{-0.994t} - 0.005) \sin(0.9996x + 0.001t).$$

This result is represented visually in Figure 3.

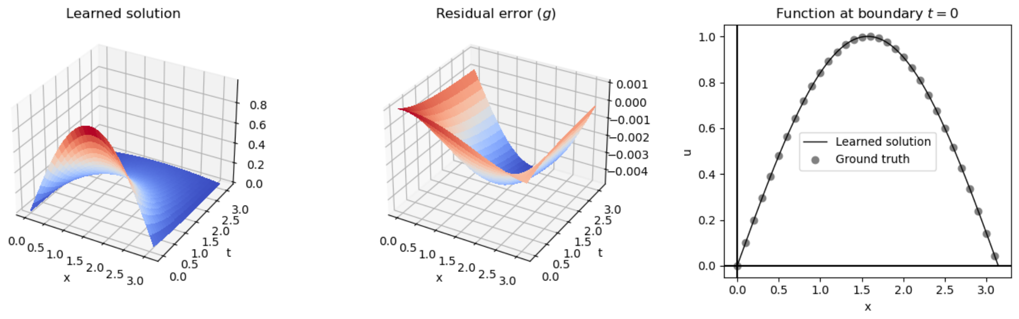

Figure 3: The symbolic solution to the heat equation (6), with residual error and boundary values.

4.3 Fokker-Planck Equations

The Fokker-Planck equation is a general model describing the behaviour of particles undergoing Brownian motion, with applications ranging from astrophysics to economics (Risken, 1996). The general format of a linear Fokker-Planck (FP) equation, also called a forward Kolmogorov equation, is

$$\frac{\partial}{\partial t}u(x,t) = -\frac{\partial}{\partial x}A(x,t)u(x,t) + \frac{\partial^2}{\partial x^2}B(x,t)u(x,t) \tag{7}$$

with initial condition $u(x,0) = f(x)$, where $A(x,t)$ and $B(x,t)$ (known as drift and diffusion coefficients, respectively), along with $f(x)$, are specified real-valued functions.

In order to fit the structure of Equation 1, we can reformulate Equation 7 as

$$g\left(u, \frac{\partial u}{\partial t}, \frac{\partial x}{\partial t}, \frac{\partial^2 x}{\partial t^2}\right) = \frac{\partial u}{\partial t} - \left(\left(\frac{\partial^2 B}{\partial x^2} - \frac{\partial A}{\partial x}\right)u + \left(2\frac{\partial B}{\partial x} - A\right)\frac{\partial u}{\partial x} + B\frac{\partial^2 u}{\partial x^2}\right) = 0. \tag{8}$$

An alternative version for an FP equation, known as the backward Kolmogorov equation, is

$$\frac{\partial}{\partial t}u(x,t) = -A(x,t)\frac{\partial}{\partial x}u(x,t) + B(x,t)\frac{\partial^2}{\partial x^2}u(x,t) \tag{9}$$

with initial condition $u(x,0) = f(x)$, where $A(x,t)$, $B(x,t)$, and $f(x)$ are specified real-valued functions. Note that this formulation is already close to matching the format of Equation 1, and does not need to be significantly rearranged.

We will look at three common examples of FP equations with known analytic solutions. These examples have been investigated in several existing works on FP equations, including those by Lakestani & Dehghan (2009), Dehghan & Tatari (2006), and Kazem et al. (2012).

4.3.1 Example 1

Consider the case of Equation 7 over $x, t \in [0,1]$ where $A(x,t) = -1$ and $B(x,t) = 1$, subject to the initial condition $f(x) = x$.

This system has the exact solution $u(x,t) = x + t$. Our method produced the symbolic solution

$$\hat{u}(x,t) = 1.0001x + 1.0001t.$$

This result is represented visually in Figure 4.

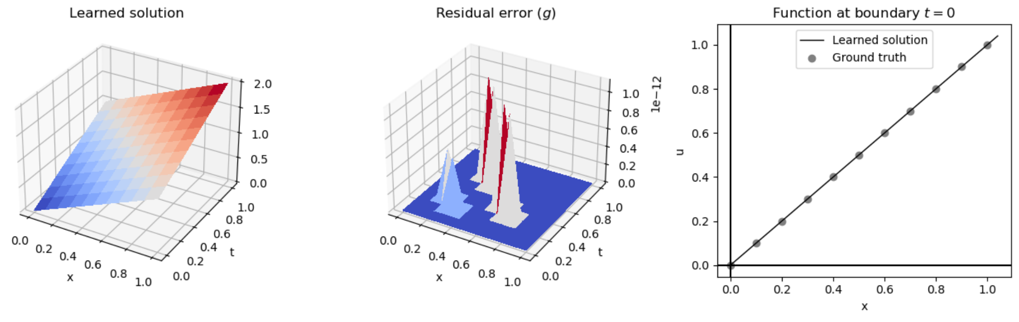

Figure 4: The symbolic solution to FP Example 1, with residual error and boundary values.

4.3.2 Example 2

Consider the case of Equation 7 over $x, t \in [0,1]$ where $A(x,t) = x$ and $B(x,t) = x^2/2$, subject to the initial condition $f(x) = x$.

This system has the exact solution $u(x,t) = xe^t$. Our method produced the symbolic solution

$$\hat{u}(x,t) = (0.972x - 0.001t + 0.002)e^{1.024t} + 0.029x - 0.002.$$

This result is represented visually in Figure 5.

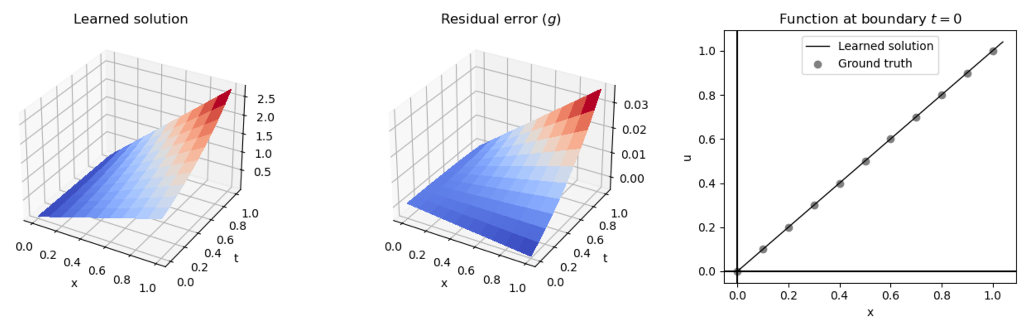

Figure 5: The symbolic solution to FP Example 2, with residual error and boundary values.

4.3.3 Example 3

Consider the case of Equation 9 over $x, t \in [0, 1]$ where $A(x, t) = -(x + 1)$ and $B(x, t) = x^2 e^t$, subject to the initial condition $f(x) = x + 1$.

This system has the exact solution $u(x, t) = (x + 1)e^t$. Our method produced the symbolic solution

$$\hat{u}(x, t) = (0.923x - 0.006t + 0.866)e^{0.011x+1.121t} + 0.057x + 0.136.$$

This result is represented visually in Figure 6.

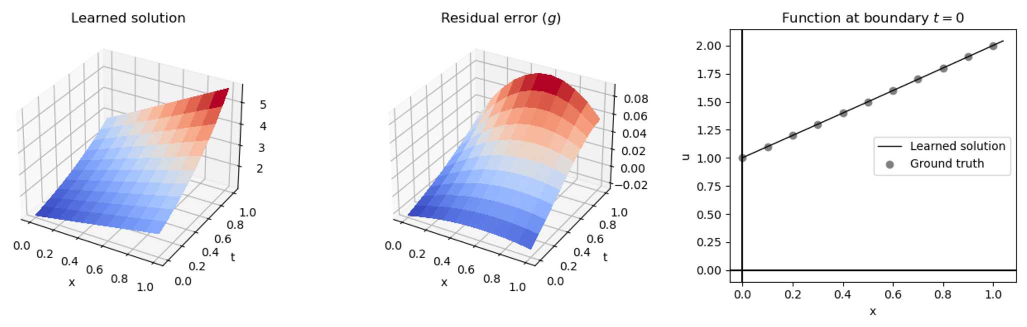

Figure 6: The symbolic solution to FP Example 3, with residual error and boundary values.

5 Conclusions and Future Work

In this work, we have shown a framework for producing symbolic solutions to partial differential equations over many variables using deep learning techniques. We have illustrated the utility of our system on a number of examples of PDEs taken from classical physics. Our method has shown its ability to generate solutions that are either exactly or approximately correct in many cases. In particular, the linear Fokker-Planck equations have provided good testing grounds for the strengths of our method.

Although the multivariate symbolic function learner has shown its capability, there is still room for improvement. As with all algorithms that seek to optimze over symbolic functions, the problem of an immensely vast search space poses a great challenge. It would be rewarding to see an MSFL sucessfully scale to spaces of increasingly complex functions over large sets of operators and variables. The modularity of our system allows the MSFL to be easily swapped with any other function learning algorithm, offering a good opportunity for future experiments.

In the bigger picture, we look forward to seeing more applications of deep learning in the realm of symbolic mathematics, and hope that this contribution will be a step towards that direction.

REFERENCES

Jens Berg and Kaj Nyström. A unified deep artificial neural network approach to partial differential equations in complex geometries. *Neurocomputing*, 317:28–41, 2018.

Ernest Davis. The use of deep learning for symbolic integration: A review of (Lample and Charton, 2019). *arXiv preprint arXiv:1912.05752*, 2019.

Mehdi Dehghan and Mehdi Tatari. The use of he's variational iteration method for solving a fokker-planck equation. *Physica Scripta*, 74(3):310, 2006.

S Kazem, JA Rad, and K Parand. Radial basis functions methods for solving fokker–planck equation. *Engineering Analysis with Boundary Elements*, 36(2):181–189, 2012.

Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

Mehrdad Lakestani and Mehdi Dehghan. Numerical solution of fokker-planck equation using the cubic b-spline scaling functions. *Numerical Methods for Partial Differential Equations: An International Journal*, 25(2):418–429, 2009.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2019.

Zichao Long, Yiping Lu, and Bin Dong. Pde-net 2.0: Learning pdes from data with a numeric-symbolic hybrid deep network. *Journal of Computational Physics*, 399:108925, 2019.

Shehryar Malik, Usman Anwar, Ali Ahmed, and Alireza Aghasi. Learning to solve differential equations across initial conditions. In *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020.

Andrew J Meade Jr and Alvaro A Fernandez. Solution of nonlinear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling*, 20(9):19–44, 1994.

Hannes Risken. Fokker-planck equation. In *The Fokker-Planck Equation*, pp. 63–95. Springer, 1996.

Subham S Sahoo, Christoph H Lampert, and Georg Martius. Learning equations for extrapolation and control. *arXiv preprint arXiv:1806.07259*, 2018.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

David Speiser and Kim Williams. *Discovering the principles of mechanics 1600-1800: essays by David Speiser*, volume 1. Springer Science & Business Media, 2008.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.