IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 19 November 2023, accepted 7 December 2023, date of publication 12 December 2023,
date of current version 19 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3342023

RESEARCH ARTICLE

Symbolic Neural Architecture Search for Differential Equations

PAULIUS SASNAUSKAS[1,2] AND LINAS PETKEVIČIUS[1], (Member, IEEE)
[1]Institute of Computer Science, Vilnius University, 08303 Vilnius, Lithuania
[2]Department of Computing, Imperial College London, SW7 2AZ London, U.K.

Corresponding author: Paulius Sasnauskas (paulius.sasn@gmail.com)

ABSTRACT In this paper, we introduce the first use of symbolic integration that leverages the machine learning infrastructure, such as automatic differentiation, to find analytical approximations of ordinary and partial differential equations. Analytical solutions to differential equations are at the core of fundamental mathematical models, which often cannot be determined analytically because of model complexity or non-linearity. Traditionally, the methods for solving these problems have used hand-designed strategies, numerical methods, or iterative methods. We propose a method that is an application of differentiable architecture search to find solutions to differential equations. We demonstrate our proposed method on a set of equations while simultaneously comparing it with numerical solutions to corresponding problems. We demonstrate that the proposed framework allows for solutions to various problems.

INDEX TERMS Symbolic integration, machine learning, PDE, neural architecture search.

I. INTRODUCTION

Mathematical models that use differential equations have various parameters that differ depending on the actual experiment, application, or area [1]. Thus, an approach is needed that would allow us to easily find analytical solutions of ordinary differential equations (ODEs), where changing these parameters and evaluating the model at different points in time or space would be easy and fast. Sometimes, an analytical solution exists, and it is possible to find one, but more often, such solutions are impossible to find, particularly if the system of equations contains a nonlinear part [2]. However, more complex ODE systems can be approximated using various numerical methods, which are most commonly iterative (e.g., finite element method, finite difference method [3]). These methods require re-evaluation of their functions at any change in the system parameters, which takes time (e.g., using a fine grid with the finite element method). Therefore, faster approaches are required for recalculation at various points when the parameters are changed.

Recent advances in machine learning have facilitated the creation of various tools that allow easy construction of machine learning models [4], [5], [6]. These frameworks allow quick iterations to construct parametric models more easily. Other software libraries have also appeared which allow for symbolic computation, for example, *SymPy* [7]. *SymPy* allows symbolic and numerical integration and differentiation, which can help solve the differential equations.

A. GENERALIZATION OF SYSTEM PARAMETERS

There have been many methods of solving differential equations, such as automating the solution of partial differential equations using the finite element method [8], [9], or a symbolic computation library solving some differential equations symbolically [7], [10], [11]. Many of them have various useful strategies and solutions, although none offer the ability to generalize over parameter intervals.

B. SMALL DATA

Another problem concerning solving differential equations is data. Usually, acquiring more data in complex systems governed by various laws is not possible [12]. We must draw conclusions from the small amount of data we have. Current state-of-the-art methods do not guarantee convergence, especially when there is no data, but it is more often the case that these small-data problems are more valuable.

The associate editor coordinating the review of this manuscript and approving it for publication was Yiming Tang.

© 2023 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

C. GENERAL ARCHITECTURE

In machine learning, specific architectures (manual selection of a parametric model form) are required to solve a particular class of problems such as classification or regression. After fixing the model, automatic differentiation [5], [13] enables the automatic use of optimization methods to estimate unknown model parameters [14]. Whenever a change in the objective or data complexity occurs, a change in architecture is required (sometimes even a complete redesign). To help solve this a novel method of *DARTS: Differentiable Architecture Search* was recently proposed [15]. The progress in the development of automatic differentiation, symbolic computation, and automatic architecture search allows for the combination of these ideas.

D. CONTRIBUTIONS

This work aims to create a machine learning architecture based on automatic differentiation, symbolic integration, and automatic neural architecture search, which could find an approximation given a differential equation, its initial and boundary conditions, its parameters, and intervals for these parameters. We aim to address the generalization of dimensionality, boundary and initial conditions, timescale, spatial resolution, and solution space smoothness.

II. RELATED WORK

Classical solvers, such as the finite difference method (FDM), finite element method (FEM), finite volume method (FVM) are numerical methods that partition the input space into a finite grid, and the solution at the elements of the grid is approximated by solving algebraic equations [3]. Semi-analytical methods, such as the homotopy perturbation method, search for an approximate analytical solution that is close enough to the exact solution [16]. These are the methods commonly used in science and engineering [8], [17] .

Learning mathematical models automatically is called *symbolic regression* [18]. In other words, it is an optimization problem over the space of model forms [19]. These methods discover equations characterizing relationships or underlying laws and physics in the data. These equations are often used as components of machine learning models, such as in the form of network nodes, loss terms, or regularization terms. There has been some work in *physics informed neural networks* (PINNs), where a neural network is trained and used to act as a solution to the differential equation [12], [20], [21], [22], [23], [24]. The yield of the PINN is the deep neural network – a black box, which can be used to perform inference. Whereas, the yield of our method is a portable symbolic formula to perform efficient use in future calculations. For a more complete overview of physics informed machine learning, we refer the reader to the work by Karniadakis et al. [18] and Cuomo et al. [25].

Brandstetter et al. [26] combine the ideas of classical solvers and physics informed machine learning methods and propose a grid-based graph neural network. They match the performance of state-of-the-art numerical solvers for some tasks. Zubov et al. [27] propose a symbolic regression method operating in a purely symbolic formulation combining various numerical and analytical techniques for finding analytical solutions. Similarly to PINNs, the yield is the whole network, not a symbolic expression.

Cranmer [28], [29] proposes an efficient open-source library for symbolic regression based on genetic algorithms. He proposed a multi-population evolutionary algorithm for equation discovery. It is a method highly dependent on data, and the output is a portable mathematical formula. These types of methods are known as Genetic programming symbolic regression (GPSR) [19], [29]. Moving beyond discovering formulas from data, such as the line of work by Cranmer [29], the methods by Oh et al. [19] propose solving differential equations with no data, i.e., only requiring the definition of the differential equation, boundary, initial conditions; and use the same GPSR approach. This aligns closely with what we want to obtain – symbolic (human interpretable) models, but in a setting for differential equations with small data. We present an approach that is different from GPSR.

We build on the work of *DARTS: Differentiable Architecture Search* [15] which proposes a differential architecture search method by continuous relaxation of the discrete architecture representation. Their results show that the architectures obtained are capable of meeting the performance of state-of-the-art architecture search methods. We examine this method by applying it to search for symbolic equations and formulas in the same way as architectures.

III. METHOD

A. DIFFERENTIAL EQUATIONS

Consider a non-linear differential equation in the domain Ω with a solution function $y = y(x)$:

$$A(y) = f(b), \quad b \in \Omega, \tag{1}$$

with boundary conditions

$$B(y, y_b) = 0, \quad b \in \partial\Omega, \tag{2}$$

where A is a differential operator, $f(b)$ is a known analytical function, B is a boundary operator, $\partial\Omega$ is a boundary of an area Ω.

The analytical solution y can be assumed to be approximated by a parametric $\hat{y} = \hat{y}(x, \theta), \theta \in \Theta \subset \mathbb{R}^{d_\theta}$ function (in some cases a neural network). In such case the search for an optimal approximation consists of two parts, first to minimize quadratic loss $\mathcal{L}_A$ (1):

$$\min_{\hat{y} \in \mathbb{R}} \quad (A(\hat{y}(x, \theta)) - f(b))^2, \quad b \in \Omega \tag{3}$$

and second to find searchable function to contain boundary conditions so to minimize the quadratic loss $\mathcal{L}_b$

$$\hat{y}(x, \theta) \in \arg\min_{\theta \in \mathbb{R}^{d_\theta}} (B(y(x, \theta), y_b(x, \theta))^2, \quad b \in \partial\Omega \tag{4}$$

However, solving such optimization problems with non-convex inner objectives is in general a NP-hard problem [30]. Approaching this to gradient-based methods (e.g., stochastic gradient descent) for optimizing such objectives is necessary due to the possible curse of dimensionality in the unknown parameter space [27], but this limits the view of the space and we may potentially become stuck in a local minimum.

Given that $\mathcal{L}_A$ is a quadratic twice continuously differentiable function the task is to find the optimal unknown model parameters $\theta^*(y)$ w.r.t. y. Under the smoothness assumption, the optimally is reached $\nabla_\theta \mathcal{L}_b(y, \theta) = \mathbf{0}$, which defines the function $\theta^*(y)$. With the assumption that $\min_\theta \mathcal{L}_b$ has a solution, there exists a (y, θ^*) such that $\nabla_\theta \mathcal{L}_b(y, \theta^*) = \mathbf{0}$. Under the condition that $\nabla_\theta \mathcal{L}_b(y, \theta^*) = \mathbf{0}$ is end-to-end continuously differentiable and that $\theta^*(y)$ is continuously differentiable at y, implicitly differentiating the last equality from both sides w.r.t. y and applying the chain rule leads to:

$$\frac{\partial(\nabla_\theta \mathcal{L}_b)}{\partial \theta}(y, \theta^*) \cdot \frac{\partial \theta^*}{\partial y}(y) + \frac{\partial(\nabla_\theta \mathcal{L}_b)}{\partial y}(y, \theta^*) = \mathbf{0}. \tag{5}$$

Assuming that the Hessian $\nabla^2_\theta \mathcal{L}_b$ is invertible, we can rewrite as follows:

$$\frac{\partial \theta^*}{\partial y}(y) = -\left(\nabla^2_\theta \mathcal{L}_b(y, \theta^*)\right)^{-1} \cdot \frac{\partial(\nabla_\theta \mathcal{L}_b)}{\partial y}(y, \theta^*). \tag{6}$$

Applying the chain rule for computing the total derivative of $\mathcal{L}_A$ with respect to y yields:

$$\frac{d\mathcal{L}_A}{dy} = \frac{\partial \mathcal{L}_A}{\partial \theta} \cdot \frac{\partial \theta^*}{\partial y} + \frac{\partial \mathcal{L}_A}{\partial y}, \tag{7}$$

where we have omitted the evaluation at (y, θ^*). Substituting and reordering yields:

$$\frac{d\mathcal{L}_A}{dy} = \frac{\partial \mathcal{L}_A}{\partial y} - \frac{\partial \mathcal{L}_A}{\partial \theta} \cdot \left(\nabla^2_\theta \mathcal{L}_b\right)^{-1} \cdot \frac{\partial^2 \mathcal{L}_b}{\partial \theta \partial y}. \tag{8}$$

which computes the gradient of $\mathcal{L}_A$, given the function $\theta^*(y)$. However, in most of the cases obtaining such a mapping is computationally expensive, and this was the reason of selecting high order architectures.

B. SYMBOLIC INTEGRATION

When we found any approximation $\hat{y}(x)$ of the solution, we would like to evaluate the quality of the analytical approximation. In contrast to numerical methods, with analytical approximation we can calculate the errors at given space points as well as at the whole domain. Let us consider this quadratic error function $\mathcal{L}$:

$$\begin{aligned}\mathcal{L} &= \mathcal{E}(\hat{y}) \\ &= \int_{x \in \Omega} (A(\hat{y}(x, \theta)) - f(b))^2 \mathrm{d}x.\end{aligned} \tag{9}$$

Since we know that the true solution must meet the conditions $A(y) = f(b), \quad b \in \Omega$, we can define the quadratic error of the solution, by integrating over the domain of interest.

The loss function $\mathcal{L}$ depends not only on the system variable $\hat{y}(x)$, but also on additional system parameters $\mathcal{E}(x_1, \ldots, x_K)$. If we fix the range of system parameters, then we can calculate the total error of the approximation, as follows:

$$\begin{aligned}\mathcal{L}_\mathrm{T} &= \mathcal{L}_\mathrm{T}(\hat{y}, x_1, \ldots, x_K) \\ &= \int_{x_{1,\mathrm{low}}}^{x_{1,\mathrm{up}}} .. \int_{x_{K,\mathrm{low}}}^{x_{K,\mathrm{up}}} \mathcal{E}\, \mathrm{d}x_1 .. \mathrm{d}x_K\,.\end{aligned} \tag{10}$$

where integration can be done by *SymPy*. In the case where the final y is a multivariate function, in practice we found it might not be possible to symbolically integrate, and in such cases the integral can be replace the Monte-Carlo simulated domain observations, when integration is applied only on a selected variable, and other domain values are calculated based on numerical estimates:

$$\begin{aligned}\mathcal{L}_\mathrm{T} &= \mathcal{L}_\mathrm{T}(\hat{y}, x_1, \ldots, x_K) \\ &= \int_{x_{j,\mathrm{low}}}^{x_{j,\mathrm{up}}} \sum_{x_1 \in [x_{1,\mathrm{low}}, x_{1,\mathrm{up}}]} .. \sum_{x_K \in [x_{K,\mathrm{low}}, x_{K,\mathrm{up}}]} \\ &\quad \times \mathcal{E}(x_1, .., x_{j-1}, x_{j+1}, .., x_K)\, \mathrm{d}x_j.\end{aligned} \tag{11}$$

C. DARTS

With recent advances in neural architecture search, several highly performant methods were discovered [15], [31], [32]. These automatically searched architectures have achieved highly competitive performance in tasks such as image classification and object detection. One such method is DARTS [15].

DARTS searches for computational cells – building blocks of the final architecture [15]. A cell is a directed acyclic graph consisting of a sequence of N nodes. Each node $x^{(i)}$ is a parametric function and each edge (i, j) between the nodes is associated with some operation $o^{(i,j)}$ that transforms $x^{(i)}$. The input of the nodes is defined as the previous layer outputs. The output of the cell is a reduction operation (e.g., concatenation, sum, etc.) with all the intermediate nodes.

The value of each intermediate node is computed using its predecessors

$$x^{(j)} = \sum_{i<j} \bar{o}^{(i,j)}(x^{(i)}) \tag{12}$$

Let $\mathcal{O}$ be a set of candidate operations where each operation is a function $o(\cdot)$ to be applied to $x^{(i)}$. The choice for the reduction operation is proposed as a softmax over the set of candidate operations, relaxing the discrete function selection into a continuous space. Let $\bar{o}^{(i,j)}$ be the mixed operation:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \sigma(\alpha^{(i,j)})_o\,,$$

where $\sigma(z)_i$ is the softmax function, or, more precisely,

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})}\, o(x) \tag{13}$$

where the the weights for each connection (i, j) are parameterized by a vector $\alpha^{(i,j)}$ of dimension $|\mathcal{O}|$. The actual architecture search is then defined as simply learning a set of variables $\alpha = \left\{\alpha^{(i,j)}\right\}$ via gradient descent, or another optimization algorithm. The remaining solution remains continuous and differentiable.

At the end of the search the architecture can be obtained by replacing each mixed operation $\bar{o}^{(i,j)}$ with the best operation:

$$o^{(i,j)} = \arg\max_{o \in \mathcal{O}} \alpha_o^{(i,j)} \tag{14}$$

ending up with a single operation for each edge between the nodes. This method can be exploited by providing a wide range of operations in $\mathcal{O}$ relevant to the problem being solved, in our case, parametric functions.

D. DARTS FOR PARAMETRIC EXPRESSIONS

The search of an analytical approximation for $\hat{y}(x)$ could be automated with inspiration from the DARTS method. The base model $\hat{y} = \hat{y}(x)$ is taken as a single cell of the DARTS network. We define the cell to have N nodes, therefore $x^{(0)} = x$, and $\hat{y} = x^{(N)}$, as shown in Fig. 1. The intermediate nodes are computed as shown in (13). The set $\mathcal{O}$ can include a zero function $o(z) = 0$ to denote no connection between the nodes, and other functions. Examples of functions include $o(z) = 1$, $o(z) = z$, $o(z) = z^2$, $o(z) = e^z$, and so on.

Then, the model is trained using gradient descent, minimizing loss $\mathcal{L}$ with respect to the weights α. Input data (x) is generated by sampling the user-specified input parameter intervals uniformly. Experiments showed that usage of *SymPy* with complex equations required much computing time. To speed up the computation and make the equations simpler the softmax function was changed to a simple product to have a regularization term which penalized the loss if the sum of weights are not equal to one

$$\bar{o}^{(i,j)}(x) = \left(\sum_{o \in \mathcal{O}} \alpha_o^{(i,j)} o(x)\right) + \beta, \tag{15}$$

and an additional regularization loss term

$$\mathcal{L}_{\text{reg.}} = \left(\left(\sum_{o \in \mathcal{O}} \alpha_o^{(i,j)}\right) - 1\right)^2, \tag{16}$$

$$\mathcal{L} = \int_{\text{k}_{\text{low}}}^{\text{k}_{\text{up}}} \mathcal{L}_{\text{eq.}}\, \text{d}k + \gamma_{\text{reg.}} \mathcal{L}_{\text{reg.}}, \tag{17}$$

where $\gamma_{\text{reg.}}$ is a hyperparameter, $k \in [\text{k}_{\text{low}}, \text{k}_{\text{up}}]$ is the interval for parameter k.

These two changes replace similar functionality that was carried out by the softmax function – all of the values after the softmax added up to one (characteristic of (16)) and allowing the weights to influence the operations (characteristic of (15)), and provide a speedup in the symbolic integration, if applicable.

E. PRACTICAL OPTIMIZATION IMPROVEMENTS

To simplify the model further, after a specific amount of epochs N_e the model would be pruned. A pruning strategy is proposed below:

1) Take the first unpruned edge's vector of weights $\alpha^{(i,j)}$.
2) Replace the vector with the largest operation like in (14) and remove the operations of the discarded weights, thus leaving the mixed operation as $\bar{o}^{(i,j)}(x) = \alpha_{o^*}^{(i,j)} o^*(x)$ for that edge, where $o^*(x)$ is the operation with the maximum weight α in that edge. Another approach is to remove a single weight with the lowest value instead of all and remove the operation of the discarded weight.
3) If the edge contains only one weight, mark the edge as pruned.
4) Repeat after N_e epochs.

When there is nothing else left to prune, the solution expression can be deduced, as shown in Fig. 1(b).

Example: Consider the example, where we have operations $\mathcal{O} = \{1, x, \sin(x)\}$, a single edge, where the model at some point is $\hat{y} = \sum_{o \in \mathcal{O}} \alpha_o o(x) + \beta$, that is, $\hat{y} = \alpha_1 + \alpha_2 x + \alpha_3 \sin(x) + \beta$. Assume the algorithm learnt the parameters $\alpha_1 = 0.01$, $\alpha_2 = 0.01$, $\alpha_3 = 0.98$. Then, the pruning step would replace that edge with just $\hat{y} = \alpha_3 \sin(x) + \beta$.

For more complex problems the differential equations might contain initial conditions, boundary conditions. Thus, there must be way of including these in the model or in the optimization algorithm. The proof of concept model has been extended to allow both types of conditions.

1) INITIAL CONDITIONS

Let the initial conditions be of the form:

$$y(x_0) = y_0, \tag{18}$$

then, a loss term of this form can be added:

$$\mathcal{L}_{\text{init.cond.}} = \left(y_0 - \hat{y}(x_0)\right)^2, \tag{19}$$

for each x_0. The loss then becomes

$$\mathcal{L} = \int_{\text{k}_{\text{low}}}^{\text{k}_{\text{up}}} \mathcal{L}_{\text{eq.}}\, \text{d}k + \gamma_{\text{reg.}} \mathcal{L}_{\text{reg.}} + \gamma_{\text{init.cond.}} \mathcal{L}_{\text{init.cond.}}, \tag{20}$$

where $\gamma_{\text{init.cond.}}$ is a hyperparameter.

2) BOUNDARY CONDITIONS

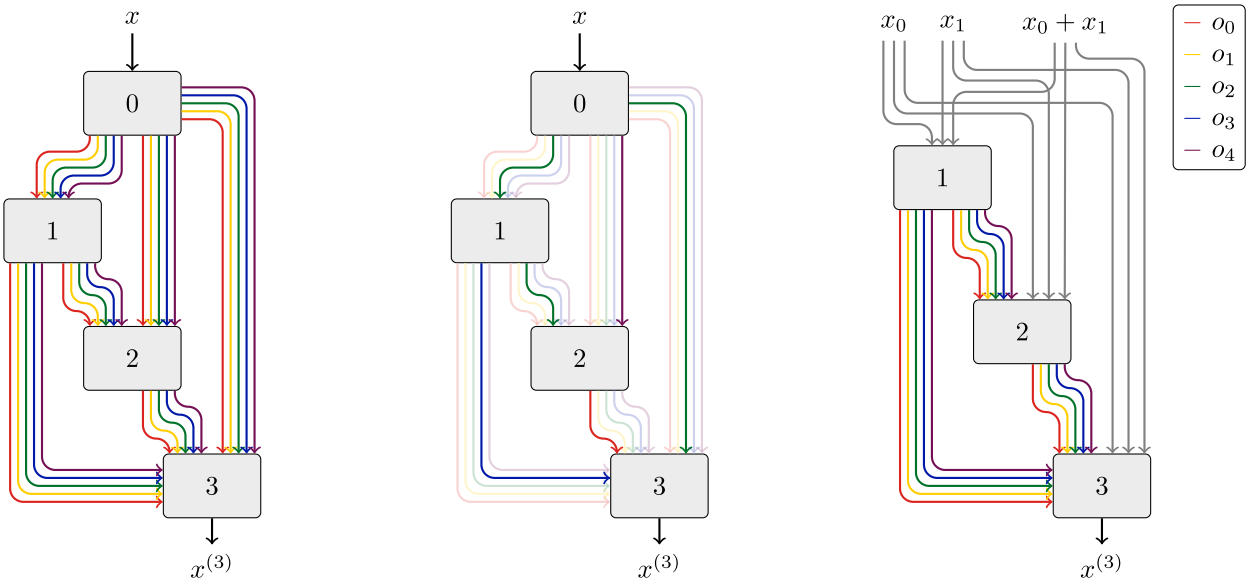
Assume the condition is in the form:

$$\left.\frac{\text{d}y}{\text{d}x}\right|_{x=x_{\text{b}}} = g(x)$$

where x_b is the boundary. These conditions can be added in a similar way how the initial conditions were added:

$$\mathcal{L}_{\text{bound.cond.}} = \left(\left.\frac{\text{d}\hat{y}}{\text{d}x}\right|_{x=x_{\text{b}}} - g(x)\right)^2 \tag{21}$$

The differentiation can be completed using autodiff or *SymPy*. Additional constraints on the problem can be added in a similar manner.



(a) Initial model with each edge containing 5 operations. The operations are $\mathcal{O} = \{o_0, o_1, o_2, o_3, o_4\}$.

(b) Trained model with each edge containing only the most prominent operation. Faded lines show pruned operations.

(c) Example model with multiple input variables and operations $\mathcal{O}_{\text{in}} = \{o_{\text{in}_0}(x_0, x_1) = x_0,\ o_{\text{in}_1}(x_0, x_1) = x_1, o_{\text{in}_2}(x_0, x_1) = x_0 + x_1\}$.

FIGURE 1. Proof of concept architecture with 5 operations. Initially the model contains the composition of the sums of all initial formulas. After training each edge is a single operation, and the symbolic formula can be inferred from the architecture. In the multiple variable case (c) each of the grey lines have a corresponding α which are also trainable.

F. MULTIVARIATE FUNCTIONS

To allow the network to support all kinds of functions, we redefine the input of the model. Let $\mathcal{O}_{\text{in}}$ be a set of candidate operations where each operation is a function $o_{\text{in}}(\cdot)$ to be applied to the input $x_0, x_1, \ldots, x_m$, where m is the number of input variables. The operations of the initial input block are swapped out for operations in $\mathcal{O}_{\text{in}}$. For example, a model with two input variables is presented in Fig. 1(b).

The mixed operations for the blocks are then calculated using

$$\bar{o}^{(i,j)}(x) = \begin{cases} \sum\limits_{o_{\text{in}} \in \mathcal{O}_{\text{in}}} \alpha_{o_{\text{in}}}^{(i,j)} o_{\text{in}}(x_0, x_1, \ldots, x_m), & i = 0, \\ \left(\sum\limits_{o \in \mathcal{O}} \alpha_o^{(i,j)} o(x) \right) + \beta, & i \neq 0. \end{cases} \tag{22}$$

The weights $\alpha_{o_{\text{in}}}^{(i,j)}$ are trained along all other weights $\alpha_o^{(i,j)}$ and participate in weight pruning in the same way.

IV. EXPERIMENTS AND RESULTS

The experiments were done in JAX [6] using Optax [33] and SymPy [7] The code developed for this method is available on github.com/PauliusSasnauskas/sasde. The hyperparameters used in the experiments below are presented in Table 1. Further experiment details and Python implementation are included in supplementary material.

A. TOY EXAMPLE

We take the Malthusian population model differential equation and conditions

$$\frac{dP}{dt} = rP, \quad P(1) = e^r, \tag{23}$$

where $P = P(t)$ is our unknown function, r and t are the variable parameters. Usually the goal is to find a good enough approximation $\hat{P} = \hat{P}(t)$ for a specific value of r and t. Instead of solving for a single value, we reduce the problem for finding the approximation for an interval $r \in [\text{r}_{\text{low}}, \text{r}_{\text{up}}] = R$ and $t \in [\text{t}_{\text{low}}, \text{t}_{\text{up}}] = T$, bound by empirical findings, laws of physics, or chosen values.

For the input the training procedure sampled 512 values for r and t uniformly from the intervals R and T respectively.

This example has a simple analytical solution $P = P_0\, e^{rt}$ (where P_0 is the initial population size), which can be compared to found solutions. The results indicated the architecture is working as expected. The final equation form obtained from this experiment is

$$\hat{y} = 0.999999940395355\, e^{1.0\, rt} - 1.12404698882074 \cdot 10^{-7}. \tag{24}$$

The solution plot can be seen in supplementary material.

The found approximation $\hat{P}$ can be integrated like presented in (10) w.r.t. both variables t and r to validate if it satisfies the conditions. Substituting $\hat{P}$ with (24) and intervals $T = [0, 1]$, $R = [1, 2]$ for variables t and r respectively we obtain an error of $3.80403517432048 \cdot 10^{-13}$.

TABLE 1. Hyperparameter choices and runtime details for the experiments.

Hyperparameter	Malthus model	Burgers' equation	Euler-Tricomi equation
Equation spec.	(23)	(25)	(30)
$\mathcal{O}_{\text{in}}$	$0, 1, r, t, rt$	$0, 1, t, x, -x, e^x, tx, e^x t$	$0, 1, x, y, xy, x + y$
$\mathcal{O}$	$0, 1, z, -z, z^2, e^z$	$0, 1, z, -z, z + 1, e^z$	$0, 1, z, -z, zx, zy, e^z, \sin(z)$
Learning rate	0.01	0.001	0.002
Alpha penalty $\gamma_{\text{reg.}}$	2	1	1
Node count	4	5	5
Num. epochs N_e	32	128	256
Num. samples	512	512	512
Batch size	64	64	128

TABLE 2. Runtime details for the experiments.

Runtime details	Malthus model	Burgers' equation	Euler-Tricomi equation
Equation spec.	(23)	(25)	(30)
Approx. time	1 min	9 min	15 min
Error*	$3.80404 \cdot 10^{-13}$	0.44282	0.15527

B. BURGERS' EQUATION

We take a specific Burgers' equation specification:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \frac{0.01}{\pi}\frac{\partial^2 u}{\partial x^2}, \tag{25}$$

with boundary and initial conditions:

$$u(0, x) = -\sin(\pi x), \tag{26}$$

$$u(t, -1) = u(t, 1) = 0, \tag{27}$$

$$t \in [0, 1], \quad x \in [-1, 1], \tag{28}$$

and apply our proposed architecture on the equation, we obtain the analytical approximation:

$$\begin{aligned} u(t, x) = {} & 0.130458233489587\, tx + 0.16775517978791\, x \\ & - 1.08303360615556\, e^{0.219613581895828\, x} \\ & - 0.142374154435032\, e^{x} \\ & + 0.431302160024643\, e^{0.365474700927734\, tx} \\ & + 0.870762419457051. \end{aligned} \tag{29}$$

The solution plots can be seen in supplementary material.

C. EULER-TRICOMI EQUATION

We take a specific Euler-Tricomi equation specification:

$$x^2\frac{d^2 u}{dx^2} + y^2\frac{d^2 u}{dy^2} = 0 \tag{30}$$

$$u(x, x^2) = \sin(x), \tag{31}$$

$$u(0, y) = y^2, \tag{32}$$

$$x \in [0, 2], \quad y \in [0, 2], \tag{33}$$

and obtain the analytical approximation:

$$\begin{aligned} u(x, y) = {} & -1.053338832861\, xy + 0.495143982030023\, x \\ & + 0.557947342762368\, y^2 + 0.0786390291381303\, y \\ & + 0.922053234446157 \sin(0.976893961429596\, y) \\ & + 0.652301847934723 \sin\Big(0.26658496260643\, x \\ & + 0.496432423591614 \sin(0.976893961429596\, y)\Big) \\ & - 0.294711053371429. \end{aligned} \tag{34}$$

The solution plot can be seen in supplementary material. The runtime details, such as the integrated error (10), are presented in Table 2. The shown error is from (10) for the toy example; for Burgers' equation and Euler-Tricomi equation the mean of the error function evaluated at a grid of points with a step of 0.01.

D. LIMITATIONS

The hyperparameter tuning procedure of each problem, such as the specification of the operations $\mathcal{O}$ is manual, which means the method may be unsuccessful in the search. Although, in mathematical modeling, practitioners suggest a set of assumptions (functions) to construct potential analytic solutions, and define permissible ways to assemble expressions (such as exponentiation, addition, multiplication, etc.) [19]. This also provides a way to suggest hints about the system we may know from the laws of physics or other observations, such as some proportionalities or scalings. The problem of huge search space [32] or scaling [31] still exists and is an open problem in neural architecture search. To bypass this problem some works [34] focus on a specific type of architecture (e.g., convolutional layers) and search in that domain only. Instead, we would like an algorithm that automatically selects the best combinations, much like in the work by Cranmer [29], which also supports function hints. Another limitation is that, for many cases symbolic integration with SymPy cannot integrate some complex equations or takes considerably longer than then whole number of epochs to integrate a single equation. In such cases we replace integration with numerical approximation as shown in (11) and obtain near-identical results, and additionally in our implementation leave the ability for the user to specify if they wish to experiment with symbolic integration.

V. CONCLUSION

We presented our proposed architecture showing how it can assist in finding solutions for differential equations. Empirical investigation revealed that this method is able to obtain satisfactory symbolic equations and agrees with analytical solutions of some differential equations. We highlight the end-to-end differentiability of this approach, combining elements from the ML pipeline and symbolic integration for searching analytical solutions. Due to the complexity of some differential equations there is no ability to compare to analytical solutions. The results suggest that the proposed architecture fits well in the toolbox of methods for solving differential equations.

Future Work: We aim to extend the work to by including automatic operation selection and support for systems of equations, and provide a basis for other applications in finding analytical approximations.

APPENDIX A
EXPERIMENTAL SETUP

The experiments were run on an *Intel(R) Core(TM) i7-1065G7* CPU. Data was generated by sampling the specified parameter intervals uniformly. The hyperparameters were chosen by manual random search. The learning rate had a linear scheduler with 1000 steps from γ to $\frac{1}{10}\gamma$, where γ is the specified learning rate. The gradients were clipped to 1 by the global norm.

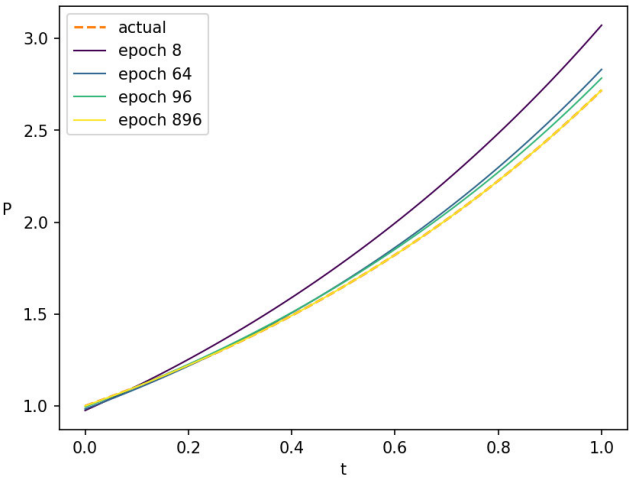


FIGURE 3. Malthus model equation solutions $P(t)$ after a number of epochs during training, compared to the actual solution $P(t) = e^{rt}$. The solution after the last epoch (yellow) overlaps with the actual solution (dashed orange).

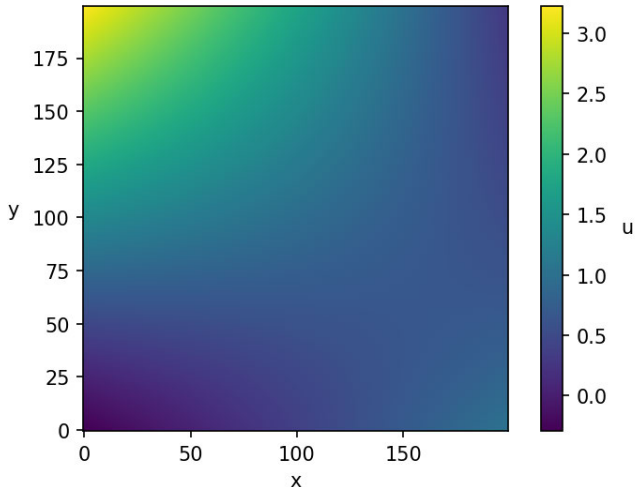


FIGURE 2. Euler-Tricomi equation (30) solution $u(x, y)$ given by (34).

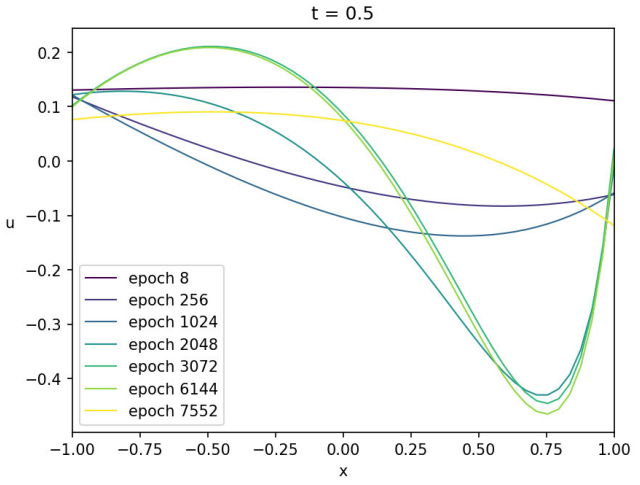


FIGURE 4. Burgers' equation solutions $u(t, x)$ after a number of epochs during training, at $t = 0.5$.

TABLE 3. Hyperparameter choices and runtime details for the experiments. * Error – loss integrated as shown in (10) for the toy example; for Burgers' equaton and Euler-Tricomi equation the mean of the error function evaluated at a grid of points with a step of 0.01.

Hyperparameter	Malthus model	Burgers' equation	Euler-Tricomi equation
Equation spec.	(23)	(25)	(30)
$\mathcal{O}_{\text{in}}$	$0, 1, r, t, rt$	$0, 1, t, x, -x, e^x, tx, e^x t$	$0, 1, x, y, xy, x + y$
$\mathcal{O}$	$0, 1, z, -z, z^2, e^z$	$0, 1, z, -z, z + 1, e^z$	$0, 1, z, -z, zx, zy, e^z, \sin(z)$
Learning rate	0.01	0.001	0.002
Alpha penalty $\gamma_{\text{reg.}}$	2	1	1
Node count	4	5	5
Num. epochs N_e	32	128	256
Num. samples	512	512	512
Batch size	64	64	128
Runtime details	**Malthus model**	**Burgers' equation**	**Euler-Tricomi equation**
Approx. time	1 min	9 min	15 min
Error*	$3.80404 \cdot 10^{-13}$	0.44282	0.15527

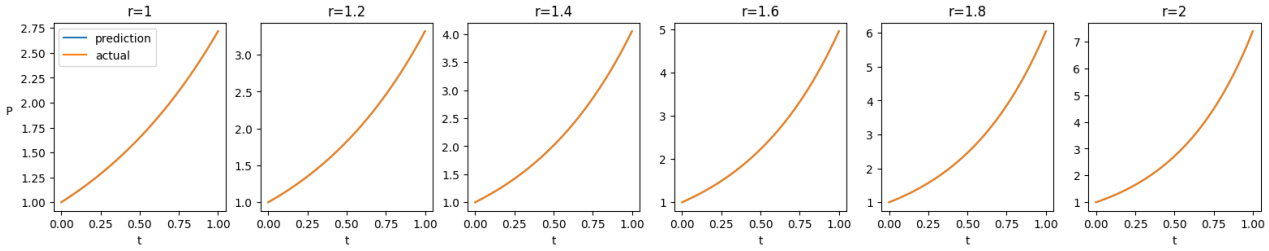


FIGURE 5. Malthus model equation (23) solution $P(t)$ given by (24) at $r \in \{1,\ 1.2,\ 1.4,\ 1.6,\ 1.8,\ 2\}$. The prediction and actual functions are overlapping.

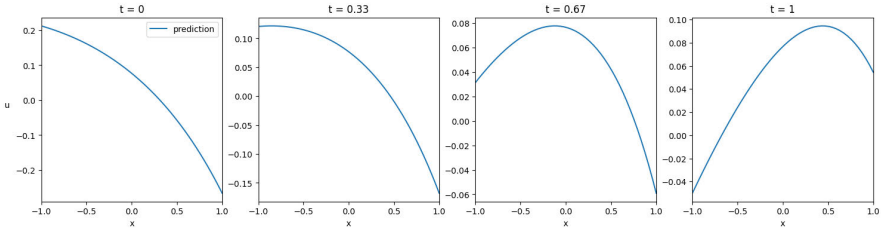


FIGURE 6. Burgers' equation (25) solution $u(t, x)$ given by (29) at $t \in \{0,\ 0.33,\ 0.67,\ 1\}$.

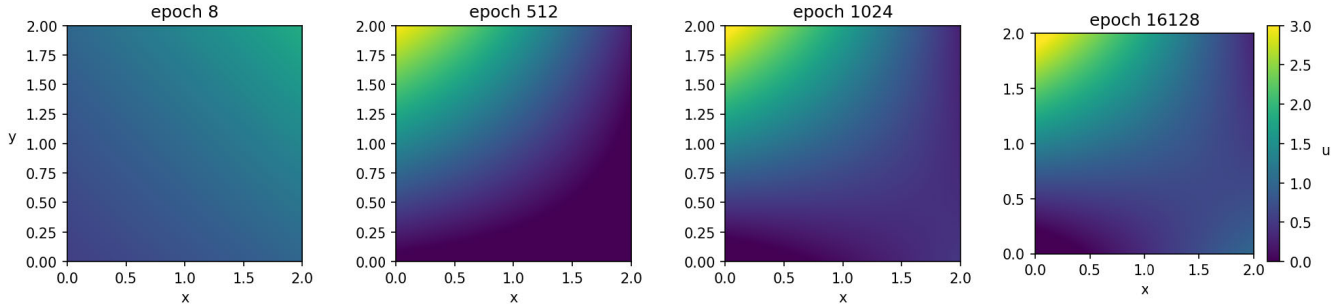


FIGURE 7. Euler-Tricomi equation solutions $u(x, y)$ after a number of epochs during training.

APPENDIX B
SOLUTION PLOTS

See Figs. 2–7 and Table 3.

REFERENCES

[1] M. Braun and M. Golubitsky, *Differential Equations and Their Applications*, vol. 1. Cham, Switzerland: Springer, 1983.

[2] D. Jordan and P. Smith, *Nonlinear Ordinary Differential Equations: An Introduction for Scientists and Engineers*. Oxford, U.K.: OUP Oxford, 2007.

[3] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*. Hoboken, NJ, USA: Wiley, 2016.

[4] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2019, pp. 8024–8035. Accessed: Nov. 9, 2023. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[5] M. Abadi et al. (2015). *TensorFlow: Large-Scale Machine learning on Heterogeneous Systems*. [Online]. Available: https://www.tensorflow.org/

[6] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. (2018). *JAX: Composable Transformations of Python+NumPy Programss*. Accessed: Nov. 9, 2023. [Online]. Available: http://github.com/google/jax

[7] A. Meurer et al., "SymPy: Symbolic computing in Python," *PeerJ Comput. Sci.*, vol. 3, p. e103, Jan. 2017.

[8] A. Logg and G. N. Wells, "DOLFIN," *ACM Trans. Math. Softw.*, vol. 37, no. 2, pp. 1–28, Apr. 2010. Accessed: Nov. 11, 2023, doi: 10.1145/1731022.1731030.

[9] V. Nehra, "MATLAB/Simulink based study of different approaches using mathematical model of differential equations," *Int. J. Intell. Syst. Appl.*, vol. 6, no. 5, pp. 1–24, Apr. 2014.

[10] G. F. Jefferson and J. Carminati, "FracSym: Automated symbolic computation of lie symmetries of fractional differential equations," *Comput. Phys. Commun.*, vol. 185, no. 1, pp. 430–441, Jan. 2014.

[11] C. Rackauckas and Q. Nie, "DifferentialEquations.Jl—A performant and feature-rich ecosystem for solving differential equations in Julia," *J. Open Res. Softw.*, vol. 5, no. 1, p. 15, May 2017, doi: 10.5334/jors.151.

[12] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (Part I): Data-driven solutions of nonlinear partial differential equations," 2017, *arXiv:1711.10561*.

[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Proc. NIPS Workshop Autodiff*, 2017., pp. 1–4.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[15] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *Proc. 7th Int. Conf. Learn. Represent.*, New Orleans, LA, USA, May 2019, pp. 1–13, Accessed: Nov. 9, 2023. [Online]. Available: https://openreview.net/forum?id=S1eYHoC5FX

[16] J.-H. He, "Homotopy perturbation technique," *Comput. Methods Appl. Mech. Eng.*, vol. 178, nos. 3–4, pp. 257–262, Aug. 1999.
[17] D. Goodarzi, K. Sookhak Lari, E. Khavasi, and S. Abolfathi, "Large eddy simulation of turbidity currents in a narrow channel with different obstacle configurations," *Sci. Rep.*, vol. 10, no. 1, p. 12814, Jul. 2020, doi: 10.1038/s41598-020-68830-5.
[18] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Rev. Phys.*, vol. 3, no. 6, pp. 422–440, May 2021, doi: 10.1038/s42254-021-00314-5.
[19] H. Oh, R. Amici, G. Bomarito, S. Zhe, R. Kirby, and J. Hochhalter, "Genetic programming based symbolic regression for analytical solutions to differential equations," 2023, *arXiv:2302.03175*.
[20] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou, "Neural-network methods for boundary value problems with irregular boundaries," *IEEE Trans. Neural Netw.*, vol. 11, no. 5, pp. 1041–1049, 2000, doi: 10.1109/72.870037.
[21] M. Raissi, P. Perdikaris, and G. Em Karniadakis, "Physics informed deep learning (Part II): Data-driven discovery of nonlinear partial differential equations," 2017, *arXiv:1711.10566*.
[22] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis, "Physics-informed neural networks (PINNs) for fluid mechanics: A review," *Acta Mechanica Sinica*, vol. 37, no. 12, pp. 1727–1738, Dec. 2021.
[23] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman, "Universal differential equations for scientific machine learning," 2020, *arXiv:2001.04385*.
[24] J. Donnelly, S. Abolfathi, and A. Daneshkhah, "A physics-informed neural network surrogate model for tidal simulations," in *Proc. ECCOMAS*, 2023, pp. 836–844.
[25] S. Cuomo, V. S. di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific machine learning through physics-informed neural networks: Where we are and what's next," 2022, *arXiv:2201.05624*.
[26] J. Brandstetter, D. E. Worrall, and M. Welling, "Message passing neural PDE solvers," in *Proc. 10th Int. Conf. Learn. Represent.*, 2022, pp. 1–27, Accessed: Nov. 9, 2023. [Online]. Available: https://openreview.net/forum?id=vSix3HPYKSU
[27] K. Zubov, Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe, N. Vinchhi, K. Balakrishnan, D. Upadhyay, and C. Rackauckas, "NeuralPDE: Automating physics-informed neural networks (PINNs) with error approximations," 2021, *arXiv:2107.09443*.
[28] M. Cranmer. (2020). *PYSR: Fast & Parallelized Symbolic Regression in Python/Julia*. Accessed: Nov. 9, 2023. [Online]. Available: http://doi.org/10.5281/zenodo.4041459
[29] M. Cranmer, "Interpretable machine learning for science with PySR and SymbolicRegression.Jl," 2023, *arXiv:2305.01582*.
[30] P. Hansen, B. Jaumard, and G. Savard, "New branch-and-bound rules for linear bilevel programming," *SIAM J. Sci. Stat. Comput.*, vol. 13, no. 5, pp. 1194–1217, Sep. 1992.
[31] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in *Proc. Int. Conf. Learn. Represent.*, 2017, pp. 1–16, Accessed: Nov. 9, 2023. [Online]. Available: https://openreview.net/forum?id=r1Ue8Hcxg
[32] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proc. 35th Int. Conf. Mach. Learn.*, in Proceedings of Machine Learning Research, vol. 80, J. Dy and A. Krause, Eds., Jul. 2018, pp. 4095–4104. Accessed: Nov. 9, 2023. [Online]. Available: https://proceedings.mlr.press/v80/pham18a.html
[33] I. Babuschkin et al. (2020). *The DeepMind JAX Ecosystem*. Accessed: Nov. 9, 2023. [Online]. Available: https://github.com/google/jax
[34] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in *Proc. 38th Int. Conf. Mach. Learn.*, in Proceedings of Machine Learning Research, vol. 139, M. Meila and T. Zhang, Eds., 2021, pp. 7588–7598. [Online]. Available: https://proceedings.mlr.press/v139/mellor21a.html

PAULIUS SASNAUSKAS received the B.S. degree in software engineering from Vilnius University, in 2022, and the M.S. degree in advanced computing from the Department of Computing, Imperial College London, in 2023. His research interests include theory of machine learning, neural architecture search, and reinforcement learning.

LINAS PETKEVIČIUS (Member, IEEE) received the Ph.D. degree in informatics from the Institute of Computer Science, Vilnius University, in 2020. Since 2022, he has been the Head of the Software Engineering Department, Institute of Computer Science. His research interests are focused on computer vision, deep learning, statistical inference, and outliers detection.