VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
SOFTWARE ENGINEERING BACHELOR STUDY PROGRAMME

Finding Analytical Approximations of Differential Equations Using Machine Learning and Symbolic Computation

Analizinių artinių diferencialinėse lygtyse radimas taikant mašininį mokymą ir simbolinius skaičiavimus

Final Bachelor Thesis

Author: Paulius Sasnauskas

Supervisor: Asst. Dr. Linas Petkevičius

Reviewer: Jr. Asst. Boleslovas Dapkūnas

Vilnius – 2022

Santrauka

Šiame darbe sprendžiama diferencialinių lygčių artinių paieškos problema. Įprastai matematiniuose modeliuose, kuriuose naudojamos diferencialinės lygtys, parametrai yra keičiami priklausomai nuo eksperimento. Šiame darbe pristatomas naujas artinių paieškos metodas, kuris apjungia diferencijuojamų architektūrų paieškos (DARTS) metodą ir simbolinį integravimą. Tokiu būdu lygčių artiniai išgaunami tam tikram parametrų intervalui. Modelio nereikia perskaičiuoti jei pasikeičia parametrai, priešingai nei tradiciniuose skaitiniuose metoduose (pvz. Eulerio metodas, baigtinių skirtumų metodas). Eksperimentai parodė, kad ši architektūra geba rasti artinius Maltuso augimo modelio lygčiai ir Michaelio-Menten kinetikos lygtims su tam tikrais parametrų intervalais.

Raktiniai žodžiai: matematinis modeliavimas, differencialinės lygtys, mašininis mokymas, simboliniai skaičiavimai

Summary

This work addresses the issue of finding approximations for differential equations. The mathematical models that use differential equations have various parameters which are changed depending on the actual experiment. A novel method of solving the modeled equations is presented, which takes the differentiable architecture search (DARTS) method and extends it with symbolic integration, where equations are solved for a particular interval of parameters. Unlike traditional numerical methods (e.g. Euler method, finite difference method), the model does not need to be rerun for each change in the parameters. Experiments show that the architecture is capable of approximating the Malthusian growth model equation and Michaelis–Menten kinetics equations for some particular parameter intervals.

Keywords: mathematical modeling, differential equations, machine learning, symbolic computation

CONTENTS

Introduction

The mathematical models that use differential equations have various parameters which are changed depending on the actual experiment, application, or area. Thus, a method is needed which would allow to easily change these parameters and evaluate the model in different points in time or space. Sometimes an analytical solution exists and it is possible to find one analytically, but more often such solutions are impossible to find. However, more complex models can be approximated with various numerical methods, which are most commonly iterative (e.g. finite element method, finite difference method). With these methods re-evaluation of their functions take a lot of time (e.g. using a fine grid with the finite element method). Therefore, a faster method is needed to recalculate the model at various points when parameters are changed.

There have been many methods of solving differential equations, such as a library aimed at automating the solution of partial differential equations using the finite element method [LW10], or a symbolic computation library solving differential equations symbolically [MSP+17]. Many of them have various useful strategies and solutions, although none offer the ability to generalize over a parameter interval.

Recent advances in the area of machine learning facilitated the creation of various frameworks and tools which allow easy construction of machine learning models [BFH+18; MAP+15; PGM+19]. These frameworks allow for quick iteration of the models to construct them more easily. Other software libraries also appeared which allow for symbolic computation, e.g. *SymPy* [MSP+17]. *SymPy* allows symbolic and numerical integration and differentiation, which can be of help in solving differential equations.

In machine learning, to solve a particular class of problems a specific architecture is needed. Whenever a change in the objective or data occurs, a change in the archtecture is also required (sometimes even a complete redesign). To help in solving this a novel method of differentiable architecture search (*DARTS*) has been recently proposed [LSY18]. These progresses in development of automatic differentiation, symbolic computation and automatic architecture search allows to combine these ideas.

Thus, the aim of this work – create a machine learning model based on automatic differentiation, symbolic computation, 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.

To reach the aim the objectives that have to be acomplished are listed below:

1. Analyze the suitability of automatic architecture search methods for solving differential equations.
2. Implement the proposed model with the automatic architecture search method for selected differential equations.

3. Evaluate the accuracy of the model under a defined metric.
4. Compare obtained approximations from the proposed model with another method.
5. Evaluate the usefulness of the model and its applicability to more general problems.

1. Background and Methods

The proposed model in this work combines the ideas of symbolic computation and the recently developed DARTS network [LSY18], and operates on differential equations. Thus, background material is presented on these topics.

1.1. Differential Equations

Differential equations are very common in science and engineering, as well as in many other fields of quantitative study, because what can be directly observed and measured for systems undergoing changes are their rates of change [Bri15]. They can describe the change in populations, heat movement, biochemical reactions and much more.

The theory of differential equations has important implications for many modeled systems, and methods for getting solutions for the equations are required. More often the case is that the solutions to these equations cannot be expressed in terms of elementary functions. Therefore, it is important to have methods that approximate or provide functions that have almost identical properties to the solution function.

1.2. Parametric Models

Parametric models consist of a finite set of parameters. Entire classes of methods using parametric models are devised with specific strategies for using these models to aid in solving specific problems.

Parametric models are used in machine learning, where they hold all the information about its outputs or predictions in a finite set of parameters [Dat20]. There are many methods of choosing parameters. One such method is maximum likelihood estimation, which provides the most likely parameter values given an input training data set. In neural networks, back propagation is applied to the training data to decide on the values of the weights that make up the model parameters. There is a tradeoff between a large number of parameters (whose calculation can become computationally expensive) and the ability to handle many data modeling problems.

1.3. Neural Networks

In the recent years with data becoming more abundant and processing power increasing, problems with data have become easier to solve. A well designed machine learning model can achieve state-of-the-art performance and for some tasks even outmach human performance [AJO+18; LSY18].

Although, designing good neural network architecture requires a good understanding of the area. It is possible to apply the conventional approaches to construct a simple supervised learning regression model that learns from data, but there is no guarantee the model performs well. Various models must be designed and tested, features manually selected, the dataset cleaned up and engineered for the specific task [LSY18; MAP+15; PGM+19].

1.3.1. Automatic Differentiation

Autograd is a technique used in machine learning to automatically compute gradients of specified functions. It is most commonly used in backpropagation, where the optimization algorithm expects some gradient of the loss with respect to the weights being learnt.

A prominent Python library that achieves these tasks is JAX [BFH+18]. JAX can automatically differentiate native Python and NumPy code. It can differentiate through a large subset of Python's features, including loops, ifs, recursion, and closures, and it can even take derivatives of derivatives of derivatives.

JAX has many useful functions for optimization and machine learning, four of them are presented here:

- `grad` – Creates a function that evaluates the gradient of the input function.
- `loss_and_grad` – Combines the initial function that computes the loss with the `grad` function, returning both the loss and the gradient of the input function.
- `vmap` – Vectorizing map. Creates a function which maps the input function over argument axes.
- `jit` – A function that sets up the input function for just-in-time compilation (JIT) with accelerated linear algebra.

In JAX these functions can be composed in any way, allowing one to take a simple Pythonic function, vectorize it, get the function that returns its gradient, and convert it to JIT compiled, all in a single line of code. This allows to use other tools to create our functions (e.g. SymPy, Sage, SciPy) and use them in JAX.

1.3.2. DARTS

With recent advances in neural architecture search, several highly performant methods were discovered. These automatically searched architectures have achieved highly competitive performance in tasks such as image classification and object detection. One such method is DARTS (Differentiable ARchiTecture Search) [LSY18].

DARTS searches for computational cells – building blocks of the final architecture. These cells can then be stacked to form a convolutional network, or recursively connected to form a recurrent network. 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

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{1}$$

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 operation is proposed as a softmax over all possible operations. 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,

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o' \in \mathcal{O}} \exp\left(\alpha_{o'}^{(i,j)}\right)} o(x) \tag{2}$$

where the the weights for each connection (i, j) are parametrized by a vector $\alpha^{(i,j)}$ of dimension $|\mathcal{O}|$. The actual architecture search is then defined as simply learning a set of variables $\alpha = \left\{\alpha^{(i,j)}\right\}$ as shown in Figure 1.

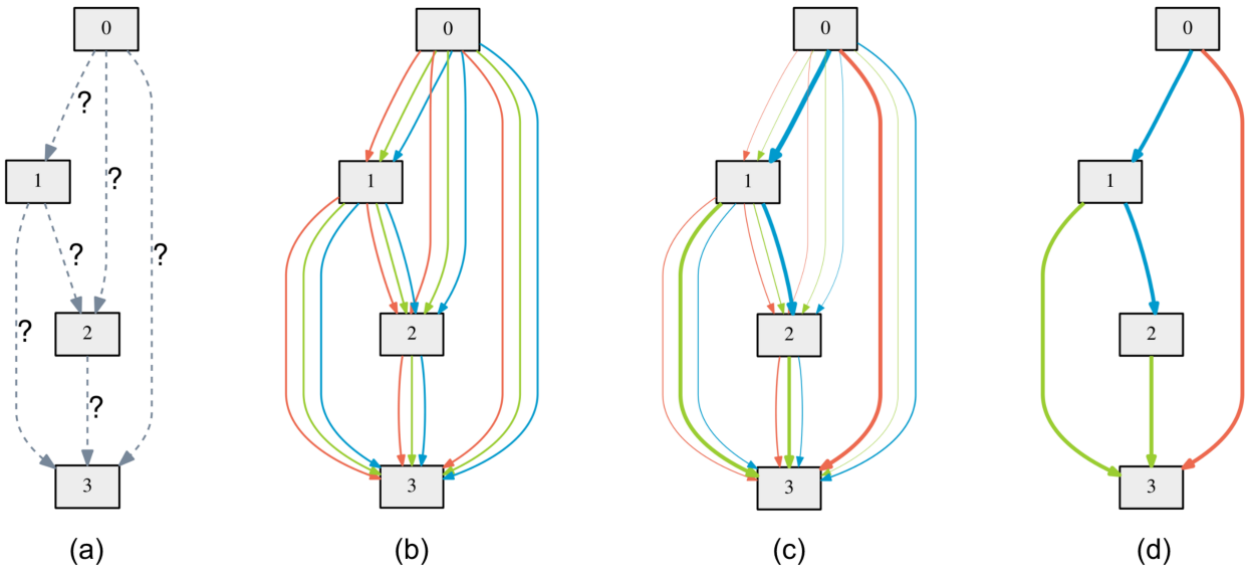

Figure 1. An overview of DARTS: (a) Operations on the edges are initially unknown. (b) Placing the mixed candidate operations on each edge, parametrized by a vector $(\alpha^{(i,j)})$ of weights. (c) Joint optimization of the network, eventually each edge contains a prominent operation. (d) Inducing the final architecture from the learned weights (i.e. $\underset{o \in \mathcal{O}}{\operatorname{argmax}}\ \alpha_o^{(i,j)}$) [LSY18].

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

$$o^{(i,j)} = \underset{o \in \mathcal{O}}{\operatorname{argmax}}\ \alpha_o^{(i,j)} \tag{3}$$

ending up with a single operation for each edge between the nodes.

This method can be exploited by providing many operations in $\mathcal{O}$ relevant to the problem being solved, e.g. convolution functions, parametric functions, etc.

The inner and outer optimization procedure from the work has been omitted for brevity.

1.4. Symbolic Computation

Computer algebra or symbolic computation is an area of computer science researching the algorithms and software that operates in and manipulates mathematical objects. Modern symbolic computation algorithms can, given the symbolic expression of a function, find derivatives, solve differential equations, compute integrals, and many more operations.

1.4.1. SymPy

SymPy is a Python library for symbolic computation [MSP+17]. It allows taking antiderivatives of symbolic expressions (with the `integrate` function) and allows transforming the symbolic expression into a pythonic function (with `lambdify`). For integration SymPy uses the Risch algorithm.

The functionality to convert a symbolic expression into a pythonic function (`lambdify`) is very powerful, because various models can be converted into Python format for manipulation by Python in different ways, loading from and saving to disk, use with other symbolic computation libraries, numerical libraries, etc.

2. Models

2.1. Informed Equation Learning

Learning mathematical models in an automated fashion is referred to as *equation learning* or *symbolic regression* [WJH+21]. The aims of informed equation learning is to learn compact and interpretable models in the form of concise mathematical equations from data. These types of models can understand the laws and physics that are featured by the dataset. To train such networks usually the mathematical expressions reside in the models themselves as a key element in training such as a network node, a loss term, a regularization term, etc.

Some informed equation learning models, with several artificial and real-world experiments from the engineering domain demonstrate learning interpretable models of high predictive power [Cra20b; WJH+21].

2.1.1. PySR

PySR is a Python symbolic regression library, which makes use of an algorithm based on regularized evolution and simulated annealing [Cra20b]. It is a very interpretable machine learning algorithm for low-dimensional problems: it searches the equation space to find algebraic relations that approximate a dataset.

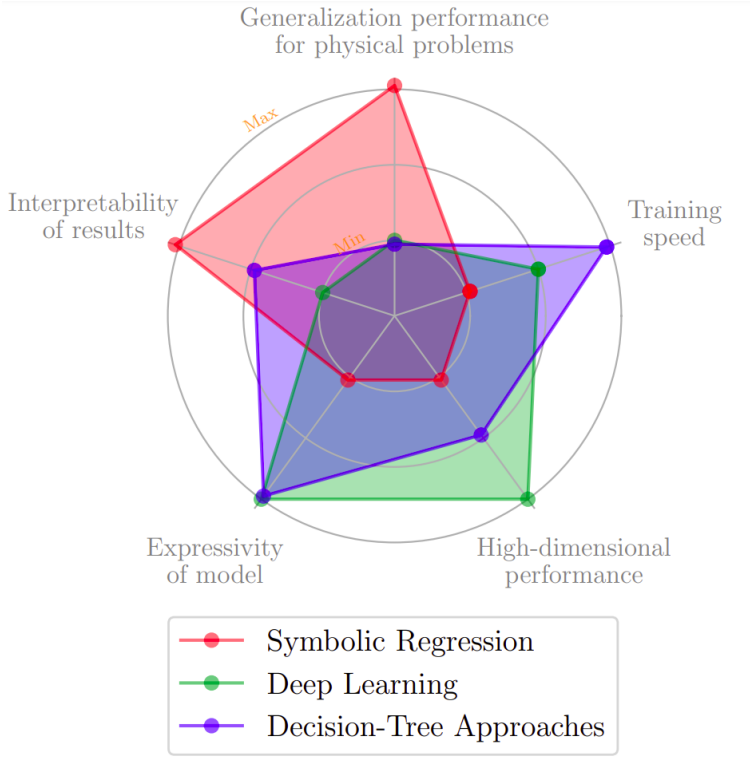

Figure 2. Various aspects of the trade-offs between machine learning (ML) techniques. Symbolic regression can robustly be applied to datasets with only $\lesssim$ 10000 data points, each with $\lesssim$ 10 parameters. On the other hand, it can provide analytic equations that are readily interpretable and generalizable [WTV+22].

The tradeoffs in available machine learning tools are illustrated along various dimensions in Figure 2. Deep learning tools like neural networks can handle very high dimensional inputs and large datasets, but are the least interpretable. Symbolic regression lies on the opposite side of this spectrum: as of today, SR can be applied to datasets with only $\lesssim$ 10,000 data points, each with $\lesssim$ 10 parameters. One must therefore simplify the problem or at times subsample the data in order to use SR on it.

PySR displays promising results when applied in the area of astrophysics (mass prediction of galaxy clusters, orbital mechanics, theoretical particle physics [Cra20a]).

3. Proposed Modeling Approach

This section presents a novel method of symbolic regression and describes the experiments executed with the proposed model, challenges faced and resolutions to problems. The full architecture of the model is presented in Section 4.

3.1. Proof of Concept

The initial proposed architecture for a proof of concept is presented below.

We take the Malthusian population model differential equation

$$\frac{dy(x)}{dx} = kx$$

where $y(x)$ is our unknown function, k is the variable parameter. The goal is to find a good enough approximation $\hat{y}$ for a specific interval $k \in [\mathrm{k_{low}}, \mathrm{k_{up}}]$.

The base model $\hat{y} = \hat{y}(x)$ is taken as a single cell of the DARTS network [LSY18]. We define the cell to have 4 nodes (1 input, 2 hidden, 1 output), therefore $x^{(0)} = x$, and $\hat{y} = x^{(3)}$, as shown in Figure 3. The intermediate nodes are computed as shown in Equation 2. For a proof of concept, the set $\mathcal{O}$ includes a zero function and four functions.

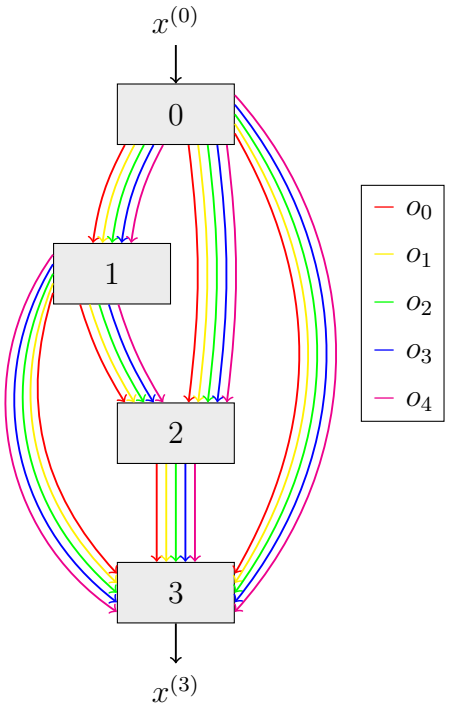

Figure 3. Proof of concept architecture with 3 nodes and 5 operations.

$$\mathcal{O} = \{o_0, o_1, o_2, o_3, o_4\}$$
$$o_0(z) = 0$$
$$o_1(z) = 1$$
$$o_2(z) = z$$
$$o_3(z) = -z$$
$$o_4(z) = e^z$$

Let $\mathcal{L}_{\text{eq.}}$ be defined as the transformed and squared equation

$$\mathcal{L}_{\text{eq.}} = \left(\frac{d\hat{y}}{dx} - kx\right)^2 \tag{4}$$

Define the loss $\mathcal{L}$ to be

$$\mathcal{L} = \int_{\text{k}_{\text{low}}}^{\text{k}_{\text{up}}} \mathcal{L}_{\text{eq.}}\, dk$$

where k_{low} and k_{up} ar the intervals for a specific problem $k \in [\text{k}_{\text{low}}, \text{k}_{\text{up}}]$ (e.g. $[1.4, 1.6]$) , and integration is done by SymPy's `integrate` method.

Initially, the model $\hat{y}$ can be constructed symbolically, lambdified using SymPy's `lambdify` method, passed to JAX's `vmap`, `loss_and_grad`, and `jit` functions (explained in Section 1.3.1). Then, the model is trained using gradient descent, minimizing loss $\mathcal{L}$ with respect to the weights α.

First tests showed that SymPy's `integrate` took a long time integrating simple equations (average of 6 min with $N = 3$, $|\mathcal{O}| = 5$, average time computed from values in Table 7). This posed a significant problem for the development of this idea. Two main solutions were tried to avert this problem:

- Changing the loss function from L2 (squared) to L1 (modulus) – this did not speed up the integration significantly.
- Changing the strategy for computing the mixed operation $\bar{o}$ to a simple product – this significantly sped up the integration process.

Therefore, the softmax function was changed to a simple product:

$$\bar{o}^{(i,j)}(x) = \left(\sum_{o \in \mathcal{O}} \alpha_o^{(i,j)} o(x)\right) + \beta \tag{5}$$

and an additional loss term added:

$$\mathcal{L}_{\text{prod.}} = \left(\sum_{o \in \mathcal{O}} \alpha_o^{(i,j)} - 1 \right)^2 \tag{6}$$

$$\mathcal{L} = \int_{\text{k}_{\text{low}}}^{\text{k}_{\text{up}}} \mathcal{L}_{\text{eq.}} \, dk + \gamma_{\text{prod.}} \mathcal{L}_{\text{prod.}} \tag{7}$$

where $\gamma_{\text{prod.}}$ is a hyperparameter.

These two changes replace similar functionallity that was carried out by the softmax function – all of the values after the softmax added up to one (characteristic of Equation 6) and allowing the weights to influence the operations (characteristic of Equation 5), and provide a speedup in integration.

The results indicate the architecture is working as expected, the training is shown in Figure 4.

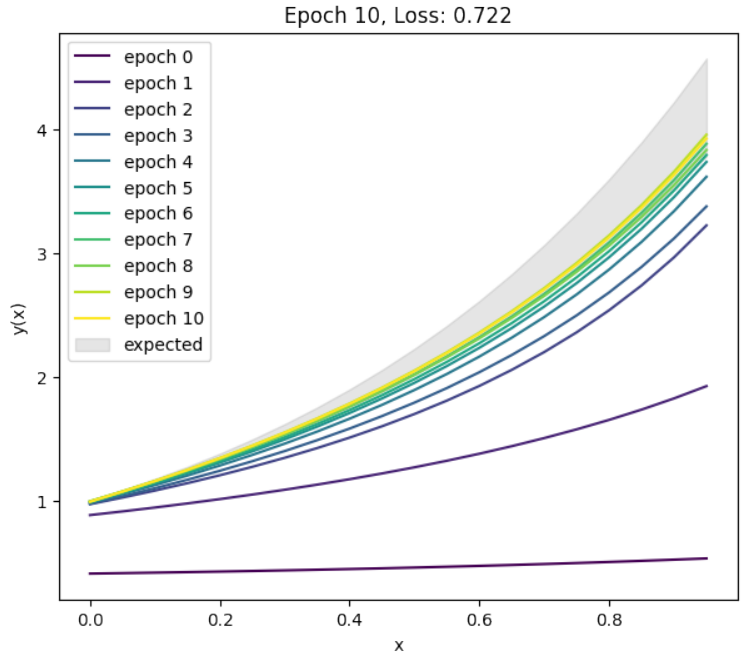

Figure 4. Details from the model training procedure. The lines represent the function $\hat{y}$ after each epoch. The light blue area in the graph represents the analytical solution $y(x) = e^{kx}$.

For the input x the training procedure sampled 1024 values for x uniformly from the interval $[0, 1]$.

Later, another improvement was made to further reduce integration times. The model $\hat{y}$, which was passed to SymPy's `integrate` function contained all of the mixed operation $\bar{o}$ symbols initially. By changing the procedure to contain unwrapped, symbolic placeholder values for $\hat{y}$ and $\frac{d\hat{y}}{dx}$ the integration time reduced to approx. 36 milliseconds (average time computed from values in Table 8).

The resulting weights can be seen in Table 1.

Table 1. $\alpha_{o_k}^{(i,j)}$ learned weights from the proof of concept model, $\beta = -0.3098$

(i,j)	o_0	o_1	o_2	o_3	o_4
$(0,1)$	0.2128	0.2036	0.3305	0.0830	0.1674
$(0,2)$	0.1618	0.1698	0.3511	0.0069	0.3146
$(0,3)$	0.1853	0.0005	0.4312	0.0093	0.3823
$(1,2)$	0.1371	0.2209	0.2895	0.1034	0.2492
$(1,3)$	0.3119	0.0002	0.3701	0.2540	0.0619
$(2,3)$	0.2873	0.0064	0.1621	0.2610	0.2851

These values can be visualized as shown in Figure 5

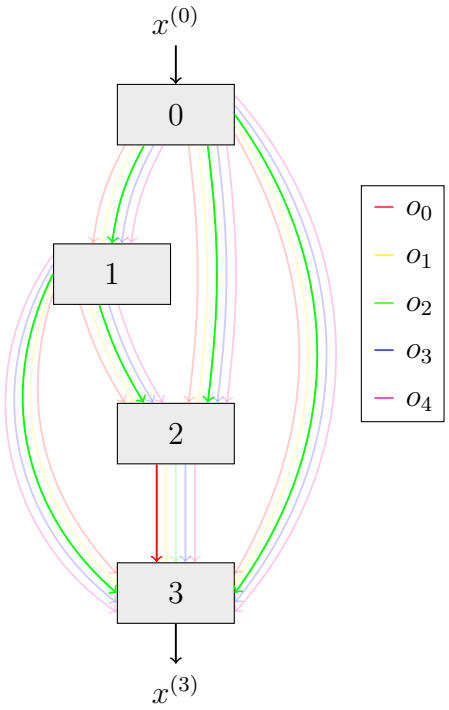

Figure 5. Trained model with each edge containing the most prominent operation. Faded lines show the operations, for which the weights α were not the maximum in their edge.

3.2. Weight Pruning

To simplify the model, after a specific amount of epochs N_e the model would be pruned. Several pruning strategies are presented below.

3.2.1. Whole Edge

- Take the first unpruned edge's vector of weights $\alpha^{(i,j)}$.
- Replace the vector with the largest operation like in Equation 3.

- 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.
- Mark the edge as pruned.
- Repeat after N_e epochs.

3.2.2. Edge Single Weights

- Take the first unpruned edge's vector of weights $\alpha^{(i,j)}$.
- Remove the weight with the lowest value and remove the operation of the discarded weight.
- If the edge contains only one weight, mark the edge as pruned.
- Repeat after N_e epochs.

3.2.3. Global Single Weights

- Find the lowest valued weight in an edge that isn't pruned.
- Remove the weight with the lowest value and remove the operation of the discarded weight.
- If the edge contains only 1 weight, mark the edge as pruned.
- Repeat after N_e epochs.

3.3. Further Improvements

With many iterative optimization algorithms the loss of the model can fluctuate during training. Therefore, during training in each iteration it was checked if the loss was the lowest out of all of the iterations. Then, the following parameters would be stored:

- Loss
- The symbolic model
- Symbolic weights & their values

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. The ways of adding them:

- Initial conditions can be added with a classic machine learning data approach – as datapoints

with an additional loss term.

- Boundary conditions can be added as special terms in the loss, depending on the usage of the parameters.

3.3.1. Initial Conditions

To add initial conditions, let the initial conditions be of the form:

$$y(x_0) = y_0$$

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

$$\mathcal{L}_{\text{init.cond.}} = (y_0 - \hat{y}(x_0))^2 \tag{8}$$

for each x_0. The loss then becomes

$$\mathcal{L} = \int_{\text{k}_{\text{low}}}^{\text{k}_{\text{up}}} \mathcal{L}_{\text{eq.}}\, dk + \gamma_{\text{prod.}}\mathcal{L}_{\text{prod.}} + \gamma_{\text{init.cond.}}\mathcal{L}_{\text{init.cond.}}$$

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

3.3.2. Boundary Conditions

For boundary conditions, assume the condition is in the form:

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

where x_0 is the boundary. These conditions can be added in a similar way how the initial conditions were added:

$$\mathcal{L}_{\text{bound.cond.}} = \left(\frac{\partial \hat{y}}{\partial x}(x_{\text{b}}) - g(x)\right)^2 \tag{9}$$

and similarly, the loss receives an additional term

$$\mathcal{L} = \ldots + \gamma_{\text{bound.cond.}}\mathcal{L}_{\text{bound.cond.}}$$

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

Additional constraints on the problem can be added in a similar manner.

3.4. Case Study – Biosensor Model

Biosensors are analytical devices used for detecting a chemical substance, they combine a biological component with a physicochemical detector [BIK21]. The action of catalytic biosensors is associated with substrate diffusion into biocatalytic membrane and its conversion to a product. The biosensor produces a signal when the analyte diffuses from the solution into the membrane.

3.4.1. Model

A part of the mathematical model is taken from the monograph [BIK21] and detailed below.

Let $x = 0$ represent the electrode surface, while $x = d$ is the boundary between the analyte source and the enzyme membrane, where d is the thickness of the enzyme layer.

Some definitions taken from the dimensionless model:

$$\hat{x} = \frac{x}{d}\,, \qquad \hat{t} = \frac{D_S\, t}{d^2}\,, \qquad \hat{S} = \frac{S}{K_M}\,, \qquad \hat{P} = \frac{P}{K_M}\,, \qquad \hat{D}_P = \frac{D_S}{D_P},$$

where $\hat{x}$ stands for the dimensionless distance from the electrode surface, $\hat{t}$ is the dimensionless time, $\hat{S}$ and $\hat{P}$ are the dimensionless concentrations of the substrate and the product, respectively, D_S and D_P are the diffusion coefficients, $V_{\max}$ is the maximal enzymatic rate and K_M is the Michaelis constant. The Michaelis constant K_M is the concentration of the substrate at which half the maximum velocity of the enzyme catalysed reaction is achieved.

The equations in the dimensionless coordinates $\hat{x}$ and $\hat{t}$ are expressed as follows:

$$\begin{aligned} \frac{\partial \hat{S}}{\partial \hat{t}} &= \frac{\partial^2 \hat{S}}{\partial \hat{x}^2} - \sigma^2 \frac{\hat{S}}{1+\hat{S}}\,, \\ \frac{\partial \hat{P}}{\partial \hat{t}} &= \hat{D}_P \frac{\partial^2 \hat{P}}{\partial \hat{x}^2} - \sigma^2 \frac{\hat{S}}{1+\hat{S}}\,, \qquad \hat{x} \in (0,1)\,, \qquad \hat{t} > 0\,, \end{aligned} \tag{10}$$

with initial conditions

$$\begin{aligned} &\hat{S}(\hat{x},0) = 0, \quad \hat{P}(\hat{x},0), \quad \hat{x} \in [0,1), \\ &\hat{S}(1,0) = \hat{S}_0, \quad \hat{P}(1,0) = \hat{P}_0, \end{aligned} \tag{11}$$

boundary conditions

$$\begin{aligned} &\hat{P}(0,1) = 0, \quad \left.\frac{\partial \hat{S}}{\partial \hat{x}}\right|_{\hat{x}=0} = 0, \\ &\hat{S}(1,\hat{t}) = \hat{S}_0, \quad \hat{P}(1,\hat{t}) = \hat{P}_0\,. \end{aligned} \tag{12}$$

where $\hat{S}_0 = \frac{S_0}{K_M}$, $\hat{P}_0 = \frac{P_0}{K_M}$, σ^2 is the dimensionless diffusion module,

$$\sigma^2 = \frac{V_{\max}\, d^2}{K_M\, D_S} = \alpha^2 d^2, \quad \sigma = \alpha d$$

3.4.2. Experiments

To model the concentration of the substrate $\hat{S}$ at a steady state using the proposed model, the equations are rewritten in the variables as specified in Section 3.1, 3.2, and 3.3:

$$\mathcal{L}_{\text{eq.}} = \left(\frac{\partial^2 \hat{y}}{\partial x^2} - k \frac{\hat{y}}{1+\hat{y}} \right)^2$$

$$\mathcal{L}_{\text{init.cond.}} = \left(y_0 - \hat{y}(0) \right)^2$$

$$\mathcal{L}_{\text{bound.cond.}} = \left(\frac{\partial \hat{y}}{\partial x}(0) \right)^2$$

from Equations 10, 11, and 12 respectively. The pruning strategy selected is Edge Single Weights, the initial condition value is selected to be $y_0 = 1$, parameters $x \in [0, 1]$, $N = 4$, $N_E = 10$, operation set $\mathcal{O} = \{o_i \mid i = 0, 1, ..., 7\}$:

$$o_0(z) = 0 \qquad o_4(z) = -z$$
$$o_1(z) = 1 \qquad o_5(z) = z^2$$
$$o_2(z) = z \qquad o_6(z) = z^4$$
$$o_3(z) = 1 + z \qquad o_7(z) = e^z$$

When k $\in [\mathbf{0.01}, \mathbf{0.1}]$ the network returns the weights presented in Table 2. The loss after the training procedure was 0.000018. The plot of the model function can be seen in Figure 6.

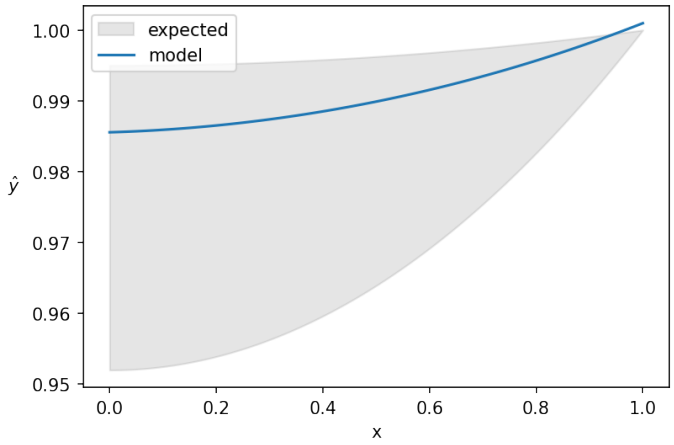

Figure 6. Learned model function (blue line) and expected model boundaries (light blue area), see Equation 13.

$\alpha_{o_4}^{(0,1)}$	0.2302	$\alpha_{o_1}^{(1,3)}$	0.1663	$\alpha_{o_7}^{(1,3)}$	0.0852	$\alpha_{o_4}^{(2,3)}$	0.2504
$\alpha_{o_4}^{(0,2)}$	0.2827	$\alpha_{o_3}^{(1,3)}$	0.1074	$\alpha_{o_0}^{(2,3)}$	0.1691	$\alpha_{o_5}^{(2,3)}$	0.0687
$\alpha_{o_1}^{(0,3)}$	0.2419	$\alpha_{o_4}^{(1,3)}$	0.1974	$\alpha_{o_1}^{(2,3)}$	0.1969	$\alpha_{o_6}^{(2,3)}$	0.0684
$\alpha_{o_4}^{(1,2)}$	0.1913	$\alpha_{o_5}^{(1,3)}$	0.1026	$\alpha_{o_2}^{(2,3)}$	0.0871	$\alpha_{o_7}^{(2,3)}$	0.0432
$\alpha_{o_0}^{(1,3)}$	0.1381	$\alpha_{o_6}^{(1,3)}$	0.1254	$\alpha_{o_3}^{(2,3)}$	0.1152	β	0.0293

Table 2. Learned weights. All other weights not specified here were pruned (= 0).

The learned model can be compared to the steady state equation specified in [BIK21]:

$$S_{ss}(x) = S_0 \frac{\cosh(\frac{\sigma}{d}\, x)}{\cosh(\sigma)} \tag{13}$$

which is shown as a light blue area in Figure 6, when $\sigma^2 \in [0.01, 0.1]$. The full expression after training is presented in Equation 14 in the appendix.

The model can be compered to Equation 13 by using the L2 distance:

$$\mathrm{L2} = \sum_{x \in X} \left(S_{ss}(x) - \hat{y}(x)\right)^2 ,$$

where the set X is a range of values from the interval $[0, 1]$ with a step size of 0.01:

σ^2	$L2$
0.01	0.0000418
0.02	0.0000082
0.03	0.0000008
0.04	0.0000191
0.05	0.0000623
0.06	0.0001300
0.07	0.0002214
0.08	0.0003362
0.09	0.0004736
0.10	0.0006332

Table 3. The model (Equation 14) L2 distance to the steady state equation (Equation 13).

When k $\in [\mathbf{10}, \mathbf{20}]$ the network returns the weights presented in Table 4. The loss after the training procedure was 1.108291. The plot of the model function can be seen in Figure 7.

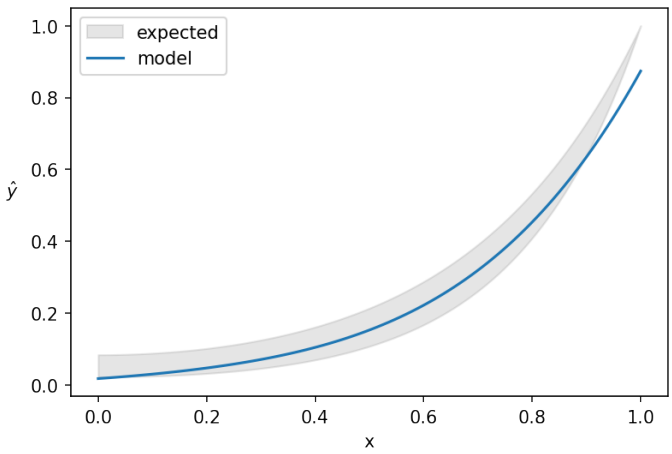

Figure 7. Learned model function (blue line) and expected model boundaries (light blue area), see Equation 13.

$\alpha_{o_2}^{(0,1)}$	0.0251	$\alpha_{o_2}^{(1,2)}$	0.1266	$\alpha_{o_2}^{(1,3)}$	0.1734	$\alpha_{o_2}^{(2,3)}$	0.1411
$\alpha_{o_4}^{(0,2)}$	0.1166	$\alpha_{o_3}^{(1,2)}$	0.1104	$\alpha_{o_3}^{(1,3)}$	0.0339	$\alpha_{o_3}^{(2,3)}$	0.0107
$\alpha_{o_0}^{(0,3)}$	0.0540	$\alpha_{o_4}^{(1,2)}$	0.1291	$\alpha_{o_4}^{(1,3)}$	0.1646	$\alpha_{o_4}^{(2,3)}$	0.2256
$\alpha_{o_2}^{(0,3)}$	0.1099	$\alpha_{o_5}^{(1,2)}$	0.1276	$\alpha_{o_5}^{(1,3)}$	0.1719	$\alpha_{o_5}^{(2,3)}$	0.1577
$\alpha_{o_5}^{(0,3)}$	0.1879	$\alpha_{o_6}^{(1,2)}$	0.1275	$\alpha_{o_6}^{(1,3)}$	0.1699	$\alpha_{o_6}^{(2,3)}$	0.1533
$\alpha_{o_6}^{(0,3)}$	0.5600	$\alpha_{o_7}^{(1,2)}$	0.1098	$\alpha_{o_7}^{(1,3)}$	0.0336	$\alpha_{o_7}^{(2,3)}$	0.0082
$\alpha_{o_0}^{(1,2)}$	0.1274	$\alpha_{o_0}^{(1,3)}$	0.1694	$\alpha_{o_0}^{(2,3)}$	0.1837	β	-0.1514
$\alpha_{o_1}^{(1,2)}$	0.1110	$\alpha_{o_1}^{(1,3)}$	0.0357	$\alpha_{o_1}^{(2,3)}$	0.0495		

Table 4. Learned weights. All other weights not specified here were pruned ($= 0$).

The model can be compered to Equation 13 by using the L2 distance:

σ^2	$L2$
1	0.284436
2	0.152818
3	0.090343
4	0.056633
5	0.036828
6	0.024486
7	0.016468
8	0.011111
9	0.007467
10	0.004972

Table 5. The model (Equation 15) L2 distance to the steady state equation (Equation 13).

Although, the monography presents a different steady state equation for large values of σ [BIK21]:

$$S_{ss}(x) = S_0 + \frac{V_{\max}(x^2 - d^2)}{2D_S},$$

the model does not seem to approximate this equation, but rather the first one, in Equation 13.

4. Description of the Architecture

A differential equation is chosen with a single variable constant k, which can be written in the form

$$f(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, ..., k) = 0$$

with initial conditions (x_0, y_0) in the form

$$y(x_0) = y_0$$

boundary conditions in the form

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

and an interval for the constant is selected $k \in [\text{k}_\text{low}, \text{k}_\text{up}]$.

The model for the solution of the equation $\hat{y}$ is taken as a single cell of the DARTS network. Define the model input being x, N_x numbers sampled uniformly from a selected domain $x \in [\text{x}_\text{low}, \text{x}_\text{up}]$ and the cell to have N nodes. Each node is computed as a sum of the mixed operations of the previous ones

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

for each $i = 1, 2, ..., N$.

Each edge connecting the nodes is a mixed operation

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

where the set $\mathcal{O}$ is the operatiion set containing a finite number of base functions.

The base network loss is the base equation L2 loss, with variable y replaced with the model $\hat{y}$:

$$\mathcal{L}_\text{eq.} = \left(f(x, \hat{y}, \frac{d\hat{y}}{dx}, \frac{d^2\hat{y}}{dx^2}, ..., k)\right)^2$$

$$\mathcal{L} = \int_{\text{k}_\text{low}}^{\text{k}_\text{up}} \mathcal{L}_\text{eq.}\, dk + \gamma_\text{prod.}\mathcal{L}_\text{prod.} + \gamma_\text{init.cond.}\mathcal{L}_\text{init.cond.} + \gamma_\text{bound.cond.}\mathcal{L}_\text{bound.cond.}$$

with losses $\mathcal{L}_\text{prod.}$, $\mathcal{L}_\text{init.cond.}$, and $\mathcal{L}_\text{bound.cond.}$ as defined in Equation 5, Equation 8, and Equation 9 respectively.

Integration done by SymPy's `integrate` method and passed to JAX using SymPy's `lambdify` method. JAX is responsible for computing the gradients automatically, as explained in Section 1.3.1.

A pruning strategy is chosen from Section 3.2. The input of the network is $x = x^{(0)}$ and the output is $\hat{y} = x^{(N)}$.

The network loss is minimized with respect to the weights α and β

$$\min_{\alpha,\beta} \mathcal{L}$$

using gradient descent, or any other optimization algorithm.

An epoch of training consists of N_x iterations of taking the inputs, calculating the loss and applying the weight updates. After N_e epochs the network is pruned using the selected pruning strategy and training continues. After all edges have been pruned the network is in its final state and training stops.

Results and Conclusions

Achieved results:

- The proposed model was defined and implemented with the automatic architecture search method for selected differential equations, see Section 4.
- The accuracy of the biosensor model was evaluated and compared to the steady state equation with the L2 distance, see Section 3.4.

Automatic architecture search methods are suitable for solving differential equations, as shown by the biosensor experiment. A novel machine learning architecture was implemented combining the ideas of differentiable architecture search and symbolic computation. Summarized results can be seen in Table 6.

Table 6. Summarized results from the biosensor model (Section 3.4.2)

k	Training Duration	Min. Loss	Avg. L2 Distance
$[0.01, 0.1]$	113.4 s	0.000018	0.000193
$[10, 20]$	124.5 s	1.108291	0.068556

Therefore, the following conclusions can be drawn:

- The proposed architecture shows promising results when applied on the proof of concept model.
- The proposed architecture can accurately find an approximation for the biosensor model with the specified parameter interval.
- For the applicability of the architecture to different problems the architecture should be fine-tuned and further improved.

References

[AJO+18] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: a survey. *Heliyon*, 4(11):e00938, 2018. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2018.e00938. URL: https://www.sciencedirect.com/science/article/pii/S2405844018332067.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, et al. JAX: composable transformations of Python+NumPy programs, version 0.2.5, 2018. URL: http://github.com/google/jax.

[BIK21] Romas Baronas, Feliksas Ivanauskas, and Juozas Kulys. *Mathematical Modeling of Biosensors*. 2021-01. ISBN: 978-3-030-65504-4. DOI: 10.1007/978-3-030-65505-1.

[Bri15] The Editors of Encyclopaedia Britannica. "differential equation". In *Encyclopedia Britannica*. 2015-04-21. URL: https://www.britannica.com/science/differential-equation.

[Cra20a] Miles Cranmer. Pysr showcase. 2020. URL: https://astroautomata.com/PySR/#/papers (visited on 2022-05-26).

[Cra20b] Miles Cranmer. Pysr: fast & parallelized symbolic regression in python/julia, 2020-09. DOI: 10.5281/zenodo.4041459. URL: http://doi.org/10.5281/zenodo.4041459.

[Dat20] DataRobot. What are parametric models? https://www.datarobot.com/blog/what-are-parametric-models/, 2020. Accessed: 2022-05-18.

[LSY18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018. arXiv: 1806.09055. URL: http://arxiv.org/abs/1806.09055.

[LW10] Anders Logg and Garth N. Wells. DOLFIN. *ACM Transactions on Mathematical Software*, 37(2):1–28, 2010-04. DOI: 10.1145/1731022.1731030. URL: https://doi.org/10.1145/1731022.1731030.

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, et al. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/. Software available from tensorflow.org.

[MSP+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017-01. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, et al. Pytorch: an imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[WJH+21] Matthias Werner, Andrej Junginger, Philipp Hennig, and Georg Martius. Informed equation learning. *CoRR*, abs/2105.06331, 2021. arXiv: 2105.06331. URL: https://arxiv.org/abs/2105.06331.

[WTV+22] Digvijay Wadekar, Leander Thiele, Francisco Villaescusa-Navarro, J. Colin Hill, et al. Augmenting astrophysical scaling relations with machine learning : application to reducing the sz flux-mass scatter, 2022. DOI: 10.48550/ARXIV.2201.01305. URL: https://arxiv.org/abs/2201.01305.

Appendix 1

Integration Times

Table 7. Integration time when the model passed to SymPy's `integrate` method contains $\hat{y}$ and $\frac{d\hat{y}}{dx}$ substituted with the full expression of the model, computation shown in Section 3.1.

Run #	Integration time (s)
1	451.806
2	688.334
3	371.367
4	375.305
5	570.060
6	367.513
7	430.686
8	409.090

Table 8. Integration time when the model passed to SymPy's `integrate` method contains $\hat{y}$ and $\frac{d\hat{y}}{dx}$ as symbolic placeholder values (only the loss function, as shown in Equation 4).

Run #	Integration time (s)
1	0.047
2	0.036
3	0.035
4	0.037
5	0.033
6	0.033
7	0.038
8	0.034

Appendix 2

Biosensor Model Results

Trained biosensor model result when $k \in [0.01, 0.1]$.

$$\begin{aligned}\hat{y} = {} & 0.000573984299538078x^4 + 0.00935040597191916x^2 + 0.0321852873662438x + \\ & + 0.857084291055799 + 0.0432434156537056e^{-0.238648342095613x} + \\ & + 0.0852382481098175e^{-0.230192229151726x}\end{aligned} \tag{14}$$

Trained biosensor model result when $k \in [10, 20]$.

$$\begin{aligned}\hat{y} = {} & 0.559994582601684x^4 + 0.187962920599667x^2 + 0.11935839572503x + \\ & + 0.000368218560399693(2.29131403768922 \cdot 10^{-7}x^4 + 0.000363628759623082x^2 - \\ & - 0.514262143846239x + 0.496010043484023 \exp(0.0251127295196056x) + 1)^4 + \\ & + 0.00772696445404676(2.29131403768922 \cdot 10^{-7}x^4 + 0.000363628759623082x^2 - \\ & - 0.514262143846239x + 0.496010043484023 \exp(0.0251127295196056x) + 1)^2 + \\ & + 0.025510901320589 \exp(0.0251127295196056x) + 0.0102351206552118 \cdot \\ & \cdot \exp(5.07225176280998 \cdot 10^{-8}x^4 + 8.04960204785666 \cdot 10^{-5}x^2 - 0.113841534716086x + \\ & + 0.109801091253757 \exp(0.0251127295196056x)) - 0.0378412777270404\end{aligned} \tag{15}$$