Universal Differential Equations Applied to Bioprocesses

by

Jyler Menard

A research project report
presented to the University of Waterloo
in fulfillment of the
research paper requirement for the degree of
Master's of Mathematics
in
Computational Mathematics

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this work. This is a true copy of the work, including any required final revisions, as accepted by my examiners.

I understand that my report may be made electronically available to the public.

Abstract

Biomanufacturing is method of manufacturing using biological organisms to produce products and molecules for a wide-range of applications. Manufacturing of pharmaceutical products and precursors alone has generated more than $90 billion in sales in 2017, with Monoclonal Antibodies (mAb) being the primary product. Representing 60% of mAb biomanufacturing cell cultures, and 84% of approved products in the period of 2015-2018, Chinese Hamster Ovary (CHO) cells, and other mammalian cells are becoming the dominant organism used for biopharmaceutical production.

The problem is that many biological organisms being used, and especially mammalian cell cultures, are not sufficiently understood to allow for accurate prediction and control using mathematical models. Moreover, models are based on mechanisms and processes understood at lab-scale, with volumes of $3.5L$ to $20L$ whereas industrial volumes are frequently on the order of $20,000L$. At industrial-scale new processes and mechanisms become salient that are not captured by traditional first-principles models. Furthermore, it is not clear how to incorporate many measured quantities into these first-principles bioprocess models; e.g. pH, temperature, osmolality, and others.

Machine learning methods frequently underuse first-principles or mechanistic knowledge about systems, primarily using them to direct feature engineering and feature selection. Neural ordinary differential equation architectures, where a neural network learns a dynamical system, are becoming one way to combine both neural networks and mechanistic models to produce hybrid predictive models. These hybrid models incorporate the flexibility of machine learning with the mechanistic knowledge of first-principles-based models. One area that has the opportunity to benefit immensely from a hybrid modelling approach is biomanufacturing process control and prediction, where the jump from lab-scale models to industrial-scale processes introduces processes and variables unaccounted for in models developed at the lab-scale. Here I demonstrate the application of hybrid modelling with a simple toy bioprocess model, and discuss other approaches of combining machine learning and mechanistic modelling present in the literature.

Acknowledgements

I would like to thank Dr. Brian Ingalls for offering his time, thoughts, and support throughout this project and report.

Table of Contents

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**

1.1 Biomanufacturing . . . 2

1.1.1 Complexities . . . 3

1.2 Combining Machine Learning and Mechanistic Modelling . . . 5

1.2.1 Sequence prediction with neural networks . . . 5

1.2.2 Sparse Identification of Nonlinear Dynamics (SINDy) . . . 6

1.2.3 System-informed Deep Learning . . . 8

1.2.4 Neural Differential Equations . . . 9

1.2.5 Hybrid Neural Differential Equations . . . 14

1.2.6 Hybrid models in process modelling . . . 16

2 Results **21**

2.1 Hybrid Modeling from Simulated Data . . . 21

2.1.1 Haldane Model . . . 21

3 Conclusion **34**

References **36**

List of Figures

1.1 Graphical representation of a recurrent neural network (RNN). x_t is the initial input. h_i is the hidden state. h_1, the first hidden state input is traditionally chosen to be zeros, or random. $\hat{y}_i$ is the predicted output. To extrapolate off of the initial input, the predicted output is fed back into the recurrent neural network as an input. 5

1.2 Graphical representation of the three broad types of hybrid models. On the left we have the two serial hybrid models (ab) where the mechanistic model and machine learning model feeds into each other; i.e. the output of a neural network is used as input to a DE system (a), or vice-versa (b). On the right we have parallel hybrid models, where the machine learning model's output is used to estimate the residual between the DE model's output and the true observations. 18

1.3 Timeline of key hybrid modelling works in the biomanufacturing setting. Note the break between years 1997 and 2016. Works referred to are as follows: 1992, [34]; 1994, [44]; 1997, [47]; 2016, [49]; 2017, [43]; 2019, [32]; 2021, [23, 14] 20

2.1 The two activation functions compared in this report. Dash-dotted line corresponds to the leaky ReLu of $\max(0.01x, x)$, while the solid line corresponds to $\tanh(x)$. 26

2.2 Representative results of integrating the true Haldane model Eq. 2.1, the assumed model Eq. 2.2, and the hybrid model Eq. 2.3, using initial conditions that were in the validation set (column a) and neither the training nor validation set (column b). Initial conditions used are specifically listed in Table 2.4. 28

2.3 Shown is the average loss over 10 trajectories against the optimization step of an ADAM optimizer. Note that initially the loss value begins at ≈ 6000, and begins to converge to ≈ 5. The blue (orange) line represents the average loss over trajectories sampled from the validation (training) set. The amplitude of the noise added to the simulated trajectories is $0.01 \cdot \mathrm{sd}(\vec{x}) \cdot \mathcal{N}(0, 1)$. . . . 30

2.4 Representative results of integrating the true Haldane model Eq. 2.1, the assumed model Eq. 2.2, and the hybrid model Eq. 2.3, using initial conditions that were not used in the training set for the hybrid model. Left and right columns are two different initial conditions. The left (right) column has begins with initial condition close to (far from) the training set. The results of the hybrid model in the right column are worse than in the left. . . 31

2.5 Representative results from the validation set when 5% noise is added (column a) and 10% noise is added (column b). With higher noise levels, the hybrid model seems to exhibit more oscillations. 33

List of Tables

2.1 Values used for parameters in the true Haldane model, along with distributions used for sampling initial conditions when building a training and validation set. Both the training and validation sets are of size N, but have distinct trajectories and initial conditions. 24

2.2 Neural network architecture for the final network used. 25

2.3 Comparison of different model architecture choices. One "epoch" corresponds to the ADAM optimizer iterating through the entire training data set once. While increasing number of layers increases training time, it tends to decrease the loss value on the validation set. Note that a leaky relu activation function seems to require more time per epoch compared to the same architecture with different activation functions. As discussed above, this might be because the tanh function is smooth while the leaky relu function is not. 27

2.4 Initial conditions for results shown in Figure 2.2. 29

2.5 Parameters estimated from the training data in each case of levels of added noise. 29

Chapter 1

Introduction

Mathematical models are frequently used to describe, predict, and control systems. Yet, they just as frequently fail to do so accurately or precisely. When they do so, we have *model mismatch*, or the discrepancy between how a mathematical model may predict a system to behave, and how it actually behaves. Model mismatch can be due to a number of factors some of which are on the modelling side of the mismatch, and some are due to experimental error and limitations. On the modelling side, mismatch is usually because of the model simply ignoring essential aspects of a system; excluding drag force when modelling projectile motion, resulting in a perfect parabola is one such example.

There are many reasons why essential aspects of a system might be missing, such as ignorance of the system, attempting to simplify the model, or not being able to translate what we measure into our model's state variables. In the first case, the system may have many processes where we do not know the exact mechanisms or interactions of the processes. In the second case, we may want a model that is relatively simple to use and simulate, and so we deliberately remove or approximate aspects of a model. In the third case, monitoring the system of interest may result in measuring variables connected to the state variables in an unclear way. Each case can contribute to model mismatch.

Machine learning, or the process of algorithmically approximating a function using data, may provide a method of augmenting differential equation models to reduce model mismatch. In particular, in the cases of system ignorance, and measurement variables being different from state variables, machine learning may be especially useful. This is because, if there is available data, machine learning offers a flexible method to determine complex non-linear relationships, that once trained is easy to use.

Yet, machine learning frequently ignores previously gathered mechanistic knowledge.

Instead, domain knowledge is only used to engineer inputs to the model, thereby providing it with more useful inputs. For example, transforming coordinates from Cartesian to polar, or using square footage instead of total length and width for a model predicting prices of houses. In this way, feature engineering enables some usage of previous domain knowledge. But, it rarely allows full usage of mathematical models, or clear theories, laws, or predictions.

Combining machine learning and mechanistic modelling may then offer a way of reducing model mismatch. The machine learning aspect of the model provides flexibility, while the rest of the model provides explicit encoding of a mechanistic principle. As a result, the model can use previously gathered mechanistic knowledge and adapt to factors driving model mismatch based on available data when needed.

This report will review methods of combining machine learning and mechanistic modelling. In addition, hybrid methods based on neural differential equations will be applied to simulation-based experiments. However, we first introduce biomanufacturing, an area where model mismatch can be caused by several complexities, and hybrid modelling may be of particular value.

1.1 Biomanufacturing

Biomanufacturing is the usage of (usually modified) living cellular organisms to manufacture products of interest, like insulin, penicillin, vaccines, antibodies, and more. The essence of it is to take a culture of cells, and grow them in a vat. It is an enormous industry with sales revenues breaking 123 billion for monoclonal Antibodies alone [50], while also being an industry producing wide-scale societal benefits in other health domains and in sustainability. Artemisin, a therapeutic for malaria, is now able to be produced at scale and cheaply, thanks in large part to large-scale biomanufacturing of precursor molecules [9]. Vaccines are manufactured in this way (though not mRNA-based vaccines like Pfizer and Moderna's [41]), as are dairy-based products and fermentation processes for brewing beers. Biomanufacturing is expected to continue increasing in importance for numerous reasons: (1) more medicine-related products are being produced in this way, (2) it is a relatively sustainable way of producing products, (3) because of the Covid-19 pandemic, countries such as Canada are making large investments in biomanufacturing research, development, training, and facilities [1].

The process of developing, predicting and controlling a biomanufacturing process is difficult and extremely important. In 2004 the FDA outlined regulations on the biomanufacturing of pharmaceutical products requiring both the product and the process of producing

it to pass stringent quality assessments before approval [51, 3]. Moreover, trying a given process at industrial-scale can be time-consuming and resource-intensive as days to weeks are required for some processes. Hence mathematical modelling of the cellular metabolism and growth processes is done in an attempt to design optimal process experiments and control them; sometimes dubbed a Digital Twin [52], the process of modelling biomanufacturing processes has become paramount in determining optimal process conditions, in order to control the process, and determine short-term scheduling and staffing, all to maximize process performance and quality.

Furthermore, prototyping, or proof-of-concepts are done at lab-scale in vats of a $\approx 3L$ and up to (but less commonly) $\approx 15L$. Yet, industrial-scale production is frequently done with tens of thousands of litres with vats of volumes up to $100,000L$. During this scale-up from lab to industrial-scale, many complexities arise prohibiting optimal production of the product of interest. Because scale-up frequently prevents lab-successes from easily being manufactured at scale, it has become dubbed a valley of death [6].

1.1.1 Complexities

Scale-up brings about new complexities, and intensifies others. One category of complexities are transfer phenomena: heat-transfer, gas-transfer, substrate- and product-transfer. Another complexity emerges from having poorly characterized cellular systems. And the third complexity is the difficulty of measuring directly quantities of interest in a vat. Each of them complicate the path from lab-success to industrial-scale production. Moreover, models must be able to accommodate these complexities, otherwise they become of limited use [52].

The jump from lab-scale reactors of $< 10L$ to $10,000 - 100,000L$ allows for transport phenomena to become important to the efficiency and productivity of a given process. Thermal energy released by cells undergoing regular metabolic processes heats the surrounding local environment, and – analogously to compost piles heating up – can increase the local temperature. Cells then respond to the temperature change in multiple ways, including initiating stress responses or undergoing metabolic changes, modifying the production rate of products-of-interest. Along with local temperature changes, cells consuming substrate (their source of nutrients) will decrease the local availability of substrate, if it decreases too much the cells may switch to another nutrient source, or stop growing, changing the rate of production of the product-of-interest. In addition, local gas concentrations become a concern when aerobic metabolic processes are desired. The aerobic process requires oxygen, and carbon dioxide is generated as a byproduct. In the absence of oxygen,

cells may slow down their growth and convert to fermentation, releasing lactic acid. Both lactic acid from fermentation and carbonic acid from dissolved carbon dioxide will change the local pH, affecting cells' growth rate and production of POI.

Along with transport phenomena complexities, cellular metabolism is another complexity. This is due to cellular organisms frequently being poorly characterized, leading to models ignoring important processes, or being unaware of them in the first place. Moreover, model parameters are also underspecified, as finding the correct parameter values given some data yields intervals of parameter values for a given confidence interval. Because the model dynamics are frequently non-linear, inaccuracies in model parameters lead to large prediction inaccuracies.

Third primary complexity is that measuring variables of importance during a biomanufacturing process is difficult. Instead, other measurements are conducted. Frequently the state variables being modelled – substrate concentration, biomass, product concentration, amino acid concentration(s) – cannot be directly measured during a manufacturing process. Rather, measurements of pH (how acidic the environment is), gas concentration (usually dissolved oxygen), optical density (estimate of biomass), osmolality (concentration of particles dissolved in solution), vessel pressure, temperature, and agitation rate are available throughout the process. Despite these measurements being available, and related to or affecting metabolic processes, incorporating them into models is not always done. E.g. temperature affects the metabolic rate of cells, and is affected by cells undergoing metabolism, yet relating temperature to the primary state variables is not clear, so the data is underused.

Each complexity along the path to commercial-scale, industrial biomanufacturing provides more room for model mismatch. Transport phenomena and the increased likelihood of heterogeneity in the reactor change the rate parameters and other dynamics involved when using mathematical models to predict product production. Unspecified or undercharacterized cellular processes become more problematic at large-scale where model mismatch can become more prominent. Being unable to directly measure many state variables or process attributes directly undermines the utility of mathematical models relying on those very concentrations.

1.2 Combining Machine Learning and Mechanistic Modelling

1.2.1 Sequence prediction with neural networks

Before going into methods of combining machine learning with mechanistic understanding of a system, first we describe how sequence prediction is already done using neural networks.

Sequence prediction corresponds to the following problem. Given a sequence $\{x_1, x_2, \ldots, x_n\}$ usually represented using a vector $\vec{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix}^T$ predict x_{n+1} the next element in the sequence. Predicting the next element in the sequence with neural networks requires a specific type of neural network architecture: the recurrent neural network, and variants built from that.

Recurrent neural networks (RNNs) have two different input/output types: hidden states, usually denoted by h_i; and, the normal inputs, x_i. The major difference from a normal neural network architecture is the outputted hidden state is recursively updated at each time step. This enables the neural network to store information about previous elements of the sequence in a compact 'memory', in the form of the hidden state as depicted in Figure 1.1. The RNN is extensively used, having seen success in handwriting synthesis and text prediction [18], and other time-series prediction tasks such as weather forecasting [25].

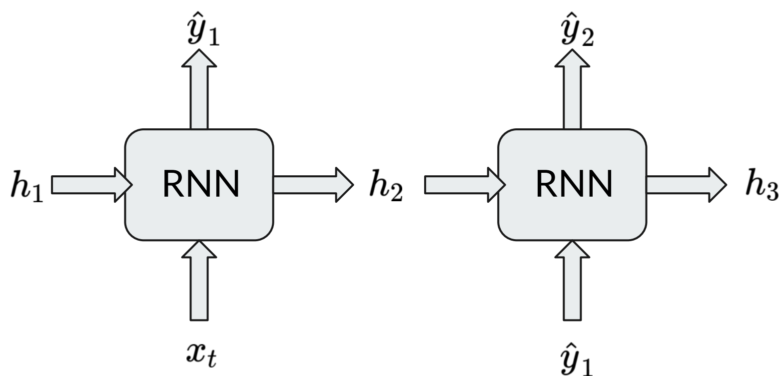

Figure 1.1: Graphical representation of a recurrent neural network (RNN). x_t is the initial input. h_i is the hidden state. h_1, the first hidden state input is traditionally chosen to be zeros, or random. $\hat{y}_i$ is the predicted output. To extrapolate off of the initial input, the predicted output is fed back into the recurrent neural network as an input.

One limitation of RNNs is their memory state is short-term, prioritizing the recent past. A variant architecture that is frequently used is the Long Short-Term Memory (LSTM) network. The LSTM resolves the short-term memory problem of RNNs, and is also effective for time-series forecasting [25].

Both RNNs and LSTMs suffer from two limitations that may be addressed with neural differential equations, as described below. The first is that they are forced into fixed time-steps. The time-stepping is implicit to the RNN because it is implicit to the sequence we are providing the RNN with. However, many problems are irregularly sampled, or we need more continuous time-series forecasting for control or scheduling purposes. The second limitation is the ambiguity regarding incorporating domain knowledge, or mechanistic knowledge, into RNNs – feature engineering and selection are the primary methods of trying to do so. A third limitation is that many systems of interest have behaviour with at least continuous first derivatives, and while neural networks can approximate any continuous function, RNNs cannot easily give results that are smooth because we do not require them to explicitly learn the dynamics governing some behaviour. The sections below will discuss different methods of incorporating mechanistic knowledge into a machine learning setting to build hybrid models.

1.2.2 Sparse Identification of Nonlinear Dynamics (SINDy)

Using machine learning and leveraging sparse encoding for determining differential equation models has been shown to be a successful method of determining governing equations for a dynamical system [7, 46, 28, 20, 29]. The method works by setting up a regression problem with an additional term inducing the regression coefficients to become equal to zero [45], Eq. 1.1. By a clever choice of model, pushing most of the regression coefficients to be zero, and leaving a small subset of coefficients non-zero, can identify a dynamical system approximating the data.

We wish to minimize

$$\sum_i (y_{\text{true},i} - \hat{y}_i)^2 + \lambda \sum_j ||\theta_j||_1 \tag{1.1}$$

where θ_j are the regression coefficients, $\hat{y}_i$ is the model's prediction at time t_i, and $y_{\text{true},i} = \frac{dx}{dt}(t_i)$. By choosing the model to be

$$\hat{y} = \sum_{j=1}^{M} \theta_j f_j(x(t)) \tag{1.2}$$

with each $f_j(x)$ a different function of x, then minimizing Eq. 1.1 with respect to θ_j results in a sparse model of non-zero coefficients. These non-zero coefficients select the functions $f_j(x)$ best approximating the dynamical system yielding the observed data. λ is a hyperparameter controlling the cost of having non-zero coefficients. A very large λ yields a model with very few coefficients, whereas $\lambda \approx 0$ may yield a model with many non-zero coefficients. The optimal choice of λ requires iterating the process of regression on a training data set, and testing on a validation set.

In this way, a lasso regression can yield a sparse model, thereby providing a data-driven method of identifying a mechanistic model. In the biological realm, this method has been used to reconstruct dynamics of the Michaelis-Menten equation, a yeast glycolysis model, and a *B. subtilis* model, albeit using simulated data from those models, and not observed experimental data [29]. Because this method seems successful at identifying model dynamics from data, it has been dubbed Sparse Identification of Nonlinear Dynamics (SINDy).

SINDy procedure

The SINDy method assumes that we have time-series observations $x_i(t_j)$. From those time-series observations, we must estimate the time-derivative at the sampled time points using a numerical differentiation technique. Once we do that we will have approximate observations of the rate of change of the system at time points t_j, $\frac{dx_i}{dt}(t_j)$. Since we have the (approximate) time-derivative for each dimension of our system, we have a matrix

$$\frac{d}{dt}X(t) = \begin{bmatrix} \frac{dx_1}{dt}(t_1) & \frac{dx_1}{dt}(t_2) & \dots & \frac{dx_1}{dt}(t_M) \\ \frac{dx_2}{dt}(t_1) & \frac{dx_2}{dt}(t_2) & \dots & \frac{dx_2}{dt}(t_M) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dx_N}{dt}(t_1) & \frac{dx_N}{dt}(t_2) & \dots & \frac{dx_N}{dt}(t_M) \end{bmatrix} = \left[\frac{dx_i}{dt}(t_j)\right] \tag{1.3}$$

where N is the number of dimensions for our dynamical system, and M is the number of time-points we sampled at. The next step is to build a 'dictionary', which contains the matrix of points put through some functions. The family of functions selected and included in the 'dictionary' is somewhat arbitrary, and depends on what functions the user supposes may be relevant to describing the observed dynamics. The choice of functions to use in the dictionary is one of the major bottlenecks of the SINDy method.

As an example, taking one of the rows of Eq. 1.3 we have that

$$\frac{dx_1}{dt} = \begin{bmatrix} 1 & \vec{x}^T & \vec{x}^{T,2} & \vec{x}^{T,p} & \sin(\vec{x}^T) & \dots \end{bmatrix} \vec{\theta}_1 \tag{1.4}$$

where, $\vec{x}^p$ is a multinomial raised to power p; e.g. for $\vec{w} \in \mathbb{R}^2$, $\vec{w}^2 = w_1^2 + w_1 w_2 + w_2^2$; $\sin(\vec{x}^T) = \begin{bmatrix}\sin(x_1) & \sin(x_2)\end{bmatrix}$. $\vec{\theta}_1$ is the vector of coefficients for the first dimension of the dynamical system. Minimizing Eq. 1.1 to find the optimal set of coefficients $\vec{\theta}_i$ picks out the functions best approximating the derivative of $\vec{x}$, giving us governing equations for the system we are observing.

One major limitation of the SINDy method is its dependence on the choice of functions f_j, and its memory usage when there is a large number of functions and a large number of dimensions of the system being modelled.

1.2.3 System-informed Deep Learning

Another method of using machine learning in concert with pre-existing knowledge, or first-principles, is to add governing equations to the loss function, thereby constraining the machine learning model to respect a physical principle we believe governs the system we are measuring.

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{ODE}} \tag{1.5}$$

where

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}} \sum_{i}^{N_{\text{data}}} \left(x_i(t_i) - \hat{x}_i(t_i)\right)^2 \tag{1.6}$$

$$\mathcal{L}_{\text{ODE}} = \frac{1}{N_{\text{ODE}}} \sum_{j}^{N_{\text{ODE}}} \left(\left.\frac{d\hat{x}}{dt}\right|_{t=\tau_j} - f(\hat{x}_j(\tau_j;\theta), \tau_j) \right)^2 \tag{1.7}$$

and times t_i do not need to equal times τ_j. Note that the derivative is the time-derivative of $\hat{x}$ the prediction from our model. We can compute the derivative of our neural network model using automatic differentiation, which is readily available and implemented in many different libraries [19, 26].

The applicability of this method has been shown in many different contexts. Initially developed for physical systems in two parts [38, 39] in which the authors described the method and used it for solving PDE problems from physics, and for determining PDE parameter values. It was then broadly applied in fluid dynamics problems [8], and in describing surface waves on a metal plate for investigating surface-breaking cracks in metal plates. [42]. In the biology realm the method was also investigated for solving a yeast glycolysis model, cell apoptosis model, and an endocrine model, where the system-informed neural network was used to determine the dynamics of unobserved state variables, and

also to infer the parameters of the DE system; albeit, this study was a simulation study with noise added to the simulated datasets to mimic data collection noise, and not using experimental data.

Both SINDy and system-informed neural networks require, or improve with, having pre-existing knowledge of the system being modelled. For SINDy, having some idea of what functions should be used in the sparse regression improves the whether the method will converge upon an interpretable and predictive system of DEs. For system-informed neural networks, having an understanding of what DEs or other constraints to include in the loss function is paramount to the methodology; for the constraint in Eq. **??** we require that we know f beforehand, this is an important distinction for the next section. The advantage of SINDy over system-informed neural networks is provides a DE model when we initially did not have one. While system-informed neural networks can be used as a way to find the solution for a system and the values of the parameters in its model. Where SINDy falls short compared to system-informed neural networks is its reliance on the functions the user pre-selects, rather than having a purely flexible neural network.

1.2.4 Neural Differential Equations

The most common neural differential equation, and especially the most well-known, is the neural ordinary differential equation of [10]. Neural ODEs are of the form

$$\frac{dx}{dt} = f_{\text{NN}}(x, t; \theta) \tag{1.8}$$

where f_{NN} is a neural network with x and possibly t as inputs. The neural network's learnable weights are θ. In this case, the neural network is defining the vector field of the dynamical system. This is different from system-informed neural networks, where the neural network is being used as the solution x of a pre-defined dynamical system f, with the f being used to constrain the neural network's outputs to satisfy the differential equation, whereas with neural differential equations, we do not know f, and instead allow it to be a neural network f_{NN} that will determine a vector field from data.

As noted in [10], we can be sure a solution exists to Eq. 1.8 because of Picard's Theorem. Since a composition of Lipschitz continuous functions is Lipschitz continuous, so long as we choose Lipschitz continuous activation functions for the neural network f_{NN}, then f_{NN} will be Lipschitz continuous as well. So, we can apply Picard's Theorem to guarantee the existence and uniqueness of a solution to Eq. 1.8, given an initial value.

In this case we are making a distinction between the neural network weights θ, and the ODE parameters p, however there is work where this distinction is not made and all

parameters are updated during training [31]. In this form we can then use any numerical integration method to obtain $x(t)$ at time t.

Evaluation of neural differential equation

Evaluating neural networks is a fairly straightforward task: go through each of the matrix multiplications and activation functions corresponding to the layers. A neural differential equation is only a little different. This time we must use a numerical integrator, along with the neural network, to provide us with the desired output x.

$$x_{\text{pred}}(t) = \text{odeint}\big(x(0), f_{\text{NN}}(x;\theta)\big) \tag{1.9}$$

with odeint$(\cdot,\cdot)$ being a ODE integrator taking an initial condition and the function outputting the derivative. Unlike other neural network architectures, neural differential equations enable the user to select any ODE integrator they wish, allowing the use of fixed or adaptive algorithms. The latter may be useful when modelling a system that is oscillatory, or having otherwise sharp changes.

Training neural differential equations

Training a neural differential equation has many similarities with training other neural networks: update the neural network's weights θ_i using the gradients of the loss function

$$\theta_{i,\text{new}} = \theta_{i,\text{old}} - \lambda \partial_{\theta_i} \mathcal{L}(x_{\text{true}}, x_{\text{pred}}) \tag{1.10}$$

where

$$\partial_{\theta_i} \mathcal{L}(x_{\text{true}}, x_{\text{pred}}) = \partial_{x_{\text{pred}}} \mathcal{L} \cdot \partial_{\theta_i} x_{\text{pred}} \tag{1.11}$$

is numerically computed using automatic differentiation [19]. A major difference, however is that we must take a derivative through the ODE integrator in order to correctly compute the derivative of the loss function. There happen to be two ways to do this. The first is using forward sensitivities, and the second is using backward sensitivities. The backward sensitivity method, or adjoint method, is popularized in [10], but used extensively in the Julia Programming Language, specifically in the Julia package DiffEqFlux [35].

Forward sensitivities compute the right-hand side of Eq. 1.11 by defining a second ODE

to solve while solving Eq.1.8

$$\partial_{\theta_i}\left(\frac{d}{dt}x_{\text{pred}}\right) = \partial_{\theta_i} f(x_{\text{pred}}(\theta_j), \theta_j) \tag{1.12}$$

$$= \partial_{x_{\text{pred}}} f \cdot \partial_{\theta_i}(x_{\text{pred}}) + \partial_{\theta_i} f \tag{1.13}$$

$$\implies \frac{d}{dt}\Big(\partial_{\theta_i}(x_{\text{pred}})\Big) = \partial_{x_{\text{pred}}} f \cdot \partial_{\theta_i}(x_{\text{pred}}) + \partial_{\theta_i} f \tag{1.14}$$

where in the last line we used Clairaut's theorem. Letting $s_i = \partial_{\theta_i}(x_{\text{pred}})$ we can write

$$\frac{d}{dt}s_i = \partial_{x_{\text{pred}}} f \cdot s_i + \partial_{\theta_i} f \tag{1.15}$$

and solving for s_i numerically. The forward sensitivity method was frequently used in the early bioprocess literature trying to involve neural networks in modelling (as will be described below). However, the forward sensitivity method has limitations. It requires computing a large number of intermediate points, each for a different neural network weight. [27] found that the forward sensitivity method is more efficient on small problems with few weights, and otherwise adjoint methods should be used.

The backward sensitivities –also known as the adjoint – method, never computes s_i. Instead, it computes $\partial_{\theta_i}\mathcal{L}$ directly, and is the more commonly used method in the neural differential equation literature. The required equations for the adjoint method are

$$\frac{da}{dt} = -a\partial_{x_{\text{pred}}} f(x_{\text{pred}}, t)$$
$$\frac{da_\theta}{dt} = -a\partial_{\theta_i} f(x_{\text{pred}}, t)$$

where

$$a(t) = \frac{dL}{dx_{\text{pred}}(t)}$$
$$a_\theta(t) = \frac{dL}{d\theta(t)}$$

with initial conditions $a(T) = \frac{dL}{dx(T)}$ and $a_\theta(T) = 0$. We solve the above dynamical equations backward in time to get the gradients of the loss function a_θ. Derivations for this method have been presented elsewhere [10], and so will not be included here.

Classification with Neural ODEs

Despite the primary focus of this report being on using neural ordinary differential equations to combine machine learning and mechanistic modelling for the sake of regression tasks, the reader may be interested in how they are used for classification tasks. We take a brief interlude to describe that now.

Performing classification with neural ODEs requires an affine layer after the neural ODE

$$\frac{d\vec{x}}{dt} = f_{\text{NN}}(t, \vec{x}) \tag{1.16a}$$

$$\hat{\vec{y}} = g(\vec{x}) \tag{1.16b}$$

$$\hat{y}_{\text{pred}} = \text{argmax}(\hat{\vec{y}}) \tag{1.16c}$$

where $\vec{x}(0) \in \mathbb{R}^{28\times28}$ is a grey-scale image with 28×28 pixels unrolled into a long vector, with an associated label from 0 to 9 (if we are looking at images of digits as in the MNIST dataset [12]). For image classification, we want to map from an image to its label. $f_{\text{NN}} \in \mathbb{R} \times \mathbb{R}^{28\times28} \to \mathbb{R}^{28\times28}$ is a neural network defining the neural ODE, and $g : \mathbb{R}^{28\times28} \to \mathbb{R}^{10}$ is an affine neural network layer outputting into the possible labels. The convention is to then integrate the neural ODE from 0 to 1 with initial condition $\vec{x}(0)$ being the inputted image. After time-evolving the image, the new vector $\vec{x}(1)$ is fed into g. $g(\vec{x})$ then provides a vector with 10 entries, we take the index of the maximum value, and that is the label predicted of the inputted image. While using this method allows us to use neural ODEs for classification tasks, the standard is still to use other neural network architectures because they have been found to perform better.

Augmented Neural ODEs

Conventional artificial neural networks are universal function approximators. However, it can be shown, and was first described in [13], that Neural ODEs by themselves are not universal function approximators. Instead, Neural ODEs must be ”augmented” with additional dimensions compared to their inputs, enabling them to use the additional dimensions to produce simpler vector fields; experimentally additional dimensions has also been found to result in performance gains [13, 30].

To show Neural ODEs cannot represent any function, we will show two cases similar to [13]. The first is in 1D. Consider a classification problem where $x = 1$ is to be classified

as 0 and $x = 0$ is to be classified as 1; let $g_{\text{swap}} : \mathbb{R} \to \mathbb{R}$ define that function. If we define the neural ODE as

$$\frac{dx}{dt} = f_{NN}(t, x) \tag{1.17}$$

then we know that $x(t)$ will be a continuous path. We can define two continuous trajectories, one starting at $x(0) = 0$ and one starting at $x(0) = 1$, x_0 and x_1 respectively. At time T they terminate at $x_0(T) = 1$ and $x_1(T) = 0$. We can define $x(t) := x_1(t) - x_0(t)$ giving us a continuous function beginning at $x(0) = 1$ and terminating at $x(T) = -1$. From the intermediate value theorem, we know that there must exist a point $c \in (0, T)$ such that $x(c) = 0 \implies x_0(c) = x_1(c)$. This implies both trajectories must intersect, despite beginning at different initial values. But because solutions to ODEs must be unique, we arrive at a contradiction, giving that a standard neural ODE cannot represent the function g_{swap}. Note that the proof provided is a variant of the one given in [13].

In higher dimensions the functions to choose to show the result are functions taking concentric circles to different values. Take $0 < r_1 < r_2 < r_3$, define two annuli:

$$g_{\text{swap,2D}}(\vec{x}) = \begin{cases} 1, & \text{if } ||\vec{x}|| < r_1 \\ 0, & \text{if } r_2 < ||\vec{x}|| < r_3 \end{cases} \tag{1.18}$$

Showing that neural ODEs cannot learn the function $g_{\text{swap,2D}}(\vec{x})$ is shown in [13], and relies on two main points:

- Solutions to Eq. 1.8 are homeomorphisms, and therefore preserve topology.
- Classification using Eq. 1.16 requires the two annuli to be linearly separable.

since the only way to make annulus $A := \{\vec{x} \in \mathbb{R}^d | ||\vec{x}||_2 < r_1\}$ linearly separable from annulus $B := \{\vec{x} \in \mathbb{R}^d | r_2 < ||\vec{x}||_2 < r_3\}$ is to break B or to have overlapping trajectories, neither of which can occur.

Remedying neural ODEs inability to approximate any function is done by 'augmenting' it with extra dimensions. This is done by introducing $\vec{a}(t) \in \mathbb{R}^{n_a}$, with n_a being the 'augmentation' dimensions, and concatenating it to the end of $\vec{x}(t)$: $\begin{bmatrix} \vec{x}(t) \\ \vec{a}(t) \end{bmatrix}$. And, we rewrite the neural ODE as

$$\frac{d}{dt} \begin{bmatrix} \vec{x}(t) \\ \vec{a}(t) \end{bmatrix} = f_{\text{NN}}(t, \begin{bmatrix} \vec{x}(t) \\ \vec{a}(t) \end{bmatrix}) \tag{1.19a}$$

and $\vec{a}(0) = \vec{0}$. The additional dimensions then enables the the neural ODE to find solutions that can separate spaces it otherwise would not be capable of separating. Numerical work also finds that augmented Neural ODEs tend to be more easily trained, and generalize better, than non-augmented ones [13, 30] because the flows it generates are 'simpler'.

Zero-augmentation, as described, is not the only way to perform augmentation. Explored in [30], augmentation can also be generalized to input-layer augmentation. In this scheme we do not necessarily choose $\vec{a}(0) = \vec{0}$, and instead generalize to other options

$$h_x(\vec{x}) : \vec{x} \to \begin{bmatrix} \vec{x}^T & \vec{h}(\vec{x})^T \end{bmatrix}^T \tag{1.20}$$

where 0-augmentation simply has

$$h(\vec{x}) : \vec{x} \to \begin{bmatrix} \vec{x}^T & \vec{0}^T \end{bmatrix}^T \tag{1.21}$$

1.2.5 Hybrid Neural Differential Equations

Now that we have covered the basics of neural ordinary differential equations, and some related work in the space of combining machine learning with mechanistic modelling, we can describe specific methods of doing so with neural ODEs. Recall that neural ODEs are of the form

$$\frac{dx_{\text{pred}}}{dt} = f_{\text{NN}}(x, t; \theta) \tag{1.22}$$

where f_{NN} is a neural network. Because many systems are modelled using differential equations, we can include explicit terms we believe should model a system of interest. We can write this as

$$\frac{dx_{\text{pred}}}{dt} = k(x, t; p) \cdot f_{\text{NN}}(x, t; \theta) + g(x, t; p) \tag{1.23}$$

where $k(x, t; p)$ and $g(x, t; p)$ are functions we select to be in the differential equation, and p are the parameters for those functions. This enables us to develop a hybrid mechanistic-machine learning dynamical system, where we have terms corresponding to mechanisms we believe to govern the system (functions k, and g) and terms for a neural network (f_{NN}). The expressivity of a neural network can then improve our mechanistic model by capturing dynamics we may have otherwise missed.

Let us consider a simple example: the Lotka-Volterra model. In the Lotka-Volterra

model, we have

$$\frac{dx_1}{dt} = \alpha x_1 - \beta x_1 x_2 \tag{1.24a}$$
$$\frac{dx_2}{dt} = -\delta x_2 + \gamma x_1 x_2 \tag{1.24b}$$

Suppose for example, that the modeller did not know the interaction terms in Eq. 1.24, but they suspected there was some interaction. Then, the modeller could include in each equation an output of a neural network

$$\frac{dx_1}{dt} = \ \alpha x_1 + f_{\theta,1}(x_1, x_2) \tag{1.25a}$$
$$\frac{dx_2}{dt} = -\delta x_2 + f_{\theta,2}(x_1, x_2) \tag{1.25b}$$

and, using observations of the predator-prey system, the modeller can train the neural network to approximate the interaction term(s) that, in this case we know to be present, but frequently we either do not have a full a model or do not have the correct mechanisms. In the latter case, we might imagine having an predator-prey system where the prey have a carrying capacity, but we are modelling it as in 1.24. If we suspected we had the wrong model, but were able to obtain observations from the system then we could modify our Lotka-Volterra model to be

$$\frac{dx_1}{dt} = \ \alpha x_1 - \beta x_1 x_2 + f_{\theta,1}(x_1, x_2) \tag{1.26a}$$
$$\frac{dx_2}{dt} = -\delta x_2 + \gamma x_1 x_2 + f_{\theta,2}(x_1, x_2) \tag{1.26b}$$

where the neural network is being used to 'correct' the mismatch between our incorrect model and the system under question.

Note that here I am calling Eq. 1.23 a 'Hybrid Neural Differential Equation' while in other works it is called a 'Universal Differential Equation' [36, 40]. Moreover, the authors in [36] go a step further: they use the SINDy method on outputs from the neural network portion of Eq. 1.23 to find a functional form for it. Having this functional form is meant to provide a mechanistic interpretation of what the neural network has learned. However, there is no reason to believe the family of functions chosen in the SINDy method is correct.

Training hybrid neural differential equations can occur in two different ways. In one method the differential equation parameters – α, β, δ, γ in Eq. 1.26– can be updated along with the neural network weights, but that removes some of the interpretability aspects

of the mechanistic part of Eq. 1.23. The alternate method is to first fit Eq.1.23 to data without the neural network terms, then holding the parameters α, β, δ, γ fixed, re-introduce the neural network terms and only update their weights $\vec{\theta}$.

Initializing the neural network weights is usually done by starting the weights $\vec{\theta}$ at, or very near, zero. Rather than initializing with random values

1.2.6 Hybrid models in process modelling

Traditionally, modelling of cells and bioprocesses consists of incorporating as many known or guessed details as possible into a reaction network. That reaction network is then translated into a system of ODEs usually resulting in high-dimensional models. At the extreme side of the spectrum is whole-cell modelling, where the modeller(s) attempt to capture details of every kinetic process in the cell. This approach faces various challenges to its effectiveness: frequently the models are under-parameterized as the parameters (e.g. rate parameters, enzyme concentrations, intracellular metabolite concentrations) cannot easily be measured; or the mechanisms inside and between cells is poorly understood, leading to phenomenological models or omitting processes known to affect the system. Despite this, modelling approaches have been invaluable for re-wiring metabolism, informing target selection, controlling bioprocesses, and design of experiments.

As bioprocesses become measured in different ways, high-throughput methods become more available, and the bioprocess industry continues to invest in process analytical technology (PAT), the industry is expected to experience a deluge of data [52, 4, 2, 33]. Because of this, akin to the petrochemical and chemical industries [52], the ground is fertile for the growth of data-driven methods such as machine learning, and specifically deep learning, to assist in leveraging this data for automation and optimized production.

Data-driven models rely heavily on data to approximate a function from given inputs to given outputs using statistical correlations between the input variables. Because they rely so heavily on existing data, rather than first-principles or mechanistic knowledge, data-driven models can have difficulty extrapolating to new situations not already explored in the experiments used to train the model. However, their flexibility allows them to find non-intuitive, non-linear relationships between input variables and target outputs.

First-principles, or mechanistic, models, are derived from physical, chemical, or biological principles. In bioprocesses they usually include mass balances, reaction kinetic pathways, transport phenomena, and thermodynamics, usually expressed as a system of differential equations. Because they are built from principles about how systems func-

tion, their parameters usually have a physical meaning, and they generally exhibit good extrapolation capability.

The avenue explored in this report, and being increasingly discussed in the literature is hybrid modelling – models incorporating both data-driven and first-principles-based methods. The basic outline of a hybrid model is to use a mechanistic model with data-driven approaches estimating unknown parts of the equations, or to estimate unknown functional relationships between variables of interest. Hybrid modelling is appealing because the flexibility of a data-driven method such as deep learning can make it easier to model unknown interactions or processes by using historical data sets or during the monitoring of a system. As discussed in the section on complexities in scale-up, osmolality, pH, temperature, and dissolved oxygen can all influence the efficiency and production of a culture used for a bioprocess. Yet, many models omit those quantities, while also being unable to systematically account for the effects heterogeneity in industrial-scale reactors has on model rate parameters. However, a neural network could be used to find a mapping between those quantities and the product of interest, and to estimate unknown interactions and processes. The mechanistic model may then improve the overall model's capacity to extrapolate (compared to a full data-driven method) and reduce the amount of data required for training.

In process systems research, hybrid mechanistic and machine learning models are usually categorized into two types: serial and parallel types [15]. This categorization depends on how the machine learning and the mechanistic model relate to each other. In parallel hybrid models, the machine learning part is trained on the residual between the mechanistic model and the plant (or observed data) Figure 1.2. Serial hybrid models come in two flavours: the machine learning model's output feeds into the mechanistic model Figure 1.2a, or the mechanistic model's output feeds into the machine learning model Figure 1.2b.

The goal of using parallel models is to correct model-plant mismatch. The ML part of the model is expected to approximate dynamics inherent in the residuals between the mechanistic model's predictions, and the observations from the plant. As the ML part is trained, it learns to correct the model. Traditionally, the ML part is used at the stage after integrating the dynamical model.

With serial hybrid models, the basic goal is to use the ML part to approximate some unknown interactions between state variables. This is done in one of two flavours. In the first flavour, the ML model's outputs are outputs into a set of differential equations while its inputs can be a combination of the state variables, online measurements from the plant, or other parameters. Most common usages of the serial flavour is to estimate the rate parameter of a process model as it is expected many details are missing from simply assuming

it to be constant. There has been some work over the past three decades to use hybrid mod-

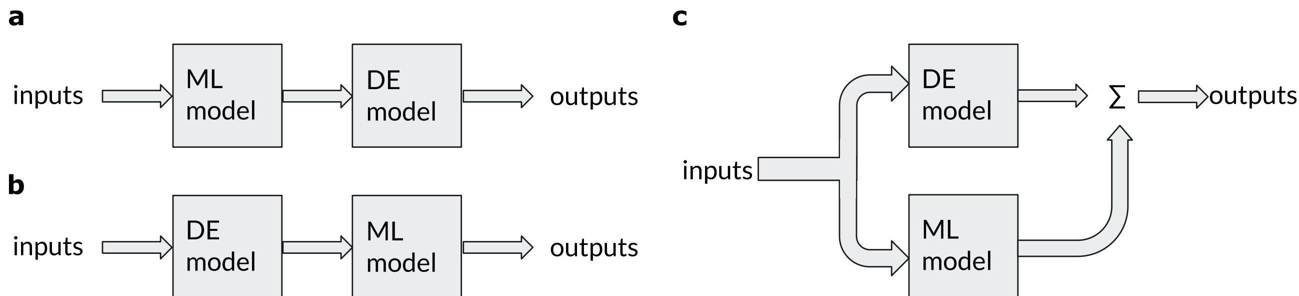

Figure 1.2: Graphical representation of the three broad types of hybrid models. On the left we have the two serial hybrid models (ab) where the mechanistic model and machine learning model feeds into each other; i.e. the output of a neural network is used as input to a DE system (a), or vice-versa (b). On the right we have parallel hybrid models, where the machine learning model's output is used to estimate the residual between the DE model's output and the true observations.

elling techniques to improve biomanufacturing processes [34, 44, 47, 49, 43, 32, 23, 14]. [34] was the first work to explicitly combine both a neural network and a mechanistic model. In it, the authors used a serial hybrid model with the neural network's output replacing a constant rate parameter, attempting to estimate the dynamics of a 'true' model from an approximate model. [44] uses a hybrid model for control of a yeast biomass production process while trying to balance amount of ethanol produced (not desired in this case); the authors find that using a hybrid model where the neural network estimates rate parameters trained with observed data performs just as good or better than a traditional mechanistic model and is more easily trained than traditional neural network models at the time. [47] has a comparison of mechanistic modelling, black box modelling, and a hybrid model for the production of penicilin G. In a pair of works published in 2016 [48, 49] von Stosch, Hamelink, and Oliveira, described an in depth case study of an industrial *E. coli* fermentation process. In it, they bring back the idea of using hybrid modeling to improve upon prediction and control of biomanufacturing processes, particularly to improve on finding optimal process condition to maximize yield of desired products while reducing variability between processes; they found that one specific advantage of a machine learning component is to enable the usage "online" process parameters (such as temperature, pH, dissolved oxygen, agitation rate, air-flow rate, vessel pressure) whose relationship to product yield is unclear. [43] in 2017 discusses the benefits hybrid models can provide in improving mam-

malian cell culture process modelling and control, especially in regards to incorporating online process parameters. In response to that, in 2019 [32] developed a hybrid model using a 1-layer neural network and a first-principles model consisting of 8 species, with the neural network being used to predict the rate parameter of the model.

Even more recently, [23] developed a serial hybrid model where a mechanistic model outputted into a Long-Short Term Memory neural network architecture to predict the pH trajectory of a cream cheese fermentation to determine how long until the fermentation is finished, ultimately finding the hybrid model outperformed the mechanistic model alone. [14] attempted to a similar thing, but compared the hybrid model of [23] to a simple 1-layer neural network with inputs pH(t), pH($t-1$), pH($t-2$) to predict pH($t+1$), and seemed to find good results.

The overall trend is that a few studies in the biomanufacturing literature has found evidence that hybrid models can be effective. Hybrid models allow for some flexibility in the rate parameters, and in what inputs are used. More recent work has tried ignoring the mechanistic approach entirely, opting instead to use only artificial neural networks. Despite these studies, there is still room for exploration and improvement of the hybrid modelling approach, especially with more and more mammalian processes being used for biomanufacturing.

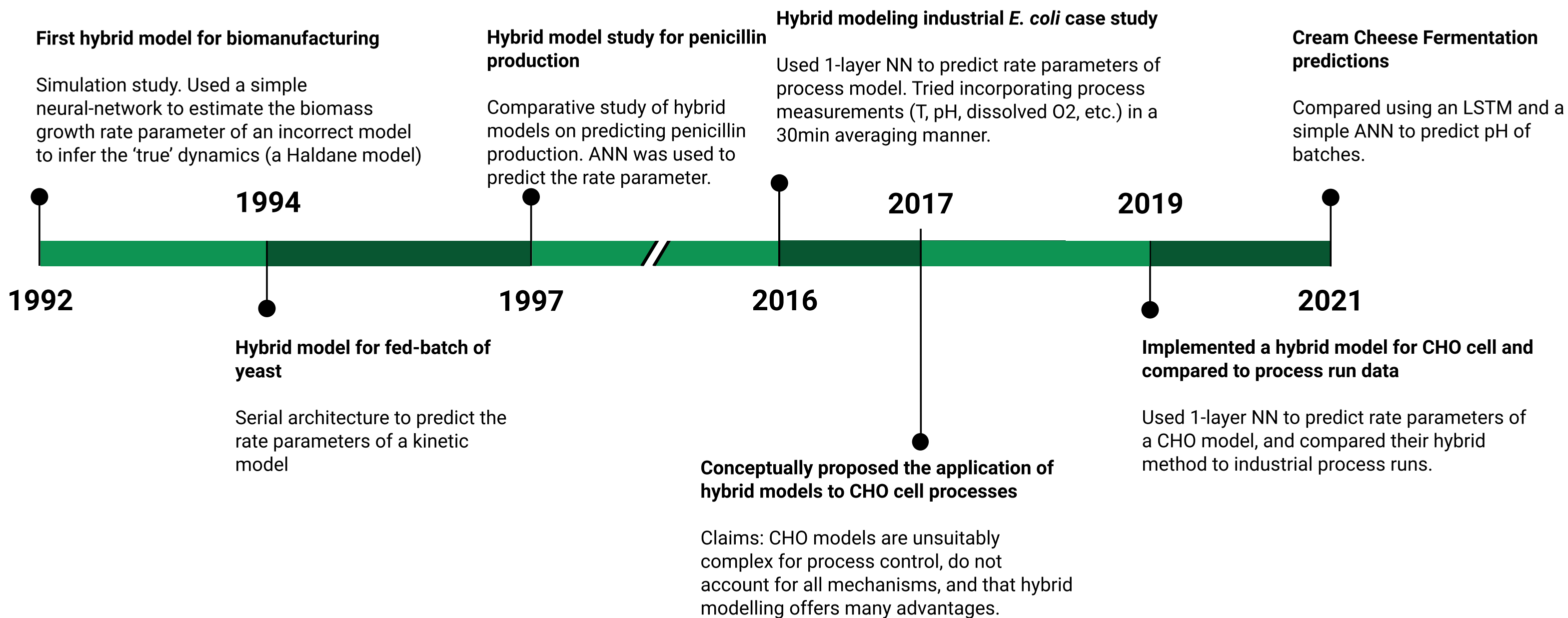

Figure 1.3: Timeline of key hybrid modelling works in the biomanufacturing setting. Note the break between years 1997 and 2016. Works referred to are as follows: 1992, [34]; 1994, [44]; 1997, [47]; 2016, [49]; 2017, [43]; 2019, [32]; 2021, [23, 14]

Chapter 2

Results

2.1 Hybrid Modeling from Simulated Data

The main results of this report are based on investigating the effectiveness of the method in simulated scenarios with simulated data. This is similar to works done elsewhere [7, 28, 29, 20, 46, 38], with the goal being to motivate using the method on observable systems where the true model is unavailable.

The methodology used here is to first take a model and treat it as the system we wish to model and predict, and pretend we do not have access to the model itself. From this true model, we generate some simulations, and treat it as observations of the system of interest. Afterwards, we modify the model so we have an incorrect, partial model – this is what we are imagining a modeller may come up with when looking at a new system. With this partial model we can apply some of the hybrid methods described above to determine if the original dynamics are recovered or approximated.

2.1.1 Haldane Model

One of the bioprocess models considered here is a variant of the Haldane model as analyzed by Ajbar [5]. The Haldane model here is a model of a continuous bioprocess system, modelling the growth of biomass X at growth rate μ as it uses the substrate S, while producing product P with yield ε, dilution factor D and a constant feed rate.

$$
\begin{aligned}
\frac{dX}{dt} &= \mu(S,X)X - DX \\
\frac{dS}{dt} &= D(S_f - S) - \sigma X \\
\frac{dP}{dt} &= \varepsilon X - DP
\end{aligned}
$$

In dimensionless form we have, where the linear assumption that $\sigma = a\varepsilon$ is also made

$$\frac{d\tilde{X}}{d\tilde{t}} = \tilde{\mu}\tilde{X} - \tilde{D}\tilde{X} \tag{2.1a}$$

$$\frac{d\tilde{S}}{d\tilde{t}} = \tilde{D}(\tilde{S}_f - \tilde{S}) - \lambda_1\tilde{\epsilon}\tilde{X} \tag{2.1b}$$

$$\frac{d\tilde{P}}{d\tilde{t}} = \lambda_2\tilde{\epsilon}\tilde{X} - \tilde{D}\tilde{P} \tag{2.1c}$$

with non-dimensionalized variables relating to the dimensionalized state variables in the following way

$$
\begin{aligned}
&\tilde{S} = \frac{S}{S_{\text{ref}}}, \quad \tilde{X} = \frac{aX}{S_{\text{ref}}}, \quad \tilde{P} = \frac{P}{P_{\text{ref}}}, \quad \tilde{D} = \frac{D}{\mu_{\text{ref}}} \\
&\tilde{t} = t\mu_{\text{ref}}, \quad \tilde{\mu} = \frac{\mu}{\mu_{\text{ref}}}, \quad \tilde{\epsilon} = \frac{\epsilon}{\epsilon_{\text{ref}}}, \quad \lambda_1 = \frac{\epsilon_{\text{ref}}}{\mu_{\text{ref}}}, \quad \lambda_2 = \frac{\epsilon_{\text{ref}}S_{\text{ref}}}{a\mu_{\text{ref}P_{\text{ref}}}}
\end{aligned}
$$

as done in [5]. In the same reference, Ajbar mapped out the parameter values leading to different regimes of stability. In this model the rate of growth and product synthesis are both inhibited by high concentrations of product. The rate of cell growth $\tilde{\mu}$ and the rate of synthesis $\tilde{\epsilon}$ are expressed as

$$
\begin{aligned}
\tilde{\mu} &= \left(\frac{\tilde{S}}{\beta_2 + \tilde{S} + \gamma_1\tilde{S}^2}\right)\left(\frac{1}{1+\tilde{P}}\right) \\
\tilde{\epsilon} &= \left(\frac{\tilde{S}}{\beta_2 + \tilde{S} + \gamma_2\tilde{S}^2}\right)\left(\frac{1}{1+\lambda_3\tilde{P}}\right)
\end{aligned}
$$

with $\lambda_3 = \frac{K_{1P}}{K_{2P}}$. As mentioned above, this model will be the true model we are trying to discover.

We will assume we are unaware of the functional forms of the rate of cell growth $\tilde{\mu}$ and the rate of synthesis $\tilde{\epsilon}$. This means we are assuming the model we constructed is

$$\frac{d\tilde{X}}{d\tilde{t}} = p_1\tilde{X} - \tilde{D}\tilde{X} \tag{2.2a}$$

$$\frac{d\tilde{S}}{d\tilde{t}} = \tilde{D}(\tilde{S}_f - \tilde{S}) - \lambda_1 p_2 \tilde{X} \tag{2.2b}$$

$$\frac{d\tilde{P}}{d\tilde{t}} = \lambda_2 p_2 \tilde{X} - \tilde{D}\tilde{P} \tag{2.2c}$$

where we do not know the rate of cell growth nor the rate of synthesis, nor do we know whether they have different functional forms. Instead, we include the trainable parameters p_1 and p_2. In order to use the hybrid method, we replace the trainable parameters with outputs from a neural network to get

$$\frac{d\tilde{X}}{d\tilde{t}} = \mathrm{NN}_1(\tilde{X}, \tilde{S}, \tilde{P}) \cdot \tilde{X} - \tilde{D}\tilde{X} \tag{2.3a}$$

$$\frac{d\tilde{S}}{d\tilde{t}} = \tilde{D}(\tilde{S}_f - \tilde{S}) - \mathrm{NN}_2(\tilde{X}, \tilde{S}, \tilde{P}) \cdot \tilde{X} \tag{2.3b}$$

$$\frac{d\tilde{P}}{d\tilde{t}} = \mathrm{NN}_3(\tilde{X}, \tilde{S}, \tilde{P}) \cdot \tilde{X} - \tilde{D}\tilde{P} \tag{2.3c}$$

where $\mathrm{NN}_1(\tilde{X}, \tilde{S}, \tilde{P})$, $\mathrm{NN}_2(\tilde{X}, \tilde{S}, \tilde{P})$, $\mathrm{NN}_3(\tilde{X}, \tilde{S}, \tilde{P})$ are outputs of our neural network.

In this case, the true system is still represented by the Haldane model discussed above. But, now we are pretending that we constructed model Eq. 2.2, and that we wish to combine that model with a neural network to construct a neural differential equation.

To make training data for the hybrid mechanistic-machine learning model, we simulated the true Haldane model (Eq. 2.1), using fixed parameter values, and initial conditions sampled from uniform distributions as listed in Table 2.1.

N	200
X distribution	Unif(0.2,0.5)
S distribution	Unif(1.7,2.3)
P0	0.0
D	0.4
β_1	0.0471
β_2	0.1
γ_1	1.2
γ_2	1.0
λ_1	0.5
λ_2	0.1
λ_3	1.0

Table 2.1: Values used for parameters in the true Haldane model, along with distributions used for sampling initial conditions when building a training and validation set. Both the training and validation sets are of size N, but have distinct trajectories and initial conditions.

To build the validation set, we sample initial conditions, and reject any that are exactly equally to any initial condition in the training set. We continue this procedure until we have a validation set of size 200.

Network Architecture

This section covers some model building and comparisons. We experiment with two different aspects of neural differential equations: (1) activation functions between layers of the neural network; (2) number of layers making up the neural network. As discussed in section 1.2.6, previous work in the hybrid modelling literature uses one-layer networks. In contrast, here we explore using deeper network architectures.

The final model architecture used here is described in Table 2.2. We used a feedforward neural network architecture with tanh activation functions. While there are 4 inputs and outputs, three of them correspond to state vectors, and the other one corresponds to the extra dimension for zero-augmentation.

Layer	N inputs	N outputs	Activation function
1	4	50	tanh
2-6	50	50	tanh
7	50	20	tanh
8	20	4	none

Table 2.2: Neural network architecture for the final network used.

Experimenting with different architectures using the same data set suggests that deeper networks with smooth activation functions perform better, albeit requiring more time to train. Results are noted in Table 2.3. The two activation functions we compared are tanh and the "leaky" ReLu $\max(\delta x, x)$ where δ is usually small, like 0.01. Leaky ReLus are intended to have the benefits of ReLu functions (where $\delta = 0$), while also mitigating the "dead neuron" problem where many coefficients in layers become and stay zero.

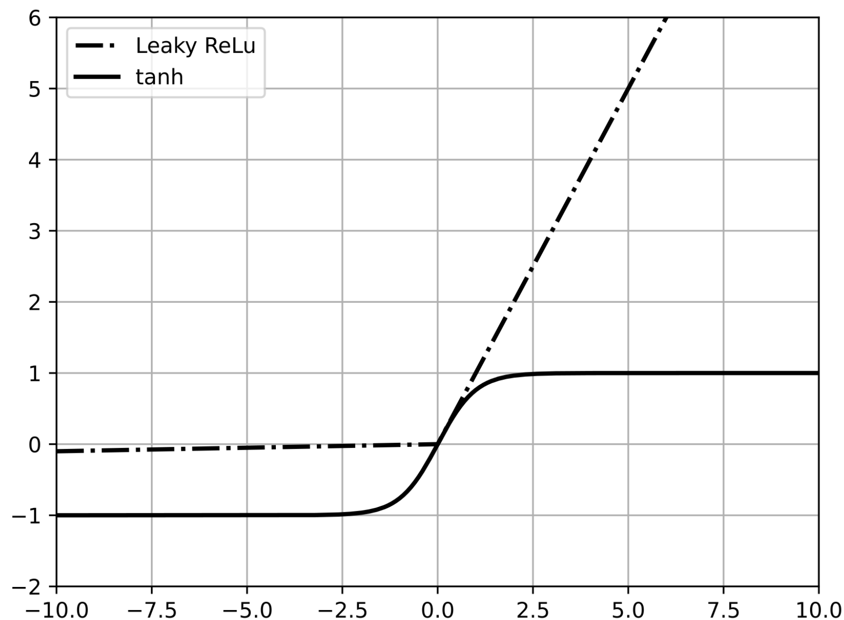

Figure 2.1: The two activation functions compared in this report. Dash-dotted line corresponds to the leaky ReLu of $\max(0.01x, x)$, while the solid line corresponds to $\tanh(x)$.

When adding layers, we are introducing more 50×50 hidden layers; e.g. architecture c of Table 2.3 has 4 layers, meaning it has a 50×4 input layer, then one 50×50 hidden layer, a 20×50 layer, and lastly a 4×20 output layer. When changing activation functions, we change the activation function between each layer to the one listed in Table 2.3.

Training the neural network requires minimizing a loss function. In each of the cases presented here, we use a mean-squared error loss function $\text{MSE}(\vec{y}_{\text{true}}, \hat{\vec{y}}) = \frac{1}{N}\sum_i^N (y_{\text{true},i} - \hat{y}_i)^2$, where $\vec{y}_{\text{true}}, \hat{\vec{y}} \in \mathbb{R}^N$

Lower loss value following increasing depth is commonly found in neural networks used in image processing and in language understanding, although the neural networks used in those domains are orders of magnitude larger than the ones used here. As described in Figure 1.2, previous work in the hybrid bioprocess modelling literature used single layer neural networks. Table 2.3 lists some evidence we found suggesting more layers may improve performance in the hybrid bioprocess modelling situation.

Architecture	Number of layers	Activation function	Average time per epoch (s)	Validation set loss
a	4	Leaky ReLu	110.1	12.46
b	4	Tanh	13.84	10.68
c	6	Tanh	33.03	9.735
d	8	Tanh	60.59	8.409

Table 2.3: Comparison of different model architecture choices. One "epoch" corresponds to the ADAM optimizer iterating through the entire training data set once. While increasing number of layers increases training time, it tends to decrease the loss value on the validation set. Note that a leaky relu activation function seems to require more time per epoch compared to the same architecture with different activation functions. As discussed above, this might be because the tanh function is smooth while the leaky relu function is not.

Case 1: no noise

Here we trained the neural differential equation on a noiseless data set. First, we simulated Eq. 2.1 up to $\tilde{t} = 75$, with 150 equally spaced data points. Next, we take the first 50 data points (i.e. up to $\tilde{t} = 25$), and store that along with its randomly chosen initial condition. We repeat that 200 times, so that we have 200 different sets of 50 data points. This data set is used to train the hybrid neural differential equation. For some of the initial conditions, the dynamics have a short transient phase reaching a stationary state before $\tilde{t} = 25$, while others have a longer transient phase not reaching a stationary state until after $\tilde{t} = 25$.

Figure 2.2 shows representative results on new initial conditions, some of which were not in the training nor in the validation set. Shown are both results from the assumed model Eq. 2.2 and the hybrid model. The parameters of the assumed model are $(p_1, p_2) \approx (0.4689, 0.3527)$, determined by using the Julia differential equation parameter estimation library DiffEqParamEstim [37] with the training data. Compared to the assumed model, the hybrid model seems to perform better, predicting the stationary states of biomass, substrate, and product.

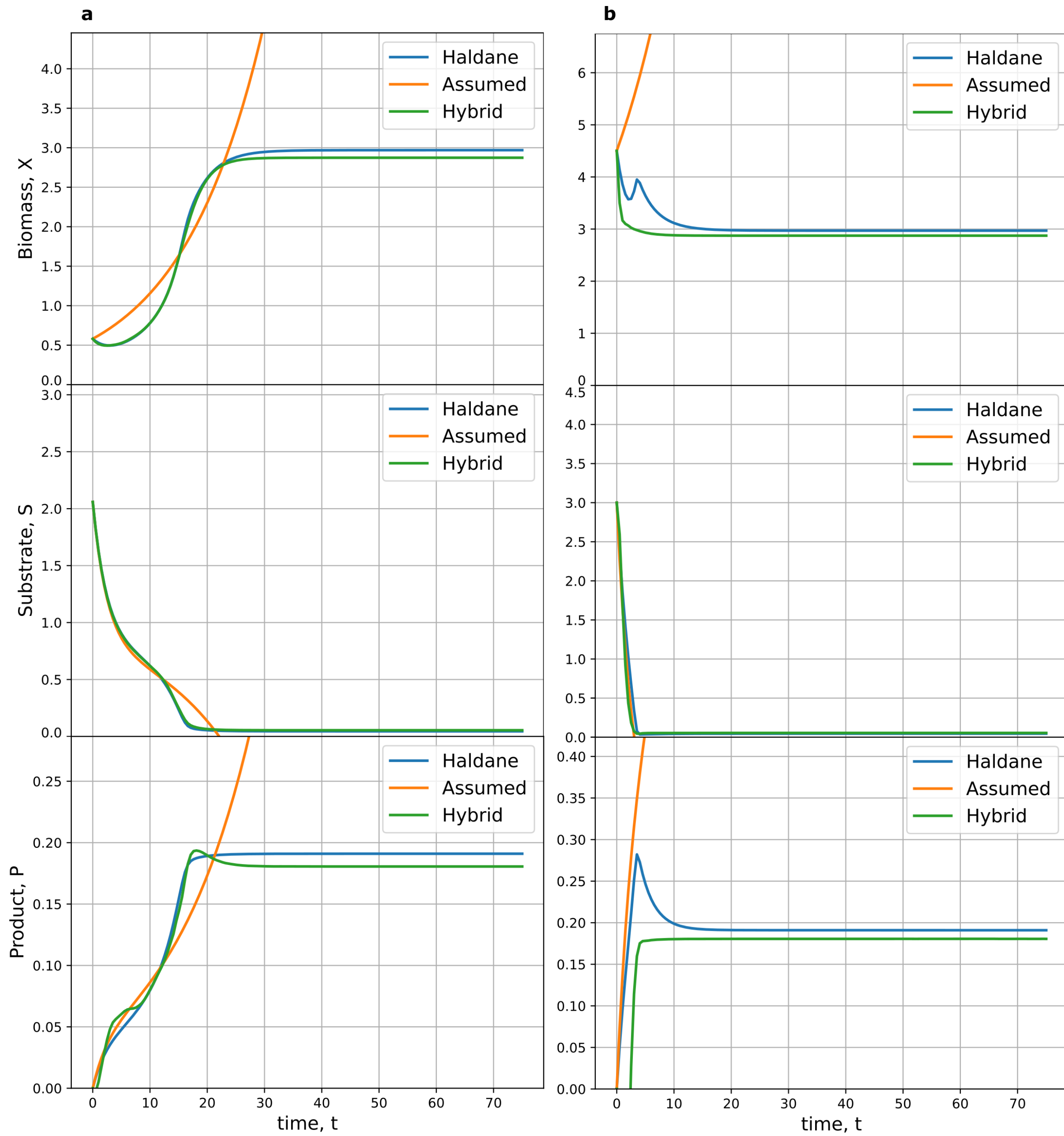

Figure 2.2: Representative results of integrating the true Haldane model Eq. 2.1, the assumed model Eq. 2.2, and the hybrid model Eq. 2.3, using initial conditions that were in the validation set (column a) and neither the training nor validation set (column b). Initial conditions used are specifically listed in Table 2.4.

Variable	Column a	Column b
Biomass, X	0.5804	4.5
Substrate, S	2.059	3.0
Product, P	0.0	0.0

Table 2.4: Initial conditions for results shown in Figure 2.2.

Case 2: noise

In this section we add varying levels of noise to the training data to emulate having some measurement error in the observed data. We first simulate the deterministic equations 2.1. Once we have the trajectories $\vec{x}(t_i)$ from Eq. 2.1, we add noise to it at each time point $\vec{y}(t_i) = \vec{x}(t_i) + \vec{\varepsilon}(t_i)$. $\vec{\varepsilon}(t_i)$ is distributed normally with standard deviation equal to s% of the standard deviation of $\vec{x}$, where below we show results for $s = 1, 5, 10$.

We used the data set $\{\vec{y}(t_i)\}$ with $i = 0, \ldots, 49$ to train the hybrid model Eq. 2.3, and a similarly built validation set for evaluation. A learning curve depicting the training progress for 1% noise level is included in Figure 2.3. To produce points along the training (validation) set loss curve, we took 10 samples randomly from the training (validation) set, and computed the average loss. In Figure 2.3 we can see the opimization procedure converges to a loss value < 6. Moreover, the training and validation set losses are fairly consistent throughout the training procedure, suggesting the model tends to generalize well to initial conditions found in the validation set. Similar learning curves were found for higher noise levels.

Following the previous section, we used the Julia DiffEqParamEstim library to estimate the parameters of Eq. 2.2 from the training set data, Table 2.5 lists the parameters estimated in each of the cases.

Case	p_1	p_2
no noise	0.4689	0.3527
1%	0.4626	0.3362
5%	0.4610	0.3497
10%	0.4621	0.3402

Table 2.5: Parameters estimated from the training data in each case of levels of added noise.

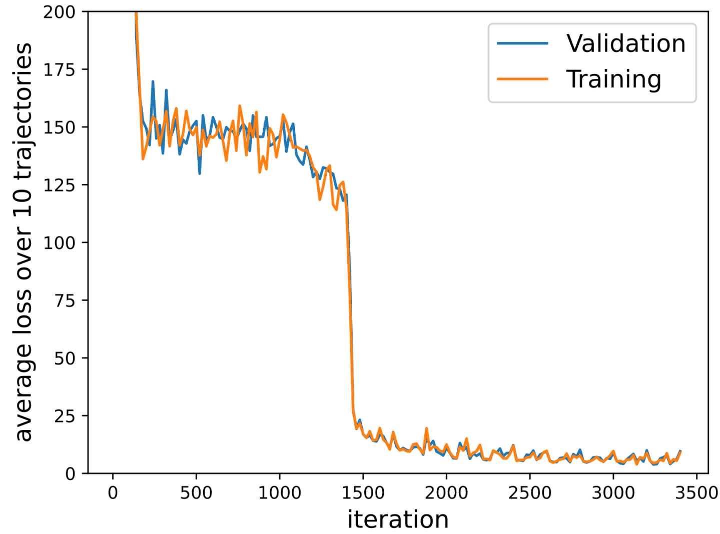

Figure 2.3: Shown is the average loss over 10 trajectories against the optimization step of an ADAM optimizer. Note that initially the loss value begins at ≈ 6000, and begins to converge to ≈ 5. The blue (orange) line represents the average loss over trajectories sampled from the validation (training) set. The amplitude of the noise added to the simulated trajectories is $0.01 \cdot \mathrm{sd}(\vec{x}) \cdot \mathcal{N}(0, 1)$.

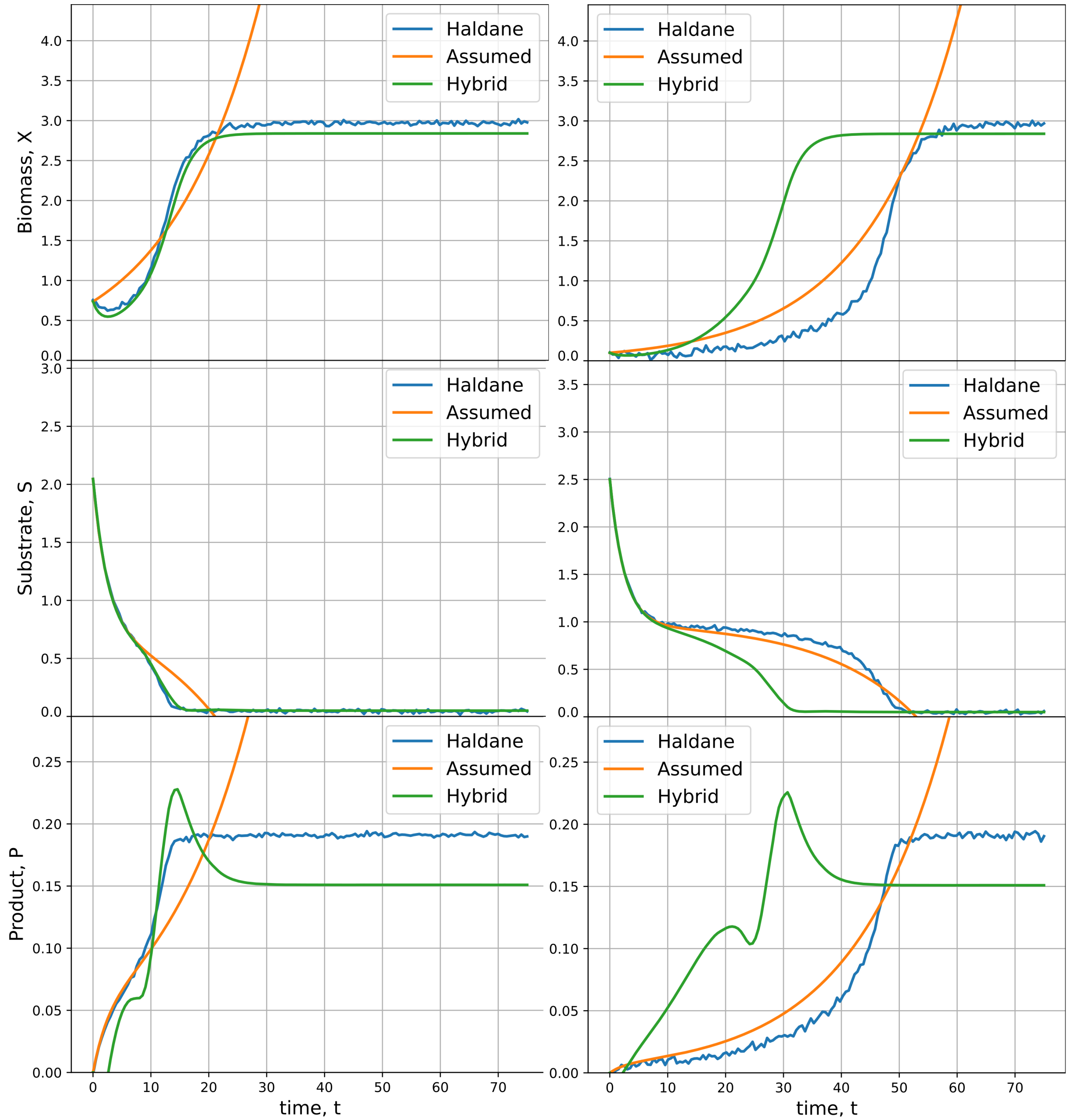

Figure 2.4: Representative results of integrating the true Haldane model Eq. 2.1, the assumed model Eq. 2.2, and the hybrid model Eq. 2.3, using initial conditions that were not used in the training set for the hybrid model. Left and right columns are two different initial conditions. The left (right) column has begins with initial condition close to (far from) the training set. The results of the hybrid model in the right column are worse than in the left.

From Figure 2.4 (left column) and Figure 2.5 we can see that the hybrid model still tends to outperform the assumed, partial model. Moreover, it tends to perform fairly well when given initial conditions within the validation set, which are similar in range to the initial conditions found in the training set. Yet, we find that as the noise level increases, the predictions from the hybrid model seem to have a growing oscillatory nature; Figure 2.4a bottom row, and the bottom row of Figure 2.5 depict this phenomena. Likely, the increase in oscillations with growing added noise is because the optimization procedure is trying to match the noisy 'measurements' exactly.

Moreover, it seems the hybrid model is able to better predict the biomass and substrate dynamics compared to the product dynamics. This limits its usage, as product prediction from new initial conditions (and parameters, but that is not explored here) are desired features of models in biomanufacturing.

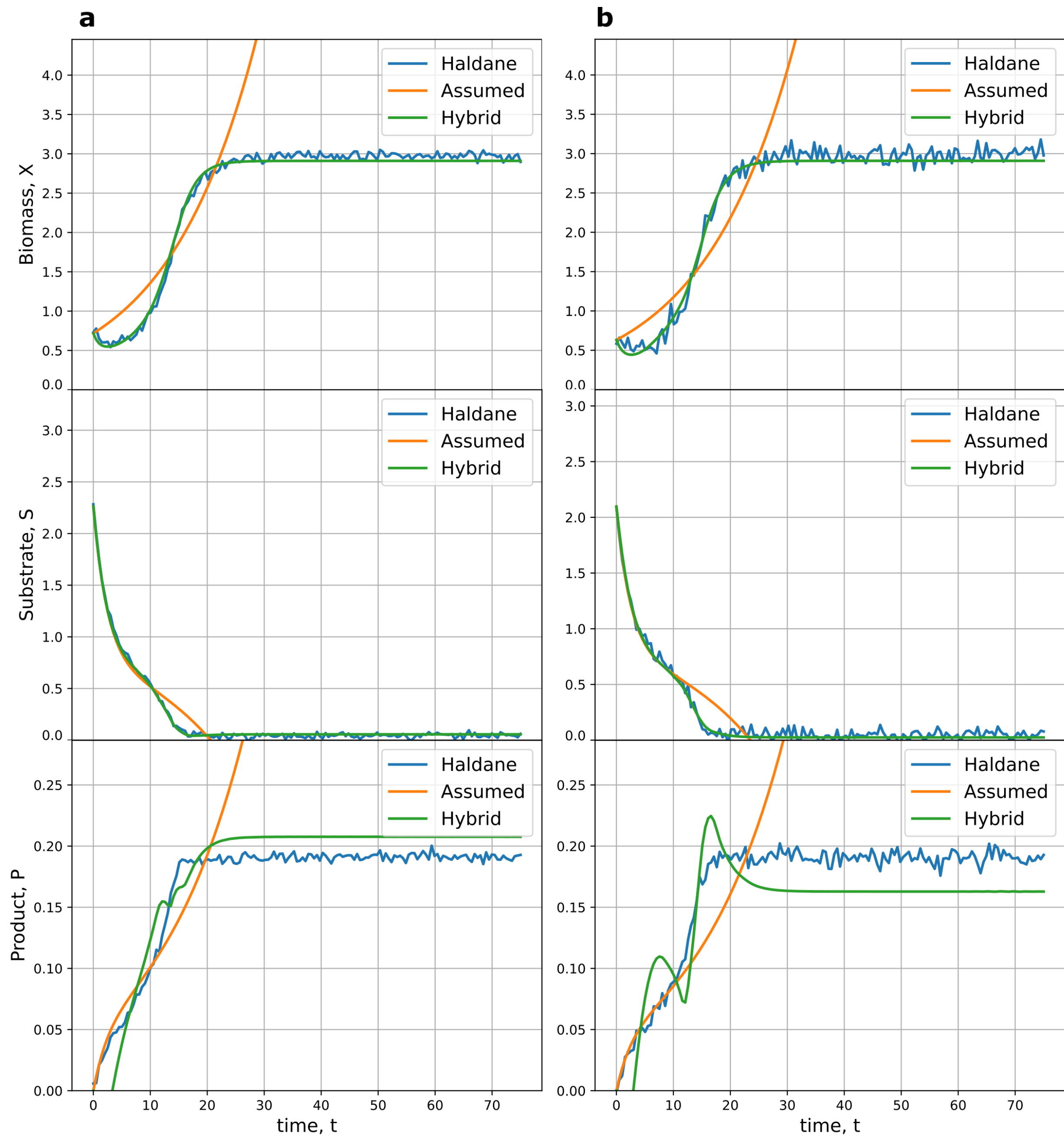

Figure 2.5: Representative results from the validation set when 5% noise is added (column a) and 10% noise is added (column b). With higher noise levels, the hybrid model seems to exhibit more oscillations.

Chapter 3

Conclusion

In this report we discussed usage of combined (or hybrid) mechanistic-machine learning models to fix model mismatch. This was set in the context of specific application to biomanufacturing processes, where the various complexities – transport phenomena diminishing homogeneity throughout the system; poor characterization of cellular processes, manifesting as unexpected phenomena at larger scales; sensors measuring variables related to, but not exactly state variables – result in difficulty designing experiments and designing processes with optimal quality and productivity. We simulated simple models of bioprocesses, then hid factors in the model. Then, using a hybrid neural differential equations, we tried to fix the partial model to re-approximate the true model. This acts as a simulation study of the situation where a modeller derives a system of differential equations to describe a process, but the model is only partially correct. By using historical process data, and online measurements, the partial model can be improved by having a neural network component to the model in the form a universal differential equation.

Overall, we found some evidence the hybrid mechanistic-machine learning models were able to approximate the true system even with few data points, and noisy measurements. Although, we note that the partial model used in this report (Eq. 2.2) may have been too constrained to be a fair comparison with the hybrid model used. One way of improving upon that is to relax the linearity assumption between the rates of growth and synthesis. Compared to previous hybrid modelling studies in the biomanufacturing literature, we also found some evidence that using deeper neural networks tends to improve performance. This suggests that hybrid methods should be attempted on realistic biomanufacturing processes in order to validate the results presented here, and to validate that this technique may be useful in an industrial context.

There are many directions for next steps that could be pursued from these explorations. Those include more simulation studies with more complex 'true' models, performing comparisons to more traditional methods, and comparing to other methods of combining machine learning and mechanistic modelling. Furthermore, an exploration of hybrid model performance sensitivity to amount of data could be performed in order to gauge how sensitive the method is to having few training examples.

References

[1] Canada's Biomanufacturing and Life Sciences Strategy.

[2] Industry 4.0: Embracing digital transformation in bioprocessing.

[3] PAT — A Framework for Innovative Pharmaceutical Development, Manufacturing, and Quality Assurance — FDA.

[4] The Biomanufacturing Facility of the Future.

[5] A. Ajbar. Classification of static behavior of a class of unstructured models of continuous bioprocesses. *Biotechnology Progress*, 17(4):597–605, aug 2001.

[6] Bradley W. Biggs, Hal S. Alper, Brian F. Pfleger, Keith E.J. Tyo, Christine N.S. Santos, Parayil Kumaran Ajikumar, and Gregory Stephanopoulos. Enabling commercial success of industrial biotechnology, dec 2021.

[7] Steven L. Brunton, Joshua L. Proctor, J. Nathan Kutz, and William Bialek. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America*, 113(15):3932–3937, 2016.

[8] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: A review. *Acta Mechanica Sinica/Lixue Xuebao*, pages ppb–ppb, may 2021.

[9] D. Ewen Cameron, Caleb J. Bashor, and James J. Collins. A brief history of synthetic biology, apr 2014.

[10] Ricky T Q Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. Technical report, 2018.

[11] Raj Dandekar, Karen Chung, Vaibhav Dixit, Mohamed Tarek, Aslan Garcia-Valadez, Krishna Vishal Vemula, and Chris Rackauckas. Bayesian Neural Ordinary Differential Equations. dec 2020.

[12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[13] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented Neural ODEs. Technical report, 2019.

[14] Misagh Ebrahimpour, Wei Yu, and Brent Young. Artificial neural network modelling for cream cheese fermentation pH prediction at lab and industrial scales. *Food and Bioproducts Processing*, 126:81–89, mar 2021.

[15] Jarka Glassey and Moritz von Stosch. *Hybrid Modeling in Process Industries*. CRC Press, 2018.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[17] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[18] Alex Graves. Generating Sequences With Recurrent Neural Networks. aug 2013.

[19] Atılım Güne¸, Güne¸s Baydin, Barak A Pearlmutter, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: a Survey. Technical Report 153, 2018.

[20] E. Kaiser, J. N. Kutz, and S. L. Brunton. Sparse identification of nonlinear dynamics for model predictive control in the low-data limit. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 474(2219), nov 2018.

[21] Donald Knuth. *The TeXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[22] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[23] Bing Li, Yinzi Lin, Wei Yu, David I. Wilson, and Brent R. Young. Application of mechanistic modelling and machine learning for cream cheese fermentation pH prediction. *Journal of Chemical Technology and Biotechnology*, 96(1):125–133, jan 2021.

[24] Xuechen Li, Ting-Kam Leonard Wong, Ricky T Q Chen, and David Duvenaud. Scalable Gradients for Stochastic Differential Equations. Technical report, jun 2020.

[25] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: A survey, apr 2021.

[26] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, feb 2021.

[27] Yingbo Ma, Vaibhav Dixit, Mike Innes, Xingjian Guo, and Christopher Rackauckas. A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions. *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021*, dec 2018.

[28] N. M. Mangan, J. N. Kutz, S. L. Brunton, and J. L. Proctor. Model selection for dynamical systems via sparse regression and information criteria. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2204), aug 2017.

[29] Niall M. Mangan, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Inferring Biological Networks by Sparse Identification of Nonlinear Dynamics. *IEEE Transactions on Molecular, Biological, and Multi-Scale Communications*, 2(1):52–63, jun 2016.

[30] Stefano Massaroli, Michael Poli, Atsushi Yamashita, and Hajime Asama. Dissecting Neural ODEs. Technical report, 2020.

[31] Viraj Mehta, Ian Char, Willie Neiswanger, Youngseog Chung, Andrew Nelson, Mark Boyer, Egemen Kolemen, and Jeff Schneider. Neural Dynamical Systems: Balancing Structure and Flexibility in Physical Prediction. In *Proceedings of the IEEE Conference on Decision and Control*, volume 2021-December, pages 3735–3742. Institute of Electrical and Electronics Engineers Inc., 2021.

[32] Harini Narayanan, Michael Sokolov, Massimo Morbidelli, and Alessandro Butté. A new generation of predictive models: The added value of hybrid models for manufacturing processes of therapeutic proteins. *Biotechnology and Bioengineering*, 116(10):2540–2549, oct 2019.

[33] Arlindo L. Oliveira. Biotechnology, Big Data and Artificial Intelligence. *Biotechnology Journal*, 14(8):1800613, aug 2019.

[34] Dimitris C. Psichogios and Lyle H. Ungar. A hybrid neural network-first principles approach to process modeling. *AIChE Journal*, 38(10):1499–1511, oct 1992.

[35] Christopher Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. Diffeqflux.jl - A julia library for neural differential equations. *CoRR*, abs/1902.02376, 2019.

[36] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal Differential Equations for Scientific Machine Learning. jan 2020.

[37] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[38] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. nov 2017.

[39] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. nov 2017.

[40] Ali Ramadhan, John Marshall, Andre Souza, Gregory LeClaire Wagner, Manvitha Ponnapati, and Christopher Rackauckas. Capturing missing physics in climate model parameterizations using neural differential equations. oct 2020.

[41] Sara Sousa Rosa, Duarte M.F. Prazeres, Ana M. Azevedo, and Marco P.C. Marques. mRNA vaccines manufacturing: Challenges and bottlenecks, apr 2021.

[42] Khemraj Shukla, Patricio Clark Di Leoni, James Blackshire, Daniel Sparkman, and George Em Karniadakis. Physics-informed neural network for ultrasound nondestructive quantification of surface breaking cracks. *Journal of Nondestructive Evaluation*, 39(3), may 2020.

[43] Wolfgang Sommeregger, Bernhard Sissolak, Kulwant Kandra, Moritz von Stosch, Martin Mayer, and Gerald Striedner. Quality by control: Towards model predictive control of mammalian cell culture bioprocesses. *Biotechnology Journal*, 12(7):1600546, jul 2017.

[44] Michael L. Thompson and Mark A. Kramer. Modeling chemical processes using prior knowledge and neural networks. *AIChE Journal*, 40(8):1328–1340, aug 1994.

[45] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. Technical Report 1, 1996.

[46] Giang Tran and Rachel Ward. Exact recovery of chaotic systems from highly corrupted data. *Multiscale Modeling and Simulation*, 15(3):1108–1129, jul 2017.

[47] H. J. L. van Can, H. A. B. te Braake, C. Hellinga, and K. C. A. M. Luyben. An efficient model development strategy for bioprocesses based on neural networks in macroscopic balances. *Biotechnology and Bioengineering*, 54(6):549–566, jun 1997.

[48] Moritz von Stosch, Jan Martijn Hamelink, and Rui Oliveira. Hybrid modeling as a QbD/PAT tool in process development: an industrial E. coli case study. *Bioprocess and Biosystems Engineering*, 39(5):773–784, may 2016.

[49] Moritz von Stosch, Jan-Martijn Hamelink, and Rui Oliveira. Toward intensifying design of experiments in upstream bioprocess development: An industrial ¡i¿Escherichia coli¡/i¿ feasibility study. *Biotechnology Progress*, 32(5):1343–1352, sep 2016.

[50] Gary Walsh. Biopharmaceutical benchmarks 2018. *Nature Biotechnology*, 36(12):1136–1145, dec 2018.

[51] Lawrence X. Yu, Gregory Amidon, Mansoor A. Khan, Stephen W. Hoag, James Polli, G. K. Raju, and Janet Woodcock. Understanding pharmaceutical quality by design, 2014.

[52] Steffen Zobel-Roos, Axel Schmidt, Lukas Uhlenbrock, Reinhard Ditz, Dirk Köster, and Jochen Strube. Digital Twins in Biomanufacturing. *Advances in biochemical engineering/biotechnology*, 176:181–262, 2021.