UC Merced
UC Merced Electronic Theses and Dissertations

Title
Block Coordinate Descent Proximal Method for ODE Estimation and Discovery

Permalink
https://escholarship.org/uc/item/8415q4q4

Author
Raziperchikolaei, Ramin

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, MERCED

Block Coordinate Descent Proximal Method for ODE Estimation and Discovery

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Ramin Raziperchikolaei

Committee in charge:

Professor Paul P. Maglio, Chair
Professor Harish S. Bhat
Professor Shawn Newsam
Professor David C. Noelle

2019

Copyright
Ramin Raziperchikolaei, 2019
All rights reserved.

The dissertation of Ramin Raziperchikolaei is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Professor Harish S. Bhat

Professor Shawn Newsam

Professor David C. Noelle

Professor Paul P. Maglio Chair

University of California, Merced

2019

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures vi

Acknowledgements ix

Vita and Publications x

Abstract xi

Chapter 1 Introduction 1
- 1.1 ODE Parameter Estimation 2
- 1.2 Learning Governing Equations 5
- 1.3 Outline 7

Chapter 2 BCD Proximal Method for Simultaneous Filtering and Parameter Estimation 8
- 2.1 Problem Definition: ODE Parameter Estimation 8
- 2.2 Iteratively Refined Principal Differential Analysis (iPDA) 9
- 2.3 Our BCD-prox Approach 10
  - 2.3.1 Optimization over the ODE parameters 12
  - 2.3.2 Optimization over the states $\mathbf{X}$ 12
  - 2.3.3 Higher-Order discretization 14
- 2.4 Convergence 15
- 2.5 Experiments 19
- 2.6 Conclusion 33

Chapter 3 Interpretable Equation Discovery with Neural Networks 34
- 3.1 Problem Definition 34
- 3.2 Our Proposed Method 35
  - 3.2.1 Parameterization of the f_j 35
  - 3.2.2 Architecture of the neural network 36
  - 3.2.3 Extending the idea to the d dimensions of $\mathbf{f}$ 38
  - 3.2.4 Objective function and optimization 40
  - 3.2.5 Overfitting in learning $\mathbf{f}$ and how to avoid it 43
  - 3.2.6 Higher-order discretization 44
- 3.3 Experiments 44
- 3.4 Conclusion 53

References 57

LIST OF FIGURES

Figure 1.1: An example of ODE parameter estimation problem. *Left:* the number of predators (lynx) and prey (snowshoe hare) over 90 years. *Right:* the number of predators and prey by simulation of the Lotka–Volterra model in (1.2). 2
Figure 1.2: An example of a highly nonlinear dynamical system, where we need to learn governing equations. This figure shows the trajectory of a fruit fly in a wind tunnel which contains an attractive food odor. 5

Figure 2.1: Prediction error at different iterations of our algorithm. Noisy observations are achieved by adding Gaussian noise with variance σ^2 to the clean observations. We consider the FitzHugh–Nagumo (first row), Rössler (second row), and Lorenz-96 (third row) models. Our learning strategy decreases the error in all cases. 22
Figure 2.2: Robustness to the hyperparameter λ in FitzHugh–Nagumo (first row) and Lotka–Volterra (second row) models. The true parameters are $\theta_0 = .5, \theta_1 = .3$, and $\theta_2 = 3$ in the FitzHugh–Nagumo and $\theta_0 = 2, \theta_1 = 1, \theta_2 = 4$, and $\theta_3 = 1$ in the Lotka–Volterra. For each λ, we report the mean error and parameter value in 10 experiments. 23
Figure 2.3: Robustness to different types and amounts of noise in the observations. We report the prediction and parameter errors on the Rössler and FitzHugh–Nagumo models. 25
Figure 2.4: Comparison with other methods on FitzHugh–Nagumo model. We create initializations by adding Gaussian noise of variance $\sigma_{\boldsymbol{\theta}}^2$ to the true parameters. We create 10 sets of observations and initializations per each $\sigma_{\boldsymbol{\theta}}^2$ and report the errors. Each error bar corresponds to the error in one of the experiments. 27
Figure 2.5: Comparison with other methods on Rössler attractor. The settings are the same as the Fig. 2.4 28
Figure 2.6: Comparison with the mean-field method [19] on Lotka–Volterra model. We add Gaussian noise with variance σ^2 to the clean states to create noisy observations. Each error bar corresponds to the error in one of the experiments. 29

Figure 2.7: Comparison with the extended Kalman filter (EKF) on the Lotka–Volterra model. We add Gaussian noise with variance $\sigma^2 = 0.1$ and $\sigma^2 = 1.5$ to the clean states to create noisy observations. We set the number of observation to $T = 20$ (left panel) and $T = 10\,000$ (right panel). For each value of T, we generate 10 sets of observations and report the average estimation and parameter errors. Each error bar corresponds to the error in one of the experiments. We have put the average error of each method for the 10 experiments below each plot. . 32

Figure 3.1: Architecture of our neural network. The first layer takes as input the scalar x_k, the kth component of $\mathbf{x}$. The output of the Dth shared (hidden) layer is B 1-dim shape functions for each dimension (a total of Bd outputs). The multiplication layer only contains multiplication neurons. It takes the Bd inputs and generates multi-dim shape functions. The last layer has d neurons with identity activation functions. Each neuron corresponds to one of the dimensions. 39

Figure 3.2: Impact of the number of epochs on the overall performance of state prediction. We train the neural network for 10, 500, and 1000 epochs, before switching to the step over learning the states. We report the error as the total number of epochs increases on FitzHugh–Nagumo and Rössler attractor models with 5% noise. 47

Figure 3.3: Avoiding the overfitting and underfitting without fixing the number of epochs. We let the neural network's training continue, as long as the current predicted states get closer to the noisy observations. We report the error as the total number of epochs increases on FitzHugh–Nagumo and Rössler attractor models with 5% noise. 48

Figure 3.4: Robustness to the hyperparameter λ. We create 10 sets of observations, each with 1%, 5%, and 10% noise. We run our algorithm with $\lambda = 0.01, 0.1$, and 1 on each set separately and report the error (each bar shows the error for one of the experiments). 49

Figure 3.5: Importance of learning the clean states. Learning the states significantly improves the results. "learn_states" is our approach, while "fix_states" fixes the states to the observations and fits the network. We run the experiments on the two models with 5% noisy observations. *Left panel:* prediction error at different epochs of the methods. *Right panel:* The Euclidean distance between the current learned states and the clean states, during the alternating optimization. 50

Figure 3.6: Comparison with other methods. We compare our method with Raissi et al. [41], Rudy et al. [46], and Brunton et al. [6] on three models with different amounts of noise in the observations. For each value of the noise percentage, we create 10 sets of observations, run the methods on each set separately, and report the prediction error. Each error bar shows the error in one experiment. 51
Figure 3.7: Visualization of the predicted states and the shape functions on Rössler ODE. We generate observations with 5% noise. *First panel:* the clean states and our predicted states over time. *Second panel:* visualization of the 1-dim shape functions. *Third panel:* visualization of the multi-dim shape functions. 54
Figure 3.8: Visualization of the predicted states and the shape functions on FitzHugh–Nagumo ODE. The panels are the same as Fig. 3.7 55
Figure 3.9: Visualization of the predicted states and the shape functions on double pendulum ODE. The panels are the same as Fig. 3.7 56

ACKNOWLEDGEMENTS

First, I would like to thank Prof. Paul Maglio. He was very supportive when I decided to switch my advisor and spent a tremendous amount of time and energy to make sure that I won't be hurt. I will always be grateful for what he did.

I would also like to thank Prof. Harish Bhat. He was very supportive at the time of switching the advisor, taught me a lot of things, and gave me the courage of exploring new ideas. I enjoyed every moment of our collaboration during the past year.

I would like to thank my committee members, Prof. Shawn Newsam and Prof. David Noelle for the time and effort they spent to serve as my committee members.

I appreciate the support from the school of engineering by offering multiple Bobcat and travel fellowships.

My friends made this journey a pleasant experience for me! I had constant support from my old friends Bahman and Joobin during these years. I had fantastic poetry nights at Merced with Arash, Dorna, Rayan, MohammadKazem, Ali, and Maryam. I greatly cherish the time I spent with Negin, Yousef, and Ariana.

I also want to thank my colleagues in the lab, Mehdi, Maksym, and MohammdaKazem. MohammdaKazem has my special thanks for the great discussions that we had on many research problems and for his help in finishing this dissertation.

There aren't enough words to express how grateful I am to my family and my grandparents. My parents, Mohsen and Fatemeh, made education my first priority and kept me on the road to academic success up until now. My brother, Samin, supported me in many difficult situations and helped me to stay in the path. My sister, Nazanin, always gave me hope, positive energy, and support in my life. This dissertation would not have been possible without you!

Last but not least, I would like to thank my wife, Maryam. She was involved in all parts of my research and Ph.D. life, helped and supported me patiently, and have been my best friend. I could not finish my study without her. We got married in Merced and made a lot of wonderful memories here. I can not wait to open a new chapter of life together.

VITA

2010	B. Sc. in Computer Engineering, Iran University of Science and Technology, Tehran, Iran.
2012	M. Sc. in Artificial Intelligence, Sharif University of Technology, Tehran, Iran. Advisor: Prof. Mansour Jamzad
2019	Ph. D. in Electrical Engineering and Computer Science, University of California, Merced. Advisor: Prof. Paul P. Maglio

PUBLICATIONS

Ramin Raziperchikolaei and Harish S. Bhat, *A Direct Method to Learn States and Parameters of Ordinary Differential Equations*, arXiv:1810.06759 [cs.LG] (In submission), 2019.

Ramin Raziperchikolaei and Miguel Á. Carreira-Perpiñán, *Learning Circulant Support Vector Machines for Fast Image Search*, In IEEE Int. Conf. Image Processing (ICIP), 2017.

Ramin Raziperchikolaei and Miguel Á. Carreira-Perpiñán, *Learning Supervised Binary Hashing: Optimization vs Diversity*, In IEEE Int. Conf. Image Processing (ICIP), 2017.

Ramin Raziperchikolaei and Miguel Á. Carreira-Perpiñán, *Learning Independent, Diverse Binary Hash Functions: Pruning and Locality*, In IEEE Int. Conf. Data Mining (ICDM), 2017.

Ramin Raziperchikolaei and Miguel Á. Carreira-Perpiñán, *Optimizing Affinity-Based Binary Hashing Using Auxiliary Coordinates*, In Advances in Neural Information Processing Systems (NIPS), 2016.

Miguel Á. Carreira-Perpiñán and Ramin Raziperchikolaei, *An Ensemble Diversity Approach to Supervised Binary Hashing*, In Advances in Neural Information Processing Systems (NIPS), 2016.

Miguel Á. Carreira-Perpiñán and Ramin Raziperchikolaei, *Hashing with binary autoencoders*, In IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), 2015.

Ramin Raziperchikolaei and Mansour Jamzad, *Visual tracking using D2-clustering and particle filter*, In IEEE Int. Symposium on Signal Processing and Information Technology (ISSPIT), 2012.

Ali Bagheri-Khaligh and Ramin Raziperchikolaei and Mohsen Ebrahimi Moghaddam, *A new method for shot classification in soccer sports video based on SVM classifier*, In IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), 2012.

ABSTRACT OF THE DISSERTATION

Block Coordinate Descent Proximal Method
for ODE Estimation and Discovery

by

Ramin Raziperchikolaei
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California Merced, 2019

Professor Paul P. Maglio, Chair

Ordinary differential equations (ODE) are used extensively in science and engineering to model dynamical systems. In this dissertation, we use noisy observations to address two learning tasks regarding dynamical systems. The first one is called ODE parameter estimation, where the shape of the ODEs are known, and we try to learn (estimate) the parameters. The second task is called learning the governing equations, where we learn both the shape and the parameters of the ODEs.

In the first part of this dissertation, we address the ODE parameter estimation problem. We propose and analyze a block coordinate descent proximal algorithm (BCD-prox) for simultaneous filtering and parameter estimation of ODE models. The main idea is to learn the states and the parameters in an alternation, where the states are restricted to change slowly from one iteration to the next. As we show on ODE systems with up to $d = 40$ dimensions, as compared to state-of-the-art methods, BCD-prox exhibits increased robustness (to noise, parameter initialization, and hyperparameters), decreased training times, and improved accuracy of both filtered states and estimated parameters. We show how BCD-prox can be used with multistep numerical discretizations, and we establish convergence of BCD-prox under hypotheses that include real systems of interest.

In the second part of this dissertation, we address the problem of learning the governing equations given the noisy observations. While we use powerful neural networks to learn the ODEs, we propose a novel structure that makes our network interpretable. The idea is to use the neural network to learn a set of one-dimensional and multi-dimensional shape functions, whose linear combinations give use the equations. To make our method robust to the noise in the observations, we learn the clean states and the parameters of the network simultaneously using the block coordinate descent proximal algorithm (BCD-prox). As we show in our experiments, our method is robust to its hyperparameter, robust to the noise, and outperforms the state-of-the-art methods by achieving more accurate state predictions.

Chapter 1

Introduction

Finite-dimensional dynamical systems form a cornerstone of mathematical modeling in the sciences and engineering. Usually, these systems take the form of systems of ordinary differential equations (ODEs). ODEs help us to understand, describe, and predict the behavior and evolution of a given process, and the way the current states of the system affect the future states. The dynamical systems and their differential equations have been used widely in different areas, such as biology, chemistry, ecology, genetics, etc.

In some cases, it might be possible to derive the ODEs from the first principles using the data and some knowledge of the system. In these cases, it is extremely common to set the parameters of the ODEs using the observed data. Our first goal in this dissertation is to solve this ODE parameter estimation problem, which means learning the parameters given the noisy observations.

In other cases, the system is so complex that it is impossible to use the first principles to come up with the ODEs. The second goal of this dissertation is to learn interpretable functions to accurately model the complex dynamical systems using noisy observations.

We first define some notations and then get into the details of each of the two problems in the next subsections. Let us consider a dynamical system in $\mathbb{R}^d$ with state

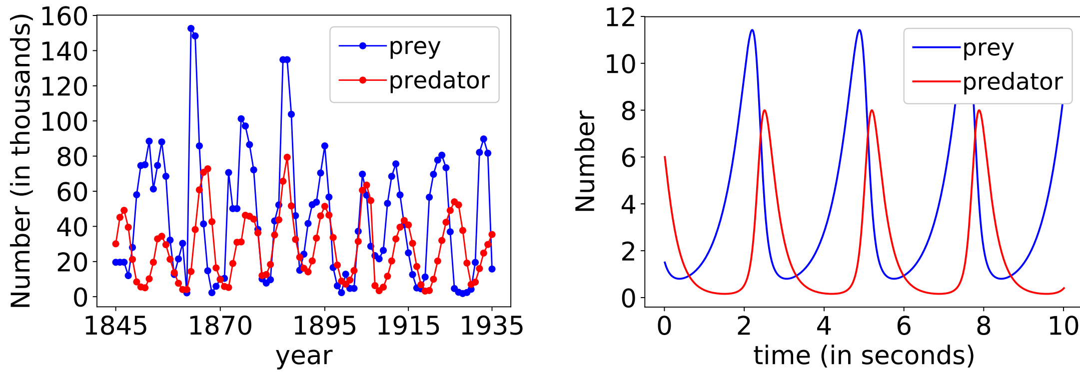

Figure 1.1: An example of ODE parameter estimation problem. *Left:* the number of predators (lynx) and prey (snowshoe hare) over 90 years. *Right:* the number of predators and prey by simulation of the Lotka–Volterra model in (1.2).

$\mathbf{x}(t)$ at time t. The time-evolution of the state is given by

$$\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t); \boldsymbol{\theta}). \tag{1.1}$$

where $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ is a vector field with parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. At T distinct times $\{t_i\}_{i=1}^T$, we are given the noisy observations $\mathbf{y}(t_i) \in \mathbb{R}^d$. Note that we do not have access to the clean data.

1.1 ODE Parameter Estimation

As mentioned before, in some cases, it is possible to derive the ODEs from the first principles, where the parameters have to be estimated from the data. We explain this problem using the predator-prey example. Assume there are only two types of animals in a closed eco-system: predator and prey. Obviously, there is a relation between the number of predators and prey at a specific time with their numbers at a later time. One of the classic studies of predator-prey interaction is the 90-year dataset of snowshoe hare and lynx pelts purchased by the Hudson's Bay Company of Canada [16]. The left plot of Fig. 1.1 shows the number of predators (blue) and prey (red) over 90 years, which have been reported in this dataset.

Several differential equations have been proposed to describe this dynamical system. One of the popular ones is the Lotka–Volterra [30] equations. Assume x_1 represents the number of predators and x_0 represents the number of prey. The Lotka–Volterra model contains the following two nonlinear equations:

$$\frac{dx_0}{dt} = \theta_0 x_0 - \theta_1 x_0 x_1 \qquad \frac{dx_1}{dt} = \theta_2 x_0 x_1 - \theta_3 x_1. \tag{1.2}$$

Note that the four parameters $\theta_0, \ldots, \theta_3$ are unknown and one needs to estimate them using the noisy observations. That is why we call this the ODE parameter estimation problem.

To see why the Lotka–Volterra model in (1.2) is a reasonable ODE for this problem, we run the simulation and generate the number of predators and prey using the ODEs in (1.2). We set the parameters to $\theta_0 = 2, \theta_1 = 1, \theta_2 = 4$, and $\theta_3 = 1$ and start from $x_0 = 1.5$ and $x_1 = 6$. We show the results in the right curve of Fig. 1.1. As we can see, the generated (simulated) data looks similar to the real data (look at the oscillations of the curves and the lag of the predator curve).

The final goal here is to set the parameters $\boldsymbol{\theta}$ such that the simulated data from the ODEs becomes as close as possible to the clean data. This problem is challenging for two main reasons: (1) the data is noisy (no access to the clean data), and (2) most ODEs do not have an analytical solution. Without these two challenges, the parameters could be estimated by solving a simple regression problem.

Literature review. There have been several different approaches to estimate ODE parameters. Nonlinear least squares methods start with an initial guess for the parameters that is iteratively updated to bring the model's predictions close to measurements [4, 5, 22, 23]. These methods are slow and have convergence issues when the initial parameters are far from the true parameters.

Varah [54] first suggested to fit splines to the noisy observations, and then consider these splines as the clean states. Since the derivatives of the splines can be found easily, the parameters are estimated by solving a regression problem. The idea of fitting splines and other smooth functions to the observations has been used extensively in the literature [9, 10, 14, 20, 28, 38, 43].

The idea of learning the splines and parameters jointly, leading to better results, has been explored [38, 43]. Spline-based methods have many hyperparameters that require careful tuning (such as the smoothing parameter, the number of knots, the positions of the knots, etc.), and are also sensitive to initialization.

Bayesian approaches have also been very popular recently [8, 15, 18, 19]. Such approaches typically have large training times and depend sensitively on a large number of hyperparameters (priors, noise variances, etc.). Another disadvantage of these methods, as has been mentioned in [19], is that they cannot simultaneously learn clean states and parameters. In [19], a variational inference approach has been used to overcome this problem, but the method is not applicable to all ODEs. Often, Bayesian methods make multiple assumptions about the distribution of the data and noise.

Finally, let us mention that the problem we solve here is that of simultaneous filtering (recovering clean states from noisy observations) and parameter estimation. Many well-known nonlinear ODE filtering methods, including extended and ensemble Kalman filters as well as particle filters, are online methods that make Gaussian assumptions.

Summary of our work. Motivated by recent advances in alternating minimization [11, 27, 57], block coordinate descent (BCD) [56], and proximal methods [37, 52], we study a BCD proximal algorithm (BCD-prox) to solve the simultaneous filtering and parameter estimation problem. Here filtering means recovering clean ODE states from noisy observations. BCD-prox works by minimizing a unified objective function that directly measures how well the states and parameters satisfy the ODE system, in contrast to other methods that use separate objectives. BCD-prox learns the states directly in the original space, instead of learning them indirectly by fitting a smoothed function to the observations. Under hypotheses that include systems of real interest, BCD-prox is provably convergent. In comparison with other methods, BCD-prox is more robust with respect to noise, parameter initialization, and hyperparameters. BCD-prox is also easy to implement and runs quickly.

BCD-prox learns parameters and states jointly, but it does not fit a smooth function to the observations. Via this approach, BCD-prox reduces the number of hyperparameters to one. BCD-prox avoids assumptions (i.e., spline or other smooth estimator)

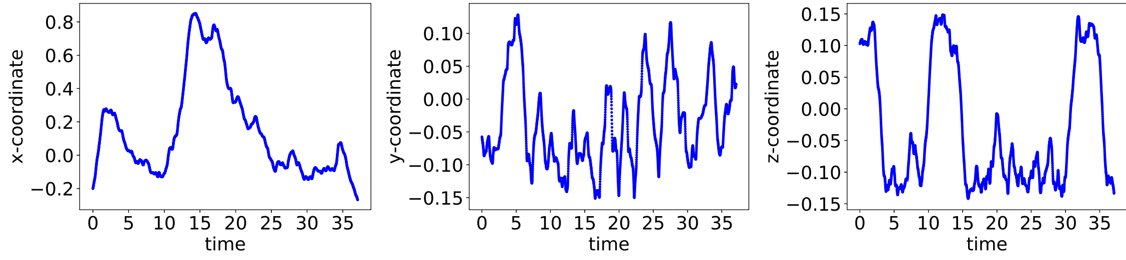

Figure 1.2: An example of a highly nonlinear dynamical system, where we need to learn governing equations. This figure shows the trajectory of a fruit fly in a wind tunnel which contains an attractive food odor.

regarding the shape of the filtered states. Furthermore, both the BCD and proximal components of the algorithm enable it to step slowly away from a poor initial choice of parameters. In this way, BCD-prox remedies the problems of other methods.

1.2 Learning Governing Equations

In many cases, it is impossible to use first principals to come up with the ODEs of the complicated dynamical systems. We have shown one example of such systems in Fig. 1.2 [36]. This figure shows the trajectory of a real fruit fly in a wind tunnel which contains an attractive food odor [36]. The goal is to find out how the fruit flies behave and make decisions in the presence of the odors. Fig. 1.2 shows the (x, y, z) position of a fruit flight over time. This is a complicated and highly nonlinear dynamical system. The goal is to use machine learning techniques to learn the governing equations given the noisy observations.

Literature review. The problem of automatically generating ODE models from time series has attracted significant recent interest [7, 12, 39, 40, 48–50, 53]. Several methods try to learn interpretable models [6, 21, 31, 32, 45, 47, 55]. The most popular work in this category is called SINDy and proposed by Brunton et al. [6]. The idea is to first create a large library of candidate nonlinear functions of the observations. This library may contain constant, polynomial, trigonometric, etc., functions. Then, they set up a sparse regression

problem to select the functions. The main issue with this approach is that since the library is pre-determined it cannot learn complicated models. Another issue is how to choose the functions in the library.

The non-interpretable methods use black-box machine learning techniques to model the complex systems [41, 42, 46]. These methods have, like our method, employed neural networks. Raissi et al. [42] first applies multi-step methods to discretize the ODE (1.1) in time. Then, it replaces the vector field $\mathbf{f}$ by a neural network and learns its parameters by minimizing the distance between the two sides of the time-discretized version of the ODE formulation in (1.1). Rudy et al. [46] uses the same objective function as [42]. The only difference is that to make it more robust to the noise, they optimize the objective over both the parameters of the network and the states, and add a regularization term to keep the states close to the noisy observations. However, both methods work well primarily when used with time series with zero to low levels of noise.

Summary of our work. We develop a method to automatically construct ODE models from noisy time series. Though we employ neural networks to parameterize the ODE's right-hand side (or vector field), we constrain the networks' architecture in such a way that the resulting model is interpretable. Essentially, the network uses training data to learn an appropriate family of one-dimensional shape functions whose tensor products can be combined linearly to accurately represent the vector field. These learned one-dimensional shape functions can be visualized and understood in the same way that we understand, e.g., Bessel, Hermite, and other special functions that arise naturally in scientific contexts.

We define our objective function based on the time-discretized formulation of the ODEs in (1.1). We optimize the objective function over the states and the parameters of the neural network alternatively. We have built into our alternating minimization algorithm a proximal step that enables accurate modeling for moderate to highly noisy time series. Over the course of many proximal steps, the algorithm filters its input (noisy time series), producing as output both a reconstruction of the system's clean states together with the system's vector field. Reproducible numerical experiments clearly show that our method outperforms three competing methods [7, 41, 42, 46].

1.3 Outline

The dissertation is structured in the following way. In chapter 2, we focus on the ODE parameter estimation problem. In section 2.1, we define the problem mathematically and define the notations that we use throughout the chapter. In section 2.2 we give the issues of the one the most important previous works, which is called iteratively Refined Principal Differential Analysis (iPDA). In section 2.3, we motivate our solution, define the objective function, propose a new way to optimize it, mention the advantages of our method over the iPDA approach, and explain how to apply our algorithm using higher-order discretizations. In section 2.4, we discuss the convergence of our algorithm from the practical and theoretical point of views. In section 2.5, we introduce several ODEs, define the evaluation metrics, run several experiments to investigate the robustness of our algorithm to the amount of noise and the hyperparameters, and compare our method with the state-of-the-art methods.

In chapter 3, we focus on learning the governing equations. In section 3.1, we define the problem mathematically and define the notations that we use throughout the chapter. In section 3.2 we explain how to parameterize the vector field in an interpretable manner, we show how to model the parameterized vector field using the neural networks, we give the details of the objective function and the optimization steps, and we discuss how to use higher-order discretizations in our algorithm. In section 3.3, we introduce several ODEs, define the evaluation metrics, run several experiments to investigate the robustness of our algorithm to the amount of noise and the hyperparameters, and compare our method with the state-of-the-art methods. At the end of this section, we visualize some of the shape functions learned by our network to confirm its interpretability.

Chapter 2

BCD Proximal Method for Simultaneous Filtering and Parameter Estimation

2.1 Problem Definition: ODE Parameter Estimation

Consider a dynamical system in $\mathbb{R}^d$ with state $\mathbf{x}(t)$ at time t. We assume the system depends on a parameter $\boldsymbol{\theta} \in \mathbb{R}^p$, in which case the time-evolution of the state is given by

$$\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t); \boldsymbol{\theta}). \tag{2.1}$$

At T distinct times $\{t_i\}_{i=1}^T$, we have noisy observations $\mathbf{y}(t_i) \in \mathbb{R}^d$:

$$\mathbf{y}(t_i) = \mathbf{x}(t_i) + \mathbf{z}(t_i), \quad i = 1, \ldots, T \tag{2.2}$$

where $\mathbf{z}(t_i) \in \mathbb{R}^d$ is the noise of the observation at time t_i. We represent the set of T d-dimensional states, noises, and observations by $\mathbf{X}, \mathbf{Z}$, and $\mathbf{Y} \in \mathbb{R}^{d \times T}$, respectively. For concision, in what follows, we write the time t_i as a subscript, i.e., $\mathbf{x}_{(t_i)}$ instead of $\mathbf{x}(t_i)$.

In this chapter, we assume that the form of the vector field $\mathbf{f}(\cdot)$ is known. *The goal is to use* $\mathbf{Y}$ *to estimate* $\boldsymbol{\theta}$ *and* $\mathbf{X}$ *(or equivalently* $\mathbf{Z}$*).*

2.2 Iteratively Refined Principal Differential Analysis (iPDA)

One of the most successful approaches in learning ODE parameters is iPDA [38, 43]. Since this approach is relevant to what we are proposing, we explain iPDA and its issues here.

The main idea of iPDA is to learn the states $\mathbf{X}$ and the parameters $\boldsymbol{\theta}$ jointly. It has the following objective function:

$$\min_{\mathbf{x}_{(t)},\boldsymbol{\theta}} \int \left\| \frac{d\mathbf{x}_{(t)}}{dt} - \mathbf{f}(\mathbf{x}_{(t)};\boldsymbol{\theta}) \right\|^2 dt + \lambda \sum_i \left\| \mathbf{x}_{(t_i)} - \mathbf{y}_{(t_i)} \right\|^2, \tag{2.3}$$

where $\mathbf{x}_{(t)}$ is a smooth spline, which is initialized by fitting the spline to the set of observations. The left term of the objective function is the parameter estimation term, and the right term is the regularization term. The objective function in (2.3) is minimized in the following two steps, iteratively:

1. Optimize over $\mathbf{x}_{(t)}$ given the parameters $\boldsymbol{\theta}$. This will be achieved by learning a smooth spline.

2. Optimize over $\boldsymbol{\theta}$ by fixing $\mathbf{x}_{(t)}$. Note that the optimization only includes the parameter estimation term since the regularization term does not depend on $\boldsymbol{\theta}$.

The main issue with the objective function in (2.3) is the regularization term. This term determines how far the clean states are going to be from noisy observations. If we set λ to a large value, then $\mathbf{x}_{(t)}$ remains close to the $\mathbf{y}_{(t)}$, and this could introduce a large error for the estimation term. If we set λ to a small value, then $\mathbf{x}_{(t)}$ can find a better local optima for the estimation term, but it could get far away from the noisy observation. It is a challenging task to set λ to the right value for two reasons: 1) this value should change every time the amount of noise or the type of ODE changes, and 2) in ODE parameter estimation, we do not have access to the clean data (all we have is the noisy observations), so we cannot find the right λ by cross validation. We get to this point later when we explain our proposed approach.

Another (less significant) problem with iPDA is that (similar to all spline-based methods) it needs to set a lot of hyperparameters carefully to achieve reasonable results.

2.3 Our BCD-prox Approach

In this chapter, we propose a method to address both issues of the iPDA approach. The first step of our approach is to discretize the ODE (2.1) in time using multistep methods. In this section, we focus on the explicit Euler method (which is a one-step method) because it is intuitive, makes our formulation simple, and helps the reader to focus purely on our novel approach. Later, we explain how higher-order multistep methods can be used in our formulation. We also compare multistep methods with different orders in our experimental results section.

The explicit Euler method discretizes the ODE (2.1) for the T time points as follows:

$$\mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)} = \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})\Delta_i, \quad i = 1, ..., T-1 \tag{2.4}$$

where $\Delta_i = t_{i+1} - t_i$. In (2.4), both states $\mathbf{X}$ and parameters $\boldsymbol{\theta}$ are unknown; we are given only the noisy observations $\mathbf{Y}$. With this discretization, we formulate our objective function:

$$E(\mathbf{X}, \boldsymbol{\theta}) = \sum_{i=1}^{T-1} \left\| \mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)} - \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})\Delta_i \right\|^2 \tag{2.5}$$

Before continuing, it is worth analyzing our objective function E. Let us define *fidelity* as the degree to which the estimated states $\mathbf{X}$ and parameters $\boldsymbol{\theta}$ actually satisfy the ODE. Our objective function directly measures time-discretized fidelity; if $E = 0$, then we have a solution to the time-discretized ODE. Note that the observations $\mathbf{Y}$ do not appear explicitly—we use $\mathbf{Y}$ to initialize Alg. (2.1).

In contrast, most prior work maximizes the likelihood function that stems from assuming the noise $\mathbf{Z}$ in (2.2) is Gaussian with mean zero and covariance matrix Σ. In such approaches, fidelity (as defined above) is treated as a secondary term, e.g., using a penalty term [43] or using a probabilistic model that accounts for temporal discretization errors [2, 3]. In the present work, we seek to show that making fidelity the primary objective yields better estimates of $\mathbf{X}$ and $\boldsymbol{\theta}$.

Theorem 2.1. *The objective function E in* (2.5) *has an infinite number of optimal solutions, each of which makes the objective value* 0*.*

Proof. Assign arbitrary real vectors to $\boldsymbol{\theta}$ and $\mathbf{x}_{(t_1)}$. Then we use the following equation sequentially with $i = 1, 2, \ldots, T-1$:

$$\mathbf{x}_{(t_{i+1})} = \mathbf{x}_{(t_i)} + \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})\Delta_i. \tag{2.6}$$

By computing the states $\mathbf{x}_{(t_2)}, \ldots, \mathbf{x}_{(t_T)}$ in this way, we ensure that each term in the objective function $E(\mathbf{X}, \boldsymbol{\theta})$—see (2.5)—is zero. Since the objective function in (2.5) is always greater than or equal to zero, we achieve a global minimum. Because $\boldsymbol{\theta}$ and $\mathbf{x}_{(t_1)}$ are arbitrary, an infinite number of solutions exist. □

It is easy to see that Theorem 2.1 generalizes to the case where we replace the Euler method—in the definition of $E(\mathbf{X}, \boldsymbol{\theta})$—by a higher-order, explicit ODE solver.

Additionally, suppose we fix $\boldsymbol{\theta}$ and consider E to be a function of $\mathbf{X}$ alone. Then, if we also fix the value of $\mathbf{x}_{(t_1)}$, we see that there is a unique global minimizer. We return to this point below when we discuss minimizing over $\mathbf{X}$.

Now the question is how to optimize (2.5) to end up in a global or local optimum which gives us a good estimate of the true parameters and states. We propose to optimize (2.5) iteratively and let the states change slowly from one iteration to the next. Specifically, we define the following Euler BCD-proximal objective function:

$$\mathbf{X}^{*(n)}, \boldsymbol{\theta}^{*(n)} = \underset{\boldsymbol{\theta}, \mathbf{X}}{\operatorname{argmin}} \left\{ E(\mathbf{X}, \boldsymbol{\theta}) + \lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|^2 \right\}, \quad \mathbf{X}^{*(0)} = \mathbf{Y}, \tag{2.7}$$

where we represent the states and parameters *learned* at iteration n by $\mathbf{X}^{*(n)}$ and $\boldsymbol{\theta}^{*(n)}$. We use a one-step alternating optimization to learn the states and parameters at each iteration n. To be more specific, we repeat the following until convergence:

$$\text{learning:} \quad \boldsymbol{\theta}^{*(n)} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left\{ E(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}) \right\} \tag{2.8a}$$

$$\text{filtering:} \quad \mathbf{X}^{*(n)} = \underset{\mathbf{X}}{\operatorname{argmin}} \left\{ E(\mathbf{X}, \boldsymbol{\theta}^{*(n)}) + \lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|^2 \right\}. \tag{2.8b}$$

Now we can explain why we call our approach BCD-prox. Repeating the two steps in (2.8) is the same as applying Block Coordinate Descent (BCD) of Gauss–Seidel type to

our original objective function in (2.5), with the proximal update over the states. The pseudocode of our method can be found in Algorithm 2.1.

Before we get into the details of optimization, let us take another look at our BCD-prox approach and compare it with the iPDA. In BCD-prox, we use the data $\mathbf{Y}$ to initialize the algorithm; subsequently, the algorithm may take many proximal steps to reach a desired optimum. If the data $\mathbf{Y}$ is heavily contaminated with noise, it may be wise to move far away from $\mathbf{Y}$ as we iterate. In contrast, iPDA's regularization term is $\lambda\|\mathbf{X}-\mathbf{Y}\|^2$. Roughly speaking, iPDA searches for $\mathbf{X}$ in a neighborhood of $\mathbf{Y}$; the diameter of this neighborhood is inversely related to λ. When the magnitude of the noise $\mathbf{Z}$ is small, searching for $\mathbf{X}$ in a small neighborhood of $\mathbf{Y}$ is reasonable. For real data problems in which the magnitude of $\mathbf{Z}$ is unknown, however, choosing λ *a priori* becomes difficult.

2.3.1 Optimization over the ODE parameters

At iteration n, we fix $\mathbf{X}$ to equal the learned states from the previous iteration: $\mathbf{X} = \mathbf{X}^{*(n-1)}$, and then minimize the objective function (2.5) over $\boldsymbol{\theta}$ only:

$$\boldsymbol{\theta}^{*(n)} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, E(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \textstyle\sum_{i=1}^{T-1} \left\| \mathbf{x}_{(t_{i+1})}^{*(n-1)} - \mathbf{x}_{(t_i)}^{*(n-1)} - \mathbf{f}(\mathbf{x}_{(t_i)}^{*(n-1)}; \boldsymbol{\theta})\Delta_i \right\|^2 \quad (2.9)$$

This is a d-dimensional regression problem where the ith input and output are $\mathbf{x}_{(t_i)}^{*(n-1)}$ and $(\mathbf{x}_{(t_{i+1})}^{*(n-1)} - \mathbf{x}_{(t_i)}^{*(n-1)})/\Delta_i$, respectively. We optimize this objective using the LBFGS algorithm, implemented in Python via the scipy.optimize.minimize function. Throughout this work, when using LBFGS, we use automatic differentiation to supply the optimizer with gradients of the objective function.

2.3.2 Optimization over the states X

At iteration n, we fix $\boldsymbol{\theta}$ to equal the parameters learned in the previous step: $\boldsymbol{\theta} = \boldsymbol{\theta}^{*(n)}$. By Theorem 2.1, directly minimizing $E(\mathbf{X}, \boldsymbol{\theta}^{*(n)})$ will yield a global minimizer such that $E = 0$, terminating the optimization procedure. To avoid this fate, we use proximal updates in learning the states:

$$\mathbf{X}^{*(n)} = \underset{\mathbf{X}}{\operatorname{argmin}} \left\{ E(\mathbf{X}, \boldsymbol{\theta}^{*(n)}) + \lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|^2 \right\}, \quad (2.10)$$

where $E(\mathbf{X}, \boldsymbol{\theta})$ has been defined in (2.5). For $\lambda > 0$, the penalty term encourages $\mathbf{X}^{*(n)}$ to remain close to $\mathbf{X}^{*(n-1)}$. As we increase λ, we tighten this closeness. To optimize this objective function, we again use the LBFGS algorithm, as in the first step.

Here, we motivate (2.10) using the notion of proximal operators [37]. Let us fix $\boldsymbol{\theta} = \boldsymbol{\theta}^{*(n)}$. When the Δ_i are all identically zero, the objective function E reduces to a quadratic form:

$$\sum_{i=1}^{T-1} \|\mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)}\|^2.$$

If we now fix $\mathbf{x}_{(t_1)}$, the Hessian with respect to the remaining variables $\mathbf{X}_{2:} = \{\mathbf{x}_{(t_i)}\}_{i=2}^{T}$ has strictly positive eigenvalues. The eigenvalues of the Hessian of E are continuous functions of the parameters Δ_i; hence there exists $\epsilon > 0$ such that if $0 \leq \Delta_i < \epsilon$, the Hessian remains positive definite. The upshot is that, under these conditions, $\widetilde{E}(\mathbf{X}_{2:}) = E(\mathbf{x}_{(t_1)}, \mathbf{X}_{2:}, \boldsymbol{\theta}^{*(n)})$ is a strictly convex function of $\mathbf{X}_{2:}$. This is why it has a unique global minimizer $\hat{\mathbf{X}}^{(n)}$, as mentioned above.

Additionally, because $\widetilde{E}$ is convex, we can form the proximal operator

$$\operatorname{prox}_{\mu \widetilde{E}}(\mathbf{X}^{*(n-1)}) = \underset{\mathbf{X}_{2:}}{\operatorname{argmin}} \left\{ \widetilde{E}(\mathbf{X}_{2:}) + (2\mu)^{-1} \|\mathbf{X}_{2:} - \mathbf{X}_{2:}^{*(n-1)}\|^2 \right\} \tag{2.11}$$

We see that when $\mu \to +\infty$, the proximal step returns $\hat{\mathbf{X}}^{(n)}$. For $\mu > 0$ sufficiently small, as shown in [37], this proximal step is approximately equal to a gradient descent step:

$$\operatorname{prox}_{\mu \widetilde{E}}(\mathbf{X}^{*(n-1)}) = \mathbf{X}_{2:}^{*(n-1)} - \mu \nabla \widetilde{E}(\mathbf{X}_{2:}^{*(n-1)}) + o(\mu) \tag{2.12}$$

One can see that our λ in (2.10) plays the role of $(2\mu)^{-1}$ in (2.11). Hence large values of λ correspond to small gradient descent step sizes, and vice versa.

Though E in (2.10) is non-convex, we can view it as a set-valued proximal operator [26]; still, the analogy with convex proximal operators gives valuable intuition. When $\lambda = 0$, we see that (2.10) ignores $\mathbf{X}^{*(n-1)}$ and directly outputs $\hat{\mathbf{X}}^{(n)}$ with $\mathbf{x}_{(t_1)}$ arbitrary. As λ increases, (2.10) approximates a small step that starts from $\mathbf{X}^{*(n-1)}$ and proceeds in the direction of the minimizer $\hat{\mathbf{X}}^{(n)}$.

Algorithm 2.1 Pseudo-code of our proposed method for ODE parameter estimation

Input: A set of T noisy observations $\mathbf{Y} = [\mathbf{y}_{(t_1)}, \ldots, \mathbf{y}_{(t_T)}] \in \mathbb{R}^{d\times T}$, the time differences $\{\Delta_i = t_{i+1} - t_i\}_{i=1}^{T-1}$, the shape of function $\mathbf{f}(\cdot)$ in (2.1), the value of the hyperparameter λ, and the initial guess of the parameters $\boldsymbol{\theta}^{*(0)}$.

1: $\mathbf{X}^{*(0)} = \mathbf{Y}$
2: $n = 0$
3: **repeat**
4: $\quad n = n + 1$
5: $\quad$ • Compute $\boldsymbol{\theta}^{*(n)}$ given $\mathbf{X}^{*(n-1)}$ via (2.8a) [Euler] or (2.15a) [multistep].
6: $\quad$ • Compute $\mathbf{X}^{*(n)}$ given $\boldsymbol{\theta}^{*(n)}$ via (2.8b) [Euler] or (2.15b) [multistep].
7: **until** convergence
8: Compute the predicted states $\hat{\mathbf{X}}$ by repeatedly applying (2.6) [Euler] or (2.13) [multistep], where $\boldsymbol{\theta} = \boldsymbol{\theta}^{*(n)}$ and $\mathbf{x}_{(t_1)} = \mathbf{x}^{*(n)}_{(t_1)}$.
9: **return** $\boldsymbol{\theta}^{*(n)}$ and $\hat{\mathbf{X}}$ as the estimated parameters and predicted states.

2.3.3 Higher-Order discretization

As we mentioned before, our approach is not limited to the Euler discretization. Here, we show that it is straightforward to use higher-order discretization methods.

The idea of multistep (m-step) methods is to use the previous m states to predict the next state. Let us consider the general formulation of the explicit linear m-step method to discretize the ODE (2.1):

$$\mathbf{x}_{(t_{i+1})} = \sum_{j=0}^{m-1} a_j \mathbf{x}_{(t_{i-j})} + \Delta_i \sum_{j=0}^{m-1} b_j \mathbf{f}(\mathbf{x}_{(i-j)}; \boldsymbol{\theta}), \tag{2.13}$$

where Δ_i is the time step. There are several strategies to determine the coefficients $\{a_j\}_{j=0}^{m-1}$ and $\{b_j\}_{j=0}^{m-1}$. Each strategy leads to a specific family of multistep methods. For example, the Adams-Bashforth method approximates $\mathbf{f}(\cdot)$ with a polynomial of order m to find the coefficients and predict the next state. Further information regarding different strategies can be found in [24, 35]. Note that to use m-step methods to predict the state at time

i, we need its previous m states. To predict the states $\{\mathbf{x}_i\}_{i=2}^{m}$ (the first few states), the maximum order we can use is $i-1$, because there are only $i-1$ states before the state $\mathbf{x}_i$. In general, to predict $\mathbf{x}_i$ we use a multistep method of order $\min(i-1, m)$.

When using a general m-step discretization method, we define our objective function as follows:

$$E_{\text{m-step}}(\mathbf{X}, \boldsymbol{\theta}) = \sum_{i=1}^{T-1} \left\| \mathbf{x}_{(t_{i+1})} - \sum_{j=0}^{k-1} a_j \mathbf{x}_{(t_{i-j})} - \Delta_i \sum_{j=0}^{k-1} b_j \mathbf{f}(\mathbf{x}_{(i-j)}; \boldsymbol{\theta}) \right\|^2, \tag{2.14}$$

where $k = \min(i-1, m)$ is the order of the discretization method to predict the state $\mathbf{x}_i$. Now, we repeat the following steps until convergence:

$$\boldsymbol{\theta}^{*(n)} = \operatorname*{argmin}_{\boldsymbol{\theta}} E_{\text{m-step}}(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}) \tag{2.15a}$$

$$\mathbf{X}^{*(n)} = \operatorname*{argmin}_{\mathbf{X}} \left\{ E_{\text{m-step}}(\mathbf{X}, \boldsymbol{\theta}^{(*n)}) + \lambda \|\mathbf{X} - \mathbf{X}^{*(n-1)}\|^2 \right\} \tag{2.15b}$$

For both steps, we use LBFGS, initialized with $\boldsymbol{\theta}^{*(n-1)}$ and $\mathbf{X}^{*(n)}$, respectively.

2.4 Convergence

There are two notions of convergence that we will discuss briefly here: practical and theoretical.

Practice. We implement the argmin steps in Alg.2.1 using the LBFGS algorithm, implemented in Python via the scipy.optimize.minimize function. Throughout this work, when using LBFGS, we use automatic differentiation to supply the optimizer with gradients of the objective function. We stop Alg.2.1 when the error E changes less than 10^{-8} from one iteration to the next.

To see when this happens, we take another look at the optimization over the states $\mathbf{X}$ in (2.10) at iteration n. This objective function has two parts. The optimal solution of the first part (E) is the predicted states $\hat{\mathbf{X}}^{(n)}$. The optimal solution of the proximal part is $\mathbf{X}^{*(n-1)}$. When we optimize this objective function to find $\mathbf{X}^{*(n)}$, there are three cases: 1) $\mathbf{X}^{*(n)} = \hat{\mathbf{X}}^{(n)}$, 2) $\mathbf{X}^{*(n)} = \mathbf{X}^{*(n-1)}$, and 3) $\mathbf{X}^{*(n)}$ is neither $\hat{\mathbf{X}}^{(n)}$ nor $\mathbf{X}^{*(n-1)}$. Our algorithm stops when we are in case 1 or 2 since further optimization over $\boldsymbol{\theta}$ and $\mathbf{X}$ does

not change anything. In case 3, the algorithm continues, leading to further optimization steps to decrease error.

Indeed, let us note that steps 5 and 6 in Alg. 2.1 together imply

$$E(\mathbf{X}^{*(n)}, \boldsymbol{\theta}^{*(n)}) \leq E(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}^{*(n-1)}). \tag{2.16}$$

The function E, bounded below by 0, is non-increasing along the trajectory $\{(\mathbf{X}^{*(n)}, \boldsymbol{\theta}^{*(n)})\}_{n\geq 1}$. Hence $\{E(\mathbf{X}^{*(n)}, \boldsymbol{\theta}^{*(n)})\}_{n\geq 1}$ must converge to some $E^* \geq 0$.

Theory. Here we will offer a convergence theory for the Euler version of BCD-prox. We believe this theory can also be established for the general m-step version of BCD-prox; however, the calculations will be lengthier. In this subsection, we let $\mathbf{x}_i = \mathbf{x}_{(t_i)}$. For T even, set

$$\mathbf{x}^+ = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{T/2}\}$$
$$\mathbf{x}^- = \{\mathbf{x}_{T/2+1}, \ldots, \mathbf{x}_T\}$$

For T odd, replace $T/2$ by $(T-1)/2$ in the above definitions. In words, $\mathbf{x}^+$ is the first half of the state series while $\mathbf{x}^-$ is the second half of the state series. Note that $\mathbf{X} = (\mathbf{x}^+, \mathbf{x}^-)$.

Suppose for the moment that all Δ_i are zero. Then (2.5) reduces to

$$E(\mathbf{X}, \boldsymbol{\theta}; \Delta_i = 0) = \sum_{i=1}^{T-1} \|\mathbf{x}_{i+1} - \mathbf{x}_i\|^2.$$

This is a quadratic form written as a sum of squares of linear terms; hence it is positive semidefinite. In general, we have for $2 \leq j \leq T-1$,

$$\nabla_{x_j} E(\mathbf{X}, \boldsymbol{\theta}; \Delta_i = 0) = -2\mathbf{x}_{j-1} + 4\mathbf{x}_j - 2\mathbf{x}_{j+1}.$$

For the edge cases, we have

$$\nabla_{\mathbf{x}_1} E(\mathbf{X}, \boldsymbol{\theta}; \Delta_i = 0) = -2\mathbf{x}_2 + 2\mathbf{x}_1,$$
$$\nabla_{\mathbf{x}_T} E(\mathbf{X}, \boldsymbol{\theta}; \Delta_i = 0) = 2\mathbf{x}_T - 2\mathbf{x}_{T-1}.$$

Now suppose we hold $\mathbf{x}^-$ fixed and only take the Hessian with respect to $\mathbf{x}^+$. We will obtain

$$A = \begin{bmatrix} 2I & -2I & & & \\ -2I & 4I & -2I & & \\ & -2I & 4I & \ddots & \\ & & \ddots & \ddots & -2I \\ & & & -2I & 4I \end{bmatrix}$$

Here each I is a $d \times d$ identity block; the d is such that $x_i \in \mathbb{R}^d$. The overall size of the matrix is therefore $d(T/2) \times d(T/2)$. The positive semidefiniteness established above implies that all eigenvalues of A are nonnegative. A simple induction argument then shows that $\det A > 0$, implying that there is no zero eigenvalue. Hence A has strictly positive spectrum, and so $E(\mathbf{X}, \boldsymbol{\theta}; \Delta_i = 0)$ restricted to x^+ (with x^- held fixed) is strictly convex. In a completely analogous way, we can show that $E(\mathbf{X}, \boldsymbol{\theta}; \Delta_i = 0)$ restricted to x^- (with x^+ held fixed) is strictly convex. Now both of these properties hold at $\Delta_i = 0$. Because the eigenvalues of both restrictions are continuous functions of Δ_i, there exists $\delta > 0$ such that for $\Delta_i \in (0, \delta)$, the eigenvalues in question remain strictly positive. Henceforth assume that all Δ_i satisfy this criterion. Then $E(\mathbf{X}, \boldsymbol{\theta})$ is strictly convex in $\mathbf{x}^+$ (with $\mathbf{x}^-$ and $\boldsymbol{\theta}$ held fixed) and strictly convex in $\mathbf{x}^-$ (with $\mathbf{x}^+$ and $\boldsymbol{\theta}$ held fixed).

Next, assume that $\mathbf{f}$ is at most linear in $\boldsymbol{\theta}$, so that

$$\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{f}_0(\mathbf{x}) + \mathbf{f}_1(\mathbf{x})\boldsymbol{\theta}.$$

Here $\mathbf{f}_0 : \mathbb{R}^d \to \mathbb{R}^d$. Because $\boldsymbol{\theta} \in \mathbb{R}^p$, we have $\mathbf{f}_1 : \mathbb{R}^d \to \mathbb{R}^{d \times p}$. That is, $\mathbf{f}_1(\mathbf{x})$ is a $d \times p$ matrix, which when multiplied by $\boldsymbol{\theta}$, gives a vector in $\mathbb{R}^d$. We assume that $\mathbf{f}_1(\mathbf{x})$ has full column rank. Then we have

$$E(\mathbf{X}, \boldsymbol{\theta}) = \sum_{i=1}^{T-1} \|\mathbf{x}_{i+1} - \mathbf{x}_i - \mathbf{f}_0(\mathbf{x}_i)\Delta_i + \mathbf{f}_1(\mathbf{x}_i)\boldsymbol{\theta}\Delta_i\|^2.$$

Now we compute the $p \times p$ Hessian

$$\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} E = 2 \sum_{i=1}^{T-1} (\mathbf{f}_1(\mathbf{x}_i))^T \mathbf{f}_1(\mathbf{x}_i) \Delta_i^2.$$

Since $\mathbf{f}_1$ has full column rank, the Hessian is positive definite; $E(\mathbf{X}, \boldsymbol{\theta})$ is strongly convex in $\boldsymbol{\theta}$ with $\mathbf{X}$ held fixed.

Now initialize $\mathbf{X}^0 = \mathbf{Y}$ and proceed sequentially with the following steps for $n \geq 1$:

$$\boldsymbol{\theta}^n = \operatorname*{argmin}_{\boldsymbol{\theta}} E(\mathbf{X}^{n-1}, \boldsymbol{\theta}) \tag{2.17a}$$

$$(\mathbf{x}^-)^n = \operatorname*{argmin}_{\mathbf{x}^-} E((\mathbf{x}^+)^{n-1}, \mathbf{x}^-, \boldsymbol{\theta}^n) + \lambda \left\|(\mathbf{x})^- - (\mathbf{x}^-)^{n-1}\right\|^2 \tag{2.17b}$$

$$(\mathbf{x}^+)^n = \operatorname*{argmin}_{\mathbf{x}^+} E(\mathbf{x}^+, (\mathbf{x}^-)^n, \boldsymbol{\theta}^n) + \lambda \left\|(\mathbf{x})^+ - (\mathbf{x}^+)^{n-1}\right\|^2 \tag{2.17c}$$

$$\mathbf{X}^n = ((\mathbf{x}^+)^n, (\mathbf{x}^-)^n) \tag{2.17d}$$

Then we have the following first convergence result.

Theorem 2.2. *Suppose all $\Delta_i \in (0, \delta)$ for the δ established above. Suppose $\mathbf{f}$ is linear in $\boldsymbol{\theta}$ with the full-rank condition described above. Then there exists an interval of λ values for which the algorithm* (2.17) *converges to a Nash equilibrium $(\overline{\mathbf{X}}, \overline{\boldsymbol{\theta}})$ of the objective function E defined in* (2.5).

Proof. The result follows directly from Theorem 2.3 from [56]; we have verified all hypotheses. □

Let us further assume that $\mathbf{f}$ satisfies the Kurdyka-Lojasiewicz (KL) property described in Section 2.2 of [56]. In particular, if each component of $\mathbf{f}$ is real analytic, the KL property will be satisfied. Together with linearity of $\mathbf{f}$ in $\boldsymbol{\theta}$, this includes numerous vector fields of interest, including all ODE in our experimental results. (For FitzHugh–Nagumo, a change of variables renders the system linear in the parameters.) Then we have a second convergence result.

Theorem 2.3. *Suppose in addition to the hypotheses of Theorem 2.2, $\mathbf{f}$ is smooth and satisfies the KL property. Then assuming the algorithm defined by* (2.17) *begins sufficiently close to a global minimizer, it will converge to a global minimizer of E defined in* (2.5).

Proof. The result follows directly from Corollary 2.7 and Theorem 2.8 of [56]; we have verified all hypotheses. □

2.5 Experiments

In this section, we first introduce four benchmark datasets. Then we show how our method works under different kinds of noise and initializations on these datasets. We finally show that our method performs well compared to other state-of-the-art methods. In the following, we create clean states using a Runge-Kutta method of order 5. In all of our experiments, unless otherwise stated, we use the three-step Adams-Bashforth method to discretize the ODE.

Lotka–Volterra model. This model is used to study the interaction between predator (variable x_0) and prey (variable x_1) in biology [30]. The model contains two nonlinear equations as follows:

$$\frac{dx_0}{dt} = \theta_0 x_0 - \theta_1 x_0 x_1 \qquad \frac{dx_1}{dt} = \theta_2 x_0 x_1 - \theta_3 x_1. \tag{2.18}$$

The state is two-dimensional and there are four unknown parameters. We use the same settings as in [15]. We set the parameters to $\theta_0 = 2$, $\theta_1 = 1$, $\theta_2 = 4$ and $\theta_3 = 1$. With initial condition $\mathbf{x}_{(1)} = [5, 3]$, we generate clean states in the time range of $[0, 2]$ with a spacing of $\Delta t = 0.1$.

FitzHugh–Nagumo model. This model describes spike generation in squid giant axons [17, 33]. It also has two nonlinear equations:

$$\frac{dx_0}{dt} = \theta_2(x_0 - \frac{(x_0)^3}{3} + x_1) \qquad \frac{dx_1}{dt} = -\frac{1}{\theta_2}(x_0 - \theta_0 + \theta_1 x_1), \tag{2.19}$$

where x_0 is the voltage across an axon and x_1 is the outward current. In this model, the states are two-dimensional and there are three unknown parameters. We use the same settings as in [43]. We set the parameters as $\theta_0 = 0.5, \theta_1 = 0.2$, and $\theta_3 = 3$. With initial condition $\mathbf{x}_{(1)} = [-1, 1]$, we generate clean states in the time range of $[0, 20]$ with a spacing of $\Delta t = 0.05$.

Rössler attractor. This three-dimensional nonlinear system has a chaotic attractor [44]:

$$\frac{dx_0}{dt} = -x_1 - x_2 \qquad \frac{dx_1}{dt} = x_0 + \theta_0 x_1 \qquad \frac{dx_2}{dt} = \theta_1 + x_2(x_0 - \theta_2). \tag{2.20}$$

The states are three-dimensional and there are three unknown parameters. We use the same settings as in [43]. We set the parameters as $\theta_0 = 0.2, \theta_1 = 0.2$, and $\theta_3 = 3$. With the initial condition $\mathbf{x}_{(1)} = [1.13, -1.74, 0.02]$, we generate clean states in the time range of $[0, 20]$, with a spacing of $\Delta t = 0.05$.

Lorenz-96 model. The goal of this model is to study weather predictability [29]. The kth differential equation has the following form:

$$\frac{dx_k}{dt} = (x_{k+1} - x_{k-2})(x_{k-1}) - x_k + \theta_0, \qquad k = 0, \ldots, d-1 \tag{2.21}$$

The model has one parameter θ_0 and d states, where d can be set by the user. This gives us the opportunity to test our method on larger ODEs. Note that to make (2.21) meaningful, we have $x_{-1} = x_{d-1}$, $x_{-2} = x_{d-2}$, and $x_d = x_0$. As suggested in [29], we set $d = 40$ and $\theta_0 = 8$. The clean states are generated in the time range $[0, 4]$ with a spacing of $\Delta_i = 0.01$. The initial state is generated randomly from a Gaussian distribution with mean 0 and variance 1.

Evaluation metrics. Let $\boldsymbol{\theta}$ and $\mathbf{X}$ denote the true parameters and the clean states, respectively. Let $\boldsymbol{\theta}^*$ and $\hat{\mathbf{X}}$ denote the estimated parameters and the predicted states. We report the Frobenius norm of $\mathbf{X} - \hat{\mathbf{X}}$ as the prediction error. We also consider $|\theta_l - \theta_l^*|$ as the lth parameter error. Recall that predicted states are achieved by considering $\boldsymbol{\theta}^*$ as the parameter and $\mathbf{x}^*_{(t_0)}$ as the initial state, and then repeatedly applying (2.6) or its multistep analogue (2.13).

Advantages of our approach. Our method is robust with respect to its only hyper-parameter λ. We will show in our experimental results later that for a broad range of λ values, our method works well. We fix it to $\lambda = 1$ in our later experiments. As explained before, previous methods have a large number of hyperparameters, which are difficult to set.

Our method can be trained quickly. On a standard laptop, it takes around 20 seconds for our method to learn the parameters and states jointly on ODE problems with

400 states. The spline-based methods take a few minutes and Bayesian methods take a few hours to converge on the same problem.

Because our method, unlike Bayesian methods, does not make assumptions about the type of the noise or distribution of the states, it performs well under different noise and state distributions. In particular, as the magnitude of noise in the observations increases, our method clearly outperforms the extended Kalman filter.

As our experiments confirm, both spline-based and Bayesian approaches are very sensitive to the initialization of the ODE parameters. If we initialize them far away from the true values, they do not converge. Our method is much more robust. This robustness stems from simultaneously learning states and parameters. Even if the estimated parameters are far from the true parameters at some iteration, they can improve later, as the estimated states converge to the clean states.

Optimization of our objective function leads to a better estimation. We first focus only on our objective function in (2.5). At each iteration n of our optimization, we compute the predicted states $\hat{\mathbf{X}}^{(n)}$ and report the prediction error. Fig. 2.1 shows the results.

In Fig. 2.1, we consider two kinds of discretization: 1) one-step Euler method, and 2) three-step Adams-Bashforth method. Note that as we increase the order, we expect to see more accurate results.

We added Gaussian noise with mean 0 and variance σ^2 to each of the clean states, where $\sigma^2 = 0.5$ in the first column and $\sigma^2 = 1$ in the second column. Fig. 2.1 shows that at the first iteration the error is significant in all models. The error is $\sim 1\,000$ for FitzHugh–Nagumo and Rössler, and $\sim 1.5 \times 10^5$ for Lorenz-96 model.

After several iterations of our algorithm, the error decreases significantly, no matter what kind of discretization we use. But, as expected, three-step Adams-Bashforth performs better than Euler in general: it converges faster and achieves a smaller error at the end. This is more clear for the Lorenz-96 model, where the error becomes almost zero when we use the three-step Adams-Bashforth, while the error becomes around 10^4 for Euler.

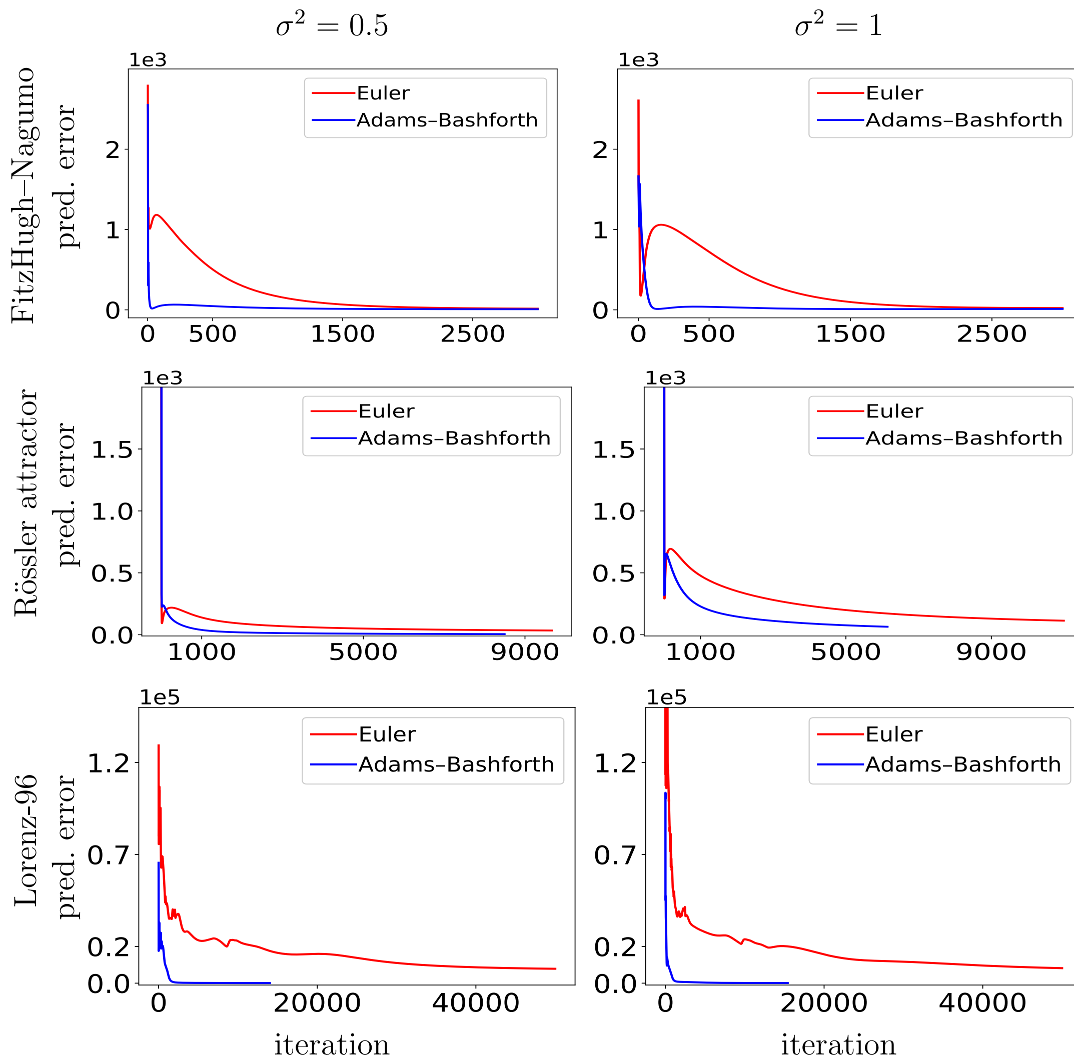

Figure 2.1: Prediction error at different iterations of our algorithm. Noisy observations are achieved by adding Gaussian noise with variance σ^2 to the clean observations. We consider the FitzHugh–Nagumo (first row), Rössler (second row), and Lorenz-96 (third row) models. Our learning strategy decreases the error in all cases.

The last point about Fig. 2.1 is that, as expected, sometimes the prediction error increases; the error does not decrease monotonically. This mainly happens at the first few iterations. The main reason for this behavior is that our objective function in (2.5) is different from the prediction error. We cannot directly optimize the prediction error

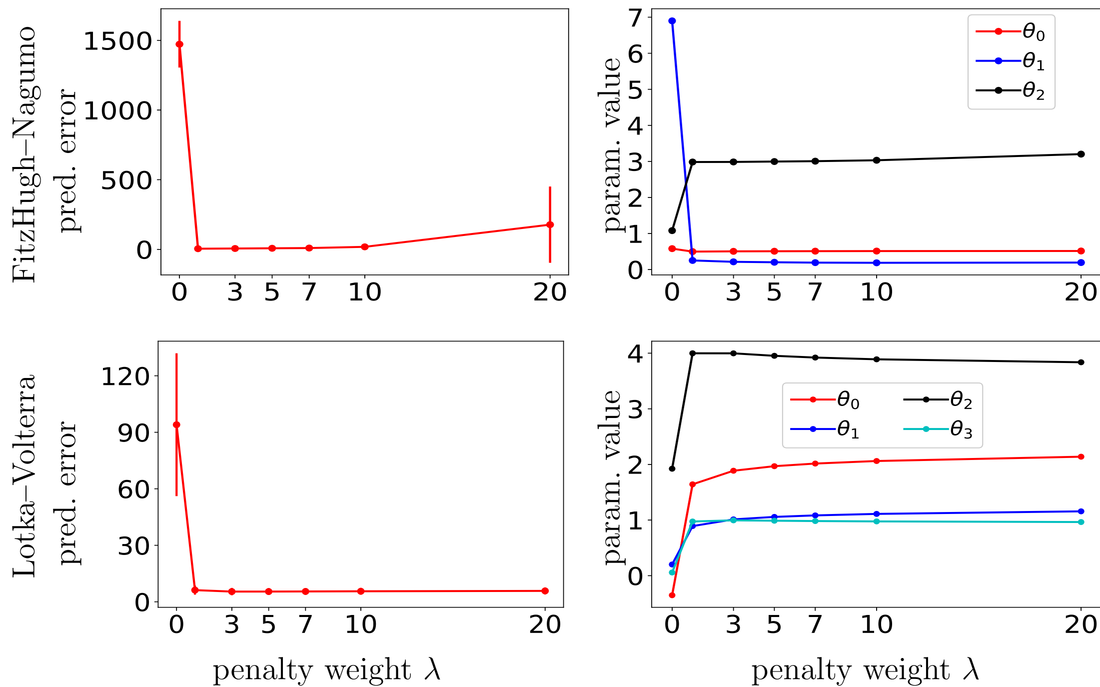

Figure 2.2: Robustness to the hyperparameter λ in FitzHugh–Nagumo (first row) and Lotka–Volterra (second row) models. The true parameters are $\theta_0 = .5, \theta_1 = .3$, and $\theta_2 = 3$ in the FitzHugh–Nagumo and $\theta_0 = 2, \theta_1 = 1, \theta_2 = 4$, and $\theta_3 = 1$ in the Lotka–Volterra. For each λ, we report the mean error and parameter value in 10 experiments.

because we do not have access to the clean states. Still, the fact that our algorithm eventually brings the prediction error close to zero suggests that minimizing the objective in (2.5) has the same effect as minimizing the prediction error.

Robustness to the hyperparameter λ. The only hyperparameter in our algorithm is λ. In Fig. 2.2, we set λ in turn to a set of values from 0 to 20, run our algorithm, and report the results after convergence. In both models, we generate observations by adding Gaussian noise with variance 0.5 to the clean states. Because of randomness included in creating noisy observations, we create 10 sets of observations, run our algorithm once for each of them, and report the mean in Fig. 2.2. We also show the standard deviation in prediction errors, but not in parameter values (to avoid clutter).

In Fig. 2.2 we report the prediction error and the estimated parameters for each value of λ. The true values for the FitzHugh–Nagumo are $\theta_0 = .5, \theta_1 = .3$, and $\theta_2 = 3$. For the Lotka–Volterra model, the true values are $\theta_0 = 2, \theta_1 = 1, \theta_2 = 4$, and $\theta_3 = 1$.

We see in Fig. 2.2 that for $\lambda > 0$, our method correctly finds the parameters and brings the error close to zero. Also, in the range of $\lambda = 1$ to 20, the errors and the estimated parameters remain almost the same. We actually increased λ to 1 000 and found that it works like the previous values of $\lambda > 0$. The only disadvantage of increasing λ to a large value is that training time increases—as explained above, increasing λ is analogous to decreasing the step size in a gradient descent method. Large λ implies that states can change very little from one iteration to another, forcing the algorithm to run longer for convergence. As explained in detail above, when $\lambda = 0$, the algorithm stops after a single iteration, with the predicted states far from the clean states.

Different types and amounts of noise in the observations. Our method does not assume anything about the type of noise. In reality, the noise could be from any distribution. In Fig. 2.3, we investigate the effect of the type of noise on the outcome of our algorithm. The red (blue) curves correspond to the case when we add Gaussian (Laplacian) noise to the observations. We set the mean to 0, change the variance of the noise, and report the prediction and parameter errors. Note that for each noise variance, we repeat the experiment 10 times and report the mean and standard deviation of the error.

In general, increasing the noise variance increases the error. We can see this in almost all plots. In both models, the error does not change much by changing the variance from 0 to 0.5. We can also see that the method performs almost as well for observations corrupted by Laplacian noise as in the Gaussian noise case. Note that the Laplacian noise has a heavier-than-Gaussian tail.

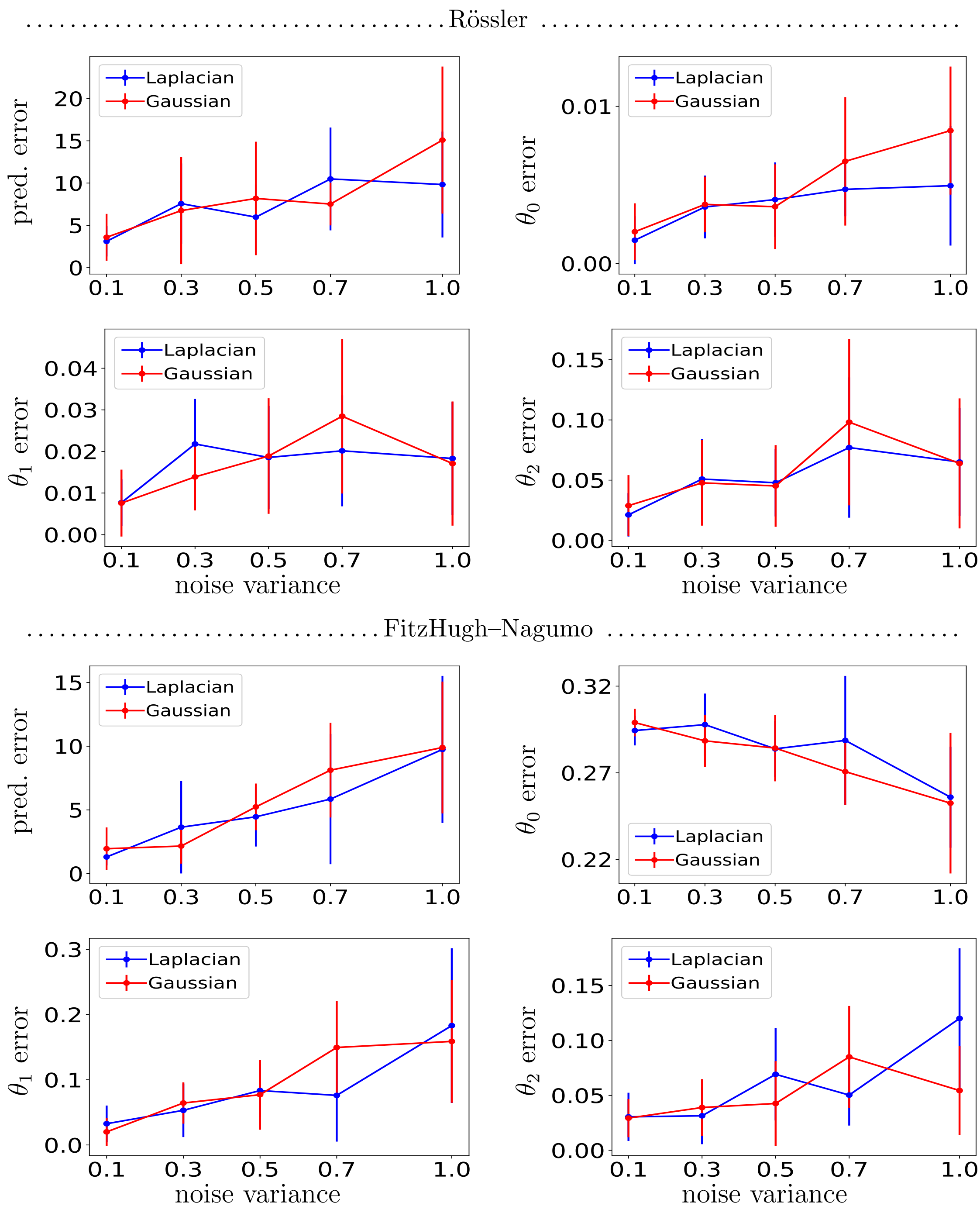

Figure 2.3: Robustness to different types and amounts of noise in the observations. We report the prediction and parameter errors on the Rössler and FitzHugh–Nagumo models.

Comparison with other methods (robustness to the initialization). As the first experiment, we compare our method with three other methods, each of them from a different category. Among the spline-based methods, we use the online MATLAB code corresponding to [43], denoted by "splines" in our experiments. Among the Bayesian approaches, we use the online R code corresponding to [15], denoted "Bayes" in our experiments. We also implement a method that uses the iterative least square approach, denoted "lsq" in our experiments. This method considers the parameters and the initial state as the unknown variables. To implement lsq, we use the Python LMFIT package [34]. We use the FitzHugh–Nagumo and Rössler models, creating noisy observations by adding Gaussian noise with mean 0 and variance 0.5 to the clean states.

All methods including ours need an initial guess for the unknown parameters. We add Gaussian noise with mean 0 and variance σ_θ^2 to the true parameter and use the result to initialize the methods. Fig.2.4 and Fig. 2.5 show the results on FitzHugh–Nagumo and Rössler attractor, where we change the variance from $\sigma_\theta^2 = 1$ to 20. Since there is randomness in both initialization and observation, we repeat the experiment 10 times. Note that the comparisons are fair, with the same observations and initializations used across all methods.

In these figures, each of the bars corresponds to the prediction or parameter error for one of the methods in one of the experiments. Hence there are 10 error bars for each of the methods in each plot. We set $\lambda = 1$ in our method for all the experiments. For the other methods, we chose the best hyperparameters we could determine after careful experimentation.

The first point regarding these experiments is that our method is robust with respect to the initialization, while the other methods are not. The total number of experiments per method is 80 (on the two models). The prediction error of our method exceeds 100 in 4 experiments. The prediction error of splines (the second best method after ours) exceeds 100 in 39 experiments (nearly half the experiments). For the other methods, the error only increases.

We can see in the figures that almost all the methods work well when the initialization is close to the true parameters (small noise). But, in reality, we do not know what

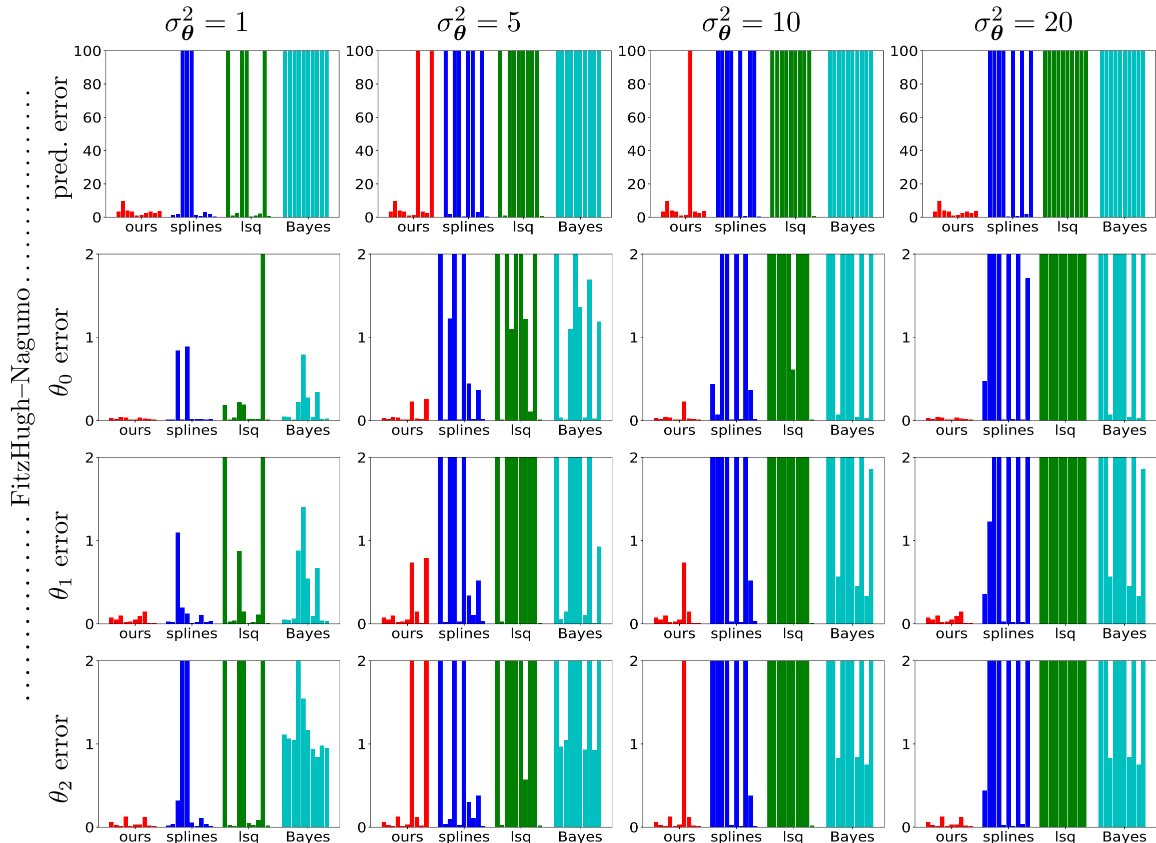

Figure 2.4: Comparison with other methods on FitzHugh–Nagumo model. We create initializations by adding Gaussian noise of variance $\sigma_{\boldsymbol{\theta}}^2$ to the true parameters. We create 10 sets of observations and initializations per each $\sigma_{\boldsymbol{\theta}}^2$ and report the errors. Each error bar corresponds to the error in one of the experiments.

the real parameters are. So it is reasonable to say that the last column of Fig. 2.4 and Fig. 2.5 (initialization with the largest noise) determines which method performs better in real-world applications. As we can see, our method outperforms the other methods in both prediction and parameter error.

In our second experiment, we compare our method with the mean-field method (also a Bayesian method) [19] on the Lotka–Volterra model. The mean-field method is only applicable to the differential equations with a specific form (see Eq. (10) in [19]). While we cannot apply it to FitzHugh–Nagumo and Rössler models, we can apply it to the

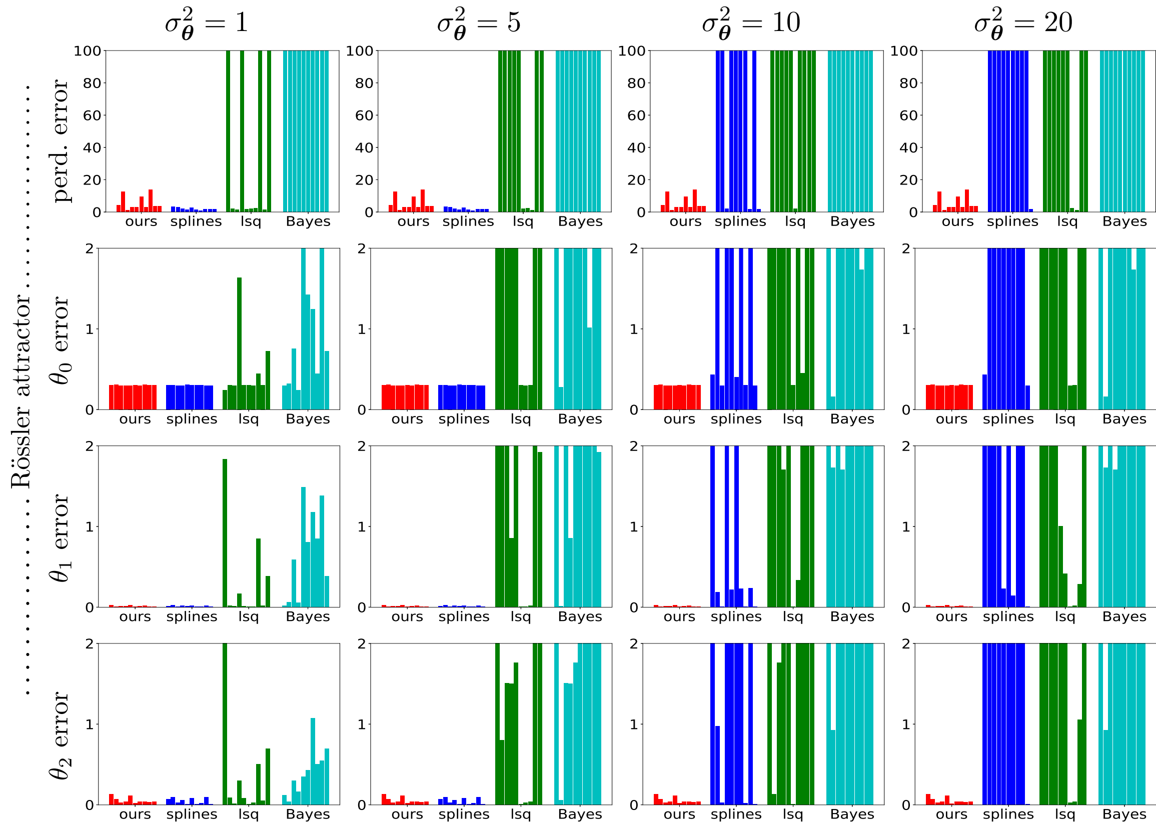

Figure 2.5: Comparison with other methods on Rössler attractor. The settings are the same as the Fig. 2.4

Lotka–Volterra model. In Fig. 2.6, we compare the methods by reporting the prediction and the parameter errors. We create noisy observations by starting with the clean states and adding Gaussian noise with variance σ^2. We show the results for different variances at different columns of Fig. 2.6. Similar to our previous experiments, we generate 10 sets of noisy observations. Each of the bars in Fig. 2.6 corresponds to the error for one of the methods in one of the experiments.

Fig. 2.6 shows that the average error of our method is less than the mean-field method in almost all cases [19]. As we increase the noise in the observations, the error of both methods increases. But, we can also see that our method is more robust than the mean-field method when it comes to observation noise. Consider the case where $\sigma^2 = 1$

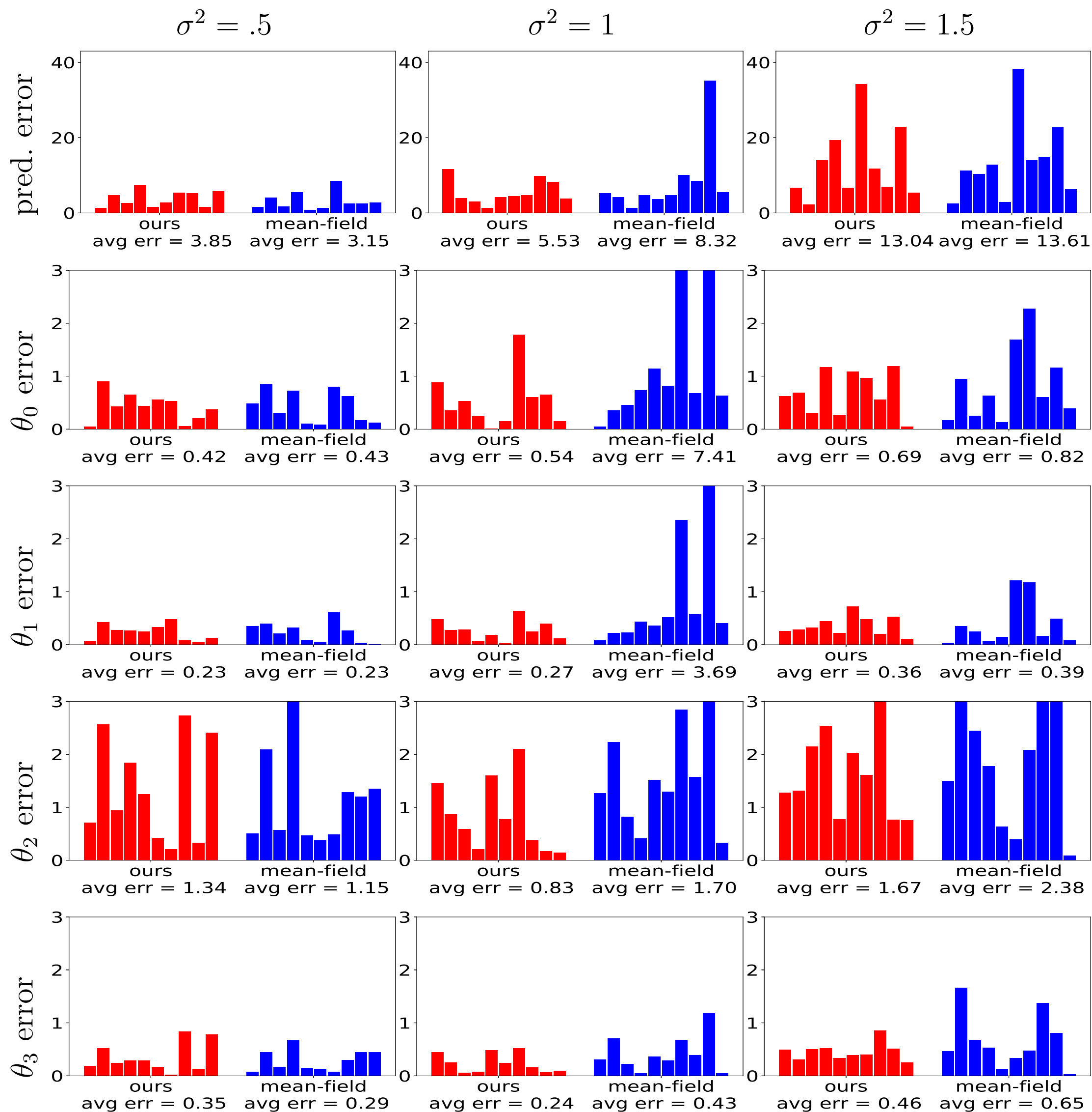

Figure 2.6: Comparison with the mean-field method [19] on Lotka–Volterra model. We add Gaussian noise with variance σ^2 to the clean states to create noisy observations. Each error bar corresponds to the error in one of the experiments.

(second column). In this case, the average parameter error of the mean-field method for θ_0 and θ_1 becomes around 3 and 8, respectively, but the average error of our method for both parameters remains less than 1.

Comparison with the extended Kalman filter (EKF). We follow [51] in applying the Kalman filter to our problem of estimating the parameters and states. We first need to write an equation that recursively finds the state $\mathbf{x}_{(t_{i+1})}$ in terms of $\mathbf{x}_{(t_i)}$. As suggested in [51], this can be achieved by discretizing the ODE in (2.1) using the Euler discretization:

$$\mathbf{x}_{(t_{i+1})} = \mathbf{x}_{(t_i)} + \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})\Delta_i. \tag{2.22}$$

Let us define $\boldsymbol{\theta}_{(t_i)}$ as the parameter estimated at time t_i by the Kalman filter. We define a joint state variable $\boldsymbol{\xi}_{(t_i)}$, which merges the states $\mathbf{x}_{(t_i)}$ and the parameters $\boldsymbol{\theta}_{(t_i)}$ as follows:

$$\boldsymbol{\xi}_{(t_i)} = \begin{pmatrix} \mathbf{x}_{(t_i)} \\ \boldsymbol{\theta}_{(t_i)} \end{pmatrix}, \quad \boldsymbol{\xi}_{(t_i)} \in \mathbb{R}^{d+p}. \tag{2.23}$$

The process model to predict the next state variable can be written as:

$$\boldsymbol{\xi}_{(t_{i+1})} = \begin{pmatrix} \mathbf{x}_{(t_{i+1})} \\ \boldsymbol{\theta}_{(t_{i+1})} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(t_i)} + \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})\Delta_i \\ \boldsymbol{\theta}_{(t_i)} \end{pmatrix}. \tag{2.24}$$

We define the observation model as follows:

$$\mathbf{y}_{(t_i)} = \mathbf{H}\boldsymbol{\xi}_{(t_i)}, \quad \mathbf{H} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \end{pmatrix}_{d \times (d+p)}, \tag{2.25}$$

where $\mathbf{H}$ is a $d \times (d+p)$ matrix, $\mathbf{I}$ is a $d \times d$ identity matrix, and $\mathbf{0}$ is a $d \times p$ matrix where all elements are 0.

In most cases, the function $\mathbf{f}(\cdot)$ is nonlinear, which makes the process model nonlinear. For this reason, we use the extended Kalman filter (EKF), which linearizes the model.

We compare our method with the EKF in Fig. 2.7 on the Lotka–Volterra model. We compare the methods in different settings by changing the amount of noise and the number of samples. To create noisy observations, we add Gaussian noise with the variances $\sigma^2 = 0.1$ and $\sigma^2 = 1.5$ to the clean states. We set the number of samples to $T = 20$ (time range $[0, 2]$) and $T = 10\,000$ (time range $[0, 1\,000]$).

We use an open-source Python code [25] to implement the EKF. We set the state covariance (noise covariance) to a diagonal matrix with elements equal to $1\,000$ (0.1). We

set the process covariance using the function Q_discrete_white_noise() provided in [25], where the variance is set to 1. Note that these parameters must be carefully tuned to obtain reasonable results; changing the state or noise covariance yields significantly worse results.

In Fig. 2.7 we report the average estimation error instead of the prediction error. Estimation error is defined as the difference between the clean states and the estimated states $\mathbf{X}^*$. We report the estimation error because the prediction error of the EKF goes to infinity. To see why this happens, note that to obtain reasonable predictions we need good estimations of the parameters and the initial state. Since the EKF is an online method, it never updates the initial state. Given that the initial state is noisy, no matter how well parameters are estimated, the prediction error becomes very large. Our method updates the initial state, yielding small prediction error.

As we can see in Fig. 2.7, the only setting in which the EKF performs comparably to our method is the case of $T = 10\,000$ and $\sigma^2 = 0.1$. In other words, EKF works fine when we have long time series with low noise. In more realistic settings, our method significantly outperforms the EKF. A key difference between the two methods is that the EKF is an online method while ours is a batch method, iterating over the entire dataset repeatedly. Consequently, our method updates parameters based on information in all the states, leading to more robust updates than is possible with the EKF, which updates parameters based on a single observation.

We can also see in Fig. 2.7 that the error of both methods becomes smaller as we increase the number of samples T. This is expected to happen because increasing T is equivalent to giving more information about the model to the methods. The average estimation error of our method becomes almost 0 for large T.

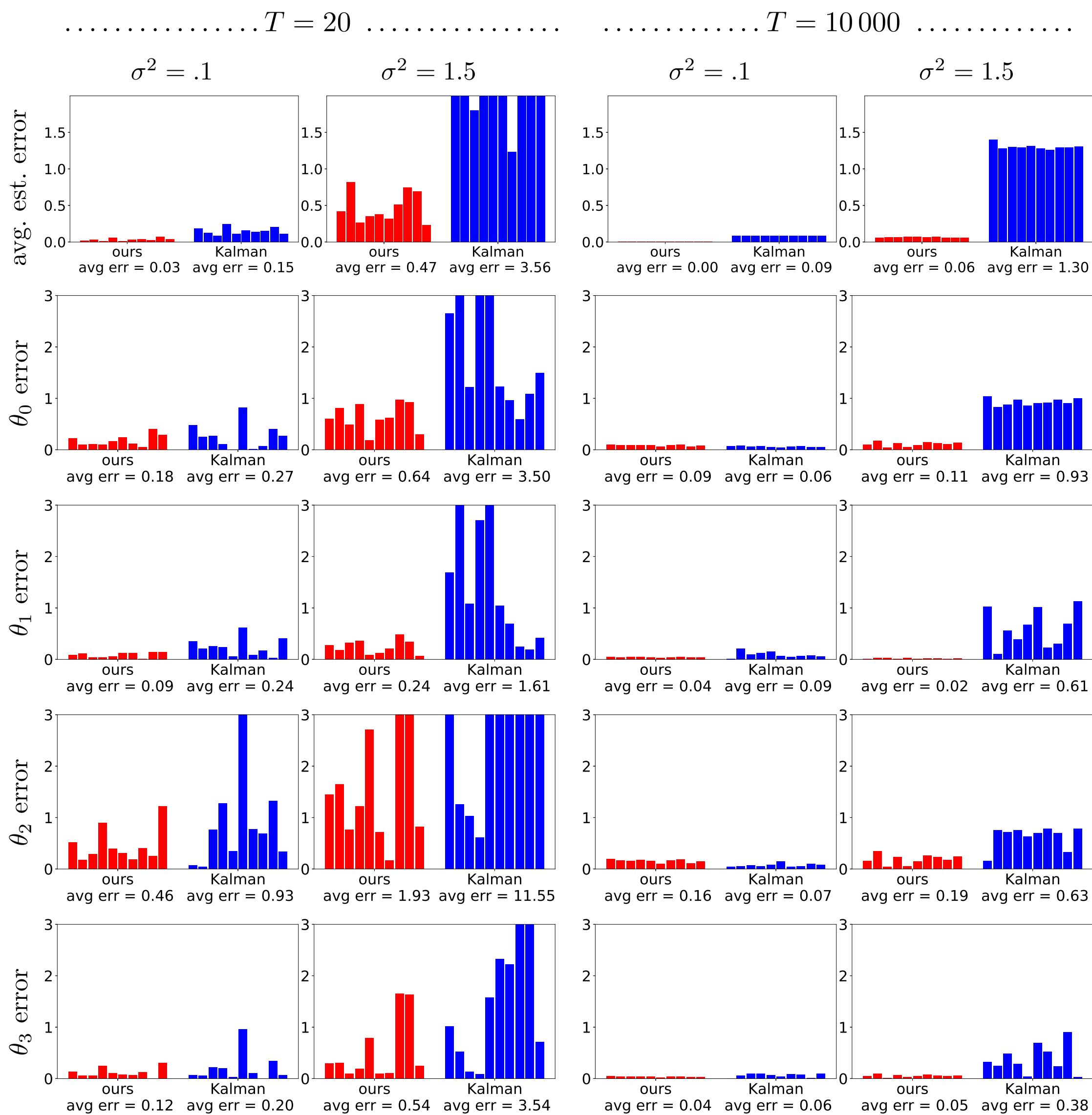

Figure 2.7: Comparison with the extended Kalman filter (EKF) on the Lotka–Volterra model. We add Gaussian noise with variance $\sigma^2 = 0.1$ and $\sigma^2 = 1.5$ to the clean states to create noisy observations. We set the number of observation to $T = 20$ (left panel) and $T = 10\,000$ (right panel). For each value of T, we generate 10 sets of observations and report the average estimation and parameter errors. Each error bar corresponds to the error in one of the experiments. We have put the average error of each method for the 10 experiments below each plot.

2.6 Conclusion

ODE parameter estimation is an important problem, made difficult due to noisy data and lack of analytical ODE solutions. Previous approaches make several assumptions about the states and the noise, leading to issues regarding the number of hyperparameters and robustness to the noise. In this chapter, we have shown that these assumptions are unnecessary. Our BCD-prox algorithm addresses issues of previous approaches to simultaneous parameter estimation and filtering, achieving fast training and robustness to noise, initialization, and hyperparameter tuning. Additional features of BCD-prox include its connection to BCD and proximal methods, its unified objective function, and a convergence theory resulting from blockwise strict convexity.

Chapter 3

Interpretable Equation Discovery with Neural Networks

3.1 Problem Definition

Consider a dynamical system in $\mathbb{R}^d$ with state $\mathbf{x}(t)$ at time t. The time-evolution of the state is given by

$$\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t); \boldsymbol{\theta}). \tag{3.1}$$

where $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ is an unknown vector field, whose components can be written as $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_d(\mathbf{x}))$, and $\boldsymbol{\theta}$ is a vector containing all parameters of $\mathbf{f}$. At T distinct times $\{t_i\}_{i=1}^T$, we have noisy observations $\mathbf{y}(t_i) \in \mathbb{R}^d$:

$$\mathbf{y}(t_i) = \mathbf{x}(t_i) + \mathbf{z}(t_i), \quad i = 1, \ldots, T \tag{3.2}$$

where $\mathbf{z}(t_i) \in \mathbb{R}^d$ is the noise of the observation at time t_i. We represent the set of T d-dimensional states, noises, and observations by $\mathbf{X}, \mathbf{Z}$, and $\mathbf{Y} \in \mathbb{R}^{d \times T}$, respectively. For concision, in what follows, we write the time t_i as a subscript, i.e., $\mathbf{x}_{(t_i)}$ instead of $\mathbf{x}(t_i)$.

In this chapter, we seek to model an unknown vector field $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ *using a neural network.*

3.2 Our Proposed Method

In this section, we first focus on the scalar-valued function $f_j : \mathbb{R}^d \to \mathbb{R}$, which is the jth component of $\mathbf{f}$, and show how we model this using a neural network. We then extend the idea to all of $\mathbf{f}$ dimensions.

3.2.1 Parameterization of the f_j

We start the process by assuming that we have access to a set of B one-dimensional (1-dim) shape functions which accurately approximate the underlying function f_j. Later, we will show how to learn these functions. Let us call these scalar-valued shape functions $h_b : \mathbb{R} \to \mathbb{R}, b = 1, \ldots, B$.

We create multi-dimensional (multi-dim) shape functions via tensor products of the 1-dim shape functions. For a given multi-index $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d) \in \mathbb{Z}_+^d$, we define

$$H_{\boldsymbol{\alpha}}(\mathbf{x}) = \prod_{k=1}^{d} h_{\alpha_k}(x_k). \tag{3.3}$$

where $\alpha_k \in \{0, 1, \ldots, B\}$, $h_0(\cdot) = 1$ is a constant function, and $h_1(x), \ldots, h_B(x)$ are the B 1-dim shape functions. For instance, if $d = 4$, then $\boldsymbol{\alpha} = (1, 2, 0, 1)$ will correspond to

$$H_{\boldsymbol{\alpha}}(\mathbf{x}) = h_1(x_1)h_2(x_2)h_0(x_3)h_1(x_4) = h_1(x_1)h_2(x_2)h_1(x_4). \tag{3.4}$$

Note that the same scalar shape function is being applied to x_1 and x_4. We define $\|\boldsymbol{\alpha}\|_0$ as the number of nonzero elements in $\boldsymbol{\alpha}$. In the previous example, we have $\|\boldsymbol{\alpha}\|_0 = 3$. There are several other $H_{\boldsymbol{\alpha}}$ functions with $\|\boldsymbol{\alpha}\|_0 = 3$. Since $\|\boldsymbol{\alpha}\|_0$ determines the number of 1-dim shape functions involved in creating $H_{\boldsymbol{\alpha}}$, we call $\|\boldsymbol{\alpha}\|_0$ the complexity of $H_{\boldsymbol{\alpha}}$.

Given the definition of $H_{\boldsymbol{\alpha}}$, we can now expand to parameterize f_j as follows:

$$f_j(\mathbf{x}; \boldsymbol{\theta}) = \sum_{m=0}^{M} \sum_{\|\boldsymbol{\alpha}\|_0 = m} \beta_{\boldsymbol{\alpha}}^j H_{\boldsymbol{\alpha}}(\mathbf{x}), \tag{3.5}$$

where $\boldsymbol{\theta}$ includes all the parameters of the f_j. We will get back to this point later when we define the structure of our neural network to model f_j. The equation (3.5) parameterizes

f_j by 1) computing the weighted sum of the multi-dim shape functions with the same complexity and 2) computing the sum over the shape functions with different complexities.

Let us consider a few cases of (3.5). If we set $M = 0$, then all elements of $\boldsymbol{\alpha}$ have to be 0. This gives us $H_{\boldsymbol{\alpha}} = 1$, which is a constant function. If we set $M = 1$, we obtain an additive model with no interactions: $f_j(\mathbf{x}; \boldsymbol{\theta}) = \beta_0^j + \sum_{b=1}^{B} \sum_{k=1}^{d} \beta_{b,k}^j h_b(x_k)$. When $M = 2$, we obtain pairwise interactions between variables, etc. In general, as we increase M, we obtain more complex combinations of the shape functions h_b.

If we consider a set A that contains all multi-indices in the double sum above, i.e., $A = \{\boldsymbol{\alpha} \,|\, 0 \leq \|\boldsymbol{\alpha}\|_0 \leq M\}$, we can equivalently write (3.5) as:

$$f_j(\mathbf{x}; \boldsymbol{\theta}) = \sum_{\boldsymbol{\alpha} \in A} \beta_{\boldsymbol{\alpha}}^j H_{\boldsymbol{\alpha}}(\mathbf{x}). \tag{3.6}$$

This equation will be helpful when we explain the structure of the output layer of our neural network in the next subsection.

This way of parameterizing f_j gives us two important advantages. First, our model remains interpretable, no matter how we learn the 1-dim shape functions, the weights $\beta_{\boldsymbol{\alpha}}^j$, and the multi-indices $\boldsymbol{\alpha}$. We can always graph the 1-dim shape functions $h_b(x)$—they are scalar functions of x. This approach contrasts directly with black-box modeling of vector fields via neural networks [41, 42].

Second, we gain direct control over the complexity of the model through M and multi-indices. If we make M smaller, then we will use *less complex* functions to approximate f_j. Also, in (3.5), we can further simplify the model by summing over a subset of multi-indices with $\|\boldsymbol{\alpha}\|_0 = m$ (instead of using all of them). In short, we limit the complexity of the model in the beginning, rather than using numerous shape functions and then relying on sparsity-promoting regularization to select only a few of them [7, 46].

3.2.2 Architecture of the neural network

We use a neural network to model the parameterized f_j in (3.5). Here, we explain the layers of the neural network in detail. The architecture has been visualized in Fig. 3.1.

The 1-dim shape functions $\{h_b\}_{b=1}^{B}$. The first D hidden layers of the network create the 1-dim shape functions. Each hidden layer is characterized by a weight matrix $\mathbf{W}_i$ and a bias vector $\mathbf{w}_i$. Let $\phi : \mathbb{R} \to \mathbb{R}$ denote the activation function. We follow the convention that when ϕ is applied to a vector $\mathbf{v} \in \mathbb{R}^m$, the result is the component-wise application $(\phi(v_1), \phi(v_2), \ldots, \phi(v_m))$.

Let us detail each layer:

- The first layer takes as input the scalar x_k, the kth component of $\mathbf{x}$. Hence, the first weight matrix $\mathbf{W}_1$ must have one column. The number of rows of $\mathbf{W}_1$ is the number of neurons or units in the first hidden layer. Then the output of this layer can be written as: $\mathbf{h}^1 = \phi(\mathbf{W}_1 x_k + \mathbf{w}_1)$. We choose $\phi(x) = \tanh(x)$.

- The intermediate weight matrices $\mathbf{W}_2, \ldots, \mathbf{W}_D$ must have dimensions that match. For instance, the number of columns of $\mathbf{W}_{i+1}$ must equal the dimension of $\mathbf{h}^i$, i.e., the number of neurons/units in layer i. Likewise the number of rows of $\mathbf{W}_{i+1}$ must be the number of neurons/units in layer $i+1$. To obtain B shape functions, $\mathbf{W}_D$ must have B rows. Then we can write: $\mathbf{h}^{i+1} = \phi(\mathbf{W}_{i+1}\mathbf{h}^i + \mathbf{w}_i), i = 1, \ldots, D-1$. For $i = 2, \ldots, D-1$ we choose $\phi = \tanh$, and for the last layer ($i = D$), we set $\phi(x) = x$, the identity.

Note that the output of the Dth hidden layer depends on the input x_k through a chain of compositions:

$$\mathbf{h}^D = \phi(\mathbf{W}_D \phi(\mathbf{W}_{D-1} \cdots \phi(\mathbf{W}_1 x_k + \mathbf{w}_1) \cdots + \mathbf{w}_{D-1}) + \mathbf{w}_D)$$

This same output, $\mathbf{h}^D$, is a B-dimensional vector; each component of this vector is a 1-dim shape function:

$$\mathbf{h}^D = [h_1(x_k), \ldots, h_B(x_k)] \in \mathbb{R}^B.$$

Note that $\{h_b(x_k)\}_{b=1}^{B}$ corresponds to the 1-dim functions defined in the previous subsection. We can now extend this idea to the full input $\mathbf{x}$. By applying $\mathbf{h}^D$ to each x_k for $k = 1, \ldots, d$, we obtain dB outputs.

Multi-dim shape functions. To model $H_{\boldsymbol{\alpha}}(\mathbf{x})$ defined in (3.3), we define a multiplication neuron. This neuron takes as input $\mathbf{v} \in \mathbb{R}^d$ and produces as output $\prod_{i=1}^{d} v_i \in \mathbb{R}$, the product of all its components.

To compute $H_{\boldsymbol{\alpha}}(\mathbf{x})$, we use the nonzero indices in $\boldsymbol{\alpha}$ to select a subset of the 1-dim shape functions from the Dth layer. We feed this subset of shape functions to the multiplication neuron, and the output is $H_{\boldsymbol{\alpha}}(\mathbf{x})$. The number of multiplication neurons is the same as the number of muti-indices, i.e., size of the set A from (3.6). We collect these multiplication neurons into a multiplication layer, which gives us multi-dim shape functions $H_{\boldsymbol{\alpha}}(\mathbf{x})$ as defined in (3.3).

Output layer. The last layer of our network is a linear layer that combines the $|A|$ multi-dimensional shape outputs of the previous layer to generate f_j. This layer contains a weight matrix $\mathbf{B} \in \mathbb{R}^{1\times|A|}$, where the values in $\mathbf{B}$ correspond to the coefficients β^j_α in (3.6).

3.2.3 Extending the idea to the d dimensions of f

Thus far we have described our model for f_j. We now extend this to a full model for the vector field $\mathbf{f}$. We keep the parameterization described above for the 1-dim and multi-dim shape functions. Given $H_{\boldsymbol{\alpha}}$, we parameterize $\mathbf{f}$ as :

$$\mathbf{f}(\mathbf{x};\boldsymbol{\theta}) = \left[\sum_{\boldsymbol{\alpha}\in A} \beta^1_{\boldsymbol{\alpha}} H_{\boldsymbol{\alpha}}(\mathbf{x}), \sum_{\boldsymbol{\alpha}\in A} \beta^2_{\boldsymbol{\alpha}} H_{\boldsymbol{\alpha}}(\mathbf{x}), \ldots, \sum_{\boldsymbol{\alpha}\in A} \beta^d_{\boldsymbol{\alpha}} H_{\boldsymbol{\alpha}}(\mathbf{x})\right]. \tag{3.7}$$

In other words, each output is achieved by a weighted combination of the fixed multi-dim shape functions. There are $|A|$ coefficients β for each dimension, which gives us a total of $|A|d$ coefficients.

We achieve this extension with only a small change to the neural network architecture. Since the 1-dim and multi-dim shape functions are shared across all f_j, we need only change the output layer. The output layer must now have d outputs instead of 1. Hence the weight matrix $\mathbf{B}$ must be $d \times |A|$. The jth row of $\mathbf{B}$ contains the coefficients that determine f_j, the jth component of $\mathbf{f}$.

As we mentioned before, $\boldsymbol{\theta}$ is a vector containing all the parameters of the $\mathbf{f}$. Since we model $\mathbf{f}$ using a neural network, $\boldsymbol{\theta}$ contains all the parameters of the network.

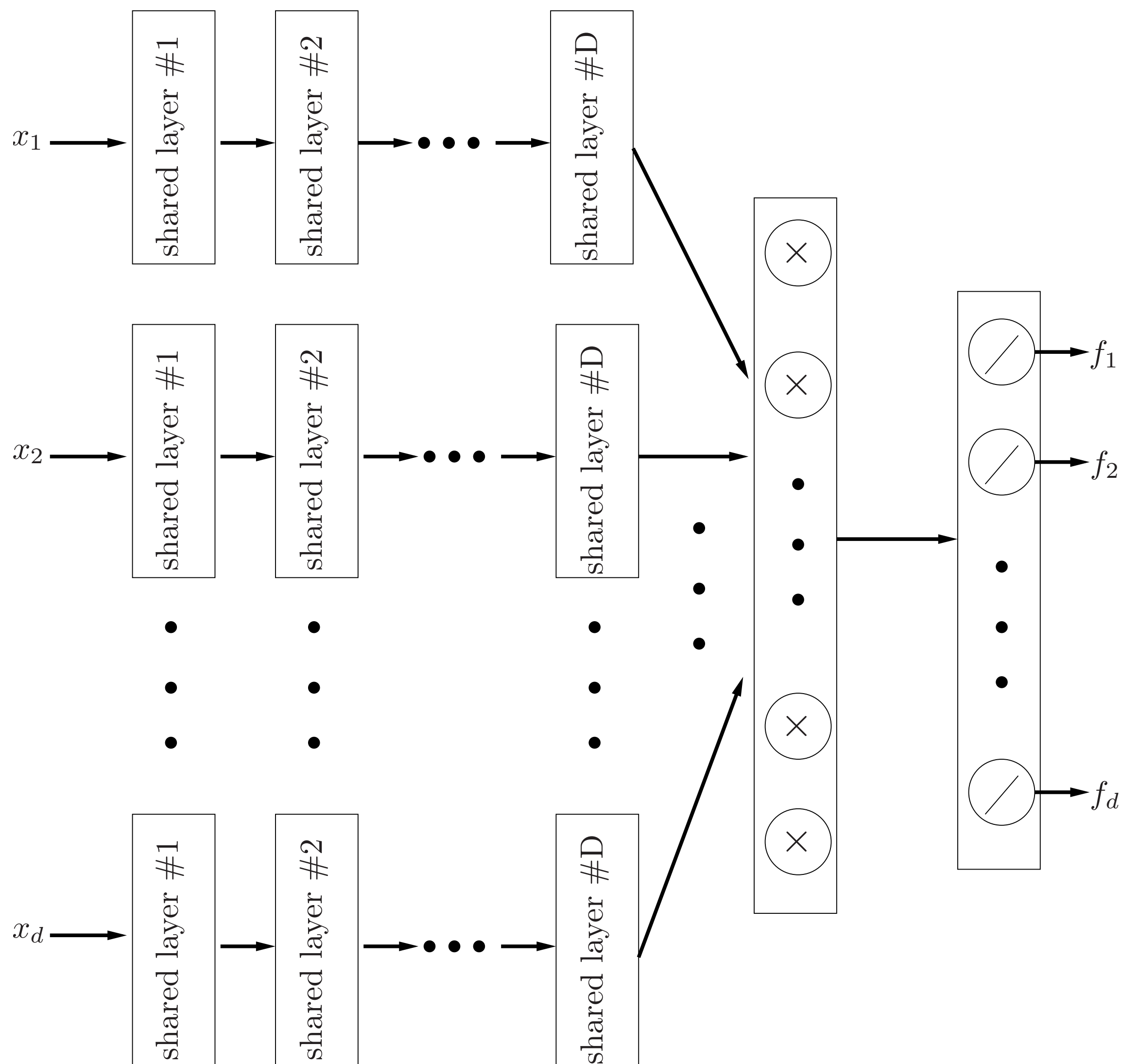

Figure 3.1: Architecture of our neural network. The first layer takes as input the scalar x_k, the kth component of $\mathbf{x}$. The output of the Dth shared (hidden) layer is B 1-dim shape functions for each dimension (a total of Bd outputs). The multiplication layer only contains multiplication neurons. It takes the Bd inputs and generates multi-dim shape functions. The last layer has d neurons with identity activation functions. Each neuron corresponds to one of the dimensions.

3.2.4 Objective function and optimization

To learn all neural network's parameters (denoted by $\boldsymbol{\theta}$), we must define an objective function and corresponding optimization problem. One possibility is to minimize the difference between the two sides of the ODE in (3.1). However, we do not have access to the clean states $\mathbf{X}$—we are given $\mathbf{Y}$, noisy observations at discrete times. Our goal is to simultaneously *filter* these noisy observations while *learning* the neural network's parameters. In what follows, we detail an alternating minimization algorithm that accomplishes this goal. This algorithm alternates between two kinds of steps: (i) a *learning* step in which we fix $\mathbf{X}$ and minimize over the neural network's parameters $\boldsymbol{\theta}$; (ii) a *filtering* step in which we fix $\boldsymbol{\theta}$ and minimize over the clean states $\mathbf{X}$.

We begin by discretizing the ODE. While we will explain later how to use higher-order discretizations, we focus on the first-order explicit Euler discretization for the moment:

$$\mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)} = \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})\Delta_i, \quad i = 1, ..., T-1 \tag{3.8}$$

Our objective function measures the L^2-norm mismatch between the left- and right-hand sides of (3.8):

$$E(\mathbf{X}, \boldsymbol{\theta}) = \textstyle\sum_{i=1}^{T-1} \left\| (\mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)})/\Delta_i - \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta}) \right\|^2. \tag{3.9}$$

Similar to our approach in chapter 2, we iteratively optimize the following Euler BCD-proximal objective function:

$$\mathbf{X}^{*(n)}, \boldsymbol{\theta}^{*(n)} = \underset{\boldsymbol{\theta}, \mathbf{X}}{\operatorname{argmin}} \left\{ E(\mathbf{X}, \boldsymbol{\theta}) + \lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|^2 \right\}, \quad \mathbf{X}^{*(0)} = \mathbf{Y}, \tag{3.10}$$

where we represent the states and parameters *learned* at iteration n by $\mathbf{X}^{*(n)}$ and $\boldsymbol{\theta}^{*(n)}$. Note that we initialize with $\mathbf{X}^{*(0)} = \mathbf{Y}$, i.e., for our first iteration, we treat the observations as the clean states. We then repeat the following steps until convergence.

$$\text{learning:} \quad \boldsymbol{\theta}^{*(n)} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left\{ E(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}) \right\} \tag{3.11a}$$

$$\text{filtering:} \quad \mathbf{X}^{*(n)} = \underset{\mathbf{X}}{\operatorname{argmin}} \left\{ E(\mathbf{X}, \boldsymbol{\theta}^{*(n)}) + \lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|^2 \right\} \tag{3.11b}$$

Note that the data $\mathbf{Y}$ is only used when $n = 0$. In subsequent iterations $n > 0$, the $\lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|_F^2$ term in (3.11b) constrains $\mathbf{X}$ to be near $\mathbf{X}^{*(n-1)}$. As the iteration

number n increases, the learned states are allowed to slowly step away from $\mathbf{Y}$. This enables the algorithm to handle situations in which $\mathbf{Y}$ is heavily contaminated with noise.

We can interpret (3.11b) naturally using the notion of proximal operators [37], as we explained in chapter 2. The conclusion is that this proximal step approximates a gradient descent step of the following form

$$\mathbf{X}^{*(n)} = \mathbf{X}^{*(n-1)} - (2\lambda)^{-1}\nabla_{\mathbf{X}} E(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}^{*(n)}) + o((2\lambda)^{-1}). \tag{3.12}$$

It is now clear that λ plays the role of an inverse step size. With this in mind, we can now contrast our method with the regularization described by [46]. We initialize our algorithm with the data $\mathbf{Y}$; subsequently, the algorithm may take many proximal steps of the form (3.12) to reach a desired optimum. If the data $\mathbf{Y}$ is heavily contaminated with noise, it may be wise to move far away from $\mathbf{Y}$ as we iterate.

In contrast, [46] uses the regularization term $\lambda\|\mathbf{X} - \mathbf{Y}\|^2$. This will constrain $\mathbf{X}$ to be in a neighborhood of $\mathbf{Y}$; the diameter of this neighborhood is inversely related to λ. When the magnitude of the noise $\mathbf{Z}$ is small, searching for $\mathbf{X}$ in a small neighborhood of $\mathbf{Y}$ is reasonable. For real data problems in which the magnitude of $\mathbf{Z}$ is unknown, however, choosing λ *a priori* becomes difficult.

By $\mathbf{X}^*$ and $\boldsymbol{\theta}^*$, we denote the *learned states* and *learned parameters* of our algorithm at the time of convergence. We use $\hat{\mathbf{X}}$ to denote our algorithm's *predicted states*. We compute $\hat{\mathbf{X}}$ by iteratively applying the following scheme, initialized with $\hat{\mathbf{x}}_{(t_1)} = \mathbf{x}^*_{(t_1)}$:

$$\hat{\mathbf{x}}_{(t_{i+1})} = \hat{\mathbf{x}}_{(t_i)} + \mathbf{f}(\hat{\mathbf{x}}_{(t_i)}; \boldsymbol{\theta})\Delta_i, \quad i = 2, \dots, T_1. \tag{3.13}$$

Note that the predicted state is initialized to be the learned state $\mathbf{x}^*_{(t_1)}$ and not the raw observation $\mathbf{y}_{(t_1)}$, since we believe $\mathbf{x}^*_{(t_1)}$ is closer to the clean state. We confirm this later in our experiments. Also note that, by construction, the predicted states will yield a zero of the mismatch function E defined in (3.9).

The pseudocode of our method can be found in Algorithm 3.1.

Algorithm 3.1 Pseudo-code of our proposed method for learning governing equations

Input: A set of T noisy observations $\mathbf{Y} = [\mathbf{y}_{(t_1)}, \ldots, \mathbf{y}_{(t_T)}] \in \mathbb{R}^{d \times T}$, the time differences $\{\Delta_i = t_{i+1} - t_i\}_{i=1}^{T-1}$, structure of the neural network (number of hidden layers, number of neurons per layer, and number of 1-dim shape functions B), the value of the hyperparameter λ, and the initialization of the neural network's weights $\boldsymbol{\theta}^{*(0)}$.

1: $\mathbf{X}^{*(0)} = \mathbf{Y}$
2: $n = 0$
3: **repeat**
4: $\quad n = n + 1$
5: $\quad$• *Learning step (training the neural network)*: Compute $\boldsymbol{\theta}^{*(n)}$ given $\mathbf{X}^{*(n-1)}$ via (3.11a) [Euler] or (3.18a) [multistep].
6: $\quad$• *Filtering step (learning the states)*: Compute $\mathbf{X}^{*(n)}$ given $\boldsymbol{\theta}^{*(n)}$ via (3.11b) [Euler] or (3.18b) [multistep].
7: **until** convergence
8: Compute the predicted states $\hat{\mathbf{X}}$ by repeatedly applying (3.13) [Euler] or (3.16) [multistep], where $\boldsymbol{\theta} = \boldsymbol{\theta}^{*(n)}$ and $\mathbf{x}_{(t_1)} = \mathbf{x}^{*(n)}_{(t_1)}$.
9: **return** $\boldsymbol{\theta}^{*(n)}$ and $\hat{\mathbf{X}}$ as the estimated neural network's parameters and predicted states.

Traning the neural network: solving the subproblem (3.11a)

In subproblem (3.11a) of our algorithm, we fix the states to $\mathbf{X} = \mathbf{X}^{*(n-1)}$ and learn the parameters $\boldsymbol{\theta}$ by minimizing the following:

$$\min_{\boldsymbol{\theta}} \textstyle\sum_{i=1}^{T-1} \left\| (\mathbf{x}^{*(n-1)}_{(t_{i+1})} - \mathbf{x}^{*(n-1)}_{(t_i)})/\Delta_i - \mathbf{f}(\mathbf{x}^{*(n-1)}_{(t_i)}; \boldsymbol{\theta}) \right\|^2, \tag{3.14}$$

where $\mathbf{f}$ is our neural network and $\boldsymbol{\theta}$ is the set of all its parameters (weights of the layers). Note that this is a regression problem, where the input is $\mathbf{x}^{*(n-1)}_{(t_i)}$ and the corresponding target is $(\mathbf{x}^{*(n-1)}_{(t_{i+1})} - \mathbf{x}^{*(n-1)}_{(t_i)})/\Delta_i$, for $i = 1\ldots, T-1$. In other words, optimizing the subproblem (3.11a) is equivalent to training a neural network to solve a regression problem.

Initializing the weights of the neural network before training is an important matter because different initializations will lead to converging to different places. Since the states change slowly at each iteration, it is reasonable to believe that the optimal parameters

at iteration n should be close to the learned parameters of the previous iteration. So we initialize the parameters at iteration n by $\boldsymbol{\theta}^{*(n-1)}$.

Learning the states: solving the subproblem (3.11b)

In subproblem (3.11b) of our algorithm, we fix the network's parameters to $\boldsymbol{\theta} = \boldsymbol{\theta}^{*(n)}$ and learn the states by minimizing the following:

$$\min_{\mathbf{X}} \sum_{i=1}^{T-1} \left\| (\mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)})/\Delta_i - \mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta}^{*(n)}) \right\|^2 + \lambda \left\| \mathbf{X} - \mathbf{X}^{*(n-1)} \right\|^2. \tag{3.15}$$

The states $\mathbf{X}$ appear as the input of the network, as a part of the target of the network, and also in the regularization term. We use the gradient descent algorithm to optimize the objective function and learn the states. We initialize the optimization at iteration n from the learned states of the previous iteration $\mathbf{X}^{*(n-1)}$.

3.2.5 Overfitting in learning f and how to avoid it

Subproblem (3.11a) of our algorithm is to fit a neural network to the current learned observations. Considering that filtering is never perfect, $\mathbf{X}^{*(n)}$ will always contain some noise. In this context, how much should we train the neural network—specifically, how many epochs?

If we overtrain the network, it will learn the noise of the target. Note that the target for $\mathbf{f}(\mathbf{x}_{(t_i)}; \boldsymbol{\theta})$ is $(\mathbf{x}_{(t_{i+1})} - \mathbf{x}_{(t_i)})/\Delta_t$. Hence overfitting in our problem does not mean that $\mathbf{f}$ will be able to predict the noisy observations instead of the clean states; it means that the $\mathbf{f}$ that we learn is wrong and cannot be used for the prediction later. So it is crucial to avoid overfitting.

Let us assume we are at iteration n of our algorithm working on the learning subproblem (3.11a). Suppose we have trained the neural network for e epochs. Let $\hat{\mathbf{X}}_e^{(n)}$ and $\mathbf{X}^{*(n-1)}$ denote, respectively, the current predicted and learned states. We call the distance between $\hat{\mathbf{X}}_e^{(n)}$ and $\mathbf{X}^{*(n-1)}$ the current prediction error, and we use this distance to detect overfitting. If we find that this distance increases as we train the network for more epochs, we stop training, i.e., we declare ourselves done with (3.11a). We then set $\boldsymbol{\theta}^{*(n)}$

to be the network's parameters that result in minimum prediction error, and we proceed with the filtering subproblem (3.11b).

3.2.6 Higher-order discretization

Extending the idea from the Euler method to the higher-order discretization is similar to the section 2.3.3 in the ODE parameter estimation problem. Here, we give a short summary.

The Euler method is a first-order method. If we seek a more accurate discretization, we can apply a multistep (m-step) method, where the idea is to use the previous m states to compute the next state. In general, an explicit linear m-step method to discretize (3.1) can be formulated as

$$\mathbf{x}_{(t_{i+1})} = \sum_{j=0}^{m-1} a_j \mathbf{x}_{(t_{i-j})} + \Delta_i \sum_{j=0}^{m-1} b_j \mathbf{f}(\mathbf{x}_{(i-j)}; \boldsymbol{\theta}), \tag{3.16}$$

where Δ_i is the time step. There are several strategies to determine the coefficients $\{a_j\}_{j=0}^{m-1}$ and $\{b_j\}_{j=0}^{m-1}$. Further information regarding different strategies can be found in [24, 35].

When using a general m-step discretization method, we define our objective function as follows:

$$E_{\mathrm{m}}(\mathbf{X}, \boldsymbol{\theta}) = \sum_{i=1}^{T-1} \left\| \mathbf{x}_{(t_{i+1})} - \textstyle\sum_{j=0}^{k-1} a_j \mathbf{x}_{(t_{i-j})} - \Delta_i \textstyle\sum_{j=0}^{k-1} b_j \mathbf{f}(\mathbf{x}_{(i-j)}; \boldsymbol{\theta}) \right\|^2, \tag{3.17}$$

where $k = \min(i-1, m)$ is the order of the discretization method to predict the state $\mathbf{x}_i$. Now, we repeat the following steps until convergence:

$$\text{learning:} \quad \boldsymbol{\theta}^{*(n)} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, E_{\mathrm{m}}(\mathbf{X}^{*(n-1)}, \boldsymbol{\theta}) \tag{3.18a}$$

$$\text{filtering:} \quad \mathbf{X}^{*(n)} = \underset{\mathbf{X}}{\operatorname{argmin}} \left\{ E_{\mathrm{m}}(\mathbf{X}, \boldsymbol{\theta}^{(*n)}) + \lambda \|\mathbf{X} - \mathbf{X}^{*(n-1)}\|^2 \right\} \tag{3.18b}$$

3.3 Experiments

In this section, we introduce the ODEs in our experiments, give the details of our implementations, show how robust our method is with respect to the hyperparameter λ, and finally compare our method with several state-of-the-art methods.

Rössler attractor. This three-dimensional nonlinear system has a chaotic attractor [44]:

$$\frac{dx_0}{dt} = -x_1 - x_2 \qquad \frac{dx_1}{dt} = x_0 + ax_1 \qquad \frac{dx_2}{dt} = b + x_2(x_0 - c). \tag{3.19}$$

The states are three-dimensional. We use the same settings as in [43]. We set the parameters as $a = 0.2, b = 0.2$, and $c = 3$. With the initial condition $\mathbf{x}_{(1)} = [1.13, -1.74, 0.02]$, we generate clean states in the time range of $[0, 20]$, with a spacing of $\Delta t = 0.05$.

FitzHugh–Nagumo model. This model describes spike generation in squid giant axons [17, 33]. It has two nonlinear equations:

$$\frac{dx_0}{dt} = c(x_0 - \frac{(x_0)^3}{3} + x_1) \qquad \frac{dx_1}{dt} = -\frac{1}{c}(x_0 - a + bx_1), \tag{3.20}$$

where x_0 is the voltage across an axon and x_1 is the outward current. In this model, the states are two-dimensional. We use the same settings as in [43]. We set the parameters as $a = 0.5, b = 0.2$, and $c = 3$. With initial condition $\mathbf{x}_{(1)} = [-1, 1]$, we generate clean states in the time range of $[0, 20]$ with a spacing of $\Delta t = 0.05$.

Double pendulum. The double pendulum is a classic mechanical system exhibiting chaos with challenging dynamics. The double pendulum may be modeled by the following equations of motion[46]:

$$\begin{aligned}
\frac{dx_0}{dt} &= \frac{l_2 x_2 - l_1 x_3 \cos(x_0 - x_1)}{l_1^2 l_2 (m_1 + m_2 \sin^2(x_0 - x_1))} \\
\frac{dx_1}{dt} &= \frac{-m_2 l_2 x_2 \cos(x_0 - x_1) + (m_1 + m_2) l_1 x_3}{m_2 l_1 l_2^2 (m_1 + m_2 \sin^2(x_0 - x_1))} \\
\frac{dx_2}{dt} &= -(m_1 + m_2) g l_1 \sin(x_0) - C_1 + C_2 \sin(2(x_0 - x_1)) \\
\frac{dx_3}{dt} &= -m_2 g l_2 \sin(x_1) + C_1 - C_2 \sin(2(x_0 - x_1)),
\end{aligned}$$

where:

$$\begin{aligned}
C_1 &= \frac{x_2 x_3 \sin(x_0 - x_1)}{l_1 l_2 (m_1 + m_2 \sin^2(x_0 - x_1))} \\
C_2 &= \frac{m_2 l_2^2 x_2^2 + (m_1 + m_2) l_1^2 x_3^2 - 2 m_2 l_1 l_2 x_2 x_3 \cos(x_0 - x_1)}{2 l_1^2 l_2^2 (m_1 + m_2 \sin^2(x_0 - x_1))^2}
\end{aligned}$$

In this model, x_0 and x_1 represent the respective angle of pendula from the vertical axis and x_2 and x_3 represent their conjugate momenta. As suggested in [46], we set the parameters $l_1 = l_2 = 1, m_1 = m_2 = 1$, and $g = 10$. With initial condition $\mathbf{x}_{(1)} = [1, 0, 0, 0]$, we generate clean states in the time range of $[0, 10]$ with a spacing of $\Delta t = 0.01$.

Evaluation metrics. Let $\mathbf{X}$ denote the clean states and $\hat{\mathbf{X}}$ denote the predicted states. We report the Frobenius norm of $\mathbf{X} - \hat{\mathbf{X}}$ as the prediction error. Recall that predicted states are achieved by considering $\boldsymbol{\theta}^*$ as the parameter and $\mathbf{x}^*_{(t_0)}$ as the initial state, and then repeatedly applying (3.13) or its multistep analogue (3.16).

Implementation details: structure of the network, optimization algorithm, etc. In all experiments, we use two hidden layers with 256 neurons, we set the maximum complexity to $M = d$, and we use all the 1-dim shape functions for the complexity $m = 1$.

In FitzHugh–Nagumo model and Rössler attractor, we use $B = 10$ 1-dim shape functions and we generate 10 random multi-indices for each complexity value $m > 1$. In the double pendulum, we use $B = 200$ 1-dim shape functions and we generate 500 random multi-indices for each complexity $m > 1$.

The activation function of all the hidden layers (all layers before the multiplication layer) is tanh, and the activation function of the last layer is the identity. We implement our method using the Keras[13] with TensorFlow backend [1]. To optimize the neural network, we use Adam optimizer with the parameters $\alpha = .001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We use the gradient descent optimizer of TensorFlow[1], with the learning rate of 0.01, to optimize the states $\mathbf{X}$ given the parameters.

To generate the observations with $p\%$ noise, we follow [46]. First, we generate the clean data, and then we add Gaussian noise with mean 0 to each dimension separately. The standard deviation of the Gaussian noise for the ith dimension is $\frac{std_i \times p}{100}$, where std_i is the standard deviation of the ith dimension of the clean data.

Avoiding overfitting and underfitting. One of the steps of our algorithm is to fit a neural network to the current learned observations. in Fig. 3.2 we explore the impact of

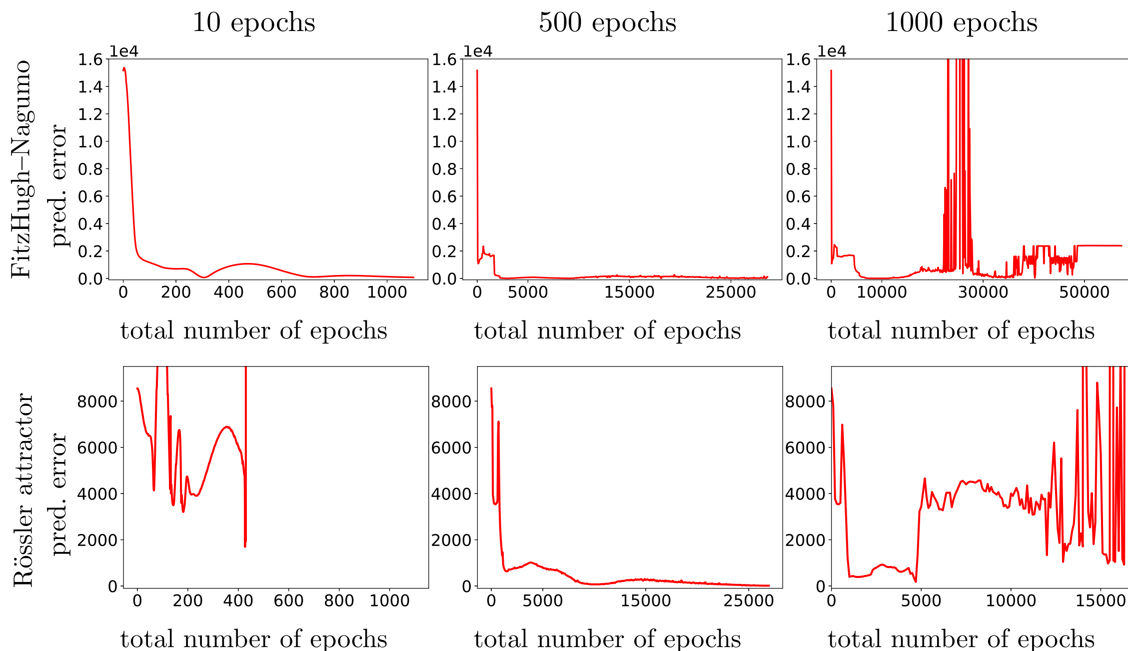

Figure 3.2: Impact of the number of epochs on the overall performance of state prediction. We train the neural network for 10, 500, and 1000 epochs, before switching to the step over learning the states. We report the error as the total number of epochs increases on FitzHugh–Nagumo and Rössler attractor models with 5% noise.

the number of epochs in training the network on the overall performance of our prediction. We show the prediction error as a function of the total number of epochs. We switch between training the network and learning the states after 10/500/1000 epochs in columns one/two/three of the Fig. 3.2. For 10 epochs in the Rössler attractor (underfitting) and for 1000 epochs in both models (overfitting), the prediction error becomes large. The error goes down smoothly when we set the number of epochs to 500.

While setting the number of epochs to 500 seems to work well in these two models, this is not necessarily the best choice for all the ODEs and for different noisy observations. In our implementation, we do not fix the number of epochs. Instead, as we train the network and after each 100 epochs, we compute the distance between the current predicted states and the noisy observations. We stop training when we find that this distance increases as we train the network for more epochs. We show the results of this implementation

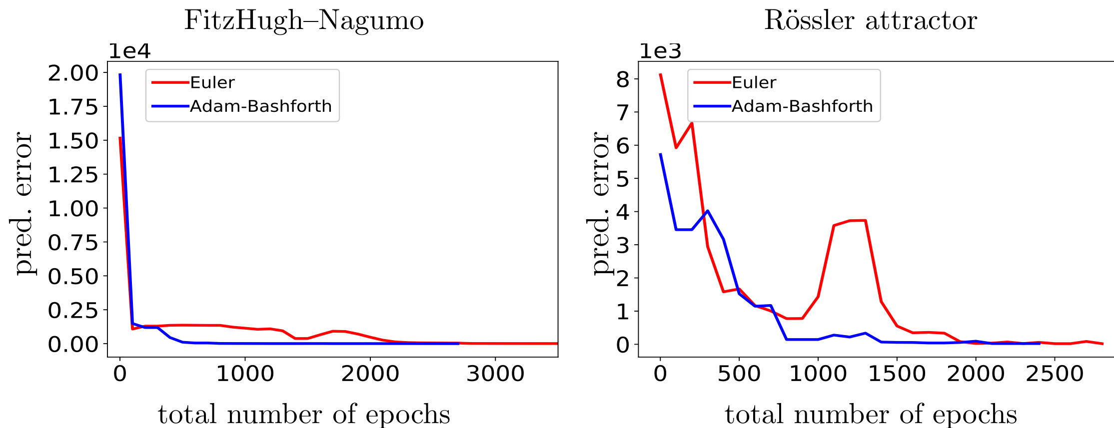

Figure 3.3: Avoiding the overfitting and underfitting without fixing the number of epochs. We let the neural network's training continue, as long as the current predicted states get closer to the noisy observations. We report the error as the total number of epochs increases on FitzHugh–Nagumo and Rössler attractor models with 5% noise.

in Fig. 3.3. As we can see, in both cases, the error goes down to 0 without any sign of overfitting or underfitting.

Higher order discretization. In Fig. 3.3, we compare the difference between the Euler and the Adam-Bashforth discretization of order three on Rössler and FitzHugh–Nagumo models with 5% noise. As we can see, the error is large at the beginning, but it gets smaller as we train the network's parameters and the states.

Robustness to the hyperparameter λ. In Fig. 3.4, we investigate the robustness of our algorithm to the hyperparameter λ in the FitzHugh–Nagumo and Rössler models, with the 1%, 5%, and 10% noisy observations. We generate 10 sets of noisy observations for each percentage of noise, run our algorithm with a specific value of λ on each of the sets of observations, and report the prediction error (each error bar in this figure corresponds to the prediction error on one set of observations).

As we see in Fig. 3.4, the errors are almost the same for different values of λ in the FitzHugh–Nagumo. In the Rössler model, we get nearly the same error with different λ values for 5% and 10% noisy observations. We get slightly better results with $\lambda = 0.01$

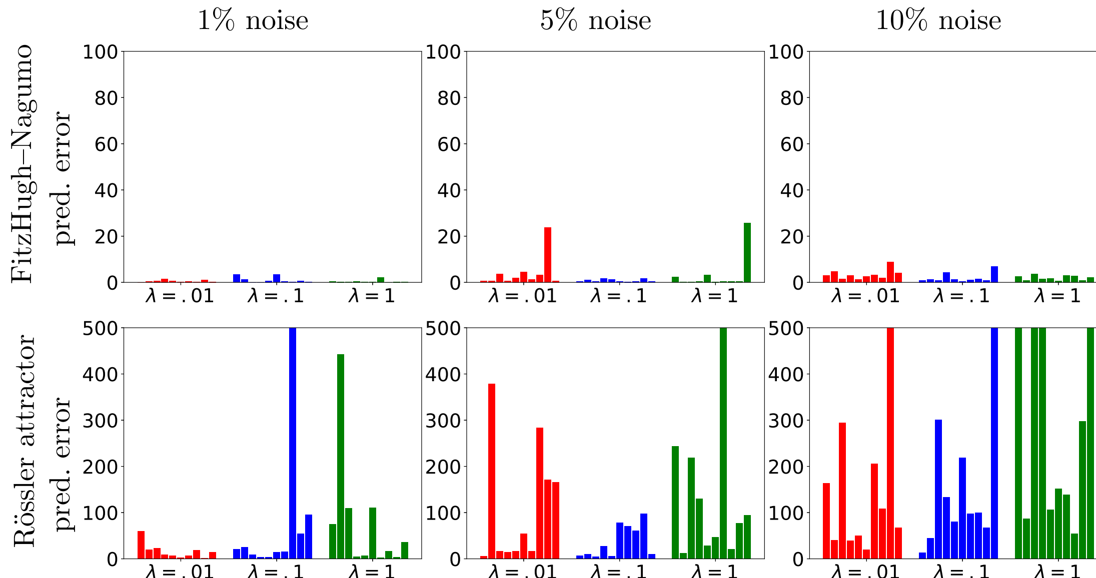

Figure 3.4: Robustness to the hyperparameter λ. We create 10 sets of observations, each with 1%, 5%, and 10% noise. We run our algorithm with $\lambda = 0.01, 0.1$, and 1 on each set separately and report the error (each bar shows the error for one of the experiments).

when we have 1% noise. This makes sense since for 1% noise, the data is almost clean, so we do not need to move $\mathbf{X}$ too much from its initializations (noisy observations). In general, we see that our algorithm is robust with respect to the value of λ.

Importance of learning the clean states X. In Fig. 3.5, we investigate how important it is to learn the states. In this figure, learn_states is our approach, where we learn the network's parameters and the states in an alternation by minimizing (3.9), and fixed_states is a method that fixes the states to the observations $\mathbf{X} = \mathbf{Y}$ and optimizes (3.9) only on the parameters of the network. The neural network structure is the same for both methods.

The left column of the Fig. 3.5 compares the prediction error of the two methods. The red curve (our approach) achieves much better errors than the other method.

The right column of this figure shows why we get a better error. We define the $\mathbf{X}$_err as the squared distance between the current states and the clean states. Note that this value is fixed for the method fixed_states and is equal to $\|Y - X\|^2$. But for our

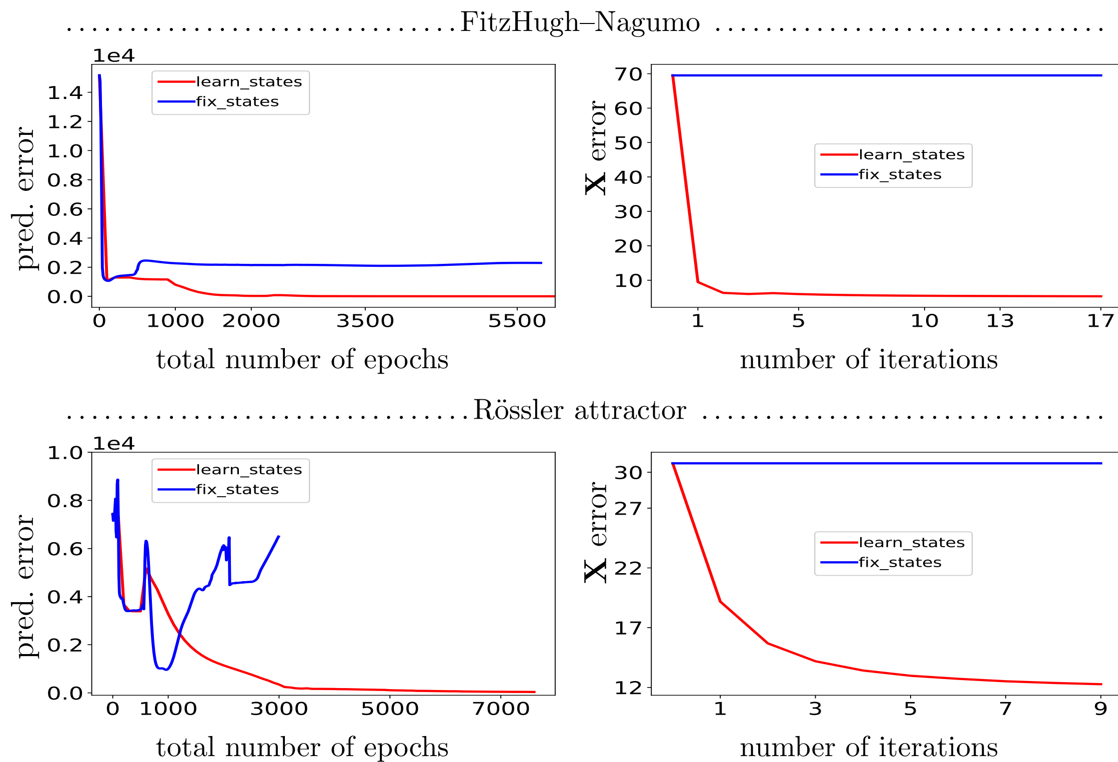

Figure 3.5: Importance of learning the clean states. Learning the states significantly improves the results. "learn_states" is our approach, while "fix_states" fixes the states to the observations and fits the network. We run the experiments on the two models with 5% noisy observations. *Left panel:* prediction error at different epochs of the methods. *Right panel:* The Euclidean distance between the current learned states and the clean states, during the alternating optimization.

method, this error changes every time we learn the states. As we can see, the error of the red curve decreases massively as we iterate over the states and the network's parameters. This makes our learned states closer to the clean states and that is why we get a better prediction error.

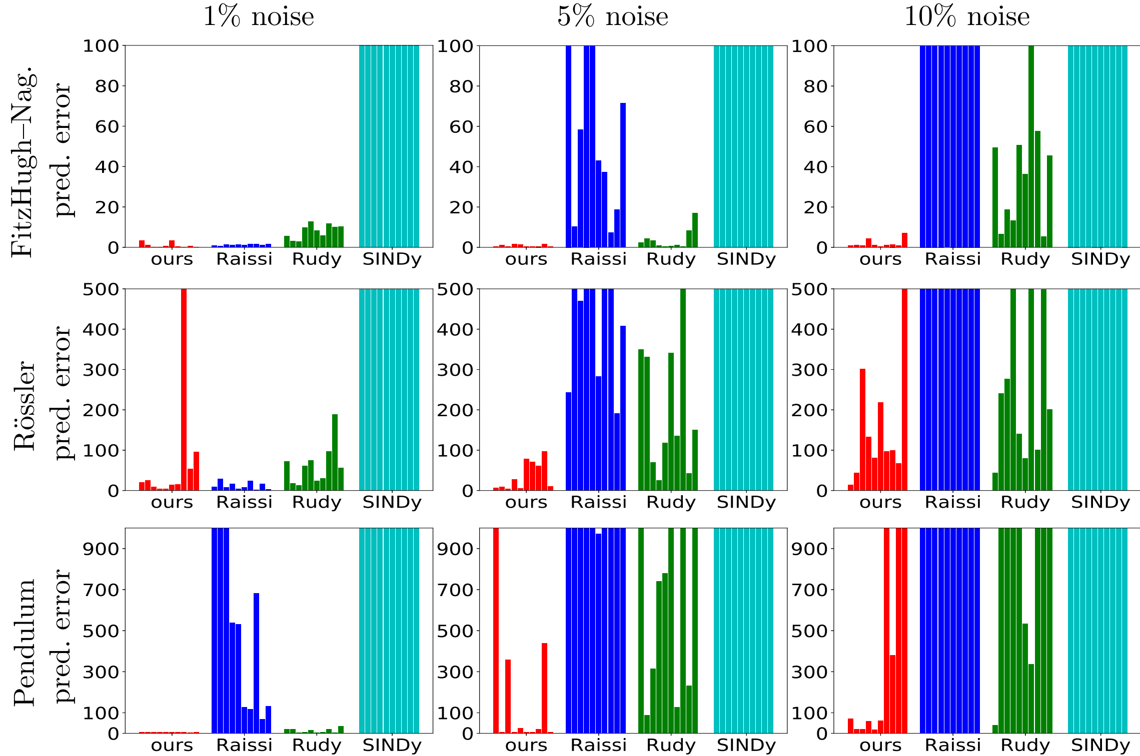

Figure 3.6: Comparison with other methods. We compare our method with Raissi et al. [41], Rudy et al. [46], and Brunton et al. [6] on three models with different amounts of noise in the observations. For each value of the noise percentage, we create 10 sets of observations, run the methods on each set separately, and report the prediction error. Each error bar shows the error in one experiment.

Comparison with other methods We compared our method with three recent works [6, 41, 46] in Fig. 3.6 on the FitzHugh–Nagumo, Rössler attractor, and double pendulum models. Noisy observations are generated in the time interval $[0, T]$, where $T = 20, 20$ and 10 seconds in the FitzHugh–Nagumo, Rössler attractor, and double pendulum models, respectively.

Both Rudy et al. [46] and Raissi et al. [41] train regular non-interpretable neural networks to learn the vector field $\mathbf{f}$. Brunton et al. [6] (SINDy) combines a set of predetermined (interpretable) shape functions to approximate the vector field $\mathbf{f}$. The results of SINDy [6] is worse than the other three methods since it uses fixed shape functions.

Raissi et al. [41] trains a network without any regularization and allows the network to overfit to the noise. We see in Fig. 3.6 that it works well on the FitzHugh–Nagumo and Rössler attractor when we have a small amount of noise (1%). But, for a larger amount of noise or a more complicated model it has a large prediction error.

Rudy et al. [46] learns both the states and the network's parameters while it tries to keep the states close to the observations. As we can see in the figure, this method is more robust to the noise than Raissi et al. [41] and Brunton et al. [6].

Finally, our method is clearly the best one, no matter what ODE we use and how much noise we add to the dataset.

Visualization of the predicted states and the shape functions We run our algorithm on the noisy observations (with 5% noise) generated from the Rössler, FitzHugh–Nagumo, and double pendulum ODEs, and visualize the predicted states and the shape functions on Fig. 3.7, Fig.3.8, and Fig. 3.9, respectively.

The first panel in each figure shows both the clean states and our predicted states over time. For better and more consistent visualization, we preprocess the data of each curve of the first panel before plotting. For all the data in a curve, we first subtract the minimum from the points, then divide them by the maximum. This makes the points of each curve between 0 and 1. For the Rössler and the double pendulum, we also multiply all the values by 5 and make them between 0 and 5. Note that we do this only for the visualization purpose. In training, we use the original values of the points.

We can see from the first panel that the predicted states and the clean states are very close, which means that our network does a good job in approximating the function $\mathbf{f}$. Note that if we train the network on clean states, then the clean and the predicted states are going to be right on top of each other (we get zero error).

The second panel shows six 1-dim shape functions that have been used in each case, and the third panel visualizes three of the multi-dim shape functions $H_{\boldsymbol{\alpha}}(\mathbf{x})$.

3.4 Conclusion

Learning governing equations given the noisy observations is an important task when the dynamical systems are complicated. Previous approaches either use fixed libraries of the interpretable shape functions or non-interpretable (black-box) powerful neural networks to learn the equations. Both approaches only work when there is a low amount of noise in observations. In this chapter, we first parameterized the vector field using the multiplication of the 1-dim shape functions. Then, we introduced a neural network with an interpretable structure to model the parameterization. By learning the clean states and parameters of the network in an alternation, we achieved an interpretable neural network which is highly robust to the noise.

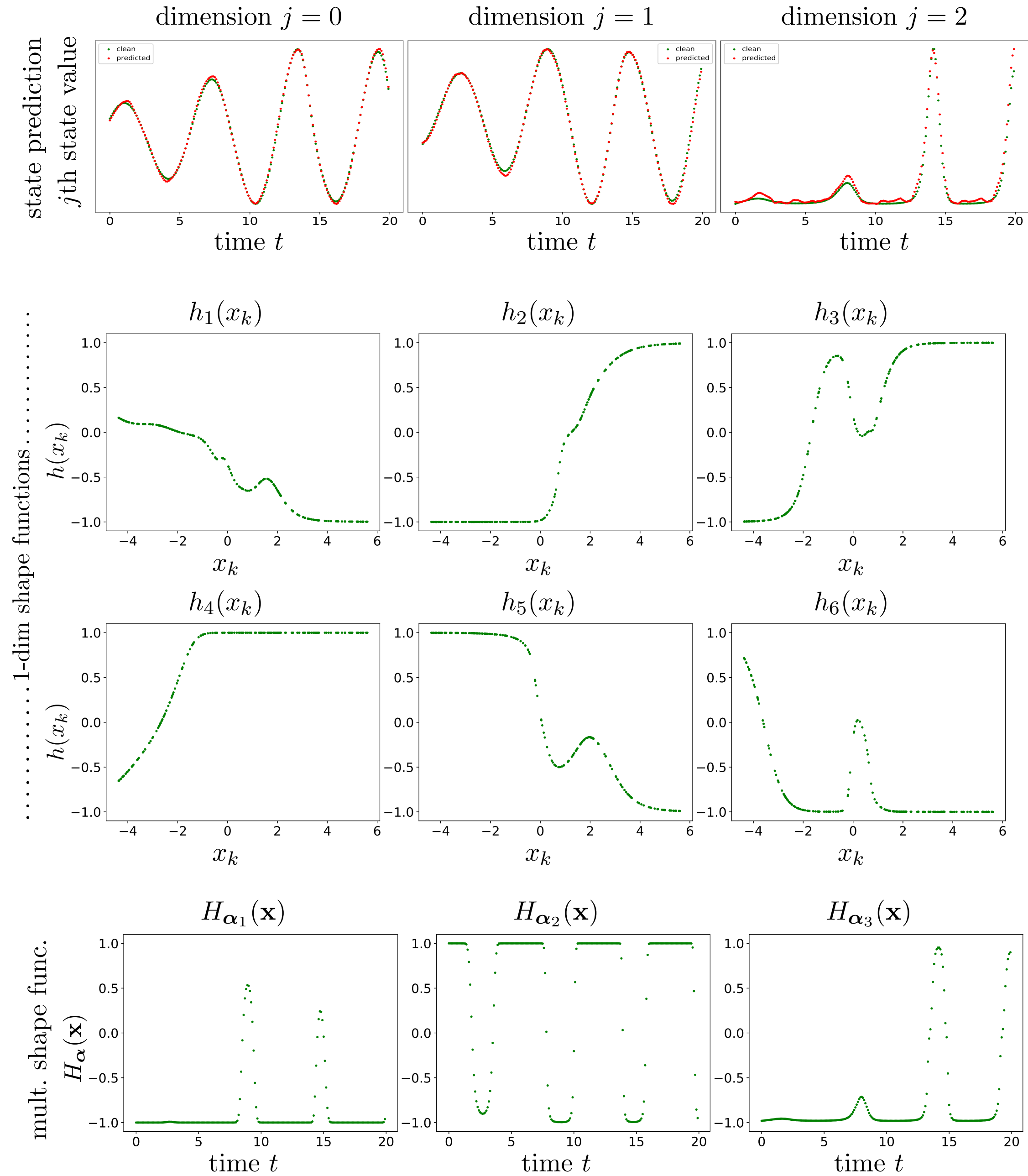

Figure 3.7: Visualization of the predicted states and the shape functions on Rössler ODE. We generate observations with 5% noise. *First panel:* the clean states and our predicted states over time. *Second panel:* visualization of the 1-dim shape functions. *Third panel:* visualization of the multi-dim shape functions.

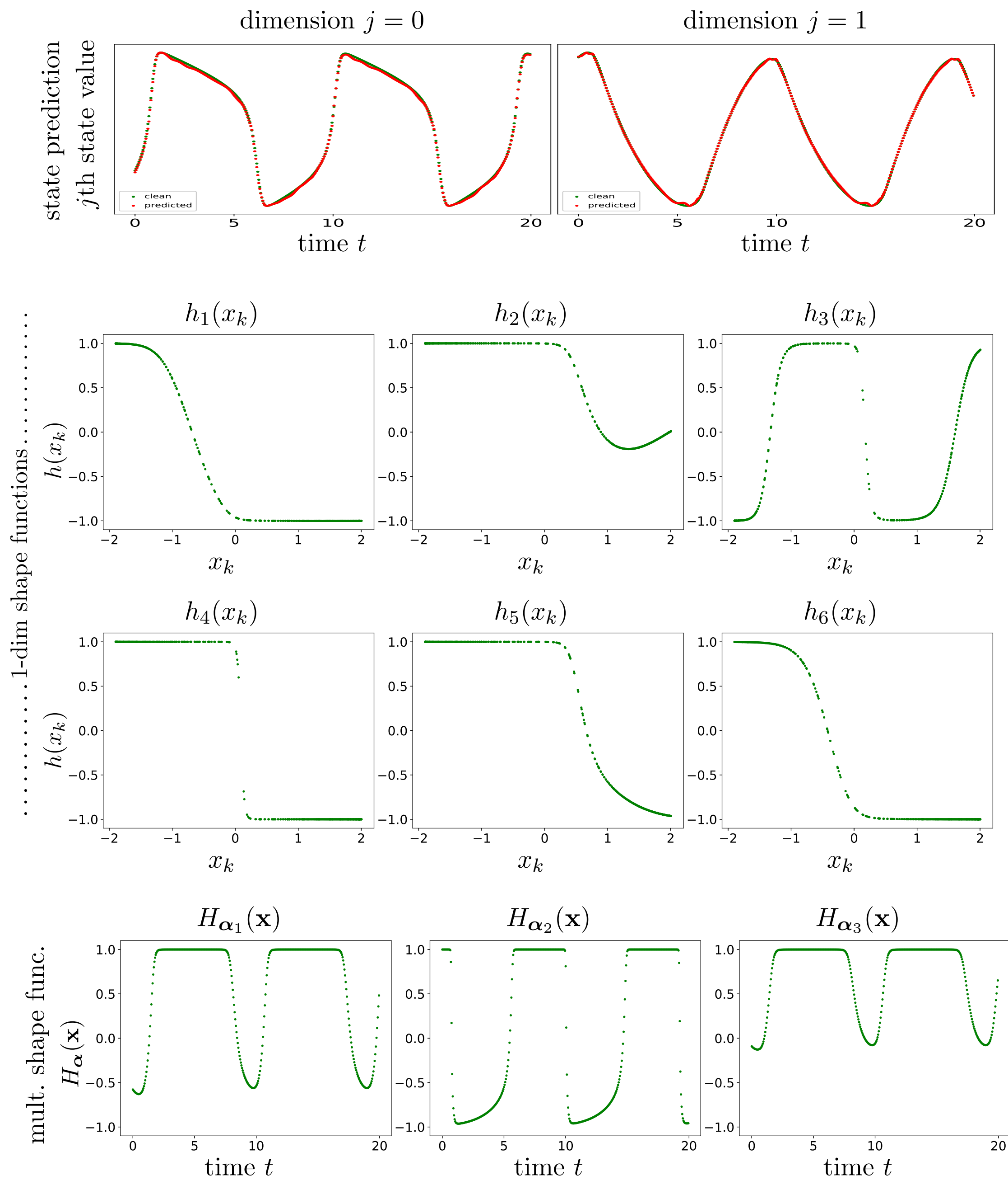

Figure 3.8: Visualization of the predicted states and the shape functions on FitzHugh–Nagumo ODE. The panels are the same as Fig. 3.7

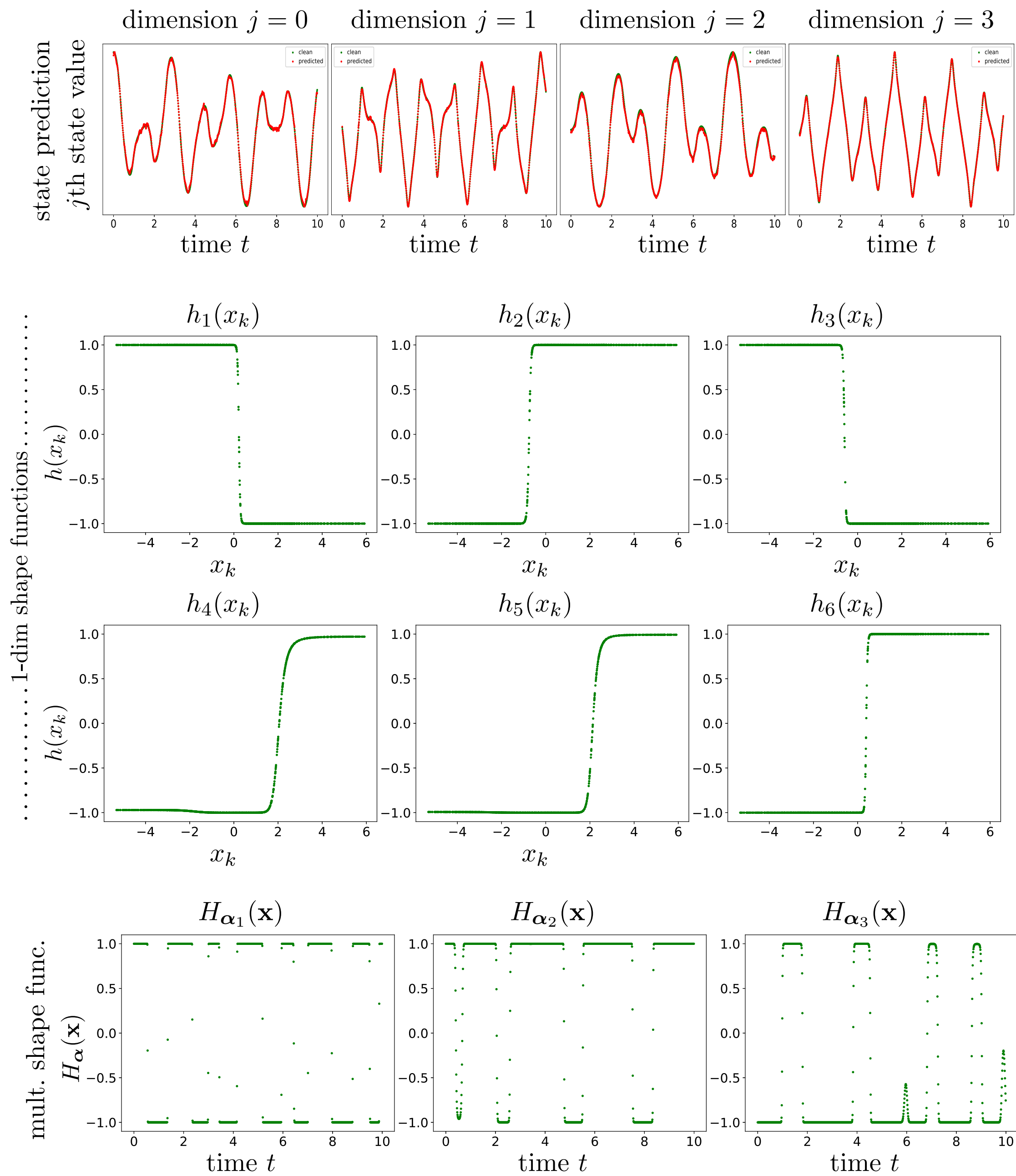

Figure 3.9: Visualization of the predicted states and the shape functions on double pendulum ODE. The panels are the same as Fig. 3.7

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

[2] Andrea Arnold, Daniela Calvetti, and Erkki Somersalo. Linear multistep methods, particle filtering and sequential Monte Carlo. *Inverse Problems*, 29(8):085007, 25, 2013.

[3] Andrea Arnold, Daniela Calvetti, and Erkki Somersalo. Parameter estimation for stiff deterministic dynamical systems via ensemble Kalman filter. *Inverse Problems*, 30 (10):105008, 30, 2014.

[4] Yonathan Bard. *Nonlinear Parameter Estimation*. Academic Press, 1973.

[5] M. Benson. Parameter fitting in dynamic models. *Ecological Modelling*, 6(2):97–115, 1979.

[6] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. 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. ISSN 1091-6490. doi: 10.1073/pnas.1517384113.

[7] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, April 2016. doi: 10.1073/pnas.1517384113.

[8] Ben Calderhead, Mark Girolami, and Neil D Lawrence. Accelerating Bayesian inference over nonlinear differential equations with Gaussian processes. In *Advances in Neural Information Processing Systems*, pages 217–224, 2009.

[9] J. Cao and H. Zhao. Estimating dynamic models for gene regulation networks. *Bioinformatics*, 24(14):1619–1624, 2008.

[10] J. Cao, L. Wang, and J. Xu. Robust estimation for ordinary differential equation models. *Biometrics*, 67(4):1305–1313, 2011.

[11] Niladri S. Chatterji and Peter L. Bartlett. Alternating minimization for dictionary learning with random initialization. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 1994–2003, 2017.

[12] Shizhe Chen, Ali Shojaie, and Daniela M. Witten. Network Reconstruction From High-Dimensional Ordinary Differential Equations. *Journal of the American Statistical Association*, 112(520):1697–1707, October 2017. doi: 10.1080/01621459.2016.1229197.

[13] François Chollet et al. Keras. `https://keras.io`, 2015.

[14] Itai Dattner and Chris A. J. Klaassen. Optimal rate of direct estimators in systems of ordinary differential equations linear in functions of the parameters. *Electronic Journal of Statistics*, 9(2):1939–1973, 2015.

[15] Frank Dondelinger, Dirk Husmeier, Simon Rogers, and Maurizio Filippone. ODE parameter inference using adaptive gradient matching with Gaussian processes. In *Artificial Intelligence and Statistics*, pages 216–228, 2013.

[16] Charles Elton and Mary Nicholson. The ten-year cycle in numbers of the lynx in canada. *Journal of Animal Ecology*, 1942.

[17] Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961.

[18] Mark Girolami. Bayesian inference for differential equations. *Theoretical Computer Science*, 408(1):4–16, 2008.

[19] Nico S Gorbach, Stefan Bauer, and Joachim M Buhmann. Scalable variational inference for dynamical systems. In *Advances in Neural Information Processing Systems*, pages 4809–4818, 2017.

[20] Shota Gugushvili and Chris A.J. Klaassen. $\sqrt{n}$-consistent parameter estimation for systems of ordinary differential equations: bypassing numerical integration via smoothing. *Bernoulli*, 18(3):1061–1098, 2012.

[21] James Henderson and George Michailidis. Network reconstruction using nonparametric additive ODE models. *PLOS ONE*, 9(4):1–15, April 2014. doi: 10.1371/journal.pone.0094003. URL `https://doi.org/10.1371/journal.pone.0094003`.

[22] D. M. Himmelblau, C. R. Jones, and K. B. Bischoff. Determination of rate constants for complex kinetics models. *Industrial & Engineering Chemistry Fundamentals*, 6(4):539–543, 1967.

[23] L.H. Hosten. A comparative study of short cut procedures for parameter estimation in differential equations. *Computers & Chemical Engineering*, 3(1-4):117–126, 1979.

[24] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, Cambridge, UK, second edition, 2009.

[25] Roger Labbe. Kalman and Bayesian filters in Python, 2014. URL `https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python`.

[26] Qunwei Li, Yi Zhou, Yingbin Liang, and Pramod K. Varshney. Convergence analysis of proximal gradient with momentum for nonconvex optimization. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2111–2119, 2017.

[27] Yuanzhi Li, Yingyu Liang, and Andrej Risteski. Recovery guarantee of weighted low-rank approximation via alternating minimization. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2358–2367, 2016. URL `http://jmlr.org/proceedings/papers/v48/lii16.html`.

[28] Hua Liang and Hulin Wu. Parameter estimation for differential equation models using a framework of measurement error in regression models. *Journal of the American Statistical Association*, 103(484):1570–1583, 2008.

[29] Edward N. Lorenz and Kerry A. Emanuel. Optimal sites for supplementary weather observations: Simulation with a small model. *Journal of the Atmospheric Sciences*, 55(3):399–414, 1998.

[30] Alfred J Lotka. The growth of mixed populations: two species competing for a common food supply. *Journal of the Washington Academy of Sciences*, 22(16/17):461–469, 1932.

[31] N. M. Mangan, S. L. Brunton, J. L. Proctor, and J. N. Kutz. Inferring Biological Networks by Sparse Identification of Nonlinear Dynamics. *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, 2(1):52–63, 2016. doi: 10.1109/TMBMC.2016.2633265.

[32] N. M. Mangan, J. N. Kutz, S. L. Brunton, and J. L. Proctor. Model selection for dynamical systems via sparse regression and information criteria. *Proc. R. Soc. A*, 473(2204):20170009, August 2017. doi: 10.1098/rspa.2017.0009.

[33] Jinichi Nagumo, Suguru Arimoto, and Shuji Yoshizawa. An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10):2061–2070, 1962.

[34] Matthew Newville, Till Stensitzki, Daniel B. Allen, and Antonino Ingargiola. LMFIT: Non-linear least-square minimization and curve-fitting for Python, 2014.

[35] Richard S. Palais and Robert Andrew Palais. *Differential Equations, Mechanics, and Computation.* Number 51 in Student Mathematical Library. American Math. Soc., Providence, RI, 2009.

[36] Richard Pang, Floris van Breugel, Michael Dickinson, Jeffrey A. Riffell, and Adrienne L. Fairhall. History dependence in insect flight decisions during odor tracking. In *PLoS Computational Biology*, 2018.

[37] Neal Parikh and Stephen P. Boyd. Proximal Algorithms. *Foundations and Trends in Optimization*, 1(3):127–239, 2014.

[38] A.A. Poyton, M.S. Varziri, K.B. McAuley, P.J. McLellan, and J.O. Ramsay. Parameter estimation in continuous-time dynamic models using principal differential analysis. *Computers & Chemical Engineering*, 30(4):698–708, feb 2006. doi: 10.1016/j.compchemeng.2005.11.008.

[39] Tong Qin, Kailiang Wu, and Dongbin Xiu. Data Driven Governing Equations Approximation Using Deep Neural Networks. *arXiv:1811.05537*, 2018. URL `https://arxiv.org/abs/1811.05537`.

[40] Markus Quade, Markus Abel, J. Nathan Kutz, and Steven L. Brunton. Sparse identification of nonlinear dynamics for rapid model recovery. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 28(6):063116, 2018. doi: 10.1063/1.5027470.

[41] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv:1711.10561*, 2017. URL `https://arxiv.org/abs/1711.10561`.

[42] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv:1711.10566*, 2017. URL `https://arxiv.org/abs/1711.10566`.

[43] J. O. Ramsay, G. Hooker, D. Campbell, and J. Cao. Parameter estimation for differential equations: a generalized smoothing approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(5):741–796, oct 2007. doi: 10.1111/j.1467-9868.2007.00610.x.

[44] Otto E Rössler. An equation for continuous chaos. *Physics Letters A*, 57(5):397–398, 1976.

[45] Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3, April 2017. doi: 10.1126/sciadv.1602614. URL `http://adsabs.harvard.edu/abs/2017SciA....3E2614R`.

[46] Samuel H. Rudy, J. Nathan Kutz, and Steven L. Brunton. Deep learning of dynamics and signal-noise decomposition with time-stepping constraints. *arXiv:1808.02578*, 2018. URL `https://arxiv.org/abs/1808.02578`.

[47] Hayden Schaeffer. Learning partial differential equations via data discovery and sparse optimization. *Proceedings of the Roya Society A*, 473(2197):20160446, 20, 2017. ISSN 1364-5021. doi: 10.1098/rspa.2016.0446.

[48] Hayden Schaeffer. Learning partial differential equations via data discovery and sparse optimization. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 473(2197):20160446, January 2017. doi: 10.1098/rspa.2016.0446.

[49] Hayden Schaeffer, Giang Tran, and Rachel Ward. Extracting Sparse High-Dimensional Dynamics from Limited Data. *SIAM Journal on Applied Mathematics*, 78(6):3279–3295, 2018.

[50] Thomas B. Schön, Andreas Svensson, Lawrence Murray, and Fredrik Lindsten. Probabilistic learning of nonlinear dynamical systems using sequential Monte Carlo. *Me-*

chanical Systems and Signal Processing, 104:866–883, May 2018. ISSN 0888-3270. doi: 10.1016/j.ymssp.2017.10.033. URL http://www.sciencedirect.com/science/article/pii/S0888327017305666.

[51] Andre Sitz, Udo Schwarz, Juergen Kurths, and Henning U. Voss. Estimation of parameters and unobserved components for nonlinear systems from noisy time series. *Phys. Rev. E*, 66(1):016210, 2002. doi: 10.1103/PhysRevE.66.016210.

[52] Jiangwen Sun, Jin Lu, Tingyang Xu, and Jinbo Bi. Multi-view sparse co-clustering via proximal alternating linearized minimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 757–766, 2015. URL http://jmlr.org/proceedings/papers/v37/sunb15.html.

[53] G. Tran and R. Ward. Exact Recovery of Chaotic Systems from Highly Corrupted Data. *Multiscale Modeling & Simulation*, 15(3):1108–1129, January 2017. doi: 10.1137/16M1086637.

[54] J. M. Varah. A spline least squares method for numerical parameter estimation in differential equations. *SIAM Journal on Scientific and Statistical Computing*, 3(1):28–46, 1982.

[55] Hulin Wu, Tao Lu, Hongqi Xue, and Hua Liang. Sparse additive ordinary differential equations for dynamic gene regulatory network modeling. *Journal of the American Statistical Association*, 109(506):700–716, 2014. ISSN 0162-1459. doi: 10.1080/01621459.2013.859617.

[56] Yangyang Xu and Wotao Yin. A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion. *SIAM J. Imaging Sciences*, 6(3):1758–1789, 2013. doi: 10.1137/120887795. URL https://doi.org/10.1137/120887795.

[57] Xinyang Yi, Constantine Caramanis, and Sujay Sanghavi. Alternating minimization for mixed linear regression. In *Proceedings of the 31th International Conference on*

Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, pages 613–621, 2014. URL http://jmlr.org/proceedings/papers/v32/yia14.html.