Chaotic Model Prediction with Machine Learning

Yajing Zhao
Brigham Young University

Chaotic Model Prediction with Machine Learning

Yajing Zhao

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Jared Whitehead, Chair
Emily Evans
Tyler Jarvis

Department of Mathematics

Brigham Young University

ABSTRACT

Chaotic Model Prediction with Machine Learning

Yajing Zhao
Department of Mathematics, BYU
Master of Science

Chaos theory is a branch of modern mathematics concerning the non-linear dynamic systems that are highly sensitive to their initial states. It has extensive real-world applications, such as weather forecasting and stock market prediction. The Lorenz system, defined by three ordinary differential equations (ODEs), is one of the simplest and most popular chaotic models. Historically research has focused on understanding the Lorenz system's mathematical characteristics and dynamical evolution including the inherent chaotic features it possesses. In this thesis, we take a data-driven approach and propose the task of predicting future states of the chaotic system from limited observations. We explore two directions, answering two distinct fundamental questions of the system based on how informed we are about the underlying model. When we know the data is generated by the Lorenz System with unknown parameters, our task becomes parameter estimation (a white-box problem), or the "inverse" problem. When we know nothing about the underlying model (a black-box problem), our task becomes sequence prediction.

We propose two algorithms for the white-box problem: Markov-Chain-Monte-Carlo (MCMC) and a Multi-Layer-Perceptron (MLP). Specially, we propose to use the Metropolis-Hastings (MH) algorithm with an additional random walk to avoid the sampler being trapped into local energy wells. The MH algorithm achieves moderate success in predicting the ρ value from the data, but fails at the other two parameters. Our simple MLP model is able to attain high accuracy in terms of the l_2 distance between the prediction and ground truth for ρ as well, but also fails to converge satisfactorily for the remaining parameters. We use a Recurrent Neural Network (RNN) to tackle the black-box problem. We implement and experiment with several RNN architectures including Elman RNN, LSTM, and GRU and demonstrate the relative strengths and weaknesses of each of these methods. Our results demonstrate the promising role of machine learning and modern statistical data science methods in the study of chaotic dynamic systems. The code for all of our experiments can be found on `https://github.com/Yajing-Zhao/`

Keywords: Lorenz system, chaotic model, machine learning, recurrent neural networks, multi-layer perceptron, Bayesian inference

Acknowledgements

First of all, I would like to express my sincere appreciation to my advisor, Jared Whitehead. Without his help and patience, I could not realize the goal of my thesis as expected. He guided me to the topic of inverse problems, helped me to master some classical models, and he encouraged me to try some relatively new methods and applied all of them to a chaotic model.

I wish to show my gratitude to professors of the mathematics department at BYU. I learned so much useful math knowledge through their courses. Especially Dr. Evans, whose Math 510 course was my first applied math class. It really opened the door to applied math for me. She also gave me many meaningful suggestions for my class selection. I would like to thank Dr. Jarvis for his patient introduction of his research projects when I was getting a knowledge of different research directions. I would also like to thank Dr. Bakker for his great help and support for my class selection, guidance as a teaching assistant, explanation of graduation requirements etc. during my masters program.

Meanwhile, I am indebted to my parents for their selfless love, helping me know how to love others and be grateful. I thank my parents in law for their support for my study. I also thank my brother and his wife for their care for my parents.

Lastly but most importantly, I wish to express my deepest gratitude to my husband. I thank him for his love, help, and support for me. I also thank him for his effort for our family. I will love and belong to him forever.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

The word *Chaos* is commonly used to express a state of utter confusion like: “The blackout causes chaos in the entire city” [2]. The word originated from the ancient Greek word χάος, which means the void and emptiness before everything comes to exist [3], as written in the first line of the famous Greek comic playwright Aristophanes's book “Birds” [4]:

At the beginning there was only Chaos, Night, dark Erebus, and deep Tartarus. Earth, the air and heaven had no existence;

Chaos Theory [5, 6] is a promising research area in applied mathematics that explores chaotic systems that exhibit this confusing, nearly unpredictable nature. Chaos theory investigates mathematical models of non-linear dynamical systems whose current states are highly sensitive to their initial state. A small perturbation of the initial state can quickly lead to large deviations in later states. Chaos theory can be intuitively described by the well-known butterfly effect [7]: a flap of butterfly wings in Brazil can cause a tornado in Texas. This property of “sensitivity to initial conditions” was first discovered by Poincare during his study of the “n-body problem” in the nineteenth century [8, 9]. In 1963, American mathematician and meteorologist Edward Lorenz found that a small rounding error in the inputs (three digits after the colon) of a simplified computer simulation of weather would lead to dramatically different outputs. He summarized this in his work “Deterministic nonperiodic flow”: [10]: *Two states differing by imperceptible amounts may eventually evolve into two considerably different states ... If, then, there is any error whatever in observing the present state—and in any real system such errors seem inevitable—an acceptable prediction of an instantaneous state in the distant future may well be impossible.*

In [10], Lorenz derived a simple three-dimensional ODE which has since been called the Lorenz System. This model illustrates the chaotic behavior of many other deterministic, nonperiodic dynamical systems. The Lorenz System is defined by the temporal evolution of the variables x, y, and z as:

$$\begin{cases} \dfrac{dx}{dt} = \sigma(y - x) \\ \dfrac{dy}{dt} = x(\rho - z) - y \\ \dfrac{dz}{dt} = xy - \beta z \end{cases} \tag{1.1}$$

Lorenz derived this system as a simplified model of a two-dimensional fluid layer which is warmed from below and cooled from above [11]. x, y, and z are proportional to the rate of convection, the horizontal temperature variation and the vertical temperature variation respectively. σ, ρ, and β are three system parameters controlling the system behavior and they are respectively proportional to the Prandt number, Rayleigh number, and certain physical dimensions of the two-dimensional fluid layer. In [10], the three canonical system parameters are set as: $\sigma = 10, \rho = 28, \beta = 8/3$. An ODE solver can be used to propagate the solution from the initial states and such a solver implemented in Python can be found in Appendix A. A Lorenz system can be visualized by its solution trajectories in the three-dimensional phase space. We visualize the Lorenz System using the canonical parameters given above in Figure 1.1. The butterfly-shaped attractor (the set to which a dynamical system evolves after a long enough time) [9] exhibits non-linear and non-periodic behavior — it doesn't repeat itself. To demonstrate the chaotic property of the Lorenz system, we consider the following case where two initial states that are only 0.001 away from each other. Their states diverge by 30 after only about 25 steps (Figure 1.1).

Previous studies [12, 13, 14] of the Lorenz system have mainly focused on its mathematical characteristics and its sensitivity to different parameters. A full review of the mathematical properties of the Lorenz System is beyond the scope of this thesis and readers are referred to Sparrow's book [13] for more details. In this work, we are interested in predicting the future states of the Lorenz System with limited observations. We formulate this task in the context of a data driven problem by making two distinct assumptions about the underlying model. We first assume that the underlying model is the Lorenz System with unknown parameters – a typical white-box problem or the "inverse" problem. We refer to this as the

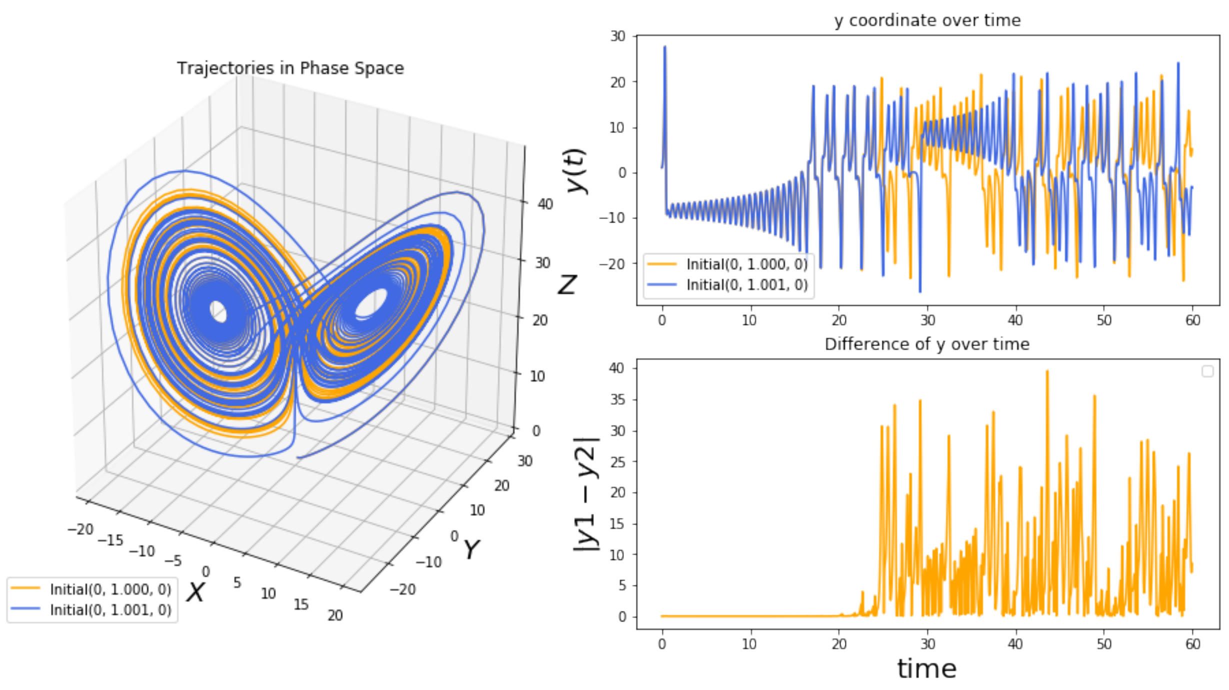


Figure 1.1: On the left are trajectories of two Lorenz System solutions with slightly different initialization. On the right we show the values of y_1, y_2 and their absolute difference for the same solutions.

parameter estimation task below. Investigation of this problem for the Lorenz system is explored in [15, 16].

A distinct problem that we consider is to assume that the underlying model is unknown – a typical black-box problem. The target task becomes a sequence prediction task. The sequence prediction task for the Lorenz System has been explored before in [17], where multi-step predictions are achieved by multi-step networks implemented by multi-layer Perceptrons (MLPs). We consider two different approaches for the sequence prediction task: future states are predicted purely from observed states or from a combination of observed and previously-predicted states. The approach in [17] is an example for the first approach. We instead focus on the second approach as it better captures the relationship between current predictions and previous ones. The Recurrent Neural Network (RNN), where the current prediction is dependent on previous predictions, is a representative algorithm for sequence prediction. It has wide applications in music/voice generation, language understanding, and

text analysis *etc.* For instance, below we quote a short paragraph generated from an RNN model learned from a Shakespearean play:
"Alas, I think he shall be come approached and the day When little srain would be attain'd into being never fed, And who is but a chain and subjects of his death, I should not sleep."
In the experiment section, we compare three different flavors of RNN networks including Long-Short Term Memory Network (LSTM) [18], Gated Recurrent Unit (GRU) [19], and Elman RNN network [20]. We find that the former two models perform the best at the sequence prediction task.

We propose to use both the traditional Bayesian Inference approach and modern neural networks for the parameter estimation task. We employ Markov Chain Monte Carlo (MCMC) [21] to estimate the probability distribution of the parameters from selected observables. MCMC works by randomly sampling the parameters guided by their probability determined by the difference between the predicted and the observed trajectory. As the vanilla Metropolis Hasting random walk (MH) [22] algorithm struggles to escape local energy wells in a multi-well distribution, we propose a simple yet effective variant called MH with random jumps [23] to help avoid local minima in the posterior distribution. The output of MH with random jumps is a probability distribution for each parameter and we take the value with highest probability (maximum a posterior estimate) as the final estimate

We also propose to use neural networks to learn to the ground truth parameter. Towards this goal, we train an MLP model and test it against a reserved testing set of data. We discuss how to collect these datasets in the experiment section of this thesis. We experiment with different hyper-parameters for the MLP model such as the number of layers, number of nodes, and the learning rate, *etc.* Our final model architecture is reported in Section 4.2.

The remainder of this thesis is organized as follows: Chapter 2 covers the basics of the MCMC algorithm and the MLP model; Chapter 3 includes an introduction to the sequential prediciton task and recurrent neural networks, and the final results including a description of the model parameters are detailed in Chapter 4.

Chapter 2. Parameter Estimation

This chapter explains the prediction task for estimating parameters of the Lorenz System from observed states. This is a type of "inverse" problem for the Lorenz system and has no analytical solution, and is likely ill-posed depending on the choice of observables. We take a data-driven approach and contrast two distinct approaches to this inverse problem: the Bayesian motivated MCMC algorithm, and the neural-network based algorithm. One of the key differences between these two algorithms is that MCMC works directly with the observed states but neural networks need to learn from a collection of states-parameter pairs. In other words, MCMC is an unsupervised learning algorithm while the neural-network algorithm is supervised. As we see below however, the implementation of MCMC that we use is significantly more computationally expensive than the neural network approach.

2.1 Parameter Estimation using Neural Networks

Artificial Neural Networks (ANN) – or more commonly referred to as Neural Networks (NN) – refer to a class of algorithms that mimic the layered structure of human brains. NNs are applied to various machine learning tasks like classification and regression. The building block of a Neural Network is a computational unit called a perceptron, which takes an input vector $\vec{x}$ of n dimensions and outputs a scalar y according to the following rule:

$$y = \sigma(\vec{w}\vec{x} + b), \tag{2.1}$$

where $\vec{w}$ is a weight vector of dimension n and b is called the bias. σ is a nonlinear function with a smooth, albeit sharp transition. A commonly used choice of σ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

The sigmoid function is bounded in the interval (0,1) and monotonically increasing. The primary benefit of using a nonlinear function such as the sigmoid into the neural network is that the composition of nonlinear functions yield more complex input-output relations. Another commonly used nonlinear function is the ReLU function. It is a piece-wise linear function whose output equals 0 when the input is negative and equals the input otherwise:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{otherwise.} \end{cases} \tag{2.3}$$

Both sigmoid and ReLU functions are differentiable everywhere given that we define the derivative of ReLU at $x = 0$ as 0.

One fundamental concept of machine learning is the different role of the training set and the testing set. The model learns from the training set and is validated against comparisons with the testing set. Both sets should be disjoint (no duplicates between them) and be sampled from the same distribution. The training set is used to update and identify the model's internal parameters or other hyper-parameters. The model will then compare its predictions against a separate never-seen-before testing set. The goal is to make sure the learned model generalizes well to unseen data instead of memorizing facts from the training set.

The perceptron model parameters are updated via backpropagation which boils down to backward differentiation of the components of the full model. The model output is compared against the ground truth labels and a loss value is computed. Different loss functions can be used for different problems: the cross entropy loss is common for classification and the Euclidean loss is common for regression. The perceptron model is composed of fully differentiable functions so that the partial derivative of the loss function with respect to each parameter can be calculated using the chain rule. A gradient descent step is performed to update the model parameters such that the future prediction is closer to the ground truth labels. Theoretically, gradient descent is only accurate when all the data samples are fed

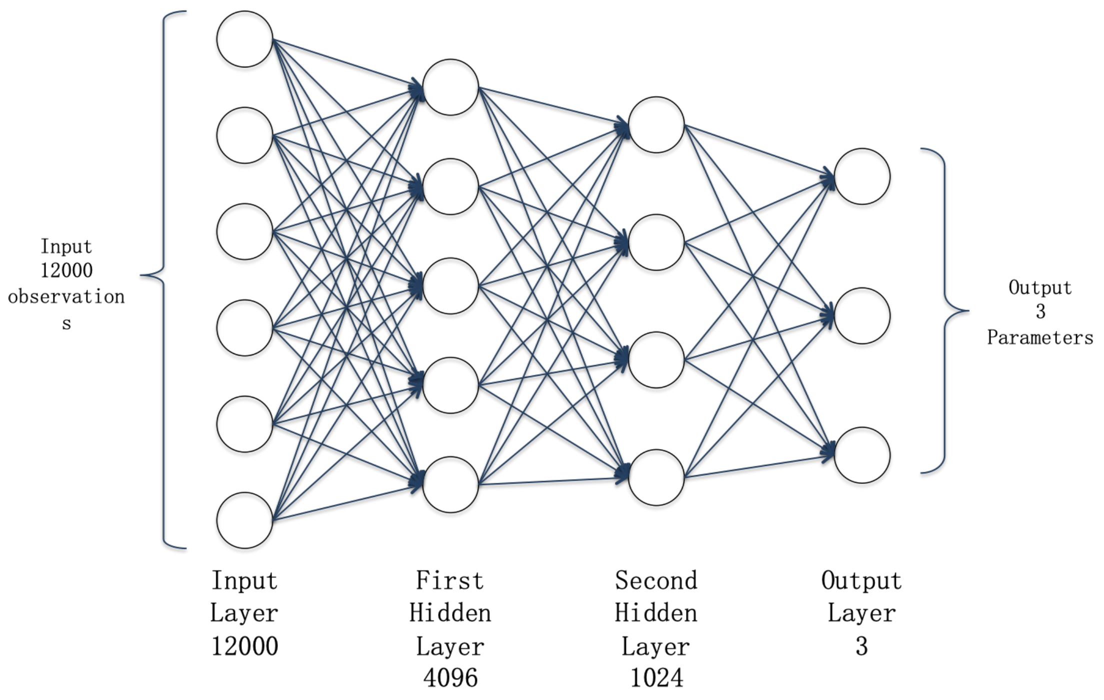


Figure 2.1: An illustration of the architecture of an MLP with two hidden layers.

into the model in a single batch, but this leads to severe memory requirements, and has the danger of memorizing the structure of the data with resultant poor generalization. Stochastic gradient descent, where the gradient is computed from a smaller batch of random data samples yields more satisfactory performance in practice.

The Multi-layer Perceptron (MLP) is a natural extension of the perceptron model. The MLP model is composed of stacked layers and each layer contains multiple perceptrons. An illustration is shown in Figure 2.1. The MLP model is a simple yet effective neural network architecture for both classification and regression problems. The simplest MLP model contains three layers, an input layer (which handles inputs), a hidden layer (which contains meaningful features) and an output layer (which produces outputs). It has been proven that this model can approximate any continuous function up to a specified accuracy.

One common issue with neural networks is the so-called internal variance shift, where the variance of the layer output is constantly shifting such that the subsequent layer parameters need to adjust frequently to adapt to these shifts. Batch normalization is proposed to deal with internal variance shift which keeps a running average and variance of the layer output and normalizes it in advance. We use batch normalization in our MLP network, which is composed of Linear-Batchnorm-ReLU modules.

2.2 Parameter Estimation using Bayesian Inference

One brute force approach for the "inverse" Lorenz System problem would be randomly sampling system parameters and comparing their solution trajectories with the observations. However, this approach wastes too much computation on parameters that are far away from ground truth. A much better way is to leverage Bayes' Rule to sample parameters based on their probability being near the ground truth under a specified metric. We provide a brief review of the MCMC algorithm and some implementation details in the following sections.

2.2.1 Bayesian Inference and Frequentist. Life is full of uncertainty. We use probability to quantitatively measure how likely an event is going to happen. Probability theory is a branch of modern mathematics that plays an important role in our everyday decision-making process. Historically, there are primarily two schools of thought in probability – the Bayesian and the Frequentist – based on their different interpretations of probability. Frequentists think of probabilities as the long-term frequency of a random event in a well-defined repeated experiment. Under this interpretation, it is difficult to associate a probability with a rare event such as the explosion of a nuclear power plant. Bayesians, on the other hand, consider probabilities as the likelihood that an event may happen, without relying on repeated experiments. Frequentist methods still dominate some applications of probability, but Bayesian methods show great vitality in fields like machine learning with recent advancements in computer technology and the appearance of big data. In this work, we apply Bayesian inference methods to predict the system parameters of the Lorenz system given specific observables, to generate a posterior probability distribution for those parameters.

2.2.2 Bayes' formula.

Bayes' rule 2.4 provides a handy tool to compute the probability of the event A conditioned on the event B as:

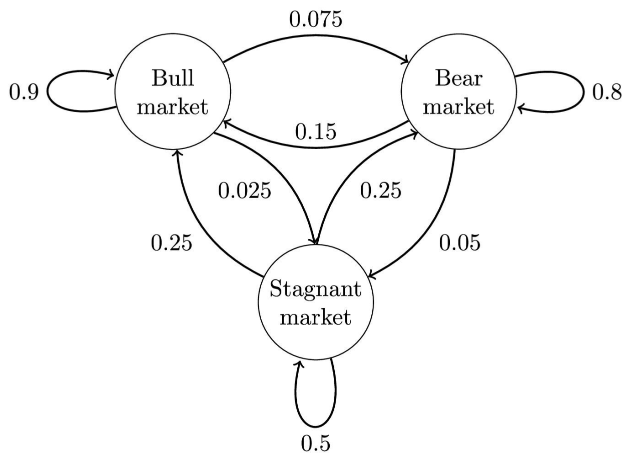
$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{2.4}$$

To see this in the context of a data science problem, we typically have observed some data D and would like to know how likely the observation is generated from the hypothesized model H. We represent this relation as the posterior probability $P(H|D)$ and it is computed by Bayes' rule as:

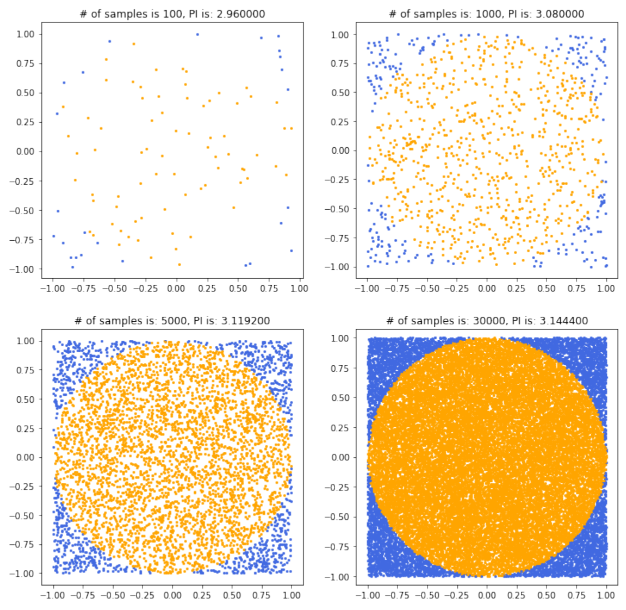
$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \tag{2.5}$$

where $P(H)$ is the prior probability of the hypothesized model, i.e. regardless of all other variables how likely is this model to be realistic? We often assume some prior knowledge of the probability distribution of all the possible hypotheses, or we simply set it to be a uniform distribution where all hypotheses are equally likely. $P(D|H)$ is the likelihood probability representing how likely the observed data is generated by the hypothesis H. $P(D)$ is the probability of the data D, which is often treated as a normalization constant since we typically are interested in ratios of the posterior probability and not the actual value of the probability itself. $P(H|D)$ is the posterior probability representing the probability that the hypothesis H is the model behind the observed data D.

Usually we're interested in the posterior probability but it is not directly observed. It can, however, be conveniently computed from the likelihood and prior as shown above. In our Lorenz System prediction task, the data D represents the observed states and the hypothesis H refers to the unknown system parameters. It is easy to define the likelihood probability of $P(D|H)$ being proportional to the similarity of the solutions from the hypothesis parameters H and the observed values. The final prediction for system parameters is then the hypothesis H with maximal posterior probability $P(H|D)$.



(a) Markov Chain state transition [24]



(b) Monte Carlo simulation of π

Figure 2.2: Examples of Markov Chains and the Monte Carlo algorithm.

2.2.3 MCMC. Markov Chain Monte Carlo (MCMC) is a method for sampling from a probability distribution using a Markov chain that is identified via a Monte Carlo simulation. We briefly these two key concepts separately and give a brief overview of the Matropolis-Hasting algorithm [25]of MCMC in this section.

A Markov chain is a stochastic model describing a sequence of events where the probability of each event depends only on the state attained in the previous event. A discrete-time Markov chain is a sequence of random variables $X_1, X_2, X_3, ...$ with the Markov property – the probability of moving to the next state depends only on the present state and not on the previous states:

$$P(X_{n+1} = x | X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = P(X_{n+1} = x | X_n = x_n) \tag{2.6}$$

As an example of an explicit Markov Chain modeling the economy, consider the diagram in Figure 2.2a. This model has 3 distinct states of a Bull Market, a Bear Market, and a Stagnant Market. The numbers and arrows represent the probability of transitioning from one state to another. For example, the probability of transitioning from a Bull Market to a Bear market is 0.075. We use the transition matrix P to represent the transition probabilities in a compact form:

$$P = \begin{bmatrix} 0.9 & 0.075 & 0.025 \\ 0.15 & 0.8 & 0.05 \\ 0.25 & 0.25 & 0.5 \end{bmatrix}. \tag{2.7}$$

Under the Markovian assumption, the future states of the economy are computed iteratively by multiplying the transition matrix against the current state's probability vector. The stationary probability is obtained when $n \to \infty$. In the above example, the stationary probability of a Bull Market, a Bear Market or a Stagnant Market are 62.5%, 31.25% and 6.25% separately:

$$\lim_{N\to\infty} P^N = \begin{bmatrix} 0.625 & 0.3125 & 0.0625 \\ 0.625 & 0.3125 & 0.0625 \\ 0.625 & 0.3125 & 0.0625 \end{bmatrix}. \tag{2.8}$$

Monte Carlo simulations refer to methods that involve a certain class of computational experiments. It uses repeated random sampling to get numerical estimation of the desired value. It can be applied to a vast range of problems and can be easily implemented. As a simple example, we demonstrate the utility of the Monte Carlo algorithm to estimate the value of π. We create a square with side length $2a$ and a circle with diameter a inscribed inside the square. The area of the square and the circle are $4a^2$ and πa^2 respectively. In the Monte Carlo experiment, points with random coordinates (x, y) are generated inside the square. The ratio between the number of points that falls inside the circle and those in the square can be computed as:

$$r = \frac{\#\text{ of points within the circle}}{\#\text{ of points within the square}} = \frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4}. \tag{2.9}$$

π can then be obtained by multiplying 4 times this ratio r. We conduct this experiment with several different numbers of random points and plot the approximated value of π in Figure 2.2b.

2.2.4 Metropolis-Hastings Algorithm.

The Metropolis-Hastings (MH) and the Gibbs Sampling Algorithm are two of the most popular implementations for MCMC. We focus on the MH algorithm because of its simplicity and also because Gibbs Sampling can be interpreted as a special case of MH.

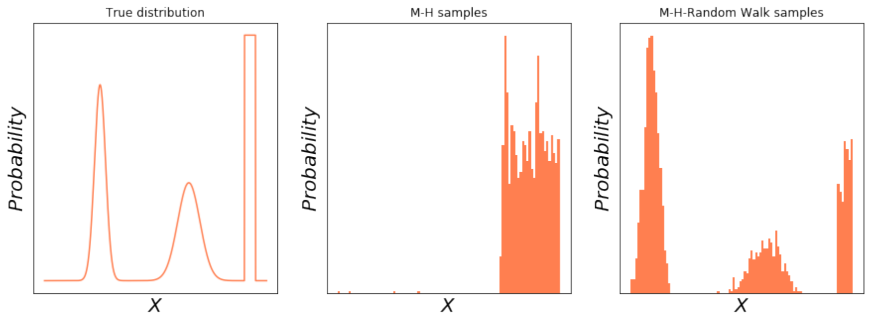
Algorithm 1: Metropolis-Hastings algorithm

```
Initialize $x^0 \sim q(x)$;
$\gamma_0 = 1$;
for iteration i=1, 2,... do
    $\gamma = \gamma_0$ ;
    $w \sim N(0, \gamma)$;
    Propose: $x^{cand} = x^{i-1} + w$;
    Acceptance Probability: $\alpha(x^{cand}|x^{i-1}) = \min(1, \frac{pr(x^{cand})li(x^{cand})}{pr(x^{i-1})li(x^{i-1})})$ ;
    $u \sim U(0,1)$ ;
    if $u < \alpha$ then
        Accept the proposal: $x^i \leftarrow x^{cand}$;
    else
        Reject the proposal: $x^i \leftarrow x^{i-1}$;
    end
end
```

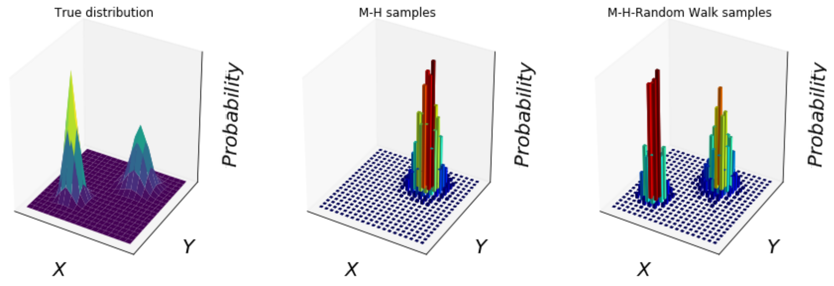
Algorithm 1 shows the pseudo code for the standard MH random walk algorithm. In every iteration, a new data point is generated by adding the old data point to a sample from a Gaussian distribution (hence the term ‘random walk’) and an acceptance probability is calculated, which decides whether to keep or discard this data point. The acceptance probability $\alpha(x^{cand}|x^{i-1})$ is computed from Bayes' Rule (Equation 2.4). For simplicity, we set the prior probability of all three system parameters as a uniform distribution which forces the learned posterior distribution to depend purely on the observations. The likelihood probability is designed to reflect the similarity between the solution trajectories of the sampled and the ground-truth data.

In order to compute the likelihood probability, a distance metric needs to be defined to capture the similarity between two trajectories. One intuitive way is to compare the trajectories one entry at a time and compute the averaged Euclidean loss between them. However, due to the chaotic nature of the Lorenz System, we observe that this distance metric produces extremely large values for two visually-similar trajectories. It is beneficial to imagine two trajectories of the same general shape, but are different at every time step except at the beginning. Small distances at each time step are accumulated into a very big distance over the entire simulation, and hence not faithfully reflecting the visual similarity between the two trajectories. The second metric which is one that we use below, is to compare the distance between the temporally averaged moments of the trajectories. We consider 1st order moments like averages of x, y, z, 2nd order moments like averages of $xy, x^2, y^2, z^2,$, *etc.* We compare the results of using different moments in Section 4.1.

One limitation of the random walk MH algorithm is that it is hard to escape local energy wells as Figure 2.3 shows. We propose to tackle this problem using MH with random jumps (Algorithm 2). The variance of the Gaussian distribution is set to a larger value ($20\gamma_0$) every n iterations, allowing the sampler to "jump to" values outside the current energy well. We visualize two probability distributions for one-dimensional and two-dimensional data, and show the estimated probability distribution using both algorithms in Figure 2.3.



(a)



(b)

Figure 2.3: Comparison of MH and MH with Random Jumps in 1D (a) and 2D (b).

Algorithm 2: Metropolis-Hastings with random jumps

```
Initialize $x^0 \sim q(x)$;
$\gamma_0 = 1$;
for iteration i=1, 2,... do
    $v \sim U(0,1)$;
    $\gamma = \gamma_0 \quad \text{if} \quad v > 0.1 \quad \text{else} \quad 20 * \gamma_0$ ;
    $w \sim N(0, \gamma)$;
    Propose: $x^{cand} = x^{i-1} + w$;
    Acceptance Probability: $\alpha(x^{cand}|x^{i-1}) = \min(1, \frac{pr(x^{cand})li(x^{cand})}{pr(x^{i-1})li(x^{i-1})})$ ;
    $u \sim U(0,1)$ ;
    if $u < \alpha$ then
        Accept the proposal: $x^i \leftarrow x^{cand}$;
    else
        Reject the proposal: $x^i \leftarrow x^{i-1}$;
    end
end
```

Chapter 3. Sequence Prediction

Sequence prediction refers to the task of predicting the next values given historical values. Popular examples include predicting the next character given previous text blocks and generating melodies one bar after another. Predicting the future states of the Lorenz System can be naturally seen as a sequence prediction task. Different from parameter estimation, the sequence prediction task does not make any assumption about the model behind the data, making it a typical black-box problem.

Recurrent Neural Networks (RNN) are the most popular models for sequence prediction. RNN is different from models like MLP in that it keeps an internal state which reflects the influence of past states on the current output. We give a brief overview of the RNN model and some of its popular variants in the following section.

3.1 Recurrent Neural Networks

One of the limitations of feed-forward networks like MLP is that it only works with fixed-length input. This requirement is easily satisfied for inputs like images, but not for sequential data with variable length. Moreover, feed-forward networks' output is only dependent on the current input, i.e. prior knowledge is not used. In many real world applications however, the current output is related to both current and previous inputs. Recurrent Neural Networks (RNN) overcome these limitations, making it a natural choice for sequence prediction.

3.2 RNN Variants

This section includes three common variants of the standard RNN: Elman RNN, Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU). Figure 3.1 provides visualizations of the network structure for each of these variants.

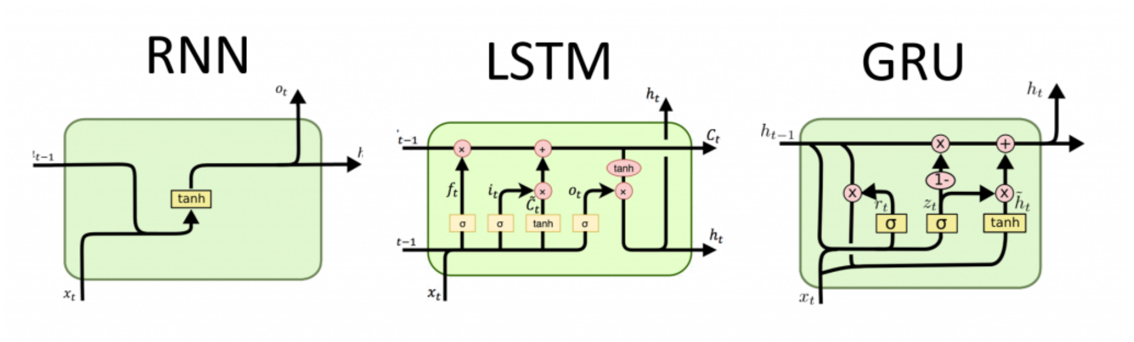


Figure 3.1: Architectures of Elman-RNN, LSTM and GRU. [1]

The simplest RNN is called an Elman Network, which is a three-layer neural network similar to the MLP. The only difference is that the hidden layer now takes both the current input and the previous output. An additional weight vector is applied to the previous output which is fed back into the hidden layer.

LSTM is especially good at modelling long-term dependency, where the current output is dependent on the input from a long time in the past. It does so by maintaining a cell state throughout the network. For each input, a forget gate is applied to discard part of the past information, an input gate is applied to store new information into the cell state and an output gate is applied to predict the current output. GRU is a recently proposed network meant to replace LSTM. It performs comparably with LSTM but is less computational expensive. GRU only has two gates: an update gate that chooses to discard and store certain information and a reset gate that resets the hidden state.

Different algorithms work better with different training data: the MH random walk algorithm with random jumps is unsupervised and thus needs no additional data to train with. The MLP and RNN algorithms, on the other hand, require a dataset to learn/train from. To form the training and testing data sets, we collected 10,000 different simulations of the Lorenz System. To generate each simulation we randomly sample the system parameters σ, ρ, β from the following uniform distributions: $U(0, 20)$, $U(10, 30)$, $U(0, 10)$, respectively. A Python ODE solver is used to propagate solutions for the sampled parameters and the initial state. These simulations are evenly split into the training and testing set, i.e. 5,000 simulations for each. In the sequence prediction task, the RNN model and its variants are trained on the first part of the solutions and is evaluated by its performance on the second part, which is unseen during the training.

4.1 Parameter Estimation using MCMC

We denote the ground truth system parameters as $\sigma_{gt}, \rho_{gt}, \beta_{gt}$, and solutions to the Lorenz system (Equation 1.1) with these parameters are denoted as (x_{gt}, y_{gt}, z_{gt}). The random walk Metropolis-Hasting algorithm with random jumps works by drawing samples in the possible parameter space. In iteration n, a random set of system parameters $(\sigma_n, \rho_n, \beta_n)$ is sampled from the Gaussian distribution centered at $(\sigma_{n-1}, \rho_{n-1}, \beta_{n-1})$ with unit variance (the chosen random walk size). The Lorenz System solution is computed as: (x_n, y_n, z_n). The acceptance ratio of $\sigma_n\rho_n\beta_n$ is then calculated as α:

$$\alpha = \frac{pr(\sigma_n, \rho_n, \beta_n) li(x_{gt}, y_{gt}, z_{gt} | \sigma_n, \rho_n, \beta_n)}{pr(\sigma_{n-1}, \rho_{n-1}, \beta_{n-1}) li(x_{gt}, y_{gt}, z_{gt} | \sigma_{n-1}, \rho_{n-1}, \beta_{n-1})}, \tag{4.1}$$

where pr is the prior and li is the likelihood probability. We assume the prior distributions on σ_n, ρ_n, and β_n are mutually independent, and satisfy a uniform distribution with range

(0, 30), (0, 40), (0, 10) respectively. Using this uniform prior probability means it can be eliminated from the acceptance ratio, i.e. the prior does not influence the acceptance of one sample over another. Hence the MAP (maximum a prior) estimation actually becomes the maximum likelihood estimation (MLE) for this choice of prior.

The likelihood $li(x_{gt}, y_{gt}, z_{gt}|\sigma_n, \rho_n, \beta_n)$ represents the probability that the ground truth solution can be obtained by the sampled parameters. In other words, the likelihood probability is proportional to the similarity between x_{gt}, y_{gt}, z_{gt} and (x_n, y_n, z_n) which is the solution of the Lorenz System for parameters σ_n, ρ_n, and β_n. As mentioned earlier, one intuitive metric is to calculate the mean squared error loss between the two sets of solutions. However, the Euclidean distance between two visually-similar phase space trajectories can be extremely large when tiny errors at each time step are accumulated. Moreover, the Euclidean distance whose value ranges from $(0, \infty)$ can not be treated as a probability without normalization. This is not surprising, as the chaotic nature of the Lorenz System makes it highly unlikely that point to point trajectories will compare favorably even though the specific parameters are very nearly the same.

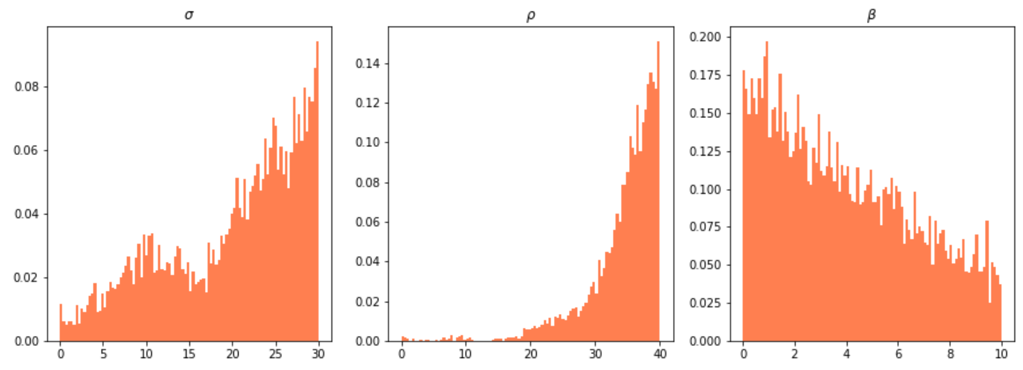
Rather than considering the Euclidean distance between two different solution trajectories, we propose to diagnose the similarity between the ground truth and the prediction based on the time averaged moments of the solutions. For instance, we designate the likelihood as a Gaussian distribution, whose mean is $\overline{x_{gt}}, \overline{y_{gt}}, \overline{z_{gt}}$ and variance is unity (or some other adjustable value). We further assume that these likelihoods over x, y, z are mutually independent and hence the joint likelihood is the product of all three. Instead of computing the probability for each individual solution at a given point in time which will very likely diverge substantially via the inherent chaos of the system, we treat the solution as a set and are only interested in its time averaged moment statistics. For example, we measure 1st order moment statistics for each sample as variations about $\overline{x_{gt}}, \overline{y_{gt}}, \overline{z_{gt}}$, where the overline denotes a time average over the entire simulation time. Considering that only the 1st order moment information may not be enough, we also consider 2nd order moments which are defined by $\overline{x_{gt}^2}, \overline{x_{gt}y_{gt}}$, etc., and the 3rd order moments like $\overline{x_{gt}^3}, \overline{x_{gt}y_{gt}z_{gt}}$, etc. However, as shown in figure 4.1, even with these higher order moments a unitary variance on the likelihood distribution will not sufficiently restrict the predicted values to give any practical estimates.

One limitation of the above method is that the variance is fixed at an arbitrary value with no reason for this choice. Alternatively we set the variance as a fraction of the mean value of the ground truth. We experimented with a proportional variance of $mean/4$, and the result is shown in Figure 4.2. It can be seen that this modification fails again for 1st and 2nd order moments, but it works for the estimation of ρ using 3rd order moments. Using a variance of $mean/10$. As shown in Figure 4.3, this setting works well for the estimation of ρ on both the 2nd and 3rd order moments. The posterior distribution for ρ is roughly Gaussian-shaped with ground truth being at the center. However, the prediction of σ and β resemble a uniform distribution, indicating that nothing predictive can be stated for these two parameters.

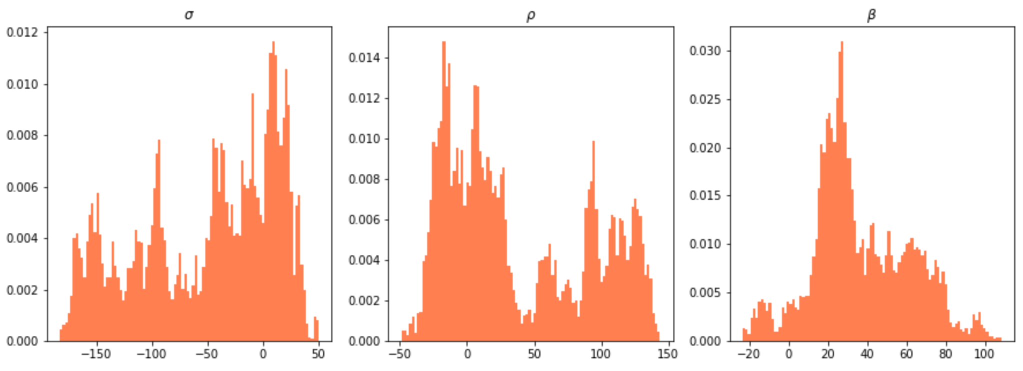
To summarize the results using MCMC to infer system parameters, inference of anything requires at least the second-order moments (possibly third-order moments) to obtain reliable accuracy. As the variance is an estimate of the error in the observations, a variance of mean/10 is a reasonable consideration. One possible reason why the MCMC failed to identify σ and β is that ρ is the most important factor to determine the appearance of the Lorenz system, i.e. ρ represents the driving force of the system whereas the other two are material parameters of the modeled fluid and a geometric constraint on the domain respectively. It makes sense then that ρ will primarily determine the time-averaged moments of the solution set.

4.2 Parameter Estimation using MLP

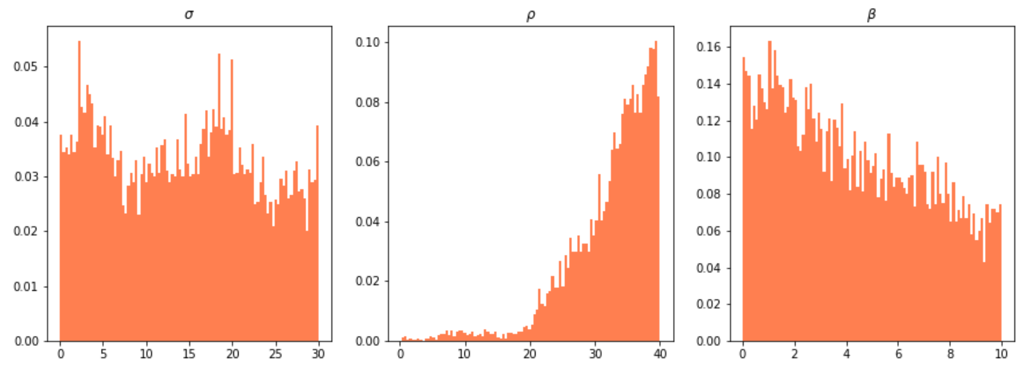
The MLP model we select is composed of three layers: an input layer, a hidden layer and an output layer. The input layer takes in 4,000 time-stamp values of the Lorenz system. As



(a) 1D moments

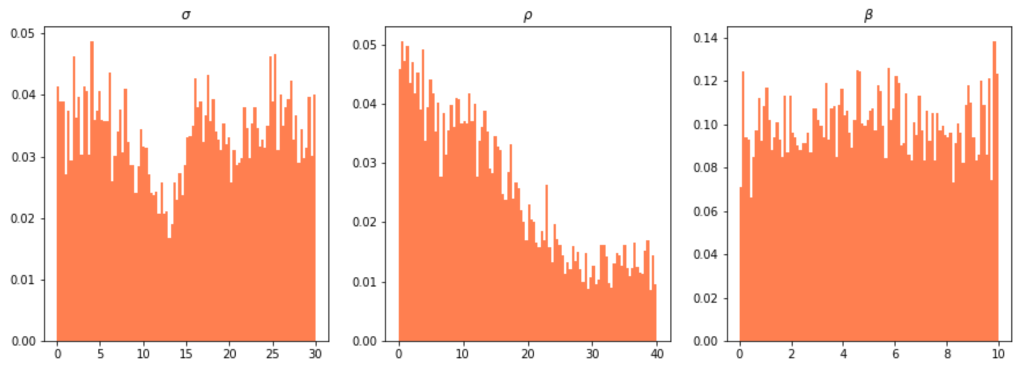


(b) 2D moments

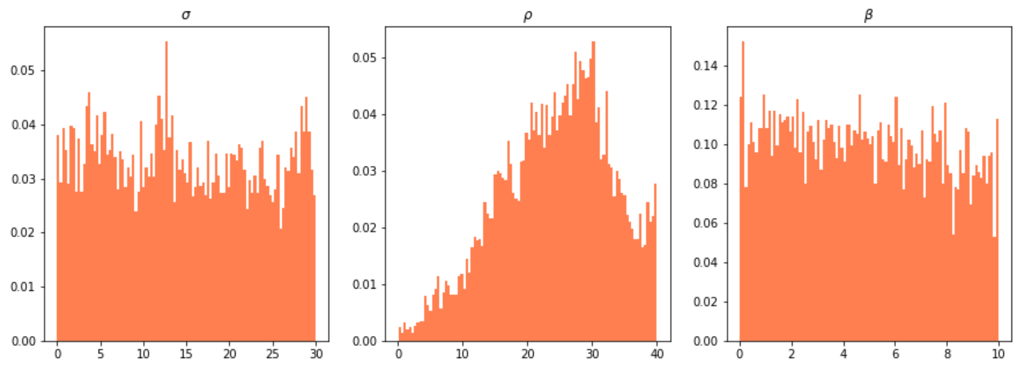


(c) 3D moments

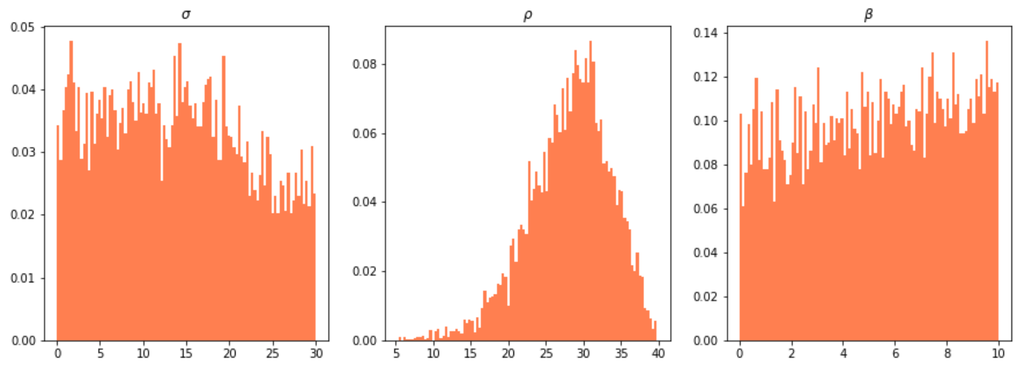
Figure 4.1: MCMC with unit variance



(a) 1D moments

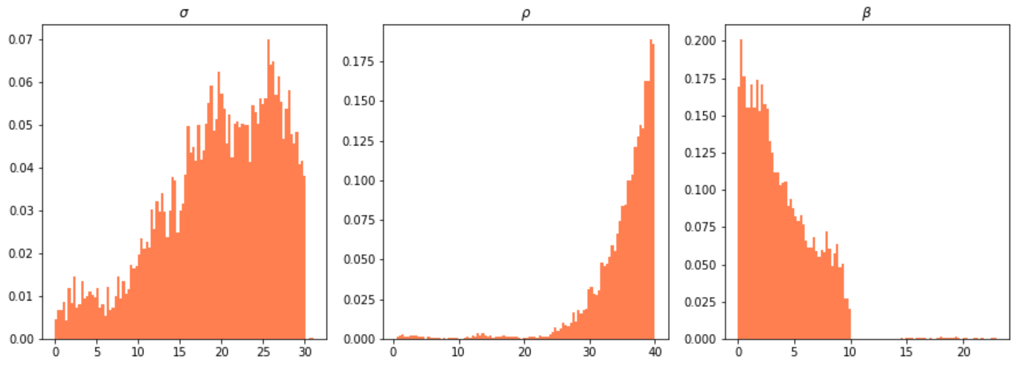


(b) 2D moments

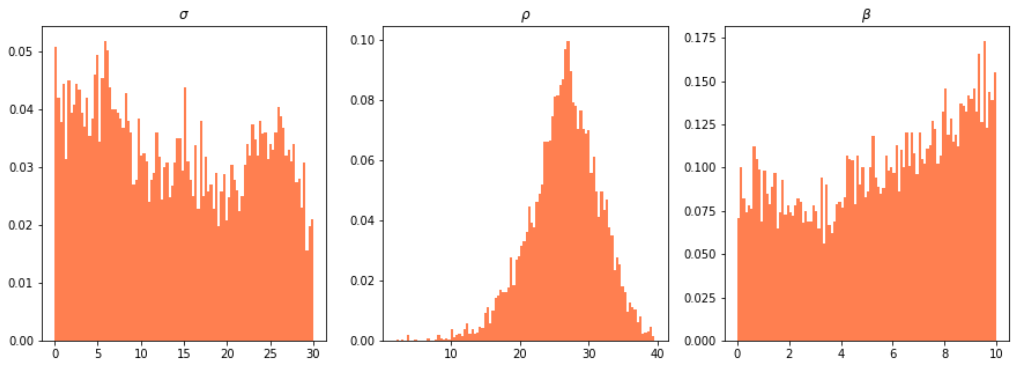


(c) 3D moments

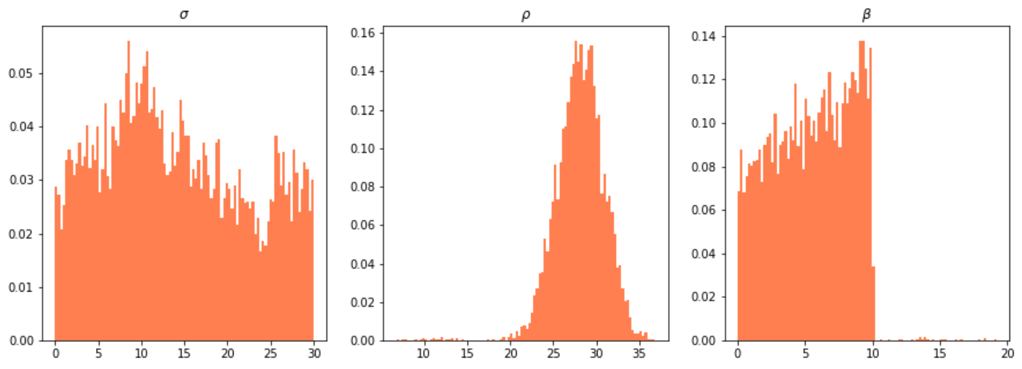
Figure 4.2: MCMC with variance of $mean/4$



(a) 1D moments



(b) 2D moments



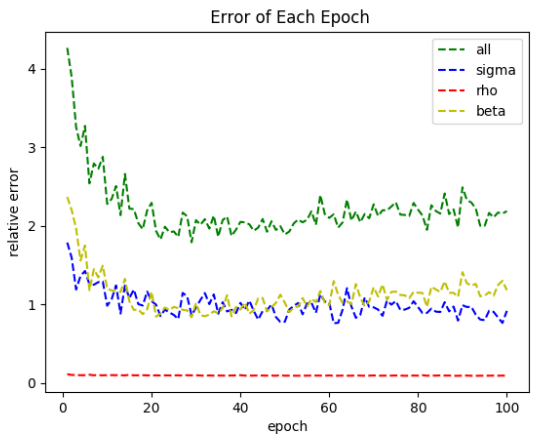
(c) 3D moments

Figure 4.3: MCMC with variance of $mean/10$

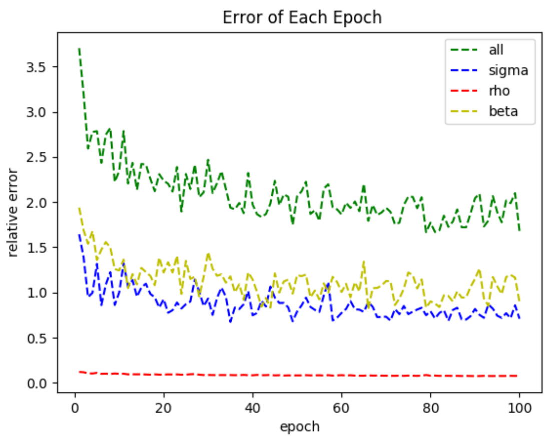
each solution is represented by three coordinates (x, y, z), the input layer contains $4,000 \times 3 = 12,000$ nodes. The hidden layer has 1024 nodes and the output layer has 3 nodes, corresponding to the system parameters σ, ρ, β. The input goes through a linear layer, a ReLU layer and another linear layer separately to get the output. The loss is the mean squared error between the output and ground truth system parameters. Stochastic gradient descent is applied with a batch size of 8. We use 5000 samples (sets of simulations) for both the training set and the testing set, and we use the mean value of the errors of 5000 samples as our absolute error. The model is trained for 100 epochs with a fixed learning rate of 0.001. The selection of these specific hyper-parameters and the architecture of the network influence the validity of the resultant estimates. In the following we discuss how variations in these settings can affect the parameter estimation.

Learning Rate Learning rate is usually the most important hyper-parameter in deep neural networks. The choice of an appropriate learning rate is often a trial-and-error process. Large learning rates will cause the loss to oscillate and never converge. It may also result in the explosion of the gradient and a complete training failure. On the other hand, too small of a learning rate will cause extremely slow convergence. In addition a very small learning rate may also lead to memorization and poor generalization. We find that 0.001 is an appropriate balance between training speed and accuracy. Another common strategy is to decay the learning rate when the training loss stops decreasing. For the limited investigations we performed for this problem, such a dynamic choice of learning rate did not yield any substantial benefits, so we have focused on a fixed learning rate instead.

Network Architecture The network architecture entails selecting the number of layers, the number of nodes per layer and the method of combining the different layers. Our network has three layers: input, hidden and output. The three-layer MLP has been proven to be able to approximate any continuous function with any given accuracy. We have also tested a deeper network with more layers, but it achieved only limited improvement. The hidden layer has 1024 nodes, which is found to be most effective while keeping a small memory



(a) Sigmoid non-linear function



(b) ReLU non-linear function

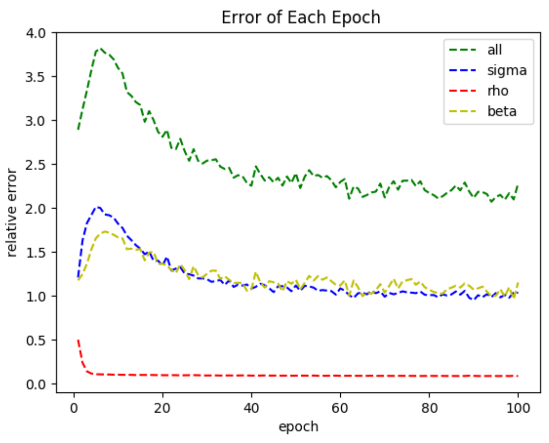
Figure 4.4: Comparison of results using either the sigmoid or the ReLU transition function.

fingerprint. There are two choices with respect to the non-linear layers as discussed above: ReLU and sigmoid. We contrast the performance of these two choices in Figure 4.4. Both non-linear functions perform comparably in our experiment.

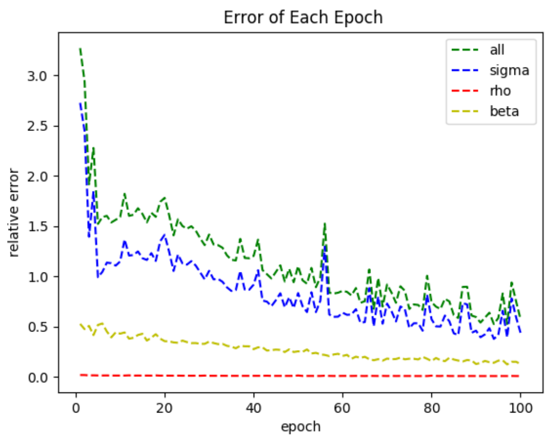
Another important design choice is the batch normalization layer. Typically the batch normalization layer is effective at reducing the variance shift between the linear layers. For this problem, we find that using a batch normalization negatively affects the training loss (Figure 4.5). We hypothesize that the normalization process of the batch normalization layer is harmful to non-probabilistic data such as the deterministic output of the Lorenz System.

We use a mean squared error for the loss function first. The result in (Figure 4.6) shows that the estimation of ρ is satisfying, while our model fails to estimate both σ and β. To capture σ and β, we to increase the weights on these parameters relative to ρ in the loss function (still with a mean-squred error). Specifically we weight errors of σ, ρ, β by (1, 1/2, 1) respectively. Unfortunately, even with the weighted loss function (as well as several other not shown choices of the weights) the model still fails to estimate σ and β.

Error Normalization The absolute error of the three system parameters is computed as the absolute value of the difference between the prediction and the ground truth. It reveals how close the prediction is to the ground truth. However, the typical values of the three

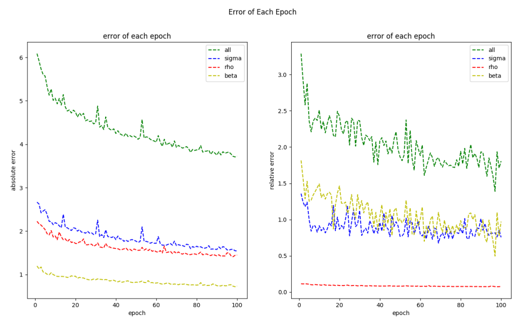


(a) With batch normalization

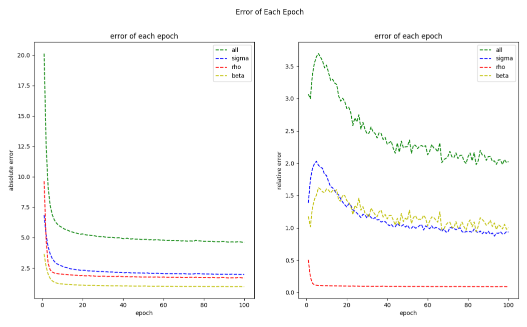


(b) Without batch normalization

Figure 4.5: Comparison of results for networks with and without batch normalization



(a) MSE



(b) MSE with Weights(1/2, 1, 1.2)

Figure 4.6: Comparison of mean squared error (MSE) loss function and MSE with weights to focus on σ and β.

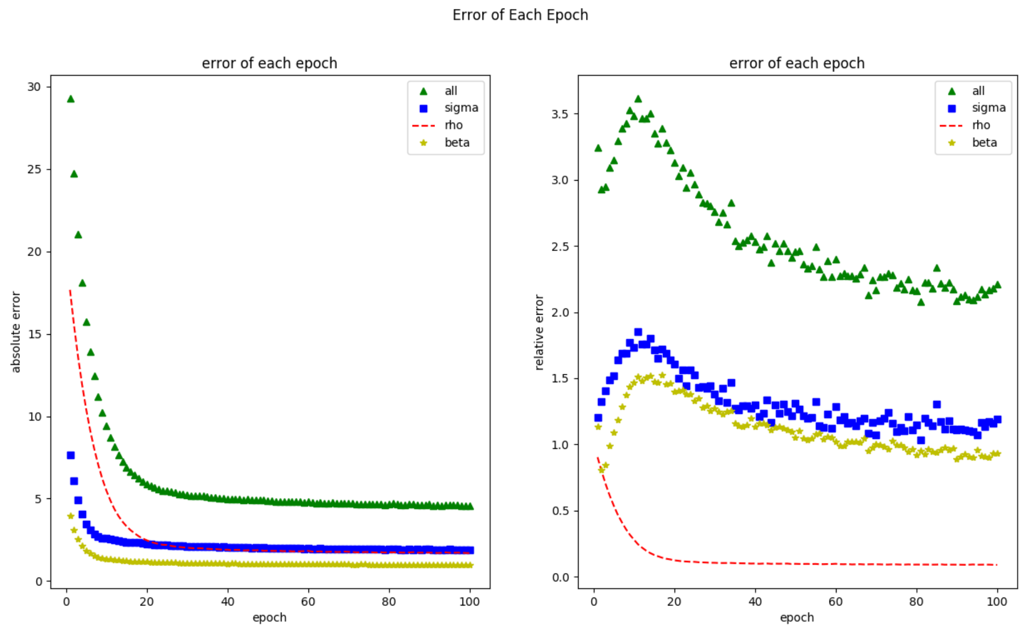
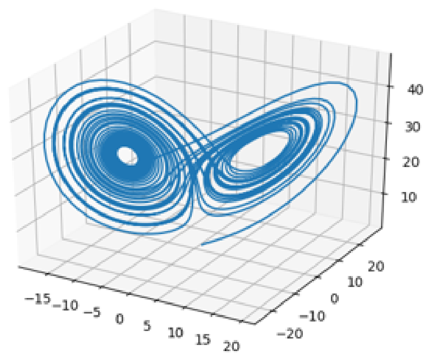


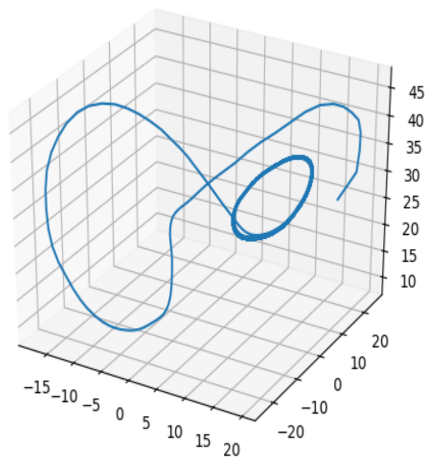
Figure 4.7: MLP testing Results

parameters vary greatly so that an absolute error of 10 for ρ is vastly different than the same absolute error for β. We therefore propose to measure the error using relative error, the absolute error divided by the ground truth value to better reflect the prediction accuracy. Here the ground truth is the mean value of the the targets of 5000 samples.

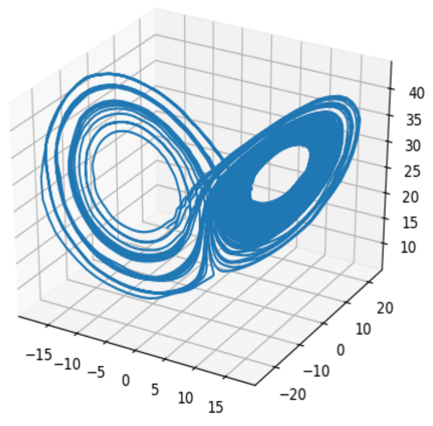
We visualize the absolute error of the three parameters in the testing set after each epoch in the left part of Figure 4.7. The absolute error of all parameters converges quickly (in terms of epochs) to values between 0 and 5. This indicates the effectiveness of the MLP algorithm. We further visualize the relative error in the right part of Figure 4.7. The relative error of ρ converges to a value which is very close to 0 and remains lower than the other two. The relative errors of σ and β remain greater than 1 after 100 epochs. At this point we remind the reader that the relative error is relative to the mean parameter values over 5000 different samples, and hence a relative error greater than 1 while not good, is not impossible to achieve. These results are similar to the MCMC investigation even though the selected observables and methods are fundamentally different, as ρ is predicted better than σ, and β.



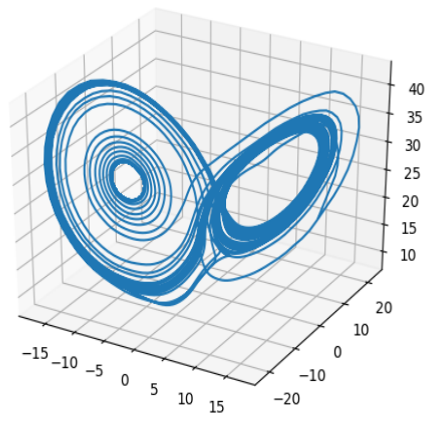
(a) ground truth trajectories with random noise of $U(0, 0.01)$



(b) Elman RNN results



(c) LSTM results



(d) GRU results

Figure 4.8: RNN sequential prediction results

4.3 Sequence Predicting using RNNs

At each time step, the RNN model takes the current state as input and outputs the next state. The previous states are fed back into the network together with the input. The loss is accumulated across all outputs to update the model parameters. For the results reported here, we have selected a batch size of 1 and learning rate of 0.01. The model is trained for 1000 iterations in total. We then use the mean squared error loss function to train the model. Each model contains an input layer, a hidden layer and an output layer. The hidden layer size is 128. The input and output layers have 3 nodes.

We study the performance of the RNN models in both qualitative and quantitative ways. To evaluate how different architectures perform in a qualitative manner, we visualize the predicted values for the networks of Elman, LSTM and GRU in Figure 4.8. The models are trained on the training set and then are given a random initialization state different from the training set. The ground truth trajectories are shown in Figure 4.8a and the predictions made by different RNN models are shown in Figure 4.8b, 4.8c, and 4.8d. The trajectories of LSTM and GRU are the most similar to the ground truth, which is not surprising as these two networks have been proven to do well at other standard machine learning tasks as well.

To evaluate the generalization of different RNN models quantitatively, it is necessary to run the models on a separate testing set. We generate 8,000 solution points using a standard Python ODE solver using the canonical parameters $\sigma = 10, \rho = 28$, and $\beta = 8/3$. We train the models on the first 4,000 time steps and test them on the second 4,000 time steps which have not been seen yet. The trajectories of the predicted states for the testing set are shown along with the trajectories of ground truth in Figure 4.9. The loss in both the training and testing sets are shown in Figure 4.9. The running time of each model is shown in Table 4.1, indicating that while the Elmann RNN is less effective at predicting the dynamical evolution, it is more computationally efficient.

In addition we note that while the GRU and LSTM networks appear to perform quantitatively at the same level and have the same relative computational cost, the phase space evolution of their predictive states are not equivalent. While the numerical differences between the predicted trajectory and the ground truth state are similar between the two network architectures, the qualitative behavior is noticeably different. For instance, the GRU network produces a relatively smooth evolution of the system that appears to fall on the attractor of the Lorenz System. The LSTM architecture produces trajectories that numerically stay as close (on average) to the ground truth solution, but the feedback mechanisms induce sharp transitions in the dynamical evolution that are not physically present in the ground truth (see Figure 4.9).

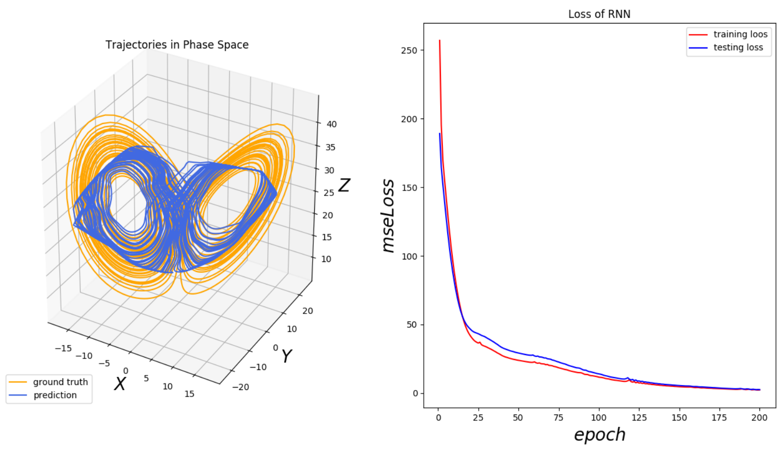
Elman RNN	GRU	LSTM
12m24s	23m11s	24m53s

Table 4.1: Running time comparison

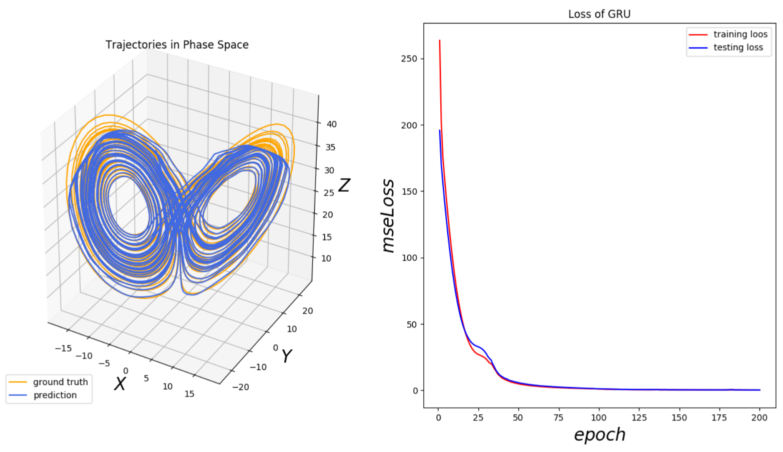
Chapter 5. Conclusions and Future Work

The results of the predictive portion of this manuscript that focus on the prediction of dynamical states from past and present values are clear. The success of all three types of RNNs indicate that these data-driven models can adequately capture the evolution of chaotic dynamical systems. Extension to more complicated models is of course of interest, but is beyond the scope of this thesis.

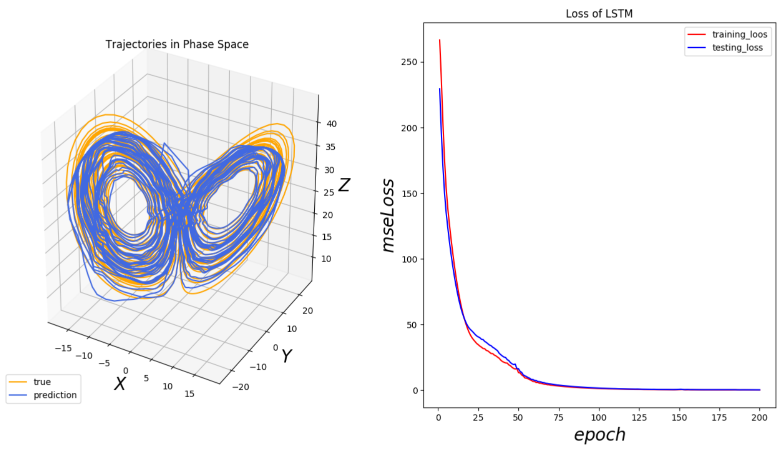
In this thesis, we have also focused on prediction of specific parameter values for the Lorenz System given observation of time-averaged moments of the solution (MCMC) and pointwise (in time) observations of the three different variables (MLP). Taking a data driven approach, we find that using either set of observations and either the Bayesian approach or via the neural network, we are able to estimate the true value of ρ, the driving force of the system, but are unable to provide accurate and reliable estimates on the other two parameters. This may be a limitation of the methods we have used (MCMC vs. MLP etc.), the particular choice of observations that we use to drive both methods, or may be



(a) Elman RNN



(b) GRU



(c) LSTM

Figure 4.9: RNN results for the Elman RNN, GRU, LSTM network structures.

an indication of the influence and physical relevance of each of the three parameters. In particular for the time averaged moments we have shown that it is apparently necessary to retain at least the 2nd order moments (if not the 3rd order moments) to accurately capture the effects of ρ. While these investigations are certainly not comprehensive, we do emphasize that our results indicate that at least for the two different types of observations used here, ρ is the most influential parameter of the Lorenz System.

Further extensions of the parameter estimation part of this thesis would include consideration of combining time-averaged moments with individual observations at specific time-steps to see if convergence to the true parameter is improved. It is also worth investigating if the critical dependence of the estimation on ρ is related to the fact that stability of the origin in the Lorenz system is dependent on ρ only, i.e. are the results presented here a reiteration of some type of structural stability result for dynamical systems? A further extension of these methods is to more complicated chaotic models. The Lorenz System was originally derived as a drastically simplified model of convection and weather prediction. Is it possible to apply these MCMC and MLP methods to more complicated weather models, or even the original partial differential equations (PDE) that describe convective processes? Can this be extended to more complicated models of the weather which inherently have a host of other parameters and tuning factors, whose impact on physically meaningful statistics is unknown?

For both the prediction of future states of the Lorenz System and for the estimation of parameter values from given observations, we have shown that modern methods of data science are applicable to chaotic systems. We emphasize that while the results indicated here are not definitive and do not necessarily yield fundamentally new insight into the dynamical evolution of the Lorenz System, they do indicate that such methods have a place in the study of dynamical systems and other physically motivated yet chaotic systems.

Bibliography

[1] Christopher Olah. Understanding lstm networks, August 27, 2015.

[2] Merriam-Webster Inc. The Merriam-Webster.com Dictionary, 2020.

[3] Wikipedia contributors. Chaos (cosmogony) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 6-March-2020].

[4] Aristophanes. *The birds*. 414 BC.

[5] STEPHEN H. KELLERT. *In the Wake of Chaos*. University of Chicago Press, 1993.

[6] Robert W Batterman. Defining chaos. *Philosophy of Science*, 60(1):43–66, 1993.

[7] Wikipedia contributors. Butterfly effect — Wikipedia, the free encyclopedia, 2020. [Online; accessed 6-March-2020].

[8] Gerhard Heinzmann and David Stump. Henri poincaré. 2013.

[9] Christian Oestreicher. A history of chaos theory. *Dialogues in clinical neuroscience*, 9(3):279, 2007.

[10] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[11] Wikipedia contributors. Lorenz system — Wikipedia, the free encyclopedia, 2020. [Online; accessed 6-March-2020].

[12] C. Sparrow. An introduction to the lorenz equations. *IEEE Transactions on Circuits and Systems*, 30(8):533–542, August 1983.

[13] C Sparrow. *The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors*. Springer, 1982.

[14] James Hateley. The lorenz system. *Lecture Notes, http://web. math. ucsb. edu/˜ jhateley/paper/lorenz. pdf,(Access date: 23.09. 2019)*.

[15] Walter JH Stortelder. Parameter estimation in dynamic systems. *Mathematics and Computers in Simulation*, 42(2-3):135–142, 1996.

[16] Klaus Schittkowski. Parameter estimation in dynamic systems. In *Progress in Optimization*, pages 183–204. Springer, 2000.

[17] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Multistep neural networks for data-driven discovery of nonlinear dynamical systems. *arXiv preprint arXiv:1801.01236*, 2018.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[20] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[21] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[22] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

[23] Chris Sherlock, Paul Fearnhead, Gareth O Roberts, et al. The random walk metropolis: linking theory and practice through a case study. *Statistical Science*, 25(2):172–190, 2010.

[24] Osho Jha. When to buy the dip.

[25] Jari P. Kaipio and Erkki Somersalo. *Statistical and Computational Inverse Problems*. Springer, 2005.