Mathematical Biosciences and Engineering

https://www.aimspress.com/journal/mbe

MBE, 22(12): 3130–3153.
DOI: 10.3934/mbe.2025116
Received: 26 March 2025
Revised: 15 September 2025
Accepted: 17 September 2025
Published: 03 November 2025

Research article

An accessible approach to density estimation neural networks with data preprocessing

Bosi Hou[1] **and Jonathan E. Rubin**[2,*]

[1] Data Science Institute, Columbia University, New York, NY 10027, USA
[2] Department of Mathematics, University of Pittsburgh, 301 Thackeray Hall, Pittsburgh, PA 15260, USA

* **Correspondence:** Email: jonrubin@pitt.edu.

Abstract: Density estimation neural networks (DENNs) represent a form of artificial neural network designed to provide an efficient approach to the Bayesian estimation of a probability density on a model parameter space, conditioned on an empirical observation of the underlying system. Despite their efficiency and potential, DENNs remain underutilized for parameter estimation in mathematical modeling. In this work, we aim to boost the accessibility of the DENN approach by providing a user-friendly introduction and code that makes it easy for users to harness existing, cutting-edge DENN software. Furthermore, we insert an easily-implemented preliminary data simulation step that reduces the computational demands of the approach and empirically demonstrates that it maintains the accuracy of parameter estimation for a stochastic oscillator model.

Keywords: parameter estimation; Bayesian inference; normalizing flows; deep learning; predator-prey cycles

1. Introduction

Mathematical models typically feature parameters with unknown values, even when data about the modeled system is available, either due to limitations on measurement precision or variability across instances of the system being measured. A classic problem in mathematical modeling is to use data x related to the modeled system either to find the single vector of parameter values θ^* in a parameter space $\mathcal{P}$ that is most likely to be consistent with the data or to compute a probability density $p(\theta \mid x)$ over values $\theta \in \mathcal{P}$, conditioned on the observation of x. A common approach for the latter is to use Bayesian inference, often employing simulation methods such as Markov Chain Monte Carlo (MCMC) to build up a density estimate $p(x \mid \theta)$ over many forward simulations of the model for various choices of parameter values θ related to an assumed prior distribution $\pi(\theta)$ on a parameter space [1]. Once this estimate is obtained, Bayes's theorem yields the following:

$$p(\theta \mid x) \propto p(x \mid \theta)\pi(\theta). \tag{1.1}$$

A more recent approach to the Bayesian estimation of a conditional probability density on a parameter space harnesses a form of density estimation neural network (DENN) [2]. In general, DENNs are a type of artificial neural network (ANN) that estimates the probability density or distribution of a random variable or a set of random variables. Early DENNs used a parametrized approximate likelihood function such that after training, the output layer produced a parameter vector that specified the features and weights of components of this likelihood function [3]. More recently, a shift occurred from parametric density estimation models to simulation-based models that incorporate a parametrized simulating function and use automatically simulated data for training [4, 5]. In this case, the idea is to use the simulated data to train an ANN to produce $p(x \mid \theta)$ such that Eq (1.1) can be used when a specific data point x is available.

Despite the computational efficiency and potential utility of DENNs, this tool has recently been characterized by Trentin [6] as "worryingly underrated in the recent literature on machine learning". Indeed, there are complications in bringing DENNs to bear for parameter estimation problems in mathematical modeling. Many practitioners in this area do not have a thorough understanding of deep learning models, along with the skills related to careful code adjustment and hyperparameter tuning required for the effective use of DENNs. As a result, traditional density estimation methods such as kernel density estimation and Gaussian mixture models remain much more widely used than DENNs. Moreover, there are many more implementation tools and packages available for these approaches than for DENNs. Recent works about DENNs proposed by Papamakarios et al. [5, 7] for Bayesian inference mostly focused on theoretical advances rather than the ease of implementation for users not in the field.

The goal of this paper is to provide clear, user-friendly guidance and code for researchers interested in studying or using DENNs. We demonstrate the workflow to employ a simulator-based DENN with a specific dynamical system, namely the Rosenzweig-MacArthur predator-prey model [8], using an existing Python package. Moreover, we introduce a modification to previous approaches by completing all of the data simulation in an initial step, with subsequent use of interpolation, rather than repeatedly generating simulated data throughout the ANN training process. Our experiments show that this approach provides both accuracy and computational efficiency, thus making it a practical alternative to generating simulated data that may be preferred for some users and applications. We hope that our systematic exposition of this approach on a specific test problem helps to publicize and enhance access to this powerful architecture for a wider population in the science and engineering communities.

2. Methodology

We describe both the theoretical and implementation details of how to use DENNs. Initially, we discuss the training objectives and the mathematical foundations which underlie DENN training. Then, we introduce the dynamical systems model that we use for our investigation, which is a stochastic version of the Rosenzweig-MacArthur system [8], tuned to produce oscillations in the absence of noise, which persist but, of course, become aperiodic when noise is present. Next, we discuss the idea of density estimation and how it applies to our model system. Finally, we present the code implementation tailored for our model. This is intended to not only illustrate our specific use case, but to also serve as a guide for those interested in exploring this tool in other contexts.

2.1. Flow-based density estimation and DENN training

Here, we denote vectors in bold font to explicitly distinguish them from their individual elements. Let $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$ denote a given data vector. Let $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots, \theta_p]$ denote the unknown or free parameters associated with a representation of the system that produces the data. Our goal is to estimate $p(\boldsymbol{\theta} \mid \boldsymbol{x})$ using (1.1), which requires $\pi(\boldsymbol{\theta})$ and $p(\boldsymbol{x} \mid \boldsymbol{\theta})$. Recall that $\pi(\boldsymbol{\theta})$ is a pre-selected distribution on the parameter space, which is often taken to be a multivariate uniform distribution but can be chosen otherwise; for example, a complete lack of prior assumptions about the distribution of the data is implemented by choosing a Jacobian prior [9]. Now, We shall discuss the mathematical formulation for the estimation of $p(\boldsymbol{x} \mid \boldsymbol{\theta})$ with synthetic data produced by a simulator.

We use $q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})$ to denote an estimate of the density $p(\boldsymbol{x} \mid \boldsymbol{\theta})$, which is parametrized by $\boldsymbol{\Phi}$. Here, $\boldsymbol{\Phi}$ represents the set of learnable parameters for the model, whose detailed structure will be introduced later. As a simple example, $q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})$ could be a sum of Gaussian functions, with the components of $\boldsymbol{\Phi}$ specifying the amplitudes and midpoint positions of each. The training process should generate a specific choice of $\boldsymbol{\Phi}$ to make the difference between $p(\boldsymbol{x} \mid \boldsymbol{\theta})$ and $q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})$ as small as possible. To measure this difference, we use Kullback–Leibler (K-L) divergence, which is a common statistical distance measure of how different two distributions are [10]. Then, we can define a training loss function using forward K-L divergence [7], which is defined as follows:

$$L(\boldsymbol{\Phi}) = D_{KL}[p(\boldsymbol{x} \mid \boldsymbol{\theta}) \,||\, q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})] := E_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}[\log p(\boldsymbol{x} \mid \boldsymbol{\theta}) - \log q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})], \tag{2.1}$$

where the notation $E_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}$ indicates that expectation is being evaluated with respect to the the distribution $p(\boldsymbol{x} \mid \boldsymbol{\theta})$.

Since $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$ under $\boldsymbol{x} \sim p(\boldsymbol{x} \mid \boldsymbol{\theta})$ is independent of $\boldsymbol{\Phi}$, the expression (2.1) for $L(\boldsymbol{\Phi})$ is equivalent to the following equation:

$$L(\boldsymbol{\Phi}) = -E_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}\left[\log q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})\right] + \ \textit{constant}. \tag{2.2}$$

In the remainder of this section, we discuss the formulation that we use for q. Once we have this formulation, we can substitute it into Eq (2.2) and then use an ANN to determine the $\boldsymbol{\Phi}$ that minimizes L. This can be done via the following standard approach:

$$\boldsymbol{\Phi}^{(t+1)} = \boldsymbol{\Phi}^{(t)} - \eta \nabla_{\boldsymbol{\Phi}} L(\boldsymbol{\Phi}).$$

While kernel density estimation or a mixture of Gaussians can be used to construct q, their versatility is constrained by their inherent parametric form. In the DENN approach, we define q as a neural density estimator by utilizing *normalizing flows*. A normalizing flow represents a transformation in the probability density using a series of mappings [11]. Each mapping is a differentiable and invertible object, called a *flow function*. Kobyzev et al. [12] showed that in a suitable setting, normalizing flows avoid the limitations of traditional density estimation, such as poor tail behaviors and the curse of dimensionality, and provide the power to approximate any target distribution, while avoiding manual selection of parameters. A normalizing flow for estimating q is defined via the following iteration:

$$\mathbf{Z_0} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})\,,\ \ \mathbf{Z}_i = g_i\left(\mathbf{Z}_{i-1}; \boldsymbol{\theta}\right)\ ,\ \ i = 1, \ldots, k. \tag{2.3}$$

where

- Z_0 is the initial random variable sampled from a standard normal distribution,
- $g_i(\cdot;\theta)$ is a differentiable, invertible function at each step i, parametrized by vector $\boldsymbol{\Phi_i} \in \mathbb{R}^{d_i}$, which consists of learnable weights in the neural network, and
- $\boldsymbol{\theta}$ represents the fixed parameters of the model, whose density we aim to estimate.

Each g_i in (2.3) is a differentiable, invertible function parametrized by vector $\boldsymbol{\Phi_i} \in \mathbb{R}^{d_i}$. $\boldsymbol{\theta}$ represents the fixed parameters of the target density $p(\boldsymbol{x} \mid \boldsymbol{\theta})$; in practice, based on the iteration (2.3), each g_i transforms a simpler distribution to a more complex one. Therefore, We can denote (2.3) as the following:

$$\mathbf{Z}_k = T_{\boldsymbol{\Phi}}(\mathbf{Z}_0;\boldsymbol{\theta}), \quad \text{where} \quad T_{\boldsymbol{\Phi}} = g_k \circ g_{k-1} \circ \cdots \circ g_1 \tag{2.4}$$

for $\boldsymbol{\Phi} = \{\boldsymbol{\Phi}_1, \ldots, \boldsymbol{\Phi}_k\}$; here, note that in general, $\boldsymbol{\Phi}_i, \boldsymbol{\Phi}_j$ can have dimensions $d_i \neq d_j$ when $i \neq j$.

Ultimately, the idea is to use Eq (2.4) to provide the following desired density estimate:

$$q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta}) \sim \mathbf{Z}_k. \tag{2.5}$$

Based on the change of variables formula associated with the transformation of random variables, if we denote a multivariate standard normal distribution as $h(\mathbf{z})$, then Eqs (2.3)–(2.5) give the following:

$$q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta}) = h\left(T_{\boldsymbol{\Phi}}^{-1}(\boldsymbol{x});\boldsymbol{\theta}\right) \cdot \mid \boldsymbol{J} \mid, \tag{2.6}$$

where $\boldsymbol{J}$ is the corresponding Jacobian matrix with respect to the transformation $T_{\boldsymbol{\Phi}}$. Now, (2.2) becomes the following:

$$L(\boldsymbol{\Phi}) = -E_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}\left[h\left(T_{\boldsymbol{\Phi}}^{-1}(\boldsymbol{x});\boldsymbol{\theta}\right) \cdot \mid \boldsymbol{J} \mid\right] + \ constant. \tag{2.7}$$

The true distribution $p(\boldsymbol{x} \mid \boldsymbol{\theta})$ is unknown to us. The actual DENN implementation [5] uses simulations of the model under study to obtain a set of training pairs $\{\boldsymbol{\theta}_i, \boldsymbol{x}_i\}_{i=1}^N$, after which Monte Carlo simulations are used to estimate the expectation in Eq (2.7) as follows:

$$L(\boldsymbol{\Phi}) = -\frac{1}{N}\sum_{i=1}^{N}\left[h\left(T_{\boldsymbol{\Phi}}^{-1}(\boldsymbol{x}_i);\boldsymbol{\theta}_i\right) \cdot \mid \boldsymbol{J} \mid\right] + \ constant. \tag{2.8}$$

The loss function in Eq (2.8) is only dependent on $\boldsymbol{\Phi}$. We can evaluate the gradient $\frac{\partial}{\partial \boldsymbol{\Phi}}L(\boldsymbol{\Phi})$ and use gradient descent to minimize the loss; then, we take $\boldsymbol{\Phi} = \arg\min L(\boldsymbol{\Phi})$ and define $q_{\boldsymbol{\Phi}}$ correspondingly. While numerical methods can also be deployed for this task, we use an ANN because of the nature of a normalizing flow. The network takes $\{\boldsymbol{\theta}_i, \boldsymbol{x}_i\}_{i=1}^N$ as the input and gives $\boldsymbol{\Phi}$ as the output. The back-propagation mechanism can use the gradient to efficiently update the parameter matrix $\boldsymbol{\Phi}$. Hence, ANNs are well suited for density estimation using normalizing flows. Once we have obtained $\boldsymbol{\Phi}$ so that we can represent q using (2.6), for any newly observed data point $\boldsymbol{x}$, we can perform MCMC sampling of the prior distribution, and hence, use Eq (1.1), with $p(\boldsymbol{x} \mid \boldsymbol{\theta})$ replaced by $q_{\boldsymbol{\Phi}}(\boldsymbol{x} \mid \boldsymbol{\theta})$, to approximate $p(\boldsymbol{\theta} \mid \boldsymbol{x})$.

2.2. Rosenzweig-MacArthur system

The Rosenzweig-MacArthur predator-prey model (R-M model) is a mathematical model that describes the population dynamics between a single predator and single prey species in an ecological system [8]. The R-M model has a similar form to the well-known Lotka-Volterra equations, but with the added realism that prey consumption saturates when the size of the prey population grows (i.e., a type 2 functional response [13]). We selected the R-M model as a focus for our demonstrations because it is a nonlinear system that supports a non-trivial attracting solution, namely a limit cycle, which can be analytically understood but is not represented in a closed form, over a well-characterized parameter range. Moreover, the R-M system is two-dimensional, which is convenient for visualization, has parameters that have distinct impacts on the system dynamics [14], and represents a setting in which stochasticity is a naturally occurring feature, thus adding an important and commonly arising challenge to parameter estimation.

The deterministic version of the R-M model is described by the following ordinary differential equation (ODE) system:

$$\begin{cases} \dfrac{dx}{dt} = rx\left(1 - \dfrac{x}{K}\right) - \dfrac{axy}{x+h} \\ \dfrac{dy}{dt} = \dfrac{abxy}{x+h} - my. \end{cases} \tag{2.9}$$

where x represents the prey population density, and y represents the predator population density. The quantities m, h, r, a, b, and K are six positive parameters that can be tuned based on the ecological setting of interest, with the following interpretations: m represents the predator death rate; h represents the prey density at the half-maximum consumption rate, or half-saturation constant; r represents the maximum growth rate of prey; a represents the maximum prey consumption rate or attack rate; b represents the prey-to-predator conversion efficiency; and K represents the environmental carrying capacity for the prey in the absence of predation.

We assume that $ab > m$, which favors non-extinction of the predator species. In this case, model (2.9) has critical points which correspond to mutual extinction at $(x, y) = (0, 0)$, predator extinction at $(x, y) = (K, 0)$, and coexistence at $(x, y) = (x_c, y_c)$ with $x_c, y_c > 0$; formulas for x_c, y_c can be obtained analytically but are not important here. Consistent with the condition $ab > m$, the $(K, 0)$ critical point is an unstable saddle, thus representing the limiting state when only the prey and no predators are present; additionally, $(0, 0)$ is a saddle, thus representing the limiting state when only the predator and no prey are present. Linearization about (x_c, y_c) and the associated analysis shows that this point can undergo an Andronov-Hopf bifurcation that gives rise to stable periodic solutions. Specifically, the model exhibits periodic oscillations when the parameters satisfy the following [14]:

$$h < K(ab - m)/(ab + m). \tag{2.10}$$

While the deterministic model perfectly supports periodic oscillations, real-world ecological systems naturally include stochasticity. A reasonable way to include stochastic effects that are uncorrelated across parameters is to allow some or all parameters to vary according to a Cox–Ingersoll–Ross stochastic process (CIR process) [14, 15]. This formulation prevents the parameters from becoming negative, and has been previously shown to have various mathematically desirable and biologically reasonable properties [16]. The CIR process for a parameter p takes the form of the following stochastic differential equation (SDE):

$$dp = \gamma(\bar{p} - p)dt + \sigma\sqrt{p} \cdot dW, \tag{2.11}$$

where $\bar{p}$ denotes the mean value of parameter p, γ represents the rate at which p reverts to its mean, and σ represents the standard deviation of the CIR process. In Eq (2.11), W is a Wiener process, which can be simulated by $dW = \sqrt{t} \cdot Z$ for $Z \sim \mathcal{N}(0, 1)$.

The R-M system cannot be analytically solved. However, it can be numerically realized via the Euler–Maruyama method, with or without stochasticity in its parameters. Figure 1 exhibits an example simulated time course of system (2.9) in the (x, y) phase plane under the baseline parameter values specified in Table 1, with both m and h governed by copies of Eq (2.11). These values are intended to illustrate certain dynamic characteristics and are not reflective of a specific biological situation. Each cycle can be viewed as one outbreak event in which, starting from low densities of both species, the prey density grows, followed by a subsequent growth in the predator density and a decline in the prey density; then, this is followed by a crash in the predator density once the prey density becomes sufficiently low. For such a cycle, we can measure an effective period by selecting a segment in the (x, y) plane (i.e., a Poincaré section) that is clearly crossed by the trajectory in the same direction on each cycle, such as $\{(x, y) : 0 < x < 0.3, y = 0.15\}$ in Figure 1, and computing the time between two successive crossings in the same direction (e.g., with decreasing y). Then, we can measure an x amplitude and a y amplitude for each cycle by computing the difference between the maximal and minimal values of each variable on that cycle.

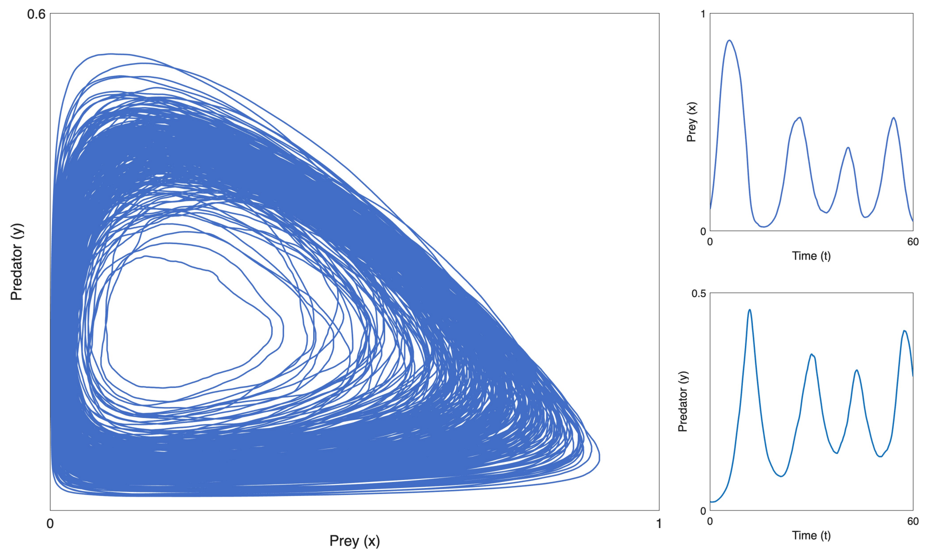

Figure 1. Prey-predator oscillation generated by the R-M model (2.9), with m and h varying as the CIR processes (2.11) and parameter values given in Table 1. The phase plane on the left shows the trajectory composed of the x, y values over 5×10^5 time units, of which the first 60 time units are shown in the time courses on the right.

2.3. Application of DENN to the R-M model

When using the R-M model or another such dynamical systems model to study a particular biological system, researchers may be interested to determine the model parameter values that best fit some observed data and thereby gain insight about the biological characteristics defined by the parameters. As such, obtaining the conditional distribution of these parameter values given the observed data can allow researchers to better understand and potentially manipulate the biological dynamics underlying the data. Additionally, this knowledge facilitates the prediction of future states under different scenarios, which is critical information when it comes to conservation planning, managing biological resources, or dealing with issues such as invasive species or insect outbreaks.

In our case, we focus on the conditional distribution of the parameters m and h, which represent the predator death rate and the prey density at a half-maximum predator consumption rate, respectively. This choice is based on a previous study that highlighted the distinctive roles of these parameters in controlling the outbreak timing variability [14]. Specifically, m is one of several parameters that enters the model linearly and has a relatively weak effect on the oscillation timing, yet does impact the oscillation amplitude and appears in the condition (2.10) for the existence of oscillations; moreover, h enters the model nonlinearly and strongly impacts oscillation timing along with existence. We fix the other 4 parameters according to the baseline values given by Rubin et al. [14], namely $a = 2, b = 0.5, K = 1$, and $r = 1$. Meanwhile, we constrain m and h to lie within the range $[0.2, 0.6]$, where oscillations have the possibility of occurring with a range of periods and amplitudes; since we are interested in the regime in which the system exhibits stable periodic oscillations in the absence of stochasticity, we also impose the conditions $m + h + mh < 1, mh < 1$. To simplify the parameter sampling process, we approximate this region with the following triangular constraints:

$$\begin{cases} m + h \leq 0.8, \\ 0.2 \leq m \leq 0.6, \\ 0.2 \leq h \leq 0.6. \end{cases} \tag{2.12}$$

Then, we turn to data simulation. There are two data simulation processes related to our network training. The first step is that for a mesh of choices of (m, h) within the region specified by (2.12); we numerically simulate the stochastic model system (2.9)–(2.11), obtain (x, y) orbits, and compute relevant training data from these orbits. The second step is to train the algorithm samples (m, h) values from the prior distribution on parameter space and map each pair to the corresponding network training data that we generated in the first step. To avoid confusion, we shall henceforth refer to the first step as data simulation and the second step as network simulation.

For the data simulation part, we use the commonly applied Euler–Maruyama method to update variables of the stochastic model system from a collection of known values at time t. The update rule based on this method is specified as follows:

$$\Delta m(t) := \gamma(\bar{m} - m(t)) - \sigma \cdot \sqrt{m(t)}\sqrt{\Delta t} \cdot Z$$
$$\Delta h(t) := \gamma(\bar{h} - h(t)) - \sigma \cdot \sqrt{h(t)}\sqrt{\Delta t} \cdot Z$$
$$m(t + \Delta t) := m(t) + \Delta m(t)$$
$$h(t + \Delta t) := h(t) + \Delta h(t)$$

$$x(t+\Delta t) := x(t) + \Delta t \cdot [rx(t)(1 - \frac{x(t)}{K}) - \frac{ax(t)y(t)}{x(t)+h(t)}]$$

$$y((t+\Delta t) := y(t) + \Delta t \cdot [\frac{abx(t)y(t)}{x(t)+h(t)} - m(t)y(t)]$$

for a fixed time step Δt.

Thus, we can sequentially update (x, y) over long simulation times and obtain large collections of oscillation cycles. To define a cycle, we set two Poincaré sections or thresholds, one on each of the x-axis and the y-axis, and specify that one cycle is completed every time the simulation trace starts from the first threshold, passes through the second threshold at least once in a specified direction, then reaches the first threshold again. Let n_i denote the number of time steps and the corresponding (x, y) points within the i^{th} cycle, and set $(x_{i_1}, y_{i_1}), \ldots, (x_{i_{n_i}}, y_{i_{n_i}})$ as the points on the system trajectory computed during that cycle. For each cycle, we calculate the x oscillation amplitude X_{amp}, the y oscillation amplitude Y_{amp}, and the cycle period T for use in the network training, where $X_{amp} = \max_j \left\{x_{i_j}\right\} - \min_j \left\{x_{i_j}\right\}$, $Y_{amp} = \max_j \left\{y_{i_j}\right\} - \min_j \left\{y_{i_j}\right\}$, and $T = \Delta t \cdot n_i$. Therefore, we obtain a collection of $[X_{amp}, Y_{amp}, T]$ values, one per cycle, which we call *synthetic data points*.

In the standard DENN implementation, the model simulation and calculation of synthetic data poins described above would be embedded in the ANN described in Section 2.1. However, doing so can greatly slow down the training process because of the time occupied by generating the synthetic data. Instead, we opt for an approach that separates the data simulation from the network training. Specifically, we define a grid plan of mesh size 0.02 over the parameter region (2.12) and use the grid points as a discretized collection of parameter vectors, denoted as (m_j, h_j). In total, we shall obtain $(21 + 1) \times 21/2 = 231$ grid points given the defined parameter region and mesh size. We note that in problems with higher-dimensional spaces of unknown parameters, such brute force sampling may become computationally intractable. Fortunately, there are a large range of Latin hypercube sampling methods that have been developed over decades that allow for efficient sampling and the subsequent use of sampled points [17], as well as other adaptive subsampling approaches [18].

Each (m_j, h_j) is used as part of the initial condition for our Euler-Maruyama simulation of (2.9)–(2.11), with the (x, y) coordinates of the initial condition fixed across all simulations. Then, we use nested loops to iterate m and h over the entire grid plan; for each, we run a simulation over multiple cycles to generate a corresponding collection of $[X_{amp}, Y_{amp}, T]$ values for each j, say $\{[X_{amp}, Y_{amp}, T]_j\}_i$, thus completing the work of data simulation before the neural network comes into play at all.

For our second step, namely the network simulations, many parameter vectors θ are sampled from the prior distribution on the parameter space. In our approach, each of these is mapped to the nearest (m_j, h_j) from our initial grid, after which the simulator randomly samples one data vector from the stored collection, $\{[X_{amp}, Y_{amp}, T]_j\}_i$ for some fixed, randomly chosen i. These (θ, x) pairs form the inputs for the ANN training process described in Section 2.1, which results in a representation of the likelihood function $p(x|\theta)$ that appears in Eq (1.1).

Figure 2 shows the working diagram for the overall algorithm that we use, which summarizes all of the steps necessary to construct the network. While some details of the workflow, such as our use of X_{amp}, Y_{amp}, and T values as data, are based on the specific details of the R-M model illustrated here, the overall idea and methodology can be naturally replicated for other systems. If the mapping from the parameters to the data is analytical, then the first step, "Obtain data", can be reduced.

We acknowledge that some readers may confuse the SDE updates of (m, h) in the R–M oscillations with the parameter updates in the normalizing flow of DENNs, as both involve iterative updates. To clarify, Figure 3 illustrates the complete workflow of our data preprocessing and model implementation. As shown in the left panel, the SDE drives the evolution of (m, h) in the R–M system under CIR noise, whereas DENN updates the flow parameters Φ in the composition $g_0 \circ \cdots \circ g_k$. These are sequential but distinct steps, and should not be conflated.

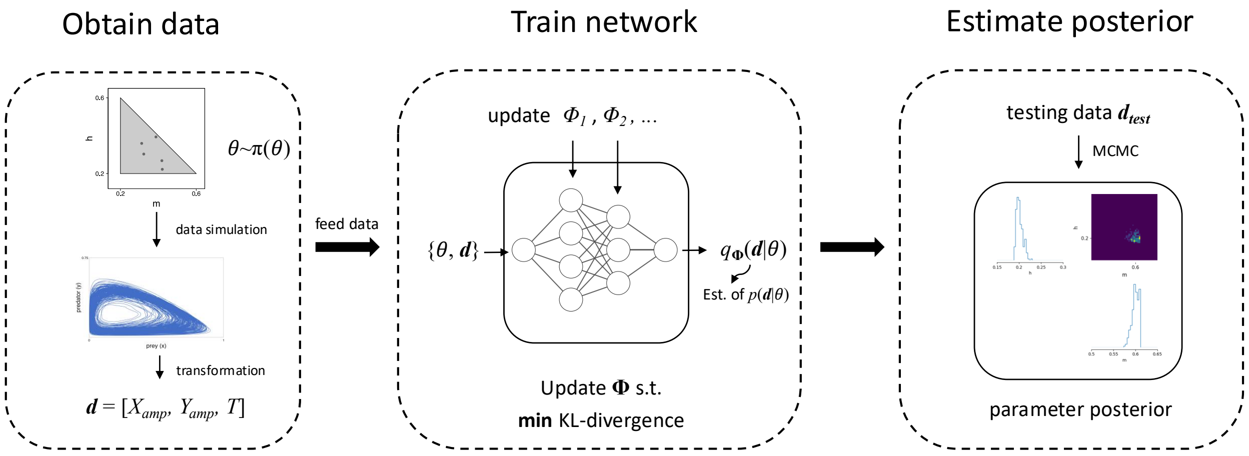

Figure 2. Schematic flowchart summarizing the workflow of constructing and using our density estimation network. We obtain data by sampling parameter vectors θ from a prior distribution π on parameter space, simulating the RM model (2.9) with the sampled parameter values used for $\bar{m}, \bar{h}$ and stochasticity in m, h given by (2.11), and computing d from the simulation. The DENN is trained, using $\{\theta, d\}$ pairs from pre-processing to generate an estimate of the posterior distribution $p(d|\theta)$. Finally, we test model performance at accurately estimating $\theta = (m, h)$ when presented with testing data d_{test} generated by simulating the original model.

After the data are obtained and the network is trained, the DENN represents a computational version of $p([m, h]|[X_{amp}, Y_{amp}, T])$, which can take data points $[X_{amp}, Y_{amp}, T]$ and yield the posterior density consisting of the joint conditional density of (m, h). To assess the performance of our trained network, we can generate new synthetic data for a set of randomly selected (m, h) values and compare the posterior density for each with the true (m, h) values. We evaluate the performance of the simulator using two performance metrics defined as follows:

- **Precision**: The width of the 95th percentile range (i.e., the interval from the 2.5th to the 97.5th percentile) of a sample from the posterior distribution for each tested (m, h) pair. Narrower range indicates more precise prediction. (Note that precision does not imply accuracy.)
- **Relative error (RE)**: The error in percentage between each true parameter value and the median value of the posterior sample for that parameter, divided by the true value.

The first metric measures the specificity of the computed posterior density, while the second represents its accuracy.

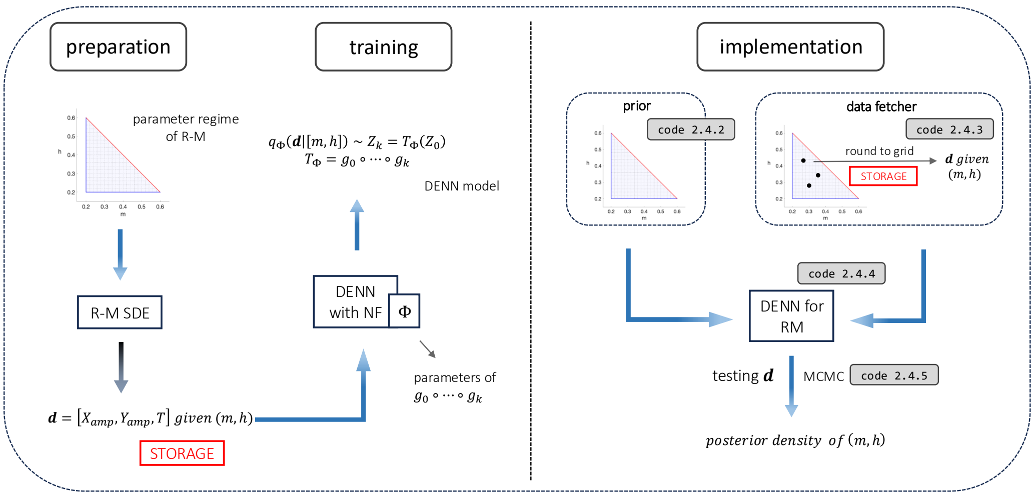

Figure 3. Comprehensive workflow diagram. The left panel summarizes the preprocessing pipeline that combines R–M SDE simulations with the DENN training process. The right panel highlights the implementation steps for the use of the trained DENN to estimate (m, h) from data, directly aligned with the Python code provided in Section 2.4.

2.4. *Code structure*

As DENN involves simulation-based training, proper coding is critical in constructing the model and obtaining reliable parameter estimates. This subsection guides the readers through how to code a DENN model based on our R-M example. We aim to provide a universal workflow for any density estimation work on parameters of dynamical systems models. The details of all code, figures, and data can be found in the GitHub repository at https://github.com/bosihou/Density_Estimation_Network_Application. Note that for the Python script, the package SBI is called and used extensively throughout the coding. SBI stands for simulation-based inference, which is the process of finding parameters of a mathematical simulator given observation data. This toolkit is provided by Tejero-Canteroe et al. [19]. Users can refer to https://github.com/sbi-dev to view detailed documentation and explore a variety of estimation algorithms. For clarity, readers may also consult the right panel of Figure 3, which links each implementation step to the corresponding code snippets.

2.4.1. Working environment

As training DENN involves utilizations of various packages, we recommend that users employ conda for virtual environments for ease of installation and maintenance of consistent dependencies. Listing 1 is a typical combination of package imports.

```python
# Import packages
import torch
import pandas as pd
import numpy as np
import h5py
from random import Random

from sbi import utils as utils
from sbi import analysis as analysis
from sbi.inference import infer
```

Listing 1. Import packages.

2.4.2. Prior

The prior specifies a distribution over the domain of model parameter values that we want to consider. It needs to be a `torch.distribution` object to be parsed correctly. In our case, we define a box-uniform domain for (m, h), as shown in Listing 2. Note that the actual domain for (m, h) is triangular, as given by (2.12). We extend our parameter domain via symmetric mapping. Specifically, we define a uniform distribution of (m, h) such that $\{(m, h) \mid 0.2 \le m \le 0.6, 0.2 \le h \le 0.6\}$. The parameter sampler of the network samples (m, h) from this box region. If the (m, h) pair falls in the parameter regime given by (2.12), it will be passed to the next step, data simulation; if the (m, h) pairs fall out of the regime, then we simply perform a mapping with the following:

$$\begin{cases} m' := 0.8 - h \\ h' := 0.8 - m \end{cases} \tag{2.13}$$

We are able to do so because this symmetric mapping takes care of all possible sampling points in this box region. Users can also implement their own prior distribution by extending `torch.distributions.Distribution`. It is, however, crucial that the self-defined class can be sampled efficiently to avoid excessive training times. We recommend that users make as much use as possible of the built-in distributions.

This step corresponds to the ‘prior’ block shown in Figure 3 (code 2.4.2).

```python
num_dim = 2 # Specify prior
uniform_prior = utils.BoxUniform(low = 0.19 * torch.ones(num_dim), high = 0.61 *
    torch.ones(num_dim))
# As the grid size is 0.02, we extend the bounds to take care of rounding for boundary
    values
```

Listing 2. Specify parameter prior.

2.4.3. Data fetcher

The code in Listing 3 defines a functional mapping from the parameters to the data. We refer to this component of the method as the data fetcher because this step involves accessing the simulation data generated in pre-processing. Specifically, it takes parameters sampled from the prior as input and produces previously generated and stored simulation data as output. The output is usually a torch tensor. The combination of pre-processing including simulation followed by data fetching in the implementation phase represents a practical option to avoid the computational expense of embedding simulation in the network training process. The last line in the code listing gives an example $[X_{amp}, Y_{amp}, T]$ output by our data fetcher. The implementation details of our data fetcher can be found in our GitHub repository.

This step corresponds to the 'data fetcher' block shown in Figure 3 (code 2.4.3).

```
# This is a data fetcher, which is a Python callable. It takes an (m,h) pair as input
    and randomly fetch one piece of [X,Y,T] data as output, which corresponds to this
    (m,h) pair

test_parameter = torch.tensor([0.2, 0.2])
ODE_simulator_on_fine_grids(test_parameter)

>> tensor([0.9224,  0.6045, 42.5000], dtype=torch.float64)
```

Listing 3. Define the data fetcher.

2.4.4. Network training

After the prior and simulator are properly defined, we can utilize the `infer()` function in `sbi` to train the network. The keyword argument `method` specifies the training method. We use the Sequential Neural Likelihood Estimation (SNLE) method because it allows the trained network to take in multiple observed data points for estimation. Keyword `num_simulations` specifies the number of data samples for which we want the network to estimate parameter values. Listing 4 shows the coding details.

This step corresponds to the 'DENN for RM' block shown in Figure 3 (code 2.4.4).

```
network = infer(ODE_simulator_on_fine_grids, # the data fetcher
                uniform_prior, # the R-M parameter regime
                method = "SNLE", # specific implementation of the DENN
                num_simulations = 10000)
```

Listing 4. Train network.

2.4.5. Posterior sampling and pair plot examination

When the training is done, we can feed new data points to the network to obtain samples of the corresponding posterior distributions on parameter space. An intuitive way to examine the network performance is to visually check the pair plots of posterior samples. Figure 4 shows example results generated by our workflow using $[X_{amp}, Y_{amp}, T]$ data generated with the code in Listing 5. Note that when an (m, h) posterior sample is obtained, the code applies the inverse transformation (2.13) if the sample is outside of our parameter regime, so that the distribution is unimodal instead of bimodal.

This step corresponds to the 'testing / MCMC' block shown in Figure 3 (code 2.4.5).

```
# Get testing data
test_data = full_data[0,20,:,:] # testing data comes from (m,h) = (0.2,0.6)
num_of_observations = 20
observation = get_sample(test_data, num_of_observations)
# Posterior sampling
posterior_samples = network.sample((200,), x = observation)
```

Listing 5. Test network performance.

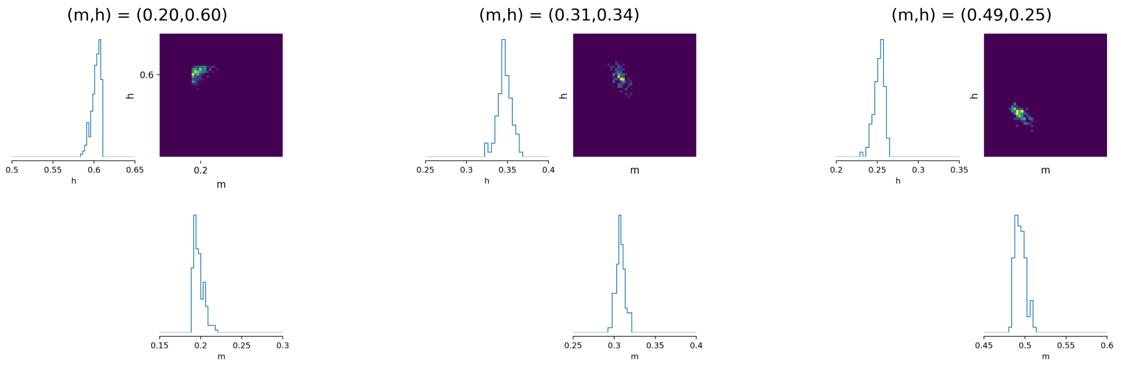

Figure 4. DENN results for data generated from the stochastic R-M model (2.9)–(2.11) on three trials with parameter values $(m, h) = (0.2, 0.6), (0.31, 0.34), (0.49, 0.25)$, respectively. In each case, the colored plot shows the joint posterior distribution obtained on (m, h) parameter space. The other plots are marginal distributions of the generated estimates for each parameter, which can all be seen to be centered on the ground truth values.

3. Results

3.1. Performance overview

In this section, we discuss the results of our tailored application of DENN to the Rosenzweig-MacArthur model, emphasizing a comparative analysis with traditional DENN implementations. We first examine the model estimation performance in terms of the metrics defined in Section 2.3. Next, we compare the performance of our implementation with that from a direct implementation of DENN. We also present algorithmic complexity and memory usage analysis between the two implementations to provide a holistic understanding of the efficacy and efficiency of our approach.

Our implementation of DENN performs preliminary data simulation, which generates synthetic data before actual training. We simulate the model (2.9)–(2.11) numerically on a discretized parameter grid within the domain specified by Eq (2.12), with $\Delta m = \Delta h = 0.02$ as our grid size. Figure 5 illustrates the parameter domain and demonstrates the ODE oscillations for a variety of (m, h) pairs distributed across our triangular domain. For each subplot, the x-axis (y-axis) represents the density of the prey (predator) population. The initial condition for the simulations has $(x, y) = (0.1, 0.02)$, and due to the choices of parameter values, all runs result in sustained oscillations. When the parameter pair (m, h) approaches the diagonal domain boundary, the variability introduced by the CIR process (2.11) increases, as reflected in the model trajectories.

We used DENN with pre-computed data to estimated the joint posterior distribution over (m, h)

parameter space and the marginal distributions of each parameter for a variety of (m, h) values and observed what appeared to be consistent success. An example of these results for three arbitrarily selected parameter vectors appears in Figure 4.

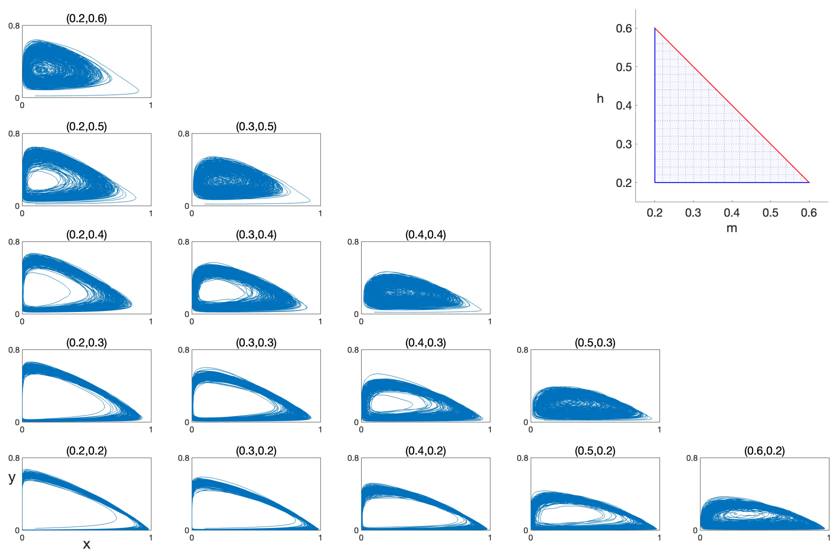

Figure 5. Trajectories of system (2.9)–(2.11) in the (x, y) plane for a collection of (m, h) values on the domain (2.12). Note that the (x, y) axis labels in the lower left apply to all subplots. The title of each subplot informs the values of (m, h) in this specific trajectory. The top-right panel illustrates the mesh region used by the sampler to generate (m, h) pairs (mesh size = 0.02), as specified by Eq (2.12).

To check its performance more systematically, we implemented our parameter estimation algorithm for the parameter values $\{(m, h) \,|\, m, h \in \{0.2, 0.3, 0.4, 0.5, 0.6\}, m + h \leq 0.8\}$. We quantified the algorithm's performance in correctly estimating m and h separately, since we want to ensure that the model yields reliable estimation for both values. Figure 6 shows the metrics computed to evaluate performance on the estimation of m. From left to right, the two color plots demonstrate precision and relative error, respectively, over the selected (m, h) values. For instance, the block in row 1, column 3 corresponds to the actual parameter values $(m, h) = (0.4, 0.2)$. The color-coding of that block in the two plots shows that for $(m, h) = (0.4, 0.2)$, the precision of m is 0.02; and the relative error based on the median of this range is around 2%. Figure 7 shows the evaluation of performance on the estimation of h, with similar interpretation.

From Figures 6 and 7, we see that both m and h are generally well estimated. Relative errors are less than 4% except in three cases, where it remains less than 8%. The precision plot suggests that the estimation of m is more precise than that for h. This difference may relate to the nonlinear way that h enters system (2.9), which leads to distinct impacts on cycle-to-cycle variability [14].

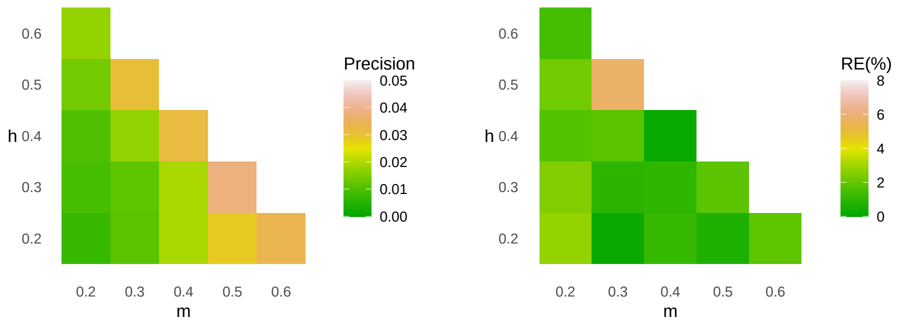

Figure 6. Evaluation of performance of DENN parameter estimation with preprocessing, with respect to parameter *m*. From left to right, the metrics are precision and relative error, as defined in Section 2.3.

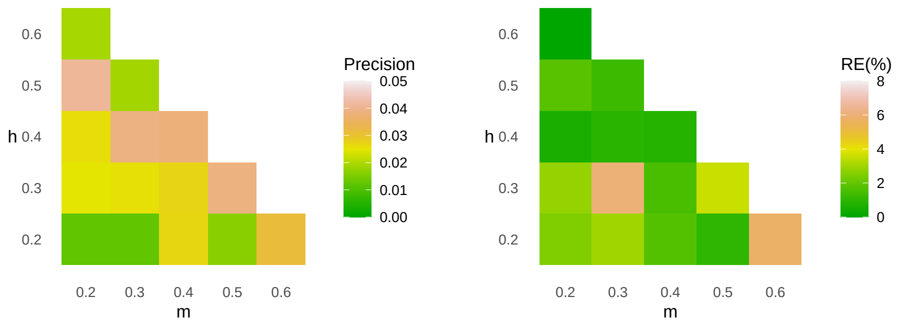

Figure 7. Evaluation of performance of DENN parameter estimation with preprocessing, with respect to parameter *h*. From left to right, the metrics are precision and relative error, as defined in Section 2.3.

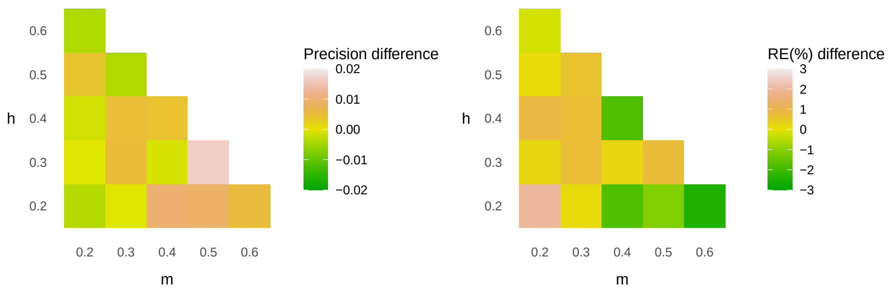

Figure 8. Performance comparison based on success in estimating the value of *m*: precision and relative error for our data-preprocessing DENN implementation minus those for direct DENN implementation.

We compared the continuous performance metrics, precision and relative error, from the direct DENN implementation published previously [20] with those from our approach based on data-preprocessing (see Section 2). Ideally, the difference of both precision and relative error should be close to zero, since the direct implementation has been shown in past work to yield high accuracy.

Figures 8 and 9 illustrate the differences between the metrics computed from our implementation minus those computed from the direct implementation. For instance, on row 1, column 3, the precision difference is 0.008, meaning that the width of the 95th percentile range of the estimate for m generated by our implementation at $(m, h) = (0.4, 0.2)$ is 0.008 greater than the 95th percentile range of the estimate from the traditional implementation. We can see that over the full range of (m, h) values tested, all performance differences are quite small, which indicates that the data-preparation approach does not sacrifice the accuracy obtained via the more training-intensive direct implementation, and in some cases may even yield improved accuracy (negative values in the figures).

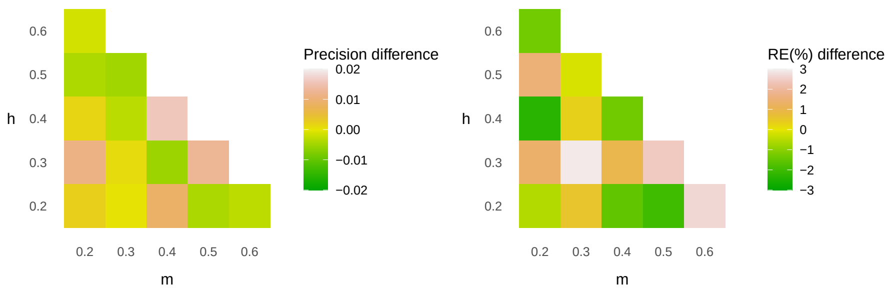

Figure 9. Performance comparison based on success in estimating the value of h: precision and relative error for our data-preprocessing DENN implementation minus those for direct DENN implementation.

The objective performance tests that we have described so far used grid-based parameter points for testing the network. We expanded our tests to include testing points sampled randomly from the parameter range specified by (2.12), without rounding to the nearest grid point. These points are directly input into a data simulator to generate synthetic data, which is then fed into the trained network. We evaluate the parameter posterior using the metrics defined in Section 2.3. Table 1 showcases the evaluation results for each randomly sampled pair of parameters (m, h). Each cell contains a tuple: the first value represents the metric associated with parameter m, and the second value pertains to parameter h. The results demonstrate that the network given our grid-point implementation provides precise and accurate estimations even for off-grid parameter values. This confirms the model's consistent performance across the entire parameter regime.

Table 1. Parameter estimation results for data from randomly selected parameter values.

True (m, h) values	Median (m, h) estimates	Precision in (m, h) estimates	RE (%) for median (m, h) estimates
(0.309, 0.446)	(0.304, 0.446)	(0.019, 0.035)	(1.745, 0.067)
(0.251, 0.416)	(0.253, 0.426)	(0.016, 0.041)	(0.633, 2.347)
(0.326, 0.248)	(0.331, 0.251)	(0.015, 0.032)	(1.570, 1.235)
(0.378, 0.393)	(0.383, 0.398)	(0.069, 0.067)	(1.408, 1.281)
(0.514, 0.221)	(0.532, 0.214)	(0.027, 0.030)	(3.347, 3.030)

3.2. *Robustness tests*

In this section we analyze the robustness of our DENN implementation under different conditions. Robustness is a key factor for evaluating the practical applicability of density estimation methods, especially in noisy or real-world scenarios. To this end, we designed two sets of experiments. First, we compare DENNs against traditionally used classical models to evaluate relative accuracy and robustness when subject to perturbations. Second, we examine the sensitivity of our DENN to deviations from the assumed Cox–Ingersoll–Ross (CIR) noise process by testing its performance on data generated with an alternative stochastic process.

3.2.1. Robustness comparison with classical models

We choose to compare DENNs against Gaussian Mixture Models (GMMs) [21] and K-Nearest Neighbor Regressors (KNNR) [22]. For GMMs, a critical hyperparameter is the number of Gaussian components, which directly impacts model performance. To determine the optimal number, we performed a grid search over component values ranging from 5 to 30 and selected the model with the lowest testing error, which resulted in an optimal choice of 25 components. Both GMMs and KNNRs are commonly used density estimation models that perform well on relatively smooth and well-structured high-dimensional data. However, in scenarios with irregular distributions, or significant noise, their effectiveness may degrade, which motivates the introduction of DENN for improved robustness. All models were trained and tested on identical conditions to ensure that the results are valid. For these experiments, we implemented the following step-by-step procedures:

1) **Training data generation:** Define a training set of size 10,000, where data is generated based on the preprocessing steps as defined in Section 2.3. Specifically, for the sampled parameter points, we round them to the nearest grid point and randomly sample one data vector from the corresponding stored distribution of $[X_{amp}, Y_{amp}, T]$.

2) **Testing data generation:** Define a separate testing set of size 20, where data is obtained by simulating the RM model with CIR noise directly for the relevant (m, h) pair.

3) **Train models and evaluate the posterior:** Train each model using the training data. Next, feed the corresponding testing data into the model to obtain posterior samples of size 100 (i.e., generating $[m, h]$ given $[X_{amp}, Y_{amp}, T]$). Finally, calculate the RE(%) of the sample medians for (m, h).

4) **Noise injection:** Add Gaussian noise to the testing data, $data \leftarrow data + STD(data) \cdot Z$, where $Z \sim N(0, 1)$ and $STD(data)$ represents the standard deviation computed for each individual column in the dataset. This step ensures that the noise added to each column is proportional to its own scale.

5) **Re-evaluate posterior:** Sample the posterior from each model based on the noisy testing data, and then re-calculate the REs.

6) **Compute robustness:** Calculate the change in performance in the noisy case for each model; for example, if RE of parameter m increases from 9% without added noise to 12% with it, then the change in percentage points is recorded as 3, such that a lower percentage change indicates higher robustness.

Table 2 demonstrates the results of these tests. We can see that DENN yields the best initial accuracy, providing stable estimates for both m and h. The second-best candidate is KNNR, yet its performance across the two parameters is inconsistent. GMMs yield the least accurate results due to their assumption that the underlying data distribution can be well-approximated by a mixture of Gaussian components. This outcome indicates that the mapping from R-M parameters to $[X_{amp}, Y_{amp}, T]$ features is more complex and cannot be well-learned by GMMs. The table also shows the accuracy of models with noisy data and the change in RE relative to the case without noise injection. We can see that all models demonstrate some degree of robustness to noise. GMMs show the smallest relative percentage change in accuracy, which is reasonable, considering that GMMs learn the joint distribution of $[m, h, X_{amp}, Y_{amp}, T]$ and, due to their parametric nature, usually sacrifice accuracy for lower model variance. While DENN is not the most robust model in terms of its estimation of h, which enters the RM model nonlinearly, it still gives the least relative error in the presence of noise, out of the three models.

Table 2. Accuracy and robustness summary for each model.

Model	RE (%)	RE (%) under noise	Absolute increase in RE (%)	Relative increase in RE (%)
GMMs	(6.0, 7.6)	(10.2, 12.9)	(4.2, 5.3)	(70.0, 69.7)
KNNR	(6.1, 0.7)	(11.2, 3.1)	(5.1, 2.4)	(83.6, 342.9)
DENN	(1.4, 1.5)	(2.3, 9.1)	(0.9, 7.6)	(64.3, 506.7)

The above analysis highlights that DENN achieves the highest accuracy among the tested models while maintaining reasonably strong robustness to the inclusion of noise in the testing data. Relative to GMMs, which sacrifice accuracy for stability, and KNNR, which struggles with imbalanced predictions even without noise, DENN delivers accurate estimates. These results support the use of DENN for complex density estimation tasks.

The disparity in performance across methods relates to certain model-level characteristics under our stochastic R–M setup. For a fixed (m, h), the stochastic oscillator induces conditional distributions over (X_{amp}, Y_{amp}, T) that are non-Gaussian and skewed, with dispersion growing as (m, h) increase. A finite GMM can only approximate such shapes with a limited set of ellipsoidal components. In addition, the expectation maximization step implemented in using GMMs tends to underfit tails [23] and is sensitive to initialization choices and to local optima when components overlap [24]. KNN-based estimators rely on local averaging, which tends to produce smoothing bias, especially when the underlying conditional density varies irregularly or is heteroskedastic [22]. By contrast, DENN learns a conditional, nonparametric mapping from (m, h) to the target density that adapts to shape changes across the parameter domain, which explains its superior accuracy and robustness in this setting.

3.2.2. Sensitivity to noise process

We next examine the performance of of our DENN, trained on data from the R-M model with CIR noise, on data generated with an alternative stochastic process. While we choose the CIR process for our pre-processing and training steps, real ecological systems might have different forms of noise correlations. More generally, it may happen that the stochastic process selected for inclusion in a model does not capture the full complexity of the noise in the physical system being modeled, and it is desirable for our parameter estimation method to retain its accuracy even under such a mismatch. It is thus important for us to test our model on synthetic data generated from the R-M model augmented with a different choice of stochastic process.

Specifically, we generate synthetic data based on the R-M model plus an Ornstein–Uhlenbeck (OU) process and then test how well our DENN, trained on the R-M model with a CIR process, estimates the underlying model parameters. The OU process [25] is a stochastic process with the following form:

$$dp = \theta(p - \bar{p})\,dt + \sigma\,dW_t.$$

Compared with the CIR process, which enforces positivity of the state variable through a square-root diffusion term and exhibits vanishing variance near zero, the OU process allows symmetric fluctuations around the mean without boundary constraints.

For data generation for these tests, we sampled R-M parameters randomly from the parameter range defined in (2.12), without rounding to the nearest grid points, and assessed DENN performance using the metric specified in Section 2.3. The key difference is that the $[X_{amp}, Y_{amp}, T]$ values presented to the DENN were generated under the OU process rather than the CIR process, although our DENN was trained on CIR data as previously. Table 3 reports the evaluation results, showing that the model continues to deliver strong precision and stable outputs. These results provide some evidence that our model remains robust even when the underlying noise structure differs from the one used during training, which suggests insensitivity to the specific choice of stochastic perturbation.

Table 3. Parameter estimation results for data generated from the R-M model with randomly selected (m, h) values and OU noise.

True (m, h) values	Median (m, h) estimates	Precision in (m, h) estimates	RE (%) for median (m, h) estimates
(0.210, 0.240)	(0.212, 0.248)	(0.009, 0.019)	(0.952, 3.333)
(0.305, 0.485)	(0.306, 0.486)	(0.017, 0.022)	(0.328, 0.206)
(0.472, 0.221)	(0.471, 0.229)	(0.022, 0.019)	(0.212, 3.62)
(0.288, 0.273)	(0.293, 0.278)	(0.010, 0.019)	(1.736, 1.832)
(0.265, 0.530)	(0.270, 0.522)	(0.016, 0.025)	(1.887, 1.509)

3.3. Analysis of computational efficiency

At this point, we have shown that the DENN approach yields good estimates of parameters for the stochastic nonlinear model (2.9)–(2.11), and that preprocessing data beforehand does not sacrifice accuracy relative to the previously proposed, training-intensive direct implementation. We now analyze the computational cost in terms of time and memory use for the two different approaches to DENN. We first consider run-time analysis from both theoretical and empirical perspectives, and then we discuss the memory use requirements of the two methods.

To recap, the data-preprocessing version of the density estimation network that we have introduced harnesses a preliminary step in which the model system of interest is simulated on a selected grid of parameter values. In the estimation process itself, therefore, after the prior on parameter space is sampled, each parameter vector obtained is mapped to the nearest grid point, and a randomly selected element of the previously computed synthetic data set for that grid point is used along with the sampled parameter vector in the training process (see Section 2.3). If the `infer()` elicits N simulations, then it will repeat this process N times to obtain N pieces of data (2.4.4). Since our version prepares and stores data for the network, the data generator is simply a data fetcher with complexity of $O(1)$, or $O(N)$ for N simulations. When all data are ready, the network will utilize the data to train the network. Empirically, the network converges within 200 training epochs, which is not time-consuming compared to the data simulation part.

The embedded model takes a different approach. Each time it samples one parameter set from the prior, it performs an entire set of stochastic DE simulations and a data transformation for each one. Suppose our stochastic DE and data transformation are numerically iterated for M time steps. This process is of $O(M)$ complexity to generate one data point and $O(MN)$ for N simulations in total. The theoretical run time of the embedded version is therefore more expensive than the data-prepared version by a factor of M. Since M needs to be large (typically 2–5 $\times 10^5$) to sufficiently simulate the R-M system, this factor is not negligible. This cost renders the embedded version as very inefficient considering that the model performance is not significantly better than the data-preprocessed version.

For empirical run-time testing, sampling 10,000 data points is often instantaneous for the data-preprocessed implementation, while it may take hours for the embedded model depending on the choice of M. Empirical run-time testing shows that if M is reduced by a factor α, the data generation time will also approximately drop by a factor α, which is consistent with this theoretical run-time analysis.

The memory usage for the two approaches is similar. Although the embedded model consumes computer memory when simulating and storing orbit data repeatedly for each parameter value sampled from the prior distribution, once a data point has been obtained, there is no need for this storage to be maintained, and hence the memory requirements stay bounded. For the preprocessing approach, there is the potential for an increased memory requirement. However, although the storage of data points generated for each parameter vector on the pre-defined grid is required, the storage of full trajectory information is not; moreover, the number of grid points that we use is only 231 based on (2.12) with $\Delta m = \Delta h = 0.02$, so in fact little memory is required for this storage. In practice, the majority of memory consumption occurs in the neural network training phase, after all synthetic data points have been obtained. This substantial memory usage is primarily driven by the mechanisms of forward and backward propagation. This dual requirement significantly increases the memory demands, as each layer's inputs, outputs, and gradients must be maintained in memory to successfully update the network [26]. Nonetheless, for both models, the memory requirement is usually between 20–60 MB, which is trivial even for a personal computer.

4. Discussion

Our work was motivated by a recent study on using DENN to estimate a posterior distribution on parameter space for a parametrized forward model, such as a nonlinear system of ODEs [7]. In this work, we aim to describe this approach in terms of an accessible workflow to help other researchers to utilize this powerful tool for their own purposes. We also suggest and illustrate the possible utility of preliminary data simulation, which separates the generation of model-derived data to be used in the network training from the DENN training process. We show that at least for the test case of the stochastic R-M system that we considered, this preliminary step results in shortened simulation times without degradation of performance. Moving forward, the demonstrated approach can be used under fairly general conditions: it should provide a useful option for parameter estimation for any model system of differential equations together with stochastic effects that can be simulated to generate values of a finite set of observables of interest, which can be model variables evaluated at specific time points or can be other quantities derived from these variables such as the oscillation period and amplitude that we used, that vary as certain parameter values are adjusted. In our study, for the testing phase, we generated proxy data by simulating our underlying model with fixed parameter values, but of course the data can come from field or laboratory measurements in practice, with knowledge about the noise correlations of the system of interest incorporated into the model used for training. Moreover, our robustness testing suggests that the model can tolerate deviation of the stochastic process assumed for training from the stochasticity present in the underlying physical system, although there is plenty of room for additional exploration of this issue.

One complication in applications of data preprocessing is that the generation of data points over a fine grid of parameter values increases accuracy, yet as the dimension of parameter space increases, the number of grid points needed will grow exponentially. This expansion of the required preprocessing steps will increase the computational time and space requirements for data generation. These issues have been dealt with in other contexts in the literature, including work on preprocessing for other purposes, providing options that could be applied in future work with higher-dimensional parameter spaces. One possibility would be to use a custom-designed non-uniform grid, inserting extra parameter values where the generated data changes rapidly with respect to parameters [27]. A second approach would be to employ Latin Hypercube Sampling (LHS) for pre-generating a limited set of representative parameter points, rather than relying on a fixed grid discretization with uniform spacing. Compared to a fixed grid, LHS ensures broader coverage of the parameter space with fewer points, which becomes particularly advantageous in high-dimensional settings where exhaustive enumeration is computationally expensive [17]. The use of LHS raises a question of how to interpolate for values outside the set of sampled points; specifically, in our context, interpolation would be applied to estimate the $[X_{amp}, Y_{amp}, T]$ values for any selected parameter vector not in the sampled training set. Kriging interpolation, which weights values from the sampled set based on both their distances from the selected vector and the variance among these values as measured along different directions in parameter space, has proved to be an effective strategy to handle this issue [28].

In the training of DENNs, z-score normalization is commonly employed to standardize the training data. This preprocessing step is typically set by default to promote fast and stable convergence and hence to achieve reliable output [29]. Z-score normalization, which adjusts data to have zero mean and unit variance, generally facilitates the learning process by making it less sensitive to the scale of features, thus smoothing the optimization landscape. However, in datasets containing extreme values or significant outliers, z-score normalization can distort the true distribution of the data, potentially leading to misleading training dynamics. For such cases, considering alternative scaling methods like robust scaling, which uses the median and interquartile range, could prove beneficial. These methods are less affected by outliers and may better preserve the original data structure, thereby supporting more accurate model training [30]. The choice of normalization technique can significantly impact the performance of DENNs, especially in applications where outliers represent critical, informative anomalies rather than mere noise, and this issue merits additional exploration in future work.

5. Conclusions

We provide a user-friendly introduction to density estimation neural networks, or DENNs. Although DENNs are not a new development, their algorithmic and implementation complexity often hinder the use of this tool. Through this paper, we hope to provide useful guidance and explanation, and we also offer a way to speed up the DENN implementation when it incorporates data from simulations, based on a preliminary data simulation step. Our example results highlight the accuracy of this efficient approach. Although DENNs come with certain limitations, this study and others suggest that they perform well on a variety of systems. We hope that this work will help to enhance access to DENNs for a range of applications.

Acknowledgments

This work was partially supported by US National Science Foundation award DMS1951095.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Conflict of interest

All authors declare no conflicts of interest in this paper.

References

1. A. F. M. Smith, G. O. Roberts, Bayesian computation via the gibbs sampler and related markov chain monte carlo methods, *J. R. Stat. Soc. B*, **55** (1993), 3–23. https://doi.org/10.1111/j.2517-6161.1993.tb01466.x
2. J. Rothfuss, F. Ferreira, S. Walther, M. Ulrich, Conditional density estimation with neural networks: Best practices and benchmarks, preprint, arXiv:1903.00954.
3. C. M. Bishop, *Mixture Density Networks*, Aston University, WorkingPaper, 1994.
4. J. M. Marin, P. Pudlo, C. P. Robert, R. J. Ryder, Approximate bayesian computational methods, *Stat. Comput.*, **22** (2012), 1167–1180. https://doi.org/10.1007/s11222-011-9288-2
5. G. Papamakarios, D. Sterratt, I. Murray, Sequential neural likelihood: Fast likelihood-free inference with autoregressive flows, in *The 22nd International Conference on Artificial Intelligence and Statistics*, **89** (2019), 837–848. https://doi.org/10.1007/s00104-018-0714-2
6. E. Trentin, Multivariate density estimation with deep neural mixture models, *Neural Process. Lett.*, **55** (2023), 9139–9154. https://doi.org/10.1007/s11063-023-11196-2
7. G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, B. Lakshminarayanan, Normalizing flows for probabilistic modeling and inference, *J. Mach. Learn. Res.*, **22** (2021), 1–64.
8. M. L. Rosenzweig, R. H. MacArthur, Graphical representation and stability conditions of predator-prey interactions, *Am. Nat.*, **97** (1963), 209–223. https://doi.org/10.1086/282272
9. D. Swigon, S. R. Stanhope, S. Zenker, J. E. Rubin, On the importance of the jacobian determinant in parameter inference for random parameter and random measurement error models, *SIAM/ASA J. Uncertainty Quantif.*, **7** (2019), 975–1006. https://doi.org/10.1137/17M1114405
10. S. Kullback, R. Leibler, On information and sufficiency, *Ann. Math. Stat.*, **22** (1951), 79–86. https://doi.org/10.1137/17M1114405

11. D. Rezende, S. Mohamed, Variational inference with normalizing flows, in *Proceedings of Machine Learning Research*, (2015), 1530–1538.

12. I. Kobyzev, S. J. Prince, M. A. Brubaker, Normalizing flows: An introduction and review of current methods, *IEEE Trans. Pattern Anal. Mach. Intell.*, **43** (2021), 3964–3979. https://doi.org/10.1109/TPAMI.2020.2992934

13. C. S. Holling, The functional response of predators to prey density and its role in mimicry and population regulation, *Mem. Entomol. Soc. Can.*, **97** (1965), 5–60. https://doi.org/10.4039/entm9745fv

14. J. E. Rubin, D. J. D. Earn, P. E. Greenwood, T. L. Parsons, T. L. Abbott, Irregular population cycles driven by environmental stochasticity and saddle crawlbys, *Oikos*, **2023** (2023), e09290. https://doi.org/10.1111/oik.09290

15. J. C. Cox, J. E. J. Ingersoll, S. A. Ross, A theory of the term structure of interest rates, *Econometrica*, **53** (1985), 385–407.

16. E. Allen, Environmental variability and mean-reverting processes, *Discrete Contin. Dyn. Syst. Ser. B*, **21** (2016), 2073–2089. https://doi.org/10.3934/dcdsb.2016037

17. F. A. Viana, A tutorial on latin hypercube design of experiments, *Qual. Reliab. Eng. Int.*, **32** (2016), 1975–1985. https://doi.org/10.1002/qre.1924

18. Y. Wen, Z. Li, Y. Xiang, D. Reker, Improving molecular machine learning through adaptive subsampling with active learning, *Digital Discovery*, **2** (2023), 1134–1142. https://doi.org/10.1039/d3dd00037k

19. A. Tejero-Cantero, J. Boelts, M. Deistler, J. M. Lueckmann, C. Durkan, P. J. Gonçalves, et al., sbi: A toolkit for simulation-based inference, *J. Open Source Software*, **5** (2020), 2505. https://doi.org/10.21105/joss.02505

20. J. Boelts, J. M. Lueckmann, R. Gao, J. H. Macke, Flexible and efficient simulation-based inference for models of decision-making, *eLife*, **11** (2022), e77220. https://doi.org/10.7554/eLife.77220

21. D. Reynolds, *Gaussian Mixture Models*, in *Encyclopedia of Biometrics* (eds. S. Z. Li, A. K. Jain), Springer US, Boston, MA, (2015), 827–832. https://doi.org/10.1007/978-1-4899-7488-4_196

22. P. Zhao, L. Lai, Analysis of knn density estimation, *IEEE Trans. Inf. Theory*, **68** (2022), 7971–7982. https://doi.org/10.1109/TIT.2022.3195870

23. D. Peel, G. J. McLachlan, Robust mixture modelling using the t distribution, *Stat. Comput.*, **10** (2000), 339–348. https://doi.org/10.1023/A:1008981510081

24. C. Jin, Y. Zhang, S. Balakrishnan, M. J. Wainwright, M. I. Jordan, Local maxima in the likelihood of gaussian mixture models: Structural results and algorithmic consequences, preprint, arXiv:1609.00978.

25. G. E. Uhlenbeck, L. S. Ornstein, On the theory of the brownian motion, *Phys. Rev.*, **36** (1930), 823–841. https://doi.org/10.1103/PhysRev.36.823

26. X. Zhou, W. Zhang, Z. Chen, S. Diao, T. Zhang, Efficient neural network training via forward and backward propagation sparsification, in *Neural Information Processing Systems*, **36** (2021), 15216–15229.

27. T. Ma, L. Zhang, F. Cao, Y. Ge, A special multigrid strategy on non-uniform grids for solving 3d convection–diffusion problems with boundary/interior layers, *Symmetry*, **13** (2021), 1123. https://doi.org/10.3390/sym13071123

28. P. Kitanidis, *Introduction to Geostatistics: Applications in Hydrogeology*, Cambridge University Press, Cambridge, UK, 1997.

29. D. Singh, B. Singh, Investigating the impact of data normalization on classification performance, *Appl. Soft Comput. J.*, **97** (2019), 105524. https://doi.org/10.1016/j.asoc.2019.105524

30. M. M. Ahsan, M. A. P. Mahmud, P. K. Saha, K. D. Gupta, Z. Siddique, Effect of data scaling methods on machine learning algorithms and model performance, *Technologies*, **9** (2021), 52. https://doi.org/10.3390/technologies9030052