Neural Stochastic Differential Equations for Multistable Dynamical Systems

Linus Heck

Master Thesis at RWTH Aachen University
November 17, 2023

First examiner: Prof. Michael Schaub, Ph.D.
Second examiner: Prof. Dr. Niklas Boers
Thesis advisor: Dr. Maximilian Gelbrecht

Abstract

Paleoclimatologists fit stochastic dynamical models to time series data to uncover fundamental mechanisms of our earth's long-term climate. This process is often manual and involves choosing a specific model framework. The advent of neural stochastic differential equations (neural SDEs) opens up an alternative approach: automatically training a general dynamical system to fit the data's distribution. We demonstrate the feasibility of this approach by successfully training latent neural SDEs to match the statistics of energy balance models and FitzHugh-Nagumo models, multistable systems standard in paleoclimatology. Latent SDEs can directly model tipping points induced deterministically, but we need to introduce another hyperparameter to match stochastic tipping points. We also present the cause of the noise underestimation problem, an open issue concerning latent SDEs for any application. Finally, we discuss the Julia implementation of neural SDEs and find an undocumented trick that has to be implemented to achieve scalable gradients in Julia.

Acknowledgements

As the field of this thesis was entirely new for me, the main reason something decent emerged was Max's continuous support and effort, for which I am deeply grateful. Niklas and Max were extremely supportive when I was trying to figure out my academic career path after my Master's degree. I'm thankful for their patience, sympathy, and support. This project was only possible because Michael took it under his wing, and his project management advice was essential for getting the thesis back on track after I ran into a dead end. Thank you to everyone at TUM and PIK, especially Keno, Alistair, Taka, Philipp, and Christof, whose curiosity and enthusiasm are inspiring and who gave essential feedback for this project. I also want to thank Bernhard for his detailed and valuable feedback. I deeply thank my parents; I could not have pursued a Master's degree without your continuous love, trust, and guidance. Finally, thank you to Karla for all of your love and support, it means the world to me.

Contents

1 Introduction

How does the climate on Earth behave on a macroscopic scale? This question is crucial in investigating the consequences of human-caused global warming. It can be investigated by analyzing data on our planet's climate in the remote past, the paleoclimate. Because nobody was measuring temperature or rainfall until very recently, researchers can only access this data through proxy records like the ratio of certain isotopes in ice and marine cores.

These records show evidence that Earth's climate exhibits nonlinear responses and stochastic forcing [33]. Stochastic dynamical systems are thus the key to modeling Earth's paleoclimate. In the 1970s, researchers such as Ghil [14] began working on single-dimensional multistable stochastic energy balance models, and by the 1980s, paleoclimatologists explored more complex models, coupling multiple systems like temperature and ice sheet equations [33]. By matching specific characteristics of paleoclimate data with the dynamics generated by an idealized dynamical system, researchers attempt to answer questions about the underlying real-world phenomena. Examples of such questions are: *Which mechanism causes the so-called Dansgaard-Oeschger events, rapid climate changes in the last glacial period [5]? Why is there a dominant peak in the records near 100kyr, which cannot be explained by orbital forcing [33]?*

The process of modeling paleoclimate data often involves picking a "framework" for a system, like a Van der Pol oscillator or a FitzHugh-Nagumo model, and then selecting values by metrics that are manually chosen [26]. Riechers et al. list more than thirty such models and sort them into categories [33]. Different models aim to model different specific behaviors of the target system.

The recent emergence of methods that couple stochastic dynamical systems with probabilistic generative machine learning [22, 25] opens up a new approach: what if the model could learn stochastic dynamics from the paleoclimate data by itself, without any expert assessment involved in selecting the underlying model, nor the statistics by which specific values are chosen? In this thesis, we make the first steps towards applying generative neural stochastic differential equations (neural SDEs) to paleoclimate data by applying them to data generated by models.

After discussing the related work in Section 2, we introduce the dynamical systems theory and motivate our research questions in Section 3. In Section 4, we explain generative neural SDEs for readers unfamiliar with either dynamical systems theory or generative machine learning. As our main contribution, we apply latent neural SDEs to data generated by models for paleoclimate systems and comprehensively

analyze the results in Section 5. We draw conclusions about the methods and their applicability. As the underlying data, we use data generated from both a bistable energy balance model and two different FitzHugh-Nagumo models, some of the most commonly used dynamical systems for the paleoclimate [33]. Because we know the underlying mechanisms of these models, we can judge whether the generative machine learning method successfully learns to match them. During this process, we slightly amend the latent neural SDE framework. We compare our results to two baseline methods in Section 6: ARIMA, a stochastic prediction method not based on machine learning, and RNNs, machine learning methods that apply to time series, but are not stochastic.

We also provide an implementation of latent neural SDEs in the Julia programming language, a tool popular with researchers concerned with dynamical systems. In Section 7, we discover a fundamental scalability issue with gradient computation for SDEs in Julia and describe it in detail, along with guidance on how to change the implementation to speed up the computation from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

2 Related Work

This thesis covers two areas of research: dynamical systems theory applied to paleoclimate records and generative machine learning. In the introduction, we have covered some context regarding the dynamical systems theory and will focus on the machine learning approaches in this section. Neural SDEs are probabilistic machine learning methods, a general term for any machine learning method where the model yields a probability distribution. In different applications of the same method, this distribution may have a different meaning to the modeler. Probabilistic machine learning methods are often used for uncertainty quantification within a Bayesian framework.

A standard probabilistic machine learning method is the Gaussian Process (GP) [40]. A GP is a generalized Gaussian distribution over the path space. GPs always have Gaussian marginals, making their distribution easy to reason about (whereas the distribution of an SDE is hard to reason about) but limiting their modeling capacity. As such, they are often used for uncertainty quantification and cannot be used for matching the probability distribution of an arbitrary stochastic time series.

Neural SDEs can also be used for uncertainty quantification. However, we use them for something else: we aim to match the probability distribution of the model to the "distribution of the data". This is the goal of generative machine learning. Genera-

tive machine learning methods add a framework around the generator, in this case, a neural SDE, so that they can learn from data in an unsupervised way. We do not propose a new method but look into applying one of two already existing methods to the problem: SDE-GANs [22] and latent neural SDEs [25]. Neural SDEs can be viewed as an extension of neural ordinary differential equations (neural ODEs) [8], but neural ODEs are not generative or otherwise probabilistic models.

Score-based diffusion models are a recent trend involving stochastic differential equations and generative machine learning [37]. The methods used in these models are interested in a probability distribution induced through the marginals of an SDE at some point. A data sample is an element of the SDE's state space. In contrast, neural SDEs are interested in the probability distribution of an SDE on the path space. A data sample is a path over the SDE's state space.

The main contributions of this thesis are a thorough investigation of applying the latent neural SDE method proposed by Li et al. [25] to datasets similar to paleoclimate data. Other papers extend that same paper. We only mention published papers or papers accepted to workshops, as many preprints exist. In [10], Fagin et al. apply latent SDEs to data from astrophysics. They use them as a drop-in replacement for Gaussian Processes, so the noise is interpreted as uncertainty. Xu et al. consider latent SDEs with a fixed diffusion, a continuous analog of Bayesian recurrent neural networks in [42]. Kidger et al. introduce a reversible Heun solver for differential equations and train a latent neural SDE on an air quality data set from Beijing in [23]. Hasan et al. propose a different latent SDE framework to recover a mapping from one high-dimensional probabilistic space to another in [15].

3 Motivation and Problem Statement

3.1 Stochastic Time Series

In the GRIP and NGRIP projects, researchers have drilled deep ice cores to bedrock in the Greenland ice sheet [1, 20]. Using the $\delta^{18}O$ measure, which is the relative concentration of the ^{18}O and ^{16}O isotopes in the ice at different depths, the NGRIP record reveals the temperatures up to 123,000 years before present as shown in Figure 1 [1]. Thus, the record covers the late Eemian period, the last glacial period, and the Holocene. Surprisingly, it shows large abrupt changes between separate climate conditions with rapid climate warming by 10 Kelvin within decades. These changes are called the Dansgaard-Oeschger events or interstadials. Most of these changes start with an abrupt warming period lasting a few decades and then slowly

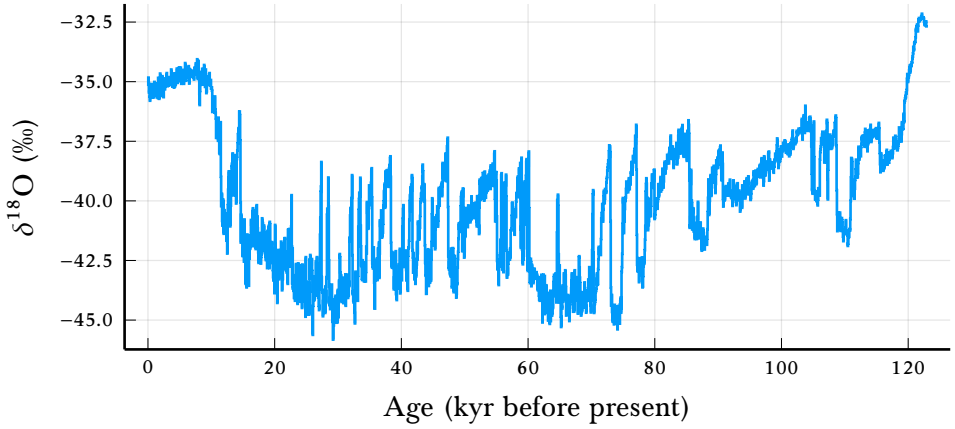


Figure 1: δ^{18}O record from NGRIP [1]

ramp down within 500 to 2000 years [20]. Ice cores drilled in several other places in Greenland agree on the events [1]. There is no consensus on the mechanisms behind the Dansgaard-Oeschger events [26].

Deep ice core records can be viewed as an example of stochastic time series, as they are "driven by noise" in some form. Not only are the records themselves noisy, but the occurrence of an interstadial event can be viewed as some process in the underlying system triggered by chance. Paleoclimatologists view the NGRIP ^{18}O record as one realization of a complex dynamic system that has stochastic properties and compare its statistics to models that exhibit noise-induced tipping [26].

This thesis uses machine learning methods to reconstruct noise-induced tipping behaviors from stochastic time series. We want to automatically synthesize a system with similar stochastic behavior to the data. By some measure, we want a model that could output the given data with some high likelihood. The goal is to understand the mechanisms behind given actual realizations of the stochastic time series.

3.2 Stochastic Differential Equations

A natural way to describe the dynamics of continuous systems is using ordinary differential equations (ODEs) [9, p. 1]. Their stochastic counterparts are stochastic differential equations (SDEs) [35, p. 6]. An ODE characterizes a function $x : \mathbb{R} \to \mathbb{R}^n$ by its rate of change $dx(t)$ at each point t through a drift function $f : \mathbb{R}^{n+1} \to \mathbb{R}^n$:

$$\frac{dx(t)}{dt} = f(x(t), t).$$

To define SDEs, we want to "add noise" to the ODE. The most commonly used

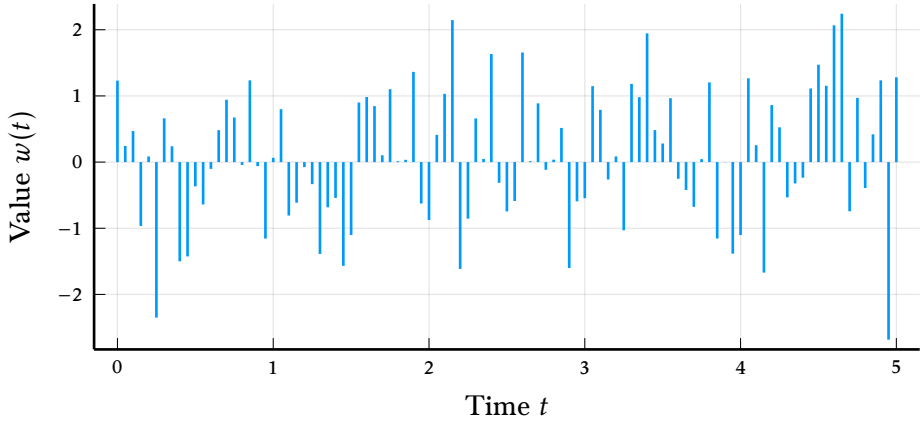


Figure 2: Discretized Gaussian white noise with expectation zero and unit variance

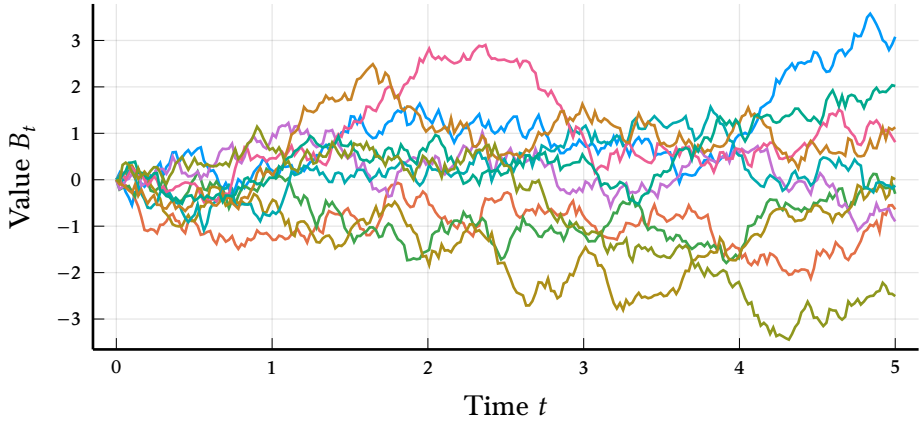


Figure 3: Multiple trajectories of Brownian motion with unit variance

form of noise in SDEs is the white noise process $w(t)$, which maps all time points t to independent Gaussians with an expectation of zero and a given constant variance [35, p. 35]. A realization of discretized white noise is shown in Figure 2. While independent Gaussian disturbances are a natural definition of noise, the white noise process is discontinuous almost everywhere, which becomes a problem once we want to integrate it over time. To solve an SDE, we want to do just that: integrate a differential equation of the form

$$\frac{dx(t)}{dt} = f(x(t), t) + g(x(t), t)w(t)$$

where the drift f expresses the deterministic term, and the diffusion g expresses the stochastic term that is multiplied with the white noise.

Due to the discontinuous nature of white noise, the usual definitions of integration do not work in this context. There is a solution, which involves two tricks: first, to

define *Brownian motion* B_t as a process that can be thought of as the integral of white noise over time, and second, to work out a way to integrate over Brownian motion specifically.

Each time *increment* $\Delta B_t = B_{t+\Delta t} - B_t$ of Brownian motion is a random variable with a Gaussian distribution that has zero mean and a variance proportional to Δt. If two timespans do not overlap, their increments are independent. While white noise is a stationary process, Brownian motion is a "random walk" starting at zero [35, p. 44]. Trajectories of Brownian motion are shown in Figure 3. The Brownian motion process could count as an integral of white noise [35, p. 25], and with caution that it is not really an integral, we will write:

$$\frac{dB_t}{dt} = w(t).$$

Using this equality, we can rewrite the above SDE to express the diffusion term over Brownian motion instead of white noise:

$$dx(t) = f(x(t), t)dt + g(x(t), t)dB_t.$$

To integrate this SDE, we need to do additional work. The usual definitions of integrals still do not work here, as Brownian motion is almost nowhere differentiable. The limit

$$\int_{t_0}^{t} g(x(t), t)dB_t = \lim_{n\to\infty} \sum_{\substack{t_k = t_0 + \frac{t-t_0}{n}, \\ k\in\{0,\dots,n-1\}}} g(x(t_k^*), t_k^*)(B_{t_{k+1}} - B_{t_k})$$

would have to converge to the same value for every $t_k^* \in [t_k, t_{k+1}]$, which is not the case for integration over Brownian motion. The solution is to fix t_k^*: The Îto integral chooses the left endpoint $t_k^* = t_k$, while the Stratonovich integral chooses the midpoint $t_k^* = \frac{t_k + t_{k+1}}{2}$. Choosing a different fixed t_k^* is also possible, but uncommon [29, p. 24]. The Îto and Stratonovich interpretations arrive at different solutions to the integral and have their distinct advantages and disadvantages [35, p. 35]. However, SDEs can be converted between their Îto and Stratonovich formulations [21, p. 74].

As an example of an SDE, consider the Ornstein-Uhlenbeck process

$$\begin{aligned} x(0) &= x_0, \\ dx(t) &= -\kappa x(t)dt + \sigma dB_t. \end{aligned}$$

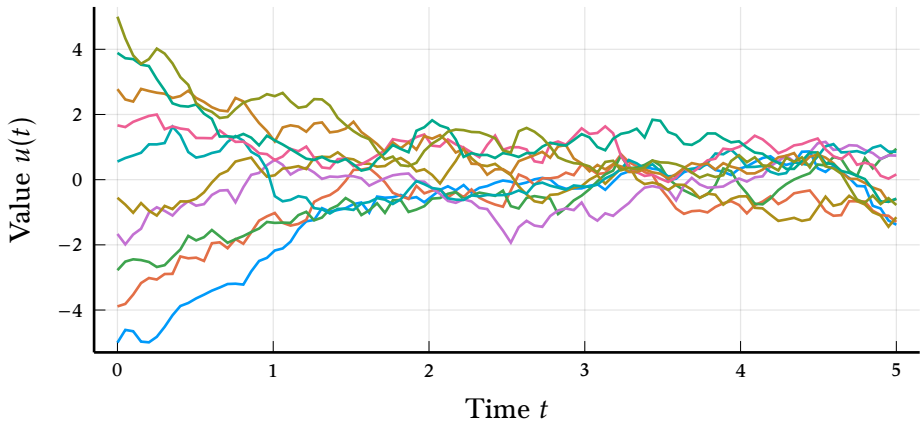


Figure 4: Trajectories of an Ornstein-Uhlenbeck process with $\kappa = \sigma = 1$

A few trajectories with $\kappa = \sigma = 1$ are integrated in Figure 4. This SDE has an analytical solution [35, p. 51]:

$$x(t) = \exp(-\kappa t) \cdot x_0 + \sigma \int_0^t \exp(-\kappa t) dB_t.$$

Most SDEs do not have an analytical solution, so they are usually simulated numerically to approximate the solution. An SDE solver is an algorithm that simulates an SDE in discrete timesteps Δt [cf. 35, p. 16]:

$$\begin{aligned} x(0) &= x_0, \\ x(t + \Delta t) &= x(t) + \int_t^{t+\Delta t} f(x(t), t) dt + \int_t^{t+\Delta t} g(x(t), t) dB_t. \end{aligned}$$

Different SDE solvers find different approximations for the integrals on the right-hand side. The simplest SDE solver is the Euler-Maruyama method which chooses

$$\begin{aligned} \int_t^{t+\Delta t} f(x(t), t) dt &\approx f(x(t), t) \Delta t, \\ \int_t^{t+\Delta t} g(x(t), t) dB_t &\approx g(x(t), t) \Delta \beta_t. \end{aligned}$$

where $\beta_t \sim \mathcal{N}(0, Q\Delta t)$ for a variance matrix Q [35, p. 132], and so the discretized algorithm to compute a solution becomes

$$\begin{aligned} x(0) &= x_0, \\ x(t + \Delta t) &= x(t) + f(x(t), t) \Delta t + g(x(t), t) \Delta \beta_t. \end{aligned}$$

The Euler-Maruyama method solves SDEs in the Îto sense as it chooses the beginning of the interval $[t, t + \Delta t]$ for the value of the diffusion at point t.

We have presented all the necessary theory for SDEs, so we can now use them to model bifurcations and noise-induced tipping in the following section.

3.3 Bifurcations and Noise-Induced Tipping

Bifurcations in feedback mechanisms like differential equations are the tool for modeling sudden changes in climate systems [3, 12]. A fundamental concept is *noise-induced tipping* in models that have a slow component (the climate, i.e., the long-term drift behavior) and a fast component (the weather, i.e., the short-term randomness from the diffusion, or a different dimension in the state space). The slow component exhibits *attractors*, which are points that attract trajectories in the area and make them converge to a fixed point, a limit cycle, or similar. At a tipping point, the fast component drives a sudden change in the slow component by escaping from an attractor [3]. Ashwin et al. and Bódai et al. refer to this as N-tipping in a closed system [3, 4]. In chaotic models, state space dimensions can also be fast components [33].

To illustrate noise-induced tipping, we create a simple process that tips back and forth between two attractors. We first choose an ODE with two attractors (the slow component) and then add noise (the fast component). For the ODE, we choose a zero-dimensional energy balance model for the Earth's temperature. A zero-dimensional model views the Earth as a single point with an average temperature and balances the incoming energy from the sun and the outgoing energy through radiation. Similar models can be found in [12], [14], and [38]. The change of the temperature is the difference between the incoming and outgoing energy

$$\frac{dT(t)}{dt} = E_{in}(T(t)) - E_{out}(T(t)).$$

The outgoing energy is given by the Stefan-Boltzmann law of black-body radiation

$$E_{out}(T(t)) = \varepsilon\sigma T(t)^4$$

with the Stefan-Boltzmann constant $\sigma = 5.67 \cdot 10^{-8} W m^{-2} K^{-4}$. We choose $\varepsilon = 0.6$ because the Earth is not a perfect black body. We model the incoming energy (technically an intensity) as

$$E_{in}(T(t)) = \left(1 - \left(\alpha_0 - \frac{\alpha_1}{2}\tanh(T(t) - \hat{T})\right)\right)\frac{S_0}{4}.$$

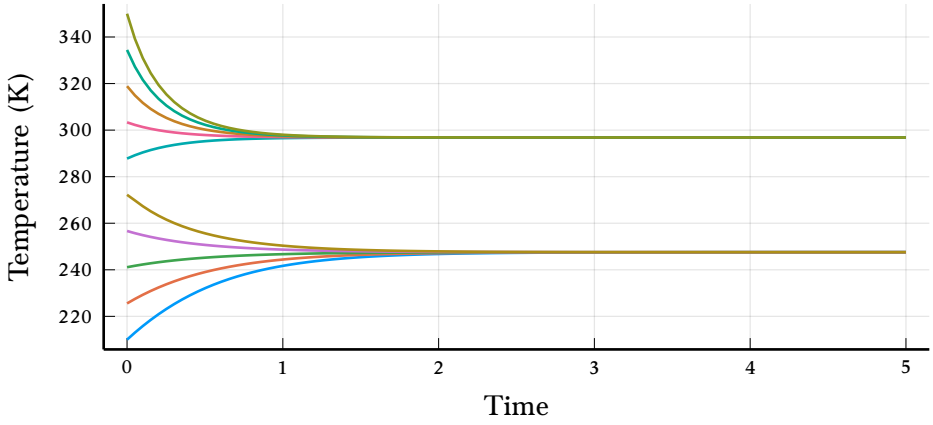


Figure 5: The zero-dimensional energy balance model

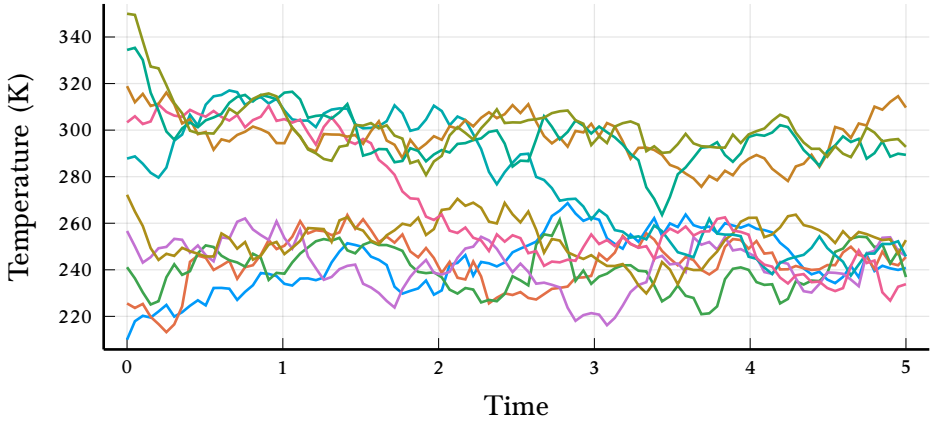


Figure 6: The zero-dimensional energy balance model with noise

$S_0 = 1636 W m^{-2}$ is the solar constant; it is divided by four because the Earth emits energy on all sides, whereas it only receives it on one side. The rest of the expression models the surface's albedo at that temperature, as snow has a higher albedo than soil. We choose $\alpha_0 = 0.425$, $\alpha_1 = 0.4$, and $\hat{T} = 273K$. These constants are similar to those found in the literature [12, 14, 38]. We have slightly increased the α values to yield closer attractors for more frequent tipping behavior.

We integrate a few trajectories of this model in Figure 5. There are two stable fixed points: the upper fixed point represents the current climate on Earth, whereas the lower fixed point represents the "snowball Earth" state. To perturb our energy balance model, we add a constant diffusion

$$dT(t) = (E_{in}(T(t)) - E_{out}(T(t)))dt + 40 dB_t$$

and plot trajectories in Figure 6. The added noise could represent random weather

fluctuations that sometimes make the system escape one attractor and tip to the other.

We mainly choose this model to give a simple example of a stochastic process with noise-induced tipping. Tipping is extremely rare in a more realistic zero-dimensional energy balance model. Sutera analyzes a similar model and finds realistic values [38].

3.4 Coarse Problem Statement

We have introduced the relevant theory for SDEs and constructed a toy zero-dimensional energy balance model that exhibits noise-induced tipping. This thesis aims to fit SDEs that can match the tipping behaviors exhibited by input data, such as data generated by the energy balance model. We want to avoid using expert assessment and manual selection techniques for this. The neural SDE is a method that seems suitable for this goal. Apart from the manual estimation of SDEs, possible alternatives include autoregressive stochastic models like ARIMA and recurrent neural networks, which we discuss in Section 6. Our research questions for neural SDEs are:

1. Can neural SDEs replicate the statistics of multistable systems? Can they capture multistability and perform well in summary statistics?
2. How do neural SDEs behave, in general, in relation to their input data set? Can they match both the drift and diffusion components of a data set?

We elaborate on these questions in Section 5.1, but before that, we explore the theory of neural SDEs and select a type of neural SDE.

4 Neural Stochastic Differential Equations

4.1 Introduction

Neural differential equations are differential equations with a neural network on the right-hand side. The simplest type of neural differential equation is a neural ordinary differential equation (neural ODE), first introduced by Chen et al. [8]:

$$\begin{aligned} u(0) &= u_0, \\ \frac{du(t)}{dt} &= f_\theta(u(t), t). \end{aligned}$$

f_θ can be any Lipschitz neural network and is usually just feedforward [21, p. 22].

A feedforward neural network [27, p. 527] is a function $f_\theta(x)$ with parameters θ and input x that is composed of layers $d \in \{1, 2, \dots, D\}$ of the form

$$a_j^{(d)} = f^{(d)}\left(\sum_l w_{j,l}^{(d)} \cdot a_j^{(d-1)} + b_j^{(d)}\right),$$

where

- $a_j^{(d)}$ are the jth elements of information at layer d,
- $a_j^{(0)} = x_{\text{in}}$ is the input layer and $a_j^{(D)}$ is the output layer,
- $w_{j,l}^{(d)}$ and $b_j^{(d)}$ are parameters from θ,
- $f^{(d)}$ is the activation function for layer d.

The gradients of the output of a neural network w.r.t. the parameters can be computed efficiently using the backpropagation algorithm [34]. Neural networks with nonlinear activation functions are universal approximators, meaning that they can learn and represent any measurable real function [18].

A neural SDE [25, 39] is a stochastic differential equation with neural networks in the place of the drift and diffusion functions:

$$\begin{aligned} u(0) &= i_\theta(v), \\ du(t) &= f_\theta(u(t), t)dt + g_\theta(u(t), t)dB_t. \end{aligned}$$

Here, $f_\theta : \mathbb{R}^{d_u} \times [0, T] \to \mathbb{R}^{d_u}$ is the drift, $g_\theta : \mathbb{R}^{d_u} \times [0, T] \to \mathbb{R}^{d_u \times d_w}$ the diffusion for d_w-dimensional Brownian motion B_t, and $i_\theta : \mathbb{R} \to \mathbb{R}^{d_u}$ represents the initial condition given a random value $v \sim \mathcal{N}(0, 1)$. We write that f, g, i are all parametrized by θ, though usually they are separate neural networks with their exclusive parameters from θ.

Given parameters θ, a neural SDE induces the stochastic process $\{Z_t\}_{t \in \mathcal{T}}$ with

$$Z_T = i_\theta(v) + \int_0^T f_\theta(u(t), t)dt + \sum_{i=0}^{m} \int_0^T g_\theta^{(i)}(u(t), t)dB_t^{(i)}.$$

This stochastic process induces a probability distribution over the path space. The distribution can be sampled from using SDE solvers but cannot be easily reasoned about in succinct mathematical form [25], as the SDE has no analytic solution. In this sense, a neural SDE is a generative model [21, p. 75]: Like a generative neural network that induces a probability distribution on images, a neural SDE induces one on paths. We will first present how to train neural SDEs by computing gradients and

then discuss leveraging this generative property to build a model that matches given probability distributions over time series.

4.2 Training Neural SDEs

Neural SDEs are trained by gradient descent, the standard method for training machine learning methods based on neural networks. To use it, we need to compute the gradients of a loss function w.r.t. the parameters of the neural networks inside the neural SDEs. These gradients are also called the *sensitivities* in differential equation parlance [36, p. 31]. There are three main ways of computing gradients:

Finite differences. The finite difference method *estimates* gradients by varying parameters slightly and computing the differences of the output values. Thus, it treats the methods (like SDEs) as a blackbox. Computing finite differences is generally applicable, but the results exhibit significant systematic and implementation-related errors [36, p. 31].

Automatic differentiation through an SDE solver. Also called "discretize-then-optimize" [23], this method treats the invocation of the SDE solver as an operation like any other program, which is differentiated using an automatic differentiation (AD) framework. An AD framework takes as input program code that computes a function $f_\theta(x)$, parametrized by θ, and automatically computes the derivatives $\frac{\partial f}{\partial \theta}(x)$. This approach is commonly used in financial applications of SDEs and is simple to implement when the practitioner has control over the source code of the solver. It usually comes with a considerable memory requirement because the program execution needs to be stored on a so-called tape when doing reverse-mode AD [25].

Continuous adjoints. This method, also called "optimize-then-discretize" [23], is specific to differential equations. It computes the gradients of an SDE by solving a different SDE, the adjoint SDE, in reverse. The adjoint SDE's state space contains the vector-Jacobian products of the original SDE at each time point. An SDE in Stratonovich formulation can be reversed by storing the Brownian motion of the trajectory. Memory-efficient approaches to store the Brownian motion are introduced by Li et al. alongside the method, resulting in a logarithmic memory cost [25]. The name "optimize-then-discretize" is not entirely accurate, as the adjoint SDE is still discretized to compute the gradients that are used to optimize afterward. It only means that a continuous characterization of the gradients exists through the adjoint SDE. We explain this method in detail in Section 7.1.

4.3 Generative SDEs

Comparing the probability distribution of a data set to an SDE is intractable: finding the probability distribution of the data would mean we have already solved the problem. Even the probability distribution of a general neural SDE is intractable (there is no closed form), as we can only sample from the distribution using an SDE solver. Generative machine learning algorithms have been considered to match the distributions and fit an SDE to data. Two major forms of generative models for SDEs have been considered: latent neural SDEs and SDE-GANs. We will first consider latent neural SDEs and, afterward, SDE-GANs.

4.3.1 Latent Neural SDEs

Latent neural SDEs [25] [see also 21, pp. 82–84] are generative models that aim to reproduce the probability distribution of a data set. A latent neural SDE consists of two neural SDEs: the posterior and the prior. We start with an informal explanation of both SDEs and precisely define the model later. We also refer to latent neural SDEs as latent SDEs.

Posterior SDE. The drift of the posterior SDE is parameterized by context, which is time-dependent information that is computed using an input trajectory from the data set. We will discuss how we compute the context later. First, we view it as some external input representing the data. In general, we can write the posterior SDE as

$$du(t) = h_{\theta \cup \phi}(u(t), t)dt + g_{\theta}(u(t), t)dB_t \qquad \text{(posterior)}$$

where θ are the latent neural SDE's parameters, and ϕ is the context. Given ϕ, the posterior SDE induces a probability distribution that aims to reconstruct and trace the data. We illustrate this in Figure 7. In training, we draw samples from this distribution and minimize their distance to the input data, corresponding to maximizing a likelihood in an observation model.

To minimize the distance between the data and the posterior, we can maximize the likelihood of an observation model we define around the data. For instance, suppose we have data x_{t_i} and a run of the posterior returns the path z_{t_i}. As an observation model, we define a distribution $\mathcal{N}(z_{t_i}, \sigma^2)$ for a fixed variance σ^2. We write $p(x_{t_i}|z_{t_i})$ for the likelihood of the observations given the model's output. For the entire path,

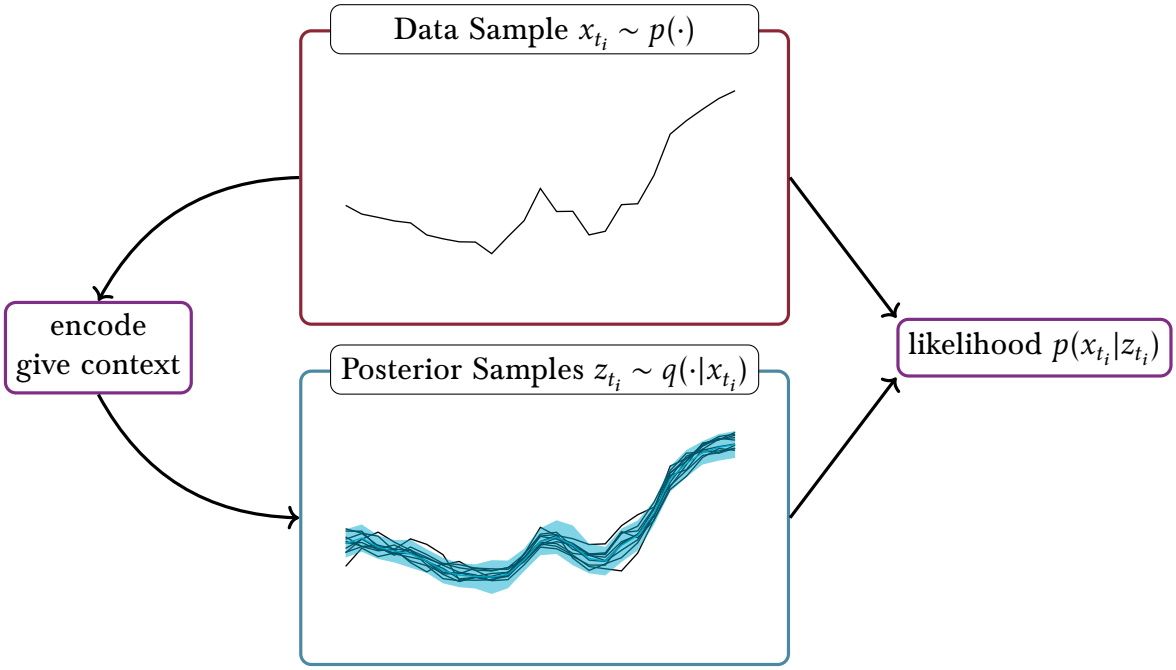


Figure 7: The data sample is encoded and given as the context ϕ to the posterior SDE, which yields a probability distribution. The likelihood of the data given realizations of the distribution is maximized in training.

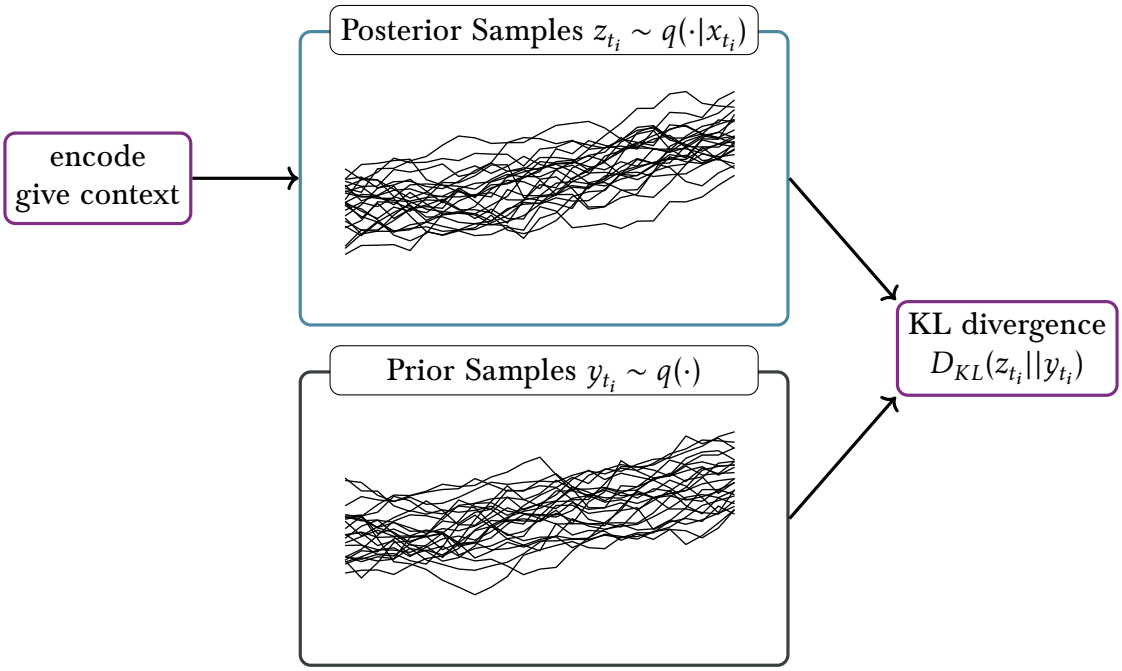


Figure 8: The distance between the posterior SDE and the prior SDE is quantified by the KL divergence, which is minimized in training.

it makes sense to quantify this as a log-likelihood (like [25], but as an integral):

$$\mathscr{L}_E = \int_0^T \log p(x_t|z_t)dt.$$

Alternatively, but not equivalently, we can minimize a squared distance between the model and data [21, p. 83]:

$$\mathscr{L}_E = \int_0^T (x_t - z_t)^2 dt.$$

The posterior alone cannot model the data's probability distribution: it always needs data as input (through the context) and thus is not a generative model alone. Additionally, if we were to train the posterior alone, its diffusion would eventually be zero, as the posterior would repeat the data we input.

Prior SDE. The prior SDE aims to model the data distribution without further input. Its general form is

$$d\tilde{u}(t) = f_\theta(\tilde{u}(t), t)dt + g_\theta(\tilde{u}(t), t)dB_t \qquad \text{(prior)}$$

which lacks the context ϕ. Note that while its drift is different from the posterior, the prior and posterior share the same diffusion. Because of this, we can quantify the *Kullback-Leibler (KL) divergence* between both SDEs, which is a measure of the difference between their probability distributions [25, 39]. Note that this is the KL divergence of the distributions over the path space, not the KL divergence over the marginals.

We can minimize the KL divergence while training to align the prior to the posterior, as illustrated in Figure 8. When we minimize the KL divergence between the prior and posterior, the posterior influences the prior to match the data distribution, and the prior also influences the posterior to prevent its diffusion from converging to zero.

We train the posterior and prior SDEs in parallel and minimize the KL divergence while maximizing the log-likelihood. After training, the posterior SDE is supposed to fit the data, while the prior SDE is supposed to fit the posterior SDE. Thus, the posterior SDE is used "as a bridge" between the prior SDE and the data set to align the prior SDE's distribution to the distribution of the data set.

Encoder and Projector. The encoder transforms a given path from data into the context ϕ; the projector transforms a path sampled from the SDEs back into data space. Theoretically, the encoder is unnecessary, as we could directly set $\phi = x_{t_i}$. However, choosing the encoder as a recurrent neural network that has seen future observations in the data is helpful, and the literature always does this [21, 25]. The intuitive advantage of an encoder is that the posterior SDE can look as far into the future as it wants to.

An RNN-based encoder works like this: suppose we are given the regularly-sampled time series $x_{t_1}, \dots, x_{t_N}$. We run the encoder on the time series in reverse, which outputs the context $a_{t_1}, \dots, a_{t_N}$. This means the output a_{t_i} depends on the input values $x_{t_i}, x_{t_{i+1}}, \dots, x_{t_N}$. To integrate the posterior SDE based on the data x, at time $t \in [t_i, t_{i+1})$, we set $\phi = a_{t_i}$.

We have yet to mention that the latent space (containing y_{t_i} and z_{t_i}) and data space (containing x_{t_i}) might differ. For instance, the latent space could have more dimensions than the data space. In this case, we need a projector in the architecture. We will choose it as a fixed vector that selects the first dimension of the latent space, whereas Li et al. choose it as a trainable linear combination from the latent dimensions [25]. We discuss these choices in the experimental section.

Training Objective. With prior and posterior as above, the KL divergence is

$$\mathscr{L}_{KL} = D_{KL}(\mu_q || \mu_p) = \mathbb{E}_{B_t} \int_0^T \frac{1}{2} \left\| \frac{h_{\theta \cup \phi}(u(t), t) - f_\theta(u(t), t)}{g_\theta(u(t), t)} \right\|_2^2 dt.$$

Here, μ_q is the probability distribution of the posterior, and μ_p of the prior SDE, and we integrate along a trajectory $u(t)$ of the posterior along Brownian motion B_t [25] [21, p. 83].

We train the distance between data and posterior and the distance between prior and posterior together, yielding the objective

$$\max_{\theta, \phi} \mathbb{E}_{x_{\text{data}}}(\mathscr{L}_E - \mathscr{L}_{KL}).$$

This objective can be interpreted as a so-called evidence lower bound (ELBO) from variational inference, making a latent neural SDE a Bayesian variational autoencoder as first proposed by Kingma and Welling [24, 25]. The names "prior" and "posterior" come from the variational inference framework. In this context, the posterior is not the actual posterior; it is only the approximate posterior, but we call it the posterior

as that is an unwieldy name.

To have more control over the model, it is desirable to weigh the KL-divergence using a tunable factor β [16, 25]:

$$\max_{\theta,\phi} \mathbb{E}_{x_{\text{data}}}(\mathscr{L}_E - \beta\mathscr{L}_{KL}).$$

We explore the selection of this hyperparameter in Section 5.

4.3.2 SDE-GANs

Because latent neural SDEs are the SDE analog of variational autoencoders, it is natural to conceive of SDE analogs of other generative machine learning methods. The most successful generative method is the Generative Adversarial Network (GAN) [6], and Kidger et al. propose the SDE-GAN [22]. We keep this introduction shorter because we do not work with SDE-GANs in this thesis (see Section 4.4 for the reasons).

GANs also consist of two components: the generator and the discriminator. The generator generates random samples from its distribution, while the discriminator takes samples as input and outputs a yes/no verdict. The discriminator is trained to differentiate samples from the data set from samples that were output by the generator. The generator is trained to trick the discriminator, making it as hard as possible for the discriminator to differentiate its output from actual data. These objectives give a min-max loss function [6]. Wasserstein GANs are a modification to classic GANs that have a loss function that is more interpretable and suffers less from training instability [2].

SDE-GANs follow the Wasserstein GAN architecture. The generator is a neural SDE, the target SDE we want to fit to the data. The discriminator could be chosen as multiple architectures; Kidger et al. choose it to be a neural controlled differential equation (neural CDE), which intuitively is a neural SDE that can take a sample as an input and come to a binary conclusion [22].

4.4 Which Generative Method to Investigate?

To fit multistable systems, we could investigate latent neural SDEs or SDE-GANs. According to Patrick Kidger, a co-author of the SDE-GAN paper, SDE-GANs have higher modeling capacity but are more challenging to train than latent SDEs [21, p. 84]. For classic GANs, a lot of the engineering details have been figured out, but

for SDE-GANs, most of this is still open, making SDE-GANs nontrivial to apply. In particular, finding a balance between generator and discriminator that leads to satisfactory results is hard [21, p. 85].

Latent SDEs are comparatively easy to train [21, p. 85], but the question of their modeling capacity is not too well investigated, as to the best of our knowledge, there are only a handful of actual experiments that are roughly similar to our problem statement in the literature (in [25], [21, pp. 89–92], and [23]). None of these experiments has a detailed evaluation, which is a good argument to use latent SDEs for this thesis: we mainly want to investigate the modeling capacity. If our experiments are successful enough for latent SDEs, there is good reason to believe that SDE-GANs, with a higher modeling capacity, will also be able to learn the same systems. Using latent SDEs makes our results less dependent on "wizardry" and gives us fewer hyperparameters to explore more exhaustively.

5 Experiments and Analysis

5.1 Problem Statement for Experimental Section

Having chosen latent SDEs as the model to investigate, the primary questions for the experimental section of this thesis, amended from Section 3.4, are the following:

1. Can neural SDEs replicate the statistics of multistable systems? Can they capture multistability and perform well in summary statistics?
2. How do *latent* SDEs behave, in general, in relation to their input data set? Can they match both the drift and diffusion components of a data set?

The first question is the primary endeavor of the thesis. However, the second question is also open: One of the few applications of latent SDEs in the literature is [25], where the authors train a latent SDE on a geometric Brownian motion data set. They note that the model underestimates the diffusion size but do not investigate or explain this effect. We also provide answers for this in this section.

Conveniently, both questions go hand-in-hand. A bistable system's marginals and tipping rate are ideal for quantifying whether a given data set's drift and diffusion components match the trained latent SDE. After introducing how we train latent SDEs using a time-dependent Ornstein-Uhlenbeck process as an example, we use two multistable data sets for the experiments: a simple one-dimensional non-chaotic system and a more involved two-dimensional chaotic system.

Hyperparameter	Value
Size of data set	1024
Batch size	1024
Latent dimensions	2
Observation model	$\mathcal{N}(z_t, 0.01)$
GD algorithm	ADAM
Learning rate	0.01
Learning rate decay	0.9995
Timespan	$[0, 50]$
Δt	0.25
Solver steps	200
Solver	Euler-Heun
Backpropagation	through solver

Component	Architecture
Encoder	GRU (hidden 64, output 32)
Prior drift	$2 \xrightarrow{f} 64 \xrightarrow{f} 64 \xrightarrow{f} 2$
Posterior drift	$2 + 32 \xrightarrow{f} 64 \xrightarrow{f} 64 \xrightarrow{f} 2$
Diffusion	$2 \xrightarrow{f} 64 \xrightarrow{g} 2 \xrightarrow{f} 2$
Projector	$(1, 0)^T$
Posterior initial conditions	$32 \xrightarrow{id} \mathcal{N}(\mu, \sigma)$
Prior initial conditions	$\mathcal{N}(\mu, \sigma)$

Table 1: Hyperparameters and architecture for the Ornstein-Uhlenbeck experiment. Above the arrows are the activation functions: f is softplus, g is sigmoid, id is identity.

The reader can replicate all experiments using the `multistablesde` package using the included `slurm` array jobs. See Section B for the link to the code archive.

5.2 Ornstein-Uhlenbeck Process

To introduce the general setup, we train a latent neural SDE on a time-dependent Ornstein-Uhlenbeck process with the equation:

$$du(t) = (0.02t - 0.1u(t))dt + 0.3dB_t.$$

This example is from [21, p. 199], but we have increased the diffusion. We use the setup in Table 1, which consists of default settings already used in [25], except for these choices:

- We use a two-dimensional latent space for data generated from a model with a two-dimensional state space (the actual state space plus the time). In contrast, Li et al. use a four-dimensional latent space for data generated from a model with a two-dimensional state space [25]. They use a trainable projector, whereas we use a projector that selects the first dimension of the SDE. We comment on this choice and higher-dimensional latent spaces in Section 5.3.6.
- Li et al. use a final sigmoid activation in the diffusion [25]. If set to that architecture, the sigmoid limits its output to a maximum value of one, which it will fully maximize in our experiments. To remove the limit on the diffusion's value but to keep the sigmoid's limiting effect, we add another layer with a final softplus activation to keep the output positive. We also attempted to leave out the sigmoid activation altogether, but this led to instabilities while training.

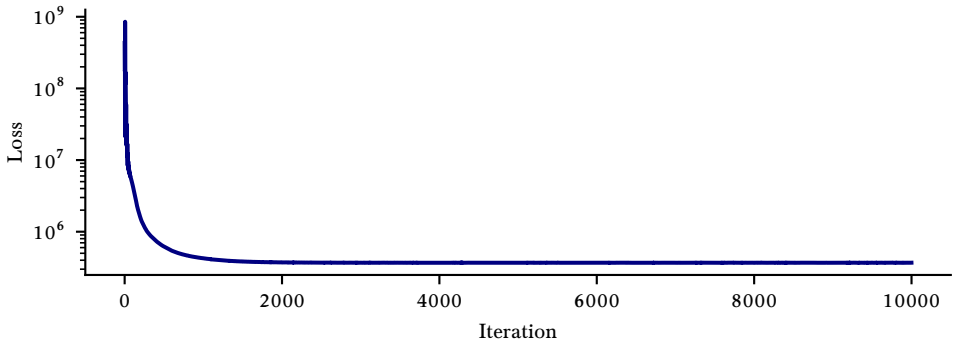


Figure 9: OU: Loss in training for $\beta = 10$

- We set the timespan to a window where the data dynamics are visible.
- We choose a Stratonovich SDE solver, not an Îto SDE solver—this is more efficient when using the continuous adjoint method for computing gradients (even though we do not use it here). The models are Îto, and we account for the correction when comparing them directly later.
- We use a larger learning rate decay factor, yielding a slower decay, and train the model with more epochs. The larger number of epochs is unnecessary for the Ornstein-Uhlenbeck process but is necessary for the other models, so we keep the number of epochs the same for consistency.
- We always use a hidden layer size of 64 instead of 100, as in [25]. This value was arbitrarily set initially and never changed; we did not explore any architectural details of the neural networks. The underlying formulas of our models are simple, so their approximation turned out to be simple for small neural networks.
- We do not corrupt or perturb the data, although we still use an observation model based on a log-likelihood. It is out of the scope of this experiment to answer whether latent SDEs can reconstruct dynamics from data that was noisily measured. The log-likelihood model worked better than squared distances, and we chose a variance of 0.01 to match Li et al. [25]. Changing the variance will impact the values for β.

As in [25], the prior and posterior SDEs are time-independent, so they do not receive the current time t as input, only the current position in state space $u(t)$. The encoder is also time-independent, so it does not receive time information alongside the data.

The main hyperparameter to vary in a latent SDE is the KL-weighting factor β. We sweep β from 0.01 to 10000 in increments of factor 10. We train for 10000 epochs and use a linear KL-annealing schedule [13] that ramps up beta from zero to its final value in 1000 epochs. We performed informal experiments without a KL scheduler

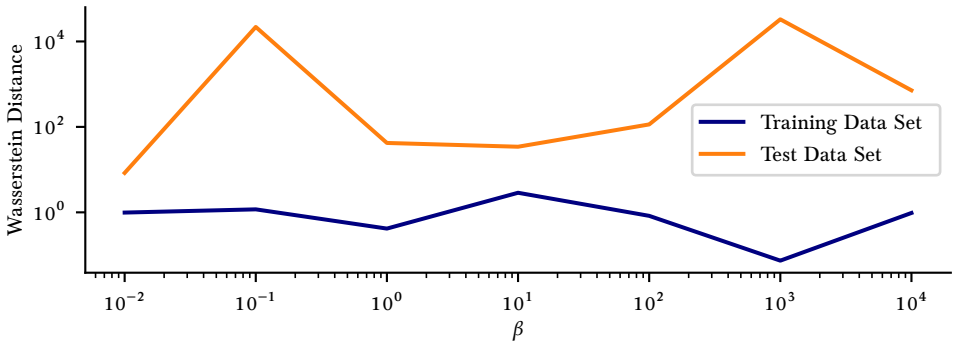
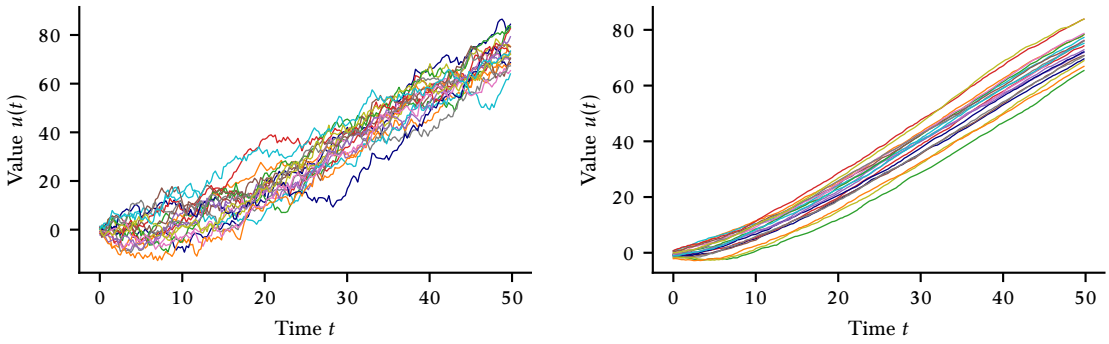


Figure 10: OU: Wasserstein distance for β



(a) OU Process, $[0, t_{\text{train}})$

(b) Prior, $\beta = 10$, $[0, t_{\text{train}})$

Figure 11: Ornstein-Uhlenbeck process and learned prior

and found that training was less stable. Training losses for each epoch (the same as an iteration because we use the entire data in every batch) are shown in Figure 9.

The model trains on the timespan $[0, t_{\text{train}})$, which can be different for each model. We set $t_{\text{train}} = 50$ for the Ornstein-Uhlenbeck process. To evaluate the statistics of the trained model, we use $[0.5t_{\text{train}}, t_{\text{train}})$ as the train set and $[t_{\text{train}}, 5t_{\text{train}})$ as the test set because we are less interested in the evolution from the Gaussian initial state and more interested in the long-run behavior of the system. We also show plots where we consider the first half of the training timespan.

We compute the marginals as histograms for the prior SDEs and the data. To quantify how close these are numerically, we use the *Wasserstein distance*, which intuitively represents the cost of turning one pile into another by moving the ground [31]. The Wasserstein distances of the train and test set on the Ornstein-Uhlenbeck data set, depending on the hyperparameter β, are shown in Figure 10.

We plot the prior for $\beta = 10$ on the training timespan in Figure 11 and note that while it follows the dynamics, it seems smooth compared to the data, i.e., it trains to have a smaller diffusion. Underestimation of the diffusion size is a known problem

Hyperparameter	Value
Size of data set	1024
Batch size	1024
Latent dimensions	1
Observation model	$\mathcal{N}(z_t, 0.01)$
GD algorithm	ADAM
Learning rate	0.01
Learning rate decay	0.9995
Timespan	$[0, 4]$
Δt	0.01
Solver steps	400
Solver	Euler-Heun
Backpropagation	through solver

Component	Architecture
Encoder	GRU (hidden 64, output 32)
Prior drift	$1 \xrightarrow{f} 64 \xrightarrow{f} 64 \xrightarrow{f} 1$
Posterior drift	$1 + 32 \xrightarrow{f} 64 \xrightarrow{f} 64 \xrightarrow{f} 1$
Diffusion	$1 \xrightarrow{f} 64 \xrightarrow{g} 1 \xrightarrow{f} 1$
Projector	none
Posterior initial conditions	$32 \xrightarrow{id} \mathcal{N}(\mu, \sigma)$
Prior initial conditions	$\mathcal{N}(\mu, \sigma)$

Table 2: Hyperparameters and architecture of the energy balance model experiments. Above the arrows are the activation functions: f is softplus, g is sigmoid, id is identity.

of latent SDEs, mentioned by Li et al. [25] and subsequently discussed in GitHub issues of packages that implement them. However, there is no answer to why the model behaves this way. In the next section, we investigate this effect by measuring the tipping rate of a bistable energy balance model.

5.3 Stochastic Zero-Dimensional Energy Balance Model

5.3.1 Exploration of Values for Beta

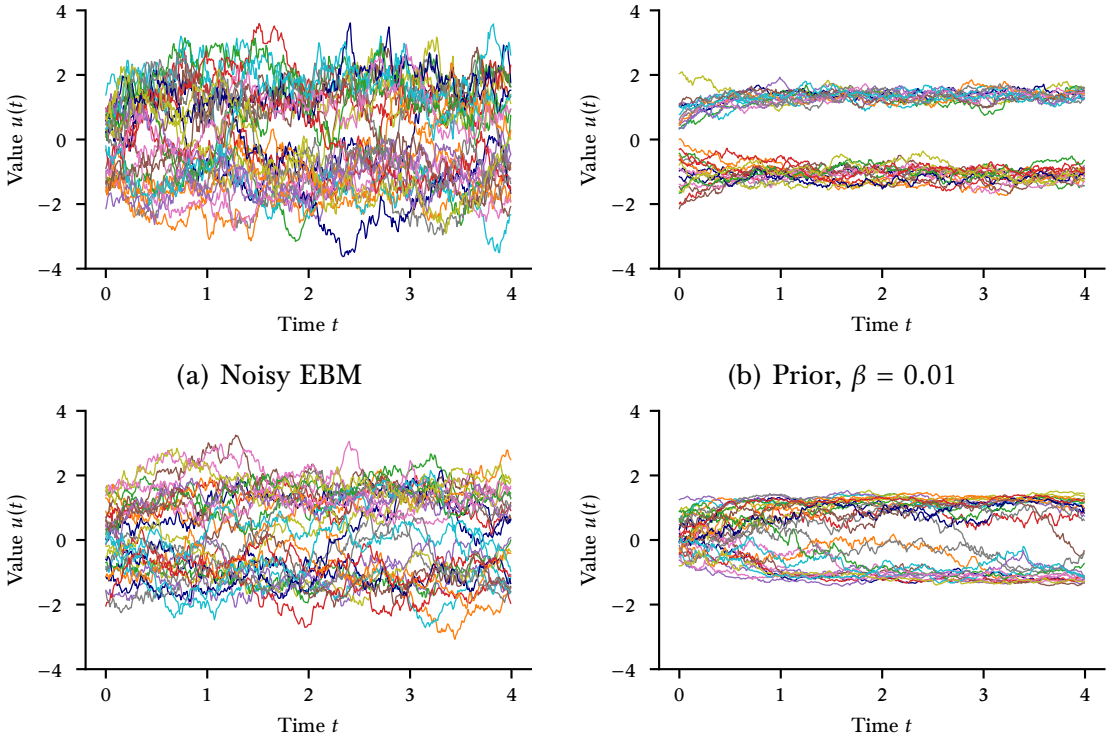
We now consider data generated by the model in Section 3.3. We are interested in whether the latent SDE can learn the drift and diffusion dynamics. Thus, we slightly modify the energy balance model from a constant to a linear diffusion term to make the dynamics more complex:

$$dT(t) = (E_{in}(T(t)) - E_{out}(T(t)))dt + 0.135 \cdot T(t)dB_t.$$

The values of $T(t)$ range between 220 and 320, so there is less noise in the lower attractor than the upper one. For the rest of this section, data from the energy balance model will be normalized and not in its original range of values.

The chosen values for the latent SDE are in Table 2. There is no change to the architecture of the latent SDE from the Ornstein-Uhlenbeck experiment, except that we use a one-dimensional latent space with the knowledge that the underlying model is one-dimensional. This is a caveat of our experiments, which we discuss in Section 5.3.6.

In Figure 12 and Figure 13, we plot samples from the training data, the posteriors,



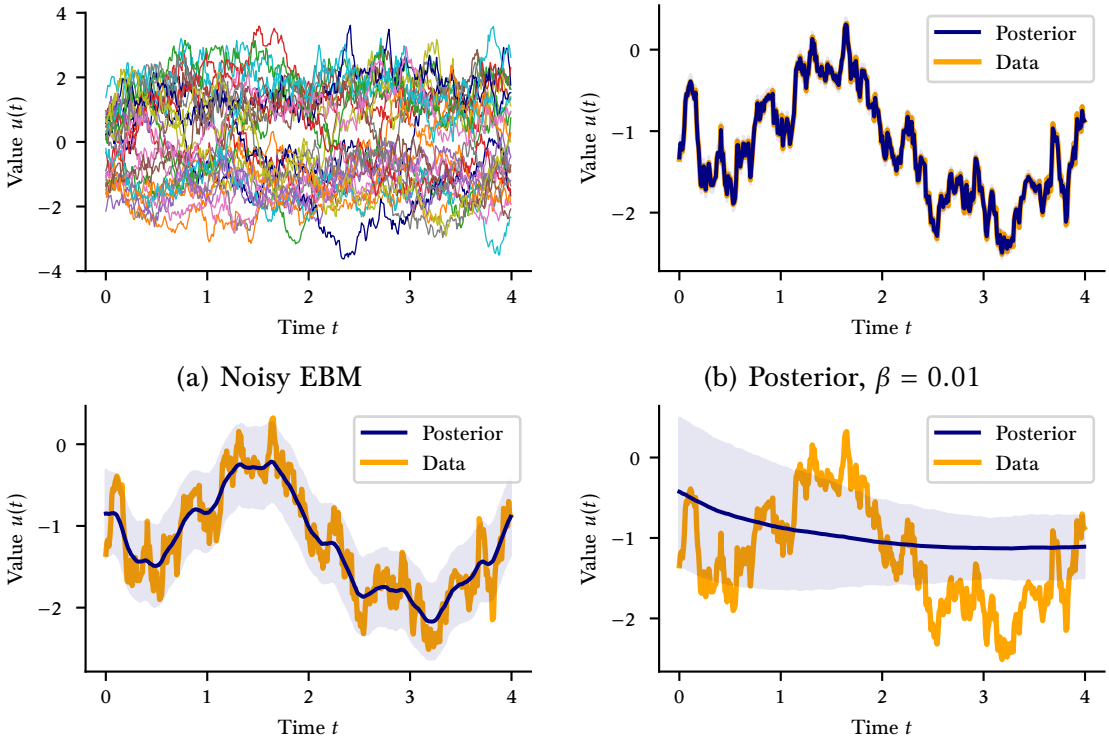
(a) Noisy EBM

(b) Prior, $\beta = 0.01$

(c) Prior, $\beta = 100$

(d) Prior, $\beta = 10000$

Figure 12: EBM: Prior SDE for values of β



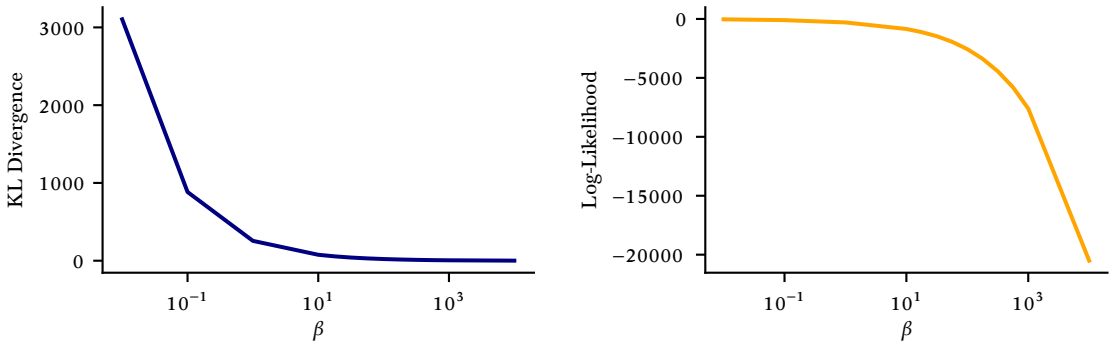
(a) Noisy EBM

(b) Posterior, $\beta = 0.01$

(c) Posterior, $\beta = 100$

(d) Posterior, $\beta = 10000$

Figure 13: EBM: Posterior SDE around data with 95% confidence interval for β



(a) KL divergence for β

(b) Log-Likelihood for β

Figure 14: EBM: Log-Likelihood and KL Divergence for β

and the priors for $\beta \in \{0.01, 100, 10000\}$, and discuss the results visually before we later quantify them.

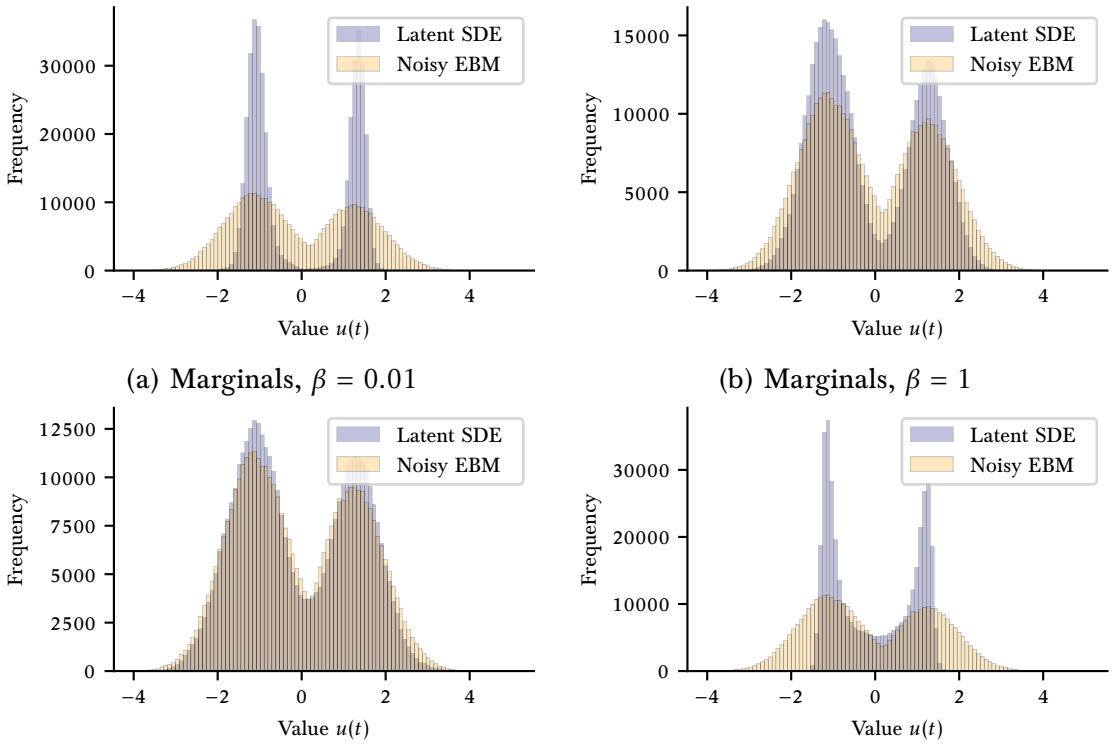
For $\beta = 0.01$, the training loss is weighted much more towards maximizing the log-likelihood between the posterior and the data. The posterior indeed tracks the data very closely, but the prior does not resemble the data distribution. While the prior has learned that there are two attractors and their approximate positions, it does not exhibit any tipping between these attractors.

For $\beta = 10000$, the training loss is weighted much more towards minimizing the KL divergence between prior and posterior. The posterior does not track the data at all. The prior also has two attractors and exhibits tipping between them, but the dynamics do not visually resemble the data set.

Finally, for $\beta = 100$, the prior visually resembles the data, although it seems less noisy; for instance, there are fewer points at which trajectories are close to 4 or -4. Still, this is the value of β for which the latent SDE's prior looks like it has the largest diffusion term. The posterior tracks the data, but has more variance around the data than with $\beta = 0.01$.

To verify that the loss is indeed weighted more towards the log-likelihoods for low β and more towards the KL divergence for high β, we plot them directly in Figure 14. Around $\beta = 10$, both values are "reasonably good", and the scores get more uneven with higher and lower values of β.

We can make our analysis more concrete by computing summary statistics. We use the marginals and their Wasserstein distances as before. We analyze the *tipping rates* to analyze the stochastic properties of the data and the model. We define the tipping rate of a data set as the expected rate at which any trajectory crosses the approximate bifurcation point. The tipping rate is computed by running a convolution over each



(a) Marginals, $\beta = 0.01$

(b) Marginals, $\beta = 1$

(c) Marginals, $\beta = 100$

(d) Marginals, $\beta = 10000$

Figure 15: EBM: Marginals for β in interval $(0, t_{\text{train}})$

time series and counting the crossings. For the simple bistable model, we estimate the bifurcation as the point in $[-1, 1]$ corresponding to the lowest bar of the histogram of the marginals with 100 bins. We check that this point is not at the edges of our chosen interval. The same experiment for $\beta \in \{10^{1.25}, 10^{1.5}, \dots, 10^{2.75}\}$ is added to the summary statistics plots for more data points for the interesting regions.

We plot the marginals in Figure 15 and compare the Wasserstein distances in Figure 16. Here, we can quantify what we have seen before in the visual analysis: judging from the data analysis, there seems to be a sweet spot for β between 10^1 and 10^3,

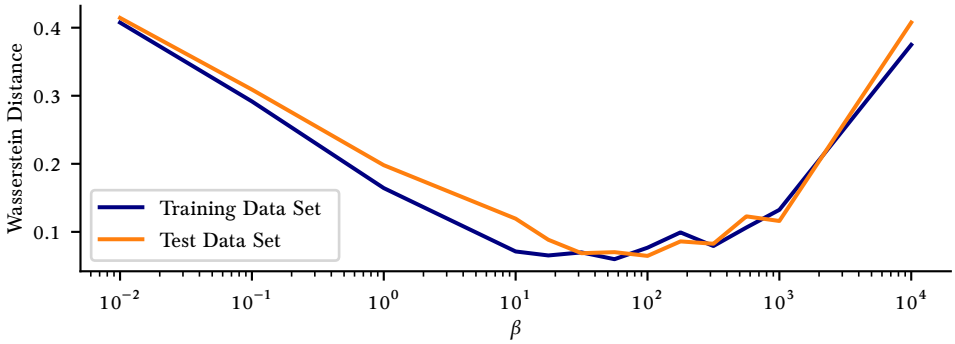


Figure 16: EBM: Wasserstein distance for β

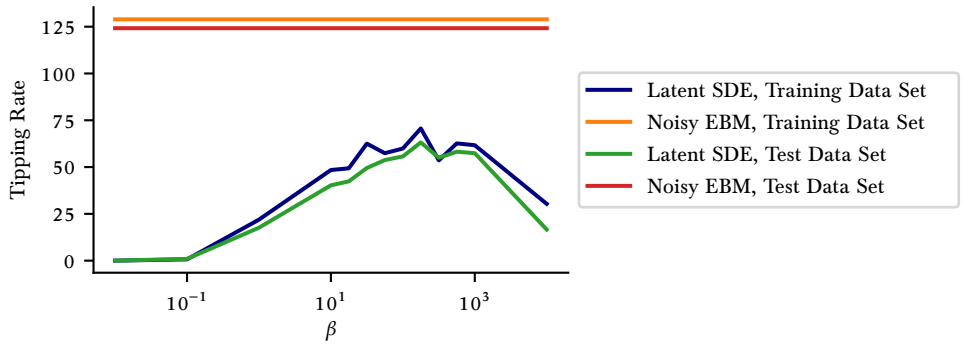
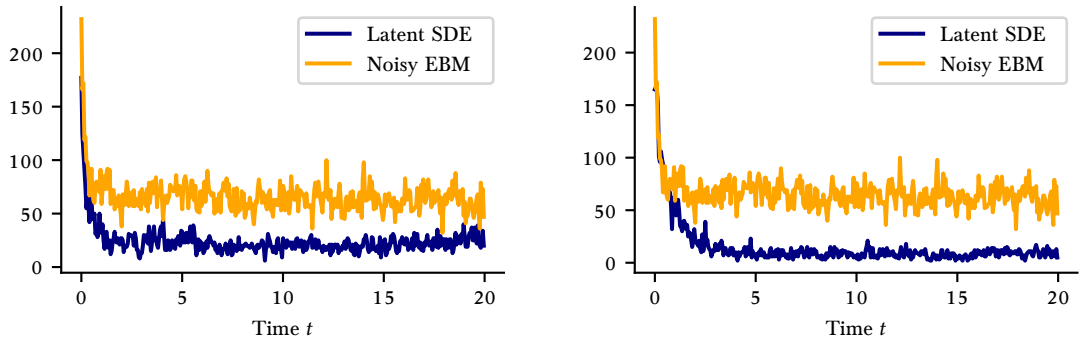


Figure 17: EBM: Tipping rate for β



(a) EBM: Tipping rate, $\beta = 10$

(b) EBM: Tipping rate, $\beta = 10000$

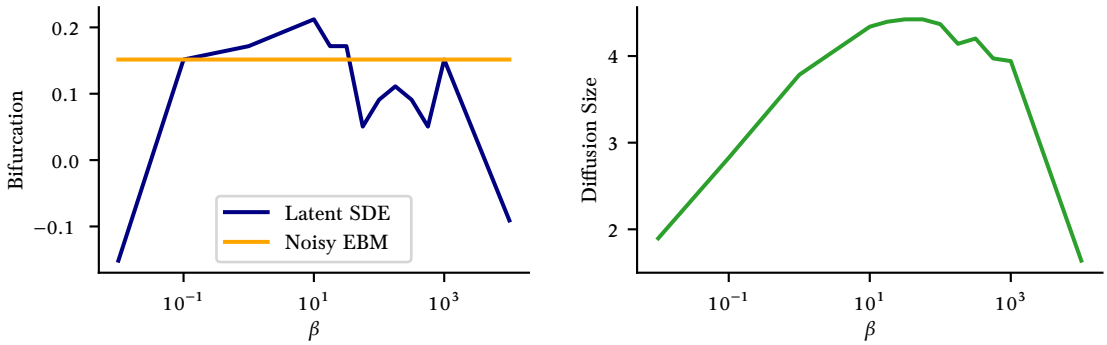
Figure 18: EBM: Tipping rate over time

where the Wasserstein distances are much lower. For β that are smaller or larger, the histograms of the latent SDEs become more concentrated on the attractor and less broadly around it. This may suggest that the diffusion size in the latent SDE is smaller there, which we confirm later.

The Wasserstein distances of the test set are almost the same as those of the training set because we chose a one-dimensional latent SDE with the knowledge that the underlying model is one-dimensional. We discuss this in Section 5.3.6.

Next, we investigate the tipping rates in Figure 17. We can immediately see that the tipping rates of all latent SDEs are too low, less than half of the EBM's tipping rates. With values of β that are too high or too low, the tipping rates decrease alongside the size of the diffusion. We investigate this effect in Section 5.3.3.

In Figure 18, we choose windows of size 0.05 and plot the tipping rate over time. We see that the latent SDE consistently underestimates the tipping rate rather than just at some point in time. Initially, the model is still evolving from its Gaussian initial state, so we chose the training timespans in the scoring plots to exclude this part.



(a) Approximate bifurcation in $[t_{\text{train}}, 5t_{\text{train}})$ (b) Size of diffusion term

Figure 19: EBM: More statistics for tipping rates depending on β

5.3.2 Direct Model Comparison

We have compared the data generated by the underlying bistable model and the trained latent neural SDE data. However, we know the underlying model and can compare the two directly. We plot the energy balance model's drift (converted to Stratonovich) and the latent neural SDE's priors in Figure 20.

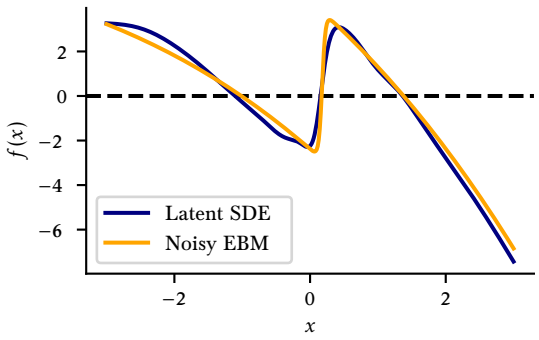
As the zeroes of the drift function $f(x)$ represent the two attractors and the repellor, the latent SDE must reproduce these points accurately. These are pretty close, although the left attractor is slightly off for all experiments. The dynamics between these zeroes are also important but less so, as with different diffusion functions, different shapes of this function could be justified and generate very similar outputs.

Surprisingly, $\beta = 0.01$ shows the closest fit for the drift function, as the data generated from this model does not fit the statistics well. For increasing values of β, the amplitude of the drift decreases. We investigated this further by choosing $\beta = 0.001$; this makes the distance between latent SDE and energy balance model larger again. We do not have an explanation, but we think it is partly a coincidence that the prior so accurately matches the drift of the EBM for this specific value. Note that the diffusion does not match the EBM's diffusion for $\beta = 0.01$ as seen in Figure 21.

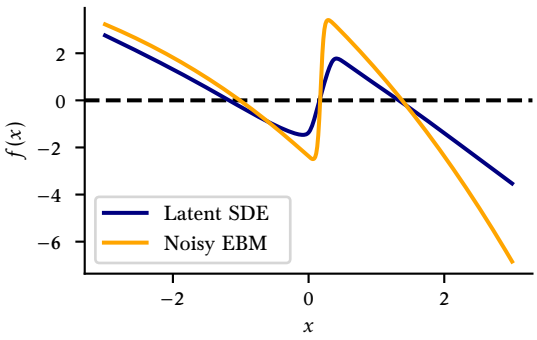
We plot diffusion values in Figure 21. For $\beta \in \{0.01, 1, 100\}$, the diffusion is constant, as opposed to the diffusion of the energy balance model, which we have chosen to be linear. For $\beta = 10000$, the latent SDE's diffusion has a cone shape. Thus, the latent SDE underestimates the magnitude of the diffusion and cannot estimate its shape.

5.3.3 Investigation of Noise Underestimation

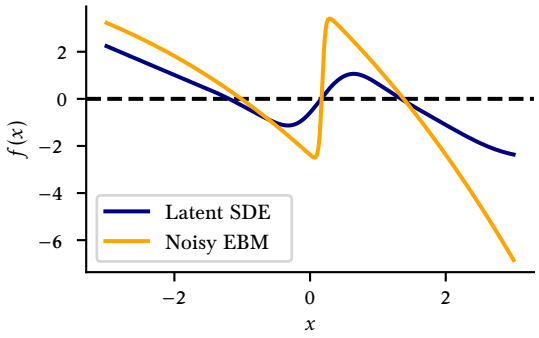
We have observed that latent SDEs underestimate the size and cannot fit the shape of the diffusion. This section reviews latent SDEs as a model and explains why this



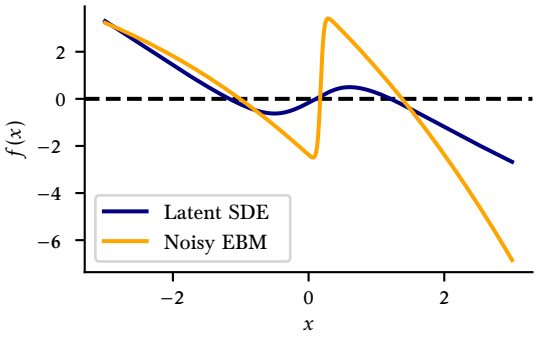
(a) Value of drift, $\beta = 0.01$



(b) Value of drift, $\beta = 1$

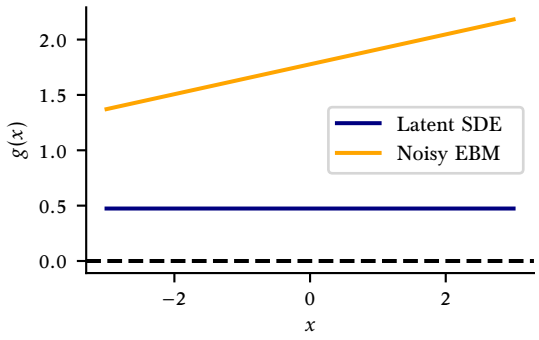


(c) Value of drift, $\beta = 100$

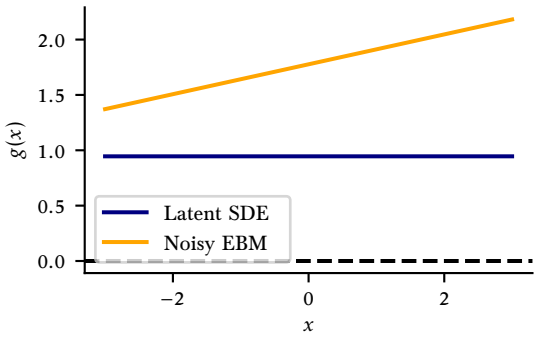


(d) Value of drift, $\beta = 10000$

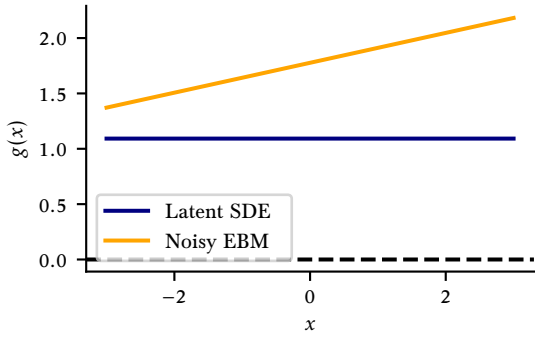
Figure 20: EBM: Value of drift for β



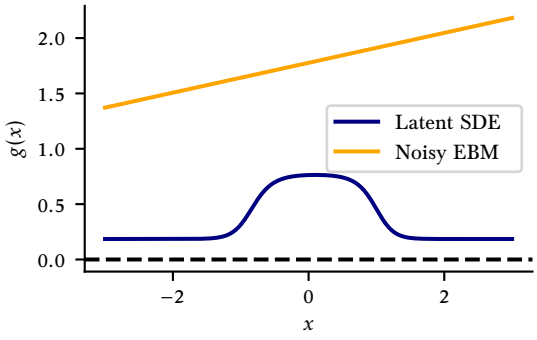
(a) Value of diffusion, $\beta = 0.01$



(b) Value of diffusion, $\beta = 1$



(c) Value of diffusion, $\beta = 100$



(d) Value of diffusion, $\beta = 10000$

Figure 21: EBM: Value of diffusion for β

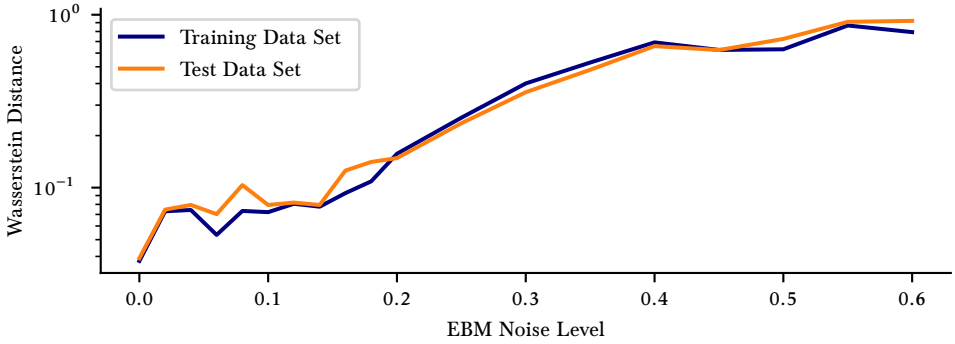


Figure 22: Wasserstein distance for EBM noise levels

might be.

What is the role of diffusion in the training procedure of latent SDEs? Recall the loss function of a latent SDE

$$\max_{\theta,\phi} \mathbb{E}_{x_{\text{data}}}(\mathscr{L}_E - \beta \mathscr{L}_{KL}).$$

The size of the diffusion term, as shown in Figure 19b, significantly decreases with high values of β. This is surprising when considering the formulation of the training loss: as the KL divergence is proportional to the inverse of the diffusion and β increases its importance, we expected the size of the diffusion term to continue to increase. A possible explanation would be that as β increases so much that the log-likelihoods become unimportant, there is no need to increase the diffusion if prior and posterior are so close that their difference is almost at zero. An extreme case of this is that the prior and posterior are equal, making the size of the diffusion arbitrary.

One can expect that for very small values of β, the magnitude of the trained diffusion will be close to zero, as only the posterior loss $\mathscr{L}_E$ is minimized. Adding diffusion will compromise its accuracy as the posterior traces a data sample. In this extreme case, the diffusion size of the latent SDE has nothing to do with the amount of noise in the data.

For other values of β, it is unclear how the data noise and the latent SDE's diffusion size relate. We check if they have any relationship by varying the diffusion size of the energy balance model and training a latent SDE with $\beta = 10$ on the data. The results are shown in Figure 22 and Figure 23, where "EBM noise level" is the size of the constant in the linear diffusion term of the energy balance model that is generating the data. We observe a correspondence between this and the size of the latent SDE's diffusion, although the latent SDE's diffusion is always smaller.

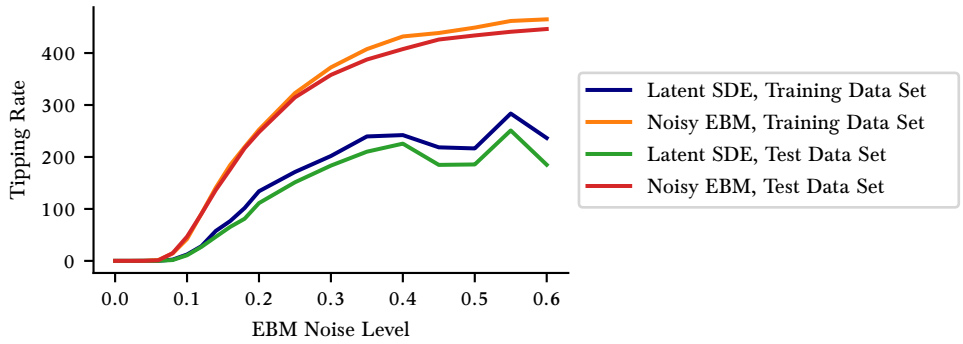


Figure 23: Tipping rate for EBM noise levels

It is more enlightening to plot the size of the KL divergence, log-likelihood, and diffusion of the latent SDE trained on EBM data with different noise levels, as in Figure 24. First, we observe that the data gets harder to approximate with increasing noise levels as both the KL divergence and log-likelihoods worsen. Like the KL divergence and log-likelihood, the latent SDE's diffusion size grows almost linearly with the EBM noise level.

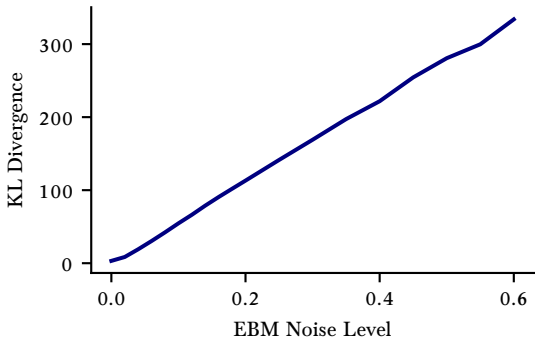
This gives rise to the following explanation: For latent SDEs, there is a "balance point" between the KL divergence and the log-likelihood. Figure 24 suggests that the diffusion size is chosen according to this balance while training. Increasing the diffusion size worsens the log-likelihoods, and decreasing it worsens the KL divergence. To show this, we now use the latent SDE for $\beta = 1$ and replace the diffusion with constant values from zero to two in Figure 25. The actual size of the final model's diffusion, as seen in Figure 21, is correlated with the point where these two scores meet. The harder the problem becomes to predict, the further this point moves to the right, leading to the relationship between the EBM's noise level and the latent SDE's diffusion size we observed.

5.3.4 Injecting Noise Into Latent SDEs

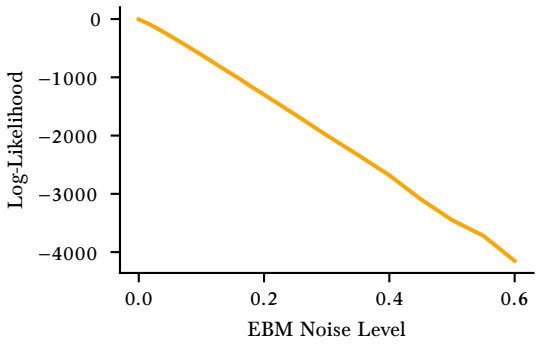
As an attempt to match the tipping rate of the data as well as the marginals, we amend the loss function to include a term for the size of the diffusion:

$$\max_{\theta,\phi} \mathbb{E}_{x_{\text{data}}}(\mathscr{L}_E - \beta \mathscr{L}_{KL} + \gamma \mathscr{L}_G)$$
$$\text{where } \mathscr{L}_G = \int_0^T g_\theta(u(t), t)dt.$$

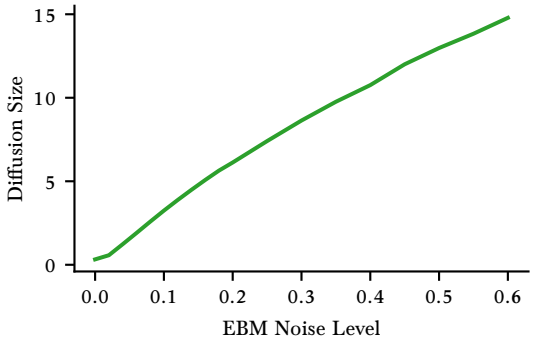
Alternatively, assuming we would know the desired value of $\mathscr{L}_N$, we could use a



(a) KL divergence for EBM noise levels



(b) Log-Likelihood for EBM noise levels



(c) Diffusion size of latent SDE for EBM noise levels

Figure 24: Training information for EBM noise levels

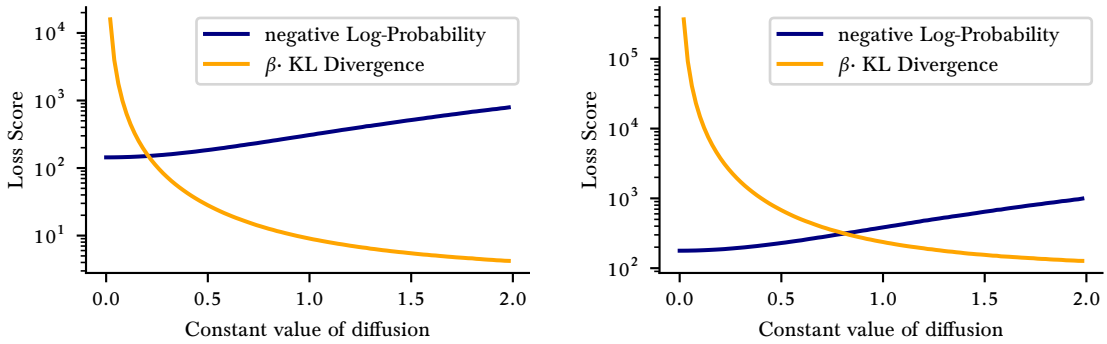
regularization-like penalty:

$$\max_{\theta,\phi} \mathbb{E}_{x_{\text{data}}}(\mathscr{L}_E - \beta\mathscr{L}_{KL} + \gamma(\mathscr{L}_G - g_{\text{target}})^2).$$

However, we do not know g_{target} in general, so this adds two hyperparameters instead of one. Thus, we choose the first loss function for simplicity. We compute $\mathscr{L}_G$ just like $\mathscr{L}_{KL}$, adding another dimension in the integration of the posterior [25, cf.]. For this experiment, we choose $\beta = 10$.

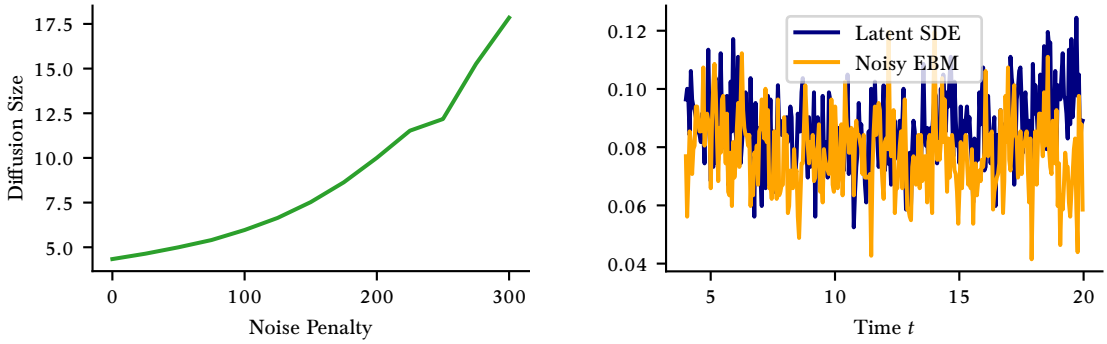
In Figure 26a, we observe that increasing γ will increase the size of the diffusion term. Its effect on the tipping rate is shown in Figure 28, which immediately correlates to the diffusion size. Relating the Wasserstein distance to γ in Figure 27, we cannot observe a clear pattern but see values similar to those that we would obtain by setting $\gamma = 0$.

Again, we also compare the models directly in Figure 30 and Figure 31. We observe that both drift and diffusion increase in amplitude when increasing γ. The latent SDE trained with $\gamma = 200$ matches the tipping rate and Wasserstein distances quite well. However, the amplitude of drift and diffusion terms are larger, and the diffusion term is constant instead of linearly increasing.



(a) Diffusion balance, $\beta = 0.01$ (b) Diffusion balance, $\beta = 1$

Figure 25: EBM: The balance between KL divergence and log-likelihood yields the diffusion size



(a) Noise penalty γ injects noise (b) Tipping for $\gamma = 200$

Figure 26: EBM-Penalty: Injecting noise

We conclude that by setting the hyperparameter γ to the correct value, we can synthesize an SDE that matches Wasserstein distances and tipping rates of the data relatively well for the noisy energy balance model. In this setup, γ cannot be learned automatically, leaving us with two hyperparameters that must be manually tuned. On the other hand, with enough data, the correct parameters can be obtained just by analyzing the data without any further knowledge of the underlying dynamics.

5.3.5 Training on Few Datapoints

How do latent SDEs fare with small data sets? We train them on different data set sizes of the energy balance model ranging from 4 to 4096 trajectories in powers of two with $\beta = 10$. We do not change the trajectories, the Δt, or the timespans. There are 400 data points per trajectory, so we trained the model on data sets with 1600 to 1638400 data points.

Summary statistics can be seen in Figure 32 and Figure 33. Unsurprisingly, the Wasserstein distances continually improve with more data. Tipping rates can only

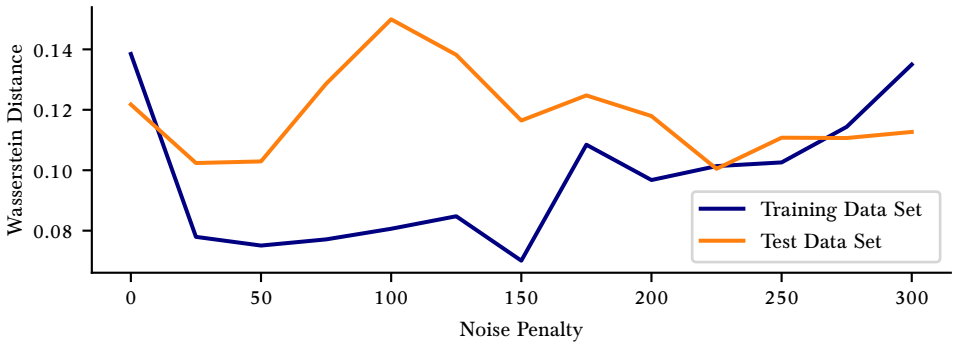


Figure 27: EBM-Penalty: Wasserstein distance for γ

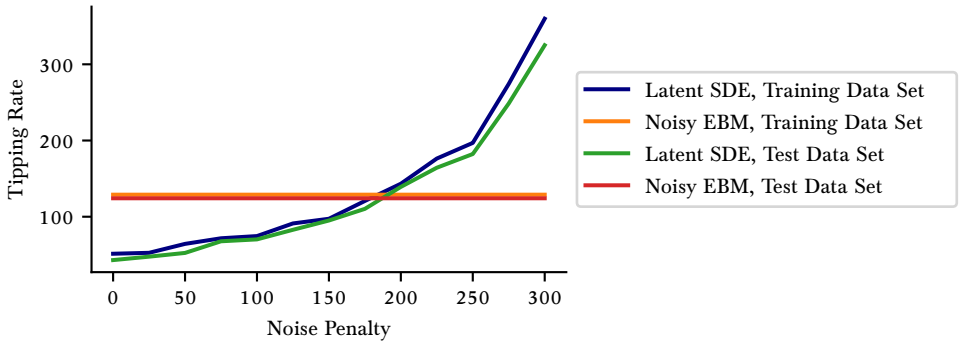
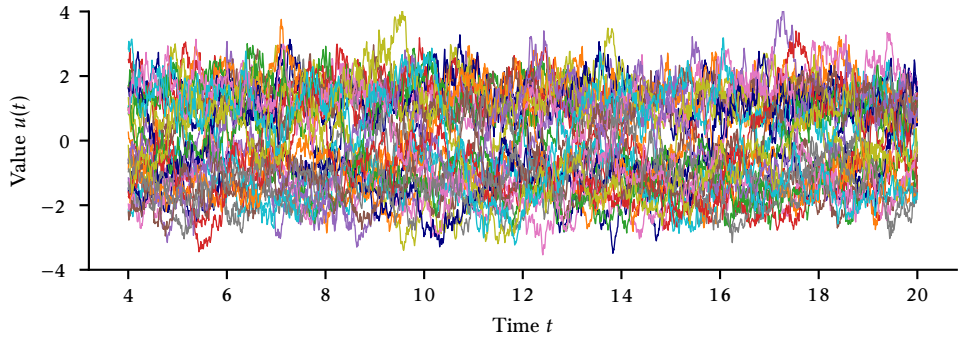
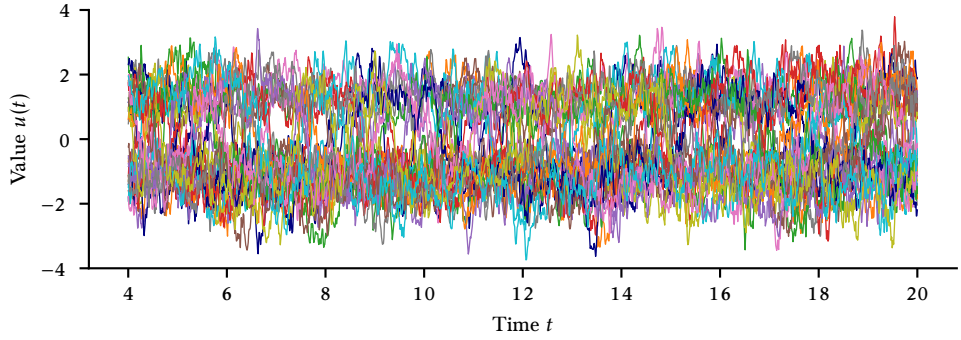


Figure 28: EBM-Penalty: Tipping rate for γ

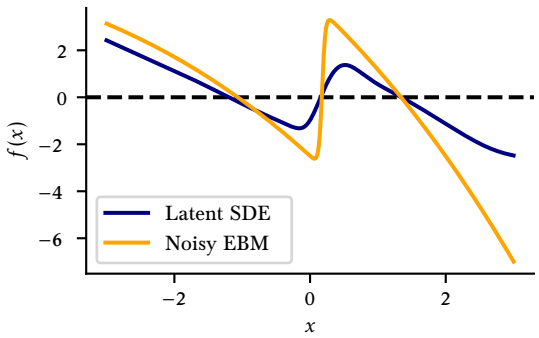


(a) Noisy EBM

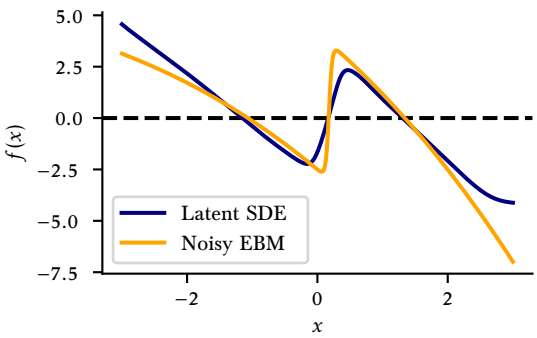


(b) Latent SDE Prior, $\beta = 10, \gamma = 200$

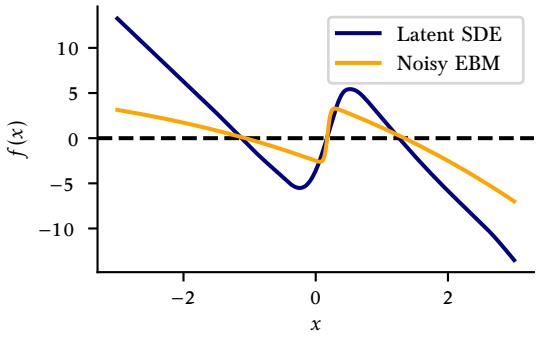
Figure 29: EBM-Penalty: Comparison of best result against ground truth



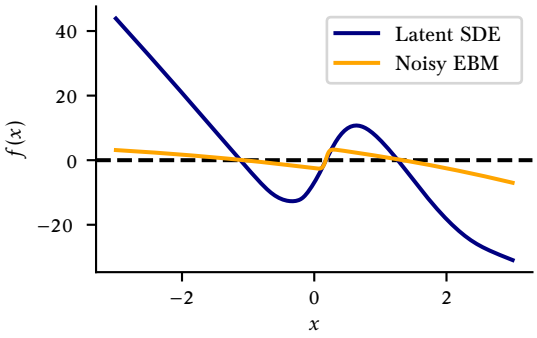
(a) Value of drift, $\gamma = 0$



(b) Value of drift, $\gamma = 100$

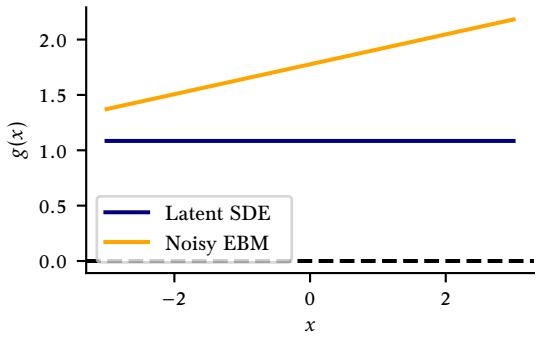


(c) Value of drift, $\gamma = 200$

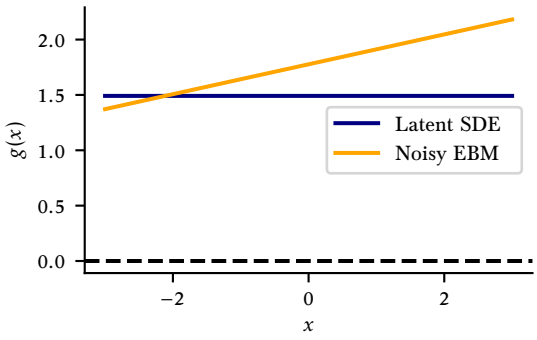


(d) Value of drift, $\gamma = 300$

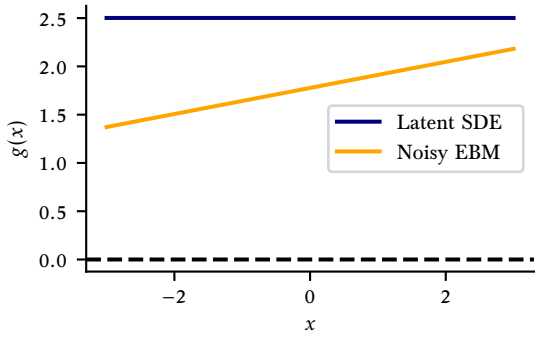
Figure 30: EBM-Penalty: Value of drift for γ



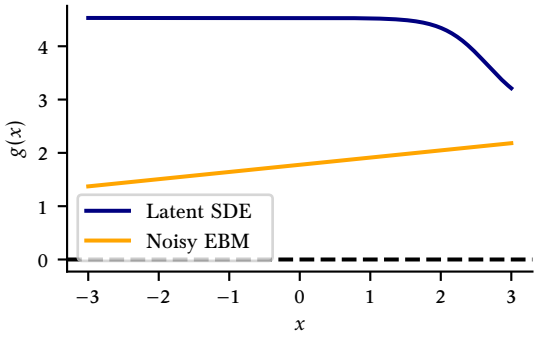
(a) Value of diffusion, $\gamma = 0$



(b) Value of diffusion, $\gamma = 100$



(c) Value of diffusion, $\gamma = 200$



(d) Value of diffusion, $\gamma = 300$

Figure 31: EBM-Penalty: Value of diffusion for γ

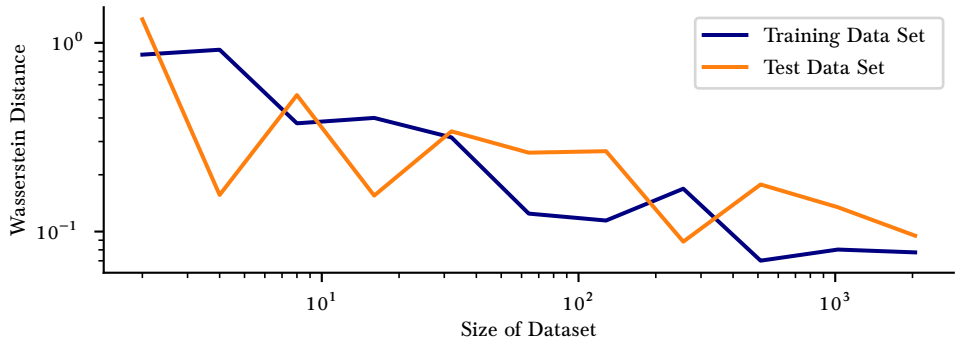


Figure 32: EBM: Wasserstein distance for amounts of data

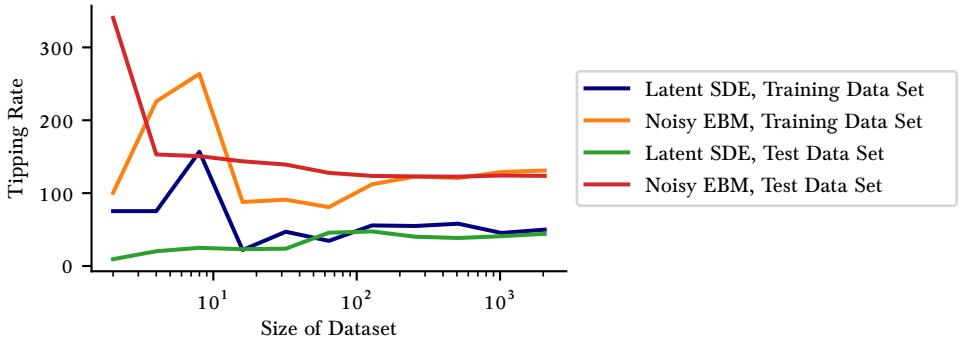
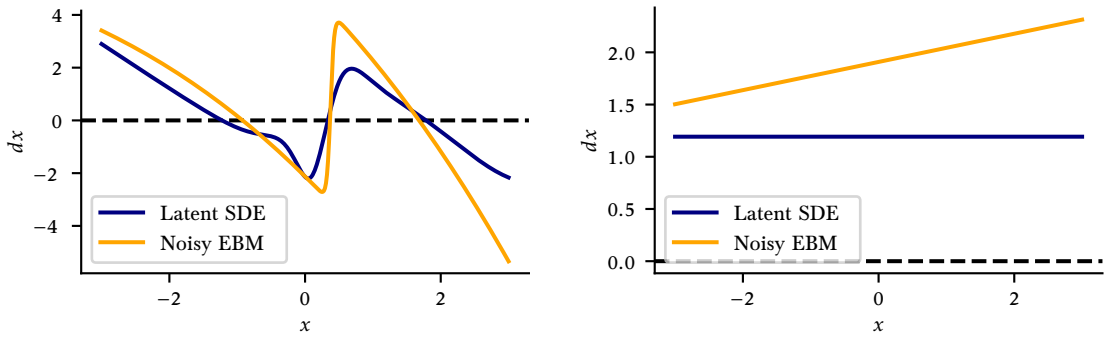


Figure 33: EBM: Tipping rate for amounts of data

be computed in a stable way somewhere above 256 trajectories. From a visual analysis of the results, the first experiment where the reconstructed drift looks close to the energy balance model's is with 64 trajectories, so 25600 data points. The results can be seen in Figure 34.

5.3.6 Discussion of Latent Space Dimensions

It seems crucial to our experiments that we chose the dimensionality of the latent space with the knowledge of the underlying model, for instance, one-dimensional in



(a) Drift, 64 trajectories, $\beta = 10$ (b) Diffusion, 64 trajectories, $\beta = 10$

Figure 34: Drift and diffusion for 64 trajectories, $\beta = 10$

the case of the energy balance model and two-dimensional in the case of the time-dependent Ornstein-Uhlenbeck process. This seems to defeat the machine learning method's purpose as we used prior knowledge about the underlying model. The small error on the test set is the main consequence of choosing a model with the same dimensionality as the data. As can be seen in Figure 16, the model performs just as well on the test set as on the training set, and this is because we know the dimensionality of the underlying model's state space. With a larger latent space, the error in the extrapolation would undoubtedly be worse. Why do we train with this assumption if it requires prior knowledge?

Li et al. have a setup without prior knowledge in their experiment and show that it works in principle. They use a four-dimensional state space for a two-dimensional data set [25]. With a general latent space, a trainable projector is necessary. With a trainable projector, the dynamics of the model itself are not comparable to the underlying model, rendering the latent SDE more opaque. There would be no simple way of comparing the direct models in Section 5.3.2 and no point in investigating the diffusion sizes as in Figure 24. Our choice of the same latent space dimensions is thus necessary for the kind of analysis we did and acceptable while keeping in mind that the test set errors are not accurate for a general application.

Latent SDEs with trainable projectors will have to use a formulation of the noise penalty that takes the projector into account, as otherwise, the model can scale up the noise and then scale down the projector.

5.4 FitzHugh-Nagumo Models

The FitzHugh-Nagumo model (FHN) is a generalized Van der Pol oscillator designed initially as a simplified model for neuron spikes in nerves. Its state space has two dimensions. The original model has an attractor P, which FitzHugh calls the resting point. When disturbed far enough from the P, a trajectory will travel through the model's state space and arrive back at P at some point [11].

The FHN can change its qualitative features by changing the parameter values. For some parametrizations, it will lose the fixed point P and shift to a so-called limit cycle, a stable oscillator. Such a shift in dynamics is called a Hopf bifurcation [33].

Because it exhibits Hopf bifurcations, the FHN is a framework that can display different qualitative behaviors, like limit cycles and mono- or multistable systems. A

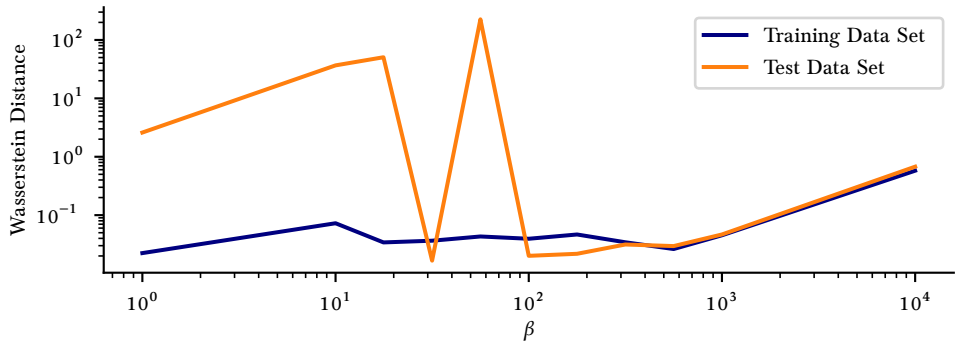


Figure 35: Limit cycle (LC) FHN: Wasserstein distance for β

general equation for the FHN is:

$$dx(t) = \frac{1}{\tau_x}(\alpha_1 x(t) - \alpha_3 x(t)^3 + y(t))dt + \sigma_x dB_{x,t},$$
$$dy(t) = \frac{1}{\tau_y}(\beta y(t) - x(t) + c)dt + \sigma_y dB_{y,t}.$$

We train a latent SDE on the data generated by two different parametrizations of the FHN. First, we choose a parametrization from Riechers et al. with which the FHN displays a limit cycle without a fixed point [33]. Second, we use an automatically synthesized parameterization from Lohmann and Ditlevsen that was selected to match certain behaviors of the NGRIP ice core record [26]. This second parametrization is monostable, although the data it produces looks like it is multistable, raising the question of whether a latent SDE can identify whether the underlying system is multistable from the data.

To find out whether the latent SDE can reproduce the behavior of an unobserved state space dimension, we only train the two-dimensional latent SDE on one of the FHN's two dimensions, hiding the second dimension. We train on the $x(t)$ dimension of this FitzHugh-Nagumo model and use the same settings as in Table 1 except that we use the timespan and Δt as in Table 2.

5.4.1 Limit Cycle FitzHugh-Nagumo with Diffusion

We first consider a FitzHugh-Nagumo model with a limit cycle as its main dynamical component. We parametrize the drift as $\alpha_1 = \alpha_3 = 2, \tau_x = 0.1, \tau_y = 1, c = \beta = 0$. We add a small diffusion term with $\sigma_x = \sigma_y = 0.1$. This setup is conceptually very similar to models presented by Riechers et al., although we swapped the x and y dimensions because they appear the other way around in Lohmann and Ditlevsen [26, 33]. We scaled the parameters τ_x and τ_y so the full dynamics of the model fit

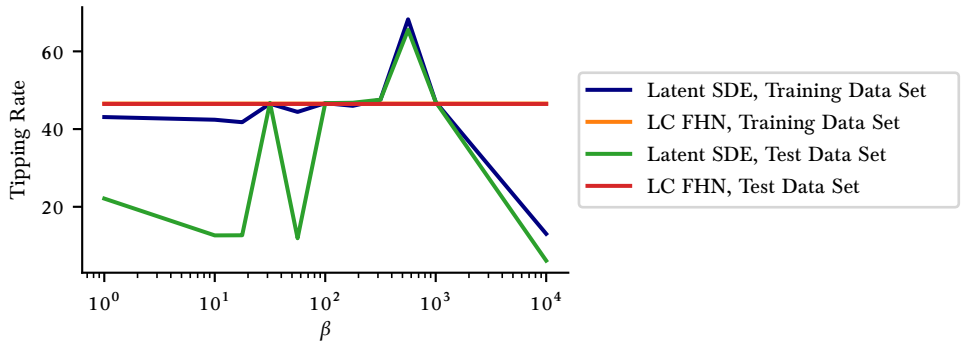


Figure 36: Limit cycle (LC) FHN: Tipping rates for β

into the timespan $[0, 4]$. Only their relation to each other is important, not their absolute size. The limit cycle can be seen in a trajectory of this model in the state space, as shown in Figure 37.

We train the latent SDE on the same values of β as always, but for values of $\beta < 1$, the Wasserstein distances between marginals are NaN, so we exclude these experiments from the plots. The Wasserstein distances are displayed in Figure 35, and the tipping rates in Figure 36. For lower values of β, we observe substantial test errors in some experiments. As we will see when plotting the state spaces, this is because latent SDEs "explode" in the state space if they do not learn the limit cycle but traverse the state space in a straight line.

Because the model exhibits a limit cycle and no stochastic tipping, the tipping rate measures the speed of this limit cycle. There are experiments where the tipping rates match precisely, and the Wasserstein distances are close, such as $\beta = 10^{1.5}$. We plot the phase portrait of this trained latent SDE in Figure 38 and see that it is a qualitatively very similar limit cycle, only that its direction of rotation is flipped and the values of $y(t)$ are different. Thus, the latent SDE can recover the limit cycle almost precisely within the constraint that y is unobserved. We can conclude that latent SDEs can reproduce the tipping rates when the "tipping" is caused by the drift component rather than the diffusion component of the underlying model.

Let us also discuss the previously mentioned explosions in the state space for smaller values of β. In Figure 40, we show an extrapolation of the trajectory for $\beta = 10^{1.25}$. This latent SDE has not learned that there is a cycle and will continue advancing through the state space after the training timespan, leading to a large test error. One could say that the latent SDE overfits to the training timespan and cannot generalize to reflect the underlying mechanism of the FHN. If this effect becomes problematic because it happens for many experiments, one could mitigate the issue by adding a

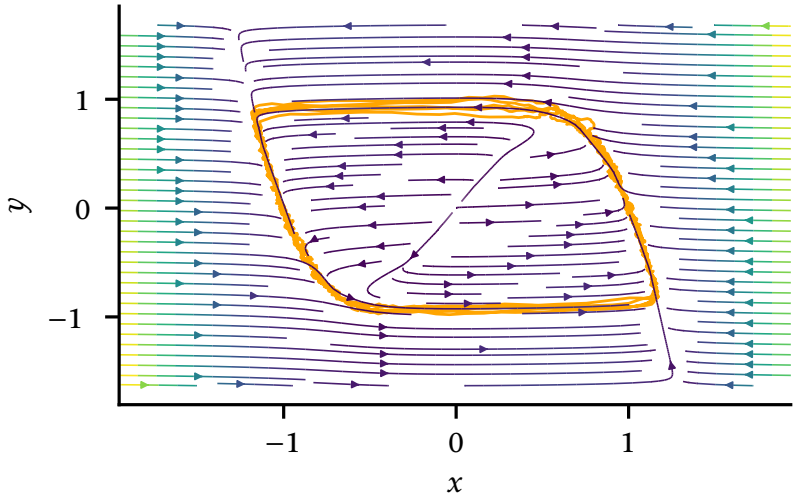


Figure 37: Phase portrait of limit cycle FHN

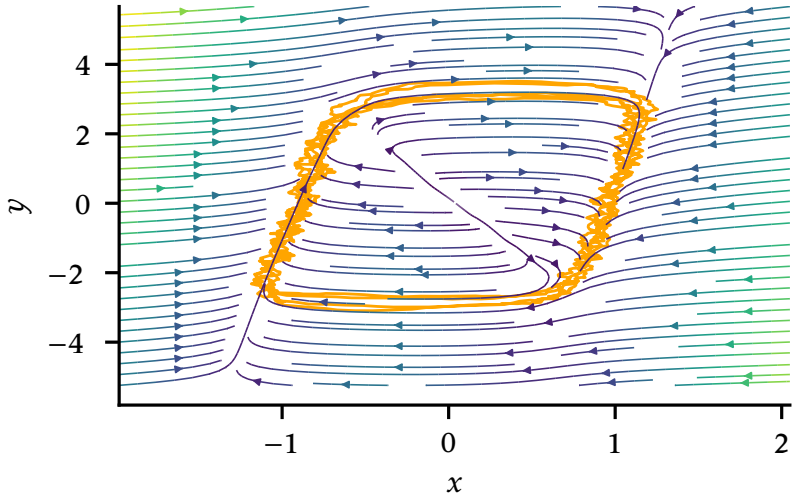


Figure 38: Limit cycle FHN: Phase portrait of latent SDE, $\beta = 10^{1.5}$

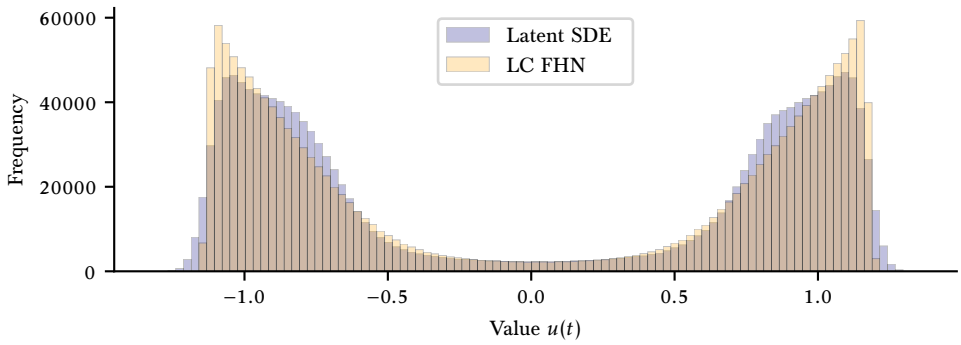


Figure 39: Limit cycle (LC) FHN: Marginals, $\beta = 10^{1.5}$

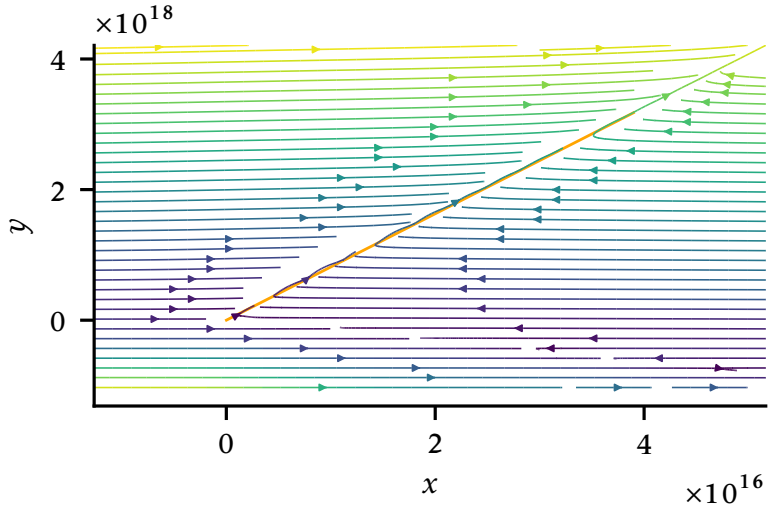


Figure 40: Limit cycle FHN: Phase Portrait of Latent SDE, $\beta = 10^{1.25}$

penalty for large unobserved values.

5.4.2 NGRIP Model by Lohmann and Ditlevsen

Lohmann and Ditlevsen use a Gaussian sampling method to select a parametrization for a bistable differential equation that most closely matches selected statistics from the NGRIP time series. They conclude that a model based on the FitzHugh-Nagumo equations is the best parametrization from their set of considered models. Lohmann and Ditlevsen call this model FHN_γ [26].

We use the resulting parametrization of the model: $\tau_x = \tau_y = 1, b = 2.55, \alpha_1 = 0.63, \alpha_3 = 2.71, c = 0.22, \sigma_x = 4.80, \sigma_y = 11.08, \beta = \tan(-0.67)$. The first dimension of the state space $x(t)$ is the observed dimension. By writing down the drift part of the equations, setting $\frac{dx}{dt} = \frac{dy}{dt} = 0$, and solving for x and y, we find a single fixed point $x = 0.256, y = -0.045$. This fixed point is an attractor. Thus, the NGRIP FHN is a monostable model in the drift term, but it looks like a bistable model when plotting the marginals of $x(t)$ because of the large diffusion terms.

As before, we train two-dimensional latent SDEs on the first dimension of the FHN's state space. We plot the resulting Wasserstein distances in Figure 41.

For low and high values of β, we again observe an overfit to the training data. To show that this overfit is again caused by unbounded state space traversal, we plot phase portraits and samples from the FitzHugh-Nagumo model and two prior SDEs in the timespan $(0, 2t_{\text{train}})$ in Figure 43. For β around 100, the trained latent SDE will remain in a fixed interval, but for low and high values of β, the hidden dimension

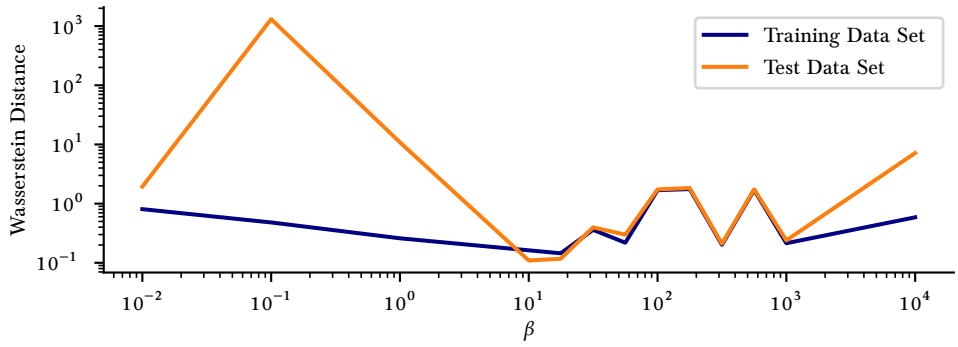


Figure 41: NGRIP FHN: Wasserstein distance for β

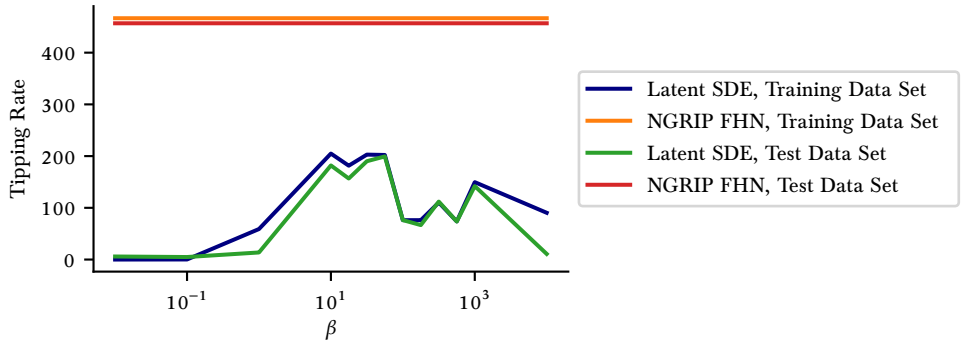
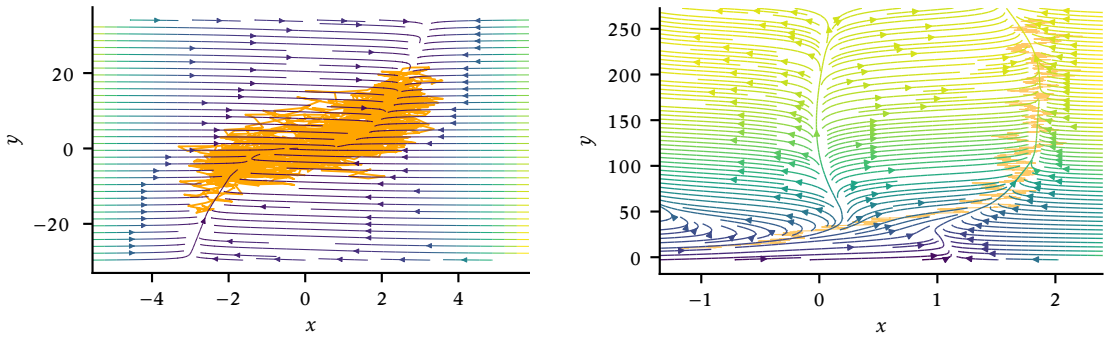
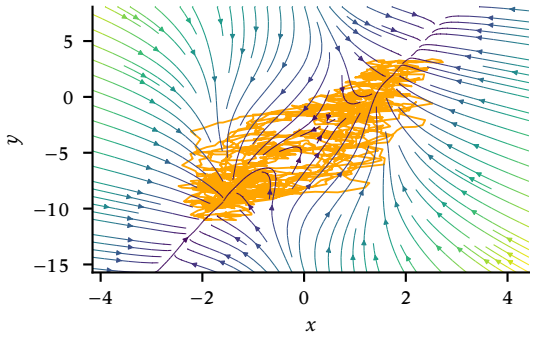


Figure 42: NGRIP FHN: Tipping rate for β

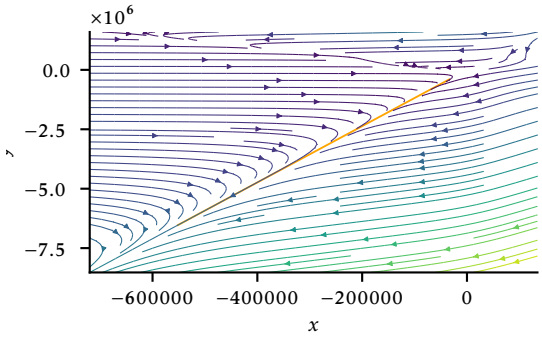


(a) Phase portrait of FHN_γ

(b) Phase portrait of prior, $\beta = 0.01$



(c) Phase portrait of prior, $\beta = 10$



(d) Phase portrait of prior, $\beta = 10000$

Figure 43: Phase portraits of trained models for values of β

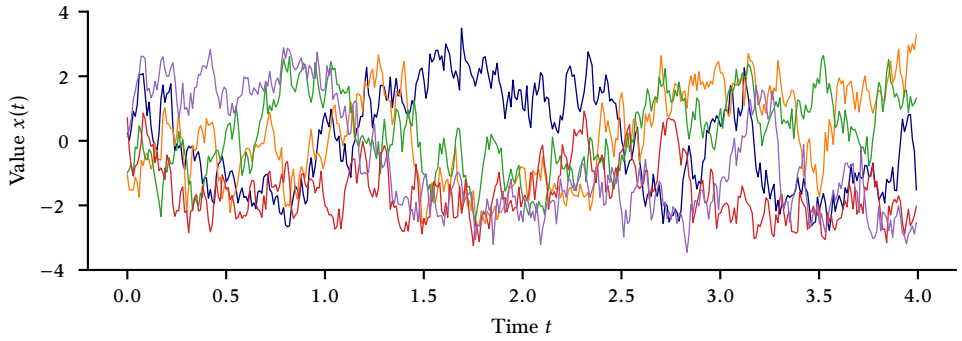


Figure 44: NGRIP FHN: NGRIP FHN model in $(0, t_{\text{train}})$

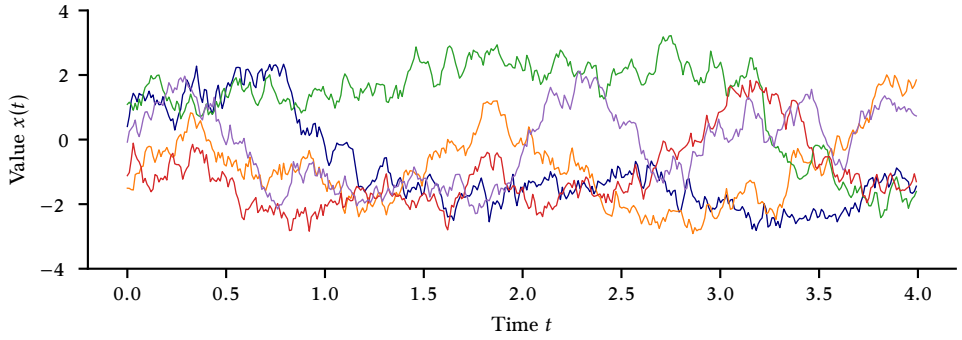


Figure 45: NGRIP FHN: Prior for $\beta = 10$ in $(0, t_{\text{train}})$

will continue to increase or decrease. In this case, the model overfits to the training timespan, which means that after the training timespan, the model's statistics will immediately diverge from the data.

The trained latent SDE for $\beta = 10$ exhibits comparatively good statistics, but it is bistable, as can be seen in its phase portrait in Figure 43c, whereas the NGRIP model is monostable. Thus, a latent SDE can not immediately recognize whether the underlying model is bistable or monostable for noise-driven tipping.

We plot the tipping rates in Figure 42. The tipping rates are too low, like for the

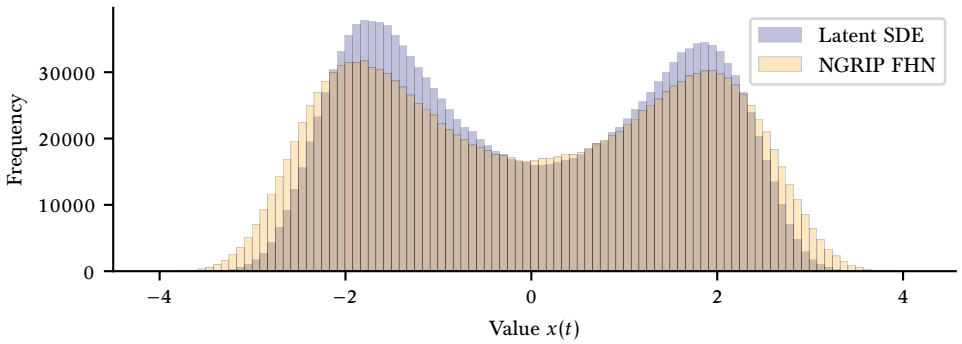


Figure 46: NGRIP FHN: Marginals for $\beta = 10$ in $[t_{\text{train}}, 5t_{\text{train}})$

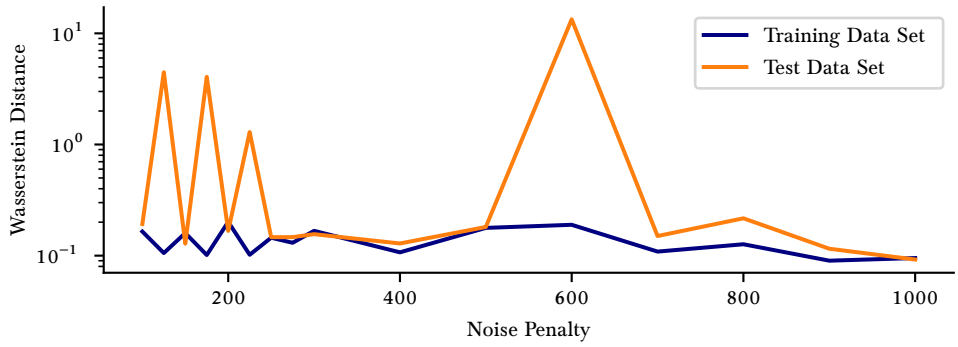


Figure 47: NGRIP FHN-Penalty: Wasserstein distance for γ

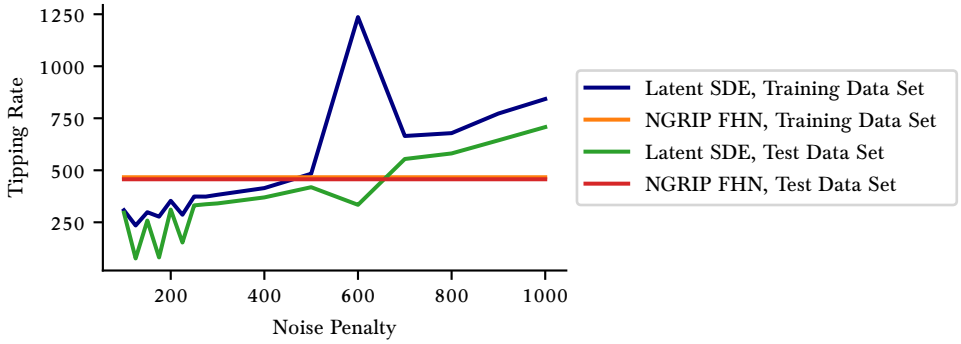


Figure 48: NGRIP FHN-Penalty: Tipping rate for γ

EBM, in contrast to the limit cycle FHN. For the NGRIP FHN, the "tipping" is driven by noise, as the drift term has a single fixed point, and as we already know, latent SDEs underestimate the diffusion size.

We add a noise penalty and show results in Figure 47 and Figure 48. The tipping rates can be increased while keeping similar Wasserstein distances if the model does not "explode" in phase space, which it sometimes does. Again, adding a regularization that prevents this explosion from happening should be possible.

5.5 Discussion of Results

Our primary questions for this section were the following:

1. Can neural SDEs replicate the statistics of multistable systems? Can they capture multistability and perform well in summary statistics?
2. How do *latent* SDEs behave, in general, in relation to their input data set? Can they match both the drift and diffusion components of a data set?

We have seen that latent SDEs can reconstruct the behavior of chaotic systems from partial information for simple dynamical systems like the zero-dimensional energy balance or FitzHugh-Nagumo models, answering the first question positively. When

the dynamics of the system are not forced by the noise, like in Section 5.4.1, the latent SDE can reproduce the state space representation accurately for correct choices of the hyperparameter β and the dimensionality of the state space. For systems forced by the noise like in Section 5.3 and Section 5.4.2, latent SDEs cannot reproduce the tipping rates without further tuning, an effect that we have explored in Section 5.3.3 and manually mitigated in Section 5.3.4. Adding another tunable hyperparameter for the diffusion size results in a model that can accurately reproduce the underlying model's dynamics. The answer to the second question is thus: latent SDEs can only match the diffusion with manual tuning.

Using SDE-GANs to estimate these systems might have advantages, although the training procedure will be more difficult. We expect that the noise underestimation problem would not arise for SDE-GANs in the same way, as the discriminator would ideally catch a trajectory with too little noise.

A modeling advantage of the latent SDE framework we have not explored is the possibility of prediction from data samples and out-of-distribution detection through the posterior SDE. One could give the posterior SDE an unseen data sample and make a forecast by switching to the prior after the last posterior point in the state space. Out-of-distribution detection might be possible by exploring the context returned by the encoder. To the best of our knowledge, this has not yet been attempted in the literature.

Our latent SDE setup was geared toward analyzing their internals and must be amended for real-world data. Most importantly, the latent SDE needs a trainable projector like the one used by Li et al. [25] because the dynamics of real-world data are not constrained by a fixed-dimensional state space. One also needs to tweak the framework towards the amount of data available. We successfully reconstructed data sets within the tens of thousands of data points, but this value also depends on the visibility of dynamics within those data points. The architecture of the neural networks inside the latent SDE might have to be tweaked; we did not explore this variable.

This concludes our experiments with neural SDEs. To demonstrate that latent SDEs are a valuable tool for stochastic time series, we use two more straightforward methods and attempt to fit the energy balance data set in the next section. In Section 7, we dive further into the workings of training neural SDEs and discover a trick only present in some of their implementations.

6 Comparison against Baselines

To show that we can solve non-trivial problems with neural SDEs, we attempt to use a completely stochastic model without any machine learning components and a machine learning model that is not stochastic to fit the energy balance data set. ARIMA models are simple autoregressive stochastic methods widely used for predicting stochastic time series. RNNs are the standard method for predicting deterministic data series in machine learning.

6.1 ARIMA Models

Autoregressive integrated moving average (ARIMA) models are some of the most common and successful methods to fit stochastic time series [7, p. 559], so we use them as a baseline. They are built around linear processes, which use weights to formulate a time series around a white noise process a_t. ARIMA models are integrated forms of autoregressive moving average (ARMA) models, which we will introduce now.

6.1.1 Introduction to AR(I)MA Models

An ARMA(p, q) [7, p. 53] process takes a white noise process a_t as input and consists of an *autoregressive process* with a horizon of p and parameters ϕ_i and a *moving average process* with a horizon of q and parameters θ_i:

$$\tilde{z}_t = \underbrace{\phi_1 \tilde{z}_{t-1} + \cdots + \phi_p \tilde{z}_{t-p}}_{\text{autoregressive process}} - \underbrace{\theta_1 a_{t-1} - \cdots - \theta_q a_{t-q}}_{\text{moving average process}} + \underbrace{a_t}_{\text{noise at } t}$$

The outputs $\tilde{z}_t$ are the deviations from the process's mean. The autoregressive process computes a weighted sum of the previous deviations of the ARMA process itself, while the moving average process computes a weighted sum of the previous white noise. The white noise a_t perturbs the process at t.

Box et al. motivate the ARMA(p, q) process as a reasonable finite approximation of the general linear filter

$$\tilde{z}_t = a_t + \sum_{j=1}^{\infty} \psi_j a_{t-j}$$

which can represent any zero-mean purely nondeterministic stationary process [7, p. 48].

In fact, they derive both the autoregressive and the moving average part as finite truncations of the general linear filter: if $\sum_{j=0}^{\infty} |\psi_j^2| < \infty$, the linear filter can be seen

both as an infinite limit of the autoregressive process as well as of the moving average process, as for appropriate π_j:

$$\tilde{z}_t = a_t + \sum_{j=1}^{\infty} \psi_j a_{t-j} = a_t + \sum_{j_1=1}^{\infty} \pi_j \tilde{z}_{t-j}.$$

A moving average process ARMA$(0, 1)$ with horizon $q = 1$ can only be represented by an infinite autoregressive process and vice-versa, so the autoregressive and moving average part of an ARMA(p, q) process have a complementary effect [7, p. 53].

A requirement for an ARMA model is that it is stationary, meaning that its mean is zero. A linear filter is stationary and can be represented by an ARMA model if $\sum_{j=0}^{\infty} |\psi_j| < \infty$ for the corresponding linear filter process formed by ψ_i as above. This requirement can be checked by computing the roots of a characteristic polynomial and testing whether they are not outside the unit circle [7, p. 9].

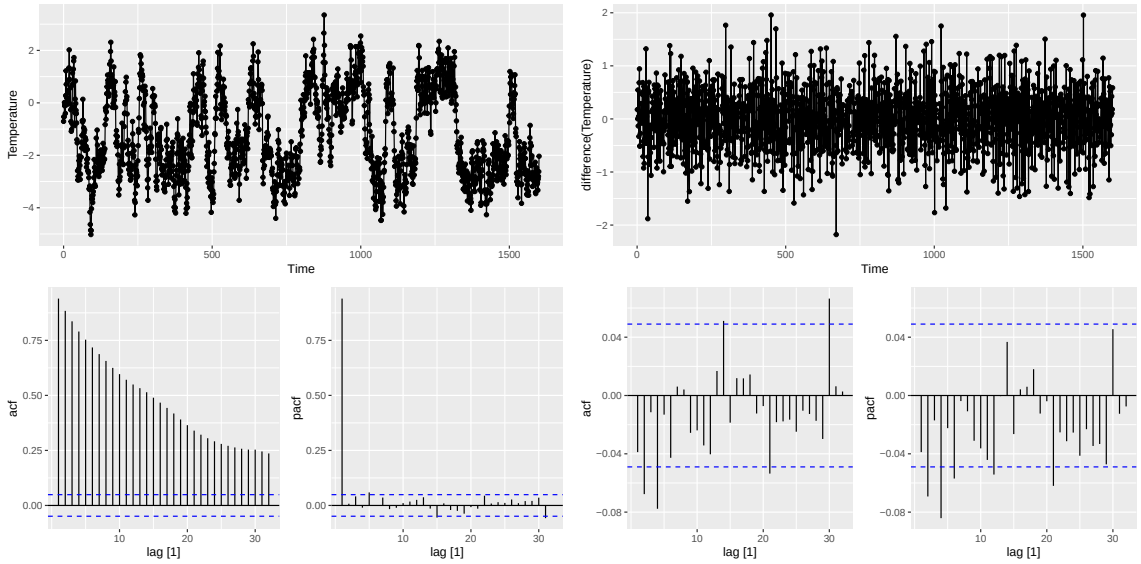
Autoregressive integrated moving average (ARIMA) processes form a generalization of ARMA processes and do not need to be stationary. Define the *difference operator* ∇ as $\nabla z_t = z_t - z_{t-1}$ and let $\nabla^d z_t$ represent this operator applied d times. For instance, $\nabla^2 z_t = \nabla z_t - \nabla z_{t-1} = z_t - 2z_{t-1} + z_{t-2}$. An ARIMA$(p, q, d)$ process [7, p. 11] is defined as

$$z_t = \underbrace{\phi_1 \nabla^d z_{t-1} + \cdots + \phi_p \nabla^d z_{t-p}}_{\text{integrated autoregressive process}} - \underbrace{\theta_1 a_{t-1} - \cdots - \theta_q a_{t-q}}_{\text{integrated moving average process}} + \underbrace{a_t}_{\text{noise at } t}.$$

An ARIMA process is the dth sum (or integration) of its corresponding ARMA process [7, p. 11]. As such, an ARIMA process's dth difference is stationary. As this is not the case for our simple multistable model, this assumption is one reason why we would not expect an ARIMA process to be able to model a multistable time series.

6.1.2 Fitting ARIMA Models

To fit an ARIMA(p, d, q) process to data, one must first choose the constants $p, q,$ and d. Box et al. call this *model identification*. d is identified by computing the dth difference of the data for a range of values in the hope of finding a stationary time series. From the differenced data, suitable p and q are identified. In the classical procedure, this identification process mainly works through analysis of the time series's sample autocorrelation and sample partial autocorrelation functions, although there are alternatives. Having obtained $p, q,$ and the dth difference of the time series, the second step is *model estimation* (fitting) of an ARMA(p, q) process [7, pp. 179–185].



(a) Autocorrelation of data, $d = 0$ (b) Autocorrelation of differenced data, $d = 1$

Figure 49: EBM: Autocorrelations for choosing d

As models with few parameters, ARIMA processes do not benefit from extremely long time series or lots of data. They perform better on coarse data sets [19, ch. 13.7]. We use a long time series sampled from the energy balance model with $\Delta t = 1$ and $t_{\text{train}} = 1600$, which should be more than enough data that is not too fine-grained.

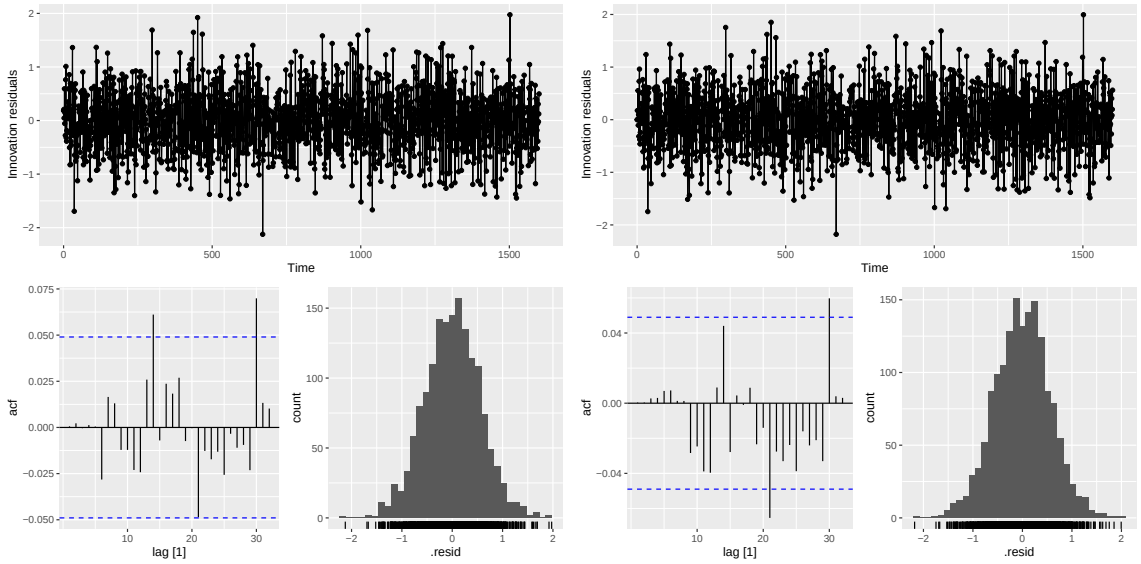
To find d, we must compute first differences until we arrive at something that looks like white noise. For the energy balance model's data, this is already the case with $d = 1$, as seen in Figure 49.

We use the Hyndman-Khandakar algorithm as described in [19, ch. 9.7], searching for models with $d \in \{0, 1, 2\}$. Interestingly, the model with the lowest (best) AICc value is the ARIMA$(1, 0, 4)$ model, i.e., the ARMA$(1, 4)$, which is also what the algorithm chooses when given no constraints. The selected models are:

```
                  nodiff          diff1          diff2
                 <model>        <model>        <model>
1 <ARIMA(1,0,4) w/ mean> <ARIMA(0,1,6)> <ARIMA(6,2,0)>
```

The library `fable` also provides information (σ^2, log-likelihoods, and various information criteria):

```
  .model sigma2 log_lik   AIC  AICc   BIC
  <chr>   <dbl>   <dbl> <dbl> <dbl> <dbl>
1 nodiff  0.306  -1321. 2656. 2657. 2694.
2 diff1   0.311  -1333. 2680. 2680. 2718.
3 diff2   0.371  -1473. 2961. 2961. 2998.
```



(a) Residuals for ARIMA$(1, 0, 4)$ (b) Residuals for ARIMA$(0, 1, 6)$

Figure 50: EBM: Residuals for fit ARIMA models

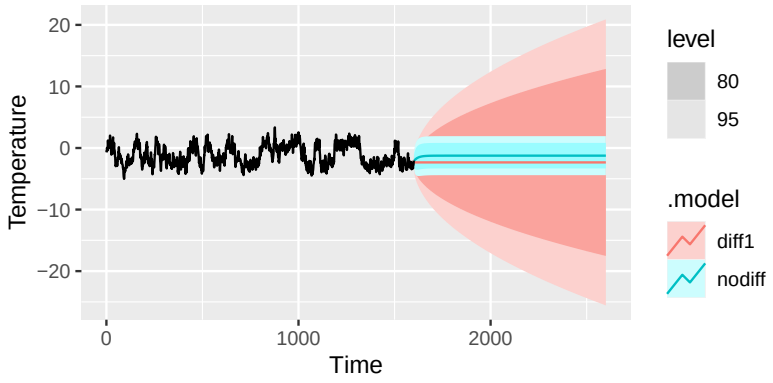


Figure 51: EBM: Forecasts from two ARIMA models

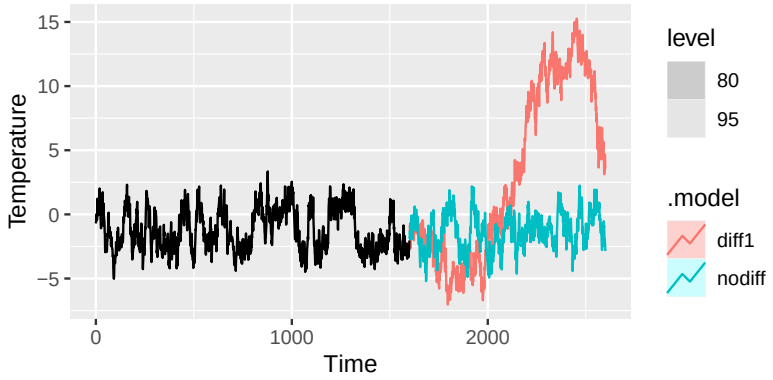


Figure 52: EBM: Simulations from two ARIMA models

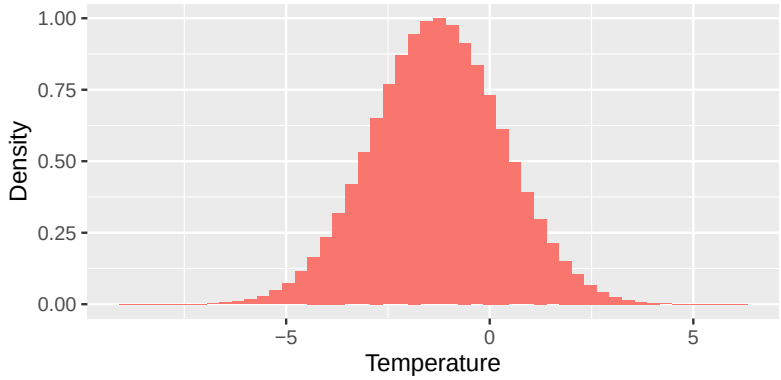


Figure 53: EBM: Histogram of the ARIMA$(1, 0, 4)$ model

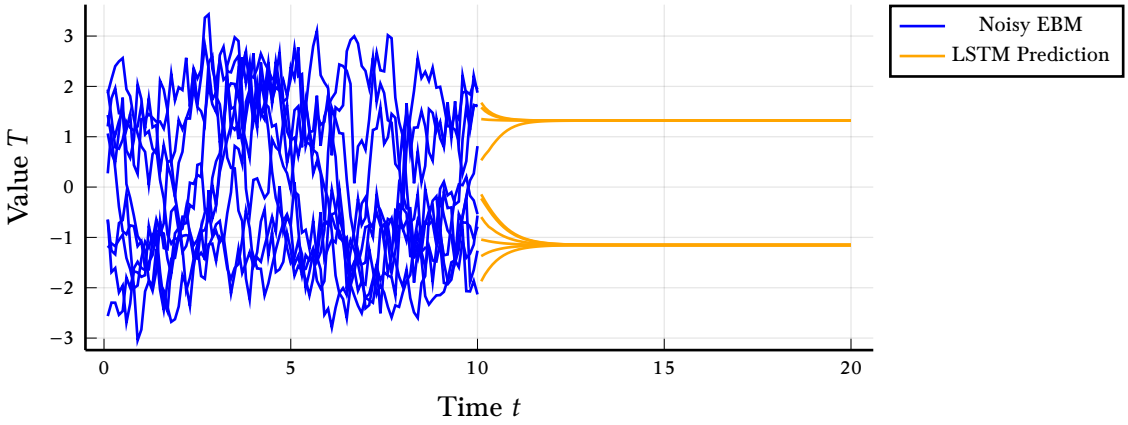


Figure 54: Autoregressive LSTM predictions

We now focus on the models with $d \in \{0, 1\}$, as these score much better. An important step is to check whether the residuals look like white noise, which we confirm in Figure 50. We forecast the time series of these models in Figure 51 and display a single simulation each in Figure 52. The ARIMA$(0, 1, 6)$ model ("diff1" in the plot) produces a trajectory that immediately seems implausible, and the confidence interval diverges. The ARIMA$(1, 0, 4)$ ("nodiff") produces a plausible confidence interval, but as it is just an ARMA model, it can only represent stationary processes. This becomes clear when plotting its histogram as we do in Figure 53, which is Gaussian. Thus, the ARMA model cannot model a bistable data set, and the ARIMA model would need some concept of an attractor, which it does not have.

6.2 Recurrent Neural Networks

ARIMA models are stochastic with no machine learning component, so to explore the other extreme, we try a machine learning method without stochastic compo-

nents. Our machine learning method should ideally be autoregressive (taking its own past outputs as input), like an ARIMA model, so that it can forecast over arbitrary timespans. Autoregressive neural networks are usually called recurrent neural networks (RNNs) [41].

The simplest way to build an RNN is to repeat the execution of a feedforward neural network n times, but this leads to issues like vanishing gradients. Long short-term memory (LSTM) [17] is a successful RNN architecture that solves these issues.

We train an LSTM with a running memory size of 100 and use a feedforward neural network to project the running memory to one dimension. As the training algorithm, we use *teacher forcing*, which always starts with the ground truth, lets the LSTM make a prediction, scores that prediction, and then starts with the next ground truth again [41]. This setup is not generative: we "score the prediction" by calculating the squared differences between the prediction and the ground truth.

We train on time series snippets of 20 steps. The testing and training sets have 100 time series each, with $\Delta t = 0.1$ and $t_{\text{train}} = 64$. We use the ADAM optimizer with the learning rate $\eta = 0.01$ and train on ten epochs; after five epochs, the training and testing losses no longer decrease.

We predict 100 steps for the trained LSTM based on 100 steps of ten time series in Figure 54. The LSTM has learned the bifurcation, which is the best it can do in this non-generative setup because it does not have any access to noise and because we scored the squared distances in training.

What would the results look like if we added noise into the RNN at each step and trained it in a generative framework? It would probably work quite well. Such a setup would be similar to a neural SDE in a generative framework. The differences are that the RNN would output the next value instead of its derivative and that the RNN can only be formulated in its discretized form. The advantage of the neural SDE is that its representation is in phase space, allowing for better explainability of the mechanisms behind the stochastic process. This also means that a neural SDE cannot represent a discontinuity in drift, whereas an RNN could jump to any value. Additionally, a neural SDE can be arbitrarily solved, whereas the randomized RNN can only be solved with a single time interval Δt fixed before training.

7 Julia Implementation

For this thesis, we implemented latent neural SDEs, a testing framework, and a `Pluto.jl` notebook with `slurm` support in Julia based on the `DifferentialEquations.jl` [32] ecosystem and the `Lux.jl` [30] library. A tutorial on how to use the package is included in Section A. The package works, but unfortunately, it is currently slow because there does not exist a competitive implementation of the gradients as discussed in [25] in Julia, which we discovered during the implementation process.

Discretize-then-optimize gradients are not functional in Julia because current AD packages like `Zygote.jl` cannot differentiate the general solver implementations that `DifferentialEquations.jl` provides. On the horizon is `Enzyme.jl` [28], which might work soon. The library `torchsde` implements the solvers as a chain of `pytorch` operations, which is why autodifferentiation works without further trouble.

Optimize-then-discretize gradients are implemented in the package `StochasticDiffEq.jl`, but they currently scale in $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n)$ (as in `torchsde`) where n is the number of dimensions in the latent space, which is not acceptable when training with large batch sizes. There is also no GPU support as a result of this. The quadratic scaling comes from a crucial optimization in the implementation of [25] that is not discussed in the paper and was previously unknown to the Julia package maintainers. We explain the issue in this section and describe how one would fix it in the Julia library but adjourn this task in the constraints of this thesis. Implementing the fast gradient computations will make our latent neural SDE package usable for large models with large batch sizes.

7.1 The Continuous Adjoint Method

We must first explain the continuous stochastic adjoint (optimize-then-discretize, backsolve) method to understand the difference between the Python and Julia adjoints. Consider an SDE

$$du(t) = f(u(t), t)dt + g(u(t), t) \circ dB_t \qquad \text{(forward)}$$

with diagonal diffusion g in the timespan $[0, T]$ and a loss function $\mathscr{L} : \mathbb{R}^{d_u} \to \mathbb{R}$ on $u(T)$. The circle $\circ$ is a notation to indicate that the SDE is Stratonovich (as opposed to Îto).

The adjoint $v(t) = \frac{d\mathscr{L}(u(T))}{du(t)}$ of Equation (forward) is

$$dv(t) = -v(t)\frac{\partial f}{\partial u}(u(t), t)dt - \sum_{k=1}^{d_w} v(t)\frac{\partial g_k}{\partial u}(u(t), t) \circ dB_t^k. \tag{adjoint}$$

It is computed backward from T to 0, with the Brownian motion B_t and the values $u(t)$ as computed in the forward pass, and the initial condition $v(T) = \frac{d\mathscr{L}(u(T))}{du(T)}$. The desired gradient is $v(0) = \frac{d\mathscr{L}(u(T))}{du(0)}$ [23, 25].

For example, consider the SDE

$$du(t) = \left(\begin{pmatrix} p & p^2 \\ 0 & -p \end{pmatrix} u(t) + \begin{pmatrix} 3 \\ 2 \end{pmatrix}\right) dt + \begin{pmatrix} p \cdot u(t)_1 & 0 \\ 0 & u(t)_2 \end{pmatrix} \circ dB_t$$

with the initial condition $u(0) = (x_1, x_2)^T$ and the loss function $\mathscr{L}(x) = \sum x$. By $u(t)_i$, we mean the ith component of the vector $u(t)$. This SDE has diagonal noise: the drift is a function yielding a two-dimensional vector, the diffusion is a function yielding a two-dimensional diagonal matrix. To compute the gradient w.r.t. p, which is an external parameter, we add a third dimension to the SDE that encodes the constant value of p by choosing the new initial condition $\hat{u}(0) = (x_1, x_2, p)^T$ and the new SDE

$$\hat{u}(t) = \left(\begin{pmatrix} \hat{u}(t)_3 & (\hat{u}(t)_3)^2 & 0 \\ 0 & -\hat{u}(t)_3 & 0 \\ 0 & 0 & 0 \end{pmatrix} \hat{u}(t) + \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix}\right) dt + \begin{pmatrix} \hat{u}(t)_3 \cdot \hat{u}(t)_1 & 0 & 0 \\ 0 & \hat{u}(t)_2 & 0 \\ 0 & 0 & 0 \end{pmatrix} \circ dB_t.$$

In this way, we can express gradients w.r.t. external parameters as gradients of the SDE w.r.t. itself and can work with the adjoint definition above, which only considers the latter. Although this makes gradients easier to work with mathematically, implementations treat parameters separately and do not add them as explicit dimensions into the SDE.

Suppose we have a forward solution of $\hat{u}(t)$ in the timespan $[0, T]$. We initialize the corresponding adjoint as defined in Equation (adjoint) with $\hat{v}(T) = \frac{d\mathscr{L}(\hat{u}(T))}{d\hat{u}(0)} = (1, 1, 0)$.

We now solve Equation (adjoint) backward with some SDE solver and consider the first evaluation of the drift and diffusion at point T. Let us compute the drift of the adjoint, i.e., the VJP (vector-Jacobian product) of the drift of the forward SDE, $\hat{v}(T)\frac{\partial \hat{f}}{\partial \hat{u}}(u(T), T)$. We name the variables of the forward solution $\hat{u}(T) = (x_1, x_2, p)$. To do this explicitly (by hand, not as the algorithm would do it), we can express the

forward SDE's drift as

$$\hat{f}((x_1, x_2, p), T) = \begin{pmatrix} px_1 + p^2 x_2 + 3 \\ -px_2 + 2 \\ 0 \end{pmatrix}$$

to compute the Jacobian w.r.t. (x_1, x_2, p):

$$\frac{\partial \hat{f}}{\partial \hat{u}}((x_1, x_2, p), T) = \begin{pmatrix} p & p^2 & 2px_2 + x_1 \\ 0 & -p & -x_2 \\ 0 & 0 & 0 \end{pmatrix}.$$

From this Jacobian, we can compute the VJP by vector-matrix multiplication with $\hat{v}(T)$. Autodifferentiation software *does not* compute the Jacobian matrix as we just did, but provides an interface for an efficient computation of the VJP $\hat{v}(t)\frac{\partial \hat{f}}{\partial \hat{u}}(u(t), t)$ given the input vector $\hat{v}(t)$ and concrete values of $\hat{u}(t) = (x_1, x_2, p)$ and t.

Now we compute the diffusion of Equation (adjoint). The difference between the Python and Julia implementations can be found in the implementation of this computation. Remember that the diffusion of the forward SDE is a function yielding a diagonal matrix, not a vector like the drift. Let us formulate $\hat{g}$ with x_1, x_2, p as above:

$$\hat{g}((x_1, x_2, p), T) = \begin{pmatrix} p \cdot x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Now we compute the Jacobians *for each dimension*:

$$\frac{\partial \hat{g}_1}{\partial \hat{u}}((x_1, x_2, p), T) = \begin{pmatrix} p & 0 & x_1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$\frac{\partial \hat{g}_2}{\partial \hat{u}}((x_1, x_2, p), T) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$\frac{\partial \hat{g}_3}{\partial \hat{u}}((x_1, x_2, p), T) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

We compute the VJPs and sum such that ith matrix is perturbed by B_t^i, leaving us

with the adjoint's diffusion

$$\hat{G}((x_1, x_2, p), T) = \begin{pmatrix} \hat{v}(t)_1 \cdot p & 0 & \hat{v}(t)_3 \cdot x_1 \\ 0 & \hat{v}(t)_2 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

7.2 Differences in the Adjoint Implementation

The Julia implementation in `DifferentialEquations.jl` computes $G(u(t), t)$ *explicitly* as an $n \times n$ matrix, where n is the number of dimensions of the forward SDE. This is not done for the external parameters because they are handled separately. The Julia implementation creates a function computing the number $g_i(u(t), t)$ and computes the VJP $\frac{\partial g_i}{\partial u}(u(t), t)$ for each $i \in \{1, \dots, n\}$. In total, the implementation runs in $\mathcal{O}(n^2)$ with n VJPs per solver step.

The Python implementations in `torchsde` and `diffrax` compute the same diffusion term with just one VJP and runtime in $\mathcal{O}(n)$ by a practical trick that is not described in the paper (although there is a comment that hints at the possibility of such an implementation in the appendix of [25]). Consider an SDE solver that has picked $\beta_t \sim \mathcal{N}(0, Q\Delta t)$ as the white noise sample at t. In practice, the Julia implementation first computes $G(u(t), t)$ and then computes $G(u(t), t) \cdot \beta_t$. The Python implementations simply ask the AD tool to compute the VJP of $\frac{\partial (g \cdot \beta_t)}{\partial u}(u(t), t)$, yielding the result $G(u(t), t) \cdot \beta_t$. $G(u(t), t)$ is never computed, saving us the $\mathcal{O}(n^2)$ operation.

Why does this work? We finish our example and then provide a proof. For our forward SDE, we have

$$\hat{g}((x_1, x_2, p), T) \cdot \beta_T = \begin{pmatrix} x_1 \cdot p \cdot \beta_T^1 \\ x_2 \cdot \beta_T^2 \\ 0 \end{pmatrix}.$$

The Jacobian is

$$\frac{\partial (\hat{g} \cdot \beta_T)}{\partial \hat{u}}((x_1, x_2, p), T) = \begin{pmatrix} p \cdot \beta_T^1 & 0 & x_1 \cdot \beta_T^1 \\ 0 & \beta_T^2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

and thus we have

$$\underbrace{\hat{v}(T)\frac{\partial(\hat{g}\cdot\beta_T)}{\partial\hat{u}}(u(T),T)}_{\text{Python implementations}} = \underbrace{\left(\sum_{i=1}^{d_w}\hat{v}(T)\frac{\partial\hat{g}_i}{\partial\hat{u}}(u(T),T)\right)\cdot\beta_T}_{\text{Julia implementation}}.$$

Proof. We have

$$\left(\sum_{i=1}^{d_w} v(t)\frac{\partial g_i}{\partial u}\right)\cdot\beta_t = v(t)\sum_{i=1}^{d_w}\frac{\partial g_i\beta_t}{\partial u} = v(t)\frac{\partial\sum_{i=1}^{d_w} g_i\beta_t}{\partial u} \overset{(*)}{=} v(t)\frac{\partial(g\cdot\beta_t)}{\partial u}.$$

For the step $(*)$, we need the assumption that the diffusion g is diagonal. □

7.3 How to Implement Scalable Gradients in Julia

Unfortunately, because of the solver implementations in the `DifferentialEquations.jl` library, it is nontrivial to implement an adjoint that only computes the VJP $\frac{\partial(g\cdot\beta_t)}{\partial u}(u(t),t)$. To illustrate this with an example, we compare the implementation of the Euler-Maruyama (EM) method for Îto SDEs. First, consider the `torchsde` implementation:

Listing 1: shortened `torchsde` implementation of EM method

```
def step(self, t0, t1, y0):
    dt = t1 - t0
    I_k = self.bm(t0, t1)

    # f(u(t)) is computed directly
    f = self.sde.f(t0, y0)

    # g(u(t))dB_t is computed abstractly
    g_prod = self.sde.g_prod(t0, y0, I_k)

    y1 = y0 + f * dt + g_prod
    return y1, ()
```

In the definition of the adjoint in `torchsde`, we can simply override the `g_prod` function to compute $\frac{\partial(g\cdot\beta_t)}{\partial u}(u(t),t) = G(u(t),t)\cdot\beta_t$. Now consider the Julia implementation:

Listing 2: shortened `StochasticDiffEq.jl` implementation of EM method

```
@muladd function perform_step!(integrator,cache::EMCache)
    @unpack tmp,rtmp1,rtmp2 = cache
    @unpack t,dt,uprev,u,W,P,c,p = integrator
```

```
    # f(u(t)) is computed directly
    integrator.f(rtmp1,uprev,p,t)
    @.. u = uprev + dt * rtmp1

    # g(u(t)) is computed directly
    integrator.g(rtmp2,u_choice,p,t)

    # g(u(t))dB_t is computed using result
    mul!(rtmp1,rtmp2,W.dW)

    @.. u += rtmp2
end
```

This implementation expects a result of `g` (which is the adjoint's diffusion G in the case of adjoint) and subsequently computes $G \cdot \beta_t$ using `mul!`. There is no generic interface like `g_prod`, but we want to avoid computing G. Apart from changing the implementation of every SDE solver, there are possible workarounds that leverage Julia's dynamic type system: we could define a struct that contains all information and is returned from the call to `g`. We then create a new implementation of `mul!` involving this struct and the noise `W.dW`, which computes the VJP.

We adjourned this implementation task because by the time we figured out the exact issue, we already moved to Python for the experiments. We were also uncertain whether using a custom struct in the solvers breaks any assumptions. The library maintainers prefer the more explicit solution that involves adding a `gprod`-like interface to every SDE solver.

To demonstrate the practical speedup, we use both methods to compute the VJP on a 10000-dimensional SDE with four parameters in this minimal example. A 10000-dimensional SDE is common in machine learning with neural SDEs: with batch gradient descent, thousands of simulations are run in parallel with a couple of dimensions each.

Listing 3: both methods in Julia

```
using Zygote, LinearAlgebra, Random, BenchmarkTools

# Parameters and number of dimensions.
p = rand(4)
m = 10000

# Arbitrarily chosen diagonal diffusion. Note this is not a diagonal matrix
# explicitly, but a vector representing a diagonal matrix, as is usual in
```

```
# SDE implementations.
function g(u, p, t)
    [p[1] * x - p[2] * t + p[3] * p[4] * t * x * x for x in u]
end

# Current point of the SDE u(t).
u = rand(m)
# Propagated gradient v(t).
v = rand(m)
# Time t.
t = rand()
# Sample from white noise beta.
beta = rand(m)

# Compute the VJP without the optimization:
_, back1 = Zygote.pullback(u, p) do u, p
    g(u, p, t)
end
out = @btime [back1(x) for x in eachcol(Diagonal(v))]

dv = stack(first.(out))
dparams = stack(last.(out))

# Compute the VJP with the optimization:
_, back2 = Zygote.pullback(u, p) do u, p
    g(u, p, t) .* beta
end
out2 = @btime back2(v)

resv = first(out2)
resparams = last(out2)

# Ensure that both implementations return the same result.
println(isapprox(dv * beta, resv)) # => true
println(isapprox(dparams * beta, resparams)) # => true
```

Executing this script in the Julia REPL on an M1 MacBook gives a runtime of 36.189s with 78.24 GiB of allocations for the unoptimized version and 2.530ms with 8.24 MiB of allocations for the optimized version according to the `@btime` invocations.

8 Conclusion and Future Work

Paleoclimatologists use stochastic multistable models to uncover fundamental mechanisms of our earth's long-term climate by fitting them to time series data. Our research aim was to fit stochastic dynamical models to data generated by multistable systems automatically, in contrast to manual estimation, as usually performed in paleoclimatology. We successfully trained latent SDEs, generative models based on neural SDEs, to match the statistics of bistable energy balance models and FitzHugh-Nagumo models, stochastic multistable systems standard in paleoclimatology.

We found that latent SDEs could directly model tipping points induced deterministically, for instance, from a limit-cycle FitzHugh-Nagumo model. Latent SDEs could not directly model stochastic tipping points because they produced tipping rates that were too low. We identified the cause of this problem and suggested a solution by adding another tunable hyperparameter.

To show that the problem of synthesizing a model from multistable data is not trivial, we showed that ARIMA models or non-generative RNNs cannot match the data's dynamics. Finally, we identified an undocumented trick in the Python implementations of neural SDEs that has to be implemented in Julia to achieve scalable gradients, as the current Julia implementation scales in the square of the latent space dimensions.

As the combination of neural SDEs and paleoclimatology is new, there are opportunities for future work. Applying neural SDEs to real-world paleoclimate data would be a big next step. Latent SDEs can, in theory, also predict and perform out-of-distribution detection, which has not yet been investigated. SDE-GANs are a promising alternative to latent SDEs because they promise to automatically match a dataset's drift and diffusion components without an explicit tuning parameter. Updating the Julia implementation of neural SDEs would open up efficient training of SDEs to a broader community of scientists.

Appendix A Julia Package

For this thesis, we implemented latent neural SDEs in the `LatentSDE.jl` Julia package. Once the issue described in Section 7.3 is fixed in the Julia libraries, this package can train latent neural SDEs efficiently. We provide a short example of how to use the library. We also have a companion repository with all of our experiments and tests. An overview of all repositories is in Section B.

For many use cases, the user can call the `StandardLatentSDE` constructor to avoid creating every network manually. Here, we are constructing a latent neural SDE with a two-dimensional latent space, where the training will happen with an Euler-Heun solver with timespan from zero to one with 20 data points:

```
julia> using LatentSDE, DifferentialEquations
julia> latent_sde = StandardLatentSDE(EulerHeun(), (0.0, 1.0), 20, latent_dims=2)
LatentSDE(
    initial_prior = Dense(1 => 4),      # 8 parameters
    initial_posterior = Dense(16 => 4),  # 68 parameters
    drift_prior = Chain(
        layer_1 = Dense(2 => 64, tanh_fast),  # 192 parameters
        layer_2 = Dense(64 => 64, tanh_fast),  # 4_160 parameters
        layer_3 = Dense(64 => 2, tanh_fast),  # 130 parameters
        layer_4 = Scale((2,)),          # 4 parameters
    ),
    drift_posterior = Chain(
        layer_1 = Dense(18 => 64, tanh_fast),  # 1_216 parameters
        layer_2 = Dense(64 => 64, tanh_fast),  # 4_160 parameters
        layer_3 = Dense(64 => 2, tanh_fast),  # 130 parameters
        layer_4 = Scale((2,)),          # 4 parameters
    ),
    diffusion = Parallel(
        layer_1 = Chain(
            layer_1 = Dense(1 => 16, tanh_fast),  # 32 parameters
            layer_2 = Dense(16 => 1, sigmoid_fast),  # 17 parameters
        ),
        layer_2 = Chain(
            layer_1 = Dense(1 => 16, tanh_fast),  # 32 parameters
            layer_2 = Dense(16 => 1, sigmoid_fast),  # 17 parameters
        ),
    ),
    encoder_recurrent = Recurrence(
        cell = GRUCell(1 => 16),        # 880 parameters, plus 1
    ),
    encoder_net = Dense(16 => 16),      # 272 parameters
    projector = Dense(2 => 1),          # 3 parameters
```

```
)         # Total: 11_325 parameters,
          #        plus 1 states, summarysize 288 bytes.
```

To replace one of these networks, just provide them as keyword argument. There are also keyword arguments for the network sizes. Here, we are giving the projector a tanh activation and providing a smaller hidden size for the prior:

```
julia> StandardLatentSDE(EulerHeun(), (0.0, 1.0), 20,
            prior_size=32, projector=Lux.Dense(2 => 1, tanh))
```

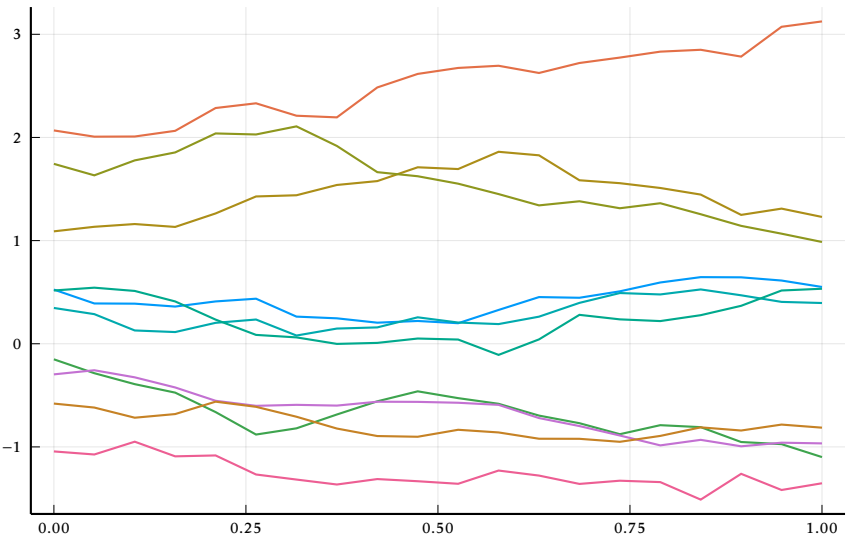
As the package is based on `Lux.jl`, we need to initialize the Latent SDE as a Lux model:

```
julia> using Lux, Random, ComponentArrays
julia> rng = Xoshiro()
julia> ps_, st = Lux.setup(rng, latent_sde)
julia> ps = ComponentArray{Float64}(ps_)
```

The `Lux.setup` function creates the parameters and state of the model. We have to create a `ComponentArray` from the parameters because sensitivity methods will only work with array-like types, but `ps_` is a dictionary. A `ComponentArray` is a flat array of parameters with an overlay that keeps the assignment of parameters to networks.

We can already plot a few samples from the prior:

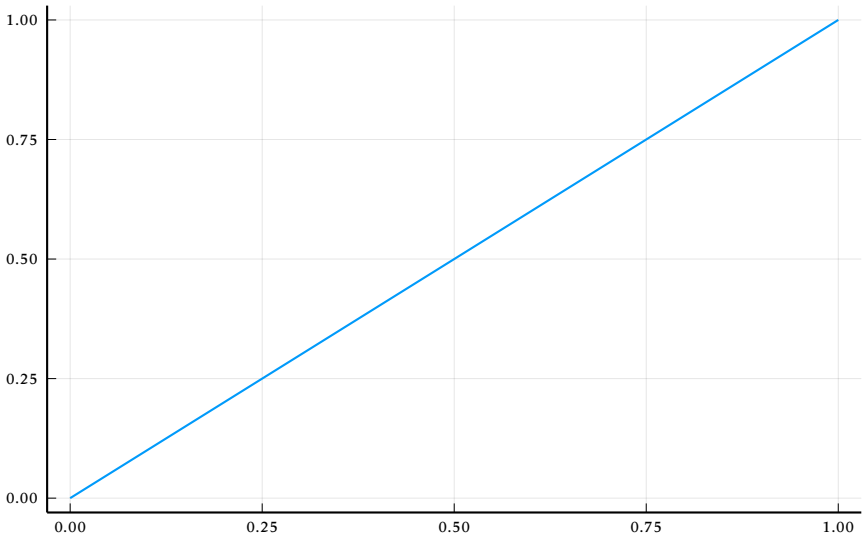
```
julia> plot(sample_prior_dataspace(latent_sde, ps, st, b=10))
```



To run the posterior, we need an input time series. Let us create one:

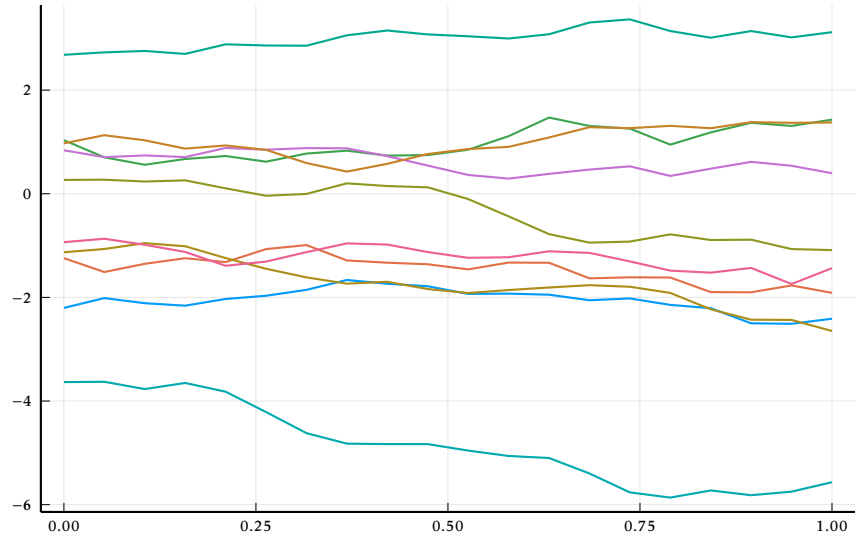
```
julia> t = collect(range(0.0, 1.0, 20))
```

```
julia> ts = Timeseries(t, [[[x] for x in t]])
julia> plot(ts)
```



We plot ten separate posteriors by repeating the input time series ten times:

```
julia> repeated_ts = repeat_ts(10, ts)
julia> _, posterior_dataspace, _, _, _ = latent_sde(repeated_ts, ps, st)
julia> plot(Timeseries(t, posterior_dataspace))
```



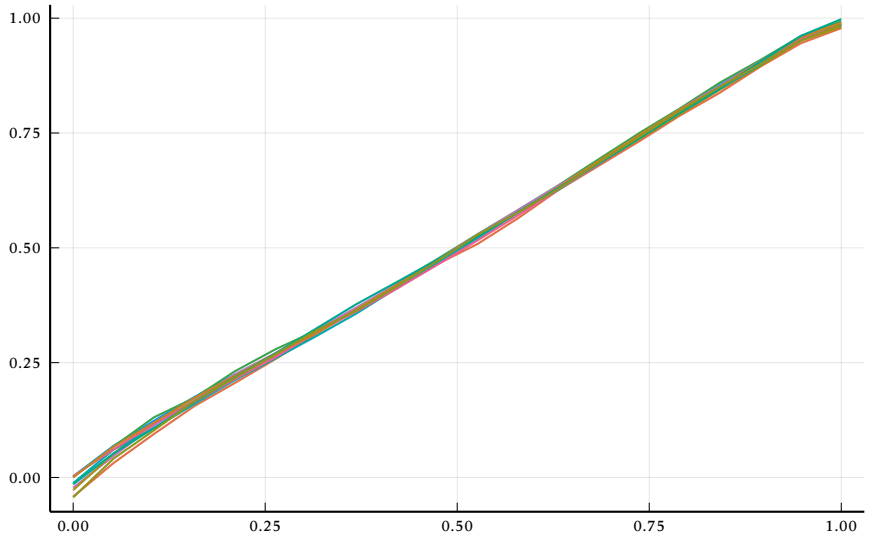
Now we train the latent SDE on the data. This can be done with the modular Julia approach. We choose the libraries `Optimisers.jl` for the ADAM algorithm and `Zygote.jl` for the autodifferentiation. As the loss function, we use the built-in loss function that uses a log-likelihood.

```
julia> using Optimisers, Zygote
julia> loss(ps) = sum(loss(latent_sde, repeat_ts(10, ts), ps, st, 0.01))
```

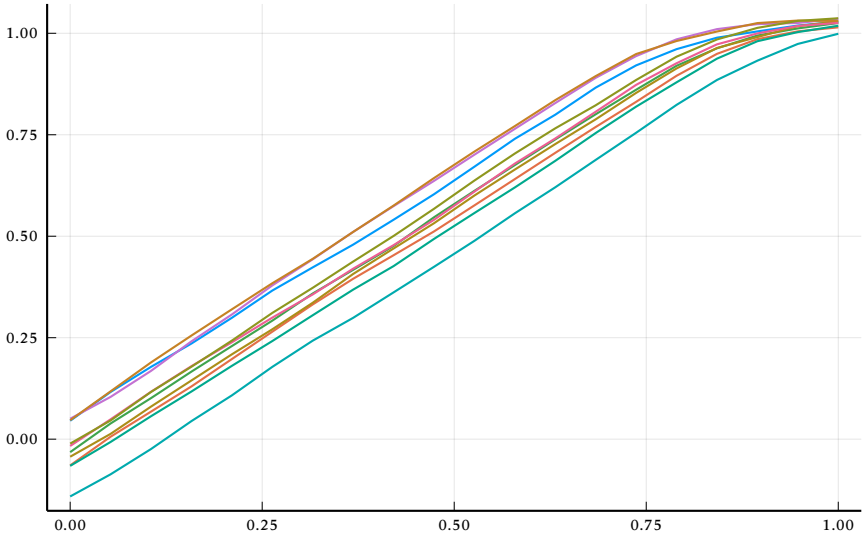
```
julia> loss(ps)
1.6216436199468267e6
julia> function train()
    for iteration in 1:200
       l, dps = Zygote.withgradient(loss, ps)
       println("Loss: $l")
       Optimisers.update!(opt_state, ps, dps[1])
   end
end
julia> opt_state = Optimisers.setup(Optimisers.Adam(0.1), ps)
julia> train()
julia> opt_state = Optimisers.setup(Optimisers.Adam(0.001), ps)
julia> train()
```

After training, let's plot prior and posterior:

```
julia> loss(ps)
-661.5151548380488
julia> _, posterior_dataspace, _, _, _ = latent_sde(repeat_ts(10, ts), ps, st)
julia> plot(Timeseries(t, posterior_dataspace))
```




```
julia> plot(sample_prior_dataspace(latent_sde, ps, st, b=10))
```



The trained prior has a slightly broader initial condition than the data, but the dynamics are accurate. More training might be necessary to align the initial condition.

Appendix B Code Archives

We implemented latent neural SDEs in Julia, although the experiments in Section 5 are in Python because of the performance issues described in Section 7. We created a repository with all Julia notebooks, plots, and analysis scripts, which also contains the experiment in Section 6.2[1]. We also created a library from this repository, which only contains the documented latent neural SDE implementation[2]. For Section 5, we wrote an extension to the `torchsde` library with models and analysis[3]. The experiments in Section 6.1 were performed in R[4].

All packages are released under the GPLv3 license. As no link on the Internet lasts forever, we provide a ZIP file with a checkout of these repositories on the Internet Archive as another backup[5]. We omit the dependencies, but the `pipenv` and `Pkg.jl` build configurations are reproducible.

[1] `https://github.com/glatteis/NeuralSDEExploration`
[2] `https://github.com/glatteis/LatentSDE.jl`
[3] `https://github.com/glatteis/multistablesde`
[4] `https://github.com/glatteis/multistablearima`
[5] `https://archive.org/details/multistable-neural-sdes`

References

A short comment concerning the references: In the machine learning community, it is not unusual that important results such as [24] are never published and only available as a preprint. In these cases, we cite the preprint on arXiv.

[1] KK Andersen, N Azuma, JM Barnola, M Bigler, P Biscaye, N Caillon, J Chappellaz, HB Clausen, D Dahl-Jensen, H Fischer, et al. "North Greenland Ice Core Project: High-resolution record of Northern Hemisphere climate extending into the last interglacial period". In: *Nature* 431.7005 (2004), pp. 147–151.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks". In: *International conference on machine learning*. PMLR. 2017, pp. 214–223.

[3] Peter Ashwin, Sebastian Wieczorek, Renato Vitolo, and Peter Cox. "Tipping points in open systems: bifurcation, noise-induced and rate-dependent examples in the climate system". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 370.1962 (2012), pp. 1166–1184.

[4] Tamás Bódai, Valerio Lucarini, Frank Lunkeit, and Robert Boschi. "Global instability in the Ghil–Sellers model". In: *Climate Dynamics* 44 (2015), pp. 3361–3381.

[5] Niklas Boers. "Early-warning signals for Dansgaard-Oeschger events in a high-resolution ice core record". In: *Nature communications* 9.1 (2018), p. 2556.

[6] Sam Bond-Taylor, Adam Leach, Yang Long, and Chris G Willcocks. "Deep generative modelling: A comparative review of VAEs, GANs, normalizing flows, energy-based and autoregressive models". In: *IEEE transactions on pattern analysis and machine intelligence* (2021).

[7] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[8] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. "Neural ordinary differential equations". In: *Advances in neural information processing systems* 31 (2018).

[9] Carmen Chicone. *Ordinary differential equations with applications*. Springer Science & Business Media, 2006.

[10] Joshua Fagin, Ji Won Park, Henry Best, and Matthew O'Dowd. "Latent Stochastic Differential Equations for Modeling Quasar Variability and Inferring Black Hole Properties". In: *ICLR 2023 Workshop on Physics for Machine Learning*. 2023.

[11] Richard FitzHugh. “Impulses and physiological states in theoretical models of nerve membrane”. In: *Biophysical journal* 1.6 (1961), pp. 445–466.

[12] Klaus Fraedrich. “Catastrophes and resilience of a zero-dimensional climate system with ice-albedo and greenhouse feedback”. In: *Quarterly Journal of the Royal Meteorological Society* 105.443 (1979), pp. 147–167.

[13] Hao Fu, Chunyuan Li, Xiaodong Liu, Jianfeng Gao, Asli Celikyilmaz, and Lawrence Carin. “Cyclical annealing schedule: A simple approach to mitigating KL vanishing”. In: *preprint, arXiv:1903.10145* (2019).

[14] Michael Ghil. “Climate stability for a Sellers-type model”. In: *Journal of Atmospheric Sciences* 33.1 (1976), pp. 3–20.

[15] Ali Hasan, Joao M Pereira, Sina Farsiu, and Vahid Tarokh. “Identifying latent stochastic differential equations”. In: *IEEE Transactions on Signal Processing* 70 (2021), pp. 89–104.

[16] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. “beta-vae: Learning basic visual concepts with a constrained variational framework”. In: *International conference on learning representations*. 2016.

[17] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5 (1989), pp. 359–366.

[19] Rob Hyndman and G. Athanasopoulos. *Forecasting: Principles and Practice*. English. 3rd. Australia: OTexts, 2021.

[20] SJ Johnsen, HB Clausen, Willi Dansgaard, K Fuhrer, N Gundestrup, CU Hammer, Pl Iversen, J Jouzel, Bernhard Stauffer, and JP Steffensen. “Irregular glacial interstadials recorded in a new Greenland ice core”. In: *Nature* 359.6393 (1992), pp. 311–313.

[21] Patrick Kidger. “On neural differential equations”. PhD thesis. University of Oxford, 2021.

[22] Patrick Kidger, James Foster, Xuechen Li, and Terry J Lyons. “Neural SDEs as Infinite-Dimensional GANs”. In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 5453–5463.

[23] Patrick Kidger, James Foster, Xuechen (Chen) Li, and Terry Lyons. "Efficient and Accurate Gradients for Neural SDEs". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 18747–18761.

[24] Diederik P Kingma and Max Welling. "Auto-encoding variational Bayes". In: *preprint, arXiv:1312.6114* (2013).

[25] Xuechen Li, Ting-Kam Leonard Wong, Ricky TQ Chen, and David K Duvenaud. "Scalable gradients and variational inference for stochastic differential equations". In: *Symposium on Advances in Approximate Bayesian Inference*. PMLR. 2020, pp. 1–28.

[26] Johannes Lohmann and Peter D Ditlevsen. "A consistent statistical model selection for abrupt glacial climate changes". In: *Climate dynamics* 52.11 (2019), pp. 6411–6426.

[27] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

[28] William Moses and Valentin Churavy. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485.

[29] Bernt Øksendal. *Stochastic Differential Equations*. Springer, 2003.

[30] Avik Pal. "On Efficient Training & Inference of Neural Differential Equations". PhD thesis. Massachusetts Institute of Technology, 2023.

[31] Victor M Panaretos and Yoav Zemel. "Statistical aspects of Wasserstein distances". In: *Annual review of statistics and its application* 6 (2019), pp. 405–431.

[32] Christopher Rackauckas and Qing Nie. "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia". In: *Journal of Open Research Software* 5.1 (2017).

[33] Keno Riechers, Takahito Mitsui, Niklas Boers, and Michael Ghil. "Orbital insolation variations, intrinsic climate variability, and Quaternary glaciations". In: *Climate of the Past* 18.4 (2022), pp. 863–893.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[35] Simo Särkkä and Arno Solin. *Applied Stochastic Differential Equations*. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2019.

[36] Andreas Sommer. “Numerical methods for parameter estimation in dynamical systems with noise with applications in systems biology”. PhD thesis. 2017.

[37] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. “Score-based generative modeling through stochastic differential equations”. In: *preprint, arXiv:2011.13456* (2020).

[38] Alfonso Sutera. “On stochastic perturbation and long-term climate behaviour”. In: *Quarterly Journal of the Royal Meteorological Society* 107.451 (1981), pp. 137–151.

[39] Belinda Tzen and Maxim Raginsky. “Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit”. In: *preprint, arXiv:1905.09883* (2019).

[40] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006.

[41] Ronald J Williams and David Zipser. “A learning algorithm for continually running fully recurrent neural networks”. In: *Neural computation* 1.2 (1989), pp. 270–280.

[42] Winnie Xu, Ricky T. Q. Chen, Xuechen Li, and David Duvenaud. “Infinitely Deep Bayesian Neural Networks with Stochastic Differential Equations”. In: *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*. Vol. 151. Proceedings of Machine Learning Research. PMLR, 28–30 Mar 2022, pp. 721–738.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature