Proceedings of Machine Learning Research vol 145:568–597, 2021 2nd Annual Conference on Mathematical and Scientific Machine Learning

A deep learning method for solving Fokker-Planck equations

Jiayu Zhai ZHAI@MATH.UMASS.EDU
Department of Mathematics and Statistics, University of Massachusetts Amherst, Amherst, MA, 01002, USA

Matthew Dobson DOBSON@MATH.UMASS.EDU
Department of Mathematics and Statistics, University of Massachusetts Amherst, Amherst, MA, 01002, USA

Yao Li YAOLI@MATH.UMASS.EDU
Department of Mathematics and Statistics, University of Massachusetts Amherst, Amherst, MA, 01002, USA

Editors: Joan Bruna, Jan S Hesthaven, Lenka Zdeborova

Abstract

The time evolution of the probability distribution of a stochastic differential equation follows the Fokker-Planck equation, which usually has an unbounded, high-dimensional domain. Inspired by Li (2019), we propose a mesh-free Fokker-Planck solver, in which the solution to the Fokker-Planck equation is now represented by a neural network. The presence of the differential operator in the loss function improves the accuracy of the neural network representation and reduces the demand of data in the training process. Several high dimensional numerical examples are demonstrated.

Keywords: Stochastic differential equation, Monte Carlo simulation, invariant measure, coupling method, data-driven and machine learning methods

1. Introduction

Stochastic differential equations are widely used to model dynamics of real world problems in the presence of uncertainty (such as events driving stock markets) or in the presence many small forces whose origins are not all tracked (such as a solvent acting on a larger molecule). The instantaneous and cumulative effects of the noise on the dynamics can be visualized through the transient and invariant probability distribution of the solution process, respectively. These probability measures can be analytically described by the Fokker-Planck equations (also known as the Kolmogorov forward equation Kolmogoroff (1933)). It is well known for the Langevin and Smoluchowski equations that if the deterministic part of the stochastic differential equation is a gradient flow, the invariant measure is the Gibbs measure whose probability density function is explicitly given. However in general, the Fokker-Planck equation can only be solved numerically. Traditional numerical PDE solvers do not work well for Fokker-Planck equations due to both the lack of a suitable boundary condition and the curse of dimensionality (see Section 2 for detailed explanation). Although many novel methods are introduced to resolve this difficulty, solving high dimensional Fokker-Planck equation remains as a challenge.

With the rapid growth of accessibility of data and demand for its analysis in various application realms, such as computer vision, speech recognition, natural language processing, and game intelligence, machine learning methods prove their strong performance in representing these high dimensional models. Mathematicians have also made quite many efforts on proving the error estimates of neural network representations using function spaces like Sobolev spaces, Besov spaces, and Barron spaces Gühring et al. (2020); Petersen and Voigtlaender (2018); Suzuki (2019); E et al. (2019). Although these results are still far away from explaining their strong performance, the success of machine learning methods in modelling high dimensional models in big data applications

motivates their consideration as a numerical scheme for solving mathematics problems, particularly in partial differential equations. Among others, we highlight the references Sirignano and Spiliopoulos (2018); Beck et al. (2018, 2019); Han et al. (2018); Macris and Marino (2020); Raissi et al. (2019); Wu et al. (2018); Chen et al. (2020); Xu et al. (2020a) that are related to this work.

In Li (2019), the author proposed a novel data-driven solver to solve the Fokker-Planck equation. The key idea is to remove the reliance on the boundary conditions and construct a constrained optimization problem that uses the Monte Carlo simulation data as the reference. We find that a low accuracy Monte Carlo data is sufficient to guide the optimization problem to produce a highly accurate solution. And the solver has very high tolerance to spatially uncorrelated noise presented in the reference data (Monte Carlo data). This approach is still grid-based, so it (including its extension in Dobson et al. (2019)) does not work well for high dimensional problems. Motivated by the progress of applying artificial neural network to traditional computational problems, in this paper we propose a mesh-free version of the data-driven solver studied in Li (2019); Dobson et al. (2019). Similar to those works, the focus here is on the steady state Fokker-Planck equation as the invariant probability measure plays a very important role in applications. The case of time-dependent Fokker-Planck equation is analogous. All our algorithms can be applied to time-dependent problems with some minor modifications.

The key idea in this paper is to replace the constrained optimization problem studied in Li (2019) by an unconstrained optimization problem, as it is not easy to use neural networks to study the constrained optimization problem on a high dimensional hyperplane. We first propose the unconstrained optimization problem and prove the convergence of its minimizer to the true Fokker-Planck solution for the discrete case. Then we propose an analogous loss function that is trainable by artificial neural networks. Our further studies find that the Fokker-Planck operator $\mathcal{L}$ plays an important role in the training. It dramatically increases the tolerance of noisy simulation data and reduces the amount of simulation data used in the training. In general, we only need 10^2 to 10^4 "reference points" with probability densities on them to train the neural network. And the probability density function obtained by Monte Carlo simulation does not have to be very accurate. (see Section 3 for explanation and Section 4 for numerical demonstrations). The reduction of demand for simulation data is significant since the stochastic dynamical systems in applications usually have high dimensionality, whereas the training data collected from either Monte Carlo simulation or experiments has high cost. The use of the PDE operator $\mathcal{L}$ in the loss function to regularize the solution has some similarity with the situation of the Physics-Informed Neural Network (PINN) Raissi et al. (2019); Wu et al. (2018); Chen et al. (2020) and the Deep Galerkin Method (DGM) Sirignano and Spiliopoulos (2018). The difference is that both PINN and DGM still rely on the boundary condition, which is hard to get for the Fokker-Planck equations defined on unbounded domains. The Fokker-Planck solver presented in this paper follows the idea of the grid-based solver studied in Li (2019); Dobson et al. (2019). We only need some rough estimates of the probability density function at a relative small number of reference points in the interior of the numerical domain. The probability density can be obtained by either the direct Monte Carlo method or the conditional Gaussian high dimensional sampler Chen and Majda (2017, 2018). Similar to the grid-based version of our Fokker-Planck solver, significant spatially uncorrelated error of probability densities at reference points can be tolerated by the neural network (Figure 4). This makes our Fokker-Planck solver different from the PINN-based Fokker-Planck solver presented in Chen et al. (2020), especially for high dimensional problems. In comparison, Chen et al. (2020) uses Kullback–Leibler divergence

as a part of the loss function, which requires a numerical integration in each evaluation of the loss function.

In Section 2, we describe the problem setting, the unconstrained optimization we study, and the idea of using neural network representation. All training and sampling algorithms are studied in Section 3. In Section 4, we use several numerical examples to demonstrate the main feature of our neural network Fokker-Planck solver.

2. Preliminaries and motivation

2.1. Fokker-Planck equation and data-driven solver.

We consider the stochastic differential equation

$$d\boldsymbol{X}_t = f(\boldsymbol{X}_t)dt + \sigma(\boldsymbol{X}_t)d\boldsymbol{W}_t\,, \tag{2.1}$$

where f is a vector field in $\mathbb{R}^n$, σ is a coefficient matrix, and $\boldsymbol{W}_t$ is an n-dimensional white noise. The time evolution of probability density of the solution process $\boldsymbol{X}_t$ is characterized by the Fokker-Planck equation, which is also known as the Kolmogorov forward equation

$$u_t = \mathcal{L}u = -\sum_{i=1}^{n}(f_i u)_{x_i} + \frac{1}{2}\sum_{i,j=1}^{n}(\Sigma_{i,j}u)_{x_i x_j}\,, \tag{2.2}$$

where $u(\boldsymbol{x}, t)$ denotes the probability density function of the stochastic process $\boldsymbol{X}_t$ at time t, $\Sigma = \sigma^T\sigma$ is the diffusion coefficient, and subscripts t and x_i denote partial derivatives. In this paper, we focus on the invariant probability measure of (2.1), whose density function $\mathbb{R}^n \ni \boldsymbol{x} \mapsto u(\boldsymbol{x}) \in \mathbb{R}$ satisfies the stationary Fokker-Planck equation

$$\begin{cases} \mathcal{L}u = 0 \\ \int_{\mathbb{R}^n} u\, d\boldsymbol{x} = 1 \end{cases} \tag{2.3}$$

Throughout the present paper, we assume the existence and uniqueness of the solution to the stationary Fokker-Planck equation.

The Fokker-Planck equation is defined on an unbounded domain with the constraint $\int_\Omega u\, d\boldsymbol{x} = 1$. Since the numerical domain has to be bounded, it is not easy to give a suitable boundary condition to describe the "zero-boundary condition at infinity". In practice, one can assume a zero boundary condition on a domain that is large enough to cover all high density areas with sufficient margin. A classic computational method, e.g., finite element method, is then applied to find a non-trivial solution. One usually needs to solve a least square problem because of the constraint $\int_{\mathbb{R}^n} u\, d\boldsymbol{x} = 1$. In general, the computational cost of classical PDE solver is too high to be practical when $n \geq 3$. The other way to solve the Fokker-Planck equation is the Monte Carlo method, which uses the fact that the empirical distribution of a long trajectory converges to the solution to the steady state Fokker-Planck equation. The Monte Carlo method is very simple regardless of the boundary condition. One only needs to divide the numerical domain into lots of "bins", run a long trajectory of the equation (2.1), and count the number of samples in each bin. However, the solution from the Monte Carlo method is much less accurate.

In Li (2019), the author introduced a data-driven method that overcomes the drawbacks of the two aforementioned methods, so that one can solve the Fokker-Planck equation locally and does

not rely on the boundary condition any more. Later in Dobson et al. (2019), the authors proved the convergence of the method and improved the method by introducing a "blocked version" that uses a divide-and-conquer strategy (see for example Cormen et al. (2009)). Let $D \subset \mathbb{R}^n$ be the numerical domain. Assume there is a rectangular grid $\{x_i\}_{i=1}^{N^n}$ defined in D with N grid points on each dimension. The key idea of this data-driven method is to solve the optimization problem

$$\begin{aligned} \min_{\boldsymbol{u}} \quad & \|\boldsymbol{u}-\boldsymbol{v}\|_2 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{u}=\boldsymbol{0}, \end{aligned} \tag{2.4}$$

where $\boldsymbol{A} \in \mathbb{R}^{(N-2)^n \times N^n}$ is a discretization of the Fokker-Planck operator $\mathcal{L}$ on D without boundary condition, and $\boldsymbol{v} \in \mathbb{R}^{N^n}$ is a Monte Carlo approximation obtained by a numerical simulation of (2.1). An entry $\boldsymbol{v}_i$ of the vector $\boldsymbol{v}$ is the probability that a long trajectory stays in a small neighborhood of x_i, which is usually a low accuracy approximation of the invariant measure. Each row of matrix $\boldsymbol{A}$ is obtained by a discretization of the Fokker-Planck equation (using the finite difference method) at a interior point x_i. Matrix $\boldsymbol{A}$ only has $(N-2)^n$ rows but N^n columns because we do not know the boundary value. The motivation is that an inaccurate Monte Carlo solution can effectively replace the boundary value. The solution to the optimization problem (2.4) projects the Monte Carlo solution $\boldsymbol{v}$ to the null space of $\boldsymbol{A}$. The projection works as a "smoother" that not only dramatically removes the error term from the Monte Carlo approximation, but also pushes most error terms to the boundary of the domain. See the proof and discussion in Dobson et al. (2019) for details.

2.2. An alternative optimization problem.

To use artificial neural network approximations, we need to convert the optimization problem in equation (2.4) to an unconstrained optimization problem. If we use the penalty method with penalty parameter 1 for (2.4), we have a new optimization problem

$$\min_{\boldsymbol{u}} \quad \|\boldsymbol{A}\boldsymbol{u}\|_2^2 + \|\boldsymbol{u}-\boldsymbol{v}\|_2^2, \tag{2.5}$$

where $\boldsymbol{A}$ and $\boldsymbol{v}$ are the same as in equation (2.4). We claim that the new optimization problem (2.5) has a similar effect as the original one in (2.4).

To compare the result, we choose the numerical solution obtained by the finite difference method, denoted by $\boldsymbol{u}^*$, as the baseline, because we have $\boldsymbol{A}\boldsymbol{u}^* = \boldsymbol{0}$. See Appendix A for a more precise description of $\boldsymbol{u}^*$. Let $\bar{\boldsymbol{u}}$ be the minimizer of the optimization problem (2.5). Denote the error terms of the Monte Carlo simulation and the optimizer by $\boldsymbol{e} = \boldsymbol{v} - \boldsymbol{u}^*$ and $\boldsymbol{z} = \bar{\boldsymbol{u}} - \boldsymbol{u}^*$ respectively. Let $\boldsymbol{A} = \boldsymbol{A}_h$ where h is the grid size of discretization. We make the following assumptions to conduct the convergence analysis.

(A1) Random vector $\boldsymbol{e}$ has i.i.d. entries whose identical expectation and variance are 0 and ζ^2 respectively.

(A2) Let $\lambda_1^h, \cdots, \lambda_r^h$ be all nonzero eigenvalues of $\boldsymbol{A}_h^T\boldsymbol{A}_h$. Let $Q(h) = h^n \sum_{i=1}^r \left(\frac{1}{1+h^4\lambda_i^h}\right)^2$. We have $Q(h) \to 0$ as $h \to 0$.

Theorem 2.1 *If (A1) and (A2) hold, then*

$$\lim_{h\to 0} \frac{\mathbb{E}[\|\boldsymbol{z}\|^2]}{\mathbb{E}[\|\boldsymbol{e}\|^2]} = 0\,.$$

Remark: One needs to multiply the volume of n-dimensional grid box when calculating the discrete L^2 error. Hence the discrete L^2 error of $\boldsymbol{v}$ is $h^{n/2}\mathbb{E}[\|\boldsymbol{e}\|] = \text{const} \cdot \zeta$. Theorem 2.1 implies that the error of $\bar{\boldsymbol{u}}$ converges to zero as $h \to 0$.

Assumption **(A1)** assumes the error term $\boldsymbol{e}$ has i.i.d entries. This is because the error terms of Monte Carlo solutions have very little spatial correlation. See Figure 1 bottom left panel as an example of the spatial distribution of the error term of a Monte Carlo solution. The real Monte Carlo simulation has smaller error than that in Assumption **(A1)**, as the absolute error is smaller in the low density area. Assumption **(A2)** is due to technical reasons. It means nonzero eigenvalues of $\boldsymbol{A}_h$ shouldn't be extremely small. This is true for all examples that we have tested. But a rigorous proof of the eigenvalue distribution of $\mathbf{A}_h$ is very difficult and beyond the scope of the present paper. See Appendix A for more discussions.

3. Neural network train algorithms

In Theorem 2.1, we show that the solution to the unconstrained optimization problem (2.5) converges to the true solution of the Fokker-Planck equation. Since it is very difficult to do spatial discretization in high dimension, it is natural to consider the mesh-free version of the optimization problem (2.5), in which the variable u is represented by an artificial neural network.

3.1. Loss function.

Now let $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta})$ be an approximation of $\boldsymbol{u}$ that is represented by an artificial neural network with parameter $\boldsymbol{\theta}$. Inspired by equation (2.5), we work on the squared error loss function

$$L(\boldsymbol{\theta}) = \frac{1}{N^X}\sum_{i=1}^{N^X}(\mathcal{L}\tilde{u}(\boldsymbol{x}_i, \boldsymbol{\theta}))^2 + \frac{1}{N^Y}\sum_{j=1}^{N^Y}(\tilde{u}(\boldsymbol{y}_j, \boldsymbol{\theta}) - v(\boldsymbol{y}_j))^2 := L_1(\boldsymbol{\theta}) + L_2(\boldsymbol{\theta}), \tag{3.1}$$

with respect to $\boldsymbol{\theta}$, where $\boldsymbol{x}_i \in \mathbb{R}^n, i = 1, 2, \ldots, N^X$ and $\boldsymbol{y}_j \in \mathbb{R}^n, j = 1, 2, \ldots, N^Y$ are collocation points sampled from D, and $v(\boldsymbol{y}_j)$ is the Monte Carlo approximation from a numerical simulation of (2.1) at $\boldsymbol{y}_j$. This loss function (3.1) is in fact the Monte Carlo integration of the following functional

$$J(u) = \|\mathcal{L}u\|^2_{L^2(D)} + \|u - v\|^2_{L^2(D)}, \tag{3.2}$$

which can be seen as the continuous version of the discrete optimization problem (2.5).

The loss function (3.1) has two parts. The minimization of $L_1(\boldsymbol{\theta})$ is to generate parameters $\boldsymbol{\theta}^*$ that guides the neural network representation $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta}^*)$ to fit the Fokker-Planck differential equation $\mathcal{L}\tilde{u} = 0$ empirically at the training points $\boldsymbol{x}_i, i = 1, 2, \ldots, N^X$ (we use automatic differentiation here to generate derivatives of $\tilde{u}$ with respect to $\boldsymbol{x}$ using the same parameters $\boldsymbol{\theta}$). It works as a regularization mechanism such that the resultant neural network representation $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta}^*)$ approximates one of the infinitely many solutions of the Fokker-Planck equation without boundary conditions. Similar to Li (2019) and Dobson et al. (2019), the second part $L_2(\boldsymbol{\theta})$ of the loss function serves as a reference for the solution. It is the low accuracy Monte Carlo approximation that guides the neural network training process to converge to the desired solution, namely the one satisfying the stationary Fokker-Planck equation (2.3). The accuracy of $v(\boldsymbol{y}_i)$ does not have to be very high. As shown in Li (2019); Dobson et al. (2019) and Section 2.2, the optimization problem removes spatially uncorrelated noise in the Monte Carlo, so that the minimizer is a good approximation of the exact solution of the Fokker-Planck equation.

Let $\mathfrak{X} := \{\boldsymbol{x}_i; i = 1, 2, \ldots, N^X\}$ and $\mathfrak{Y} := \{\boldsymbol{y}_j; j = 1, 2, \ldots, N^Y\}$ be two training sets that consists of collocation points. To distinguish them, we call $\mathfrak{X}$ the "training set" and $\mathfrak{Y}$ the "reference set". We find that these two sets do not have to be very large. In our simulations N^X ranges from 10^4 to 10^5, while N^Y ranges from 10^2 to 10^4. This loss function can be easily trained in a simple feedforward neural network architecture. (See Appendix E.1.) We remark that the choice of loss function has some similarity to the so called physics-informed neural network (PINN) studied in Raissi et al. (2019); Wu et al. (2018). The first part $\|\mathcal{L}u\|^2_{L^2(D)}$ serves a similar role by using the differential operator from the physics laws there, whereas the second part of the loss function plays a similar role as the boundary and initial data in PINN. The difference from PINN is that we do not use boundary conditions. Instead, a rough estimate of probability density at reference points is used to guide the neural network training.

The neural network approximation learns the differential operator over collocation points and learns the probability density function from the reference data points. It is proved in many related works that it works effectively to recover a complicated solution function (see Raissi et al. (2019); Sirignano and Spiliopoulos (2018); Wu et al. (2018)). To further accelerate the training process, we introduce a "double shuffling" method that only uses a small batch of $\mathfrak{X}$ and $\mathfrak{Y}$ in each iteration to update the parameter. Since L_1 and L_2 could have very different magnitude, in each iteration, we use Adam optimizer Kingma and Ba (2015) to train L_1 and L_2 and update the parameter $\boldsymbol{\theta}$ separately (Because Adam optimizer is invariant to rescaling. See Kingma and Ba (2015).) This method avoids the trouble of rebalancing the weight of L_1 and L_2 during the neural network training. See Algorithm 1 for detailed implementation of the "double shuffling" method.

Algorithm 1: Neural network training

Input: Training set $\mathfrak{X}$ and reference set $\mathfrak{Y}$.

Output: Minimizer $\boldsymbol{\theta}^*$ and $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta}^*)$.

Initialize a neural network representation $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta})$ with undetermined parameters $\boldsymbol{\theta}$.

Run Monte Carlo simulation to get an approximate density $v(\boldsymbol{y}_j)$ at each reference data points $\boldsymbol{y}_j, j = 1, 2, \ldots, N^Y$.

repeat

  Pick a mini-batch in $\mathfrak{X}$, calculate the mean gradient of L_1, and use the mean gradient to update $\boldsymbol{\theta}$.

  Pick a mini-batch in $\mathfrak{Y}$, calculate the mean gradient of L_2, and use the mean gradient to update $\boldsymbol{\theta}$.

until *Losses of L_1 and L_2 are both small enough*

Return $\boldsymbol{\theta}^*$ and $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta}^*)$.

3.2. Sampling collocation points and reference data.

For many stochastic dynamical systems (2.1), the invariant probability measure is concentrated near some small regions or low dimensional manifolds, while the probability density function is close to zero far away. Hence samples of the collocation points in $\mathfrak{X}$ and $\mathfrak{Y}$ must effectively represent the concentration of the invariant probability density function. The solution is to use the dynamics of the system to choose representative $\mathfrak{X}$ and $\mathfrak{Y}$. We run a numerical trajectory of the stochastic differential equation (2.1), and pick $\alpha \in [0, 1]$ of the collocation points $\boldsymbol{x}_j$ and $\boldsymbol{y}_j$ from this trajectory. When

sampling from the long trajectory, we set up an "internal burn-in time" s_0 and only sample at time $\{ns_0\}_{n=1,2,\cdots}$ to avoid samples being too close to each other. Then to represent the complement set so that the network can learn small values from it, we sample the other $1-\alpha$ of the collocation points $\boldsymbol{x}_j$ and $\boldsymbol{y}_j$ from the uniform distribution on D. Since the concentration part preserves more information of the invariant distribution density, we usually set $\alpha = 0.5 \sim 0.9$. See Algorithm 2 for the full detail.

Algorithm 2: Data collocation sampling

Input: Rate $\alpha \in [0.5, 0.9]$.
Output: Training collocation points $\boldsymbol{x}_i, i = 1, 2, \ldots, N^X$ (or $\boldsymbol{y}_j, j = 1, 2, \ldots, N^Y$).
Initialize $\boldsymbol{X}_0$. ;
Run a numerical trajectory of (2.1) to time t_0 to "burn in".;
Choose an internal "burn in" time s_0 ;
for $i = 1$ *to* N^X **do**
  Generate a random number $c_i \sim U([0, 1])$. ;
  if $c_i \leq \alpha$ **then**
    Let $t_i = t_{i-1} + s_0$;
    Run the numerical trajectory of (2.1) up to time t_i;
    Let $\boldsymbol{x}_i = \boldsymbol{X}_{t_i}$. ;
  else
    Let $t_i = t_{i-1}$.;
    Generate a random point $\boldsymbol{x}_i \sim U(D)$.;
  end
end
Return $\boldsymbol{x}_i, i = 1, 2, \ldots, N^Y$.;

It remains to discuss how to sample the probability density $v(\boldsymbol{y}_i)$ for $\boldsymbol{y}_i \in \mathfrak{Y}$. If the dimension is low, one can sample $v(\boldsymbol{y}_i)$ use grid-based approaches as in Li (2019). For higher dimensional problems, some improvements on sampling techniques are needed. A memory-efficient Monte Carlo sampling algorithm for higher dimensional problems is discussed in Appendix B. For some stochastic differential equations with conditional linear structure, a conditional Gaussian sampler developed in Chen and Majda (2018) can be used. See Appendix C for the full detail.

4. Numerical examples

In this section, we use three numerical examples with explicit exact solution to demonstrate several properties of our Fokker-Planck solver. Then a six dimensional example is used to demonstrate its performance in higher dimensions. In addition to these three numerical examples in the main text, a few additional numerical tests are carried out to further examine the performance and the limitation of our data-driven Fokker-Planck solver. We test the early stopping technique (Appendix D.1), the robustness of the algorithm against random initializations (Appendix D.2), a comparison of neural network architectures (Appendix D.3), the performance of the Fokker-Planck solver for multimodal distributions under the weak noise setting (Appendix D.4), and the performance in higher dimensional problems (Appendix D.5). See Appendix D for the full detail.

4.1. A 2D ring density

Consider a two dimensional stochastic gradient system

$$\begin{cases} dX_t = (-4X_t(X_t^2 + Y_t^2 - 1) + Y_t)\,dt + \sigma\,dW_t^x, \\ dY_t = (-4Y_t(X_t^2 + Y_t^2 - 1) - X_t)\,dt + \sigma\,dW_t^y, \end{cases} \tag{4.1}$$

where W_t^x and W_t^y are independent Wiener processes, and we choose diffusion coefficient $\sigma = 1$. The drift part of equation (4.1) is a gradient flow of the potential function

$$V(x, y) = (x^2 + y^2 - 1)^2$$

plus a rotation term orthogonal to the equipotential lines of V. Hence the probability density function of the invariant measure of (4.1) is

$$u(x, y) = \frac{1}{K} e^{-2V(x,y)/\sigma^2},$$

where $K = \pi \int_{-1}^{\infty} e^{-2t^2/\sigma^2}\,dt$ is the normalization parameter. Note that the orthogonal rotation term does not change the invariant probability density function. This can be verified by substituting $u(x, y)$ into the Fokker-Planck equation of (4.1). In this example and other numerical examples, unless otherwise specified, we use the small feed-forward neural network with fixed architecture of hidden layers as described in Appendix E.1.

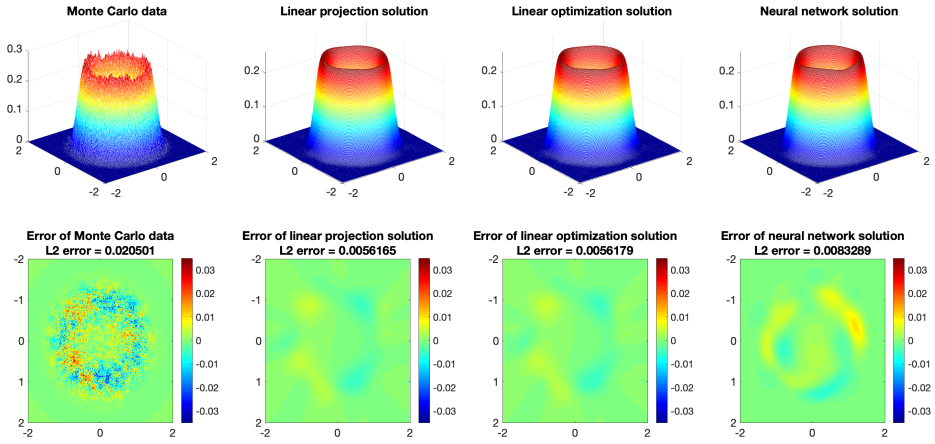

Figure 1: A comparison of probability density function of the invariant measure obtained by Monte Carlo simulation, linear projection method, linear optimization method, and optimization through an artificial neural network. First row: probability density functions. Second row: Distributions of error against the exact solution. (Grid size = 200×200. Sample size of Monte Carlo: 10^7.)

Our first goal is to compare the performance of the neural network representation, solution to the constrained optimization problem, and solution to the unconstrained optimization problem (2.5). In Figure 1, the first column gives a Monte Carlo approximation and its error distribution. As

expected, the Monte Carlo approximation is both noisy and inaccurate. Note that the error term of the Monte Carlo approximation has very little spatial correlation. This motivates Assumption **(A1)**. In the second and the third columns of Figure 1 respectively, we see the data-driven solvers (2.4) and (2.5) can clearly "smooth out" the fluctuation in the Monte Carlo approximation. This confirms the convergence result proved in Dobson et al. (2019) and Theorem 2.1. The last column of Figure 1 show that the artificial neural network method with loss function (3.1) has a similar "smoothing" effect. In the neural network training of this example, we let two sets of collocation points $\mathfrak{X}$ and $\mathfrak{Y}$ be the set of grid points to compare the result. This result validates the use of the loss function (3.1) as a continuous version of the unconstrained optimization problem (2.5). We can see when a grid-based approach is available, it usually has higher accuracy. However, the neural network method is more applicable to higher dimensional problems.

Instead of grid points, the second numerical simulation uses Algorithm 2 with randomly sample collocation points (using Algorithm 2). Figure 2 shows the neural network representations learnt from various amounts of reference points $v(\boldsymbol{y}_j)$. The discrete L^2 errors is computed with respect to this grid and demonstrated on the title of each subfigures. The neural network is then trained with Algorithm 1, in which the norm of $\mathcal{L}u$ is evaluated at each training point. In order to numerically check the effect of the Fokker-Planck operator in the loss function, we train the neural network without calculating $\mathcal{L}u$, and demonstrate the result in Figure 3. More precisely, in Figure 3, we only use a large training data set $\mathfrak{Y}$. Step 3 in Algorithm 1 is skipped. See Appendix E.2 for implementation details.

In Figure 2, we can see a clear underfitting when using too few reference points. The training result becomes satisfactory when the number of training points is 256 or larger. As a comparison, if $\mathcal{L}u$ is not added to the loss function, one needs as large as 16384 training points to reach the same accuracy. This shows the advantage of including $\mathcal{L}u$ into the loss function. The differential operator $\mathcal{L}u$ helps the neural network to find a solution to the Fokker-Planck equation. And the role of reference points is to make sure that the Fokker-Planck solution is the one we actually need. We can train the neural network with only a few hundreds reference points, and the accuracy of the probability density at those reference points does not have to be very high. Similarly to the discrete case in Theorem 2.1, the spatially uncorrelated noise can be effectively removed by training the loss function L_2. This observation is very important in practice, as in high dimension it is not practical to obtain the probability densities for a very large reference set, and the result from a high dimension Monte Carlo simulation is unlikely to be accurate.

4.2. A 2D Gibbs measure

We consider a two dimensional stochastic gradient system

$$\begin{cases} dX_t = (X_t^2 Y_t - X_t^5)\,dt + \sigma\,dW_t^x, \\ dY_t = (\frac{1}{3}X_t^3 - \frac{7}{3}Y_t)\,dt + \sigma\,dW_t^y, \end{cases} \tag{4.2}$$

where W_t^x and W_t^y are independent Wiener processes, and $\sigma = 1$ in this example is the strength of the white noise. The drift part of equation (4.2) is a gradient flow of the potential function

$$V(x,y) = -\frac{1}{3}x^3 y + \frac{1}{6}x^6 + \frac{7}{6}y^2 = \frac{1}{6}(x^3 - y)^2 + y^2.$$

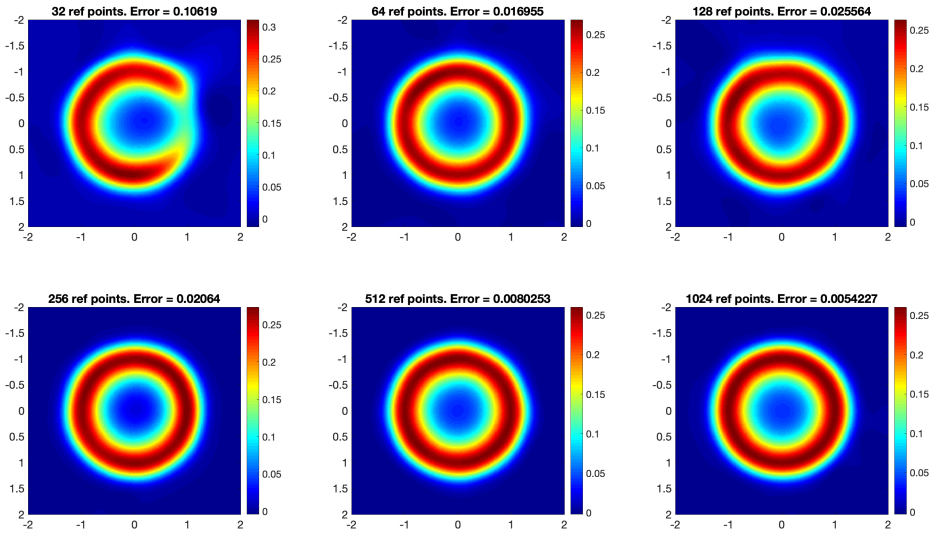

Figure 2: A comparison of different sizes of reference set with $\mathcal{L}u$ being in the loss function. Top left to bottom right: heat map of the invariant probability density function if the "ring model" with $32, 64, 128, 256, 512$, and 1024 reference points are used. The L_2 error is shown in the title of each subplot.

So the invariant measure of (4.2) is the Gibbs measure with probability density function

$$u(x, y) = \frac{1}{K} \exp(-2V(x, y)),$$

where $K = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \exp(-2V(x, y))\, dxdy$ is the normalization parameter. We choose this system because Y_t is conditionally linear with respect to X_t. We can use this system to test the conditional Gaussian sampler.

One aim of this numerical experiment is to show the unconstrained optimization problem used by the artificial neural network can tolerate spatially uncorrelated noise at a very high level. To demonstrate this, we artificially add a noise to the exact solution $u(x)$ of the Gibbs measure to get the reference data v. We first run Algorithm 2 to get four sets of collocation points $\boldsymbol{y}_j, j = 1, 2, \ldots, 1024$. Then we generate four sets of reference data v at these collocation points by injecting an artificial noise with maximal relative error α for $\alpha = 0.01, 0.05, 0.1$, and 0.5, such that $v(\boldsymbol{y}_j) = u(\boldsymbol{y}_j)(1-\alpha+2\alpha U)$, where U is uniformly distributed on $[0, 1]$. Then we run Algorithm 1 with these sets of reference data $\{v(\boldsymbol{y}_j)\}_{j=1}^{1024}$. The first row of Figure 4 shows how the artificial noise is applied by increasing α and the second row shows the neural network approximation. Observing from the third row of Figure 4, it is surprising that even when the magnitude of the multiplicative noise is increased to 0.5, namely, the relative error of the Monte Carlo approximation is 50%, the correction $\tilde{\boldsymbol{u}}(\cdot, \theta)$ is still quite accurate. This shows our method has high tolerance to spatially uncorrelated noise, which is usually the case of the reference data obtained from Monte Carlo simulations.

Then we use the conditional Gaussian sampler (Algorithm 4 in Appendix C) to generate the probability density function. In Figure 5, we can see there is a small but systematic bias in the

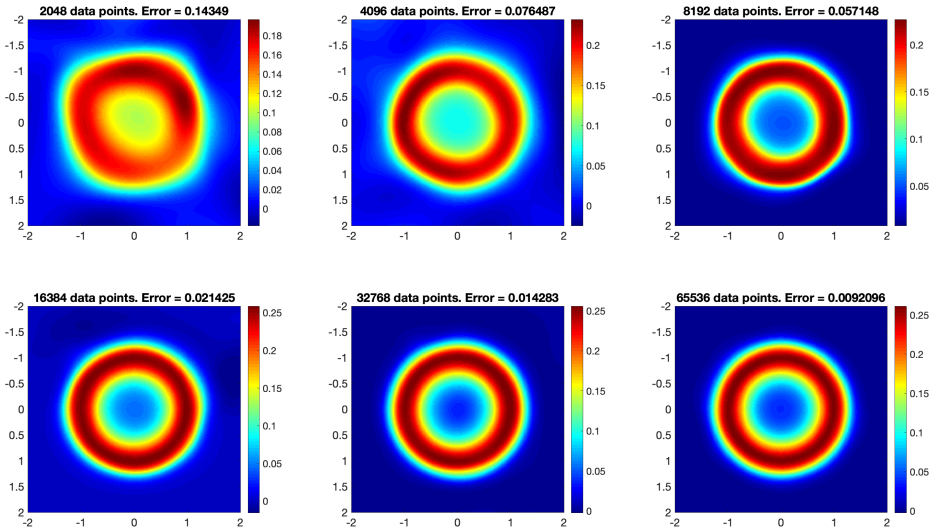

Figure 3: A comparison of different sizes of reference set without $\mathcal{L}u$ being in the loss function. Top left to bottom right: heat map of the invariant probability density function if the "ring model" with 2048, 4096, 8192, 16 384, 32 768, and 65 536 reference points are used. The L_2 error is shown in the title of each subplot.

probability density function given by Algorithm 4. We suspect that this bias comes from the use of one long trajectory in Algorithm 4. As a result, if we use it to generate reference data $v(\boldsymbol{y}_j)$ for $\boldsymbol{y}_j \in \mathcal{Y}$, the error will be systematic, which is very different from the spatially uncorrelated noise seen in the Monte Carlo result. This systematic bias makes the differential operator $\mathcal{L}u$ in the loss function hard to guide the training, because there are infinitely many functions that solve $\mathcal{L}u = 0$. To maintain a minimization of the two parts of the loss function (3.1), a balance between them forces the neural network approximation to produce an approximation biased from the exact solution. In other words, $\boldsymbol{e} = \boldsymbol{v} - \boldsymbol{u}^*$ as defined in Section 2.2 for this conditional Gaussian approximation does not satisfy Assumption **(A1)**. Consequently, the convergence of $\mathbb{E}[\boldsymbol{z}] = \mathbb{E}[\bar{\boldsymbol{u}} - \boldsymbol{u}^*]$ is not guaranteed. However, the conditional Gaussian sampler has its advantage in higher dimensions. See Section 4.4 for more discussion.

4.3. A 4D ring density

Consider a generalization of the stochastic gradient system in Subsection 4.1 in four dimensional state space

$$\begin{cases} dX_t = (-4X_t(X_t^2 + Y_t^2 + Z_t^2 + S_t^2 - 1) + Y_t)\,dt + \sigma\,dW_t^x, \\ dY_t = (-4Y_t(X_t^2 + Y_t^2 + Z_t^2 + S_t^2 - 1) - X_t)\,dt + \sigma\,dW_t^y, \\ dZ_t = (-4Z_t(X_t^2 + Y_t^2 + Z_t^2 + S_t^2 - 1))\,dt + \sigma\,dW_t^z, \\ dS_t = (-4S_t(X_t^2 + Y_t^2 + Z_t^2 + S_t^2 - 1))\,dt + \sigma\,dW_t^s, \end{cases} \tag{4.3}$$

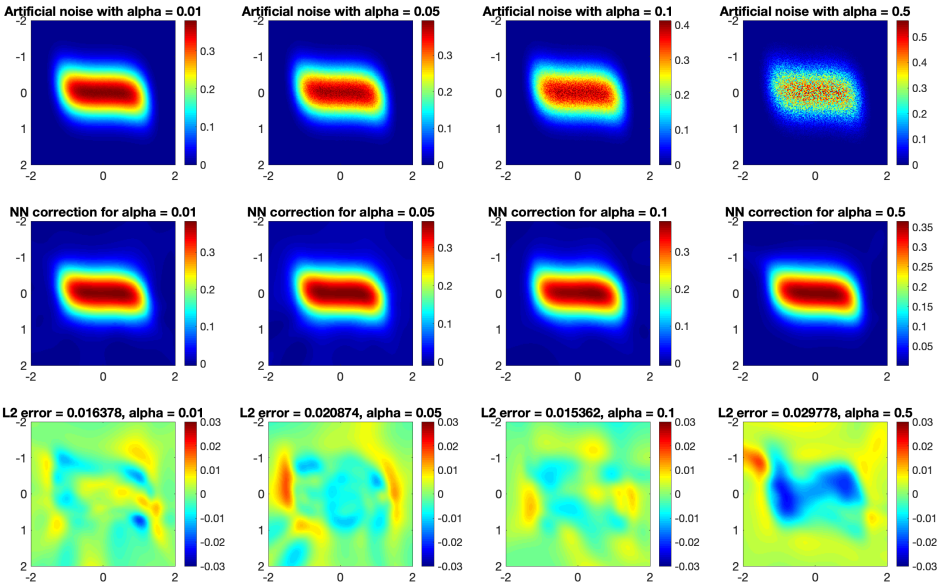

Figure 4: Neural network representations with different level of artificial noise. First row: Artificial noises added to the exact solution. Second row: Neural network approximation with 1024 reference points and 10000 training points. Third row: The error of the neural network approximation and the discrete L_2 error.

where W_t^x, W_t^y, W_t^z and W_t^s are independent Wiener processes, and $\sigma = 1$ in this example is the strength of the white noise. The drift part of equation (4.3) is a gradient flow of the potential function

$$V(x, y, z, s) = (x^2 + y^2 + z^2 + s^2 - 1)^2$$

plus a rotation term orthogonal to the equipotential lines of V in the first two dimensions of variables x and y. Hence the invariant measure of (4.3) is

$$u(x, y, z, s) = \frac{1}{K} \exp(-2V(x, y, z, s)),$$

where $K = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \exp(-2V(x, y, z, s))\, dxdydzds$ is the normalization parameter. Similar to Subsection 4.1, the rotation term does not change the invariant probability density function, which can be verified by substituting $u(x, y, z, s)$ into the Fokker-Planck equation of (4.3).

The aim of this example is to demonstrate the accuracy of neural network representation in 4D. The numerical domain is $D = [-2, 2]^2$. We use Algorithm 2 to sample 10^4 reference points and 10^5 training points. Probability densities at training points are obtained by Algorithm 3. After we get the neural network approximation by Algorithm 1, we evaluate it on the four x-y slices for $(z, s) = (0, 0), (0.5, 0.5), (1, 0)$ and $(1, 1)$ in Figure 6. Figure 6 also shows the error distributions and the L^2 error on these slices. The errors at all points in these 4 slices are controlled at a very low level ≤ 0.04. And the discrete L^2 errors are satisfactory. Figure 6 illustrates that after training the loss function (3.1) on a sparse set of reference data, this solution function is accurate at any

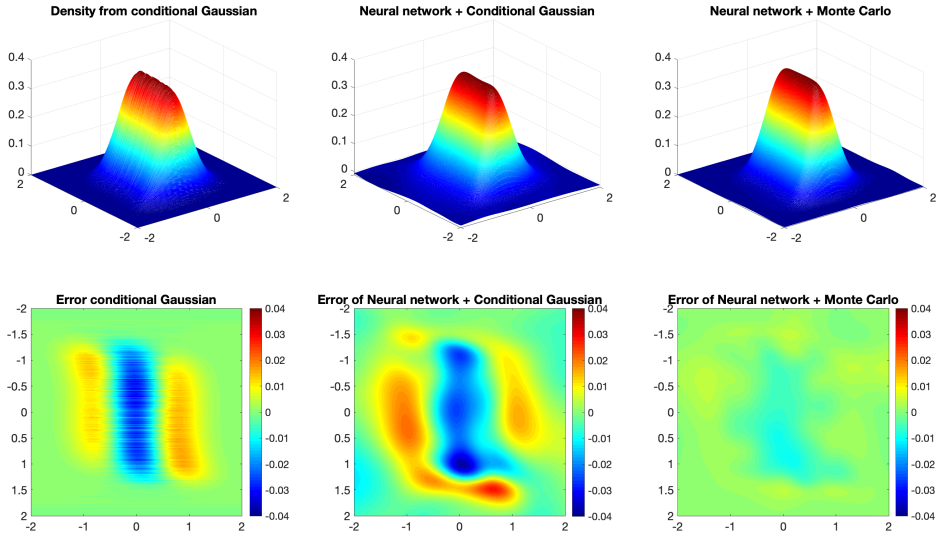

Figure 5: A comparison of the conditional Gaussian method and the direct Monte Carlo method. First column: The invariant probability density function and its error obtained by the conditional Gaussian method. Second column: Top: The invariant probability density function and its error obtained by the neural network approximation, with 1024 training points whose densities are obtained by Algorithm 4. Bottom: The invariant probability density function and its error obtained by the neural network approximation, with 1024 training points whose densities are obtained by Monte Carlo simulation.

point in D. It demonstrates strong representing power of the neural network approximation, both globally and locally. We remark that it is not possible to solve this 4D Fokker-Planck equation with traditional numerical PDE approach. The divide-and-conquer strategy in Dobson et al. (2019) would be difficult to implement as well, due to the high memory requirement of a 4D mesh.

4.4. A 6D conceptual dynamical model for turbulence

In this subsection, we consider a six dimensional stochastic dynamical system with conditional Gaussian structure as (C.1)-(C.2) with $\boldsymbol{X}^l_{\mathbf{I}}(t) = X_t$ and $\boldsymbol{X}^l_{\mathbf{II}}(t) = (Y^{(1)}_t, Y^{(2)}_t, Y^{(3)}_t, Y^{(4)}_t, Y^{(5)}_t)^T$

$$\begin{cases} dX_t = (-0.1X_t + 0.5 + 0.25X_t(Y^{(1)}_t + Y^{(2)}_t + Y^{(3)}_t + Y^{(4)}_t + Y^{(5)}_t))\,dt + 2\,dW^x_t, \\ dY^{(1)}_t = (-0.2Y^{(1)}_t - 0.25X^2_t)\,dt + 0.5\,dW^{(1)}_t, \\ dY^{(2)}_t = (-0.5Y^{(2)}_t - 0.25X^2_t)\,dt + 0.2\,dW^{(2)}_t, \\ dY^{(3)}_t = (-Y^{(3)}_t - 0.25X^2_t)\,dt + 0.1\,dW^{(3)}_t, \\ dY^{(4)}_t = (-2Y^{(4)}_t - 0.25X^2_t)\,dt + 0.1\,dW^{(4)}_t, \\ dY^{(5)}_t = (-5Y^{(5)}_t - 0.25X^2_t)\,dt + 0.1\,dW^{(5)}_t, \end{cases} \tag{4.4}$$

where W^x_t and $W^{(i)}_t, i = 1, 2, \ldots, 5$, are independent Wiener processes. This model has been studied in Chen and Majda (2018) as a numerical example.

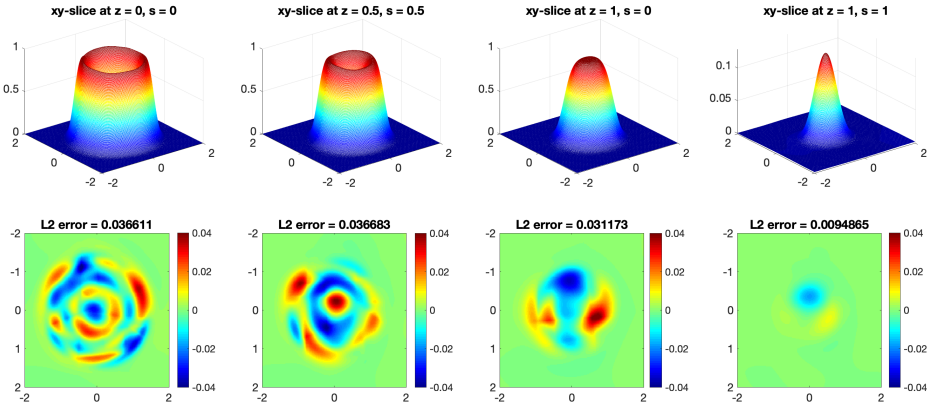

Figure 6: Invariant probability density function of the 4D ring (Equation (4.3)). Total number of reference points is 10 000. Probability density at reference points is obtained by direct Monte Carlo method with 10^{10} sample points. First row: Invariant probability density functions restricted on the x-y slices with $z = 0, s = 0$. $z = 0.5, s = 0.5$, $z = 1, s = 0$, and $z = 1, s = 1$. Second row: Error of probability density functions when comparing with the exact solution. The discrete L_2 error is shown in the title of each subplot.

In this high dimensional system, we compare the direct Monte Carlo approximation and neural network approximation with the reference data obtained from both Monte Carlo and the modified conditional Gaussian sampler in Algorithm 4. It is not possible to visualize a 6D probability density function, so we compare probability densities on the central slices in the 6D state space, namely, the u-$v_i, i = 1, 2, \ldots, 5$ hyperplanes with $v_j = 0, j \neq i$. The first row demonstrates the probability density functions at the five slices obtained by a direct Monte Carlo simulation. The solution has low resolution and low accuracy because It is difficult to collect enough samples in high dimension.

In this example, we generate a reference point set $\mathfrak{Y}$ with size $N^Y = 20000$ using Algorithm 2. These collocations are very sparse in this six dimensional region D. Then we use both Monte Carlo approximation and the conditional Gaussian sampler in Algorithm 4 to generate the probability density $v(\boldsymbol{y}_i)$ for $\boldsymbol{y}_i \in \mathfrak{Y}$. Note that the simulation time of Monte Carlo sampler is about 100 times more than the conditional Gaussian sampler. Then in both cases, we use Algorithm 1 to obtain a neural network approximation of the invariant probability measure. After the neural network is trained in the whole region D, we evaluate and plot it on the five central slices (see the second and third row in Figure 7). Although a closed-form solution for this example is not possible, we can still see that the solution obtained by three different approaches are not very far away from each other. This confirms the validity of the solutions. The neural network has low demand (20000 points) on reference data points and fast training speed (less than one hour). After the training, we can use it to predict the invariant probability density at any point in the domain. This is a remarkable result, because it is impossible to solve such as six dimensional problem by using traditional grid-based approaches.

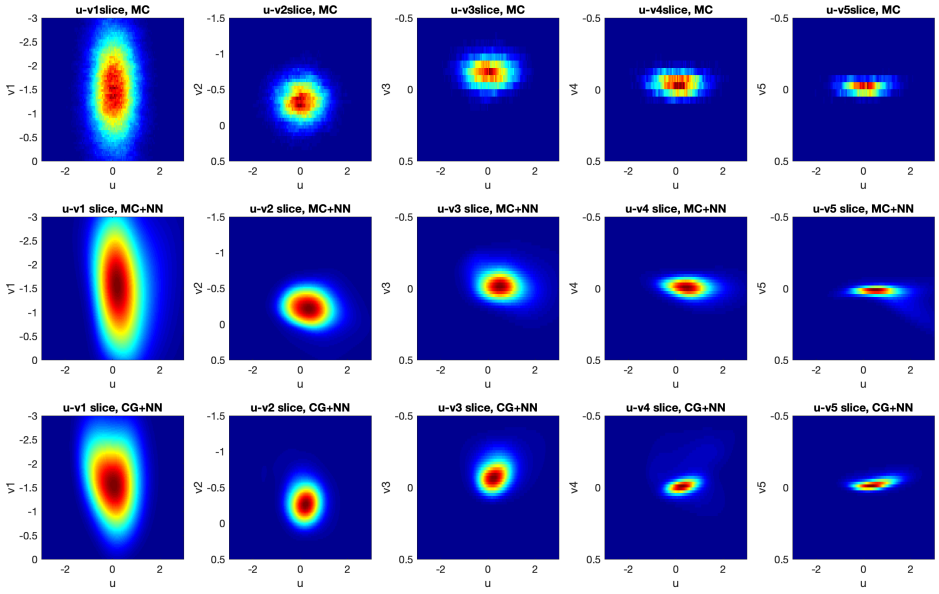

Figure 7: Heat maps of the invariant probability density function of the 6D turbulence model restricted on 2D slices. From left to right: the ith plot is the u-$v_i, i = 1, 2, \ldots, 5$ slice with $v_j = 0, j \neq i$. Top row are invariant probability density functions obtained by the direct Monte Carlo method. Middle row and bottom row are the neural network approximation using Algorithm 1 with probability densities $\{v(\boldsymbol{y}_i)\}$ obtained by Monte Carlo simulation and the conditional Gaussian sampler, respectively.

5. Conclusion and Prospective Works

We proposed a neural network approximation method for solving the Fokker-Planck equations. The motivation is that the data-driven method studied in Li (2019); Dobson et al. (2019) can be converted to a similar unconstrained optimization problem, and a mesh-free neural network solver can be used to solve the "continuous version" of this unconstrained optimization problem. We only present the case of the stationary Fokker-Planck equation that describes the invariant probability measure, because the case of the time-dependent Fokker-Planck equation is analogous. By introducing the differential operator of the Fokker-Planck equation into the loss function, the demand for large training data in the learning process is significantly reduced. Our simulation shows that the neural network can tolerate very high noise in the training data so long as it is spatially uncorrelated. In terms of performance, one training epoch that goes through all training points once takes a few seconds for 2D problems and about six minutes for the 20D problem shown in Appendix D.5. It takes 20 to 30 epochs for the neural network to converge to the desired solution. This is significantly faster than the traditional grid-based method in dimension 3 or higher. We believe this work provides an effective numerical approach to study many high dimensional stochastic dynamics.

In this paper, the convergence of minimizer of the new unconstrained optimization problem is only carried out for the discrete case. It is tempting to extend this result to the space of functions. The problem becomes trivial and not interesting if we work with the space of C^∞ functions and

assume that the error term of the Monte Carlo simulation is a spatial white noise. So we need to consider the more realistic case such as the Barron space, and find a more realistic assumption to describe the reference data obtained by the Monte Carlo simulation. We plan to carry out this study in our subsequent work. In the future, we will also apply this method to more complicated but interesting systems such as systems with non-Gaussian Lévy noises and quasi-stationary distributions.

References

Rikard Anton, David Cohen, and Lluis Quer-Sardanyons. A fully discrete approximation of the one-dimensional stochastic heat equation. *IMA Journal of Numerical Analysis*, 40(1):247–284, 2020.

Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

Christian Beck, Sebastian Becker, Philipp Grohs, Nor Jaafari, and Arnulf Jentzen. Solving stochastic differential equations and Kolmogorov equations by means of deep learning. arXiv:1806.00421, 2018.

Christian Beck, Weinan E, and Arnulf Jentzen. Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. *Journal of Nonlinear Science*, 29:1563–1619, 2019.

Nan Chen and Andrew J Majda. Beating the curse of dimension with accurate statistics for the Fokker-Planck equation in complex turbulent systems. *Proceedings of the National Academy of Sciences*, 114(49):12864–12869, 2017.

Nan Chen and Andrew J Majda. Efficient statistically accurate algorithms for the Fokker-Planck equation in large dimensions. *Journal of Computational Physics*, 354:242–268, 2018.

Xiaoli Chen, Liu Yang, Jinqiao Duan, and George Em Karniadakis. Solving inverse stochastic problems from discrete particle observations using the fokker-planck equation and physics-informed neural networks. *arXiv:2008.10653*, 2020.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Aaron R Dinner, Jonathan C Mattingly, Jeremy OB Tempkin, Brian Van Koten, and Jonathan Weare. Trajectory stratification of stochastic dynamics. *SIAM Review*, 60(4):909–938, 2018.

Matthew Dobson, Yao Li, and Jiayu Zhai. An efficient data-driven solver for Fokker-Planck equations: algorithm and analysis. arXiv:1906.02600, 2019.

Weinan E, Chao Ma, and Lei Wu. Barron spaces and the compositional function spaces for neural network models. arXiv:1906.08039, 2019.

Ingo Gühring, Gitta Kutyniok, and Philipp Petersen. Error bounds for approximations with deep ReLU neural networks in $W^{s,p}$ norms. *Analysis and Applications*, 18(5):803–859, 2020.

Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *PNAS*, 115(34):8505–8510, 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Andrey Kolmogoroff. Zur theorie der stetigen zufälligen prozesse. *Mathematische Annalen*, 180: 149–160, 1933.

Yao Li. A data-driven method for the steady state of randomly perturbed dynamics. *Communications in Mathematical Sciences*, 17(4):1045–1059, 2019.

Nicolas Macris and Raffaele Marino. Solving non-linear kolmogorov equations in large dimensions by using deep learning: a numerical comparison of discretization schemes. *arXiv:2012.07747*, 2020.

Jay M Newby. Isolating intrinsic noise sources in a stochastic genetic switch. *Physical biology*, 9 (2):026002, 2012.

Philipp Petersen and Felix Voigtlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *Neural Networks*, 108:296–330, 2018.

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:586–707, 2019.

Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

Taiji Suzuki. Adaptivity of deep reLU network for learning in besov and mixed smooth besov spaces: optimal rate and curse of dimensionality. In *International Conference on Learning Representations*, 2019. arXiv:1810.08033.

Jin-Long Wu, Heng Xiao, and Eric Paterson. Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework. *Phys. Rev. Fluids*, 3:074602, Jul 2018. doi: 10.1103/PhysRevFluids.3.074602. URL https://link.aps.org/doi/10.1103/PhysRevFluids.3.074602.

Yong Xu, Hao Zhang, Yongge Li, Kuang Zhou, Qi Liu, and Jürgen Kurths. Solving fokker-planck equation using deep learning. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30: 013141, 2020a.

Zhi-Qin John Xu, Yaoyu Zhang, Tao Luo, Yanyang Xiao, and Zheng Ma. Frequency principle: Fourier analysis sheds light on deep neural networks. *Communications in Computational Physics*, 28(5):1746–1767, 2020b.

Appendix A. Proof of Theorem 2.1

We first need to describe how the "baseline" solution $\boldsymbol{u}^*$ is obtained. Let $\hat{\boldsymbol{u}} \in \mathbb{R}^{N^n}$ denote the true solution of the Fokker-Planck equation restricted on the rectangular grid $\{x_i\}_{i=1}^{N^n}$ in D, that is, $\hat{\boldsymbol{u}}_i$ is the solution to equation (2.2) at point x_i. Then $\boldsymbol{u}^*$ is the numerical solution obtained by the finite difference method such that $\boldsymbol{A}\boldsymbol{u}^* = 0$ and $\boldsymbol{u}^* = \hat{\boldsymbol{u}}$ at all grid points on ∂D. More precisely, we need to solve a new linear system

$$\begin{bmatrix} \boldsymbol{A} \\ P_0 \end{bmatrix} \boldsymbol{u}^* = \begin{bmatrix} \boldsymbol{0} \\ P_0\hat{\boldsymbol{u}} \end{bmatrix},$$

where $\boldsymbol{A}$ is the aforementioned $(N-2)^n \times N^n$ matrix, P_0 is an $(N^n - (N-2)^n) \times N^n$ matrix such that $P_0\boldsymbol{u}$ gives entries of $\boldsymbol{u}$ on grid points on ∂D. Since the finite difference method is convergent for second order elliptic PDEs with given boundary value, when $h \ll 1$, $\boldsymbol{u}^*$ is a good approximation of $\hat{\boldsymbol{u}}$. And the accuracy of $\boldsymbol{u}^*$ is considerable higher than the result of Monte Carlo simulations.

Proof [Proof of Theorem 2.1] Let $F(\boldsymbol{u}) = \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} + (\boldsymbol{u}-\boldsymbol{v})^T(\boldsymbol{u}-\boldsymbol{v})$. It is easy to see that the minimizer solves

$$\frac{\partial F(\boldsymbol{u})}{\partial \boldsymbol{u}} = 2\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} + 2\boldsymbol{u} - 2\boldsymbol{v} = \boldsymbol{0}.$$

Therefore, the quadratic form (2.5) has a unique minimizer $\bar{\boldsymbol{u}} = (\boldsymbol{I} + \boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{v}$.

Denote $\boldsymbol{B} = (\boldsymbol{I} + \boldsymbol{A}^T\boldsymbol{A})^{-1}$. Then

$$\boldsymbol{B}^{-1}\boldsymbol{u}^* = (\boldsymbol{I} + \boldsymbol{A}^T\boldsymbol{A})\boldsymbol{u}^* = \boldsymbol{u}^*,$$

and

$$\boldsymbol{v} = \boldsymbol{B}^{-1}\bar{\boldsymbol{u}} = \boldsymbol{B}^{-1}\bar{\boldsymbol{u}} - \boldsymbol{B}^{-1}\boldsymbol{u}^* + \boldsymbol{u}^* = \boldsymbol{B}^{-1}\boldsymbol{z} + \boldsymbol{u}^*.$$

So $\boldsymbol{z} = \boldsymbol{B}\boldsymbol{e}$. Therefore, $\mathbb{E}[\boldsymbol{z}] = 0$ by assumption **(A1)**, and $\mathrm{cov}(\boldsymbol{z},\boldsymbol{z}) = \zeta^2\boldsymbol{B}\boldsymbol{B}^T = \zeta^2\boldsymbol{B}^2$ by the symmetry of $\boldsymbol{B}$. Furthermore,

$$\mathbb{E}[\|\boldsymbol{z}\|_2^2] = \mathrm{Trace}(\mathrm{cov}(\boldsymbol{z},\boldsymbol{z})) = \zeta^2\mathrm{Trace}(\boldsymbol{B}^2) = \zeta^2\sum_{k=1}^{M}\lambda_k^2,$$

where $M = N^n$ and $\{\lambda_k\}_{k=1}^M$ are the eigenvalues of $\boldsymbol{B}$.

For the sake of simplicity let $M = N^n$. Recall that $\boldsymbol{A}_h = h^2\boldsymbol{A}$. Since $\boldsymbol{A}_h^T\boldsymbol{A}_h$ is symmetric, there is a orthonormal matrix $\boldsymbol{Q}$ such that $\boldsymbol{A}_h^T\boldsymbol{A}_h = \boldsymbol{Q}\boldsymbol{\Lambda}_h\boldsymbol{Q}^T$, where $\boldsymbol{\Lambda}_h = \mathrm{diag}\{\lambda_1^h, \cdots \lambda_M^h\}$ is the diagonal matrix whose diagonal elements are the eigenvalues of $\boldsymbol{A}_h^T\boldsymbol{A}_h$. A short calculation shows

$$\lambda_k = \frac{1}{1 + h^{-4}\lambda_k^h}.$$

Since $\boldsymbol{A}_h^T\boldsymbol{A}_h$ is positive semi-definite and $\boldsymbol{A}_h \in \mathbb{R}^{(N-2)^n \times N^n}$ has full rank, $\boldsymbol{A}_h^T\boldsymbol{A}_h$ has $N^n - (N-2)^n$ zero eigenvalues, while $r = (N-2)^n$ eigenvalues are positive. So for all sufficiently small $h > 0$, we have

$$\begin{aligned}\mathbb{E}[\|\boldsymbol{z}\|_2^2] &= \zeta^2\sum_{k=1}^{M}\lambda_k^2 = \zeta^2\sum_{k=1}^{M}\left(\frac{1}{1+h^{-4}\lambda_k^h}\right)^2 \\ &= \zeta^2\left[N^n - (N-2)^n + \sum_{i=1}^{r}\left(\frac{1}{1+h^{-4}\lambda_i^h}\right)^2\right].\end{aligned}$$

Since $\mathbb{E}[\|\boldsymbol{e}\|^2] = \zeta^2 N^n$ and $N = O(h^{-1})$, by Assumption **(A2)**, we have

$$\frac{\mathbb{E}[\|\boldsymbol{z}\|_2^2]}{\mathbb{E}[\|\boldsymbol{e}\|^2]} \leq \frac{N^n - (N-2)^n}{N^n} + O(1)Q(h) \to 0$$

as $h \to 0$. ■

It remains to discuss why Assumption **(A2)** is a valid assumption. Because $\boldsymbol{A}$ is obtained by finite difference method, it has the form $\boldsymbol{A} = h^{-2}(\boldsymbol{A_0}(h) + h\boldsymbol{A_1}(h))$, where $\boldsymbol{A}_0$ (resp. $\boldsymbol{A}_1$) is a constant matrix if σ (resp. f) is constant. It is extremely difficult to rigorously prove Assumption **(A2)** when σ and f in equation (2.1) are location-dependent. In general, the smallest nonzero eigenvalue is $O(h^4)$, but other eigenvalues are significantly larger than $O(h^4)$. There are $O(h^{-n})$ terms in the summation in the definition of $Q(h)$. Hence $Q(h)$ approaches to zero when $h \ll 1$. In Figure 8, we numerically verify Assumption **(A2)** for 1D and 2D Fokker-Planck equations. We can see that $Q(h) \to 0$ as $h \to 0$ in both cases. This numerical result is for $\sigma = \mathrm{Id}_n$ and $f = 0$. Note that f determines $\boldsymbol{A}_1(h)$, which is only a small perturbation of the matrix $\boldsymbol{A}(h)$. So we expect Assumption **(A2)** to hold for any bounded f.

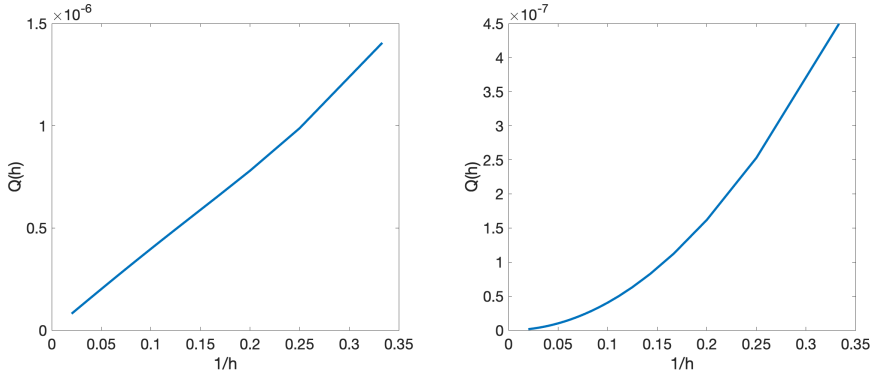

Figure 8: Left: $Q(h)$ vs. h for the discretized 1D Fokker-Planck operator. Right: $Q(h)$ vs. h for the discretized 2D Fokker-Planck operator.

Appendix B. Sampling probability densities – direct Monte Carlo.

It remains to discuss the sampling technique to obtain $\boldsymbol{v}(y_i)$ for $\boldsymbol{y}_i \in \mathcal{Y}$. This step is trivial when using traditional grid-based method. One only needs to set up a grid in the numerical domain and count sample points in the neighborhood of each grid from a long trajectory. However, instead of the full space, Algorithm 1 only requires probability densities at thousands of reference points $\boldsymbol{y}_i \in \mathcal{Y}$. Also a high dimensional grid could occupy an unrealistic amount of memory. So we need to improve the efficiency of the Monte Carlo sampler.

As mentioned in Section 2.1, to obtain a desirable accuracy of the reference data $v(\boldsymbol{y}_j)$ using a Monte Carlo method, one needs to run very long numerical trajectories of (2.1) to guarantee enough points on the trajectories are counted around $\boldsymbol{y}_j$. On the other hand, when the dimensionality increases, the size of reference set $\mathcal{Y}$ in the training process also increases. For example, to solve a

6D Fokker-Planck equation, N^Y needs to be as large as tens of thousands. If we do not optimize the algorithm, then every time a new sample point is obtained from a long trajectory, one must check whether it belongs to the neighborhood of each collocation points $\boldsymbol{y}_j \in \mathcal{Y}$. This will make the long trajectory sampler too slow to be useful.

An alternative approach is to create an N mesh with N^n grid points on $D = \prod_{\iota=1}^{n}[a_\iota, b_\iota]$ and denote the vector of grid points by $\boldsymbol{y}^k, k = 1, 2, \ldots, N^n$. Let $h_\iota = (b_{iota} - a_{iota})/h$ be the grid size. This gives an n-dimensional "box" $\prod_{\iota=1}^{n}[\boldsymbol{y}_\iota^k - h_\iota, \boldsymbol{y}_\iota^k + h_\iota]$ around each mesh point $\boldsymbol{y}^k$. Instead of using Algorithm 2 to sample $\boldsymbol{y}_j$ directly, we choose the closest mesh point $\boldsymbol{y}^{k_j}$ for each $\boldsymbol{y}_j$ given by Algorithm 2. So we have $\mathcal{Y} = \{\boldsymbol{y}^{k_j}\}, j = 1, \cdots, N^Y$. With the help of the mesh, we can put a sample point into the corresponding n-dimensional "box" after implementing $2n$ comparisons. After running a sufficiently long trajectory, the number of samples in each box gives the approximate probability density at each grid point. Then we can look up the probability density of $\boldsymbol{y}^{k_j}$ from the corresponding boxes. This approach dramatically improves the efficiency, at the cost of storing a big array with N^n points. When $n \geq 4$, this method could have an unrealistically high demand of the memory.

We propose the following "splitting" method to balance the efficiency and the memory pressure. The idea is to split the dimensions in to groups to reduce the size of vector stored in the memory. To be specific, we use $n = 6$ as an example to state this method. One can easily generalize it to other dimensions. For $\mathbb{R}^6 = \mathbb{R}^3 \times \mathbb{R}^3$, we create an array of arrays $\mathcal{Q}$ with $2 \times N^3$ entries. The first and second N^3 entries are for the first and second $\mathbb{R}^3$, respectively. Each entry of $\mathcal{Q}$ is an array of indices of training points. More precisely, for a collocation point $\boldsymbol{y}_j = (y_1^j, y_2^j, \ldots, y_6^j) \in \mathcal{Y}$ that is also a mesh point, we denote its index by $(n_1, n_2, \ldots, n_6)$, where $n_\iota = (y_\iota^j - a_\iota)/h_j, \iota = 1, 2, \ldots, 6$. Then we record the numbering j in two arrays corresponding to the $(n_1N^2 + n_2N + n_3)$-th and the $(N^3 + n_4N^2 + n_5N + n_6)$-th entries of $\mathcal{Q}$. When a sample point $\boldsymbol{x} = (x_1, x_2, \ldots, x_6)$ is obtained from the Monte Carlo sampler, we compute its mesh index $(n_1^{\boldsymbol{x}}, n_2^{\boldsymbol{x}}, \ldots, n_6^{\boldsymbol{x}})$, where $n_\iota^{\boldsymbol{x}} = \lfloor (x_\iota - a_\iota)/h_\iota + 1/2 \rfloor, \iota = 1, 2, \ldots, 6$. Then we check the arrays at the $(n_1^{\boldsymbol{x}}N^2 + n_2^{\boldsymbol{x}}N + n_3^{\boldsymbol{x}})$-th and the $(N^3 + n_4^{\boldsymbol{x}}N^2 + n_5^{\boldsymbol{x}}N + n_6^{\boldsymbol{x}})$-th entries of $\mathcal{A}$. The sample point $\boldsymbol{x}$ is associated to the training point $\boldsymbol{y}_j$ if and only if the intersection of the two aforementioned arrays is j. See Algorithm 3 for the full detail.

Appendix C. Sampling probability densities – conditional Gaussian sampler.

As discussed before, the direct Monte Carlo method still suffers from the curse of dimensionality. It is more and more difficult to collect enough samples in a higher dimensional box. To maintain the desired accuracy in high dimensional spaces, the requirement of sampling grows exponentially with the dimension. We need to either run much longer numerical trajectories of (2.1) or make the grid more coarse. Otherwise the simulation will give a lot of $v(\boldsymbol{y}_i) = 0$ at reference points $\boldsymbol{y}_i$ whose invariant probability density is not zero. This makes it not applicable as a reference data in the neural network training. For some high dimensional problems with conditional linear structure, the conditional Gaussian framework introduced in Chen and Majda (2017, 2018) can be effectively applied to solve the problem of curse-of-dimensionality. Consider a stochastic differential equation with the following the conditional linear structure

$$d\boldsymbol{X}_{\mathbf{I}} = [\boldsymbol{A}_0(t, \boldsymbol{X}_{\mathbf{I}}) + \boldsymbol{A}_1(t, \boldsymbol{X}_{\mathbf{I}})\boldsymbol{X}_{\mathbf{II}}]\, dt + \boldsymbol{\Sigma}_{\mathbf{I}}(t, \boldsymbol{X}_{\mathbf{I}})\, d\boldsymbol{W}_{\mathbf{I}}(t), \tag{C.1}$$

$$d\boldsymbol{X}_{\mathbf{II}} = [\boldsymbol{a}_0(t, \boldsymbol{X}_{\mathbf{I}}) + \boldsymbol{a}_1(t, \boldsymbol{X}_{\mathbf{I}})\boldsymbol{X}_{\mathbf{II}}]\, dt + \boldsymbol{\Sigma}_{\mathbf{II}}(t, \boldsymbol{X}_{\mathbf{I}})\, d\boldsymbol{W}_{\mathbf{II}}(t), \tag{C.2}$$

Algorithm 3: Reference data sampling with Monte Carlo method for high dimensional spaces $\mathbb{R}^6$

Input: Reference set $\mathcal{Y} = \{\boldsymbol{y}_1, \cdots, \boldsymbol{y}_{N^Y}\}$. $\boldsymbol{y}_i$ are grid points.
Output: Probability densities $v(\boldsymbol{y}_j)$ at $\boldsymbol{y}_j, j = 1, 2, \ldots, N^Y$.
Set a zero array $\boldsymbol{\eta}$ with length N^Y and an array of arrays $\mathcal{Q}$ that contains $2 \times N^3$ empty arrays. ;
Sample N^Y collocation points using Algorithm 2 ;
for $j = 1$ *to* N^Y **do**
  Compute $n_\iota = (y_\iota^j - a_\iota)/h_\iota, \iota = 1, 2, \ldots, 6.$;
  Add j to the $(n_1 N^2 + n_2 N + n_3)$-th and the $(N^3 + n_4 N^2 + n_5 N + n_6)$-th elemental arrays of $\mathcal{Q}$. ;
end
Initialize $\boldsymbol{X}(0)$ and run a numerical simulation of equation (2.1)) for sometime t_0 and "burn in" time t_0.;
Reset $\boldsymbol{X}(0) = \boldsymbol{X}(t_0)$.;
for $l = 1$ *to* L **do**
  Continue the numerical simulation of (2.1) with step size Δt to get a new sample point $\boldsymbol{x} = \boldsymbol{X}(l\Delta t)$. ;
  Compute $n_\iota^{\boldsymbol{x}} = \lfloor (x_\iota - a_\iota)/h_\iota + 1/2 \rfloor, \iota = 1, 2, \ldots, 6.$;
  Check the intersection $\mathcal{B}_{\boldsymbol{x}}$ of the $(n_1^{\boldsymbol{x}} N^2 + n_2^{\boldsymbol{x}} N + n_3^{\boldsymbol{x}})$-th and the $(N^3 + n_4^{\boldsymbol{x}} N^2 + n_5^{\boldsymbol{x}} N + n_6^{\boldsymbol{x}})$-th elemental arrays of $\mathcal{Q}$.;
  if $\mathcal{B}_{\boldsymbol{x}} = \{j\}$ **then**
    $\boldsymbol{\eta}(j) = \boldsymbol{\eta}(j) + 1.$;
  end
end
Return $v(\boldsymbol{y}_j) = \boldsymbol{\eta}(j) \prod_{\iota=1}^6 h_\iota^{-1}/L, j = 1, 2, \ldots, N^Y$. ;

where $\boldsymbol{X}(t) = (\boldsymbol{X}_{\mathbf{I}}(t), \boldsymbol{X}_{\mathbf{II}}(t)) \in \mathbb{R}^{n_{\mathbf{I}}} \times \mathbb{R}^{n_{\mathbf{II}}}$ is the solution stochastic process. Then given the current path $\boldsymbol{X}_{\mathbf{I}}(s), s \le t$, the conditional distribution of $\boldsymbol{X}_{\mathbf{II}}(t)$ is approximated by a Gaussian distribution

$$(\boldsymbol{X}_{\mathbf{II}}(t) | \boldsymbol{X}_{\mathbf{I}}(s), s \le t) \sim N(\overline{\boldsymbol{X}}_{\mathbf{II}}(t), \boldsymbol{R}_{\mathbf{II}}(t)),$$

where the expectation $\overline{\boldsymbol{X}}_{\mathbf{II}}(t)$ and variance $\boldsymbol{R}_{\mathbf{II}}(t)$ follow the ordinary differential equations

$$d\overline{\boldsymbol{X}}_{\mathbf{II}} = [\boldsymbol{a}_0 + \boldsymbol{a}_1 \overline{\boldsymbol{X}}_{\mathbf{II}}]\, dt + (\boldsymbol{R}_{\mathbf{II}} \boldsymbol{A}_1^* (\boldsymbol{\Sigma}_{\mathbf{I}} \boldsymbol{\Sigma}_{\mathbf{I}}^*)^{-1} [d\boldsymbol{X}_{\mathbf{I}} - (\boldsymbol{A}_0 + \boldsymbol{A}_1 \overline{\boldsymbol{X}}_{\mathbf{II}})\, dt], \tag{C.3}$$

$$d\boldsymbol{R}_{\mathbf{II}} = [\boldsymbol{a}_1 \boldsymbol{R}_{\mathbf{II}} + \boldsymbol{R}_{\mathbf{II}} \boldsymbol{a}_1^* + (\boldsymbol{\Sigma}_{\mathbf{I}} \boldsymbol{\Sigma}_{\mathbf{I}}^*) - (\boldsymbol{R}_{\mathbf{II}} \boldsymbol{A}_1^* (\boldsymbol{\Sigma}_{\mathbf{I}} \boldsymbol{\Sigma}_{\mathbf{I}}^*)^{-1} (\boldsymbol{R}_{\mathbf{II}} \boldsymbol{A}_1^*)^*]\, dt. \tag{C.4}$$

The original algorithm in Chen and Majda (2018) is for simulating the time evolution of the probability density function. The probability density function is obtained by averaging the conditional probability density of many independent trajectories of equation (2.1). Since the focus of this paper is the invariant probability density function, we make some modification to the conditional Gaussian framework in Chen and Majda (2018). The main difference is that we use one long trajectory to simulate the conditional probability density. This is because the speed of convergence of the evolution of transient distribution to the invariant distribution of equation (2.1) is unknown. In

a simulation, we don't know when the probability density function becomes a satisfactory approximation of the invariant probability density function.

In equation (C.1)-(C.2), the first part $\boldsymbol{X}_{\mathbf{I}}$ is usually in a relatively low dimension $n_{\mathbf{I}}$. So for this part, a Monte Carlo approximation is reliable. Let $\mathcal{Y}$ be the set of reference points. Denote the two coordinates of a reference point $\boldsymbol{y}_i \in \mathcal{Y}$ by $\boldsymbol{y}_i^{\mathbf{I}}$ and $\boldsymbol{y}_i^{\mathbf{II}}$ respectively. Then we run a long numerical trajectory $\boldsymbol{X}$, for (C.1)-(C.2) and evaluate the trajectory at discrete times $0 = t_0 < t_1 < t_2 < \cdots < t_I = T$. Denote the visiting times of $\boldsymbol{X}_{\mathbf{I}}$ to an h-neighborhood of $\boldsymbol{y}_i^{\mathbf{I}}$ by $t_{k_1}, \cdots, t_{k_{S(j)}}$. Then at time t_{k_i}, the conditional probability density of at $\boldsymbol{y}_i^{\mathbf{II}}$ is

$$\begin{aligned} v_{i,j} &= f_{(\boldsymbol{y}_j^{\mathbf{II}}(t_{k_i})|\boldsymbol{y}_j^{\mathbf{I}}(s), s\le t_{k_i})}(\boldsymbol{y}_j^{\mathbf{II}}) \\ &= \frac{1}{\sqrt{(2\pi)^{n_{\mathbf{II}}}|\boldsymbol{R}_{\mathbf{II}}(t_{k_i})|}} \exp(-\tfrac{1}{2}(\boldsymbol{y}_j^{\mathbf{II}} - \overline{\boldsymbol{X}}_{\mathbf{II}}(t_{k_i}))^T \boldsymbol{R}_{\mathbf{II}}(t_{k_i})^{-1}(\boldsymbol{y}_j^{\mathbf{II}} - \overline{\boldsymbol{X}}_{\mathbf{II}}(t_{k_i}))) \end{aligned} \tag{C.5}$$

according to the Gaussian distribution $N(\overline{\boldsymbol{X}}_{\mathbf{II}}(t_{k_i}), \boldsymbol{R}_{\mathbf{II}}(t_{k_i}))$.This gives a more reliable approximation for the reference data $v(\boldsymbol{y}_j)$

$$v(\boldsymbol{y}_j) = \frac{1}{S(j)} \sum_{i=1}^{S(j)} v_{i,j}. \tag{C.6}$$

See Algorithm 4 for the full detail.

Algorithm 4: Reference data sampling for high dimensional spaces with conditional Gaussian structure

Input: Conditional linear stochastic differential equations (C.1) and (C.2). **Output:** Reference approximation $v(\boldsymbol{y}_j)$ at $\boldsymbol{y}_j = (\boldsymbol{y}_j^{\mathbf{I}}, \boldsymbol{y}_j^{\mathbf{II}}), j = 1, 2, \ldots, N^Y$.
Initialize $\boldsymbol{X}(0)$ and run a numerical simulation of equation (C.1)-(C.2) for sometime t_0 and "burn in". ;
Reset $\boldsymbol{X}(0) = \boldsymbol{X}(t_0)$.;
Continue the numerical simulation of (2.1) for a relatively large T and collect $\boldsymbol{X}_{\mathbf{I}}(s), s \le t$.;
Run a numerical solver for (C.3) and (C.4) to get $\overline{\boldsymbol{X}}_{\mathbf{II}}(t)$ and $\boldsymbol{R}_{\mathbf{II}}(t)$.;
for $j = 1$ *to* M **do**
  Record $\boldsymbol{X}(t_{k_i}), i = 1, 2, \ldots, S(j) \in B(\boldsymbol{y}_j^{\mathbf{I}}, h)$ in $\mathbb{R}^{n_{\mathbf{I}}}$.;
  Evaluate $v_{k,j}$ using (C.5).;
end
Return $v(\boldsymbol{y}_j), j = 1, 2, \ldots, N^Y$ using (C.6). ;

Appendix D. Additional numerical tests

In this section, we perform a few additional numerical tests to show the effectiveness of our data-driven Fokker-Planck solver.

D.1. Test on early stopping method.

The first test is on the early stopping method when the loss function only includes $L_2(\boldsymbol{\theta})$. Because it is known that a neural network learns the low frequency part of a function first Xu et al. (2020b),

there is a possibility that early stopping can make the solution more smooth. Hence one needs to exclude the possibility that the low performance when training the neural network without $\mathcal{L}u$ is caused by overtraining. To check this, we run the same numerical test that generates Figure 3 with smaller amount of training epochs. The early stopping method does not improve the accuracy in all cases that we have tested. In Figure 9, we show the result of using only 20% of the training epochs. We can see that early stopping does not improve the performance of training without $\mathcal{L}u$. This further justifies the use of $\mathcal{L}u$ term in the loss function.

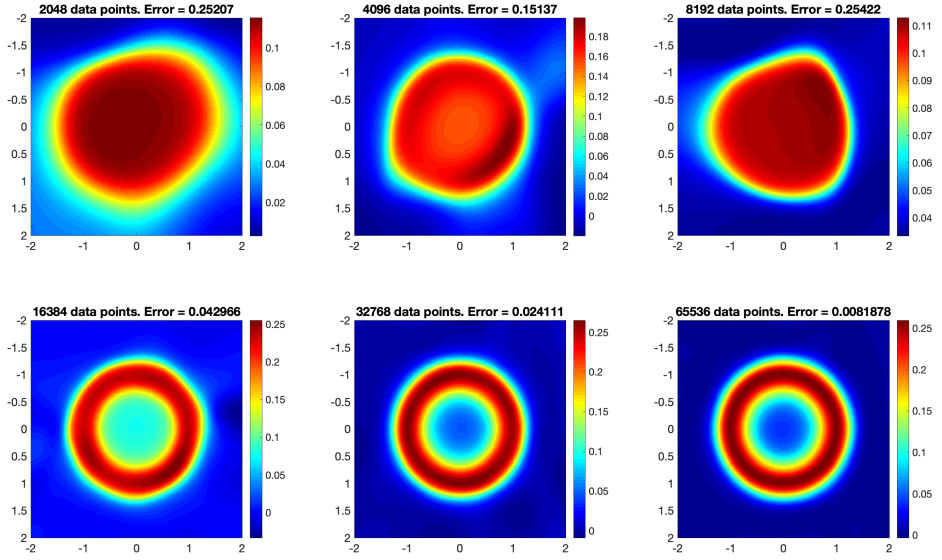

Figure 9: Ring example. A comparison of different sizes of reference set without $\mathcal{L}u$ being in the loss function when training with early stopping. Top left to bottom right: heat map of the invariant probability density function if the "ring model" with 2048, 4096, 8192, $16\,384$, $32\,768$, and $65\,536$ reference points are used. The L_2 error is shown in the title of each subplot.

D.2. Test on robustness of solutions.

The second numerical experiment tests the robustness of solutions. We run the same example in Figure 2 with 512 reference points 20 times, with different initializations each time. The initial values of the neural network parameters are drawn from a standard normal distribution. Each test stops after 20 epochs. Three runs out of 20 have slow convergence and have not converged very well when stopped at 20 epochs. Results of the rest 17 tests are in line with the results demonstrated in Figure 2. The result of three slow convergence tests is shown in Figure 10 together with the exact solution. We can see that they are already fairly close to the true solution. We expect them

to further converge to the true solution after a few more epochs. In general, the simulation result is more robust to different network initializations if more reference points are used.

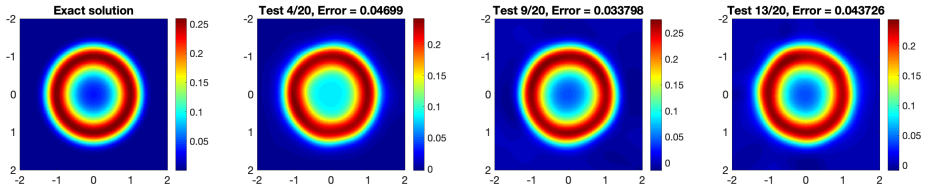

Figure 10: Ring example. Results of 3 slow converging tests out of 20 total tests and a comparison with the true invariant probability density function. The L_2 error is shown in the title of each subplot.

D.3. Test on smaller neural networks.

The third test compares the solution with smaller neural network. We run the same example in Figure 2 with 512 reference points with six smaller feed-forward neural network, denoted by small network 1 to 6. The first network has 5 hidden layers with $16, 128, 128, 128, 12$ neurons respectively. The second network has 4 layers with $128, 128, 128, 12$ neurons respectively. The third network has 3 hidden layers with $128, 128, 128$ neurons respectively. The fourth network has 3 hidden layers with $128, 128, 12$ neurons respectively. The fifth network has 3 hidden layers with $64, 64, 8$ neurons respectively. The sixth network has 2 hidden layers with $265, 16$ neurons respectively. The training result (after 20 epochs) is demonstrated in Figure 11. From these training results we can find that our Fokker-Planck solver still works on a suitable smaller neural network with $4 - 5$ layers. But networks with 3 layers or fewer do not have satisfactory performance, mainly because it cannot approximate the target function properly. In addition, the number of neurons in each layer should not drop too quickly when feeding to the output layer. Otherwise errors in the second-to-last layer can accumulate to the output layer and significantly interrupt the training result, which is the case of small network 3. Based on the result in this example and numerical tests in Appendix D.4 and D.5, we can see that for most problems, it is sufficient to have about 100 neurons in each hidden layer. However, if the target solution is either highly concentrated or in high dimension, wider hidden layers can significantly improve the approximating ability of the neural network.

D.4. Test on multimodal distribution.

The fourth test checks the performance of our Fokker-Planck solver on a multimodal distribution with weak noise, which is known to be a challenging issue for both numerical computations and sampling. In addition, it is known that the upper bound of the neural network approximation of a function f depends on the Barron norm of f Barron (1993), which is considerably higher if f is highly concentrated at the vicinity of some low dimensional sets. Hence it is important to understand the performance and limitations of the neural network Fokker-Planck solver in the setting of weak noise and multimodal distribution. Consider a Toggle Switch model that models a gene circuits with two genes G_A and G_B that produces proteins A and B respectively Newby (2012). Assume

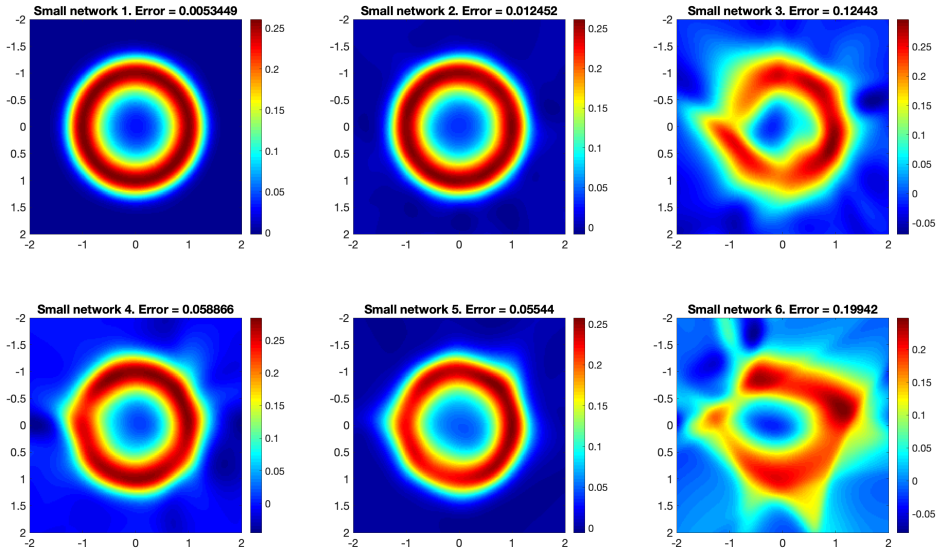

Figure 11: Ring example. Training results of 6 smaller neural networks are compared with the true solution. The L_2 error is shown in the title of each subplot.

that each protein turns off the other gene (A turns off G_B and B turns off G_A). Each off-gene turns itself back on with a certain rate. Let x and y be the concentration of proteins A and B. The system is

$$\mathrm{d}X_t = \left(\frac{b+x^2}{b+x^2+y^2} - x\right)\mathrm{d}t + \epsilon \mathrm{d}W_t^{(1)} \tag{D.1}$$

$$\mathrm{d}Y_t = \left(\frac{b+y^2}{b+x^2+y^2} - y\right)\mathrm{d}t + \epsilon \mathrm{d}W_t^{(2)}\,, \tag{D.2}$$

where $b = 0.25$ is a constant, ϵ is a changing parameter, and $W_t^{(1)}, W_t^{(2)}$ are two independent Wiener processes. It is easy to see that the deterministic part of equation (D.1) admits two stable equilibria P, Q and one saddle equilibrium R. When $0 < \epsilon \ll 1$, the solution concentrates in the vicinity of P and Q, while the probability density everywhere else is very low. This system has a very slow rate of convergence in this situation.

We compute the results produced by the data-driven solver with $\epsilon = 0.05, 0.1$, and 0.15. Since the invariant probability density of equation (D.1) cannot be explicitly given, we use the grid-based data-driven solver to compute it again on a 512×512 mesh as a comparison. Because of the very slow convergence of equation (D.1), we use 8 parallel trajectories from different initial values to balance the probability of sampling the neighborhood of P and Q. The result is shown in Figure 12. When the noise is small ($\epsilon = 0.05$), we enlarge the first hidden layer to 256 neurons because a small first hidden layer does not represent a function with very large derivatives very well. As seen in Figure 12, the neural network successfully converges in all three cases. But the weight of

two "wells" has some error when ϵ is very small. Overall the performance of the neural network solver is satisfactory when ϵ is as small as 0.1. However, very small noise does cause some problems. We expect this situation to be improved by using better Monte Carlo sampler such as the stratified Markov Chain Monte Carlo Dinner et al. (2018). In addition, wider neural networks could approximate probability distributions that are highly concentrated in low dimensional sets better than narrow networks.

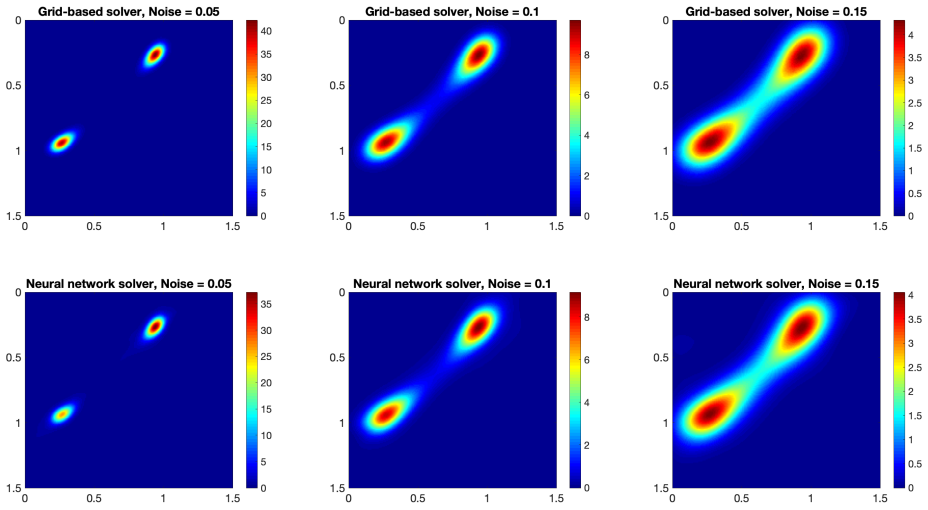

Figure 12: Toggle Switch model with $\epsilon = 0.05, 0.1$, and 0.15. Top row: result from the grid-based data-driven solver. Bottom row: result from the mesh-free data-driven solver.

D.5. Test on high dimensional problems.

The last numerical example tests the performance of our data-driven Fokker-Planck solver on a higher dimensional problem. It is very difficult to verify the result in high dimension, so we choose a linear system with known invariant probability measure. Consider the following SDE with linear deterministic part

$$\mathrm{d}X_t^{(1)} = (-2X_t^{(1)} + X_t^{(2)})\mathrm{d}t + \mathrm{d}W_t^{(1)} \tag{D.3}$$

$$\mathrm{d}X_t^{(i)} = (X^{(i-1)} - 2X_t^{(i)} + X_t^{(i+1)})\mathrm{d}t + \mathrm{d}W_t^{(i)} \qquad i = 2, \cdots, N-1 \tag{D.4}$$

$$\mathrm{d}X_t^{(N)} = (X_t^{(N-1)} - 2X_t^{(N)})\mathrm{d}t + \mathrm{d}W_t^{(N)}\,, \tag{D.5}$$

where $W_t^{(i)}, i = 1, \cdots, N$ are independent Wiener processes. This equation can be seen as a discrete approximation of the 1D stochastic heat equation Anton et al. (2020). Let $\mathbf{X} = (X^{(1)}, X^{(2)}, \cdots, X^{(N)})^T$ be the column vector of variables. The deterministic part of equation (D.3) has the matrix form $A\mathbf{X}$. The invariant probability measure of equation (D.3) satisfies

$$u(\mathbf{X}) = (2\pi)^{-N/2}|\Sigma|^{-1/2}\exp(-\frac{1}{2}\mathbf{X}^T\Sigma^{-1}\mathbf{X})\,,$$

where Σ solves the Lyapunov equation $A\Sigma + \Sigma A^T + 2I = 0$. Since A is symmetric, we have $\Sigma = \frac{1}{2}A^{-1}$. It is easy to see that the probability density on the $(X^{(1)}, X^{(2)})$-slice at the origin is independent of the dimension N.

We use the mesh-free data-driven solver to solve the invariant probability measure of equation (D.3) with $N = 10$ and $N = 20$. The probability density at reference points are from the theoretical invariant probability measure. We use 4000 reference points and 40000 training points for the 10D problem and 8000 reference points and 80000 referene points for the 20D problem. To accommodate more input neurons, we change the size of first hidden layer from 16 to 32. Number of neurons in other hidden layers are the same as in other numerical examples. It is not possible to fully visualize a high dimensional probability density function so we only plot the $(X^{(1)}, X^{(2)})$-slice at the origin. The result is compared with the theoretical density function in Figure 13. We can see in the top row that the result in 10D is still satisfactory. However, the solution has higher variance than it is expected to have in 20D. We believe this is because a high dimensional Gaussian distribution is actually more concentrated (because the volume of high dimensional unit sphere converges to zero). Also it is easy to check that the Barron norm of a high dimensional Gaussian distribution is exponentially higher, so the theoretical upper error bound of the neural network approximation grows exponentially. This may contribute to higher error that we have observed in the 20D problem. In general, the small neural network we use has some limitation in dealing with highly concentrated probability density functions in higher dimensions. The performance of the data-driven Fokker-Planck solver in high dimension could be improved by using wider neural networks. In the second row of Figure 12, we further increase the width of the first hidden layer from 32 to 128. And the problem of "artificial diffusion" seen in the numerical result is clearly alleviated. We expect larger neural networks to have better performance in those high dimensional problems.

Appendix E. Numerical simulation details

E.1. Parameter of the neural network.

Throughout this paper, unless otherwise specified, we use a small feed-forward neural network with 6 hidden layers, each of which contains $16, 128, 128, 128, 16, 4$ neurons respectively, to approximate the solution to Fokker-Planck equation in all numerical examples. The output layer always has one neuron. Number of neurons in the input layer depends on the problem. All activation functions are the sigmoid function. We choose sigmoid function because (1) the solution of the Fokker-Planck equation is everywhere nonnegative and (2) the second order derivative of the neural network output is included in the loss function.

E.2. Numerical example 1.

In Figure 1, the Monte Carlo solution is obtained by running an Euler-Maruyama numerical scheme for (4.1) with 10^7 steps and calculating the empirical probability on a 200×200 mesh of the region $D = [-2, 2] \times [-2, 2]$. The time step size is 0.001. Optimization problems in equation (2.4) and (2.5) are solved by linear algebra solvers. The neural network training with loss function 3.1 uses all probability densities at grid points obtained by the same Monte Carlo simulation. The architecture of the artificial neural network is described in Section E.1 with two input neuron and one output neuron.

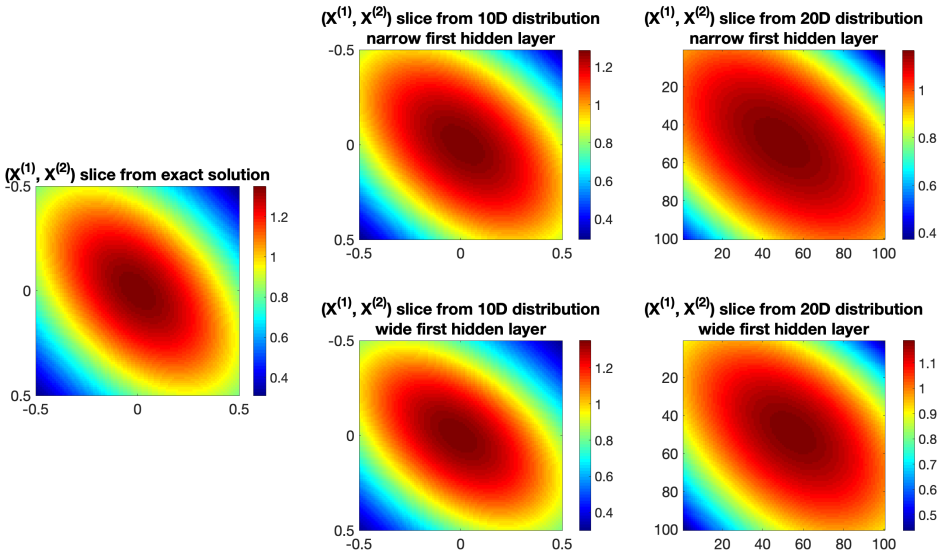

Figure 13: Invariant probability density of system (D.3) on $(X^{(1)}, X^{(2)})$ slice at the origin. Left: true solution. Top Middle: 10 dimensional problem with narrow first hidden layer (32 neurons). Bottom Middle: 10 dimensional problem with wide first hidden layer (128 neurons). Top Right: 20 dimensional problem with narrow first hidden layer (32 neurons). Bottom Right: 20 dimensional problem with wide first hidden layer (128 neurons).

In Figure 2, Algorithm 2 is used to generate reference points and training points. The number of reference points in six panels of Figure 2 are $32, 64, 128, 256, 512$, and 1024 respectively. The number of training points is 10000 in all cases. All probability densities $v(\boldsymbol{y}_i)$ at training points are obtained by Algorithm 3, which runs the Euler-Maruyama scheme for 10^8 steps. Then we train the artificial neural network with loss function (3.1). The architecture of the artificial neural network is described in Section E.1 with two input neuron and one output neuron. The trained neural network is evaluated on a 400×400 grid.

When generating Figure 3, we use the loss function without $\mathcal{L}u$, so there is no training set $\mathfrak{X}$. In six panel of Figure 3, the numbers of reference points $v(\boldsymbol{y}_j)$ with probability densities are 2048, 4096, 8192, $16\,384$, $32\,768$, and $65\,536$, respectively. Reference points are obtained by Algorithm 2. The probability density at each reference point is exact (obtained from the Gibbs density). The architecture of the artificial neural network is described in Section E.1 with two input neuron and one output neuron. The trained neural network is evaluated on a 400×400 grid.

E.3. Numerical example 2.

In Figure 4, Algorithm 2 is used to generate 1024 reference points and 10000 training points. Then we artificially inject some noise into the training data. For a reference point $\boldsymbol{y}_i \in \mathfrak{Y}$, we have $v_i(\boldsymbol{y}_j) = r_i(\boldsymbol{y}_j)u(\boldsymbol{y}_j)$, where u is the Gibbs density, $r_i \sim U([1-\alpha, 1+\alpha])$ is a random variable uniformly distributed in a range $[1-\alpha, 1+\alpha]$. Here, α controls the "strength" of the artificial noise. Four different values $\alpha = 0.01, 0.05, 0.1$ and 0.5 are used to generate four different reference data sets $\{v(\boldsymbol{y}_j)\}_{j=1}^{1024}$. The architecture of the artificial neural network is described in Section E.1 with two input neuron and one output neuron. The trained neural network is evaluated on a 400×400 grid.

In Figure 5, the conditional Gaussian simulation is obtained by running Algorithm 4. The trajectory is recorded at 50000 discrete times. The probability density at 400×400 grid points are evaluated by using Algorithm 4. (Two panels in the first column). Next, Algorithm 2 is used to generate 1024 reference points and 10000 training points. The probability densities at those training points are evaluated by running Algorithm 4 (same sample size as above) and Algorithm 3, respectively. Two sets of probability densities at reference points are used in the neural network training (with loss function (3.1)) to generate subplots in the second and third column, respectively. The architecture of the artificial neural network is described in Section E.1 with two input neuron and one output neuron. The trained neural network is evaluated on a 400×400 grid.

E.4. Numerical example 3.

In Figure 6, algorithm 2 is used to sample 10000 reference points and 10^5 training points in the domain $D = [-2, 2]^4$. Then we run algorithm 3 with 10^{10} steps of the Euler-Maruyama scheme to estimate the probability densities at training points. The values of the probability density function are rescaled on the whole domain D such that the maximum is 1 (for otherwise the neural network cannot easily learn the distinction among small values). Then we train the artificial neural network with loss function (3.1). The architecture of the artificial neural network is described in Section E.1 with four input neuron and one output neuron. The trained neural network is evaluated at four (x, y)-slices for $(z, s) = (0, 0), (0.5, 0.5), (1, 0)$ and $(1, 1)$ respectively. Each (x, y) slice contains 400×400 grid points.

E.5. Numerical example 4.

In Figure 7, the numerical domain is $[-3,3]\times[-3,0]\times[-1.5,0.5]\times[-0.5,0.5]\times[-0.5,0.5]\times[-0.5,0.5]\subset\mathbb{R}^6$. A direct Monte Carlo simulation that uses 8×10^9 steps of Euler-Maruyama scheme is used to generate subplots in the first row. The grid size of the Monte Carlo simpler is 0.05. Then we use Algorithm 2 to sample 20000 reference points and 10^5 training points in the domain D. Two sets of probability densities at reference points are obtained using two approaches. The first approach uses Algorithm 3 with 4×10^{10} steps of the Euler-Maruyama scheme. The second approach uses 8×10^5 samples from the conditional Gaussian sampler in Algorithm 4. The values of the probability density function are rescaled on the whole domain D such that the maximum is 1. Then we train two artificial neural networks (with loss function (3.1)) using the same collocation points but two sets of probability densities at reference points. The results are shown in the second and third row of Figure 7 respectively. The architecture of the artificial neural network is described in Section E.1 with six input neuron and one output neuron. The trained neural network is evaluated at five (u,v_i)-slices for $i=1,\cdots,5$ centering at the origin.