Applied Mathematics and Computation 457 (2023) 128185

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

Artificial neural network solver for time-dependent Fokker–Planck equations

Yao Li*, Caleb Meredith

Department of Mathematics and Statistics, University of Massachusetts Amherst, Amherst, MA 01002, USA

ARTICLE INFO

Article history:
Received 3 January 2023
Revised 27 April 2023
Accepted 10 June 2023

ABSTRACT

Stochastic differential equations (SDEs) play a crucial role in various applications for modeling systems that have either random perturbations or chaotic dynamics at faster time scales. The time evolution of the probability distribution of a stochastic differential equation is described by the Fokker–Planck equation, which is a second order parabolic partial differential equation (PDE). Previous work combined artificial neural networks and Monte Carlo data to solve stationary Fokker–Planck equations. This paper extends this approach to time dependent Fokker–Planck equations. The main focus is on the investigation of algorithms for training a neural network that has multi-scale loss functions. Additionally, a new approach for collocation point sampling is proposed. A few 1D and 2D numerical examples are demonstrated.

1. Introduction

The Fokker–Planck equation plays an important role in various applications because it describes the time evolution of a stochastic differential equation, which is widely used to study noise-perturbed systems or models. Since most Fokker–Planck equations have no explicit solution, numerical Fokker–Planck solvers are necessary. Previously, the main difficulty of solving a Fokker–Planck equation was that the long-term stability of the Fokker–Planck solution comes from the drift term of the stochastic differential equation rather than its own boundary condition. The lack of a suitable boundary condition on the numerical domain, coupled with high dimensionality, makes many traditional methods less effective. This problem is partially solved by the first author's series of papers [1,2], in which a data-driven Fokker–Planck solver is developed. An artificial neural network version of the data-driven Fokker–Planck solver for stationary Fokker–Planck equations is proposed and studied in Zhai et al. [3]. In this paper, we will both extend the work in Zhai et al. [3] to time-dependent Fokker–Planck equations and further investigate the training methods of artificial neural networks for the neural network Fokker–Planck solver. Many unaddressed problems regarding neural network training and training point sampling in Zhai et al. [3] are studied in this paper.

The main idea of the data-driven solver is that the Fokker–Planck equation has a probabilistic representation, hence its solution can be approximated by a Monte Carlo simulation. The data-driven solver only requires rough Monte Carlo simulation data, which is highly noisy but can be obtained at low computational cost. One important observation is that

* Corresponding author.
E-mail addresses: yaoli@math.umass.edu (Y. Li), cmeredith@umass.edu (C. Meredith).

https://doi.org/10.1016/j.amc.2023.128185
0096-3003/

the error term in the Monte Carlo simulation data is largely spatially uncorrelated. Therefore, the Monte Carlo simulation data can be used to guide either a classical PDE solver or an artificial neural network. The data-driven Fokker–Planck solver can be seen as a data regularization process: the noisy Monte Carlo data is regularized by the Fokker–Planck operator. The goal of training is to make the solution fit the data and satisfy the Fokker–Planck equation (or its discretization). This idea is similar to the physics-informed neural network (PINN) [4,5]. The main difference is that the values at collocation points are from Monte Carlo sampling rather than initial or boundary conditions.

The loss function of the neural network Fokker–Planck solver has two parts: one comes from the Fokker–Planck operator, denoted by L_1^{loss}, and the other comes from the Monte Carlo approximation of the Fokker–Planck solution, denoted by L_2^{loss}. Due to the low accuracy of the Monte Carlo simulation, the neural loss has a multi-scale feature. In the early phase of training, a randomly generated artificial neural network usually has large second-order derivatives, so we have $L_1^{\text{loss}} \gg L_2^{\text{loss}}$. Later in the training, L_2^{loss} may become the dominant term due to errors in the Monte Carlo approximation. In [3], the problem of optimizing two loss functions at different scales was solved by an algorithm called "Alternating Adam," which alternates two Adam optimizers [6] for the two loss functions. Interestingly, we later found that this approach does not work well when the differential operator in the loss term is not fully elliptic, which includes time-dependent Fokker–Planck equations and the stationary Fokker–Planck equation with a degenerate elliptic term.

Therefore, in this paper we use time-dependent Fokker–Planck equations as an example to carefully examine the methods of optimizing artificial neural networks and sampling training points. We test and compare several different methods. Ultimately, we conclude that the most robust training method is the "Gradient-Based Momentum Weight" method, which gradually changes the relative weight of the two loss functions based on the gradients of the two loss functions from the previous epoch. We also test different methods of sampling training points. In addition to sampling training points proportional to the probability density, as discussed in Zhai et al. [3], we find that it is beneficial to concentrate collocation points (meaning training points with approximated probability density) at the initial distribution and a few selected time "anchors." We believe this is because the temporal variable is only regularized by the first-order derivative. As a result, the neural network can easily learn the "shape" of the solution but needs more data at "anchors" to learn the correct scale of the solution.

We remark that this paper is *not* a trivial generalization of Zhai et al. [3]. It carefully investigates the training methods of the artificial neural network with multiple loss functions at different scales. It is known that PINNs have similar issues when the data at collocation points come from experiments [7]. Many non-PDE neural network training methods also need to balance training loss functions at different scales. In the examples that we have tested, the new training methods developed in this paper have superior performance compared to both the "Alternative Adam" proposed in Zhai et al. [3] and the idea of trainable weight proposed in Gu et al. [8], Liu and Wang [9], McClenny and Braga-Neto [10]. We expect these new discoveries to be applied to other applications in the future.

The organization of this paper is as follows. Section 2 reviews stochastic differential equations, the Fokker–Planck equation, and the data-driven Fokker–Planck solver with both the discretization version and the neural network version. The neural network solver is described in Section 3. Section 4 investigates a few different ideas for training a neural network with multi-scale loss functions, which is one of the main focuses of this paper. The numerical examples are demonstrated in Section 5. Section 6 presents some numerical examples to demonstrate the improved training results from a better sampling method. Section 7 is the conclusion. The Appendix discusses the implementation of training methods, hyper-parameter selections, training point selection, and further performance improvements in full detail.

2. Preliminary

2.1. Stochastic differential equations and the Fokker–Planck equation

Consider a stochastic differential equation (SDE) that is of the form

$$d\boldsymbol{X}_t = f(\boldsymbol{X}_t)dt + \sigma(\boldsymbol{X}_t)\mathrm{d}\boldsymbol{W}_t, \tag{1}$$

where f is a vector field in $\mathbb{R}^d$, σ is a $d \times m$ matrix-valued function, and $\boldsymbol{W}_t$ is the standard Wiener process in $\mathbb{R}^m$. The solution of (1), denoted by $\boldsymbol{X} = \{\boldsymbol{X}_t \mid t \in \mathbb{R}\}$, is a stochastic process on $\mathbb{R}^d$. Since the theme of this paper is about the numerical method, throughout this paper we assume that f and σ have sufficient regularities such that Eq. (1) admits a weak solution. It is well known that $\boldsymbol{X}_t$ is a continuous-time Markov process with an infinitesimal generator $\mathcal{L}$ satisfying

$$\mathcal{L}h = -\sum_{i=1}^{n} f_i h_{x_i} + \frac{1}{2}\sum_{i,j=1}^{n} D_{i,j} h_{x_i,x_j}, \tag{2}$$

where $D = \sigma^T\sigma$ is a $d \times d$ matrix-valued function.

The Fokker–Planck equation is a parabolic partial differential equation that describes the time evolution of the probability density function of an SDE. More precisely, let $u = u(t, \boldsymbol{x})$ be the probability density function of the solution $\boldsymbol{X}_t$ to Eq. (1), such that $u(t, \boldsymbol{x})$ is the probability density at $\boldsymbol{x} \in \mathbb{R}^d$ at time t. Let $D = \sigma^T\sigma$ be the diffusion matrix. The Fokker–Planck

equation reads

$$u_t = \mathcal{L}^* u = -\sum_{i=1}^{n} (f_i u)_{x_i} + \frac{1}{2} \sum_{i,j=1}^{n} (D_{i,j} u)_{x_i, x_j}, \tag{3}$$

where $\mathcal{L}^*$ is the adjoint operator of the generator $\mathcal{L}$.

In addition, if the SDE (1) admits an invariant probability measure π, then the probability density function of π, denoted by $u(\boldsymbol{x})$, must satisfy the stationary Fokker–Planck equation, which is given by

$$\mathcal{L}^* u = 0 \text{ and } \int_{\mathbb{R}^d} u d\boldsymbol{x} = 1, \tag{4}$$

2.2. Data-driven stationary Fokker–Planck equation solver

The Fokker–Planck solver studied in this paper is based on the data-driven solver for stationary Fokker–Planck equations described in Li [2]. As discussed in the introduction, when solving the Fokker–Planck equation, many traditional PDE solvers have problems with unbounded domains and high dimensionality, while Monte Carlo simulations usually have accuracy issues. This problem is partially solved by the data-driven hybrid method proposed in Dobson et al. [1], which considers local Fokker–Planck equations on a subset of the entire domain without the knowledge of the boundary condition. Instead, Monte–Carlo simulation is used to provide a reference solution that makes up for the lack of a boundary condition.

Take the 2D stationary Fokker–Planck as an example. Let $D = [a_0, b_0] \times [a_1, b_1]$ be the numerical domain, which is further split into an $N \times M$ grid of boxes. The numerical solution $\boldsymbol{u} \in \mathbb{R}^{N \times M}$ of the stationary Fokker–Planck equation is an approximation of the probability density of u at the center of each grid box. Now let $\boldsymbol{A}$ represent the discretization of the operator $\mathcal{L}^*$ on D with respect to all interior boxes. Then $\boldsymbol{A}$ is an $(N-2)(M-2) \times (NM)$ matrix that provides the linear constraint on $\boldsymbol{u}$ given by

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{0} \tag{5}$$

Next, we run a long trajectory of $\boldsymbol{X}_t$ and count the sample points in each grid box, which gives an approximated invariant probability density function denoted by $\boldsymbol{v} = \{v_{i,j}\}_{i=1,j=1}^{i=N,j=M}$ The numerical solution $\boldsymbol{u}$ is then given by the optimization problem

$$\begin{cases} \min_{\boldsymbol{u}} ||\boldsymbol{u} - \boldsymbol{v}||_2 \\ \text{subject to } \boldsymbol{A}\boldsymbol{u} = \boldsymbol{0} \end{cases} \tag{6}$$

It is further proved in Dobson et al. [1] that the error in the reference solution $\boldsymbol{v}$ is significantly removed by the projection in solving the optimization problem.

The data-driven solver for stationary Fokker–Planck equations has an artificial neural network version proposed in Zhai et al. [3]. The idea is that the constrained optimization problem above can be replaced by a unconstrained optimization problem

$$\min_{\boldsymbol{u}} ||\boldsymbol{A}\boldsymbol{u}||_2^2 + ||\boldsymbol{u} - \boldsymbol{v}||_2^2 \tag{7}$$

that preserves the key numerical properties of the original data-driven solver in Li [2]. This motivates us to represent $\boldsymbol{u}$ by an artificial neural network $\tilde{u}(\boldsymbol{x}_i, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the trainable parameters. Since artificial neural networks are differentiable, we can further replace the discretized operator $\boldsymbol{A}$ by the differential operator $\mathcal{L}^*$. Instead of the whole domain, the optimization problem is solved with respect to a set of training points.

Mimicking the unconstrained optimization problem in (7), a loss function $\bar{L}(\boldsymbol{\theta})$ is given by

$$\bar{L}(\boldsymbol{\theta}) = \frac{1}{N^X} \sum_{i=1}^{N^X} (\mathcal{L}^* \tilde{u}(\boldsymbol{x}_i, \boldsymbol{\theta}))^2 + \frac{1}{N^Y} \sum_{j=1}^{N^Y} (\tilde{u}(\boldsymbol{y}_j, \boldsymbol{\theta}) - v(\boldsymbol{y}_j))^2 \tag{8}$$

where $\boldsymbol{\theta}$ represents the neural network parameters that can be updated during training, $\tilde{u}(\boldsymbol{x}, \boldsymbol{\theta})$ is the neural network approximation for the probability density at $\boldsymbol{x}$ for specific parameters $\boldsymbol{\theta}$, $\boldsymbol{x}_i, \boldsymbol{y}_j \in \mathbb{R}^n$ for $i \in 1, 2, \ldots, N^X$ and $j \in 1, 2, \ldots, N^Y$ are training points without Monte Carlo approximation and collocation points with Monte Carlo approximation respectively, and $v(\boldsymbol{y}_i)$ are the Monte Carlo approximations for the probability density at those collocation points.

3. Neural network solver for time-dependent Fokker–Planck equations

The general idea behind our artificial neural network solver for time-dependent Fokker–Planck equations largely resembles the stationary case, although the implementation and training details have many differences.

Consider the initial value problem

$$\begin{cases} u_t = \mathcal{L}^* u = -\sum_{i=1}^{n} (f_i u)_{x_i} + \frac{1}{2} \sum_{i,j=1}^{n} (D_{i,j} u)_{x_i, x_j} \\ u(0, \boldsymbol{x}) = u_0(\boldsymbol{x}) \end{cases} \tag{9}$$

Similarly to the stationary case, the Fokker–Planck equation is defined on an unbounded domain and the only boundary condition is that $u(0, \boldsymbol{x})$ vanishes at infinity. Let $[0, T] \times D \subset \mathbb{R} \times \mathbb{R}^d$ be the numerical domain that we are interested in. Let $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ be the neural network approximation of the time dependent solution to the Fokker–Planck Eq. (9). Let $\mathfrak{X} := \{(t_i, \boldsymbol{x}_i) \in [0, T] \times D \mid i \in 1, 2, \ldots, N^X\}$ be training points without Monte Carlo approximation, $\mathfrak{Y} := \{(t_j, \boldsymbol{y}_j) \in [0, T] \times D \mid j \in 1, 2, \ldots, I^Y, I^Y + 1, \ldots, N^Y\}$ be collocation points from the initial distribution for $j \leq I^Y$ and collocation points with Monte Carlo approximation for $j > I^Y$, and $v(t_j, \boldsymbol{y}_j)$ for $j \in 1, 2, \ldots, N^Y$ be $u_0(\boldsymbol{y}_j)$ if $j \leq I^Y$ and Monte Carlo approximation of the probability density at collocation point $(t_j, \boldsymbol{y}_j)$ for $j > I^Y$. Similarly to (8), we attempt to minimize the optimization function

$$L(\boldsymbol{\theta}) = \frac{1}{N^X} \sum_{i=1}^{N^X} (\mathcal{L}^* \tilde{u}(t_i, \boldsymbol{x}_i, \boldsymbol{\theta}) - \tilde{u}_t(t_j, \boldsymbol{x}_i, \boldsymbol{\theta}))^2 + \frac{1}{N^Y} \sum_{j=1}^{N^Y} (\tilde{u}(t_j, \boldsymbol{y}_j, \boldsymbol{\theta}) - v(t_j, \boldsymbol{y}_j))^2 := L_1^{\text{loss}} + L_2^{\text{loss}} \tag{10}$$

Below we will address three key components of the neural network Fokker–Planck solver, i.e., the selection of collocation points, the Monte Carlo simulation that provides a reference solution, and the training of the artificial neural network.

3.1. Sampling collocation points

To train the neural network, we must first sample points for $\mathfrak{X}$ and $\mathfrak{Y}$. The standard method is based on the sampling method used in Dobson et al. [1], and can be used for sampling both $\mathfrak{X}$ and $\mathfrak{Y}$. It consists of two parts, sampling uniformly across the entire numerical domain, and sampling proportional to the probability density function. Due to the fact that the density tends to concentrate near global attractors of the deterministic part of the SDE, solely uniform sampling may not be sufficient, as too many points may be chosen from low density regions. On the other hand, solely sampling according to the probability density leaves scarce points in low density regions, which can cause notable error. This can be resolved by sampling α% of the points uniformly and $(1 - \alpha)$% of the points proportional to density for some $\alpha \in [0, 1]$. Sampling collocation points according to the probability density of the solution means to sample them from trajectories of equation (1). Those samples are collected at discrete times because Eq. (1) is computed numerically at multiples of a fixed step size $0 < \delta t \ll 1$. Since δt is very small (0.001 in our simulations), it would not affect the result because very few, if any, collocation points may have the same time component. To facilitate Monte Carlo approximation of the probability density function, when sampling $\mathfrak{Y}$ we move the collocation point $(t_i, \boldsymbol{y}_i)$ to the center of the grid h-box it is in if $i > I^Y$. However, when uniformly sampling for $\mathfrak{X}$ there is no need to constrain the points to a grid, so the uniformly sampled points are left unaltered. The pseudocode for this algorithm can be seen below in Algorithm 1. For simplicity, assume that $D = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$ has been split into a grid of boxes with side length h, and $[0, T]$ has been discretized with time steps of δt.

An alternative of Algorithm 1 is the "anchor method", which is introduced in this paper and explored in more detail in Section 6. The main motivation of the anchor method is that the spatial and temporal variables of the solution are regularized by the first and second order derivatives respectively. The second order derivative is more sensitive against changes of network parameters. Hence the spatial variables are much easier to train. Since the solution of the Fokker–Planck equation is uniquely determined by its initial value, training L_2^{loss} with just points from the initial distribution and L_1^{loss} with the standard set of points (See Section 3.3 for details) can usually determine the shape of the solution. However, the scale of the solution is usually less accurate the further away from the initial distribution it is calculated because the temporal variable is less regularized. Zhai et al. [3] demonstrated that the inclusion of the Fokker–Planck operator in the loss function allowed comparable training to be done with a far sparser set of collocation points, indicating that it should also be possible to use a sparse set of collocation points with respect to the time domain to "anchor" the scale of the solution. However, we found that it is sufficient to concentrate these points at as few as one time step. To update Algorithm 1 for use with the anchor sampling method with one set of anchoring points, we simply let $t_i = T$ for $i > I^Y$ ($\delta t = T$) and set I^Y close to N^Y when sampling collocation points in $\mathfrak{Y}$. The rest is identical to what is described in Algorithm 1.

3.2. Monte Carlo simulation

In order to run the neural network Fokker–Planck solver, an estimate of the probability density at the collocation points $(t_j, \boldsymbol{y}_j)$ is essential. Generally speaking the number of collocation points does not need to be very large. They only provide the role of "anchoring" the solution at the right place, while L_1^{loss} drives the neural network approximation to the solution of the Fokker–Planck equation.

The first step is to sample the initial distribution $u_0(\boldsymbol{x})$, which is done here using a rejection-based method. More precisely, a random variable $\boldsymbol{y} \in \mathbb{R}^d$ is sampled uniformly from the numerical domain D. Next, an auxiliary random variable p is uniformly sampled from $(0, \sup_{\boldsymbol{x} \in D} u_0(\boldsymbol{x}))$. The sample $\boldsymbol{y}$ is accepted if and only if $p < u_0(\boldsymbol{y})$. The process is repeated until an initial value $\boldsymbol{y}$ is accepted. We note that if the initial distribution is highly concentrated, other methods such as an MCMC sampler can also be used.

Next, we use Monte Carlo simulation to approximate the probability densities at the collocation points away from the initial distribution. Trajectories of Eq. (1) are approximated by the Euler–Maruyama scheme. The time interval $[0, T]$ is divided into L steps with $\delta t = T/L$. Let $\tau_j = j\delta t$ for $j \in 0, \ldots, L$ and $\boldsymbol{X}_j = \boldsymbol{X}_{\tau_j}$. The initial value $\boldsymbol{X}_0$ is given by $\boldsymbol{y}$ sampled from

Algorithm 1 Training and collocation point sampling.

Input: $\alpha \in [0,1]$, δt, N^X or N^Y and I^Y.
Output: $\mathfrak{X}$ or $\mathfrak{Y}$.

if $\mathfrak{X}$ **then**
  $M = N^X$, $t_0 = 0$, $i_1 = 1$
else
  $M = N^Y$, $t_0 = \delta t$, $i_1 = I^Y + 1$
  Sample I^Y points from initial distribution for $\boldsymbol{V}_1$ through $\boldsymbol{V}_{I^Y}$.
end if
for $i = i_1$ to $i = M$ **do**
  Uniformly sample $c_i \in [0,1]$
  if $c_i < \alpha$ **then**
    Uniformly sample $(t_i, \boldsymbol{r}_i) \in [0,T] \times D$ with $\boldsymbol{r}_i = (r_1, \ldots, r_d)$
    if $\mathfrak{X}$ **then**
      Set $\boldsymbol{V}_i = (t_i, \boldsymbol{r}_i)$
    else
      Set t_i to closest lower multiple of δt with $t_i = \lfloor \frac{t_i}{\delta t} \rfloor \delta t$.
      Set $\boldsymbol{r}_i$ to the center of the h-box it belongs to with $r_j = \lfloor \frac{r_j - a_j}{h} \rfloor h + a_j + \frac{h}{2}$
      Set $\boldsymbol{V}_i = (t_i, \boldsymbol{r}_i)$
    end if
  else
    Uniformly sample $t_i \in [t_0, T + \delta t]$
    Set t_i to closest lower multiple of δt with $t_i = \lfloor \frac{t_i}{\delta t} \rfloor \delta t$.
    Run a numerical trajectory of the SDE to time t_i.
    Let $\boldsymbol{r}_i = \boldsymbol{X}_{t_i} = (r_1, \ldots, r_d)$
    if $\mathfrak{X}$ **then**
      Set $\boldsymbol{V}_i = (t_i, \boldsymbol{r}_i)$
    else
      Set $\boldsymbol{r}_i$ to the center of the h-box it belongs to with $r_j = \lfloor \frac{r_j - a_j}{h} \rfloor h + a_j + \frac{h}{2}$
      Set $\boldsymbol{V}_i = (t_i, \boldsymbol{r}_i)$
    end if
  end if
end for
Return $\boldsymbol{V}$

the initial distribution. Then we have

$$\boldsymbol{X}_j = \boldsymbol{X}_{j-1} + f(\boldsymbol{X}_{j-1})\delta t + \sigma(\boldsymbol{X}_{j-1})\sqrt{\delta t}\mathcal{N}^m(0,1), \tag{11}$$

where $\mathcal{N}^m(0,1)$ is a vector in $\mathbb{R}^m$ with each entry an i.i.d. standard normal random variable.

The approximate probability density function can be obtained from repeatedly computing and recording numerical trajectories. As mentioned previously, all t_j of the collocation points $(t_j, \boldsymbol{y}_j)$ are integer multiples of the time step δt, i.e., $t_j = j\delta t$. After simulating one numerical trajectory of the Euler–Maruyama scheme, we check whether the trajectory hits the h-box centering at each collocation point $\boldsymbol{y}_j$ at time t_j, or whether $\|\boldsymbol{X}_j - \boldsymbol{y}_j\|_\infty < h/2$. This process is repeated M times. If we assume that the h-box centering at collocation point $\boldsymbol{y}_j$ is hit by realizations of $\boldsymbol{X}_t$ a total of M_j times at time t_j, the probability density $u(t_j, \boldsymbol{y}_j)$ is approximated by $v(t_j, \boldsymbol{y}_j) = M_j h^{-d} M^{-1}$. If the dimension of D is high (4 or larger), one can use kernel density estimation to improve the Monte Carlo estimation of $u(t_j, \boldsymbol{y}_j)$. But in general M does not have to be very large, as the neural network solver can tolerate large spatially uncorrelated noise [3]. In practice $M = 10^6$ to 10^8 is sufficient for most 2D and 3D problems.

The pseudocode for the Monte Carlo simulation algorithm can be seen below in Algorithm 2. The Monte Carlo probability density approximation at can be found at $G[\lfloor \frac{t}{\delta t} \rfloor][\lfloor \frac{y_1 - a_1}{h} \rfloor][\lfloor \frac{y_2 - a_2}{h} \rfloor] \ldots [\lfloor \frac{y_d - a_d}{h} \rfloor]$ for $t \geq \delta t$, since this algorithm does not approximate the probability density at the initial distribution where it is already explicitly known. If the dimension of D is too high to use a grid G, the grid-free sampling method described in Zhai et al. [3] can also be used.

3.3. Neural network training

The last and the most important step is to train the artificial neural network to minimize the loss function $L(\boldsymbol{\theta})$ given in (10). Throughout this paper, we use an artificial neural network with 6 feed forward hidden layers. The neural network architecture for the neural network has node counts given by $(d+1) \to 16 \to 256 \to 256 \to 256 \to 16 \to 4 \to 1$, where d is

Algorithm 2 Monte Carlo probability density estimation.

Input: $u_0(\boldsymbol{y})$, δt, M, $s = \sup_{\boldsymbol{x}\in D} u_0(\boldsymbol{x})$.
Output: Grid of Monte Carlo estimations G.

for $k = 1$ to $k = M$ **do**
  Uniformly sample $\boldsymbol{y} \in D$ and $p \in (0, s)$
  while $u_0(\boldsymbol{y}) > p$ **do**
    Regenerate $\boldsymbol{y} \in D$ and $p \in (0, s)$
  end while
  Let $\boldsymbol{X}_0 = \boldsymbol{y}$
  for $i = 1$ to $i = L$ **do**
    $\boldsymbol{X}_i = \boldsymbol{X}_{i-1} + f(\boldsymbol{X}_{i-1})\delta t + \sigma(\boldsymbol{X}_{i-1})\sqrt{\delta t}\mathcal{N}^m(0, 1)$
    Denote $\boldsymbol{X}_i = (x_1, \ldots, x_d)$
    Compute grid position of point with $\tilde{x}_j = \lfloor \frac{x_j - a_j}{h} \rfloor$
    if $\boldsymbol{X}_i \in D$ **then**
      $G[i][\tilde{x}_1][\tilde{x}_2]\ldots[\tilde{x}_d] \mathrel{+}= M^{-1}h^{-d}$
    end if
  end for
end for
Return G

the dimension of the phase space. Each layer uses a sigmoid activation function, and the Adam Optimizer is used for the optimization.

The loss function (10) is a combination of two loss functions

$$\begin{cases} L_1^{\text{loss}}(\boldsymbol{\theta}) = \frac{1}{N^X}\sum_{i=1}^{N^X}(\mathcal{L}^*\tilde{u}(t_j, \boldsymbol{x}_i, \boldsymbol{\theta}) - \tilde{u}_t(t_j, \boldsymbol{x}_i, \boldsymbol{\theta}))^2 \\ L_2^{\text{loss}}(\boldsymbol{\theta}) = \frac{1}{N^Y}\sum_{j=1}^{N^Y}(\tilde{u}(t_j, \boldsymbol{y}_j, \boldsymbol{\theta}) - v(t_j, \boldsymbol{y}_j))^2 \end{cases} \tag{12}$$

Depending on the accuracy of Monte Carlo sampler, the loss values for these two loss functions may have very different scales. Generally speaking L_1^{loss} is large in the beginning of the training because a randomly given neural network usually has large second order derivatives. However, L_2^{loss} could be larger than L_1^{loss} at the end of training if the Monte Carlo approximation v is not very accurate. This property of the loss functions needs to be carefully addressed. If one simply runs the Adam optimizer for the sum $L_1^{\text{loss}} + L_2^{\text{loss}}$, or the weighted sum $L_1^{\text{loss}} + \theta L_2^{\text{loss}}$ for some θ, then one loss function can dominate the other and yield unsatisfactory results. To resolve these issues, training algorithms needed to be developed to evenly balance the two loss functions. This is one of the main focuses of this paper, and will be addressed in detail in the next section.

4. Training algorithms

This section provides an overview of a number of training algorithms introduced or applied in this paper, as well as the motivation for their use. Hyper-parameter selection and performance sensitivity to those values will be discussed in the Appendix.

4.1. Alternating adam

The first training algorithm we will consider is Alternating Adam, the training algorithm used for the stationary case in Zhai et al. [3]. It will later serve as a performance benchmark.

The idea behind this algorithm is that the Adam optimizer is scaling free. Therefore, to account for the difference in scale between the loss functions, they can be separated and alternatively trained on their own mini-batches until both of their loss values are low enough. Doing so avoids the need to find a way to balance the two loss functions given that they are being trained in isolation. Alternating Adam is a relatively simple algorithm to implement and served well for the stationary case where the loss function dynamics were less extreme. The pseudocode for Alternating Adam can be seen below in Algorithm 3.

Algorithm 3 Alternating adam.

1: Initialize a neural network representation $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ with undetermined parameters $\boldsymbol{\theta}$.
2: Pick a mini-batch in $\mathfrak{X}$, calculate the mean gradient of L_1^{loss}, and use the mean gradient to update $\boldsymbol{\theta}$.
3: Pick a mini-batch in $\mathfrak{D}$, calculate the mean gradient of L_2^{loss}, and use the mean gradient to update $\boldsymbol{\theta}$.
4: repeat steps 2 and 3 until L_1^{loss} and L_2^{loss} are both small enough.
5: Return $\boldsymbol{\theta}$ and $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ for epoch with minimum L^{loss}

4.2. Fixed weight

The next training algorithm is the Fixed Weight algorithm, which is a widely used simple algorithm. It uses a fixed weighted sum of L_1^{loss} and L_2^{loss} as the overall loss function, which is given by, for $\theta \in [0, 1]$,

$$L^{\text{loss}}(\boldsymbol{\theta}) = (1-\theta)L_1^{\text{loss}}(\boldsymbol{\theta}) + \theta L_2^{\text{loss}}(\boldsymbol{\theta}) \tag{13}$$

Using a fixed weight works well if the ratio of the gradients of the two loss functions remains approximately the same, since θ can be adjusted so that the loss functions are equally influential on the overall loss. Additionally, combining the two loss functions resolves a drawback of Alternating Adam, which is that moving along the negative gradient of one loss function can potentially result in the increase of the other loss function. However, as will be seen later, the optimal θ value is a problem specific hyper-parameter that can be difficult to choose.

4.3. Trainable weight

Due to the difficulty of selecting an optimal θ for the Fixed Weight algorithm, the remaining algorithms explore ways to adjust θ during training based on the performance from prior epochs. If we assume that the ratio of the two loss functions remains approximately fixed, we would like our algorithms to make θ converge to the optimal θ. However, given that this assumption is rarely satisfied, we instead wish to update θ after each epoch so that it will do a better job of balancing the two loss functions for the next epoch.

Our first attempt to do this used the loss function (13), and adjusted θ in the direction of the positive gradient $\frac{\partial L^{\text{loss}}}{\partial \theta}$ after each epoch. However, this approach did not look stable, as θ would simply converge to either 0 or 1, since this would allow the neural network to solely minimize one loss function at the cost of the other. In particular, since $\tilde{u}(t_j, \boldsymbol{x}_j, \boldsymbol{\theta}) = 0$ satisfies the Fokker–Planck equation, the neural network had a tendency to move to $\theta = 0$ and produce a zero solution.

An alternative idea developed in McClenny and Braga-Neto [10] is to move θ following the negative gradient $\frac{\partial L^{\text{loss}}}{\partial \theta}$ for the loss function $L^{\text{loss}}(\boldsymbol{\theta}) = L_1^{\text{loss}}(\boldsymbol{\theta}) + \theta L_2^{\text{loss}}(\boldsymbol{\theta})$. Since θ can only increase this approach works like a constraint optimization problem: $L_1^{\text{loss}}(\boldsymbol{\theta})$ is optimized under the constraint that $L_2^{\text{loss}}(\boldsymbol{\theta})$ is already near its optimal value. Theoretically θ should converge to a certain saddle point. We refer to Liu and Wang [9] for the mathematical details.

4.4. Loss-based momentum weight

If L_1^{loss} and L_2^{loss} remain roughly proportional over all epochs, then the optimal θ for the Fixed Weight algorithm would be the loss ratio

$$\frac{L_1^{\text{loss}}}{L_1^{\text{loss}} + L_2^{\text{loss}}} \tag{14}$$

However, in practice L_1^{loss} and L_2^{loss} have significant fluctuations, at the very least because of the randomly sampled mini-batches. As a result, simply updating θ to (14) after each mini-batch or each epoch would seriously interrupt the training. In particular, we observed that L_1^{loss} drops rapidly in the beginning of training because an artificial neural network with random weights usually has very large second order derivatives. If θ is immediately updated according to the loss ratio, a rapid increase of the weight of L_1^{loss} may cause the training process to completely focus on optimizing L_1^{loss}. This negative feedback loop will eventually reach the trivial solution $u(t, \boldsymbol{x}) = 0$ of the Fokker–Planck equation. Instead, we must employ some method to slow the updates of θ far enough that this feedback loop is avoided. To do this we applied the idea of "momentum" to stabilize the change of θ during each update.

The Loss-Based Momentum Weight algorithm can be implemented in a few different ways. An initial training period of T epochs of Fixed Weight training are used to stabilize the results, as well as determine a better value of θ than what was used for those T epochs. After that, θ is set equal to a weighted average of itself and some function of the loss ratios from the previous epochs. At the $(T+1)$th epoch, the initial value r could be chosen as either the average of the loss ratios from the first T epochs, or the loss ratio at the Tth epoch. The weight α is a hyper-parameter that must be determined before training. See the Appendix for the discussion of suitable values of α. Algorithm 4 shows the pseudo code for the implementation in more detail. In practice we mainly use the Tth epoch loss ratio as the initial value of r because it has less training failures. Without further specification, Algorithm 4 takes approach (1) in line 8.

Alternatively, one can further stabilize the fluctuation of θ by taking a historic average of (14) throughout all training epochs, and update θ based on this average weight ratio. Since each additional loss ratio will have a progressively smaller impact on the average loss ratio, this will make the θ updates smaller over time and further prevent the negative feedback loop. The training may take longer but the value of θ is more likely to converge. We call this the "alternative implementation" of the loss-based momentum weight method when comparing algorithms. The pseudo code for this can be seen below in Algorithm 5.

Algorithm 4 Loss-based momentum weight method.

Initialize a neural network representation $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ with undetermined parameters $\boldsymbol{\theta}$.
Set $\theta = \theta_0$
Set $r = 0$
for epochs $= 0$ to epochs $= T - 1$ **do**
    Train using $L^{\text{loss}} = (1-\theta)L_1^{\text{loss}} + \theta L_2^{\text{loss}}$
    Record L_1^{loss} and L_2^{loss}
end for
Set $r =$ (1) the most recent $\frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$ or (2) the average of $\frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$ over epochs 0 to $T-1$
for epochs $= T$ to epochs $= N - 1$ **do**
    Set $\theta = \alpha\theta + (1-\alpha)r$
    Train using $L^{\text{loss}} = (1-\theta)L_1^{\text{loss}} + \theta L_2^{\text{loss}}$
    Set $r = \frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$
end for
Return $\boldsymbol{\theta}$ and $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ for epoch with minimum L^{loss}

Algorithm 5 Alternative implementation of the loss-based momentum weight.

Initialize a neural network representation $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ with undetermined parameters $\boldsymbol{\theta}$.
Set $\theta = \theta_0$
Set $r = 0$
for epochs $= 0$ to epochs $= T - 1$ **do**
    Train using $L^{\text{loss}} = (1-\theta)L_1^{\text{loss}} + \theta L_2^{\text{loss}}$
    Set $r = r + \frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$
end for
for epochs $= T$ to epochs $= N - 1$ **do**
    Set $\theta = \alpha\theta + (1-\alpha)\frac{r}{\text{epochs}}$
    Train using $L^{\text{loss}} = (1-\theta)L_1^{\text{loss}} + \theta L_2^{\text{loss}}$
    Set $r = r + \frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$
end for
Return $\boldsymbol{\theta}$ and $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ for epoch with minimum L^{loss}

4.5. Gradient-based momentum weight

The Gradient-Based Momentum Weight algorithm is motivated by the Loss-Based Momentum Weight implementations, but does not use loss ratios for the θ updates. Instead, it uses $\left|\left|\frac{\partial L_1^{\text{loss}}}{\partial \boldsymbol{\theta}}\right|\right|_2$ and $\left|\left|\frac{\partial L_2^{\text{loss}}}{\partial \boldsymbol{\theta}}\right|\right|_2$ in place of L_1^{loss} and L_2^{loss} for the Loss-Based Momentum Weight method. This completely circumvents the feedback problem with Loss-Based Momentum Weight training, as the norms of the loss gradients do not behave the same way as the loss values themselves. In addition, since the Monte Carlo data $v(t_i, \boldsymbol{y}_i)$ has some error, the value of L_2^{loss} with respect to the theoretical solution is nonzero. Hence a Loss-Based Momentum Weight algorithm may cause the optimization to focus too much on L_2^{loss} in the late phase of training and cause over fitting. This problem is also avoided by updating the weight according to the gradient.

Because it is very computationally expensive to compute the norms of the gradients for each mini batch, at the end of each epoch an additional batch with 500 randomly selected collocation points is run, and the gradients are computed based off of this batch. We find that averaging the ratio over the first five Fixed Weight training epochs does not make a meaningful difference. Hence throughout this paper, the initial value is chosen to be the ratio of gradients after the 5th epoch. The pseudo code for the Gradient-Based Momentum Weight algorithm can be seen below in Algorithm 6.

5. Numerical example with performance analysis

5.1. 1D Example

The first numerical example studies the solution to the simple stochastic differential equation

$$dX_t = (-X_t^3 + X_t)dt + dW_t \tag{15}$$

Algorithm 6 Gradient-based momentum weight.

Initialize a neural network representation $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ with undetermined parameters $\boldsymbol{\theta}$.
Set $\theta = \theta_0$
Set $a, b, r = 0$
for epochs $= 0$ to epochs $= T - 1$ **do**
    Train using $L^{\text{loss}} = (1-\theta)L_1^{\text{loss}} + \theta L_2^{\text{loss}}$
    Compute $a = \left|\left|\frac{\partial L_1^{\text{loss}}}{\partial \boldsymbol{\theta}}\right|\right|_2$ and $b = \left|\left|\frac{\partial L_2^{\text{loss}}}{\partial \boldsymbol{\theta}}\right|\right|_2$ based on an additional batch with 500 collocation points, and set $r = \frac{a}{a+b}$
end for
for epochs $= T$ to epochs $= N - 1$ **do**
    Set $\theta = \alpha\theta + (1-\alpha)r$
    Train using $L^{\text{loss}} = (1-\theta)L_1^{\text{loss}} + \theta L_2^{\text{loss}}$
    Compute $a = \left|\left|\frac{\partial L_1^{\text{loss}}}{\partial \boldsymbol{\theta}}\right|\right|_2$ and $b = \left|\left|\frac{\partial L_2^{\text{loss}}}{\partial \boldsymbol{\theta}}\right|\right|_2$ based on an additional batch with 500 collocation points, and set $r = \frac{a}{a+b}$
end for
Return $\boldsymbol{\theta}$ and $\tilde{u}(t, \boldsymbol{x}, \boldsymbol{\theta})$ for epoch with minimum L^{loss}

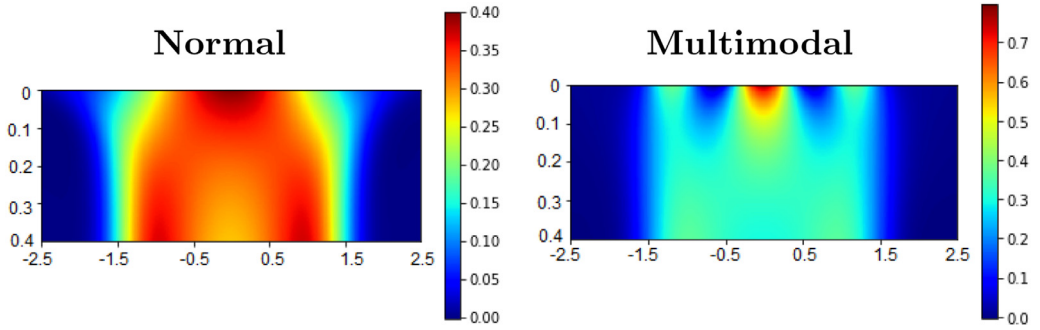

Fig. 1. 1D median neural network solutions from 31 samples for the normal and multimodal initial distribution using the Gradient-Based Momentum Weight method.

on the numerical domain $[0, 0.4] \times [-2.5, 2.5]$, which is discretized into a 200 by 500 grid. Two different initial distributions are considered. The first is the standard normal distribution with probability density function

$$u_0(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}, \tag{16}$$

and the second one is a multimodal distribution given by

$$u_0(x) = \frac{1}{Z}(1 + \cos(5x))e^{-\frac{1}{2}x^2}, \tag{17}$$

where $Z = (1 + \exp(-25/2))\sqrt{2\pi}$ is the normalizer that makes u_0 a probability density function. Generally speaking, it is more difficult to train a neural network to accurately fit a multimodal distribution.

The details of algorithm implementation and parameter selection of Eq. (15) are discussed in the Appendix. In summary, the size of collocation points $\mathfrak{X}$ and training points $\mathfrak{Y}$ are 1500 and 20,000 respectively. The hyper-parameters selected were $\theta = 0.975$ for Fixed Weight, $\theta_0 = 0$ for Trainable Weight, $\theta_0 = 0.99$ for Loss-Based Momentum Weight with $\alpha = 0.4$ for the regular implementation and $\alpha = 0.6$ for the alternative implementation, and $\theta_0 = 0.99$ with $\alpha = 0.4$ for Gradient-Based Momentum Weight, all with $T = 5$. Each of these training algorithm configurations were trained 31 times for both initial distributions, the normal distribution and the multimodal distribution, so as to demonstrate the average performance and the variation in performance for simple and complex initial distributions. The training time is around 10–11 min. The neural network solutions from the Gradient-Based Momentum Weight method are shown in Fig. 1. On the left is the median solution for the normal initial distribution, and on the right is the median solution for the multimodal initial distribution. Due to the relatively low L^2 error in comparison to the 2D SDE discussed later, the solutions look very similar across training methods, so only these solutions are provided as an example.

5.1.1. Analysis of 1D performance results

In Fig. 2, we demonstrate the distribution of error from all six training algorithms. Here the ground truth solution comes from the Crank–Nicolson PDE solver on a mesh that is further refined by 4 times. It is well known that the Crank–Nicolson scheme is a second order scheme [11]. Its theoretical magnitude of error on the refined mesh is around 10^{-5}.

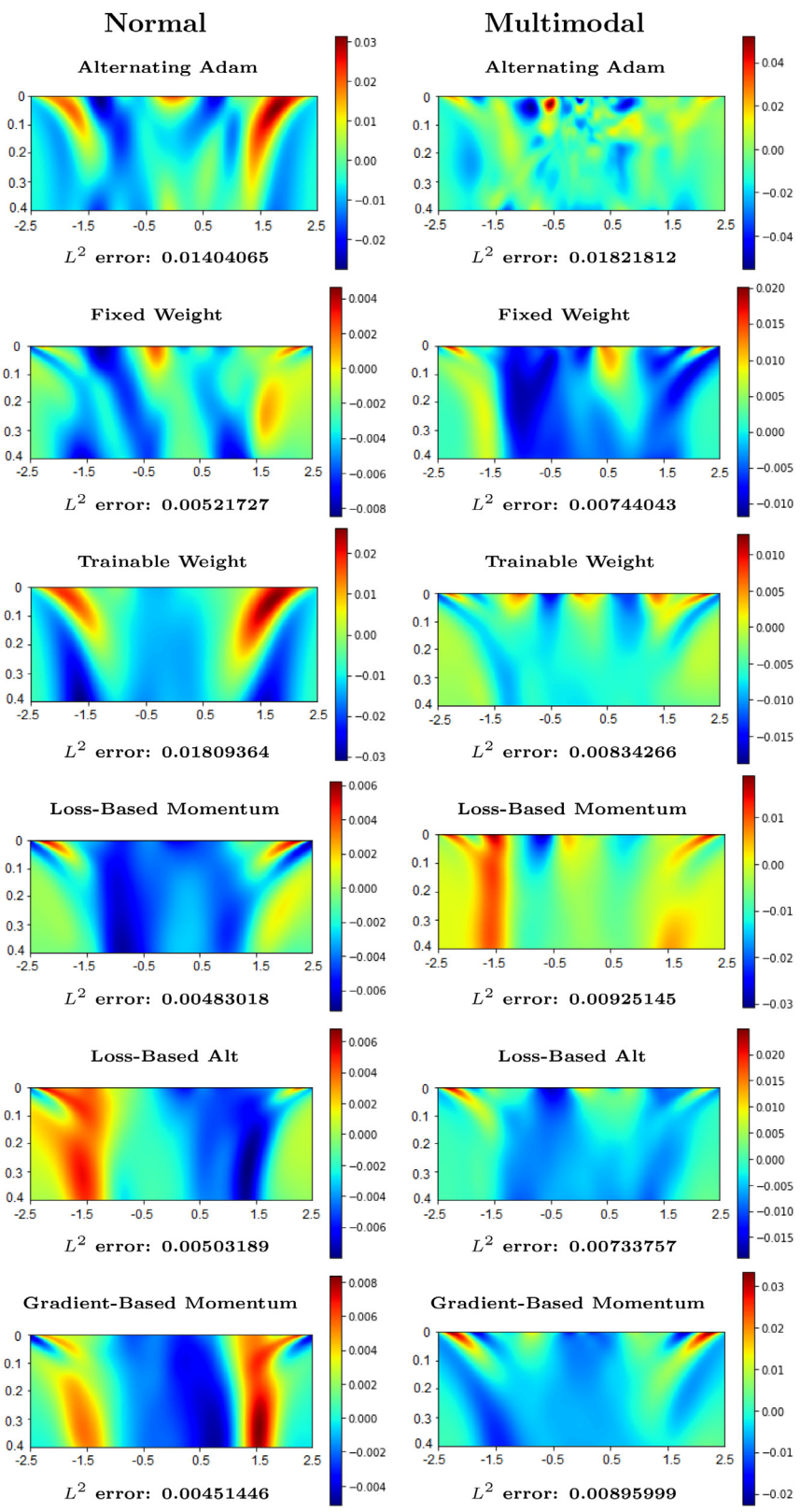

Fig. 2. 1D SDE median L^2 error heat maps for each training algorithm from 31 samples. Left column: normal initial distribution. Right column: multimodal initial distribution.

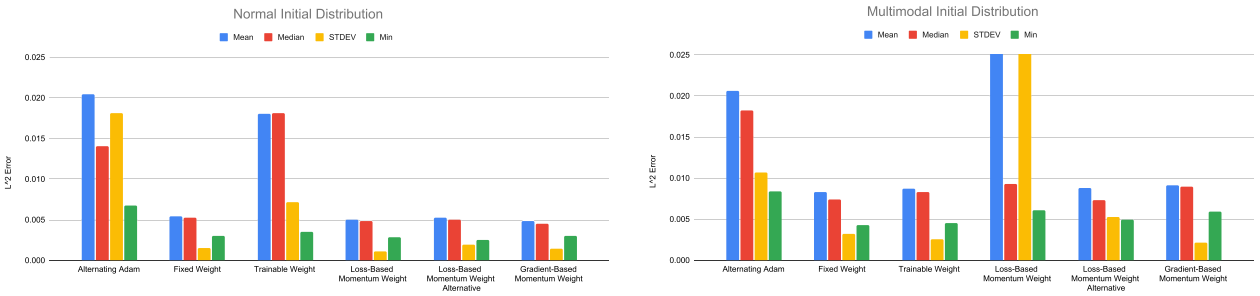

Fig. 3. L^2 error for the normal and multimodal initial distributions. Because of the 6 failed trainings for Loss-Based Momentum Weight, the mean and standard deviation of their results extend above the graph to 0.04531 and 0.08335 respectively.

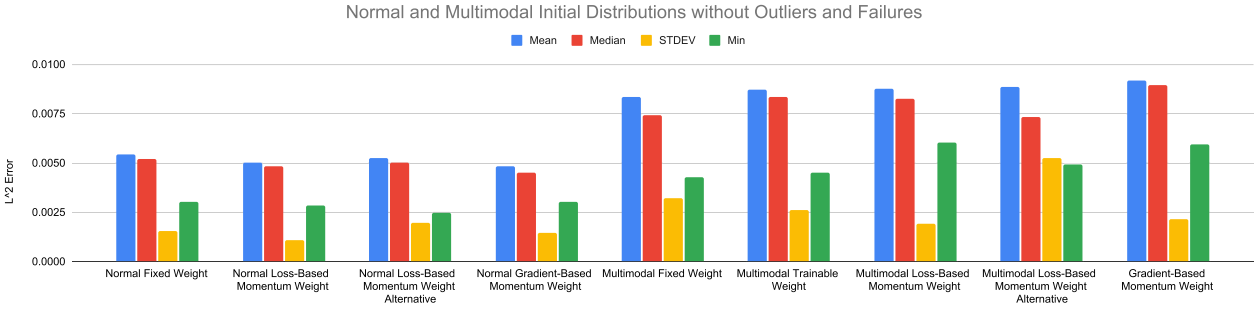

Fig. 4. L^2 error results for both initial distributions without outliers or training failures.

The L^2 error of the six neural network train algorithms with the two initial distributions is demonstrated in Fig. 3. It is easy to see that the performance for the simpler $\mathcal{N}(0,1)$ initial distribution is better than for the multimodal initial distribution in Eq. (17). This is expected because training a neural network to fit a multimodal function is more difficult. Additionally, this confirms our expectation that hyper-parameters selected for a more complicated multimodal initial distribution can be used directly on simpler initial distributions. As seen in Fig. 2, the error for the multimodal initial distribution is mostly concentrated at the initial distribution, most notably near $x = -2.34$ and $x = 2.34$ where the smallest peaks are typically only about half the height they should be. While there is also error concentration there for Loss-Based Momentum Weight method for the normal initial distribution, this is far less of a problem given that the distribution should be close to zero there, and the error itself is much lower. The more noticeable problem for the Loss-based Momentum Weight method is the maximum possible errors for the multimodal initial distribution. This is because a portion of trainings (6 out of 31) made θ converge towards zero. Given how visually apparent these errors are, it is worth considering the performance for these methods once we discard those training results, as will be shown in Fig. 4.

Alternating Adam is a clear outlier for both initial distributions as was to be expected based on previous results. The Trainable Weight algorithm does not work well either. Interestingly, it gives higher error for the standard normal initial distribution. This likely means that the optimal Fixed Weight θ value is actually lower for the normal initial distribution because the L_1 error is proportionately smaller, so revisiting Fixed Weight for the normal distribution could possibly produce better results.

For better comparison, in Fig. 4 we only demonstrate the error statistics of training algorithms with good results. This means that Alternative Adam for both initial distribution and Trainable Weight for the standard normal distribution are removed, as well as the 6 failed training results in the Loss-Based Trainable Weight method.

When taking both sets of results into account, combine with the hyper-parameter selection and implementation details discussed in the Appendix, we can see that the Alternating Adam method is the easiest to implement but has much higher error than for the stationary Fokker–Planck equation reported in Zhai et al. [3]. The Fixed Weight method works the best but to the best of our knowledge the weight has to be manually selected for each problem, because it is very difficult to estimate the scales of L_1^{loss} and L_2^{loss} without training the neural network. This makes the Fixed Weight method less practical. The Trainable Weight method is supported by some literature theoretically but has no performance advantage in our testing, especially considering the relatively high error for the case of the normal initial distribution. The momentum algorithms are very comparable to, or better than, the Fixed Weight method. The Loss-Based Momentum method converges quickly but has some stability issues, meaning one must manually check whether θ converges to either 0 or 1. The alternative implementation has better stability, but the hyper-parameter selection process shows that it could have a slower convergence to the optimal θ. (See Fig. 18 in Appendix.) The Gradient-Based Momentum Weight is a better balance of stability, easier implementation, and performance.

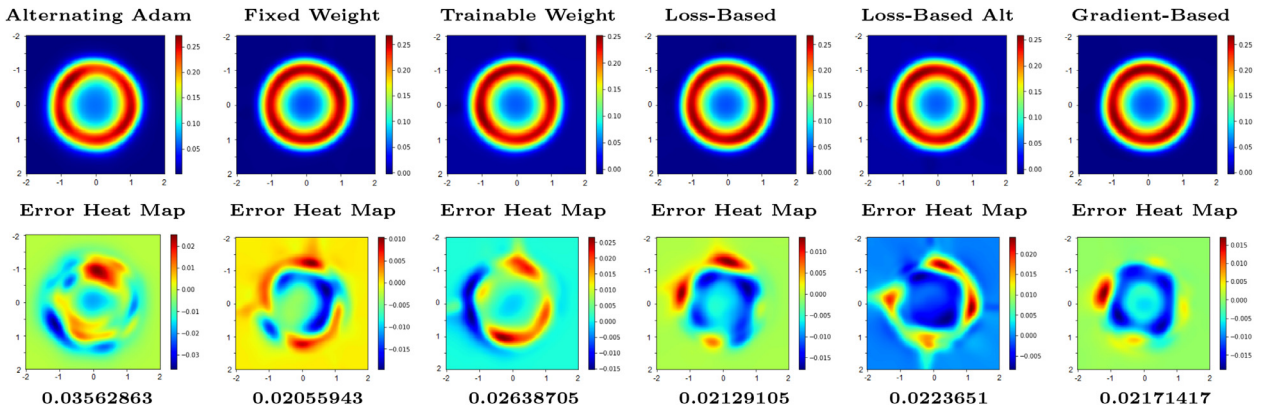

Fig. 5. 2D SDE median L^2 error results at $t = 0.2$ for each training algorithm from 55 samples across training point counts. First row: neural network output at $t = 0.2$. Second row: error heat maps, with respective L^2 error listed below.

5.2. 2D example

5.2.1. 2D SDE overview and performance results

Next, we apply our training methods to the same "ring example" studied in Zhai et al. [3]. The SDE is given by

$$\begin{cases} dX_t = (-4X_t(X_t^2 + Y_t^2 - 1) + Y_t)dt + dW_t^x \\ dY_t = (-4Y_t(X_t^2 + Y_t^2 - 1) - X_t)dt + dW_t^y \end{cases} \tag{18}$$

where W_t^x and W_t^y are independent one dimensional Wiener processes. It is easy to check that Eq. (18) admits an explicit stationary distribution $\exp(-2(x^2 + y^2 - 1)^2)/K$ with $K = \pi \int_{-1}^{\infty} \exp(-2t^2)dt$, which concentrates at the unit circle. For this example the initial distribution used is the multivariate normal distribution $\exp(-0.5(x^2 + y^2))/(2\pi)$. Because the vector field symmetrically pushes the density to the unit circle, the solution converges to the stationary distribution quickly. We selected this equation as a numerical example because if a neural network can accurately approximate a fast-changing ring-shaped solution, we expect it can also approximate Fokker–Planck solutions in simpler shapes.

The numerical domain used was $[0, 0.2] \times [-2, 2] \times [-2, 2]$ discretized into a 200 by 200 by 200 grid. The same neural network architecture as the 1D example was used, although the Fixed Weight training algorithm was slightly modified to have $\theta = 0.984$, which was selected based on the loss ratios from some preliminary trainings. The training data used here is also the same as the 1D case, however the average run time is around 17–18 min instead of the 10–11 min from before. Each training algorithm configuration was trained 11 times at 5 different collocation point counts, namely 1083, 1875, 3468, 7500, and 13,467, to see whether more points would be required for this higher dimensional SDE. The same ratios between N^X, N^Y, and I^Y were maintained. Additionally, $\mathfrak{Y}$ was again 20,000 points sampled uniformly. The training results are demonstrated in Fig. 5. The first row shows the training result at $t = 0.2$ for the six different neural network training algorithms. The second row is the difference between neural network solution and the solution from Crank–Nicolson scheme. For each algorithm, the solution demonstrated in Fig. 5 is the solution with median error among the 55 training results combined across the different training point counts, as the performance difference between them is negligible.

Comparisons here are for the slice of the distribution when $t = 0.2$, since by then the distribution is close to stationary and can be compared to the results from Zhai et al. [3]. Additionally, performance at $t = 0.2$ is a decent indicator of overall performance, and slices at multiple times will be addressed in the next subsection.

5.2.2. Analysis of 2D performance results

Fig. 6 shows the median (top left) and mean (top right) L^2 error for each algorithm and training point count. It is easy to see that Trainable Weight is a clear outlier here and therefore should not be used. The Loss-Based Momentum Weight method also has 4 failed trainings (out of 55). After removing the Trainable Weight method and the failed training results from the Loss-based Momentum Weight method, a more refined result is demonstrated at the bottom of Fig. 6. It is easy to see that Alternating Adam has clearly higher error, while the rest of the algorithms have rather similar performance.

Fig. 6 also demonstrates that there is no apparent difference between training with more than 1083 points, which is close to the maximum of 1024 training points used for this 2D ring example in Zhai et al. [3]. Because of this, the results are combined across training point counts in Fig. 7 to increase the sample size. On the left we have the L^2 error values, and on the right we have those values normalized by the benchmark Alternating Adam method. Similarly to the 1D case, Fixed Weight has the lowest mean, median and standard deviation, but requires manual selection of θ. The Gradient-Based Momentum Weight is better than all implementations of the Loss-Based Momentum Weight method in almost all categories. Considering all factors across the 1D and 2D cases, the Gradient-Based Momentum Weight has the best performance and will be used in our future studies, including the next section.

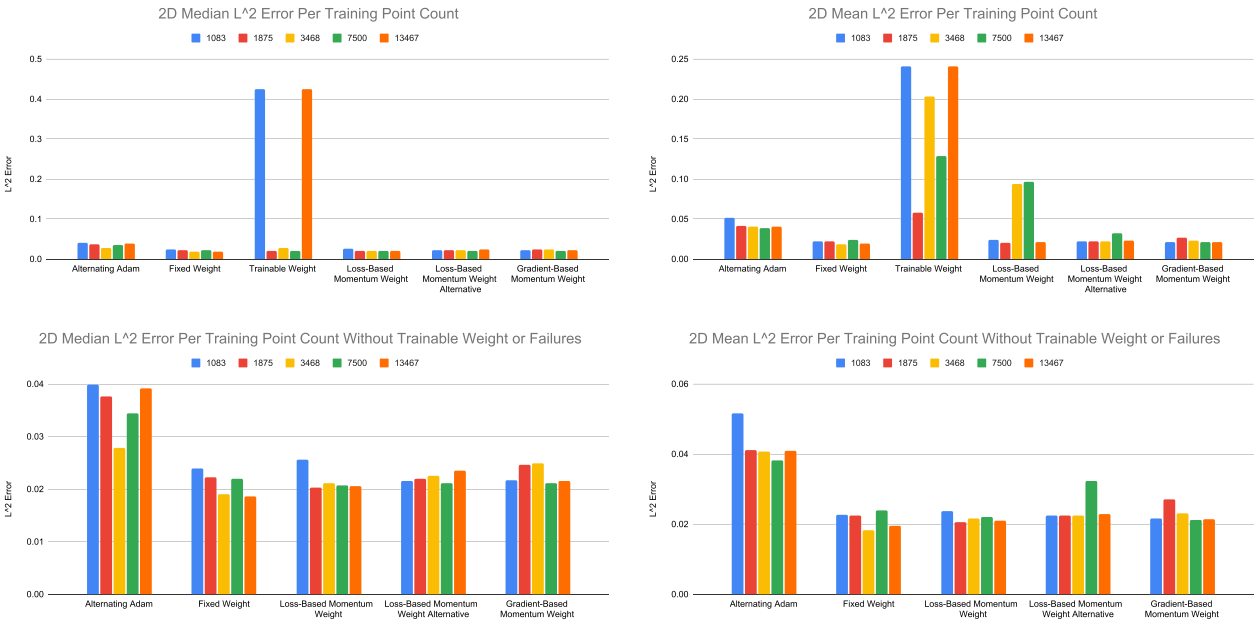

Fig. 6. Median and Mean L^2 error for 2D example for all training algorithms using 1083, 1875, 3468, 7500 and 13,467 training points. Top: including Trainable Weight method and the Loss-Based Momentum Weight training failures. Bottom: not including them.

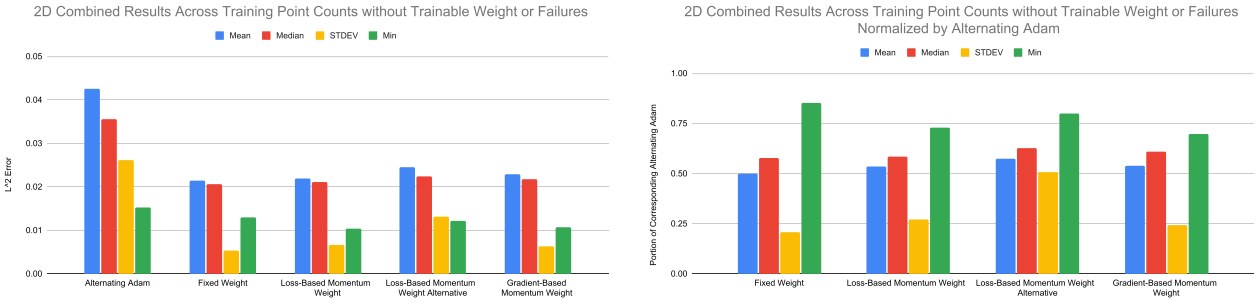

Fig. 7. Median, Mean, Standard Deviation and Minimum L^2 error for 2D example for all training algorithms except Trainable Weight, using combined training point counts without Loss-Based Momentum Weight training failures. Left: pure values. Right: normalized by Alternating Adam.

6. Comparison with anchor sampling method

The idea behind Anchor Sampling is that theoretically the solution to the Fokker–Planck equation is uniquely determined by the initial distribution, if it is given. Therefore, we should not treat the time variable simply as "yet another dimension". Instead, we find that it is beneficial to concentrate collocation points at the initial time and the terminal time. To see this, in the 2D ring example, we let $\mathfrak{Y}$ consist of just 40,000 points from the initial distribution and $\mathfrak{X}$ the standard set of points uniformly sampled throughout the entire numerical domain. The result is shown in Fig. 8, in which the error heat maps show the Crank–Nicolson solution minus the neural network solution. We can see that although the L^2 error increases with the time, the general shape of the distribution is largely preserved. Fig. 9 shows equivalent results for the 1D SDE using 3000 points from the initial distribution as our $\mathfrak{Y}$ for both the multimodal and normal initial distributions. Like for the 2D case, the general shape is preserved, but the scale of the solution drifts upward as time progresses. However, this drift is much slower for the normal initial distribution, which is easier for the neural network to represent and produces much faster convergence to the stationary distribution. These results indicate that producing an accurate solution only requires correcting the scales away from the initial distribution, who's rate of drift depends on the initial distribution. This motivates us to add a relatively small amount of collocation point at $t = 0.2$ that serves as an "anchor" for the scale.

6.1. 2D ring SDE anchor sampling numerical results

In the next numerical result, the set of collocation points $\mathfrak{Y}$ is 40,000 collocation points from the initial distribution and 1156 points at $t = 0.2$ (see Fig. 20 in the Appendix.) We tested two different ways of sampling collocation points at $t = 0.2$: one uses the standard sampling method described in Algorithm 1, the other selects points from a sparser grid

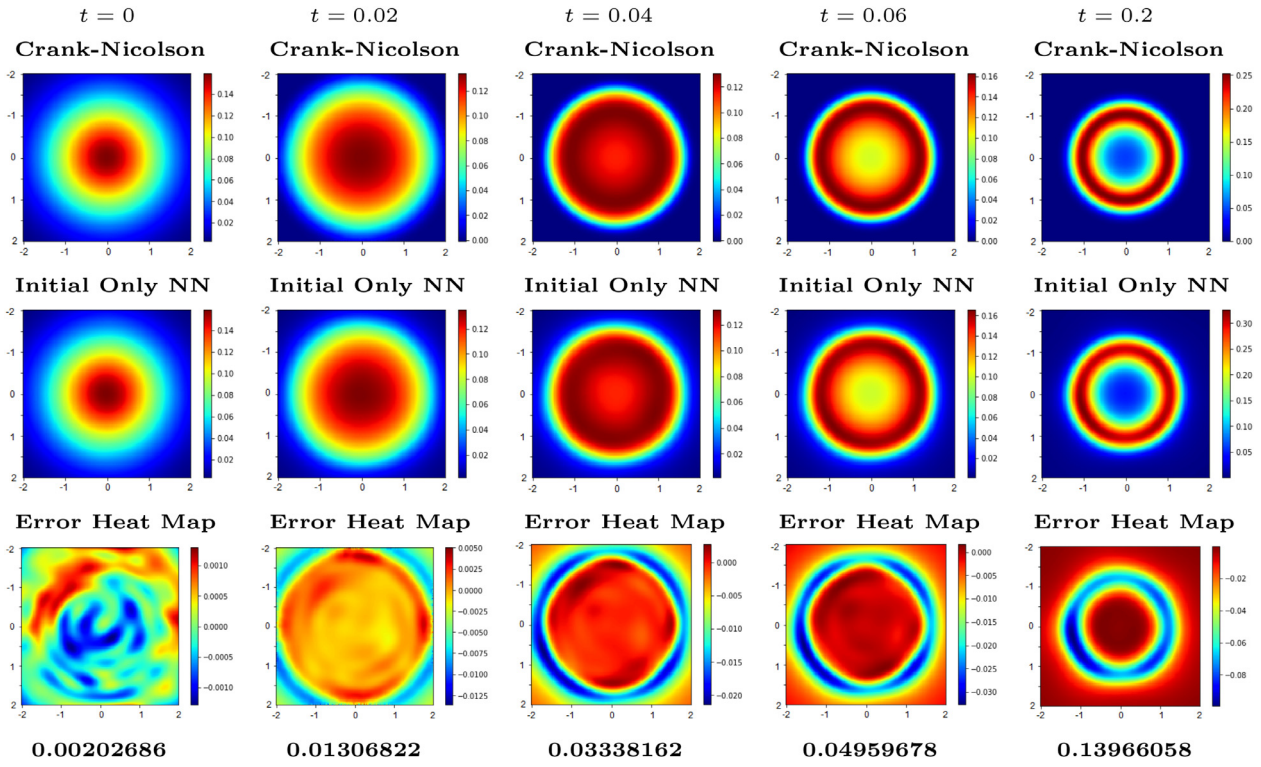

Fig. 8. Crank–Nicolson solution, neural network solution and error heat maps at $t = 0$, 0.02, 0.04, 0.06, and 0.2 after training the neural network with 40,000 points from the initial distribution as $\mathfrak{Y}$. L^2 error is listed below the respective heat map.

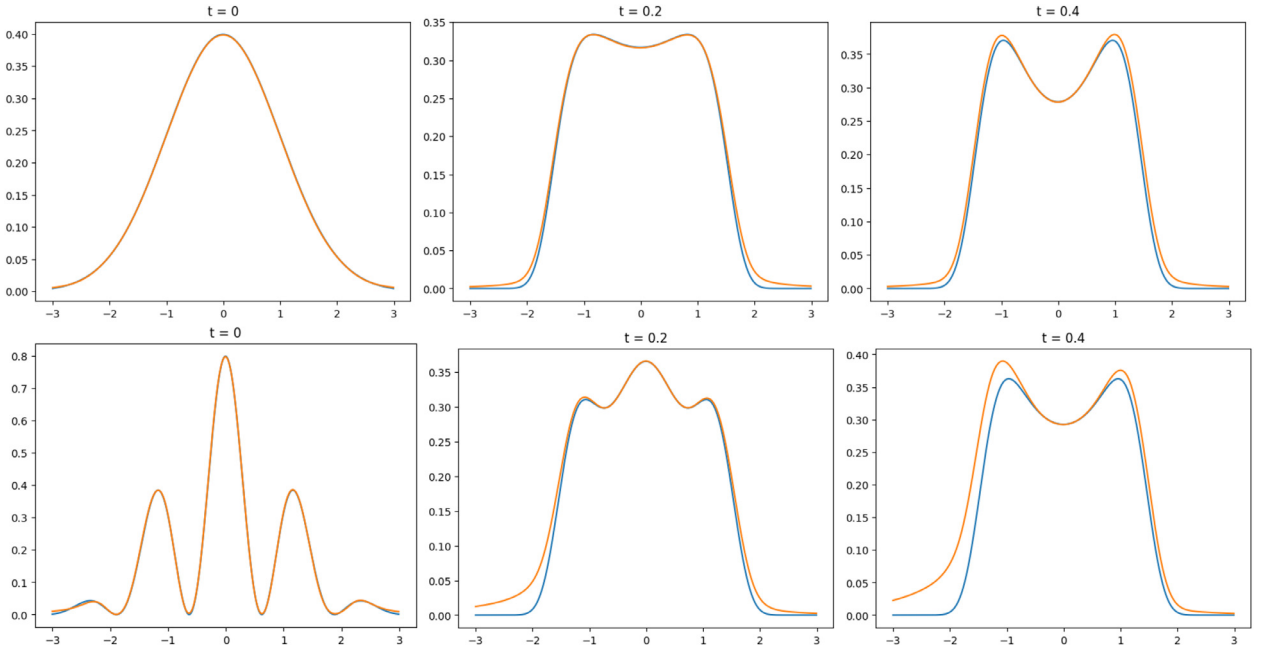

Fig. 9. Crank–Nicolson solution (blue) and neural network solution (orange) at $t = 0$, $t = 0.2$ and $t = 0.4$ for the 1D SDE with normal initial distribution (top) and multimodal initial distribution (bottom) after training the neural network with 3000 points from the initial distribution as $\mathfrak{Y}$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

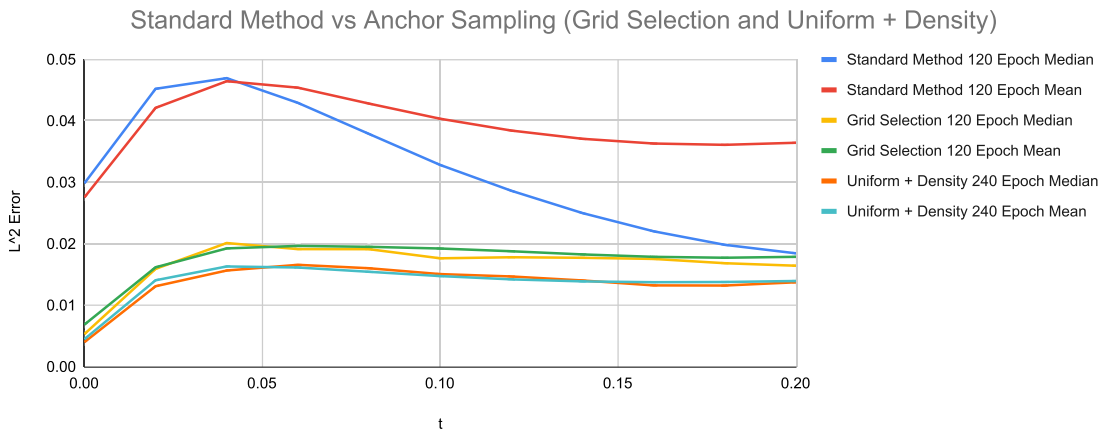

Fig. 10. Median and Mean L^2 error comparison between Standard method for 120 epochs, Anchor sampling with Grid Selection for 120 epochs, and Anchor sampling with 0.5 Uniform Sampling 0.5 Proportional to Density Sampling for 240 epochs. 11 sample trainings used for each configuration.

laid over the grid at $t = 0.2$, in this case 33 by 33 squares. The numerical result, error heat maps, and median L^2 error at $t = 0, 0.02, 0.04, 0.06, 0.2$ are demonstrated in Fig. 11, and the median and mean L^2 error across all time slices is shown in Fig. 10. In Fig. 11, the two aforementioned sampling methods are called "U+D" and "Grid" respectively, because Algorithm 1 samples, in this case, half the collocation points uniformly and the other half from the probability density. The Standard Method and Grid based Anchor Sampling method were trained for 120 epochs, whereas the U+D based Anchor Sampling method was trained for 240 epochs. This is because when the Grid based and U+D based Anchor Sampling methods were both trained at 120 and 240 epochs, one slightly outperformed the other both times. We can see that there are diminished returns from doubling the training epochs, and that even at 120 epochs the Anchor Sampling method reduced the median L^2 error by about one half away from the initial distribution and the terminal time. Due to the high concentration of points at the initial distribution for the Anchor Sampling method, the L^2 error there is considerably lower. Additionally, Fig. 10 shows that while the median L^2 error for the Standard Method approaches that of the Anchor Sampling method near $t = 0.2$, the mean L^2 error does not. This confirms the advantage of the Anchor Sampling method. It is beneficial to use most collocation points to approximate the initial distribution, and a relative small number collocation points at the terminal time to "anchor" the solution.

6.2. 1D multimodal SDE anchor sampling numerical results

For the 1D SDE studied previously, only the multimodal initial distribution was tested with Anchor Sampling, as the solution from the normal initial distribution is already satisfactory. In our computation, we sampled 2500 points from the initial distribution and 500 points at the terminal time ($t = 0.4$). Since there were only 500 grid points at the terminal time, all of them were used and therefore no true sampling was required. The neural network was trained for 240 epochs. The result is demonstrated in Fig. 12. The ground truth is still obtained from Crank–Nicolson scheme with a refined mesh. However, inspecting the range of error values, one can see that the Anchor sampling is clearly outperforming the Standard method, especially when it comes to the initial distribution. The peak of the local maximums of the initial probability density function near -2.34 and 2.34 is about 0.0432. Even in the best case scenario, the Standard method is missing about half the density there, and often misses it entirely, leaving the density at zero. In comparison, Anchor sampling is able to almost entirely eliminate that error.

6.3. Longer time frames

So far we have only considered Anchor Sampling for relatively short time frames, 0.2 and 0.4 for the 2D and 1D SDEs respectively. To extend this technique to longer time frames, multiple slices, or Anchors, are required. This is because there is a short effective range of influence before and after each Anchor where the scale of the distributions are correct. Additionally, the shape of the distribution starts to deform in addition to the scale drifting for sufficiently large distances from any training points. Because of this, if just the initial distribution and terminal Anchor are used for a long time frame, error will be low near the start and end but rise considerably in between, away from the influence of either set of points.

To demonstrate the use of multiple Anchors, the 2D ring example was trained using Anchors at $t = 0.2, 0.4, 0.6, 0.8,$ and 1.0, along with the entire initial distribution. The L^2 error results from this can be seen below in Fig. 13. We can see that during the time interval $[0, 0.2]$, having multiple "anchors" does not change the result very much. In addition, when multiple anchors are used, the L^2 error is largely consistent throughout the entire time domain.

One additional question is that how the positions of "anchors" should be selected. We believe this should be related to the spectral gap of the infinitesimal generator of Eq. (1). Larger spectral gap means faster convergence to the steady-state.

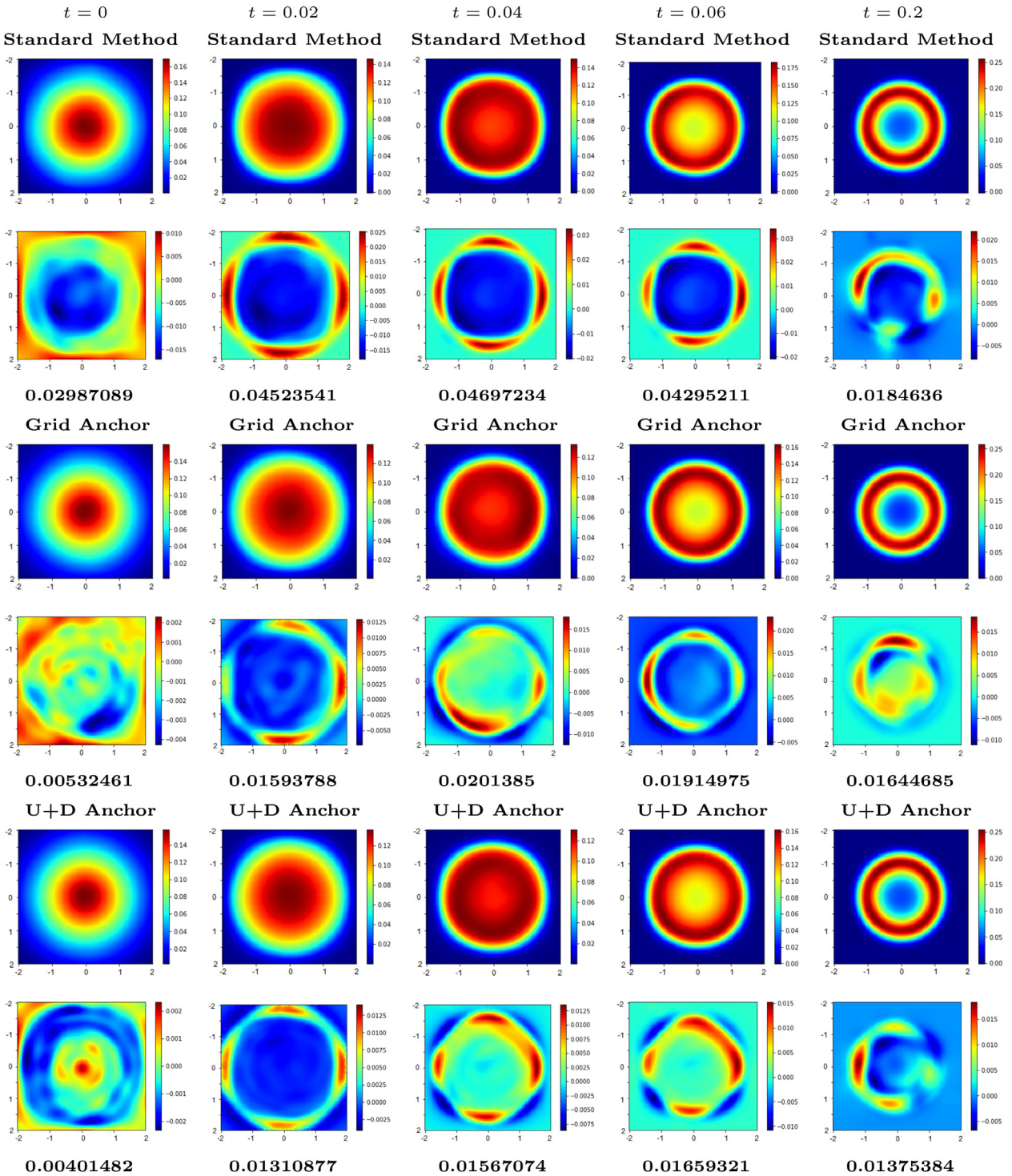

Fig. 11. Neural network solutions and error heat maps of median results from 11 sample trainings per configuration at $t = 0, 0.02, 0.04, 0.06, 0.2$ for the Standard method, Grid Selection Anchor sampling for 120 epochs (Grid Anchor), and Uniform + Density Anchor Sampling for 240 epochs (U+D Anchor). L^2 error is listed below the respective heat map.

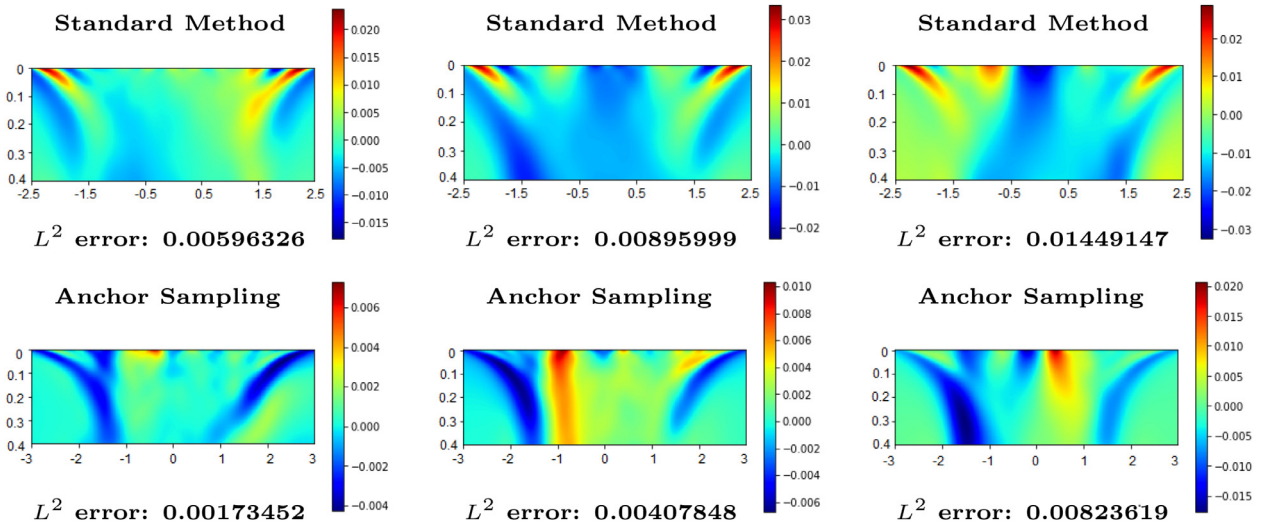

Fig. 12. Minimum, Median and Maximum L^2 error results from 31 trainings of the 1D Multimodal SDE using the Standard Method and Anchor Sampling for 240 epochs.

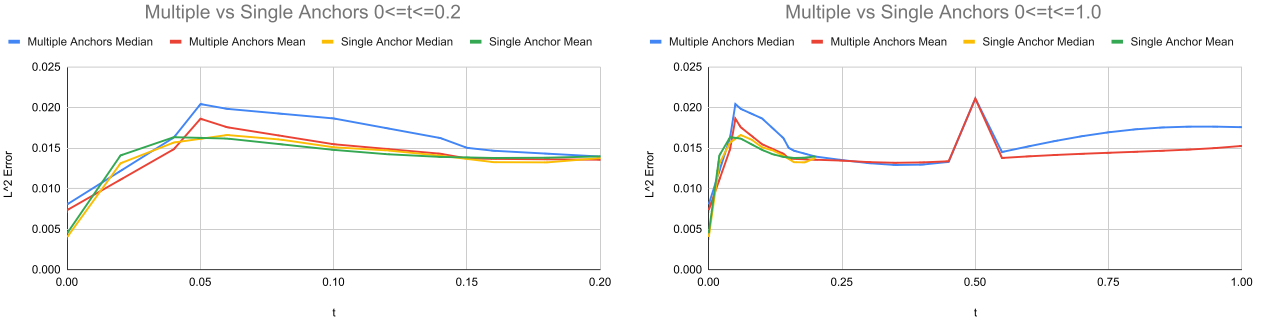

Fig. 13. Comparison of mean and median L^2 error after 240 epochs of single vs. multiple Anchors for the 2D ring SDE.

The operator in the first part of the loss is also further away from a singular operator. This corresponds to less "drift" described at the beginning of this section. Hence the neural network solver can tolerate longer gaps between "anchors", i.e., times at which collocation points in $\mathfrak{Y}$ concentrate. However, a practical relation between the spectral gap of the infinitesimal generator and the gap between "anchors" requires deeper studies. We will address this issue in our future work.

7. Conclusion

In this paper, we examined the neural network training for the neural network Fokker–Planck solver in full detail. The main challenge here is the presence of multiple loss functions at different scales. We believe this challenge can also appear when using PINN to solve equations with noisy experimental data. One very interesting finding is that the optimization method for training the neural network seems to be problem-dependent. The idea of "Alternating Adam" that worked very well for the stationary Fokker–Planck equation does not have satisfactory performance for the time-dependent Fokker–Planck equation. Instead, we tested a few different ways to balance multiple loss functions. Our analysis shows that the most robust approach is to let the relative weight of a loss function depend on the norm of the gradient of this loss function because each update is based on the gradient rather than the value of each loss function. In addition, one needs a "momentum" term to gradually change the weight of the loss functions to avoid stability issues.

Our study motivates a challenging question: what does the loss landscape look like? There are some known studies about the loss landscape of a few commonly used loss functions [12]. But to the best of our knowledge, the loss landscape of a loss function that involves the norm of a differential operator of the neural network has not been investigated. If the neural network can well approximate PDE solutions in H^1 norm as suggested by Chen et al. [13], Weinan and Wojtowytsch [14], the "bottom" of a loss function given by the norm of a differential operator should be like a very high-dimensional valley because any boundary condition (resp. initial and boundary condition) can uniquely decide the solution of an elliptic (resp. parabolic) PDE. The neural network training process aims to find a local minimum in this "valley" that also matches the initial distribution, the boundary condition, or the Monte Carlo approximation in our paper. However, the gradient of

the loss function orthogonal to those "valleys" may have qualitative differences between a loss function given by an elliptic operator and a loss function given by a parabolic operator. This is because the second-order derivative of the neural network approximation is likely much more sensitive to a random change of connection weights than the first-order derivatives. The lack of second-order derivatives in some directions makes the "valley" less steep. We believe this could be the root cause that makes the Alternating Adam method less effective for time-dependent Fokker–Planck equations. We also find that the Alternating Adam method fails frequently when a stationary Fokker–Planck equation has a degenerate elliptic term. This further supports our conjecture.

Currently, it appears that the best training method for a PINN-like problem is very problem-specific. Alternating Adam works best for the stationary Fokker–Planck equation. The Gradient-Based Momentum Weight method works well for the time-dependent Fokker–Planck equation. Many PINNs are trained by second-order methods such as BFGS [15]. The Trainable Weight method works well for some applications of PINN [8,10]. However, there is no theory that supports the selection of training methods. After writing this paper, we believe the choice of a suitable training method should be dependent on properties of the loss surface. We will address this in our future work.

Data availability statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

CRediT authorship contribution statement

Yao Li: Conceptualization, Methodology, Writing – review & editing. **Caleb Meredith:** Data curation, Formal analysis, Visualization, Writing – original draft, Writing – review & editing.

Acknowledgments

YL is supported by NSF grants DMS-1813246 and DMS-2108628. CM is supported by the REU part of NSF DMS-1813246.

We thank Prof. George Karniadakis and Dr. Shengze Cai for helpful discussions about neural network training, particularly the use of trainable weight.

Appendix A. Algorithm implementation, hyper-parameter and training point selection, and performance optimization

The majority of this Appendix covers the implementation details, hyper-parameter selection, and training data sampling based on the 1D SDE given by Eq. (15) with the multimodal initial distribution. For each training, the error is calculated in the L^2 sense in comparison to the ground truth, which is a numerical solution obtained using a Crank–Nicholson solver. The Standard Method is used for selecting training points. This Appendix also covers training point count selection for the Anchor Sampling method and investigates the effects of changing the numerical domain, using additional compute, and adding an additional anchor.

In this example we used 1500 collocation points with a breakdown of $N^X = 500$, $N^Y = 1000$, $I^Y = 500$. Additionally, $\mathfrak{Y}$ is composed of 20,000 points uniformly sampled from the entire domain. The Monte Carlo sampling part runs 10^7 samples to approximate the probability density at non-initial collocation points.

A1. Alternating adam

While Alternating Adam doesn't have any hyper-parameters to select, a slight modification must be made based on the performance for this SDE. Unlike for the stationary case, the loss of a given epoch is not a good indicator of performance. The lowest loss is usually seen in the first few epochs. Because of this, the last epoch of training rather that the one with the minimum loss value is selected as the solution of the neural network solver.

A2. Fixed weight

The only hyper-parameter that must be selected here is the weight term θ, which serves as a benchmark when comparing the results of non Fixed Weight training methods. Generally L_1^{loss} is much larger than L_2^{loss}, so θ must be close to 1. In the first numerical example we tested the performance of the neural network solver for 10 different values of θ in the interval $[0.95, 0.995]$. This interval was selected after a few preliminary tests, which showed that the training result outside of this interval is less satisfactory in general. After training with each θ value in this interval 5 times, the optimal choice was found to be $\theta = 0.975$.

This weight is further confirmed by evaluating 45 weight ratios $\frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$ using $\theta = 0.975$. One can see that the L^2 error is the lowest when the final weight ratio is between 0.96 and 0.97, and from the results from Fig. 14 we observed that the final loss ratio averaged around 0.012 less than the θ value used (Fig. 15).

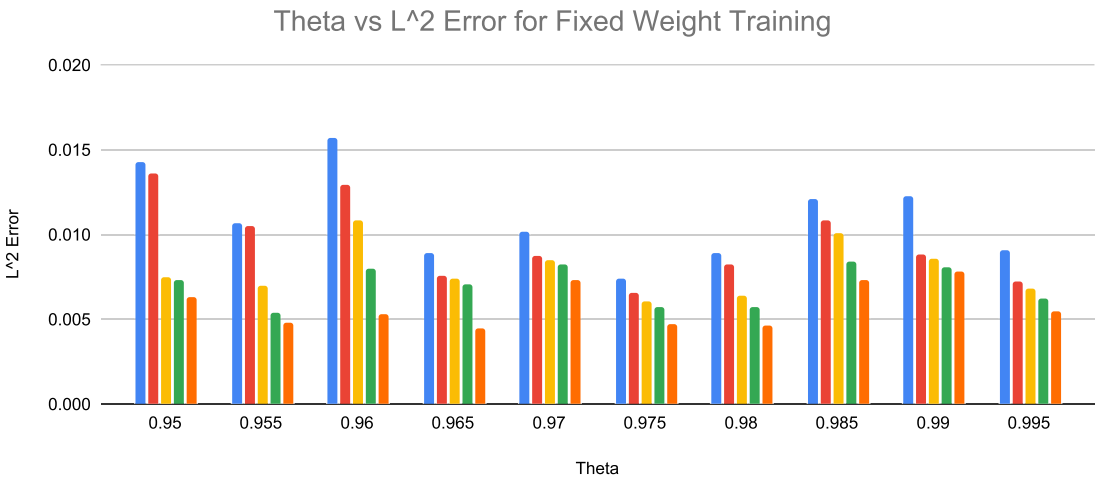

Fig. 14. L^2 error for different θ values using Fixed Weight training.

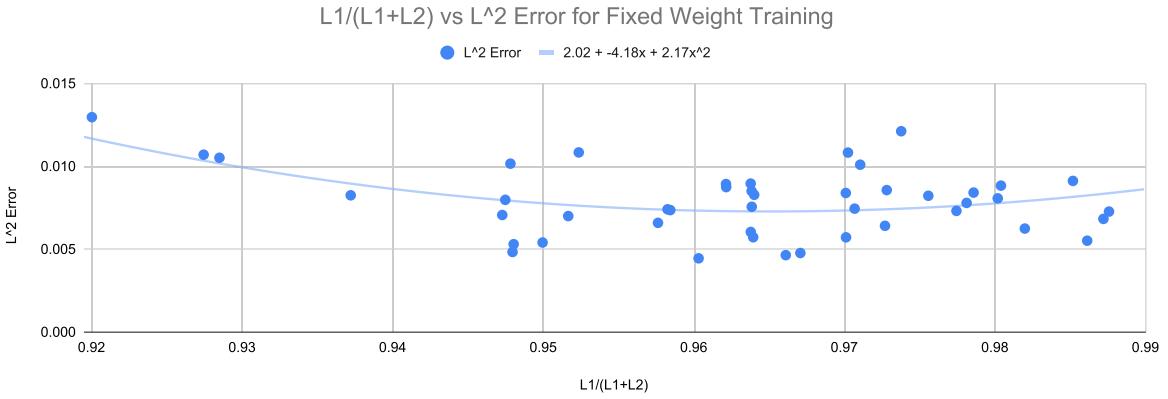

Fig. 15. Comparison of final $\frac{L_1^{\text{loss}}}{L_1^{\text{loss}}+L_2^{\text{loss}}}$ and L^2 error for all 45 Fixed Weight training results using $\theta = 0.975$.

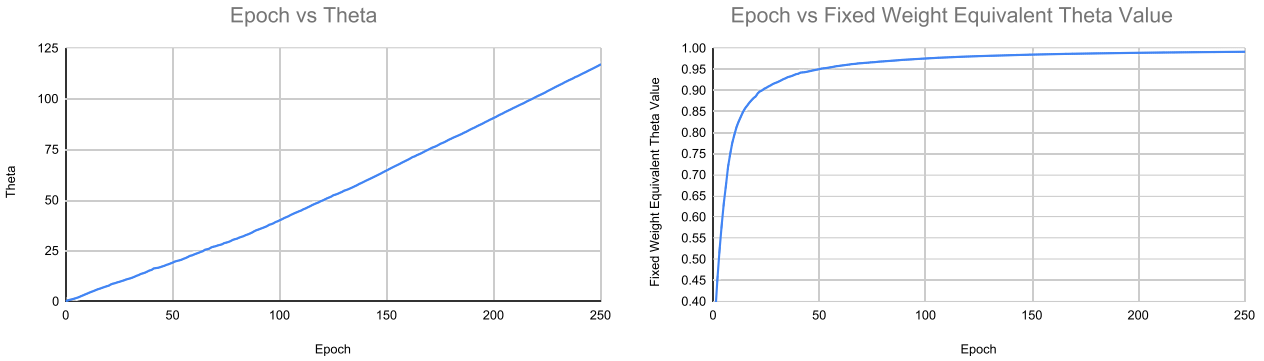

Fig. 16. Left: Theta per Epoch for Trainable Weight using $\theta_0 = 0$. Right: Equivalent Theta for Fixed Weight per Epoch.

A3. Trainable weight

The Trainable Weight algorithm only requires selecting an initial value for θ_0. The simplest choice is $\theta_0 = 0$. The training results from this initial value can be seen below in Fig. 16. The left panel shows the value of θ versus the epoch, and the right panel shows the equivalent θ values if translated to the Fixed Weight algorithm. As θ increases roughly linearly, the Fixed Weight equivalent θ asymptotically approaches 1. Recall that previously we found that the optimal Fixed Weight θ is 0.975. This means that the Trainable Weight algorithm quickly moves θ into the neighborhood of this optimal value then passes this optimal value. Although theoretically the minimax weighting seeks to find a saddle point in the weight space

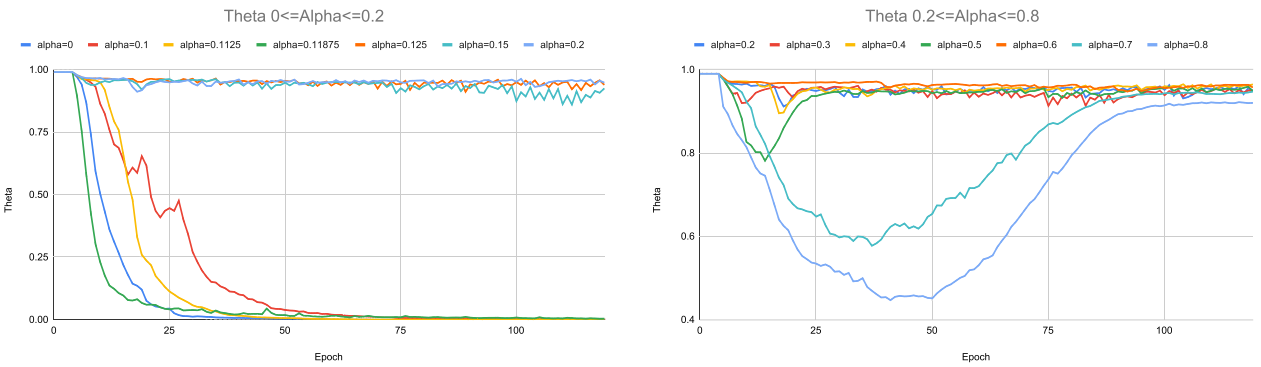

Fig. 17. Loss-based Momentum Weight θ value per epoch for different α values using Loss-Based Momentum Weight. Left: $0 \leq \alpha \leq 0.2$. Right: $0.2 \leq \alpha \leq 0.8$.

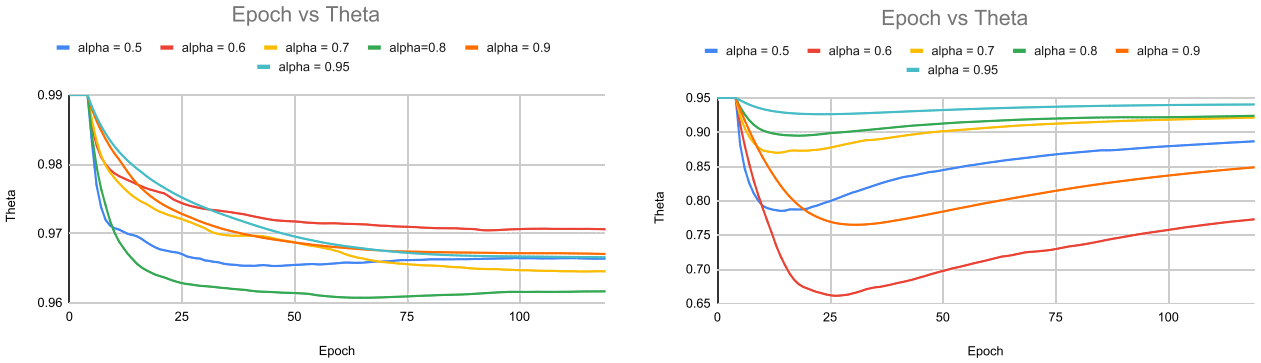

Fig. 18. Loss-based Momentum Weight Alternative implementation θ per epoch for different θ_0 and α values.

[9], throughout our study, we have not seen the stabilization of θ as theoretically predicted. As discussed in Gu et al. [8], the stabilization may be related to the use of a suitable "mask function".

A4. Loss-based momentum weight

The Loss-Based Momentum Weight algorithm requires us to choose the initial value θ_0 the weight α, and T, which was chosen to be 5 for based on a few trainings. Since L_1^{loss} is usually a large quantity in the early phase of training, we chose a θ_0 that is above the expected range of optimal θ values. Based on some preliminary testing, $\theta_0 = 0.99$ was selected. The weight α also significantly determines the training dynamics. In our numerical tests the initial loss ratio comes from the 5th epoch (instead of the average of the first 5 epochs). Fig. 17 shows how θ changes over the epochs depending on the different α values. This is split into two graphs, since there are two different phenomena present at the low and high range of the α values.

The left panel of Fig. 17 shows that small values of α make θ converge towards zero quickly. This is expected because the value of θ is not properly stabilized, hence the problem of the negative feedback loop is not prevented. The transition starts when α reaches approximately 0.125, at which point the dynamics of θ stabilize. On the right panel of Fig. 17, we see a very different behavior going on for larger values of α. When α is greater than 0.5, it becomes harder to revert the initial decreasing trend of θ because the updates on θ are too small. As seen in the figure, θ takes large dips that are followed by increases back towards the stable area near $\theta = 0.95$. Larger values of α make the stabilization even slower. In summary, we conclude that the optimal range of α values appears to be roughly between 0.2 and 0.4. As larger values make convergence of θ to 0 less likely, $\alpha = 0.4$ was selected.

As mentioned previously, the alternative implementation of Loss-Based Momentum Weight method uses the average of all historical loss ratios, hence both θ_0 and α needed to be selected again. Fig. 18 demonstrates the evolution of θ across the epochs when starting from $\theta_0 = 0.99$ (left panel) and $\theta_0 = 0.95$ (right panel). As seen in Fig. 18, the dynamics of θ are relatively stable for a wide range of values of α, although θ moves too slowly when α is close to 1. Here we believe $\theta_0 = 0.99$ remains a good choice of the initial value, and the L^2 error results indicate that $\alpha = 0.6$ is optimal.

A4.1. Gradient-based momentum weight

Finally, Gradient-Based Momentum Weight also requires selecting θ_0 and α. Our numerical experiment shows that the performance is not very sensitive against the choice of these hyper-parameters, possibly because the dynamics of θ have

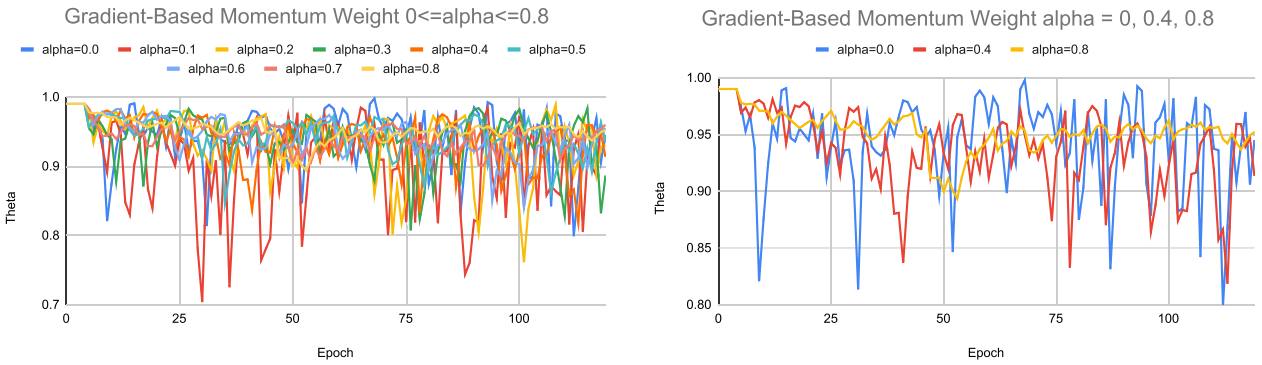

Fig. 19. Gradient-Based Momentum Weight θ value per epoch for Gradient-Based Momentum Weight using different α values.

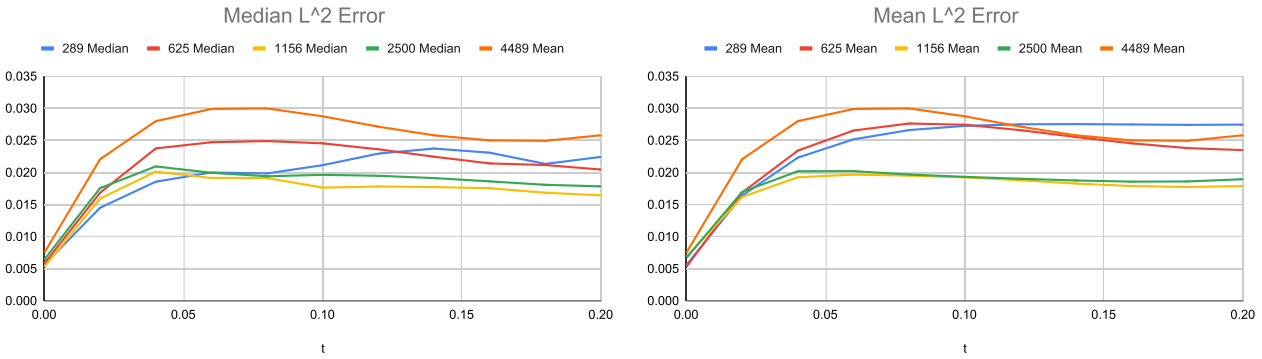

Fig. 20. Median and Mean L^2 error from 11 trainings using Anchor Sampling with Grid Selection for 120 epochs. Sample counts of 289, 625, 1156, 2500, and 4489 are used at $t = 0.2$.

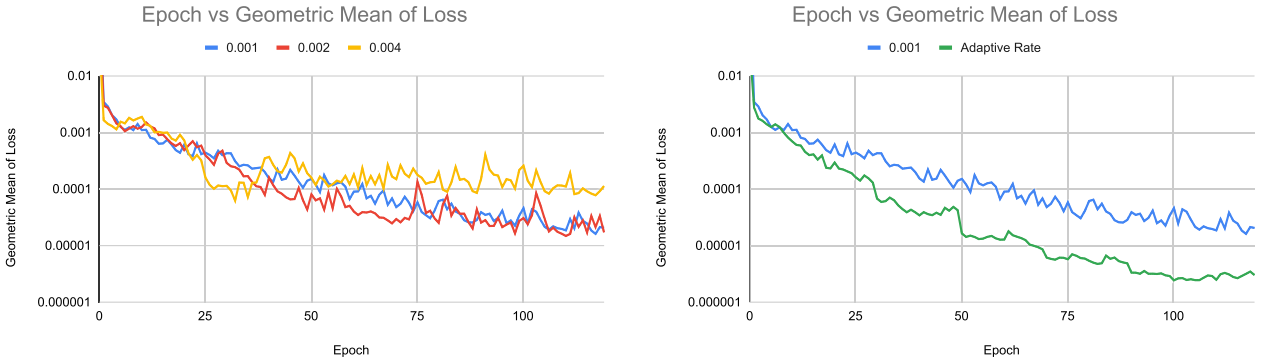

Fig. 21. Left: Geometric mean of loss per epoch for learning rates of 0.001, 0.002 and 0.004 using 5 trainings each. Right: Geometric mean of loss per epoch for learning rate of 0.001 and adaptive learning rate using 5 trainings each.

significant fluctuation anyway. For consistency $\theta_0 = 0.99$ was selected. Fig. 19 shows θ per epoch for different α values. Increasing α decreases the variance and vice versa, but no other behavior is introduced by changing α, and performance remains similar across all α values. Further tests showed that L^2 error is lowest at $\alpha = 0.4$, so this was selected.

A5. Anchor sampling implementation

The median and mean L^2 error over the time domain for five training point counts at $t = 0.2$ can be seen below in Fig. 20. Grid selection and 120 epochs of training were used here. 1156 and 2500 points produced similar results, while the rest produced worse results. Because of this 1156 was selected.

A6. Adaptive learning rate

The performance of neural network training can be further improved by using an adaptive learning rate. This is demonstrated on the left in Fig. 21, which shows the geometric mean of the loss for five trainings at different learning rates. Larger

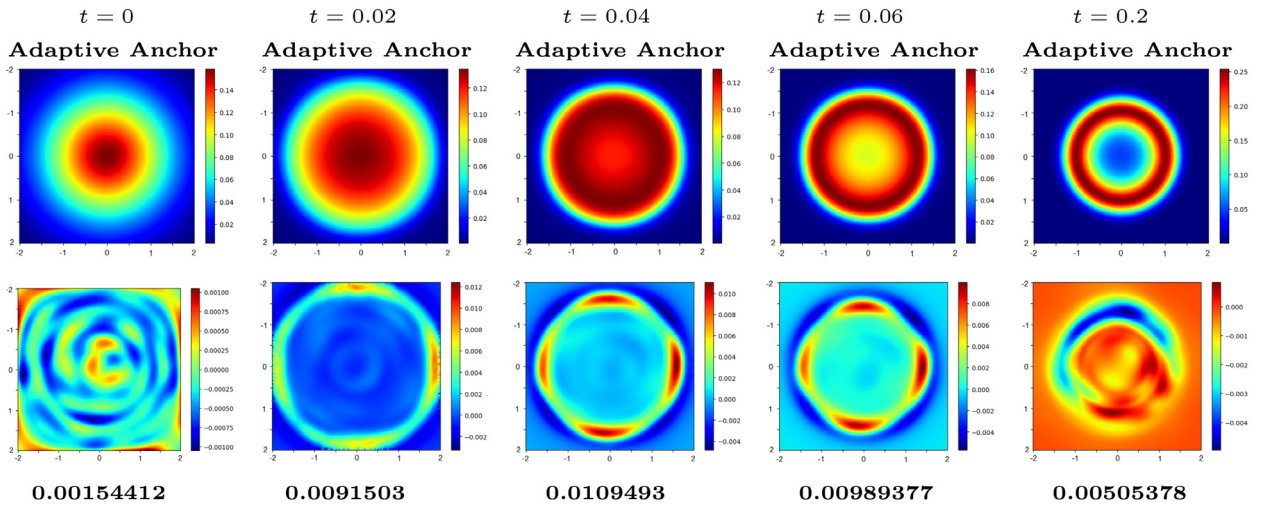

Fig. 22. Neural network solutions and error heat maps of median results from 11 sample trainings per configuration at $t = 0, 0.02, 0.04, 0.06, 0.2$ for Anchor Sampling using the adaptive learning rate for 120 epochs. L^2 error is listed below the respective heat map.

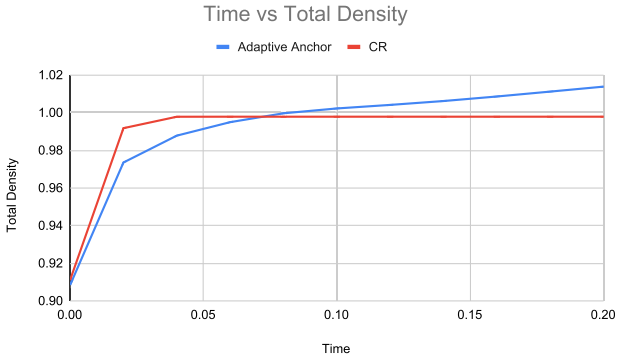

Fig. 23. Total density at each time for Adaptive Anchor and Crank–Nicolson for the 2D SDE on domain $[-2, 2] \times [-2, 2]$.

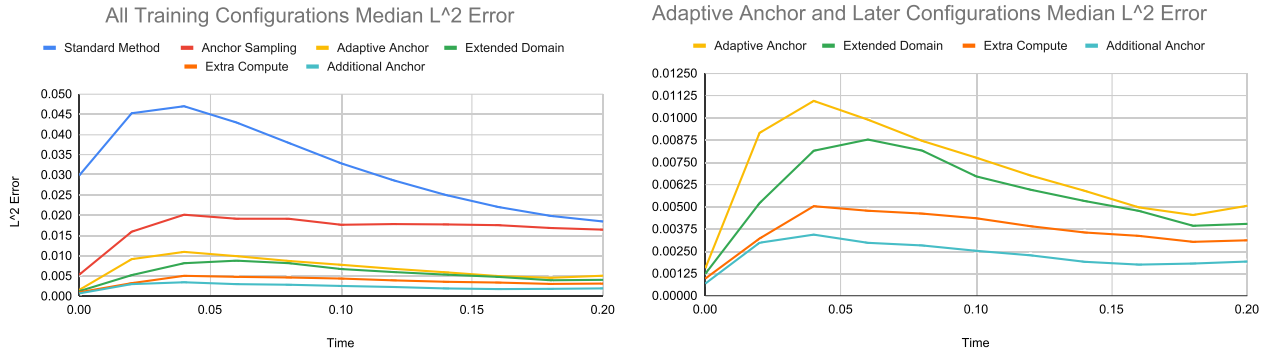

Fig. 24. Left: L^2 error for all training configurations. Right: L^2 error for adaptive anchor and later training configurations.

learning rates give faster initial convergence but higher final loss. Starting with a learning rate of 0.004, after the first 20 epochs we check every 10 epochs if the loss in the past 10 epochs is at least half of the loss from the 10 epochs before those, and if not we halve the learning rate. This training technique enables much faster training as well as lower final loss after 120 epochs.

A7. Numerical domain

In addition to hyper-parameters studied above, the numerical domain plays a role in the performance of the Fokker–Planck solver. Fig. 22 shows the median solutions and corresponding error heat maps from 11 training outputs using the

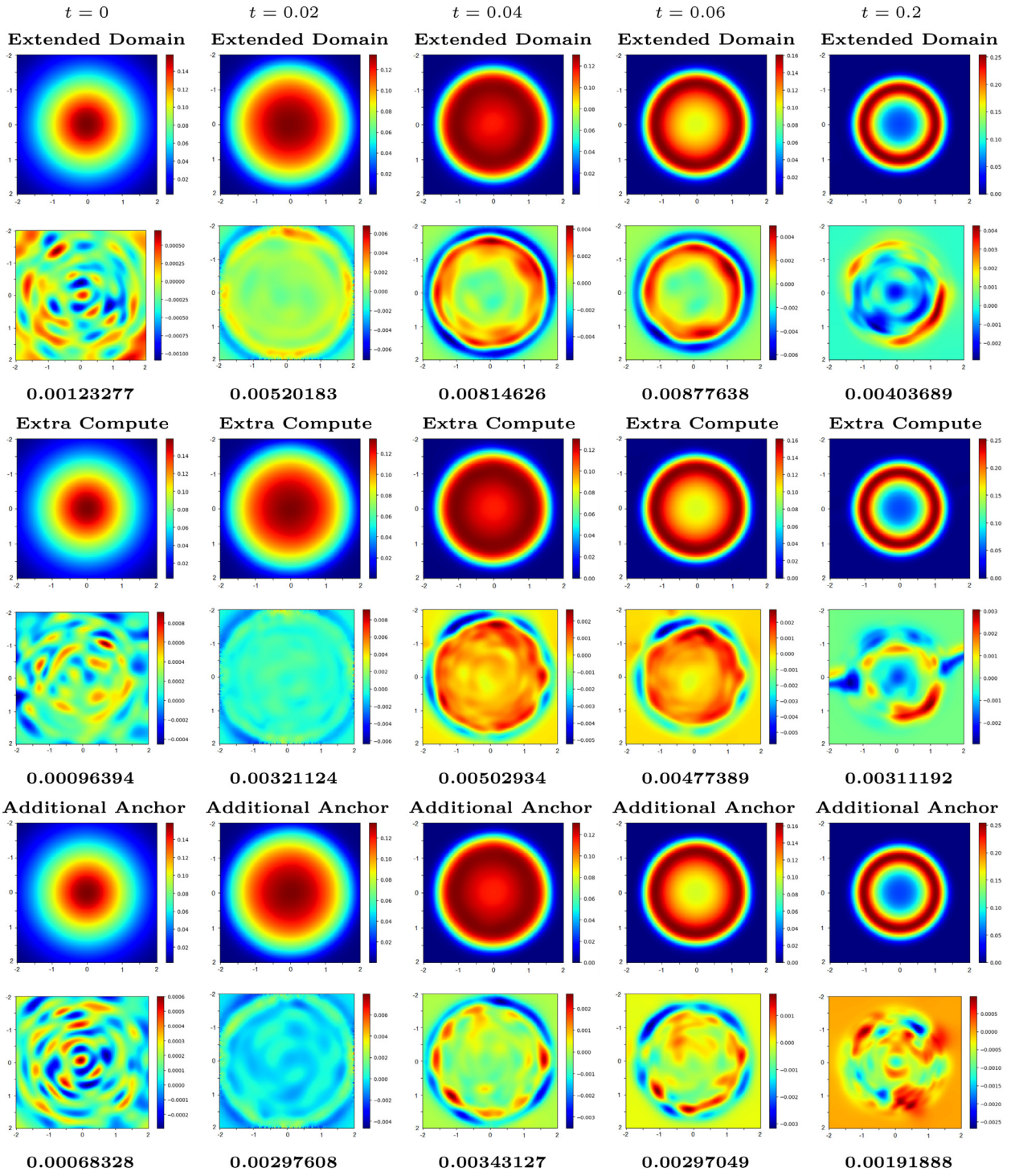

Fig. 25. Neural network solutions and error heat maps of median results from 11 sample trainings per configuration at $t = 0, 0.02, 0.04, 0.06, 0.2$ for Anchor Sampling using the adaptive learning rate for 120 epochs. Extended Domain extends the domain to $[-2.5, 2.5] \times [-2.5, 2.5]$, Extra Compute increases the mini-batch size to 128 and mini-batch count to 1250 on top of the extended domain, and Additional Anchor adds an extra anchor at $t = 0.05$ on top of the extended domain and extra compute. L^2 error is listed below the respective heat map.

adaptive learning rate with anchor sampling for 120 epochs. While the adaptive learning rate is able to reduce the L^2 error, if we compare the results to those in Fig. 11 we can see the structured nature of the error is even more apparent. Given that the error is being computed using a Crank–Nicolson solution on a larger domain with a very fine mesh as the ground truth, it appears that this boundary error is a product of the neural network training in some way.

A potential problem with the current numerical domain is that it does not include 8.893 percent of the density from the initial distribution, most of which is excluded at the center of the four sides due to the circular distribution. The error heat maps in Fig. 22 show that much of the error is from an underestimate of the density near these four edges, which propagates inward as the density concentrates at the unit circle. Although theoretically in our algorithm the boundary information is replaced by the Monte Carlo data, this example shows that the boundary effect still affects the accuracy of the solution. Despite this, the total density at each time remains relatively close to the true density within that domain, as can be seen below in Fig. 23.

If we extend the numerical domain to $[-2.5, 2.5] \times [-2.5, 2.5]$ we only exclude 2.468 percent of the density from the initial distribution. To maintain the density and ratio of training points we update $\mathfrak{X}$ to 62,500 uniformly sampled points and $\mathfrak{Y}$ to 62,500 points from the initial distribution and 1800 from the anchor at $t = 0.2$. Additionally, we increase the batch sizes to 64 from 32. The results from this can be seen in the first two rows of Fig. 25.

A8. Additional compute

Finally, we demonstrate the effect of even longer training by increasing the mini-batch sizes to 128 and doubling the number of batches, while keeping the training data the same. This means that each batch covers the training data multiple times, but we find that increasing the size of the training sets in addition to the longer training does not make a meaningful difference. It takes slightly under one hour to train the neural network. Fig. 25 shows that this reduces both the L^2 error and much of the structure of the error. Then we add another anchor of 1800 points at $t = 0.05$ in addition to the extended domain and additional compute. This has a less notable impact than increasing the mini-batch size and count, but still produces the best results. The L^2 error for all training configurations can be seen in Fig. 24 for comparison. Estimating the theoretical error of neural network PDE solvers is a very challenging problem. However, this experiment shows that in practice it is difficult to make L^2 error much lower than 10^{-3} with reasonable training cost.

References

[1] M. Dobson, Y. Li, J. Zhai, An efficient data-driven solver for Fokker–Planck equations: algorithm and analysis, Commun. Math. Sci. 20 (3) (2022) 803–827.
[2] Y. Li, A data-driven method for the steady state of randomly perturbed dynamics, Commun. Math. Sci. 17 (4) (2019) 1045–1059.
[3] J. Zhai, M. Dobson, Y. Li, A deep learning method for solving Fokker–Planck equations, in: J. Bruna, J. Hesthaven, L. Zdeborova (Eds.), Proceedings of the 2nd Mathematical and Scientific Machine Learning Conference, 16–19 Aug, Proceedings of Machine Learning Research, PMLR, 2022, pp. 568–597.
[4] X. Chen, L. Yang, J. Duan, G.E. Karniadakis, Solving inverse stochastic problems from discrete particle observations using the Fokker–Planck equation and physics-informed neural networks, SIAM J. Sci. Comput. 43 (3) (2021) B811–B830.
[5] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.
[6] D.P. Kingma, J.B. Adam, A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).
[7] H. Eivazi, R. Vinuesa, Physics-informed deep-learning applications to experimental fluid mechanics, arXiv preprint arXiv:2203.15402 (2022).
[8] Y. Gu, H. Yang, C. Zhou, SelectNet: self-paced learning for high-dimensional partial differential equations, J. Comput. Phys. 441 (2021) 110444.
[9] D. Liu, Y. Wang, A dual-dimer method for training physics-constrained neural networks with minimax architecture, Neural Netw. 136 (2021) 112–125.
[10] L. McClenny, U. Braga-Neto, Self-adaptive physics-informed neural networks using a soft attention mechanism, arXiv preprint arXiv:2009.04544 (2020).
[11] J.W. Thomas, Numerical Partial Differential Equations: Finite Difference Methods, vol. 22, Springer Science & Business Media, 2013.
[12] D.D. Lee, P. Pham, Y. Largman, A. Ng, Advances in neural information processing systems 22, Technical report, Tech. Rep., Tech. Rep, 2009.
[13] Z. Chen, J. Lu, Y. Lu, On the representation of solutions to elliptic PDEs in barron spaces, in: M. Ranzato, A. Beygelzimer, Y.N. Dauphin, P. Liang, J.W. Vaughan (Eds.), Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6–14, 2021, Virtual, 2021, pp. 6454–6465.
[14] E. Weinan, S. Wojtowytsch, Some observations on high-dimensional partial differential equations with barron data, Math. Sci. Mach. Learn. (2022) 253–269.
[15] D.C. Liu, J. Nocedal, On the limited memory BFGS method for large scale optimization, Math. Program. 45 (1) (1989) 503–528.