Learning Generative Neural Networks with Physics Knowledge

Kailai Xu[1*], Weiqiang Zhu[3*] and Eric Darve[1,2*]

[1*]Institute for Computational and Mathematical Engineering, Stanford University, Stanford, 94305, CA, USA.
[2*]Mechanical Engineering Department, Stanford University, Stanford, 94305, CA, USA.
[3*]Department of Geophysics, Stanford University, Stanford, 94305, CA, USA.

*Corresponding author(s). E-mail(s): kailaix@stanford.edu; zhuwq@stanford.edu; darve@stanford.edu;

Abstract

Deep generative neural networks have enabled modeling complex distributions, but incorporating physics knowledge into the neural networks is still challenging and is at the core of current physics-based machine learning research. To this end, we propose a Physics Generative Neural Network (PhysGNN), a new class of generative neural networks for learning unknown distributions in a physical system described by partial differential equations (PDE). PhysGNN couples PDE systems with generative neural networks. It is a fully differentiable model that allows back-propagation of gradients through both numerical PDE solvers and generative neural networks, and is trained by minimizing the discrete Wasserstein distance between generated and observed probability distributions of the PDE outputs using the stochastic gradient descent method. Moreover, PhysGNN does not require adversarial training like standard generative neural networks (GAN), which offers better stability than adversarial training. We show that PhysGNN can learn complex distributions in stochastic inverse problems, where conventional methods such as maximum likelihood estimation and momentum matching methods may be inapplicable when little knowledge is known about the form of unknown distributions or the physical model is too complex. Our method allows physics-based generative neural network training for learning complex distributions in the context of differential equations.

Keywords: Generative Neural Networks, Automatic Differentiation, Partial Differential Equations

1 Introduction

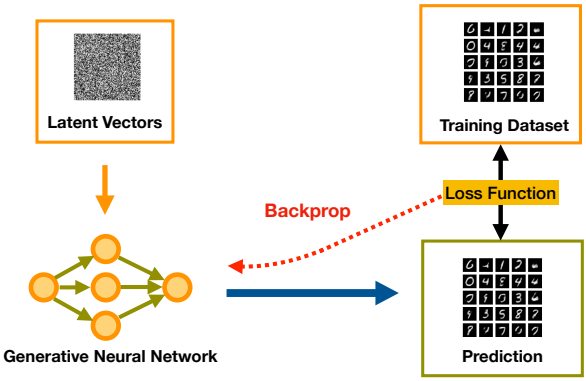

Fig. 1 Traditional deep generative neural networks.

In this paper, we consider how to incorporate physics knowledge into training generative neural networks for learning distributions of physical parameters. Unlike traditional deep generative neural networks [1–4] (Fig. 1) that are directly trained to represent data distribution, our physics generative neural networks (PhysGNN) (Fig. 2) are trained to represent underlying distributions of physical parameters by imposing physical constraints on the final layer outputs of the generative neural networks. In doing so, we can generalize deep generative models to solve a wide variety of physical problems.

Physical constraints are usually described by partial differential equations (PDE) and solved by numerical solvers such as finite element analysis [5]. We consider a PDE that contains random processes. We will denote by $\boldsymbol{\omega}$ a known random process and by $\boldsymbol{\theta}$ an unknown random process that we are trying to learn. We will denote by $\mathbf{y}$ the solution of the PDE. It is typically a function of space and/or time. We define formally the map f as follows

$$f : (\boldsymbol{\theta}, \boldsymbol{\omega}) \mapsto \mathbf{y}_{\boldsymbol{\theta},\boldsymbol{\omega}}, \tag{1}$$

where $\mathbf{y}_{\boldsymbol{\theta},\boldsymbol{\omega}}(r,t)$ may be for example a function of the space variable r and time variable t. For example, in a stochastic finite element analysis for static linear elasticity [6], the equilibrium equation has the form

$$\mathbf{K}(\boldsymbol{\theta})\,\mathbf{y}_{\boldsymbol{\theta},\boldsymbol{\omega}} = \mathbf{h}(\boldsymbol{\omega}), \tag{2}$$

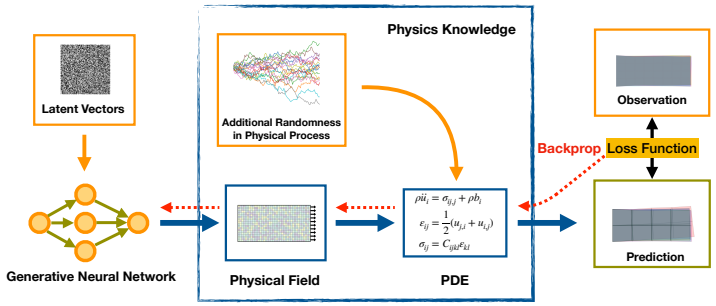

Fig. 2 Physics generative neural networks. The physics knowledge is described by partial differential equations (PDE) that require as input physical parameters produced by the generative neural network. The gradients are computed by automatic differentiation through both the numerical PDE solver and the generative neural network. In the "observation" and "prediction" blocks, each color represents a deformed domain and we can observe the displacements. We have multiple different deformations because the media is random.

where $\mathbf{K}(\boldsymbol{\theta})$ is the stochastic stiffness matrix, $\mathbf{y}_{\boldsymbol{\theta},\boldsymbol{\omega}}$ is now the stochastic displacement vector, and $\mathbf{h}(\boldsymbol{\omega})$ is the stochastic force vector. $\boldsymbol{\theta}$ is a sample of certain properties of a random media, and $\boldsymbol{\omega}$ is a Gaussian distribution, which we use to generate the force vector.

The generative neural network is used to generate $\boldsymbol{\theta}$ samples:

$$\boldsymbol{\theta} = g(\mathbf{z};\ \mathbf{w}),$$

where $\mathbf{w}$ represents the weights and biases of the neural network, and $\mathbf{z}$ is the latent random vector.

Since $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ are random variables, the hypothetical output $\mathbf{y}$ is also a random variable. The loss function in PhysGNN is formulated by measuring the distance between the distributions of the hypothetical output $\mathbf{y}$ (e.g., in the stochastic finite element example, $\mathbf{y}$ can be a realization of the stochastic displacements at certain locations) and the observed output using the discrete Wasserstein distance [7].

We require that the numerical PDE model in Eq. (1) is differentiable so that the gradients can be calculated by automatic differentiation (AD) [8–10] and the coupled system of PDEs and generative neural networks can be trained by minimizing the loss function using gradient descent methods. One challenge in implementing AD is back-propagating the gradient through *implicit* numerical PDE solvers. We derived an efficient algorithm based on the implicit function theorem, which fits into the reverse-mode AD framework. This algorithm allows us to couple generative neural networks and both explicit and implicit numerical PDE solvers, making PhysGNN applicable to most physical equations and problems for modeling distributions of underlying physical parameters.

2 Literature Review

Physics informed neural networks

Physics informed neural networks (PINNs) [11–13] approximates the solutions to PDEs with a deep neural network and the PDE is solved by minimizing a loss function consisting of governing equation and data mismatch at different locations. PINNs leverage automatic differentiation (AD) of current deep learning frameworks to calculate the partial derivative terms in a PDE. To handle stochasticity, PINNs has been extended to stochastic partial differential equations [14], Bayesian frameworks [15], and uncertainty quantification [16]. Instead of solving the PDE by minimizing a loss function, PhysGNN solves the PDE with a numerical scheme, which is more accurate and faster [17].

Deep generative models

Deep generative models, such as generative adversarial neural networks (GAN) [18] and variational auto-encoders (VAE) [19, 20], usually train a generative neural network using samples from the target distribution (Fig. 1). Flow-based generative models [21] explicitly learn the probability density function of the input data. However, the flow-based models require a sequence of invertible transformation, which is not applicable for our problems. PhysGNN couples PDEs with generative neural networks to learn an underlying distribution of physical parameters (target distribution). The training data are then the realizations of the state variables of the PDEs instead of samples from the target distribution. Consequently, the gradients must be back-propagated through both the neural networks and the numerical PDE solvers.

Computational optimal transport

Computational optimal transport provides an efficient way to compute the discrepancy based on the optimal transport distance like the Wasserstein distance [7, 22, 23], which is used in PhysGNN to measure the discrepancy between the distributions of hypothetical outputs and observed outputs. [24] presents a method to train large-scale generative models by minimizing the discrete Wasserstein distance with an entropy penalty term. In this work, we apply netflow solvers [25] to directly minimize the discrete Wasserstein distance, which achieves a similar performance on our benchmark problem but avoids tuning the penalty term.

Automatic differentiation and adjoint state method

The reverse-mode automatic differentiation and the adjoint state method can be shown to be mathematically equivalent [26]. The adjoint state method has been widely used for PDE-constrained optimization problems in scientific computing. It has also been used in [27] to obtain an efficient computational method for training neural ordinary differential equations (ODE). Neural ODEs build interpretable neural architectures by parameterizing derivatives of the state variables in ODEs using neural networks. In PhysGNN, we derive and

implement automatic differentiation algorithms for both explicit and implicit numerical PDE solvers.

Markov Chain Monte Carlo

It is worth mentioning that Markov Chain Monte Carlo (MCMC) [28–32], which has been widely applied to sampling-based algorithms, is not easily applicable to our problems. MCMC usually requires the likelihood function of $\boldsymbol{\theta}$'s distribution to generate samples by constructing a Markov chain. However, it is difficult to obtain the likelihood function of $\boldsymbol{\theta}$ analytically given the PDE model in Eq. (1). However, for comparison, we will present some benchmarks against MCMC for cases where we can approximate the likelihood function using the method of Gaussian kernel density estimation. We purposely design the benchmark case so that MCMC is applicable.

3 PhysGNN: Training Generative Neural Networks with Physics Knowledge

3.1 Minibatch Discrete Wasserstein Distance

In general, the mismatch between two distributions can be computed using any probability metrics; in this paper, we consider *the discrete Wasserstein distance,* which offers attractive features such as differentiability, efficiency, and stability. Assume that $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^{s\times n}$ and $\mathbf{y} = (y_1, \ldots, y_m) \in \mathbb{R}^{s\times m}$, where m and n are sample sizes and s is the dimension of x_i and y_j. The cost matrix is given by $\underline{c} \in \mathbb{R}^{n\times m}$, $\underline{c}_{ij} = c(x_i, y_j)$, where c is the cost function. When x_i and y_j are in the same metric space $\mathcal{X}$, a common choice for c is $c(x, y) = d_{\mathcal{X}}(x, y)$, where $d_{\mathcal{X}}$ is a distance associated with the space. The cost of mapping from $\mathbf{x}$ to $\mathbf{y}$ with the joint probability matrix p_{ij} is quantified as $\sum_{i,j} p_{ij}\underline{c}_{ij}$. The discrete Wasserstein distance can be computed by solving an optimization problem [33, 34]

$$\begin{aligned} &\min_{p_{ij}} W(\mathbf{p};\ \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij}\underline{c}_{ij}, \\ &\text{s.t.}\ \ \sum_{j=1}^{m} p_{ij} = \frac{1}{n},\ \ \sum_{i=1}^{n} p_{ij} = \frac{1}{m},\ \ 0 \le p_{ij} \le 1. \end{aligned} \tag{3}$$

In our case, $\mathcal{X}$ is the Euclidean space and we let $d_{\mathcal{X}}(x, y) = \|x - y\|_1$ (or other choices such as $d_{\mathcal{X}}(x, y) = \|x - y\|_2$). The corresponding discrete Wasserstein distance (referred to as "loss function") is

$$D(\mathbf{x}, \mathbf{y}) = W(\mathbf{p}^*;\ \mathbf{x}, \mathbf{y}) = \sum_{1\le i\le n, 1\le j\le m} p^*_{ij}\|x_i - y_j\|_1. \tag{4}$$

Here $[\mathbf{p}^*]_{ij} = p^*_{ij}$ is the optimal solution to Eq. (3) and is a function of $\mathbf{x}$ and $\mathbf{y}$.

The rationale behind using the discrete Wasserstein distance is that the distance is a good approximation to the infinite-dimensional optimal transportation cost between two distributions with the cost function $c(x, y)$. It can be shown that the solution to Eq. (3) converges to the optimal transference plan between those two distributions under certain assumptions [35]. However, the optimization problem Eq. (3) implemented directly is computationally too expensive for large m and n because the problem size scales with mn. With efficient net-flow solvers or interior point methods, the worst case of the computational cost for Eq. (3) scales with $\mathcal{O}(k^3 \log k)$ (assume $m = n = k$) [22, 36]. We can further improve the computational challenging by using stochastic gradient methods with moderate m and n, which restricts the optimization problem size to an appropriate range.

There are also variants of computational methods for solving Eq. (3). For example, [22] proposed a computational approach for minimizing the original loss function with an entropic regularization term

$$W_\alpha(\mathbf{p}) = W(\mathbf{p}) - \alpha H(\mathbf{p}),\ H(\mathbf{p}) = -\sum_{i,j} p_{ij} \log p_{ij}. \tag{5}$$

This modification converts the original problem into a strictly convex problem, which is solved efficiently with the Sinkhorn-Knopp matrix scaling algorithm. However, using a regularization term smears distinct features in gradient flows and distorts the original optimization problem [37]. Therefore, it is desirable to use a penalty term α as small as possible. But a small α results in numerical instability of the Sinkhorn-Knopp algorithm. We found that using a net-flow solver to solve the original linear programming problem Eq. (3) is effective for our applications. We also compare our method with the Sinkhorn-Knopp algorithm in Section 4.1 and show that results using different optimizers are consistent and the net-flow solver is superior in terms of numerical stability.

3.2 Automatic differentiation

Differentiating

$$D(\{\tilde{\mathbf{y}}_i(\mathbf{w}^k)\}_{i=1}^{n_{\text{batch}}}, \{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}}) \tag{6}$$

with respect to $\mathbf{w}^k$ requires carrying out the forward step of Algorithm 1, storing all intermediate results, and computing the gradients with the chain rule. Note that

$$D(\{\tilde{\mathbf{y}}_i(\mathbf{w}^k)\}_{i=1}^{n_{\text{batch}}}, \{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}})$$

only depends on the pairwise distance matrix $\underline{c}$ between $\{\tilde{\mathbf{y}}_i(\mathbf{w}^k)\}_{i=1}^{n_{\text{batch}}}$ and $\{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}}$

$$\underline{c}_{ij} := d_{\mathcal{X}}(\tilde{\mathbf{y}}_i, \mathbf{y}_j) = \|\tilde{\mathbf{y}}_i - \mathbf{y}_j\|_1.$$

Because the value of $D(\{\tilde{\mathbf{y}}_i(\mathbf{w}^k)\}_{i=1}^{n_{\text{batch}}}, \{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}})$ can be determined by $[d_{\mathcal{X}}(\tilde{\mathbf{y}}_i, \mathbf{y}_j)]$, without confusion we can write

$$D([d_{\mathcal{X}}(\tilde{\mathbf{y}}_i, \mathbf{y}_j)]) := D(\{\tilde{\mathbf{y}}_i\}_{i=1}^{n_{\text{batch}}}, \{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}}).$$

Algorithm 1 Optimization algorithm for PhysGNN. Here $g(\mathbf{z};\ \mathbf{w})$ is a neural network with weights and biases $\mathbf{w}$ and inputs $\mathbf{z}$, n_{batch} is the batch size, $f(\boldsymbol{\theta}, \boldsymbol{\omega})$ is the physical model, $D(\cdot, \cdot)$ is the discrete Wasserstein distance between two *discrete* distributions, and $\alpha^k > 0$ is the step size for the gradient descent at step k.

1: Initialize the neural network weights and biases $\mathbf{w}^1$
2: **for** $k = 1, 2, 3, \ldots$ **do**
3: Sample $\{\mathbf{z}_i\}_{i=1}^{n_{\text{batch}}}$ from $\mathcal{N}(0, I_N)$ (I_N is the identity matrix of N dimension)
4: Compute $\tilde{\boldsymbol{\theta}}_i = g(\mathbf{z}_i;\ \mathbf{w}^k)$, $i = 1, 2, \ldots, n_{\text{batch}}$
5: Sample $\{\boldsymbol{\omega}_i\}_{i=1}^{n_{\text{batch}}}$ from the known distribution of $\boldsymbol{\omega}$
6: Compute $\tilde{\mathbf{y}}_i = f(\tilde{\boldsymbol{\theta}}_i, \boldsymbol{\omega}_i)$, $i = 1, 2, \ldots, n_{\text{batch}}$ (Note that this step requires solving a PDE. For example, in the stochastic finite element example Eq. (2), we have $\tilde{\mathbf{y}}_i = \mathbf{K}(\tilde{\boldsymbol{\theta}}_i)^{-1}\mathbf{f}(\boldsymbol{\omega}_i)$.)
7: Sample $\mathbf{y}_i$ from the observation distribution
8: Compute the gradients

$$G^k = \nabla_{\mathbf{w}^k} D(\{\tilde{\mathbf{y}}_i(\mathbf{w}^k)\}_{i=1}^{n_{\text{batch}}}, \{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}})$$

Here $\tilde{\mathbf{y}}_i$ depends on $\mathbf{w}^k$ through $\tilde{\boldsymbol{\theta}}_i$; therefore, we write $\tilde{\mathbf{y}}_i(\mathbf{w}^k)$. The associated optimization problem Eq. (3) is solved with net-flow solvers.
9: Update the weights and biases of the neural network

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha^k G^k$$

10: **end for**

The chain rule can be expressed by

$$\frac{\partial D(\{\tilde{\mathbf{y}}_i(\mathbf{w}^k)\}_{i=1}^{n_{\text{batch}}}, \{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}})}{\partial \mathbf{w}^k} = \sum_{i,j=1}^{n_{\text{batch}}} \frac{\partial D([\underline{c}_{ij}])}{\partial \underline{c}_{ij}} \frac{\partial d_{\mathcal{X}}(\tilde{\mathbf{y}}_i, \mathbf{y}_j)}{\partial \tilde{\mathbf{y}}_i} \frac{\partial f(\tilde{\boldsymbol{\theta}}_i, \boldsymbol{\omega}_i)}{\partial \tilde{\boldsymbol{\theta}}_i} \frac{\partial g(\mathbf{z}_i;\ \mathbf{w}^k)}{\partial \mathbf{w}^k}. \tag{7}$$

The last term $\frac{\partial g(\mathbf{z}_i;\ \mathbf{w}^k)}{\partial \mathbf{w}^k}$ is the derivative of the neural network's outputs with respect to its weights and biases and can be computed using AD. $\frac{\partial d_{\mathcal{X}}(\tilde{\mathbf{y}}_i, \mathbf{y}_j)}{\partial \tilde{\mathbf{y}}_i}$ can be evaluated analytically and thus also implemented using AD. In the following, we describe how the other two terms in Eq. (7) are computed.

AD for the discrete Wasserstein distance

The first term $\frac{\partial D([\underline{c}_{ij}])}{\partial \underline{c}_{ij}}$ can be computed using the following theorem [38].

Theorem 1 *Assume that* $\mathbf{p}^*([\underline{c}_{ij}])$ *is the optimal solution to the linear programming problem Eq. (3) given the cost matrix* $[\underline{c}_{ij}]$*, then we have*

$$\frac{\partial W(\mathbf{p}^*([\underline{c}_{ij}]))}{\partial \underline{c}_{ij}} = p^*_{ij}\,. \tag{8}$$

Eq. (8) is an elegant result and implies that the gradients can be computed as a by-product of solving the optimization problem Eq. (3). Given Eq. (4), we have

$$\frac{\partial D([\underline{c}_{ij}])}{\partial \underline{c}_{ij}} = p^*_{ij}\,.$$

AD for implicit numerical solvers

Although we write the physical model in the explicit form $\mathbf{y} = f(\boldsymbol{\theta})$ (we omit $\boldsymbol{\omega}$ since it is irrelevant here), in practice, $\mathbf{y}$ is usually numerically solved using an implicit formula

$$F(\mathbf{y};\ \boldsymbol{\theta}) = \mathbf{0} \in \mathbb{R}^d, \quad \mathbf{y} \in \mathbb{R}^d, \tag{9}$$

for reasons of accuracy, stability, or model nonlinearity. In this case, it is impossible to compute $\partial f(\boldsymbol{\theta})/\partial \boldsymbol{\theta}$ directly using AD since the analytical form of f is unknown and $f(\boldsymbol{\theta})$ must be solved from Eq. (9). In this work, we use the Physics Constrained Learning (PCL), proposed in [17], to design AD methods for implicit numerical solvers. To derive a formula that fits into the automatic differentiation framework, we apply the implicit function theorem [39] and take the gradient with respect to $\boldsymbol{\theta}$ on both sides of Eq. (9)

$$F_{\mathbf{y}} \frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + F_{\boldsymbol{\theta}} = 0.$$

In a reverse-mode AD framework, a top level gradient $\frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}}$ (a row vector) is passed to Eq. (9), where D is a scalar loss function and "..." denotes additional arguments. The gradient to be computed is

$$\frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}} \frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}} F_{\mathbf{y}}^{-1} F_{\boldsymbol{\theta}}. \tag{10}$$

There are two ways to compute Eq. (10):

1. First compute $X = F_{\mathbf{y}}^{-1} F_{\boldsymbol{\theta}} \in \mathbb{R}^{d\times p}$, which requires solving p linear systems $F_{\mathbf{y}} X = F_{\boldsymbol{\theta}}$, and then compute $\frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}} X$, where p is the number of parameters in $\boldsymbol{\theta}$.
2. First compute $Y = \frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}} F_{\mathbf{y}}^{-1} \in \mathbb{R}^{1\times d}$, which requires solving only one linear system $Y F_{\mathbf{y}} = \frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}}$, and then compute $Y F_{\boldsymbol{\theta}}$.

Note the costs for computing $\frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}} X$ and $Y F_{\boldsymbol{\theta}}$ are both $\mathcal{O}(pd^2)$. In the context of PDEs, the computational costs for both methods are dominated by

solving the linear system $F_{\mathbf{y}}X = F_{\boldsymbol{\theta}}$ or $YF_{\mathbf{y}} = \frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}}$. The second approach is generally more efficient than the first due to fewer linear systems that need to be solved. Thus, we use the second approach in PhysGNN.

A simple example is $F(\mathbf{y};\ \boldsymbol{\theta}) = A(\boldsymbol{\theta})\mathbf{y} - \mathbf{h} = 0$ where the gradients in Eq. (10) are expressed by

$$\frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}}\frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\partial D(\mathbf{y},\ldots)}{\partial \mathbf{y}}A(\theta)^{-1}\left(\frac{\partial A(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}A(\boldsymbol{\theta})^{-1}\mathbf{h}\right),$$

where $\mathbf{h}$ is a vector and $A(\boldsymbol{\theta})$ is a coefficient matrix whose entry depends on $\boldsymbol{\theta}$. Such operators are encountered in Sections 4.1, 4.2 and 4.4.

3.3 Implementation

Putting everything together, we can now implement automatic differentiation for computing the gradients Eq. (7) for training PhysGNN by complementing the deep learning frameworks, such as `TensorFlow` [40] and `PyTorch` [41], with numerical PDE solvers. Specifically, we use the `Julia` [42, 43] package `ADCME.jl` [44], which allows for high performance scientific computing capabilities and provides an interface to `TensorFlow` for automatic differentiation in numerical PDE solvers and deep neural networks.

The numerical solvers in this work are implemented using the `AdFem.jl` package, which is a finite element method library—built on ADCME—that supports gradient backpropagation through numerical schemes. The operators in AdFem.jl are implemented using the Physics Constrained Learning approach described in Section 3.2.

3.4 Limitations of PhysGNN

Despite its many strengths, PhysGNN has some potential limitations, which we point out. Firstly, a key assumption of its applicability is that the physical model is differentiable. Although this is usually the case for PDEs, the physical laws may not always be conveniently expressed by PDEs; for example, the physical laws may be in the form of principles of minimum energy [45], in which case the techniques proposed in [46] are relevant.

Secondly, since we are training a generative neural network with stochastic gradient descent methods, training the neural network may require a large number of iterations. The training can be prohibitively expensive for large-scale physical problems because each iteration requires solving the PDE systems multiple times. One potential solution is to introduce reduced-order modeling methods [47, 48], which trade accuracy for speed.

Thirdly, since the optimization problem is highly non-convex, the loss function may have many local minima or may be very stiff. Proper handling of local minima is crucial for the robustness and applicability of PhysGNN. This is indeed a direction that needs more investigation in the future.

4 Numerical Benchmarks

4.1 Estimating the mean of a Gaussian random field

In the first example, we investigate the use of the discrete Wasserstein distance; for simplicity, this example does not involve deep neural networks. We only estimate one parameter $\boldsymbol{\theta} = E_0$ in a Gaussian random field $E(x)$ (x is the coordinate), where the mean is given by a constant E_0, i.e., $\mathbb{E}(E(x)) = E_0$, and the covariance matrix is given by

$$\operatorname{cov}(E(x), E(x')) = \sigma^2 \exp(-\|x - x'\|^2/\rho). \tag{11}$$

We consider a linear elasticity model, where the balance of linear momentum equation is given by [49]

$$\begin{aligned}
&\sigma_{ij,j} + \rho_d b_i = 0, \ x \in \Omega, \\
&\varepsilon_{ij} = \frac{1}{2}(u_{j,i} + u_{i,j}), \ x \in \Omega, \\
&\sigma_{ij} = \lambda \delta_{ij} \varepsilon_{kk} + \mu(\varepsilon_{ij} + \varepsilon_{ji}), \ x \in \Omega, \\
&\sigma_{ij} n_j = t_j, \ x \in \Gamma_N; \qquad u_i = (u_0)_i, \ x \in \Gamma_D, \\
&\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E\nu}{1-\nu^2}.
\end{aligned} \tag{12}$$

Here we have used Einstein notation [50], $[u_i]_{i=1,2}$ is the horizontal and vertical displacement, $[\sigma_{ij}]_{i,j=1,2}$ is the stress (internal forces per unit area), $[\varepsilon_{ij}]_{i,j=1,2}$ is the strain (local deformation), δ_{ij} is the Dirichlet Delta function, $[t_i]_{i=1,2}$ is the external pressure, u_0 is the fixed displacement, Ω is the computational domain, ρ_d is the density, $[b_i]_{i=1,2}$ is the external forces, and Γ_N and Γ_D are the Neumann boundary for the external force and the Dirichlet boundary for the fixed displacement such that $\Gamma_N \cap \Gamma_D = \emptyset$, $\partial\Omega = \Gamma_N \cup \Gamma_D$.

Assume we can observe the x-direction displacement at a specific location in Fig. 3. The Young's modulus $E(x)$ of the material is given by a Gaussian random field (Fig. 4), and the Poisson's ratio is given by $\nu = 0.3$. We generate 10 observations using different $E(x)$ sampled from the Gaussian random fields:

- Sample $\{\mathbf{z}_i\}_{i=1}^{n_{\text{batch}}}$ from $\mathcal{N}(0, I_N)$
- Compute the Cholesky factorization of the covariance matrix $C = LL^T$, where $C_{ij} = \operatorname{cov}(E(x_i), E(x_j))$, and x_i, x_j are coordinates of the Gauss points.
- Compute $E_i = L\mathbf{z}_i + E_0$

The inverse problem is to find E_0 ($\boldsymbol{\theta}$ in Eq. (1)) using 10 x-direction displacement data at the location marked by a red dot in Figure 3. Because we have a small set of data (10 samples), we consider a fixed sampling strategy to make use of all the data: we fixed the sample $\{\mathbf{z}_i\}_{i=1}^{n_{\text{batch}}}$, $n_{\text{batch}} = 10$, and the observations $\{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}}$ in Algorithm 1. Thus for given samples $\{\mathbf{z}_i\}_{i=1}^{n_{\text{batch}}}$

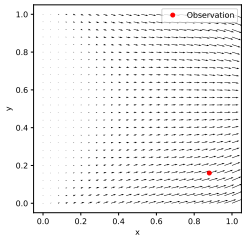

Fig. 3 An example of the solution to Eq. (12) with $\rho = 0.01$ in Eq. (11). The red dot is the location where the observation is made.

and $\{\mathbf{y}_i\}_{i=1}^{n_{\text{batch}}}$, we have a deterministic loss function. We need to solve a linear system, which is implicit, and we apply the technique in Section 3.2, to obtain the solution. We use the L-BFGS-B optimizer to minimize the loss function, because this optimizer is shown to converge faster than first order methods in similar applications [51].

The left panel in Fig. 5 shows the estimated E_0 versus different ρ values in multiple simulations. We can see that as the value of ρ becomes larger, the deviation of the estimate tends to be larger. This is because for large ρ, $E(x)$ at different locations in the Gauss random field tends to be more correlated. The large correlation means that the values $E(x)$ at the different locations are simultaneously biased upwards or downwards, making the estimate biased; when $E(x)$ is less correlated, the upward and downward biases cancel each other. In the right panel, we also show the estimates for both the net-flow solver approach and the Sinkhorn algorithm with an entropic regularization (Eq. (5)). We see that both methods yield similar results. However, we observed that for the Sinkhorn algorithm the optimizer diverges more frequently because of numerical instabilities for small α.

4.2 Poisson's Equation

In the previous example, we estimate one parameter in a random field. In this example, we show the ability of PhysGNN to learn a distribution of two parameters in deterministic physical models, i.e., no additional randomness ($\boldsymbol{\omega}$ in Eq. (1)) exists in the physical process. In Section 4.5, we consider a stochastic ODE that contains random processes in addition to the one we are estimating.

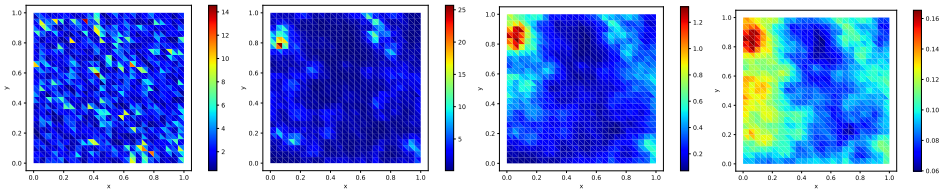

Fig. 4 Samples of the Gaussian random fields $E(x)$ in Eq. (11). From left to right: $\rho = 0.01$, 0.1, 1.0, and 10.0.

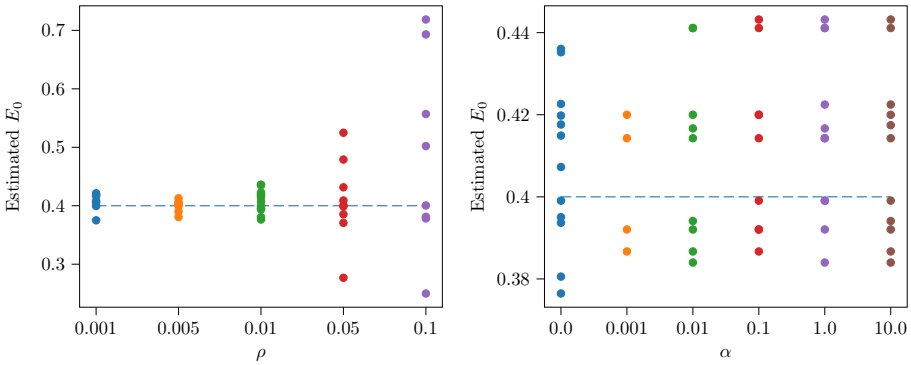

Fig. 5 Left: estimated E_0 for different ρ; each dot represents one simulation. Right: estimated E_0 for different α in the loss function. $\alpha = 0$ corresponds to the net-flow solver, and the others correspond to the Sinkhorn algorithm. Only converged results are shown in the plot; for example, there are only 4 orange dots because most of the experiments fail to converge. We use $\rho = 0.01$ in the simulation.

We consider a Poisson equation with Dirichlet boundary condition

$$\begin{cases} -\frac{\partial\left(a(x)\frac{\partial u(x)}{\partial x}\right)}{\partial x} = 1, \quad x \in (0,1), \\ u(0) = u(1) = 0, \end{cases} \tag{13}$$

where the diffusion coefficient $a(x)$ is modelled by

$$a(x) = 1 - 0.9 \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Here (μ, σ), $\sigma > 0$, is drawn from a bivariate Dirichlet distribution. For each (μ, σ) we can observe the corresponding solution to Eq. (13). Our task is to estimate the distribution of (μ, σ) from the observed distribution of u.

Since in most cases, it is impossible to solve partial differential equations analytically, equations like Eq. (13) are often solved using numerical methods. The equation is discretized on a uniform grid $0 = x_1 < x_2 < \cdots < x_{n+1} = 1$, $x_{i+1} - x_i = \frac{1}{n}$, and the solution u is represented by discrete values $u_i = u(x_i)$. The standard finite difference method [52] leads to a linear system

$$A(\mu, \sigma)\mathbf{u} = \mathbf{f}, \tag{14}$$

where $A(\mu, \sigma)$ is a tridiagonal matrix whose entries depend on μ and σ, $\mathbf{u}$ is the solution vector, and $\mathbf{f}$ is computed from the right hand side of Eq. (13). The operator to solve $\mathbf{u}$ from $\mu, \sigma, \mathbf{f}$ is implicit in nature, and we apply the technique in Section 3.2 to calculate the gradients.

We construct a generative neural network such that $(\mu, \sigma) = g(\mathbf{z}; \mathbf{w})$. The weights and biases $\mathbf{w}$ can be learned by matching the numerical solution $\mathbf{u}$

and observations $\mathbf{u}_{\text{obs}}$. Using the notations of Eq. (1) we have

$$\boldsymbol{\theta} = (\mu, \sigma), \qquad \mathbf{y} = \mathbf{u}.$$

We establish a baseline for training PhysGNN: following the standard GAN framework, we introduce a discriminative neural network $d(\mathbf{u};\ \tilde{\mathbf{w}})$ and train both $g(\mathbf{z};\ \mathbf{w})$ and $d(\mathbf{u};\ \tilde{\mathbf{w}})$ in a two-player minimax game

$$\min_{\mathbf{w}} \max_{\tilde{\mathbf{w}}} \mathbb{E}_{\mathbf{u}\sim p_{\text{data}}(u)}[\log d(\mathbf{u};\ \tilde{\mathbf{w}})] + \mathbb{E}_{\mathbf{z}\sim p_{\text{latent}}(z)}[\log(1 - d(\mathcal{S}(g(\mathbf{z};\ \mathbf{w}));\ \tilde{\mathbf{w}}))].$$

Here $\mathcal{S}((\mu, \sigma)) = A(\mu, \sigma)^{-1}\mathbf{f}$ is the numerical solver defined in Eq. (14), which is not present in traditional GANs. The GAN is trained with the RMSProp [53] optimizer as suggested by [54] and [55] while the PhysGNN is trained with the Adam optimizer where the batch size equals 32.

Fig. 6 shows that the generative neural network trained with the discrete Wasserstein distance is more accurate than that with adversarial training. Interestingly, in the adversarial training approach, the generator loss keeps oscillating and the stopping criterion is unclear. In PhysGNN, the validation error, which is computed from the average loss of 10 batches of randomly sampled (μ, σ), decreases to a certain level with small oscillation. This implies, for the current dataset and problem, that the PhysGNN approach enjoys better stability and has a good indicator (validation error) for convergence, which is important for engineering applications.

4.3 Comparison with Markov Chain Monte Carlo

In this example, we compare PhysGNN with Markov Chain Monte Carlo (MCMC) [56], which is one of the state-of-the-art methods for sampling probability distributions. We consider a 2D Poisson's equation

$$\begin{aligned} \nabla \cdot (\kappa \nabla u) &= 2\pi^2 \sin(\pi x)\sin(\pi y), && (x, y) \in (0, 1)^2, \\ u(x, y) &= 0, && (x, y) \in \partial(0, 1)^2. \end{aligned} \tag{15}$$

Here κ is a random variable, and we want to estimate the distribution of κ from observations of u at **only one point**: $u(0.8, 0.8)$. The true distribution of κ is given by

$$\kappa = |v|, \quad v \sim \mathcal{N}(1.0, 0.3^2).$$

In practice, when using a pseudo-random number generator, we exclude the case $v = 0$.

In MCMC, the density function of u is calculated using Gaussian kernel density estimation. We generate a chain of 20,000 samples and the first 4,000 samples serve as "burn-in." In PhysGNN, κ is approximated with a deep neural network with 3 hidden layers, 20 neurons per layer, and tanh activation functions. The input to the DNN is $\mathcal{N}(0, I_{10})$. The DNN is trained

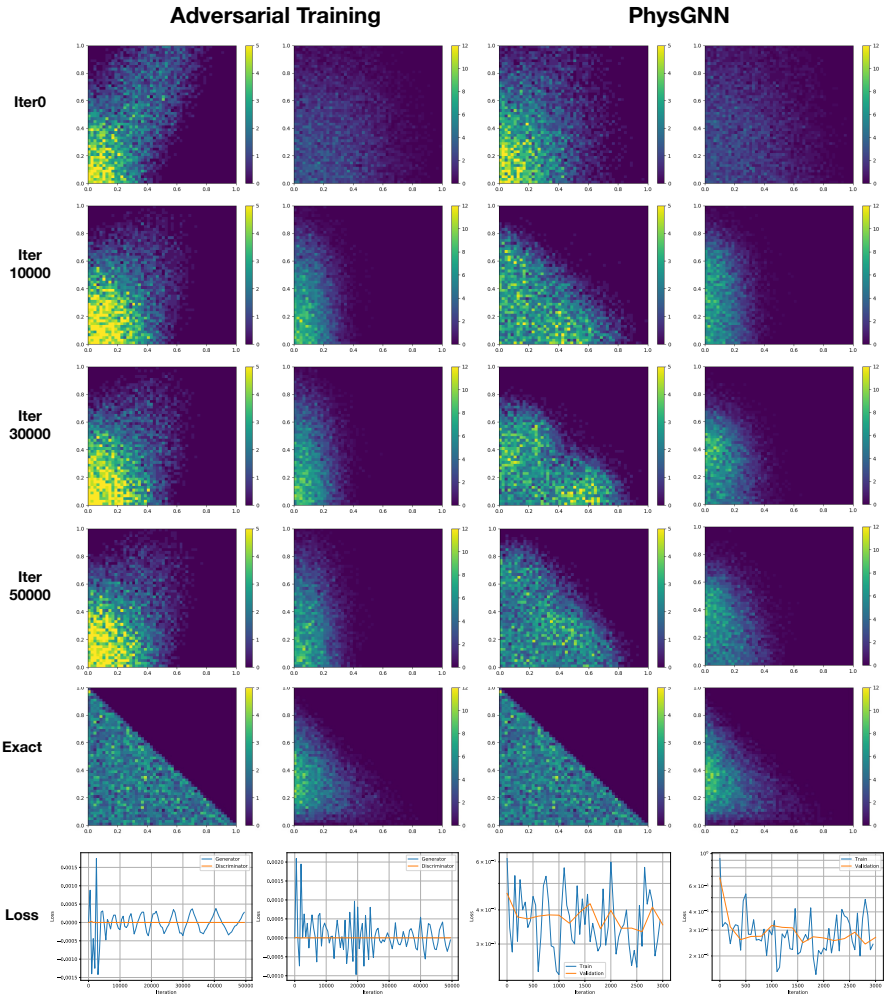

Fig. 6 Evolution of the learned distributions for (μ, σ) for Poisson's equation. The exact solution is shown in the second last column. μ is on the x-axis and σ on the y-axis for each plot. The first two rows correspond to the method of **adversarial training**. We consider two different distributions for (μ, σ). Each row is a different (μ, σ) distribution. The last two rows are **PhysGNN** (with the same two cases for the (μ, σ) distributions). Iteration counts are shown at the top. Going left to right, we see that the distributions with PhysGNN converge to the exact solution. The convergence with adversarial training is much slower.

with ADAM optimizer with a learning rate 0.001 for 600 iterations. The batch size is 16 so that the total number of forward and backward evaluations ($600 \times 16 \times 2 = 19200$) are roughly the same as forward samplings in MCMC. Note that the computations in MCMC are essentially sequential, but in PhysGNN can be parallelized within each batch. Therefore, PhysGNN is significantly faster in MCMC.

Figure 7 shows the result of κ for PhysGNN and MCMC. We can see that PhysGNN performs better than MCMC. Additionally, we find that MCMC is quite sensitive to the parameters used in the proposal distribution.

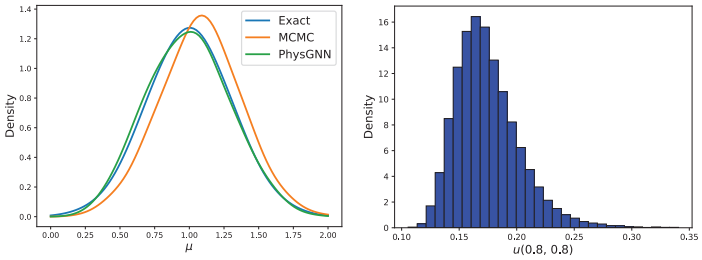

Fig. 7 Left: Results of MCMC and PhysGNN. The total number of samples in MCMC is 20000, and in PhysGNN we run the ADAM optimizer for 600 iterations, where the total number of forward and backward evaluations is 19200. Right: Histogram of the observation $u(0.8, 0.8)$.

4.4 Linear Elasticity Problem

In Section 4.2, we use a generative neural network to approximate a 2D unknown distribution in a Poisson's equation. In this section, we consider learning a high-dimensional distribution in a static linear equation using PhysGNN.

We consider linear elasticity under equilibrium conditions in mechanics as an illustration of the representation power of PhysGNN for time-independent stochastic physical models. See Section 4.1 for a detailed description of the linear elasticity problem. We consider a simple case where we pull a thin plate on the right side whose left side is fixed (Fig. 8).

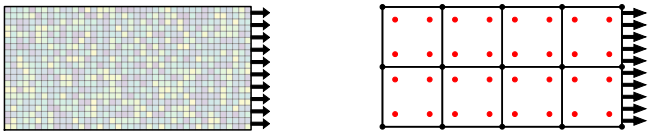

Fig. 8 The thin plate used in the linear elasticity problem. The arrows represent external forces. Left: different colors denote different physical properties. Right: the red dots denote the Gauss quadrature points used in the finite element method.

When discretized using finite element analysis, Eq. (12) can be written as a linear system

$$A(E(x), \nu(x))\mathbf{u} = \mathbf{f} \tag{16}$$

whose coefficient matrix A depends on two important physical parameters, the Young's modulus $E(x)$ and the Poisson's ratio $\nu(x)$; here $\mathbf{u}$ is a discrete value of u at element nodes and $\mathbf{f}$ is computed from body and external forces.

The system Eq. (16) is implicit and we apply the technique in Section 3.2. If the material property is heterogeneous and random, which is often seen in practice, $(E(x), \nu(x))$ can be modeled as spatially independent and identically-distributed bivariate random variables with respect to x. Note that the spatial distribution of $(E(x), \nu(x))$ is 2D; however, because we have one realization per Gauss quadrature point, the total dimension ($2\times$ number of Gauss quadrature points) of unknown distribution is high dimensional. Estimating the distributions of $(E(x), \nu(x))$ from observed plate deformations is interesting both from a methodological and practical standpoint.

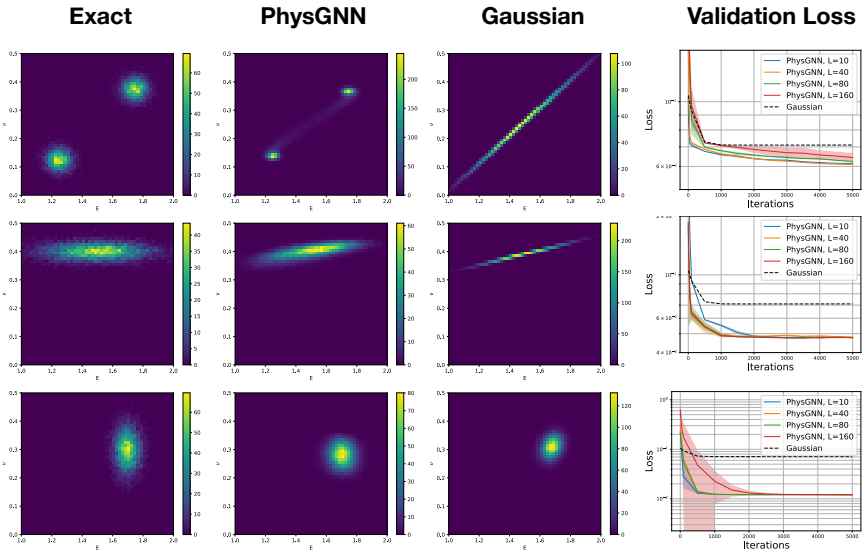

Fig. 9 Training results for the linear elasticity problem. For plots on the left panel, E is on the x-axis while ν is on the y-axis. L is the latent vector dimension. Each row is a different case. In the "Validation Loss" plots, we show the validation loss (calculated using newly generated samples) for different latent vector dimensions. We also show a one standard deviation bound for each case.

We implemented a multilayer feedforward neural network with 3 hidden layers, 20 neurons per layer, `tanh` activation functions, and a latent space of dimension $L = 20$, 40, 80, and 160. The physics consists of a finite element solver for Eq. (12), where we use 8 four-node rectangular elements with 4 Gauss quadrature points per element (Fig. 8). For assembling the stiffness matrix A, the generative neural network generates a specific $(E(x), \nu(x))$ for each element, thus 8 pairs in total. The gradient back-propagation requires differentiating $\mathbf{u} = A(E(x), \nu(x))^{-1}\,\mathbf{f}$ with respect to $(E(x), \nu(x))$, which is implemented following Eq. (10). Going back to Eq. (1), we have

$$\boldsymbol{\theta} = \{(E(x), \nu(x)) : x \in \Omega\}, \quad \mathbf{y} = \mathbf{u}.$$

We minimized the discrete Wasserstein distance using the Adam optimizer where the batch size equals 64. The training data set is generated on-the-fly and the validation data set consists of 640 deformation data using physical parameters sampled from the true distribution of $(E(x), \nu(x))$. We report the mean and the standard deviation of the validation loss in Fig. 9 based on 10 sets of random initialization of neural network weights and biases. The learned distributions shown in Fig. 9 are selected based on the validation loss. We contrast PhysGNN with the bivariate Gaussian model [57], where the target distribution is parametrized by a Gaussian distribution with trainable mean and covariance matrix, which are estimated by minimizing the same discrete loss function using the same training strategy as PhysGNN.

Fig. 9 shows that PhysGNN can learn the exact distribution of physical parameters in all test cases and achieves smaller validation losses than the Gaussian model. The Gaussian model only works for simple distributions like that in the third test case. Note that the first case is challenging because the support of the probability density function is disconnected. Nevertheless, PhysGNN has no difficulty distinguishing the two disconnected components.

4.5 Generative neural networks for stochastic physical models

Unlike deterministic physical models, stochastic physical models carry additional randomness in the physical process ($\boldsymbol{\omega}$ in Eq. (1)), which makes the learning problem more complicated.

We consider the jump diffusion process [58]

$$dX_t = \mu dt + \sigma dB_t + \sum_{i=1}^{N_t(\lambda)} Y_i, \tag{17}$$

where μ, σ are the drift mean and standard deviation, B_t is the standard Brownian motion, $N_t(\lambda)$ is a Poisson process with parameter λ, and $Y_i \in \mathbb{R}^2$ are independent and identically distributed random variables. To make the problem more challenging, we only assume that the values $X_T \in \mathbb{R}^2$ at the terminal time $T > 0$ are observable. The validation loss is computed by averaging loss function values from 150 randomly sampled latent vectors.

The target is to estimate the distributions of Y_i based on observations data X_T. Conventionally, restricting to certain forms of Y_i, maximum likelihood estimation (MLE) [59] can be used to estimate the parameters in the parametrized distribution of Y_i. However, in our case we have no assumptions on the specific form of Y_i, thus MLE is not suitable. In our method, we construct a generative neural network to approximate Y_i. The neural network is a multilayer feedforward neural network with 4 hidden layers, 20 neurons per layer, `tanh` activation functions, and a 20-dimensional latent space. The neural

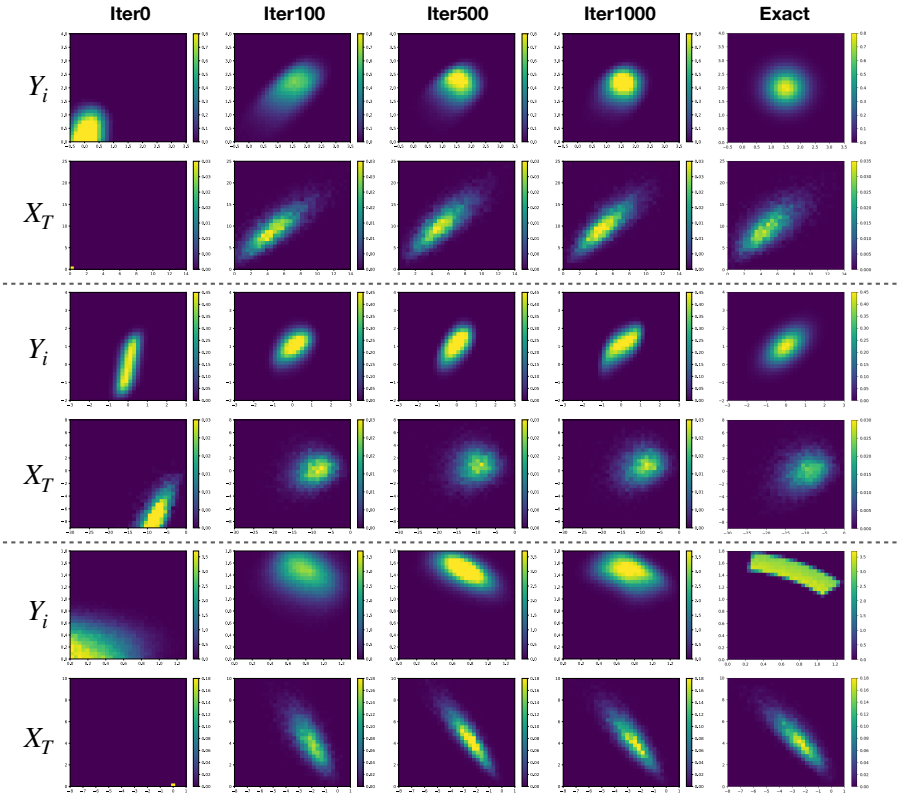

Fig. 10 Evolution of the learned jump distribution Y_i (2D random variable) and the associated 2D data distribution X_T, which is computed from Y_i using Eq. (17). The exact distributions for Y_i and X_T are shown in the last column. We look at three different distributions for Y_i and X_T; each case is shown in a pair of rows. The iteration counts are shown at the top of the figure. As the number of iterations increases (going left to right), the distribution of Y_i and X_T converge to the exact solution.

network is trained according to Algorithm 1, where

$$\boldsymbol{\omega} = (B_t, N_t(\lambda)), \quad \boldsymbol{\theta} = (Y_1, Y_2, Y_3, \ldots), \quad \mathbf{y} = X_T.$$

Fig. 10 shows the training plots, the estimations of latent jump distribution Y_i, and the observation data distribution X_T. We observe that PhysGNN reconstructs a distribution of Y_i consistent with ground truth. The problem we consider here is a typical but challenging nonparametric statistical inference problem, and the results suggest that PhysGNN can potentially benefit a wide variety of similar nonparametric estimation problems in stochastic processes.

5 Conclusion

In this paper, we develop PhysGNN, a novel way to train generative neural networks with physics knowledge. We couple the numerical PDE solvers and neural networks to learn the distributions of physical parameters and train the neural network by minimizing the discrete Wasserstein distance. We evaluate the stability of the training algorithm and the expressivity of neural network architectures in PhysGNN. We showcase a series of promising results in which we model physical parameter distributions in the Poisson's equation, the linear elasticity problem, and the stochastic jump diffusion process using deep generative neural networks. In sum, PhysGNN provides a viable approach for training generative neural networks using physics knowledge in the context of differential equations. Despite many advantages of PhysGNN, there are many questions to be investigated further. For example, the problem may not be well-posed; either the PDE system is ill-conditioned or there is too much noise in the dataset. The use of DNNs provides implicit regularization [60], which mitigates the ill-posedness problem. This line of research is left to future work.

Acknowledgements

This work is supported by the Applied Mathematics Program within the Department of Energy (DOE) Office of Advanced Scientific Computing Research (ASCR), through the Collaboratory on Mathematics and Physics-Informed Learning Machines for Multiscale and Multiphysics Problems Research Center (DE-SC0019453).

References

[1] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., Chen, X.: Improved techniques for training GANs. In: Advances in Neural Information Processing Systems, pp. 2234–2242 (2016)

[2] Brock, A., Donahue, J., Simonyan, K.: Large scale GAN training for high fidelity natural image synthesis. arXiv preprint arXiv:1809.11096 (2018)

[3] Zhu, J.-Y., Park, T., Isola, P., Efros, A.A.: Unpaired image-to-image translation using cycle-consistent adversarial networks. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2223–2232 (2017)

[4] Isola, P., Zhu, J.-Y., Zhou, T., Efros, A.A.: Image-to-image translation with conditional adversarial networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1125–1134 (2017)

[5] Hughes, T.J.: The finite element method: linear static and dynamic finite element analysis (2012)

[6] Arregui-Mena, J.D., Margetts, L., Mummery, P.M.: Practical application of the stochastic finite element method. Archives of Computational Methods in Engineering **23**(1), 171–190 (2016)

[7] Santambrogio, F.: Optimal transport for applied mathematicians. Birkäuser, NY **55**, 58–63 (2015)

[8] Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: a survey. The Journal of Machine Learning Research **18**(1), 5595–5637 (2017)

[9] Margossian, C.C.: A review of automatic differentiation and its efficient implementation. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 1305 (2018)

[10] van Merriënboer, B., Breuleux, O., Bergeron, A., Lamblin, P.: Automatic differentiation in ML: Where we are and where we should be going. In: Advances in Neural Information Processing Systems, pp. 8757–8767 (2018)

[11] Raissi, M., Perdikaris, P., Karniadakis, G.E.: Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics **378**, 686–707 (2019)

[12] Shin, Y., Darbon, J., Karniadakis, G.E.: On the convergence and generalization of physics informed neural networks. arXiv preprint arXiv:2004.01806 (2020)

[13] Kharazmi, E., Zhang, Z., Karniadakis, G.E.: hp-vpinns: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering **374**, 113547 (2021)

[14] Zhang, D., Guo, L., Karniadakis, G.E.: Learning in modal space: Solving time-dependent stochastic pdes using physics-informed neural networks. SIAM Journal on Scientific Computing **42**(2), 639–665 (2020)

[15] Yang, L., Meng, X., Karniadakis, G.E.: B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with noisy data. Journal of Computational Physics **425**, 109913 (2021)

[16] Zhang, D., Lu, L., Guo, L., Karniadakis, G.E.: Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems. Journal of Computational Physics **397**, 108850

(2019)

[17] Xu, K., Darve, E.: Physics constrained learning for data-driven inverse modeling from sparse observations. arXiv preprint arXiv:2002.10521 (2020)

[18] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Advances in Neural Information Processing Systems, pp. 2672–2680 (2014)

[19] Kingma, D.P., Welling, M.: Auto-encoding variational Bayes. arXiv preprint arXiv:1312.6114 (2013)

[20] Doersch, C.: Tutorial on variational autoencoders. arXiv preprint arXiv:1606.05908 (2016)

[21] Ho, J., Chen, X., Srinivas, A., Duan, Y., Abbeel, P.: Flow++: Improving flow-based generative models with variational dequantization and architecture design. In: International Conference on Machine Learning, pp. 2722–2730 (2019). PMLR

[22] Cuturi, M.: Sinkhorn distances: Lightspeed computation of optimal transport. In: Advances in Neural Information Processing Systems, pp. 2292–2300 (2013)

[23] Santambrogio, F.: Introduction to optimal transport theory. Notes (2014)

[24] Genevay, A., Peyré, G., Cuturi, M.: Learning generative models with Sinkhorn divergences. arXiv preprint arXiv:1706.00292 (2017)

[25] Flamary, R., Courty, N.: POT: Python Optimal Transport library (2017). https://github.com/rflamary/POT

[26] Li, D., Xu, K., Harris, J.M., Darve, E.: Time-lapse full waveform inversion for subsurface flow problems with intelligent automatic differentiation. arXiv preprint arXiv:1912.07552 (2019)

[27] Chen, T.Q., Rubanova, Y., Bettencourt, J., Duvenaud, D.K.: Neural ordinary differential equations. In: Advances in Neural Information Processing Systems, pp. 6571–6583 (2018)

[28] Owen, A.B.: Monte carlo theory, methods and examples (2013)

[29] Andrieu, C., De Freitas, N., Doucet, A., Jordan, M.I.: An introduction to MCMC for machine learning. Machine learning **50**(1-2), 5–43 (2003)

[30] Turner, B.M., Van Zandt, T.: A tutorial on approximate Bayesian

computation. Journal of Mathematical Psychology **56**(2), 69–85 (2012)

[31] Sisson, S.A., Fan, Y.: Likelihood-free mcmc (2011)

[32] Kousathanas, A., Leuenberger, C., Helfer, J., Quinodoz, M., Foll, M., Wegmann, D.: Likelihood-free inference in high-dimensional models. Genetics **203**(2), 893–904 (2016)

[33] Villani, C.: Optimal transport: old and new **338** (2008)

[34] Villani, C.: Topics in optimal transportation (58) (2003)

[35] Oberman, A.M., Ruan, Y.: An efficient linear programming method for optimal transportation. arXiv preprint arXiv:1509.03668 (2015)

[36] Pele, O., Werman, M.: Fast and robust earth mover's distances. In: 2009 IEEE 12th International Conference on Computer Vision, pp. 460–467 (2009). IEEE

[37] Schmitzer, B.: Stabilized sparse scaling algorithms for entropy regularized transport problems. SIAM Journal on Scientific Computing **41**(3), 1443–1481 (2019)

[38] Castillo, E., Mínguez, R., Castillo, C.: Sensitivity analysis in optimization and reliability problems. Reliability Engineering & System Safety **93**(12), 1788–1800 (2008)

[39] Fikhtengol'ts, G.M.: The fundamentals of mathematical analysis (2014)

[40] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)

[41] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch (2017)

[42] Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: A fast dynamic language for technical computing. arXiv preprint arXiv:1209.5145 (2012)

[43] Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM review **59**(1), 65–98 (2017)

[44] Huang, D.Z., Xu, K., Farhat, C., Darve, E.: Learning constitutive relations from indirect observations using deep neural networks. arXiv e-prints, 1905–12530 (2019) https://arxiv.org/abs/1905.12530 [math.NA]

[45] Huang, H.Q., Chang, H.H., Nanson, G.C.: Minimum energy as the general form of critical flow and maximum flow efficiency and for explaining variations in river channel pattern. Water Resources Research **40**(4) (2004)

[46] Amos, B., Kolter, J.Z.: Optnet: Differentiable optimization as a layer in neural networks. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 136–145 (2017). JMLR. org

[47] Mignolet, M.P., Przekop, A., Rizzi, S.A., Spottswood, S.M.: A review of indirect/non-intrusive reduced order modeling of nonlinear geometric structures. Journal of Sound and Vibration **332**(10), 2437–2460 (2013)

[48] Lucia, D.J., Beran, P.S., Silva, W.A.: Reduced-order modeling: new approaches for computational physics. Progress in aerospace sciences **40**(1-2), 51–117 (2004)

[49] de Souza Neto, E.A., Peric, D., Owen, D.R.: Computational methods for plasticity: theory and applications (2011)

[50] Heinbockel, J.H.: Introduction to tensor calculus and continuum mechanics **52** (2001)

[51] Xu, K., Darve, E.: Adversarial numerical analysis for inverse problems. arXiv preprint arXiv:1910.06936 (2019)

[52] Larsson, S., Thomée, V.: Partial differential equations with numerical methods **45** (2008)

[53] Hinton, G., Srivastava, N., Swersky, K.: Neural networks for machine learning lecture 6a: overview of mini-batch gradient descent. Notes (2012)

[54] Arjovsky, M., Chintala, S., Bottou, L.: Wasserstein GAN. arXiv preprint arXiv:1701.07875 (2017)

[55] Mescheder, L., Geiger, A., Nowozin, S.: Which training methods for GANs do actually converge? arXiv preprint arXiv:1801.04406 (2018)

[56] Hastings, W.K.: Monte carlo sampling methods using markov chains and their applications (1970)

[57] Roweis, S., Ghahramani, Z.: A unifying review of linear gaussian models. Neural computation **11**(2), 305–345 (1999)

[58] Tankov, P., Voltchkova, E.: Jump-diffusion models: a practitioner's guide. Banque et Marchés **99**(1), 24 (2009)

[59] Ramezani, C.A., Zeng, Y.: Maximum likelihood estimation of asymmetric jump-diffusion processes: Application to security prices. Available at SSRN 606361 (1998)

[60] Fan, T., Xu, K., Pathak, J., Darve, E.: Solving inverse problems in steady state navier-stokes equations using deep neural networks. arXiv preprint arXiv:2008.13074 (2020)